

Projektarbeit
BioMechatronik
Technische Fakultät

Universität Bielefeld
AG Kognitronik & Sensorik
Prof. Dr.-Ing. Ulrich Rückert

Potentialfeldbasierte Navigation mit dem AMiRo

Jan O'Sullivan

Betreuer: M.Sc. Timo Korthals

Bielefeld, Oktober 2018

Inhaltsverzeichnis

1 Einleitung	1
2 Grundlagen	3
2.1 Potentialfeldmethode	3
2.1.1 Anziehendes Potential	3
2.1.2 Abstoßendes Potential	6
2.1.3 Alternative Interpretation	8
2.2 Navigation nach der Potentialfeldmethode	9
3 Methodik	13
3.1 Rahmenbedingungen	13
3.1.1 Software-Framework <i>ROS</i>	13
3.1.2 <i>AMiRo</i>	14
3.1.3 Telewerkbank	14
4 Definition der Steuerungsarchitektur	15
4.1 Autonomous Robot Architecture <i>AuRA</i>	16
4.1.1 Theorie der Schemata	16
4.2 Aufbau und Komponenten der <i>AuRA</i>	20
4.2.1 <i>Missionplanner</i>	20
4.2.2 <i>Path Sequencer</i>	20
4.2.3 <i>Plan Sequencer</i>	21
4.2.4 <i>Schema Controller</i>	22
4.2.5 Fehlerbehandlung (<i>Exception</i>)	23
5 Übertragung der Steuerungsarchitektur in das Software-Framework <i>ROS</i>	24
5.1 <i>Schema Controller</i>	24
5.1.1 Entwurf	25
5.2 <i>Plan Sequencer</i>	28
5.2.1 Entwurf	28

6 Anwendung der Steuerungsarchitektur auf die Problemstellung <i>AMiRo Assembly Line</i>	30
6.1 AAL (<i>AMiRo Assembly Line</i>)	30
6.2 Definition der <i>motor schemas</i>	31
6.3 Umsetzung des <i>Plan Sequencers</i>	32
6.4 Umsetzung des <i>Schema Controllers</i>	33
6.4.1 <i>move-robot</i>	34
6.4.2 <i>wait-for-obstacle</i>	36
6.4.3 <i>avoid-obstacle</i>	37
6.4.4 <i>move-to-target</i>	40
6.4.5 <i>stay-on-path</i>	44
7 Ergebnisse	46
7.1 Simulationsumgebung	46
7.1.1 <i>Navigate to gateway</i>	46
7.1.2 <i>Navigate to descent</i>	47
7.1.3 <i>Navigate to goal</i>	49
7.2 Telewerkbank	50
8 Bewertung und Diskussion der Ergebnisse	51
9 Anhang	53
9.1 Pfadverfolgung mit der Potentialfeldmethode	53
9.1.1 Vektorfeld zur Bewegung entlang des Pfades	53
9.1.2 Vektorfeld zur Bewegung zum Pfad	61
Literaturverzeichnis	67

1 Einleitung

„Warum hybridisieren?“

Diese Frage greift Ronald Arkin in [Ark98] auf. Die Ausgangsposition für diese Fragestellung bilden die zwei Extreme der Roboterarchitekturen, die reaktive und die deliberative Architektur. Eine reaktive Architektur definiert die direkte Projektion von Sensorinformationen auf ein Bewegungsresultat. Eine deliberative Architektur hingegen enthält eine interne Repräsentation der Umgebung und plant anhand von Sensorinformationen eine Serie von Bewegungen. Diese Architekturen bilden Lösungen für Aufgaben, in entweder hoch dynamischen oder konstanten Umgebungen. Jedoch bilden diese Umgebungen Grenzfälle und entsprechen nur selten der Realität. Beispielsweise sind aus der Natur keine rein deliberativen Verhaltensweisen bekannt [Ark98]. Arkin formuliert an dieser Stelle die Aussage, dass Maschinen erst dann dem Menschen ähnlich agieren können, wenn sich diese auch ähnlich verhalten. Hiermit beschreibt Arkin den generellen Anspruch an eine Roboterarchitektur. Dieser besteht darin, ein Verhalten hervorzubringen, dass dem menschlichen Verhalten ähnelt oder es sogar übertrifft. Aus diesem Grund muss ein Modell definiert werden, dass das menschliche oder menschenähnliches Verhalten beschreibt. Ein solches Modell wurde 1987 von [AL95] vorgeschlagen und besagt, dass komplexes Verhalten die Summe simpler Verhaltensweisen darstellt. Diese These beschreibt nach dem heutigen Verständnis die verhaltensbasierten Roboterarchitekturen. Allerdings können verhaltensbasierte Roboterarchitekturen, ebenso wie die reaktiven und die deliberativen Architekturen, das menschliche Verhalten nicht vollständig abbilden. Arkin beschreibt daher das menschliche Verhalten als hybrid, genauer gesagt als eine Kombination aus deliberativer und verhaltensbasierter Steuerungsarchitektur. Auf dieser Grundlage definiert Arkin in [Ron97] die *AuRA* (*Autonomous Robot Architecture*).

Im Rahmen dieser Arbeit wird die von Arkin definierte *AuRA* als Ausgangspunkt verwendet. Im Verlauf der Arbeit werden zunächst notwendige Grundlagen erarbeitet und nachfolgend wird die *AuRA* neu interpretiert und in das Software-Framework *ROS* überführt. Des weiteren wird beschrieben, wie eine moderne Problemstellung aus der Industrie mit Hilfe der *AuRA* gelöst werden kann. Der Titel dieser Arbeit

1 Einleitung

bezieht sich dabei auf das zugrundeliegende Steuerungskonzept der *AuRA*, das der Potentialfeldmethode fast vollständig ähnelt.

2 Grundlagen

2.1 Potentialfeldmethode

Die Grundidee der Potentialfeldmethode bildet die Definition eines Agenten als Punktladung in einem zugrundeliegenden euklidischen Raum beliebiger Dimension n . Die Bewegung des Agenten im Raum wird dabei durch ein künstliches Potentialfeld beeinflusst. Das Potentialfeld ist so beschaffen, dass der Zielpunkt das globale Minimum, also den energieärmsten Punkt, bildet. Der Agent bewegt sich in einem solchen Potentialfeld entlang des negativen Gradienten in Richtung des Zielpunktes, um selbst den energieärmsten Zustand einzunehmen. Im Idealfall bildet ein Potentialfeld ein einzelnes, globales Minimum aus, was jedoch in realen Anwendungsfällen nur selten möglich ist. Entsprechende Beispiele sind unter Abschnitt 2.2 aufgeführt.

Im Allgemeinen bildet ein künstliches Potentialfeld die Kombination aus abstoßenden und anziehenden Potentialen. In diesem Zusammenhang beschreibt das anziehende Potential einen Zielpunkt und abstoßende Potentiale beschreiben nicht passierbare Bereiche, wie zum Beispiel Hindernisse. Die Intensität des Potentials und die damit verbundene Wirkung auf die Punktladung, wird durch eine Funktion der Distanz zum Potential beschrieben. Die genaue Definition einer solchen Funktion ist von mehreren Faktoren abhängig und muss in Abhängigkeit der Problemstellung definiert werden. Zu diesen Faktoren zählt unter anderem das gewünschte Geschwindigkeitsprofil, mit dem sich die Punktladung, beziehungsweise der Agent, innerhalb des Potentialfelds bewegen soll. Hierbei entspricht die Bewegungsgeschwindigkeit des Agenten dem Betrag des Gradienten in Abstiegsrichtung. Daher ist die Funktion des Potentials für das Geschwindigkeitsprofil maßgebend. Im Folgenden werden die grundlegenden Funktionen für Potentialfelder beschrieben, sowie dessen Vor- und Nachteile diskutiert.

2.1.1 Anziehendes Potential

Ein anziehendes Potential lässt sich durch eine Funktion der Distanz beschreiben, die im gesamten Definitionsbereich mit zunehmender Distanz stetig steigende Funktions-

werte aufweist. Die Distanz beschreibt hierbei den euklidischen Abstand zwischen der aktuellen Position der Punktladung und dem Zentrum des anziehenden Potentials, also dem Zielpunkt. Ein solcher Funktionscharakter lässt sich im einfachsten Fall durch eine lineare oder quadratische Funktion beschreiben. Die Gleichungen 2.1 und 2.3 bilden einen allgemeinen Ansatz dieser Funktionen. Die Gleichungen 2.2 und 2.4 definieren den Betrag der Gradienten in Abhängigkeit der Distanz zum Zielpunkt. Zur Vereinfachung wird nachfolgend die Bezeichnung „Gradient“ mit dem Betrag des Gradienten gleichgesetzt.

$$P_{att,lin}(d_{goal}) = \eta_{att} d_{goal} \quad (2.1)$$

$$\frac{dP_{att,lin}(d_{goal})}{dd_{goal}} = \eta_{att} \quad (2.2)$$

$$P_{att,quad}(d_{goal}) = \frac{\eta_{att}}{2} d_{goal}^2 \quad (2.3)$$

$$\frac{dP_{att,lin}(d_{goal})}{dd_{goal}} = \eta_{att} d_{goal} \quad (2.4)$$

Dabei weist der Faktor η_{att} die Einheit $\frac{1}{m}$ auf und daher ist das Potential P_{att} hier einheitenlos. Die resultierenden Funktionsgraphen sind in der Abbildung 2.1 dargestellt. Für die Darstellung wurde $\eta_{att} = 1$ angenommen. Der Wert d_{goal} beschreibt die minimale Distanz zum Zielpunkt. Der Gradient der linearen Funktion weist im gesamten Definitionsbereich einen konstanten Wert auf, welcher ein gleichmäßiges Geschwindigkeitsprofil verursacht. Jedoch würde eben dieser konstante Gradient das Erreichen des Ziels unmöglich machen, da der Gradient nur exakt im Zielpunkt verschwinden würde und folglich ein konstantes Überschwingen des Zielpunktes resultiert. Der Gradient der quadratischen Funktion hingegen weist einen Funktionswert auf, der sich mit abnehmender Distanz zum Zielpunkt linear reduziert. Daraus folgt eine zunehmende Geschwindigkeit in weiter Entfernung sowie eine abnehmende Geschwindigkeit in der Nähe des Zielpunktes. Allerdings führt die lineare Reduktion der Geschwindigkeit für sehr kleine Distanzen dazu, dass der Zielpunkt nicht erreicht werden kann.

Aus dem Vergleich der beiden Funktionen geht hervor, dass eine Kombination der beiden Charakteristika eine mögliche Lösung für die Beschreibung des anziehenden Potentials bildet.

2.1 Potentialfeldmethode

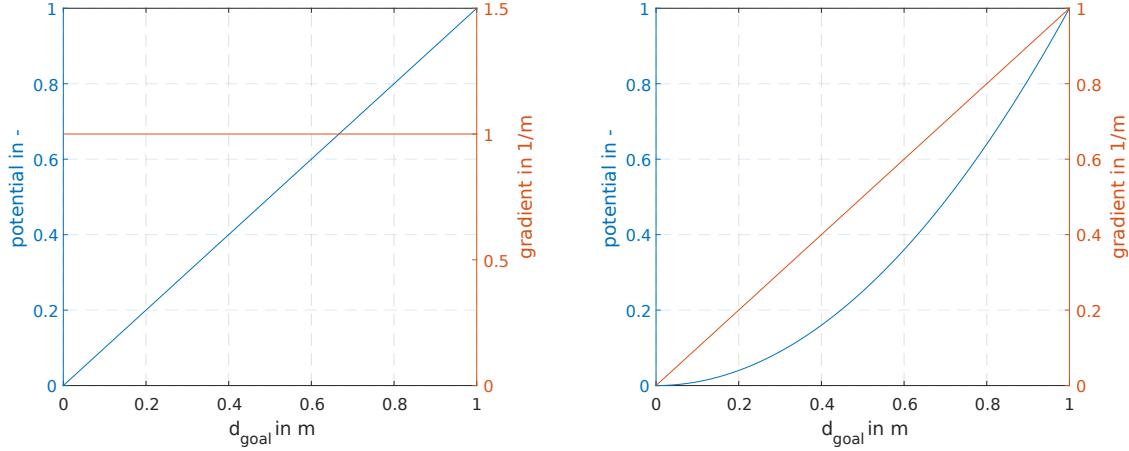


Abbildung 2.1: **links** lineare Potential-Funktion (blau) und Betrag des Gradienten (orange) gemäß der Gleichungen 2.1 und 2.2. **rechts** quadratische Potential-Funktion (blau) und Betrag des Gradienten (orange) gemäß der Gleichungen 2.3 und 2.4. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (gradient/Gradient), (potential/Potential).

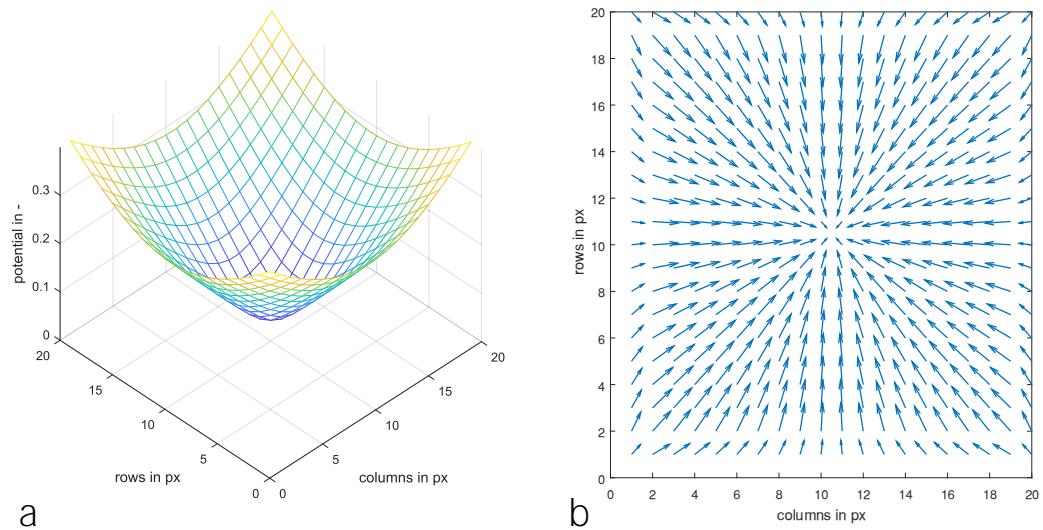


Abbildung 2.2: Pixel werden hierbei als Zellen eines diskreten Gitters angenommen
a resultierendes Potentialfeld eines einzelnen anziehenden Potentials im Zentrum eines 20x20 Gitters gemäß Gleichung 2.5. **b** negativer Gradient des Potentialfelds a

Die Abbildung 2.2 zeigt als Beispiel das resultierende Potentialfeld sowie dessen negative Gradienten für ein anziehendes Potential im Zentrum eines 20x20 Gitters. Dabei gilt die folgende Funktion für die Bestimmung des Potentialfelds. Die Distanz d_{goal} wird hierbei als einheitenlos angenommen.

$$P_{att}(d_{goal}) = \frac{1}{450} \cdot d_{goal} \quad (2.5)$$

2.1.2 Abstoßendes Potential

Ein abstoßendes Potential lässt sich durch eine Funktion der Distanz beschreiben, die im gesamten Definitionsbereich mit zunehmender Distanz stetig abnehmende Funktionswerte aufweist. Ein entsprechender Funktionscharakter lässt sich durch eine hyperbolische oder quadratische Funktion beschreiben. Die Gleichungen 2.6 und 2.8 bilden einen allgemeinen Ansatz dieser Funktionen. Die Gleichungen 2.7 und 2.9 entsprechen dem Betrag des Gradienten, abhängig von der minimalen Distanz zum nächstgelegenen Hindernis. Zur Vereinfachung wird nachfolgend die Bezeichnung „Gradient“ mit dem Betrag des Gradienten gleichgesetzt.

$$P_{rep,hyp}(d_{obst}) = \begin{cases} \frac{\eta_{rep}}{2} \left(\frac{1}{d_{obst}} - \frac{1}{d_0} \right)^2 & \text{falls } d_{obst} \leq d_0 \\ 0 & \text{sonst} \end{cases} \quad (2.6)$$

$$\frac{dP_{rep,hyp}(d_{obst})}{dd_{obst}} = \eta_{rep} \frac{d_{obst} - d_0}{d_0 d_{obst}^3} \quad (2.7)$$

$$P_{rep,quad}(d_{obst}) = \begin{cases} \frac{\eta_{rep}}{2} (d_{obst} - d_0)^2 & \text{falls } d_{obst} \leq d_0 \\ 0 & \text{sonst} \end{cases} \quad (2.8)$$

$$\frac{dP_{rep,quad}(d_{obst})}{dd_{obst}} = \eta_{rep} (d_{obst} - d_0) \quad (2.9)$$

Dabei weist der Faktor η_{rep} die Einheit $\frac{1}{m}$ auf und daher ist das Potential P_{rep} hier einheitenlos. Die resultierenden Funktionsgraphen sind in der Abbildung 2.3 dargestellt. Für die Darstellung wurde $\eta_{rep} = 1$ und $d_0 = 0.5 m$ angenommen. Der Wert d_{obst} beschreibt die minimale Distanz zum betrachteten Hindernis. Der Parameter d_0 definiert einen Grenzwert, ab dem die minimale Distanz zum Hindernis einen Funktionswert zur Folge hat. Die hyperbolische Funktion 2.7 weist für eine verschwindende Distanz eine Singularität auf. Solch eine Singularität würde ohne weitere Abfangmechanismen eine unkontrollierte Bewegung des Agenten hervorrufen. Jedoch bildet eine Singularität für

eine verschwindende Distanz zum Potential große Gradienten in dessen Umgebung aus. Aus diesen Gradienten folgt eine abrupte Entfernung vom betrachteten Hindernis. Eine Alternative bietet die quadratische Funktion 2.9, welche keine Singularität aufweist und dessen Gradient für eine geringe Distanz zum Hindernis über den Faktor η_{rep} skaliert werden kann.

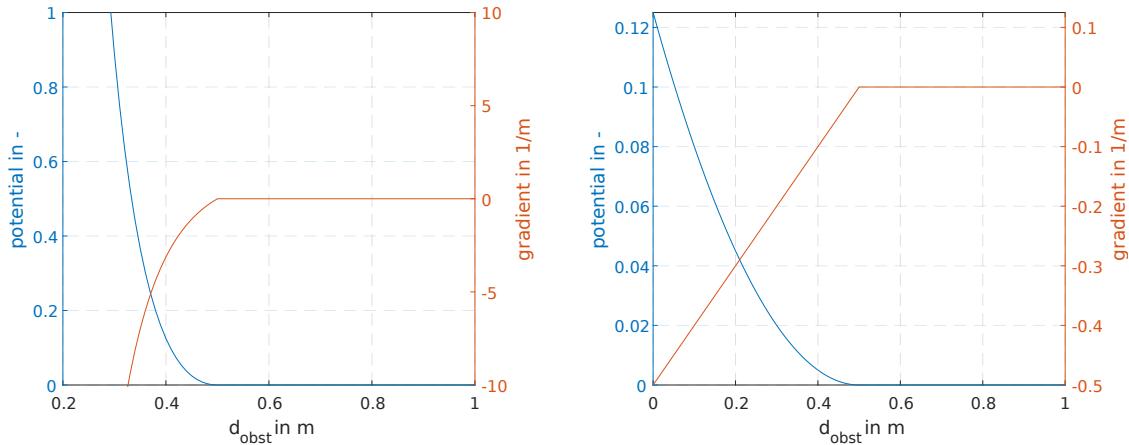


Abbildung 2.3: links hyperbolische Potential-Funktion (blau) und Betrag des Gradienten (orange) gemäß der Gleichungen 2.6 und 2.7, $\eta_{rep} = 1$ und $d_0 = 0.5m$ rechts quadratische Potential-Funktion (blau) und Betrag des Gradienten (orange) gemäß der Gleichungen 2.8 und 2.9, $\eta_{rep} = 1$ und $d_0 = 0.5m$. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (gradient/Gradient), (potential/Potential)

Die Abbildung 2.4 zeigt als Beispiel das resultierende Potentialfeld sowie dessen Gradienten für ein abstoßendes Potential im Zentrum eines 20x20 Gitters. Dabei gilt die folgende Funktion für die Bestimmung des Potentialfelds. Die Distanz d_{obst} wird hierbei als einheitenlos angenommen.

$$P_{rep}(d_{obst}) = \frac{1}{8 d_{obst}} \quad (2.10)$$

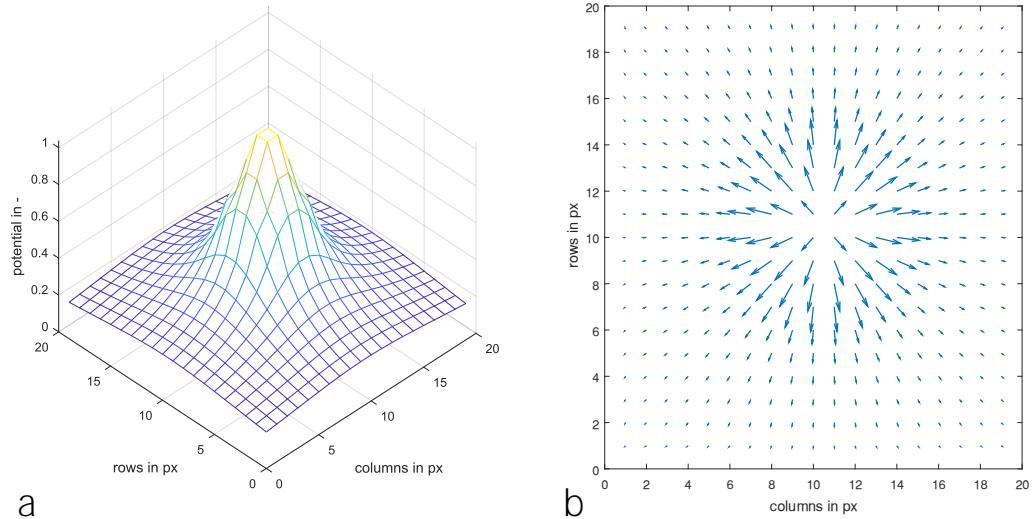


Abbildung 2.4: Pixel werden hierbei als Zellen eines diskreten, einheitenlosen Gitters angenommen **a** resultierendes Potentialfeld eines einzelnen abstoßenden Potentials im Zentrum eines 20×20 Gitters gemäß Gleichung 2.10. **b** negativer Gradient des Potentialfelds a

2.1.3 Alternative Interpretation

Die Beschreibungen in den Abschnitten 2.1.2 und 2.1.1 beziehen sich auf die Interpretation eines Potentials als Punktladung. Die Ladungen sind dabei als positiv/anziehend oder als negativ/abstoßend anzunehmen und der Betrag der Ladung wird durch die entsprechenden Potential-Funktionen beschrieben. Jedoch lässt die Potentialfeldmethode eine weitere, abweichende Interpretation von Potentialen zu. Dies lässt sich implizit durch die Betrachtung der Gradienten in den Abbildungen 2.2 und 2.4 zeigen. Die Potentialfeldmethode liefert im weiten Sinne keine Einschränkung, welche die Transformation der Gradienten betrifft. Somit liegt es nah, die Gradienten der jeweiligen Potentiale zu rotieren. Hierbei sei allein die Rotation um 90° in mathematisch positive Richtung betrachtet, wobei die Rotation beliebig sein kann. Die daraus resultierenden Wirbelfelder sind in der Abbildung 2.5 dargestellt. Führt man die entstandenen Wirbelfelder nun zurück auf eine Ursache, findet man eine abweichende physikalische Interpretation der ursprünglichen Ladungen. Die Ursache der Wirbelfelder kann als elektrischer Fluss oder Strom interpretiert werden. Demnach verhält sich das Wirbelfeld wie ein magnetisches Feld, dass sich um einen stromdurchflossenen Leiter

ausbreitet. Dabei können die ursprünglichen Ladungen, je nach Polarität, als Ströme in die unterschiedlichen Flussrichtungen betrachtet werden. Dabei bildet, nach der Korkenzieherregel, ein Strom in die Ebene das Analogon zur negativen Ladung und ein Strom aus der Ebene das Analogon zur positiven Ladung.

Somit bildet das Wirbelfeld, neben den anziehenden und abstoßenden Vektorfeldern, eine weitere Möglichkeit die Bewegung innerhalb eines künstlichen Potentialfeldes zu gestalten.

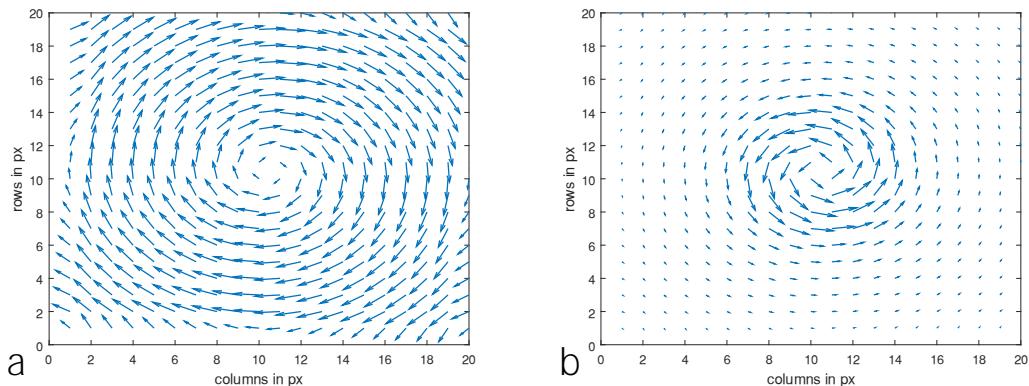


Abbildung 2.5: Pixel werden hierbei als Zellen eines diskreten, einheitenlosen Gitters angenommen **a** Wirbelfeld durch Rotation der Gradienten aus 2.2 um $+90^\circ$ **b** Wirbelfeld durch Rotation der Gradienten aus 2.4 um $+90^\circ$

2.2 Navigation nach der Potentialfeldmethode

Der Begriff Navigation beschreibt nach [Gal95] einen Prozess des Bestimmens und Beibehaltens eines Kurses oder einer Trajektorie zu einem Zielpunkt. Im Kontext der Robotik werden zwei grundlegende Ansätze zur Lösung dieser Problemstellung unterschieden. Ein Ansatz beschreibt die Lokalisation und die Pfadplanung anhand einer geometrischen Karte. Der zweite Ansatz hingegen beschreibt die direkte Bestimmung des Kurses zum Ziel anhand einer topologischen Karte [Möl17].

Die Potentialfeldmethode beschreibt ein Konzept für eine kollisionsfreie Navigation innerhalb einer geometrischen Karte. Damit lässt sich die potentialfeldbasierte Navigation zunächst dem ersten der beiden genannten Ansätze zur Navigation unterordnen. Wie bereits im Abschnitt 2.1 beschrieben, setzt sich ein Potentialfeld im Allgemeinen aus abstoßenden und anziehenden Potentialen zusammen. Ein Ziel bildet ein

2 Grundlagen

anziehendes Potential und Hindernisse¹ bilden abstoßende Potentiale. Dieses Konzept kann verwendet werden, um das Ziel oder Hindernisse in einer geometrischen Karte (diskret oder kontinuierlich) beschreiben zu können. Aus dieser Beschreibung lässt sich mit den in den Abschnitten 2.1.1 und 2.1.2 definierten Funktionen ein künstliches Potentialfeld über die zugrundeliegende geometrische Karte legen. Die Abbildung 2.6 zeigt beispielhaft eine diskrete geometrische Karte der Größe 50x50 Zellen sowie das Potentialfeld, das aufgrund der Karte bestimmt wurde. Dabei gelten die folgenden Funktionen für die Bestimmung des Potentialfeldes.

$$P_{att}(d_{goal}) = \frac{d_{goal}}{450} \quad (2.11)$$

$$P_{rep}(d_{obst}) = \frac{1}{8 d_{obst}} \quad (2.12)$$

Dabei weisen die Distanzen d_{goal} und d_{obst} keine Einheit auf und daher ist das Potential P_{att} sowie P_{rep} hier einheitenlos.

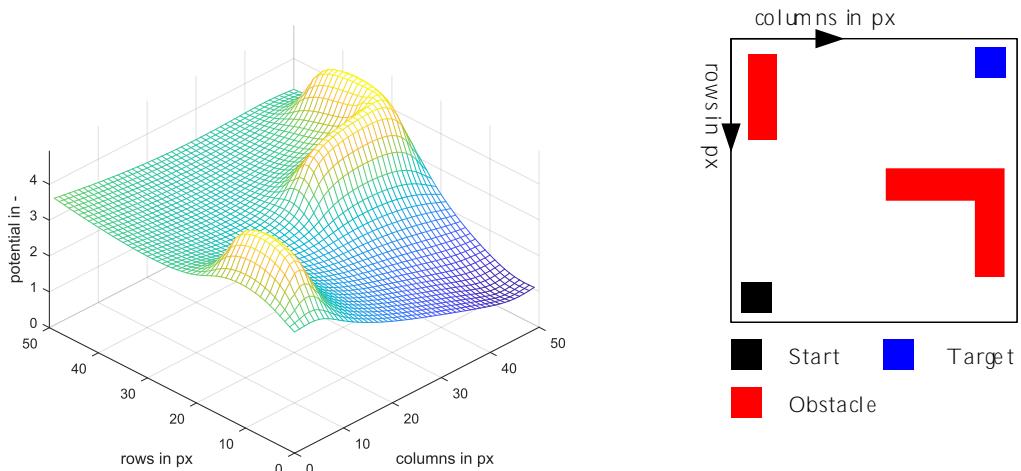


Abbildung 2.6: Pixel werden hierbei als Zellen eines diskreten, einheitenlosen Gitters angenommen **links:** Potentialfeld der diskreten Karte (rechts) gemäß der Gleichungen 2.11 und 2.12 **rechts:** Beispielhafte diskrete geometrische Karte der Größe 50x50 Zellen. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (*Target/Ziel*), (*Obstacle/Hinderniss*)

¹Allgemein: nicht passierbare Bereiche

Die abstoßenden und anziehenden Potentiale werden für jede Zelle in Abhängigkeit ihrer Distanz zu einer blauen oder roten Zelle bestimmt und anschließend summiert. Hierbei ist zu beachten, dass der Zielpunkt aufgrund der gewählten Diskretisierung keine einzelne Zelle darstellt, sondern eine Gruppe von Zellen.

Das resultierende Potentialfeld ermöglicht somit eine Navigation vom Start- (Abbildung 2.6: *Start*) zum Zielpunkt (Abbildung 2.6: *Target*). Die Navigation entspricht hierbei einem Gradientenabstieg, weshalb die Ermittlung einer Trajektorie zum Zielpunkt ein Optimierungsproblem darstellt. Eine Lösung dieses Optimierungsproblems ist in Abbildung 2.7 dargestellt.

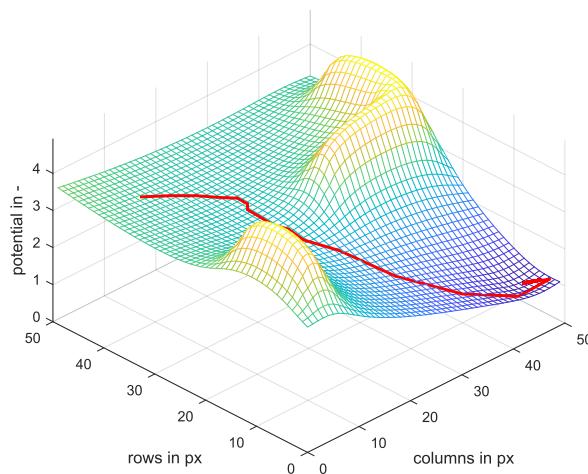


Abbildung 2.7: Pixel werden hierbei als Zellen eines diskreten, einheitenlosen Gitters angenommen. Abstieg entlang des Gradienten des zugrundeliegenden Potentialfelds vom Start- zum Zielpunkt, siehe auch Abbildung 2.6 **rechts**

Die gewählte geometrische Karte in Abbildung 2.7 bildet ein teilweise idealisiertes Szenario. Eine solche Karte führt unter den oben angegebenen Funktionen zu einem Potentialfeld mit einem globalen Minimum. Eine Lösung dieser Navigationsaufgabe ist dementsprechend trivial. Die geometrische Karte in Abbildung 2.8 und das resultierende Potentialfeld² zeigen ein leicht verändertes Szenario. Das Potentialfeld weist in diesem Fall verschiedene lokale Minima auf. Die Lösung dieses Problems ist weitaus weniger trivial als die des idealisierten Problems.

²Die Bestimmung erfolgt ebenfalls anhand der Gleichungen 2.5 und 2.10

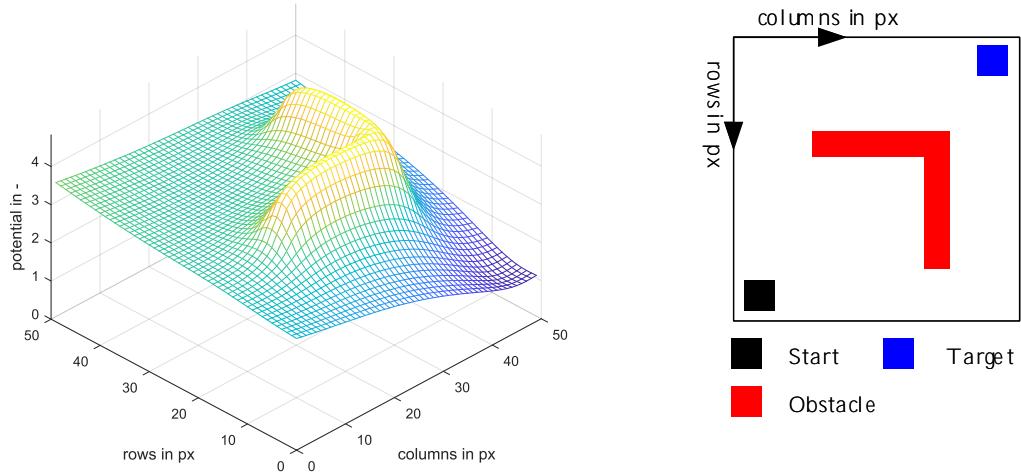


Abbildung 2.8: Pixel werden hierbei als Zellen eines diskreten, einheitenlosen Gitters angenommen **links**: Potentialfeld der diskreten Karte (rechts) gemäß der Gleichungen 2.11 und 2.12 **rechts**: Beispielhafte diskrete geometrische Karte der Größe 50x50 Zellen. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (*Target/Ziel*), (*Obstacle/Hinderniss*)

Zusammenfassend bildet die potentialfeldbasierte Navigation eine geeignete Möglichkeit zur lokalen Pfadplanung. Dies lässt sich dadurch begründen, dass eine globale Pfadplanung nur uneingeschränkt möglich ist, wenn die gesamte geometrische Karte nur ein globales Minimum aufweisen würde. Das Beispiel in Abbildung 2.8 hat gezeigt, dass geringe Veränderungen auch in relativ kleinen Karten zu lokalen Minima führen. Daher ist ein ideales Potentialfeld für eine globale Pfadplanung selten zu erwarten. Dennoch weist die potentialfeldbasierte Navigation einen Vorteil gegenüber anderen Methoden der Navigation auf. Das Potentialfeld bildet, bei geeigneter Wahl der Potential-Funktionen, in jedem Fall einen stetigen Verlauf. Demnach verlaufen die Gradienten ebenfalls stetig und können über einen entsprechenden Faktor direkt in Fahrbefehle überführt werden [Möl17].

3 Methodik

3.1 Rahmenbedingungen

3.1.1 Software-Framework ROS

Moderne Robotersysteme basieren auf Softwarelösungen, die häufig nicht mehr in einem gemeinsamen System vereint sind. Vielmehr wird das Gesamtsystems in einzelne Module aufgeteilt, um dessen Komplexität so beherrschbar wie möglich zu gestalten [M.O14]. Diese Modularisierung ist eine Herausforderung der Softwareentwicklung und bietet Chancen wie einen hohen Wiederverwendungsgrad und eine erleichterte Wartbarkeit von Software. Des Weiteren besteht während der Entwicklung von Softwarelösungen ein hoher Anspruch an eine schnelle und kostengünstige Testbarkeit einzelner Module. Dies motiviert vor allem in frühen Entwicklungsphasen hoch frequentierte Testzyklen und eine Früherkennung sowie Abschaltung von Fehlerquellen. Jedoch gestaltet sich die Testbarkeit auf realen Systemen als kosten- und zeitintensiv. In diesem Zusammenhang bieten realitätsnahe Simulationsmodelle eine Möglichkeit, in frühen Entwicklungsphasen weitestgehend unabhängig von realen Prototypen zu arbeiten. Um diesen Ansprüchen gerecht zu werden liefert das quellenoffene Software-Framework *Robot Operating System (ROS)* verschiedene Werkzeuge und Bibliotheken für die Entwicklung von Roboterkomponenten. *ROS* bietet damit einen vollständigen Rahmen für eine kreative aber zielorientierte Softwareentwicklung. Das *Framework* schlägt zudem eine modulare Struktur von Systemen vor, die eine Wiederverwendbarkeit von vorhandenen Lösungen begünstigt. Darüber hinaus schafft *ROS* durch das Konzept einer Hardwareabstraktion die Möglichkeit das Zielsystem durch ein Simulationsmodell zu ersetzen. Weiterhin bietet das *Framework* verschiedene Gerätetreiber, eine Paketverwaltung sowie ein transparentes Konzept der Kommunikation [M.O14]. Die Entwicklung innerhalb der *ROS*-Frameworks erfolgt anhand von Knoten, welche jeweils einen eigenständigen Prozess definieren. Das Zusammenwirken aller Knoten bildet in der Regel die Steuerung des Roboters. Die Knoten kommunizieren direkt untereinander und erhalten alle notwendigen Verbindungsinformationen von einem *Master*-Knoten. Die Kommunikation unterliegt dabei dem *TCPROS* Standardkommunikationsproto-

koll [ROS]. *TCPROS* bildet eine Transportschicht für die interne Kommunikation des Frameworks.

3.1.2 *AMiRo*

Der *AMiRo* (*Autonomous Mini Robot*) bildet das Ergebnis einer interdisziplinären Entwicklung innerhalb des Exzellenzclusters Kognitive Interaktionstechnologie (*CITEC*) [Her17]. Die Konfiguration des Miniroboters ist modular gestaltet, sodass sich die Basiskonfiguration um weitere Komponenten erweitern lässt und daher flexibel an verschiedene Anwendungsbereiche angepasst werden kann [KS218]. So könnte der *AMiRo* beispielsweise um einen Greifer erweitert werden, um *Pick-and-Place* Aufgaben bewältigen zu können. In seiner Basiskonfiguration ist der *AMiRo* unter anderem mit 12 Näherungssensoren (acht umlaufende und vier nach unten gerichtete), acht RGB-Leuchtdioden, Magnetometern, einer *IMU* sowie einer Kamera ausgestattet. Der Roboter wird mit Hilfe von zwei Flachgetriebemotoren angetrieben und erreicht eine maximale Geschwindigkeit von 800 mm/s [Her17].

Die grundlegenden physikalischen und funktionalen Eigenschaften des *AMiRo* stehen im *URDF* (*Unified Robot Description Format*) zur Verfügung. Insofern kann der *AMiRo* als vollständiges Simulationsmodell in Umgebungen wie *Gazebo* eingebunden werden, um unabhängig von der Hardware Algorithmen zu implementieren und zu testen.

3.1.3 Telewerkbank

Die Telewerkbank bildet eine einheitliche Experimentierplattform für Multi-Roboter-Experimente. Die Plattform umfasst eine ca. 36 m² große quadratische Fläche, die in vier gleichgroße Sektoren aufgeteilt ist. Jeder der Sektoren wird durch ein eigenständiges Kamerasystem abgedeckt, dass zur optischen Erfassung von Positionsinformationen eingesetzt wird. Ein lokales Serversystem dient der Protokollierung relevanter Informationen, wie Positions- oder Kommunikationsdaten. Aufgenommene Daten können im Rahmen von Experimenten in die zu testende Software rückgekoppelt werden. Dafür stellt die Plattform eine Kommunikationsschnittstelle bereit, die eine Kopplung zwischen dem Software-Framework *ROS* und dem Zielsystem ermöglicht. Folglich kann die Softwareentwicklung leicht durch einen experimentellen Rahmen erweitert werden und reduziert daher den Aufwand von Softwaretests unter realen Bedingungen [TGM⁺13].

4 Definition der Steuerungsarchitektur

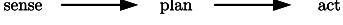
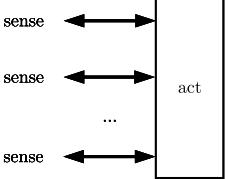
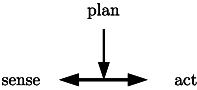
Eine Steuerungsarchitektur beschreibt den Zusammenhang aller Systemkomponenten, die maßgeblich an der Steuerung eines Roboters beteiligt sind. Zudem definiert die Architektur einen Rahmen, in dem die Softwarelösung eingebettet werden kann. Grundlegend wird zwischen einigen wenigen Klassen von Steuerungsarchitekturen unterschieden. Zu diesen zählen die deliberativen, die reaktiven, die verhaltensbasierten und die hybriden Architekturen. Eine kurze Beschreibung und Vor- und Nachteile der einzelnen Architektur-Klassen sind in der Tabelle 4.1 aufgeführt.

Im Rahmen dieser Arbeit wird die Steuerungsarchitektur *AuRA* (*Autonomous Robot Architecture*) für die Einbettung und Untersuchung der potentialfeldbasierten Navigation verwendet. *AuRA* bildet eine Kombination aus einer hierarchischen und einer reaktiven Komponente und wird daher den hybriden Architekturen zugeordnet (siehe auch Tabelle 4.1: hybrid). Prägnant für diese Steuerungsarchitektur ist die reaktive Komponente, die durch Emergenz einzelner Subkomponenten einen Bewegungsvektor analog zur Potentialfeld-Methode bildet. Somit liefert *AuRA* ein Grundkonzept für eine potentialbasierte Navigation. Darüber hinaus gilt die Steuerungsarchitektur als sehr modular und verallgemeinerbar. Die Modularität folgt maßgeblich aus der reaktiven Komponente, dessen Subkomponenten mit geringem Aufwand austauschbar sind. Dieser Aspekt macht die Steuerungsarchitektur gerade für die Forschung und die Lehre interessant. Die Modularität der *AuRA* bedingt ebenfalls die Verallgemeinerbarkeit der Architektur. Zu den Anwendungsbereichen gehören unter anderem die dreidimensionale Navigation, Navigation im Innen- sowie Außenbereich und militärische Szenarien [Ark98].

Im Folgenden wird das theoretische Konzept der reaktiven Komponente, die Theorie der Schemata, erarbeitet. Anschließend wird der genaue Aufbau, die Funktion und das Zusammenspiel der einzelnen Komponenten der *AuRA* beschrieben.

4 Definition der Steuerungsarchitektur

Tabelle 4.1: Gegenüberstellung gängiger Steuerungsarchitekturen für Roboter, vergleiche [Oub09]

Architektur	Beschreibung	Vor-/Nachteile
deliberativ	 <pre> sense → plan → act </pre>	<ul style="list-style-type: none"> + optimale Lösung durch Weltmodell - Weltmodell benötigt - hohe Reaktionszeit
reakтив	 <pre> sense ↔ act </pre>	<ul style="list-style-type: none"> + geringe Reaktionszeit + kein Weltmodell benötigt - geringe Flexibilität - keine optimale Lösung ohne Weltmodell
verhaltensbasiert	 <pre> sense ↔ act sense ↔ act ... sense ↔ act </pre>	<ul style="list-style-type: none"> + hohe Skalierbarkeit + geringe Reaktionszeit + kein Weltmodell benötigt - keine optimale Lösung ohne Weltmodell
hybrid	 <pre> sense → plan sense ↔ act </pre>	vereint Vor- und Nachteile der kombinierten Architekturen

4.1 Autonomous Robot Architecture AuRA

4.1.1 Theorie der Schemata

Eine eindeutige Handlungsvorschrift, auch als Algorithmus bezeichnet, beschreibt eine isolierte Komponente für die Lösung eines spezifischen Problems. Ferner definiert eine Problemstellung die Diskrepanz zwischen einer Ausgangssituation und einem Ziel. Im Kontext der Robotik kann dieses Ziel beispielsweise das Erreichen einer Position oder die Vermeidung von Hindernissen definieren. Die Gemeinsamkeit aller Ziele besteht allein darin, dass diese ein Ergebnis von Verhaltensweisen darstellen. Letztendlich beschreibt eine eindeutige Handlungsvorschrift ein Verhalten, dass je nach Definition

als einfach oder komplex eingestuft werden kann. Im Rahmen dieser Arbeit wird gemäß [Ark98] und [RR02] ein Verhalten als komplex bezeichnet, sofern es durch mehrerer simplere Verhalten darstellbar ist. Diese These wird durch Erkenntnisse von [AL95] gestützt, die bereits 1987 ein Modell vorschlugen, dass das Jagdverhalten von Fröschen beschreibt. Das Modell kombiniert die Bewegungsresultate simpler Verhalten, wie Hindernisvermeidung und Beuteannäherung zu einem Bewegungsprofil. Entsprechende Bewegungsprofile konnten daraufhin in einer experimentellen Umgebung vorhergesagt werden.

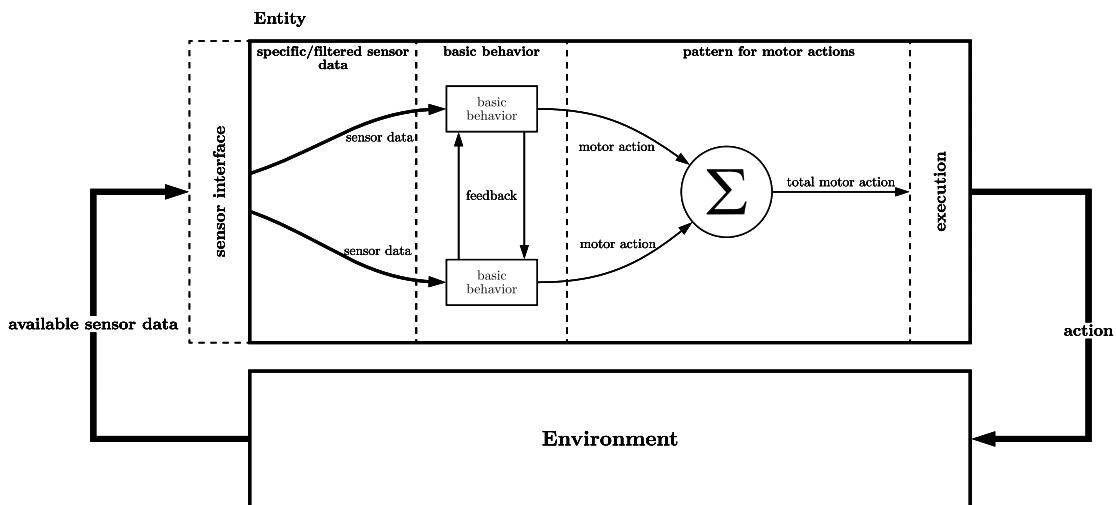


Abbildung 4.1: schematische Prozessdarstellung der Überführung extrahierter (Sensor-)Informationen, innerhalb eines Wesens, in eine Reaktion bzw. ein Bewegungsmuster sowie dessen Einbettung in einen Aktions-Reaktions-Zyklus. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (*basic behavior*/einfache Verhaltensweise), (*specific/filtered sensor data*/spezifische Sensorinformationen), (*total motor action*/Gesamtbewegungsresultat), (*feedback*/Rückinformation)

Diese Erkenntnis zeigt unter anderem, dass simple Verhaltensweisen spezifische Informationen aus der Umwelt auf ein Bewegungsresultat projizieren und in Kombination ein Gesamtbewegungsresultat bilden. Dabei ist zu beachten, dass einzelne Verhaltensweisen nicht isoliert voneinander agieren, sondern untereinander Informationen austauschen können [AD92]. Eine solche Rückinformation kann beispielsweise eine hemmende oder verstärkende Wirkung auf eine oder mehrere bestimmte Verhaltensweisen hervorrufen. Die so resultierende Gewichtsverlagerung in den Verhaltensweisen ermöglicht es einem Wesen bestimmte Situationen adäquat zu bewältigen. Beispielswei-

se können so essentielle Verhaltensweisen gestärkt und redundante gehemmt werden, um Stresssituationen (unter anderem Flucht, Jagd und Nährstoffmangel) bewältigen zu können. Diese Zusammenhänge lassen sich anhand der Abbildung 4.1 verdeutlichen. Folglich beschreibt eine einfache Verhaltensweise ein Muster, wie spezifische Informationen aus der Umwelt ein Bewegungsresultat hervorrufen. Ein solches Muster wird im Folgenden als Schema bezeichnet¹. [Ark98] definiert daher ein einfaches Verhalten als Kombination aus einem *motor schema* und endlich vielen *perceptual schemas*, Abbildung 4.2 **a** und **b**. Die *perceptual schemas* dienen dabei der Extraktion spezifischer Informationen aus der Umwelt und das *motor schema* überführt diese Informationen in das Bewegungsresultat. Abbildung 4.2 **c** beschreibt dieses Prinzip anhand des konkreten Beispiels einer Hindernissvermeidung. Die spezifischen Sensorinformationen werden in Form von Distanzdaten eines Lasersensors vom *perceptual schema get distance* aufgenommen, aufbereitet und dem *motor schema avoid obstacle* übergeben. Es ist zu beachten, dass nach [Ark98] das *motor schema* die Bezeichnung der Verhaltensweise erhält. *avoid obstacle* projiziert dann die aufbereiteten Sensorinformationen auf das Bewegungsresultat und bedingt, anhand interner Regeln, eine Ausweichbewegung im Fall einer bevorstehenden Kollision.

Demnach kann festgehalten werden, dass komplexe Verhaltensweisen die Kombination grundlegender Verhaltensweisen bilden. Es liegt zudem nahe, dass grundlegende Verhaltensweisen einen hohen Wiederverwendungsgrad aufweisen. Beispielsweise kann eine Hindernisvermeidung eine Komponente vieler komplexer Verhalten, wie z.B. der Navigation oder auch der Flucht, ausmachen.

Gleichermaßen gelten die *perceptual schemas* als hochgradig wiederverwendbar. Spezifische Informationen, die aus der Umwelt extrahiert werden, können die Grundlage mehrerer Verhaltensweisen bilden. Genauer gesagt können Informationen wie z.B. die aktuelle Position eine Quelle diverser Verhaltensweisen darstellen.

Schließlich lässt sich das Konzept der Schemata auf die Robotik übertragen. In diesem Zusammenhang können grundlegende Verhalten bzw. Handlungsvorschriften als wieder verwendbare Module abgegrenzt werden. Ebenfalls können *perceptual schemas* als Submodule in mehrere Module integriert werden, um Sensorinformationen aufzubereiten und den *motor schemas* bereitzustellen. Die Kombination der einzelnen *motor actions* kann ebenfalls durch ein eigenständiges Modul, also eine Kombination aus *motor* und *perceptual schemas*, beschrieben werden. Dieses Verhalten, nachfolgend als *move-robot* bezeichnet, gewährleistet im einfachsten Fall eine Summation einzelner Bewegungsresultate zu einem Gesamtresultat und dessen Übersetzung in einen Befehl an die Aktorik.

¹Definition gemäß [Nei76]

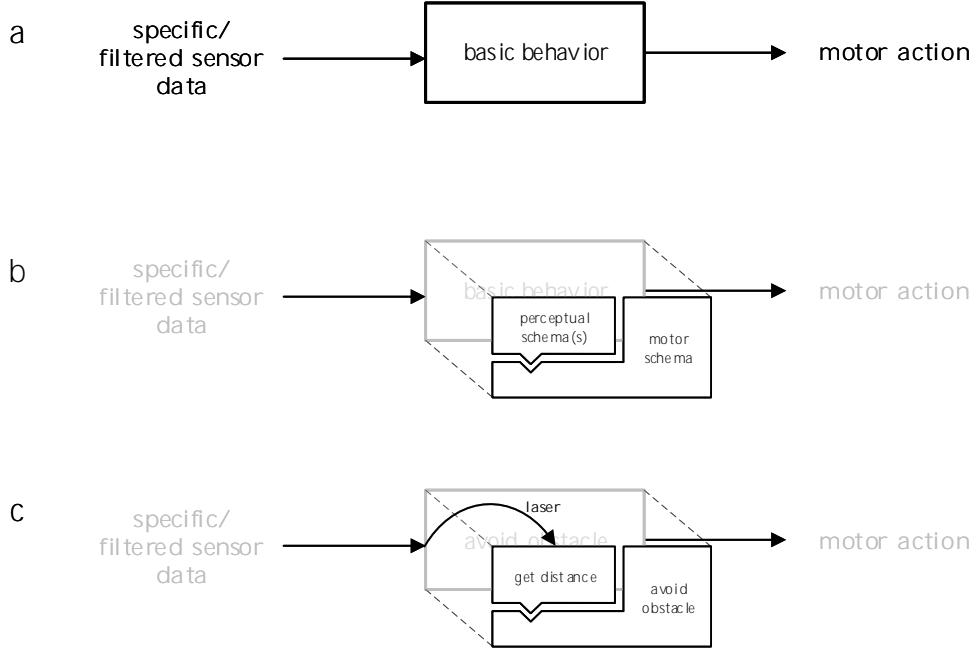


Abbildung 4.2: **a** Darstellung einer einfachen Verhaltensweise als System mit spezifischen Informationen bzw. Sensorwerten als Eingabe und ein Bewegungsresultat als Ausgabe **b** interne Struktur einer einfachen Verhaltensweise nach [Ark98] **c** schematische Darstellung eines konkreten Beispiels anhand der Verhaltensweise „Hindernissvermeidung“. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (*basic behavior*/einfache Verhaltensweise), (*specific/filtered sensor data*/spezifische Sensorinformationen), (*motor action*/Bewegungsresultat)

Die Instrumentalisierung der Schema-Theorie im Kontext der Robotik strebt ein emergentes Verhalten des Roboters an. Ein emergentes Verhalten beschreibt das zielführende Zusammenwirken von Komponenten (hier: simple Verhaltensweisen), so dass die Fähigkeit des Systems über die Fähigkeit der Komponenten hinausgeht, ähnlich der komplexen Verhaltensweisen. Das entsprechende Zusammenwirken soll sich dabei autonom, allein durch die Interaktion der Komponenten, organisieren [Qua00]. Dieses Konzept bildet die Grundlage für die reaktive Komponente der AuRA. Verhaltensweisen sollen einzeln spezifische Sensorinformationen auf ein Bewegungsresultat projizieren und in ihrem Zusammenwirken ein Bewegungsmuster bilden, dass die Lösung einzelner Probleme ermöglicht. Solche Probleme können beispielsweise eine kollisionsfreie Navigation für mobile Roboter oder Manipulatoren beschreiben.

Demzufolge bildet der Ansatz der Schemata eine Grundlage für eine Steuerungsarchitektur, die einerseits eine Wiederverwendung von Software motiviert und andererseits eine Modularisierung von Verhaltensweisen vorgibt.

4.2 Aufbau und Komponenten der AuRA

Ronald C. Arkin hat die Theorie der Schemata [AD92] als Grundlage genutzt, um zusammen mit Tucker Balch in [Ron97] ein Konzept für die hybride Steuerungsarchitektur zu definieren. Die so entstandene Architektur *AuRA* beschreibt eine Kombination aus einer deliberativen und reaktiven Komponente. Die Architektur ist hierarchisch aufgebaut, sodass auf höchster Ebene eine rechenzeitintensive Aufgabenplanung und auf niedrigster Ebene eine reaktive Ausführung von kollaborativen Verhalten erfolgen kann.

Nachfolgend werden die einzelnen Komponenten des Architekturkonzeptes anhand der Darstellung in Abbildung 4.3 beschrieben und diskutiert.

4.2.1 Missionplanner

Der *Mission Planner* bildet die höchste hierarchische Ebene der *AuRA* und dient als Schnittstelle zwischen dem Client (Anwender oder vorgesetzte Software) und der Architektur. Auf dieser Ebene werden konkrete Zielvorgaben (Abbildung 4.3: *cmd*) vom Client, wie zum Beispiel die Endposition einer Navigationsaufgabe, entgegengenommen und weiterverarbeitet. Zudem ermöglicht die Schnittstelle eine Rückmeldung (Abbildung 4.3: *resp*) an den Client, um beispielsweise Informationen bezüglich des Bearbeitungsstatus zurückzugeben. Ebenfalls bildet die Rückmeldung an den Client das letzte Element der Fehlerbehandlung (Abbildung 4.3: *Exception*).

4.2.2 Path Sequencer

Gemäß der Hierarchie folgt auf den *Missionplanner* der *Path Sequencer*. Diese Komponente übersetzt das Gesamtziel in eine Abfolge von Teilzielen. Teilziele können im Kontext einer Navigationsaufgabe beispielsweise Knotenpunkte einer topologischen Karte definieren, die angefahren werden müssen, um das Gesamtziel zu erreichen.

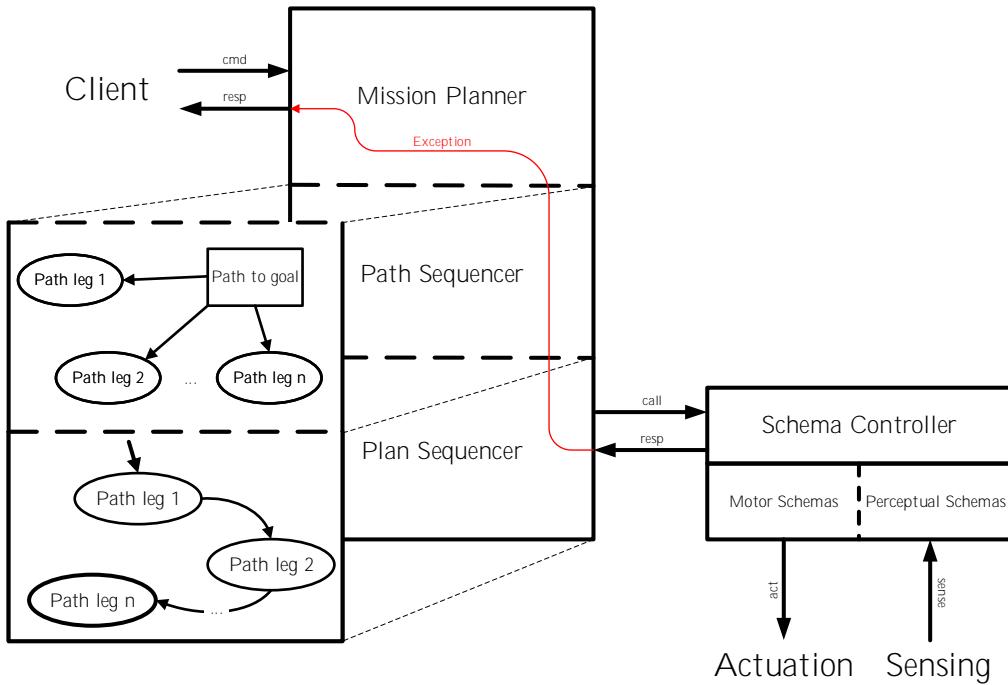


Abbildung 4.3: Struktur der hierarchischen Steuerungsarchitektur AuRA nach Ronald C. Arkin und Tucker Balch, vergleiche [Ron97]. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (*Exception*/Fehlerbehandlung), (*Path leg*/Teilziel), (*Actuation*/Aktorik), (*Sensing*/Sensorik)

4.2.3 Plan Sequencer

Der *Plan Sequencer* nimmt die Teilziele des *Path Sequencer* entgegen und bildet diese in einer Zustandsmaschine ab. Die einzelnen Teilziele definieren die Transitionsbedingungen zwischen einzelnen Zuständen. Ein Zustand umfasst die Instanziierung von Verhaltensweisen, die für das Erreichen des Teilziels erforderlich sind. Im Umkehrschluss sorgt ein Zustand ebenfalls für die Deinstanziierung nicht benötigter Verhaltensweisen. Die Definition einzelner Gruppen von Verhaltensweisen, die zur Erfüllung von Teilzielen notwendig sind, erfordert ein genaues Verständnis der Problematik. Somit können solche Gruppen nur unter hohem Aufwand automatisiert erstellt werden und müssen in den meisten Fällen vordefiniert werden. Diese Eigenschaft entzieht dem *Schema-Controller*, oder ferner der reaktiven Komponente ein Maß an Autonomie und reduziert dessen Handlungsspielraum. Ebenfalls widerspricht diese Herangehensweise der Theorie der Schemata insofern, dass die Theorie kein Konzept für die Existenz oder

Aktivität einzelner Verhaltensweisen definiert. Es wird vielmehr impliziert, dass alle Verhaltensweisen aktiv sind aber, abhängig von den aktuellen Sensorinformationen, mehr oder weniger gewichtet werden. Jedoch wäre die Umsetzung unter technischen Aspekten wesentlich komplexer. Es müsste ein Regelwerk erstellt werden, dass abhängig von bestimmten Situationen die gegenseitige Hemmung oder Verstärkung einzelner Verhaltensweisen definiert. Ein solches Regelwerk kann auch für weniger komplexe Gesamtverhalten schnell unverhältnismäßig aufwendig in der Erstellung sein. Solch eine Herangehensweise würde im Kontext des maschinellen Lernens (adaptive Reaktion) wesentlich handhabbarer sein aber unter den hier zugrunde gelegten Aspekten² wenig sinnvoll.

Demnach wird für jeden einzelnen Zustand eine Gruppe an Verhaltensweisen definiert und während des Eintritts in den entsprechenden Zustand ausschließlich diese Gruppe aktiv gehalten.

4.2.4 Schema Controller

Der *Schema Controller* bildet das untere Ende der Hierarchie und kann als Ausführungs-ebene bezeichnet werden, die in direkter Verbindung mit der Aktorik und der Sensorik des Roboters steht (Abbildung 4.3: *act* und *sense*). Die Komponente nimmt Anfragen des *Plan Sequencer* entgegen und de- oder instanziert spezifische Verhaltensweisen, um das aktuelle Teilziel zu erfüllen. Wie zuvor beschrieben, umfasst jede Verhaltensweise ein *motor schema* und endlich viele *perceptual schemas*. Die einzelnen Verhaltensweisen reagieren eigenständig auf Sensorinformationen und bilden in Summe ein emergentes, zielführendes Verhalten des Roboters aus. Dafür bildet jede Verhaltensweise die eingehenden Sensorinformationen auf einen Vektor (4.1.1: Bewegungsresultat) analog zur Potentialfeld-Methode ab. Die einzelnen Vektoren werden von einer übergeordneten Verhaltensweise (*move-robot*) zu einem resultierenden Gesamtvektor (4.1.1: Gesamtbewegungsresultat) verrechnet. Dies impliziert, dass eine Kommunikation der Verhaltensweisen zulässig ist, was ebenfalls durch die Schema-Theorie definiert wird. Gemäß der Definition einer Verhaltensweise (Abschnitt 4.1.1) muss, wie auch die Sensorinformationen, die Kommunikation über ein *perceptual schema* erfolgen. Dieses Konzept ermöglicht neben dem Zusammenfassen aller Bewegungsresultate ebenfalls eine Wechselwirkung der Verhaltensweisen untereinander. Somit können einzelne Verhaltensweisen beispielsweise, in bestimmten Situationen, gegenseitig ihre Bewegungsresultate hemmen oder verstärken. Das ist vor allem dann sinnvoll, wenn die Lösung eines akuten Problems ein Ungleichgewicht der Verhaltensweisen erfordert. Es

²Determinismus auf der Planungsebene sowie schnelle Austauschbarkeit einzelner Verhaltensweisen

ist zu beachten, dass die Kommunikation der Verhaltensweisen untereinander asynchron erfolgt.

4.2.5 Fehlerbehandlung (*Exception*)

Die hierarchische Struktur der Steuerungsarchitektur ermöglicht eine Fehlerbehandlung nach dem *Bottom-Up* Prinzip (Abbildung 4.3: *Exception*). Somit können entsprechende Verhaltensweisen auf unvorhersehbare Zustände, wie zum Beispiel Hindernisse in einer dynamischen Umwelt, adäquat reagieren. Im Falle eines nicht lösbarer Problems auf der Ebene der Verhaltensweisen, zum Beispiel ein Stillstand des Roboters durch die gegenseitige Auslöschung der Verhaltensweisen, reagiert der *Schema Controller*. Eine Reaktion wäre zum Beispiel die Instanziierung weiterer Verhaltensweisen, um das aktuelle Problem zu lösen. Führt diese Verhaltensänderung nicht zur Lösung, reagieren die höheren Ebenen mit einer Umplanung der Lösung. Kann dennoch keine Lösung gefunden werden, meldet der *Missionplanner* ein Problem an den Client und beendet die Ausführung.

5 Übertragung der Steuerungsarchitektur in das Software-Framework *ROS*

Dieser Abschnitt beschreibt das Ergebnis einer Machbarkeitsanalyse, in wie fern das Konzept der festgelegten Steuerungsarchitektur in das Software-Framework *ROS* übertragbar ist. Dabei wird der Fokus besonders auf jene Komponenten gelegt, die der Ausführungsebene am nächsten gelegen sind. Dies schließt den *Plan Sequencer* sowie den *Schema Controller* ein. Beide Komponenten weisen, im Gegensatz zu den höheren Ebenen, eine geringere Abstraktion auf und verfügen daher über technischere Schnittstellen. Ebenfalls ist zu erwähnen, dass die Ausführungsebene den sinnvollsten Ausgangspunkt für die Umsetzung der Steuerungsarchitektur bildet. Diese Entwicklung nach dem *Bottom-Up*-Prinzip ist weniger anfällig für zeitintensive Korrekturschleifen in fortgeschrittenen Entwicklungsstufen und begünstigt frühe Modultests auf dem Zielsystem. Ebenfalls sei erwähnt, dass der angestrebte Entwurf eine objektorientierte Struktur zum Ziel hat. Das liegt zum einen darin begründet, dass *schemas* einen hohen Grad an Wiederverwendbarkeit aufweisen und daher gut mit der Objektorientierung vereinbar sind. Zum anderen ermöglicht die objektorientierte Kapselung von Komponenten Modultests und trägt damit zur Reduktion einer Fehlerwirkung bei.

5.1 *Schema Controller*

Aus der Beschreibung des *Schema Controller* in Abschnitt 4.2 und der Zieldefinition lassen sich konkrete Anforderungen für die Komponente ableiten. Zum einen muss jede Verhaltensweise ein Interface bereitstellen, das die In- und Deinstanziierung sowie die Kommunikation mit und unterhalb der Verhaltensweisen ermöglicht. Darüber hinaus muss eine Verhaltensweise im Sinne der Wiederverwendbarkeit und der Testbarkeit als eigenständiges Modul auftreten. Demnach wird der *Schema Controller* weniger als eine Komponente umgesetzt, sondern umfasst alle Verhaltensweisen und definiert ein

einheitliches Interface für jede dieser Module.

Der Einfachheit halber, wird im Folgenden eine Verhaltensweise als *motor schema* definiert und jegliche *perceptual schemas* bilden Submodule der *motor schemas*. Dies ist der Definition in [Ark98] geschuldet, die eine Verhaltensweise mit einem *motor schema* gleichsetzt.

5.1.1 Entwurf

Das Software-Framework *ROS* stellt ein Knoten-Konzept zur Verfügung, welches eine Modularisierung von Software ermöglicht. Ein Knoten kann dabei mit einem eigenständigen Prozess gleichgesetzt werden, der in Kombination mit anderen Knoten einen Gesamtprozess definiert. Knoten kommunizieren untereinander über das *ROS*-eigene Kommunikationsprotokoll auf sogenannten *Topics*. Die Kommunikation erfolgt dabei nach dem *Subscriber-Publisher*-Prinzip und gilt daher als asynchron und die Koppelung untereinander als lose. *ROS* stellt keine Methode bereit, die es ermöglicht einzelne Knoten zur Laufzeit zu starten. Deshalb werden alle Knoten gemeinsam mit dem Gesamtsystem gestartet. Dies legt fest, dass zum einen alle notwendigen *motor schemas* zur Problemlösung im Vorfeld definiert werden und zum anderem die *motor schemas* und *perceptual schemas* innerhalb der jeweiligen Knoten implementiert sind. Die Ausführung der *motor schemas* innerhalb der jeweiligen Knoten wird dann über das Interface ermöglicht.

Abbildung 5.1 stellt eine mögliche Lösung dar, inwiefern der *Schema-Controller* in das *ROS*-Framework übertragen werden kann. Dementsprechend orientiert sich die folgende Beschreibung an der genannten Abbildung. In diesem Zusammenhang stellen die jeweiligen *motor schemas* einzelne Module, beziehungsweise Knoten, dar. Jeder Knoten stellt ein Interface in Form eines *Action Servers* bereit, dass eine De- und Instanziierung der *motor schemas* zur Laufzeit ermöglicht. Der Aufbau und die Funktionsweise eines *Action Servers* sind im Abschnitt 5.1.1.1 erläutert. Die Kommunikation zwischen den *motor schemas* wird anhand der *topics Interconnection* und *Feedback* abgebildet. Dabei beschreibt *Interconnection* ein *topic*, das die Ausgabe (*motorAction_m*) aller instanziierten *motor schemas* bündelt. Demgegenüber definiert das *topic feedback* die Möglichkeit einer Kommunikation der *motor schemas* untereinander. Dieser Ansatz orientiert sich an der Theorie der Schemata und ermöglicht unabhängig von *Interconnection* eine gegenseitige Wechselwirkung der *motor schemas*. Somit kann beispielsweise eine gegenseitige Skalierung der *motorActions* erfolgen. Eine genaue Beschreibung der *motorActions*, findet sich im Abschnitt 5.1.1.1. Das *motor schema move-robot* entspricht der Summation aller *motorActions*, siehe auch 4.2. Daher bildet auch allein diese Komponente

die Schnittstelle zwischen der Hardwareabstraktion und dem *Schema Controller*. Die Hardwareabstraktion wird unter den *Communication topics* zusammengefasst und gliedert sich in *topics* mit Anbindung an die Aktorik sowie Sensorik des Zielsystems.

5.1.1.1 ***motor schema***

Ein *motor schema* definiert einerseits einen eigenständigen Knoten als auch ein Objekt¹ innerhalb des jeweiligen Knotens. Der Knoten beschreibt dabei ausschließlich eine Rahmenstruktur, die für alle Knoten identisch aufgebaut ist. Dies bezieht sich vor allem auf das Interface, welches die De- und Instanziierung der *motor schemas* als Objekte ermöglicht. Das Interface wird hierbei durch einen *Action Server* abgebildet. Eine *Action* ist ein standardisiertes Konzept innerhalb des *ROS*-Frameworks, um einzelne Knoten als Zustand einer endlichen Zustandmaschine zu kapseln. Die *Action* verfolgt dabei das Server/Client-Paradigma und definiert für Knoten einen Server als Interface. Der Client kann daraufhin die Ausführung des Knotens starten oder stoppen. Darüber hinaus kann der Client dem Server zu Beginn der Ausführung Informationen, wie zum Beispiel eine Zieldefinition übergeben. Ebenfalls können während der Ausführung Informationen zwischen dem Server und dem Client ausgetauscht werden [Sai18]. Folglich bildet die *Action* ein standardisiertes Konzept ab, welches den Anforderungen einer De- und Instanziierung zur Laufzeit gleichkommt.

Das Objekt eines spezifischen *motor schemas* definiert ein konkretes Regelwerk, den *Code of Behavior*, inwiefern das Objekt eine *motorAction* bildet. Hierbei sei erwähnt, dass unter *motorAction* ein Vektor verstanden wird. Somit definiert eine *motorAction* eine Richtung innerhalb des Zielkoordinatensystems sowie dessen Betrag. Das Regelwerk repräsentiert die Aussage der Schema-Theorie und definiert, wie die *values* auf eine *motorAction* projiziert werden. Im Sinne der objektorientierten Programmierung erfolgt die Umsetzung der *perceptual schemas* ebenfalls als Objekte. Die *perceptual schemas* sind dabei dem *motor schema* als Attribute untergeordnet und liefern die notwendigen *value*. Dafür ist jedes *perceptual schema* an ein spezifisches *topic* geknüpft und überführt anliegende *sensorValues* in *values*. Die *values* stellen in diesem Zusammenhang lediglich aufbereitete Sensorinformationen, beziehungsweise *sensorValues*, dar.

¹Objekt im Kontext einer objektorientierten Programmierung, also die Instanz einer Klasse

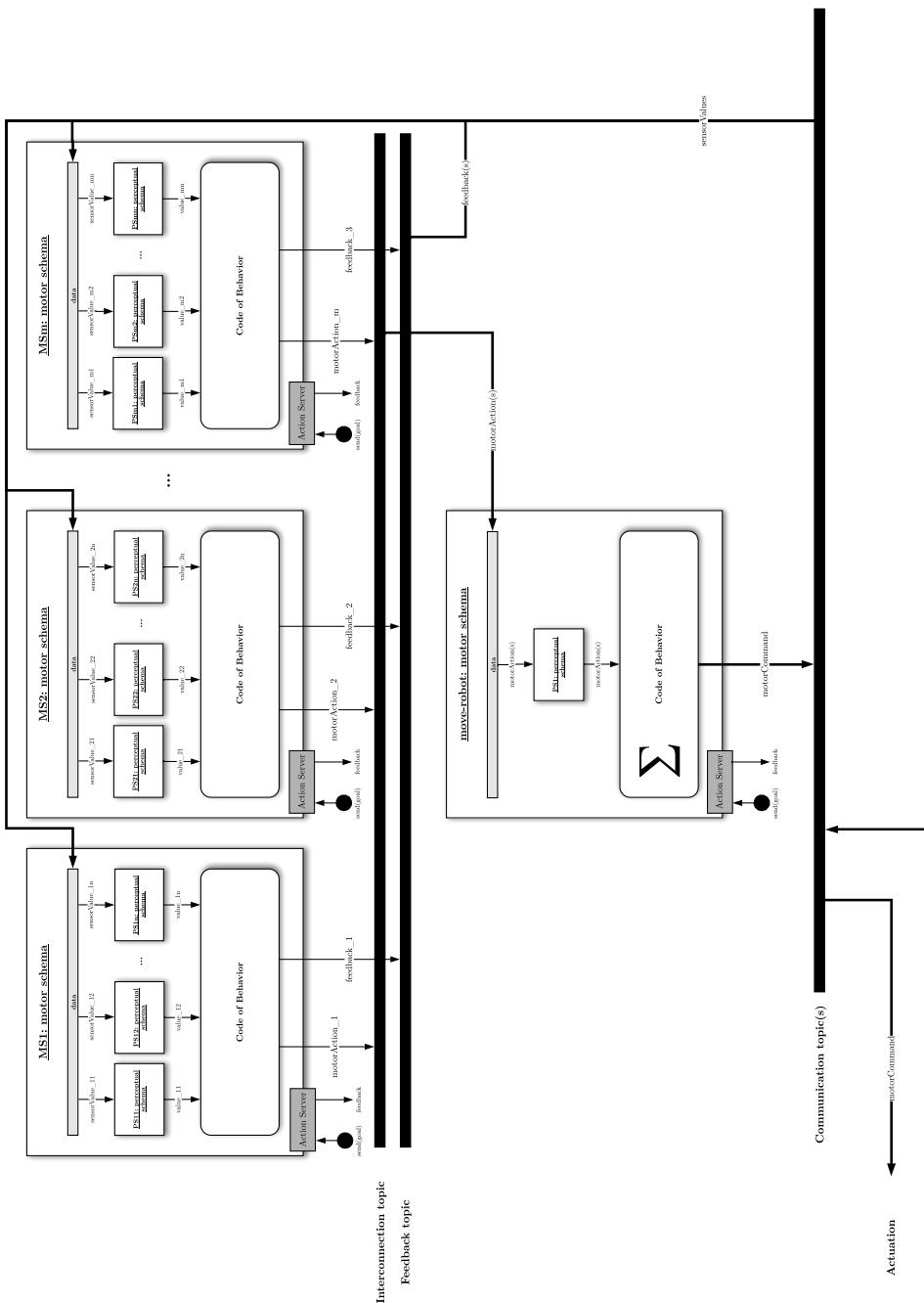


Abbildung 5.1: Darstellung der Architektur des Schema Controllers. Im Fließtext verwendete Übersetzungen für Fachtermini
(eng/de): (Actuation/Aktorik), (Sensing/Sensorik)

5.2 Plan Sequencer

Analog zum *Schema Controller* lassen sich aus der Beschreibung in Abschnitt 4.2 Anforderungen an den *Plan Sequencer* ableiten. Der *Plan Sequencer* ist im Grunde durch einen endlichen Zustandsautomaten charakterisiert. Die einzelnen Zustände beschreiben jeweils spezifische Gruppen von *motor schemas*, die zur Erfüllung eines Teilziels instanziert werden müssen. Demnach muss der *Plan Sequencer* das Interface des *Schema Controller* implementieren. Durch die Verwendung der *Action* als standardisiertes Interface, implementiert der *Plan Sequencer* somit für jedes *motor schema* den *Action-Client* als Gegenstück zum *Action-Server*.

Weiterhin ermöglicht die Verwendung der *Action* als Interface den Vorteil, dass Server und Client nicht in derselben Programmiersprache umgesetzt sein müssen. Dies ermöglicht unter anderem die Verwendung von verfügbaren *ROS*-Bibliotheken zur Abbildung einer Zustandsmaschine. Aus diesem Grund wird hier die Bibliothek *SMACH* gewählt, um die Implementierung einer Zustandsmaschine zu vereinfachen. *SMACH* bildet eine Python-Architektur zur intuitiven Erstellung von *State*-Objekten, die über Transitionen zu endlichen Automaten verknüpft werden. Die einzelnen *State*-Objekte können vom Benutzer als Klassen definiert werden [Boh18]. Somit können einzelne Objekte einerseits eine umfassende Logik implementieren und andererseits *ROS*-spezifische Interfaces verwenden. Dies schließt unter anderem die Verknüpfung mit *topics* sowie die Verwendung von *Action*-Clients ein. Folglich können jedem *State*-Objekt die notwendigen *Action*-Clients zugeordnet werden, um eine Gruppe von *motor schemas* zu de- oder instanzieren. Des weiteren bietet *SMACH* die Möglichkeit mehrere simultane Zustände abzubilden sowie Zustandsautomaten in Zustände einzubetten [Boh18].

Im Rahmen der Machbarkeitsanalyse wurde eine Evaluation der *SMACH*-Bibliothek anhand eines Multi-Roboter-Szenarios durchgeführt. Auf die detaillierte Beschreibung der Umsetzung wird hier verzichtet und auf ein Videoaufnahme des abschließenden Ergebnisses verwiesen: <https://youtu.be/GSYV1CjFuUY>.

5.2.1 Entwurf

Ein Zustand, nachfolgend als *State* bezeichnet, definiert innerhalb der *ROS*-Bibliothek *SMACH* eine Klasse, dessen Objekte mit weiteren Objekten zu einem Zustandsautomaten zusammengefasst werden können. Für die Implementierung einer *AuRA* ist es notwendig, ein Gesamtziel in Teilziele zu zerlegen und für die Erfüllung der Teilziele jeweils eine Gruppe von *motor schemas* zu definieren. Die Ziele, die einzelnen Zustände sowie die benötigten *motor schemas* sind dabei individuell von der Problemstellung

abhängig und müssen daher problembezogen definiert werden. Ebenso wie bei dem *Schema Controller* kann daher nur ein Rahmen entworfen werden, in den die *Action-Clients* eingebettet werden. Darüber hinaus kann definiert werden, unter welchen Voraussetzungen ein Zustandswechsel erfolgen darf. Der grundsätzliche Aufbau eines *States* lässt sich durch die Abbildung 5.2 visualisieren. Der innere Aufbau eines *States* kann in drei Komponenten aufgeteilt werden. Die erste Komponente *deinstantiate* repräsentiert den Eintritt in den *State*. Diese Komponente deinanstanziiert alle *motor schemas*, die im vorherigen Zustand instanziert aber im aktuellen Zustand nicht länger benötigt werden. Die nachfolgende Komponente *call* instanziert alle *motor schemas*, die im aktuellen Zustand benötigt, bisher aber noch nicht aktiv sind. Nachdem alle notwendigen *motor schemas* instanziert sind, befindet sich der *State* in einem Wartezustand. Der Wartezustand wird solange eingenommen, bis das emergente Verhalten der *motor schemas* das Teilziel erfüllt oder ein Fehler eintritt. Dafür besteht zwischen der Komponente *success control* und den aktiven *motor schemas* eine Rückverbindung. Diese Rückverbindung ermöglicht es jedem *Action-Server* Informationen an die Clients weiterzugeben. Somit können Erfolg, Misserfolg oder weitere Informationen zum Ausführungszustand an den *State* kommuniziert werden. Dabei besteht grundsätzlich keine Einschränkung, durch welches *motor schema* ein Zustandswechsel ausgelöst werden kann. Abhängig von Erfolg oder Misserfolg kann dieser Wechsel in den Nachfolgezustand oder einen Fehlerzustand führen (Abbildung 5.2: roter Ausgang). Der Fehlerzustand bildet einen möglichen Nachfolger jedes Zustands, außer sich selbst. Damit wird gewährleistet, dass ein Fehlerzustand gemäß der Fehlerbehandlung (siehe auch Abschnitt 4.2) abgefangen und zunächst im *Plan Sequencer* bearbeitet werden kann.

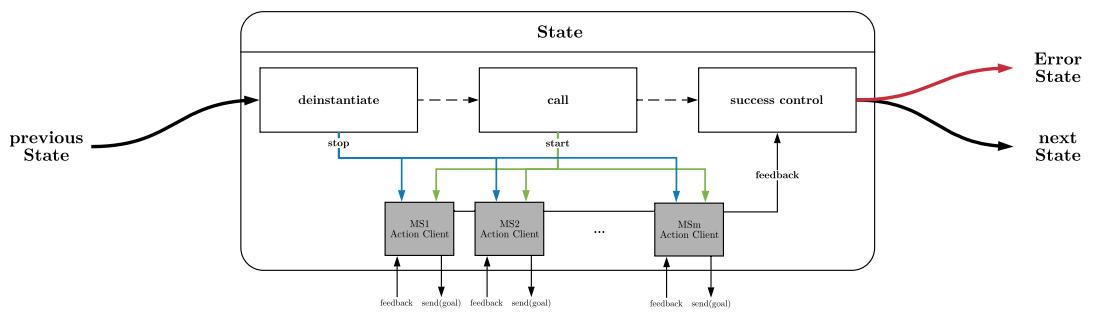


Abbildung 5.2: Entwurf eines *States* bzw. eines Zustands des *Plan Sequencer* in Komponentenform. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (*Error State*/Fehlerzustand), (*next State*/Nachfolgezustand), (*previous State*/vorheriger Zustand)

6 Anwendung der Steuerungsarchitektur auf die Problemstellung *AMiRo Assembly Line*

Dieser Abschnitt beschreibt die Implementierung des Steuerungsentwurfs in das *ROS*-Framework sowie dessen Anwendung auf eine konkrete Problemstellung. Hierbei wird der Fokus weniger auf den tatsächlichen Quellcode gelegt, sondern eher auf die konzeptionelle Ebene der Umsetzung. Dies liegt darin begründet, dass der Anspruch dieser Arbeit nicht in einer endgültigen Lösung besteht. Das hier entwickelte Konzept soll weitestgehend von einer spezifischen Programmiersprache unabhängig sein und einen möglichen Lösungsweg aufzeigen. Dennoch sind alle Details zur Implementierung und der Quellcode unter https://github.com/JanOS92/AuRA_AAL.git einsehbar. Folglich bezieht sich die nachfolgende Beschreibung auf die Übersetzung einer konkreten Problemstellung in den Kontext der *AuRA*. Dies umfasst zum Einen die Überführung der Problemstellung in eine Zustandsmaschine und zum anderen die Ermittlung aller notwendigen *motor schemas*.

6.1 AAL (*AMiRo Assembly Line*)

Die Problemstellung *AAL*, siehe auch Abbildung 6.1, beschreibt einen Teiletransport im Rahmen einer industriellen Fertigung. Der *AMiRo* dient in diesem Zusammenhang als Transportmittel für Kleinteile, wie zum Beispiel Schrauben oder Dichtungen, innerhalb der Fertigungsstraße. Somit beschreibt *AAL* ein Multi-Roboter-Szenario, indem endlich viele *AMiRos* weitestgehend unabhängig voneinander arbeiten. Dabei reagieren einzelne *AMiRos* auf Anfragen einer höheren Instanz und navigieren daraufhin autonom zu einer gewünschten Endposition. Die Navigation erfolgt anhand von *RFID*-Tags, die jeweils Pfadkreuzungen sowie Endpositionen (*Assembly 1,2, Testing, Stock 1,2*)

definieren. Die gerichtete Bewegung zwischen Start- und Endposition erfolgt anhand eines Pfadnetzwerks, dem der *AMiRo* auf kürzestem Weg zur Zielposition folgt. Neben der Pfadverfolgung muss der *AMiRo* angemessen auf verschiedene Ausnahmesituationen reagieren. Dies schließt das Einreihen von *AMiRos* vor Endpositionen sowie das Überholen von *AMiRos* im Transitbereich (Abbildung 6.1 Rundstrecke) ein [SK17].

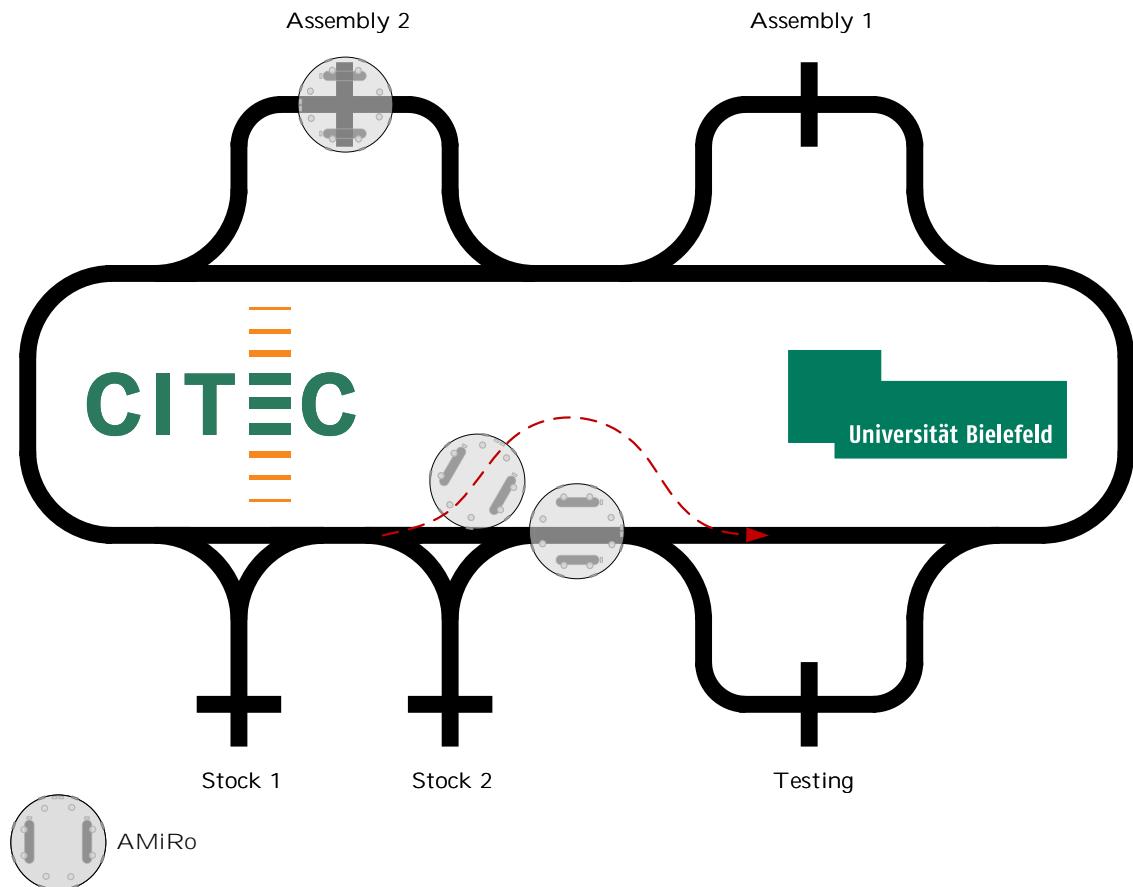


Abbildung 6.1: schematische Darstellung des AAL-Szenarios.

6.2 Definition der *motor schemas*

Für die Implementierung einer Lösung innerhalb der *AuRA* müssen zunächst aus der Problemstellung einzelne Verhaltensweisen abgeleitet werden. Diese Verhaltensweisen müssen in unterschiedlichen Kombinationen alle Anforderungen an die Implementierung erfüllen. Die Verhaltensweisen dienen dann als Grundlage für die Definition von *motor schemas*. Die notwendigen *perceptual schemas* können jedoch erst während der

konkreten Planung des *Schema Controllers* definiert werden. Eine zu frühe Planung der *perceptual schemas* würde einen zu engen Rahmen für die Implementierung der *motor schemas* definieren. Ein Grund dafür liegt im Grad der Spezifikation von *perceptual schemas*. Diese müssen bereits zum Zeitpunkt der Definition eine spezifische Ausgabe aufweisen, wohingegen *motor schemas* zunächst nur abstrakt eine Verhaltensweise repräsentieren.

Aus der Problemstellung lassen sich die folgenden notwendigen Verhaltensweisen ableiten: Pfadverfolgung in Zielrichtung, Einreihen vor Endpositionen und Überholen im Transitbereich. Darauf aufbauend lassen sich neben dem *move-robot* folgende *motor schemas* definieren: *move-to-target*, *stay-on-path*, *wait-for-obstacle* und *avoid-obstacle*.

6.3 Umsetzung des *Plan Sequencers*

Wie bereits beschrieben, legt diese Arbeit den Fokus auf die Umsetzung der Architekturkomponenten *Schema Controller* und *Plan Sequencer*. Somit beschreibt der *Plan Sequencer* die höchste hierarchische Ebene. Folglich müssen die einzelnen Zustände, beziehungsweise *States*, zunächst definiert und anschließend manuell implementiert werden. Den Rahmen für die Implementierung liefert der Entwurf des *Plan Sequencers* in Abschnitt 5.

Die Zustandsmaschine lässt sich aus dem grundsätzlichen Ablauf einer Navigation ableiten. Zu Beginn einer Navigation muss der *AMiRo* in jedem Fall von einer Startposition der Nebenstrecke¹ auf die Transitstrecke auffahren. In diesem Zustand werden gemäß der Problemstellung die folgenden *motor schemas* benötigt: *move-to-target*, *stay-on-path*, *wait-for-obstacle* und *move-robot*. Befindet sich der *AMiRo* auf der Transitstrecke, muss er dieser so lange folgen, bis die Ausfahrt zur Endposition erreicht wird. Dieser Zustand kann also durch die folgenden *motor schemas* abgebildet werden: *move-to-target*, *stay-on-path*, *avoid-obstacle* und *move-robot*. Hat der *AMiRo* die Transitstrecke verlassen und befindet sich auf der Nebenstrecke, folgt er dieser bis zum Erreichen der Endposition. Für diesen Zustand werden die selben *motor schemas*, wie im initialen Zustand benötigt. Zusammenfassend bildet die Zustandsmaschine in Abbildung 6.2 einen möglichen Lösungsansatz.

¹Abbildung 6.1: Pfad, der von der Transitstrecke zu den Endpositionen führt

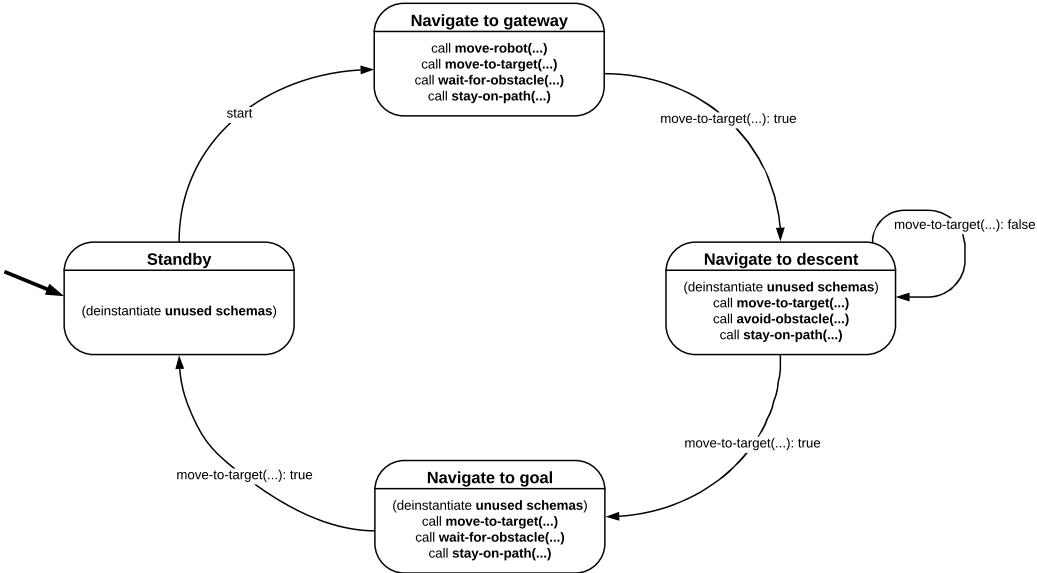


Abbildung 6.2: Zustandsmaschine der Problemstellung AAL. Im Fließtext verwendete Übersetzungen für Fachtermini (eng/de): (*gateway*/Auffahrt), (*descent*/Abfahrt)

Die Zustandsmaschine in Abbildung 6.2 zeigt bereits auf, inwiefern die Implementierung der *States* innerhalb des *Plan Sequencers* erfolgt. Dabei wird auf die Darstellung der *Action-Clients* sowie der einzelnen Komponenten eines *States* verzichtet. Darüber hinaus ist für die hier beschriebene Beispielimplementierung kein Fehlerzustand vorgesehen. Die einzelnen Zustandsübergänge werden in diesem Fall durch *move-to-target* eingeleitet. Das *motor schema* übermittelt der *State*-Komponente *success control*, dass die erwarteten Zielposition erreicht wurde. Eine solche Zielposition wird durch die ID eines *RFID-Tags* repräsentiert und kann entweder eine Auffahrt, eine Ausfahrt oder die Endposition definieren. Die *motor-schemas* *move-to-target* sowie *stay-on-path* müssen in jedem *State* neu initialisiert werden, bleiben aber instanziert. Dies ist der letztendlichen Umsetzung geschuldet und wird im Abschnitt 6.4 genauer erläutert.

6.4 Umsetzung des Schema Controllers

Im Folgenden werden die einzelnen *motor schemas* anhand ihrer Zielsetzung sowie dem genauen Funktionsumfang beschrieben. Dafür wird für jedes *motor schema* ein Schema (*Code of Behavior*) beschrieben, inwiefern das geforderte Verhalten erzielt

wird. Das Schema definiert hierbei, wie spezifische Sensorinformationen *sensorValues* auf ein Bewegungsresultat *motorAction* projiziert werden. Anschließend wird aus dem Schema abgeleitet, welche Sensorinformationen notwendig sind und demnach welche *perceptual schemas* implementiert werden müssen. Diese zielorientierte Gestaltung der *motor schemas* erlaubt eine kreative Umsetzung von Lösungsansätzen, jedoch muss im Vorfeld ein grober Entwurfsrahmen abgesteckt werden. Dieser bezieht sich vor allem auf die Umsetzbarkeit von *perceptual schemas*, welche vom Funktionsumfang des Zielsystems abhängig sind.

Die nachfolgende Beschreibung ist nah an der Darstellung des *Schema Controller* in Abbildung 5.1 orientiert.

6.4.1 move-robot

Das *move-robot motor schemas* dient der Konvertierung aller *motorActions* in einen Fahrbefehl *motorCommand*. Das *motorCommand* ist dabei bereits an das Zielsystem angepasst und kann direkt von der Aktorik interpretiert werden. Dabei ist *move-robot* zu keinem Zeitpunkt bekannt, wie viele *motor schemas* instanziert sind.

6.4.1.1 Schema

move-robot verarbeitet *motorActions* blockbasiert. Dafür werden zunächst *motorActions* innerhalb eines definierten Zeitfensters in einen Pufferspeicher geladen. Der Pufferspeicher hält dabei nicht jede einzelne *motorAction*, sondern lediglich die letzte *motorAction* jedes *motor schemas*. Die Unterscheidung der jeweiligen Absender, also der *motor schemas*, kann anhand des hier verwendeten Kommunikationsformats getroffen werden. Das ROS-Framework ermöglicht dem Benutzer die Definition eigener Kommunikationsformate. Somit enthält eine *motorAction* neben einem Vektor (Richtung und Betrag) ebenfalls den Namen des zugrundeliegenden *motor schemas*.

Nach Ablauf des Zeitfensters wird der Inhalt des Pufferspeichers in das *motorCommand* konvertiert. Dafür werden die Vektoren der einzelnen *motorActions* zu einem resultierenden Vektor addiert. Der resultierende Vektor wird anschließend mit einem Faktor gewichtet und somit in die Dimension eines Fahrbefehls übersetzt. Anschließend wird der Fahrbefehl in eine translatorische und eine rotatorische Geschwindigkeit aufgeteilt. Die translatorische Geschwindigkeit entspricht dabei dem Betrag des Fahrbefehls. Die rotatorische Geschwindigkeit wird aus der Abweichung zwischen der aktuellen Orientierung des *AMiRos* und der Orientierung des Fahrbefehls ermittelt.

Die blockbasierte Verarbeitung der *motorActions* bietet den Vorteil, dass die Frequenz der Fahrbefehle frei wählbar ist. Somit kann der *Schema Controller* an die Arbeitsfrequenz des *AMiRos* oder eines anderen *Zielsystems* angepasst werden.

6.4.1.2 *perceptual schemas*

Zum einen benötigt der Algorithmus die einzelnen *motorActions* der *motor schemas* und zum anderen wird die Orientierung des *AMiRos* im Referenzkoordinatensystem benötigt. Dafür werden die zwei *perceptual schemas* *get motorAction* und *get pose* definiert. Dabei umfasst *get motorAction* einen *Subscriber*, der die formatierten *motorActions* vom *Interconnection topic* liest. *get pose* hingegen umfasst einen *Subscriber*, der die Pose (Orientierung und Position) des *AMiRos* vom *Communication topic* liest. Ebenfalls wird ein *perceptual schema* *get feedback* definiert, um *move robot* an das *Feedback topic* anzubinden.

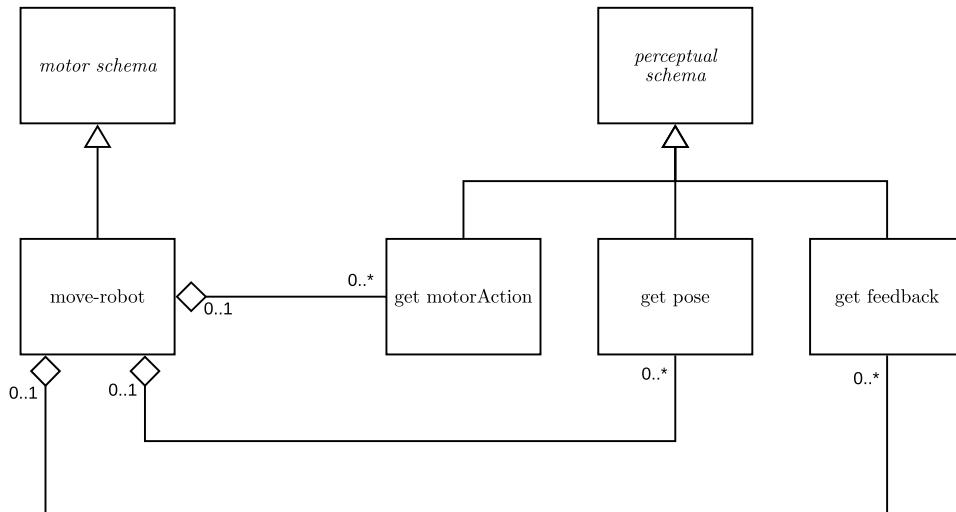


Abbildung 6.3: Domänenmodell des *move-robot motor schemas*. Die Multiplizitäten sollen hierbei andeuten, dass die *perceptual schemas* unabhängig von *move-robot* verwendbar sind.

Zur Visualisierung des objektorientierten Zusammenhangs, ist das *motor schema* *move-robot* in Abbildung 6.3 als Domänenmodell dargestellt. Die Darstellung ist hierbei rein qualitativ zu verstehen und soll einen Eindruck der Umsetzung vermitteln.

6.4.2 ***wait-for-obstacle***

Das *wait-for-obstacle motor schema* gewährleistet, dass sich der *AMiRo* auf der Nebenstrecke in eine Warteschlange einreibt. Dafür darf ein *AMiRo* eine minimale Entfernung zum vorhergehenden *AMiRo* nicht unterschreiten.

6.4.2.1 Schema

wait-for-obstacle ermittelt den Abstand zu einem *AMiRo* in Bewegungsrichtung und stoppt den *AMiRo*, falls eine minimale Distanz unterschritten wird. Dafür werden die Werte der vorderen Näherungssensoren² am Umfang des *AMiRos* mit einem Grenzwert abgeglichen. Überschreitet einer dieser Werte einen Grenzwert, entspricht dies der Unterschreitung einer minimalen Distanz. Daraufhin werden alle *motor schemas* gehemmt. Diese Hemmung erfolgt über das *Feedback topic*, auf dem ein Faktor an alle aktiven *motor schemas* kommuniziert werden kann. Es sei erwähnt, dass zusammen mit dem Faktor ein Adressat (*motor schema*) definiert werden kann, an welchen der Faktor gerichtet ist. Dafür werden Faktor und Sender in einem Kommunikationsformat über das *Feedback topic* vermittelt. Der Adressat kann so abgleichen, ob der Faktor an ihn gerichtet ist oder nicht. Aufgrund des Entwurfs muss hier zum Zeitpunkt der Implementierung entschieden werden, welche *motor schemas* über das *Feedback topic* untereinander wechselwirken können. Im Fall von *wait-for-obstacle* wird der Faktor Null an alle *motor schemas* adressiert. Somit werden alle *motorActions* zu null und der *AMiRo* stoppt solange, bis die minimale Distanz nicht mehr unterschritten wird.

6.4.2.2 *perceptual schemas*

Der Algorithmus benötigt die Werte der Näherungssensoren am Umfang des *AMiRos*. Dafür wird das *perceptual schema get ring values* definiert. *get ring values* umfasst einen *Subscriber*, der die Werte der Näherungssensoren vom *Communication topic* liest und dem *motor schema* zur Verfügung stellt. Darüber hinaus wird *get feedback* benötigt, um *wait-for-obstacle* an das *Feedback topic* anzubinden. Zur Visualisierung des objektorientierten Zusammenhangs, ist das *motor schema wait-for-obstacle* in Abbildung 6.4 als Domänenmodell dargestellt. Die Darstellung ist hierbei rein qualitativ zu verstehen und soll einen Eindruck der Umsetzung vermitteln.

²Sensoren 2 (*LEFT_FRONT*), 3 (*FRONT_LEFT*), 4 (*FRONT_RIGHT*) und 5 (*RIGHT_FRONT*)

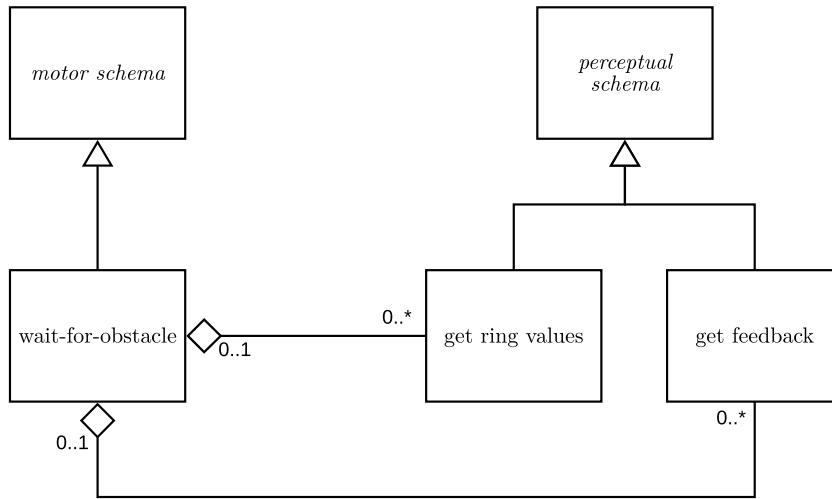


Abbildung 6.4: Domänenmodell des *wait-for-obstacle motor schemas*. Die Multiplizitäten sollen hierbei andeuten, dass die *perceptual schemas* unabhängig von *wait-for-obstacle* verwendbar sind.

6.4.3 ***avoid-obstacle***

avoid-obstacle ermöglicht einem *AMiRo* die Vermeidung von Hindernissen auf der Transitstrecke. Hierbei sei erwähnt, dass die Vermeidung von Hindernissen kein Überholmanöver abbilden kann. Dafür wäre es notwendig, dass *avoid-obstacle* den *AMiRo* von der Transitstrecke, am Hindernis vorbei, wieder zurück auf die Transitstrecke navigiert. Anstatt dieses Verhalten in einem *motor schema* zu vereinen, wird auf das emergente Verhalten von *avoid-obstacle* und *stay-on-path* gesetzt. *stay-on-path* beschreibt ein Verhalten, dass den *AMiRo* aus einem Toleranzbereich (Bereich aus dem eine Navigation zurück zur Strecke noch möglich ist) zurück auf die Strecke navigiert. Somit würde *avoid-obstacle* den *AMiRo* ausweichen lassen und *stay-on-path* führt den *AMiRo* zurück auf die Strecke, sobald das Hindernis passiert wurde. Folglich bildet das Zusammenspiel der *schemas* das geforderte Überholmanöver ab.

6.4.3.1 Schema

avoid-obstacle erfasst den Abstand zu einem Hindernis über die vorderen Närerungssensoren³ am Umfang des *AMiRos*. Anschließend werden die Werte der einzelnen Närerungssensoren normiert. Die normierten Sensorwerte werden dann verwendet,

³Sensoren 2 (*LEFT_FRONT*), 3 (*FRONT_LEFT*), 4 (*FRONT_RIGHT*) und 5 (*RIGHT_FRONT*)

um Vektoren zu skalieren. Die Ausrichtung der Vektoren ist, wie in Abbildung 6.5 a dargestellt, relativ zum Roboterkoordinatensystem fest durch ϕ_{outer} und ϕ_{inner} definiert. Jeder Vektor wird durch einen zugehörigen Sensorwert skaliert. Der resultierende Gesamtvektor wird durch Addition der einzelnen Vektoren gebildet. Die Beträge der einzelnen Vektoren geben somit an, wie schwer die jeweilige Ausrichtung gewichtet wird. Zusätzlich ist ein konstanter Vektor v_f definiert, der in Bewegungsrichtung des AMiRos zeigt. Dieser konstante Anteil stellt sicher, dass der Gesamtvektor stets eine translatorische Bewegung des AMiRos hervorruft. Somit werden rein rotatorische Bewegungen des AMiRos vermieden. Die Bestimmung der Vektoren lässt sich durch die Gleichung 6.1 beschreiben.

$$\|v_i\| = \alpha_i \cdot \frac{\text{getRingValue}(i)}{\text{maxValue}}, i = 2, 3, 4, 5 \quad (6.1)$$

wobei $\alpha_3 = 1,5 \cdot \alpha_2$, $\alpha_4 = 1,5 \cdot \alpha_3$ und $\alpha_5 = \alpha_2$

Dabei weisen die Faktoren α_i keine Einheit auf. Die Nebenbedingungen für α_i entsprechen empirisch ermittelten Verhältnissen. Somit gilt es lediglich den Faktor α_2 anzupassen, sodass der resultierende Gesamtvektor an den geforderten Wertebereich angeglichen ist. Der Wertebereich ergibt sich aus dem durchschnittlichen Betrag aller *motorActions*. Die Nebenbedingung $\alpha_4 = 1,5 \cdot \alpha_3$ gewährleistet, dass der AMiRo vorzugsweise in Richtung der Vektoren v_4 und v_5 ausweicht. So wird verhindert, dass die gegenseitige Auslöschung der Vektoren v_3 und v_4 zu einer Kollision des AMiRos mit einem Hindernis führt. Abbildung 6.5 b zeigt anhand eines Beispiels, wie ein Gesamtvektor in Abhängigkeit der Sensorwerte gebildet wird.

6.4.3.2 perceptual schemas

Für den Algorithmus werden die Werte der Näherungssensoren am Umfang des AMiRos benötigt. Daher implementiert *avoid-obstacle*, wie auch *wait-for-obstacle*, das *perceptual schema get ring values*. Darüber hinaus wird *get feedback* benötigt, um *wait-for-obstacle* an das *Feedback topic* anzubinden.

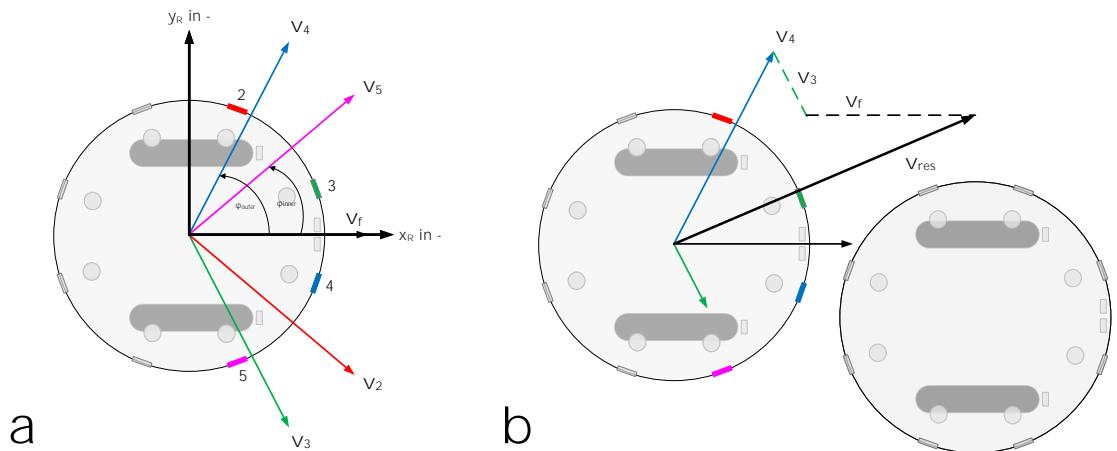


Abbildung 6.5: **a** Darstellung der Vektoren sowie der Nummerierung aller betrachteter Sensoren. Die Indizes der Vektoren definieren den zugrundeliegenden Sensor. **b** Beispiel für die Ermittlung von v_{res} durch die normierten Werte der Sensoren 3 und 4

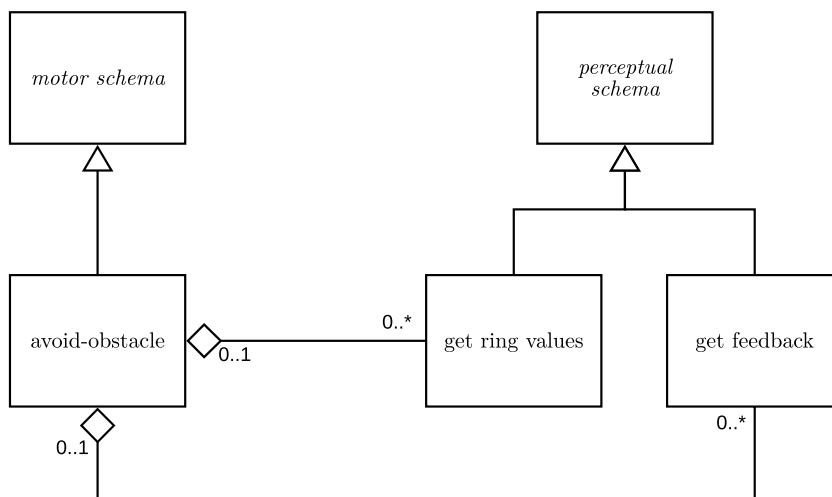


Abbildung 6.6: Domänenmodell des *avoid-obstacle motor schemas*. Die Multiplizitäten sollen hierbei andeuten, dass die *perceptual schemas* unabhängig von *avoid-obstacle* verwendbar sind.

6.4.3.3 Domänenmodell

Zur Visualisierung des objektorientierten Zusammenhangs, ist das *motor schema avoid-obstacle* in Abbildung 6.6 als Domänenmodell dargestellt. Die Darstellung ist hierbei rein qualitativ zu verstehen und soll einen Eindruck der Umsetzung vermitteln.

6.4.4 move-to-target

move-to-target ermöglicht die Navigation zwischen Start- und Zielposition durch die Verfolgung des kürzesten Pfades. Dafür muss *move-to-target* die Bewegungsrichtung entlang des Pfades ermitteln und in eine *motorAction* überführen. Jedoch kann die alleinige Bewegungsrichtung keine robuste Pfadverfolgung gewährleisten. In diesem Zusammenhang wird eine Pfadverfolgung als robust bezeichnet, sofern einer tolerierbaren Abweichung vom Pfad entgegengewirkt werden kann. Abweichungen sind zum Beispiel durch die Latenz der Steuerung oder durch die physikalische Trägheit des *AMiRos* zu erwarten. Wie auch bei *avoid-obstacle*, wird eine robuste Pfadverfolgung erst durch das emergente Verhalten der *motor schemas* ermöglicht. Die geforderte Robustheit der Pfadverfolgung lässt sich durch das Zusammenwirken von *stay-on-path* und *move-to-target* erreichen. Wobei *move-to-target* die Richtung entlang eines Pfades vorgibt und *stay-on-path* sicherstellt, dass vom Pfad nicht abgewichen wird.

6.4.4.1 Schema

Die Instanziierung von *move-to-target* erfolgt anhand einer Zielposition. Dafür wird die Zielposition als ID eines *RFID-Tags* an *move-to-target* übermittelt. Die ID wird vor dem Start der Ausführung im *Plan Sequencer* definiert und über den Action-Client vermittelt. Die ID wird von *move-to-target* zum Zeitpunkt der Instanziierung an den separaten *ROS-Knoten beacon_node* übermittelt. Dieser Knoten enthält eine interne Repräsentation des Pfadnetzwerks. Die ID eines *RFID-Tags* wird daraufhin vom *beacon_node* mit einem Schlüsselwort beantwortet. Das Schlüsselwort gibt an, welches Vektorfeld von *move-to-target* geladen werden muss, um vom aktuellen *RFID-Tag* zum geforderten *RFID-Tag* zu navigieren. Das Vektorfeld bildet den hier gewählten Ansatz zur Verfolgung eines Pfades. Die Vektorfelder werden während der Initialisierung des Programms in einem weiteren *ROS-Knoten image_converting_node* vorberechnet. Dafür enthält der *image_converting_node* das gesamte Pfadnetzwerk als Bilder, wie in Abbildung 6.7 dargestellt. Diese Bilder stellen in ihrer Kombination alle Möglichkeiten dar, wie innerhalb des Pfadnetzwerks zwischen Start- und Endpositionen navigiert

werden kann. Die Bilder weisen im Einzelnen keine Schnittpunkte, beziehungsweise Doppeldeutigkeiten auf. Innerhalb des *image_converting_node* werden die einzelnen Bilder über einen Algorithmus zunächst in Potentialfelder überführt. Die Erstellung der Potentialfelder ähnelt dabei dem Vorgehen in Abschnitt 2.2. Aus den gebildeten Potentialfeldern werden dann die Gradienten abgeleitet und durch eine weitere Verarbeitung in das resultierende Vektorfeld überführt. Der Algorithmus wird detailliert im Abschnitt 9.1.1 beschrieben. Die resultierenden Vektorfelder repräsentieren folglich eine Bewegung entlang der Pfade. Die Richtung, in die der *AMiRo* dem Pfad folgen soll, kann vor dem Programmstart festgelegt werden.

Das erzeugte Schlüsselwort wird anschließend an den *image_converting_node* übermittelt und mit einem Vektorfeld beantwortet. Hiermit ist die Initialisierung von *move-to-target* abgeschlossen. Die *motorAction* wird daraufhin aus dem Vektorfeld abgeleitet. Dafür wird anhand der Position des *AMiRos* der korrespondierende Vektor aus dem Vektorfeld gewählt. Die Vektoren sind vom Algorithmus so skaliert, dass diese ohne weitere Verarbeitung als *motorAction* ausgegeben werden können.

Abgesehen von der Bewegung entlang des Pfades leitet *move-to-target* ebenfalls die Zustandswechsel im *Plan Sequencer* ein. Die jeweiligen Zustandswechsel werden ausgeführt, sobald eine Wegpunkt der Navigation erreicht wird. Für den Zustand *Navigate to gateway* bildet zum Beispiel der *RFID*-Tag den Nachfolger, der als nächstes auf die Startposition folgt. Wird ein *RFID*-Tag vom *AMiRo* registriert, wird dieser ebenfalls an den *beacon_node* übermittelt. Abhängig von Start- und Zielposition der Navigation ermittelt der *beacon_node*, ob ein Wegpunkt erreicht wurde und meldet *move-to-target* einen Erfolg oder Misserfolg. Wurde ein Wegpunkt der Navigation erreicht, leitet *move-to-target* über die Rückverbindung zum *Plan Sequencer* einen Zustandswechsel ein. Darüber hinaus kann der Zustandswechsel, abhängig vom aktuellen Zustand, an weitere Bedingungen geknüpft sein. Dies wird im Zustand *Navigate to goal* verwendet, um den *AMiRo* genau in die Endposition zu navigieren. Dafür werden neben der Detektion des *RFID*-Tag die Unterbodensensoren des *AMiRo* mit einbezogen. Diese können den Querstrich an einer Endposition (siehe auch Abbildung 6.1) ermitteln und zusammen mit der Detektion des *RFID*-Tag die Endposition über dem *RFID*-Tag erzielen. Demgegenüber werden in den Zuständen *Navigate to gateway* und *Navigate to descent* die Zustandswechsel lediglich durch Wegpunkte eingeleitet. Nach jedem Zustandswechsel muss *move-to-target* neu initialisiert werden, um aufgrund der aktuellen Position das notwendige Vektorfeld zu laden. Dabei wird das bereits vorhandene Vektorfeld nur dann aktualisiert, wenn ein alternatives Vektorfeld für die Navigation nötig ist.

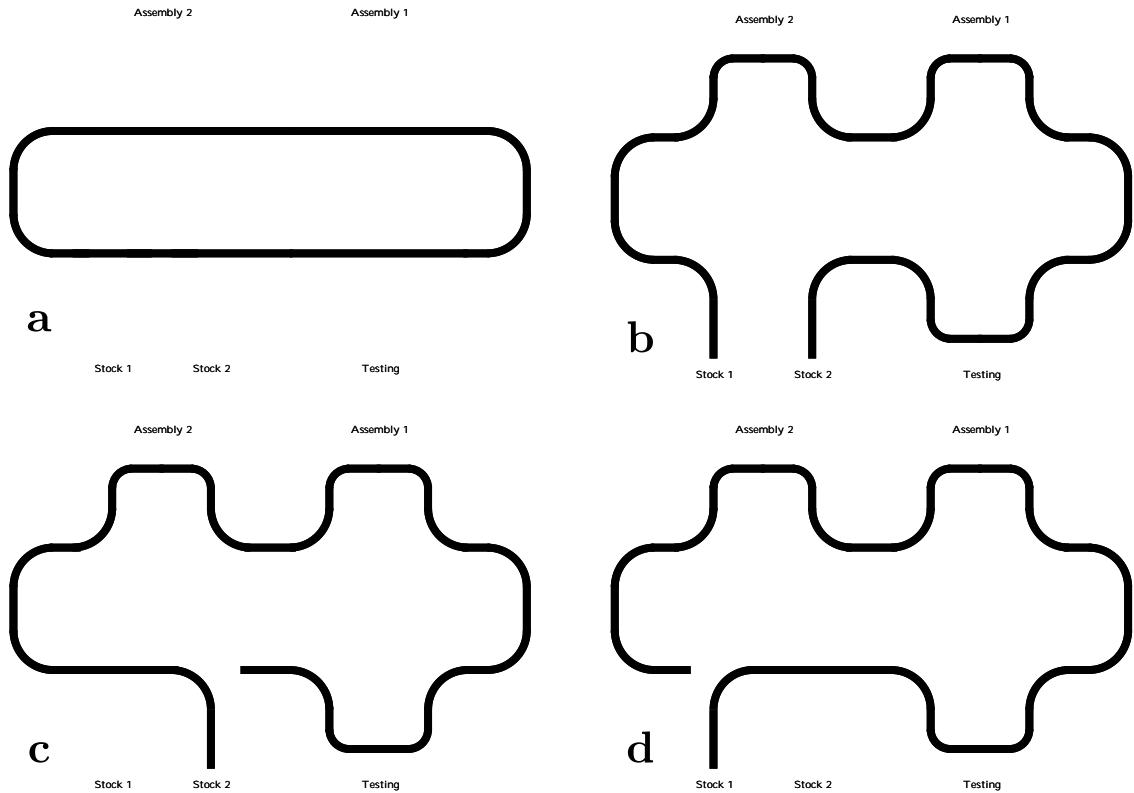


Abbildung 6.7: Aufteilung des Pfadnetzwerks AAL, um Doppeldeutigkeiten durch Kreuzungspunkte aufzuheben. Kreuzungspunkte beschreiben hierbei Abfahrten von der Transitstrecke. In Kombination bilden die Strecken a,b,c und d alle möglichen Pfade ab, die zur Navigation innerhalb des Pfadnetzwerks notwendig sind. **a** Transitstrecke **b** generelle Nebenstrecke **c** Nebenstrecke für Start gegen den Uhrzeigersinn aus Stock 2 **d** Nebenstrecke für Start im Uhrzeigersinn aus Stock 1

6.4.4.2 *perceptual schemas*

Der Algorithmus benötigt ein *perceptual schema*, dass die Kommunikation mit den ROS-Knoten *beacon_node* und *image_converting_node* ermöglicht. Für diesen Zweck wird *get vectorfield* definiert. *get vectorfield* implementiert Service-Clients, über die eine Kommunikation zwischen dem *perceptual schema* und den Knoten möglich ist. Ein Service bildet, ebenso wie die *Action*, ein standardisiertes Interface innerhalb des ROS-Frameworks. Im Gegensatz zur *Action* verfolgt ein *Service* ein Frage-Antwort-Konzept aber unterliegt ebenfalls dem Server-Client-Paradigma [Con11]. Demnach implementieren die Knoten *beacon_node* und *image_converting_node* die notwendigen

Service-Server als Gegenstück zu den Clients. Die einzelnen Client-Anfragen werden dann in den Knoten verarbeitet und beantwortet.

get vectorfield stellt dem *motor schema* das notwendige Vektorfeld zur Verfügung. Wie oben beschrieben, benötigt *get vectorfield* den aktuellen *RFID*-Tag als Startposition sowie den *RFID*-Tag der Zielposition. Mit diesen *RFID*-Tags kann über einen *Service* ein Schlüsselwort bei dem Knoten *beacon_node* angefragt werden. Anhand des Schlüsselworts kann dann über einen weiteren *Service* ein Vektorfeld bei dem Knoten *image_converting_node* angefragt werden. Es ist zu beachten, dass der aktuelle *RFID*-Tag über ein separates *perceptual schema* ermittelt wird. Dafür wird *get RFID* definiert. Das *perceptual schema* umfasst einen *Subscriber*, der den detektierbaren⁴ *RFID*-Tags vom *Communication topic* liest. Die Weiterleitung des *RFID*-Tags an *get vectorfield* erfolgt innerhalb des *motor schemas*. Neben dem *Service*-Client zur Anfrage eines Schlüsselworts, implementiert *get vectorfield* ebenfalls einen *Service*-Client zur Identifikation einzelner *RFID*-Tags. Diese Identifikation umfasst, wie oben beschrieben, den Abgleich eines gegenwärtigen *RFID*-Tags mit den erwarteten Wegpunkten der Navigation.

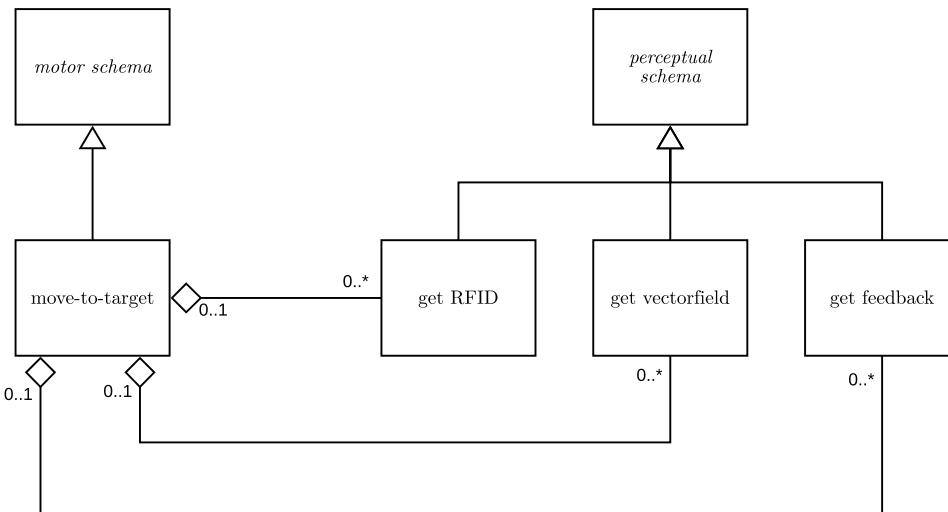


Abbildung 6.8: Domänenmodell des *move-to-target motor schemas*. Die Multiplizitäten sollen hierbei andeuten, dass die *perceptual schemas* unabhängig von *move-to-target* verwendbar sind.

Ebenfalls werden die bereits definierten *perceptual schemas* *get position* und *get feedback* benötigt.

⁴Hierbei wird angenommen, dass sich stets nur ein einziger *RFID*-Tag in Reichweite befindet

Zur Visualisierung des objektorientierten Zusammenhangs, sind die notwendigen *schemas* in Abbildung 6.8 als Domänenmodell dargestellt. Die Darstellung ist hierbei rein qualitativ zu verstehen und soll einen Eindruck der Umsetzung vermitteln.

6.4.5 *stay-on-path*

stay-on-path gewährleistet, dass der *AMiRo* während der Navigation den Pfad nicht verlässt. Dafür ist es notwendig, dass *stay-on-path* einer Abweichung vom Pfad entgegenwirkt. Eine Abweichung wird toleriert, sofern diese eine maximale Abweichung nicht überschreitet. Ein Austritt aus diesem Toleranzbereich führt zu einem Stillstand des *AMiRos*. Dieser Toleranzbereich ist näher im nachfolgenden Abschnitt beschrieben.

6.4.5.1 Schema

Die Instanziierung von *stay-on-path* erfolgt, ebenso wie *move-to-target*, anhand der Zielposition. Des Weiteren gleicht das Schema von *stay-on-path* dem von *move-to-target*. Demnach ermittelt *stay-on-path* ebenfalls ein Schlüsselwort und lädt ein Vektorfeld zur Ermittlung von *motorActions*. Auch die notwendigen Vektorfelder werden über die einzelnen Strecken in Abbildung 6.7 ermittelt. Jedoch bilden die resultierenden Vektorfelder keine Bewegung entlang des Pfades ab, sondern eine Bewegung zum Pfad. Diese Vektoren wirken so einer Abweichung vom Pfad entgegen, unabhängig in welcher Richtung der Pfad verlassen wird. Der Bereich, in dem um den Pfad Vektoren berechnet werden, wird als Toleranzbereich bezeichnet und kann beliebig angepasst werden. Somit kann ein relativ breiter Bereich genutzt werden, um komplexe Verhaltensweisen, wie das Überholen, zu ermöglichen. Wie bereits im Abschnitt 6.4.3 beschrieben, ist in diesem Zusammenhang ein Überholmanöver nur möglich, wenn *stay-on-path* und *avoid-obstacle* gemeinsam wirken. Andernfalls kann ein relativ schmaler Bereich im Zusammenwirken mit *move-to-target* die Pfadverfolgung präziser⁵ gestalten.

Der verwendete Algorithmus wird detailliert im Abschnitt 9.1.2 beschrieben. Die Vektorfelder werden ebenfalls während der Initialisierung des Programms vorberechnet. Der einzige Unterschied zwischen *stay-on-path* und *move-to-target* besteht darin, dass *stay-on-path* keinen Zustandswechsel einleiten kann. Dennoch muss das *motor schema* nach jedem Zustandswechsel neu initialisiert werden, damit das Vektorfeld an den aktuellen Zustand und die Position angepasst werden kann.

⁵Präzision bezieht sich hierbei auf eine mittlere Abweichung vom Pfad während der Navigation

6.4.5.2 *perceptual schemas*

Das Schema benötigt die gleichen *perceptual schemas* wie *move-to-target*. Zur Visualisierung des objektorientierten Zusammenhangs, sind die notwendigen *schemas* in Abbildung 6.9 als Domänenmodell dargestellt. Die Darstellung ist hierbei rein qualitativ zu verstehen und soll einen Eindruck der Umsetzung vermitteln.

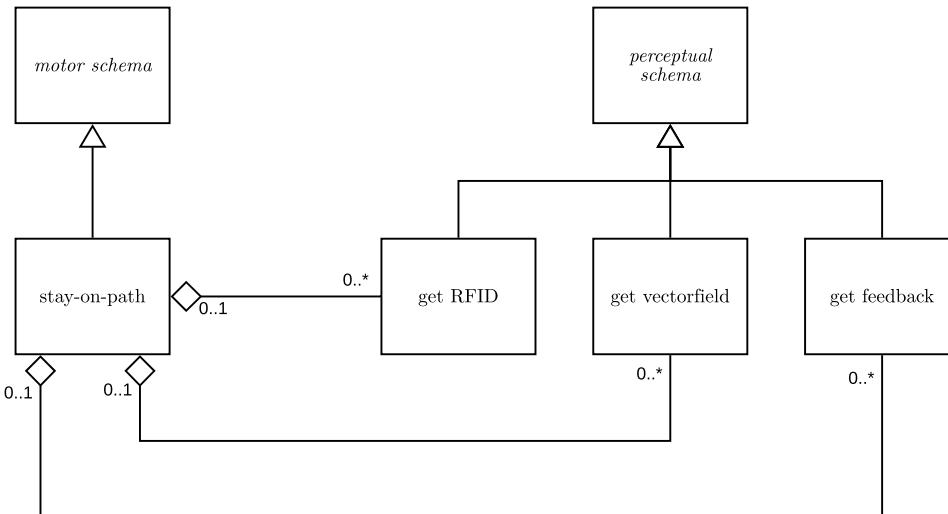


Abbildung 6.9: Domänenmodell des *stay-on-path* *motor schemas*. Die Multiplizitäten sollen hierbei andeuten, dass die *perceptual schemas* unabhängig von *stay-on-path* verwendbar sind.

7 Ergebnisse

Dieser Abschnitt beschreibt das Ergebnis dieser Arbeit anhand der Anwendung der Implementierung auf die Problemstellung *AAL (AMiRo Assembly Line)*. Die Implementierung wurde zum einen innerhalb der Simulationsumgebung *Gazebo* und zum anderen auf dem realen System in der Telewerkbank untersucht.

Das Ergebnis wird anhand von zwei ausgewählten Szenarien dargestellt. Diese werden nachfolgend, anhand des Zustandsautomaten (*Plan Sequencer*) und der *motor schemas (Schema Controller)* beschrieben. Für eine detaillierte Beschreibung der einzelnen Komponenten sei auf den Abschnitt 6 verwiesen. Als Grundlage für die Beschreibung sind Videos der einzelnen Szenarien verlinkt.

7.1 Simulationsumgebung

Video: <https://www.youtube.com/watch?v=XzC6PFxVVj8>

Das Szenario beschreibt die Navigation des *AMiRo* von der Startposition *Assembly 2* entlang der Transitstrecke zurück zur Startposition. Der Startzustand ist in der Abbildung 7.1 dargestellt.

7.1.1 *Navigate to gateway*

Zum Startzeitpunkt befindet sich der Zustandsautomat im Zustand *Navigate to gateway*. Zunächst werden die *motor schemas stay-on-path* und *move-to-target* instanziert. Anhand der Start- und Zielposition laden die beiden *motor schemas* die Vektorfelder zur Pfadverfolgung. Für die Darstellung der Vektorfelder wird im Video sowie in den nachfolgenden Abbildungen der HSV-Farbraum verwendet. Eine Beschreibung der Farbcodierung befindet sich im Abschnitt 9.1.1 in der Abbildung 9.12. Die Kombinationen der

Vektorfelder sind in den Abbildung 7.3 und 7.4 dargestellt. Die Darstellung beschränkt sich auf die Richtung, um die Übersichtlichkeit zu bewahren. Anschließend werden die weiteren *motor schemas* instanziert. Der *AMiRo* startet hier in entgegengesetzter Bewegungsrichtung. Durch die zusammenwirkenden Vektorfelder wird der *AMiRo* ausgerichtet und folgt dem Pfad in Richtung der Transitstrecke. Wird der *RFID-Tag* an der Auffahrt der Transitstrecke erreicht, leitet *move-to-target* den Zustandswechsel ein. Die Situation ist in der Abbildung 7.2 dargestellt.

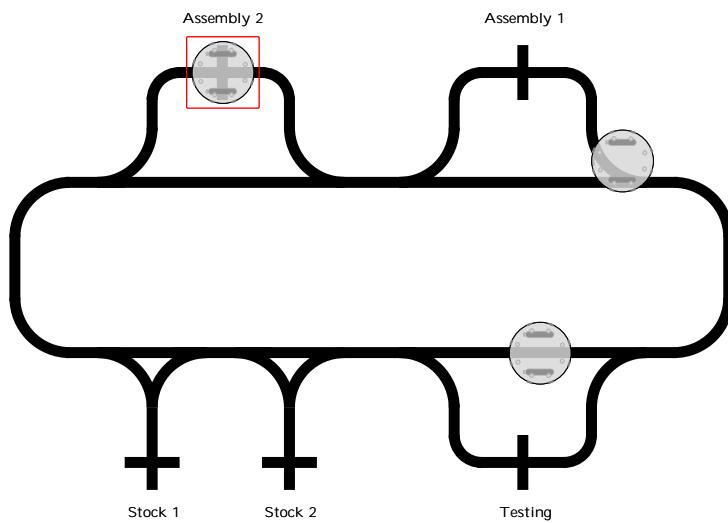


Abbildung 7.1: Position des *AMiRo* zum Startzeitpunkt. Der betrachtete *AMiRo* ist durch das rote Rechteck markiert.

7.1.2 Navigate to descent

Durch den Wechsel in den Zustand *Navigate to descent* wird das *motor schema* *wait-for-obstacle* deinstanziert und *avoid-obstacle* instanziert. *stay-on-path* und *move-to-target* werden neu initialisiert, laden aber keine neuen Vektorfelder. Eine Aktualisierung der Vektorfelder ist an dieser Stelle noch nicht notwendig. Erst bei dem Erreichen einer möglichen Ausfahrt werden die Vektorfelder für die Transitstrecke geladen. Das Laden der Vektorfelder wird durch die Rückverbindung *move-to-target(...):false* im Zustandsautomaten eingeleitet. Diese erzwingt einen erneuten Eintritt in den Zustand *Navigate to descent*. Demnach wird ebenfalls die Initialisierung von *stay-on-path* und *move-to-target* wiederholt. Somit wird an jeder möglichen Ausfahrt der Transitstrecke geprüft, ob die Vektorfelder aktualisiert werden müssen. Dieser Zustand wird solange beibehalten, bis die Ausfahrt vor der Zielposition erreicht wird. In diesem Fall leitet

7 Ergebnisse

move-to-target den Zustandswechsel ein. Die Situation wird in der Abbildung 7.5 dargestellt. Solange der AMiRo auf der Transitstrecke navigiert, verhindert *avoid-obstacle* eine Kollision mit einem Hindernis. Eine entsprechende Situation ist im Video aufgezeigt. Der AMiRo überholt Hindernisse auf der Transitstrecke durch das Zusammenwirken von *stay-on-path* und *avoid-obstacle*.

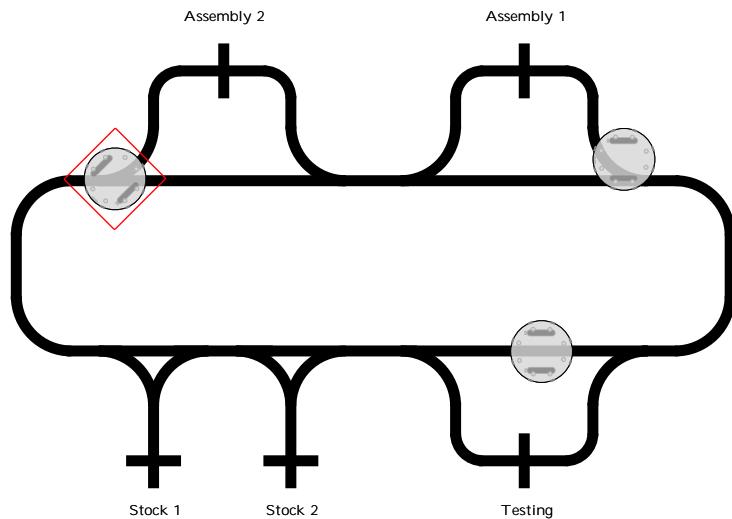


Abbildung 7.2: Position des AMiRo zum Zeitpunkt des Zustandswechsels zwischen *Navigate to gateway* und *Navigate to descent*. Der betrachtete AMiRo ist durch das rote Rechteck markiert.

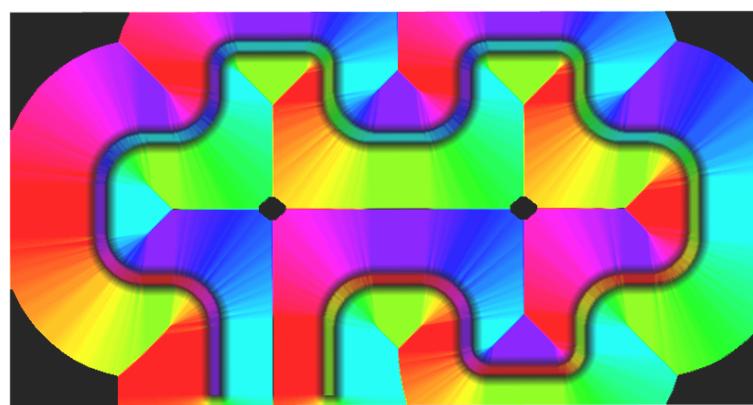


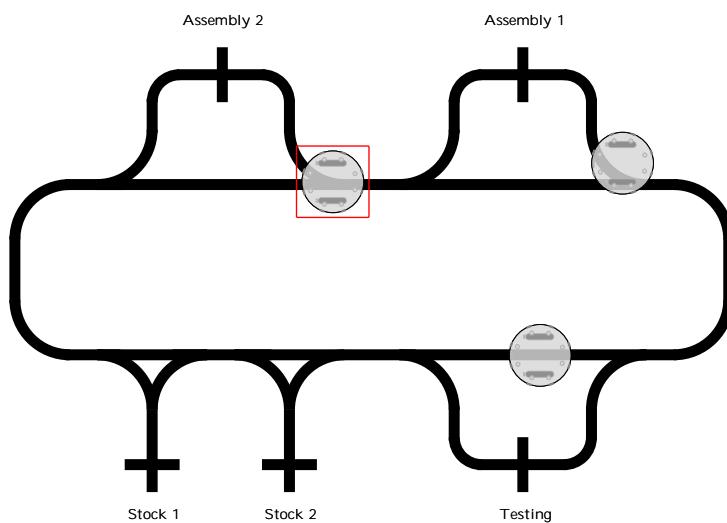
Abbildung 7.3: Kombiniertes Vektorfeld der Nebenstrecke, siehe auch 6.7. Die Berechnung der Vektorfelder erfolgt wie in Abschnitt 9 beschrieben



Abbildung 7.4: Kombiniertes Vektorfeld der Transitstrecke, siehe auch 6.7. Die Berechnung der Vektorfelder erfolgt wie in Abschnitt 9 beschrieben

7.1.3 *Navigate to goal*

Der Wechsel in den Zustand *Navigate to goal* führt zur Deinstanziierung von *avoid-obstacle* und zur erneuten Instanziierung von *wait-for-obstacle*. Die Initialisierung von *stay-on-path* und *move-to-target* führt zur finalen Aktualisierung der Vektorfelder. Daraufhin wird die Nebenstrecke bis zur Zielposition verfolgt. *move-to-target* beendet die Navigation, wenn der *RFID-Tag* der Zielposition sowie der Querstrich über dem Ziel erkannt werden.



7 Ergebnisse

Abbildung 7.5: Position des *AMiRo* zum Zeitpunkt des Zustandswechsels zwischen *Navigate to descent* und *Navigate to goal*. Der betrachtete *AMiRo* ist durch das rote Rechteck markiert.

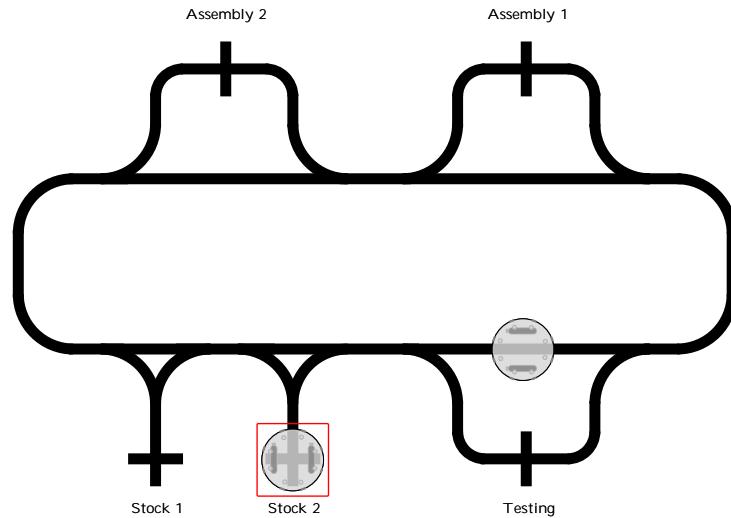


Abbildung 7.6: Position des *AMiRo* zum Startzeitpunkt. Der betrachtete *AMiRo* ist durch das rote Rechteck markiert.

7.2 Telewerkbank

Video: https://www.youtube.com/watch?v=U0d7B_Q4kNs

Das Szenario beschreibt die Navigation des *AMiRo* von der Startposition *Stock 2* entlang der Transitstrecke zur Zielposition *Assembly 2*. Der Startzustand ist in der Abbildung 7.6 dargestellt. Darüber hinaus erfolgt die Navigation analog zu der obigen Beschreibung und daher wird auf eine detaillierte Betrachtung verzichtet. Des Weiteren sei auf das verlinkte Video verwiesen.

8 Bewertung und Diskussion der Ergebnisse

Die Ergebnisse im Abschnitt 7 haben gezeigt, dass die erarbeitete Steuerungsarchitektur in das *ROS*-Framework übertragbar und auf industrielle Probleme, wie *AAL*, anwendbar ist. Solche Problemstellungen lassen sich in der Regel nur durch deterministische Algorithmen lösen. Ein Algorithmus wird als deterministisch bezeichnet, sofern dieser nur definierte und reproduzierbare Zustände zulässt. Jedoch bildet die implementierte Steuerungsarchitektur ein emergentes Verhalten ab. Ein derartiges Verhalten ist durch das konstruktive Zusammenwirken einzelner Komponenten charakterisiert. Das Zusammenwirken soll sich dabei autonom, allein durch die Interaktion der Komponenten, organisieren [Qua00]. Somit kann davon ausgegangen werden, dass emergentes Verhalten nur in wenigen Fällen definierte und reproduzierbare Zustände zulässt. Demnach bilden Emergenz und Determinismus gegensätzliche Ansätze. Allerdings zeigen die Ergebnisse, dass sich ein deterministischer Algorithmus durch Emergenz abbilden lässt. Dieses Ergebnis lässt sich nur schwer anhand einer Metrik quantifizieren und daher werden nachfolgend die Vor- und Nachteile der Implementierung beschrieben.

Ein Vorteil der Implementierung bildet die hohe Robustheit auf der Ausführungsebene. Ein Beispiel dafür bildet die implementierte Pfadverfolgung. Diese ermöglicht eine Stabilisierung der Pfadverfolgung im Rahmen einer tolerierbaren Abweichung vom Pfad. Folglich reagiert die Pfadverfolgung weniger sensibel auf den Verlust des Pfads als klassische Verfahren¹. Diese Eigenschaft ermöglicht ebenfalls die triviale Umsetzung komplexer Verhaltensweisen, wie das Überholmanöver. Ein solches Verhalten lässt sich deterministisch nur unter mäßigem Aufwand abbilden. Durch die hier verwendete Pfadverfolgung kann beliebig vom Pfad abgewichen und anschließend wieder zurückgekehrt werden. Dabei wird angenommen, dass die Abweichung vom Pfad den Toleranzbereich nicht überschreitet. Zusammenfassend gilt, dass Emergenz einen größeren Handlungsspielraum ermöglicht als klassische, deterministische Lösungsansätze. Dieser Handlungsspielraum ermöglicht es der Implementierung, Lösungswege zu verfolgen, die nicht fest definiert sind. Die Akzeptanz suboptimaler Lösungen redu-

¹Pfadverfolgung durch die sensorisch Erfassung des Pfades, z.B. durch Infrarotsensoren

8 Bewertung und Diskussion der Ergebnisse

ziert den Entwicklungsaufwand und trägt dazu bei, Konzepte robuster zu gestalten. Darüber hinaus bietet die Implementierung die Möglichkeit, einzelne Verhaltensweisen (*motor schemas*) einfach auszutauschen und in die vorhandene Steuerungsarchitektur zu integrieren.

Einen Nachteil der erarbeiteten Steuerungsarchitektur bildet der hohe Implementierungsaufwand. Unabhängig von der Komplexität des Anwendungsfalls, muss die gesamte Architektur implementiert werden. Dies umfasst, neben dem Zustandsautomaten im *Plan Sequencer*, die Kommunikationswege sowie die *motor schemas*. Darüber hinaus agieren die *motor schemas* nahezu in Echtzeit und deshalb besteht einer relativ hoher Anspruch an die Kommunikationsrate. Dies bezieht sich vor allem auf die Kommunikation zwischen den *motor schemas* und dem *move-robot schema*. Abgesehen von dem hohen initialen Implementierungsaufwand gestalten sich die Wartung und Erweiterung der Steuerungsarchitektur als weniger zeitaufwendig.

Als Nachteil der Implementierung kann zum Teil die geringe Präzision² der Pfadverfolgung angesehen werden. Die Videoaufnahmen der Ergebnisse zeigen, dass sich der *AMiRo* nur selten genau auf der Linie befindet. Dies ist vor allem auf die Latenz zurückzuführen, die aus der blockbasierten Erzeugung von Steuerkommandos resultiert. Es ist aber zu beachten, dass der geringen Präzision eine hohe Robustheit der Pfadverfolgung gegenübersteht.

²mittlere Abweichung vom Pfad während der Navigation

9 Anhang

9.1 Pfadverfolgung mit der Potentialfeldmethode

Im Rahmen dieser Arbeit wird ein Algorithmus definiert, der beliebige zweidimensionale Pfade in zwei unterschiedliche Vektorfelder überführt. Dabei gilt jedoch die Einschränkung, dass die Pfade keine Kreuzungspunkte aufweisen dürfen. Ein Kreuzungspunkt entspricht einer Doppeldeutigkeit und eine solche Form kann nicht ohne weitere Maßnahmen interpretiert werden. Darüber hinaus wird angenommen, dass ein Pfad eine schwarze Linie auf weißem Grund darstellt. Die Linie kann dabei eine geschlossene oder offene Form darstellen.

Die Kombination der resultierenden Vektorfelder soll eine Verfolgung der Pfade ermöglichen. Ein Vektorfeld repräsentiert dabei die Bewegung entlang des Pfades und das andere Vektorfeld repräsentiert die Bewegung zum Pfad.

9.1.1 Vektorfeld zur Bewegung entlang des Pfades

Gesucht wird eine Möglichkeit einen zweidimensionalen Pfad, anhand seiner Pixelgrafik, in ein Vektorfeld zu überführen. Das Vektorfeld soll eine eindeutige Bewegungsrichtung entlang des Pfades aufweisen.

Einen Ansatz für die Lösung dieser Problemstellung bildet die Potentialfeldmethode. Der Ansatz soll an einem einfachen Beispiel verdeutlicht werden.

Betrachtet seien zwei gefärbte Pixel in der Ebene, wie in Abbildung 9.1 dargestellt. Die Pixel werden anhand ihrer Färbung in ein abstoßendes und ein anziehendes Potential aufgeteilt. Dafür wird die Färbung aus dem Abschnitt 2 verwendet. Demnach entspricht das rote Pixel einem abstoßenden und das blaue Pixel einem anziehenden Potential. Werden die Pixel nun isoliert betrachtet, resultieren die in Abbildung 9.2 dargestellten Potentialfelder und die korrespondieren negativen Gradienten in 9.3. Die Potentialfelder werden dabei durch die Gleichungen 9.1 und 9.2 beschrieben.

$$P_{att}(d_{min,i}) = -\frac{1}{d_{min,i}} \quad (9.1)$$

$$P_{rep}(d_{min,i}) = \frac{1}{d_{min,i}} \quad (9.2)$$

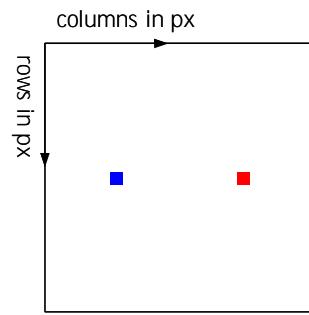


Abbildung 9.1: Beispiel zur Herleitung: Das rote Pixel repräsentiert ein abstoßendes Potential und das blaue Pixel repräsentiert ein anziehendes Potential. Pixelgrafik der Größe 25x25

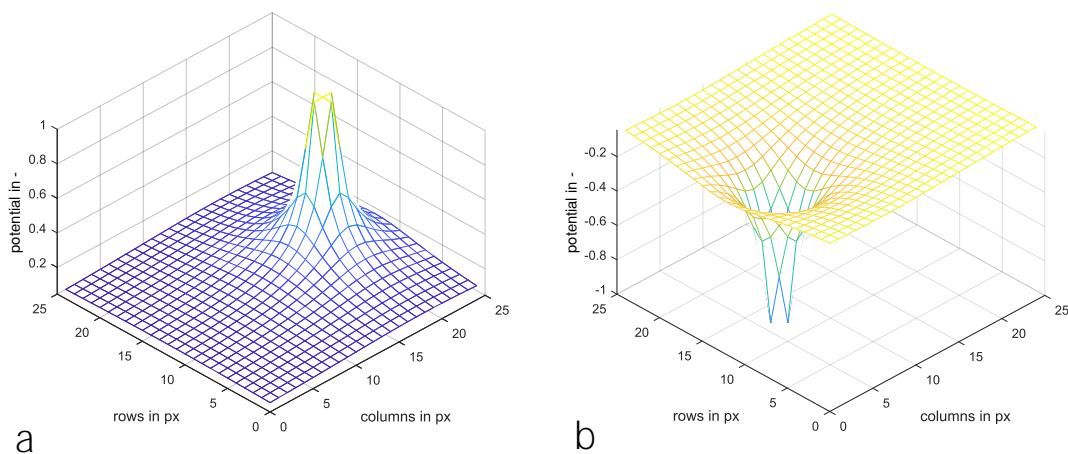


Abbildung 9.2: **a** Potential der abstoßenden Ladung gemäß Gleichung 9.2 **b** Potential der anziehenden Ladung gemäß Gleichung 9.1

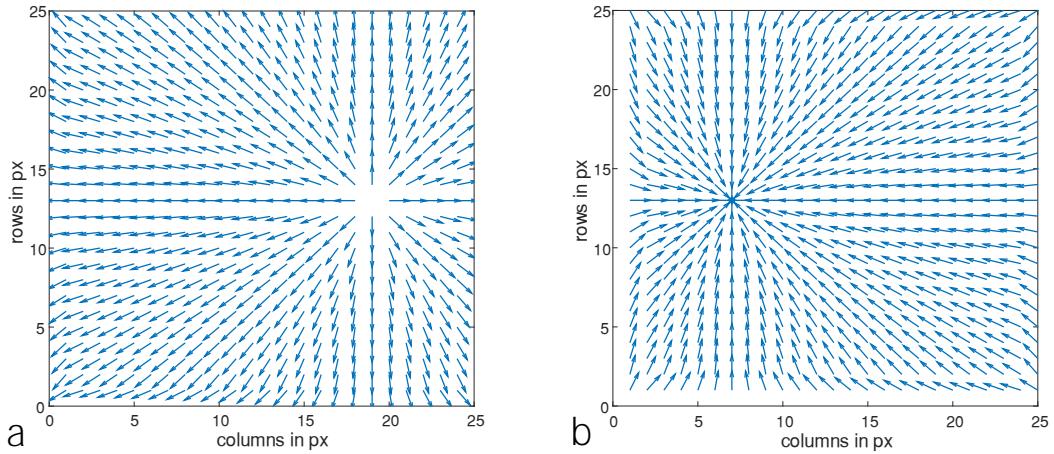


Abbildung 9.3: **a** negativer Gradient der abstoßenden Ladung **b** negativer Gradient der anziehenden Ladung

Hierbei gibt die Distanz $d_{min,i}$ den Abstand zwischen einem Pixel i und dem nächstgelegenen gefärbten Pixel an. Dabei werden für die Bestimmung von P_{att} nur die blauen Pixel und für die Bestimmung von P_{rep} nur die roten Pixel betrachtet. Ein Pixel i definiert jeden möglichen weißen Pixel. Es sei erwähnt, dass alle dargestellten Gradienten eine konstante Länge von $0,125 \frac{1}{px}$ aufweisen. Auch nachfolgend werden die Gradienten, beziehungsweise die Vektoren, eine konstante Länge aufweisen. Diese Annahme liefert die besten Ergebnisse und wurde empirisch ermittelt.

Werden nun die Pixel nicht als unterschiedliche Ladungen betrachtet, sondern als entgegengerichtete Flüsse, resultiert die Darstellung in Abbildung 9.4. Isoliert liefert die Darstellung nicht mehr Informationen, als die Erkenntnisse in Abschnitt 2. Werden die Wirbelfelder aber miteinander kombiniert, resultiert die Darstellung in 9.5. Es wird deutlich, dass zwischen den gefärbten Pixeln nahezu parallele Vektoren resultieren. Dieses Verhalten ist analog zu den magnetischen Feldlinien zwischen zwei stromdurchflossenen Leitern. Die Stromflüsse müssen eine entgegengesetzte Richtung aufweisen, ansonsten würden sich die magnetischen Felder auslöschen. Wird nun die Anzahl der gefärbten Pixel erhöht, resultiert zwischen den Pixeln ein Vektorfeld mit einer eindeutigen Richtung. Die Richtung der Vektoren ist dabei von der Anordnung der gefärbten Pixel abhängig und somit würde ein Austausch der Farben die Richtung umkehren. Eine schrittweise Erhöhung der gefärbten Pixel ist in Abbildung 9.6 dargestellt. Somit liegt es nah, den gezeigten Ansatz auf die Konturen beliebiger zweidimensionaler Pfade anzuwenden. Dafür müssen jedoch die Konturen erkannt und eindeutig voneinander getrennt werden. Anschließend müssen die Konturen nach dem obigen

9 Anhang

Prinzip eingefärbt werden. Hierfür muss ein Algorithmus implementiert werden, der die Konturen von offenen oder geschlossenen Linien extrahiert und einfärbt.

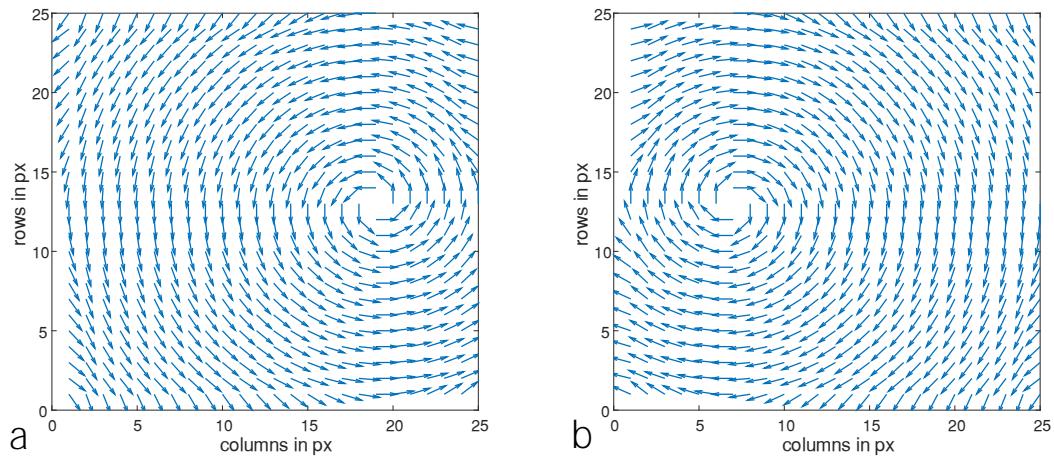


Abbildung 9.4: **a** Wirbelfeld des positiven Flusses **b** Wirbelfeld des negativen Flusses

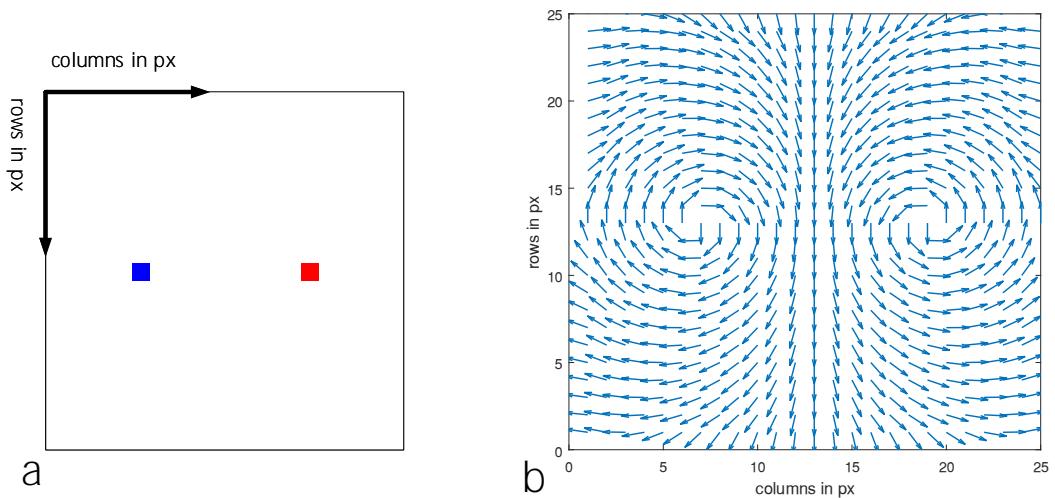


Abbildung 9.5: **a** Pixelgrafik (25x25) als Grundlage des Vektorfelds **b** kombiniertes Wirbelfeld

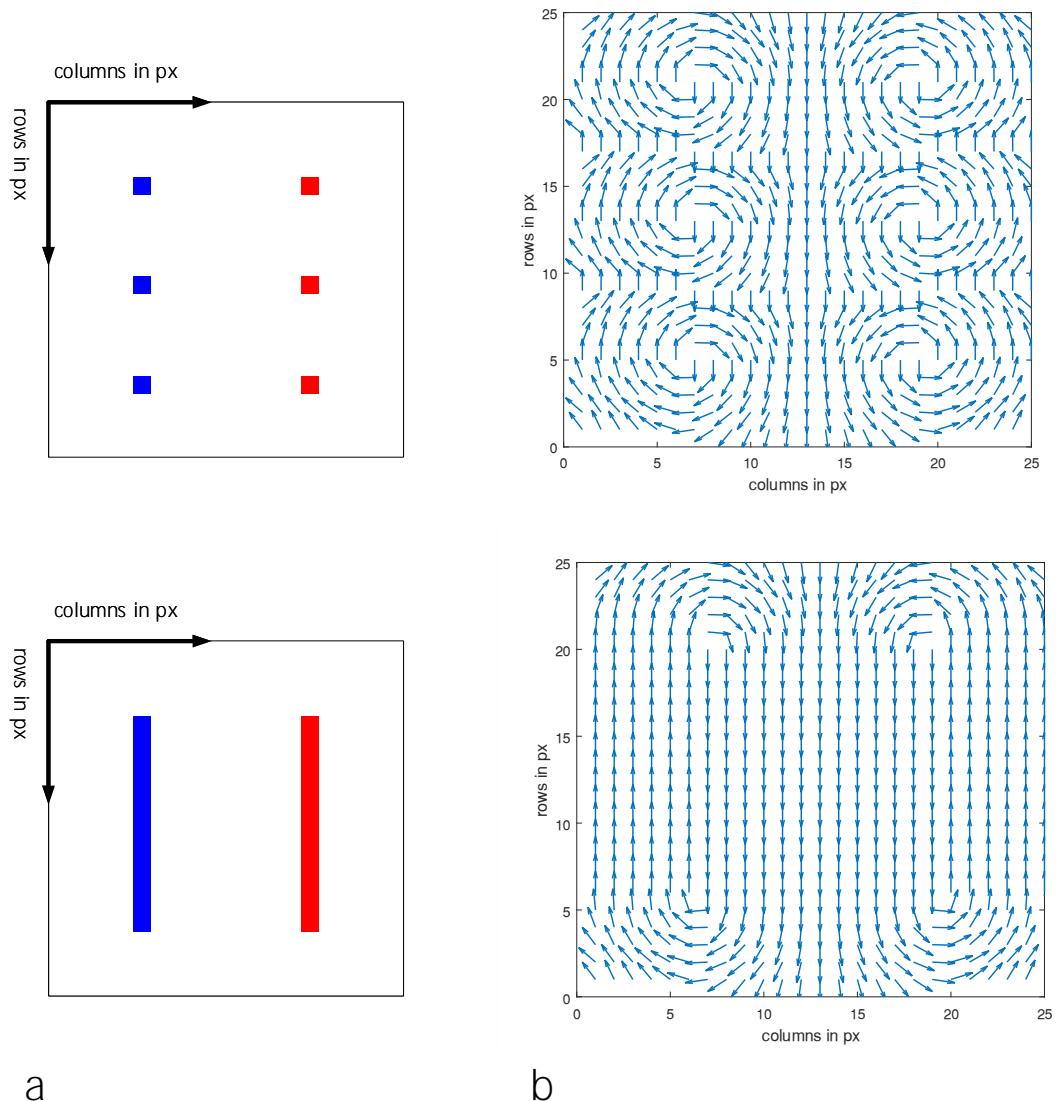


Abbildung 9.6: **a** Pixelgrafiken (25x25) als Grundlage der Vektorfelder **b** kombinierte Wirbelfelder

9.1.1.1 *contour and redye* Algorithmus

Der *contour and redye*-Algorithmus beschreibt, wie die Konturen zweidimensionaler Pfade extrahiert und anschließend eingefärbt werden. Die Einfärbung dient anschließend der Berechnung einer Bewegungsrichtung entlang des Pfades. Der Ablauf des Algorithmus und seine Komponenten werden anhand der Darstellung in 9.7 beschrieben.

Als Eingabe dient eine *24bit RGB* Pixelgrafik mit einer schwarzen Form auf weißem Hintergrund. Im ersten Schritt ermittelt die Komponente *Closeness Detection*, ob die gegebenen Form offen oder geschlossen ist. Falls die Form offen ist, also eine Linie darstellt, werden die Enden der Linie über *Corner Detection* ermittelt. Nachfolgend extrahiert die Komponente *Contour Extraction* die Konturen der Form. Hierbei wird, falls eine offene Form den Algorithmus durchläuft, die Kontur der Linie an den zuvor ermittelten Enden geöffnet. Somit resultieren für geschlossene als auch für offene Formen zwei eindeutige Konturen. Abschließend werden die Konturen sowie der Hintergrund der Pixelgrafik durch die Komponente *Contour Clustering and Dyeing* eingefärbt. Dafür werden die Konturen zunächst als Objekte erfasst und eingefärbt. Die Färbung ist dabei abhängig von der erwarteten Bewegungsrichtung auf dem Pfad. Darüber hinaus wird der Hintergrund der Pixelgrafik grau eingefärbt, damit die Berechnung des Vektorfelds ausschließlich auf dem Pfad durchgeführt wird. Wie bereits oben erwähnt, wird ausschließlich für weiße Pixel ein Potential bestimmt. Durch diese Maßnahme werden die Berechnungszeit und der Speicheraufwand der Vektorfelder reduziert. Allerdings sind an der Grenze zwischen weißen und grauen Bereichen Unstetigkeiten im Potentialfeld zu erwarten. Diese Unstetigkeiten führen zu Singularitäten im Vektorfeld, diese werden von der Berechnung der Vektoren erkannt und auf zu null reduziert.

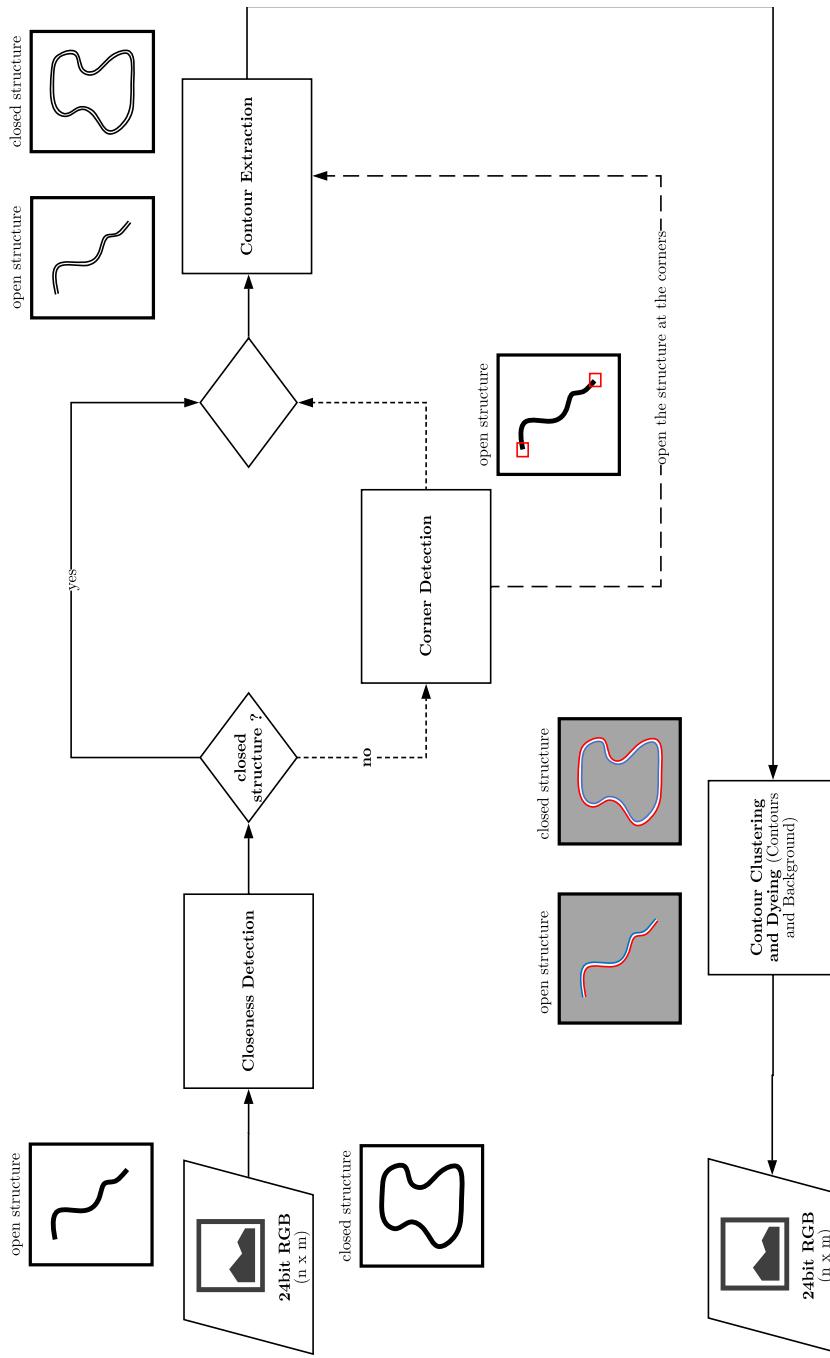


Abbildung 9.7: Prozessdarstellung des Algorithmus *contour and redye* zur Extraktion und Einfärbung von Konturen

9.1.1.2 Beispiele

Die Abbildungen 9.8, 9.9, 9.10 und 9.11 zeigen diverse Beispiele offener oder geschlossener Formen, die nach dem beschriebenen Konzept in Vektorfelder konvertiert sind. Die Vektorfelder sind im HSV-Farbraum dargestellt. Der Betrag der Vektoren wird durch die Extrusion, als auch durch den Hellwert, visualisiert. Die Richtung der Vektoren ist hierbei durch den Farbwert visualisiert. Jeder Farbwert entspricht dabei einem Winkel auf dem Farbkreis. Der Farbkreis ist in der Abbildung 9.12 dargestellt. Der Winkel, beziehungsweise die Richtung der Vektoren, bezieht sich auf ein rechtshändiges XYZ-Koordinatensystem in der unteren linken Ecke der grauen Ebene. Wie bereits erwähnt, sind die Beträge der Vektoren alle konstant und gleich.

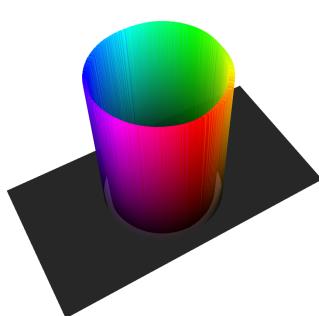


Abbildung 9.8: Vektorfeld zur Bewegung entlang eines Kreises

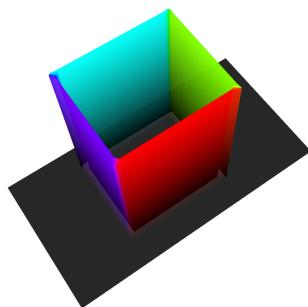


Abbildung 9.9: Vektorfeld zur Bewegung entlang eines Rechtecks

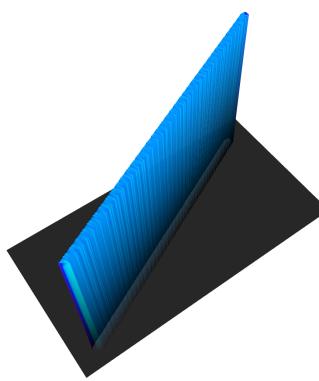


Abbildung 9.10: Vektorfeld zur Bewegung entlang einer Linie

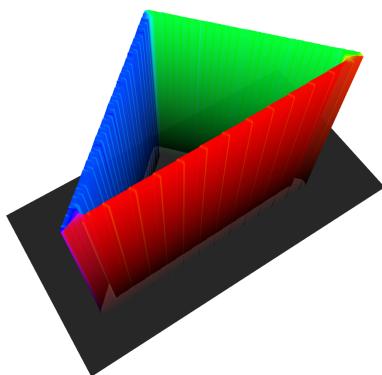


Abbildung 9.11: Vektorfeld zur Bewegung entlang eines Dreiecks

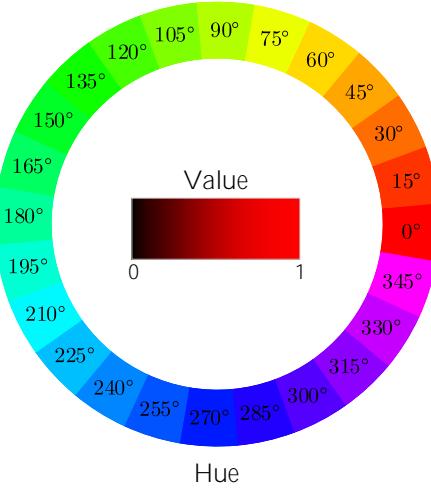


Abbildung 9.12: Farbkreis des HSV-Farbraumes mit Angabe der jeweiligen Winkel und Darstellung des Hellwertes *Value*

9.1.2 Vektorfeld zur Bewegung zum Pfad

Wie auch im Abschnitt 9.1.1, wird eine Möglichkeit gesucht einen Pfad anhand seiner Pixelgrafik in ein Vektorfeld zu überführen. Allerdings soll das Vektorfeld hierbei keine Bewegung entlang des Pfades, sondern zum Pfad hin darstellen.

Für die Bildung eines solchen Vektorfelds kann ebenfalls die Potentialfeldmethode verwendet werden. Der hier verwendete Ansatz wird nachfolgend anhand eines Beispiels beschrieben. Betrachtet sei ein einzelner, blau gefärbter Pixel in der Ebene, wie in Abbildung 9.13. Der Pixel wird per Definition als anziehendes Potential interpretiert. Das resultierende Potentialfeld sowie der negative Gradient sind in Abbildung 9.14 dargestellt. Das Potentialfeld wird dabei durch die Gleichung 9.3 beschrieben.

$$P_{att}(d_{min,i}) = \frac{d_{min,i}^2}{250px} \quad (9.3)$$

Hierbei gibt die Distanz $d_{min,i}$ den Abstand zwischen einem Pixel i und dem nächstgelegenen blauen Pixel an. Dabei werden für die Bestimmung von P_{att} nur die blauen Pixel betrachtet. Ein Pixel i definiert jeden möglichen weißen Pixel. Es sei erwähnt, dass alle dargestellten Gradienten eine konstante Länge von $0,125 \frac{1}{px}$ aufweisen. Diese Annahme liefert die besten Ergebnisse und wurde empirisch ermittelt.

9 Anhang

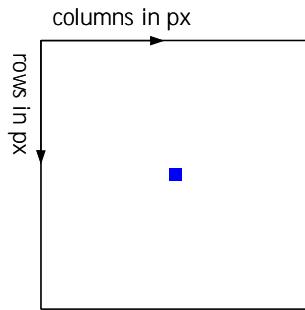


Abbildung 9.13: Beispiel zur Herleitung, das blaue Pixel repräsentiert ein anziehendes Potential. Pixelgrafik der Größe 25x25

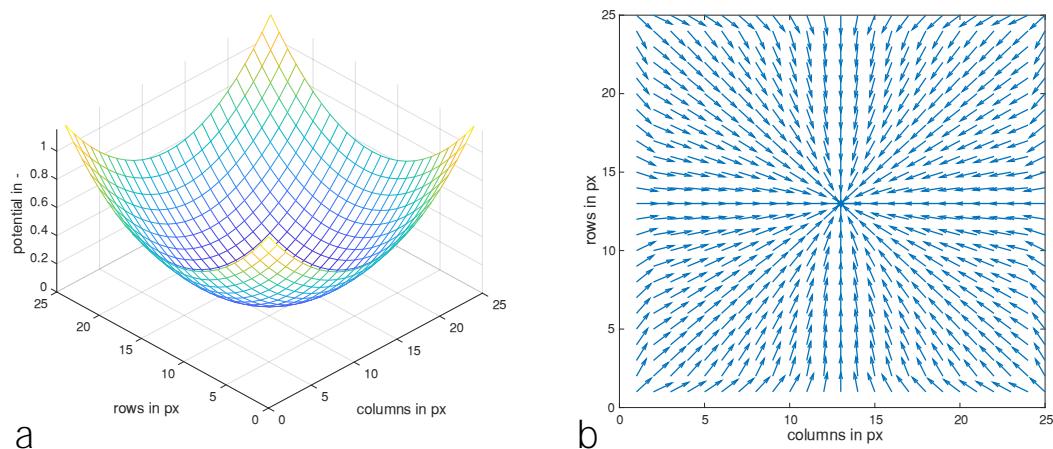


Abbildung 9.14: **a** Potential des anziehenden Potentials gemäß Gleichung 9.3 **b** negativer Gradient des Potentialfelds in a

Wie im Abschnitt 2 beschrieben, weist das Potentialfeld ein Minimum an der Stelle des anziehenden Potentials auf. Folglich zeigen alle negativen Gradienten in Richtung des anziehenden Potentials. Überträgt man diesen Ansatz nun auf eine Linie blau gefärbter Pixel, wie in Abbildung 9.15 dargestellt, resultieren daraus die Felder in Abbildung 9.16. Die negativen Gradienten definiert somit eine Bewegung zur dargestellten Linie.

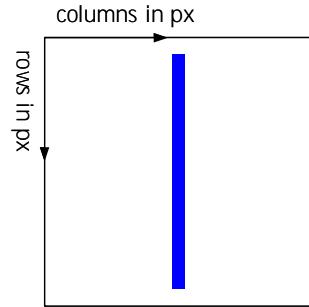


Abbildung 9.15: Linie aus blauen Pixeln. Pixelgrafik der Größe 25x25

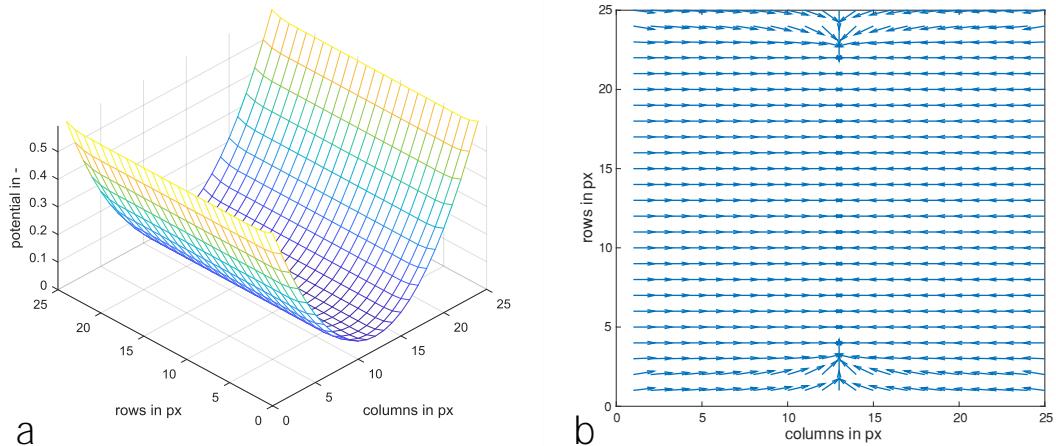


Abbildung 9.16: a Potential einer Linie aus anziehenden Potentialen gemäß Gleichung 9.3 b negativer Gradient des Potentialfelds in a

Somit kann der beschriebene Ansatz auf blau gefärbte, beliebige Pfade in der Ebene angewendet werden. Dafür muss lediglich der schwarze Pfad blau eingefärbt werden und anschließend kann das gesuchte Vektorfeld abgeleitet werden. Hierfür wird ein Algorithmus implementiert, der den Pfad als Linie erkennt, vorverarbeitet und letztendlich blau einfärbt.

9.1.2.1 *redye center* Algorithmus

Der *redye center* Algorithmus beschreibt, wie beliebige Pfade blau eingefärbt werden. Die Einfärbung bildet anschließend die Grundlage zur Berechnung eines Vektorfelds.

9 Anhang

Das resultierende Vektorfeld definiert dann die Bewegung zum dargestellten Pfad. Der Ablauf des Algorithmus und seine Komponenten werden anhand der Darstellung in 9.17 beschrieben. Die Eingabe des Algorithmus bildet eine *24bit RGB* Pixelgrafik beliebiger Größe. Die Grafik muss dabei den Pfad als eine offene oder geschlossene schwarze Linie auf weißem Grund darstellen. Anders als bei dem Algorithmus *contour and redye* wird hier nicht zwischen offenen und geschlossenen Formen unterschieden. Der erste Schritt des Algorithmus teilt sich in zwei Teilprozesse auf, die jeweils eine Teillösung hervorbringen. Die einzelnen Teillösungen werden dann im zweiten Schritt zu der Gesamtlösung kombiniert. Im oberen Prozess wird die schwarze Linie durch die Komponente *Dilate and Dyeing (Background)* erfasst und wird geweitet. Das Ergebnis der Verbreiterung wird dann als weiße Pixel in eine gleichgroße Pixelgrafik mit grauem Hintergrund eingebracht. Die Weitung ist beliebig anpassbar und definiert den im Abschnitt 6 beschriebenen Toleranzbereich, also den Bereich, in dem Potentiale berechnet und Vektoren abgeleitet werden. Im unteren Prozess wird die Linie erodiert und anschließend eingefärbt. Die erodierte Linie bietet den Vorteil, dass das resultierende Potentialfeld annähernd das Zentrum der Linie abbildet. Somit kann mit Hilfe des Vektorfeldes eine Zentrierung auf einer gegebenen Linie erreicht werden. Im zweiten Schritt werden die geweitete weiße Linie und die erodierte blaue Linie zu einer Pixelgrafik fusioniert. Wie bereits erwähnt, wird ausschließlich für weiße Pixel ein Potential bestimmt. Demnach kann hier, neben einer Einsparung von Rechenzeit und Speicherplatz, ein Toleranzbereich festgelegt werden. Der Toleranzbereich gibt an, in welchem Bereich um den Pfad einer Abweichung entgegengewirkt wird.

9.1 Pfadverfolgung mit der Potentialfeldsmethode

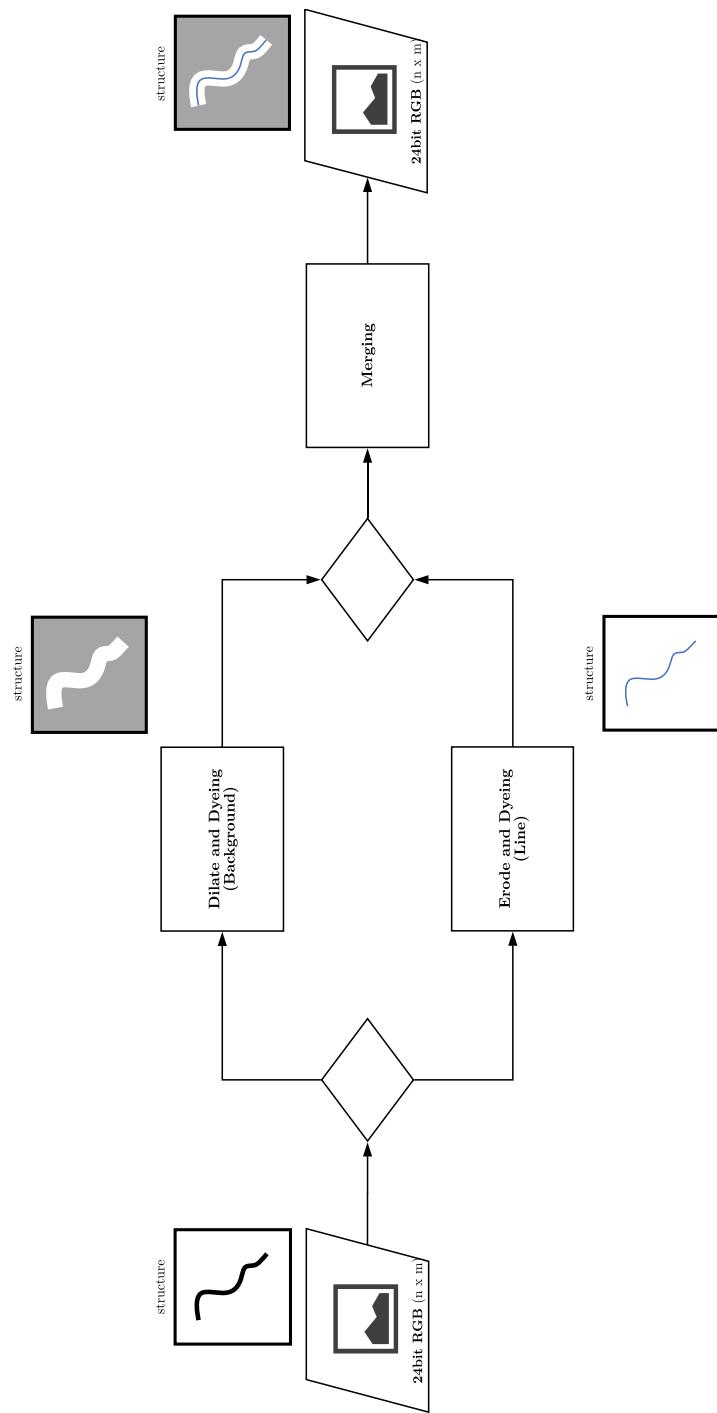


Abbildung 9.17: Prozessdarstellung des Algorithmus *redye center* zur Einfärbung von Linien

9.1.2.2 Beispiele

Die Abbildungen 9.18, 9.19, 9.20 und , 9.21 zeigen diverse Beispiele offener oder geschlossener Formen, die nach dem beschriebenen Konzept in Vektorfelder konvertiert sind. Die Vektorfelder sind im HSV-Farbraum dargestellt. Der Betrag der Vektoren wird durch die Extrusion, als auch durch den Hellwert, visualisiert. Die Richtung der Vektoren ist hierbei durch den Farbwert visualisiert. Jeder Farbwert entspricht dabei einem Winkel auf dem Farbkreis. Der Farbkreis ist in der Abbildung 9.12 dargestellt. Der Winkel, beziehungsweise die Richtung der Vektoren, bezieht sich auf ein rechtshändiges XYZ-Koordinatensystem in der unteren linken Ecke der grauen Ebene. Wie bereits erwähnt, sind die Beträge der Vektoren alle konstant und gleich.

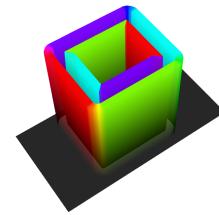
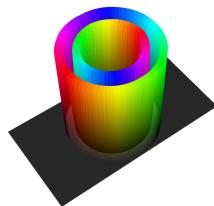


Abbildung 9.18: Vektorfeld zur Bewe-
gung zum kreisförmigen Pfad

Abbildung 9.19: Vektorfeld zur Bewe-
gung zum rechteckigen Pfad

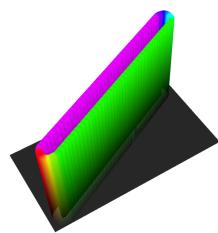


Abbildung 9.20: Vektorfeld zur Bewe-
gung zur Linie

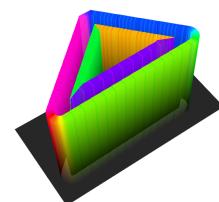


Abbildung 9.21: Vektorfeld zur Bewe-
gung zum dreieckigen Pfad

Literaturverzeichnis

- [AD92] Michael A Arbib and San Diego. Schema Theory. (January), 1992.
- [AL95] Michael A. Arbib and Jim Shih Liaw. Sensorimotor transformations in the worlds of frogs and robots. *Artificial Intelligence*, 72(1-2):53–79, 1995.
- [Ark98] Ronald Arkin. *Behavior-based robotics*. The MIT Press, 1998.
- [Boh18] Jonathan Bohren. smach - ROS Wiki, 2018.
- [Con11] Ken Conley. rosservice - ROS Wiki, 2011.
- [Gal95] CR Gallistel. Insect navigation: Brains as symbol-processors. *An Invitation to Cognitive Science*, 4:1–51, 1995.
- [Her17] Stefan Herbrechtsmeier. Modell eines agilen Leiterplattenentwurfsprozesses basierend auf der interdisziplinären Entwicklung eines modularen autonomen Miniroboters. 2017.
- [KS218] AMiRo – flexibles Helferlein passt sich an seine Umgebung an | CITEC, 2018.
- [M.O14] Jason M.O’Kane. *A Gentle Introduction to ROS*. University of South Carolina, Columbia, 2.1.1 edition, 2014.
- [Möl17] Ralf Möller. *Mobile Roboter*. Bielefeld, 2017.
- [Nei76] U Neisser. *Cognition and Reality: Principles and Implications of Cognitive Psychology*. Books in psychology. W. H. Freeman, 1976.
- [Oub09] Mohamed Oubbati. *Einführung in die Robotik*. PhD thesis, Universität Ulm, 2009.
- [Qua00] Uwe Quasthoff. Emergentes Verhalten, 2000.
- [Ron97] Tucker Balch Ronald C. Arkin. AuRA: Principles and Practice in Review. page 12, 1997.
- [ROS] de/ROS/Concepts - ROS Wiki.
- [RR02] Klaus Richter and Jan-Michael Rost. Komplexe Systeme. (30305), 2002.

Literaturverzeichnis

- [Sai18] Isaac Saito. actionlib - ROS Wiki, 2018.
- [SK17] Thomas Schöpping and Timo Korthals. AMiRo Assembly Line. pages 1–7, 2017.
- [TGM⁺13] Andry Tanoto, Javier V. Gomez, Nikolaos Mavridis, Hanyi Li, Ulrich Ruckert, and Santiago Garrido. Teletesting: Path planning experimentation and benchmarking in the Teleworkbench. *2013 European Conference on Mobile Robots, ECMR 2013 - Conference Proceedings*, pages 343–348, 2013.

Erklärung

Ich versichere, dass ich die vorliegende wissenschaftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, wurden unter Angabe der Quelle als Entlehnung deutlich gemacht. Das Gleiche gilt auch für beigegebene Skizzen und Darstellungen. Diese Arbeit hat in gleicher oder ähnlicher Form meines Wissens noch keiner Prüfungsbehörde vorgelegen.

Bielefeld, Oktober 2018

Jan O'Sullivan