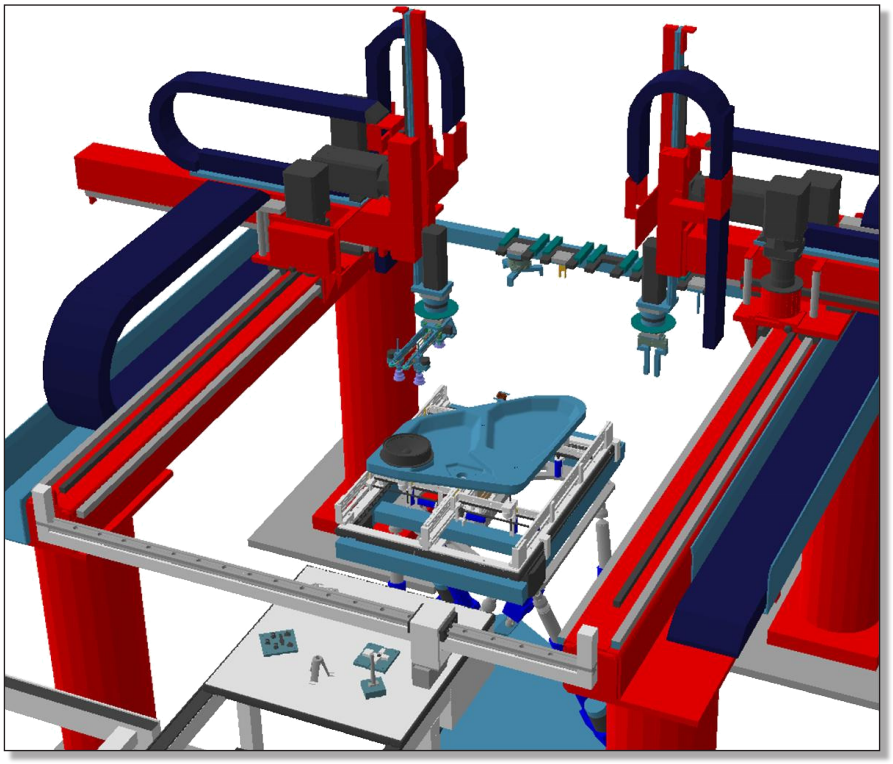
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Fertigungsautomatisierung und Produktionssystematik

Prof. Dr.-Ing. K. Feldmann

Entwurf und Implementierung einer Bahnplanungsmethode für kooperierende Industrieroboter



Studienarbeit im Fach Informatik

Betreuer: Prof. Dr.-Ing. Klaus Feldmann
Prof. Dr. Klaus Meyer-Wegener
Dipl.-Ing. Christian Ziegler
Dipl.-Inf. Christoph Neumann

Bearbeiter: cand.-inf. Florian Rampp

Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 12.02.2008

(Florian Rampp)

Inhaltsverzeichnis

1 Einleitung

Die Begriffe „Automatisierung“ oder „Robotereinsatz in der Fertigung“ wecken bei manchen Arbeitnehmern im Hochlohnland Deutschland Bedenken, Existenzängste keimen auf. Durch die zunehmende Mechanisierung von Fertigungsprozessen würden Arbeitsplätze verloren gehen, gelernte Facharbeiter würden durch moderne Industrieroboter ersetzt, die schneller, präziser und rund um die Uhr arbeiten und das ohne Lohnnebenkosten und Streikgefahr. Dabei wird vergessen, dass jeder Roboter auch neue Arbeitsplätze in anderen Bereichen schafft. Robotikfirmen sind auf der ständigen Suche nach qualifizierten Mitarbeitern für die Bereiche Forschung und Entwicklung, aber auch Service, Inbetriebnahme und Montage. Die Konsequenz ist die Verlagerung von Arbeitsplätzen in neue Sektoren, mit höheren Anforderungen an Bildung und Qualifikation.

Dennoch sind viele Tätigkeiten auch auf länger Sicht nicht durch Roboter ausführbar. Zu unterschiedlich die Arbeitsabläufe, zu flexibel die Anforderungen, die hergestellten Serien zu klein oder die Reaktion auf sich ändernde Umstände zu komplex, als dass Industrieroboter diese Arbeiten durchführen könnten. Erst mit großen Stückzahlen und uniformen Abläufen lohnte sich der Einsatz.

Doch auch für kleinere Serien und flexiblere Eingaben sind inzwischen Automatisierungslösungen auf dem Markt. Durch aufwändige Sensorsysteme, wie Bauteilscanner und den Einzug der Computertechnologie wird der Einsatz von Industrierobotern auch für flexiblere Aufgaben möglich. Die Steuerungssoftware wird zunehmend komplexer und immer mehr der Bahnberechnungen werden automatisch erledigt, so dass sich der Anwender um die Ansteuerung der Roboter auf einer höheren Ebene und größere Probleme wie der kooperativen Verbindung mehrerer Industrieroboter widmen kann.

Ein Teilgebiet des kooperativen Problembereichs ist die parallele Montage von Bauteilgruppen mittels Pick- und Place-Operationen durch mehrere Roboter. Die vorliegende Arbeit beschreibt einen Bahnplaner für einen solchen Montageprozess anhand einer Roboterzelle am Lehrstuhl für Fertigungsautomatisierung und Produktionssystematik der Universität Erlangen-Nürnberg. In dieser Zelle kommen zwei Industrieroboter mit Linearkinematik der Firma Reis Robotics zum Einsatz, die kooperierend Pick- und Place-Operationen durchführen. Als Beispielablauf dient eine Montagesequenz, bei der verschiedene Bauteile auf einem Autotürblech platziert werden. Die Roboter teilen sich dabei den Arbeitsbereich und verschiedene Wechselgreifer.

Um auch Bauteile in einer schrägen Fügeebene platzieren zu können, wird zur Verkippung des Trägerbauteils ein Hexapod als Montagetisch verwendet. Bei der Einbringung in die Zelle über ein Transportsystem können die beteiligten Bauteile und deren Positionen mit Hilfe eines Laserlinienscanners erkannt werden. Die Daten dieser Bauteile wie Material, Form und Maße, aber auch mögliche Montagesequenzen oder ein Bauteilabhängigkeitsgraph können direkt aus einem CAD-Programm exportiert und zusammen mit der erkannten Position zur Montageplanung benutzt werden. Diese findet auf einem handelsüblichen Personal Computer als Leitrechner statt, der per Ethernet-Netzwerk an die Steuerung der Roboter angeschlossen ist.

Die Anforderungen an den zu entwickelnden Montage- und Bahnplaner sind eine flexible Reaktion auf zu montierende Bauteile und deren Position und die möglichst schnelle, kooperierende Montage. Deswegen ist das Ziel – im Gegensatz zur oft verwendeten Ablaufprogrammierung mittels Teach-In von festen Positionen – der Entwurf einer autonomen und dynamischen Bahnplanung. Montageaufgaben sollen an beide Industrieroboter so vergeben werden, dass die Montagereihenfolge eingehalten und die Taktzeit optimiert wird. Die Verfahrbewegungungen sollen kollisionsfrei erfolgen und müssen deswegen vom Leitrechner zwischen den Robotern synchronisiert werden. Dazu wird der Arbeitsbereich in die Unterbereiche Pick, Place und Bahnhof zerlegt, die von jedem Roboter explizit reserviert werden müssen, falls in sie eingefahren werden soll. Zur Auflösung von Konflikten existiert für jeden Roboter ein Rückzugsraum, in dem er sich unabhängig vom anderen Roboter frei bewegen kann. Darüber hinaus ist auch die Synchronisation des Zugriffs auf Greifer und den Hexapod notwendig.

Zunächst wird in dieser Arbeit auf den aktuellen Stand der Technik im Bereich autonome und dynamische Bahnplanung eingegangen. Insbesondere sollen hier die schon existierenden Ansätze Landkarten-, Zellzerlegungs- und Potentialfeldmethode kurz erläutert werden. Anschließend werden der Aufbau der Roboterzelle, die einzelnen Komponenten sowie deren Ansteuerung und Zusammenspiel näher vorgestellt.

Daraufhin werden die Konzepte eingeführt, die dem entworfenen Bahnplaner zugrunde liegen. Es wird beschrieben, wie eine Montageaufgabe in kleinere Elementaroperationen zerlegt wird und diese ausgeführt werden. Besonderes Augenmerk liegt in der Aufteilung des Arbeitsbereichs in Unterbereiche und die Synchronisation der Roboter.

Um diese Konzepte nun in ein Softwaresystem umzusetzen, werden einige Grundlagen der Softwaretechnik benötigt. Deswegen werden das Architekturmuster „Schichtenmodell“ und einige Entwurfsmuster eingeführt. Der Grobentwurf des Bahnplaners und die Unterteilung in vier Schichten wird dargelegt und die Wahl für die Programmiersprache Java wird begründet.

Im Anschluss daran wird darauf eingegangen, wie die grundlegenden Konzepte in eine Bahnplanungssoftware integriert werden. Diese wird sich in die vier Schichten „User Interface“, „Operation“, „Cell“ und „Communication“ gliedern, denen jeweils ein Kapitel gewidmet wird.

Die Kernkonzepte des Bahnplaners finden sich in der Operation-Schicht, die mit einer Bedienoberfläche angesprochen werden kann. Die Cell-Schicht modelliert die bestehende Hardware, die mit Hilfe der Communication-Schicht angesprochen wird.

Beim Entwurf des Bahnplaners soll besonderes Augenmerk auf Wartbarkeit und Modularität gelegt werden. Erweiterungen oder neue Ansätze für die Bahnplanung sollen sich einfach integrieren lassen. Im anschließenden Ausblick werden deswegen einige Möglichkeiten aufgezeigt, wie der Bahnplaner ausgebaut und weitere Funktionalität hinzugefügt werden kann.

2 Stand der Technik

2.1 Kooperierende Roboter

Von einer Kooperation von Robotern wird gesprochen, wenn sich mehrere Roboter bei einem Fertigungsprozess gegenseitig unterstützen oder gemeinsam Aufgaben erledigen. Sie verfügen meist über einen gemeinsamen Arbeitsbereich, in dem Teile gemeinsam gehandhabt oder verarbeitet werden. So können Füge- und Handhabungsoperationen getätigt werden, die ein einzelner Roboter allein nicht erledigen kann. Teile von Fertigungsoperationen lassen sich nun auch parallelisieren, was in einer verkürzten Taktzeit resultiert.

2.2 Online- und Offline-Programmierung

Bisherige Lösungen zur Programmierung von kooperierenden Robotern lassen sich in mehrere Bereiche einteilen. Eine einfache Lösung, mehrere kooperierende Roboter, deren Arbeitsbereich sich überschneidet zu programmieren, ist das *statische Teach-In* von fest vorgegebenen Positionen. Von allen Robotern wird bei der Montage ein Programm abgefahren, das einmalig einprogrammiert wurde. Diese Lösung ist unflexibel und kann nicht auf neue Eingaben reagieren. Die Programmierung muss für jeden Roboter einzeln erfolgen, ist zeitaufwändig und kann nur von einem Spezialisten durchgeführt werden. Auch ist sie im Ablauf meist nicht optimal, da die anzufahrenden Positionen nicht aus einer Berechnung stammen, die die zeitliche Abfolge optimiert.

Eine weitere Möglichkeit stellt der *Master-Slave-Ansatz* dar. Hier fungiert ein Roboter als Master, der die Bewegungsbahn vorgibt. Andere Roboter sind an diesen geometrisch gekoppelt. Sie führen die gleichen oder gespiegelte Bewegungen des Masters aus. Es existieren Synchronisierungspunkte, die von allen Robotern erreicht werden müssen, bis eine weitere Aktion gestartet werden kann. Die Programmierung wird hier vereinfacht, da Positionen nur noch für einen Roboter vorgegeben werden müssen. Der Bewegungsablauf ist dennoch statisch und nicht flexibel. Eine Kollisionsüberwachung findet nicht statt.

Sowohl der Teach-In- als auch der Master-Slave-Ansatz lassen sich unter dem Begriff Online-Programmierung einordnen. Es muss direkt an der Roboterzelle mit den vorhandenen Robotern gearbeitet werden, so dass eine längere Unterbrechung des Fertigungsprozesses notwendig ist.

Bei der Offline-Programmierung hingegen wird mit Hilfe eines Simulationsprogramms der Fertigungsprozess am Computer nachgebildet. Mit einer Kinematiksimulation können die gewünschten Bewegungen des Roboters im Programm vorgegeben werden. Die Programmierung erfolgt jedoch weiterhin manuell. Durch diesen Ansatz wird aber die Zeit, in der die Roboter nicht produktiv arbeiten minimiert. Sowohl bei der Online- als auch bei der Offlineprogrammierung ist der Anwender für eine kollisionsfreie Bewegung verantwortlich.

2.3 Autonome Bahnplanung

Für dynamische Eingaben wie veränderbare Bauteilgruppen, deren Position a priori nicht bekannt ist, empfiehlt sich der Einsatz von *autonomen Bahnplanern*. Diese berechnen mit der aktuellen Position und dem Zielpunkt als Eingabe die kollisionsfreien Trajektorien von Bahnen in Echtzeit. So wird der Anwender von der Programmierung der konkreten Bahnen entlastet. Auf den Teach-In-Vorgang von Positionen kann gänzlich verzichtet werden. Durch Nutzung eines Scannersystems können die Positionen der zu fügenden Bauteile auf dem Werkstückträger bestimmt und an die Bahnplanung übergeben werden. Deswegen werden Formnester, die die Bauteile tragen, zur Positionsbestimmung nicht mehr benötigt. Im Vordergrund steht hierbei vor allem der Zeitgewinn durch den Wegfall des Teach-In-Prozesses. Dies fällt vor allem bei Kleinserien deutlich ins Gewicht, wo eine häufige Neuprogrammierung der Roboter entfällt.

Existierende autonome Bahnplaner lassen sich auf drei Grundprinzipien zurückführen, die im folgenden kurz vorgestellt werden. Die Beschränkung auf den zweidimensionalen Fall bei allen Bahnplanern ist der Einfachkeit und dem Umstand geschuldet, dass auch der in dieser Arbeit vorgestellte Bahnplaner zweidimensional arbeitet.

2.3.1 Landkartenmethode

Hiermit werden die bekannten Hindernisse in ein Modell des Arbeitsraums eingetragen [5]. Zwischen den Eckpunkten des Arbeitsraums, dem Start- und Endpunkt der Bewegung und allen Eckpunkten von Hindernissen werden nun Geraden eingezeichnet. Schneidet eine Gerade ein Hindernis, so wird sie weggelassen. Nun wird folgendermaßen ein Graph gebildet: Als Knoten dienen alle Mittelpunkte der eingezeichneten Geraden. Kanten zwischen den Knoten werden gezogen, wenn die Kante kein Hindernis schneidet. Der Graph stellt nun alle möglichen Verfahrwege dar.

Ein Alternative ist das Voronoi-Diagramm, bei dem Verfahrwege so einzeichnet werden, das sie in jedem Punkt den maximalen Abstand zu allen Hindernissen einnehmen.

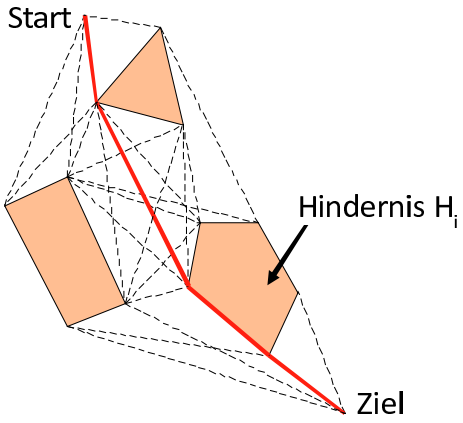


Bild 2.1: Die Planung von Verfahrwegen mittels der Landkartenmethode.

2.3.2 Zellzerlegungsmethode

Bei dieser Bahnplanungsmethode wird der Arbeitsbereich in kleinere Zellen zerlegt [4]. Diese Methode kann exakt oder näherungsweise durchgeführt werden. Bei der exakten Methode wird der Freiraum um die Hindernisse in Trapeze unterteilt. Verbindet man Mittelpunkte von Trapezkanten, die zwei Freiraumzellen verbinden, so ensteht eine kollisionsfreie Bahn, die befahren werden kann.

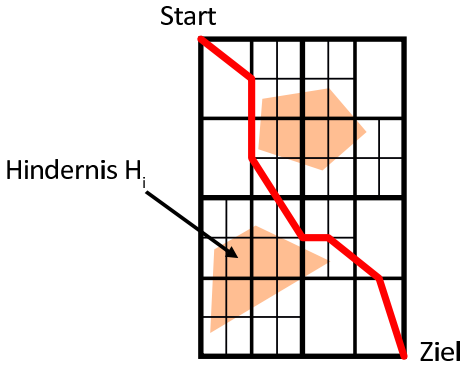


Bild 2.2: Die Zerlegung der Zelle in Unterbereiche mit Hilfe der Zellzerlegungsmethode.

Bei der näherungsweisen Zerlegung kann der Arbeitsbereich in vier gleichgroße Rechtecke zerlegt werden. Jedes Rechteckt ist entweder vollkommen von einem Hindernis belegt, vollkommen frei oder teilweise belegt. Die Zellen, die teilweise belegt sind, werden rekursiv in vier weitere Teilzellen zerlegt, deren Belegungszustand wiederum gespeichert wird. Dies wird solange wiederholt, bis eine gewünschte Granularität erreicht ist. Es entsteht hierbei ein Graph in Baumform, ein sogenannter Quad-Tree. Mittels geeigneter Algorithmen kann dieser Baum auf der Suche nach einem freien Pfad traversiert werden.

2.3.3 Potentialfeldmethode

Die Potentialfeldmethode [6] wird im Gegensatz zur Landkarten- oder Zellzerlegungsmethode zu den lokalen Bahnplanungsmethoden gerechnet. Hierbei muss kein ganzes Modell des Arbeitsraums erstellt, sondern nur die unmittelbare Umgebung des Roboters analysiert werden. Es wird immer nur der nächste zu fahrende Wegabschnitt und kein Gesamtweg vom Ausgangs- zum Zielpunkt berechnet.

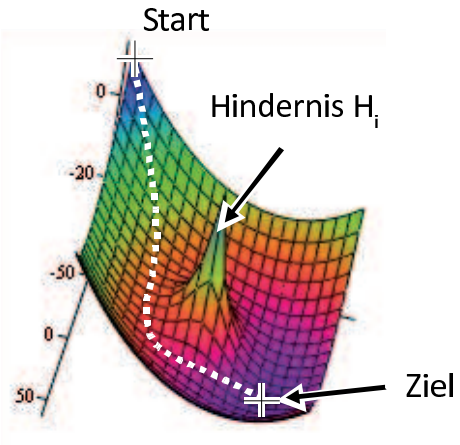


Bild 2.3: Die Potentialfunktion der Potentialfeldmethode mit dem Ziel der Bewegung als Senke und Hindernissen als Quellen.

Das Grundprinzip ist die Definition einer Potentialfunktion, wie sie aus der Elektrostatik bekannt ist. Hindernisse und der Roboter sind positive Ladungen, die sich in der Potentialfunktion als Erhebungen auswirken. Der Zielpunkt ist eine negative Ladung, die eine Senke im Potentialgebirge ergibt. Nun wird am momentanen Aufenthaltspunkt des Roboters der Gradient der Potentialfunktion berechnet. Der negative Gradient zeigt in die Richtung des steilsten Abfalls der Potentialfunktion und somit den besten Weg zum Zielpunkt an. Wie ein Ball in einem Gebirge bewegt sich der Roboter so auf den Zielpunkt zu.

Problematisch ist der Auftritt von lokalen Minima, zum Beispiel bei konkaven Hindernissen. Hierfür müssen Strategien entwickelt werden, die den Roboter von lokalen Minima weg oder aus angesteuerten Minima herausführen.

2.4 Weitere Probleme bei der Bahnplanung mit realen Robotern

Im Gegensatz zu den momentan vorgestellten Methoden ist die Bahnplanung für reale Roboter komplexer. Die Vorstellung vom Roboter als bewegter Punkt im Zweidimensionalen muss aufgegeben werden, denn jeder reale Arbeitsraum ist dreidimensional. Der Roboter besteht aus einer Kette von Gliedern, die in die Kollisionsberechnung miteinbezogen werden müssen.

Dies geschieht beispielsweise durch die Umhüllung des Roboterarms durch meist in Quaderform gestalteten Hüllkörpern. Es sind hinreichend viele Algorithmen zur Kollisionsüberprüfung von geometrisch einfachen Objekten bekannt. Diese werden nun paarweise auf das Hindernis und die Menge der Hüllkörper des Roboters angewendet.

Des Weiteren sind Hindernisse dynamisch und bewegen sich im Laufe der Zeit, da auch die anderen beteiligten Roboter bei der Bahnplanung eines Roboters als Hindernisse betrachtet werden. Meist ist die Position dieser Hindernisse nicht im Voraus bekannt, sondern muss durch periodische Positionsabfrage ermittelt werden. Dies dauert eine bestimmte Zeit, welche eine Restriktion bei den Verfahrgeschwindigkeiten der Roboter auferlegt.

2.5 Zusammenfassung

Nach einer kurzen Definition des Begriffs „kooperierende Roboter“ wurde die Onlineprogrammierung Roboterzelle mittels festem Teach-In von Positionen sowie die Offlineprogrammierung mit einer vorhergehenden Simulation der Roboter in einem Computer vorgestellt. Beide Verfahren resultieren in einem festen Montageablauf und bieten keine Möglichkeiten der flexiblen Reaktion auf veränderte Eingaben. Aus diesem Grund wurden drei existierende Verfahren zur autonomen Bahnplanung vorgestellt. Die Landkartenmethode sucht mögliche Verfahrwege durch die Zelle mittels Aufbau eines Wegegraphen. Bei der Zellzerlegungsmethode wird der Arbeitsbereich der Zelle in kleinere Segmente unterteilt, die entweder als frei oder als von einem Hindernis belegt markiert werden. Die Potentialfeldmethode plant Verfahrwege über die Richtung des steilsten Abstiegs einer Potentialfunktion, in der Hindernisse Quellen und das Ziel der Bewegung eine Senke bewirkt.

Alle Verfahren weisen bei der praktischen Umsetzung Probleme auf, die im vorhergehenden Abschnitt kurz aufgezeigt wurden. Im Folgenden soll nun ein Bahnplaner entwickelt werden, der diese Probleme umgeht und in der in der vorgestellten Roboterzelle operiert. Dieses neu entwickelte Verfahren setzt auf die explizite Sperrung von Teilarbeitsbereichen der Zelle auf.

3 Bereichssperrverfahren und die Montagezelle

Die im vorigen Kapitel vorgestellten Bahnplaner weisen bei der praktischem Umsetzung inhärente Probleme auf. Aus diesem Grund wird im Folgenden ein neuer Bahnplaner entwickelt, der die angesprochenen Probleme umgeht. Das grundlegende Prinzip ist die exklusive Reservierung von Teilbereichen des Arbeitsbereich für einen Roboter, falls er in diesen einfahren soll. Zur Vermeidung von Deadlocks gibt es für jeden Roboter einen Rückzugsbereich, in dem dieser unabhängig vom anderen Roboter bewegt werden kann.

Der entwickelte Bahnplaner kommt in einer Montagezelle mit einem beispielhaften Montageablauf zum Einsatz, der anschließend genauer vorgestellt werden soll. Der zentrale Gedanke dabei ist, die Gesamtmontage in einer Zelle durchzuführen, was durch den flexiblen Aufbau der Zelle gewährleistet ist. Als Beispielszenario dient die Montage einer Autotür. Bauteile wie Fensterkurbel und Lautsprecher werden nach und nach auf dem eingebrachten Türinnenblech platziert.

Die Roboterzelle besteht aus zwei Linearrobotern, die sich gegenüber stehen, einem Hexapod als Montagetisch und einem Greiferbahnhof, in dem Greifer automatisch während der Montage gewechselt werden können. Der Aufbau der Montagezelle wird im Weiteren mit besonderem Augenmerk auf die softwareseitige Ansteuerung der aktiven Komponenten vorgestellt.

3.1 Das Bereichssperrverfahren

Beim Bereichssperrverfahren wird der Arbeitsbereich einer Montagezelle in mehrere Unterbereiche zerlegt. Soll ein Roboter im Zuge einer Montageoperation in einen dieser Bereiche einfahren, so muss dieser vom Bahnplaner zuerst gesperrt werden. Nur in für den verfahrenden Roboter gesperrte Bereiche darf eingefahren werden. Hat ein anderer Roboter den Zielbereich bereits reserviert, weil er in diesen einfahren will, oder sich schon in diesem befindet, so kommt eine Ausweichstrategie zum Tragen. Diese sieht in diesem Fall eine Bewegung in einen für jeden Roboter exklusiven Rückzugsbereich vor. Im Rückzugsbereich wird gewartet, bis der gewünschte Zielbereich frei wird. Anschließend wird er für den verfahrenden Roboter reserviert und der Befehl zum Einfahren in diesen Bereich kann an den Roboter gesendet werden. Zielpunkte von Bewegungen dürfen nur innerhalb von Bereichen liegen. Das Anfahren eines Zielpunkts

außerhalb eines Bereichs ist nicht zulässig. In Bild 3.1 sind beispielhaft einige Arbeitsbereiche und Rückzusbereiche für zwei Roboter dargestellt.

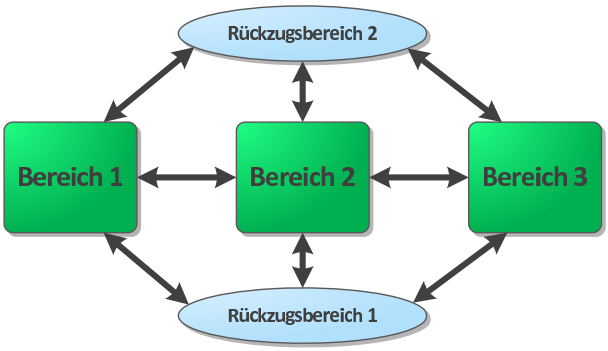


Bild 3.1: Drei Arbeitsbereiche, die im Rahmen des Bereichssperrverfahrens vor dem Einfahren reserviert werden müssen und Rückzugsbereiche zur Deadlock-Vermeidung für zwei Roboter.

Es ist nicht zwingend notwendig, dass von jedem Arbeitsbereich in jeden anderen direkt gefahren werden kann. Eine Bewegung von einem Bereich in einen anderen kann auch über den Rückzugsbereich oder einen dritten Arbeitsbereich erfolgen, der in diesem Falle ebenfalls reserviert werden muss. Bei einer direkten Bewegung von Bereich 1 in Bereich 3, welche in Bild 3.1 dargestellt sind, muss Bereich 2 ebenfalls reserviert werden. Verlässt ein Roboter einen Bereich, so wird dieser freigegeben und ist für andere wieder frei verfügbar. In Abschnitt 4.2.3 auf Seite 33 wird auf die Konzepte von Sperren und Freigeben der Arbeitsbereiche genauer eingegangen.

3.2 Ablauf des Beispielmontageprozesses

Der Montageprozess besteht aus einer Reihe von Pick- und Place-Operationen, die zu montierende Komponenten auf einem Innenblech einer Autotür platzieren. Dazu werden zwei Linearroboter und ein Hexapod benutzt, die in einer Roboterzelle angeordnet sind, die in Bild 3.2 auf der nächsten Seite zu sehen ist.

Der erste Schritt der Montage besteht aus dem Transport des Autotürblechs auf den Hexapod, welcher sich so an das Förderbandsystem annähern kann, dass die Übergabe der Trägerpalette möglich ist. Zu montierende Bauteile werden anschließend auf einer Werkstückträgerpalette über das Transportsystem in die Zelle eingebracht. Die Bauteile werden mit einem an der Vorderseite der Zelle befestigten Laserlinienscanner erkannt und mit einer Mustererkennungssoftware identifiziert. Nach Abschluss des Scanvorgangs wird die Werkstückpalette fixiert und die Montage beginnt.

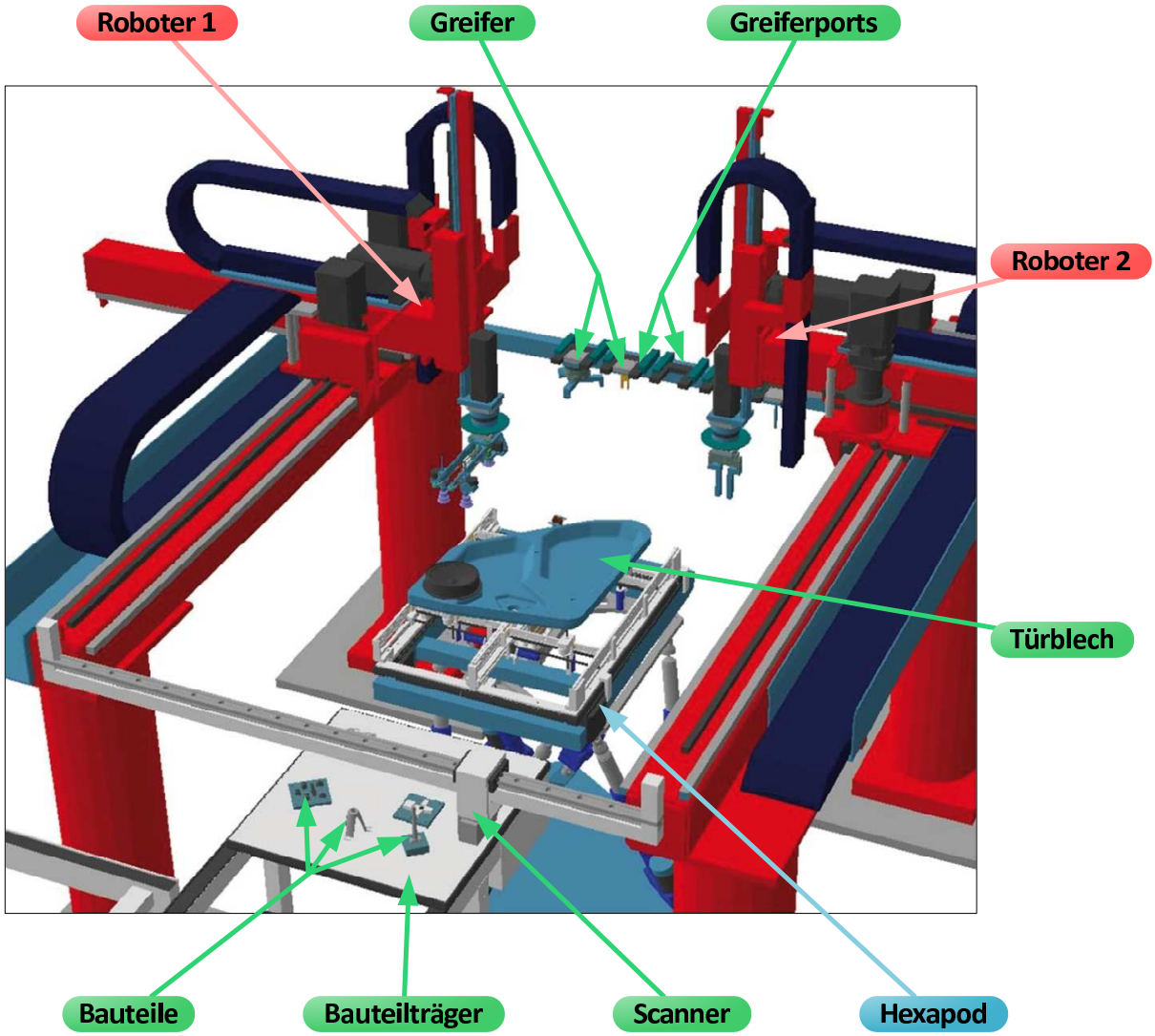


Bild 3.2: Die Gesamtzelle mit Robotern, dem Transportsystem und dem Hexapod

Beide Roboter nehmen nacheinander Bauteile von der Palette auf und setzen sie auf das Türblech auf dem Hexapod. Dazu werden Greifer benutzt, die in einzelne Greiferports am hinteren Ende der Zelle eingehängt sind. Ein Wechsel des Greifers ist automatisch und während des Montageablaufs möglich. Der Hexapod muss vor jedem Place-Vorgang so verfahren werden, dass die Fügeoperation senkrecht erfolgen kann. Sind alle Bauteile auf dem Türblech platziert, ist der Montageablauf beendet.

Der gesamte Montageprozess wird von einem Leitrechner aus gesteuert. Dies ist ein handelsüblicher Personal Computer auf dem die Steuerungssoftware läuft. Die Anbindung an die Steuerungen der Roboter und des Hexapods erfolgt über Ethernet und das TCP/IP[1]-Protokoll.

3.3 Die beiden Industrieroboter

Zur Montage sind zwei Industrieroboter des Typs *RL-16* der Firma *Reis Robotics* vorhanden. Diese stehen sich punktsymmetrisch gegenüber und spannen einen Arbeitsraum von ca. $2\text{m} \times 1\text{m} \times 1\text{m}$ auf. Der *RL-16* ist ein Linearroboter und verfügt über drei translatorisch verfahrbare Achsen. Zusätzlich kann der angeflanschte Greifer um die z-Achse gedreht werden. Ein Simulationsmodell eines RL-16 ist in Bild 3.3 gezeigt. Die Traglast des Roboterarms beträgt 16 kg, die Positionierungsgenauigkeit wird von Reis mit $\pm 0{,}1\,\text{mm}$ angegeben. Die beiden Roboter in der Zelle werden im Folgenden Roboter 1 und Roboter 2 genannt.

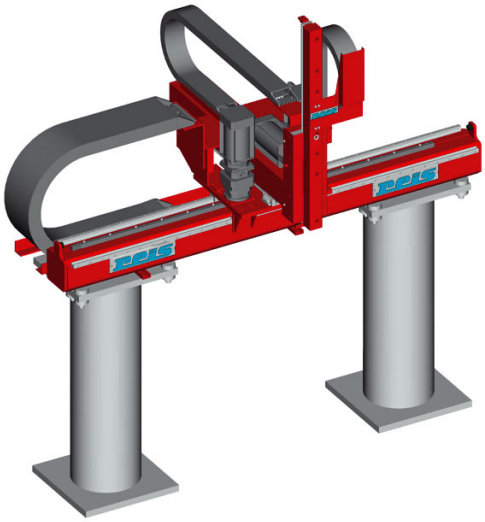


Bild 3.3: Simulationsmodell des *RL-16* Roboter der Firma *Reis Robotics*

Über eine Handsteuerung kann der Roboter manuell verfahren werden. Ablaufprogramme können in das PHG[2] geladen und ausgeführt werden. So ist das feste Einteachen von Bewegungsabläufen möglich. Zusätzlich besteht die Möglichkeit, den Roboter über Ethernet anzusteuern. Mittels des TCP/IP-Protokolls werden XML[3]-Kommandos an den Roboter gesendet, der den

1 Transmission Control Protocol / Internet Protocol
2 Programmierbares Handgerät
3 Extensible Markup Language

Befehl ausführt und eine Antwort ebenfalls im XML-Format zurückliefert. So kann der Roboter dynamisch programmiert werden, indem die Ablaufsteuerung in einem Host-Computer läuft. Die Kommandos dienen vor allem dem Setzen und Auslesen der Systemvariablen des Roboters und dem Setzen und Löschen von Bits in einer Integer-Robotervariablen.

Positionen können in verschiedenen Koordinatensystemen an den Roboter gesendet und Ist-Positionen abgefragt werden. Es lassen sich in den Robotersteuerungen beliebige Koordinatensysteme definieren, indem bestimmte Punkte angefahren und ein Konfigurationsskript auf dem PHG ausgeführt wird. Zusätzlich zum Standardkoordinatensystem „Base“ wurde für beide Roboter schon im Voraus ein gemeinsames Weltkoordinatensystem definiert, mit welchem im Weiteren gearbeitet wird.

Fährt der Roboter über den Zellbereich hinaus, so wird ein Endschalter aktiviert um eine Beschädigung des Roboters zu vermeiden. Auch für die rotatorische Achse gibt es bei einem gewissen Winkel einen Endschalter, über den hinaus nicht gedreht werden kann. Treten zu hohe Achsgeschwindigkeiten (translatorisch, als auch rotatorisch) auf, so reagiert der Roboter ebenfalls mit einem Abbruch um vor Beschädigung zu schützen.

Der *RL-16* kann in zwei verschiedenen *Verfahrmodi* bedient werden: PTP[1], wo versucht wird, den Zielpunkt zu erreichen, indem alle Achsen asynchron schnellstmöglich die Zielkoordinate anstreben und CP-LIN[2], der eine lineare Bewegung zwischen Start- und Endpunkt erzwingt. Anzufahrende Positionen werden dem Roboter mitgeteilt, indem sie per XML-Kommando in die Systemvariable `_PEXTPOS` geschrieben werden. Wird der eingebaute *Ringpuffer* benutzt, so wird die Position nach dem Schreiben in `_PEXTPOS` in den Ringpuffer kopiert. Dieser hält bis zu 50 Positionen, die sequenziell angefahren werden. Ist die Option *Überschleifen* aktiviert, so ist es möglich, Zwischenpositionen nicht exakt anzufahren. Die Geschwindigkeit wird beim Erreichen einer Zwischenposition dabei nicht auf Null reduziert. Vielmehr fährt der Roboter ab einem gewissen Abstand zum anzufahrenden Punkt eine Kurve in Richtung des nächsten Punktes.

3.4 Die Greifer und der Greiferbahnhof

An die beiden Roboter können speziell konstruierte Greifer angeflanscht werden. Diese werden aufgenommen, indem der Roboter von oben in den Flansch des Greifers einfährt und dann durch Druckluft Verriegelungsbolzen ausfährt. Die Greifer verfügen über einen oder mehrere Anschlüsse an weitere Druckluftleitungen, durch die der Greifer geöffnet, geschlossen oder – falls er dafür ausgelegt ist – gedreht werden kann.

1 Point-To-Point
2 Controlled Path - linear

Es existieren verschieden Typen von Greifern. Zwei verschiedene Größen von Zweibackengreifern fassen Bauteile seitlich, ein Dreibackengreifer ist für zylinderförmige Bauteile vorhanden. Außerdem gibt es noch zwei Greifer, deren Backen waagrecht orientiert sind und die je nach manueller Einstellung um die horizontale Achse um bis zu 225° gedreht werden können. Dies kann zum Beispiel für eine kooperierende Werkstückdrehung genutzt werden.

Die Greifer können in speziellen sogenannten Greiferports abgelegt werden. Diese befinden sich am hinteren Ende der Zelle in einem Bahnhofsbereich. Dieser wird von beiden Robotern zusammen genutzt. Greifer können hier eingeparkt und abgeholt werden. Somit ist ein automatischer Wechsel der Greifer während des Montageablaufs möglich.

3.5 Ansteuerung der externen Peripheriegeräte

Für die Bauteilzubringung existieren Förderbänder der Firma Bosch. Zum Festspannen der Montagepalette und zum Absperren der Förderbänder werden Hebebolzen verwendet, die mittels Druckluft bedient werden. Außerdem melden induktive Sensoren die Position der Montagepalette. Diese Peripheriegeräte sind über einen Schaltkasten an die binären Ein- und Ausgänge von Roboter 1 angeschlossen und können somit ebenfalls mit Kommandos an den Roboter gesteuert und ausgelesen werden.

3.6 Hexapod zur Verkippung des Trägerbauteils

Durch die Linearkinematik können Bauteile nur senkrecht platziert werden. Um nun auch von dieser Richtung abweichende Fügeoperationen zuzulassen, wird als Montagetisch ein Hexapod verwendet, welcher in Bild 3.4 auf der nächsten Seite dargestellt ist. Dieser kann das Trägerbauteil durch Verfahren von sechs Elektrozylindern der Firma *Raco* heben und senken, seitlich kippen und verdrehen. Somit kann die Fügeebene in eine horizontale Position gebracht werden, so dass die Montageoperation möglich ist.

Die Ansteuerung erfolgt ebenfalls über TCP/IP und Ethernet mit einem proprietären Anwendungsprotokoll. Gesendet werden Positionen für jeden einzelnen Zylinder. Die gewünschte Position (bestehend aus drei kartesischen und drei Winkelkoordinaten) muss auf dem Steuerungscomputer auf die Einzelwerte für jeden Zylinder umgerechnet werden.

Bild 3.4: Der Hexapod mit sechs Elektrozylindern der Firma *Raco* als Montagetisch

3.7 Laserscanner zur Bauteilerkennung und Lagebestimmung

Auf der Vorderseite der Zelle ist an einer Querstrebe ein Laserscanner vom Typ *scanCONTROL 2800* der *Firma Micro-Epsilon* befestigt, welcher in Bild 3.5 auf der nächsten Seite dargestellt ist. Dieser kann über das Triangulationsprinzip ein Höhenprofil eines Winkelbereichs erstellen. Wird die Palette unter dem Laser hindurch bewegt, entsteht ein dreidimensionales Abbild, das per Firewire an den Steuerrechner übertragen wird. Auf diesem läuft eine Mustererkennungssoftware, mit der die zu montierenden Bauteile, die auf der Palette liegen, erkannt werden. Die Positionen auf der Palette werden in eine Datei geschrieben, aus der die Montagesteuerung dann die Pick-Positionen auslesen kann.

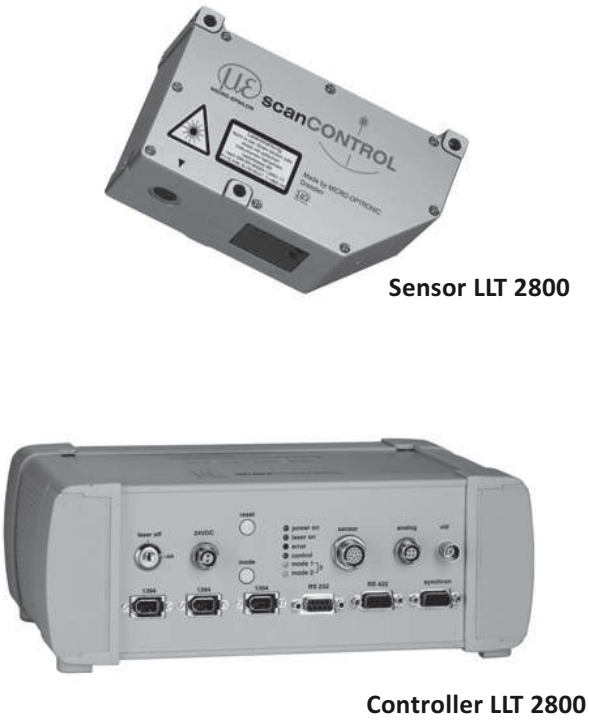


Bild 3.5: Der Laserscanner *scanCONTROL 2800* der Firma *Micro-Epsilon*

3.8 Die Umsetzung des Bereichssperrverfahrens in der vorgestellten Montagezelle

Zur Umsetzung des Bereichssperrverfahrens muss der vorhandene Arbeitsbereich in Unterbereiche zerlegt werden. Da die Montage eines Bauteils aus dem Greiferwechsel im Bahnhofsbereich, der Aufnahme eines Bauteils und der Platzierung auf dem Türblech besteht wird für jeden dieser Schritte ein Bereich definiert, in dem der Zielpunkt dieses Teilschritts liegt. So ergeben sich die Bereiche *Pick*, *Place* und *Station* und für jeden Roboter ein Rückzugsbereich, die in Bild 3.6 auf der nächsten Seite grün eingezeichnet sind. Die Rückzugsbereiche sind linienförmig, da in diese nur zur Kollisionsvermeidung eingefahren wird und in ihnen keine physischen Operationen durchgeführt werden, die eine zweidimensionale Ausdehnung rechtfertigen würden. In Abschnitt 4.2.3.1 auf Seite 33 wird genauer auf die Definition dieser Arbeitsbereich in der Montagezelle eingegangen. Die Arbeitsbereiche werden über eine externe Konfiguration eingelesen, so dass deren Maße und Positionen einfach verändert werden können.

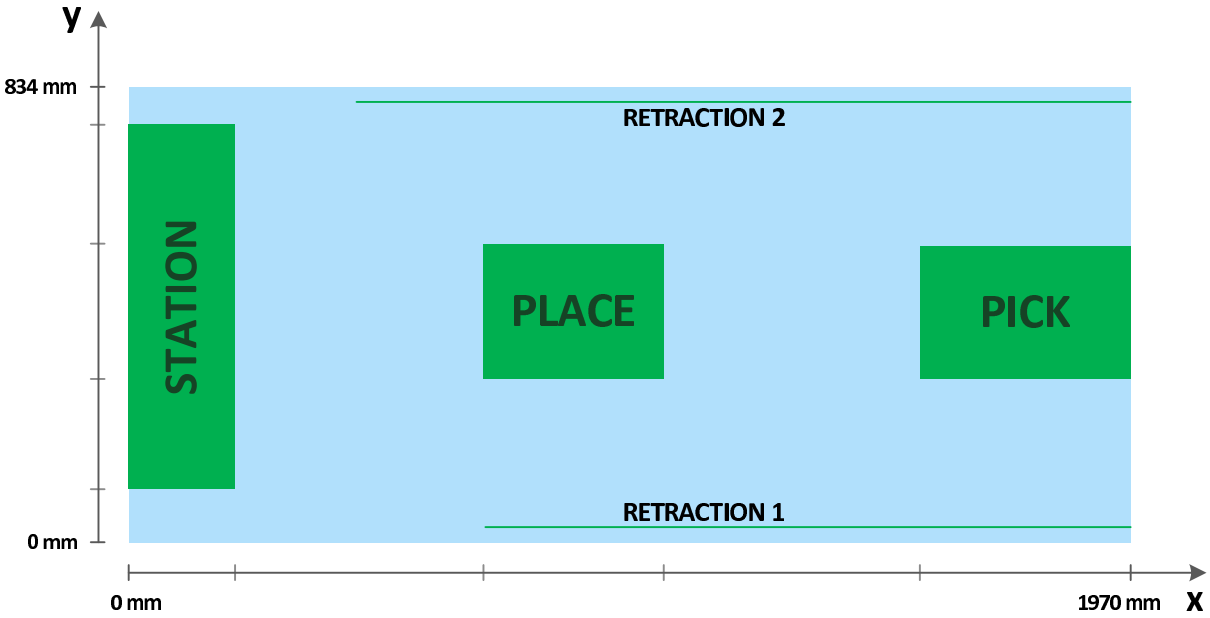


Bild 3.6: Die im Rahmen des Bereichssperrverfahrens für die Montagezelle definierten Arbeitsbereiche Pick, Place und Station sowie Retraction 1 und Retraction 2 als Rückzugsbereiche für die beiden Roboter

3.9 Zusammenfassung

Als Lösung für die Probleme der im vorigen Kapitel vorgestellten Bahnplaner wurden die Grundsätze des Bereichssperrverfahrens vorgestellt. Dieser Bahnplaner kommt im Rahmen einer Beispielmontagesequenz zum Einsatz, welche anschließend beschrieben wurde. Der genauere Aufbau der Montagezelle schloss sich an. Hierbei wurden die Industrieroboter der Firma Reis und der Hexapod als Montagetisch, sowie die softwareseitige Ansteuerung dieser Komponenten vorgestellt. Um verschiedene Bauteile handhaben zu können, werden unterschiedliche Greifer benutzt, welche anschließend beschrieben wurden. Ebenfalls wurde der automatische Wechsel dieser Greifer über spezielle Greiferports dargestellt. Um die Position der zu montierenden Bauteile auf der Trägerpalette zu bestimmen wird ein Laserscanner benutzt, welcher im darauffolgenden Abschnitt dargestellt wurde. Abschließend wurden die in dieser Zelle verwendeten Arbeitsbereiche des Bereichssperrverfahrens vorgestellt. Im folgenden Kapitel soll nun genauer auf die Grundlagen des Bereichssperrverfahren und weitergehende Aspekte zur Planung des Montageablaufs eingegangen werden.

4 Grundlegende Konzepte des Bahnplaners

Dieses Kapitel stellt einige grundlegende Konzepte des entwickelten Montage- und Bahnplaners vor. Es wird beschrieben, wie eine Montageaufgabe ausgewählt und in einzelne Elementaroperationen zerlegt wird, welche nacheinander ausgeführt werden. Das Ziel ist zwar die Kooperation von zwei Robotern, aber die konzeptionelle Zerlegung einer Aufgabe und die Ausführung der Elementaroperationen erschließt sich auch schon am Beispiel der Steuerung nur eines Roboters. Deswegen wird im ersten Teil dieses Kapitels zuerst ein einziger Roboter betrachtet.

Erst anschließend wird die Bahnplanung so erweitert, dass die Kooperation mit einem zweiten Roboter möglich ist. Dazu werden verschiedene Arbeitsbereiche definiert, die explizit reserviert werden müssen, wenn ein Roboter diese betreten will. Die Reservierung und Freigabe dieser Bereiche und anderer Ressourcen geschieht über weitere Elementaroperationen, die erst im Kontext der Kooperation und gegenseitigen Abstimmung von mehreren Robotern eingeführt werden müssen. Sämtliche Konzepte und Operationen der Steuerung eines Roboters werden auch bei der kooperierenden Steuerung benötigt.

4.1 Montage- und Bahnplanung mit einem Roboter

Wird ein Roboter allein betrachtet, so können schon anhand dieses Szenarios einige Konzepte des Montageablaufs gezeigt werden. Montageaufgaben werden durch Analyse eines Abhänigkeitsgraphen ausgewählt und über einen Zwischenschritt in Elementaroperationen zerlegt. Diese werden in Abarbeitungseinheiten, den sogenannten Executors ausgeführt. Zur Anforderung und Freigabe der verschiedenen Ressourcen gibt es eine zentrale Verwaltungsinstanz, welche am Ende vorgestellt wird.

4.1.1 Der Task als elementarer Baustein der Montageplanung

Der Montageablauf, der mit einem Roboter ausgeführt werden soll, besteht aus einzelnen Montageaufgaben. Eine Montageaufgabe entspricht dabei der Montage eines Bauteils. Dazu muss gegebenenfalls zuerst der Greifer gewechselt werden. Anschließend wird das zu montierende Bauteil im *Pick-Bereich* aufgenommen und im *Place-Bereich* abgesetzt. Diese Sequenz aus bedingtem Greiferwechsel, Pick und Place wird im Folgenden als *Assembly-Task* bezeichnet.

Der Assembly-Task trägt als Parameter die Pick- und Place-Position für den Roboter und die Hexapodposition für den Place-Vorgang. Außerdem hat jeder Assembly-Task eine eindeutige Nummer und eine Beschreibung des Bauteils, das montiert wird.

Zusätzlich gibt es noch Aufgaben, die Entwicklungs-, Test- und Initialisierungszwecken dienen, die keine Montageaufgabe darstellen. Sie werden vom Benutzer des Systems manuell ausgelöst, um z.B. einen Greifer zu parken oder abzuholen oder eine bestimmte Position anzufahren. Diese Aufgaben werden *Manual-Tasks* genannt. Auf diese wird im Weiteren nicht genauer eingegangen.

Jeder Task befindet sich in einem Zustand, der angibt, ob der Task bearbeitet werden kann, gerade ausgeführt wird, oder ob die Ausführung schon beendet ist.

4.1.2 Der Montagegraph und Scheduling der Tasks

Bei der Montage der Bauteile müssen oft Reihenfolgen eingehalten werden. Das heißt, ein Bauteil muss zwingend vor einem anderen montiert werden. Dennoch gibt es Bauteile, die zeitlich unabhängig voneinander gesetzt werden können. So existiert zu jedem Zeitpunkt eine Menge an Tasks, die zur Ausführung bereit sind. Von diesen Tasks können eventuell andere abhängen, die erst danach ausgeführt werden können. Somit ergibt sich ein *Abhängigkeitsgraph* der Bauteile, welcher auch als *Montagegraph* bezeichnet wird. Tasks bilden die Knoten, gerichtete Kanten stellen Montageabhängigkeiten dar.

Alle Quellen im Graph (Knoten mit keinen eingehenden Kanten) stellen ausführbare Tasks dar. Diese sind in Bild 4.1 grün markiert. Soll ein nächster auszuführender Task bestimmt werden, so wird aus allen Quellen ein Task ausgewählt, was als *Scheduling* bezeichnet wird. Dabei muss

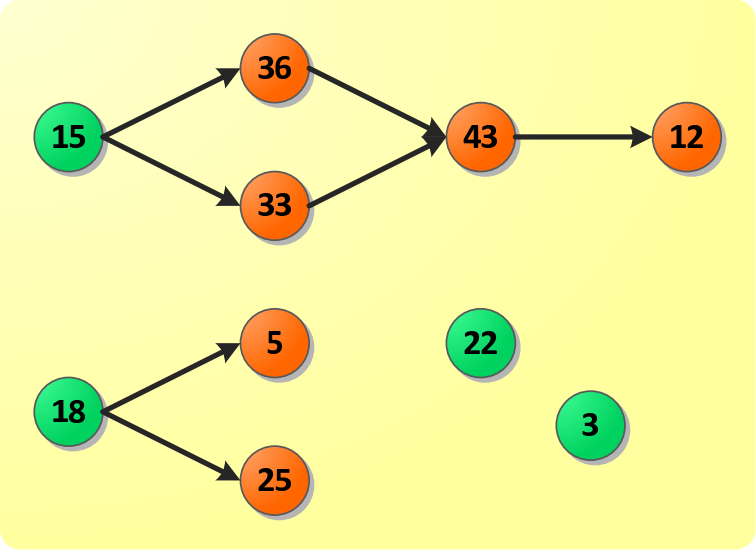


Bild 4.1: Ein Beispiel für einen Montagegraph. Jede Montageaufgabe trägt eine eindeutige Nummer.

entschieden werden, welcher Task aus der Menge der möglichen Aufgaben ausgewählt werden soll.

Die implementierte Schedulingstrategie berücksichtigt den aktuell angeflanschten Greifer und wählt Tasks aus, die möglichst den gleichen Greifer benötigen. Da ein Greiferwechsel zeitlich aufwändig ist, wird angenommen, dass durch diese Strategie ein möglichst kurzer Montageablauf resultiert. Möglich wäre aber auch eine Analyse des Graphen, so dass stets versucht wird, einen abschließenden Leerlauf eines Roboters, während der Andere noch Aufgaben verarbeitet, zu vermeiden (siehe Abschnitt 9 auf Seite 90 für weitere Möglichkeiten). Nach der Abarbeitung des Assembly-Tasks wird dieser aus dem Montagegraph entfernt.

4.1.3 Die Komplexoperationen

Komplexoperationen stellen einzelne, logische Abarbeitungsschritte eines Tasks dar, die aus dem grundlegenden Montageprozess heraus motiviert sind. Folgende COPs[1] wurden eingeführt:

- `FetchGripperOp(Greifertyp)`: Aufnehmen eines Greifers vom gegebenen Typ
- `ParkGripperOp`: Ablegen eines Greifers in einen freien Port
- `ChangeGripperOp(Greifertyp)`: Bedingtes Ablegen des angeflanschten und Aufnehmen des benötigten Greifers
- `PickOp(Pick-Position)`: Abholen eines Bauteils im Pick-Bereich
- `PlaceOp(Place-Position Roboter, Place-Position Hexapod)`: Platzieren eines Bauteils im Place-Bereich

4.1.4 Zerlegung eines Tasks in einzelne Komplexoperationen

Um einen Task feingranularer zu behandeln und Synchronisationspunkte innerhalb der Abarbeitung eines Tasks zu ermöglichen, wird eine Montageaufgabe bei der Ausführung in die vorgestellen COPs unterteilt. Ein Assembly-Task wird hierbei in eine `ChangeGripperOp`, eine `PickOp` und eine `PlaceOp` zerlegt. Dies entspricht der logischen Reihenfolge der Montage eines Bauteils. Außerdem kann jeder Komplexoperation einer der drei Arbeitsbereiche Station, Pick und Place zugeordnet werden.

Ein Manual-Task wird ebenfalls in eine Liste von Complex Operations transformiert, worauf hier aber nicht weiter eingegangen wird.

1 Complex Operations

4.1.5 Die Elementaroperationen

Elementaroperationen stellen die einzelnen Elementarschritte der Abarbeitung eines Montageabschnitts dar. Im Gegensatz zu den COPs, die logisch motiviert waren, sind die EOPs[1] Detailschritte, die konzeptuell der tatsächlichen Ansteuerung der aktiven Komponenten näher liegen. Sie bilden eine Hierarchie, die sie in Untergruppen teilt. Diese ist in Bild 4.2 dargestellt. Im folgenden Abschnitt werden die COPs in EOPs zerlegt.

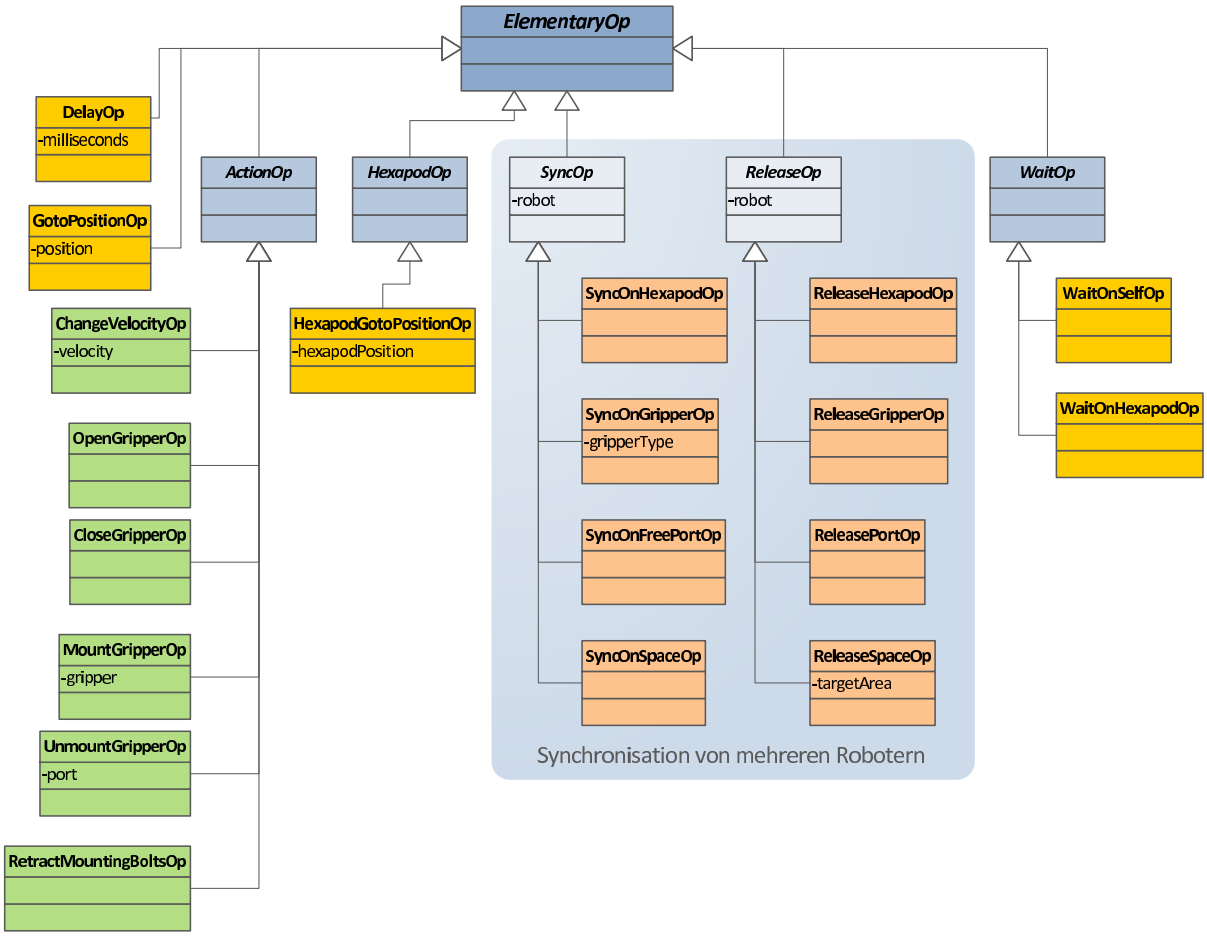


Bild 4.2: Die Hierarchie der Elementaroperationen

Die erste Gruppe bilden die `ActionOps`. Hier gibt es die `ChangeVelocityOp`, die `OpenGripperOp` bzw. `CloseGripperOp`, `MountGripperOp` bzw. `UnmountGripperOp` und `RetractMountingBoltsOp` zum Rückzug der Bolzen, die den Greifer am Roboter halten. `ChangeVelocityOp` ist mit der Geschwindigkeit parametrisiert, `MountGripperOp` mit dem anzubringenden Greifer und `UnmountGripperOp` mit dem Port, in den der Greifer nach dem

1 Elementary Operations

Abmontieren eingehängt ist. Aus Operationen für den Hexapod besteht die zweite Gruppe. Hier existiert bisher nur die `HexapodGotoPositionOp`, die dem Hexapod eine anzufahrende Position sendet. Die Gruppe der `WaitOps` besteht aus `WaitOnHexapodOp` und `WaitOnSelfOp`, deren Ausführung solange blockiert, bis Hexapod bzw. Roboter an der zuletzt gesendeten Position angekommen sind.

Darüber hinaus gibt es noch die EOP `DelayOp`, die die weitere Ausführung um ein bestimmtes Zeitintervall verzögert und `GotoPositionOp`, die als Parameter eine Position trägt, die an den Roboter gesendet wird. Die Operation verläuft asynchron, d.h. es wird nicht gewartet, bis der Roboter die Position erreicht, sondern nur, bis die Position gesendet wurde. Dies erklärt die Notwendigkeit für die `WaitOnSelfOp`. Außerdem trägt die `GotoPositionOp` eine semantische Markierung der enthaltenen Position, z.B.: „`direkt über dem Port`“. Diese wird im Folgenden bei der Nennung einer `GotoPositionOp` in Klammern angegeben.

4.1.6 Zerlegung der Komplex- in Elementaroperationen

Die COPs, in die ein Task zerlegt wurde, werden der Reihe nach ausgeführt und dabei wiederum in Ketten von EOPs unterteilt.

4.1.6.1 Zerlegung der `ChangeGripperOp`

Die `ChangeGripperOp` dient dem Wechsel des Greifers. Es wird geprüft, welcher Greifer momentan am Roboter befestigt ist. Ist der benötigte Greifer angeflanscht, werden keine EOPs erzeugt. Die `ChangeGripperOp` gilt dann als abgearbeitet. Trägt der Roboter keinen Greifer, so wird eine `FetchGripperOp` erzeugt und diese wiederum zerlegt. Ist der falsche Greifer angeflanscht, so wird die `ChangeGripperOp` in eine Folge aus `ParkGripperOp` und `FetchGripperOp` umgewandelt und deren Zerlegung zurückgegeben. `ChangeGripperOp` bildet die einzige Ausnahme in der Zerlegung der COPs, da sie nicht direkt in EOPs, sondern in andere COPs zerlegt wird.

4.1.6.2 Zerlegung der `FetchGripperOp`

Die `FetchGripperOp` dient dem Holen eines Greifers. Wegen der Kollisionsgefahr mit dem Greiferport muss dabei langsam in den Greifer eingefahren werden. Als erstes wird die Geschwindigkeit auf eine Basisgeschwindigkeit gesetzt und ein Punkt mit maximaler z-Koordinate über dem Greiferport angefahren. Danach wird die Verfahrgeschwindigkeit verlangsamt und ein Punkt einige Zentimeter über dem Port angefahren. Schließlich wird die Geschwindigkeit nochmal verringert und die Bolzen zurück gefahren, sodass in den Flansch des im Port hängenden Greifers eingefahren werden kann. Der Greifer wird angehängt, indem die Fixierungsbolzen ausgefahren werden. Nach einem kurzen Delay wird langsam in x-Richtung aus dem Port ausgefahren. Einige

Zentimeter vor dem Port wird die Geschwindigkeit erhöht und ein weiterer Punkt vor dem Port wird an den Roboter gesendet, der die Endposition einer `FetchGripperOp` bildet.

Wenn der Roboter einen Greifer holt und in diesen einfährt, bzw. mit Greifer aus dem Port fährt wird sehr langsam gefahren. Da der Gesamtablauf trotzdem schnell erfolgen soll, wird der Weg, der langsam gefahren wird auf einige wenige Zentimeter beschränkt, indem oben beschriebene Zwischenschritte eingeführt werden. Diese Zwischenschritte bedingen jeweils ein `ChangeVelocityOp`, ein `GotoPositionOp` und ein `WaitOnSelfOp`. Falls der Zwischenschritt entfallen soll, können diese Befehle entfernt werden. Grundsätzlich gilt für die Geschwindigkeiten:

$$\text{Basisgeschwindigkeit} < \text{Zwischengeschwindigkeit} < \text{Portgeschwindigkeit}$$

Folgende Liste wird erzeugt. Eventuelle Parameter der EOPs folgen der Operation in Klammern. Die EOPs für die Zwischenschritte sind grau dargestellt.

1. `ChangeVelocityOp(Basisgeschwindigkeit)`
2. `GotoPositionOp(über dem Port - z maximal)`
3. `WaitOnSelfOp`
4. `ChangeVelocityOp(Zwischengeschwindigkeit)`
5. `GotoPositionOp(direkt über dem Port)`
6. `WaitOnSelfOp`
7. `ChangeVelocityOp(Portgeschwindigkeit)`
8. `RetractMountingBoltsOp`
9. `GotoPositionOp(in Port)`
10. `WaitOnSelfOp`
11. `MountGripperOp(Greifer)`
12. `DelayOp(timeout)`
13. `GotoPositionOp(direkt vor dem Port)`
14. `WaitOnSelfOp`
15. `ChangeVelocityOp(Zwischengeschwindigkeit)`
16. `GotoPositionOp(vor dem Port)`
17. `WaitOnSelfOp`

Die `WaitOnSelfOp` nach jedem Senden einer Position ist notwendig, da `GotoPositionOp` asynchron arbeitet und somit nicht wartet, bis die Position erreicht ist. Da aber beispielsweise die Änderung auf die langsamere Geschwindigkeit erst erfolgen soll, wenn der Roboter die zuletzt gesendete Position erreicht hat, muss gewartet werden. Ohne die `WaitOnSelfOp` würde die Geschwindigkeit direkt nach dem Senden der Position und nicht nach Erreichen der Position gesetzt werden.

4.1.6.3 Zerlegung der `ParkGripperOp`

Ähnlich zum Holen wird das Parken eines Greifers zerlegt. Es wird nun – ebenfalls mit Zwischenschritten, bei denen die Geschwindigkeit verringert wird – von vorne in den Port eingefahren und der Greifer gelöst. Der Roboter wird – wieder mit einem Zwischenschritt – nach oben abgezogen.

Es ergibt sich folgende Liste:

1. `ChangeVelocityOp(Basisgeschwindigkeit)`
2. `GotoPositionOp(vor dem Port)`
3. `WaitOnSelfOp`
4. `ChangeVelocityOp(Zwischengeschwindigkeit)`
5. `GotoPositionOp(direkt vor dem Port)`
6. `WaitOnSelfOp`
7. `ChangeVelocityOp(Portgeschwindigkeit)`
8. `GotoPositionOp(in Port)`
9. `WaitOnSelfOp`
10. `UnmountGripperOp(Zielport)`
11. `DelayOp(timeout)`
12. `GotoPositionOp(direkt über dem Port)`
13. `WaitOnSelfOp`
14. `ChangeVelocityOp(Zwischengeschwindigkeit)`
15. `GotoPositionOp(über dem Port - z maximal)`
16. `WaitOnSelfOp`

Die Liste entspricht weitgehend der umgekehrten Zerlegung einer `FetchGripperOp`. Die EOPs für die Zwischenschritte sind ebenfalls ausgegraut. Als Parameter für die `UnmountGripperOp` wird der Zielport übergeben. Dies ist notwendig, um in der internen Greiferverwaltung zu registrieren, wo sich welcher Greifer befindet.

4.1.6.4 Zerlegung der `PickOp`

Genauso wie beim Aufnehmen und Ablegen eines Greifers wird beim Pickvorgang aus Gründen der Kollisionsgefahr das Bauteil sehr langsam angefahren. Der Greifer wird um das Bauteil herum geschlossen und langsam wieder nach oben abgezogen. Da jedoch der Gesamtablauf möglichst schnell sein soll, wird die langsame Geschwindigkeit nur für eine kurze Strecke beibehalten um dann auf eine höhere Geschwindigkeit zu wechseln.

Eine `PickOp` wird folgendermaßen zerlegt:

1. `ChangeVelocityOp(Basisgeschwindigkeit)`
2. `GotoPositionOp(über der Pick-Position)`
3. `WaitOnSelfOp`
4. `OpenGripperOp`
5. `ChangeVelocityOp(Zwischengeschwindigkeit)`
6. `GotoPositionOp(direkt über der Pick-Position)`
7. `WaitOnSelfOp`
8. `ChangeVelocityOp(Pick-Geschwindigkeit)`
9. `GotoPositionOp(Pick-Position)`
10. `WaitOnSelfOp`
11. `CloseGripperOp`
12. `GotoPositionOp(direkt über der Pick-Position)`
13. `WaitOnSelfOp`
14. `ChangeVelocityOp(Zwischengeschwindigkeit)`
15. `GotoPositionOp(über der Pick-Position)`
16. `WaitOnSelfOp`

4.1.6.5 Zerlegung der `PlaceOp`

Die Place-Operation ist analog zur Pick-Operation. Es muss jedoch noch der Hexapod mit einbezogen werden, der in die richtige Place-Position gefahren werden muss, ehe der Place-Vorgang weiter ausgeführt werden kann. Eine `PlaceOp` wird folgendermaßen zerlegt (die Hexapodoperationen sind grün dargestellt):

1. `ChangeVelocityOp(Basisgeschwindigkeit)`

2. HexapodGotoPositionOp(Hexapod-Place-Position)
3. GotoPositionOp(über der Place-Position)
4. WaitOnSelfOp
5. ChangeVelocityOp(Zwischengeschwindigkeit)
6. GotoPositionOp(direkt über der Place-Position)
7. WaitOnSelfOp
8. ChangeVelocityOp(Place-Geschwindigkeit)
9. WaitOnHexapodOp
10. GotoPositionOp(Place-Position)
11. WaitOnSelfOp
12. OpenGripperOp
13. GotoPositionOp(direkt über der Place-Position)
14. WaitOnSelfOp
15. ChangeVelocityOp(Zwischengeschwindigkeit)
16. GotoPositionOp(über der Place-Position)
17. WaitOnSelfOp

Die Operationen zum Senden einer Position an den Hexapod wird direkt nach dem Holen des Hexapod-Locks in die Kette eingefügt, die Operation zum Warten auf den Hexapod sehr spät, da der Hexapod langsamer verfährt als die Roboter.

4.1.6.6 Verschmelzung der GotoPositionOp und der WaitOnSelfOp

In den vorgestellten Zerlegungen der COPs folgt eine WaitOnSelfOp stets auf eine GotoPositionOp. Deswegen könnte die GotoPositionOp geändert werden, so dass diese synchron arbeitet. Die Abarbeitung der GotoPositionOp würde also erst zurückkehren, wenn die Position erreicht ist und WaitOnSelfOp wäre überflüssig.

Wird die Bahnplanung mit mehreren Robotern betrachtet, müssen zwischen die GotoPositionOp und die WaitOnSelfOp jedoch weitere Operationen eingefügt werden. Somit ist eine Verschmelzung dieser beiden Operationen nicht mehr möglich.

4.1.7 Ausführung der Elementaroperationen

Bisher wurde nur gezeigt, wie ein Task in verschiedene EOPs zerlegt wurde. Diese haben an sich keine Funktionalität und sind passiv. Deshalb existiert für jede EOP ein sogenannter *Executor*, der die EOP ausführt. Somit wird die semantische Bedeutung der EOP in einer Liste zusammen mit anderen EOPs von ihrer Ausführung getrennt.

Die Einheit, die EOPs ausführt, hat eine Liste solcher Executor. Jede dieser ausführenden Einheiten wird nun gefragt, ob sie die EOP, die am Kopf der Liste steht bearbeiten kann. Ist dies möglich, so wird dem Executor die gesamte Liste übergeben. Dieser verarbeitet nun das EOP am Kopf der Liste und gibt die Restliste zurück.

Der ausführende Executor kann die Liste am Kopf auch um weitere EOPs ergänzen oder EOPs in der Liste verändern. Von diesen Möglichkeiten werden diejenigen Executor Gebrauch machen, die mehrere Roboter über spezielle EOPs synchronisieren. Dann wird auch der Vorteil der schon angesprochenen strikten Trennung zwischen den EOPs im Kontext einer Liste und deren Ausführung zum Tragen kommen. Dazu mehr in Abschnitt 4.2.1 auf der nächsten Seite.

4.1.8 Die vier Ressourcentypen

Im Bahnplanungssystem werden Roboter als aktive Instanzen angesehen, die Ressourcen reservieren. Ressourcen sind Betriebsmittel, die den Robotern zugänglich sind und um die diese konkurrieren. Ressourcen können von jeweils einem Roboter für sich reserviert werden, woraus der exklusive Zugriff auf diese folgt.

Folgenden Ressourcentypen werden im Bahnplaner berücksichtigt:

1. Greifer
2. freie Ports
3. Hexapod
4. räumliche Bereiche

Es wird zwischen einem Greifer und dessen Typ unterschieden. Von einem Typ kann es mehrere Greiferinstanzen geben. So hat die COP `ChangeGripperOp` einen Parameter für den Greifertyp und nicht einen Greifer. Dies ist wichtig, weil erst bei der Ausführung entschieden wird, welche konkrete Ausprägung eines Typs nun geholt werden soll. Von der Ressource Hexapod gibt es nur eine Ausprägung. Dennoch rechtfertigt dies die Bezeichnung als Ressourcentyp, da beide Roboter um die Ansteuerung des Hexapods konkurrieren. Außerdem wird so die zentrale Verwaltung der Ressourcen in einem Ressourcenmanager möglich.

4.2 Montage- und Bahnplanung mit mehreren Robotern

Im Gegensatz zum Montageablauf mit nur einem Roboter müssen bei der Benutzung von mehreren Robotern zusätzliche Vorkehrungen zur Kollisionsvermeidung, Kooperation und Synchronisierung von Ressourcen getroffen werden. Dazu werden unter anderem zusätzliche EOPs und deren Executor eingeführt. Außerdem kann nun nicht mehr a priori bestimmt werden, welchen Greifer die `MountGripperOp` oder welchen Port die `UnmountGripperOp` trägt. Analog dazu sind auch die Positionen in den `GotoPositionOps` nicht mehr im Voraus bestimmt, sondern müssen nach der Auswahl eines konkreten Greifers/Ports in die `GotoPositionOps` eingefügt werden. Im Weiteren wird die Ausführung mit zwei Robotern betrachtet. Die Konzepte greifen jedoch genauso bei einer höheren Anzahl von Robotern.

4.2.1 Zusätzliche Elementaroperationen zur Synchronisierung

Wie oben schon beschrieben, konkurrieren die Roboter um Ressourcen. Jeder Ressource ist ein sog. *Lock* zugeordnet, der angefordert werden muss, um die Ressource exklusiv für sich zu reservieren. Neue Elementaroperationen werden zum exklusiven Reservieren einer Ressource und zu deren Freigabe eingeführt. Die EOPs zum Reservieren von Ressourcen sind von der `SyncOp` abgeleitet. Jede `SyncOp` trägt als Parameter den Roboter, der die Ressource anfordern will. Das Anfordern und Freigeben von Ressourcen klammert Operationen, die auf der angeforderten Ressource operieren. Folgende `SyncOps` gibt es:

`SyncOnHexapodOp` Diese Elementaroperation dient zum Anfordern des Hexapod-Locks. Die Ausführung dieser EOP blockiert so lange, bis der Lock erfolgreich belegt wurde.

`SyncOnGripperOp` Falls beide Roboter zur gleichen Zeit den gleichen Greifer aufnehmen wollen und kein Lock für einen Greifer existieren würde, so nähme der erste Roboter den Greifer auf. Der Zweite bekäme davon nichts mit und würde versuchen, den Port anzufahren, in dem der Greifer gerade eben noch hing. Deswegen ist es nötig, auch Greifer explizit zu reservieren. Dazu dient die `SyncOnGripperOp`. Diese EOP enthält als zusätzlichen Parameter den Typ des Greifers, der benötigt wird.

Bei der Ausführung wird nun ein freier Greifer des benötigten Typs ausgewählt. Ist kein passender Greifer verfügbar, weil ein anderer Roboter diesen momentan für sich reserviert hat, so muss eine *Ausweichstrategie* gefahren werden. Der ausgewählte Greifer wird nun reserviert. Die `MountGripperOp` kennt bisher den konkreten Greifer nicht. Der Executor der `SyncOnGripperOp` ersetzt also den bisherigen `null`-Wert in der `MountGripperOp` durch den ausgewählten Greifer. Außerdem wird die Position des Ports, in dem der Greifer hängt

in der `GotoPositionOp(Port)` ergänzt. Die Koordinaten der weiteren `GotoPositionOps` können aus der Portposition hergeleitet werden.

Nach der Reservierung des ausgewählten Greifers, dem Ersetzen der Position in der `GotoPositionOp` und des Greifers in der `MountGripperOp` ist die `SyncOnGripperOp` fertig ausgeführt.

`SyncOnFreePortOp` Aus ähnlichen Gründen wie beim expliziten Reservieren eines Greifers müssen auch freie Ports vor der Benutzung reserviert werden. Ansonsten wäre es möglich, dass der erste Roboter einen Greifer in einen Port parkt und der zweite Roboter – der diesen Port noch als frei wahrnimmt – versucht, einen weiteren Greifer in diesem Port abzulegen.

Die Ausführung der `SyncOnFreePortOp` wählt zuerst den freien Port aus, der der aktuellen Position des Roboters am nächsten ist. Dann wird der Lock dieses Ports angefordert und der `null`-Wert in der `UnmountGripperOp` mit dem ausgewählten Port überschrieben. Anschließend wird wie bei der `SyncOnGripperOp` die Zielposition des Ports in der `GotoPositionOp(Port)` ergänzt.

`SyncOnSpaceOp` Diese EOP dient zum Reservieren von räumlichen Bereichen und ermöglicht die kollisionsfreie Bewegung von mehreren Robotern. Dies ist das Herzstück der Bewegungsplanung und wird im Abschnitt 4.2.3 auf Seite 33 genauer beschrieben.

Die beiden Operationen `SyncOnGripperOp` und `SyncOnFreePortOp` laufen sehr ähnlich ab. Es wird ein freier Greifer vom benötigten Typ bzw. ein freier Port ausgewählt und dieser reserviert. Der Greifer wird in die `MountGripperOp` eingetragen, der Port in die `UnmountGripperOp`. Außerdem wird bei beiden Operationen die Zielposition in der `GotoPositionOp(Port)` ergänzt. Zum *Freigeben von Ressourcen* gibt es weitere EOPs, die von `ReleaseOp` ableiten. Sie tragen alle als Parameter den Roboter, der die Ressource momentan belegt.

`ReleaseHexapodOp` Diese EOP dient dem Freigeben des Hexapods.

`ReleaseGripperOp` Mit dieser EOP kann der Lock des aktuell vom Roboter belegten Greifers wieder freigegeben werden.

`ReleasePortOp` Die `ReleasePortOp` gibt einen belegten Port wieder frei. Diese EOP wurde nicht `ReleaseFreePortOp` genannt, da der Port bei der Freigabe des Locks nicht mehr frei ist, sondern jetzt einen Greifer enthält.

`ReleaseSpaceOp` Diese EOP wird benutzt, um räumliche Bereiche wieder freizugeben. Diese EOP enthält außerdem einen Parameter, der den Zielbereich der Bewegung angibt. Ist der Zielbereich erreicht, so kann davon ausgegangen werden, dass das Freigeben von Bereichen beendet ist und die Ausführung dieser EOP kehrt zurück.

Von einem Roboter kann maximal ein freier Port, ein Greifer und ein Hexapod belegt werden. Ansonsten ist beim Freigeben nicht klar, welche der belegten Ressourcen nicht mehr benötigt wird. Dies wird als *Single-Resource-Restriction* bezeichnet. Für die Ressource Bereich gilt diese Einschränkung nicht.

4.2.2 Erweiterte Zerlegung der Komplex- in Elementaroperationen

Die neu eingeführten Elementaroperationen ergänzen nun die bisherigen, indem sie bei der Zerlegung der COPs in die EOP-Listen mit eingefügt werden. Die neuen EOPs sind grün markiert. Die Zwischenschritte zur Geschwindigkeitsreduzierung wurden weggelassen.

4.2.2.1 Zerlegung der `FetchGripperOp`

Vor dem Holen eines Greifers muss dieser mit Hilfe der `SyncOnGripperOp` reserviert werden. Erst in der COP `ParkGripperOp` wird der Lock für den Greifer mittels einer `ReleaseGripperOp` wieder freigegeben. Um in den Bahnhofsbereich einfahren zu können muss dieser zuvor explizit reserviert werden. Dazu dient die eingefügte `SyncOnSpaceOp`. Dass der Zielbereich der Bahnhofsbereich ist, kann aus der Position der nachfolgenden `GotoPositionOp` bestimmt werden. Sobald die anzufahrende Position an den Roboter gesendet wurde kann das Freigeben der Bereiche gestartet werden, die nicht mehr weiter benötigt werden. Dies geschieht noch vor dem Warten auf die Ankunft an der Zielposition, so dass die Verfahrzeit schon zum Freigeben der Bereiche benutzt werden kann, sodass ein weiterer Roboter noch während der Bewegung des freigebenden Roboters in einen gerade freigegebenen Bereich einfahren kann. Der Bereich, der in der `SyncOnSpaceOp` vor der `GotoPositionOp` angefordert wurde, ist Zielbereich der Bewegung und wird deswegen bei der Ausführung der `ReleaseSpaceOp` nicht freigeben. Erst bei der Ausführung der nächsten `ReleaseSpaceOp` im Rahmen der nächsten COP wird der Bahnhofsbereich entsperrt. Die resultierende Zerlegung wird im folgenden dargestellt. Die neuen Operationen sind grün dargestellt.

1. `ChangeVelocityOp(Basisgeschwindigkeit)`
2. `SyncOnGripperOp(GripperType)`
3. `SyncOnSpaceOp`
4. `GotoPositionOp(über dem Port - z maximal)`
5. `ReleaseSpaceOp`
6. `WaitOnSelfOp`
7. `ChangeVelocityOp(Portgeschwindigkeit)`

8. RetractMountingBoltsOp
9. GotoPositionOp(in Port)
10. WaitOnSelfOp
11. MountGripperOp(Greifer)
12. DelayOp(timeout)
13. GotoPositionOp(vor dem Port)
14. WaitOnSelfOp

4.2.2.2 Zerlegung der ParkGripperOp

Ebenso wie bei der FetchGripperOp muss beim Parken eines Greifers der Bahnhofbereich zuerst über eine SyncOnSpaceOp angefordert werden. Zuvor muss allerdings noch ein freier Port reserviert werden, was durch die SyncOnFreePortOp erfolgt. Am Ende der Liste wird der Port mit einer eingefügten ReleasePortOp wieder freigegeben. Zuvor wird auch der Lock des Greifers, der bis zum Ablegen an den Roboter angeflanscht war wieder freigegeben.

1. ChangeVelocityOp(Basisgeschwindigkeit)
2. SyncOnFreePortOp
3. SyncOnSpaceOp
4. GotoPositionOp(vor dem Port)
5. ReleaseSpaceOp
6. WaitOnSelfOp
7. ChangeVelocityOp(Portgeschwindigkeit)
8. GotoPositionOp(in Port)
9. WaitOnSelfOp
10. UnmountGripperOp(Zielport)
11. DelayOp(timeout)
12. ReleaseGripperOp
13. GotoPositionOp(über dem Port - z maximal)
14. ReleasePortOp
15. WaitOnSelfOp

4.2.2.3 Zerlegung der `PickOp`

Die Zerlegung der `PickOp` ändert sich nur geringfügig, da der Pick-Bereich angefordert werden muss. Nach dem Senden einer Position im Pick-Bereich werden alle anderen Bereiche, die nicht mehr benötigt werden freigegeben. Bei der Ankunft im- Pick-Bereich wird nur noch der Bereichs-Lock auf den Pick-Bereich selbst gehalten.

1. `ChangeVelocityOp(Basisgeschwindigkeit)`
2. `SyncOnSpaceOp`
3. `GotoPositionOp(über der Pick-Position)`
4. `ReleaseSpaceOp`
5. `WaitOnSelfOp`
6. `OpenGripperOp`
7. `ChangeVelocityOp(Pick-Geschwindigkeit)`
8. `GotoPositionOp(Pick-Position)`
9. `WaitOnSelfOp`
10. `CloseGripperOp`
11. `GotoPositionOp(über der Pick-Position)`
12. `WaitOnSelfOp`

4.2.2.4 Zerlegung der `PlaceOp`

Bei der Zerlegung der `PlaceOp` kommt zusätzlich zur Anforderung des `PlaceOp`-Bereichs und der Freigabe von nicht mehr benötigten Bereichen noch die Sperrung des Hexapods vor dessen Ansteuerung und die Rückgabe des Locks nach dem Place-Vorgang hinzu. So erfolgt die Ansteuerung des Hexapods in einer Klammer zwischen dem Anfordern und der Freigabe des Locks.

1. `ChangeVelocityOp(Basisgeschwindigkeit)`
2. `SyncOnSpaceOp`
3. `SyncOnHexapodOp`
4. `HexapodGotoPositionOp(Hexapod-Place-Position)`
5. `GotoPositionOp(über der Place-Position)`
6. `ReleaseSpaceOp`

7. `WaitOnSelfOp`
8. `ChangeVelocityOp(Place-Geschwindigkeit)`
9. `WaitOnHexapodOp`
10. `GotoPositionOp(Place-Position)`
11. `WaitOnSelfOp`
12. `OpenGripperOp`
13. `GotoPositionOp(über der Place-Position)`
14. `ReleaseHexapodOp`
15. `WaitOnSelfOp`

4.2.3 Synchronisation über das Sperren von Arbeitsbereichen

Die schon erwähnten räumlichen Bereiche zur Synchronisation können von verschiedener Gestalt sein. So können direkte Verfahrwege der Roboter räumliche Korridore definieren, die explizit angefordert werden müssen (mehr dazu in Abschnitt 9 auf Seite 90). Im Folgenden wird allerdings davon ausgegangen, dass ein räumlicher Bereich einem bestimmten, festen Arbeitsbereich in der Zelle entspricht.

4.2.3.1 Definition der Arbeitsbereiche

Die gesamte Arbeitsbereich der Roboterzelle wird in fünf Unterarbeitsbereiche unterteilt. Diese sind zweidimensional in der Draufsicht auf die Zelle in Bild 4.3 auf der nächsten Seite zu erkennen. Folgende Bereiche wurden definiert:

Pick der Arbeitsbereich, in dem Bauteile aufgenommen werden

Place der Arbeitsbereich, in dem Bauteile platziert werden

Station der Bereich um den Greiferbahnhof

Retraction1 der Rückzugsbereich des ersten Roboters

Retraction2 der Rückzugsbereich des zweiten Roboters

Jeder Roboter kann in seinem Rückzugsbereich in der x-Koordinaten frei verfahren, ohne auf den anderen Roboter zu achten. Den Rückzugsraum des anderen Roboters darf er nicht betreten. Die drei gemeinsamen Bereiche Pick, Place und Station dürfen erst nach Belegen des zugehörigen Locks befahren werden. Außerdem gilt die Regel, dass Zielpunkte von Bewegungen immer

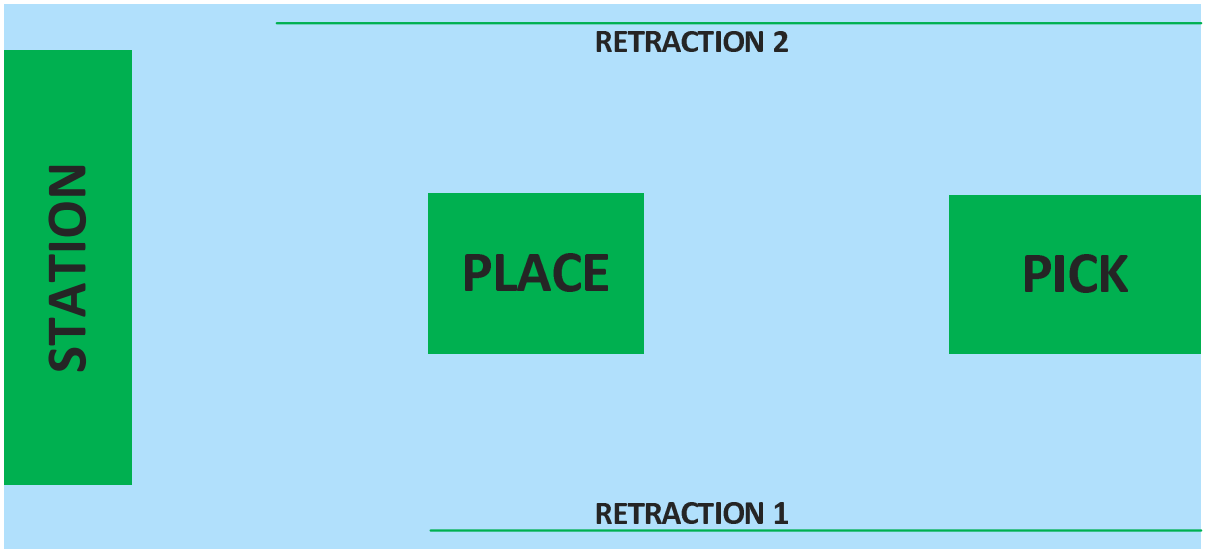


Bild 4.3: Die verschiedenen Arbeitsbereiche der Zelle in der Draufsicht.

innerhalb von Bereichen liegen müssen. Bei der folgenden Behandlung von Arbeitsbereichen wird der Roboter immer als Punkt behandelt, indem nur der TCP[1] des Roboters betrachtet wird.

Die Bereiche wurden folgendermaßen ausgemessen: Von jedem Roboter wurde die umgebende Hülle (im Folgenden: Bounding Box) ermittelt. Anschließend wurde für Roboter 1 die y-Koordinate des Rückzugsraums bestimmt und dieser dorthin gefahren. Unter Beachtung eines Sicherheitsabstandes im Zentimeterbereich wurde Roboter 2 nun so weit wie möglich auf Roboter 1 zu bewegt. Die y-Position von Roboter 2 markiert nun die untere Kante der Bereiche Pick und Place. Analog wird die obere Kante bestimmt. Die beiden x-Werte von Pick ergeben sich aus der Aufnahmefläche der Bauteile. Die x-Koordinaten des Bereichs Station können ebenfalls ermittelt werden. Nun ergeben sich die beiden x-Werte des Place-Bereichs aus der minimalen x-Position eines Roboters, wenn der andere auf der maximalen x-Koordinate im Bereich Station steht. Analog wird die maximale x-Koordinate für Place bestimmt. Die Rückzugsbereiche verlaufen nicht über die ganze Länge der Zelle, da die y-Ausdehnung des Bahnhofbereichs größer ist, als die von Pick und Place.

Allgemein sind die Bereiche also so gewählt, dass ein Roboter, der sich innerhalb eines Bereichs bewegt, Bewegungen in anderen Bereichen nicht beeinträchtigt.

1 Tool-Center-Point

4.2.3.2 Belegen und Freigeben von Arbeitsbereichen

Soll ein Roboter in den Pick-Bereich einfahren, so muss er diesen zuvor belegen. Nach Erhalten des Locks kann er sich in diesem frei bewegen. Befindet er sich im Pick-Bereich und will in den Bereich Station, so muss er Station und Place belegen, da er Place auch überstreicht. Den Lock für Pick hält er in diesem Beispiel schon, da er sich momentan in diesem Bereich aufhält. Nachdem die Bereiche reserviert wurden, kehrt die Ausführung der `SyncOnSpaceOp`, die für das Reservieren von Bereichen zuständig ist, zurück.

Die Liste der EOPs wird weiter abgearbeitet, solange bis ein `ReleaseSpaceOp` erreicht wird. Diese EOP dient dem Freigeben der Bereiche. In regelmäßigen Intervallen fragt der Executor der `ReleaseSpaceOp` die aktuelle Position des Roboters ab. Diese Position kann auf einen Bereich zurückgeführt werden, in dem sich der Roboter momentan aufhält. Wird ein Bereich verlassen, so merkt sich der Executor den verlassenen Bereich. Wird ein neuer Bereich betreten, so kann nun der zuvor verlassene, alte Bereich freigegeben werden. Dies wird so lange wiederholt, bis der Zielbereich erreicht wird. Nach Freigeben des alten Bereichs ist die Ausführung der `ReleaseSpaceOp` beendet.

Steht der Roboter also im Pick-Bereich, belegt er zuerst die Bereiche Place und Station. Nun folgt ein `GotoPositionOp`, das ihn zur Zielposition im Station-Bereich sendet. Nach Beginn der Bewegung wird die `ReleaseSpaceOp` ausgeführt. Es wird erkannt, dass der Pick-Bereich verlassen wird. Beim Betreten des Place-Bereichs, wird der Pick-Bereich freigegeben. Place wird wieder verlassen und beim Eintreten in den Bahnhofsbereich wird Place freigegeben. Da der Bahnhofsbereich der Zielbereich der Bewegung ist (dies ist aus dem Parameter der `ReleaseSpaceOp` bekannt) ist das Freigeben der Bereiche beendet.

4.2.3.3 Verfahrwege bei der Kollisionsvermeidung

Beim Zerlegen einer COP in eine Liste von EOPs wird der Zielpunkt der COP, z.B. eine Place-Position, direkt in ein `GotoPositionOp` eingefügt. Es wird also optimistisch davon ausgegangen, dass der Zielpunkt direkt angefahren werden kann. Ist dies nicht der Fall, weil entweder ein zwischenliegender Bereich oder der Zielbereich momentan vom anderen Roboter belegt ist, so greift eine weitere Funktionalität der `SyncOnSpaceOp`. Dies ist das Ergänzen von zusätzlichen Positionen um Ausweichbewegungen zu fahren. Dabei wird die `SyncOnSpaceOp` am Beginn der Liste belassen und zuvor weitere `GotoPositionOps` und `SyncOnSpaceOps` ergänzt. Im Folgenden werden alle Ausweichbewegungen am Beispiel des Roboters 1 dargestellt.

Pick-Place-Ausweichbewegung

Steht der Roboter im Bereich Pick und soll in den Place-Bereich, welcher belegt ist, so fährt er die Ausweichbewegung, die in Bild 4.4 abgebildet ist. Am Punkt P_2 wird gewartet, bis der Place-Bereich vom anderen Roboter verlassen und freigegeben wird.

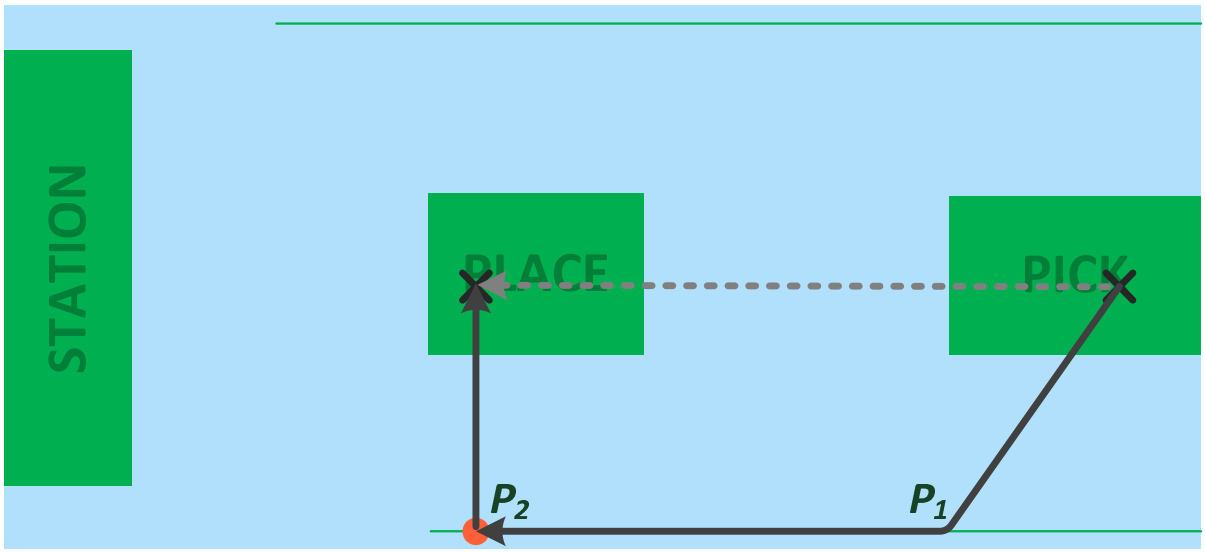


Bild 4.4: Ausweichbewegung bei der Pick-Place-Bewegung

Die Liste der EOPs verändert sich bei der Ausführung der `SyncOnSpaceOp` durch den zugehörigen Executor folgendermaßen:

Vorher:

1. `SyncOnSpaceOp`
2. `GotoPositionOp(Zielpunkt Place)`
3. ...

Nachher:

1. `GotoPositionOp(`P_1`)`
2. `GotoPositionOp(`P_2`)`
3. `SyncOnSpaceOp`
4. `GotoPositionOp(Zielpunkt Place)`
5. ...

Die momentan ausgeführte `SyncOnSpaceOp` wird also nicht aus der Liste entfernt. Zwei `GotoPositionOps` werden hinzugefügt und die Liste in diesem Zustand zurückgegeben. Somit

wird die `SyncOnSpaceOp` eigentlich zweimal ausgeführt. Bei der ersten Ausführung wird erkannt, dass Place nicht direkt angefahren werden kann und die Liste wird um die `GotoPositionOps` erweitert. Bei der zweiten Ausführung kann keine Ausweichbewegung mehr gefahren werden und es muss daher gewartet werden, bis der Place-Bereich frei ist. Die umgekehrte Ausweichbewegung von Place nach Pick verläuft analog.

Place-Station-Ausweichbewegung

Soll der Bereich Station von Place aus angefahren werden, und ist Station bei Ausführung der `SyncOnSpaceOp` belegt, so wird die Ausweichbewegung gefahren, die in Bild 4.5 zu sehen ist. Bei P_1 wird gewartet, bis der Bahnhofsbereich frei ist. Dies ist der äußerste Punkt des Rückzugsbereich der angefahren werden kann und welcher am nächsten am Bahnhofsbereich liegt.

Die Ausweichbewegung von Station nach Place ist in Bild 4.6 auf der nächsten Seite zu sehen. Hier ist der dem Ziel am nächsten gelegene Punkt im Rückzugsbereich die Projektion des Zielpunkts auf diesen. Dieser ist in diesem Beispiel P_2 benannt. Hier wird gewartet, bis der Lock für den Place-Bereich verfügbar ist.

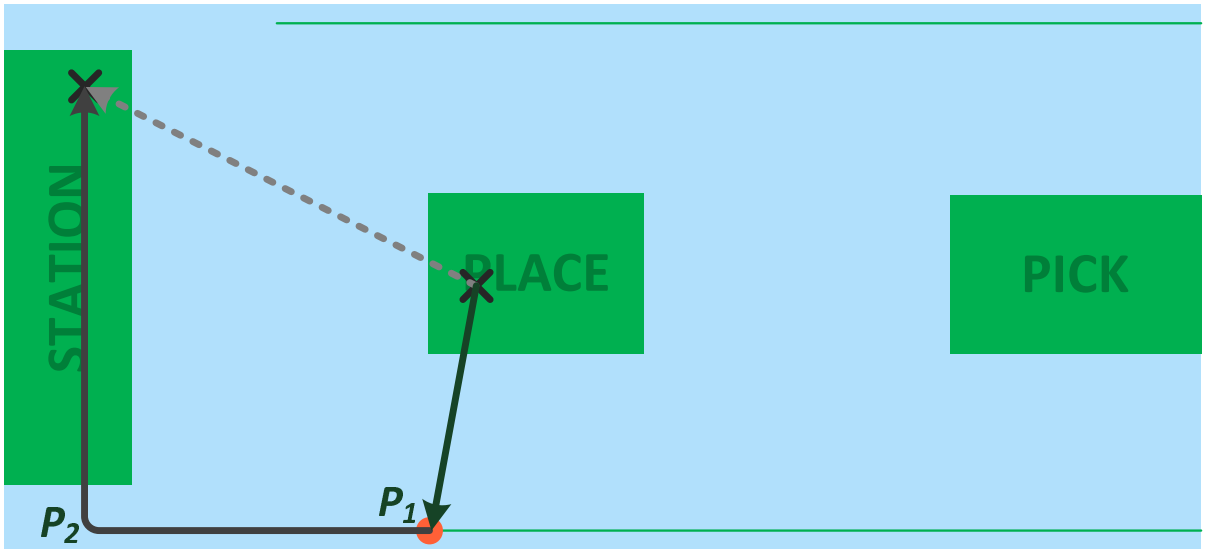


Bild 4.5: Ausweichbewegung bei der Place-Station-Bewegung

Pick-Station-Ausweichbewegung

Bild 4.7 auf der nächsten Seite zeigt die Ausweichbewegung bei der Bewegung vom Pick- zum Station-Bereich. Hier wird ähnlich wie bei der Ausweichbewegung vom Place- in den Station-Bereich am äußersten Punkt des Rückzugsbereichs gewartet.

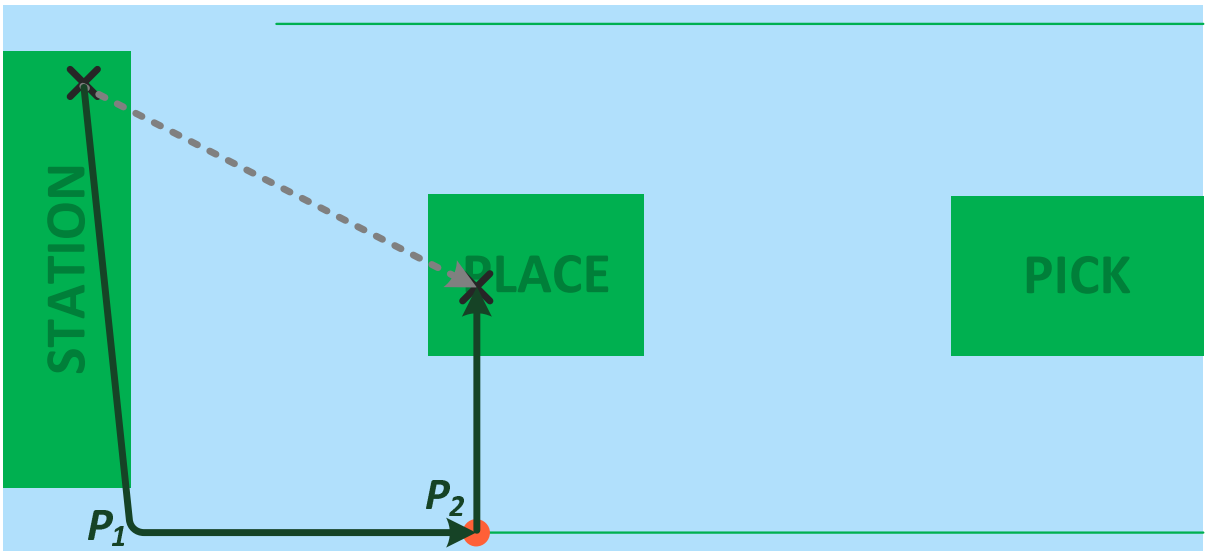


Bild 4.6: Ausweichbewegung bei der Station-Place-Bewegung

Sollte die Bewegung über den Place-Bereich hinweg direkt in den Bahnhofsbereich möglich sein, so muss gewährleistet werden, dass die Bewegung nur innerhalb des in Bild 4.8 auf der nächsten Seite dunkel eingezeichneten Korridors möglich ist. Dazu wird am Schnittpunkt des Korridors mit dem Station-Bereich ein Stützpunkt (P_1) eingefügt. Dieser wird angefahren, falls der Zielpunkt im Bahnhofbereich außerhalb des Korridors liegt. Dies muss auch bei der Bewegung vom Place- in den Station-Bereich beachtet werden.

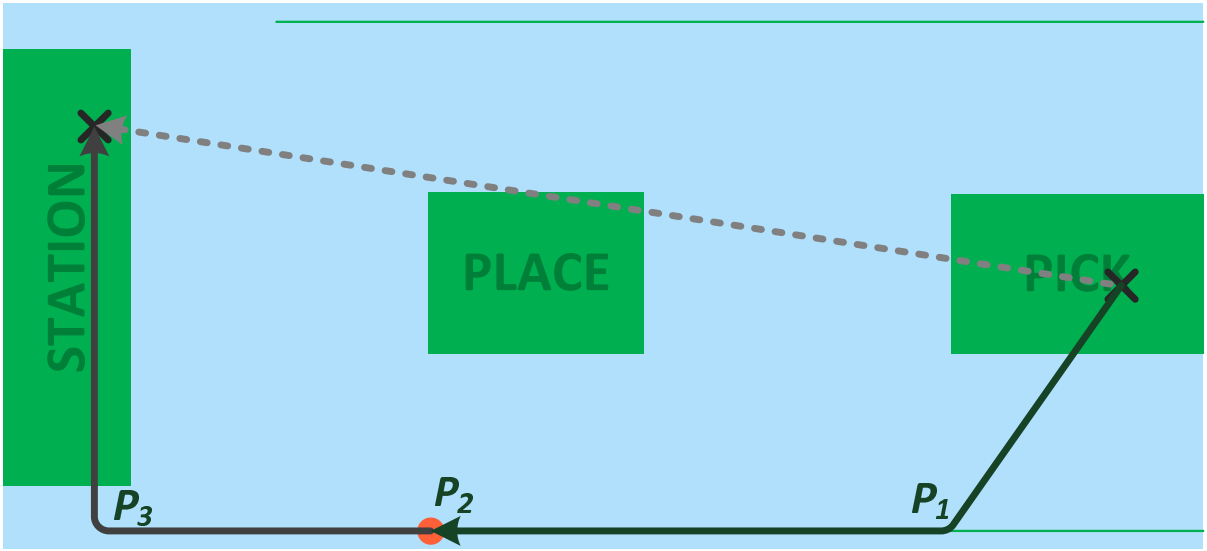


Bild 4.7: Ausweichbewegung bei der Pick-Station-Bewegung

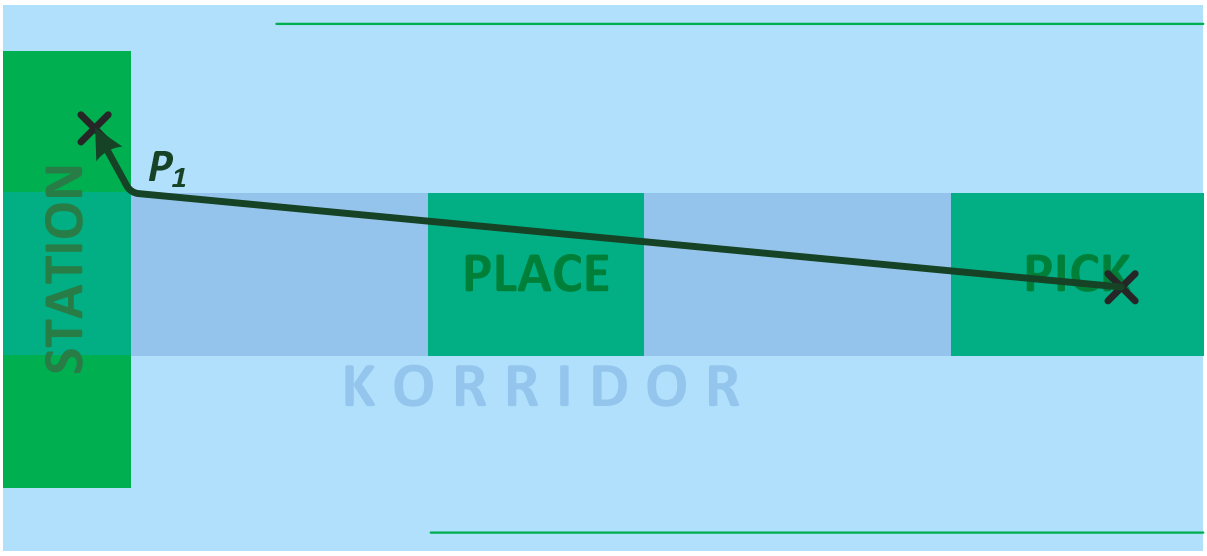


Bild 4.8: Direktbewegung Pick-Station

4.3 Zusammenfassung

Schon am Beispiel eines einzelnen Roboters konnte gezeigt werden, wie die grundlegende Montageplanung abläuft. Bei der Ausführung des Montageplaners wird eine Montageaufgabe durch Analyse des Montageabhängigkeitsgraphen ausgewählt und in Komplexoperationen zerlegt, die die logischen Schritte in der Montagesequenz abbilden. Diese werden wiederum in Elementaroperationen zerlegt, die Detailschritte kapseln und die Möglichkeit bieten, Synchronisationspunkte in der Abarbeitung einzufügen.

Werden mehrere Roboter eingesetzt, so müssen weitere Elementaroperationen eingeführt werden, die für Synchronisationsaspekte verantwortlich sind. Bei deren Ausführung wird eine Ausprägung der vier verschiedenen Ressourcentypen Bereich, Greifer, Port und Hexapod angefordert oder freigegeben. Die Listen aus Elementaroperationen aus dem Beispiel mit einem einzelnen Roboter werden um die neuen Synchronisationsoperationen ergänzt. Der wichtigste Aspekt der Synchronisierung ist dabei die Planung der Verfahrwege mittels Ausweichbewegungen zur Deadlock-Vermeidung.

5 Architekturentwurf

Dieses Kapitel behandelt die Softwarearchitektur des Bahnplaners. Zuerst werden einige Grundlagen an Architektur- und Entwurfsmustern eingeführt. Dann wird ein Überblick über die Architektur des Bahnplaners gegeben, indem die einzelnen Schichten der Software kurz vorgestellt werden. Sie werden in späteren Kapiteln genauer beschrieben. Als letztes wird die Wahl der Programmiersprache Java begründet.

5.1 Grundlagen an Architektur- und Entwurfsmuster

Generell wird zwischen Architektur- und Entwurfsmustern unterschieden. Architekturmuster bestimmen den Grundaufbau einer Anwendung, bieten also ein makroskopisches Abbild des Softwaresystems. Entwurfsmuster lösen konkrete Implementierungsprobleme und beschränken sich auf kleinere Teilgebiete des Systems. Diese entstanden, indem stetig wiederkehrende Problemstellungen während der Programmierung erkannt und generische Lösungen entwickelt wurden.

Der Bahnplaner verwendet das Architekturmuster *Schichtenmodell*, welches im Folgenden etwas genauer vorgestellt werden soll. Zwei wichtige Entwurfsmuster, die benutzt werden, sind das *Observer-Pattern* und das *Strategy-Pattern*.

5.1.1 Schichtenmodell

Das Schichtenmodell ist ein weit verbreitetes Architekturmuster. Dabei wird das Softwaresystem vertikal in mehrere hierarchische Schichten unterteilt. Nach oben hin erhöht sich der Grad der Abstraktion. Die unterliegende Schicht fungiert jeweils als Diensterbringer, die darüber liegende Schicht als Dienstnutzer. Jede Schicht stellt nach oben klar definierte Schnittstellen zur Verfügung. Die Kommunikation erfolgt immer über die Schnittstellen der unterliegenden Schicht und nie über mehrere Schichten hinweg. Die Implementierung einer Schicht ist verborgen. Sie erfüllt nur den durch die Schnittstelle geforderten Vertrag. Die Software ist leichter wartbar, da die Implementierung einer Schicht getrennt von den anderen Schichten geändert werden kann, solange die Schnittstelle gleich bleibt.

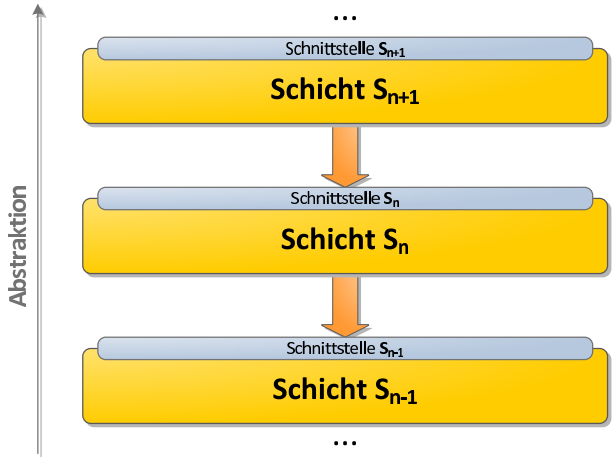


Bild 5.1: Ein Ausschnitt aus einem konzeptionellen Schichtenmodell

Ein prominentes Beispiel für eine Schichtenarchitektur ist das OSI[1]-Modell für Kommunikationsprotokolle. Hiermit können durch Austausch von tiefer liegenden Schichten, die gleichen Anwendungsprotokolle mit verschiedenen Datenübertragungsprotokollen benutzt werden.

5.1.2 Observer-Pattern

Das Observer-Pattern [3], auch Publisher-/Subscriber-Pattern genannt, bietet eine Möglichkeit, Objekte über Änderungen anderer Objekte zu informieren. Erstere werden Observer oder Beobachter genannt, letztere Subjects oder Veröffentlicher. Beobachter können sich beim Subjekt über die Methode registerObserver registrieren und werden von nun an informiert, wenn das Subjekt sich ändert. Alle Beobachter implementieren ein Interface Observer, welches die Methode notify vorschreibt. Diese Methode wird vom Subjekt für jeden registrierten Beobachter aufgerufen. Jeder

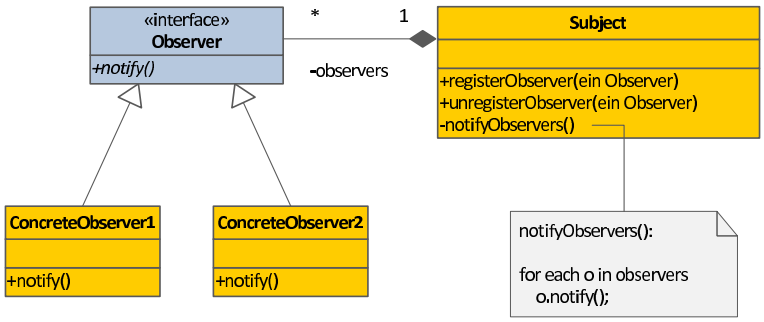


Bild 5.2: Klassendiagramm eines Observer-Patterns

1 Open Systems Interconnection

Beobachter kann nun eine konkrete Reaktion auf die Statusänderung in seiner notify-Methode implementieren.

Mit diesem Entwurfsmuster ist also eine Art *Callback* möglich. Ein Beobachter muss nicht mehr periodisch das Subjekt nach einer Statusänderung abfragen, sondern wird von diesem über eine Änderung informiert.

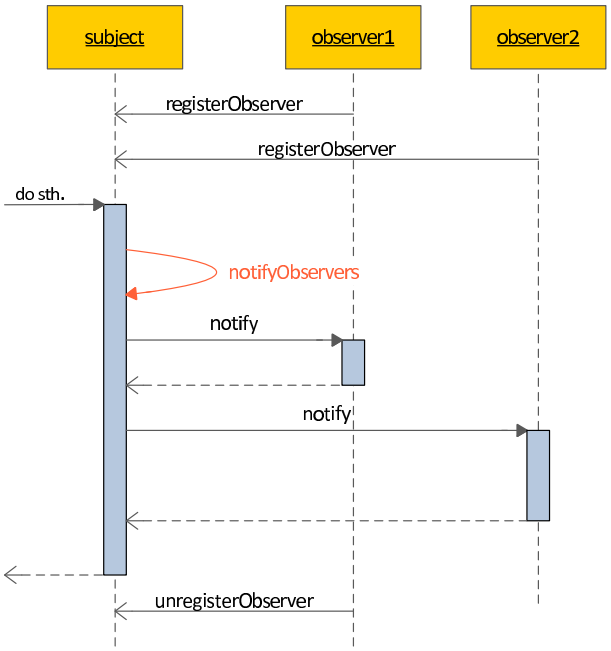


Bild 5.3: Sequenzdiagramm eines Observer-Patterns mit zwei Beobachtern

Die Reihenfolge in der die registrierten Beobachter informiert werden, ist nicht festgelegt, erfolgt aber meistens sequentiell in der Reihenfolge der Registrierung der Beobachter. Es kann auch ein Subjekt implementiert werden, das jeden Beobachter in einem eigenen Thread informiert, d.h. die notify-Methode für jeden Beobachter in einem eigenen Thread aufruft. Beobachter können sich von einem Subjekt auch wieder abmelden und werden ab diesem Zeitpunkt nicht mehr über Änderungen informiert.

5.1.3 Strategy-Pattern

Das Strategy-Pattern findet Verwendung, wenn das Verhalten eines Objekts dynamisch zur Laufzeit geändert oder ein Objekt mit verschiedenen Verhaltensweisen initialisiert werden soll. Die auszutauschende Funktionalität wird dabei in eigene Objekte ausgelagert, die alle ein gemeinsames Strategie-Interface implementieren. Das aufrufende Objekt wird dabei Kontext genannt. Der Kontext enthält eine Referenz auf eine Strategie, die bei der Initialisierung oder

auch während der Laufzeit mit einer konkreten Implementierung des Strategie-Interfaces belegt wird. Der Kontext ruft dann im Rahmen eines Ablaufs die Strategie auf.

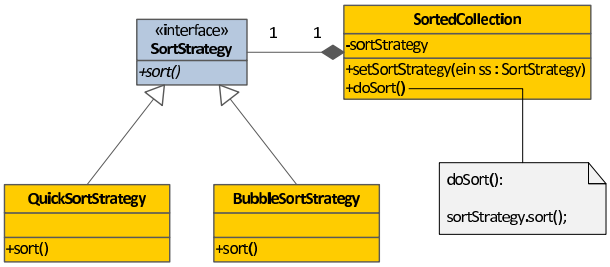


Bild 5.4: Klassendiagramm eines Strategy-Patterns

Soll eine Klasse beispielsweise eine Datenstruktur implementieren, die eine Sortierung von Elementen verlangt, so könnte der konkrete Sortieralgorithmus als eine Strategie ausgelagert werden. Das Strategie-Interface würde nun aus der Methode sort bestehen. Konkrete Implementierungen dieser Strategie würden dann diese Methode auf verschiedene Arten umsetzen.

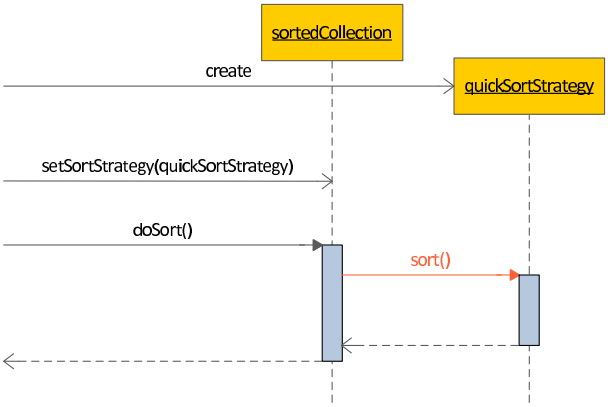


Bild 5.5: Sequenzdiagramm eines Strategy-Patterns

5.2 Überblick über die Architektur

Das Bahnplanungssystem gliedert sich in die vier Schichten *Oberfläche*, *Operation*, *Cell* und *Communication*. Die letzten drei können umfangreich konfiguriert werden. Ein Überblick wird in Bild 5.6 auf der nächsten Seite gegeben.

5.2.1 Graphische Bedienoberfläche

Die Oberflächenschicht bietet dem Benutzer eine komfortable Bedienoberfläche. Mit dieser kann er die Montagesequenz anstoßen, einzelne Tasks manuell ausführen oder Greifer parken und holen.

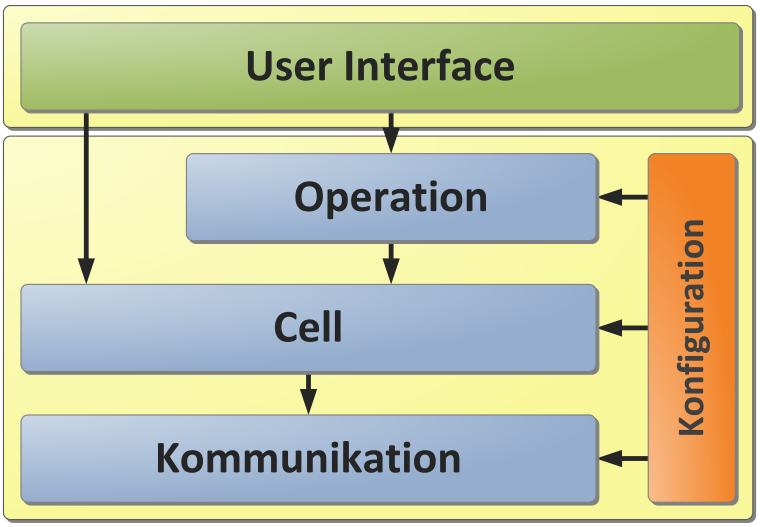


Bild 5.6: Die vier Schichten der Bahnplanungssoftware

Dazu wird auf die unterliegende *Operation-Schicht* zugegriffen. Außerdem wird der aktuelle Montagegraph und eine Übersicht über die Ressourcen dargestellt.

Des Weiteren bietet die Oberfläche direkten Zugriff auf das Hardwaremodell, das durch die *Cell-Schicht* abgebildet wird. Damit können die Druckluftventile, die Greifer öffnen und schließen oder anflanschen und die Laufbänder und Bolzen des Transfersystems gesteuert werden. Positionen können durch Zugriff auf die Cell-Schicht direkt angefahren werden.

5.2.2 Operation-Schicht und Implementierung der Bahnplanungslogik

Die Operation-Schicht verarbeitet auf höchster Ebene einen *Task*, der aus einer Menge von ausführbaren Tasks ausgewählt wird (Scheduling). Dieser wird einem Roboter dann über einen *Robot Manager* zugewiesen. Hier wird er in *Complex Operations* zerlegt, welche bei deren Abarbeitung wiederum in eine Reihe von *Elementary Operations* zerlegt werden. Elementaroperationen werden dann mit Hilfe von Executoren ausgeführt. Dazu wird auf die unterliegende Cell-Schicht und auf die Ressourcen durch *ResourceLockers* zugegriffen.

Die Operation-Schicht beinhaltet also den eigentlichen Bahnplaner und dessen Konzepte. Hier befindet sich die Ausführungslogik: Einerseits der Montagegraph und die Auswahl des nächsten auszuführenden Tasks, andererseits die Bahnplanung und die Aspekte der Kooperation, wie bereits in Kapitel 4 beschrieben.

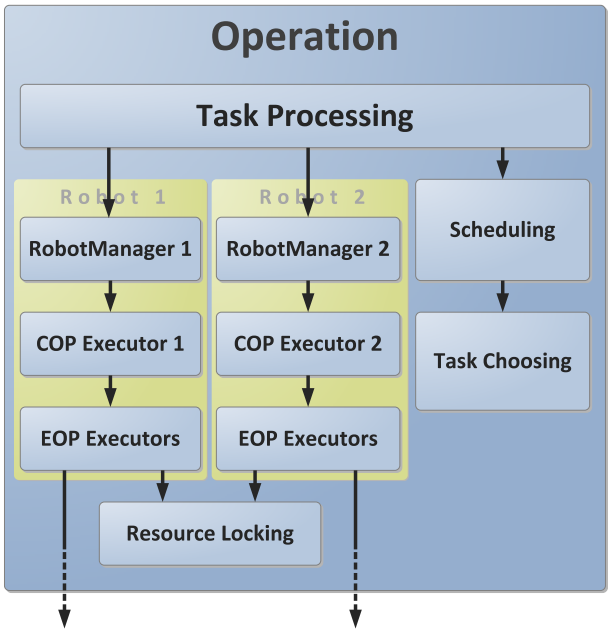


Bild 5.7: Die vier Unterschichten der Operation-Schicht

5.2.3 Cell-Schicht und Modell der Hardware

Die Cell-Schicht bildet die Hardware der Zelle als Modell ab. Es existieren Komponenten, die die Roboter, den Hexapod und das Transportsystem abbilden. Außerdem gibt es Klassen, die Greifer und den Greiferbahnhof verwalten. Diese Schicht ist nicht weiter in Unterschichten zerlegt. Die Unterteilung ist eher horizontaler Natur, da ihre einzelnen Komponenten sehr unabhängig voneinander agieren.

Diese Schicht könnte jederzeit benutzt werden, um von Grund auf einen neuen Bahnplaner zu schreiben. Es wurde sehr darauf geachtet, die Operation-Schicht mit dem Modell von Tasks,

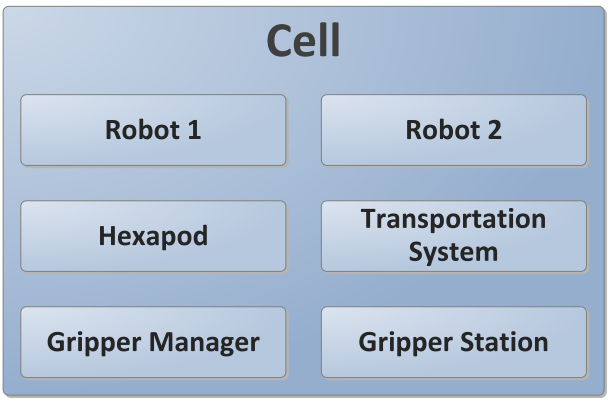


Bild 5.8: Die Untergliederung der Cell-Schicht

Komplex- und Elementaroperationen nicht mit der Cell-Schicht zur Ansteuerung der aktiven Komponenten zu vermischen, um die Austauschbarkeit der Operation-Schicht zu gewährleisten.

5.2.4 Communication-Schicht und XML-Umsetzung

Die Kommunikation mit den Robotern und dem Hexapod ist in dieser Schicht gekapselt. Die Schnittstelle nach oben bilden sog. Robotervariablen. Diese können gelesen und beschrieben werden. Die Umsetzung und Übertragung dieser Lese- und Schreibbefehle an den Roboter wird in dieser Schicht implementiert. Es können verschiedene Low-Level-Optimierungen vorgenommen werden, wie das Puffern von Variablen oder die Anpassung der Sendewiederholungen im Fehlerfall. Bestehende Fehler der Roboter im Ablauf der Kommunikation können schon in dieser Schicht ausgeglichen werden, so dass Schichten über der Communication-Schicht diese Fehlern nicht bemerken.

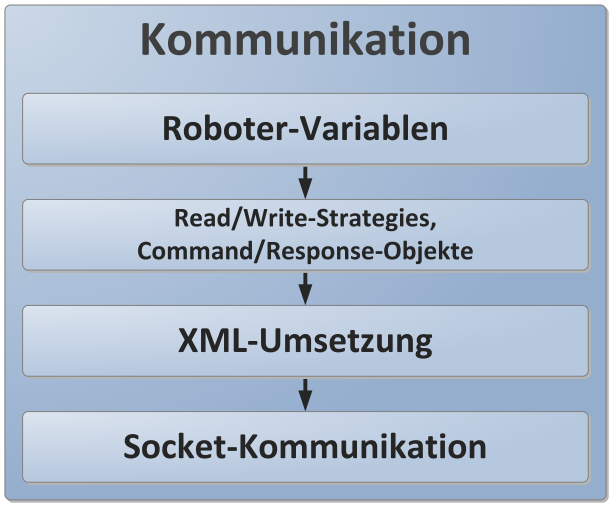


Bild 5.9: Die vier Unterschichten der Communication-Schicht

Diese Schicht bietet eine sehr klare Unterteilung in Unterschichten, die die einzelnen Schritte der Umsetzung eines Befehls in eine einfache Zeichenkette, die an den Roboter gesendet wird, abbilden. Sollte sich die momentane XML-Ansteuerung der Roboter einmal ändern, so können Unterschichten dieser Schicht einfach ausgetauscht werden.

5.3 Entscheidung für die Programmiersprache Java

Der Bahnplaner wurde komplett mit Java entwickelt. Die Begründung dafür liegt vor allem in der *Einfachheit und Portabilität* von Java. Es ist mit Java dank der Einfachkeit der Sprache, Garbage

Collection und sehr guter IDE[1]-Unterstützung schneller und leichter als mit anderen Sprachen möglich, korrekten Code zu schreiben. Java bietet eine sehr umfangreiche Standardbibliothek. Außerdem gibt es eine Fülle an Bibliotheken für Aspekte wie *Logging, Konfiguration, XML-Verarbeitung und Graphstrukturen*. Mit *Eclipse* steht eine IDE zur Verfügung, die dank Debugger, Code-Completion, Refactoring, Oberflächeneditor, SVN-Plugin und Aufgabenverwaltung sehr effizientes Programmieren ermöglicht. Mit JUnit können Modultests geschrieben werden, mit dem Spring Framework wird die Applikation konfiguriert. Außerdem sind Java-Applikationen plattformunabhängig. Somit kann die Anwendung unter Linux genauso entwickelt und betrieben werden, wie unter Windows. Diese Gründe gaben den Ausschlag für die Entwicklung mit Java und der Entwicklungsumgebung Eclipse. Als *Bibliotheken* wurde Spring zur Initialisierung der Anwendung, Apache Commons Log4j zum Logging, JUNG für Graphstrukturen und JUnit für Modultests verwendet.

1 Integrated Development Environment

6 Graphische Bedienoberfläche

Um den Bahnplaner komfortabel anzusteuern und auch darüber hinaus Kontrolle über die Zelle zu haben, wurde eine Bedienoberfläche mit Java Swing erstellt. Mit dieser Oberfläche kann die Demonstrations-Applikation einfach initialisiert und gestartet werden. Über eine Übersicht ist die einfache Kontrolle des Ablaufs möglich. Hiermit kann die Position der Roboter, die belegten Bereiche und andere Ressourcenbelegungen eingesehen werden. Des weiteren bietet die graphische Oberfläche die Möglichkeit, in die Arbeit die Logik des Bahnplaners einzugreifen, indem Ressourcen gelockt oder bestimmte Tasks vergeben werden. Darüber hinaus können auch andere Aufgaben wie der Wechsel des aktuellen Greifers oder das Anfahren einer bestimmten Position ausgeführt werden. Eine weitere Sicht bietet die manuelle Kontrolle über die binären Ein- und Ausgänge des Roboters, womit das Transfersystem gesteuert und Sensoren ausgelesen werden können.

6.1 Aufbau der Oberfläche

Die Bedienoberfläche untergliedert sich in verschiedene Sichten, die über die Karteikarten am oberen Rand aufgerufen werden können. Unterhalb der Karteikarten findet sich ein gemeinsamer Bereich, von dem aus die Montagedemo initialisiert und ausgeführt werden kann. Auch können einzelne Tasks manuell durch Eingabe ihrer Nummer gestartet werden. Ebenfalls gibt es hier Buttons, die mit beliebigen Tests belegt werden können und einen Button für die Initialisierung einer weiteren Demo.

Wichtig ist der Schieberegler, der nach Klick auf `wait until sending command` aktiviert wird. Hiermit kann angegeben werden, dass jeder Befehl, der ein Senden eines Kommandos an den Roboter nach sich zieht um eine bestimmte Zeit verzögert gesendet wird. Ein verzögerbares Kommando ist in der Oberfläche mit dem Hinweis `(delayed)` markiert.

6.2 Die Sicht „robot information“

Die erste Karteikarte ist eine allgemeine Übersicht über die Position der Roboter und die Reservierung der Bereiche. Dargestellt ist eine Draufsicht auf die Zelle mit den einzeichneten Bereichen und den Bounding Boxen der Roboter. Sie ist in Bild 6.1 dargestellt.

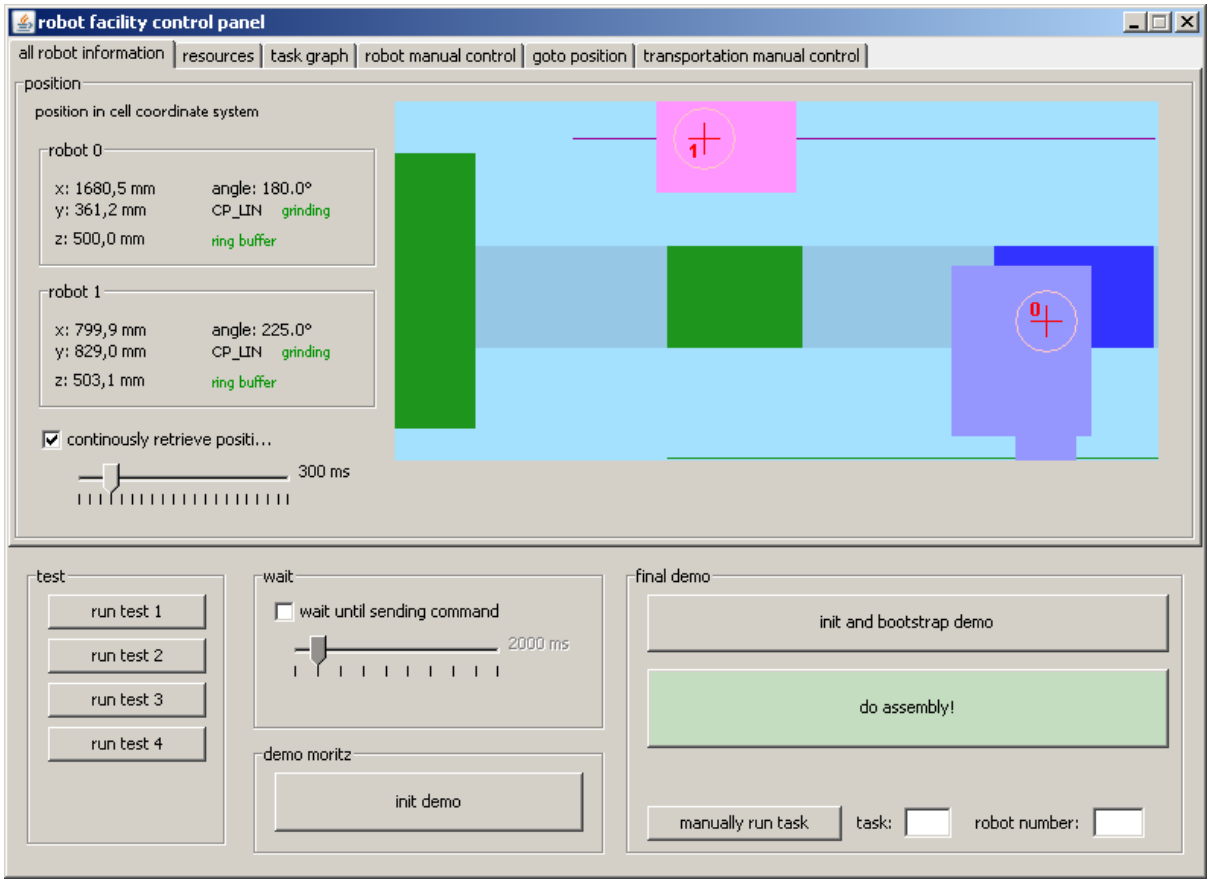


Bild 6.1: Übersicht über die Position der Roboter und belegte Bereiche

Ist die Checkbox `continously retrieve positions` aktiviert, so wird die Position der beiden Roboter in regelmäßigen Intervallen, die mit dem Schieber eingestellt werden abgefragt. Ansonsten wird die Postion der Roboter in der Grafik nicht aktualisiert. Die x-, y- und z-Koordinaten und der Drehwinkel wird für jeden Roboter auf der linken Seite angezeigt. Der graphische Bereich rechts davon bildet die Zelle ab. Die verschiedenen Arbeitsbereiche sind grün eingezeichnet. Die Bounding Box des unteren Roboters wird pink, die des oberen blau dargestellt. Die TCPs sind durch ein Kreuz markiert um das ein Kreis gezeichnet ist. Wird mit der Maus in diesen Kreis und dann auf eine weitere Position außerhalb dieses Kreises geklickt, so bewegt sich der Roboter an die angeklickte Stelle. Eine Bewegung des Mausrads innerhalb des Kreises verfährt die z-Achse des jeweiligen Roboters. Diese Kommandos werden direkt an die Cell-Schicht weitergegeben. Die Operation-Schicht wird übergangen, so dass unabhängig vom Bahnplaner Postionen gesendet werden können. Somit werden belegte Arbeitsbereiche nicht berücksichtigt, da diese in der Cell-Schicht unbekannt sind. Wird ein Bereich von Roboter 1 reserviert, so wird dieser violett

eingefärbt. Ein von Roboter 2 belegter Bereich wird marineblau gezeichnet. Ein grüner Bereich indiziert somit keine Belegung durch einen Roboter.

6.3 Die Sicht Ressourcen

Die Karteikarte `resources` gibt einen Überblick über alle im System registrierten Ressourcen. Vier Listen zeigen die Ressourcen „Greifer“, „Ports“, „Hexapod“ und „Bereiche“ an. Alle Ressourcen des jeweiligen Typs werden aufgelistet. Ist eine Ressource von einem Roboter belegt, so erscheint hinter dieser der Name des Roboters und die Zeile wird in der Farbe des Roboters eingefärbt: blau für Roboter 1, violett für Roboter 2. In einem weiteren Feld werden die Ports des Greiferbahnhofs und die jeweils in ihnen geparkten Greifer aufgeführt. Eine zweite Liste darunter zeigt explizit den Aufenthaltsort jedes Greifers an. Ändert sich an der Ressourcenbelegung etwas, so wird diese Sicht automatisch aktualisiert.

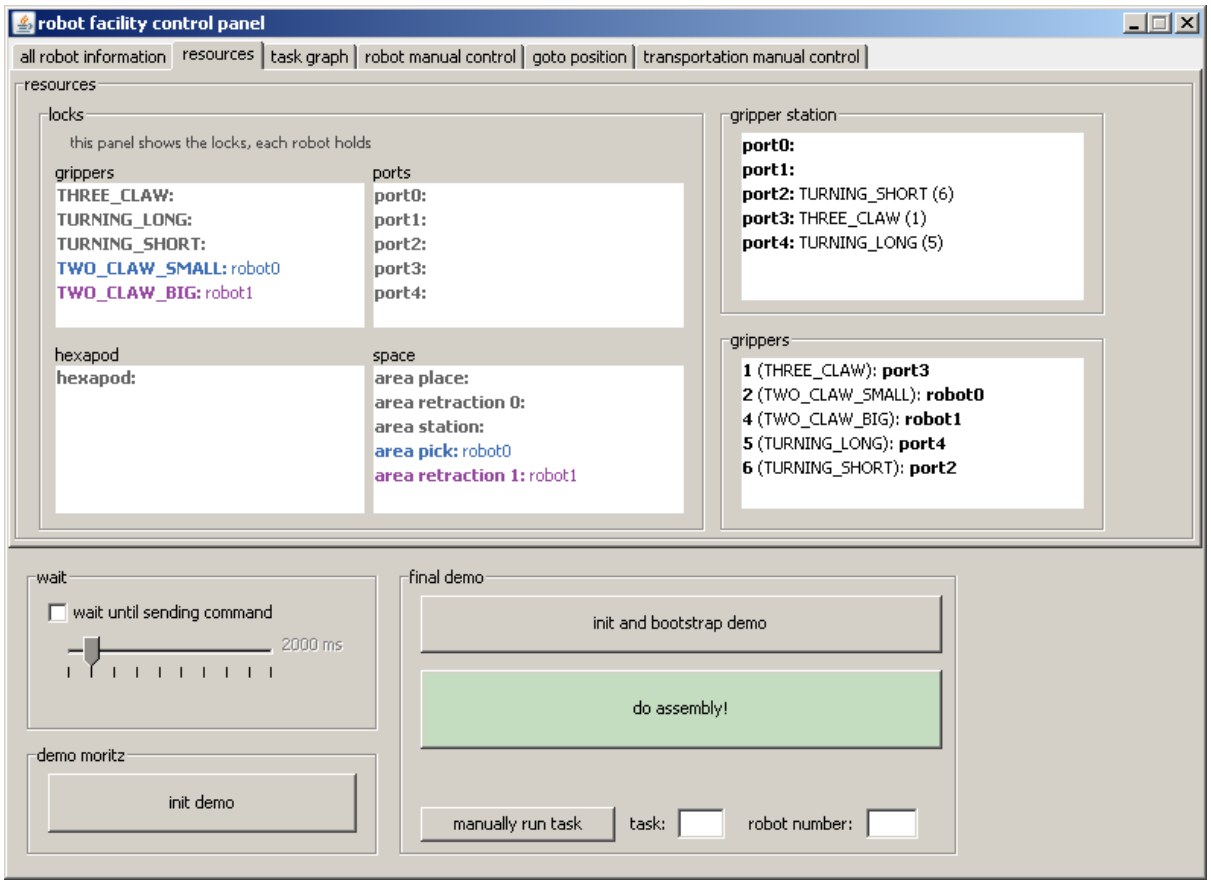


Bild 6.2: Die Ressourcen-Sicht

6.4 Die Montagegraph-Sicht

In dieser Sicht wird der Montagegraph dargestellt. Das Layout des Graphen wird von der Bibliothek `JUNG` übernommen. Grüne Knoten markieren ausführbare Tasks, blau werden Tasks angezeigt, die gerade in Ausführung sind. Graue Tasks können momentan nicht ausgeführt werden, da sie von anderen Tasks abhängen, die noch nicht fertig abgearbeitet wurden. Abgeschlossene Tasks werden aus dem Graphen entfernt.

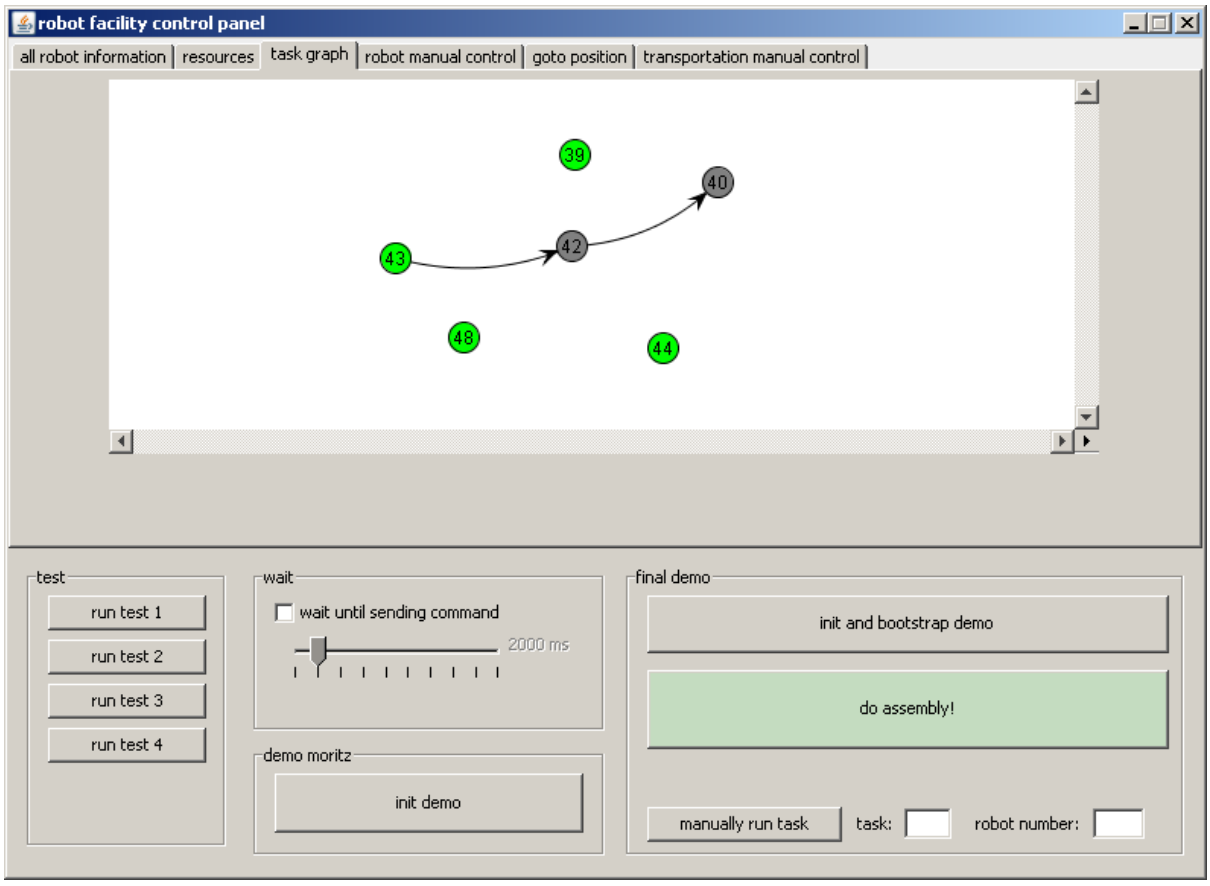


Bild 6.3: Der Montagegraph mit Tasks als Knoten

Wird mit der Maus über einen Knoten gefahren, so erscheint als Tooltip der Name des Bauteils und weitere Informationen zum Task. Durch Klicken und Ziehen der Maus wird der Ausschnitt des Montagegraphen verschoben. Zwischen den beiden Scrollleisten befindet sich ein kleiner Pfeil. Wird auf diesen geklickt, so kann der Bearbeitungsmodus ausgewählt werden. In diesem kann ein Knoten per Drag-and-Drop verschoben und das Layout somit manuell beeinflusst werden.

6.5 Die Sicht „robot manual control“

Diese Sicht bietet die Möglichkeit einen Roboter Aufgaben manuell erledigen zu lassen und in die Logik des Bahnplaners einzugreifen. Dies dient dem Herstellen von Initialbedingungen, der manuellen Handhabung der Roboter und dem Debugging. Mit dem Button `open gripper` kann der aktuell angehängte Greifer geöffnet und geschlossen werden. Mittels `unmount gripper` wird der aktuelle Greifer abgeworfen, bzw. ein Neuer aufgenommen. `Goto start` schickt den Roboter in eine festgelegte Startposition. Diese Buttons greifen direkt auf die Cell-Schicht zu und umgehen die Operation-Schicht.

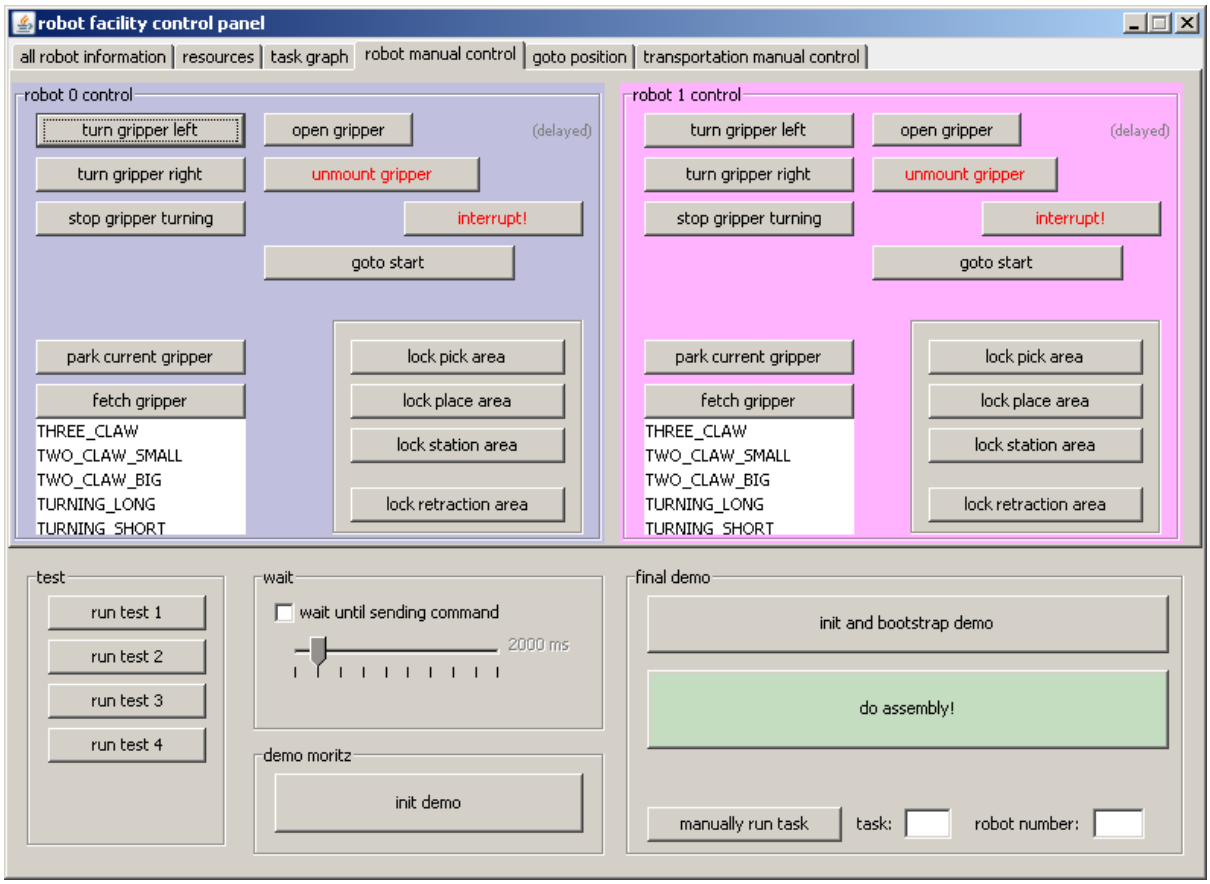


Bild 6.4: Die Sicht „robot manual control“

`Interrupt!` unterbricht die Verarbeitung des momentanen Tasks. Im Falle eines Assembly-Tasks, wird dieser nach der Unterbrechung nicht aus dem Montagegraphen entfernt, sondern als nicht bearbeitet markiert. Mittels vier Buttons können die jeweiligen Bereiche explizit belegt bzw. freigegeben werden. Dies ist zum Debuggen praktisch, da so eine Initialkonfiguration der Bereichsbelegung eingestellt werden kann. Zwei weitere Buttons und die zugehörige Liste bieten die Möglichkeit, bestimmte Greifer zu holen oder den aktuell angeflanschten Greifer in den

nächstgelegenen Port zu parken. Diese beiden Aktionen sind Beispiele für einen *Manual-Task*, der genauso wie ein Assembly-Task in der Operation-Schicht in eine Reihe von Elementaroperationen zerlegt wird.

6.6 Die Sicht „goto position“

Mit Hilfe dieser Sicht können den beiden Robotern direkt Positionen im Cell- oder World-Koordinatensystem geschickt werden. Diese werden direkt an die Cell-Schicht gegeben. Somit werden belegte Bereiche ignoriert, da der Cell-Schicht die Abstraktion „Arbeitsbereich“ nicht bekannt ist. Diese Sicht kann beispielsweise zum Senden von Positionen zur Ausmessung der Place- oder der Port-Position benutzt werden.

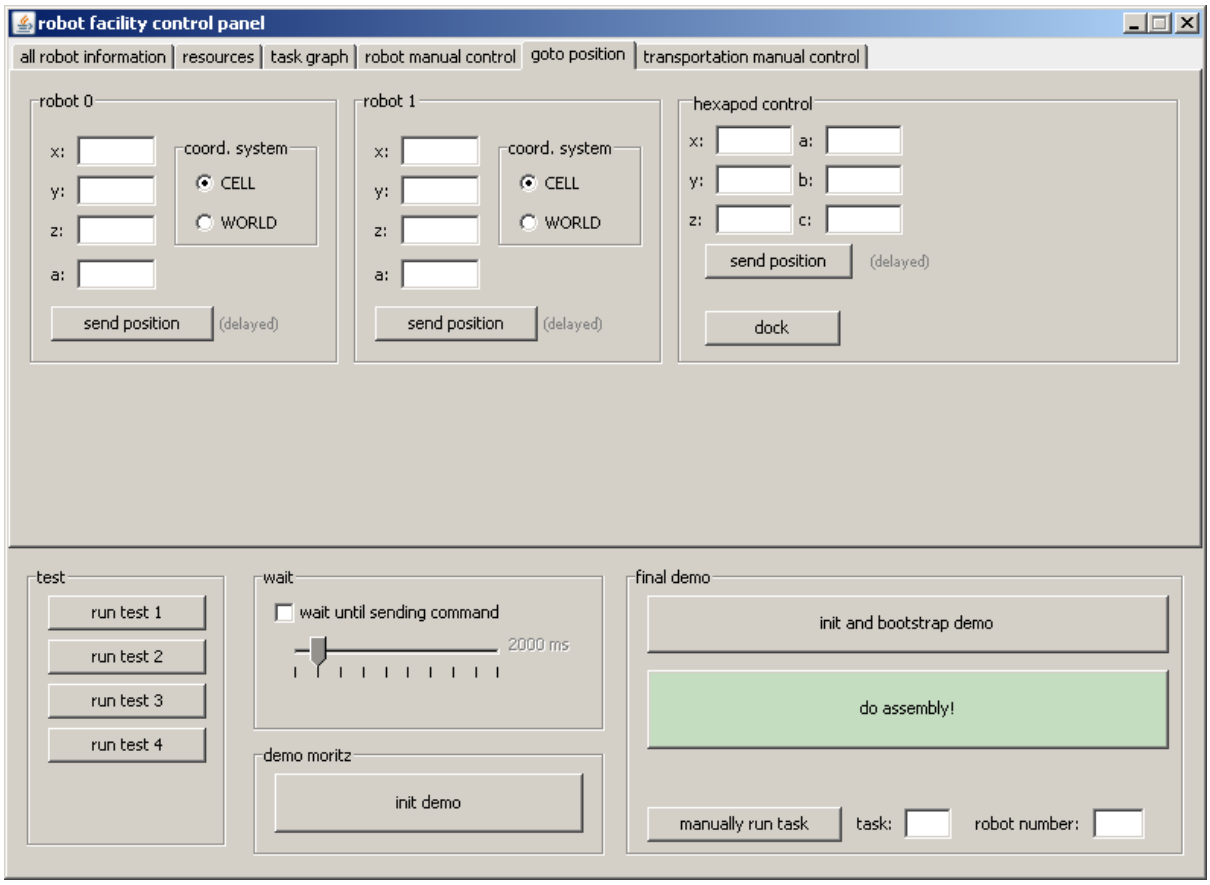


Bild 6.5: Die Sicht „goto position“

Auch dem Hexapod können Positionen geschickt werden. Außerdem bietet der Button `dock` die Möglichkeit, den Hexapod über mehrere Zwischenschritte an das Transportband „anzudocken“, so dass eine Palette vom Förderband auf den Hexapod transportiert werden kann.

6.7 Die Sicht für das Transportsystem

Um die verschiedenen Bit-Ein- und -Ausgänge der Roboter anzusteuern, mit denen das Transportsystem verbunden ist, wurde diese Sicht programmiert. Angeschlossen an diese Ein- und Ausgänge sind unter anderem die Förderbänder, so dass diese hiermit in beide Richtungen bewegt und angehalten werden können. Die Sicherheits- und Hebebolzen werden durch Klick auf den jeweiligen Umschalt-Button aktiviert. Um die verschiedenen Sensoren abzufragen, gibt es einen Button `update sensors`. Mittels Checkboxen wird dargestellt, ob die Sensoren 1 bis 5 aktiviert sind.

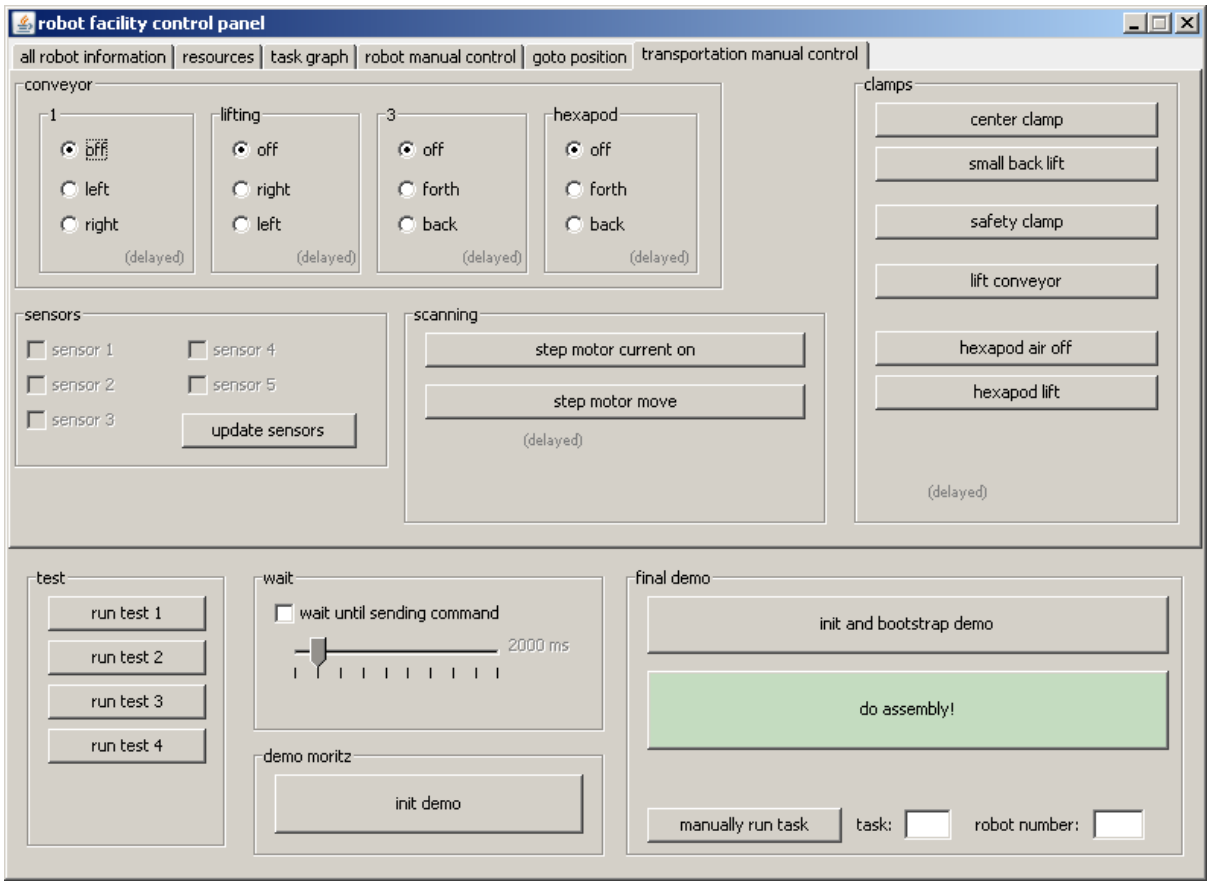


Bild 6.6: Die Sicht für das Transportsystem

7 Operation-Schicht und Implementierung der Bahnplanungslogik

In der Operation-Schicht kapselt sich die Logik der Bahnplanung. Hier wird ein Task ausgewählt, in verschiedene COPs zerlegt, die wiederum jeweils in eine Liste von EOPs zerlegt werden. Die Elementaroperationen werden dann unter Berücksichtigung der Synchronisierung über Ressourcen ausgeführt.

Dieses Kapitel beschreibt die softwaretechnische Umsetzung der Bahnplanungslogik. Die Architektur dieser Schicht soll vorgestellt und die entworfenen Klassen dokumentiert werden. Mittels dieses Kapitels soll die Weiterentwicklung des Bahnplaners oder die Adaption auf andere Zellen erleichtert werden.

7.1 Die Klasse TaskProcessor und das Scheduling der Tasks

Die Klasse TaskProcessor ist die zentrale Instanz zur Verarbeitung von Tasks. Die Ausführung von manuellen und Assembly-Tasks teilt sich auf die beiden Klassen ManualTaskProcessor und AssemblyTaskProcessor auf. Außerdem stehen einige Methoden zur Initialisierung der Demos bereit. Beide Task-Ausführungsklassen erhalten im Konstruktor eine Liste von Objekten vom Typ RobotManager, die für die Ausführung eines Tasks auf einem Roboter zuständig sind. ManualTasksProcessor bietet zwei Methoden um einen einzelnen bzw. eine Liste von ManualTasks auszuführen.

Die Klasse AssemblyTaskProcessor enthält ebenfalls eine Liste von RobotManager, die Methode doAssembly zum Anstoßen der Demo und eine Instanz von TaskScheduler. Diese ist für das *Scheduling der AssemblyTasks* zuständig. Neben Methoden zur Initialisierung bietet das Interface TaskScheduler die Methode getNextTask, die aufgrund von übergebenen Informationen über den zu bedienenden Roboter den nächsten Task auswählt. Über removeTask kann ein Task nach seiner Ausführung aus dem Graph gelöscht werden. HasMoreTasks überprüft, ob noch weitere Tasks zur Abarbeitung bereit stehen.

Die Klasse DependencyGraphScheduler implementiert dieses Interface und nutzt zur Auswahl der möglichen Tasks einen Montagegraphen. Im Konstruktor werden die zwei Interfaces AssemblyTaskReader und TaskDependencyReader übergeben, die die Knoten (die AssemblyTasks) und die Kanten (die Abhängigkeiten zwischen den Tasks) des Montagegraphen einlesen. Verschiedene

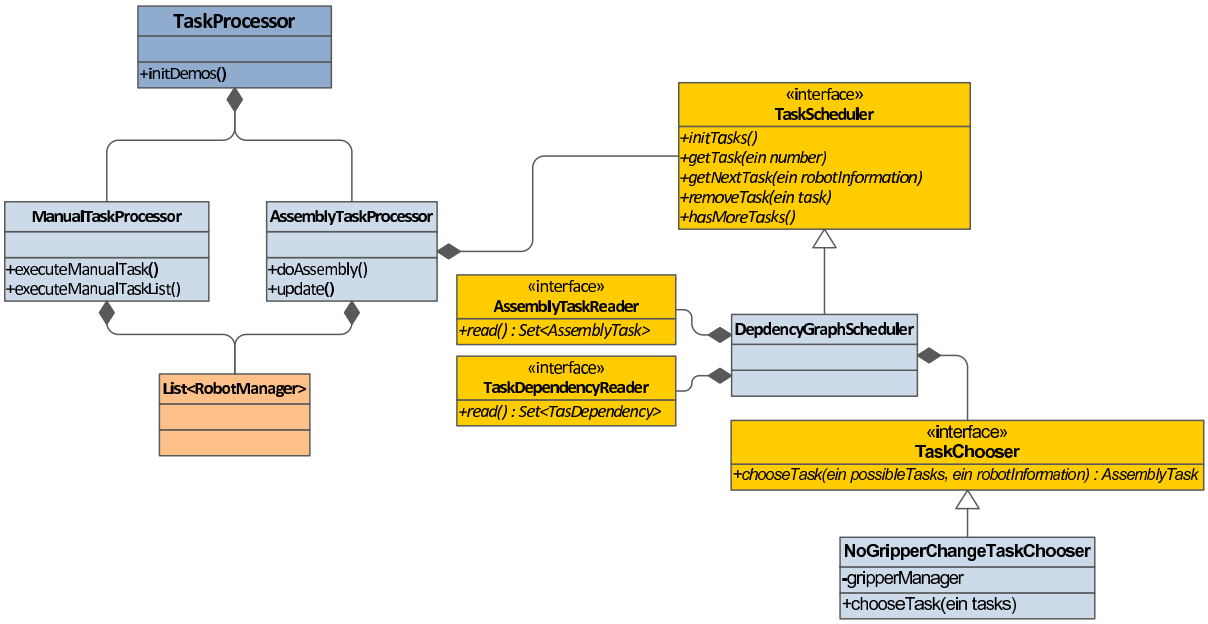


Bild 7.1: Die Klassenstruktur des Softwaremoduls für die Vergabe und Ausführung von Tasks

Implementierungen dieser Interfaces lesen die Informationen auf unterschiedliche Arten ein. Erwähnt sei hier die Implementierung CombinedAssemblyTaskReader, die ein sehr flexibles Einlesen der verschiedenen Bestandteile eines Tasks aus unterschiedlichen Quellen bietet. So können die Place-Positionen aus einer anderen Datei als die Komponentendaten oder die Pick-Positionen gelesen werden. Beliebige, eingelesene Positionen können durch andere überschrieben oder um einen bestimmten Offset verschoben werden. Das Einlesen der Pick-Positionen und deren Verarbeitung ist über das Proxy-Entwurfsmuster realisiert.

Beim Aufruf der Methode getNextTask von TaskScheduler läuft folgendes ab: Aus allen Tasks im Graph werden diejenigen ausgewählt, die keine Abhängigkeit von weiteren Tasks haben (Quellen) und die momentan nicht schon von anderen Robotern ausgeführt werden. Diese Menge an AssemblyTasks wird nun einer Implementierung des Interfaces TaskChooser übergeben, welcher den geeignetsten Task auswählt und zurück liefert. Dieser wird von der Methode getNextTask als der nächste auszuführende Task zurück geliefert.

Die momentan einzige Implementierung von TaskChooser ist NoGripperChangeTaskChooser. Der aus dem Interface TaskChooser überschriebenen Methode chooseTask werden die möglichen Tasks und Informationen zum anfordernden Roboter übergeben. Aufgrund dieser Informationen wird der am besten passende Task ausgewählt. Dies geschieht durch einen Vergleich des momentan angeflanschten Greifers mit dem Greifer, der vom jeweiligen Task benötigt wird. Es wird ein Task ausgewählt, der keinen Greiferwechsel benötigt. Ist dies nicht möglich, so wird ein Task

gewählt, der einen Greifer benötigt, der momentan nicht vom anderen Roboter belegt ist. So wird versucht, die Gesamtmontagezeit klein zu halten.

Weitere Implementierungen des Interfaces TaskChooser können bessere Strategien zur Auswahl eines Tasks aus der Menge der Möglichen bereitstellen. Die Implementierung des Task-Schedulings und der Task-Auswahl ist ein Beispiel für die Verwendung des Strategie-Patterns.

Der AssemblyTaskProcessor fungiert als *Publisher* für Änderungen und implementiert somit die Subjektrolle des Observer-Patterns. Beobachter der Oberflächenschicht können sich registrieren und werden über Änderungen im Montagegraph (Löschen eines AssemblyTasks, Ausführung eines neuen Tasks) benachrichtigt. So kann die Oberfläche automatisch aktualisiert werden.

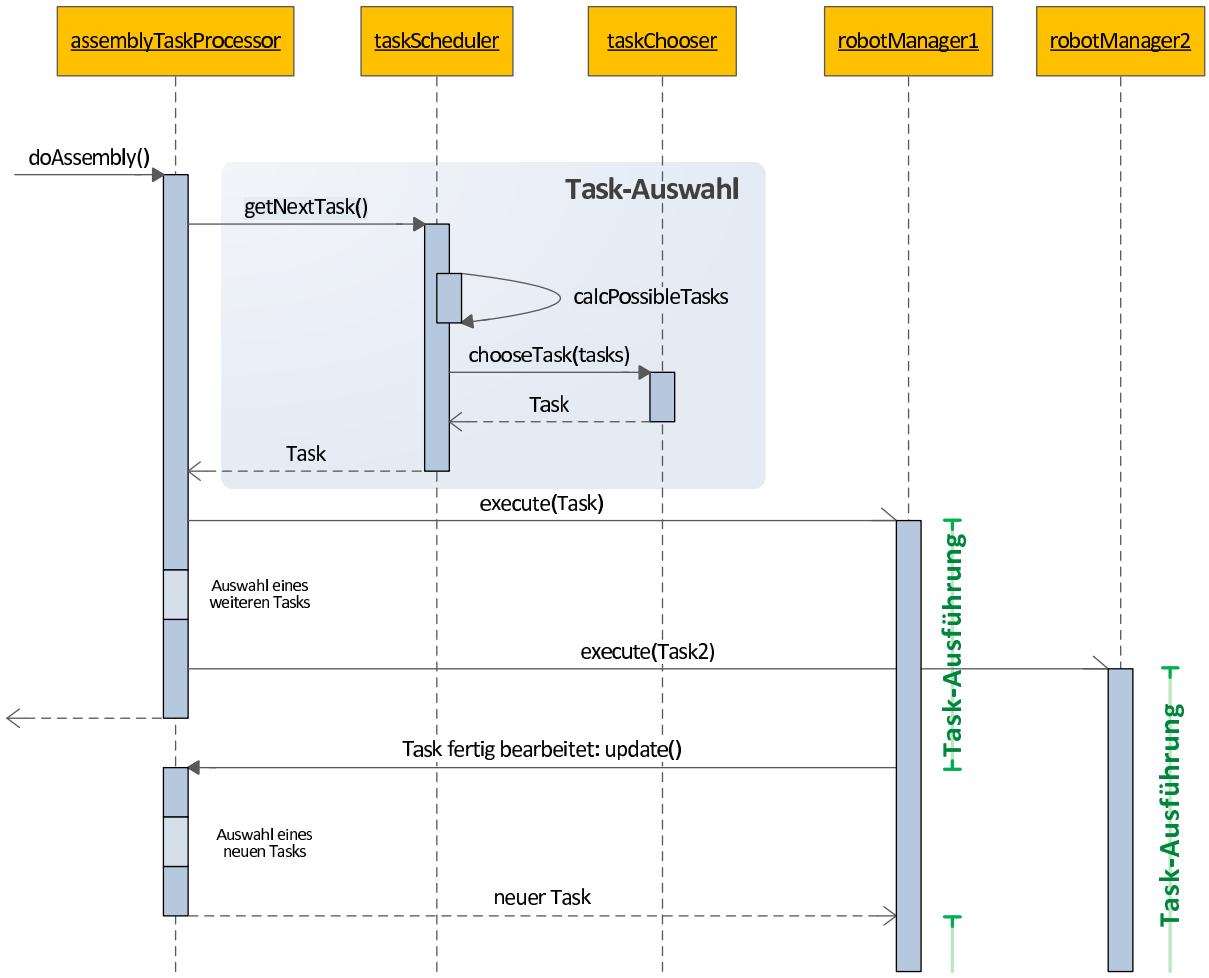


Bild 7.2: Der Ablauf des Task-Scheduling und der Task-Ausführung

Gleichzeitig tritt der AssemblyTaskProcessor als Subscriber für einen RobotManager auf. Dieser meldet Statusänderungen, z.B. die fertige Abarbeitung eines Task, indem er die update-Methode des AssemblyTaskProcessors aufruft. So kann der RobotManager den AssemblyTaskProcessor über die fertige Abarbeitung eines Tasks informieren. Wurde der Task regulär abgearbeitet (und

nicht durch Klicken des `interrupt`-Buttons der Oberfläche unterbrochen), wird er aus dem Montagegraphen entfernt. Hat der RobotManager keine weiteren Tasks abzuarbeiten, wird im AssemblyTaskProcessor der nächste Task bestimmt und dem RobotManager übergeben. So wird sichergestellt, dass ein neuer Task ausgewählt wird, sobald der alte abgearbeitet wurde. Nur die beiden ersten Tasks werden beim Aufruf von doAssembly explizit vergeben. Nachdem jeder Roboter den Task erhalten hat, kehrt die Methode doAssembly zurück. Die weiteren Tasks werden nach einem Callback von den beiden RobotManager-Threads mittels des Beobachterentwurfsmusters vergeben, sobald diese einen Task fertig ausgeführt haben. Dieser Ablauf der Auswahl eines Tasks und dessen Ausführung durch den RobotManager ist in Bild 7.2 auf der vorherigen Seite illustriert.

7.2 Die Klassen RobotManager und TaskDemuxer

Ist ein nächster Assembly-Task bestimmt oder soll ein Manual-Task ausgeführt werden, so wird jeweils die Methode executeTask von RobotManager mit dem Task als Parameter aufgerufen. Für jeden Roboter existiert eine Instanz der Klasse RobotManager. Jede dieser Instanzen läuft in einem eigenen Thread, der vom TaskProcessor bei der Initialisierung gestartet wird. Die Methode executeTask setzt den übergebenen Task an das Ende einer internen Queue dieses RobotManagers und springt zurück. Es wird also nicht gewartet, bis der Task ausgeführt wurde, sondern asynchron zurückgekehrt.

Jedem RobotManager ist, wie schon erwähnt, genau eine Instanz von Robot (siehe Abschnitt 8.1 auf Seite 81) zugeordnet. Außerdem können sogenannte TaskHooks gesetzt werden. Es gibt zwei Arten: PreExecuteTaskHook, der vor und PostExecuteTaskHook, der nach Abarbeitung eines Tasks ausgeführt wird. Jeder abzuarbeitende Task wird zuerst mit Hilfe des TaskDemuxers in eine Folge an COPs zerlegt. Diese werden dann der Reihe nach vom COPExecutor ausgeführt.

7.3 Die Klassen COPExecutor und COPDemuxer

Der COPExecutor zerlegt die aktuell bearbeitete COP mit Hilfe des COPDemuxers in eine Reihe von Elementaroperationen. Für jeden Typ von Elementaroperation gibt es einen EOPExecutor, der die EOP ausführt. Eine Liste aller EOPExecutors ist beim COPExecutor registriert. Für jede Elementaroperatione muss es genau einen Executor geben, der diese ausführt. Jeder RobotManager hat einen eigenen COPExecutor.

Ist der passende EOPExecutor aus der Liste bestimmt, wird ihm die aktuelle EOP-Liste übergeben. Es wird immer die gesamte Liste zur Bearbeitung an den EOPExecutor gereicht, da er zwar das

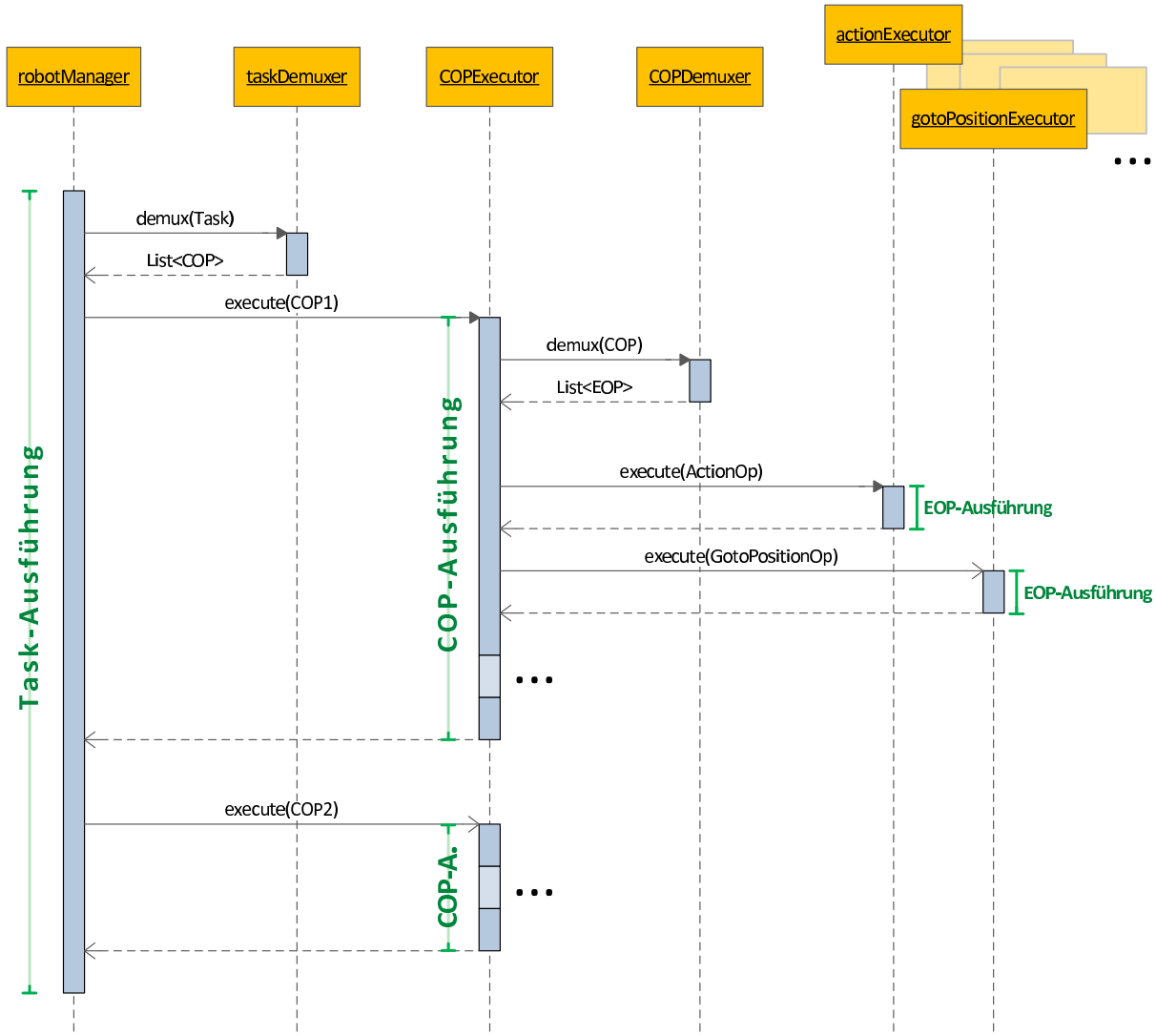


Bild 7.3: Der Ablauf der Zerlegung eines Tasks in EOPs und deren Ausführung

erste Element abarbeitet, jedoch auch weitere EOPs der Liste verändern kann. Die Ausführung und Zerlegung eines Tasks ist in Bild 7.3 dargestellt.

7.4 Die verschiedenen EOPExecutors

Das Interface EOPExecutor schreibt zwei Methoden vor: canExecute, der eine Liste von EOPs übergeben wird und die überprüft, ob diese Liste mit diesem Executor ausgeführt werden kann. Des Weiteren gibt es die Methode execute, die eine Liste von Elementaroperationen ausführt. Diese Methode gibt die Liste der EOPs nach der Ausführung zurück.

7.4.1 ActionExecutor

Die Klasse ActionExecutor ist in der Lage, alle `ActionOps` auszuführen. Jedem ActionExecutor ist eine Referenz auf Robot und den zentralen GripperAttacher zugeordnet (siehe Abschnitt 8.4 auf Seite 87). Mit Hilfe dieser beiden Abhängigkeiten können Greifer geöffnet oder geschlossen und angeflanscht oder abmontiert werden. Nach jeder dieser Operationen wird eine kurze Zeit gewartet um der Pneumatik Zeit für die Ausführung des Vorgangs zu geben.

Das Ändern der Geschwindigkeit, das eine `ChangeVelocityOp` anstößt, wird durch Setzen einer Variable von Robot erreicht. Bei der Ausführung der Operationen `MountGripperOp` und `UnmountGripperOp` wird dafür Sorge getragen, dass der aktuelle Aufenthaltsort des Greifers in der Klasse GripperAttacher aktualisiert wird.

7.4.2 HexapodExecutor

Alle `HexapodOps` werden durch den HexapodExecutor ausgeführt. Dies beschränkt sich momentan auf die `HexapodGotoPositionOp`, die eine Hexapodposition trägt. Der Executor hält eine Referenz auf eine Instanz der Klasse Hexapod (siehe Abschnitt 8.2 auf Seite 86), mittels derer die Position an den Hexapod gesendet wird.

7.4.3 WaitOnSelf- und WaitOnHexapodExecutor

Diese beiden Executor führen die zugehörigen Warteoperationen aus. Wenn die execute-Methode des WaitOnSelfExecutor aufgerufen wird, aktiviert dieser den Polling-Thread des zugehörigen Robot, registriert sich bei diesem als Beobachter und legt sich selbst schlafen. Der aufrufende Thread wird also vorübergehend deaktiviert. Durch den Polling-Thread des Roboters wird jedoch regelmäßig die update-Methode des registrierten WaitOnSelfExecutor aufgerufen, der die aktuelle Position des Roboters übergeben wird. Bei jedem Aufruf der Callback-Methode update wird überprüft, ob die aktuell abgefragte Position des Roboters bis auf eine gewisse Toleranz der zuletzt gesendeten Position entspricht. Ist dies der Fall, wird der aufrufende Thread wieder aufgeweckt. Dieser kehrt aus der Methode execute zurück und die `WaitOnSelfOp` ist fertig abgearbeitet.

Technisch läuft die Synchronisation über eine Variable des Typs `java.util.concurrent.locks.Condition`. Der aufrufende Thread blockiert sich auf diese Variable mittels eines Aufrufs von await(). Wird in der update-Methode festgestellt, dass die gewünschte Position erreicht wurde, so werden alle Threads, die sich auf die Condition-Variable blockieren mittels signal() aufgeweckt.

Sehr ähnlich ist der WaitOnHexapodExecutor implementiert. Da es aber nicht möglich ist vom Hexapod eine aktuelle Position abzufragen, wird jeweils nur die Methode isMoving des Hexapods aufgerufen. Liefert diese false zurück, so bewegt sich der Hexapod nicht mehr und der wartende Thread kann aufgeweckt werden, woraufhin er die execute-Methode verlässt.

Exemplarisch sei hier die etwas vereinfachte Implementierung der beiden Methoden execute und update von WaitOnSelfExecutor gezeigt:

```
1  protected List<ElementaryOp> executeEOPs(List<ElementaryOp> eops) {
2      eops.remove(0);
3
4      // add this as listener for positions of the robot
5      robot.addSubscriber(this, pollingInterval);
6      try {
7          lock.lock();
8          // wait for arrival of position
9          arrivedAtPosition.await();
10     }
11     finally {
12         lock.unlock();
13         // remove listener again
14         robot.removeSubscriber(this);
15     }
16
17     return eops;
18 }
19
20
21 public void update(Publisher publisher, Object arg) {
22     CellPosition cellPosition = (CellPosition)arg;
23
24     if(cellPosition.equalsWithTolerance(robot.getLastSentCellPosition())) {
25         lock.lock();
26         arrivedAtPosition.signal();
27         lock.unlock();
28     }
29 }
```

Bevor eine Methode einer Condition-Variable aufgerufen werden kann, muss erst die Methode lock der zu jeder Condition-Variable zugehörigen Lock-Variable aufgerufen werden.

Die Klasse CellPosition kapselt eine Position eines Roboters in der Zelle. Mehr dazu in Abschnitt 8.7 auf Seite 89.

7.4.4 DelayExecutor

Dieser Executor führt `DelayOps` aus. Dazu wird der aktuelle Thread mit der statischen Methode Thread.sleep schlafen gelegt. Die zu schlafende Zeit in Millisekunden wird in der `DelayOp` als Parameter übergeben.

7.4.5 GotoPositionExecutor und PathPreparing

Der GotoPositionExecutor verarbeitet `GotoPositionOps` indem er die enthaltenen Positionen an einen Roboter sendet. Dabei muss jedoch beachtet werden, dass eine Cell-Position spezielle Koordinatenwerte enthalten kann. Folgende Werte für Koordinaten sind erlaubt:

numerische Werte Einfache numerische Werte geben Absolutkoordinaten in der Zelle an.

additive Werte Diese stellen einen Versatz zur direkt zuvor gesendeten Position dar. Sie stehen also für relative Werte.

additive Rückwärtswerte Ähnlich wie additive Werte stellen diese eine Relativkoordinate dar. Der Bezug ist jedoch nicht auf eine Koordinate, die vor der aktuellen Position verarbeitet wurde, sondern auf eine Position, die der Aktuellen in der EOP-Liste folgt. So ist es möglich einen Offset in Bezug auf die folgende Koordinate anzufahren. Bei additiven Vorwärts- und Rückwärtswerten gibt ein relativer Wert von `0` an, dass die vorhergehende bzw. nachfolgende Koordinate übernommen werden soll.

Interpolationswerte Dieser Koordinatenwert zeigt an, dass die aktuelle Koordinate interpoliert werden soll. Dazu wird ein Wert berechnet, der zwischen der vorhergehenden und der nachfolgenden Koordinate im Pfad liegt.

Vor dem Senden einer Position muss diese also vorbereitet werden, indem nichtnumerische Werte durch Numerische ersetzt werden. Dies erledigt eine Instanz der Klasse PathPreparer. Aus der zuletzt gesendeten Position und allen folgenden Positionen von `GotoPositionOps` in der EOP-Liste wird ein Pfad – eine Liste aus Positionen – erstellt. In der ersten Position in diesem Pfad werden – falls vorhanden – nichtnumerischen Werte durch numerische ersetzt. Dazu werden zuerst vorhandene additive Werte, dann die additiven Rückwärtswerte und schließlich Interpolationswerte berechnet und durch das Ergebnis dieser Berechnung, welches einen Absolutwert darstellt, ersetzt. Nach

dieser Vorbearbeitung kann die Position an den Roboter gesendet werden. Anschließend kehrt die execute-Methode zurück.

7.4.6 SyncOnHexapodExecutor

Die execute-Methode des SyncOnHexapodExecutors besteht aus dem blockierenden Anfordern des Locks für den Hexapod. Die Methode kehrt zurück, sobald der Hexapod-Lock für den anfordernden Roboter gewährt wird. Ist der Lock momentan von einem anderen Roboter belegt, so muss gewartet werden, bis dieser den Lock freigibt. Anschließend wir der Hexapod reserviert und die `SyncOnHexapodOp` ist fertig abgearbeitet.

7.4.7 SyncOnGripperExecutor

Bei der Ausführung einer `SyncOnGripperOp` wird zuerst ein freier Greifer vom benötigten Typ, der als Parameter in der Operation gespeichert ist, ausgewählt. Ist kein Greifer frei, so greift eine Ausweichstrategie. Diese wird jedoch momentan nicht benötigt, da einem Roboter nur Tasks zugewiesen werden, für die ein freier Greifer verfügbar ist. Sind nur noch Tasks vorhanden, die einen Greifer benötigen, der momentan vom anderen Roboter belegt ist, ist die Arbeit für diesen Roboter beendet. Schon beim Scheduling der Tasks wird Sorge getragen, dass die Tasks so vergeben werden, dass beide Roboter in etwa gleich lange beschäftigt sind.

Ist ein konkreter Greifer ausgewählt, so wird dessen Lock geholt. Bisher sind allerdings die anzufahrenden Positionen in den `GotoPositionOps` in der Liste der EOPs noch nicht bestimmt. Es wird also jetzt die Position des Greifers in die passende `GotoPositionOp`, die mittels der semantischen Markierung gefunden wird, eingesetzt. Die Positionen davor und danach ergeben sich durch Relativkoordinaten aus der Greiferposition. Darüber hinaus wird auch noch der null-Wert in der `MountGripperOp` durch den ausgewählten Greifer ersetzt.

7.4.8 SyncOnFreePortExecutor

Ähnlich wie der SyncOnGripperExecutor arbeitet der SyncOnFreePortExecutor. Zuerst wird der freie Port ausgewählt, der dem Roboter, der einen Greifer parken will, am nächsten liegt. Dieser wird belegt und die fehlende Position analog zur Ausführung im SyncOnGripperExecutor ersetzt. In der `UnmountGripperOp` wird null durch den Zielport ersetzt, um bei deren Ausführung über den Aufenthaltsort des Greifers Buch führen zu können.

7.4.9 SyncOnSpaceExecutor und PathPlanners

Der SyncOnSpaceExecutor ist für die Bahnplanung zuständig. Hier werden `SyncOnSpaceOps` ausgeführt und die EOP-Liste ergänzt, falls eine Ausweichbewegung notwendig ist. Die aktuelle

Position ist die zuletzt Gesendete. Die Zielposition ist die Erste in einer `GotoPositionOp` in der Liste der Elementaroperationen. Daraus können die Start- und Zielarbeitsbereiche berechnet werden, indem bestimmt wird, in welchem Bereich der Start- und der Zielpunkt liegt.

Ist der Ziel- gleich dem Startbereich, so ist die Ausführung beendet und die EOP-Liste ohne die `SyncOnSpaceOp` am Anfang wird zurückgegeben. Andernfalls muss die Verfahrbewegung genauer geplant werden. Dazu werden Implementierungen des Interfaces PathPlanner verwendet. Jeder PathPlanner hat einen zugeordneten Start- und Zielbereich. Diese können über eine get-Methode abgefragt werden. Für jede Kombination aus Start- und Zielbereich existiert eine PathPlanner-Instanz. Die Schnittstellenmethode planPath nimmt als Parameter den Roboter, für den der Pfad geplant werden soll und den Zielpunkt der Bewegung. Sie gibt eine neue EOP-Liste zurück, die am Anfang der momentan ausgeführten EOP-Liste eingefügt wird. So können mittels der Pfadplanung EOPs am Beginn der Liste ergänzt werden.

Wird also eine `SyncOnSpaceOp` ausgeführt und Start- und Zielpunkt der Bewegung liegen in unterschiedlichen Bereichen, wird die Planung der Bewegung an den PathPlanner delegiert, der dem entsprechenden Start- und Zielbereich zugeordnet ist.

7.4.9.1 Die PathPlanners

Eine Klasse, die PathPlanner implementiert, ist dabei zuständig für die Bewegung von Bereich A nach Bereich B und von B nach A. Von dieser Klasse werden dann zwei Instanzen erzeugt. Über einen Konstruktorparameter wird angegeben, ob die Instanz für die „Vorwärtsbewegung“ (A→B) oder die „Rückwärtsbewegung“ (B→A) zuständig ist. Beide Bewegungen werden in einer Klasse verarbeitet, da sich die Planung von Vorwärts- und Rückwärtsbewegung sehr ähnelt.

Alle PathPlanner werden mit Hilfe des Entwurfsmusters Factory erzeugt. Der Methode createPathPlanner wird der Start- und der Zielbereichstyp übergeben. Zu beachten ist hier, dass der Typ der Bereiche übergeben wird und nicht der Bereich selbst. Dies macht nur einen Unterschied beim Bereichstyp Retraction. Von diesem gibt es zwei Ausprägungen (Retraction1 und Retraction2). Von allen anderen Bereichstypen (Pick, Place, Station) gibt es nur eine Ausprägung, weshalb im Folgenden statt von Bereichstypen auch von Bereichen gesprochen wird. Obwohl sich die Bereiche Retraction1 und Retraction2 unterscheiden, ist die Planung von Bewegungen von und nach beiden Bereichen gleich. Durch den an die Methode planPath übergebenen Roboter wird letztendlich entschieden, ob Punkte für die Bewegung über den Rückzugsbereich 1 oder 2 eingefügt werden sollen.

Da vier verschiedene Bereichstypen unterschieden werden, aber für die Bewegung innerhalb eines Bereichs keine Planung benötigt wird, ergeben sich sechs PathPlanner-Klassen. Von jeder

gibt es zwei Instanzen, sodass die Liste der PathPlanner im SyncOnSpaceExecutor zwölf Einträge umfasst.

Im Folgenden soll nun auf die Planung der Bewegungen zwischen den verschiedenen Arbeitsbereichen eingegangen werden. Grafiken, die die Ausweichbewegungen beschreiben, finden sich in Abschnitt 4.2.3.3 auf Seite 35.

Da Zielpunkte einer Bewegung immer in den Hauptbereichen Pick, Place oder Station liegen und bei der Zerlegung der COPs optimistisch davon ausgegangen wird, dass der Zielpunkt direkt, ohne Umweg über den Rückzugsbereich angefahren werden kann, wird zunächst immer eine Bewegung zwischen den Hauptbereichen geplant. Wird in einem Pfadplaner, der Hauptbereiche verbindet erkannt, dass die Bewegung nicht direkt erfolgen kann, wird über das Einfügen einer weiteren `SyncOnSpaceOp` und einer `GotoPositionOp` mit einem Punkt im Rückzugsbereich indirekt an einen Pfadplaner delegiert, der Bewegungen über den Rückzugsraum plant. Somit ergeben sich zwei logische Ebenen von Pfadplanern:

1. PathPlanner mit Zuständigkeiten für Bewegungen zwischen den Hauptbereichen Pick, Place und Station (drei Klassen) und
2. PathPlanner für die Planung von Bewegungen zwischen dem Rückzugsbereich und einem Hauptbereich (weitere drei Klassen)

Wie schon erwähnt, werden von jeder PathPlanner-Klasse zwei Instanzen erzeugt, die die beiden Bewegungsrichtungen planen. Intern ruft die Methode planPath eine Weitere auf, die folgende Signatur trägt: protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint, Area retractionArea). Der Parameter retractionArea trägt dabei den für den übergebenen Roboter passenden Rückzugsbereich. Außerdem sind in dieser Methode zwei Klassenvariablen bekannt: Area fromArea und Area toArea, die mit Start- und Zielbereich belegt sind.

An dieser Stelle ist es nötig, noch eine weitere EOP einzuführen, die nur aufgrund der Pfadplanung erforderlich ist. Die Elementaroperation `AcquireSpaceLockOp` dient dazu, einen Lock für eine räumliche Ressource (im Folgenden: einen Arbeitsbereich) anzufordern. Dies geschieht also nicht mehr bei der Ausführung der `SyncOnSpaceOp`, sondern in einem extra Executor namens AcquireSpaceLockExecutor, welcher in der execute-Methode solange blockiert, bis der benötigte Lock zugeteilt wurde. Die Notwendigkeit für diese EOP erschließt sich im Folgenden.

Zuerst werden die PathPlanner für die Hauptbereiche vorgestellt. Der wesentliche Teil der ausgeführten EOP-Liste besteht dabei aus der aktuell bearbeiteten `SyncOnSpaceOp` und der folgenden `GotoPositionOp`.

7.4.9.2 PickPlacePathPlanner

Dieser Pfadplaner kommt zum Einsatz, falls der Startbereich Pick und der Zielbereich Place ist, oder umgekehrt. Zur Erklärung dient der folgende Code:

```
1  protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint,
       Area retractionArea) {
2      List<ElementaryOp> eops = new LinkedList<ElementaryOp>();
3
4      if(areaResourceLocker.isAvailable(toArea)) {
5          // possible violation of the Retract-On-Locked-Area-Paradigm
6          eops.add(new AcquireSpaceLockOp(robot, toArea));
7      }
8      else {
9          // PICK to PLACE _and_ PLACE to PICK
10         eops.add(new SyncOnSpaceOp(robot, retractionArea));
11         int positionNumber = forth ? 1 : 2;
12         eops.add(new GotoPositionOp(positionLookup.lookupPosition(robot,
               positionNumber), SpecialPositionType.RETRACTION));
13
14         eops.add(new SyncOnSpaceOp(robot, toArea));
15     }
16
17     return eops;
18 }
```

Hier wird klar, warum Vor- und Rückwärtsbewegung in einem PathPlanner ausgeführt werden können: Über die beiden Variablen fromArea und toArea werden beide Bewegungen generisch gehandhabt. Eine Unterscheidung von beiden Fällen ist nur für die Bestimmung des Rückzugspunktes notwendig. Dazu dient die Variable forth, die true ist, falls die Bewegung von rechts nach links stattfindet, andernfalls ist sie false. Dies kann auch am Namen des PathPlanner erkannt werden. Forth ist wahr, wenn die Bewegung vom im Namen des PathPlanner erstgenannten zum zweitgenannten Bereich erfolgt. Beide Instanzen eines PathPlanners unterscheiden sich nur im Wert von forth.

Die Verhaltensweise im Falle eines freien bzw. eines belegten Bereichs soll im Folgenden kurz vorgestellt werden. Die Einführung von virtuellen Parametern hilft beim Verständnis der Pfadplanung. Die Punkte der Pfadplanung können aus der Klasse PositionLookup ausgelesen werden, auf was kurz eingegangen werden soll. Außerdem kann in der Pfadplanung ein unerwünschtes Verhalten auftreten, was als *Retraction-On-Locked-Area-Paradigm*-Verletzung bezeichnet wird.

Freier Zielbereich

Falls der Zielbereich frei ist, würde der Lock für diesen über eine `AcquireSpaceOp` reserviert werden. Hier könnte der Lock auch direkt angefordert werden, eine Notwendigkeit für die neue EOP `AcquireSpaceLockOp` ergibt sich hieraus noch nicht. Bild 7.4 zeigt diesen Fall. Vom PathPlanner neu hinzugefügte EOPs sind in den EOP-Listen grün markiert. Die virtuellen Parameter Start- und Zielbereich sind blau dargestellt.

Bild 7.4: Die EOP-Liste vor und nach Abarbeitung der `SyncOnSpaceOp` wenn der Place-Bereich frei ist

Belegter Zielbereich

Ist der Zielbereich nicht frei, wird eine neue `SyncOnSpaceOp` und eine `GotoPositionOp(Retraction-Point)` und eine weitere `SyncOnSpaceOp` eingefügt. Danach wird die resultierende Liste zurückgegeben. Bild 7.5 beschreibt diesen Fall genauer.

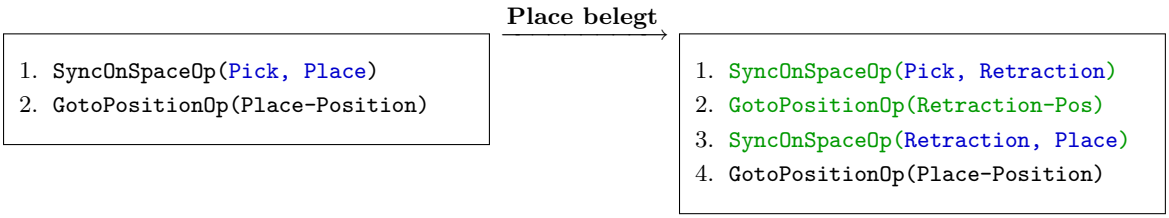


Bild 7.5: Die EOP-Liste vor und nach Abarbeitung der `SyncOnSpaceOp` wenn der Place-Bereich belegt ist

Virtuelle Parameter Start- und Zielbereich

Die Reaktion auf einen belegten Zielbereich wird deutlicher, wenn gedanklich zwei virtuelle Parameter für die `SyncOnSpaceOp` eingefügt werden: Start- und Zielbereich. Der Startbereich wird aus der zuletzt an den Roboter gesendeten Position berechnet. Diese Position war Parameter der `GotoPositionOp`, die in der Liste vor dem `SyncOnSpaceOp` stand und schon abgearbeitet wurde. Der Zielbereich folgt aus der Position der `GotoPositionOp`, die der `SyncOnSpaceOp` als nächstes folgt. Dies wird in Bild 7.6 auf der nächsten Seite illustriert.

Der Startbereich der ersten `SyncOnSpaceOp`, die angefügt wird, ist der Bereich fromArea. Da zwischen der Ausführung der aktuellen Methode und der Ausführung dieser `SyncOnSpaceOp`

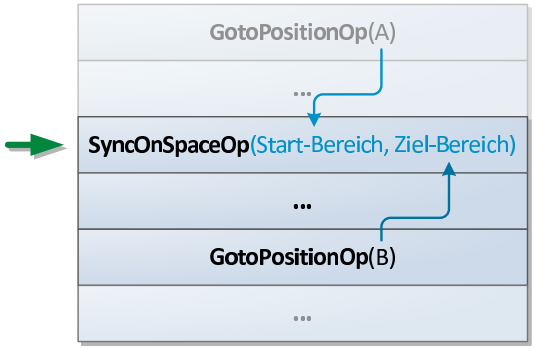


Bild 7.6: Über die gedanklich eingefügten Parameter Start- und Zielbereich wird die Ausführung einer `SyncOnSpaceOp` deutlicher. Start- und Zielbereich entsprechen genau den Variablen fromArea und toArea bei der Ausführung des PathPlanner.

keine weitere Position gesendet wird, bleibt der Startbereich also der gleiche wie in der aktuellen Methode, nämlich fromArea. Der Zielbereich ergibt sich aus der nächsten `GotoPositionOp`. Da die zugehörige Position im Rückzugsbereich liegt, ist der Zielbereich Retraction.

Die zweite eingefügte `SyncOnSpaceOpOp` trägt den Startbereich Retraction, da bei deren Ausführung die zuvor gesendete Position (`GotoPositionOp(Retraction-Pos)`) im Retraction-Bereich liegt. Der Zielbereich toArea ist gleich dem Zielbereich der aktuellen Methode.

Position-Lookup

Die konkrete Position im Retraction-Bereich, die in die `GotoPositionOp` eingefügt wird, wird mit Hilfe der Klasse PositionLookup und deren Methode lookupPosition bestimmt. Diese Methode gibt für einen Roboter und eine Positionsnummer einen Punkt zurück. Diese Positionen können in der externen Konfigurationsdatei des Bahnplaners verändert werden. Bild 7.7 auf der nächsten Seite zeigt, welche Punkte welche Nummern tragen.

ROLAP-Verletzung

Ein weiterer Aspekt aus dem obigen Code ist eine mögliche Verletzung des Prinzips, dass bei einem belegten Zielbereich die Bewegung über den Rückzugsbereich erfolgen muss. Dieses Prinzip wird als *ROLAP*[1] bezeichnet. Angenommen, es wird eine Bewegung von Pick nach Place geplant. Falls der Place-Bereich zum Zeitpunkt der Abfrage frei ist, so setzt der PathPlanner die Ausführung im if-Zweig fort und fügt die `AcquireSpaceLockOp` an den Anfang der EOP-Liste.

Findet nun zufällig zwischen der Abfrage auf den freien Bereich und der Ausführung des `AcquireSpaceLockOp` ein Thread-Kontextwechsel statt und ein zweiter Thread führt ebenfalls

1 Retraction On Locked Area Paradigm

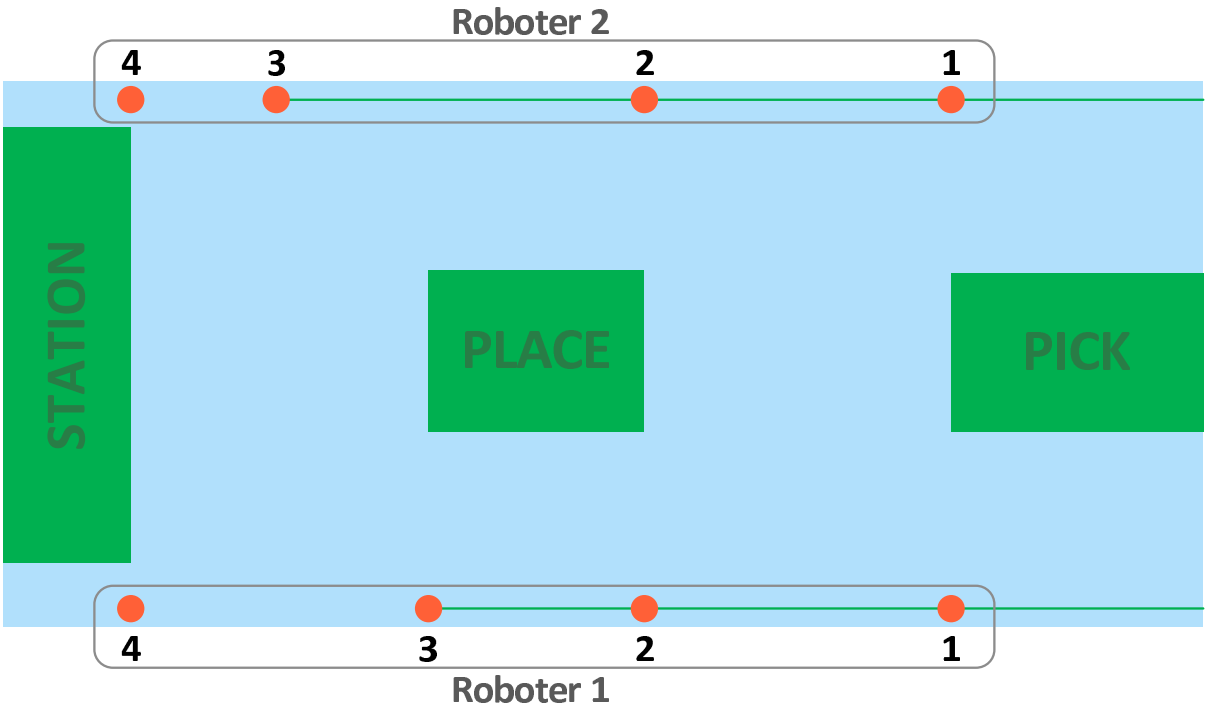


Bild 7.7: Die Punkte, die die Methode lookupPosition zurückgibt

eine Abfrage durch, ob der Place-Bereich frei ist, so wird diese Abfrage ebenfalls ergeben, dass der Bereich nicht belegt ist, da der erste Thread den Lock noch nicht geholt hat. Sperrt nun der zweite Thread zuerst den Place-Bereich für sich, so hätte im ersten Thread eigentlich der else-Pfad betreten und eine Ausweichbewegung geplant werden müssen. Die Konsequenz ist folgende: Der Roboter, dessen Weg im ersten Thread geplant wurde, versucht nun, während er noch im Pick-Bereich steht, den Lock für den Place-Bereich zu erwerben. Dieser ist jedoch schon vom zweiten Thread für den anderen Roboter belegt worden. Der erste Roboter verharrt also im Pick-Bereich bis er den Lock für den Place-Bereich erhält und fährt keine Ausweichbewegung. Dieses Verhalten wird ROLAP-Verletzung genannt.

Dies tritt nur auf, wenn ein anderer Thread den ersten Thread zwischen der Abfrage auf die Verfügbarkeit und die Reservierung des Bereiches unterbricht. Es führt aber zu keinem Deadlock, da der zweite Roboter, falls er nach dem Place- den Pick-Bereich betreten will, bemerkt, dass dieser belegt ist und eine Ausweichbewegung fährt. Sobald dieser Roboter den Place-Bereich frei gibt, kann der im Pick-Bereich Wartende den Lock für den Place-Bereich erwerben und diesen betreten, was die Freigabe des Pick-Bereiches zur Konsequenz hat.

Eine Kollision kann dadurch auch nicht auftreten, da ein Bereich immer erst betreten wird, wenn der Lock dieses Bereiches belegt wurde. Die ROLAP-Verletzung ist also ein unvorhergesehener Ablauf und kein Problem, das zu einem Fehlverhalten oder gar zu einer Kollision führen würde.

7.4.9.3 PlaceStationPathPlanner

Die Bewegung zwischen dem Place-Bereich und dem Pick-Bereich wird mit Hilfe des PlaceStationPathPlanners geplant. Folgender Code aus der Implementierung dieses PathPlanners soll die Pfadplanung verdeutlichen.

```
1  protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint,
       Area retractionArea) {
2      List<ElementaryOp> eops = new LinkedList<ElementaryOp>();
3
4      if(areaResourceLocker.isAvailable(toArea)) {
5          // possible ROLAP-VIOLATION
6
7          // MOVE DIRECTLY
8          eops.add(new AcquireSpaceLockOp(robot, toArea));
9
10         // INSERTION OF SUPPORT POINTS
11         // ...
12         // ...
13     }
14     else {
15         // PLACE to STATION _and_ STATION to PLACE
16         eops.add(new SyncOnSpaceOp(robot, retractionArea));
17         int positionNumber = forth ? 3 : 4;
18         eops.add(new GotoPositionOp(positionLookup.lookupPosition(robot,
               positionNumber), SpecialPositionType.RETRACTION));
19
20         eops.add(new SyncOnSpaceOp(robot, toArea));
21     }
22
23     return eops;
24 }
```

Für den Fall, dass der Zielbereich belegt ist, verläuft die Bewegung analog zu der Bewegung vom Pick- in den Place-Bereich. Es unterscheiden sich nur die Positionsnummern im Rückzugsbereich.

Falls die Bewegung direkt erfolgen kann, müssen allerdings weitere Stützpunkte eingefügt werden. Dies ist im obigen Codebeispiel ausgelassen und im nächsten zu sehen. Zuerst wird dazu die Position bestimmt, die in der Station angefahren wurde oder wird (je nachdem, ob aus dem Bahnhof heraus oder in ihn hinein bewegt wird). Liegt dieser Punkt außerhalb des Bewegungskorridors, so muss am Eck zwischen Korridor und Bahnhofsbereich ein Stützpunkt eingefügt werden. Außerdem muss beim Eintreten oder Verlassen des Bahnhofsbereichs immer ein Punkt vor und über dem Zielport angefahren werden. Bei Bewegungen innerhalb des

Bahnhofbereichs ist dies nicht notwendig, so dass dieser Punkt hier beim Pathplanning eingefügt werden muss und nicht schon bei der Erzeugung der anfänglichen EOP-Liste im COPDemuxer.

```
// INSERTION OF SUPPORT POINTS

// the part of the movement corridor inside the station
Rectangle2D stationCorridor = new Rectangle2D.Double(
        stationArea.getRectangle().getX(),
        placeArea.getRectangle().getY(),
        stationArea.getRectangle().getWidth(),
        placeArea.getRectangle().getHeight());

// the position in the station from where we came or to which we go
CellPosition stationPosition = forth ? targetPoint : robot.
    getLastSentCellPosition();

// support point in the corner of the above stationCorridor rectangle if
    movement target point is outside of the corridor
CellPosition supportPoint = null;
if(!stationCorridor.contains(stationPosition.getXYPoint())) {
    // => we need a support point
    int supportPointNo = stationPosition.y() < stationCorridor.getMinY() ? 0 :
        1;
    supportPoint = positionLookup.lookupStationSupportPoint(supportPointNo,
        stationPosition.getToolAngle());
}

// point above and in front of port
CellPosition aboveAndIfoPortPoint = new CellPosition(
        positionLookup.getStationRailX(),
        stationPosition.y(),
        positionLookup.getUpperZ(),
        stationPosition.getToolAngle());

if(forth) {
    // PLACE to STATION (directly)
    if(supportPoint != null)
        eops.add(new GotoPositionOp(supportPoint, SpecialPositionType.
            SUPPORT_POINT));
    eops.add(new GotoPositionOp(aboveAndIfoPortPoint, SpecialPositionType.
        ABOVE_AND_IFO_PORT));
}
else {
    // STATION to PLACE (directly)
```

```
36      eops.add(new GotoPositionOp(aboveAndIfoPortPoint, SpecialPositionType.
            ABOVE_AND_IFO_PORT));
37      if(supportPoint != null)
38          eops.add(new GotoPositionOp(supportPoint, SpecialPositionType.
                SUPPORT_POINT));
39  }
```

Abhängig davon, ob die Bewegung in den Bahnhofsbereich hinein oder heraus erfolgt, müssen die beiden Stützpunkte in verschiedener Reihenfolge in die EOP-Liste eingefügt werden. Dies wird in Bild 7.8 dargestellt.

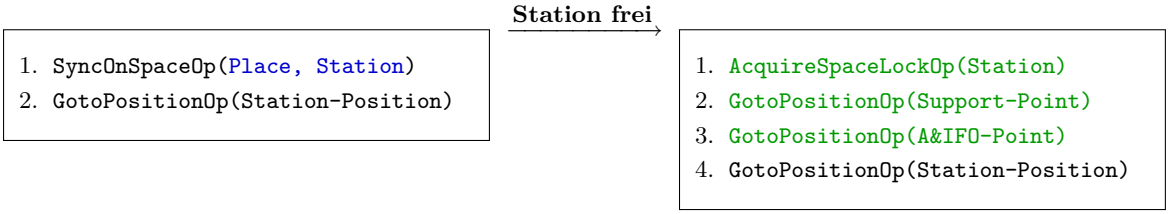


Bild 7.8: Die EOP-Liste vor und nach Abarbeitung der `SyncOnSpaceOp` wenn direkt in den Bereich Station gefahren werden kann. Die Zielposition liegt außerhalb des Korridors. Es müssen also Stützpunkte eingefügt werden.

7.4.9.4 PickStationPathPlanner

Die Planung der Bewegung vom Pick- in den Station-Bereich verläuft wiederum ähnlich wie bei der Place-Station-Bewegung. Folgender Code-Ausschnitt soll dies verdeutlichen:

```
1   protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint,
        Area retractionArea) {
2       List<ElementaryOp> eops = new LinkedList<ElementaryOp>();
3
4       List<SpatialResource> placeAndToArea = new ArrayList<SpatialResource>(2);
5       placeAndToArea.add(placeArea);
6       placeAndToArea.add(toArea);
7
8       if(areaResourceLocker.areAvailable(placeAndToArea)) {
9           // possible ROLAP-VIOLATION
10
11          // MOVE DIRECTLY
12          eops.add(new AcquireSpaceLockOp(robot, placeAndToArea));
13
14          // INSERTION OF SUPPORT POINTS
15          // ...
16          // ...
```

```
17      }
18      else {
19          // PICK to STATION _and_ PICK to PLACE
20          eops.add(new SyncOnSpaceOp(robot, retractionArea));
21          int positionNumber = forth ? 1 : 3;
22          eops.add(new GotoPositionOp(positionLookup.lookupPosition(robot,
                positionNumber), SpecialPositionType.RETRACTION));
23
24          eops.add(new SyncOnSpaceOp(robot, toArea));
25      }
26
27      return eops;
28  }
```

Falls Place- *und* Zielbereich verfügbar sind, kann die Bewegung direkt erfolgen. Beide Bereiche werden reserviert. Genauso wie bei der Planung der Place-Station-Bewegung müssen Stützpunkte eingefügt werden, um zu verhindern, dass die Bewegung den Korridor verlässt. Dies wurde hier nicht abgedruckt, sondern nur durch einen Kommentar verdeutlicht.

Falls entweder der Zielbereich oder der Place-Bereich nicht frei ist, so erfolgt die Bewegung über den Rückzugsraum entweder über Position 1 oder 3. Dies wird in Bild 7.9 illustriert.

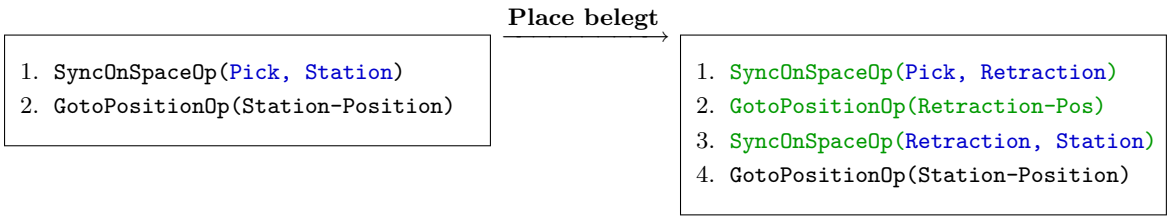


Bild 7.9: Die EOP-Liste vor und nach Abarbeitung der `SyncOnSpaceOp` wenn der Place-Bereich belegt, Station aber frei ist

7.4.9.5 RetractionPickPathPlanner

Die zweite Ebene der PathPlanner plant eine Bewegung vom Rückzugs- in einen Hauptbereich. Diese Pfadplaner fügen keine weiteren `SyncOnSpaceOps` ein. Im Gegensatz zur ersten Ebene der PathPlanner findet hier aber eine klare Unterscheidung zwischen Vor- und Rückwärtsbewegung statt. Um sich diese Unterscheidung zu sparen, könnten aus einem PathPlanner auch zwei gemacht werden, die jeweils für eine Richtung der Bewegung zuständig sind. Aus dem momentanen RetractionPickPathPlanner mit einer Unterscheidung der Variablen forth könnten also die zwei Pfadplaner RetractionPickPathPlanner und PickRetractionPathPlanner *ohne* Unterscheidung der Richtung gemacht werden.

Im Folgenden ein Auszug der Bewegungsplanung zwischen Pick- und Retraction-Bereich.

```
protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint,
    Area retractionArea) {
    List<ElementaryOp> eops = new LinkedList<ElementaryOp>();

    if(forth) {
        // RETRACTION to PICK
        eops.add(new GotoPositionOp(getXProjection(targetPoint),
            SpecialPositionType.WAIT_POINT));
        eops.add(new ReleaseSpaceOp(robot, retractionArea));
        eops.add(new AcquireSpaceLockOp(robot, pickArea));
    }
    else {
        // PICK to RETRACTION
        eops.add(new AcquireSpaceLockOp(robot, retractionArea));
    }

    return eops;
}
```

Vom Rückzugsbereich in den Pick-Bereich kommt man, indem die x-Projektion des Zielpunkts auf den Rückzugsbereich angefahren wird. Anschließend werden nicht mehr benötigte Bereiche freigegeben und der Lock für den Pick-Bereich wird erworben. Hier wird nun deutlich, warum die Einführung der `AcquireSpaceLockOp` nötig wurde. Bevor der Lock für einen Bereich angefordet werden soll, müssen weitere Operationen ausgeführt werden. Würde der Lock direkt an dieser Stelle mittels eines Funktionsaufrufs geholt, so wären die zwei Operationen `GotoPositionOp` und `ReleaseSpaceOp` noch gar nicht ausgeführt, da diese ja an dieser Stelle erst in die Liste eingereiht und nicht ausgeführt werden.

Die Bewegung vom Pick- in den Rückzugsbereich erfolgt nach Anfordern des Locks für den Rückzugsbereich. Dies ist streng genommen nicht notwendig, da sich jeder Roboter nur in seinem eigenen Rückzugsbereich bewegt. Dennoch wird der Retraction-Bereich gesperrt, um eine konsistente Ausführung der `SyncOnSpaceOps` und `ReleaseSpaceOps` zu gewährleisten.

Das Beispiel aus Bild 7.5 auf Seite 67 wird in Bild 7.10 auf der nächsten Seite fortgesetzt.

7.4.9.6 RetractionPlacePathPlanner

Die Bewegungsplanung zwischen Rückzugsbereich und Place-Bereich ist analog zur Planung der Bewegung in den Pick-Bereich. Nach Abarbeitung der `AcquireSpaceLockOp` und der `GotoPositionOp` aus Bild 7.10 auf der nächsten Seite wird die `SyncOnSpaceOp(Retraction,`

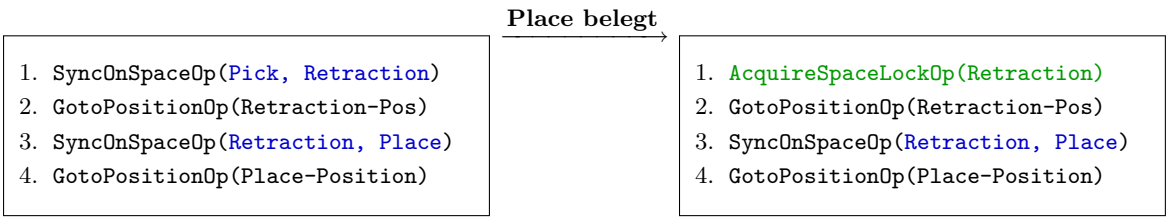


Bild 7.10: Die EOP-Liste vor und nach Abarbeitung der `SyncOnSpaceOp` wenn der Place-Bereich belegt ist

`Place`), wie in Bild 7.11 dargestellt, ausgeführt. Nachdem der Bereich freigegeben wurde, blockiert die Ausführung bei der `AcquireSpaceLockOp(Place)` solange, bis der Lock für Place erworben wurde.

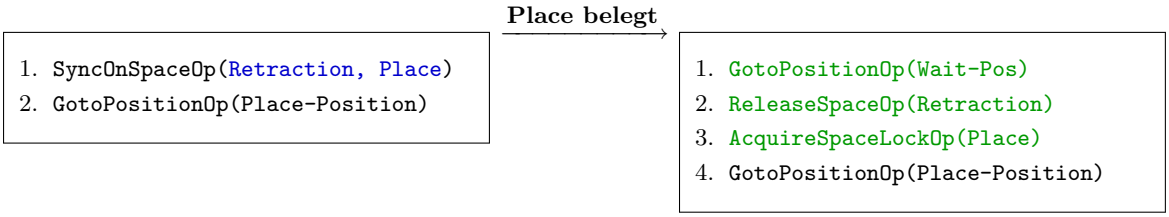


Bild 7.11: Die EOP-Liste vor und nach Abarbeitung der `SyncOnSpaceOp` wenn der Place-Bereich belegt ist

7.4.9.7 RetractionStationPathPlanner

Eine der komplexesten Pfadplanungen findet bei der Bewegung zwischen dem Station- und dem Rückzugsbereich statt.

```
1  protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint,
       Area retractionArea) {
2      List<ElementaryOp> eops = new LinkedList<ElementaryOp>();
3
4      if(forth) {
5          // RETRACTION to STATION
6          if(areaResourceLocker.isAvailable(stationArea)) {
7              // possible ROLAP-VIOLATION
8              eops.add(new AcquireSpaceLockOp(robot, stationArea));
9
10             // close the gripper to avoid collision in the station
11             if(robot.hasAttachedGripper())
12                 eops.add(new CloseGripperOp());
13
14             eops.add(new ReleaseSpaceOp(robot, retractionArea));
15         }
```

```
16          else {
17              CellPosition waitPoint = positionLookup.lookupPosition(robot, 3);
18              CellPosition lastSent = robot.getLastSentCellPosition();
19              if(!lastSent.getXYPoint().equals(waitPoint.getXYPoint()))
20                  eops.add(new GotoPositionOp(waitPoint, SpecialPositionType.
                        WAIT_POINT));
21
22              eops.add(new ReleaseSpaceOp(robot, retractionArea));
23              eops.add(new AcquireSpaceLockOp(robot, stationArea));
24
25              // close the gripper to avoid collision in the station
26              if(robot.hasAttachedGripper())
27                  eops.add(new CloseGripperOp());
28          }
29          // goto x projection of target and UP
30          eops.add(new GotoPositionOp(getXProjection(targetPoint, new Coord(
                positionLookup.getUpperZ()), targetPoint.getToolAngle()),
                SpecialPositionType.SUPPORT_POINT));
31          // above target point
32          eops.add(new GotoPositionOp(new CellPosition(targetPoint.x(),
                targetPoint.y(), positionLookup.getUpperZ(), targetPoint.
                getToolAngle()), SpecialPositionType.SUPPORT_POINT));
33      }
34      else {
35          // STATION to RETRACTION
36          eops.add(new AcquireSpaceLockOp(robot, retractionArea));
37
38          // close the gripper to avoid collision in the station
39          if(robot.hasAttachedGripper())
40              eops.add(new CloseGripperOp());
41
42          eops.add(new GotoPositionOp(new CellPosition(Coord.keep, Coord.keep, new
                 Coord(positionLookup.getUpperZ()), ToolAngle.keep),
                SpecialPositionType.SUPPORT_POINT));
43          eops.add(new GotoPositionOp(positionLookup.lookupPosition(robot, 4,
                Coord.keep, ToolAngle.keep), SpecialPositionType.SUPPORT_POINT));
44      }
45      return eops;
46  }
```

Wird ein Punkt im Bahnhofsbereich über den Rückzugsbereich angefahren, so muss zur Kollisionsvermeidung der Greifer geschlossen werden. Außerdem wird die x-Projektion des Zielpunkts auf die Verlängerung des Rückzugsbereichs mit maximaler z-Höhe angefahren. Anschließend wird

eine Position über dem Zielpunkt, wiederrum mit maximaler z-Höhe gesendet. Beim Verlassen des Bahnhofbereichs in den Rückzugsbereich wird ähnlich vorgegangen.

Bei der Bewegungsplanung vom Rückzugs- in den Bahnhofsbereich wird unterschieden, ob der Bahnhofsbereich frei ist. Ist dies der Fall, wird der Lock angefordert und nach Schließen des Greifers und Einfügen einer `ReleaseSpaceOp` der Bereich betreten.

Ist der Station-Bereich jedoch belegt, wird Punkt 3 angefahren. Vorige Bereiche werden zuerst freigegeben und dann erst wird der Lock für den Bahnhofsbereich belegt. Wird dieser erhalten kann nach Schließen des Greifers wiederum in den Bahnhofsbereich eingefahren werden.

In Bild 7.12 wird das Beispiel aus Bild 7.9 auf Seite 73 nach Abarbeitung der `SyncOnSpaceOp(Pick, Retraction)` und der `GotoPositionOp(Retraction-Pos 1)` fortgesetzt.

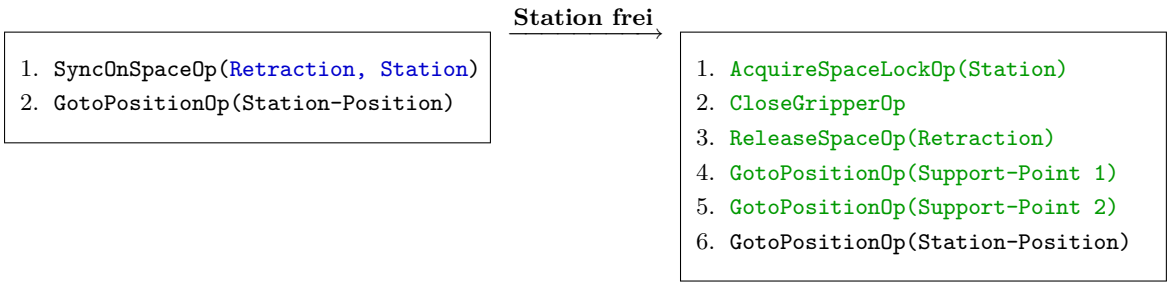


Bild 7.12: Die EOP-Liste vor und nach Abarbeitung der `SyncOnSpaceOp` wenn der Place-Bereich belegt, Station aber frei ist

7.4.10 ReleaseHexapodExecutor

Dieser Executor gibt den von einem Roboter gehaltenen Lock für den Hexapod zurück. Alle `ReleaseOps` im Folgenden tragen als Parameter den Roboter, der den Lock, der freigegeben werden soll momentan hält.

7.4.11 ReleaseGripperExecutor

Um den Lock für einen Greifer freizugeben, wird die `ReleaseGripperOp` benutzt. Folgender Code zeigt die Ausführung der Operation im ReleaseGripperExecutor.

```
1 protected List<ElementaryOp> executeEOPs(List<ElementaryOp> eops) {
2     ReleaseGripperOp op = (ReleaseGripperOp)eops.remove(0);
3
4     List<Gripper> acquiredPorts = gripperResourceLocker.getAllResourcesForRobot(
          op.getRobot());
5     // Single-Resource-Restriction
6     assert(acquiredPorts.size() == 1);
7
```

```
8       // Single-Resource-Restriction
9       gripperResourceLocker.releaseResource(acquiredPorts.get(0), op.getRobot());
10
11      return eops;
12  }
```

Nach Entfernen der EOP aus der Liste wird eine Liste aller vom Roboter reservierten Greifer geholt. Da aber a priori nicht klar ist, welcher der Greifer in dieser Liste freigegeben werden soll, tritt hier die sogenannte *Single-Resource-Restriction* in Kraft. Diese besagt, dass ein Roboter von der Ressource Greifer maximal eine Instanz sperren kann.

Diese Instanz wird dann mit Hilfe des passenden ResourceLockers freigegeben. Die verbleibende EOP-Liste wird zurückgegeben.

7.4.12 ReleasePortExecutor

Der ReleasePortExecutor arbeitet analog zum ReleaseGripperExecutor. Nach Bestimmung des vom Roboter reservierten Ports, wird dieser freigegeben. Auch hier ist die *Single-Resource-Restriction* in Kraft, so dass von einem Roboter nur maximal ein Port belegt werden darf.

7.4.13 ReleaseSpaceExecutor

Der Executor zur Ausführung der `ReleaseSpaceOp` arbeitet etwas komplizierter. Jede `ReleaseSpaceOp` trägt als Parameter neben dem Roboter, dessen Bereiche freigegeben werden sollen auch den Zielbereich der Bewegung.

Des Weiteren gibt es im ReleaseSpaceExecutor zwei weitere Zeiger auf Bereiche: Die Referenz current, die auf den Bereich zeigt, in dem sich der Roboter momentan aufhält oder null ist, wenn der Roboter sich in keinem Bereich befindet. Der Zeiger old zeigt anfänglich auf den gleichen Bereich wie current, im Verlauf der Abarbeitung aber auf den Bereich, der zuletzt verlassen wurde.

Um den current-Zeiger immer aktuell zu halten, registriert sich der ReleaseSpaceExecutor für den Zeitraum der Bearbeitung der Operation beim zugehörigen Roboter als Beobachter, der über neu empfangene Ist-Positionen informiert werden soll. Sobald der Poller-Thread des Roboters eine neue Position empfängt, wird die update-Methode aufgerufen. Die Kernfunktionalität findet sich in dieser Methode. In der Methode execute wird dagegen nur die Initialisierung der Zeiger vorgenommen. Anschließend blockiert sich der ausführende Thread in der execute-Methode auf eine Condition-Variable.

Die weitere Ausführung liegt nun in der Funktion update, die per Callback vom Poller-Thread aufgerufen wird. Der folgende Code zeigt diese Methode.

```
1  public void update(Publisher publisher, Object arg) {
2      CellPosition cellPosition = (CellPosition)arg;
3
4      if(current != null)
5          old = current;
6
7      current = areaManager.getAreaForPointWithTolerance(cellPosition.getXYPoint()
           );
8
9      if(current != null && current != old)
10         areaResourceLocker.releaseResource(old, robot);
11
12     if(current == target) {
13         lock.lock();
14         reachedTargetArea.signal();
15         lock.unlock();
16     }
17 }
```

current wird aus der aktuellen Position neu berechnet. Falls current nicht null ist – also der Roboter sich in einem Bereich befindet – aber old noch auf den alten Bereich zeigt, wird dieser freigegeben. Falls der Zielbereich erreicht wurde, ist die Bearbeitung der Operation fertig. Vor dem Aktualisieren des current-Zeigers muss nun noch old aktualisiert werden.

Die Signalisierung der Condition-Variable reachedTargetArea bewirkt das Aufwecken des Threads, der die execute-Methode aufgerufen hat. Dieser meldet den Executor als Beobachter des Roboters ab. Danach wird die update-Methode nicht mehr aufgerufen.

7.5 Die ResourceLocker

Das Interface ResourceLocker<T> stellt eine Schnittstelle zum Sperren und Freigeben von Ressourcen zur Verfügung. Es ist mit dem Typparameter T versehen, der die Klasse der Ressourcen angibt, die mit der aktuellen Instanz des ResourceLockers verwaltet werden soll. So können mit einer Implementierung beliebige Klassen als Ressourcen verwaltet werden, ohne dass diese eine gemeinsame Oberklasse haben müssten. Folgende Methoden sind in ResourceLocker enthalten:

boolean isAvailable(T resource) zum Prüfen, ob eine Ressource verfügbar oder gesperrt ist.

boolean areAvailable(List<T> resources) zur atomaren, nicht unterbrechbaren Überprüfung, ob mehrere Ressourcen frei sind.

List<T> getAllResourcesForRobot(Robot robot) liefert alle Ressourcen vom Typ, den der aktuelle ResourceLocker verwaltet, die vom übergebenen Roboter reserviert sind.

List<T> getAllResources() liefert alle Ressourcen, die von dieser ResourceLocker-Instanz verwaltet werden.

void acquireResource(T resource, Robot robot) dient dem Anfordern einer Ressource. Diese Methode blockiert, bis der Lock für die Ressource erworben wurde.

acquireResources(List<T> resources, Robot robot) fordert eine Liste von Ressourcen atomar an. Der Aufruf der Methode blockiert solange, bis alle Ressourcen der Liste gesperrt sind.

void releaseResource(T resource, Robot robot) gibt eine Ressource frei, die zuvor vom übergebenen Roboter reserviert war.

void releaseAllResourcesForRobot(Robot robot) gibt alle von einem Roboter belegten Ressourcen frei.

Für dieses Interface existiert die Implementierung PublishableResourceLocker<T>, welche auch noch das Interface Publisher implementiert, so dass sich Beobachter registrieren können, die über Updates des Ressourcenzustands informiert werden. Die Oberfläche registriert sich hier, um den aktuellen Ressourcenstand anzuzeigen.

7.6 Zusammenfassung

Montageaufgaben werden vom TaskProcessor entgegengenommen und ausgeführt. Dazu werden sie einer Instanz der Klasse RobotManager übergeben, die für jeden Roboter existiert. Diese Instanz nutzt die Klasse TaskDemuxer um einen Task in eine Liste an COPs zu zerlgen, welche dann vom COPExecutor ausgeführt werden. Dieser zergliedert jedes COP mit Hilfe eines COPDemuxer in eine Reihe EOPs. Für jeden Typ von EOP existiert ein EOPExecutor, der für die Ausführung dieser EOP verantwortlich ist. Diese EOPExecutor wurden vorgestellt, wobei besonderes Augenmerk auf die Executors gelegt wurden, die die `SyncOps` ausführen. Der Kern der Bahnplanung war in der Implementierung des SyncOnSpaceExecutor zu finden, der wiederum eine Reihe von PathPlanners nutzt, um den Verfahrweg zu planen. Abschließend wurde noch auf die Implementierung des Sperrens und Freigebens von Ressourcen in der Klasse ResourceLocker eingegangen.

8 Cell-Schicht und Modell der Hardware

Die Cell-Schicht bildet die Hardware, die in die Montageplanung involviert ist, als Modell ab. Es existieren Schnittstellen für die Roboter, den Hexapod, für Greifer, den Greiferbahnhof und das Transportsystem. Auch ist hier ein für beide Roboter gemeinsames Koordinatensystem definiert.

Methoden von Klassen der Cell-Schicht können grob in zwei Gruppen geteilt werden. Die erste Gruppe umfasst Methoden, deren Aufruf ein Senden eines Kommandos an einen Roboter nach sich zieht. Hierzu zählen beispielsweise Methoden zum Öffnen des aktuellen Greifers oder zum Abfragen der aktuellen Position des Roboters. Die Methoden der zweiten Gruppe haben keine externe Kommunikation zur Folge, sondern ändern interne Datenstrukturen im Montageprogramm. Manche Methoden haben Charakteristiken beider Gruppen. So sendet die Methode zum Anflanschen eines Greifers ein Kommando an den Roboter, um ein Bit zu setzen, das die Druckluft für die Haltebolzen kontrolliert. Gleichzeitig werden Einträge in einer internen Greiferverwaltungsdatenstruktur geändert. Sollte im Folgenden ein Funktionsaufruf ein Senden eines Kommandos an den Roboter nach sich ziehen, wird dies explizit erwähnt.

Zusätzlich bietet die Kommunikationsschicht die Möglichkeit, Robotervariablen im Softwaresystem zu puffern. So hat auch das Lesen einer Robotervariable auf Softwareseite manchmal kein Kommando an den Roboter zur Folge. Mehr dazu in Abschnitt B.1.1.2 auf Seite 102.

8.1 Das Modell der Roboter

Um einen Roboter anzusteuern, wurde das Interface Robot geschaffen. Dieses leitet von den Interfaces MoveableRobot, GripperAwareRobot und PollingPublisherToGUITier ab und definiert selbst noch einige Methoden. Die Struktur dieser Klassen und Interfaces wird in Bild 8.1 auf der nächsten Seite verdeutlicht. Folgende zusätzliche Methoden werden vom Interface Robot vorgeschrieben:

getNumber() liefert für jeden Roboter eine eindeutige Nummer zurück. Obwohl in der vorliegenden Arbeit immer von Roboter 1 und 2 die Rede ist, werden sie intern unter den Nummern 0 und 1 angesprochen.

getBoundingBox() bietet die Möglichkeit für jeden Roboter die zugehörige umgebende Hülle abzufragen.

isRingBufferUsed() fragt ab, ob der Roboter bei der Steuerung den Ringpuffer benutzt. Dies wird beim Start der Anwendung gesetzt und muss danach nicht mehr vom Roboter erfragt werden, da der Wert lokal gepuffert ist.

isGrinding() überprüft, ob das Überschleifen von Positionen aktiviert ist. Auch dies wird einmalig beim Start gesetzt.

getDriveMode() liefert den Verfahrmodus (entweder CP-LIN oder PTP) zurück. Dieser Wert wird ebenfalls einmalig belegt und lokal gepuffert.

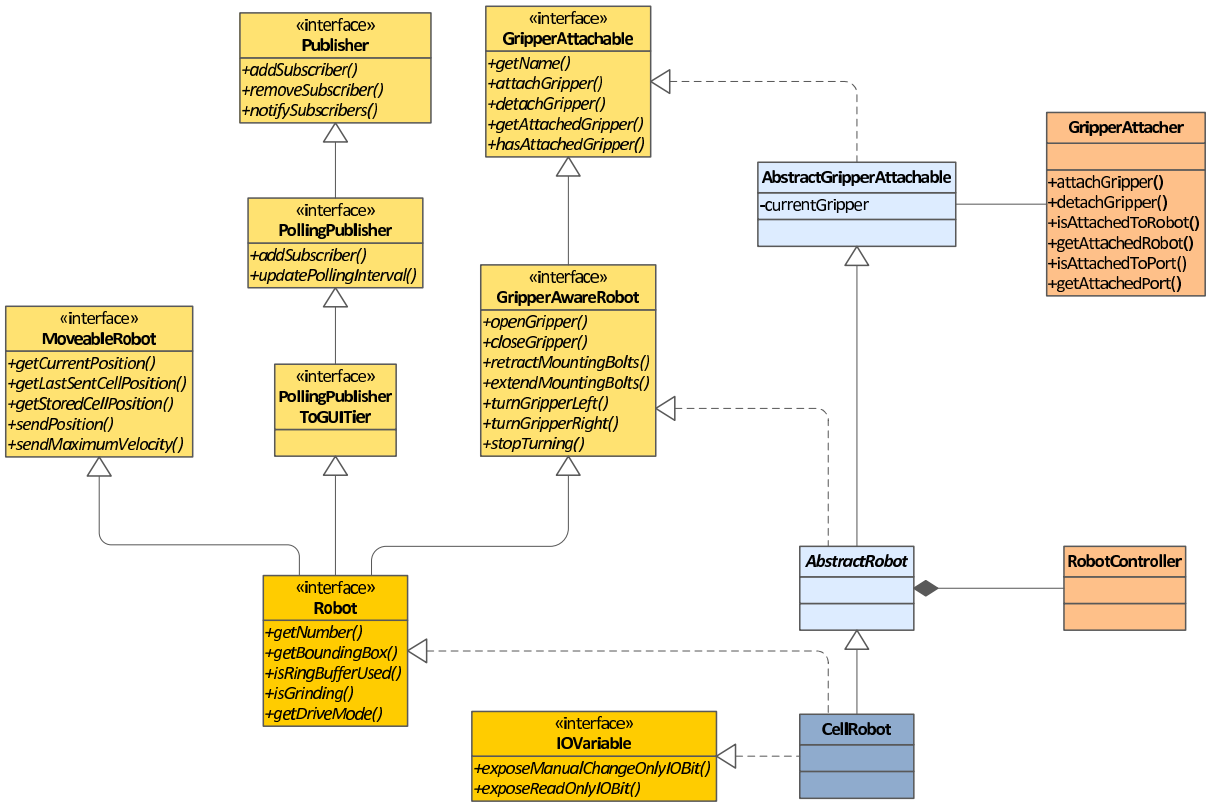


Bild 8.1: Die Struktur des Interfaces Robot und der damit verbundenen Klassen und Interfaces

8.1.1 Das Interface MovableRobot

Diese Schnittstelle definiert Methoden für einen Roboter, der bewegt werden und mit Positionsdaten umgehen kann. Es existiert die Methode getCurrentPosition zum Abfragen der aktuellen Position, welche ein Senden eines Abfragekommandos an den Roboter nach sich zieht. Die Methode sendPosition schickt eine Position an den Roboter. Die letzte gesendete Position wird softwareseitig gespeichert und kann mit Hilfe der Methode getLastSentCellPosition abgefragt werden.

Mit sendMaximumVelocity kann die Maximalgeschwindigkeit, bis zu der beschleunigt wird, an den Roboter gesendet werden.

Außerdem bietet das Interface noch die Methode getStoredCellPosition, die als Parameter einen String nimmt, der eine Position benennt, die zurückgeliefert wird. So können für jeden Roboter eigene Positionen gespeichert werden.

8.1.2 Das Interface GripperAwareRobot

Um einem Roboter die softwareseitige Handhabung mit Greifern zu ermöglichen, gibt es das Interface GripperAwareRobot. Dieses leitet vom Interface GripperAttachable ab, welches Methoden zum An- und Abhängen von Greifern bietet und in Abschnitt 8.4 auf Seite 87 näher beschrieben wird.

Außer den geerbten Methoden gibt es zwei Methoden zum Öffnen und Schließen des Greifers und zum expliziten Ein- und Ausfahren der Haltebolzen für Greifer. Dies wird von den beiden Methoden zum Mounten und Unmounten eines Greifers, die in GripperAttachable definiert sind, ebenfalls erledigt. Mount- und unmountGripper führen aber noch eine Überprüfung durch, ob auch wirklich ein Greifer als angehängt registriert ist und ändern außerdem interne Greiferverwaltungsstrukturen. Für Greifer, die eine Drehung über Druckluft ermöglichen, gibt es die Methoden turnGripperLeft, turnGripperRight und stopGripperTurning. Alle vorgestellten Methoden senden Kommandos an den Roboter, die bestimmte Bits setzen, welche die Druckluft kontrollieren.

8.1.3 Das Interface PollingPublisherToGUITier

Dieses Interface schreibt keine eigenen Methoden vor, sondern leitet vom Interface PollingPublisher ab. Es dient somit in erster Linie als Marker-Interface. Die Schnittstelle PollingPublisher schreibt die Methode addSubscriber vor, die einen Beobachter hinzufügt. Beim Hinzufügen muss auch ein Intervall in Millisekunden angegeben werden. Dieses Interface wird von Klassen implementiert, die periodisch einen Status abfragen und diesen über das Observer-Pattern anderen bekannt machen. So kann ein Robot zum Beispiel durch Implementieren dieses Interfaces Positionen periodisch abfragen und registrierte Beobachter über eine neu empfangene Position benachrichtigen. Ist kein Beobachter registriert, so wird auch keine Positionsabfrage durchgeführt. Sind mehrere Beobachter angemeldet, so ist das minimale Abfrageintervall aller Beobachter der Maßstab.

Die Methode updatePollingInterval von PollingPublisher aktualisiert das Abfrageintervall für einen registrierten Beobachter. Weitere Methoden werden von der Schnittstelle Publisher übernommen, von der PollingPublisher ableitet. Publisher schreibt eine Schnittstelle für ein allgemeines Subjekt

vor, das Beobachter über Statusänderungen informieren will. Hier existieren noch Methoden zum Entfernen und zum Benachrichtigen der registrierten Beobachter.

Diese Funktionalität wird vom WaitOnSelfExecutor (siehe Abschnitt 7.4.3 auf Seite 60) genutzt. Er registriert sich bei jeder Ausführung einer `WaitOnSelfOp` als Beobachter beim zugehörigen Robot und wird somit periodisch mit der aktuellen Position versorgt. Auch der ReleaseSpaceExecutor nutzt dies.

8.1.4 Die Klasse CellRobot

Die Klasse CellRobot implementiert die Schnittstelle Robot und bietet die Standardimplementierung für einen Roboter in einer Zelle. Sie leitet von der Klasse AbstractRobot ab, welche das Interface GripperAwareRobot umsetzt und damit die Implementierung für alle Greiferbelange umfasst. Diese Klasse leitet wiederum von AbstractGripperAttachable ab, was eine Standardimplementierung für das Interface GripperAttachable ist.

CellRobot implementiert die anderen benötigten Methoden aus dem Interface Robot. Für die Kommunikation mit dem Roboter wird die Klasse RobotController aus der Kommunikationsschicht verwendet. Mehr dazu in Abschnitt B.1.2 auf Seite 104.

Das periodische Abfragen der aktuellen Position wird nicht von der Klasse CellRobot selbst durchgeführt, sondern von der Klasse PollerThread, die eine periodische Positionsabfrage in einem eigenen Thread implementiert. Dem Konstruktor dieser Klasse wird eine Implementierung der Schnittstelle RunnablePoller übergeben, die das konkrete Abfragen der Position umsetzt.

8.1.4.1 Das Interface IOVariableExposingRobot

Neben Robot implementiert die Klasse CellRobot noch das Interface IOVariableExposingRobot. Dieses trägt dem Umstand Rechnung, dass externe Armaturen wie das Transportsystem und die Druckluftsteuerung für die Greifer ebenfalls über Bits gesteuert werden, die im Roboter gesetzt werden müssen. Das hat aber mit der eigentlichen Robotersteuerung nichts zu tun. Somit bietet dieses Interface die Möglichkeit, solche IO-Bit-Variablen zu exportieren, sodass andere Klassen diese Bits setzen können.

Die Methode exposeManualChangeOnlyIOBit nimmt die Byte- und die Bitnummer des benötigten IO-Bits und liefert eine Instanz der Klasse BitVariable zurück, die nur manuell geändert werden kann. Diese Instanz kann dann in einer Klasse, die dieses IO-Bit ansteuern will, referenziert werden, welche aber von einem Roboter nichts mehr weiß, da eine Instanz von CellRobot nur zur Initialisierung der Bitvariablen, aber nicht zur Laufzeit benötigt wird.

ExposeReadOnlyIOBit exportiert eine nur-lesbare Bitvariable und nimmt zusätzlich zu Byte- und Bitnummer noch ein Lese-Timeout zum Puffern von gelesenen Werten. Mehr zum Thema Bit- und Robotervariablen findet sich in Abschnitt B.1.1.2 auf Seite 102.

8.1.5 Die Vermeidung von Drehwinkeln außerhalb des gültigen Rotationsbereichs

Wird eine Position an den Roboter gesendet, so versucht dieser die Winkeländerung so zu verfahren, dass der überstrichene Winkel möglichst klein ist. So werden immer maximal 180° überstrichen. Nun ist aber der Drehbereich des Roboters nicht beliebig groß, sondern eingeschränkt. Von einem Winkel von -5° beginnend kann der Greifer bis zu einem maximalen Winkel von 365° gedreht werden. Bei einem Versuch, über die Randwinkel dieses 370°-Intervalls hinaus zu rotieren, wird der Drehendschalter aktiviert, der eine Beschädigung der Hardware vermeidet. Der Roboter nimmt dann keine Befehle mehr entgegen und das Programm muss neu gestartet werden.

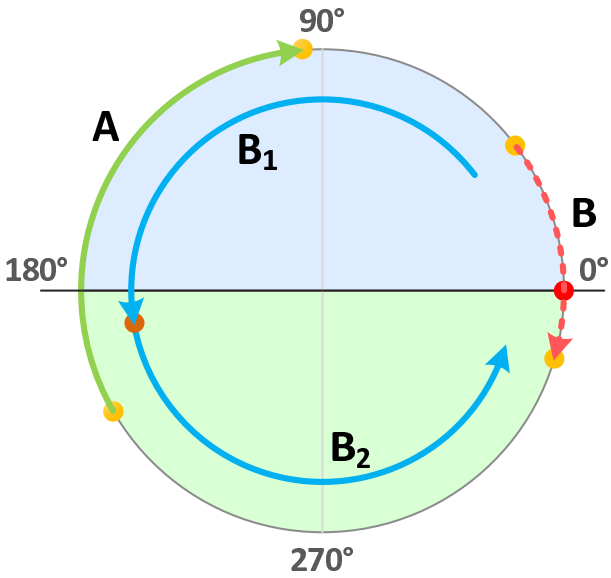


Bild 8.2: Die Teilung des Drehbereichs in zwei Halbkreise. Bewegung A kann direkt erfolgen. Bewegung B muss über einen Stützpunkt erfolgen und wird somit in B_1 und B_2 zerlegt.

Um dies zu vermeiden, wird nun der Kreis von 360° gedanklich in zwei Hälften geteilt. Die erste umfasst die Winkel von 0° (einschließlich) bis 180°, der zweite Halbkreis geht von 180° (inklusive) bis 360° (exklusive). Obwohl dabei 10° „verloren“ gehen, sind alle anfahrbaren Winkelstellungen abgedeckt. Der Endschalter wird gedanklich bei 0° eingerichtet. Liegen nun Start- und Zielwinkel im selben Halbkreis, so kann die Bewegung ohne weitere Vorkehrungen erfolgen.

Wird jedoch von einem Halbkreis in den anderen gedreht, so ist es nötig zu überprüfen, ob die Winkeldrehung 180° oder 0° überstreicht. Im ersten Fall sind ebenfalls keine weiteren Maßnahmen

notwendig. Sollte die Drehung aber 0° einschließen, so muss auf halbem Weg ein Stützpunkt gesetzt werden. Dieser trägt den halben Winkel des größeren Kreissegments zwischen dem Start- und dem Zielwinkel. Somit erfolgt die Drehung erzwungenermaßen über 180° und nicht über 0° und eine Bewegung der Drehachse in den Endschalter wird vermieden. Diese Funktionalität findet sich in der Methode sendPosition von CellRobot und nutzt einige Path-Preparer-Klassen, die schon vom GotoPositionExecutor (siehe Abschnitt 7.4.5 auf Seite 62) gebraucht werden.

8.2 Die Klasse Hexapod

Die Klasse Hexapod bildet ein Modell des Hexapods mit Methoden zur Steuerung und Statusabfrage. Die init-Methode initialisiert die Socketverbindung zum Hexapod und führt einen Reset der Achsen durch. Die Geschwindigkeit, mit der sich der Hexapod bewegt, wird auf die Geschwindigkeit aus der Konfigurationsdatei gesetzt.

Die Methode gotoPosition sendet eine Position an den Hexapod. Eine solche besteht aus den drei kartesischen Koordinaten x, y und z und den drei Winkeln a, b und c, die eine Drehung um die jeweilige kartesische Achse vorgeben. Die kartesischen Koordinaten stimmen mit den Koordinaten, die an die Roboter gesendet werden nicht überein. Der Hexapod hat ein eigenes Koordinatensystem, das um die z-Achse um 90° gedreht ist, so dass auch die Richtung der x- und y-Achsen nicht mit dem Roboterkoordinatensystem übereinstimmen.

Außerdem existiert die Methode isMoving, die den Bewegungszustand des Hexapods abfragt. Die Klasse Hexapod erweitert die Klasse DefaultPollingPublisher, die eine Standardimplementierung für einen Publisher des Beobachter-Patterns darstellt. Beim Aufruf der Methode isMoving werden alle registrierten Beobachter über den aktuellen Bewegungszustand informiert.

Ähnlich zur Klasse CellRobot existiert ein Poller-Thread, der periodisch die Methode isMoving aufruft, wenn Beobachter registriert sind. Diese setzen bei der Registrierung ebenfalls ein Abfrageintervall. So registriert sich der WaitOnHexapodExecutor (siehe Abschnitt 7.4.3 auf Seite 60) zur Ausführungszeit einer `WaitOnHexapodOp` als Beobachter bei der Klasse Hexapod und kann so auf einen Stillstand des Hexapods warten.

Zur Kommunikation mit dem Hexapod wird eine Implementierung des Interfaces HexapodController benutzt. Dazu mehr in Abschnitt B.2 auf Seite 109.

8.3 Die Klasse Gripper und deren Unterklassen

Die abstrakte Klasse Gripper definiert eine Vorgabe für alle konkreten Greiferklassen. Es kann im Konstruktor ein Name gesetzt und über die Methode getName abgefragt werden. Die abstrakten Methoden type und canTurn müssen von den Unterklassen überschrieben werden.

Außerdem wird innerhalb der Klasse Gripper noch der Aufzählungstyp GripperType definiert, der den Typ des Greifers darstellt und als Rückgabewert der Methode type fungiert. Jedem Typ ist eine Nummer zugeordnet. Folgende Unterklassen existieren:

ThreeClawGripper: kann nicht gedreht werden. Typnummer: 1

TurningLongGripper: kann gedreht werden. Typnummer: 5

TurningShortGripper: kann gedreht werden. Typnummer: 6

TwoClawBigGripper: kann nicht gedreht werden. Typnummer: 4

TwoClawSmallGripper: kann nicht gedreht werden. Typnummer: 2

Greifer werden über die GripperFactory instanziiert.

8.4 Das Interface GripperAttachable und die Klasse GripperAttacher

Jede Einheit im System, an die ein Greifer angedockt werden kann, implementiert das Interface GripperAttachable. Folgende Methoden werden in diesem Interface deklariert:

void attachGripper(Gripper g) zum Anhängen eines Greifers

Gripper detachGripper() zum Abhängen eines Greifers

Gripper getAttachedGripper() liefert den momentan angehängten Greifer

boolean hasAttachedGripper() überprüft, ob ein Greifer angehängt ist

String getName() liefert für das Objekt, an das ein Greifer angedockt werden kann, einen eindeutigen Namen

Die Klasse AbstractGripperAttachable stellt eine Standardimplementierung dieses Interfaces. Von dieser Klasse leitet die Klasse AbstractRobot ab. In der Implementierung der Methoden attachGripper und detachGripper wird eine private Variable currentGripper auf den neuen Greifer bzw. auf null gesetzt. Zusätzlich wird der Klasse GripperAttacher mitgeteilt, dass der Greifer nun an der aktuellen Instanz hängt.

Die Klasse GripperAttacher stellt eine zentrale Greiferregistrierung dar. Hier wird eine Liste aller Greifer und aller GripperAttachables geführt. Zusätzlich existiert eine Map, die jeden Greifer auf ein GripperAttachable abbildet, in dem sich dieser Greifer momentan befindet. Jede Änderung eines Greiferaufenthaltsortes wird der Klasse GripperAttacher über einen Aufruf der Methoden attachGripper und detachGripper gemeldet. Außerdem wird bei jeder Änderung eine Datei aktualisiert, die den aktuellen Aufenthaltsort eines jeden Greifers speichert. So muss nicht bei jedem Start des Montageplaners wieder eingestellt werden, wo sich welcher Greifer befindet. Über die Methode

isAttachedToRobot kann überprüft werden, ob der Greifer an einen der beiden Roboter angehängt ist. Dieser kann mittels getAttachedRobot erhalten werden. Alle Methoden von GripperAttachable und GripperAttacher ändern nur interne Datenstrukturen, senden aber keine Kommandos an den Roboter.

8.5 Die Klassen GripperPort und GripperStation

Die Vorrichtung, die einen Greifer im Bahnhofsbereich aufnimmt, wird im Folgenden als Greiferport bezeichnet. Für jeden Port existiert eine Instanz der Klasse GripperPort. Jede Instanz trägt eine eindeutige Nummer, die den Port nummeriert und die Position des Ports. Diese beiden Eigenschaften können durch get-Methoden abgefragt werden. Die Klasse GripperPort leitet von AbstractGripperAttachable ab. So kann softwareseitig einem Port ein eingehängter Greifer zugeordnet werden. Die Klasse GripperAttacher bietet außerdem die beiden Methoden isAttachedToPort und getAttachedPort, die analog zu den beiden Robot-Methoden funktionieren.

Die Klasse GripperStation stellt eine Abstraktion des gesamten Greiferbahnhofs dar und verwaltet eine Liste der Ports. Es existieren Methoden, die einen Port, dem eine bestimmte Nummer zugewiesen ist, zurückgeben, eine Methode die überprüft, ob freie Ports vorhanden sind und eine Methode, die eine Liste aller freien Ports zurückliefert.

8.6 Die Klasse TransportationSystem

Diese Klasse bietet ein Modell des Transportsystems, bestehend aus Sensoren, Sicherheitsbolzen und Förderbänder. Außerdem kann hiermit der Schrittmotor für den Laserscanner gesteuert werden. Der Konstruktor nimmt eine Referenz auf einen IOVariableExposingRobot (siehe Abschnitt 8.1.4.1 auf Seite 84). Hiermit werden die Bitvariablen angelegt, mit denen die Sensoren ausgelesen und die Bolzen und Hebevorrichtungen gesteuert werden.

Die zugehörigen Bitvariablen und Sensoren können über get-Methoden angefragt werden. Ein Teil des Transportsystems bildet die Klasse Conveyor-System. Diese verwaltet die vier Förderbänder. Jedes Laufband ist durch eine Instanz des Interfaces Conveyor abgebildet, welches Methoden zur Vorwärts- und Rückwärtsbewegung und zum Stoppen des Bandes beinhaltet. Des Weiteren können Beschriftungen für „vorwärts“ und „rückwärts“ gesetzt und abgefragt werden. So kann „forth“ beispielsweise durch „left“ und „back“ durch „right“ ersetzt werden. Da es unterschiedliche Bitkodierungen für die Steuerungen der Laufbänder gibt, existieren zwei Implementierungen, die jeweils für einen Typ von Bitansteuerung passen. Die beiden Implementierungen unterscheiden sich nur in den Methoden forth, back und off. Eine Methode on existiert nicht, da bei deren Aufruf nicht klar wäre, in welcher Richtung das Band laufen soll.

Bei Betätigung von Funktionen, die Zustände des Transportsystems ändern oder abfragen (Bolzen ein-/ausfahren, Sensoren abrufen, Bänder steuern) werden Kommandos zum Setzen und Abfragen von Bitvariablen an den Roboter gesendet.

8.7 Die Klasse CellPosition und Koordinatenumrechnung

Die Klasse CellPosition kapselt eine Koordinate im Zellkoordinatensystem. Diese Position besteht aus drei kartesischen Koordinaten x, y, und z und einem Werkzeugwinkel. Die einzelnen kartesischen Koordinaten sind Instanzen der Klasse Coord. Diese Klasse kann im Konstruktor numerische Werte aufnehmen, was die Angabe von absoluten Koordinaten erlaubt. Außerdem hat Coord vier öffentliche, statische Klassenvariablen, die Referenzen auf Instanzen der eigenen Klasse halten und unter folgenden Namen erreichbar sind:

Coord.keep der vorige Wert dieser Koordinate in einem Pfad (Liste an Positionen) soll beibehalten werden.

Coord.revKeep der im Pfad als nächstes kommende Koordinatenwert soll eingesetzt werden.

Coord.interpolate der Koordinatenwert soll aus vorhergehenden und nachfolgenden Werten interpoliert werden.

Coord.nn der Wert der Koordinaten ist zum Zeitpunkt der Erzeugung der CellPosition noch nicht bekannt.

Es existieren zwei Subklassen von Coord: AddCoord und RevAddCoor. Diese Klassen indizieren relative Vorwärts- oder Rückwärtskoordinaten. Die Relativkoordinate wird im Konstruktor angegeben. Die nichtnumerischen Werte werden beim *Path-Preparing* im GotoPositionExecutor durch numerische Werte ersetzt (siehe Abschnitt 7.4.5 auf Seite 62).

Die Klasse CellPosition beinhaltet darüber hinaus noch Methoden zur Abfrage der einzelnen Koordinaten (als double oder als Coord), als Vektor (x-, y- und z-Koordinate), als Punkt (nur x- und y-Koordinate) zur Abfrage des Winkels und zur Überprüfung zweier Positionen auf Gleichheit und auf Gleichheit mit einer gewissen Toleranzschwelle.

Das Zellkoordinatensystem stellt ein für beide Roboter gemeinsames Koordinatensystem dar und basiert auf dem Weltkoordinatensystem der Roboter. Die Umrechnung zwischen beiden geht in Klassen von statten, die das Interface PositionTransformer implementieren. Hier werden Methoden zur Berechnung von Zell- aus Weltkoordinaten und umgekehrt vorgeschrieben. Da das Weltkoordinatensystem für beide Roboter nicht genau übereinstimmt, gibt es zwei Implementierungen von PositionTransformer. Eine für Roboter 1 (WorldPositionTransformer), der als Referenz für das Weltkoordinatensystem angesehen wird und dann die Klasse ShiftedWorldPositionTransformer, die den Versatz, um den Roboter 2 abweicht, einberechnet.

9 Ausblick

Das Bahnplanungssystem wurde mit besonderem Augenmerk auf Erweiterbarkeit und Modularität entworfen. Deswegen sollen hier nun einige Ideen für Erweiterungen vorgestellt werden. Dabei können Bestandteile des Bahnplaners ausgetauscht oder um weitergehende Funktionen ergänzt werden. So ist die Anpassung auf vergleichbare Zellen einfach und auf beliebige Zellen vorgesehen, da die grundlegenden Konzepte der Synchronisierung und Kooperation auf beliebig viele Roboter übertragbar sind. Auch ist es einfach möglich, neue Funktionalität hinzuzufügen, die den Gesamtablauf optimiert und neue Möglichkeiten bietet.

Momentan wird beim *Scheduling der Montageaufgaben* aus der Menge der möglichen Tasks derjenige ausgewählt, der den Greifer benötigt, der am Roboter montiert ist, so dass kein Greiferwechsel notwendig ist. Sind mehrere Tasks dieser Art vorhanden, so wird ein beliebiger ausgewählt. Dies kann deutlich optimiert werden, indem versucht wird, den Montagegraph genauer zu analysieren und die Tasks so zu vergeben, dass ein Roboter nicht deutlich länger arbeitet als der Andere. Weitere Scheduling-Methoden sind denkbar. Dazu müsste das Interface TaskChooser, das einen Task aus der Menge der Möglichen auswählt neu implementiert werden.

Die momentane *Bahnplanung* basiert auf der Synchronisation über die Arbeitsbereiche. Neue Bahnplanungsmethoden können implementiert werden, indem die `SyncOnSpaceOp` anders verarbeitet wird. Hier liegt die Logik der Bahnplanung gekapselt.

Ein neuer SyncOnSpaceExecutor könnte beispielsweise folgendes Verfahrmodell implementieren: Roboter reservieren bei der Bewegung von einem Start- zu einem Zielpunkt immer den Korridor, den die umgebende Hülle überstreicht. Eine Bewegung ist nur möglich, wenn sich der benötigte Korridor nicht mit einem anderen überlappt. Der ReleaseSpaceExecutor würde diesen Bereich während der Bewegung verkürzen, so dass er sich nur noch von der aktuellen Position bis zur Zielposition erstreckt. Ist eine Bewegung nicht möglich, weil der Verfahrkorridor des anderen Roboters im Weg liegt, so kann die Bewegung soweit erfolgen, bis sich beide Korridore annähernd berühren. Hier muss gewartet werden, bis der Korridor des anderen Roboters durch den zugehörigen ReleaseSpaceExecutor so verkürzt wurde, dass der Weg nun frei ist. Nun kann die Bewegung fortgesetzt und der zu fahrende Korridor gesperrt werden. Aufgrund von direkten Verfahrwegen ohne explizite Ausweichbewegungen ist mit diesem *Bewegungskorridorsperrverfahren* eine effiziente Wegeplanung zu erwarten.

Eine weitere Möglichkeit, den aktuellen Bahnplaner zu erweitern, ist die Einführung weiterer Synchronisationselemente. Soll beispielsweise ein Roboter an einer Stelle warten, bis der andere eine weitere Stelle erreicht hat, so muss hierzu eine neue EOP eingeführt werden. Dies könnte bis zur direkten Signalisierung zwischen den Robotern gehen. Eine Anwendung davon wäre das Verschrauben von Bauteilen, die der andere Roboter fixiert. Ist das Bauteil an der Stelle angekommen, muss der handhabende Roboter so lange warten, bis der Roboter mit dem Schrauber an die richtige Position fährt und das Bauteil verschraubt. Dann kann der handhabende Roboter seinen Greifer öffnen und das nächste Bauteil holen.

Zusätzlich zur Einführung von Synchronisationspunkten mittels neuer COPs und EOPs können neue Typen von Montageaufgaben eingeführt werden. Diese würden nicht mehr von einem Roboter unabhängig vom anderen, sondern von beiden echt kooperierend ausgeführt. Beispielhaft dafür steht die beschriebene Verschraubung von Bauteilen und die Montageblechdrehung mit zwei Robotern. Diese neuen Tasks stellen wiederum neue Anforderungen an die Vergabe der Montageaufgaben und deren Repräsentation im Montagegraph.

Jede dieser Erweiterungen kann wegen der Anordnung des Bahnplaners in Schichten, der Kappselung der Funktionalitäten und der strengen Achtung auf lose Koppung der Komponenten einfach integriert werden.

10 Zusammenfassung

Mit der beschriebenen und implementierten Bahnplanungssoftware ist es möglich, Pick- und Place-Operationen mit zwei kooperierenden Robotern und einer autonomen Bahn- und dynamischen Montageplanung durchzuführen. Durch die Aufteilung der Zelle in Arbeitsbereiche und die explizite Reservierung dieser, wird die kollisionsfreie Bewegung sichergestellt. Greifer, Ports und der Hexapod tragen Locks, die angefordert werden müssen, bevor die Ressourcen benutzt werden können. Die Tasks werden in einem Montagegraphen abgelegt, aus dem jeweils der nächste Auszuführende über Scheduling-Strategien bestimmt wird. Dieser wird in kleinere Einheiten, die Elementaroperationen zerlegt, welche der Reihe nach ausgeführt werden. Es gibt Operationen zur Bewegung von Robotern und Hexapod, zum Steuern der einzelnen Bit-Ein- und -Ausgänge und zur Reservierung und Freigabe der Ressourcen. Die ausführenden Einheiten greifen auf die unterliegende Schicht zu, die die Hardware als Modell abbildet und eine einfache Schnittstelle zur Ansteuerung dieser bietet. Die grundlegende Kommunikation mit den Robotern und dem Hexapod wird in der untersten Schicht gekapselt. Die gesamte Bahnplanungssoftware kann über eine graphische Bedienoberfläche gesteuert und überwacht werden.

Es ergibt sich eine streng hierarchische Aufteilung des Softwaresystems in Schichten und Unterschichten. Dies führt zu einer sehr guten Wart- und Erweiterbarkeit des Softwaresystems. Über den konkreten Bahnplaner hinaus wurde mit dieser Arbeit eine Infrastruktur für weitergehende Projekte geschaffen. Deswegen wurden im Kapitel „Ausblick“ einige weiterführende Gedanken zur Erweiterung des Bahnplaners angestellt.

Die Beispielmontagesequenz der Türblechmontage wird mit dem entwickelten Bahnplaner erfolgreich und schnell abgearbeitet. Somit wurde am praktischen Beispiel gezeigt, dass sich das Bereichssperrverfahren für die beschriebene Roboterzelle sehr gut eignet und aufgrund diverser Charakteristiken der eingesetzten Roboter schneller und erfolgreicher arbeitet als vorhergehende Verfahren, wie Potentialfeld- oder Zellzerlegungsmethode, die ebenfalls mit dieser Zelle umgesetzt wurden.

Literaturverzeichnis

[1] Brunn, Arno: *XML-Kommunikation.* Reis Robotics GmbH, 2004.

[2] Elter, Thomas: *Externes Verfahren.* Reis Robotics GmbH, 2004.

[3] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns - Elements of reusable software.* Addison-Wesley Professional, January 1995.

[4] Ginal, Peter: *Entwicklung und Implementierung einer Montageablaufsteuerung für kooperierende Industrieroboter auf Grundlage der Zellzerlegungsmethode.* Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2005.

[5] Latombe, Jean-Claude: *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[6] Merhof, Jochen: *Entwicklung und Implementierung eines Bahnplaners für kooperierende Industrieroboter auf Grundlage der Potentialfeldmethode.* Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2006.

11 Abbildungsverzeichnis

12 Abkürzungsverzeichnis

CAD	Computer-Aided Design
COP	Complex Operation
CP-LIN	Controlled Path - linear
DLL	Dynamic-Link Library
DOM	Document-Object-Model
EOP	Elementary Operation
IDE	Integrated Development Environment
JNI	Java-Native-Interface
NAK	No Acknowledgement
OSI	Open Systems Interconnection
PHG	Programmierbares Handgerät
PTP	Point-To-Point
ROLAP	Retraction On Locked Area Paradigm
RTT	Round-Trip-Time
TCP/IP	Transmission Control Protocol / Internet Protocol
TCP	Tool-Center-Point
XML	Extensible Markup Language
XML-RPC	XML-Remote-Procedure-Call

A Instanziierung und Konfiguration des Systems

A.1 Die Klasse RobotFacility zur Instanziierung des Systems

Die Klasse RobotFacility kann keiner Schicht zugeordnet werden sondern dient vor allem der Instanziierung und der Initialisierung des Softwaresystems. Sie wird in der Klasse CLRControlPanelWindow der Oberflächenschicht instanziiert, welche die zentrale Klasse für die Erzeugung der Oberfläche und die Steuerung des Bahnplaners darstellt.

RobotFacility enthält zwei konstante Referenzen auf ein Objekt der Klasse Cell und ein Objekt vom Typ TaskProcessor. Da diese öffentlich sind, kann der TaskProcessor – die Bahnplanungslogik – und die Cell – das Hardwaremodell – direkt angesprochen werden, wenn ein Objekt vom Typ RobotFacility vorliegt.

Die Klasse Cell der Cell-Schicht fasst alle funktionalen Aspekte dieser Schicht in einer Klasse zusammen. Über get-Methoden können Referenzen auf beide Roboter, den Hexapod, die Klassen zur Greiferverwaltung und das Transportsystem erhalten werden.

Das gesamte System wird mit Hilfe des Spring-Frameworks und dessen Konzept der *Dependency Injection* initialisiert und verschaltet. Alle Konstruktoren nehmen Referenzen auf Abhängigkeiten, die von dieser Klasse benötigt werden und belegen damit private Attribute. Das Spring-Framework sorgt für die tatsächliche Instanziierung. Wie genau das System verschaltet wird und welche Klassen einer anderen Klasse als Abhängigkeiten „injiziert“ werden, wird in einer XML-Konfigurationsdatei festgelegt.

Zur Instanziierung des Systems wird ein ApplicationContext erzeugt, der dazu die XML-Konfiguration des Systems benötigt. Alle Klassen werden dann instanziiert und miteinander gemäß der Konfiguration mittels Dependency Injection verschaltet. Durch einen Aufruf von applicationContext.getBean("robotFacility") kann die RobotFacility-Instanz erhalten werden, mit der alles weitere angesteuert wird.

A.2 Die Klasse RobotFacilityProperties

Zusätzlich zur Bestückung der Klassen mit ihren Abhängigkeiten, die zur aktiven Ausführung benötigt werden, muss auch noch die Konfiguration mit statischen Konfigurationsdaten vorgenommen werden. Erstere Abhängigkeiten sind aktive Objekte, letztere passive Daten.

Dazu dient die Klasse RobotFacilityProperties. Dieser wird bei der Erzeugung ein `java.lang.Properties`-Objekt übergeben, das die Konfiguration aus einer externen Java-Properties-Datei enthält.

Jede Konfigurationseigenschaft hat einen Namen und einen Wert. Eigenschaften können von verschiedenen Typen sein. Alle Typen repräsentieren eine Klasse aus der Problemdomäne. Sie werden im folgenden also auch Domänenklassen genannt. Einerseits gibt es skalare Typen, welche nur einen einzelnen Eigenschaftswert umfassen, andererseits gibt es zusammengesetzte Typen, die aus mehreren Werten bestehen.

Jede Domänenklasse hat eine Repräsentation in der Konfigurationsdatei. Boolean zum Beispiel kann einen beliebigen Namen haben, der die Eigenschaft identifiziert. Mögliche Werte sind true oder false. Die Repräsentation der Domänenklasse Vector schaut folgendermaßen aus:

```
1 NAME.x=5
2 NAME.y=10
3 NAME.z=42
```

Da Vector eine zusammengesetzte Domänenklasse ist, erstreckt sich die Konfiguration über mehrere Zeilen.

Die Darstellung des Typs Rectangle in der Konfigurationsdatei ist wie folgt:

```
1 NAME.x0=10
2 NAME.x1=100
3 NAME.y0=10
4 NAME.y1=200
```

NAME muss in allen vier Zeilen der gleiche sein, kann ansonsten aber frei gewählt werden. Um nun innerhalb der Konfigurationsdatei Eigenschaften zu gruppieren, wird der Name mit einem Präfix versehen, das angibt, in welchem Bereich des Systems die Eigenschaft benötigt wird. Dieses Präfix ist hierarchisch aufgebaut und durch Punkte untertrennt. Die Eigenschaft vom Typ Rectangle mit dem Namen cell.robot1.boundingBox wird also in der Cell-Schicht verwendet, um Roboter 1 zu konfigurieren. Das Präfix ist cell.robot1, der Name boundingBox.

Folgendes taucht also in der Konfigurationsdatei auf:

```
1 cell.robot0.boundingBox.x0=-243
2 cell.robot0.boundingBox.x1=120
3 cell.robot0.boundingBox.y0=-300
```

```
4 cell.robot0.boundingBox.y1=140
```

Zu jeder Domänenklasse gibt es in der Klasse RobotFacilityProperties eine get-Methode, deren Namen sich folgendermaßen zusammensetzt: get + Name der Domänenklasse + Property. In jeder get-Methode wird das Domänenobjekt erzeugt, mit den Werten aus der Konfigurationsdatei bestückt und zurückgegeben.

Zusätzlich gibt es noch die Methode hasProperty zum Überprüfen, ob eine Eigenschaft vorhanden ist und subProperties. subProperties nimmt als Parameter einen Präfixstring und gibt eine neue Instanz von RobotFacilityProperties zurück. Diese wird gebildet, indem alle Eigenschaften, deren Namen mit dem übergebenen Präfix beginnen, in die neue Instanz eingefügt werden. Das Präfix wird dabei vom Namen abgeschnitten. So erzeugt der Aufruf von properties.subProperties("cell") ein neues Objekt vom Typ RobotFacilityProperties, das alle Eigenschaften von properties umfasst, deren Name mit "cell" beginnt.

Jeder Konstruktor einer Klasse, die konfigurierbar ist, erhält als Parameter ein Objekt der Klasse RobotFacilityProperties, aus dem mittels der get-Methoden benötigte Eigenschaften ausgelesen werden.

B Communication-Schicht und XML-Umsetzung

B.1 Kommunikation mit den Robotern

Wie schon ausgeführt, sind die Roboter an ein *Ethernet-Netzwerk* angeschlossen, über welches Kommandos abgesetzt und Antworten empfangen werden können. Dazu wird das weit verbreitete *TCP/IP*-Protokollpaar genutzt, das zuverlässiges Senden und Empfangen von Antworten bietet.

B.1.1 Das Lesen und Schreiben von Robotervariablen

B.1.1.1 Die Datentypen und Systemvariablen des Roboters

Der Roboter *RL-16* der Firma *Reis Robotics* hat über 600 Systemvariablen, die mittels Kommandos gelesen und beschrieben werden können. Der Roboter unterscheidet zwischen einigen verschiedenen Datentypen: Frame, Integer, Position, Real, String, Tool und Vector. Eine Variable vom Typ *Frame* ist eine 4×4-Matrix, die ein Koordinatensystem beschreibt. *Integer* und *Real* stellen Ganz- bzw. Gleitkommazahlen dar. *String* repräsentiert eine Zeichenkette und mittels einer Variable des Datentyps *Tool* kann das an den Roboter angebrachte Werkzeug genauer bestimmt und der TCP verschoben werden. *Position* besteht aus mehreren Komponenten:

1. Anzahl der Hauptachsen (Integer)
2. Anzahl der Zusatzachsen (Integer)
3. Positionstype (Integer)
4. Nummer des Frames (Integer)
5. x, y und z-Koordinate des TCP (jeweils Real)
6. x, y und z-Koordinate des Tool-Frames in z-Richtung (jeweils Real)
7. x, y und z-Koordinate des Tool-Frames in x-Richtung (jeweils Real)

Der *Reis RL-16* Roboter hat vier Haupt- und keine Nebenachse.

Framenummer 0 steht für den Base-Frame. Dies ist das jedem Roboter eigene, grundlegende Koordinatesystem. Die Framenummern 1 bis 30 stehen für den Benutzer zur Verfügung, um sich eigene Koordinatensysteme zu definieren.

Ist der TCP nicht über eine programmierte Tool-Variable verschoben, so stellen die TCP-Werte die Koordinaten im Zentrum des Flansches der Roboter dar.

Im Falle des *RL-16* stellen die x- und y-Komponenten des Tool-Frames in x-Richtung den Cosinus bzw. den Sinus des Drehwinkels um die z-Achse dar.

Jede Systemvariable hat einen Namen, über den sie angesprochen werden kann. Dieser Name beginnt mit dem Präfix `_T`, wobei `T` der erste Buchstabe des Datentyps ist. So ist `_IKEY_CODE` eine Variable vom Typ Integer und `_SVERSION` eine Variable vom Typ String.

Außerdem unterstützt die Steuerungssoftware der Roboter Arrays von Variablen. `_ROPTI_OUT[6]` ist ein Feld aus sechs Real-Variablen, `_FCALAX[24]` ein Feld aus 24 Frame-Variablen.

Manche Variablen können gelesen und beschrieben werden, andere nur ausgelesen oder nur beschrieben werden.

B.1.1.2 Das Interface RobotVariable

Die Interface RobotVariable steht für genau eine Variable des Roboters. Ihr Typparameter T gibt den Java-Datentyp an, auf den der Datentyp des Roboters abgebildet wird. *Integer* und *String* werden zu den gleichnamigen Java-Typen umgewandelt, *Real* auf *Double* und *Position* auf die gleichnamige, selbstdefinierte Klasse. *Frame* und *Tool* werden nicht benutzt, könnten aber über die Definition eigener Frame- und Tool-Klassen auch in das System eingebunden werden.

Das Interface RobotVariable leitet von den beiden Interfaces ReadableRobotVariable und WriteableRobotVariable ab. ReadableRobotVariable enhält die Methode read() zum Lesen, WriteableRobotVariable die Methode write(T value) zum Beschreiben von Werten. Ist eine Variable nur lesbar, so kann diese als ReadableRobotVariable definiert werden. Äquivalent verhält es sich mit nur schreibbaren Variablen.

Das Interface RobotVariable wird von der Klasse RobotVariableImpl implementiert. Dem Konstruktor dieser Klasse wird dabei eine Instanz der Klasse RobotCommunication übergeben, die die Kommunikation mit dem Roboter kapselt. Ein Objekt der Klasse VariableSpec definiert die Variable genauer. Dazu wird bei der Erzeugung von VariableSpec der *Name*, der *Roboterdatentyp* und der *Index der Variablen*, falls diese in einem Feld liegt angegeben. Da die Sichtbarkeit des Konstruktors von RobotVariable auf das Package variables beschränkt ist, müssen Variablen über eine Instanz der Factory VariableFactory erzeugt werden.

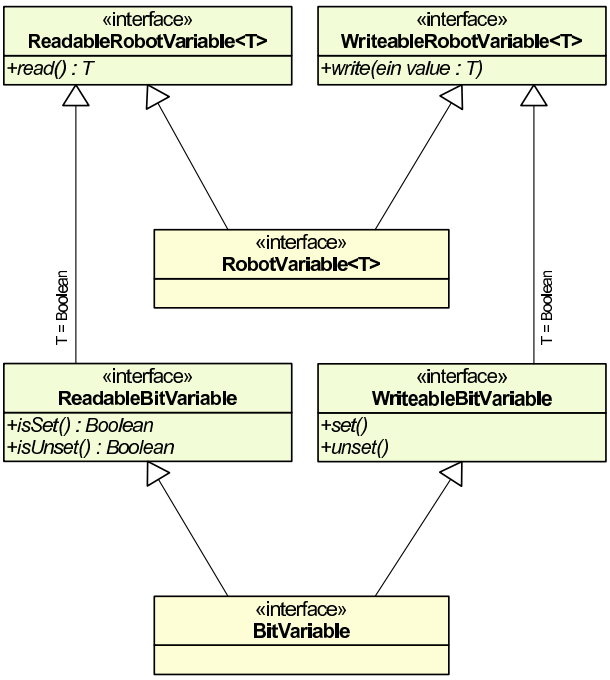


Bild B.1: Die Interfaces RobotVariable und BitVariable

Hierbei gibt es drei Methoden, die eine `Read-Only`-, eine `Write-Only`- und eine `Manual-Change-Only`-Variable erzeugen. `Manual-Change-Only` bedeutet, dass die Variable nur vom Anwender geschrieben wird. Der Roboter selbst überschreibt diese Variable nicht eigenmächtig. Damit wird beim Lesen der Variable das Senden eines Kommandos an den Roboter gespart, da der schon geschriebene Wert auf dem Host-PC gepuffert werden kann. Beispielhaft stehen dafür die Variablen gripperOpening zum Öffnen und Schließen eines Greifers oder maxCartVelocity zum Setzen der kartesischen Maximalgeschwindigkeit.

Zusätzlich gibt es das Interface BitVariable, welches ein einzelnes Bit einer Integervariable eines Roboters repräsentiert. BitVariable erbt von den beiden Interfaces ReadableBitVariable und WriteableBitVariable. ReadableBitVariable stellt die beiden Methoden isSet() und isUnset() zur Verfügung und erbt von ReadableRobotVariable<Boolean>. WriteableBitVariable schreibt die beiden Methoden set() und unset() vor und erbt von WriteableRobotVariable<Boolean>. Um eine Bitvariable zu erzeugen, können auch die oben vorgestellten Methoden der VariableFactory benutzt werden. Allerdings wird nun eine Instanz der Klasse BitSpec übergeben, welche von VariableSpec erbt. Im Konstruktor kann der Name und der Index der Integervariablen, in der das Bit liegt und die Bitnummer des Bits innerhalb dieser Integervariable angegeben werden. Zwei spezielle Subklassen von BitSpec sind IOBitSpec zur Ansteuerung von binären Ein-/Ausgängen und MarkerSpec zur Spezifikation von Marker-Bits (Bits in der Systemvariable `_IPLC`).

B.1.1.3 Das Lesen und Schreiben von Robotervariablen

Die Methoden zum Lesen und Schreiben einer Robotervariablen delegieren auf Implementierungen der Interfaces ReadStrategy und WriteStrategy. Hier kommt das *Strategiepattern* zum Tragen, das einfache Austauschbarkeit verschiedener Implementierungen einer Funktionalität bietet. Verschiedene Lesestrategien sind:

AlwaysQueryReadStrategy zum *normalen, sofortigen* Lesen.

BufferedReadStrategy zum *gepufferten* Lesen. Solange seit dem letzten Lesen der Variable weniger als ein einstellbarer Timeout vergangen ist, wird der zuvor gelesene Wert sofort zurück gegeben.

ReturnLastSentElseQueryReadStrategy gibt den *zuletzt gesendeten Wert* zurück. Wurde bisher kein Wert an den Roboter gesendet, wird der aktuelle Wert der Variable auf dem Roboter *ausgelesen*.

ReturnLastSentReadStrategy gibt den *zuletzt gesendeten Wert* zurück. Wurde bisher kein Wert an den Roboter gesendet, wird null zurückgegeben.

Folgende Schreibstrategie gibt es:

AlwaysWriteStrategy zum sofortigen Senden des Werts an den Roboter.

IfNotAlreadySetWriteStrategy zum bedingten Senden des Werts. Falls der zuletzt gesetzte Wert dem neu zu setzenden entspricht, wird kein Wert gesendet.

Auf die Implementierung des Lese-/Schreibvorgangs einer Variable wird später unter Abschnitt B.1.3 auf der nächsten Seite eingegangen.

Eine `Read-Only-Variable` benutzt eine ReadStrategy vom Typ BufferedReadStrategy, falls ein Timeout angegeben wurde, ansonsten AlwaysQueryReadStrategy. Eine `Write-Only-Variable` hat die Schreibstrategie AlwaysWriteStrategy. Eine `Manual-Change-Only-Variable` benutzt die Schreibstrategie IfNotAlreadySetWriteStrategy und die Read-Strategy ReturnLastSentReadStrategy.

B.1.2 Die Klasse RobotController

Die Klasse RobotController beinhaltet eine *Sammlung von benötigten Robotervariablen*. Weitere können durch einfache Deklaration hinzugefügt werden. Folgende Variablengruppen sind vorhanden:

- Kontrolle des Verfahrmodus (PTP vs. CP-LIN, Überschleifen, Ringpuffer)
- Senden von Positionen und Lesen der IST-Position

- Setzen und Auslesen der Ist- und Maximalgeschwindigkeit und der Beschleunigung (sowohl kartesisch, als auch rotatorisch)
- BitVariablen zum Öffnen und Schließen und zur Aufnahme und Ablage der Greifer

Das gewünschte *Koordinatensystem* kann eingestellt werden. Standardeinstellung ist das `BASE`-Koordinatensystem. Wird dieses benutzt, wird die aktuelle Position aus der Variable `_PACTISTPOS` gelesen. Ist das eingestellt Koordinatensystem `WELT`, so wird `_PUSER[1]` gelesen.

Außerdem bietet RobotController noch die Möglichkeit, IO-Bit-Variablen nach außen zu exportieren. Mittels zweier Methoden können so andere Klassen IO-Bits lesen und setzen. Dies ist zum Beispiel für das Transportsystem relevant, welches sich über System-Bitvariablen des ersten Roboters ansteuern lässt (siehe Abschnitt 8.6 auf Seite 88).

B.1.3 Die Command- und Response-Objekte

Die Kommunikation zwischen Roboter verläuft nach einem einfachen *Query-Response-Protokoll*. Ein Kommando wird an den Roboter geschickt, welcher es ausführt und dann eine Antwort an den Host zurück sendet. Das Protokoll baut – wie oben schon erwähnt – auf TCP/IP auf und verwendet als Datenformat XML. Es gibt unterschiedliche Typen von Kommandos:

`getVar` zum Abfragen einer Robotervariable

`setVar` zum Setzen einer Robotervariable

`bitsetVar` zum Setzen und Löschen von Bits in einer Robotervariable vom Typ Integer

Zu jedem Kommando gibt es eine Antwort vom gleichen Typ. Außerdem gibt es noch eine Antwort vom Typ NAK[1]. Diese meldet einen Fehler in der Verarbeitung.

Ein Kommando kapselt sich in eine Unterklasse der abstrakten Klasse RobotCommand. Eine Antwort wird durch eine Unterklasse von RobotResponse repräsentiert.

Jedes Kommando trägt eine sog. *client stamp*, welche einen eindeutigen Stempel darstellt und mit der Antwort wieder zurückgesendet wird. Obigen drei Nachrichtentypen ist auch die Spezifikation des Namens der abgefragten Variable gemeinsam. Diese beiden Werte können für jede RobotMessage gesetzt und gelesen werden. Außerdem hat jede RobotMessage eine Methode type(), die entweder CMD oder RES zurückliefert. Die Methode subType() bestimmt den Typ des Kommandos. Diese Methoden werden durch die Unterklassen überschrieben, um die richtigen Werte per Polymorphie zurückzugeben.

Jede RobotResponse liefert noch einen Zeitstempel mit, der den Zeitpunkt der Ausführung des zugehörigen Kommandos darstellt.

1 No Acknowledgement

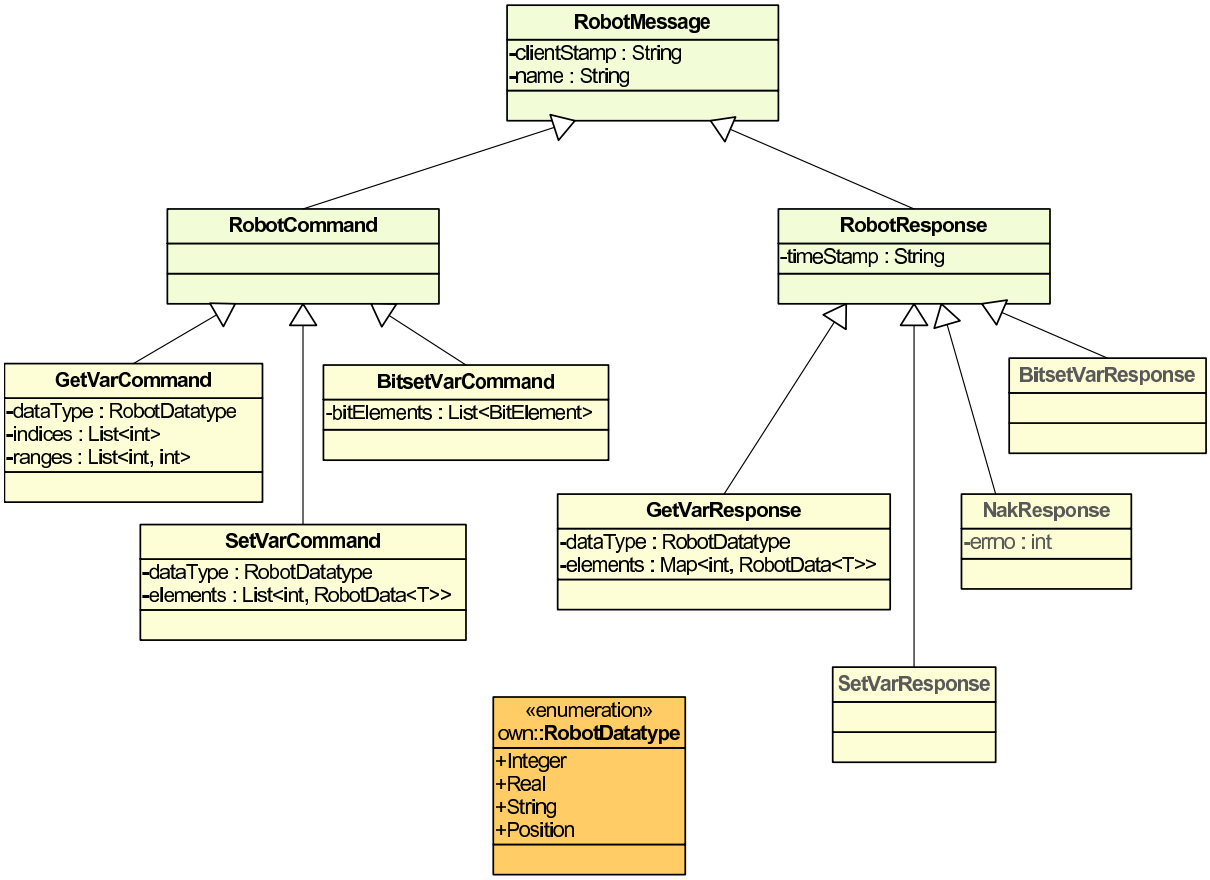


Bild B.2: Die Hierarchie der Command- und Response-Objekte

Für jeden Nachrichtentyp gibt es eine Unterklasse von RobotCommand und RobotResponse. Dies wird in Bild B.2 dargestellt.

B.1.3.1 Kommando und Antwort für den Typ `getVar`

Ein `getVar`-Kommando kann eine Variable oder mehrere Felder eines Arrays abfragen. Dazu können beliebig viele Feldindizes und Feldbereiche angegeben werden. Außerdem muss der Datentyp der abzufragenden Variable gesetzt werden. Die Antwort liefert dann Index-Wert-Paare zurück.

Mittels addIndex(int) und addRange(Range) können Indizes und Bereiche zu der Klasse GetVarCommand hinzugefügt werden. Der Datentyp wird mit setDataType(RobotDataType) gesetzt. Die Klasse GetVarResponse bietet die Methode element(int i) an, die den abgefragten Wert zum Index i zurückliefert.

B.1.3.2 Kommando und Antwort für den Typ `setVar`

Analog zu `getVar` kann mit dem `setVar`-Kommando eine Variable oder mehrere Felder eines Arrays gesetzt werden.

Der Klasse SetVarCommand werden dazu mittels der Methode addElement Index-Wert-Paare übergeben. Die Antwort (Klasse: SetVarResponse) enthält keine wichtigen Informationen. Sie dient nur der Bestätigung der Ausführung.

B.1.3.3 Kommando und Antwort für den Typ `bitsetVar`

Mittels eines `bitsetVar`-Kommandos können Bits in einer Robotervariable vom Typ Integer gesetzt und gelöscht werden. Die Klasse BitsetVarCommand bietet dazu die Methode addBitElement(BitElement) an. Ein BitElement kann entweder das Setzen oder das Löschen eines Bits bedeuten. Somit können durch Hinzufügen mehrere Bitelemente mehrere Bits auf einmal gelöscht oder gesetzt werden. Ähnlich wie bei der `setVar`-Antwort trägt die Klasse BitsetVarResponse keine wichtigen Informationen und dient nur zur Benachrichtigung der korrekten Ausführung.

B.1.4 Die Erzeugung von XML-Kommandos und das Parsen der XML-Antworten

Wird die Methode read() einer RobotVariable aufgerufen, so delegiert diese an die read-Methode der zugehörigen Lesestrategie. Hier wird ein GetVarCommand erzeugt und mit den übergegeben Angaben zur Variable, die gelesen werden soll gefüllt. Anschließend wird die Methode sendCommand des Interfaces RobotCommunication aufgerufen. RobotCommunication stellt einen Kommunikationskanal zum Roboter dar, über den Kommandos gesendet und Antworten empfangen werden.

Eine Implementierung dieses Interfaces ist RobotXMLRPCCommunication. Diese wandelt die Command- und Response-Objekte in eine Art XML-RPC[1] um. Dazu wird eine Implementierung des Interfaces CommandSerializer und eine Implementierung von ResponseParser benutzt.

B.1.4.1 Die Klasse XMLCommandSerializer

Eine Implementierung von CommandSerializer ist die Klasse XMLCommandSerializer. Zur Umsetzung eines Command-Objekts nach XML wird die DOM[2]-Implementierung der Java-Klassenbibliothek (Paket `javax.xml`) benutzt. Nach und nach werden die Elemente eines Commands in den XML-Baum eingefügt, bis das XML-Dokument vollständig ist. Siehe Bild B.3 auf der nächsten Seite für eine genauere Beschreibung. Außerdem kann dem XMLCommandSerializer eine Implementierung

1 XML-Remote-Procedure-Call
2 Document-Object-Model

des Interfaces ClientStampGenerator übergeben werden, welche oben beschriebene ClientStamps erzeugt.

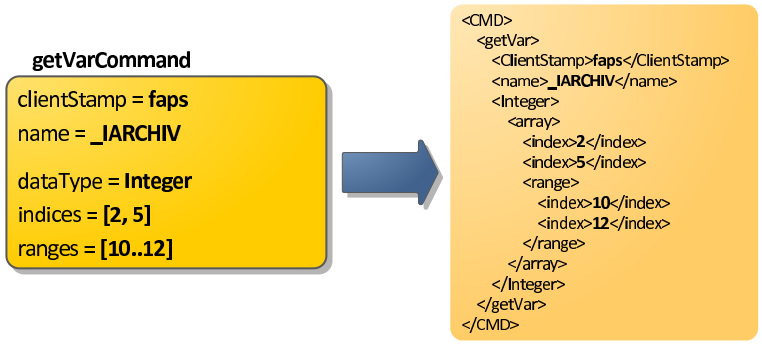


Bild B.3: Die Umsetzung eines `GetVarCommand` in die XML-Form

B.1.4.2 Die Klasse XMLResponseParser

Zur Transformierung eines empfangenen XML-Strings vom Roboter in ein Response-Objekt wird die Klasse XMLResponseParser – eine Implementierung von ResponeParser – benutzt. Das Parsen des Strings wird ebenfalls von der Java-DOM-Implementierung unterstützt. Es wird ein neues Response-Objekt des passenden Typs angelegt und nach und nach mit den Informationen aus dem XML-String gefüllt. Bild B.4 verdeutlicht diesen Vorgang.

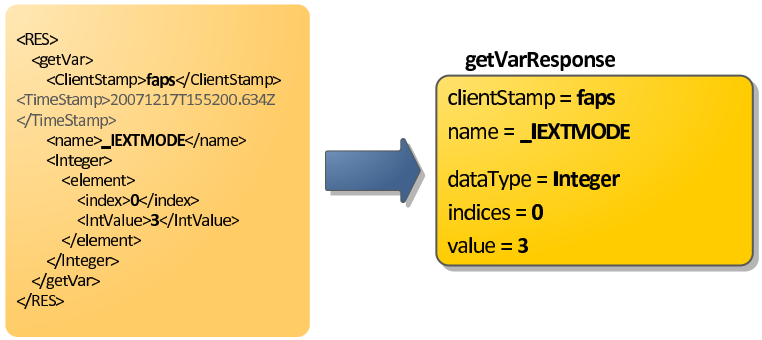


Bild B.4: Die Transformation eines emfangenen XML-Strings in eine `GetVarResponse`

B.1.5 Senden und Empfangen über einen Socket

Ist das Kommando in XML umgesetzt, so benutzt RobotXMLRPCCommunication eine Instanz der Klasse SocketCommunication, um einen Socket zu öffnen, das Kommando zu senden, die Antwort zu empfangen und den Socket wieder zu schließen. Pro Kommando wird also immer ein neuer Socket geöffnet und somit eine neue TCP-Verbindung gestartet. Für jeden Verbindungsaufbau muss ein sogenannter *Three-Way-Handshake* durchgeführt werden, der aus dem Senden eines

Verbindungsaufbauwunsches, dem Empfang einer Antwort sowie aus dem erneuten Senden einer Bestätigung besteht. Somit zieht jeder Aufbau einer TCP-Verbindung eine Verzögerung von drei Round-Trip-Times nach sich. Dabei ist eine RTT[1] die Dauer, die eine Nachricht vom Sender zum Empfänger benötigt. Dies ist zusätzlich zur Verarbeitung des Befehls im Roboter eine weitere Verzögerung der Ausführung.

Zusätzlich können die Zeiten, die zwischen Senden des Kommandos und Empfangen der Antwort vergehen aufgezeichnet werden.

B.1.6 Socket-Verbindungsabbrüche und erneutes Senden

Es kommt vor, dass der RL-16-Roboter von Zeit zu Zeit nach dem Empfangen des Kommandos die Socket-Verbindung abbricht. Die Lösung für dieses Problem ist das erneute Senden des Kommandos. Es kann eingestellt werden, wie oft maximal versucht wird, das Kommando erneut zu senden. Zwischen Abbruch des Sockets und erneutem Senden wird eine bestimmte Zeit gewartet, die mit der Anzahl der Sendeversuche linear ansteigt.

Diese Funktionalität des *erneuten Sendens* ist in der Klasse RobotXMLRPCCommunication zu finden.

B.2 Kommunikation mit dem Hexapod

Für die Kommunikation mit dem Hexapod steht eine externe Bibliothek in Form einer DLL[2] zur Verfügung. Diese implementiert die Steuerung des Hexapods und das Senden von Kommandos. Mittels des JNI[3] wird diese Bibliothek angesprochen.

Das Interface HexapodController bietet einige Methoden zur Steuerung des Hexapods. Dabei ist vor allem die Methode gotoPosition(HexapodPosition) zum Senden einer Hexapodposition und isMoving zur Abfrage der Bewegung wichtig. Eine Hexapodposition besteht aus drei kartesischen Koordinaten x, y und z und drei Winkeln a, b und c, die Drehwinkel um die jeweiligen kartesischen Achsen darstellen.

Die Klasse HexapodNativeController implementiert das Interface HexapodController und delegiert alle Methodenaufrufe an die Native-Implementierung in der DLL-Datei.

1 Round-Trip-Time
2 Dynamic-Link Library
3 Java-Native-Interface