

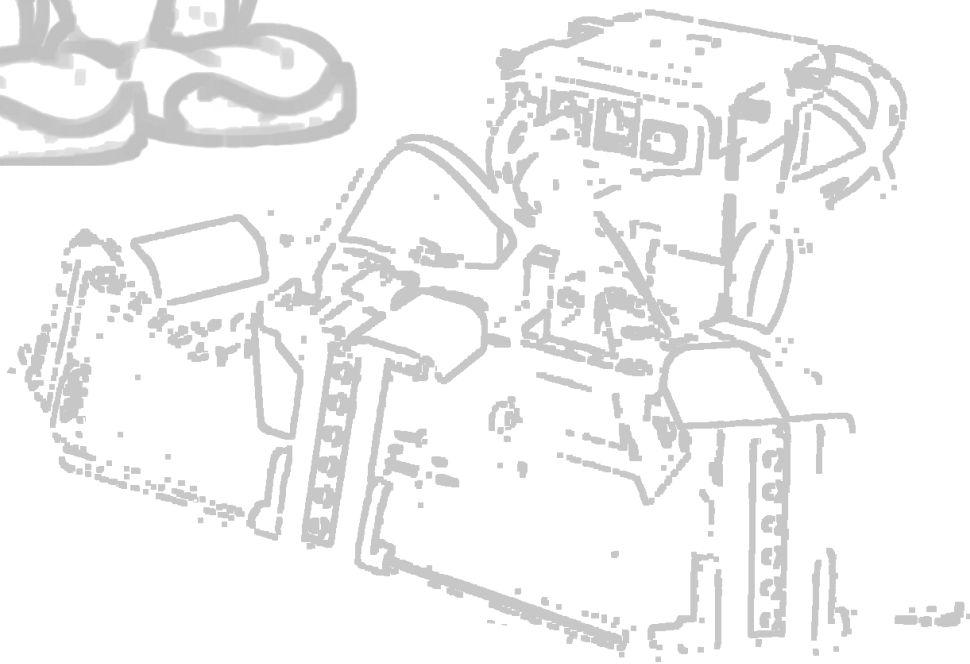


Dokumentation

Semesterprojekt - Softwaretechnik für autonome Roboterteams

Jens Bork, Denis Erfurt, Lorenz Fichte, Robert Fritz, Josef Hufnagl, Sebastian Günther, Sven Schröder, Jonathan Sielhorst und Cordt Voigt

Berlin, Sevilla, Zuerich; den 10.10.2012



Inhaltsverzeichnis

I	Allgemein	2
1	Vorwort	3
2	Einleitung	4
3	Aufgabenstellung	5
3.1	Formulierung der Aufgabe	5
3.2	Verfügbare Ressourcen	5
4	Anforderungserhebung	7
4.1	Systemanforderungen	7
4.2	Benutzeranforderungen	8
5	Projekt- / Zeitplanung	10
5.1	Gruppenaufteilung	10
5.2	Grobe Ablaufplanung	10
II	Projektumsetzung	11
6	NXT	12
6.1	Entwurf	12
6.1.1	Software	12
6.1.2	Idee 1 → Modell 1	13
6.1.3	Idee 2 → Modell 2	14
6.1.4	Idee 3 → Modell 3	15
6.1.5	Fazit und Entscheidung	17
6.2	Kommunikation	17
6.2.1	Bluetooth	17
6.2.2	Kommunikationsprotokoll PC ↔ NXT	17
6.2.3	Kommunikation mit dem MCC	18
6.3	Logik	18
6.3.1	Explorationsalgorithmen	19
6.3.2	GoToPoint	20
III	Fazit	22

I Allgemein

1	Vorwort	3
2	Einleitung	4
3	Aufgabenstellung	5
3.1	Formulierung der Aufgabe	5
3.2	Verfügbare Ressourcen	5
4	Anforderungserhebung	7
4.1	Systemanforderungen	7
4.2	Benutzeranforderungen	8
5	Projekt- / Zeitplanung	10
5.1	Gruppenaufteilung	10
5.2	Grobe Ablaufplanung	10

Kapitel 1

Vorwort

Bei der vorliegende Projektdokumentation handelt es sich um die abschließende Dokumentation des Semesterprojektes “Softwaretechnik für autonome Roboterteams” im Sommersemester 2012 am Institut für Informatik der Humboldt-Universität zu Berlin. Das Projekt ist verpflichtender Bestandteil zum Erlangen des Bachelor of Science am Institut.

Diese Dokumentation wurde Frau Prof. Dr. sc. nat. Verena Hafner und Herrn Prof. Dr. rer. nat. habil. Bernd-Holger Schlingloff, den Betreuern des Projekts, vorgelegt. Durchgeführt wurde das Projekt von Jonathan Sielhorst, Jens Bork, Sven Schröder, Lorenz Fichte, Denis Erfurt, Cordt Voigt, Robert Fritz, Josef Hufnagl und Sebastian Günther. Außerdem wurde das Projekt von Marcus Scheunemann vom Lehrstuhl Kognitive Robotik betreut. In der Studienordnung ¹ ist das Semesterprojekt folgendermaßen beschrieben: “[...] Studierende üben die Fähigkeit, in einem Team ein komplexes System, das eine gegebene Aufgabenstellung löst, in Hard- und/oder Software zu entwerfen, zu entwickeln, zu testen und zu dokumentieren sowie die Ergebnisse in geeigneter Form zu präsentieren. Sie erlangen Kenntnisse über die typischen Probleme bei Projekten mit mehr als 2 Beteiligten. Sie erhalten die Fähigkeit zur selbstkritischen Präsentation des Erreichten und der vorgenommenen Entscheidungen.”

Diese Dokumentation soll also nicht nur den Zweck haben das Problem und die Lösungsstrategie zu beschreiben und zu dokumentieren, sondern auch die während des Projekts aufgetretenen Probleme, Herangehensweisen und Schwierigkeiten beleuchten und somit anderen Studierenden die Möglichkeit geben daraus zu profitieren. Berlin, Sevilla, Zürich, den 10.10.2012

¹<http://www.informatik.hu-berlin.de/institut/dokumente/ordnungen>

Kapitel 2

Einleitung

Das in den nachfolgenden Kapiteln beschriebene Projekt wurde in einer vollständig heterogenen Soft- und Hardwareumgebung durchgeführt, es wird daher weitestgehend auf betriebssystem-, software- oder hardware-spezifische Beschreibungen verzichtet - da es jedoch auch Teilaufgaben gab, bei denen ein homogenes Umfeld maßgeblich für einen korrekten Ablauf war, werden diese Notwendigkeiten gesondert erläutert. Insbesondere die Arbeitsumgebungen, der in diesem Projekt verwendeten Roboter, werden genauer erläutert.

Die Programmiersprache die durchgehend verwendet wird ist Python in der Version 2.6; lediglich bei der hardware-nahen Ansteuerung einer Robotergruppe wurde aus pragmatischen Gründen NXC (Not exactly C,¹) verwendet. Verwendete anwendungsspezifische Begriffe werden gesondert im Glossar erläutert, eine grundsätzliche Verständnis der Themengebiete Informatik und Programmierung wird allerdings vorausgesetzt.

¹<http://bricxcc.sourceforge.net/nbc/>

Kapitel 3

Aufgabenstellung

3.1 Formulierung der Aufgabe

Die grobe Aufgabenstellung, wie sie ursprünglich ausgeschrieben war lautet: “Bei Erkundungsaktionen in für Menschen schwer zugänglichen Gebieten - etwa in Katastrophengebieten wie der Region Fukushima - können künftig Teams von autonomen Robotern eingesetzt werden, die miteinander kooperieren. Ein Flugroboter erkundet dabei das Gebiet aus der Luft und gibt den am Boden agierenden Rettungsrobotern Lage- und Geländeinformationen. Damit solch ein Szenario reibungslos funktioniert, müssen eine Reihe software-technischer Probleme gelöst werden, von der Modellierung der Anforderungen über die Implementierung der Kommunikation und Sensorverarbeitung bis hin zur Konstruktion geeigneter Sensorik, Motorik und Verhalten der mobilen Plattformen. In diesem Semesterprojekt soll ein autonomes Roboterteam, bestehend aus einem Quadroptopter, humanoiden Nao-Robotern und mehreren NXT-Robotern, entworfen und realisiert werden, welches eine kooperative Aufgabe selbständig in schwer zugänglichem Territorium durchführt. Schwerpunkt liegt dabei auf der Modellierung und dem Entwurf der Software für das verteilte System, welche nachweisbar bestimmte Korrektheitseigenschaften erfüllen muss. Begleitend zum Projekt werden das Seminar *Schwarmverhalten* und die Vorlesung *Software-Verifikation* angeboten. Die Teilnehmerzahl ist begrenzt. Von den Teilnehmern werden Vorkenntnisse in Programmierung und verteilten Systemen vorausgesetzt. Die Themen sind eng mit den Forschungsarbeiten der LS Kognitive Robotik und SVT verbunden und können zu Abschlussarbeiten führen. Besonders wichtig ist die Zusammenarbeit im Team.”¹

Im Verlauf der ersten Treffen der Gruppenmitglieder mit den Professoren wurde diese grobe Aufgabenstellung konkretisiert, indem verschiedene Möglichkeiten, das Problem zu lösen, diskutiert wurden. Einerseits wurde zunächst von einem Flugroboter abgesehen, da die Auswertung bewegter Bilddaten bereits ein sehr anspruchsvolles und umfangreiches Problem für sich allein ist - optional konnte dieser bei ausreichenden Ressourcen noch hinzu genommen werden. Das Gebiet indem der Zielbereich gefunden werden soll, kann durch Wände oder andere glatte Oberflächen begrenzt werden; die Hindernisse sollen durch Ziegel oder ähnliche, gleich beschaffene, Steine modelliert werden. Auch hier ist eine Anpassung des Schwierigkeitsgrades durch die Anzahl der Steine und die Größe des Spielfelds möglich. Eine zeitliche Beschränkung besteht automatisch durch die Akkulebensdauer der Roboter (s. dazu Abschnitt Nao, bzw. NXT). Die erste Idee für die allgemeine Vorgehensweise ist, die NXT's für die Erkundung der Umgebung zu verwenden, die dadurch erhaltenen Daten durch ein steuerndes System auszuwerten und anschließend zu benutzen, um den NAO zum Ziel zu führen. Die genauen Rahmenbedingungen durch die Aufgabe und die Einschränkungen und Anforderungen an das System und die Roboter sind in der Anforderungserhebung formuliert (siehe Kapitel 4 auf Seite 7).

3.2 Verfügbare Ressourcen

Für das Projekt sind laut Studienordnung 12 SP angesetzt - dies entspricht etwa einem Aufwand von 360 Stunden, also etwa 25 bis 30 Stunden pro Person pro Woche. Zusätzlich haben Frau Professor Hafner, Herr Professor Schlingloff und Marcus Scheuenemann einmal wöchentlich an einer gemeinsamen Sitzung von 90 Minuten teilgenommen und waren darüber hinaus auch während des gesamten Semesters als Ansprechpartner verfügbar. Die für das Projekt eingeplanten Roboter waren zwei NAO's, sowie mehrere NXT-Bausätze, die je nach Bauart ausreichend für 3-5 Roboter sind. Für Test- und Arbeitszwecke wurde

¹<https://goya3.informatik.hu-berlin.de/goyacs/course/showCourseDetails.do?id=8363&caller=overview>

ein Raum im Fraunhofer Institut in Berlin-Adlershof zur Verfügung gestellt, der uneingeschränkt genutzt werden konnte. Weiterhin wurden zur Ausführung und Entwicklung der Software die privaten Rechner der Studenten verwendet - Lizenzen für eine Python-Entwicklungsumgebung wurden von der Universität zur Verfügung gestellt.

Kapitel 4

Anforderungserhebung

Die Anforderungserhebung wurde aus der groben Aufgabenstellung (s. dazu “Aufgabenstellung”) und einer Einigung der Projektteilnehmer über den konkreten Ablauf der Mission erstellt. Sie ist in System- und Benutzeranforderungen gegliedert maßgeblich für das gesamte Projekt:

4.1 Systemanforderungen

1. NAO

- 1.1. Der NAO soll helfen können die NXT’s zu kalibrieren
 - 1.1.1. Die NXT’s tragen TAGs die gemäß der Bedienungsanleitung vom NAO optisch erkennbar sind, wenn der NXT in einem Abstand von 30 - 100 cm vom NAO ist
 - 1.1.2. Der NAO muss das ganze ihm mögliche Sichtfeld absuchen können in dem er im Stehen sein Kopf bewegt
 - 1.1.3. Wenn ein NAO ein freies Sichtfeld auf den NXT hat, sendet er dem MCC auf Anfrage Informationen über die relative Lage, sonst muss der NAO dem MCC den Fehler, dass der NXT nicht gesehen werden kann, melden
- 1.2. Der NAO muss sich zu Beginn nach dem Aufstehen und wenn er hinfällt mit Hilfe der NXT’s auf 2 cm genau kalibrieren können
 - 1.2.1. Der NAO hat ein freies Sichtfeld auf 2 NXT’s, sonst muss der NAO dem MCC den Fehler, dass er einen oder beide NXT’s nicht sehen kann, melden
 - 1.2.2. Die NXT’s sind in einem Abstand von 30 - 100 cm, sonst muss der NAO dem MCC den Fehler, dass ein oder beide NXT’s nicht im gewünschten Radius sind, melden
- 1.3. Der NAO muss sich anhand von Orientierungspunkten (Steine, NXT’s, Tags??) einem gegebenen Pfad mit einem Fehler von höchstens 5 cm folgen können

2. NXT

- 2.1. Der NXT muss zu einem gegebenen Punkt fahren können mit einer maximalen Abweichung von 10 cm
- 2.2. Der NXT darf für 1 m Luftlinie maximal 1 min brauchen
- 2.3. Der NXT muss das Ziel identifizieren können wenn er darüber fährt
- 2.4. Der NXT muss selbstständig Kollisionen verarbeiten können, so dass er anschließend weiter fahren kann
- 2.5. Der NXT muss sich selbstständig, effektiv in einem unbekannten Gebiet bewegen können
- 2.6. Der NXT muss seine Position, den Winkel seiner aktuellen Ausrichtung und ob er sich auf dem Ziel befindet oder nicht laufend an das MCC übermitteln können

3. MCC

- 3.1. Das MCC muss mit 2 NAO’s kommunizieren können
- 3.2. Das MCC muss mit bis zu 5 NXT’s kommunizieren können
- 3.3. Das MCC muss die 5 Zustände *Initial*, *AutonomicExploration*, *GuidedExploration*, *PathVerification* und *NAOWalk* mit folgenden Zustandsübergangsanforderungen verwalten können:

Initial

Tritt nach Beginn der Mission ein

AutonomicExploration

Tritt ein nachdem die NAO's stehen, kalibriert sind und die NXT's ihre absolute Position erhalten haben

GuidedExploration

Tritt nach 5 Minuten des vorherigen Zustands ein

PathVerification

Tritt ein nachdem das Ziel gefunden wurde, ein Pfad dorthin berechnet wurde und 80

NAOWalk

Tritt ein nachdem die vorherige Phase erfolgreich abgeschlossen wurde; sonst wird die *GuidedExploration* wiederholt

- 3.3.1. Das MCC muss den NAO's und den NXT's die aktuelle Missionsphase mitteilen können
- 3.4. Das MCC muss eine Karte aus den Messungen der NXT's generieren können
 - 3.4.1. Das MCC kann die Karten in vereinfachter Form an die Roboter schicken können
 - 3.4.2. Das MCC muss nach einer Kalibrierung die Messfehler mit Hilfe der Abweichung, die die Kalibrierung ergeben hat, verringern können
- 3.5. Das MCC muss mit Hilfe der beiden NAO's einen NXT kalibrieren können
 - 3.5.1. Das MCC muss die Funktion kalibrieren bereit stellen, Die eine Anfrage an die beiden NAO's stellt einen bestimmten NXT zu kalibrieren
 - 3.5.2. Wenn beide NAO's relative Koordinaten für den NXT liefern, dann ermittelt das MCC die absolute Position des NXTs und teilt ihm diese mit
 - 3.5.3. Nach erfolgter Kalibrierung, sendet das MCC dem NXT seine aktuelle Koordinaten.
 - 3.5.4. Das MCC muss den NXT auf 2 cm genau identifizieren können
 - 3.5.5. Wenn ein oder beide NAO's den Fehler melden, dass der NXT nicht in Sichtfeld ist, muss das MCC dem NXT eine neue Position geben können, die das Problem möglicherweise behebt
- 4. NAOWalk
 - 4.1. Das MCC muss einen Pfad zum Ziel bestimmen und diesen an den NAO übermitteln können
 - 4.2. Der NAO muss mit Hilfe von einem NXT mit einem roten Ball zum Ziel laufen können
 - 4.2.1. Der NAO muss dem 1 Meter entfernten NXT mit dem roten Ball folgen können und ihm mitteilen, dass er angekommen ist
 - 4.2.2. Der NAO muss sich das letzte Teilstück zum NXT merken und auf die ehemalige position des NXT laufen können wenn dieser sich weiterbewegt hat.
 - 4.3. Der NAO muss den Becher am Ziel platzieren können

4.2 Benutzeranforderungen

- 1. Der Benutzer muss die Roboter auf initiale Positionen bringen, so dass alle NXT's im Sichtfeld beider NAO's sind
- 2. Der Benutzer muss die Mission starten können
- 3. Die Mission muss bis auf den Systemstart ohne menschlichen Eingriff ablaufen und nach maximal 30 Minuten erfolgreich beendet sein
- 4. Es sollen 2 NAOs und bis zu 5 NXT's verwendet werden
- 5. Das Spielfeld soll eine Größe von 3m * 3m haben
- 6. Auf dem Spielfeld befinden sich Hindernisse
 - 6.1. Die verwendeten Hindernisse sind so zu plazieren, dass mindestens ein möglicher Weg vom NAO zum Ziel existiert
 - 6.2. Die Hindernisse sind von den NXT's erkennbar, d.h. sie dürfen sich nicht verschieben oder umfallen, wenn der NXT dagegen fährt

- 6.3. Die Größe der Hindernisse entsprechen den Kalkziegelsteinen von Hellweg für 89 Cent
- 7. Am Ende der Mission muss ein Becher auf dem Zielpunkt positioniert sein
 - 7.1. Der Durchmesser des Ziels beträgt mindestens 15 cm.
 - 7.2. Im Umfeld von 30 cm um das Ziel befinden sich keine Hindernisse.

Kapitel 5

Projekt- / Zeitplanung

5.1 Gruppenaufteilung

Ausgehend von der Anforderungserhebung kann das Projekt in drei große Teilaufgaben gegliedert werden:

- Implementation des Mission Control Centers
- Konstruktion der NXT's und Implementation der für die Roboter benötigten Routinen
- Implementation der für die NAO's benötigten Routinen

Diese drei Teilaufgaben wurden durchgehend von jeweils einer Gruppe von 2-4 Studenten bearbeitet, wobei ein Student im speziellen die Kommunikation zwischen dem MCC und dem NAO-Roboter bearbeitet hat. Durch diese Einteilung konnte - nachdem die exakten Schnittstellen zwischen den Gruppen, bzw. den Programmen festgelegt wurden - die Projektarbeit auf drei Kleingruppen heruntergebrochen werden. Innerhalb dieser Kleingruppen wurden die anstehenden Aufgaben in der Regel gemeinschaftlich oder zumindest in durchgängiger Absprache durchgeführt.

5.2 Grobe Ablaufplanung

In der Anforderungserhebung ist bereits die zu entwickelnde Funktionalität beschrieben - darauf basierend wird die Mission mit folgendem Ablauf implementiert:

1. Die NXT's erkunden die Umgebung in ausreichendem Maße, so dass für den NAO ein Weg zum Ziel existiert. Die Informationen werden dem MCC übermittelt und in einer Karte visualisiert, um den Ablauf der Mission verfolgen zu können.
2. Das MCC hat die Möglichkeit die auftretenden Ungenauigkeiten zu reduzieren, indem eine Kalibrierung eines NXT's durch einen NAO erfolgt.
3. Nachdem die Karte ausreichend erkundet ist und ein Weg zum Ziel berechnet wurde, wird der NAO, geführt von einem NXT, zum Ziel gebracht um dort den Becher abzustellen. Formal ist der Ablauf der Mission in einer "State-Machine" implementiert (s. dazu "Statusorientierter Ablauf").

II Projektumsetzung

6	NXT	12
6.1	Entwurf	12
6.1.1	Software	12
6.1.1.1	nxc – Not eXactly C	12
6.1.1.2	nxt-python-framework	12
6.1.1.3	hybrider Ansatz	12
6.1.2	Idee 1 → Modell 1	13
6.1.2.1	Idee	13
6.1.2.2	Konstruktion	13
6.1.2.3	Test	13
6.1.2.4	Pros & Cons	14
6.1.2.5	Fazit	14
6.1.3	Idee 2 → Modell 2	14
6.1.3.1	Idee	14
6.1.3.2	Konstruktion	14
6.1.3.3	Test	14
6.1.3.4	Pros & Cons	15
6.1.3.5	Fazit	15
6.1.4	Idee 3 → Modell 3	15
6.1.4.1	Idee	15
6.1.4.2	Konstruktion	16
6.1.4.3	Test	16
6.1.4.4	Pros & Cons	16
6.1.4.5	Fazit	17
6.1.5	Fazit und Entscheidung	17
6.2	Kommunikation	17
6.2.1	Bluetooth	17
6.2.2	Kommunikationsprotokoll PC ↔ NXT	17
6.2.3	Kommunikation mit dem MCC	18
6.3	Logik	18
6.3.1	Explorationsalgorithmen	19
6.3.1.1	Exploration – simple	19
6.3.1.2	Exploration – circle	19
6.3.1.3	Exploration – radar	20
6.3.2	GoToPoint	20

Kapitel 6

NXT

6.1 Entwurf

6.1.1 Software

Beim Entwurf der Software für unseren Teil der Aufgabe hatten wir zwei Dinge zu beachten, die eingeschränkte Rechenleistung¹ des LEGO[®] Mindstorms[®] NXT Brick (im folgenden nur noch Brick genannt) und die durch LEGO[®] begrenzte Anzahl von Robotern auf maximal 4.

6.1.1.1 nxc – Not eXactly C

nxc ist die für LEGO[®] NXT native Programmiersprache mit einem Compiler, welche für diversen Plattformen erhältlich ist. Obwohl die Syntax von nxc der Programmiersprache C ähnlich ist, ist sie in ihrem Umfang erheblich eingeschränkt. Aus dem Fehlen des Pointerkonzept resultiert unter anderem der Verlust auf die Speicherverwaltung direkt einwirken zu können.

6.1.1.2 nxt-python-framework

Das nxt-python-framework² agiert als Interface, welches die von LEGO[®] in dem „Bluetooth Development Kit“ veröffentlichten direkten Kommandos, nutzt um mit der Hardware zu interagieren.

Wie wird dies erreicht?

Mit LEGO[®] NXT ist es möglich, dass sich bei maximal vier NXTs einer zum Master erklärt. Der Master kann nun durch speziell kodierte Befehle die anderen drei NXTs fernsteuern. Dieses Verhalt macht sich das nxt-python-framework zu nutze und täuscht maximal 3 NXTs vor, dass es ein NXT-Master sei. Wenn dies geschehen ist können die NXTs von PC-Seite ferngesteuert werden.

Vorteil: Durch die Nutzung von nxt-python integriert sich die Komponente NXT-Erkunder nahtlos in das übrige System.

Nachteil: Der synchrone Start bzw. Stopp von zwei Motoren ist nur schwerlich realisierbar, da zwei Befehle benötigt würden, die nacheinander verschickt und auf NXT-Seite nacheinander ausgewertet werden würden.

Wegen dem immensen Vorteil der einfachen Integration in das Restsystem und dem Nachteil der asynchronen Ansteuerung von Motoren entstand die Idee die beiden Programmiersprachen (nxc und python) zu kombinieren.

6.1.1.3 hybrider Ansatz

Die Kombination von nxc und nxt-python wurde wie folgt realisiert. Es wurde in einfaches Kommunikationsprotokoll (siehe Abbildung 6.5 auf Seite 18) konzipiert durch welches der Aufruf von in nxc implementierten Funktionen durch nxt-python ermöglicht wird.

¹8-Bit ARM mit 48 MHz Takt, 64KB RAM

²<http://code.google.com/p/nxt-python/>

6.1.2 Idee 1 → Modell 1

6.1.2.1 Idee

Unsere erste Idee bestand im Prinzip aus zwei unabhängigen Ideen. Zum Einen wollten wir ein Fahrgestell konzipieren, das auch bei unwegsamem Gelände eine kontrollierte Bewegung des Explorers ermöglichen würde und zum Anderen wollten wir einen Sensor der schon viele Informationen über die Umgebung sammelt ohne, dass der Explorer jeden Quadratzentimeter abfahren muss.

6.1.2.2 Konstruktion

Die Konstruktion bestand aus einem kettengetriebenen Fahrzeug, welches mit Hilfe eines Radars seine Umgebung wahrnahm (siehe Abbildung 6.1 auf Seite 13). Die Ketten waren dabei fest gespannt um mögliches Schlüpfen der Kette über die Achse zu vermeiden und so eine Ungenauigkeit bei der Fahrt zu vermeiden. Des Weiteren bestanden sie aus Gummi und hatten eine große Auflagefläche zum Boden und somit eine möglichst hohe Reibung zum Boden. Der Radar bestand aus einem Motor der über einige Zahnräder und einer Schnecke einen Ultraschallsensor bewegte.

Verbaute Sensoren und Motoren:

- zwei Motoren für den Antrieb
- ein Motor für den Radar
- ein Kompass-Sensor zur Ermittlung der Ausrichtung des Fahrzeuges
- ein Lichtstärke-Sensor zur Zielfindung

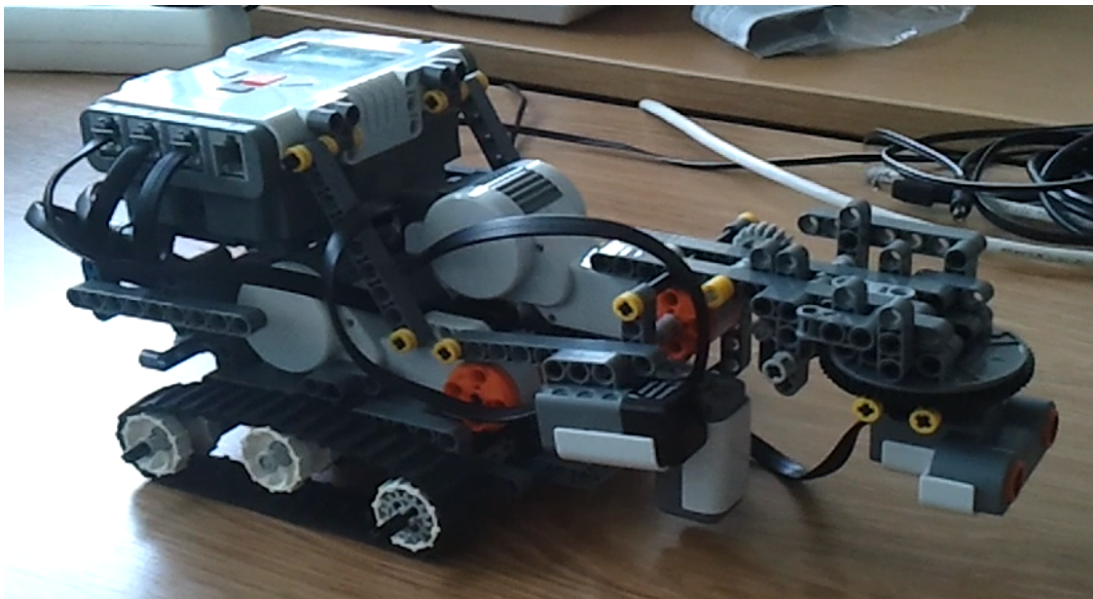


Bild 6.1: Modell 1

6.1.2.3 Test

Getestet haben wir sowohl das Fahrverhalten des Kettenfahrzeuges als auch die Möglichkeit mit dem Radar die Umgebung wahrzunehmen. Hierbei wurde besonders Wert darauf gelegt die Abweichung zum Ist-Wert zu ermitteln.

Beim Fahrzeug war es wichtig herauszufinden ob es geradeaus fahren kann und ja mit welcher Abweichung auf verschiedenen Distanzen (20cm, 50cm, 1m) zu rechnen war. Des Weiteren musste ermittelt werden ob das Fahrzeug in der Lage war sich auf der Stelle zu drehen und dies möglichst genau nach der Vorgabe eines vorher angegebenen Winkels. Hierfür wurden mehrere Tests mit verschiedensten Winkeln durchgeführt bei denen das Fahrzeug sich so oft drehen musste bis es wieder auf seine Ausgangsposition angelangt ist z.B. vier mal eine Drehung um 90° im Uhrzeigersinn. Der Radar wurde auf seine Genauigkeit getestet, als auch auf sein Verhalten bei verschiedenen Materialien und Winkel zu den verschiedenen Objekten.

6.1.2.4 Pros & Cons

Pros:

- geringe Abweichung bei der Fahrt durch die hohe Reibung der Gummiketten
- hohe Geländetauglichkeit durch Kettenantrieb
- kennt das Gebiet vor sich und kann vorausschauend fahren

Cons:

- hohe Abweichung beim drehen
- sehr hohe Abweichung des Ultraschallsensors wenn nicht im 90° zum Hindernis
- hohe Ungenauigkeit beim drehen des Radarkopfes durch das Getriebe

6.1.2.5 Fazit

Die Idee ist alles in allem nicht schlecht aber die Umsetzung mittels LEGO[®] ist nicht praktikabel. Die hohen Ungenauigkeiten und die komplett Aussetzer des Ultraschallsensors lassen uns keine andere Wahl als ein neues Fahrzeug zu entwerfen. Hierbei muss sowohl der Antrieb als auch die Sensorkonstruktion überdacht werden. An einen Einsatz dieses Modells ist nicht zu denken.

6.1.3 Idee 2 → Modell 2

6.1.3.1 Idee

Da wir bei unserer ersten Idee feststellen mussten das ein kettengetriebenes Fahrzeug zu hohe Ungenauigkeiten verursachte, musste hier eine Alternative gefunden werden, welche aber keine Einschränkungen bei der Bewegungsfreiheit des Fahrzeuges macht d.h. möglichst genaues (vorwärts/rückwärts) Fahren und auf der Stelle wenden können.

Auch eine Alternative für den Radar musste her. Hierbei wurde in Kooperation mit dem MCC-Team vereinbart das nicht Hindernisse gefunden werden, sondern davon ausgegangen wird das die gefahrene Strecke des Explores frei ist, sozusagen wurde das Bild der Karte invertiert. Dies ermöglichte uns nur auf Hindernisse reagieren zu müssen und nicht wie vorher Informationen über den Bereiche des Einsatzgebietes zu sammeln.

6.1.3.2 Konstruktion

Unsere Konstruktion 2 (siehe Abbildung 6.2 auf Seite 15) erhielt nun ein 2-Achsen Antrieb, welcher es uns ermöglichte durch gleichzeitiges ansteuern der Motoren gerade Strecken zu fahren, als auch durch entgegengesetztes ansteuern sich auf der Stelle zu drehen. Als Zusatz wurde noch ein Omni-Wheel verbaut, welches dem Gefährt mehr Stabilität verleihen sollte. Um auf Hindernisse reagieren zu können, wurden an der Vorderseite des Fahrzeuges drei Touchsensoren befestigt. Diese sollten auslösen sobald das Fahrzeug vor ein Hindernis fuhr.

Verbaute Sensoren und Motoren:

- zwei Motoren für den Antrieb
- drei Touch-Sensoren um Hindernisse zu finden
- ein Lichtstärke-Sensor zur Zielfindung

6.1.3.3 Test

Auch beim zweiten Model wurden die oben schon beschriebenen Tests für den Antrieb durchgeführt. Die Sensoren wurden in ihrer Anordnung getestet um eine möglichst optimale Anordnung zu finden.

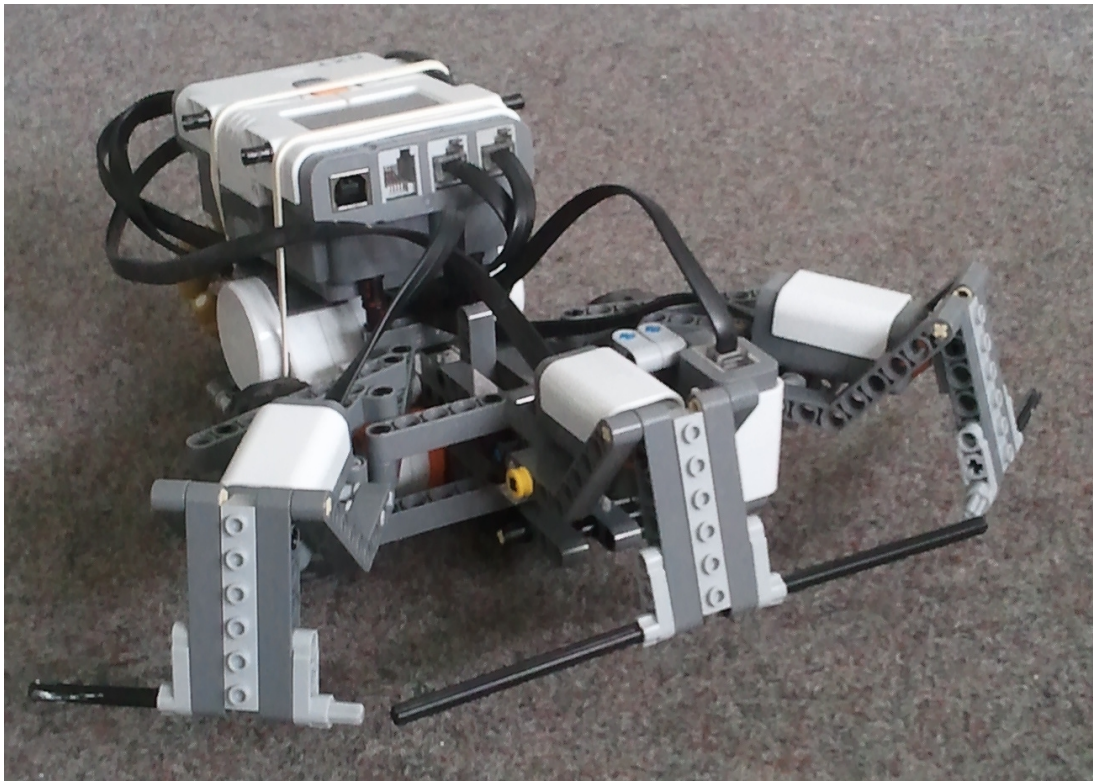


Bild 6.2: Modell 2

6.1.3.4 Pros & Cons

Pros:

- geringe Abweichung beim fahren
- geringe Abweichung beim drehen
- hohe Aussagekraft bzgl. Sensorevents da nur Touch-Sensoren verbaut wurden

Cons:

- keine Aussagen über das Gebiet vor dem Fahrzeug möglich
- Fahrzeug kann nur noch reagieren und kaum intelligent handeln
- Verhaken der Touch-Sensorkonstruktion

6.1.3.5 Fazit

Bei der zweiten Konstruktion überzeugt das Antriebsmodell durch seine geringen Abweichungen und seine Agilität. Die Sensoranordnung hingegen lässt noch einige Wünsche offen. Die fehlende Voraussicht ist dabei das größte Manko. An einen Einsatz dieses Model ist ebenfalls nicht zu denken. Eine Verbesserung des Sensorkonstrukts ist nötig.

6.1.4 Idee 3 → Modell 3

6.1.4.1 Idee

Das in Idee 2 entwickelte Antriebsmodell hat uns überzeugt. Im dritten Anlauf wird sollte nur noch die Sensoranordnung überdacht werden. Dem Bereich vor dem Fahrzeug sollte dabei mehr Beachtung geschenkt werden, um bessere und intelligentere Entscheidungen treffen zu können.

6.1.4.2 Konstruktion

Bei der dritten Konstruktion (siehe Abbildung 6.3 auf Seite 16) wurde das Antriebsmodell der zweiten Konstruktion übernommen. Bei den Sensoren wurde der mittlere Touch-Sensor durch ein Ultraschall-Sensor ersetzt. Dieser ermöglichte es die freie Strecke vor dem Fahrzeug zu ermitteln.

Verbaute Sensoren und Motoren:

- zwei Motoren für den Antrieb
- zwei Touch-Sensoren um Hindernisse zu finden
- ein Ultraschall-Sensor um den Bereich vor dem Fahrzeug zu überblicken.
- ein Lichtstärke-Sensor zur Zielfindung

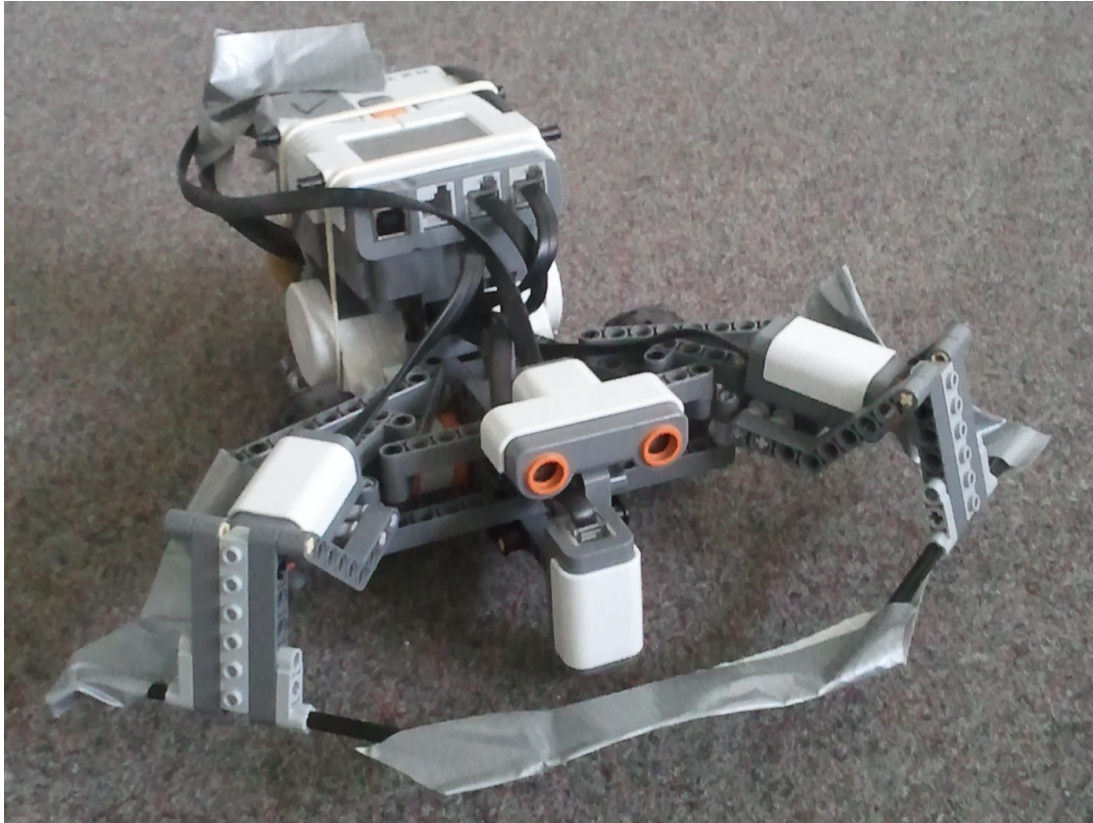


Bild 6.3: Modell 3

6.1.4.3 Test

Auch beim dritten Model wurden alle Antriebtests durchgeführt. Die neue Sensoranordnung wurde mit mehreren Testaufbauten auf ihre Einsatztauglichkeit geprüft. Hierbei wurden mögliche Anordnungen von Hindernissen konstruiert und die Sensorevents ausgewertet.

6.1.4.4 Pros & Cons

Pros:

- geringe Abweichung beim fahren
- geringe Abweichung beim drehen
- freie Strecke vorm Fahrzeug bekannt

Cons:

- Verhaken der Touch-Sensorkonstruktion

6.1.4.5 Fazit

Das dritte Model überzeugt durch seinen Antriebsmodel, sowie durch die verbesserte Sensoranordnung gegenüber des zweiten Models. Aber auch dieses Model ist mit Vorsicht zu benutzen. Die Möglichkeiten des Verhaken der Fahrzeuge ist ein großes Problem das es noch zu beseitigen gibt. An einen Einsatz ist nur bedingt zu denken.

6.1.5 Fazit und Entscheidung

Das erste Model ist für raues Gelände bestens geeignet. Durch seine hohen Abweichungen beim Fahren und des nicht zuverlässigen Ultraschall-Sensors im Radar aber kein Kandidat für die Lösung unseres Problems.

Das zweite Model überzeugt durch seine Genauigkeit beim Fahren und auch die Aussagekraft der Sensoranordnung. Allerdings fehlt uns hier das gewisse etwas um ein möglichst elegante Lösung für die autonome Erkundung eines Gebietes.

Von allen Model schneidet das dritte am besten ab. Es erbt die guten Fahrteigenschaften des zweiten Models und biete uns dazu noch die Möglichkeit durch seine überarbeitete Sensoranordnung möglichst intelligent das Problem zu lösen.

6.2 Kommunikation

6.2.1 Bluetooth

Bluetooth ist ein Funkstandard speziell für die Nahbereichskommunikation. Er wurde in den 1990er-Jahren von der Bluetooth Special Interest Group (SIG) entwickelt und in IEEE 802.15.1 festgeschrieben. Grundsätzlich sind verbindungslose und verbindungsorientierte Übertragungen möglich.

Bei der Konzeptionierung von LEGO[®] Mindstorms[®] NXT hat LEGO[®] die Verbindungsorientierung für ihre Geräte festgelegt.

6.2.2 Kommunikationsprotokoll PC ↔ NXT

Bei einem der Treffen im Rahmen des Semesterprojektes wurde durch den begleitenden Professor angeregt, dass durch das Team NXT sicherzustellen sei, dass alle Steuerkommandos und Antworten (über Bluetooth) auch von der Gegenstelle erhalten werden müssen. Als Denkanstoß verwies er auf den 3-way-handshake des TCP. Dieser Aspekt wurde von uns mit dem Protokoll aus Abbildung 6.4 implementiert, sodass auf jede Nachricht vom Typ „m“ eine Antwort vom Typ „r“ folgte und der Sende diese Typ „r“ Nachricht mit einer abschließenden Typ „a“ Nachricht quittierte. Dieses Protokoll hatte eine Datenflut zur Folge, die kaum zu beherrschen war, da bei überschreiten von Fristen Nachrichten wiederholt gesandt werden mussten.

Da bei LEGO[®] NXT Bluetooth verbindungsorientiert verwendet wird, war die oben beschriebene Herangehensweise nicht nötig, da bei Verbindungsorientiertheit die Zustellung von Nachrichten garantiert wird. Aus diesem Grund wurde das Protokoll auf das in Abbildung 6.5 reduziert.

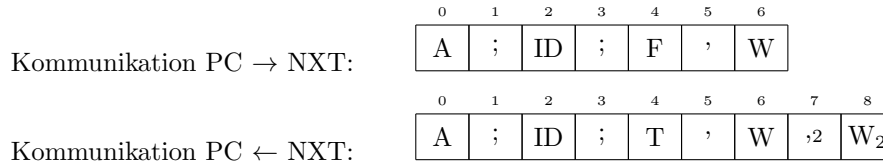
Definitions- und Wertebereich der im jüngsten Protokoll (Abbildung 6.5) verwendeten Platzhalter:

Kommunikation PC → NXT:

F	W	Beschreibung
0	beliebig	Stopp, der NXT in Hardware bricht alle seine Aktionen ab und wartet auf weitere Kommandos
1	0 – MAX_INT	Vorwärtsfahren: bei W = 0 bis Steuerkommando 0 oder Auslösen eines Sensors, bei W > 0 vorwärts fahren für W Grad in Motorumdrehungen
2	1 – MAX_INT	rückwärts fahren für W Grad in Motorumdrehungen
3	1 – MAX_INT	Fahrzeugachse um W Grad nach links drehen
4	1 – MAX_INT	Fahrzeugachse um W Grad nach rechts drehen
5	beliebig	Messung mit dem Ultraschallsensor ausführen

Kommunikation PC ← NXT:

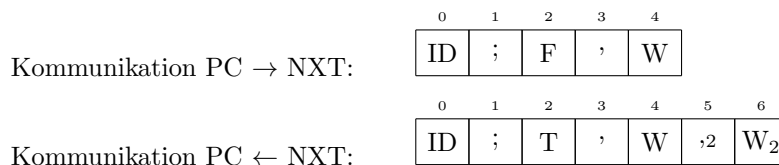
T	W	Beschreibung
1	0 – MAX_INT	ständige Standortmeldung: für W cm vorwärts gefahren (ohne Kollision)
2	1 – MAX_INT	Kollision: W cm vorwärts gefahren, Sensor W_2 hat ausgelöst
3	1 – MAX_INT	Fahrziel erreicht: für W cm vorwärts gefahren (ohne Kollision)
4	beliebig	Drehung/Rückwärtsfahren beendet
5	0 – 255	Ergebnis der Messung mit dem Ultraschallsensor
9	beliebig	Bodensensor hat Ziel gefunden



Legende:

A = Typ der Nachricht (m = Nachricht, r = m erhalten, a = r erhalten)
F = Funktion die aufgerufen werden soll
W = Zahlwert
T = Typ der Antwort
₂ = optional

Bild 6.4: Kommunikationsprotokoll 3-way-handshake



Legende:

F = Funktion die aufgerufen werden soll
W = Zahlwert
T = Typ der Antwort
₂ = optional

Bild 6.5: Kommunikationsprotokoll

6.2.3 Kommunikation mit dem MCC

Für die Kommunikation mit dem MCC kam das Python Framework Twisted³ zum Einsatz. Twisted ist ein sogenanntes „event-driven“ framework, das heißt der Programmierer stellt für Aktionen Callback-Funktionen bereit. Twisted abstrahiert von der darunterliegenden Netzwerkarchitektur und dem verfügbaren Netzwerkprotokollstack (wenn auch nicht vollständig transparent), dies vereinfacht die Programmierung eines Verteilten Systems erheblich.

6.3 Logik

Für die logische Programmierung des „Explorers“ bestand vorwiegend das Problem, dass die Hardware (der Brick) Zeit benötigt um die Aktionen auszuführen, also blockiert, jedoch die Softwarekomponente asynchrones Messaging nutzt. Um diese Hürde zu überwinden wurde bei der Konzeptionierung des Systems darauf geachtet, dass auf jedes Steuerkommando, welches eine Handlung der Hardware hervorruft, nach Abschluss der Handlung eine Antwort des Brick gesendet werden muss. Dass versetzte uns in die Lage mit einer synchronisierten boole'schen Variablen (blockiert = True) die Software solange zu blockieren, bis die Antwort vom Brick eingegangen ist und damit die boole'sche Variable wieder auf False gesetzt wird.

³<http://twistedmatrix.com>

6.3.1 Explorationsalgorithmen

In der Welt der autonomen Rasenmäh- und Staubsaugroboter haben sich vier Algorithmen durchgesetzt⁴:

1. Touch and Go⁵
2. circle
3. radar
4. Wandverfolgung

1 – 3 werden im Folgenden näher besprochen. 4 konnte aufgrund der begrenzten Anzahl an Sensoren nicht implementiert werden und bleibt deshalb außen vor.

6.3.1.1 Exploration – simple

Der erste und auch gleichzeitig einfachste Algorithmus. Der Roboter fährt solange geradeaus bis er auf ein Hindernis trifft. Nach dem Auslösen eines Sensors wechselt der Roboter beliebig seine Fahrtrichtung und fährt wieder geradeaus bis zum nächsten Ereignis. Der Explorer wechselt die Richtung nicht vollständig beliebig, er wählt die Drehrichtung in Abhängigkeit von dem auslösenden Sensor, soll heißen wenn der linke Sensor angeschlagen wurde, dreht sich der Explorer um einen beliebigen Winkel zwischen 30° und 160° nach rechts.

6.3.1.2 Exploration – circle

Wie in Abbildung 6.6 zu sehen ist macht der Explorer eine Bewegungsabfolge in Form einer eckigen Nautiluschnecke. Durch diese Bewegungsfolge wird der Bereich in dem sich der Explorer befindet schnell erkundet.

In Abbildung 6.7 wird an Hand eines Zustandsautomaten gezeigt, wie der Explorationsalgorithmus umgesetzt ist. Es gibt keinen ausgezeichneten Endzustand, was damit begründet ist, dass der Algorithmus von einem externen Zähler abgebrochen wird. Der Abbruch kann in jedem der Zustände geschehen. Man kann erkennen, dass zusätzlich zur reinen Bewegung weitere Zustände und Transitionen eingefügt wurden um Messungen mit dem Ultraschallsensor durchzuführen. Die Messungen dienen als Grundlage zur Berechnung eines Korrekturfaktors für alle Bewegungen.

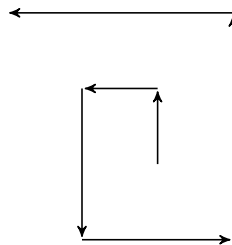


Bild 6.6: Bewegungsmuster Exploration – circle

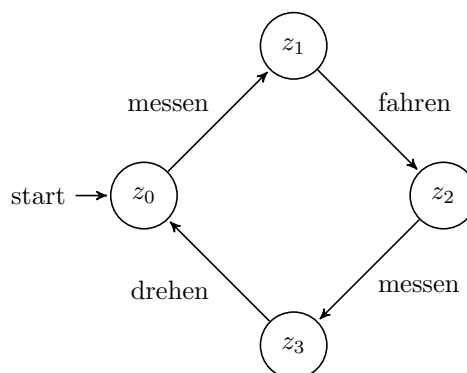


Bild 6.7: state-machine Exploration – circle

⁴Quelle: Roboter mit Mikrocontrollern, Heinz Schmid, Franzis Verlag, 2011

⁵hier simple genannt

6.3.1.3 Exploration – radar

Wie in Abbildung 6.8 zu sehen ist, macht der Explorer eine Bewegungsabfolge in Form einer eckigen Wellenausbreitung. Durch diese Bewegungsfolge wird der Bereich vor dem Explorer schnell erkundet und der Explorer verlässt langsam seinen bisherigen Bereich.

Wie im Abschnitt Exploration – circle wird hier auch ein Zustandsautomat (Abbildung 6.9) zur Darstellung des Algorithmus verwendet. Er unterliegt den gleichen Kriterien wie oben.

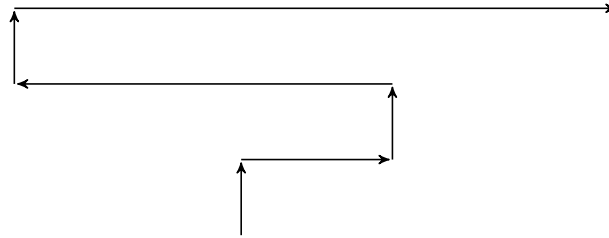


Bild 6.8: Bewegungsmuster Exploration – radar

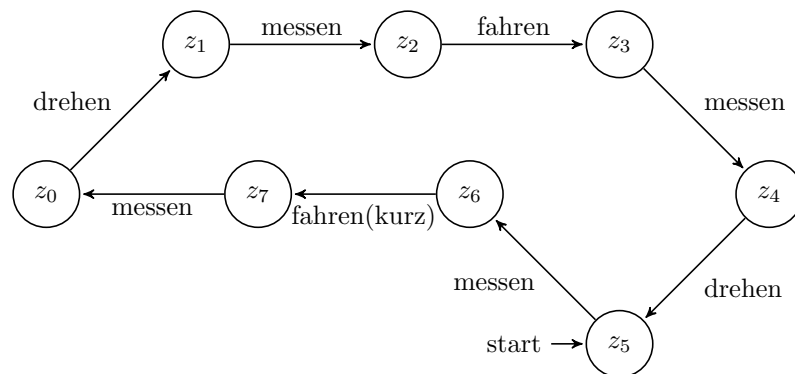


Bild 6.9: state-machine Exploration – radar

6.3.2 GoToPoint

Für die GoToPoint – Funktionalität wurden die zwei Funktionen aus Listing 6.1 und Listing 6.2 benötigt. Mit der Funktion berechnePunkt() kann aus der eigenen Ausrichtung, der gefahrenen Entfernung und dem ehemaligen Standort der neue Standort berechnet werden. Die Funktion berechneVektor() aus Listing 6.2 berechnet zu zwei gegebenen Punkten den verbindenden Vektor, dazu wird erst ein relatives Koordinatensystem geschaffen, dann wird über den Satz von Pythagoras die Entfernung, d.h. der Betrag des Vektor berechnet und danach Winkel berechnet.

Listing 6.1: berechnePunkt()

```
1 def berechnePunkt(ausrichtung, entfernung, standort = {'x':0.0, 'y':0.0}):
2     return {'x': standort['x'] + entfernung * GRAD2CM *
3             math.cos(ausrichtung * (math.pi / 180.0)),
4             'y': standort['y'] + entfernung * GRAD2CM *
5             math.sin(ausrichtung * (math.pi / 180.0))}
```

Listing 6.2: berechneVektor()

```

1  def berechneVektor(standort = {'x':0.0, 'y':0.0}, ziel = {'x': 0.0, 'y': 0.0}):
2      relativ_ziel = {'x': abs(ziel['x'] - standort['x']),
3                      'y': abs(ziel['y'] - standort['y'])}
4      dbg_print("St0: (%d,%d), Z: (%d,%d), rZ: (%d,%d)"%(standort['x'],
5                                                         standort['y'],
6                                                         ziel['x'],
7                                                         ziel['y'],
8                                                         relativ_ziel['x'],
9                                                         relativ_ziel['y']), 1, 0)
10     entfernung = math.sqrt(relativ_ziel['x'] ** 2 + relativ_ziel['y'] ** 2)
11     if relativ_ziel['x'] == 0:
12         winkel = 90
13     elif relativ_ziel['x'] < 0:
14         winkel = math.atan(float(relativ_ziel['y']) / float(relativ_ziel['x']))
15                     (180.0 / math.pi) + 180
16     else:
17         winkel = math.atan(float(relativ_ziel['y']) / float(relativ_ziel['x']))
18                     (180.0 / math.pi) + 360
19     return {'winkel': winkel % 360,
20            'entfernung': entfernung,
21            'rel_x': relativ_ziel['x'],
22            'rel_y': relativ_ziel['y']}

```

III Fazit
