Autonome mobile Kleinroboter

Eine Einführung in Grundkonzepte der Robotik
mit LEGO-MINDSTORMS-Robotern

J. Wachter
Bochum, 19.03.2002

In diesem Papier werden einige Grundbegriffe der Robotik autonomer, mobiler Kleinroboter erläutert. Im Zentrum der Betrachtungen sollen dabei Programmierkonzepte und Konzepte des maschinellen Lernens sein. Umgesetzt werden sollen die vorgestellten Verfahren mittels Kleinrobotern vom Typ LEGO-MINDSTORMS und der Programmiersprache NOT QUITE C (NQC). Der Text ist noch in Arbeit und wird weiter aktualisiert.

1 Allgemeine Einführung

Der Begriff *Roboter* geht auf den tschechischen Schriftsteller Karel Čapeck zurück, der ihn für sein Theaterstück *Rossums Universalroboter* vom tschechischen Wort *robota* für Frondienst ableitete.
Im Allgemeinen verstehen wir unter einem Roboter eine Maschine, die dem Aussehen eines Menschen nachgebildet ist und Tätigkeiten eines Menschen nachahmt. Während zu Beginn der Roboterentwicklung das Interesse im Fordergrund stand, dass Roboter für den Menschen körperlich aufwendige oder gefährliche Aufgaben übernehmen, so ist es nun auch von Interesse, dass Roboter Tätigkeiten übernehmen, die kognitive Fähigkeiten verlangen.

1.1 Historische Entwicklung der Robotik

Ausgangspunkt für die Entwicklung der Robotik waren im Wesentlichen zwei zentrale Technologien.

- Telemanipulatoren
- Numerische Steuerung von Werkzeugmaschinen

Bei den *Telemanipulatoren* handelt es sich um vom Menschen ferngesteuerte Apparaturen, die für spezielle Tätigkeiten eingerichtet wurden. Häufig handelt es sich dabei um Arm- und Greiferkonstruktionen deren Bewegungen durch eine bedienende Person gesteuert werden, in dem diese die auszuführenden Bewegungen über eine spezielle Steuereinheit direkt vorgibt. Bekannte Beispiele für Telemanipulatoren findet man zum Beispiel im Bereich der Manipulation von radioaktivem Material.
Bei der numerischen Steuerung von Werkzeugmaschinen (NC Numerical Control) verfolgt man das Ziel, die Bewegungen einer Werkzeugmaschine präszise in Bezug auf ein Koordinatensystem vorzugeben. Dabei wurden solche Maschinen zunächst in der Weise programmiert, dass das Programm die anzufahrenden Koordinaten und die auszuführenden Aktionen in sequenzieller Weise vorgab. Die Programmdaten wurden dabei auf Lochstreifen codiert. Diese Technik wurde 1952 von einer eine Arbeitsgruppe des *Massachusetts Institute of Technologie* (MIT) eingeführt. Zur Programmierung dieser NC-Maschine wurde die Programmiersprache APT (Automatic Programmed Tools) entwickelt und 1961 veröffentlicht.
Der erste kommerzielle Roboter wurde 1959 von der *Planet Corporation* für einfache Handhabungsaufgaben wie z.B. Punktschweißen vertrieben. In den kommenden Jahren wurden verschiedenste Roboter industriell eingesetzt.
1984 wird von der *Waseda University, Tokio* ein *antropmorpher* (menschenähnlicher) Roboter namens WABOT-2 mit der Fähigkeit des Keyboardspielens mit 2 Händen, 2 Beinen und Kamera vorgestellt.

2 Klassifikation von Robotern

Es gibt vielfältige Möglichkeiten, Roboter zu klassifizieren.
Eine Möglichkeit, ist die Klassifizierung nach Mobilität, äußerem Erscheinen und Unabhängigkeit. Wir gelangen damit zu folgenden Kategorien:

Stationäre Roboter

Stationäre - also an einem Ort festgelegte - Roboter haben folgende Aufgaben:

- Kollisionsfreies Bewegen von Arm und Greifern
- Handhabung und Manipulation von Objekten
- Erkennung und Verfolgung von Objekten
- Planung von Bewegungs- und Handhabungsfolgen und deren Durchführung

Mobile Roboter

Mobile Roboter - also Roboter, die den Ort wechseln können - haben folgende Aufgaben.

- Bewältigung verschiedenen Transportaufgaben
- Bewegung in unterschiedlich stark strukturierten und dynamischen Umgebungen
- Erkunden unbekannter Umgebungen
- Messungen, Probennahmen, Sichtungen durchführen

Humanoide Roboter

Humanoide - also menschenähnliche - Roboter haben folgenden Aufgaben.

- Nachbildung menschlicher Fähigkeiten, wie Fortbewegung, Manipulation etc.
- Kooperation mit dem Menschen

Autonome Roboter

Autonome Roboter sind Roboter, die folgende Anforderunge erfüllen.

- Unabhängigkeit von menschlichen Bedienern
- Weitgehend Resourcenunabhängig (z.B. eigene Energieversorgung)
- Eigenständige Handlung aufgrund der Programmvorgaben

3 Aufbau und Teilsysteme von Robotern

Ein Roboter ist ein komplexes System aus Hardware- und Softwareelementen. In diesem Abschnitt sollen jene Elemente behandelt werden, die für die Konstruktion mobiler Kleinroboter von Relevanz sind. Die folgenden Ausführungen halten sich im Wesentlichen an [5]

3.1 Mechanik

Die mechanische Struktur eines Roboters bestimmt maßgeblich seine Fähigkeiten in seiner Umwelt zu agieren. Bei mobilen Kleinrobotern, die verschiedenste Aufgaben zu lösen haben, unterscheiden wir:

- **Fahrzeug:** Bewegung des Roboters
- **Roboterarm:** Führung des Effektors
- **Hand, Greifer:** Endeffektor, Manipulation von Gegenständen

Fahrzeug

- **Freiheitsgrade:** Ein auf dem Boden bewegliches Fahrzeug besitzt drei Freiheitsgrade:
  - Translation auf der Bodenfläche in x- und y-Richtung
  - Rotation um die senkrecht zur Bodenfläche stehend z-Achse

  Ein Fahrzeug im Weltraum oder unter Wasser kann bis zu sechs Freiheitsgraden besitzen:
  - Translation in Richtung jeder Achse (y,x,z)
  - Rotation um jede der Achsen x,y,z
- Bei einigen Anwendungen übernehmen Zusatzachsen (Bodenschienen) oder Portale (Laufkräne unter der Decke) die Funktion des Fahrzeugs (1 Freiheitsgrad)
- **Fortbewegungsmittel** können Räder, Ketten oder Beine sein. Kombinationen sind prinzipiell möglich.
  - **Räder:** Roboter mit Rädern haben eine einfache Mechanik und lassen sich leicht zusammenbauen. Häufig findet man bei Kleinrobotern Konstrutionen, bei denen die Räder der beiden Roboterseiten (rechts/links) getrennt über Motoren angetrieben werden. Dadurch ist eine gute Manövierfähigkeit gewährleistet: Schnelle Drehungen auf der Stelle sind durch gegenläufige Drehung der Räder möglich. Selten findet man Antriebskonstruktionen, bei denen eine (starre) Achse den Vortrieb bewirkt, während eine „Lenkung“den Richtungswechsel ermöglicht, wie das bei Automobilen mit Heckantrieb üblich ist. Der Nachteil eines Radantriebs ist die Abhängigkeit von einem geeigneten Untergrund.
  - **Ketten:** Antriebssysteme mit Ketten erlauben einem Roboter, sich auch in schwierigem Gelände zu bewegen. Die größere Lauffläche der Ketten ist sogar Allradantrieben überlegen. Die Umsetzung eines Kettenantriebs ist jedoch häufig mit dem Nachteil höheren Gewichtes und einer komplexeren Hardware verbunden.
  - **Beine:** Auf Beinen gehende Roboter können in der Regel unebenes Gelände sehr viel besser überwinden als Fahrzeuge mit Rädern oder Ketten. Die Technische Realisierung ist jedoch viel schwieriger. So sind mindestens zwei Motoren pro Bein notwendig, um das Heben und das Fortsetzen des Beines zu ermöglichen. Roboter mit Beinen erfordern meisten komplexe Steuerungsalgorithmen zur Koordination der Beinbewegungen. Trotz der zahlreichen Erfolge, die auf dem Gebiet der *Laufmaschinen* gemacht wurden, sind die Probleme und Grenzen dieser Forbewegungsart zu erkennen.

Roboterarme

Im dreidimensionalen Raum besitzt ein frei beweglicher starrer Körper sechs Freiheitsgrade.

- Translation in Richtung jeder Achse (y,x,z)
- Drehung um jede der Achsen x,y,z

Ein Roboterarm

- übernimmt das Führen des Endeffektors (z.B. Greifers, Werkzeugs)
- besteht aus Armelementen (Gliedern), die über Bewegungsachsen (Gelenke) miteinander verbunden sind.

Effektor

Effektor ist ein Oberbegriff für

- **Greifer** zur Manipulation von Objekten
- **Werkzeuge** zur Werkstückbearbeitung
- **Meßmittel** zur Ausführung von Prüfaufträgen
- **Kamerasysteme** bei einem nur beobachtenden Roboter

3.2 Kinematik

Die **Kinematik** beschäftigt sich mit der Geometrie und den zeitabhängigen Aspekten der Bewegung, wobei die Kräfte, die die Bewegung bewirken, außer acht gelassen werden. Aus der Sicht der Kinematik sind also von interesse,

- die Position (Translation, Rotation),
- die Geschwindigkeit,
- die Beschleunigung und
- die Zeit.

Die Kinematik beschreibt

- die Beziehung zwischen Lage des Effektors bezüglich der Roboterbasis und der Einstellung der Gelenkparameter
- die Drehwinkel und Translationswege der Gelenke.

3.3 Antriebe

Antriebe bewirken die Fortbewegung des Roboters und die Bewegung seiner Arme und Endeffektoren (z.B.Greifer). Man unterscheidet drei Antriebsarten mit unterschiedlichen Vor- und Nachteilen:

- **pneumatischer Antrieb:**
  - **Prinzip:** Komprimierte Luft bewegt Kolben. Es gibt keine Getriebe.
  - **Vorteile:** Kostengünstig, einfacher Aufbau, schnelle Reaktionszeit, auch in ungünstigen Umgebungen brauchbar
  - **Nachteile:** Laut, keine Steuerung der Geschwindigkeit bei der Bewegung, nur Punkt-zu-Punkt-Betrieb möglich, schlechte Positioniergenauigkeit.
  - **Anwendungsbereiche:** Kleinere Roboter mit schnellen Arbeitszyklen und wenig Kraft, zum Beispiel Palettierung kleinerer Werkstücke.
- **hydraulischer Antrieb:**
  - **Prinzip:** Öldruckpumpe und steuerbare Ventile

  - **Vorteile:** Sehr große Kräfte, mittlere Geschwindigleit
  - **Nachteile:** laut, zusätzlicher Platz für Hydraulik nötig, Ölverlust führt zur Verunreinigung, Viskosität des Öls erlaubt keine guten Reaktionszeiten und keine hohen Positionier- oder Wiederholgenauigkeiten
  - **Anwendungsbereiche:** Einsatz für große Roboter zum Beispiel zum Schweißen
- **elektrischer Antrieb:**
  - **Prinzip:** (Elektro)motoren
  - **Vorteile:** wenig Platzbedarf, kompakt, ruhig, gute Regelbarkeit der Drehzahl und des Drehmoments, hohe Positionier- und Wiederholgenauigkeit, daher auch Abfahren von Flächen oder gekrümmten Bahnen möglich
  - **Nachteile:** wenig Kraft, keine hohen Geschwindigkeiten
  - **Anwendungsbereiche:** Einsat für kleinere Roboter mit Präszisionsarbeiten

3.4 Sensoren

Die Sensoren eines Roboters haben einerseits die Aufgabe, eine Schnittstelle zwischen dem Roboter und seiner Umwelt zu bilden, andererseit innere Zustände des Roboters selbst zu erfassen. Entsprechend haben Sensoren die folgenden Funktionen:

- Erfassen der Zustände der Umgebung und ggf. der Handhabungsgegenstände
- Erfassen der inneren Zustände des Roboters, wie zum Beispiel seine Lage, Geschwindigkeit, Kräfte, Momente
- Messen physikalischer Größen

Nach [5] kann man drei Hauptkategorien unterscheiden:

- **externe Sensoren:** erfassen Umwelteigenschaften
  - Licht
  - Wärme
  - Schall
  - Kollision
  - physikalische Größen im technischen Prozess
  - Entfernungen
  - Lage von Positioniermarken und Objekten
  - Kontur von Objekten
  - Pixelbilder der Umwelt
- **interne Sensoren:** messen Zustandsgrössen des Roboters selbst
  - Position und Orientierung des Roboters selbst
  - Messung des Batteriestandes
  - Temperatur im Innern des Roboters
  - Motorstrom
  - Stellung der Gelenke
  - Geschwindigkeit mit der sich die Gelenke bewegen
  - Kräfte und Momente, die auf die Gelenke einwirken
- **Oberflächensensoren:**
  - Tastsensoren
  - Lichtsensoren, die die Lichtreflexion von Oberflächen aufnehmen

3.5 Steuerung und Programmierung

Als Robotersteuerung wird nach [5] die Hard- und Software eines einzelnen Roboters bezeichnet. Sie hat folgende Funktionen:

- Entgegennahme und Abarbeitung von Roboterbefehlen und -programmen
- Steuerung und Überwachung von Bewegungs- und Handhabungssequenzen und Fahraufträgen
- Synchronisation und Anpassung des Manipulators an den Handhabungsprozess
- Vermeidung bzw. Auflösen von Konflicksituationen

3.6 Roboter-Programmiersprachen

Roboter werden größtenteils mittels spezieller, für diesen Zweck entwickelten Programmiersprachen, programmiert. In diesem Paier soll besonders auf die einfache Programmiersprache NOT QUIT C (NQC) eingegangen werden, die auf die Programmierung von Kleinrobotern vom Typ LEGO-MINDSTORMS ausgerichtet ist.

3.6.1 Die Programmiersprache NOT QUITE C

Die Programmiersprache NOT QUITE C (NQC) ist eine von Dave Baum speziell für die Programmierung von Kleinrobotern vom Typ LEGO-MINDSTORMS entwickelte Programmiersprache. Sie zeichnet sich durch folgende Eigenschaften aus.

- Der Programmiersprache C ähnelnde Syntax
- Multitasking
- Spezielle Befehle zur Sensorabfrage und Aktuatorsteuerung
- Einfache Integer-Arithmetik
- Imperative Programmierung

Prozesse und Multitasking

Ein *Prozess* oder eine *Task* (engl. Aufgabe) ist ein Programm, das mit anderen Prozessen oder Programmen scheinbar gleichzeitig abläuft. Der gleichzeitige Ablauf mehrerer Prozesse wird als *Multitasking* bezeichnet. Ein *Scheduler* ist ein Steuerprogramm, das den gleichzeitigen Ablauf mehrerer Prozesse koordiniert, indem einem Prozess für eine kurze Zeit (Bruchteil einer Sekunde) die ausschließliche Kontrolle über den Prozessor gegeben wird, um sie ihm anschließend zu entziehen und einem anderen Prozess zu übergeben. Durch diesen schnellen Wechsel der Prozesse, ist es scheinbar möglich, Prozesse parallel ablaufen zu lassen. Man spricht dann von einem quasiparallelen Ablauf.
Beim Multitasking unterscheidet man nach [5] zwei Verfahren:

- **Präemptives Multitasking:** Der Scheduler ist in der Lage, einen Prozess nach einer bestimmten Zeit zu unterbrechen und dann einen anderen Prozess zu laden und auszuführen.
- **Kooperatives Multitasking:** Hier entscheidet der Prozess, wann die Kontrolle an den Scheduler zurückgegeben wird, so dass der nächste Prozess ablaufen kann. Die Prozesse sind hier als deterministische endliche Automaten aufzufassen.

Die Programmiersprache NQC setzt präemptives Multitasking um, wenn jeder Prozess als Task z.B. in der Form

```
task Prozess1() {
     Anweisung 1
     Anweisung 2
     ...
     Anweisung n
}
```

```
task Prozess2() { //
     Anweisung 1   //
     Anweisung 2   //
     ... //
     Anweisung m //
}
```

umgesetzt wird, und der Scheduler durch Starten der Prozesse in der *main*-Task damit beauftragt wird, die Prozesse quasiparallel ablaufen zu lassen.

```
task main() { //
     start Prozess1;//
     start Prozess2;//
     ...//
     start ProzessN;//
}
```

3.7 Kleinroboter vom Typ LEGO-MINDSTORMS

Die in diesem Papier vorgestellten Konzepte werden kommen bei Kleinroboter vom Typ LEGO-MINDSTORMS zur Anwendung. LEGO-MINDSTORMS-Roboter sind einfache Roboter, die von der Firma LEGO als didaktisches Spielzeug für Kinder und Jugendliche kommerziell entwickelt wurden. Sie basieren auf einem Programmierbaren Mikrocontroler, der drei Sensoren und drei Aktuatoren (Elektromotoren) ansteuern kann. Als Sensoren hat die Firma LEGO folgende Sensoren vorgesehen:

- Berührungssensoren
- Lichtsensoren
- Wärmesensoren
- Rotationssensoren

Der Hersteller der LEGO-MINDSTORMS-Roboter vertreibt mit seinen Robotern eine einfache Programmiersprache, mit der eine auf Blockstrukturen beruhende, bedienungsfreundliche Programmierung möglich ist.
Außerdem gibt es zahlreiche,alternativ angebotene, kostenlos erhältliche, Programmiersprachen und Entwicklungsumgebungen, mit denen eine komfortable Programmentwicklung möglich ist.

4 Programmierung von Robotern

Im Bereich der Programmierung von Robotern können nach [5] folgende Grundprinzipien unterschieden werden:

- Programmierung durch Beispiele
- Programmierung durch Training
- Roboterorientierte Programmierung
- Aufgabenorientierte Programmierung

Zunächst sollen diese Grundprinzipien kurz erläutert werden und danach auf einige Prinzipien vertiefend eingegangen werden. Besonders behandelt werden

- die verhaltensorientierte Programmierung nach BROOKS,
- Programmierung mit dynamischen Prozessen

4.1 Programmierung durch Beispiele

Bei dieser Programmiertechnik wird der Roboter auf der gewünschten Bahn geführt. Dabei werden Bahnpunkte abgespeichert, so dass diese Punkte vom Roboter später immer wieder angefahren werden können.
Folgende Spezialfälle der Programmierung durch Beispiele können unterschieden werden.

- **Manuelle Programmierung:** Der Robotor (Effektor) wird von Hand entlang der gewünschten Bahn geführt.
- **Teach-In-Programmierung:** Der Robotor wird von einer bedienenden Person über ein spezielles Eingabegerät (Teach-In-Box) gesteuert.
- **Master-Slave-Programmierung:** Die bedienende Person führt einen kleinen und leicht bewegbaren Master-Roboter wobei die Bewegung auf den schweren, großen Slave-Roboter übertragen wird.
- **Teleoperation:** Hier wird wie bei der Master-Slave-Programmierung gearbeitet, wobei die Daten der abgefahrenen Bahn jedoch nicht abgespeichert werden.

Vorteile gegenüber textueller Programmierung sind:

- keine Programmierkenntnisse erforderlich
- keine zusätzlichen Rechner für die Programmierung erforderlich
- direkter Bezug zur realen Aufgabe und Umgebung unter Berücksichtigung aller konstruktiver Ungenauigkeiten und sonstiger Störgrößen
- schnelle Anpassung an neue Gegebenheiten durch unkomplizierte Umprogrammierung möglich

Der Hauptnachteil ist, dass ein Einbezug von Sensoren und die Korrektur von Bahnen aufgrund von Sensordaten nicht möglich ist. Der Roboter agiert gewissermaßen „blind“ für seine Umwelt.

4.2 Programmierung durch Training

Bei der Programmierung durch Training wird dem Roboter die auszuführende Tätigkeit vorgeführt, die dieser über Sensoren aufnimmt. Der Roboter führt die Aktion dann solange immer wieder aus (Training), bis sie gewissen Gütekriterien entspricht. Dabei muß die Abweichung von der gewünschten Zielvorgabe über externe Sensoren festgestellt werden. Mit Hilfe der gemessenen Abweichungen wird jeweils versucht, das Programm weiter zu verbessern.
Programmierung durch Training gilt als Forschungsthema und ist für reale Aufgaben bisher noch nicht anwendbar.

Hauptprobleme sind:

- Extrahierung bzw. Generierung einer Handlungssequenz aufgrund der beobachteten Aktionen
- Einbeziehung unterschiedlicher Sensoren in die Lösung einer Aufgabe und die Integration der Sensordaten für die Auswertung
- Bildverarbeitung und -interpretation (Erkennen von Objekten, Positionen und Orientierungen im Raum - Verfolgen bewegter Objekte)

4.3 Roboterorientierte Programmierung

Bei dieser Programmiertechnik erfolgt die Steuerung des Roboters durch ein Roboterprogramm mit expliziten Bewegungsbefehlen. Das Programm ist in einer speziellen Roboterprogrammiersprache geschrieben.
Roboterprogrammiersprachen basieren zum Teil auf Hochsprachen wie Pascal, PL/1, C oder Java und setzen verschiedene Programmierparadigma um. Die für die Programmierung von LEGO-MINSTORMS-Robotern häufig eingesetzte Programmiersprache NQC (Not-Quite-C) ist eine an die Programmiesprache C angelehnte, imperative Programmiersprache, die gewissen Einschräkungen (z.B. nur Integer-Arithmetik) unterliegt jedoch *Multitasking* erlaubt.

4.4 Aufgabenorientierte Programmierung

Während bei roboterorientierten Programmierung durch das Programm spezifiziert wird, *wie* der Roboter eine Aufgabe zu bearbeiten hat, so wird bei der *aufgabenorientierten* Programmierung lediglich spezifiziert, *was* der Roboter zu tun hat. Ein **Aufgabenplaner** erzeugt dann ein Roboterprogramm in einer roboterorientierten Programmiersprache.
Bei der aufgabenorientierten Programmierung erfolgt die Programmierung auf einer deutlich höheren Abstraktionsebene. Deshalb wird diese Art der Programmierung häufig mit *intelligenten* Robotern in Verbindung gebracht. Zentrale Konzepte dieses Ansatzes sind:

- Weltmodellierung
- Aktionsplanung

Das Roboterprogramm implementiert dann folgende Funktionseinheiten:

1. Eingabedaten von den Sensoren aufnehmen
2. Sensorendaten interpretieren
3. Weltmodellierung
4. Aktionsplanung
5. Ausführung von Aktionen
6. Ausgabe an die Aktuatoren

5 Verhaltensbasierte Programmierung

Bei der verhaltensbasierten Programmierung geht man davon aus, die verschiedensten Verhaltensweisen eines Roboters, die miteinander in Wechselwirkung stehen, sich hemmen und verstärken können, zu realisieren und zu einem Ganzen zusammenzusetzen. Ein weitverbreitete Technik besonders bei der Programmierung von Kleinrobotern geht dabei auf den sogenannten **Subsumtionsansatz** von BROOKS [1] zurück.

5.0.1 Der Subsumtionsansatz von BROOKS

Bei der Programmierung eines Roboters nach dem Subsumtionsansatz wird das Gesamtverhalten eines Roboters durch hierarchisch aufeinander aufbauende Einzelverhalten bestimmt. Die Einzelverhalten laufen (quasi)parallel ab und werden aufgrund von Sensorinformationen ausgelöst. Mit Hilfe eines Auswahlschemas wird das für die jeweilige Situation dominate Verhalten bestimmt. Stichwortartig läßt sich das Konzept wie folgt darstellen:

- Es gibt höherwertige und niederwertige Verhalten.

- Alle Verhalten laufen parallel ab.
- Kein Verhalten ruft eine anderes Verhalten (im Sinne eines Unterprogramms) auf.
- Höherwertige Verhalten können niederwertige Verhalten zeitweise unterdrücken (inhibieren, Inhibited Wire) oder Überschreiben (Suppressed Wire). Unterdrücken eines Verhaltens geschieht durch Blockieren eines Ausgabekanals, Überschreiben, durch (Er)setzen eines Eingangssignals (auf) durch einen bestimmten Wert.
- Die Gesamtarchitektur des Robotersystems wird vom einfachen Verhalten auf niedrigster Ebene zum höherwertigen Verhalten auf höchster Ebene *bottom-up* entwickelt.
- Es gibt keine einheitliche Datenstruktur oder ein explizite Repräsentation der Umwelt (Weltmodell)
- Erweiterbarkeit der Architektur soll jederzeit möglich sein.

5.0.2 Einfache Verhaltensnetze

In diesem Abschnitt sollen besonders einfache Verhaltensweisen von mobilen Kleinstrobotern nach dem Konzept der Subsumtion vorgestellt werden.

Beispiel 5.1 (Hindernisumgehung). Ein mobiler Kleinroboter (z.B. Typ LEGO-MINDSTORMS) sei mit zwei einfachen Berührungssensoren (Bumper) ausgestattet und verfüge über zwei Motoren, die getrennt (rechts/links) angesteuert werden können. Durch seine Konstruktion als Dreirad (Ackermannlenkung) ist er gut manövrierfähig.
Der Roboter soll im Raum umherfahren und beim Berühren eines Hindernisses, zurückweichen und dieses anschließend umfahren. Das in Abbildung 1 dargestellte, einfache Subsumtionsprogramm implementiert diese Verhaltensweisen.

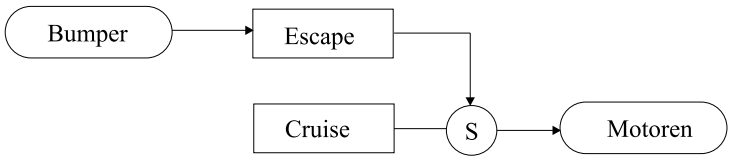

Abbildung 1: Blockdiagramm eines einfachen Subsumtionsprogramms

Die graphische Darstellung der Subsumtionsprogramme orientiert sich an [5]: Abgerundete Kästchen bezeichnen physische Sensoren oder Aktuatoren einschließlich der Softwaretreiber, die sie direkt steuern. Eckige Kästchen bezeichnen das Programmmodul, das ein bedingtes Verhalten realisiert. Als Kreis mit dem Symbol *S* ist ein sogenannter Unterbrecherknoten dargestellt, des Entscheidungen fällt, welches Verhalten entsprechend seine Priorität zum Wirkung kommt.
Das in Abbildung 1 dargestellte Programm arbeitet folgendermaßen:

- Das Modul *Cruise* hat allein den Zweck, den Roboter permanent geradeaus fahren zu lassen. Es implementiert eine ungerichtete Suche, eine Erkundung des Raums. Dieses Modul erhält keine Eingaben von Sensoren, sondern steuert lediglich die Motoren für einen Geradeauslauf.
- Das Modul *Escape* wird aktiviert, wenn einer der Berührungssensoren eine Kollision verzeichnet. Entsprechend der Seite, auf der die Kollision verzeichnet wurde, dreht sich der Roboter weg, nachdem er zunächst ein Stück zurückgewichen ist.

Um die bottom-up-Struktur der Subsumtionsarchitektur noch weiter zu verdeutlichen, soll noch ein weiteres Beispiel angegeben werden, aus dem das Hierarchische noch besser zu erkennen ist. Das Beispiel ist, ebenso wie das obige Beispiel, leicht mit LEGO-MINSTORMS-Robotern zu realisieren.

Beispiel 5.2 (Linienverfolgung). Ein mobiler Kleinroboter (z.B. Typ LEGO-MINDSTORMS) sei mit einem einfachen Berührungssensor (Bumper) und einem gegen den Boden gerichteten Lichtsensor ausgestattet und verfüge über zwei Motoren, die getrennt (rechts/links) angesteuert werden können. Durch seine Konstruktion als Dreirad (Ackermannlenkung) ist er gut manövrierfähig.
Der Roboter soll einer auf dem Boden angebrachten Linie folgen, wobei er sich vorwärts bewegt. Sollte er die Linie verloren haben, so soll er anhalten und die Linie suchen, um wieder darauf weiter zu laufen. Beim Berühren eines Hindernisses, soll er sofort anhalten und erst nach einer Weile erst wieder losfahren. Das in Abbildung 2 dargestellte, einfache Subsumtionsprogramm implementiert diese Verhaltensweisen.

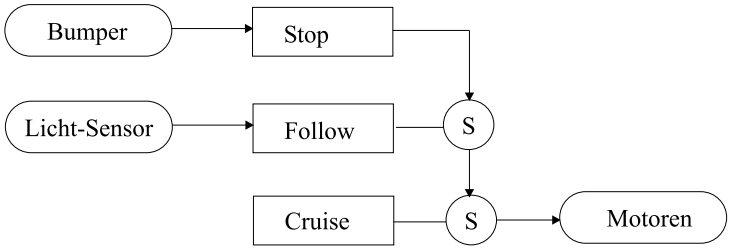

Abbildung 2: Blockdiagramm eines einfachen Subsumtionsprogramms

- Das Modul *Cruise* hat in diesem Fall den Zweck, den Roboter permanent fahren zu lassen. Dieses Modul erhält keine Eingaben von Sensoren, sondern steuert lediglich die Motoren für einen Geradeauslauf. Es ist jedoch von höherwertigen Verhalten inhibierbar.
- Das Modul *Follow* implementiert eine Heuristik, die zu verfolgende Linie wieder aufzusuchen, falls diese verlassen worden ist.
- Das Modul *Stop* wird aktiviert, wenn der Berührungssensor eine Kollision verzeichnet. Der Roboter soll dann für eine Weile unbewegt verharren, in der Hoffnung, der Auslöser der Kollison (z.B. ein Mensch) hätte den Weg des Roboters verlassen und erst später wieder losfahren.

Der Darstellung ist zu entnehmen, dass das Verhalten *Stop* höchste Priorität hat. Es wird davon ausgegangen, dass eine Aktivierung dieses Verhaltens dazu führt, dass alle anderen, niederwertigen Verhalten inhibiert werden.
Das dargestellte Programm ist natürlich sehr einfach, der damit ausgestatte Roboter nur unter vereinfachenden Bedingungen einsetzbar. Es ist jedoch möglich, weitere Verhalten auf dieser Architektur aufzusetzen und mit einer definierten Priorität wirken zu lassen.

5.0.3 Implementierung von Subsumtionsprogrammen

Viele Programmiersprachen für Kleinroboter - so auch die Sprache NQC für LEGO-MINSTORMS-Roboter - erlauben **Multitasking**, also die (quasi)parallele Abarbeitung verschiedener Aufgaben, sog. *tasks* auch auf einem Ein-Prozessor-System.
Für eine Implementation von Subsumtionsprogrammen identifiziert man ein Verhalten mit einer solchen *task*, d.h. jedem Verhaltensmodul wird eine Task zugeordnet, die Eingaben von den Sensoren aufnimmt, verarbeitet und Aktuatordaten sendet.
Betrachten wir nun eine Implementation von Beispiel 5.1 in der Programmiersprache NQC.

Beispiel 5.3 (NQC Implementation der Hindernisumgehung). Jeder Ebene des Subsumtionsprogramms wird eine NQC-Task zugeordnet. Das Verhalten *Cruise*, das auf niedrigster Ebene angeordnet ist, ist leicht zu Implementieren.

```
task Cruise() {
     while (true) {
```

```
            OnFwd(LinkerMotor+RechterMotor);
            }
}
```

Etwas aufwendiger ist die Implementierung des Verhaltens *Escape*.

```
task Escape() {
     while (true) {
           if (LinkerSensor==Beruehrung)
              { stop Cruise;
                OnRev(LinkerMotor+RechterMotor);Wait(60);
                OnFwd(LinkerMotor);Wait(Random(60));
                Float(LinkerMotor+RechterMotor);
                start Cruise;
              }
              else if (RechterSensor==Beruehrung)
              {stop Cruise;
               OnRev(LinkerMotor+RechterMotor);Wait(60);
               OnFwd(RechterMotor);Wait(Random(60));
               Float(LinkerMotor+RechterMotor);
               start Cruise;
              }
     }
}
```

Die Task *Escape* realisiert eine einfache Strategie: Verzeichnet einer der beiden Berührungssensoren eine Berührung, so fährt der Roboter für eine bestimmte Zeit (60 ns) zurück, um anschließend zu der Seite wegzudrehen, die der Berührungsseite gegenüberliegt. Damit sich der Roboter nicht in Ecken verfängt (deadlock), ist das Ausmaß des Wegdrehens randomisiert (0 bis 60 ns) umgesetzt. Die angegebenen Zeitmaße sind, ebenso wie das Abschalten der Stromversorgung der Motoren mit *Float()*, wodurch die Motoren nicht gebremst werden sondern auslaufen, konstruktionsbedingt.
Beide Tasks werden in einer Task *main* gestartet und laufen dann paralell ab. Die Task *Escape* blockiert die Task *Cruise*, wenn sie die Aktuatoren ansteuert und hebt die Blockierung wieder auf, sobald die Aktuatoren nicht mehr von ihr angesteuert werden. Die Hierarchie der beiden Verhalten ist hier also deutlich zu erkennen.

5.1 Programmierung dynamischer Prozesse

Das im Folgenden vorgestellte Konzept der verhaltensbasierten Programmierung beruht auf der Implementierung dynamischer Prozesse. Die Ausführungen halten sich im Wesentlichen an [3] sowie [2].
Als Programmiersprache wir PDL (Process Description Language) eingesetzt, eine Programmiersprache, die an der Freien Universität Brüssel von LUC STEELS entwickelt wurde. Die in die Programmiersprache C eingebettete Sprache wurde in zahlreichen Experimenten mit verschiedenen (mobilen) Robotern eingesetzt.
Die zentralen Elemente des auf dynamischen Systemen basierenden Ansatzes sind:

- **Quantitäten**, die zum Abspeichern, von Sensorwerten, Aktuatorwerten und und zur Representation interner Informationen verwendet werden,
- **Prozesse**, die als Bausteine des Informationsverarbeitungsprozesses von Verhaltensweisen aufgefasst werden können.

Das Gesamtsystem durchläuft Zyklen, wobei folgende Schritte abgearbeitet werden:

1. Auslesen von Sensorwerten und Einfrieren dieser in Form von Sensorquantitäten

2. (quasi)paralleles Abarbeiten von Prozessen, die auf Quantitäten arbeiten und diese incrementell positiv oder negativ beeinflussen

3. Verrechung der Beiträge der verschiedenen Prozesse zu den Quantitäten, die Verhalten repräsentieren sollen

4. Auswertung und Umsetzung von Aktuatorquantitäten in Roboterverhalten

Die Architektur des Roboterssystems ist so beschaffen, dass ein kontinuierlicher Eingabedatenstrom existiert, der vom Steuerprogramm über Quantitäten abgegriffen werden kann sowie ein kontinuierlicher Ausgabedatenstrom, der die Aktuatoren ebenfalls über entsprechende Quantitäten beeinflusst. Zudem gibt es Quantitäten, die innere Zustände des Roboters repräsentieren.

Die Werte von Quantitäten können über die Methode *AddValue(Quantität,Additionsterm)* verändert werden, wobei die Veränderung erst am Ende eines Zyklus wirksam wird. Der Wert einer Quantität kann über die Methode *Value(Quantität)* abgefragt werden.
Abbildung 3 verdeutlicht den Ablauf eines PDL-Zyclus.

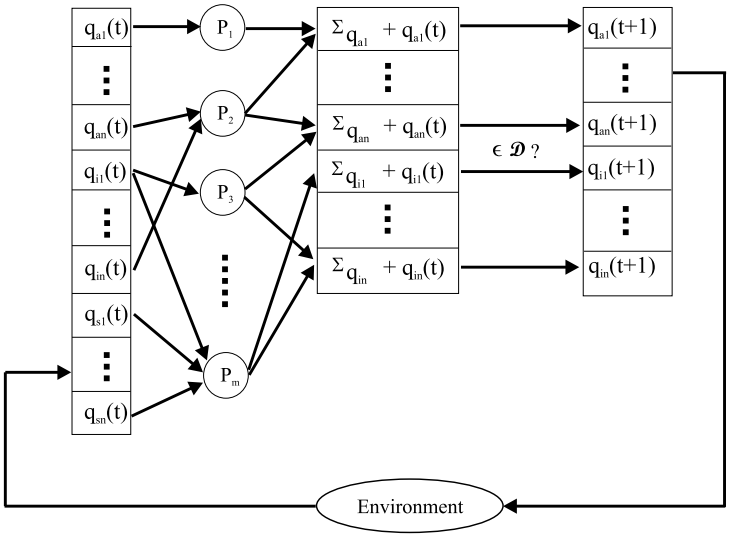

Abbildung 3: Darstellung eines PDL-Zyklus. Quantitäten zum Zeitpunkt t und $t+1$. Abb. nach [2]

Beispiel 5.4 (PDL-Implementierung einer Hindernisvermeidung). In diesem Beispiel wird die PDL-Implementation einer Hindernisvermeidung unter Verwendung zweier Infrarot-Sensoren nach [3] beschrieben.

- **Sensorquatitäten:**
  IRLeft und IR Right: Infrarot-Sensor-Quantitäten (Werte zwischen 0 und 255. Konstruktionsbedingt verhalten sich die Quantitätenwerte fallend, wenn die Infrarot-Reflektion ansteigt.

- **Actuatorquantitäten:**
  MotorLeft und MotorRight: Motorwerte für Rückwärts- und Forwärtsfahrt. (Werte zwischen -255 und $+255$).

- **Konstanten:**
  IR-reaction-threshold: Grenzwert für die Aktivierung der Hindernisvermeidung. Umgebungsabhängig.

- **Prozesse:**
  Die folgende Hindernis-Vermeidung verwendet die Sensor- und Actuatorquantitäten:

```
process Obstacle-avoidance;
 if ((Value(IRRight)<IR-reaction-threshold)
      &&(Value(IRRight)<Value(IRLeft)))
    AddValue(MotorLeft,(Value(IRRight)-Value(IRLeft)));
 if ((Value(IRLeft)<IR-reaction-threshold)
      &&(Value(IRLeft)<Value(IRRight)))
    AddValue(MotorRight,(Value(IRLeft)-Value(IRRight)));
endprocess
```

Der Prozess zur Hindernisvermeidung wird mit einem Prozess kombiniert, der das Vorwärtsfahren gewährleitet.

```
process Move-Forward;
        AddValue(MotorLeft,+1.0);
        AddValue(MotorRight,+1.0);
endprocess
```

5.1.1 Umsetzung in der Programmiersprache NQC

Bei dem in [3] umgesetzten Verfahren werden die Berechnungen in Fleißkommaarithmetik vorgenommen, die von der dort eingesetzten Programmiersprache PDL (Prozess Description Language) [4] unterstützt wird. In der Programmiersprache NQC - die im Rahmen dieses Papiers verwendet wird - ist nur eine Integerarithmetik möglich. Zudem wird bei der Verwendung von PDL davon ausgegangen, dass das System in Zyklen mit einer konstanten Periode von 40 Zyklen pro Sekunde läuft. Eine Synchronisation in dieser Form ist mit NQC nicht möglich. Statt dessen wird mit einer Endlosschleife gearbeitet, die die Prozesse immer wieder aufruft. Es wird also versucht, die Ideen von STEELS mit den begrenzten Möglichkeiten von NQC weitgehend umzusetzen.

Beispiel 5.5 (Linienverfolgung). Ein Roboter vom Typ LEGO-MINDSTORMS soll eine auf dem Boden aufgebrachte Linie verfolgen. Als Lösungsstrategie wird folgende Idee umgesetzt:
Der Roboter soll auf der Linie laufen, indem er bogenförmig auf der Linie fährt, bis er die Linie verliert. Hat er die Linie verloren nachdem er zum Beispiel einen Rechtsbogen fuhr, so soll er die Linie wieder finden, indem er in einem engen Bogen nach links zurück zur Linie fährt, bis er diese wieder auffindet. Umgekehrt verfährt er analog.
Zu Umsetzung dieser Strategie nach dem Prinzip der dynamischen Systeme werden folgende Quantitäten eingerichtet:

```
// Sensorquantitaeten
   int aufLinie;              // 0=Linie verloren, 1=Linie gefunden

// interne Quantitaeten
   int LetzteFahrtRichtung;  // links=linksgerichteter, rechts=rechtsgerichteter Bo
   int LetzterZustand;       // Panik=Linie verloren, Sicherheit=f"ahrt auf Linie

// Aktuatorquantitaeten
   int RichtungRechterMotor;    // vorwaerts oder rueckwaerts
   int RichtungLinkerMotor;
```

Die drei Prozesse *Sensor()*, *FolgeLinie()* und *Antrieb()* werden in einer Endlosschleife in der *main*-Methode immer wieder aufgerufen, womit eine (quasi)parallele Abarbeitung simuliert werden soll.

```
task main () {
     Initialisierung();
     while (true) {
           // Ablesen und Einfrieren der Quantitaeten
              Sensor();
           // Auswerten  der Quantitaeten
              FolgeLinie();
           // Interpretieren der Quantitaeten als Aktuatorwerte
              Antrieb();
     }
}
```

Durch die Implementation als Inline-Funktion ist beim Prozess *Sensor()* das Einfrieren der Sensordaten in Form der Quantität *aufLinie* gewährleistet.

```
void Sensor() {
     if (LichtSensor > LichtGrenzwert)  // LichtGrenzwert=47 konstrutionsbedingt
        aufLinie=0;
       else
        aufLinie=1;
}
```

Der Prozess *FolgeLinie* fragt die Quantität *aufLinie* ab und setzt die Quantitäten *RichtungRechterMotor* und *RichtungLetzterMotor* je nachdem, ob eine Kreisbogenbewegung auf der Linie oder - im Falle des Linienverlustes - ein Wiederfinden der Linie nötig ist. Beim Linienverlust wird die Quantität *Letzter-Zustand* auf *Panik* gesetzt. Beim Wiederauffinden der Linie wird die Kreisbogenbewegung des Roboter gegenläufig gesetzt und die innere Quantität *LetzterZustand* auf *Sicherheit* gesetzt.

```
void FolgeLinie() {
  if (!aufLinie)
     { LetzterZustand=Panik;
       if (LetzteFahrtRichtung==rechts)
        { RichtungRechterMotor=vorwaerts;    // drehe nach links rein
          RichtungLinkerMotor=rueckwaerts;
         }
        else // LetzteFahrtRichtung=links
        { RichtungRechterMotor=rueckwaerts;    // drehe nach rechts rein
          RichtungLinkerMotor=vorwaerts;
        }
      }
   else  { RichtungRechterMotor=vorwaerts;    // normales Vorwaertsfahren
           RichtungLinkerMotor=vorwaerts;
           if (LetzterZustand==Panik)// falls von auerhalb der Linie kommend
             { if (LetzteFahrtRichtung==rechts)       //wechsel die Fahrtrichtung
                        LetzteFahrtRichtung=links;
                   else  LetzteFahrtRichtung=rechts;
               LetzterZustand=Sicherheit;             //und versetze in Sicherheit
             }
          }
 }
```

Der Prozess *Antrieb()* setzt die Werte der Aktuatorenquantitäten in Motorbewegungen um.

```
void Antrieb() {
```

```
  if (RichtungRechterMotor==RichtungLinkerMotor) //falls Vorwaertsfahrt gew"unscht
     { if (LetzteFahrtRichtung==rechts)
          { SetOutput(RechterMotor,OUT_FLOAT);
            SetOutput(LinkerMotor,OUT_ON);
          }
       else
          { SetOutput(LinkerMotor,OUT_FLOAT);
            SetOutput(RechterMotor,OUT_ON);
          }
      }
  else {
        if(RichtungRechterMotor==vorwaerts && RichtungLinkerMotor==rueckwaerts)
            { SetDirection(RechterMotor,OUT_FWD);
              SetDirection(LinkerMotor,OUT_REV);
              SetOutput(RechterMotor+LinkerMotor,OUT_ON);
             }
           else
             {SetDirection(LinkerMotor,OUT_FWD);
              SetDirection(RechterMotor,OUT_REV);
              SetOutput(RechterMotor+LinkerMotor,OUT_ON);
             }
         }
}
```

Da die Aktuatorenquantitäten nur diskrete Werte (vorwaerts, rueckwaerts) annehmen, ist eine incrementelle Veränderung und Summation am Ende eins Zyklus in diesem einfachen Beispiel nicht notwendig. Die in der *main*-Methode aufgeführte Methode *Initialisierung* setzt den Roboter in einen definierten Anfangszustnad, wobei davon ausgegangen wird, dass er auf der Linie startet und mit einem linksgerichteten Kriesbogen beginnt.
Versteht man die Prozesse *Sensor()* und *Antrieb()* als Treiberprogramme für Senoren und Aktuatoren und stellt diese nicht explizit graphisch dar, so ergibt sich der in Abbildung 4 dargestellte Zyklus. Die Quan-

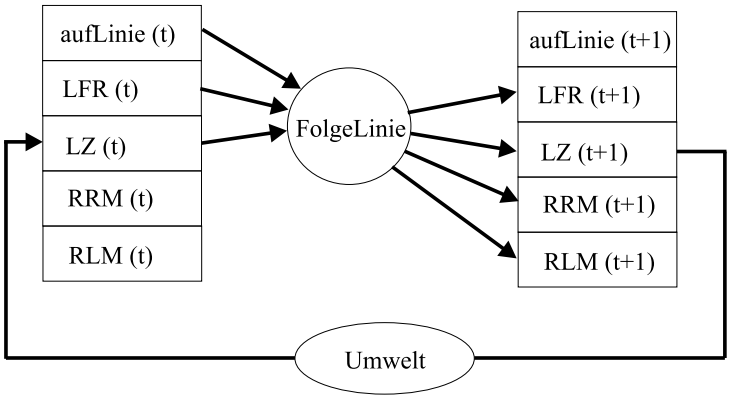

Abbildung 4: Darstellung eines Zyklus. Quantitäten zum Zeitpunkt t und $t+1$.

titäten sind dabei in der Reihenfolge Sensor-, innere und Aktuatorenquantitäten aufgeführt, wobei mit *LFR* die Quantität *LetzteFahrRichtung*, mit *LZ* die Quantität *LetzterZustand* und mit *RRM* und *RLM* die Quantitäten *RichtungRechterMotor* und *RichtungLinkerMotor* bezeichnet sind. Der Prozess *FolgeLinie* ist dann der einzige, die Quantitäten verändernde, Prozess.

Literatur

[1] Brooks, R.A.: A Robust Layered Control System for Mobile Robot. *IEEE Journal on Robotics and Automation*, RA-2:14-23, April 1986.

[2] Knoll, A. C., Burgard, W., Christaller, T.: Robotik. In: Görz/Rollinger/Schneeberger (Hrsg.): Handbuch der Künstlichen Intelligenz. 3. vollst. überarb. Aufl.. München; Wien; Oldenbourg, 2000.

[3] Christaller, T., Dautenhahn, K., Pauer, M.,Schlottmann, E., Spenneberg, D.: A modular design approach towards behavior oriented robotics. 1997. http://www...

[4] Steels, L.: The PDL reference manual. VUB AI Lab memo 92-5, AI laboratory, Brussels. 1992

[5] Steinmüller, J.: Robotik. Vorlesung an der TU Chemnitz WS 2000/2001.