Entwurf und Implementierung einer Bahnplanungsmethode für kooperierende Industrieroboter
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 1

2 Stand der Technik 4

2.1 Kooperierende Roboter .............................................. 4
2.2 Online- und Offline-Programmierung .................................. 4
2.3 Autonome Bahnplanung .............................................. 5
  2.3.1 Landkartenmethode ........................................... 5
  2.3.2 Zellzerlegungsmethode ........................................ 6
  2.3.3 Potentialfeldmethode .......................................... 7
2.4 Weitere Probleme bei der Bahnplanung mit realen Robotern .......... 7
2.5 Zusammenfassung .................................................. 8

3 Bereichssperrverfahren und die Montagezelle 9

3.1 Das Bereichssperrverfahren ...................................... 9
3.2 Ablauf des Beispielmontageprozesses .................................. 10
3.3 Die beiden Industrieroboter ....................................... 12
3.4 Die Greifer und der Greiferbahnhof ................................. 13
3.5 Ansteuerung der externen Peripheriegeräte ......................... 14
3.6 Hexapod zur Verkippung des Trägerbauteils ....................... 14
3.7 Laserscanner zur Bauteilerkennung und Lagebestimmung .......... 15
3.8 Die Umsetzung des Bereichssperrverfahrens in der vorgestellten Montagezelle ........................................... 16
3.9 Zusammenfassung .................................................. 17

4 Grundlegende Konzepte des Bahnplaners 18

4.1 Montage- und Bahnplanung mit einem Roboter ....................... 18
  4.1.1 Der Task als elementarer Baustein der Montageplanung ............ 18
  4.1.2 Der Montagegraph und Scheduling der Tasks ....................... 19
  4.1.3 Die Komplexoperationen ....................................... 20
  4.1.4 Zerlegung eines Tasks in einzelne Komplexoperationen ............ 20
  4.1.5 Die Elementaroperationen ..................................... 21
Inhaltsverzeichnis

4.1.6 Zerlegung der Komplex- in Elementaroperationen .................. 22
4.1.7 Ausführung der Elementaroperationen ................................. 27
4.1.8 Die vier Ressourcentypen ............................................. 27
4.2 Montage- und Bahnplanung mit mehrerenRobotern ...................... 28
  4.2.1 Zusätzliche Elementaroperationen zur Synchronisierung .......... 28
  4.2.2 Erweiterte Zerlegung der Komplex- in Elementaroperationen ...... 30
  4.2.3 Synchronisation über das Sperren von Arbeitsbereichen ........ 33
4.3 Zusammenfassung .................................................. 39

5 Architekturentwurf ................................................. 40
  5.1 Grundlagen an Architektur- und Entwurfsmuster ...................... 40
    5.1.1 Schichtenmodell ............................................... 40
    5.1.2 Observer-Pattern ............................................. 41
    5.1.3 Strategy-Pattern ............................................. 42
  5.2 Überblick über die Architektur ..................................... 43
    5.2.1 Graphische Bedienoberfläche ................................ 43
    5.2.2 Operation-Schicht und Implementierung der Bahnplanungslogik 44
    5.2.3 Cell-Schicht und Modell der Hardware .......................... 45
    5.2.4 Communication-Schicht und XML-Umsetzung ................... 46
  5.3 Entscheidung für die Programmiersprache Java ....................... 46

6 Graphische Bedienoberfläche ....................................... 48
  6.1 Aufbau der Oberfläche ............................................ 48
  6.2 Die Sicht „robot information“ .................................... 48
  6.3 Die Sicht Ressourcen ............................................ 50
  6.4 Die Montagegraph-Sicht ........................................... 51
  6.5 Die Sicht „robot manual control“ ................................ 52
  6.6 Die Sicht „goto position“ ..................................... 53
  6.7 Die Sicht für das Transportsystem ................................ 54

7 Operation-Schicht und Implementierung der Bahnplanungslogik ... 55
  7.1 Die Klasse TaskProcessor und das Scheduling der Tasks ................ 55
  7.2 Die Klassen RobotManager und TaskDemuxer .......................... 58
  7.3 Die Klassen COPExecutor und COPDemuxer .......................... 58
  7.4 Die verschiedenen EOPExecutors .................................. 60
    7.4.1 ActionController ........................................... 60
    7.4.2 HexapodExecutor ........................................... 60
Inhaltsverzeichnis

7.4.3 WaitOnSelf- und WaitOnHexapodExecutor ........................................... 60
7.4.4 DelayExecutor ......................................................................................... 62
7.4.5 GotoPositionExecutor und PathPreparing ............................................ 62
7.4.6 SyncOnHexapodExecutor ........................................................................ 63
7.4.7 SyncOnGripperExecutor .......................................................................... 63
7.4.8 SyncOnFreePortExecutor .......................................................................... 63
7.4.9 SyncOnSpaceExecutor und PathPlanners ................................................ 63
7.4.10 ReleaseHexapodExecutor ....................................................................... 77
7.4.11 ReleaseGripperExecutor ........................................................................ 77
7.4.12 ReleasePortExecutor ............................................................................... 78
7.4.13 ReleaseSpaceExecutor ........................................................................... 78
7.5 Die ResourceLocker .................................................................................... 79
7.6 Zusammenfassung ........................................................................................ 80

8 Cell-Schicht und Modell der Hardware ............................................................. 81
  8.1 Das Modell der Roboter .............................................................................. 81
    8.1.1 Das Interface MovableRobot ............................................................... 82
    8.1.2 Das Interface GripperAwareRobot ..................................................... 83
    8.1.3 Das Interface PollingPublisherToGUITier ......................................... 83
    8.1.4 Die Klasse CellRobot ........................................................................... 84
    8.1.5 Die Vermeidung von Drehwinkeln außerhalb des gültigen Rotationsbereichs 85
  8.2 Die Klasse Hexapod .................................................................................... 86
  8.3 Die Klasse Gripper und deren Unterklassen .............................................. 86
  8.4 Das Interface GripperAttachable und die Klasse GripperAttacher .............. 87
  8.5 Die Klassen GripperPort und GripperStation .......................................... 88
  8.6 Die Klasse TransportationSystem ........................................................... 88
  8.7 Die Klasse CellPosition und Koordinatenumrechnung .............................. 89

9 Ausblick ........................................................................................................... 90

10 Zusammenfassung ........................................................................................ 92

11 Abbildungsverzeichnis .................................................................................. 94

12 Abkürzungsverzeichnis ................................................................................... 97

A Instanziierung und Konfiguration des Systems ............................................... 98
  A.1 Die Klasse RobotFacility zur Instanziierung des Systems ....................... 98
Inhaltsverzeichnis

A.2 Die Klasse RobotFacilityProperties ........................................ 99

B Communication-Schicht und XML-Umsetzung ................................. 101

B.1 Kommunikation mit den Robotern ........................................... 101
  B.1.1 Das Lesen und Schreiben von Robotervariablen ....................... 101
  B.1.2 Die Klasse RobotController .............................................. 104
  B.1.3 Die Command- und Response-Objekte .................................. 105
  B.1.4 XML-Kommandos und -Antworten ...................................... 107
  B.1.5 Senden und Empfangen über einen Socket .............................. 108
  B.1.6 Socket-Verbindungsabbrüche und erneutes Senden .................... 109

B.2 Kommunikation mit dem Hexapod ........................................... 109
1 Einleitung


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.


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.


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.


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.

1 Einleitung

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


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.

### 2.3 Autonome Bahnplanung


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 Einfachheit und dem Umstand geschuldet, dass auch der in dieser Arbeit vorgestellte Bahnplaner zweidimensional arbeitet.

#### 2.3.1 Landkartenmethode


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

Bild 2.1: Die Planung von Verfahrwegen mittels der Landkartenmethode.

2.3.2 Zellzerlegungsmethode


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

2.3.3 Potentialfeldmethode


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.
2.5 Zusammenfassung

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


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


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

3.2 Ablauf des Beispielmontageprozesses

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

\[\text{Bild 3.1: Drei Arbeitsbereiche, die im Rahmen des Bereichsspererverfahrens vor dem Einfahren reserviert werden müssen und Rückzugsbereiche zur Deadlock-Vermeidung für zwei Roboter.}\]


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.

3.2 Ablauf des Beispielmontageprozesses


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

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/IP¹-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. 2m × 1m × 1m 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 Positionierungs genauigkeit wird von Reis mit ±0,1 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


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


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.


### 3.4 Die Greifer und der Greiferbahnhof


---

\(^1\) Point-To-Point

\(^2\) Controlled Path - linear
3.5 Ansteuerung der externen Peripheriegeräte


3.5 Ansteuerung der externen Peripheriegeräte


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.

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.
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 Arbeitsbereiche in der Montagezelle eingegangen. Die Arbeitsbereiche werden über eine externe Konfiguration eingelesen, so dass deren Maße und Positionen einfach verändert werden können.

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

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.


4.1 Montage- und Bahnplanung mit einem Roboter


4.1.1 Der Task als elementarer Baustein der Montageplanung

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.


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.
4.1 Montage- und Bahnplanung mit einem Roboter

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


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


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 Detailschritte, die konzeptionell 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


---

1 Elementary Operations
4.1 Montage- und Bahnplanung mit einem Roboter


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


4.1.6.2 Zerlegung der FetchGripperOp

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.


Basisgeschwindigkeit < Zwischengeschwindigkeit < 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
10. WaitOnSelfOp
11. MountGripperOp(Greifer)
12. DelayOp(timeout)
14. WaitOnSelfOp
15. ChangeVelocityOp(Zwischengeschwindigkeit)
17. WaitOnSelfOp
4.1 Montage- und Bahnplanung mit einem Roboter


4.1.6.3 Zerlegung der \textit{ParkGripperOp}


Es ergibt sich folgende Liste:

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

4.1 Montage- und Bahnplanung mit einem Roboter

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)
7. WaitOnSelfOp
8. ChangeVelocityOp(Pick-Geschwindigkeit)
10. WaitOnSelfOp
11. CloseGripperOp
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)
4.1 Montage- und Bahnplanung mit einem Roboter

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
14. WaitOnSelfOp
15. ChangeVelocityOp(Zwischengeschwindigkeit)
17. WaitOnSelfOp


4.1.6.6 Verschmelzung der GotoPositionOp und der WaitOnSelfOp


4.1 Montage- und Bahnplanung mit einem Roboter

4.1.7 Ausführung der Elementaroperationen


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

4.2 Montage- und Bahnplanung mit mehreren Robotern


4.2.1 Zusätzliche Elementaroperationen zur Synchronisierung


4.2 Montage- und Bahnplanung mit mehreren Robotern


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.


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


4.2 Montage- und Bahnplanung mit mehreren Robotern


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


1. \texttt{ChangeVelocityOp}(Basisgeschwindigkeit)
2. \texttt{SyncOnGripperOp}(GripperType)
3. \texttt{SyncOnSpaceOp}
4. \texttt{GotoPositionOp}(über dem Port - z maximal)
5. \texttt{ReleaseSpaceOp}
6. \texttt{WaitOnSelfOp}
7. \texttt{ChangeVelocityOp}(Portgeschwindigkeit)
4.2 Montage- und Bahnplanung mit mehreren Robotern

8. RetractMountingBoltsOp
10. WaitOnSelfOp
11. MountGripperOp(Greifer)
12. DelayOp(timeout)
14. WaitOnSelfOp

4.2.2.2 Zerlegung der ParkGripperOp


1. ChangeVelocityOp(Basisgeschwindigkeit)
2. SyncOnFreePortOp
3. SyncOnSpaceOp
4. GotoPositionOp(vor dem Port)
5. ReleaseSpaceOp
6. WaitOnSelfOp
7. ChangeVelocityOp(Portgeschwindigkeit)
9. WaitOnSelfOp
10. UnmountGripperOp(Zielport)
11. DelayOp(timeout)
12. ReleaseGripperOp
14. ReleasePortOp
15. WaitOnSelfOp
4.2 Montage- und Bahnplanung mit mehreren Robotern

4.2.2.3 Zerlegung der PickOp


1. ChangeVelocityOp(Basisgeschwindigkeit)
2. SyncOnSpaceOp
4. ReleaseSpaceOp
5. WaitOnSelfOp
6. OpenGripperOp
7. ChangeVelocityOp(Pick-Geschwindigkeit)
8. GotoPositionOp(Pick-Position)
9. WaitOnSelfOp
10. CloseGripperOp
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
4.2 Montage- und Bahnplanung mit mehreren Robotern

7. WaitOnSelfOp
8. ChangeVelocityOp(Place-Geschwindigkeit)
9. WaitOnHexapodOp
10. GotoPositionOp(Place-Position)
11. WaitOnSelfOp
12. OpenGripperOp
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

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.

\begin{figure}[h]
  \centering
  \includegraphics[width=\textwidth]{image}
  \caption{Die verschiedenen Arbeitsbereiche der Zelle in der Draufsicht.}
\end{figure}

\footnote{1 \textit{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.


4.2.3.3 Verfahrwege bei der Kollisionsvermeidung

4.2 Montage- und Bahnplanung mit mehreren Robotern

Pick-Place-Ausweichbewegung

Steh 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. ...


\textbf{Place-Station-Ausweichbewegung}

Soll der Bereich \textit{Station} von \textit{Place} aus angefahren werden, und ist \textit{Station} bei Ausführung der \texttt{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.


\textbf{Pick-Station-Ausweichbewegung}

Bild 4.7 auf der nächsten Seite zeigt die Ausweichbewegung bei der Bewegung vom \textit{Pick} zum \textit{Station}-Bereich. Hier wird ähnlich wie bei der Ausweichbewegung vom \textit{Place}- in den \textit{Station}-Bereich am äußersten Punkt des Rückzugsbereichs gewartet.
4.2 Montage- und Bahnplanung mit mehreren Robotern

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 4.6: Ausweichbewegung bei der Station-Place-Bewegung

Bild 4.7: Ausweichbewegung bei der Pick-Station-Bewegung
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.


Bild 4.8: Direktbewegung Pick-Station
5 Architekturentwurf


5.1 Grundlagen an Architektur- und Entwurfsmuster


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

5.1 Grundlagen an Architektur- und Entwurfsmuster

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

\[\text{Bild 5.1: Ein Ausschnitt aus einem konzeptionellen Schichtenmodell}\]

\[\text{Bild 5.2: Klassendiagramm eines Observer-Patterns}\]

---

\(^1\) Open Systems Interconnection
5.1 Grundlagen an Architektur- und Entwurfsmustern

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.

![Sequenzdiagramm eines Observer-Patterns mit zwei Beobachtern](image)

**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
überblick über die Architektur

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.
5.2 Überblick über die Architektur

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 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.
5.2 Überblick über die Architektur

5.2.3 Cell-Schicht und Modell der Hardware


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,
5.3 Entscheidung für die Programmiersprache Java

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


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 Einfachheit der Sprache, Garbage

---

\(^1\) Integrated Development Environment
6 Graphische Bedienoberfläche


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.


6.2 Die Sicht „robot information“

Die erste Karteikarte ist eine allgemeine Übersicht über die Position der Robotier und die Reservierung der Bereiche. Dargestellt ist eine Draufsicht auf die Zelle mit den einzeichneten Bereichen und den Bounding Boxen der Robotier. Sie ist in Bild 6.1 dargestellt.
6.2 Die Sicht „robot information“


6.3 Die Sicht Ressourcen


Bild 6.2: Die Ressourcen-Sicht
6.4 Die Montagegraph-Sicht


![Montagegraph mit Tasks als Knoten](image)

**Bild 6.3:** Der Montagegraph mit Tasks als Knoten

6.5 Die Sicht „robot manual control“


**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
6.6 Die Sicht „goto position“


Bild 6.5: Die Sicht „goto position“

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 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
7.1 Die Klasse TaskProcessor und das Scheduling der 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.

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.

![Diagramm: Der Ablauf des Task-Scheduling und der Task-Ausführung]

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
Die Klassen RobotManager und TaskDemuxer


Die Klassen COPExecutive und COPDemuxer


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
7.3 Die Klassen COPExecutor und COPDemuxer

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


7.4.1 ActionExecutor


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

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:

```java
protected List<ElementaryOp> executeEOPs(List<ElementaryOp> eops) {
    eops.remove(0);
    // add this as listener for positions of the robot
    robot.addSubscriber(this, pollingInterval);
    try {
        lock.lock();
        // wait for arrival of position
        arrivedAtPosition.await();
    }
    finally {
        lock.unlock();
        // remove listener again
        robot.removeSubscriber(this);
    }
    return eops;
}

public void update(Publisher publisher, Object arg) {
    CellPosition cellPosition = (CellPosition)arg;
    if(cellPosition.equalsWithTolerance(robot.getLastSentCellPosition())) {
        lock.lock();
        arrivedAtPosition.signal();
        lock.unlock();
    }
}
```
7.4 Die verschiedenen EOPExecutors

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


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 relater 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.

7.4 Die verschiedenen EOPExecutors

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

7.4.6 SyncOnHexapodExecutor


7.4.7 SyncOnGripperExecutor


7.4.8 SyncOnFreePortExecutor


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
7.4 Die verschiedenen EOPExecutors


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


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.


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.


7.4 Die verschiedenen EOPExecutors

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:

```java
protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint,
   Area retractionArea) {
    List<ElementaryOp> eops = new LinkedList<ElementaryOp>();
    if(areaResourceLocker.isAvailable(toArea)) {
        // possible violation of the Retract-On-Locked-Area-Paradigm
        eops.add(new AcquireSpaceLockOp(robot, toArea));
    } else {
        // PICK to PLACE _and_ PLACE to PICK
        eops.add(new SyncOnSpaceOp(robot, retractionArea));
        int positionNumber = forth ? 1 : 2;
        eops.add(new GotoPositionOp(positionLookup.lookupPosition(robot,
            positionNumber), SpecialPositionType.RETRACTION));
        eops.add(new SyncOnSpaceOp(robot, toArea));
    }
    return eops;
}
```

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.
7.4 Die verschiedenen EOPExecutors

**Freier Zielbereich**


```
1. SyncOnSpaceOp(Pick, Place)
2. GotoPositionOp(Place-Position)
```

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

**Belegter Zielbereich**


```
1. SyncOnSpaceOp(Pick, Place)
2. GotoPositionOp(Place-Position)
```

```
1. SyncOnSpaceOp(Pick, Retraction)
2. GotoPositionOp(Retraction-Pos)
3. SyncOnSpaceOp(Retraction, Place)
4. GotoPositionOp(Place-Position)
```

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

**Virtuelle Parameter Start- und Zielbereich**


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


**Position-Lookup**


**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 unvorhergesehen Ablauf und kein Problem, das zu einem Fehlverhalten oder gar zu einer Kollision führen würde.
7.4 Die verschiedenen EOPExecutors

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.

```java
protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint, Area retractionArea) {
    List<ElementaryOp> eops = new LinkedList<ElementaryOp>();
    if(areaResourceLocker.isAvailable(toArea)) {
        // possible ROLAP-VIOLATION
        // MOVE DIRECTLY
        eops.add(new AcquireSpaceLockOp(robot, toArea));
        // INSERTION OF SUPPORT POINTS
        // ...   // ...
    } else {
        // PLACE to STATION _and_ STATION to PLACE
        eops.add(new SyncOnSpaceOp(robot, retractionArea));
        int positionNumber = forth ? 3 : 4;
        eops.add(new GotoPositionOp(positionLookup.lookupPosition(robot, positionNumber), SpecialPositionType.RETRACTION));
        eops.add(new SyncOnSpaceOp(robot, toArea));
    }
    return eops;
}
```

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 Bahnhofsereich ein Stützpunkt eingefügt werden. Außerdem muss beim Eintreten oder Verlassen des Bahnhofsereichs 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.

```java
// 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

// 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)
}
```
7.4 Die verschiedenen EOPExecutors

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.


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:

```java
protected List<ElementaryOp> planPathImpl(Robot robot, CellPosition targetPoint, Area retractionArea) {
    List<ElementaryOp> eops = new LinkedList<ElementaryOp>();
    List<SpatialResource> placeAndToArea = new ArrayList<SpatialResource>(2);
    placeAndToArea.add(placeArea);
    placeAndToArea.add(toArea);
    if(areaResourceLocker.areAvailable(placeAndToArea)) {
        // possible ROLAP-VIOLATION
        // MOVE DIRECTLY
        eops.add(new AcquireSpaceLockOp(robot, placeAndToArea));
        // INSERTION OF SUPPORT POINTS
        // ...
        // ...
```
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 eingesetzt 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.

7.4.9.5 RetractionPickPathPlanner

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

```java
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;
}
```


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

### 7.4.9.6 RetractionPlacePathPlanner

7.4 Die verschiedenen EOPExecutors

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

Place belegt

1. SyncOnSpaceOp(Pick, Retraction)
2. GotoPositionOp(Retraction-Pos)
3. SyncOnSpaceOp(Retraction, Place)
4. GotoPositionOp(Place-Position)

Place belegt

1. AcquireSpaceLockOp(Retraction)
2. GotoPositionOp(Retraction-Pos)
3. SyncOnSpaceOp(Retraction, Place)
4. GotoPositionOp(Place-Position)

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.

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

    if (forth) {
        // RETRACTION to STATION
        if (areaResourceLocker.isAvailable(stationArea)) {
            // possible ROLAP-VIOLATION
            eops.add(new AcquireSpaceLockOp(robot, stationArea));

            // close the gripper to avoid collision in the station
            if (robot.hasAttachedGripper())
                eops.add(new CloseGripperOp());

            eops.add(new ReleaseSpaceOp(robot, retractionArea));
        }
    }
```
7.4 Die verschiedenen EOPExecutors

```java
else {
    CellPosition waitPoint = positionLookup.lookupPosition(robot, 3);
    CellPosition lastSent = robot.getLastSentCellPosition();
    if (!lastSent.getXYPPoint().equals(waitPoint.getXYPPoint()))
        eops.add(new GotoPositionOp(waitPoint, SpecialPositionType.WAIT_POINT));

    eops.add(new ReleaseSpaceOp(robot, retractionArea));
    eops.add(new AcquireSpaceLockOp(robot, stationArea));

    // close the gripper to avoid collision in the station
    if (robot.hasAttachedGripper())
        eops.add(new CloseGripperOp());
}

// goto x projection of target and UP
eops.add(new GotoPositionOp(getXProjection(targetPoint, new Coord(positionLookup.getUpperZ()), targetPoint.getToolAngle()), SpecialPositionType.SUPPORT_POINT));

// above target point
eops.add(new GotoPositionOp(new CellPosition(targetPoint.getX(), targetPoint.getY(), positionLookup.getUpperZ(), targetPoint.getToolAngle()), SpecialPositionType.SUPPORT_POINT));
else {
    // STATION to RETRACTION
    eops.add(new AcquireSpaceLockOp(robot, retractionArea));

    // close the gripper to avoid collision in the station
    if (robot.hasAttachedGripper())
        eops.add(new CloseGripperOp());

    eops.add(new GotoPositionOp(new CellPosition(Coord.keep, Coord.keep, new Coord(positionLookup.getUpperZ()), ToolAngle.keep), SpecialPositionType.SUPPORT_POINT));
    eops.add(new GotoPositionOp(positionLookup.lookupPosition(robot, 4, Coord.keep, ToolAngle.keep), SpecialPositionType.SUPPORT_POINT));
}
return eops;
```

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
7.4 Die verschiedenen EOPExecutors


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.

1. SyncOnSpaceOp(Retraction, Station)  
2. GotoPositionOp(Station-Position)

1. AcquireSpaceLockOp(Station)  
2. CloseGripperOp  
3. ReleaseSpaceOp(Retraction)  
4. GotoPositionOp(Support-Point 1)  
5. GotoPositionOp(Support-Point 2)  
6. GotoPositionOp(Station-Position)

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


7.4.11 ReleaseGripperExecutor


```
protected List<ElementaryOp> executeEOPs(List<ElementaryOp> eops) {
    ReleaseGripperOp op = (ReleaseGripperOp)eops.remove(0);
    List<Gripper> acquiredPorts = gripperResourceLocker.getAllResourcesForRobot(op.getRobot());
    // Single-Resource-Restriction
    assert(acquiredPorts.size() == 1);
```
7.4 Die verschiedenen EOPExecutors


7.4.12 ReleasePortExecutor


7.4.13 ReleaseSpaceExecutor


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

```java
// Single-Resource-Restriction
gripperResourceLocker.releaseResource(acquiredPorts.get(0), op.getRobot());
return eops;
```
public void update(Publisher publisher, Object arg) {
    CellPosition cellPosition = (CellPosition)arg;

    if(current != null)
        old = current;

    current = areaManager.getAreaForPointWithTolerance(cellPosition.getXYPoint());

    if(current != null && current != old)
        areaResourceLocker.releaseResource(old, robot);

    if(current == target) {
        lock.lock();
        reachedTargetArea.signal();
        lock.unlock();
    }
}

7.5 Die ResourceLocker


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.
### 7.6 Zusammenfassung

**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

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.


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.
Das Modell der Roboter

`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`

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`


### 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 Roboter 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.


### 8.1.4 Die Klasse CellRobot


#### 8.1.4.1 Das Interface IOVariableExposingRobot


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.
8.1 Das Modell der Roboter


8.1.5 Die Vermeidung von Drehwinkeln außerhalb des gültigen Rotationsbereichs


**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 B₁ und B₂ 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
8.2 Die Klasse Hexapod


8.2 Die Klasse Hexapod


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.


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 instanziert.

**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


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...
8.5 Die Klassen GripperPort und GripperStation


8.5 Die Klassen GripperPort und GripperStation


Die Klasse GripperStation stellt eine Abstraktion des gesamten Greiferbahnhofs dar und verwaltet eine Liste der Ports. Es existieren Method en, 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 zurücklie fert.

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.

Beim Betätigen 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.


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.

9 Ausblick


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.


Jede dieser Erweiterungen kann wegen der Anordnung des Bahnplaners in Schichten, der Kapselung der Funktionalitäten und der strengen Achtung auf lose Kopplung der Komponenten einfach integriert werden.
10 Zusammenfassung


Die Beispielmontagesequenz der Türblechmontage wird mit dem entwickelten Bahnplaner erfolgreich und schnell abgearbeitet. Somit wurde am praktischen Beispiel gezeigt, dass sich das Bereichsspererverfahren 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


11 Abbildungsverzeichnis

2.1 Die Planung von Verfahrwegen mittels der Landkartenmethode.............. 6
2.2 Die Zerlegung der Zelle in Unterbereiche mit Hilfe der Zellzerlegungsmethode........ 6
2.3 Die Potentialfunktion der Potentialfeldmethode mit dem Ziel der Bewegung als Senke und Hindernissen als Quellen................................. 7

3.1 Drei Arbeitsbereiche, die im Rahmen des Bereichssperrverfahrens vor dem Einfahren reserviert werden müssen und Rückzugsbereiche zur Deadlock-Vermeidung für zwei Roboter.......................................................... 10
3.2 Die Gesamtzelle mit Robotern, dem Transportsystem und dem Hexapod..... 11
3.3 Simulationsmodell des RL-16 Roboter der Firma Reis Robotics.................. 12
3.4 Der Hexapod mit sechs Elektrozyllindern der Firma Raco als Montagetisch.... 15
3.5 Der Laserscanner scanCONTROL 2800 der Firma Micro-Epsilon................... 16
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................................. 17

4.1 Ein Beispiel für einen Montagegraph. Jede Montageaufgabe trägt eine eindeutige Nummer................................................................. 19
4.2 Die Hierarchie der Elementaroperationen............................................ 21
4.3 Die verschiedenen Arbeitsbereiche der Zelle in der Draufsicht............... 34
4.4 Ausweichbewegung bei der Pick-Place-Bewegung................................ 36
4.5 Ausweichbewegung bei der Place-Station-Bewegung............................ 37
4.6 Ausweichbewegung bei der Station-Place-Bewegung............................ 38
4.7 Ausweichbewegung bei der Pick-Station-Bewegung............................... 38
4.8 Direktbewegung Pick-Station............................................................. 39

5.1 Ein Ausschnitt aus einem konzeptionellen Schichtenmodell.................... 41
5.2 Klassendiagramm eines Observer-Patterns......................................... 41
5.3 Sequenzdiagramm eines Observer-Patterns mit zwei Beobachtern.............. 42
5.4 Klassendiagramm eines Strategy-Patterns.......................................... 43

94
<table>
<thead>
<tr>
<th>Seite</th>
<th>Abbildungsverzeichnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Sequenzdiagramm eines Strategy-Patterns</td>
</tr>
<tr>
<td>5.6</td>
<td>Die vier Schichten der Bahnplanungssoftware</td>
</tr>
<tr>
<td>5.7</td>
<td>Die vier Unterschichten der Operation-Schicht</td>
</tr>
<tr>
<td>5.8</td>
<td>Die Untergliederung der Cell-Schicht</td>
</tr>
<tr>
<td>5.9</td>
<td>Die vier Unterschichten der Communication-Schicht</td>
</tr>
<tr>
<td>6.1</td>
<td>Übersicht über die Position der Roboter und belegte Bereiche</td>
</tr>
<tr>
<td>6.2</td>
<td>Die Ressourcen-Sicht</td>
</tr>
<tr>
<td>6.3</td>
<td>Der Montagegraph mit Tasks als Knoten</td>
</tr>
<tr>
<td>6.4</td>
<td>Die Sicht „robot manual control“</td>
</tr>
<tr>
<td>6.5</td>
<td>Die Sicht „goto position“</td>
</tr>
<tr>
<td>6.6</td>
<td>Die Sicht für das Transportsystem</td>
</tr>
<tr>
<td>7.1</td>
<td>Die Klassenstruktur des Softwaremoduls für die Vergabe und Ausführung von Tasks</td>
</tr>
<tr>
<td>7.2</td>
<td>Der Ablauf des Task-Scheduling und der Task-Ausführung</td>
</tr>
<tr>
<td>7.3</td>
<td>Der Ablauf der Zerlegung eines Tasks in EOPs und deren Ausführung</td>
</tr>
<tr>
<td>7.4</td>
<td>Die EOP-Liste vor und nach Abarbeitung der SyncOnSpaceOp wenn der Platzbereich frei ist</td>
</tr>
<tr>
<td>7.5</td>
<td>Die EOP-Liste vor und nach Abarbeitung der SyncOnSpaceOp wenn der Platzbereich belegt ist</td>
</tr>
<tr>
<td>7.6</td>
<td>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.</td>
</tr>
<tr>
<td>7.7</td>
<td>Die Punkte, die die Methode lookupPosition zurückgibt</td>
</tr>
<tr>
<td>7.8</td>
<td>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.</td>
</tr>
<tr>
<td>7.9</td>
<td>Die EOP-Liste vor und nach Abarbeitung der SyncOnSpaceOp wenn der Platzbereich belegt, Station aber frei ist</td>
</tr>
<tr>
<td>7.10</td>
<td>Die EOP-Liste vor und nach Abarbeitung der SyncOnSpaceOp wenn der Platzbereich belegt ist</td>
</tr>
<tr>
<td>7.11</td>
<td>Die EOP-Liste vor und nach Abarbeitung der SyncOnSpaceOp wenn der Platzbereich belegt ist</td>
</tr>
<tr>
<td>7.12</td>
<td>Die EOP-Liste vor und nach Abarbeitung der SyncOnSpaceOp wenn der Platzbereich belegt, Station aber frei ist</td>
</tr>
<tr>
<td>8.1</td>
<td>Die Struktur des Interfaces Robot und der damit verbundenen Klassen und Interfaces</td>
</tr>
</tbody>
</table>
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. 85

B.1 Die Interfaces RobotVariable und BitVariable 103
B.2 Die Hierarchie der Command- und Response-Objekte 106
B.3 Die Umsetzung eines GetVarCommand in die XML-Form 108
B.4 Die Transformation eines empfangenen XML-Strings in eine GetVarResponse 108
12 Abkürzungsverzeichnis

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

A.1 Die Klasse RobotFacility zur Instanziierung des Systems


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.


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 |
A.2 Die Klasse RobotFacilityProperties


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.
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


1. Anzahl der Hauptachsen (Integer)
2. Anzahl der Zusatzachsen (Integer)
3. Positionstyp (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.
B.1 Kommunikation mit den Robotern

Framenummer 0 steht für den Base-Frame. Dies ist das jedem Roboter eigene, grundlegende Koordinatensystem. 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.


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

B.1.1.2 Das Interface RobotVariable


B.1 Kommunikation mit den Robotern


B.1 Kommunikation mit den Robotern

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.

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.


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
B.1 Kommunikation mit den Robotern

- 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


B.1.3 Die Command- und Response-Objekte


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.


---
\(^1\) No Acknowledgement

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


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 Kommunikation mit den Robotern

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.


B.1.3.3 Kommando und Antwort für den Typ bitsetVar


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


B.1.4.1 Die Klasse XMLCommandSerializer


---

1 XML-Remote-Procedure-Call
2 Document-Object-Model
B.1 Kommunikation mit den Robotern

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**


**Bild B.4:** Die Transformation eines empfangenen 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

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

### B.1.6 Socket-Verbindungsabbrüche und erneutes Senden


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