Entwicklung und Implementierung einer Steuerungsarchitektur für einen autonomen mobilen Roboter in Java am Beispiel eines modifizierten Pioneer 2DX

von

Sebastian Ries

Diplomarbeit in Informatik

am Fachgebiet Simulation und Systemoptimierung
der Technischen Universität Darmstadt

bei Prof. Dr. Oskar von Stryk
Betreuung durch
Dipl. Math. Markus Glocker

August 2005

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, August 2005

Sebastian Ries

Zusammenfassung

Diese Diplomarbeit befasst sich mit der Entwicklung und Implementierung einer Steuerungsarchitektur für einen autonomen mobilen Roboter in Java. Die entwickelte Architektur HydrA (**Hy**brid **d**elibe**r**ative **A**rchitektur) soll auf dem modifizierten Pioneer 2DX des Fachgebietes Simulation und Systemoptimierung eingesetzt werden können. HydrA entspricht dem hybrid deliberativen Ansatz. Teil der Entwicklung und Implementierung ist die reaktive Steuerung und erste Komponenten zur Sensordateninterpretation und Verhalten, um die Funktionalität der Architektur zu demonstrieren. Des Weiteren ist die Schnittstelle für eine deliberative Komponente realisiert. Auf die Entwicklung einer solchen Komponente wird im Rahmen dieser Arbeit verzichtet, da dies den Umfang übersteigen würde.

Abstract

This diploma thesis deals with the development and implementation of a control architecture for a autonomous mobile robot in Java. The developed architecture HydrA (**H**ybrid **d**elibe**r**ative **A**rchitecture) is expected to run on the modified Pioneer 2DX of the Simulation and Systems Optimization Group. HydrA corresponds to the hybrid deliberative paradigm. Part of the development and implementation is the reactive control and first components for sensor interpretation and behaviours to demonstrate the functionality of the architecture. Furthermore, the interface to a deliberative component is realized. Due to the restricted extent of this thesis, the development of a deliberative component has been neglected.

Notation und Symbole

$\mathbb{N}$	Menge der natürlichen Zahlen (einschließlich 0)
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$[0, 1]$	$\{r \mid r \in \mathbb{R}, 0 \leq r \leq 1\}$
$i, j, n \in \mathbb{N}$	Indizes
$\mathcal{M}$	Menge $\mathcal{M}$
$\lvert\mathcal{M}\rvert$	Mächtigkeit der Menge $\mathcal{M}$
$\{A_i, ..., A_n\}$	$\{A_j \mid i \leq j \leq n\}$
$\sum_{i=j, i\in\mathcal{M}}^{n} B_i$	$\sum_{i=j}^{n} C_i$ mit $\begin{cases} C_i = B_i & \text{falls } i \in \mathcal{M} \\ 0 & \text{sonst} \end{cases}$
$H, P, S \in \mathbb{R}^2$	Punkte in einem zweidimensionalen Koordinatensystem
$R \in \mathbb{R}^2$	Punkt, der den Roboter repräsentiert
$\mathcal{K}_W$	kartesisches (Welt-)Koordinatensystem mit $(0, 0)$ als Ursprung
$\mathcal{K}_R$	kartesisches Koordinatensystem mit R als Ursprung, Ausrichtung der x-Achse in „Vorwärtsrichtung“ des Roboters
H_W	Koordinaten von H bezüglich $\mathcal{K}_W$
H_R	Koordinaten von H bezüglich $\mathcal{K}_R$
H^T	H transponiert
$\lVert H^T \rVert_2$	euklidische Länge des Vektors H^T

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

2050 mit einer Mannschaft von Robotern den Weltmeister im Fußball zu schlagen, lautet das ehrgeizige Ziel der Organisatoren des RoboCup (www.robocup.org). Wie Roboter zu diesem Zeitpunkt aussehen werden, weiß noch niemand genau. Sie könnten menschenähnlich sein und künstliche Haut und Muskeln besitzen. Ihr Erscheinungsbild könnte uns aus dem alltäglichen Leben vertraut sein. Die Entwicklung könnte schon soweit fortgeschritten sein, dass sich Roboter autonom in unserer Umwelt zurechtfinden, selbständig mit Menschen interagieren und uns in den verschiedensten Bereichen unterstützen.

1.1 Stand der Technik

Aus heutiger Sicht scheint das eben beschriebene mögliche Bild der Roboter visionär. Die verbreitetsten Roboter sind Industrieroboter, die gewisse Arbeiten zwar schneller und zum Teil sogar besser als Menschen ausführen, aber nur unter genau festgelegten Bedingungen eingesetzt werden können, beziehungsweise dürfen. Autonome mobile Roboter, die sich selbständig in ihrer Umwelt zurechtfinden, ohne dass diese vorher für sie präpariert wird, sind hingegen noch sehr selten. Die Forschung ist auf den verschiedensten Feldern aktiv. Neben der Entwicklung künstlicher Intelligenz, die es dem Roboter ermöglicht, selbstständig Aufgaben zu lösen, ist vor allem die Steuerung des Bewegungsapparates ein Thema, das den Einsatz im Alltag verhindert. Bei humanoiden Robotern konzentriert sich die Forschung noch auf die Koordination der grundlegenden Bewegungen der Roboter. Die Fähigkeit zum Treppensteigen von Hondas humanoidem Roboter ASIMO wurde auf dessen Europavorstellung 2003 wie eine kleine Sensation gefeiert. Der Einsatz vierbeiniger Roboterhunde von Sony (www.aibo.com) findet im erwähnten RoboCup seinen Höhepunkt. Neben ständigen Verbesserung des Bewegungsablaufs beschäftigen sich die Entwickler der teilnehmenden Teams vor allem mit Algorithmen zur Koordination des Verhaltens der Roboter auf dem Spielfeld. Die Steuerung der Bewegung radgetriebener Roboter ist weniger kompliziert. Sie kön-

nen bereits im kleinem Rahmen im Alltag, beispielsweise als Wach- oder Serviceroboter, eingesetzt werden (siehe Minerva www-2.cs.cmu.edu/~minerva, Rhino www.informatik.uni-bonn.de/~rhino/tourguide/, Neobotix www.neobotix.de).
Am Fachgebiet Simulation und Systemoptimierung dient der radgetriebene Roboter Pioneer 2DX der Firma ActivMedia als Entwicklungsplattform. An ihm soll die Fusion von Sensordaten mit realen Messzeitpunkten untersucht werden. Da eine sensornahe Zeitstempelung im Originalzustand des Pioneers nicht möglich gewesen ist, sind in mehreren Diplomarbeiten Änderungen an Hard- und Software vorgenommen worden. Von C. Vollrath ist ein neues Mikrocontrollerbetriebssystem entwickelt worden, das eine erweiterte Sensordatenkommunikation ermöglicht (vgl. [13]), und U.R. Vollrath hat eine Roboter-API implementiert, die den Zugriff auf die Sensoren und Aktoren des Roboters in Java erlaubt (vgl. [14]). Da die vom Hersteller zur Verfügung gestellte Steuerungsarchitektur Saphira aufgrund der bisherigen Modifikationen nicht mehr eingesetzt werden kann, soll im Rahmen dieser Arbeit eine neue Steuerungsarchitektur entwickelt werden.

Abbildung 1.1: Pioneer 2DX

1.2 Ziele und Anforderungen

Das wesentliche Kennzeichen eines autonomen mobilen Roboter ist, dass er sich selbständig in einer unbekannten oder auch dynamischen Umgebung bewegen kann. Dazu benötigt er Sensoren, um Informationen aus der Umgebung aufzunehmen, und Aktoren, um seine Umgebung zu verändern, beziehungsweise, um sich zu bewegen. Der Pioneer 2DX des Fachbereiches erfüllt diese Voraussetzungen. Um sich kollisionsfrei zu bewegen oder Karten zu erstellen, ist darüber hinaus eine Komponente erforderlich, welche die Sensoren und Aktoren steuert und die anfallenden Daten verarbeitet. Diese wird als Steuerungsarchitektur bezeichnet. Nach Mataric bestimmt die Steuerungsarchitektur die prinzipielle Art und Weise

wie die Steuerung eines Roboters organisiert wird (vgl. [8] S.10). Eine Steuerungsarchitektur legt demnach die wesentlichen Funktionsblöcke und Datenstrukturen fest, die als Grundlage zur Steuerung dienen. Das Ziel dieser Arbeit ist die Entwicklung einer Steuerungsarchitektur für den modifizierten Pioneer 2DX. Des Weiteren werden erste Module zur Sensordatenverarbeitung und Verhalten implementiert, welche die Funktionalität der Architektur demonstrieren und als Grundlage für weitere Entwicklungen dienen können.
Die zentrale Aufgabenstellung ist damit zwar treffend beschrieben, jedoch gilt es weitere Kriterien zu erfüllen, auf die im Folgenden eingegangen wird. Die Architektur soll dem Roboter eine gewisse Robustheit in der Interaktion mit seiner Umgebung ermöglichen, so dass er sich auch bei einzelnen fehlerhaften Sensordaten noch zurechtfinden kann. Beispielsweise ist es möglich, die Position des Roboters nur aus den Odometriedaten der Radencoder zu berechnen. Für ein erstes Verfahren zur Positionsbestimmung scheint dieses Vorgehen auch sinnvoll, allerdings führt in diesem Fall bereits ein durchdrehendes Rad dazu, dass der Roboter seine Position nicht mehr korrekt bestimmt. Um das Verhalten des Roboters robust gegen solche Einflüsse zu machen, ist es nötig, solche Situationen durch Fusion der Daten mehrerer Sensoren zu erkennen und soweit möglich zu korrigieren. Um dabei das bestmögliche Ergebnis zu erzielen, müssen alle verfügbaren Sensordaten innerhalb der Architektur zur Verfügung gestellt werden. Darüber hinaus soll der Roboter sowohl um Sensoren wie auch um Module zur Sensorfusion beziehungsweise -interpretation erweiterbar sein. Damit ist es denkbar, dieselbe Steuerungsarchitektur für verschiedene Roboter mit unterschiedlichen Aufgaben einzusetzen. Ein weiterer Aspekt, den es zu erfüllen gilt, ist die Remote-Fähigkeit der Architektur. Es soll möglich sein, die Steuerungsarchitektur auf einem anderen Computer zu betreiben, als dem Roboter selbst. Hierzu muss es möglich sein, dass der Roboter die Sensordaten, die er aufnimmt, über ein Netzwerk, beispielsweise LAN oder WLAN, an einen entfernten Computer schickt und auf dem gleichen Weg Befehle zur Steuerung zu empfängt. Der Grund für diese Anforderung ist die beschränke Rechenleistung des Pioneer 2DX. Dieser ist mit einem Pentium II mit einer Taktfrequenz von 400MHz und 256MB Hauptspeicher ausgestattet und entspricht somit nicht mehr dem aktuellen Stand der Technik. Die Basisfunktionalität der Remote-Schnittstelle ist verhältnismäßig einfach zu integrieren, da sich keine Änderungen am Steuerungsablauf ergeben. Der Anspruch, eine robuste Steuerungsarchitektur zu schaffen, die auch bei Verzögerungen in der Datenübertragung oder einem Verbindungsabbruch ein stabiles Verhalten liefert, stellt hier die Herausforderung dar. Des Weiteren scheint „Echtzeitfähigkeit“ auf den ersten Blick ein wünschenswertes Kriterium zu sein. Um dieses Schlagwort etwas greifbarer zu machen, soll es im Sinne der folgenden Definition verwendet werden.

> „Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit

> sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. ... “ (DIN 44300 - siehe [10], Kapitel Einführung)

Die Kernaussage dieser Norm ist, dass die Verarbeitungsergebnisse eines echtzeitfähigen Systems innerhalb einer vorgegebenen Zeitspanne vorliegen müssen. Aufgrund der Geschwindigkeit (bis ca. 2 m/s) mit der sich der Pioneer bewegt, werden zur Steuerung Zeitspannen in der Größenordnung von 100ms eingesetzt (vgl. [2]). Da die Steuerungsarchitektur in Java realisiert werden soll und auf dem Pioneer das Betriebssystem Windows eingesetzt wird, ist in diesem Rahmen keine Echtzeitfähigkeit möglich. Als Stichworte seien hier Garbage-Collection in Java und Multitasking unter Windows 2000 aufgeführt. Auch die TCP/IP Verbindung, die der Remote-Schnittstelle zu Grunde liegt, gibt keine Garantien, innerhalb welcher Zeitspanne Pakete ausgeliefert werden. Statt einem echtzeitfähigen System soll ein praxistaugliches System entwickelt werden, mit dem es möglich ist, den Roboter flüssig zu bewegen. Es gilt folglich größere Verzögerungen soweit wie möglich zu vermeiden, beziehungsweise wenn nicht anderes möglich, den Roboter sicher zu stoppen, um eine unkontrollierte Fahrt zu verhindern. Als letzter Gesichtspunkt sei hier das Softwaredesign genannt. Da diese Architektur die Grundlage für weitere Entwicklungen sein soll, ist es wichtig, auch hierauf besonders zu achten. Die Software soll den Konzepten der Objektorientierung entsprechen und vor allem Erweiterbarkeit und Austauschbarkeit von Komponenten unterstützen.

1.3 Vorgehensweise

Im folgenden Kapitel werden die drei wesentlichen Ansätze für Steuerungsarchitekturen vorgestellt, die in der Fachliteratur bekannt sind. Im Kapitel 3 wird die Funktionalität der neuen Architektur HydrA und der entwickelten Komponenten erklärt. Das vierte Kapitel beschreibt die wesentlichen Merkmale der Implementierung der Architektur und der Remote-Schnittstelle. Zusätzlich wird der realisierte Simulator vorgestellt, der es ermöglicht die Komponenten in einer virtuellen Umgebung zu testen. Im letzten Kapitel wird ein Vergleich zwischen den gestellten Anforderungen und der entwickelten Architektur vorgenommen und Möglichkeiten für zukünftige Erweiterungen aufgezeigt.

Kapitel 2

Ansätze für Steuerungsarchitekturen

In diesem Kapitel wird die Entwicklung der drei wesentlichen Ansätze für Steuerungsarchitekturen seit den 60er Jahren vorgestellt (vgl. [8] S.5). Alle drei Ansätze können mit Hilfe der Komponenten SENSE, PLAN und ACT beschrieben werden, welche die wesentlichen Funktionsblöcke einer Architektur repräsentieren. SENSE steht dabei für die Komponente, welche für das Erfassen und Verarbeiten der Sensordaten zuständig ist. PLAN ermittelt eine geeignete Vorgehensweise zur Erfüllung einer vorgegeben Aufgabe. Die Komponente ACT steuert die Aktoren und damit letztlich die Bewegungen des Roboters. Die Abgrenzung der verschiedenen Ansätze erfolgt durch Unterschiede im Zusammenspiel dieser Komponenten (vgl. Abbildung 2.1).

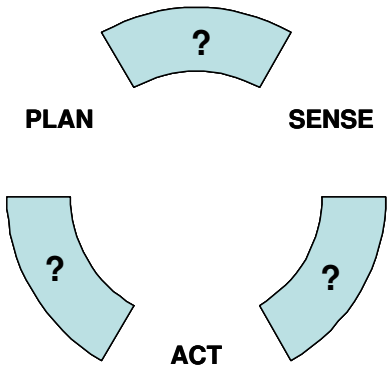

Abbildung 2.1: SENSE-PLAN-ACT

2.1 Hierarchischer Ansatz

Der hierarchische Ansatz ist die älteste Vorgehensweise, die vorgestellt werden soll. Der Roboter führt in diesem Modell die Schritte SENSE, PLAN und ACT streng sequenziell aus (siehe Abbildung 2.2). Die Grundlage dieses Ansatzes ist

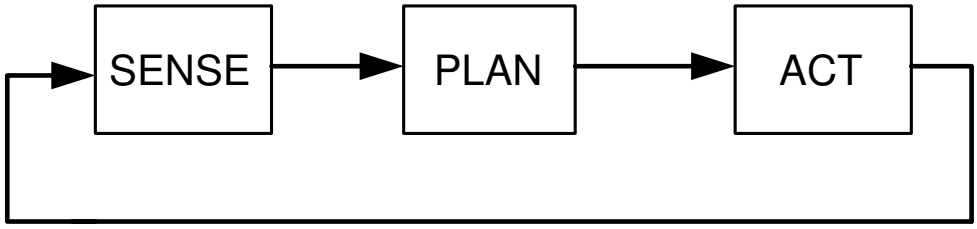

Abbildung 2.2: Hierarchischer Ansatz

die horizontale Dekomposition einer Aufgabe, also die Zerlegung in zeitlich aufeinander folgende Teilaufgaben. Im ersten Schritt werden hier die Sensordaten erfasst und aufbereitet. Anschließend erstellt die Planungskomponente aus diesen Daten ein Modell der Umwelt und ermittelt eine Abfolge von Steueranweisungen, mit der die gestellte Aufgabe erfüllt werden kann. Die erste Anweisung wird an die Steuereinheiten der Motoren weitergegeben und umgesetzt. Wenn die Durchführung der Aktion abgeschlossen ist, beginnt der Zyklus von neuem. Die Planungskomponente hat hier sehr viel zu leisten, da in jedem Zyklus der komplette Plan zur Lösung der gesamten Problemstellung ab dem aktuellen Stand neu berechnet wird. Shakey, der erste Roboter mit künstlicher Intelligenz (vgl. [8] S.42) arbeitete auf diese Weise. Er wurde ab 1967 am Stanford Research Institute (SRI) entwickelt. Die Planungskomponente wurde mit *Strips* realisiert. Dies ist ein logikbasierter „Allgemeiner Problem Löser“ (engl. general problem solver). Das Weltmodell wurde dabei mit Hilfe der Annahme der Weltabgeschlossenheit (engl. closed world assumption) aus einer Menge von vorgegeben Fakten und Regeln, den jeweiligen Sensordaten und den mittels des verwendeten Logikkalküls herleitbaren Schlüssen gebildet (vgl. [8] S.53 ff). Neben den Problemen, die sich beim Erstellen des Weltmodells ergaben, war ein wesentlicher Nachteil dieses Vorgehens, dass die ständige Neuberechnung des Plans keine flüssige Bewegungen zu Stande kommen ließ, sondern immer wieder Denkpausen erforderte.

2.2 Reaktiver Ansatz

Der reaktive Ansatz, der seit Ende der 80er verfolgt wird und den hierarchischen Ansatz ablöste, sollte das Problem mit den langen Planungspausen lösen. Bei diesem Vorgehen wurde die Planungskomponente komplett aus der Architektur entfernt und die Module SENSE und ACT direkt verbunden.

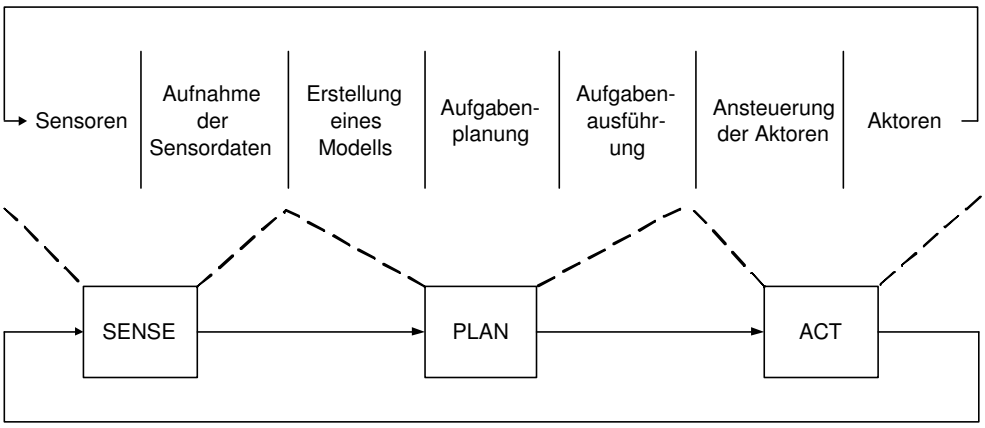

Abbildung 2.3: Horizontale Dekomposition (vgl. [8] S.106)

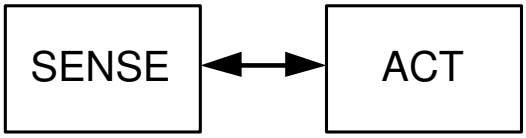

Abbildung 2.4: Reaktiver Ansatz

Jede Aufnahme von Sensordaten führt damit zu einer Aktion beziehungsweise Reaktion. Die Wahrnehmung eines Hindernisses kann beispielsweise direkt eine Ausweichbewegung einleiten. Die einzelnen SENSE-ACT Module werden auch Verhalten genannt. Eine der bekanntesten reaktiven Steuerungsarchitekturen wurde von Rodney Brooks entwickelt und ist unter dem Namen „Subsumption Architecture“ bekannt [4, 8]. Anders als beim hierarchischen Ansatz gliedert Brooks seine Architektur vertikal nach der Abstraktionsebene der zu erfüllenden Aufgaben (siehe Abbildung 2.5). In der Abbildung steigt der Abstraktionsgrad von unten nach oben an. Die Idee ist, durch Überlagerung einfacher Verhalten eine Steuerung für komplexe Aufgaben zu entwickeln. Dabei sollen höher gestellten Verhalten die Aktionen unterstellter Verhalten unterdrücken können. Es sollte möglich sein, die Architektur an einer beliebigen Stelle zwischen zwei Ebenen zu trennen und den unteren Teil als eigenständige Steuerungsarchitektur zu verwenden. Allerdings beschreibt Brooks nur das Zusammenspiel der unteren drei Schichten in [4], wie die weiteren Schichten aufgebaut werden sollen, lässt er offen.

In der Integration der abstrakteren Schichten liegt das Problem einer reaktiven Architektur. Da jedes Verhalten einen Reiz zur Aktivierung benötigt und nur solange aktiv ist, wie der Reiz anhält, kann auf ein Weltmodell verzichtet werden. Die Planung, welche Verhalten wann aktiviert werden sollen, wird nicht von einer Planungskomponente ausgeführt, sondern dem Entwickler der Verhalten überlassen. Dieser legt beim Design der Verhalten fest, welche Reize ein Verhalten auslösen sollen. Soll eine Sequenz von Verhalten ausgeführt werden, so

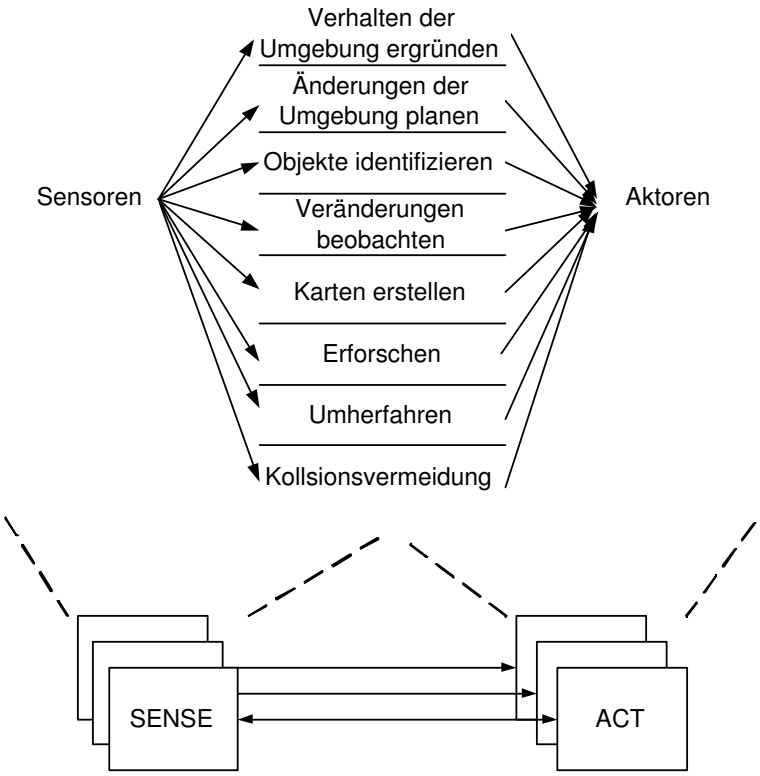

Abbildung 2.5: Vertikale Dekomposition (vgl. [4])

können lediglich wechselnde Reize zur Steuerung dieser Sequenz genutzt werden, weshalb die Entwicklung komplexer Verhalten sehr viel Zeit und Erfahrung erfordert. Erlernen, welche Verhalten in welchen Situationen zum Ziel führen, kann der Roboter nicht, da er kein Weltmodell besitzt.

2.3 Hybrid deliberativer Ansatz

Der letzte Ansatz, der hier vorgestellt werden soll, dominiert seit Anfang der 90er Jahre die Entwicklung von Steuerungsarchitekturen. Beim hybrid deliberativen Ansatz wird die enge Kopplung von SENSE und ACT wie im reaktiven Ansatz beibehalten. Zusätzlich wird wieder eine Planungskomponente (deliberative Komponente) eingeführt, die über den reaktiven Verhalten steht. Diese Planungskomponente wird wie beim hierarchischen Ansatz für die Erstellung eines Weltmodells und Ausarbeitung eines Plans zur Aufgabenerfüllung eingesetzt. Allerdings plant sie nicht wie im hierarchischen Ansatz jede kleinste Aktion, sondern wählt die Verhalten aus, die nötig sind, um ein bestimmtes Ziel zu erreichen. Bei komplexen Aufgabenstellungen zerlegt die Planungskomponente die Aufgabe in Teilziele und aktiviert schrittweise die nötigen Verhalten, um von einem Ziel zum nächsten zu kommen. Die Planungskomponente hält in diesem Fall den

Roboter nicht ständig auf, da sie im Hintergrund ablaufen kann, während die eingesetzten reaktiven Verhalten schnell auf Umwelteinflüsse reagieren können.

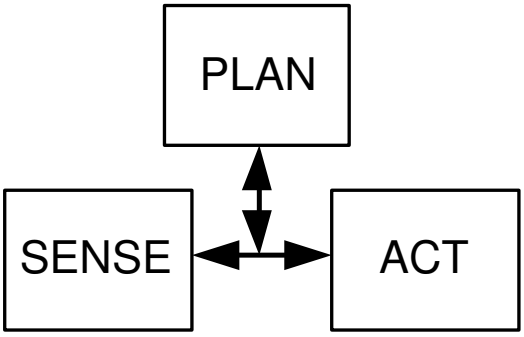

Abbildung 2.6: Hybrid deliberativer Ansatz

Saphira ist eine hybrid deliberative Steuerungsarchitektur und wurde am SRI in einer Gruppe um Kurt Konolige entwickelt [6, 7]. Die Architektur wurde unter anderem auf dem Roboter Flakey eingesetzt und gewann 1992 und 1993 Preise bei den AAAI Robot Competitions (siehe: http://www.ai.sri.com/timeline/). Saphira ist aus einer deliberativen Komponente und einer reaktiven Komponente zusammengesetzt. Ein Kernstück von Saphira ist der so genannte lokale Wahrnehmungsraum LPS (engl. local perceptual space). In ihm werden alle Sensordaten zu einem Bild der Umgebung zusammengeführt. Der LPS ist eine Schnittstelle zwischen der deliberativen und der reaktiven Komponente, siehe Abbildung 2.7, und kann mehrere Abstraktionsebenen besitzen. Beispielsweise kann eine sehr einfache Variante des LPS konstruiert werden, in den nur die Position des Roboters und die gemessenen Sonarwerte eingehen. Eine komplexere Variante könnte zusätzlich die Daten einer vorher bekannten Karte einbringen. Die Verhalten in der reaktiven Schicht arbeiten entweder direkt auf den Sensordaten oder auf den abstrakten Daten des LPS. Die Planungskomponente PRS-Lite ist ein Logikinterpreter und erfüllt die deliberativen Steuerungsaufgaben. Hier werden komplexe Aufgaben in einfachere Teilaufgaben zerlegt und deren Ausführung überwacht. Beispielsweise lässt sich die Aufgabe „Finde eine bestimmte Person in diesem Gebäude“, vereinfachen zu „Gehe in jeden Raum und überprüfe jeweils, ob die Person anwesend ist“. Der erste Raum, der aufgesucht werden soll, wird als Ziel zur Bahnplanung an den topologischen Planer gegeben und von diesem wieder in Teilaufgaben zerlegt. Diese werden von reaktiven Verhalten umgesetzt. Auf der reaktiven Ebene können sich beispielsweise zielgerichtete und kollisionsvermeidende Verhalten ergänzen. Die verschiedenen, von den Verhalten vorgeschlagenen Aktionen werden mittels Fuzzy-Logik koordiniert, wodurch ein kontextsensitives Überblenden zwischen den einzelnen Verhalten möglich wird. Die Steuerbefehle des so ermittelten Gesamtverhaltens werden an die Aktoren weitergegeben. Durch die Software-Agenten (rechts in Abbildung 2.7) ist es möglich weitere Fähigkeiten in die Architektur zu integrieren. Saphira wurde seit Beginn der 90er

weiterentwickelt und ist heute noch aktuell. Die Architektur wird von ActivMedia vertrieben und ist für Roboter wie den Pioneer 2DX erhältlich. Allerdings ist Saphira auf dem Pioneer des Fachbereiches aufgrund der bisher durchgeführten Modifikationen nicht mehr einsetzbar, so dass eine neue Steuerungsarchitektur entwickelt werden muss. An dieser Stelle sei erwähnt, dass es seit der Version 8.x von Saphira eine Zweiteilung der Architektur in ARIA, welche die Grundlagen der Steuerungsarchitektur implementiert, und Saphira, für anspruchsvollere Aufgaben, gibt.

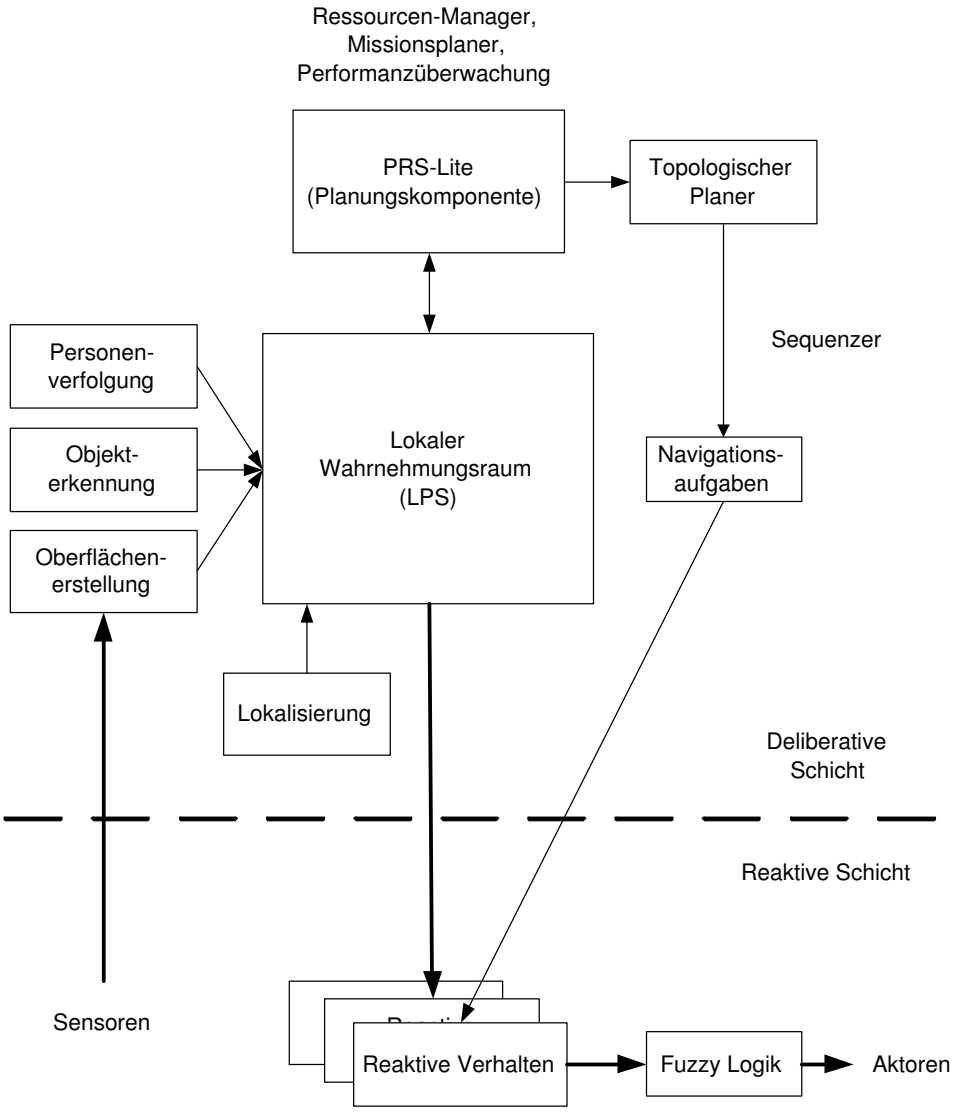

Abbildung 2.7: Saphira - vereinfachtes Modell (vgl. [8] S.279)

Kapitel 3

HydrA - Eine hybrid deliberative Architektur

Nachdem ein Grundwissen über Steuerungsarchitekturen vermittelt worden ist, wird die Steuerungsarchitektur HydrA (**Hy**bird **d**elibe**r**ative **A**rchitektur) vorgestellt, die im Rahmen dieser Arbeit für den modifizierten Pioneer entwickelt wird. Die neue Architektur entspricht dem hybrid deliberativen Ansatz und ist ähnlich wie ARIA aufgebaut (vgl. [1], z.B. ARIA Overview). Die Architektur besteht im Wesentlichen aus einem reaktiven Steuerungszyklus, der eine verhaltensbasierte Steuerung des Roboters erlaubt. Hierfür sind Komponenten zur Realisierung von Sensorinterpretation und Verhalten implementiert worden, die einerseits als Grundlage für eine Weiterentwicklung der Architektur dienen, andererseits aber auch die Funktionalität der Architektur demonstrieren und dem Roboter autonomes Agieren ermöglichen. Zudem ist eine Schnittstelle für eine deliberative Komponente vorhanden, welche die Steuerung der reaktiven Schicht ermöglicht (siehe Abbildung 3.1). Auf die Implementierung einer deliberativen Komponente muss in dieser Arbeit verzichtet werden, da dies den Rahmen übersteigen würde. Im Folgenden wird der Aufbau der realisierten Architektur beschrieben und die Arbeitsweise der Komponenten erklärt.

3.1 Reaktive Steuerung

Die reaktive Steuerung erfolgt durch den in Abbildung 3.1 beschrieben Zyklus. Die Begriffsbildung erfolgt hierbei in Anlehnung an ARIA. Die Sensordaten gelangen durch eine Schnittstelle zum Roboter in den Steuerungszyklus. Bevor die Daten weiterbearbeitet werden, werden sie im Zustandsreflektor gespeichert. Dieser stellt die unbearbeiteten Sensordaten den Komponenten zur Sensordateninterpretation zur Verfügung. Diese können durch Sensorfusion beziehungsweise -interpretation aus den Rohdaten abstrakte Sensordaten erstellen. Die Resultate dieser Komponenten werden den Verhalten zu Verfügung gestellt. Der Block,

der die Verhalten repräsentiert, wird in Abbildung 3.8 weiter aufgegliedert. Die Verhalten ermitteln Anweisungen für die Steuerung des Roboters. Diese Steuerbefehle werden in einem Speicher gesammelt, bis sie zur Ausführung an den Roboter weitergegeben werden. Die Zeitspanne, die für diesen Zyklus zur Verfügung steht, ist konfigurierbar. In der aktuellen Konfiguration beträgt sie 50ms. Wenn der Roboter sich mit seiner maximalen Geschwindigkeit von ca. 2m/s fortbewegt, bedeutet dies, dass er eine Strecke von ca. 10cm zurücklegt, bis er die nächsten Sensordaten auswertet. Die Architektur ist so entwickelt, dass alle Daten, die auf der Seite des Roboters ermittelt werden, solange gespeichert werden, bis sie in den Steuerungszyklus übergeben werden. Dies ermöglicht, die Dauer des Steuerungszyklus unabhängig von den Aktualisierungsraten der Sensoren zu wählen. Wird ein größeres Intervall für die Zyklusdauer eingesetzt, beispielsweise 100ms, dauert es zwar länger bis der Roboter auf Änderungen in der Umgebung reagiert, es gehen aber keine Daten verloren.

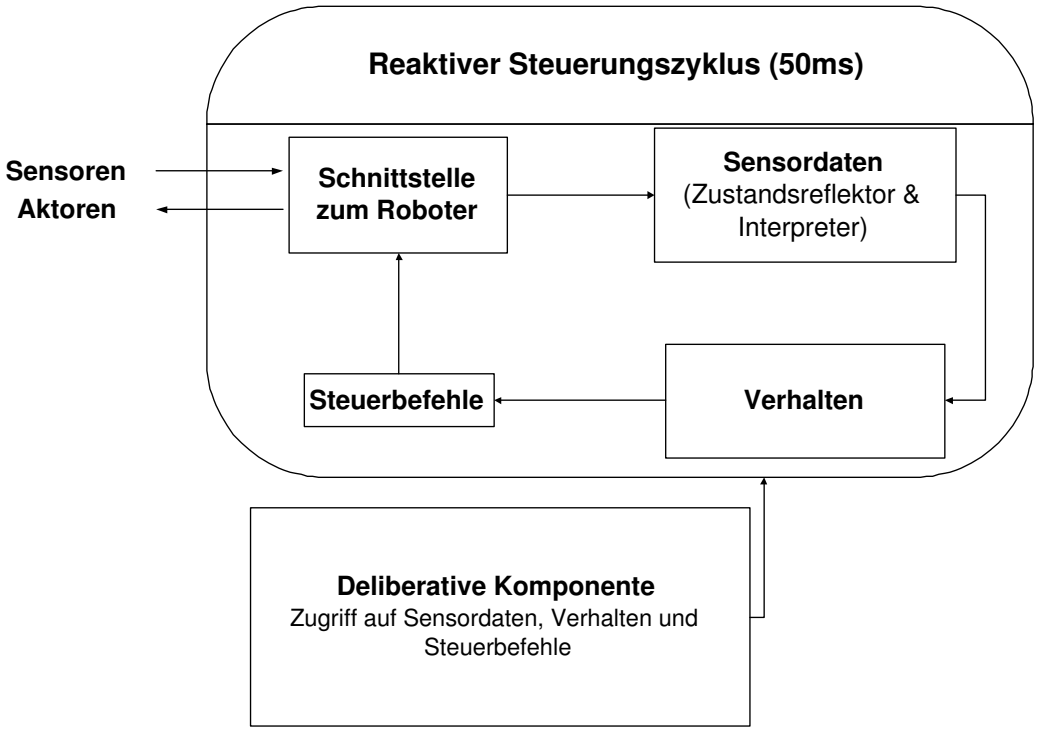

Abbildung 3.1: Realisierte Architektur im Überblick

3.2 Sensordatenverarbeitung

Der erste Schritt in jedem Steuerungszyklus ist das Erfassen und Aufbereiten der Sensordaten. Das Konzept, das der Verarbeitung der Sensordaten zugrunde liegt, wird im Folgenden erklärt.

3.2.1 Sensordatenerfassung

Zum Sammeln der Sensordaten gibt es zwei Strukturen - den Zustandsreflektor und den Ereigniscontainer. Der Zustandsreflektor dient in erster Linie zum Sammeln aller unbearbeiteten Sensordaten, die der Roboter regelmäßig erfasst. Im aktuellen Zustand des Pioneer sind die Daten folgender Sensoren verfügbar. Die Aktualisierungsraten ergeben sich aus [14] und stehen in Klammern.

- 2D Beschleunigungssensoren (40 ms)
- Gyroskop (40 ms)
- 16 Sonare (alle 40 ms je 2 Sonare)
- Radencoder (40 ms)
- Batteriespannung (40 ms)
- Temperatur (40 ms)

Dabei sind für alle, bis auf die letzten beiden Messwerte Zeitstempel verfügbar, die hardwarenah zum Zeitpunkt der Aufnahme gemacht werden. Die Daten des Zustandsreflektors sollen den letzten verfügbaren Stand der Sensordaten wiedergeben. Neben den Sensordaten liefert der Roboter Ereignisse, die in unregelmäßigen Abständen auftreten. Beispielsweise löst der Greifer ein Ereignis aus, wenn er seine vorgegebene Position erreicht oder die Lichtschranke unterbrochen wird. Diese Ereignisse, werden in einem Container gesammelt und können von den Verhalten konsumiert werden. Wenn ein Verhalten ein Ereignis konsumiert, wird es nach Ablauf dieses Steuerungszyklus aus dem Container entfernt, ansonsten wird es erst entfernt, nachdem seine maximale Lebensdauer (im Moment 10 Zyklen) abgelaufen ist. Welche Ereignisse es gibt und wann sie ausgelöst werden, kann in [14] nachgelesen werden. Für Navigationsaufgaben spielen sie im Moment keine Rolle.

3.2.2 Sensorinterpretation

Die Interpretation beziehungsweise die Fusion von Sensordaten ist ein wichtiger Aspekt einer stabilen Steuerung. Erst durch das Zusammenführen der Informationen mehrerer Sensoren kann ein zuverlässiges Bild der Umgebung erstellt werden, da es so möglich ist, die Schwächen einzelner Sensoren, beispielsweise die Ungenauigkeit der Sonare oder der Radencoder, auszugleichen. Die Komponenten zur Sensorinterpretation beziehungsweise -fusion, werden im Weiteren kurz als Interpreter bezeichnet. Neben dem Verarbeiten von Sensordaten, die von am Roboter vorhandenen Sensoren gemessen werden, ermöglichen die Interpreter die Integration abstrakter Sensordaten in die Architektur. Abstrakte Sensordaten sind keinem Sensor direkt zuzuordnen, sondern können beispielsweise durch Fusion

mehrerer Sensoren ermittelt werden. Die Position kann als abstraktes Sensordatum betrachtet werden. Sie ist an keinen Sensor direkt gebunden und kann auf unterschiedliche Art und Weise, mit Hilfe eines oder mehrerer Sensoren, bestimmt werden. Ein weiterer Vorteil abstrakter Sensordaten liegt in der Möglichkeit des Einbringens beliebiger Informationen des Weltmodells. Ein Verhalten, das zur Kollisionsvermeidung auf Sonarwerte reagiert, ist einerseits davon abhängig, dass der Roboter über Sonare verfügt, andererseits davon, dass die Sonare die Hindernisse erkennen. Durch Verwendung eines abstrakten Sensors für Hindernisse ist das Verhalten nicht mehr an die Sonare gebunden und kann beispielsweise auch auf Hindernisse reagieren, die aus einer vorher bekannten Karte der Umgebung extrahiert werden (siehe Abbildung 3.2).
Das Konzept für die Interpreter ist modular aufgebaut und für die Integration weiterer Komponenten ausgelegt. Darüber hinaus ist es möglich, je nach Aufgabenstellung, zur Laufzeit Interpreter zum Steuerungszyklus hinzuzufügen oder aus ihm zu entfernen. Jeder Interpreter der Teil des Steuerungszyklus ist, wird nach dem Aktualisieren der Daten des Zustandsreflektors ein Mal ausgeführt.

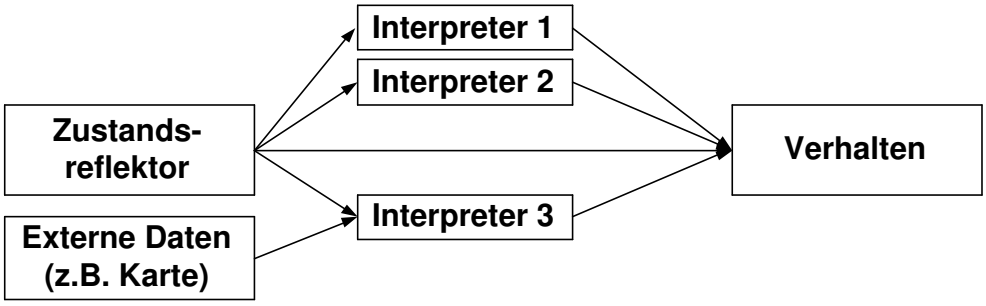

Abbildung 3.2: Interpreter

3.2.3 Entwickelte Interpreter

Im Weiteren werden die entwickelten Komponenten zur Sensorinterpretation mit ihrer Funktionalität vorgestellt.

Position-Interpreter

Der Position-Interpreter bestimmt die Position einschließlich der Orientierung des Roboters. Zusätzlich errechnet er die lineare Geschwindigkeit V und die Rotationsgeschwindigkeit ω. Das Vorgehen wird in [12] im Kapitel 4.5 beschrieben. Grundlage für diese Berechnung sind die Streckenzählung der Radencoder und die Zeitstempel dieser Sensordaten. Die Ticks des rechten und linken Rades, $ticks_r$ und $ticks_l$, um die sich die Räder im Zeitintervall Δt gedreht haben, können aus den Sensordaten der Radencoder ermittelt werden. Die Faktoren zur Um-

rechnung von Ticks in mm, $const_r$ (rechts) und $const_l$ (links), können für beide Räder getrennt festgelegt werden.
In Abbildung 3.3 beschreibt das Tupel (x, y) die Position und θ die Orientierung des Roboters bezüglich des Weltkoordinatensystems $\mathcal{K}_W$. Dabei liegt (x, y) in der Mitte der Antriebsräder des Roboters. Der Abstand der Räder ist d. Die Geschwindigkeiten des rechten bzw. linken Rades werden durch v_r bzw. v_l beschrieben. V steht für die Geschwindigkeit des Roboters. Unter der Annahme, dass v_r und v_l konstant sind und $v_l \neq v_r$ ist, bewegt sich der Roboter auf einer Kreisbahn mit konstanter Rotationsgeschwindigkeit ω. ICC beschreibt das Zentrum der Drehung und r den Radius der Kreisbahn, die der Roboter abfährt.

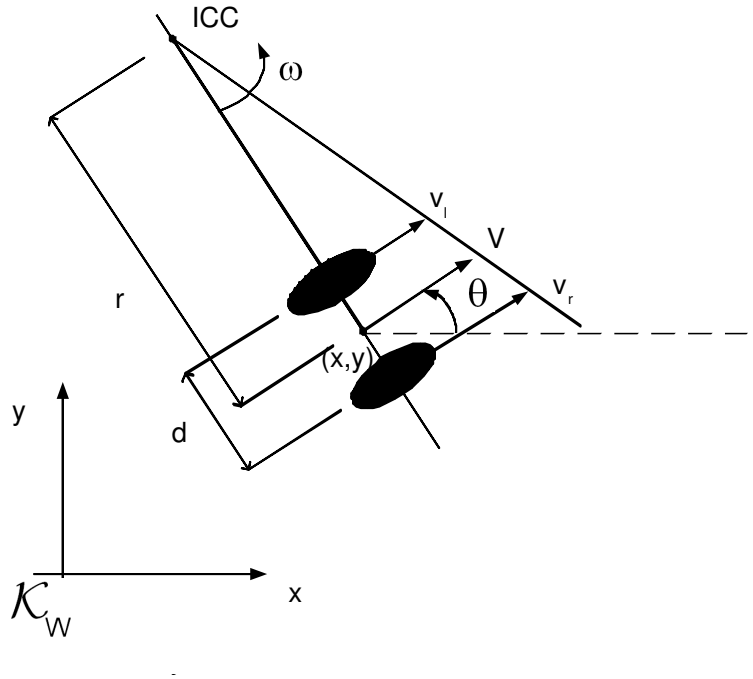

Abbildung 3.3: Geschwindigkeit eines dreirädrigen Radroboters (vgl. [12] Kapitel 4.5)

Mit Hilfe der obigen Annahme können die Geschwindigkeiten für den Zeitraum Δt folgendermaßen bestimmt werden:

$$v_l = const_l \cdot ticks_l/\Delta t \text{ und } v_r = const_r \cdot ticks_r/\Delta t \tag{3.1}$$

$$V = 1/2 * (v_l + v_r) \tag{3.2}$$

$$\omega = 1/d * (v_r - v_l) \tag{3.3}$$

Sei die Position des Roboters $S(t_0) = (x_0, y_0, \theta_0)$ bekannt und bewege sich der Roboter im Intervall $\Delta t = t_1 - t_0$ (mit $t_1 > t_0$) mit einer linearen Geschwindigkeit $V(t)$ und einer Rotationsgeschwindigkeit $\omega(t)$, so gilt für die Position $S(t_1) = (x_1, y_1, \theta_1)$.

$$x_1 = x_0 + \int_{t_0}^{t_1} V(\tau) \cdot \cos(\theta(\tau)) d\tau \tag{3.4}$$

$$y_1 = y_0 + \int_{t_0}^{t_1} V(\tau) \cdot \sin(\theta(\tau)) d\tau \tag{3.5}$$

$$\theta_1 = \theta_0 + \int_{t_0}^{t_1} \omega(\tau) d\tau \tag{3.6}$$

Mit der Annahme, dass $V(t)$ und $\omega(t)$ in betrachteten Zeitraum konstant sind, also $V(t) = V$ und $\omega(t) = \omega$, folgt:

$$x_1 = \begin{cases} x_0 + V \cdot \cos(\theta_0) \cdot \Delta t & \text{falls } \omega = 0 \\ x_0 + V/\omega \cdot (\sin(\theta_1) - \sin(\theta_0)) & \text{sonst} \end{cases} \tag{3.7}$$

$$y_1 = \begin{cases} y_0 + V \cdot \sin(\theta_0) \cdot \Delta t & \text{falls } \omega = 0 \\ y_0 + V/\omega \cdot (-\cos(\theta_1) + \cos(\theta_0)) & \text{sonst} \end{cases} \tag{3.8}$$

$$\theta_1 = \theta_0 + \omega \cdot \Delta t \tag{3.9}$$

Mit dem beschriebenen Verfahren ist es theoretisch möglich, die Position des Roboters zu verfolgen, wenn die zugrunde liegenden Annahmen gelten. In der Praxis ist festzustellen, dass der Roboter, wenn er geradeaus fährt oder auf der Stelle dreht, seine Position sehr gut bestimmen kann. Die Abweichung beim Drehen auf der Stelle beträgt bei fünf Umdrehungen ca. 1°. Wird der Pioneer auf gerader Strecke über ebenen Boden geschoben, so ergeben sich nach ca. 18m Fahrt folgende Messdaten:

	Position (x , y, theta) [mm,mm,°]
wirkliche Position	(18230,0,0)
vom Pioneer ermittelte Position	(18036,-363,-1)
Differenz	(194,-363,-1)

Abbildung 3.4 dient als Visualisierung des Wegs dieser Testfahrt. Die Abbildung ist mit Hilfe des Moduls zur Kartenerstellung von Gerhard Rohe erstellt worden (vgl. [9]). Der Weg, der tatsächlich zurückgelegt worden ist, wird durch die grüne Linie repräsentiert. Der Weg, der sich aus den Berechnungen des Interpreters

ergibt, ist rot eingezeichnet. Die Punkte links und rechts des Roboters stellen durch die Sonare ermittelte Hindernisse dar. Ein Kästchen in der Karte hat jeweils die Kantenlänge 1m. Der Roboter ist von links nach rechts bewegt worden. Aus der Abbildung wird ersichtlich, wie die Abweichung zwischen tatsächlicher und errechneter Position langsam wächst.

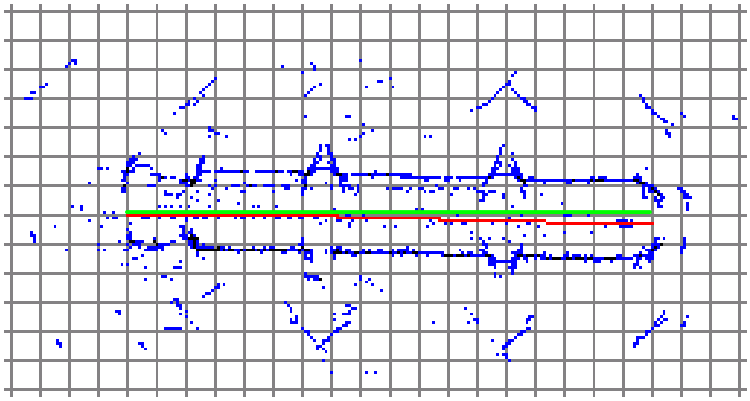

Abbildung 3.4: Fahrt geradeaus (geschoben)

Bei kurvenreichen Fahrten über wenige Meter wächst der Unterschied zwischen der realen Position und der mit diesem Verfahren ermittelten schneller. In einer Testfahrt sind folgende Ergebnisse ermittelt worden:

	Position (x , y, theta) [mm,mm,°]
wirkliche Position	(16000,+100, ca. -30)
vom Pioneer ermittelte Position	(16887,-1362,-35)
Differenz	(887,-1462, ca. -5°)

Die Karte (siehe Abbildung 3.5) zeigt den Weg, der bei dieser Testfahrt von links nach rechts aus den Daten des Interpreters ermittelt worden ist, in rot. Die grüne Linie ist von Hand eingefügt worden und beschreibt ungefähr die Bahn, die der Roboter tatsächlich zurückgelegt hat. Das Anwachsen der Abweichung zwischen interner Repräsentation und Realität ist deutlich zu erkennen.
Gründe hierfür können darin gefunden werden, dass die Annahme der konstanten Radgeschwindigkeiten falsch ist, oder darin, dass gelegentlich ein Rad etwas durchdreht oder blockiert und damit die Ticks verfälscht. Die Probleme auf diesem Gebiet sind im Rahmen dieser Arbeit nicht weiter verfolgt worden, da Andreas Grunewald sich in seiner Diplomarbeit mit Verbesserungsmöglichkeiten für die Positionsbestimmung beschäftigt [5].
An dieser Stelle sei noch bemerkt, dass der Position-Interpreter die Positionsdaten über die letzten 800ms speichert, um diese für andere Module zur Verfügung zu

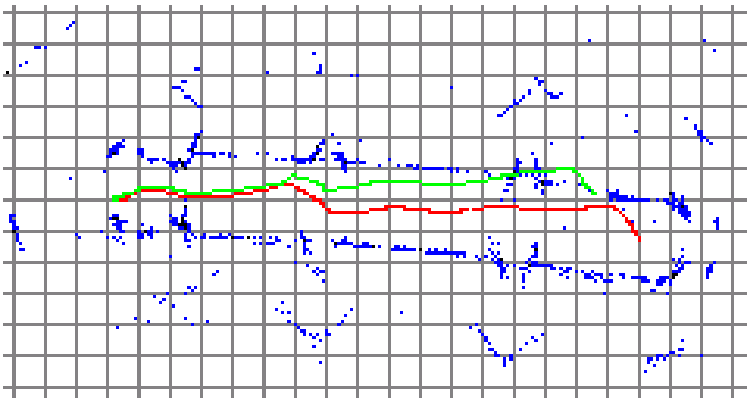

Abbildung 3.5: Freie Fahrt

stellen. Mit jeder Position wird außerdem ein Zeitstempel gespeichert, der dem Zeitstempel der Radticks entspricht, aus denen die Position berechnet worden ist.

Sonar-Interpreter

Der Sonar-Interpreter arbeiten auf den Sensordaten der Sonare. Sei d die Distanz, welche ein Sonar gemessen hat, so berechnet der Sonar-Interpreter aus dieser Distanz d, der Position eines Sonars S und dem Abstrahlwinkel α, die Position des Reflexionspunktes H (siehe Abbildung 3.6). Unter der Annahmen, dass die Messung korrekt ist, entspricht H dem Punkt, an dem der Schall des Sonars reflektiert worden ist. Der Sonar-Interpreter ergänzt die Sonardaten um die absoluten Koordinaten H_W des Reflexionspunktes. Es wird davon ausgegangen, dass die Positionen der Sonare am Roboter und deren Abstrahlwinkel konstant sind. Diese sind in einer Konfigurationsdatei gespeichert und können so bei Änderungen der Hardware angepasst werden.
Die Koordinaten des Roboters R_W, die zur Berechnung von H_W erforderlich sind, werden mit Hilfe des Position-Interpreter ermittelt. Da der Position-Interpreter Zugriff auf die berechneten Positionen der letzten 800ms ermöglicht, können für R_W die Koordinaten der Position des Roboters bestimmt werden, deren Zeitstempel dem der Sonardaten am nächsten ist. Wird davon ausgegangen, dass die Positionsdaten alle 40ms aktualisiert werden, und die Sonardaten nicht älter als 800ms sind, so beträgt die maximale Differenz zwischen den beiden Zeitstempeln 20ms. Wird die Position des Roboters korrekt bestimmt, kann unter der Annahme, dass die maximale Geschwindigkeit des Roboters 2m/s beträgt, die Position von H bis auf 4cm genau bestimmt werden.
Die Berechnung von H_W soll mit Hilfe von Abbildung 3.6 erklärt werden. Sei R der Punkt, der den Roboter und den Ursprung des Koordinatensystems $\mathcal{K}_R$ repräsentiert. Weiter sei $\mathcal{K}_R$ so am Roboter verankert, dass die x-Achse des Koordinatensystems vom Roboter aus „vorwärts“ zeigt. In Abbildung 3.6 ist $\mathcal{K}_R$ ge-

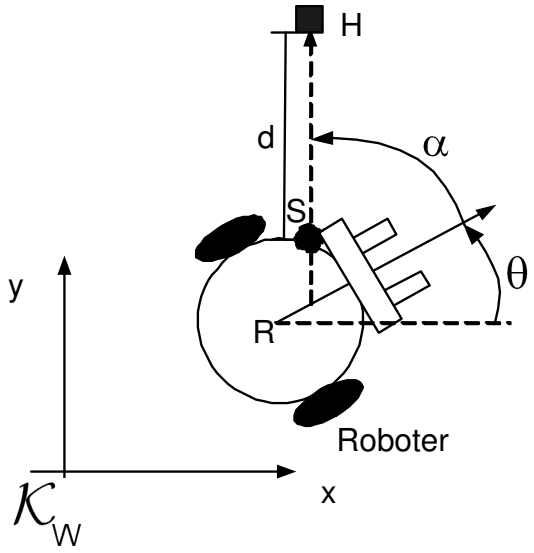

Abbildung 3.6: Bestimmung der Koordinaten von Sonarwerten

genüber $\mathcal{K}_W$ um θ ausgelenkt. Sind die Koordinaten eines Sonars S_R und dessen Abstrahlrichtung α bekannt und ist R_W die Position des Roboters zum Aufnahmezeitpunkt, so lässt sich H_W folgendermaßen bestimmen:

$$H_W^T = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \left(d \cdot \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix} + S_R^T \right) + R_W^T \tag{3.10}$$

Der Sonar-Interpreter hält für jedes Sonar die letzten verfügbaren Daten. Werden die für die einzelnen Sonarwerte berechneten absoluten Koordinaten der Reflexionspunkte dauerhaft aufgezeichnet, kann aus ihnen eine Karte der Umgebung erstellt werden. Zu den Ungenauigkeiten in dieser Karte kommen, neben der bereits erwähnten Unschärfe von 4cm, zum Einen Fehler die bereits bei der Berechnung der Position des Roboters gemacht werden, zum Anderen die Ungenauigkeit der Sonare. Gerhard Rohe beschäftigt sich zum Zeitpunkt der Erstellung dieser Diplomarbeit in seiner Studienarbeit [9] ausführlich mit der Erstellung von Karten. Das von ihm entwickelte Programm, wird auch als Interpreter in HydrA integriert.

Hindernis-Interpreter

Der letzte Interpreter, der im Rahmen dieser Arbeit entwickelt worden ist, ist der Hindernis-Interpreter. Er liefert Informationen über Hindernisse in der Umgebung des Roboters. Dazu benötigt er nur die absolute Position des Roboters R_W und die absoluten Koordinaten des Hindernisses H_W (siehe Abbildung 3.6). Bei jedem Aufruf durch den Steuerungszyklus erfolgt eine Neuberechnung der

relativen Koordinaten des Hindernisses H_R. Die relativen Koordinaten werden folgendermaßen berechnet:

$$H_R^T = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \cdot (H_W^T - R_W^T) \tag{3.11}$$

Die Verhalten haben so die Möglichkeit abzufragen, ob sich in bestimmten Bereichen um den Roboter Hindernisse befinden. Zum Einen kann abgefragt werden, ob sich Hindernisse in einem bestimmten Kreissektor befinden. Dieser besitzt R als Mittelpunkt und wird durch den Radius r und die Winkel α und β festgelegt (siehe Abbildung 3.7). Zum Anderen kann abgefragt werden, ob sich Hindernisse in einem bestimmten Rechteck befinden. Das Rechteck wird definiert durch R, Höhe, Breite und den Winkel γ (siehe Abbildung 3.7). Das Rechteck ist derart mit R verbunden, dass R die Seite, die es enthält, halbiert.

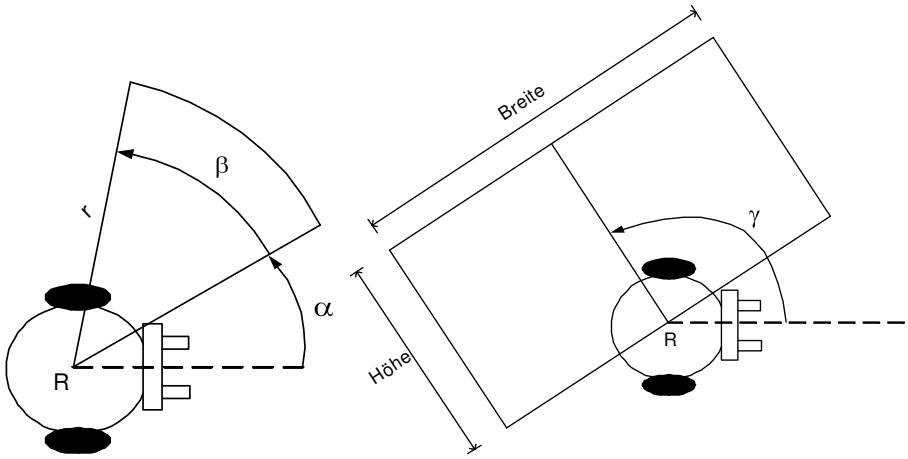

Abbildung 3.7: Hindernis-Interpreter

In der aktuellen Konfiguration bezieht der Hindernis-Interpreter die Daten, die er verwaltet, aus dem Sonar-Interpreter. Dabei wandelt er die Sensordaten der Sonare in abstrakte Sensordaten vom Typ Hindernis um. Die Hindernisse unterscheiden sich von den Daten des Sonars dadurch, dass sie zusätzlich ihre relative Position zum Roboter kennen. Der Hindernis-Interpreter speichert alle Hindernisse, die er aufgenommen hat, solange bis deren Zeitstempel älter als 5000ms sind. Dies ermöglicht, das Erstellen eines Bildes der Umgebung, das genauer ist als ein Bild, das nur die letzten Sonardaten heranzieht.
Ein wesentliches Merkmal dieses Interpreters ist, dass er nicht nur in der Lage ist, Sonardaten in Hindernisse umzuwandeln, sondern er kann auch abstrakte Hindernisse als Eingangsdaten verwenden. Diese Eigenschaft kann von Nutzen sein, wenn es einmal möglich ist, die Position des Roboters auf einer vorher erstellten Karte zu lokalisieren. Dann kann der Hindernis-Interpreter nicht nur Hindernisse, die aus den Sonaren ermittelt werden, zur Verfügung stellen, sondern auch

weitere Hindernisse, beispielsweise Wände, die auf der Karte eingezeichnet sind, aber gerade nicht von den Sonaren wahrgenommen werden.

3.3 Verhalten

Verhalten werden zur reaktiven Steuerung des Roboters eingesetzt. Jedes Verhalten wird in jedem Durchlauf des Steuerungszyklus nach den Interpretern einmal ausgeführt. Sie können auf alle Sensordaten zugreifen und schnell auf Veränderungen in der Umwelt reagieren. Der folgende Abschnitt erklärt, wie aus einzelnen Verhalten ein Gesamtverhalten gebildet wird. Danach werden die wesentlichen Eigenschaften beschrieben, die bei der Entwicklung von Verhalten zu berücksichtigen sind und wie es möglich ist Verhalten zu gruppieren. Außerdem werden die Verhalten, die als Teil dieser Arbeit entwickelt worden sind, vorgestellt.

Da in mehreren Beispielen die Verhalten „Move To“ und „Avoid Front“ herangezogen werden, soll deren grundlegende Funktionalität hier kurz erklärt werden. „Move To“ ist ein Verhalten, welches versucht den Roboter auf geradem Weg zu einen vorgegeben Ziel zu bringen. Es bestimmt die Richtung zum Ziel und die Geschwindigkeit, mit der sich der Roboter dem Ziel näheren soll. „Avoid Front“ soll vermeiden, dass der fahrende Roboter mit Hindernissen zusammenstößt. Taucht ein Hindernis vor dem Roboter auf, so schlägt es vor, nach links (90°) beziehungsweise rechts (-90°) wegzudrehen, je nachdem welche Seite weiter von dem Hindernis entfernt ist.

3.3.1 Verhaltenskoordination

Ein wichtiger Aspekt der Steuerungsarchitektur ist die Koordination der Verhalten. Es ist möglich, mehrere Verhalten sozusagen nebeneinander in dieser Architektur zu verwalten und durch Überlagerung ihrer Ergebnisse, oder präziser gesagt, ihrer vorgeschlagenen Aktionen, ein komplexes Gesamtverhalten zu erzeugen. Dies soll zunächst in einem Beispiel erklärt werden. Die beiden Verhalten „Move To“ und „Avoid Front“ sollen zusammen aktiv sein. Angenommen, „Move To“ schlägt, um auf ein vorgegebenes Ziel zuzufahren, eine Geschwindigkeit von 1m/s und eine relative Richtungsänderung von 0° vor (Geradeausfahrt). „Avoid Front“ beeinflusst die Geschwindigkeit nicht weiter, fordert aber eine relative Richtungsänderung von 90°, um einem Hindernis auszuweichen. Jetzt gilt es beide Verhalten zu koordinieren, damit ein sinnvolles Gesamtverhalten entsteht. In diesem Fall wäre es möglich, den Roboter mit einer Geschwindigkeit von 1m/s weiterfahren zu lassen und gleichzeitig mit 90°/s zu drehen. Die Abbildung 3.8 zeigt grafisch wie der Algorithmus zur Verhaltenskoordination aufgebaut ist. Die Bestandteile der Abbildung werden im Weiteren genauer beschrieben.

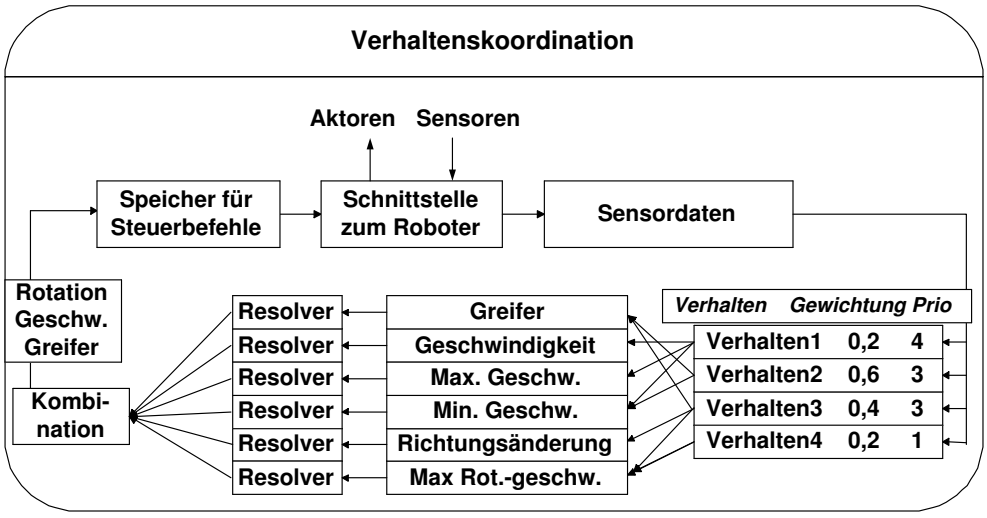

Abbildung 3.8: Verhaltenskoordination

Kanäle

Jedes Verhalten kann für folgende sechs Kanäle Aktionen vorschlagen, die Einheiten stehen in Klammern:

- Geschwindigkeit (in mm/s)
- relative Richtungsänderung (in °)
- maximale Geschwindigkeit (in mm/s)
- minimale Geschwindigkeit (in mms/s)
- maximale Rotationsgeschwindigkeit (in mm/s)
- Greifer (Steuerbefehle)

Die ersten fünf Kanäle sind in Anlehnung an ARIA gewählt und basieren auf folgendem Konzept. Die Verhalten, die den Roboter aktiv steuern, beispielsweise „Move To“ oder „Avoid Front“, nutzen hauptsächlich die Kanäle für Geschwindigkeit und relative Richtungsänderung. Passive Verhalten, die den Roboter nicht selbst bewegen, aber seine Bewegungsmöglichkeiten einschränken, nutzen in erster Linie die Kanäle für maximale und minimale Geschwindigkeiten. Ein Beispiel wäre ein Verhalten, das die Geschwindigkeit in Abhängigkeit von Hindernissen in der näheren Umgebung des Roboters begrenzt.
Da zur Steuerung des Roboters Geschwindigkeiten an die Roboter-API weitergegeben werden, wird die Richtungsänderung beim Kombinieren der Kanäle in eine Rotation umgesetzt. Auf den ersten Blick ist nicht offensichtlich, weshalb die Verhalten nicht direkt eine Rotation (in °/s) statt einer Richtungsänderung (in °)

vorschlagen. Unter Einbeziehung der Überlegung, wie ein Verhalten wie „Move To“ aufgebaut ist, scheint es intuitiver zu sein, dass dieses Verhalten eine relative Richtungsänderung vorschlägt. Angenommen, das Verhalten kennt die aktuelle Position einschließlich der Richtung, in die der Roboter steht, und die Position des Zieles. Dann ist es leicht, Richtung und Abstand des Ziels zu bestimmen und somit eine Richtungsänderung um einen bestimmten Winkel vorzuschlagen. Die Rotation, die letztlich beim Kombinieren der Kanäle aus der Richtungsänderung errechnet wird, ist nur von dieser abhängig. Die Werte hierfür sind experimentell ermittelt worden (siehe Tabelle 3.1). Die Geschwindigkeit hingegen kann jedes Verhalten direkt vorschlagen. Dies ermöglicht den Verhalten, die Geschwindigkeit in Abhängigkeit von mehreren Parametern, beispielsweise der Entfernung und der Richtung zum Ziel, festzulegen.

Betrag der relative Richtung α in $^\circ$	Betrag der Rotation
$0^\circ < \alpha \leq 10^\circ$	$10^\circ/\mathrm{s}$
$10^\circ < \alpha \leq 20^\circ$	$20^\circ/\mathrm{s}$
$20^\circ < \alpha \leq 40^\circ$	$40^\circ/\mathrm{s}$
$40^\circ < \alpha \leq 60^\circ$	$60^\circ/\mathrm{s}$
$60^\circ < \alpha$	$90^\circ/\mathrm{s}$

Tabelle 3.1: Umsetzung relative Richtung in Rotation

Ein Unterschied zu ARIA ist der Kanal für den Greifer. Dieser Kanal ermöglicht es, Verhalten unterschiedliche Steuerbefehle für den Greifer vorzuschlagen, da diese nicht direkt an die Motorsteuerung des Greifers weitergeleitet, sondern koordiniert werden. Ein weiteres Merkmal ist, dass die Kanäle so entwickelt sind, dass problemlos weitere Kanäle hinzugefügt werden können. Sollen Kanäle entfernt werden, so ist dies auch möglich. Jedoch ist zu bedenken, dass die bereits entwickelten Verhalten dann eventuell nicht mehr genutzt werden können, beziehungsweise angepasst werden müssen, da sie Steueranweisungen für die beschriebenen Kanäle liefern.

Resolver

Die Resolver werden eingesetzt, um Konflikte zu lösen, die entstehen, wenn auf einem Kanal von mehreren Verhalten unterschiedliche Werte, beispielsweise unterschiedliche Geschwindigkeiten, vorgeschlagen werden. Jeder Kanal besitzt einen eigenen Resolver. Dieser ist unabhängig von den Resolvern der anderen Kanäle und kennt auch die Vorgänge auf den anderen Kanälen nicht. Da abhängig von der Funktion eines Kanals unterschiedliche Methoden zur Konfliktlösung sinnvoll erscheinen, werden unterschiedliche Resolver entwickelt. Im Fall, dass für einen Kanal keine Aktionen vorgeschlagen sind, wird der Resolver des entsprechenden

Kanals nicht ausgeführt. Im Folgenden wird davon ausgegangen, dass von den Verhalten für jeden Kanal mindestens eine Aktion vorgeschlagen wird.

Minimum- und Maximumresolver

Als relativ einfache Konfliktlösungsmechanismen sind Minimum- und Maximumresolver entwickelt worden. Der jeweilige Resolver gibt als Kompromiss unter allen vorgeschlagen Werten den Kleinsten beziehungsweise den Größten zurück. Dies ist vor allem sinnvoll für die Kanäle, die über Maximalgeschwindigkeiten bestimmen. Wollen mehrere Verhalten die maximale Geschwindigkeit des Roboters nach unterschiedlichen Kriterien, beispielsweise dem Vorhandensein von Hindernissen in der Nähe des Roboters oder der aktuellen Rotationsgeschwindigkeit, festlegen, so ist das Minimum dieser Werte ein geeigneter Kompromiss.

Prioritäts- und gewichtungsgesteuerter Resolver

Ein weiterer Resolver ist prioritäts- und gewichtungsgesteuert und wird in Anlehnung an ARIA entwickelt (vgl. [1]). Für diesen Resolver muss jedes Verhalten seine vorgeschlagenen Aktionen mit einer Priorität und einer Gewichtung versehen. Als Wertebereich für die Prioritäten sind ganze Zahlen größer null vorgesehen. Aktionen mit höherer Priorität werden bevorzugt. Der Wertebereich für die Gewichtungen sind reelle Zahlen im Intervall $[0, 1]$. Die Gewichtung gibt an, wie stark eine Aktion das Gesamtverhalten beeinflussen soll. Ist beispielsweise eine Aktion mit höchster Priorität und der Gewichtung 1.0 vorgeschlagen, so wird genau diese Aktion ausgeführt. Die Arbeitsweise dieses Resolvers kann folgendermaßen beschrieben werden.
Der Resolver erhält eine nicht leere, endliche Menge an vorgeschlagenen Aktionen $\mathcal{A}$, wobei jede vorgeschlagene Aktion $A \in \mathcal{A}$ aus einem 3-Tupel (p_A, u_A, w_A) besteht, in dem $p_A \in \mathbb{N}\backslash\{0\}$ für die Priorität, $u_A \in [0, 1]$ für die Gewichtung und $w_A \in \mathbb{Z}$ für den vorgeschlagenen Wert dieser Aktion steht. Sei $\mathcal{P}$ eine Menge von Prioritäten mit $\mathcal{P} = \{p \mid \exists p = p_A, A \in \mathcal{A}\}$ und $\mathcal{A}_p = \{A \in \mathcal{A} \mid p_A = p\}$. Für jede vorkommende Priorität $p \in \mathcal{P}$ wird die durchschnittliche Gewichtung U_p und die gewichtete Summe der Werte W_p berechnet.

$$U_p = \sum_{A \in \mathcal{A}_p} u_A / |\mathcal{A}_p| \qquad \text{für } \forall p \in \mathcal{P} \tag{3.12}$$

$$W_p = \sum_{A \in \mathcal{A}_p} (w_A \cdot u_A) / \sum_{A \in \mathcal{A}_p} u_A \qquad \text{für } \forall p \in \mathcal{P} \tag{3.13}$$

Nun werden die Aktionen $A_p = (p, U_p, W_p)$, die für alle Prioritäten ermittelt werden, zusammengefasst zur Aktion $A' = (p_{low}, U', W')$ mit $p_{low} = \max\{p \in \mathcal{P} \mid U' <= 1.0\}$. Sei $p_{max} = \max\{p \in \mathcal{P}\}$, so lassen sich U' und W' festlegen mit:

$$U' = \sum_{i=p_{low}, i \in \mathcal{P}}^{p_{max}} U_i \tag{3.14}$$

$$W' = \sum_{i=p_{low}, i \in \mathcal{P}}^{p_{max}} W_i \cdot U_i \tag{3.15}$$

Begonnen bei der höchsten Priorität gehen die Aktionen A_p, solange in A' ein, wie gilt $U' \leq 1$. Um so viele Aktionen wie möglich im Ergebnis zu berücksichtigen, wird, falls noch Aktionen vorhanden sind, die nicht in A' berücksichtigt sind und $U' < 1$ gilt, die Aktion $A_{p_{next}} = (p_{next}, U_{p_{next}}, W_{p_{next}})$, mit $p_{next} = \max\{p \in \mathcal{P} \mid p < p_{low}\}$, in das Ergebnis einbezogen. Da aufgrund des bisherigen Vorgehens $U_{p_{next}} > 1 - U'$ gilt, wird der Wert $W_{p_{next}}$ skaliert.

$$W = \begin{cases} W'/U' & \text{falls } U' = 1 \text{ oder } p_{low} = \min\{p \in \mathcal{P}\} \\ W' + W_{p_{next}} \cdot (1 - U') & \text{sonst} \end{cases} \tag{3.16}$$

Zurückgegeben wird dann vom Resolver eine Aktion A_{erg} mit

$$A_{erg} = \begin{cases} (p_{low}, U', W) & \text{falls } U' = 1 \text{ oder } p_{low} = \min\{p \in \mathcal{P}\} \\ (p_{next}, 1, W) & \text{sonst} \end{cases} \tag{3.17}$$

Ein Beispiel soll diesen Vorgang verdeutlichen. Angenommen der Resolver wird mit folgenden Werten gestartet:

Aktion	Priorität	Gewichtung	Wert
Akt 5A	5	0.75	400
Akt 5B	5	0.25	160
Akt 4A	4	0.3	0
Akt 2A	2	1.0	100
Akt 2B	2	0.25	200
Akt 1A	1	0.5	100

Schritt 1: Zusammenfassen der Aktionen gleicher Priorität

Aktion	Priorität	Gewichtung	Wert
Akt 5	5	0.5	340
Akt 4	4	0.3	0
Akt 2	2	0.625	120
Akt 1	1	0.5	100

Schritt 2: Aufaddieren der Gewichtungen der neuen Aktionen (prioritätsübergreifend), solange die gemeinsame Gewichtung kleiner oder gleich 1.0 ist

Aktion	Priorität	Gewichtung	Wert
Akt 45	4	0.8	170
Akt 2	2	0.625	120
Akt 1	1	0.5	100

Schritt 3: Auffüllen der Gewichtung auf 1.0

Aktion	Priorität	Gewichtung	Wert
Akt 245	2	1.0	194
Akt 1	1	0.5	100

Die Aktion A_{erg}, die letztendlich vom Resolver zurückgegeben wird, ist $A_{erg} = (2, 1.0, 194)$.
Dieser Resolver wird eingesetzt für die Kanäle für Geschwindigkeit und relative Richtung. In dem zu Beginn des Abschnitts 3.3.1 eingeführten Beispiel, in welchen „Move To“ eine Richtungsänderung um 0° vorschlägt, und „Avoid Front“ eine Richtungsänderung um 90°, ist das Gesamtverhalten davon abhängig, welche Priorität und Gewichtung die vorgeschlagenen Werte bekommen. Hat das „Avoid Front“ Verhalten eine höhere Priorität als „Move To“ und die Gewichtung 1.0, so wird sich die Richtungsänderung mit 90° durchsetzen, ansonsten gibt es einen Kompromiss, wie eben beschrieben. Eine interessante Möglichkeit zur Erweiterung wäre, den Resolver für die Richtungsänderung mittels Vektoraddition zu implementieren, indem die vorgeschlagenen Werte als Richtungen und die Gewichtungen als Länge der Vektoren interpretiert werden. Das Prioritätsmanagement könnte dafür beibehalten werden.
Eine weitere Variante dieses Resolvers arbeitet mit den gleichen Eingaben, allerdings wählt sie nur eine Aktion maximaler Priorität und Gewichtung aus und gibt diese zurück. Dies ist sinnvoll, wenn sich die verschiedenen Verhalten auf einem Kanal nicht überlagern sollen, sondern sich nur ein Verhalten durchsetzen soll. Dieser Resolver wird im Moment für den Greifer eingesetzt. Der Grund ist, dass die Position des Greifers nicht abgefragt werden kann, sondern lediglich nach einer ausgeführten Aktion zurückgemeldet wird, dass die gewünschte Position erreicht ist. Würden sich die Verhalten überlagern, wäre nicht feststellbar, an welcher Position sich der Greifer nach Ausführung einer Aktion befindet.

Kombinieren der Kanäle

Nachdem die Konflikte auf den Kanälen gelöst sind, müssen Aktionen der verschiedenen Kanäle noch kombiniert werden. Hierzu sei bemerkt, dass in einer

Konfigurationsdatei die maximalen und minimalen Geschwindigkeiten festgelegt werden können. Diese sind vom Roboter unter allen Umständen einzuhalten. Im Weiteren werden diese als V_{MAX} (maximale Geschwindigkeit), V_{MIN} (minimale Geschwindigkeit) und V_{ROT} (maximale Rotationsgeschwindigkeit) referenziert. Zu Beginn des Kombinationsvorgangs wird überprüft, ob es Kanäle gibt, für die keine Aktion vorgeschlagen ist. Sollte das der Fall sein, wird auf Standardwerte zurückgegriffen. Im Folgenden steht der Index res an den Werten, welche von den Resolvern der einzelnen Kanäle ermittelt worden sind. Der Index $komb$ markiert die Werte, die im Weiteren zur Kombination eingesetzt werden. Diese ermitteln sich wie folgt:

- Geschwindigkeit:
$$v^{komb} = \begin{cases} 0\text{mm/s} & \text{falls keine Aktion vorgeschlagen} \\ v^{res} & \text{sonst} \end{cases}$$

- Richtungsänderung:
$$\alpha^{komb} = \begin{cases} 0° & \text{falls keine Aktion vorgeschlagen} \\ \alpha^{res} & \text{sonst} \end{cases}$$

- Maximale Geschwindigkeit:
$$v_{max}^{komb} = \begin{cases} V_{MAX} & \text{falls keine Aktion vorgeschlagen} \\ \min\{v_{max}^{res}, V_{MAX}\} & \text{falls } v_{max}^{res} > V_{MIN} \\ V_{MIN} & \text{sonst} \end{cases}$$

- Minimale Geschwindigkeit:
$$v_{min}^{komb} = \begin{cases} V_{MIN} & \text{falls keine Aktion vorgeschlagen} \\ \max\{v_{min}^{res}, V_{MIN}\} & \text{falls } v_{min}^{res} < V_{MAX} \\ V_{MAX} & \text{sonst} \end{cases}$$

- Maximale Rotationsgeschwindigkeit:
$$v_{rot}^{komb} = \begin{cases} V_{ROT} & \text{falls keine Aktion vorgeschlagen} \\ \min\{|v_{rot}^{res}|, |V_{ROT}|\} & \text{sonst} \end{cases}$$

Falls für den Greifer keine Aktion vorgeschlagen ist, wird an diesen kein Steuerbefehl weitergegeben. Ansonsten wird der Steuerbefehl weitergeleitet, der vom Resolver ausgewählt worden ist.
Beim Kombinieren wird die Semantik der Bezeichnungen der Kanäle berücksichtigt. Sollte bei den ermittelten Geschwindigkeiten $v_{min}^{komb} > v_{max}^{komb}$ gelten, wird

der Roboter angehalten. Ansonsten wird die Geschwindigkeit so begrenzt, dass sie zwischen maximaler und minimaler Geschwindigkeit liegt. Die relative Richtungsänderung wird, wie beschrieben, in eine Rotation umgesetzt, die mit der Richtungsänderung wächst (siehe Tabelle 3.1). Die endgültige Rotationsgeschwindigkeit darf nicht größer als die maximale Rotationsgeschwindigkeit sein. Seien beispielsweise folgende Größen gegeben:

- Geschwindigkeit $v^{komb} = 194\text{mm/s}$
- Richtungsänderung $\alpha^{komb} = 120°$
- max. Geschwindigkeit $v^{komb}_{max} = 150\text{mm/s}$
- min. Geschwindigkeit $v^{komb}_{min} = -300°/\text{s}$
- max. Rotationsgeschwindigkeit $v^{komb}_{rot} = 90°/\text{s}$

Die Geschwindigkeit, welche nach dem Kombinieren zurückgegeben wird, beträgt 150mm/s. Dies entspricht der maximalen Geschwindigkeit, auf welche reduziert wird. Die Richtungsänderung wird zuerst in eine Rotation umgesetzt, hier $90°/\text{s}$. Dieser Wert ist nicht größer als die maximale Rotationsgeschwindigkeit und wird zurückgegeben.

3.3.2 Eigenschaften von Verhalten

Als nächstes wird auf wichtige Eigenschaften der Verhalten eingegangen, die bei der Entwicklung von Verhalten, unabhängig von ihrer Funktionalität, berücksichtigt werden müssen.

Zustand und Fortschritt

In der Steuerungsarchitektur ist es vorgesehen, dass ein Verhalten Auskunft darüber geben kann, ob es Fortschritte macht und in welchem Zustand es sich befindet. Dies ermöglicht der deliberativen Komponente zu überprüfen, ob die eingesetzten Verhalten geeignet sind eine bestimmte Aufgabe zu erfüllen, oder ob es Probleme gibt und die Verhalten eventuell ausgetauscht werden müssen. Was Zustand und Fortschritt genau bedeuten, ist von dem jeweiligen Verhalten abhängig. Es ist vorgesehen, dass der Zustand Werte zwischen 0 und 100 annehmen kann. Die Verhalten, die im Rahmen dieser Arbeit implementiert werden, nutzen nur die Werte 0, 90 und 100. Die Bedeutung dieser Werte ist in Tabelle 3.2 beschrieben.
Für ein Verhalten wie „Avoid Front“ lässt sich der Zustand in Abhängigkeit vom Vorhandensein von Hindernissen im Frontbereich angeben. Falls in diesem Bereich Hindernisse erkannt werden, ist das Verhalten im Zustand „Ziel nicht erreicht“ und sonst im Zustand „Ziel erreicht“. Für das Verhalten „Move To“ gilt,

Zustandswert	Semantik
0	Ziel nicht erreicht
90	Ziel fast erreicht
100	Ziel erreicht

Tabelle 3.2: Zustände

der Zustand „Ziel nicht erreicht“, solange die Entfernung zum Ziel größer als 20cm ist. Zwischen 20cm und 5cm ist der Zustand „Ziel fast erreicht“, und wenn die Entfernung kleiner 5cm ist, ist der Zustand „Ziel erreicht“. Die Zustand „Ziel fast erreicht“ kann gut für Navigationsaufgaben genutzt werden. Wenn Zwischenziele nicht exakt angefahren werden müssen, kann, sobald der Zustand „Ziel fast erreicht“ zurückgegeben wird, das nächste Ziel angesteuert werden. Der Fortschritt eines Verhaltens wird nach ähnlichen Gesichtspunkten gewählt, jedoch sind hier nur die Werte „wahr“ und „falsch“ vorgesehen. „Wahr“ bedeutet in diesem Zusammenhang, dass das Verhalten Fortschritte macht und sich seinem Ziel nähert, und „falsch“, dass das Verhalten keine Fortschritte macht.

Das Verhalten „Avoid Front“ macht solange Fortschritt, wie Hindernisse im Frontbereich sind und sich der Abstand zum nächsten Hindernis vergrößert, sonst nicht. „Move To“ macht nur Fortschritte, wenn sich Distanz oder Richtung zum Ziel verkleinern. Beim Implementieren weiterer Verhalten ist darauf zu achten, dass diese auch ihren Zustand und Fortschritt mit sich führen.

Prioritäten und Gewichtungen

Weiter ist für die Verhalten wichtig, dass die vorgeschlagenen Aktionen für die verschiedenen Kanäle zu den jeweiligen Resolver passen. Ist für einen bestimmten Kanal ein prioritätsgesteuerter Resolver vorgesehen, so muss jede für diesen Kanal vorgeschlagenen Aktion auch eine Priorität und eine Gewichtung besitzen. Dabei ist es möglich, dass ein Verhalten für jeden Kanal unterschiedliche Prioritäten und Gewichtungen verwendet. In der gegenwärtigen Implementierung nutzen die Verhalten für alle Kanäle die gleiche Priorität und Gewichtung, weshalb diese als Eigenschaft eines Verhaltens angesehen werden können. Die Werte für Priorität und Gewichtung können durch die deliberative Komponente oder vom Verhalten selbst gesteuert werden. Dieses Konzept erlaubt es, schnell und einfach zu definieren, wie sich Verhalten überlagern sollen. Dabei erweist es sich als sinnvoll, den Verhalten mit steigender Sicherheitsrelevanz eine höhere Priorität zuzuweisen. Wenn „Avoid Front“ die maximale Priorität besitzt und mit näher kommenden Hindernissen seine Gewichtung auf 1.0 steigert, so kann es den Roboter in Gefahrensituationen alleine steuern und vor Kollisionen schützen.

„Single vote“ und „multiple vote“

Dieser Abschnitt erläutert die unterschiedlichen Ansätze von „single vote“ und „multiple vote“ Verhalten. Schlägt ein Verhalten für einen Kanal nur einen Wert vor, wird von „single vote“ gesprochen. Beispielsweise das bereits beschriebene “Avoid Front“ Verhalten ist hier einzuordnen. Dieses Verhalten schlägt beispielsweise in Abbildung 3.9 eine Richtungsänderung von -90° (nach rechts) vor, um dem Hindernis auszuweichen. Besitzt dieses Verhalten die maximale Priorität und Gewichtung, so steuert es den Roboter alleine und führt ihn damit zwar weg vom Hindernis, aber auch weg vom Ziel. Ist ein „Move To“ Verhalten aktiv, wel-

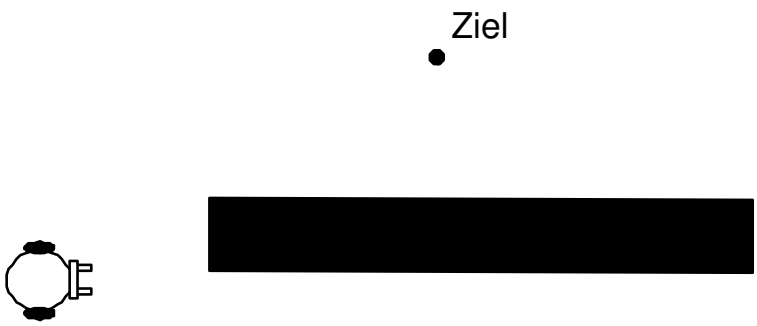

Abbildung 3.9: Überlagern von Verhalten

ches den Roboter zu dem markierten Ziel steuert, so schlägt es im Beispiel eine Richtungsänderung um 30° vor. Besitzen „Move To“ und „Avoid Front“ gleiche Priorität und Gewichtung, ist das Ergebnis der Überlagerung der Verhalten eine Richtungsänderung von -30°. Dies führt dazu, dass der Roboter knapper am Hindernis vorbeifährt, aber auf der dem Ziel abgewandten Seite bleibt.
Eine Lösung dieses Problems besteht in der Verwendung von „multiple vote“ Verhalten. Ein solches Verhalten gibt statt eines Wertes für einen bestimmten Kanal eine Bewertung für alle möglichen Werte ab, die angibt, inwieweit der jeweilige Wert dem Verhalten geeignet scheint, sein Ziel zu erreichen. Beispielsweise könnte das „Avoid Front“ beziehungsweise das „Move To“ Verhalten dann folgende Bewertungen für die möglichen Richtungsänderungen abgeben (siehe Abbildung 3.10,3.11): In den beiden Abbildungen (3.10,3.11) ist an der x-Achse die Richtungsänderung in Grad angetragen, an der y-Achse die Bewertung dieser Richtungsänderung. Der Wertebereich von y sind in diesem Beispiel die natürlichen Zahlen in $[0; 1000]$, wobei größere Werte für eine bessere Bewertung der jeweiligen Richtungsänderung stehen. Um die beiden Verhalten auf diesem Kanal zu koordinieren, können die bereits beschriebenen Resolver genutzt werden. Wird der Minimumresolver eingesetzt, lässt sich das Ergebnis der Überlagerung der Verhalten folgendermaßen darstellen (siehe Abbildung 3.12). Die Richtungsänderungen mit maximaler Bewertung liegen zwischen 60° und 75°. Wird ein Winkel aus diesem Bereich zur Steuerung des Roboter ausgewählt, fährt er nicht wie oben

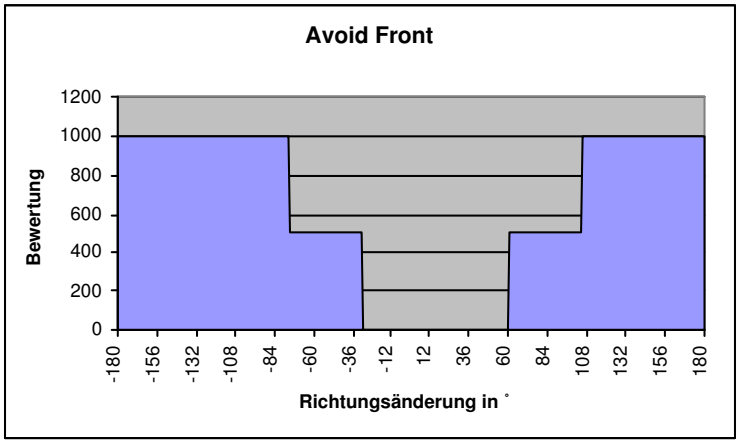

Abbildung 3.10: Multiple Vote - Avoid Front

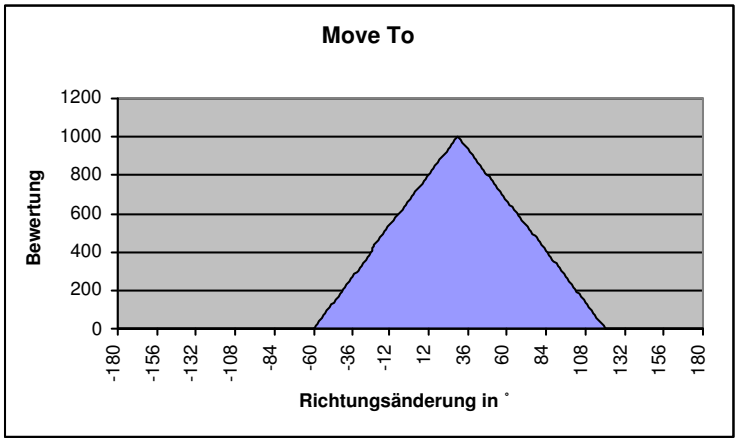

Abbildung 3.11: Multiple Vote - Move To

rechts am Hindernis vorbei, sondern auf der linken, dem Ziel zugewandten Seite, mit ausreichendem Abstand. Der wesentliche Unterschied ist, dass im ersten Fall das „Avoid Front“ Verhalten einen einzigen Wert vorschlagen muss, ohne die Ziele der anderen Verhalten zu kennen. Im zweiten Fall, ist es möglich, dass dieses Verhalten vorschlägt, rechts oder links vorbeizufahren, solange der Abstand groß genug ist.

In der gegenwärtigen Implementierung sind alle Verhalten als „single vote“ implementiert. Da bereits die „single vote“ Verhalten ein brauchbares Ergebnis liefern, um die Einsatzfähigkeit der Steuerungsarchitektur zu demonstrieren. Die Integration von „multiple vote“ Verhalten ist bereits vorgesehen. Es ist möglich, „multiple vote“ auf einzelnen Kanälen einzuführen. Dazu müssen lediglich die Verhalten dieses unterstützen und die Methode zum Kombinieren der Kanäle geändert werden. Weitere Information zu „multiple vote“ Verhalten sind unter anderem bei [11]

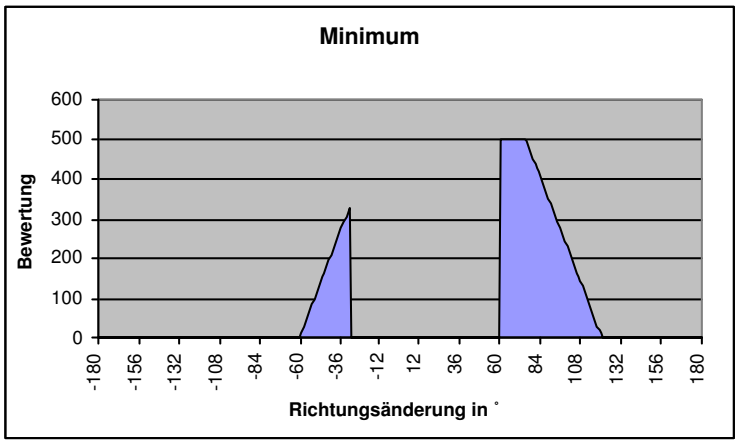

Abbildung 3.12: Multiple Vote - Minimum

und in der Dokumentation von Saphira im Rahmen der Fuzzy-Logik verfügbar (siehe [2], [3]).

3.3.3 Gruppieren von Verhalten

Eine weitere Eigenschaft der Verhalten besteht darin, dass Verhalten entwickelt werden können, die auf bereits bestehende Verhalten aufbauen oder sich aus diesen zusammensetzen.
Das bereits bekannte „Avoid Front“ Verhalten reagiert beispielsweise auf Hindernisse im Frontbereich, die näher liegen als 0.80m. Dieses Verhalten ist für eine langsame Fahrt des Roboters entwickelt. Soll der Pioneer schneller fahren, ist es sinnvoll, bereits auf Hindernisse zu reagieren, die noch weiter entfernt sind. Um die Größe des Bereiches in dem „Avoid Front“ auf Hindernisse reagiert, dynamisch an die gefahrene Geschwindigkeit anzupassen, soll das Verhalten „Avoid Front Velocity“ entwickelt werden, welches das bereits entwickelte „Avoid Front“ Verhalten nutzt. Bei jedem Aufruf von „Avoid Front Velocity“ passt dieses Verhalten zunächst die Größe des Frontbereiches von „Avoid Front“ an die Geschwindigkeit an, bringt anschließend „Avoid Front“ zur Ausführung und gibt dessen vorgeschlagene Aktionen an den Koordinationsalgorithmus zurück. Bei der Entwicklung des neuen „Avoid Front Velocity“ Verhaltens muss berücksichtigt werden, dass es seinen Zustand und Fortschritt mitführt, und eventuell seine Priorität und Gewichtung in die zurückgegebenen Aktionen einbringt. Da dies den Aufwand der Implementierung vergrößert, kann hier auch auf Vererbung zurückgegriffen werden.
Das Design der Verhalten ermöglicht nicht nur, dass ein Verhalten ein anderes nutzen kann, sondern auch, dass ein Verhalten auf mehrere andere Verhalten zurückgreift. Damit können Verhalten geschaffen werden, die eine Sammlung anderer Verhalten besitzen, und deren Eigenschaften so aufeinander abstimmen,

dass sie in geeigneter Weise zusammenspielen, um ein bestimmtes Ziel zu erfüllen. Um Kollisionen zu vermeiden, könnte ein Verhalten „Avoid Collisions“ entwickelt werden, das nicht nur aus dem Verhalten „Avoid Front“ besteht, sondern zusätzlich die Verhalten „Avoid Right“ und „Avoid Left“ besitzt, um Abstand zu Hindernissen rechts und links zu halten. Wird diese Sammlung von Verhalten zum Steuerungszyklus hinzugefügt, so lassen sich damit nicht nur alle Verhalten die „Avoid Collisions“ besitzt zur Ausführung bringen, sondern auch deren Priorität und Gewichtung aufeinander abstimmen. Nachdem „Avoid Collisions“ seine Verhalten ausgeführt hat, sammelt es die Aktionen, welche diese vorgeschlagen haben, ein und gibt sie an den Koordinationsalgorithmus weiter.
Dieses Vorgehen bringt bei prioritätsgesteuerten Verhalten allerdings einen wesentlichen Nachteil mit sich. Falls neben einer solchen Sammlung von Verhalten weitere Verhalten im Steuerungszyklus aktiv sind, kann es zu unerwünschten Interferenzen bei der Verhaltenskoordination kommen. Der Grund hierfür ist, dass die Prioritäten und Gewichtungen der Verhalten einer solchen Sammlung vom Entwickler zwar leicht untereinander abgestimmt werden können, sie aber zur Laufzeit mit wechselnden Verhalten außerhalb der Sammlung abzustimmen, ist ungleich schwerer. Deshalb scheint wünschenswert, dass aus den Verhalten einer solchen Sammlung zuerst ein gemeinsames Gesamtverhalten ermittelt wird, bevor eine Koordination mit den anderen Verhalten erfolgt. Das hat den Vorteil, dass sich die Interferenzen mit anderen Verhalten besser vorhersehen lassen. Um dies zu Verwirklichen sind die Gruppen von Verhalten entwickelt worden. Eine Gruppe hat einen eigenes Modul zur Koordination ihrer Verhalten (siehe 3.13). Die Gruppe „Gruppe1“ besteht im Beispiel aus den Verhalten VG_1 bis VG_4. Die Aktionen dieser Verhalten werden an die gruppeneigenen Kanäle und Resolver weitergeben. Die Steueranweisungen, die bei der Verhaltenskoordination ermittelt werden, werden an „Gruppe1“ zurückgeben und können neu mit Priorität und Gewichtung versehen werden, bevor eine Koordination mit den weiteren Verhalten im Steuerungszyklus erfolgt. Diese Gruppen von Verhalten erlauben es, komplexe Verhalten aus einfachen aufzubauen, und diese kontrolliert mit anderen Verhalten zusammenarbeiten zu lassen. In der aktuellen Version [1] unterstützt ARIA solche Gruppen nicht.

3.3.4 Entwickelte Verhalten

Die Tabelle 3.3 stellt die bisher entwickelten Verhalten im Überblick dar. Die Werte für Priorität und Gewichtung haben sich bei den Testfahrten bewährt.

Key Motion

Das „Key Motion“ Verhalten erlaubt die Steuerung des Pioneer mittels der Computermaus oder des Nummernblocks der Tastatur. Das GUI-Design (siehe Abbildung 3.14) ist aus der Diplomarbeit von Uwe Rene Vollrath [14] entnommen. Da

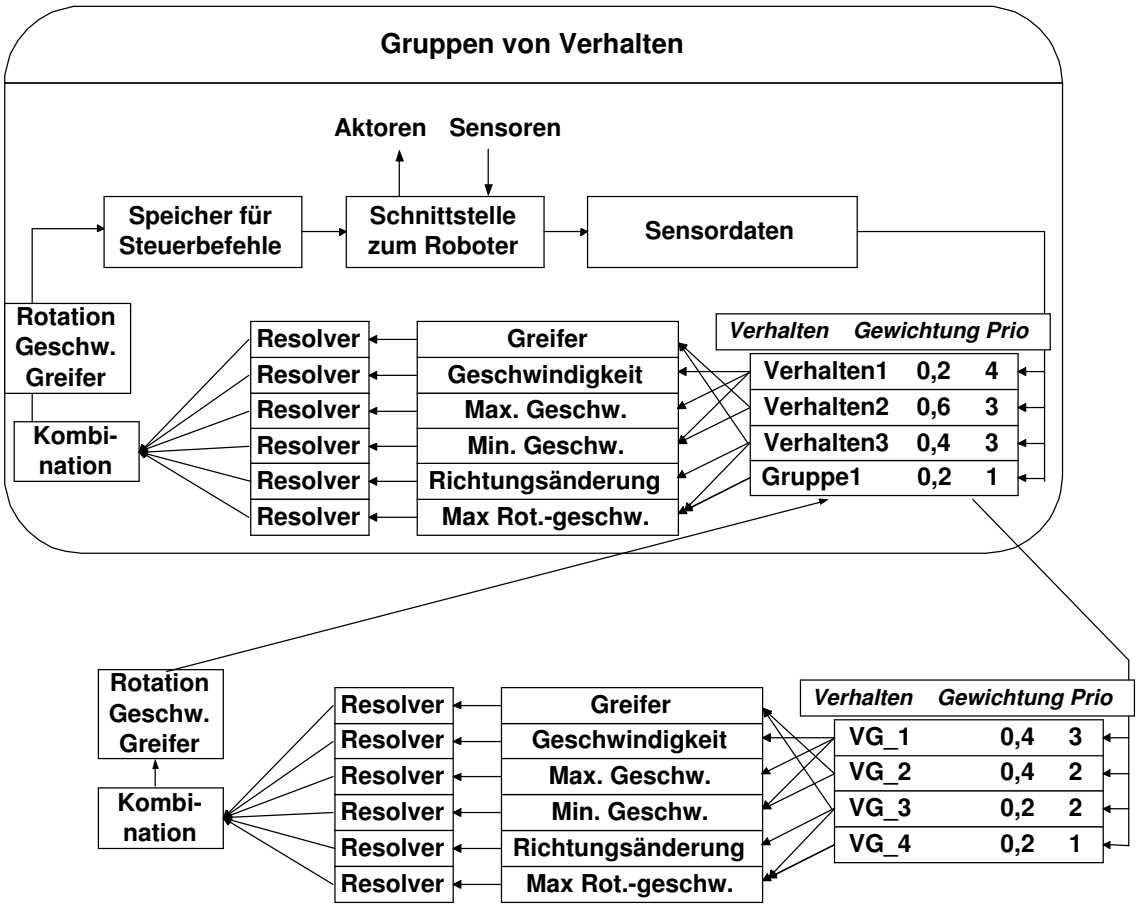

Abbildung 3.13: Gruppe von Verhalten

noch keine Navigationskomponente entwickelt ist, baut ein Großteil der durchgeführten Testfahrten auf diesem Verhalten auf. Wird mit Hilfe dieses Verhaltens eine konstante Geschwindigkeit festgesetzt und sind zusätzlich kollisionsvermeidende Verhalten aktiv, so ist das Ergebnis ein zielloses Umherfahren des Roboters, das sich beispielsweise für Kartenerstellung einsetzen lässt. Dieses Verhalten ermöglicht auch die Ansteuerung der Greifer.

Move To

Das “Move To“ Verhalten ist für die Steuerung des Pioneers durch eine Navigationskomponente gedacht. Wird diesem Verhalten ein Ziel, beschrieben durch seine Koordinaten bezüglich $\mathcal{K}_W$, übergeben, steuert es den Roboter dort hin. Dabei erfolgt erst eine Drehung des Roboters in Richtung des Ziels. Wenn die Abweichung von der Zielrichtung weniger als 60° beträgt, beginnt der Roboter sich zum Ziel zu bewegen. Die Richtung und der Abstand zum Ziel werden dabei

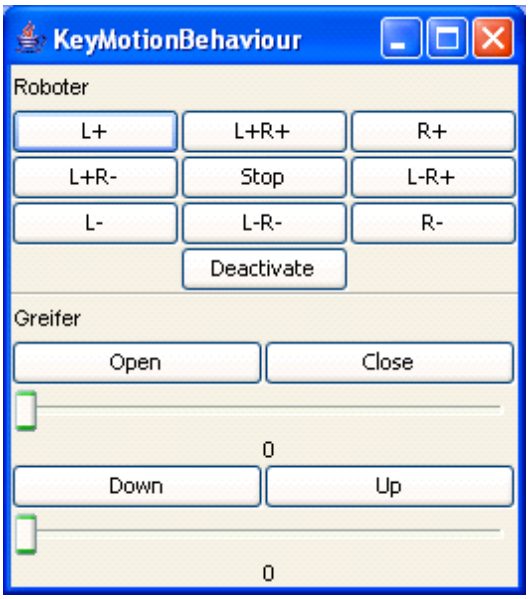

Abbildung 3.14: GUI - Key Motion

ständig überwacht. Die Geschwindigkeit v [mm/s], mit der dieses Verhalten den Roboter zum Ziel steuert, ist abhängig von der Entfernung d [mm] zum Ziel (siehe Abbildung 3.15). Die Geschwindigkeit wird beschrieben durch:

$$v = \begin{cases} \sqrt{d * 100} & \text{falls } d < 500 \\ \sqrt{d * 200} & \text{sonst} \end{cases} \tag{3.18}$$

Als Funktion ist wie in ARIA eine Wurzelfunktion gewählt. Der Sprung bei $d = 500$ dient um zu verhindern, dass der Roboter über das Ziel hinaus fährt.

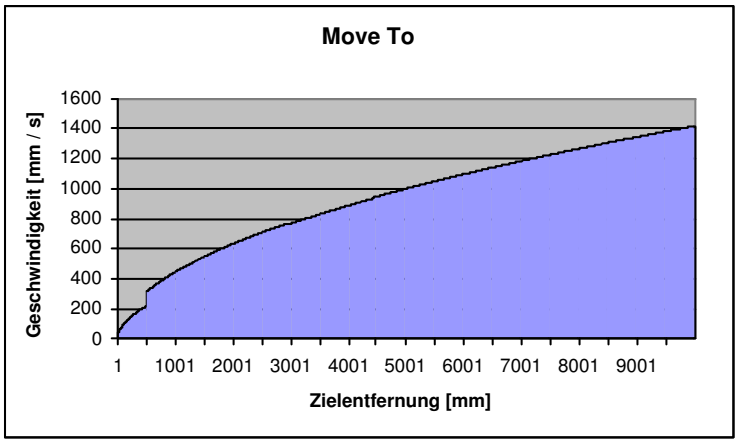

Abbildung 3.15: Geschwindigkeit des „Move To“ Verhaltens

Avoid Front

Dieses Verhalten ist entwickelt, um den Roboter vor Hindernissen im Frontbereich ausweichen zu lassen. Der Frontbereich wird durch einen Kreissektor mit einem Öffnungswinkel $\alpha = 60°$ und variablem Radius r festgelegt (siehe Abbildung 3.16). Um die Hindernisse in diesem Bereich zu ermitteln, wird, wie bei den weiteren „Avoid“ Verhalten, der Hindernis-Interpreter eingesetzt. Wenn Hindernisse in diesem Bereich sind, wird das Ausweichen durch eine Richtungsänderung entweder nach links (+90°) oder rechts (-90°) eingeleitet. Das Verhalten steuert den Roboter nur solange, wie auch Hindernisse erkannt werden. Die Bestimmung der Drehrichtung erfolgt mit Hilfe abstoßender Potentialfelder ([8] S. 122 - 147). Dadurch wird erreicht, dass der Roboter, auch wenn er von mehreren Hindernissen umgeben ist, sich geschickt zwischen diesen bewegen kann. Sei $\mathcal{H} = \{H_{1_R}, ..., H_{n_R}\}$ eine Menge von Punkten, welche Hindernisse repräsentieren, die im Frontbereich erkannt werden (siehe Abbildung 3.16). Sei

$$S^T = \sum_{i=1}^{n} H_{i_R}^T / \|H_{i_R}^T\|_2, \tag{3.19}$$

so gilt für die Ausweichrichtung

$$\beta = \begin{cases} +90° & \text{falls } S \text{ unterhalb der } x\text{-Achse ist} \\ -90° & \text{sonst.} \end{cases} \tag{3.20}$$

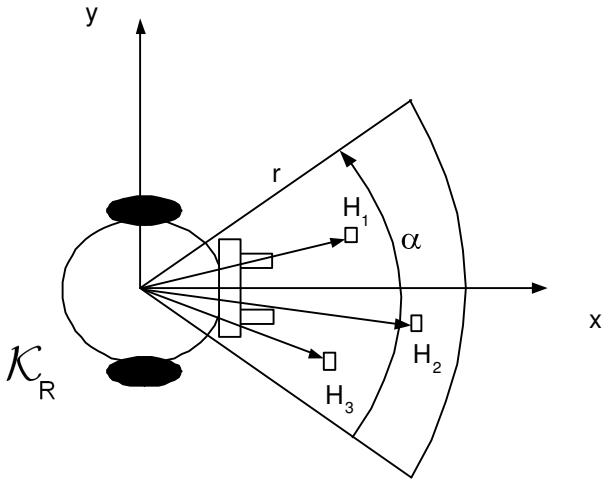

Abbildung 3.16: Ausweichen mittels „Avoid Front“

Da dieses Verhalten in der aktuellen Zusammenstellung die maximale Priorität besitzt, steuert es die Richtungsänderung des Roboters alleine, falls der Roboter einem Hindernis zu nahe kommt.

Avoid Side

„Avoid Side“ hat die Aufgabe, den Roboter von Hindernissen, die sich seitlich von ihm befinden, fernzuhalten. Es kann für die rechte und für die linke Seite genutzt werden. Das Verhalten prüft, ob sich innerhalb eines Kreissektors mit variablem Radius Hindernisse befinden und versucht den Roboter gegebenenfalls von diesen Weg zu lenken. Die Priorität ist kleiner als die des „Avoid Front“ Verhaltens. Eine grafische Darstellung des seitlichen Sicherheitsbereiches ist in Abbildung 3.17 zu sehen.

Avoid Side Box

Das Verhalten „Avoid Side Box“ hat wie „Avoid Side“ die Aufgabe, den Roboter von Hindernissen, die sich neben ihm befinden, weg zu lenken. „Avoid Side Box“ reagiert dabei auf Hindernisse, die sich in einem Rechteck neben dem Roboter befinden. Eine grafische Darstellung ist in Abbildung 3.17 zu sehen.

Avoid Front Velocity

Das Verhalten „Avoid Front Velocity“ passt den Radius r [cm] des Frontbereiches an die Geschwindigkeit v [mm/s] an. Es gilt:

$$r = \begin{cases} 900 & \text{falls } v < 200 \\ (1300 - 900)/200 * (v - 200) + 900 & \text{falls } 200 <= v < 400 \\ 1300 & \text{sonst.} \end{cases} \tag{3.21}$$

Avoid Side Velocity, Avoid Side Velocity Group

Ein dynamisches „Avoid Side“ Verhalten ist als Sammlung von Verhalten und als Gruppe von Verhalten implementiert. Die Funktionalität ist in beiden Fällen gleich. Ist die Geschwindigkeit kleiner 200mm/s, wird das „Avoid Side“ Verhalten eingesetzt. Ansonst wird „Avoid Side Box“ genutzt, um den seitlichen Sicherheitsbereich zu vergrößern (siehe Abbildung 3.17).

Velocity Delimiter

Dieses Verhalten dient dazu indirekt die Beschleunigung des Roboters zu steuern. Es erlaubt nur Geschwindigkeitserhöhungen oder -verminderung um 264mm/s ausgehend von der aktuellen Geschwindigkeit. Der Hintergrund ist, dass einerseits die Radmotoren des Pioneer blockieren, wenn die Geschwindigkeitsänderungen zu groß sind, andererseits kippt der Pioneer nach vorne, wenn bei hohen Geschwindigkeiten zu stark gebremst wird.

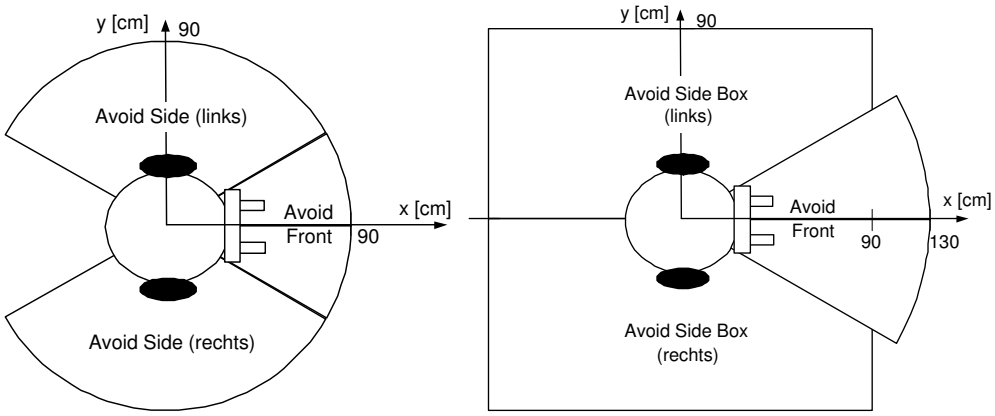

Abbildung 3.17: Kombinierter Sicherheitsbereich bei niedrigen Geschwindigkeiten (links) und hohen Geschwindigkeiten (rechts)

Obstacle Velocity Delimiter

Dieses Verhalten bremst den Roboter, wenn er sich auf Hindernisse zubewegt. Um dies zu erreichen, beschränkt er die maximale Geschwindigkeit v_{max} [mm/s] abhängig von der minimalen Distanz d_{front} [mm] zu Hindernissen vor dem Roboter. Die minimale Geschwindigkeit v_{min} [mm/s] wird analog in Abhängigkeit vom minimalen Abstand d_{rear} [mm] zu Hindernissen hinter dem Roboter gewählt. Die Geschwindigkeiten v_{max} und v_{min} berechnen sich jeweils zu:

$$v_{max} = \begin{cases} 0 & \text{falls } d_{front} \leq 680 \\ 2000/(5000 - 680) * (d_{front} - 680) & \text{sonst} \end{cases} \tag{3.22}$$

$$v_{min} = \begin{cases} 0 & \text{falls } d_{rear} \leq 680 \\ 2000/(5000 - 680) * (d_{rear} - 680) & \text{sonst} \end{cases} \tag{3.23}$$

Avoid Collisions

„Avoid Collisions“ ist eine Gruppe von Verhalten, die den Roboter vor Kollisionen schützen soll. Diese Gruppe nutzt die Verhalten:

- Avoid Front Velocity
- Avoid Side Velocity (für die linke Seite)
- Avoid Side Velocity (für die rechte Seite)
- Obstacle Velocity Delimiter

Verhalten (Name)	Funktion	Priorität	Gewichtung	Kanäle
Key Motion	Steuerung per Tastatur oder Maus	3	0.5	Geschwindigkeit, Richtungsänderung, Greifer
Move To (x, y)	Steuert den Roboter zu den Koordinaten (x, y) des Weltkoordinatensystems $\mathcal{K}_W$	3	0.5	Geschwindigkeit, Richtungsänderung
Turn α	Dreht den Roboter auf der Stelle um α	3	0.5	Richtungsänderung
Avoid Front	Steuert das Ausweichverhalten bei Hindernissen im Frontbereich	7	1.0	Richtungsänderung
Avoid Side	Steuert das Ausweichverhalten bei Hindernissen im Seitenbereich (rechts oder links)	6	1.0	Richtungsänderung
Avoid Side Box	Steuert das Ausweichverhalten bei Hindernissen im Seitenbereich (rechts oder links)	6	1.0	Richtungsänderung
Avoid Front Velocity	Steuert geschwindigkeitsabhängig das Ausweichverhalten bei Hindernissen im Frontbereich	7	1.0	Richtungsänderung
Avoid Side Velocity, Avoid Side Velocity Group	Steuert geschwindigkeitsabhängig das Ausweichverhalten bei Hindernissen im Seitenbereich (rechts oder links)	6	1.0	Richtungsänderung
Velocity Delimiter	Begrenzt die Geschwindigkeiten in Abhängigkeit von der aktuellen Geschwindigkeit	-	-	Max. und Min. Geschwindigkeit
Obstacle Velocity Delimiter	Begrenzt die Geschwindigkeit in Abhängigkeit von Hindernissen	-	-	Max. und Min. Geschwindigkeit
Avoid Collisions	Komplexes Verhalten zum Vermeiden von Kollisionen	7	1.0	Geschwindigkeit, Richtungsänderung, Max. und Min. Geschwindigkeit
Follow Mouse	Interaktive Steuerung mittels Computermaus - durch Klicks in die GUI (nutzt Move To)	3	0.5	Geschwindigkeit, Richtungsänderung

Tabelle 3.3: Entwickelte Verhalten

3.4 Schnittstellen zur deliberativen Komponente

Die aktuelle Implementierung sieht eine deliberative Komponente vor, jedoch hätte deren Entwicklung den Rahmen dieser Arbeit überschritten. Es gibt, wie in Abbildung 3.1 angedeutet, eine Schnittstelle, die es der deliberativen Komponente erlaubt auf den reaktiven Steuerungszyklus zuzugreifen. Diese Schnittstelle ermöglicht den Zugriff auf alle Daten des Zustandsreflektors, des Ereigniscontainers und der Interpreter. Diese können genutzt werden, um situationsabhängig die Verhalten zu kontrollieren. Zusätzlich ist es möglich, Interpreter zur Laufzeit hinzuzufügen oder zu entfernen, je nachdem welche Informationen für die aktuelle Aufgabe nötig sind. Des Weiteren kann die deliberative Komponente Verhalten in den Steuerungszyklus einbringen und aus ihm entfernen. Um zu überwachen, ob die eingesetzten Verhalten das gewünschte Resultat liefern, kann der Zustand und Fortschritt der Verhalten abgefragt werden. Über die Schnittstelle können zudem Steuerbefehle direkt an den Roboter weitergeben werden. Dies ist vor allem für Steuerbefehle, die nicht die Navigation betreffen gedacht. Beispielsweise lässt sich so das Ein- und Ausschalten der Sonare steuern.
Da im Moment noch keine Komponente vorliegt, welche die Planungsaufgaben übernimmt, werden die verfügbaren Verhalten und Interpreter in der GUI zur Verfügung gestellt. Dies ermöglicht dem Anwender zur Laufzeit, Komponenten in den reaktiven Steuerungszyklus einzufügen bzw. zu entfernen.

Kapitel 4

Implementierung

In diesem Kapitel werden die wesentlichen Merkmale des Softwaredesigns der Steuerungsarchitektur vorgestellt. Zur Realisierung ist die Entwicklungsumgebung Eclipse 3.0 von IBM und Java in der Version 1.5.0_02 verwendet worden. Eine ausführliche Dokumentation des Codes ist mit `javadoc` erstellt worden. Die Komponenten, die im Rahmen dieser Arbeit entwickelt worden sind, sind in den Packages `tud.sim.pioneer.controlArchitecture` (Steuerungsarchitektur) und `tud.sim.pioneer.comArchitecture` (Remote-Schnittstelle) abgelegt. Testanwendungen befinden sich im Package `tud.sim.pioneer.test.application`. Da eine Remote-Schnittstelle zwischen Roboter und Steuerungszyklus eine grundlegende Anforderung an diese Arbeit ist, sieht die Implementierung vor, dass im Regelfall die Steuerungsarchitektur auf einem von Pioneer unabhängigen Computer ausgeführt wird und über WLAN auf den Roboter zugreift.

4.1 Implementierung der Steuerungsarchitektur

Abbildung 4.1 zeigt in einem Modell, wie die verschiedenen Schichten der Software auf der Hardware aufbauen. Diese werden in den nachfolgenden Abschnitten vorgestellt. Die Treiber und die Roboter-API [14] waren bereits vor Beginn dieser Arbeit vorhanden. Die wesentlichen Parameter der Architektur (z.B. Dauer des Steuerungszyklus) beziehungsweise der Hardware (z.B. Anzahl und Position der Sonare) können in einer Konfigurationsdatei festgelegt werden (siehe `tud.sim.pioneer.controlArchitecture.Const.controlArchitecture.properties`). Die Zeitangaben im nachfolgenden Abschnitt beziehen sich auf die aktuelle Konfiguration.

4.1.1 Schnittstelle zum Roboter

Die Steuerung des Roboters durch die Steuerungsarchitektur erfolgt nach dem Client-Server-Prinzip. Die Steuerungsarchitektur ist hierbei der Client, der Robo-

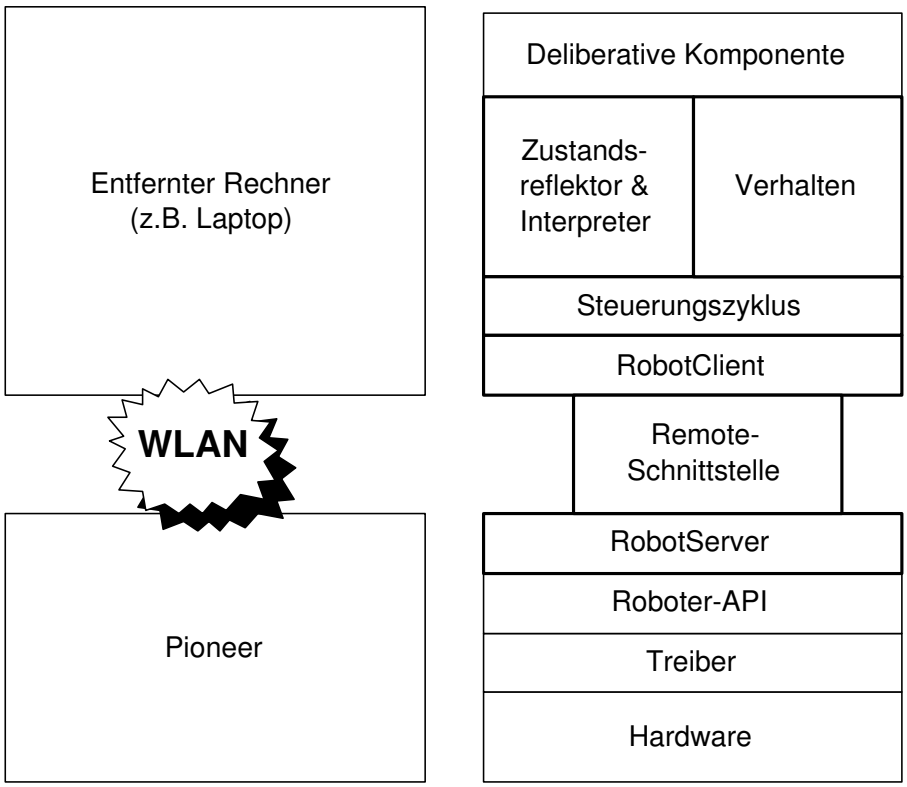

Abbildung 4.1: Abstaktionsebenen im Softwaredesign

ter der Server. Im Softwaredesign ist die Schnittstelle zwischen dem Roboter und der Architektur durch *RobotServer* `tud.sim.pioneer.controlArchitecture.interfaces.IRobotServer` und *RobotClient* `tud.sim.pioneer.controlArchitecture.interfaces.IRobotClient` realisiert. *RobotServer* und *RobotClient* haben im Wesentlichen die Aufgabe, die Remote-Schnittstelle zu kapseln und Daten auf beiden Seiten dieser Schnittstelle zu puffern. Dies ermöglicht, negative Auswirkungen durch Verzögerungen bei der Datenübertragung auf die Steuerungsarchitektur weitgehend zu vermeiden.

Die Implementierung des *RobotServer* für den Pioneer `tud.sim.pioneer.controlArchitecture.interfaces.p2dx.RobotServer` besitzt lokalen Zugriff auf die Roboter-API. Der *RobotServer* sammelt die Sensordaten, welche vom Pioneer ermittelt werden, und speichert diese, bis sie vom *RobotClient* abgeholt werden. Die Sensordaten der Roboter-API werden ca. alle 40ms aktualisiert. Um keine Sensordaten zu verlieren, wird alle 25ms vom *RobotServer* überprüft, ob neue Sensordaten vorliegen (siehe Abbildung 4.2). Steuerbefehle, die der *RobotServer* vom *RobotClient* erhält, gibt er an die Roboter-API weiter, welche diese zur Ausführung bringt (siehe Abbildung 4.3). Zusätzlich hat der *RobotServer* die Aufgabe den Roboter zu stoppen, falls es zu einem Verbindungsabriss oder zu größeren Verzögerung in der Kommunikation mit dem *RobotClient* kommt,

um zu verhindern, dass der Roboter unkontrolliert weiterfährt. Da speziell auf dem Pioneer Schwierigkeiten beim Anfahren mit niedrigen Geschwindigkeiten festgestellt worden sind, besitzt der *RobotServer* einen Thread zum Steuern der Radgeschwindigkeiten, der dazu dient, das Anfahrverhalten zu verbessern.
Die Implementierung des *RobotClient* in der Klasse `tud.sim.pioneer.controlArchitecture.interfaces.p2dx.RobotClient` ist das Gegenstück zum *RobotServer* auf der Seite der Steuerungsarchitektur. Der *RobotClient* tauscht die Sensordaten mit dem *RobotServer* über die Remote-Schnittstelle aus. Die Steuerungsarchitektur holt sich über synchronisierte lokale Zugriffe auf den *RobotClient* Sensordaten ab und gibt die am Ende eines Steuerungszyklus ermittelten Steuerbefehle an den *RobotClient* weiter.
Die Verteilung der Sensordaten vom Roboter an die Steuerungsarchitektur erfolgt nach dem Hol-Prinzip (engl. pull).

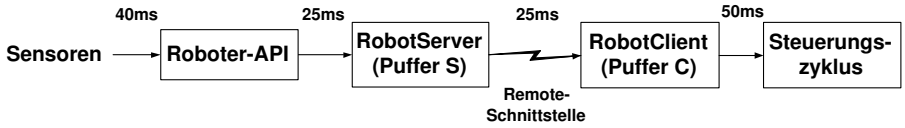

Abbildung 4.2: Verteilung der Sensordaten (mit Angabe der Abfrageintervalle)

Um eine kurze Übertragungszeit über die Remote-Schnittstelle zu gewährleisten, erfolgt die Übertragung der Sensordaten mittels eines Array vom Typ `byte[][]`. Für die Übertragung des Arrays mit einer Länge ca. 110byte, werden ca. 10ms benötigt. Gegenwärtig setzt sich der Array aus folgenden Bestandteilen zusammen:

- Sensordaten, welche die Roboter-API regelmäßig zur Verfügung stellt. In der Regel werden 108bytes für diese Sensordaten beansprucht (siehe `tud.sim.pioneer.controlArchitecture.sensors.SensorFactory`). Falls aufgrund von Verzögerungen mehr als ein Satz Sensordaten im *RobotServer* vorliegt, werden diese hintereinander gehängt. Die Länge der Sensordaten beträgt dann ein Vielfaches von 108bytes.

- Ereignisse des Greifers
  Details, wann solche Ereignisse auftreten, werden in [14] beschrieben.

- Ereignisse der Kamera
  Details, wann solche Ereignisse auftreten, werden in [14] beschrieben.

Sobald der *RobotClient* ein Array mit Sensordaten und Ereignissen erhält, erstellt er aus diesem objektorientierte Datenstrukturen (siehe Abbildung 4.5) (siehe Package `tud.sim.pioneer.controlArchitecture.sensors` und `tud.sim.pioneer.controlArchitecture.events`). Um die Handhabbarkeit möglichst einfach zu halten, werden zur Erstellung dieser Datenstrukturen Fabriken (engl. factories) eingesetzt. Diese befinden sich im jeweiligen Package.

Die Übertragung der Steuerbefehle erfolgt nach dem Schiebe-Prinzip (engl. push). Ähnlich wie bei den Sensordaten, erfolgt auch hier die Übertragung der Steuerbefehle auf Basis eines Array vom Typ `byte[][]`. Die Implementierung der Steuerbefehle findet sich in `tud.sim.pioneer.controlArchitecture.commands`.

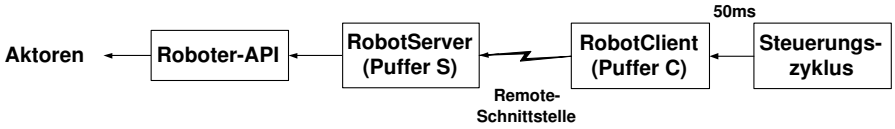

Abbildung 4.3: Verteilung der Steuerbefehle - Initiierung am Ende des Steuerungszyklus

4.1.2 Steuerungszyklus

Als nächstes werden die Komponenten erläutert, die Teil des reaktiven Steuerungszyklus sind. Der Ablauf des Steuerungszyklus wird durch die Klasse *ControlCenter* `tud.sim.pioneer.controlArchitecture.ControlCenter` gesteuert. Diese startet den Zyklus in einem eigenen Thread. Die Klasse hat Zugriff auf alle Datenstrukturen, die innerhalb des Steuerungszyklus eingesetzt werden (siehe Abbildung 4.4).

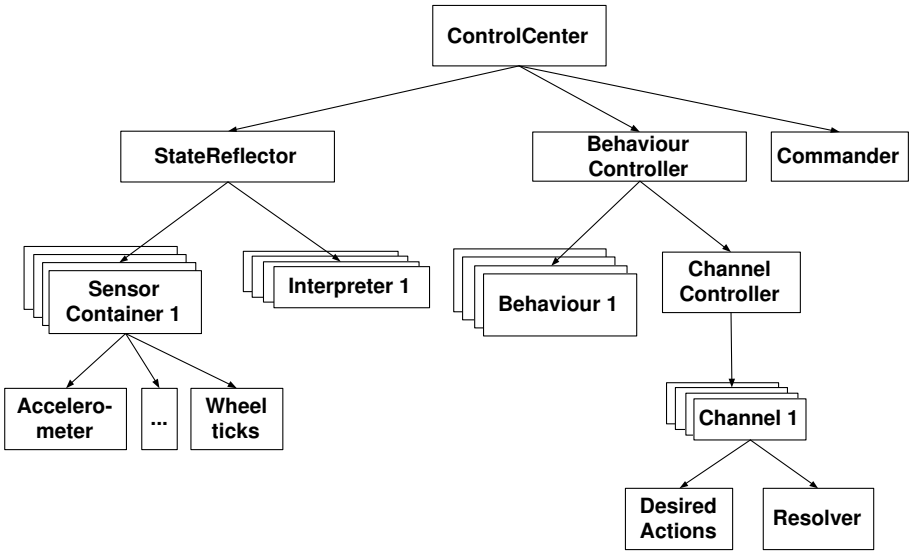

Abbildung 4.4: Struktur der Komponenten des Steuerungszyklus

Als erstes werden in jedem Zyklus die Sensordaten aktualisiert, anschließend die Interpreter ausgeführt und dann die Verhalten gestartet. Der Steuerungszyklus läuft sequentiell ab, das heißt innerhalb des Zyklus ist keine Synchronisation nötig. Im Moment werden für den Steuerungszyklus inklusive Interpreter und

Verhalten ca. 10ms Rechenzeit pro Durchlauf benötigt (auf einem Laptop mit 1600MHz Taktfrequenz und 256MB RAM). Die restliche Zeit der 50ms, die dem Steuerungszyklus in der gegenwärtigen Konfiguration zur Verfügung stehen, ist noch ungenutzt. Es ist nicht sinnvoll den Steuerungszyklus sofort nach Beendigung wieder neu zu starten, da die Sensordaten mindestens eine Aktualisierungsrate von 40ms haben.

Sensordatenverarbeitung

Die Implementierung des Zustandsreflektors ist die Klasse *StateReflector* `tud.sim.pioneer.controlArchitecture.StateReflector`. Der Zustandsreflektor aktualisiert seine Sensordaten zu Beginn jedes Zyklus und sammelt diese in einer eigenen Datenstruktur. Die zentrale Einheit dieser Datenstruktur ist die Klasse *SensorContainer* `tud.sim.pioneer.controlArchitecture.sensors.SensorContainer` (siehe Abbildung 4.5), welche die Daten der einzelnen Sensoren verwaltet und Zugriff auf diese ermöglicht. Wird die Hardware um zusätzliche Sensoren erweitert, sind die Daten dieser Sensoren in den Sensorcontainer zu integrieren. Die dazu benötigten neuen Klassen sind von den abstrakten Klassen *AbstractSensor* beziehungsweise *AbstractTimeSensor* im Package `tud.sim.pioneer.controlArchitecture.sensors` abzuleiten. Der Zustandsreflektor stellt Zugriff auf die letzten zehn erzeugten Sensorcontainer zur Verfügung.

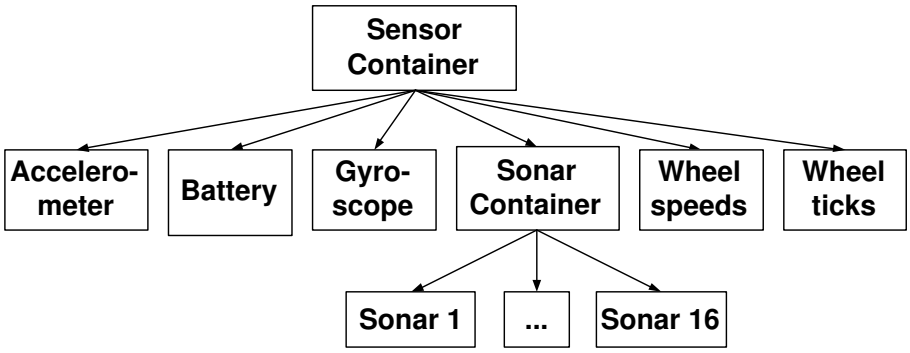

Abbildung 4.5: Struktur der Sensordaten

Die Interpreter sollen wie in 3.2.2 beschrieben, Sensordaten verarbeiten, beziehungsweise zur Verfügung stellen. Dazu wird jeder Interpreter nach Aktualisierung der Sensordaten des Zustandsreflektors aufgerufen, um seine Daten zu aktualisieren. Außerdem sollen Interpreter dynamisch zur Laufzeit hinzugefügt und entfernt werden können. Um diese Funktionalität zu realisieren, ist das Interface *Fireable* (`tud.sim.pioneer.controlArchitecture.Fireable`) entwickelt, das jeder Interpreter implementieren muss. Die Interpreter werden vom Zustandsreflektor verwaltet (siehe Abbildung 4.4).

Verhaltenssteuerung

Die Steuerung aller aktiven Verhalten erfolgt über die Klasse *BehaviourController* `tud.sim.pioneer.controlArchitecture.BehaviourController` (siehe Abbildung 4.4). Diese führt alle Verhalten aus und übergibt die vorgeschlagenen Aktionen an die Klasse *DefaultChannelController* `tud.sim.pioneer.controlArchitecture.channels.DefaultChannelController`, welche die Verhaltenskoordination übernimmt. Die Klasse *DefaultChannelController*, wie auch die Kanäle (engl. channels), die diese besitzt, sind erweiter- beziehungsweise austauschbar. Die Klasse *BehaviourUpdater* `tud.sim.pioneer.controlArchitecture.BehaviourUpdater` erlaubt durch synchronisierte Zugriffe, Verhalten von außerhalb des Steuerungszyklus hinzuzufügen und zu entfernen.

Verhalten

Alle implementierten Verhalten müssen von der abstrakten Basisklasse *Behaviour* `tud.sim.controlArchitecture.Behaviour` abgeleitet werden. Diese besitzt neben der abstrakten Methode `fire()`, die in jedem Steuerungszyklus aufgerufen wird und die Funktionalität eines Verhaltens enthalten soll, die Grundlagen für Zustand und Fortschritt. Verhalten, welche die bestehenden Kanäle und Resolver nutzen sollen, können von der Klasse *PriorityBehaviour* `tud.sim.pioneer.controlArchitecture.PriorityBehaviour` abgeleitet werden. Diese Klasse besitzt, neben Variablen für Priorität und Gewichtung, Methoden zum einfachen Erstellen von Aktionen für die verschiedenen Kanäle. Gruppen von Verhalten können von der abstrakten Basisklasse *PriorityBehaviourGroup* `tud.sim.pioneer.controlArchitecture.PriorityBehaviourGroup` erben.

Zur Veranschaulichung, wie weitere Verhalten entwickelt werden können, wird der Quellcode der Methode `fire()` ausgewählter Verhalten, die in dieser Arbeit implementiert worden sind, im Anhang aufgeführt.

4.1.3 Schnittstelle zur deliberativen Komponente

Die Schnittstelle zur deliberativen Komponente ist durch das Interface *IControlCenter* `tud.sim.pioneer.controlArchitecture.IControlCenter` definiert. Die Methoden des Interface erlauben den Zugriff auf den Zustandsreflektor, und damit auch auf die Interpreter, auf die aktiven Verhalten und auf die Klasse *BehaviourUpdater*. Zusätzlich ist es mittels der Methode `getCommander()` möglich Zugriff auf den Speicher für die Steuerbefehle zu erhalten. Über diesen können Steuerbefehle an den Roboter weitergegeben werden. Die Methoden `connect()` und `disconnect()` werden benötigt, um die Verbindung zwischen der Steuerungsarchitektur und der Roboter-API herzustellen, beziehungsweise zu trennen.

4.2 GUI

Mit Hilfe der GUI (engl. graphical user interface) kann der Nutzer die Aktivitäten der Steuerungsarchitektur überwachen. Die Klasse *ControlCenterFrame* `tud.sim.pioneer.controlArchitecture.gui.ControlCenterFrame` besteht im Wesentlichen aus einer Menüleiste zur Steuerung und drei Feldern (engl. panels) (siehe Abbildung 4.6). Das Feld mit der Tabelle links oben enthält die aktuellen Sensordaten. Das Feld darunter zeigt eine Karte mit dem Pioneer und den Hindernisse, die der Hindernis-Interpreter ermittelt. Auf der rechten Seite werden die aktuellen Verhalten und die Aktionen angezeigt, die sie vorschlagen. Über Menüs ist es außerdem möglich, die Kartenerstellung (Menüpunkt „Interpreters“) und Verhalten (Menüpunkt „Behaviours“) zu aktivieren, beziehungsweise zu deaktivieren. Der Menüpunkt „Maps“ ermöglicht unter anderem sich eine Karte anzeigen zu lassen. Dies ist bisher allerdings nur für den Betrieb im Simulator sinnvoll, da die Positionsbestimmung über die Odometrie noch zu ungenau ist, um die Position des Roboters auf einer Karte wiederzugeben (siehe Kapitel 3.2.3).

4.3 Simulator

Des Weiteren ist im Rahmen dieses Projekts ein Simulator entwickelt. Dieser simuliert die Bewegungen und Sensordaten des Pioneer in einer virtuellen Umgebung. Allerdings sind im Moment lediglich die Odometriedaten (Radticks und Radgeschwindigkeiten) und die Sonare aus dem Simulator verfügbar. Die Simulation der restlichen Sensoren ist noch zu implementieren. Um die Sonarwerte zu berechnen, ist es nötig, dem Simulator eine Karte, welche die virtuellen Umgebung repräsentiert, zur Verfügung zu stellen.
Die erstellten Karten befinden sich im Package `tud.sim.pioneer.controlArchitecture.Maps`. In der Datei `MapName.properties` in diesem Package, wird festgelegt, welche Karte für die Simulation beziehungsweise die GUI verwendet wird. Die Konfigurationsdatei für eine Karte ist folgendermaßen aufgebaut. In der ersten Zeile steht die Anzahl n der Einträge in der Datei, in den weiteren Zeilen stehen die Linien, beziehungsweise Wände, welche die Karte bilden in folgendem Format:

$$\text{line}j = x1\ y1\ x2\ y2$$

Das j am Ende von „line$j$“ steht für den Index und läuft von 0 bis $n-1$. Der Startpunkt einer Linie ist $(x1, y1)$, der Endpunkt $(x2, y2)$. Die Einheit von x und y ist mm. Bisher startet der Roboter bei $(0, 0)$. Es wäre allerdings auch denkbar, die Startposition des Roboters in dieser Datei zu festzulegen. Die Karte, die in Abbildung 4.6 zu sehen ist, hat folgende Konfigurationsdatei:

count = 6
line0 = -5000 -5000 5000 -5000
line1 = -5000 5000 -5000 -5000
line2 = 5000 -5000 5000 5000
line3 = 5000 5000 -5000 5000
line4 = 0 500 0 2000
line5 = -2000 4500 -2000 2500

Die vom Simulator errechneten Sensordaten sind ohne Rauschen. Durch Austauschen des Simulators ist es möglich, dies zu ergänzen.

4.4 Remote-Schnittstelle

Die entwickelte Remote-Schnittstelle ist vielfältig einsetzbar. Die Schnittstelle baut auf RMI („Remote Method Invokation“) auf und bietet einige Vorteile gegenüber einer direkten Verwendung von RMI. Die Dienste, die entfernten Computern zu Verfügung gestellt werden sollen und für dieses Framework entwickelt sind, werden im Weiteren Module genannt. Wesentlicher Bestandteil der Remote-Schnittstelle ist das Modul „Load And Register“. Es ermöglicht das dynamische Laden von Modulen durch entfernte Computer. Das heißt, es ist nicht erforderlich alle Module, die für entfernte Computer verfügbar sein sollen, sofort zu starten, sondern das „Load And Register“ Modul übernimmt dies, sobald ein Modul angefragt wird. Die Module müssen sich dazu lediglich im Klassenpfad der Java Virtual Machine (JVM) befinden, welche das *„LoadAndRegister“* Modul gestartet hat. Ein weiterer Unterschied zu RMI ist, dass der Zugriff auf Module, die in der lokalen JVM vorhanden sind, auch lokal erfolgt. Dies bringt in diesem Spezialfall gegenüber einer direkten Verwendung von RMI einen erheblichen Geschwindigkeitsvorteil. Im Weiteren wird davon ausgegangen, dass auf jedem Computer genau eine JVM läuft und der Zugriff auf ein Module, beziehungsweise auf dessen Methoden, ein Zugriff auf einen entfernten Computer ist. Der Computer, der auf ein Modul zugreift, wird Client genannt, der Computer auf dem das Modul ausgeführt wird, Server. Welche Implementation eines Moduls geladen wird, ist auf jedem Server konfigurierbar. Dadurch ist es möglich, auf einem Server die Implementation eines Moduls auszutauschen, ohne dass dazu Änderungen im Code nötig sind. So können auf verschiedenen Computern verschiedene Implementierung eines Dienstes laufen, die beispielsweise an die Rechenleistung des jeweiligen Servers angepasst sind. Der Name der Klasse, welche die Funktionalität implementiert, wird in einer Datei hinterlegt und kann ausgetauscht werden. Dies ist wesentlich übersichtlicher als die Verwendung unterschiedlicher Codebasen.
Für die Anwendung in der Steuerungsarchitektur bedeutet dies, dass das Modul *RobotServer* so konfiguriert werden kann, dass, falls es auf dem Pioneer ausgeführt wird, der Roboter auch tatsächlich angesteuert wird. Falls der *RobotServer*

aber zu Testzwecken auf einem anderen Computer ausgeführt werden soll, kann als Implementierung der Simulator angegeben werden, der Code der Steuerungsarchitektur bleibt unverändert.
Die Klassen der Remote-Schnittstelle befinden sich im Package `tud.sim.pioneer.comArchitecture`. In der Dokumentation des Interfaces *IComCenter* in diesem Package, findet sich auch eine Anleitung, wie Module für diese Schnittstelle zu entwickeln sind. Es werden allerdings Kenntnisse über RMI und eventuell Reflection vorausgesetzt.

4.5 Hinweise zum Einsatz der Steuerungsarchitektur

Das Starten der Steuerungsarchitektur kann über eine Instanz der Klasse *ControlCenterStarter* `tud.sim.pioneer.test.application.ControlCenterStarter` vorgenommen werden. Diese Klasse enthält Methoden, um die bereits entwickelten Interpreter und Verhalten zu laden und in der GUI verfügbar zu machen.
Als Testanwendung kann die Klasse *ControlCenterApp* `tud.sim.pioneer.test.application.ControlCenterApp` genutzt werden. Diese greift dabei auf die Klasse *ControlCenterStarter* zurück und startet die Steuerungsarchitektur einschließlich grafischer Oberfläche. Ein weiteres Beispiel für eine einfache Anwendung der Steuerungsarchitektur findet sich im Anhang (siehe A.1 Testanwendung).
In der aktuellen Konfiguration ist die IP-Adresse unter der nach dem *RobotServer* gesucht wird, auf „localhost“ festgelegt. Der lokale *RobotServer* startet den Simulator. Die Konfiguration hierfür wird in `tud.sim.pioneer.controlArchitecture.moduls.robotServer.RobotServerAdapter.properties` vorgenommen.
Soll der Pioneer gesteuert werden, so ist der Klasse ControlCenterApp beim Starten die IP-Adresse des Pioneers zu übergeben. Auf dem Pioneer muss die Klasse *PioneerApp* `tud.sim.pioneer.test.application.PioneerApp` ausgeführt werden. Diese ermöglicht den Zugriff auf den Pioneer durch entfernte Computer. Als Klasse für die Implementierung des *RobotServer* ist auf dem Pioneer in `tud.sim.pioneer.controlArchitecture.moduls.robotServer.RobotServerAdapter.properties` die Klasse `tud.sim.pioneer.controlArchitecture.interfaces.p2dx.RobotServer` anzugeben.

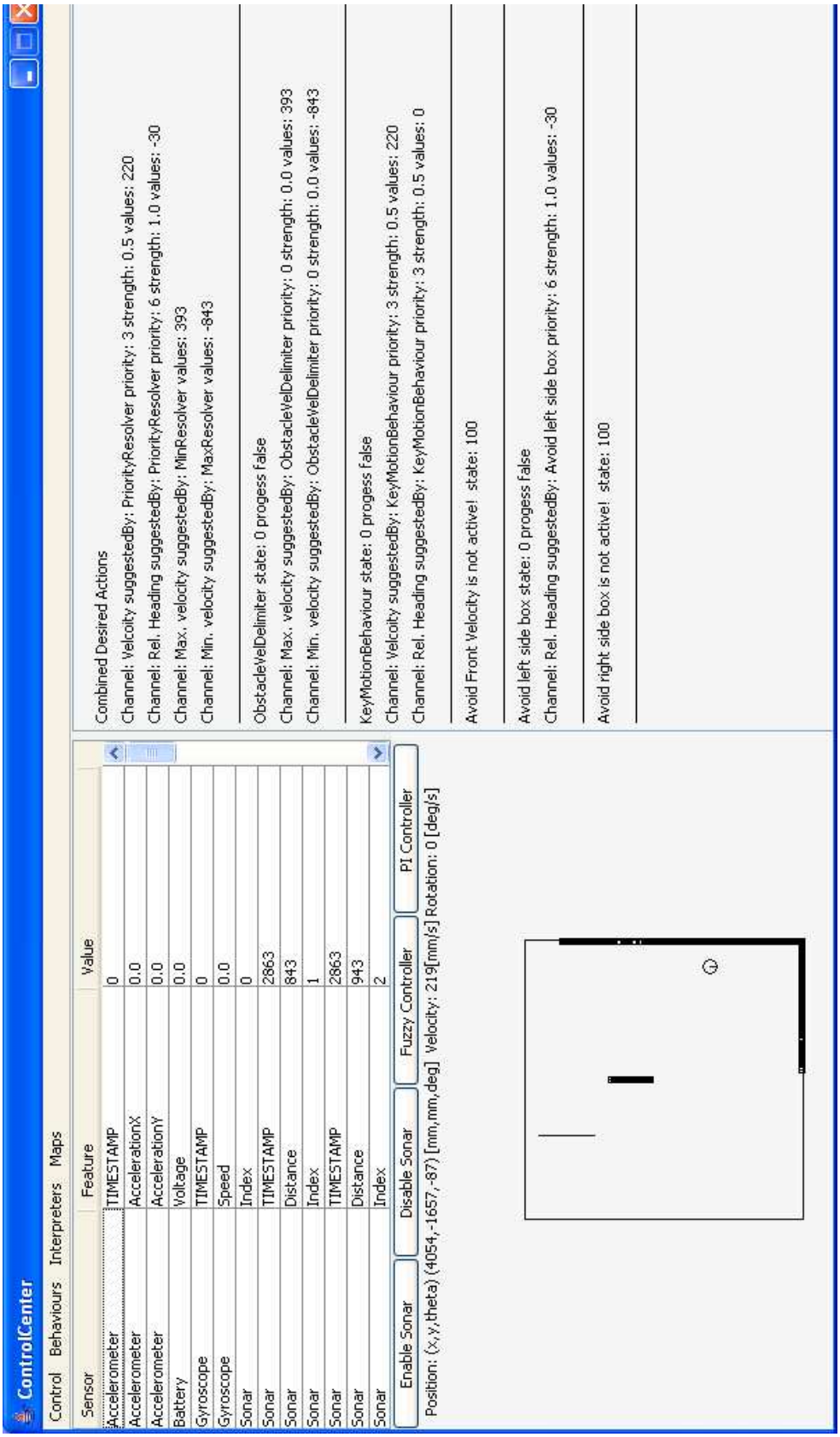

Abbildung 4.6: GUI - ControlCenterFrame

Kapitel 5

Zusammenfassung und Ausblick

Die bisherigen Tests zeigen, dass die Steuerungsarchitektur HydrA den zu Beginn der Arbeit festgelegten Anforderungen entspricht. Mit Hilfe der entwickelten Interpreter und Verhalten ist bereits autonomes kollisionsfreies Umherfahren bei Geschwindigkeiten bis 500mm/s möglich. Außerdem ist es möglich, die manuelle Steuerung des Roboters mit kollisionsvermeidenden Verhalten zu überlagern. Der Test der Verhalten im Simulator hat gezeigt, dass vor allem die niedrige Aktualisierungsrate der Sonare und deren Ungenauigkeit in der Realität Probleme bei der Steuerung machen. Die angesteuerten Geschwindigkeiten erreicht der Roboter in der Praxis im Allgemeinen nicht genau, bei niedrigen Geschwindigkeiten ist er in der Regel etwas langsamer, bei Geschwindigkeiten über ca. 500mm/s etwas schneller als gewünscht. Allerdings sollte dies kein Problem für die Steuerbarkeit des Roboters darstellen. Die Erweiterbarkeit zeigte sich bereits in der problemlosen Integration der von Gerhard Rohe entwickelten Kartenerstellung. Da wesentliche Parameter des Roboters konfigurierbar sind, können Teile der Hardware ausgetauscht werden. Nach einer Anpassung der Schnittstellen könnte die Architektur auch auf einem dem Pioneer ähnlichen Roboter eingesetzt werden. Durch Multi-Threading ist die Architektur sehr performant. Die Dauer des Steuerungszyklus (50ms) orientiert sich bisher nur an den Aktualisierungsraten der Sensoren (40ms). Wird die Steuerungsarchitektur auf einem Laptop gestartet (1600MHz Taktfrequenz und 256 MB RAM), beanspruchen die aktiven Verhalten und Interpreter zum autonomen ziellosen Umherfahren ca. 10ms. Die restlichen 40ms sind noch ungenutzt. Die Datenübertragung vom Pioneer auf den Laptop benötigt durchschnittlich 10ms. Trotz gelegentlich längerer Übertragungszeiten kann der Roboter flüssig gesteuert werden. Die GUI und der Simulator bilden eine gute Grundlage zur Entwicklung weiterer Interpreter und Verhalten. Andreas Grunewald arbeitet bereits an einer Verbesserung der Positionsbestimmung durch Verwendung einer optischen Maus als weiteren Sensor. Darüber hinaus können weitere reaktive Verhalten entwickelt und Komponenten zur Lokalisierung und Navigation integriert werden. Auch eine Integration der Kamera wäre sinnvoll, allerdings gibt es zum Zeitpunkt der Erstellung dieser Arbeit Probleme mit

der Hardware. Die Bilder der Kamera könnten für eine Steuerung des Roboters durch menschliche Anwender über ein Web-Interface eingesetzt werden. Schließlich könnte an einer deliberativen Komponente gearbeitet werden, die eine aufgabenorientierte Steuerung des Pioneers ermöglicht. Über die Remote-Schnittstelle ist außerdem eine Kommunikation mit anderen Robotern denkbar.

Literaturverzeichnis

[1] ActivMedia. *ARIA 2.2-0 for Windows Data Sheets.* `http://robots.activmedia.com/ARIA/download/ARIA-2.2-0.exe`

[2] ActivMedia. *Saphira 8.4-1 for Windows Data Sheets.* `http://robots.activmedia.com/Saphira/download/Saphira-8.4-1.exe`

[3] ActivMedia. *Saphira Behaviour.* `http://robots.activmedia.com/docs/`. 1996

[4] Brooks, R.: A robust layered control system for a mobile robot. In: *IEEE Journal of Robotics and Automation* RA-2 (1986), Nr. 1, S. 14–23

[5] Grunewald, A.: *Odometriedatenverbesserung unter Verwendung einer optischen Maus als Bodenkorrelator [vorläufiger Titel]*, Darmstadt, TU, Diplomarbeit, 2005

[6] Konolige, K. ; Myers, K. L.: The Saphira Architecture for Autonomous Mobile Robots. In: Kortenkamp, David (Hrsg.) ; Bonasso, R. P. (Hrsg.) ; Murphy, Robin (Hrsg.): *AI-based Mobile Robots: Case studies of successful robot systems.* MIT Press, 1996

[7] Konolige, K. ; Myers, K. L. ; Ruspini, E. ; Saffiotti, A.: The Saphira Architecture: A Design for Autonomy. In: *Journal of Experimental and Theoretical AI* (1996)

[8] Murphy, R. R.: *Introduction to AI Robotics.* Cambridge, MA, USA : MIT Press, 2000. – ISBN 0262133830

[9] Rohe, G.: *Erstellung und Implementierung eines Algorithmus zur Kartenerstellung [vorläufiger Titel]*, Darmstadt, TU, Studienarbeit, 2005

[10] Schürr, A.: *Echtzeitsysteme, SS 05*, Darmstadt, TU, Vorlesungsskript, 2005

[11] Stenzel, R.: *Steuerungsarchitekturen für autonome mobile Roboter*, Aachen, RWTH, Dissertation, 2002. – Standort: Monographien: Stenzel; DokNr.: 9704481

[12] Stryk, O. von: *Robotik I, WS 03/04*, Darmstadt, TU, Vorlesungsskript, 2004

[13] Vollrath, C.: *Entwicklung und Analyse einer Programmierschnittstelle für eine Pioneer-2DX-Plattform unter Berücksichtigung nicht-echtzeitfähiger Softwareumgebungen*, Darmstadt, TU, Diplomarbeit, 2004

[14] Vollrath, U. R.: *Entwicklung und Analyse einer Programmierschnittstelle für eine Pioneer-2DX-Plattform unter Berücksichtigung nicht-echtzeitfähiger Softwareumgebungen*, Darmstadt, TU, Diplomarbeit, 2004

Anhang A

Quellcodes

Die folgenden Quellcodes sind ausgewählte Beispiele zur Veranschaulichung wie die entwickelte Steuerungsarchitektur eingesetzt werden kann.
Die Testanwendung zeigt, wie eine deliberative Komponente auf den reaktiven Steuerungszyklus zugreifen kann. Die weiteren Codebeispiele zeigen wie Verhalten implementiert werden können. Dabei wird von allen Verhalten nur die Methode `fire()` vorgestellt, welche die Funktionalität eines jeden Verhaltens implementiert und in jedem Steuerungszyklus einmal ausgeführt wird. Die Klassen mit der Implementierung der Verhalten sind im Package `tud.sim.pioneer.controlArchitecture.behaviours` zu finden. Die Bezeichnungen der Variablen und Methoden sind weitgehend selbsterklärend. Das Verständnis wird durch Kommentare unterstützt.
Der komplette Quellcode, der im Rahmen dieser Arbeit entwickelt worden ist, und die dazugehörige Dokumentation können auf der beigelegten CD eingesehen werden.

A.1 Testanwendung

```
/**
 * Testanwendung:
 * Ein einfaches Beispiel, welches zeigen soll,
 * wie eine deliberative Komponente die reaktive
 * Steuerung beeinflussen kann.
 *
 * Die Anwendung lässt den Roboter 4m vor fahren,
 * und anschließend wieder zum Ausgangspunkt zurück.
 */
public class DemoDelibControlApp {

    /**
     * Starte die Testanwendung auf dem lokalen Computer
     * (Vewendet den Simulator)
     */
    public static void main(String[] args) throws Exception{
        //Erzeuge ein Objekt zur Konfiguration der
        //Steuerungsarchitektur
        ControlCenterStarter ccs=
            new ControlCenterStarter("localHost");
        //Starte keine Verhalten
        ccs.setLoadStandardBehaviours(false);
        //Starte den Steuerungszyklus
        ccs.startControlCenter();

        //Initialisiere MoveTo
        MoveTo moveTo=new MoveTo();
        //Setzte als Ziel (4000mm,0mm)
        moveTo.setGoal_Cartesian(4000,0);
        ccs.getControlCenter().getBehaviourUpdater().add(moveTo);
        //Warte bis der Roboter dort ist.
        while (moveTo.getState()!=Behaviour.STATE_GOAL_REACHED){
            Thread.yield();
        }
        //Setze als neues Ziel (0,0) (Ausgangsposition)
        moveTo.setGoal_Cartesian(0,0);
        //Warte bist der Roboter dort ist
        while (moveTo.getState()!=Behaviour.STATE_GOAL_REACHED){
            Thread.yield();
        }
```

```
        //Beende das Programm
        ccs.stopControlCenter();
        System.exit(0);
    }
}
```

A.2 Move To

```
/**
 * Ausführen des Verhaltens
 */
public void fire() {
    IPosition currentPos = positionInterpreter.getLastPosition();
    if (currentPos == null) return;
    int currentX = currentPos.getX();
    int currentY = currentPos.getY();
    //Richtung und Distanz zum Ziel bestimmen
    goalDistance = MyMath.getDistance(
            currentX, currentY, goalX, goalY);
    goalHeading = MyMath.getAngle(
            currentX, currentY, goalX, goalY)
                - currentPos.getHeading();

    //Fortschritt
    setHasProgress(goalDistance < lastGoalDistance
        || Math.abs(goalHeading) < Math.abs(lastGoalHeading));
    lastGoalDistance = goalDistance;
    lastGoalHeading = goalHeading;
    //Zustand
    if (goalDistance <= goalReachedDistance) {
        setState(STATE_GOAL_REACHED);
        keepPosition = true;
    } else if (goalDistance <= goalNearlyReachedDistance) {
        setState(STATE_GOAL_NEARLY_REACHED);
        keepPosition=true;
    } else setState(STATE_GOAL_NOT_REACHED);

    if (hasGoalReached()) return;

    //Geschwindigkeit
    int velocity;
    //Ziel ist näher als 50cm
    if (goalDistance < 500) {
        velocity = (int) Math.sqrt(goalDistance * 100);
    }
    //Ziel ist weiter weg als 50cm
    else {
        velocity = (int) Math.sqrt(goalDistance * 100 * 2);
    }
```

```
    //Ziel erreicht
    if (goalDistance <= goalReachedDistance) {
        velocity = 0;
    }

    //Wenn die Richtung zum Ziel
    //mehr als 60° beträgt, soll der Roboter auf der
    //Stelle drehen.
    if (Math.abs(goalHeading) > 60 && keepPosition) {
        velocity = 0;
    } else if (Math.abs(goalHeading) > 50) {
        velocity = Math.max(velocity, 120);
        keepPosition = false;
    } else {
        keepPosition = false;
    }
    //Setze Aktionen für Richtung und Geschwindigkeit
    setRelHeading(goalHeading);
    setVelocity(velocity);
}
```

A.3 Avoid Front

```
/**
 * Ausführen des Verhaltens
 * (Vereinfacht)
 */
public void fire() {
    //Ermittle die aktuelle Position & Geschwindigkeit
    IPosition currentPos = positioninterpreter.getLastPosition();

    setGoalReached(true);

    //Ermittle die Hindernisse im Frontbereich
    ObstaclePoint[] obstacles = obstacleInterpreter.
        getObstaclesArc(
            IObstacleInterpreter.START_ANGLE_FRONT,
            IObstacleInterpreter.WIDTH_FRONT_REAR, 0);
    //Wenn keine Hindernisse im Frontbereich sind
    if (obstacles == null) return;

    //Ermittle die Ausweichrichtung
    //Hindernisse die näher sind werden stärker berücksichtigt
    double x = 0;
    double y = 0;
    for (int i = 0; i < obstacles.length; i++) {
        if (obstacles[i].getDistance() < maxDist) {
            x += obstacles[i].getRelativeCoord().x
                / Math.pow(obstacles[i].getDistance(), 2);
            y += obstacles[i].getRelativeCoord().y
                / Math.pow(obstacles[i].getDistance(), 2);
        }
    }

    //Falls kein Hindernis näher als maxDist ist
    if (x == 0 && y == 0) return;

    //Lege die Ausweichrichtungrichtung auf +90° oder -90° fest
    int heading = (int) Math.toDegrees(Math.atan2(y, x));
    if (heading >= 0) heading = -90;
    else heading = 90;
    //Setze die gewünschte Aktion für den Kanal relative Richtung
    //mit Priorität und Stärke des Verhaltens
    setRelHeading(heading);
```

```
    //Distanz zum nächsten Hindernis ermitteln
    int currentMinDist = obstacleInterpreter.getMinDistArc(
            IObstacleInterpreter.START_ANGLE_FRONT,
            IObstacleInterpreter.WIDTH_FRONT_REAR, 0);

    //Zustand
    if (currentMinDist > maxDist) setGoalReached(true);
    else setGoalReached(false);

    //Fortschritt
    if (lastDist > currentMinDist)  setHasProgress(false);
    else setHasProgress(true);

    lastDist = currentMinDist;
}
```

A.4 Avoid Front Velocity

```
/**
 * Ausführen des Verhaltens
 * Dieses Verhalten ist von Avoid Front abgelitten.
 */
public void fire() {
    //Bestimme die aktuelle Geschwindigkeit
    int currentVelocity = positioninterpreter.getLastPosition()
            .getVelocity();
    //Passe die Größe des Bereiches an, in dem
    //AvoidFront auf Hindernisse reagieren soll.
    if (currentVelocity <= 200) {//niedrige Geschwindigkeiten
        setMaxDistance(900);
    } else if (currentVelocity >= 400) {//hohe Geschwindigkeiten
        setMaxDistance(1300);
    else {//Dazwischen lineares Ansteigen
        //2 = (1300-900)/200
        setMaxDistance( 2*(currentVelocity-200)+900);
    }
    //Führe Avoid Front aus
    super.fire();
}
```

A.5 Avoid Side

```
// Ausführen des Verhaltens
public void fire() {
    //Initialisieren der Interpreter
    if (obstacleInterpreter == null)
        obstacleInterpreter = (IObstacleInterpreter) getStateReflector()
            .getInterpreter(IObstacleInterpreter.NAME);
    if (positionInterpreter == null)
        positionInterpreter = (IPositionInterpreter) getStateReflector()
            .getInterpreter(IPositionInterpreter.NAME);
    //Ermittle aktuelle Position & Geschwindigkeit
    IPosition currentPos = positionInterpreter.getLastPosition();
    if (currentPos.getVelocity() < minVelocity) return;

    //Ermittle minimalen Abstand zu Hindernissen auf der Seite
    int currentDist = obstacleInterpreter.getMinDistArc(startAngle,
            IObstacleInterpreter.WIDTH_SIDE_BIG, 0);
    //Zustand
    if (currentDist > maxDist) {
        lastDist = maxDist;
        setGoalReached(true);
    } else setGoalReached(false);
    if (hasGoalReached()) return; //Ziel erreicht

    //Ausweichrichtung
    int heading = avoidHeading;

    //Fortschritt
    if (currentDist > lastDist) {
        setHasProgress(true);
        //dann bewegt sich der Roboter schon in die richtige Richtung
        //=> nur noch kleine Korrektur
        //Ein Problem ist bei Sonarwerten unterhalb von ca. 60cm
        //(40cm Abstand vom Sonar) wachsen die Werte wieder an,
        //obwohl sich der Roboter dem Hindernis weiter nähert
        heading = avoidHeadingSmall;
    } else setHasProgress(false);
    lastDist = currentDist;

    //Setze die Aktion für den Kanal Richtungsänderung
    setRelHeading(heading);
}
```

A.6 Avoid Side Box

```
/**
 * Ausführen des Verhaltens
 */
public void fire() {
    //Initialisieren der Interpreter
    if (obstacleInterpreter == null)
        obstacleInterpreter = (IObstacleInterpreter)
            getStateReflector().getInterpreter(
                IObstacleInterpreter.NAME);
    if (positionInterpreter == null)
        positionInterpreter = (IPositionInterpreter)
            getStateReflector().getInterpreter(
                IPositionInterpreter.NAME);

    //Ermittle Position & Geschwindigkeit
    IPosition currentPos = positionInterpreter.getLastPosition();

    //Ermittle Hindernisabstand
    int currentDist = obstacleInterpreter.getMinDistBox(
            startAngle,minDist, width, 0);
    //Zustand
    if (currentDist == Integer.MAX_VALUE) {
        lastDist = Integer.MAX_VALUE;
        setGoalReached(true);
    } else setGoalReached(false);
    if (hasGoalReached()) return;

    //Ausweichrichtung festlegen
    int heading = avoidHeading;
    //Fortschritt & Richtung
    if (currentDist > lastDist) {
        //dann bewegt er sich schon in die richtige Richtung
        //=> nur noch kleine Korrekturen
        setHasProgress(true);
        heading = avoidHeadingSmall;
    } else setHasProgress(false);

    lastDist = currentDist;
    //Setze die Aktion für den Kanal Richtungsänderung
    setRelHeading(heading);
}
```

A.7 Avoid Side Velocity

```
/**
 * Ausführen des Verhaltens
 * Diese Verhalten besitzt die Verhalten
 *  - AvoidSide
 *  - AvoidSideBox
 * und setzt sie abhängig von der Geschwindigket
 * des Roboters ein.
 */
public void fire() {

    if (positionInterpreter.getLastPosition().getVelocity() <= 200) {
        //langsame Fahrt - kleiner Sicherheitsbereich
        activeBehaviour = avoidSide;
    } else {
        //schnelle Fahrt - großer Sicherheitsbereich
        activeBehaviour = avoidSideBox;
    }
    //activeBehaviour wird jetzt genutzt

    //Weitergeben von Priorität und Gewichtung,
    //für den Fall, dass diese durch die deliberative
    //Komponenten geändert wurden
    activeBehaviour.setPriority(getPriority());
    activeBehaviour.setStrength(getStrength());

    //Zurücksetzen des Verhaltens (löscht desiredActions)
    activeBehaviour.reset();

    //Ausführen von AvoidSide bzw. AvoidSideBox
    activeBehaviour.fire();
    //Aktionen des aktiven Verhaltens übernehmen
    addDesiredActions(activeBehaviour.getDesiredActions());
    //Zustand und Fortschritt übernehmen
    setState(activeBehaviour.getState());
    setHasProgress(activeBehaviour.hasProgress());
}
```

A.8 Avoid Collisions

```
/**
 * Diese Methode verarbeitet die Aktionen,
 * die aus den Verhalten der Gruppe ermittelt worden sind.
 *
 * Diese Gruppe nutzt folgende Verhalten
 *  - Avoid Front Velocity,
 *  - Avoid Side Velocity (für links und rechts),
 *  - Obstacle Velocity Delimiter
 */
protected void createDesiredActions(int velocity, int heading,
    DesiredActionContainer resolvedDesiredActions) {
    //Wenn keine Aktionen ermittelt worden sind, tue nichts
    if (resolvedDesiredActions == null ||
            resolvedDesiredActions.isEmpty())
        return;

    //Sonst, ermittle die gewünschten Aktionen,
    //und gebe sie mit der eigenen Priorität und
    //Gewichtung weiter.
    Iterator i = resolvedDesiredActions.iterator();
    DesiredAction resolvedDesired;
    while (i.hasNext()) {
        resolvedDesired = ((DesiredAction) (i.next()));
        if (ArchitectureConsts.MAX_VELOCITY_CHANNEL.equals(
                resolvedDesired.getChannelName())) {
            //max. Geschwindigkeit
            addDesiredAction(resolvedDesired);
        } else if (ArchitectureConsts.MIN_VELOCITY_CHANNEL
                .equals(resolvedDesired.getChannelName())) {
            //min. Geschwindigkeit
            addDesiredAction(resolvedDesired);
        } else {
            //Geschwindigkeit und Richtungsänderung
            PriorityDesiredAction pda =
                    (PriorityDesiredAction) resolvedDesired;
            pda.setPriority(getPriority());
            pda.setStrength(getStrength());
            addDesiredAction(pda);
        }
    }
}
```