Dokumentation – NXT

Sven Schröder, Jens Bork

29. September 2012

Inhaltsverzeichnis

1 Entwurf **1**
1.1 Software . . . 1
1.1.1 nxc – Not eXactly C . . . 2
1.1.2 nxt-python-framework . . . 2
1.1.3 hybrider Ansatz . . . 2
1.2 Idee 1 → Modell 1 . . . 2
1.2.1 Idee . . . 2
1.2.2 Konstruktion . . . 2
1.2.3 Test . . . 3
1.2.4 Pros & Cons . . . 3
1.2.5 Fazit . . . 3
1.3 Idee 2 → Modell 2 . . . 3
1.3.1 Idee . . . 3
1.3.2 Konstruktion . . . 3
1.3.3 Test . . . 4
1.3.4 Pros & Cons . . . 4
1.3.5 Fazit . . . 4
1.4 Idee 3 → Modell 3 . . . 4
1.4.1 Idee . . . 4
1.4.2 Konstruktion . . . 4
1.4.3 Test . . . 4
1.4.4 Pros & Cons . . . 4
1.4.5 Fazit . . . 5
1.5 Fazit und Entscheidung . . . 5

2 Kommunikation **5**
2.1 Bluetooth . . . 5
2.2 Kommunikationsprotokoll PC ↔ NXT . . . 5
2.3 Kommunikation mit dem MCC . . . 5

3 Logik **5**
3.1 Explorationsalgorithmen . . . 6
3.1.1 Exploration – simple . . . 6
3.1.2 Exploration – circle . . . 6
3.1.3 Exploration – radar . . . 6
3.2 GoToPoint . . . 6

1 Entwurf

1.1 Software

Beim Entwurf der Software für unseren Teil der Aufgabe hatten wir zwei Dinge zu beachten, die mäßige Rechenleistung[1] des LEGO© Mindstorms© NXT Brick (im folgenden nur noch Brick genannt) und die durch LEGO© begrenzte Anzahl von Robotern auf maximal 4.

[1] 8-Bit ARM mit 48 MHz Takt, 64KB RAM

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.

1.1.2 nxt-python-framework

Das nxt-python-framework[2] 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.

1.1.3 hybrider Ansatz

Die Kombination von nxc und nxt-python wurde wie folgt realisiert. Es wurde in einfaches Kommunikationsprotokoll (siehe Abbildung 2 auf Seite 5) konzipiert durch welches der Aufruf von in nxc implementierten Funktionen durch nxt-python ermöglicht wird.

1.2 Idee 1 → Modell 1

1.2.1 Idee

Unsere erste Idee bestand im Prinzip aus zwei unabhängigen Ideen. Zum Einen wollten wir ein Fahrgestell konzipieren, das auch bei unwegsamen Gelände eine kontrollierte Bewegung des Explorer 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.

1.2.2 Konstruktion

Die Konstruktion bestand aus einem kettengetriebenen Fahrzeug, welches mit Hilfe eines Radars seine Umgebung wahrnahm (siehe Abbildung 2 auf Seite 5). 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

[2] http://code.google.com/p/nxt-python/

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 verschieden 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 angegeben Winkels. Hierfür wurden mehrere Test 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.

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

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 eine neues Fahrzeug zu entwerfen. Hierbei muss sowohl der Antrieb als auch die Sensorkonstruktion überdacht werden. An einen Einsatz dieses Models ist nicht zu denken.

1.3 Idee 2 → Modell 2

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 gefahren 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.

1.3.2 Konstruktion

Unsere Konstruktion 0.2 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

1.3.3 Test

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

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

1.3.5 Fazit

Bei der zweiten Konstruktion überzeugt das Antriebsmodel 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.

1.4 Idee 3 → Modell 3

1.4.1 Idee

Das in Idee 2 entwickelte Antriebsmodel 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.

1.4.2 Konstruktion

Bei der dritten Konstruktion wurde das Antriebsmodel 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

1.4.3 Test

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

1.4.4 Pros & Cons

Pros:

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

Cons:

- Verhaken der Touch-Sensorkonstruktion

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.

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.

2 Kommunikation

2.1 Bluetooth

2.2 Kommunikationsprotokoll PC ↔ NXT

anfänglich 3-way-handshake wegen missverständnis

Kommunikation PC → NXT:

0	1	2	3	4	5	6
A	;	ID	;	F	,	W

Kommunikation PC ← NXT:

0	1	2	3	4	5	6	7	8
A	;	ID	;	T	,	W	,2	W_2

```
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
```

Abbildung 1: Kommunikationsprotokoll 3-way-handshake

Kommunikation PC → NXT:

0	1	2	3	4
ID	;	F	,	W

Kommunikation PC ← NXT:

0	1	2	3	4	5	6
ID	;	T	,	W	,2	W_2

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

Abbildung 2: Kommunikationsprotokoll

2.3 Kommunikation mit dem MCC

3 Logik

Problem das der Robo blockiert, das Program aber nicht

3.1 Explorationsalgorithmen

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

1. Touch and Go[3]
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.

3.1.1 Exploration – simple

3.1.2 Exploration – circle

3.1.3 Exploration – radar

3.2 GoToPoint

[3]hier simple genannt