Robotik II
Programmierung von Robotern

Skriptum zur Vorlesung

Vorversion
Rückmeldungen und Fehler bitte an `robotik2@ira.uka.de`

Lehrstuhl für Industrielle Anwendungen der Mikrosystemtechnik
Institut für Technische Informatik
Fakultät für Informatik
Universität Karlsruhe (TH)

Herausgeber:
Prof. Dr.-Ing. R. Dillmann
Dipl.-Ing. S. Knoop

April 2007

Autoren:

Michael Demel, Jonas Firl, Steffen Knoop, Michael Pardowitz, Martin Pruzinec,
Matthias Rambow, Steffen Rühl, Sven R. Schmidt-Rohr, Markus Westphal, Tom Zentek

Inhaltsverzeichnis

Vorwort

In den letzten Jahren hat sich die Wissenschaft der Robotik mit Riesenschritten entwickelt. War sie vor Jahrzehnten noch eine rein für industrielle, repetetive Aufgaben verwendete Technologie, so wandelt sich die Robotik inzwischen immer mehr: Die Anwendungen werden immer breiter, und angrenzende Wissenschaften werden integriert und eingegliedert. Waren vor Jahrzehnten nur vollständig bekannte Szenarien mit Robotern handhabbar, so wagen sich Forscher heutzutage mit den Roboter in völlig unbekannte Umgebungen. Prominentes Beispiel sind die erfolgreich auf den Mars geschickten teilautonomen Fahrzeuge *Spirit* und *Opportunity*, die inzwischen seit weit über zwei Jahren den Roten Planeten erkunden.

Allerdings müssen Roboter nicht auf den Mars geflogen werden, um den Herausforderungen unstrukturierter und unbekannter Umgebungen zu begegnen: Menschenzentrierte Umgebungen wie private Wohnungen, öffentliche Gebäude oder Kaufhäuser bieten ein hohes Maß an Dynamik und stochastischem Verhalten. Der Umgang in direkter Interaktion mit dem Menschen selbst ist schließlich eine der höchsten Herausforderungen an die moderne Robotik: Interaktionsmechanismen müssen nicht nur zur Verfügung stehen, sondern vor allem auch interpretiert und in die Planung des Roboters mit einbezogen werden.

Auf Basis dieser Entwicklung und der Herausforderungen der modernen Robotik will die Vorlesung *Robotik 2 – Programmieren von Robotern* einen Überblick über existierende Verfahren zur Programmierung solcher autonomen Maschinen geben. Der Bogen wird gespannt von den klassischen Anwendungen in der Industrie bis hin zu modernsten Planungs- und Entscheidungsmechanismen, die bisher nur in der reinen Forschung untersucht werden und für deren Anwendbarkeit im großen Maßstab noch viel Entwicklungsarbeit notwendig ist.

Aufbauend auf der Vorlesung *Robotik 1 – Grundlagen der Robotik* werden die Schwerpunkte auf der Modellierung der Umwelt und der Aufgaben sowie auf Planungs- und Entscheidungsverfahren gesetzt. Die Vorlesung stellt sowohl klassische Programmiermethoden wie *Teach-In* als auch moderne Verfahren wie *Programmieren durch Vormachen* vor. Ausgiebig werden Methoden zur Aktionsplanung betrachtet, deren Aufgabe in der Bestimmung einer Handlungskette besteht, um ein vorgegebenes Ziel zu erreichen. Soll der Roboter autonom situationsabhängig eine Verhaltensentscheidung durchführen können, so kommen probabilistische Ansätze zum Einsatz. Hier werden beispielhaft die *Partially Observable Markov Decision Processes* vorgestellt.

Das vorliegende Skript zur Vorlesung *Robotik 2* entstand in den Jahren 2005 und 2006 unter Mithilfe von Studenten teilweise im Rahmen des Roboterseminars. Michael Demel, Jonas Firl, Martin Pruzinec, Matthias Rambow, Steffen Rühl, Markus Westphal und Tom Zentek gebührt großer Dank für ihren Einsatz und das Engagement.

Das Skript ist in seiner vorliegenden Form sicher noch nicht perfekt und fehlerfrei. Nichtsdestotrotz wollen wir den Hörern der Vorlesung die Möglichkeit bieten, die Inhalte auch in schriftlicher Form zu erhalten. Die Qualität dieses Skriptums kann allerdings nur weiterhin gesteigert werden, wenn uns auftauchende Fehler und Unzulänglichkeiten mitgeteilt werden. Der Aufruf geht also an alle aufmerksamen Studenten, auch durch eigene Mitarbeit die Qualität der Lehre weiterhin zu steigern!

In diesem Sinne wünschen wir allen interessierten Studenten viel Erfolg.

R. Dillmann, S. Knoop

Kapitel 1

Grundlagen

In diesem Kapitel werden die wichtigen Grundlagen der Roboterprogrammierung für diese Vorlesung behandelt. Neben geschichtlichen Aspekten wird eine kurze Übersicht über unterschiedliche Arten der Klassifizierung von Roboterprogrammierverfahren gegeben. Dazu wird eine Einteilung von Programmierverfahren nach

- dem Programmierort,
- der Programmierart und
- dem Abstraktionsgrad

vorgenommen. Auf eine genauere Betrachtung wird allerdings hierbei verzichtet, dies geschieht in den folgenden Kapiteln.

1.1 Geschichtliches

Das Wort "Roboter" geht geschichtlich zurück auf das slawische Wort für Fronarbeit: *robota*. In modernen slawischen Sprachen wird dieses aber eher im Sinne von *Maloche* verwendet. Die, im Laufe der Zeit stark veränderte, Bedeutung des Ausdrucks „Roboter“ prägte der international wohl bekannteste tschechische Schriftsteller Karel Čapek durch seine Satiren über den technischen Fortschritt. So tauchte 1920 in seinem Schauspiel R.U.R. (Rossum's Universal Robots) erstmal das Wort Roboter auf.

Weitere Vordenker waren Stanislaw Lem und der US-Amerikaner Isaac Asimov (Abb. 1.1). Letzterer wurde vor allem durch die Asimovschen Gesetze bekannt , erstmals zu finden in seinem 1942 erschienen Roman *Robots and Empire* (dt. *Das Galaktische Imperium*). Diese drei grundlegenden Regeln lauten:

- Ein Roboter darf keinem Menschen verletzen oder durch Untätigkeit zu Schaden kommen lassen.

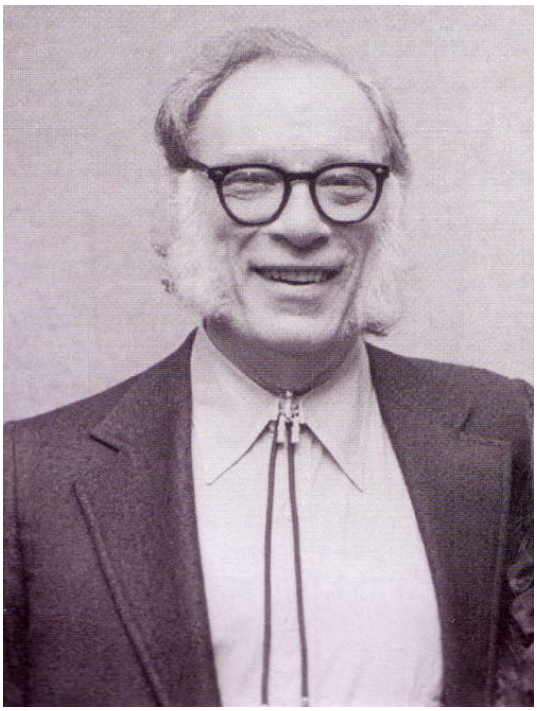


Abbildung 1.1: Isaac Asimov

- Ein Roboter muss den Befehlen der Menschen gehorchen - es sei denn, solche Befehle stehen im Widerspruch zum Ersten Gesetz.

- Ein Roboter muss seine eigene Existenz schützen, solange dieses sein Handeln nicht dem Ersten oder Zweiten Gesetz widerspricht.

Hierbei ist zu beachten, dass die drei Gesetze hierarchisch aufgebaut sind, d.h. das erste Gesetz steht "über" den beiden anderen. Das wichtigste und oberste Prinzip eines Roboters sollte also stets sein durch seine Handlungen keine Menschen zu Schaden kommen zu lassen.

Später lässt Asimov in seinem Roman *Das Galaktische Imperium* das sog. „Nullte Robotergesetz" entwickeln:

- Ein Roboter darf der Menschheit keinen Schaden zufügen oder durch seine Untätigkeit gestatten, dass die Menschheit zu Schaden kommt.

Die ersten drei Gesetze wurden dahingehend verändert, dass dieses Nullte Gesetz in der Hierarchie über ihnen steht. [Wik]

1.2 Klassifizierungen von Roboterprogrammierverfahren

Im Folgenden werden Roboterprogrammierverfahren nach verschiedenen Kriterien klassifiziert:

1.2.1 Programmierort

Es gibt im Wesentlichen zwei unterschiedliche Orte der Roboterprogrammierung:

Direkte Programmierung
Die Direkte Programmierung (oder auch *on-line Programmierung*) zeichnet sich dadurch aus, dass die eigentliche Programmierung *direkt* am Roboter, also unmittelbar an der Robotersteuerung, erfolgt. Dadurch ist es auch verständlich, dass diese Art der Programmierung als *prozessnah* bezeichnet wird, da der Bediener direkten Kontakt zum zu programmierenden Roboter hat.

Indirekte Programmierung
Im Gegensatz zur Direkten Programmierung erfolgt die Programmierung hier ohne den Roboter, sondern mit Hilfe von textuellen, graphischen oder interaktiven Methoden, auf welche später noch genauer eingegangen wird. Da also hier nicht direkt am Roboter gearbeitet wird, bezeichnet man diese Methode auch als *off-line* oder *prozessferne* Programmierung.

Wir werden sehen dass beide Verfahren Vor- und Nachteile haben, weswegen heutzutage, vor allem bei komplexeren Programmierungen, beide Verfahren zum Einsatz kommen. Deswegen ist es oftmals nicht möglich eindeutig von einer *on-line* oder einer *off-line* Programmierung zu reden, da unzählige *Hybride Verfahren* zwischen diesen beiden bestehen.

1.2.2 Art der Programmierung

Die Roboterprogrammierung lässt sich in vier Arten einteilen:

Direkte Programmierung
Diese Art fällt als Einzige der hier vorgestellten Arten in den Bereich *on-line* Programmierung. Man kann diese als wirkliches „Einstellen“ des Roboters betrachten. Hierbei gibt es etliche unterschiedliche Verfahren, wie z.B. die *Teach-In-*, oder die *Play-Back* Programmierung. Diese und weitere Beispiele der direkten Programmierung werden in Kapitel 2 genauer betrachtet.

Textuelle Verfahren
Bei den *textuellen Verfahren* erfolgt die Programmierung durch höhere Programmiersprachen, wie z.B. PasRo oder Val, wodurch ein wirkliches *Robotersteuerprogramm* entsteht. Da dies jedoch nicht direkt am Roboter geschieht, zählt dieses Verfahren zu der *off-line* Programmierung. Zu den Vorteilen dieser Methode zählt, dass die Programmierung un-

abhängig vom Roboter erfolgen kann und dadurch theoretisch für mehrere Roboter zu verwenden ist. Auch die Möglichkeit der Erstellung sehr komplexer Programme ist ein weiterer positiver Aspekt. Allerdings muss der Bediener über gute Programmierkenntnisse verfügen um solche Programme erstellen zu können.

Graphische Verfahren
Hier wird die Umgebung des Roboters neben der textuellen Beschreibung auch graphisch dargestellt. Es wird versucht durch eine möglichst exakte Darstellung die Roboterprogramme möglichst gut zu simulieren. Als Vorteile zählen, neben der schon bei den textuellen Verfahren erwähnte Unabhängigkeit vom Roboter, die Tatsache dass weniger Programmierkenntnisse benötigt werden, die Programmierung für den Benutzer also einfacher wird. Als größter Nachteil dieses Verfahrens kann genannt werden, dass man sehr komplexe Modelle für eine möglichst realistische Simulation benötigt, und dass dadurch ein wesentlich größerer Bedarf an leistungsfähiger Hardware besteht als bei den rein textuellen Verfahren.

Gemischte Verfahren
Bei den *gemischten Verfahren* basiert diese graphische Programmierung auf einer sensoriellen Erfassung einer Benutzervorführung. Dadurch muss nicht mehr die gesamte Umwelt des Roboters simuliert werden, sondern diese wird durch Sensoren erfasst. Da aber diese sensorielle Erfassung bis jetzt noch meist zu ungenau, und dadurch unbrauchbar ist, findet diese Methode noch nicht sonderlich viele Anwendungen. Es ist allerdings zu erwarten, dass sich dies in Zukunft stark ändern wird. Als Vorteile kann man, wie bei den *graphischen Verfahren*, den geringen Bedarf an Programmierkenntnissen und ein dadurch einfacheres Programmieren nennen, als Nachteil der Bedarf an leistungsstarker Hardware für komplexe Modelle.

1.2.3 Abstraktionsgrad der Programmierung

Die dritte und letzte Möglichkeit der Klassifizierung von Roboterprogrammierungen ist der Abstraktionsgrad. Man unterscheidet dafür die *explizite* und *implizite* Programmierung:

Explizite Programmierung
Die *explizite* Programmierung lässt sich gut durch die Frage „Wie ist es zu tun?“ charakterisieren. Z.B. bei der Bewegung eines Roboters von einem Punkt A zu einem Punkt B wird hier genau gesagt welche einzelnen Punkte abzufahren sind (um möglichen Hindernissen auszuweichen etc.). Dabei wird darauf vertraut, dass die einzelnen Befehle richtig und sinnvoll eingegeben wurden da keine spätere Überprüfung (z.B. durch eine Simulation der Umwelt) erfolgt.

Implizite Programmierung
Im Gegensatz dazu lässt sich die *implizite* Programmierung durch die Frage „Was ist zu tun?“ beschreiben. Betrachtet man hier die gleiche Bewegung eines Roboters von A nach B, so bekommt dieser nun einfach den Befehl nach B zu fahren. Mögliche Hindernisse muss er selbstständig erkennen und diesen gegebenenfalls ausweichen. Dafür ist natürlich eine genaue Modellierung der Umwelt erforderlich, und erst nach dieser ist eine Spezifikation der

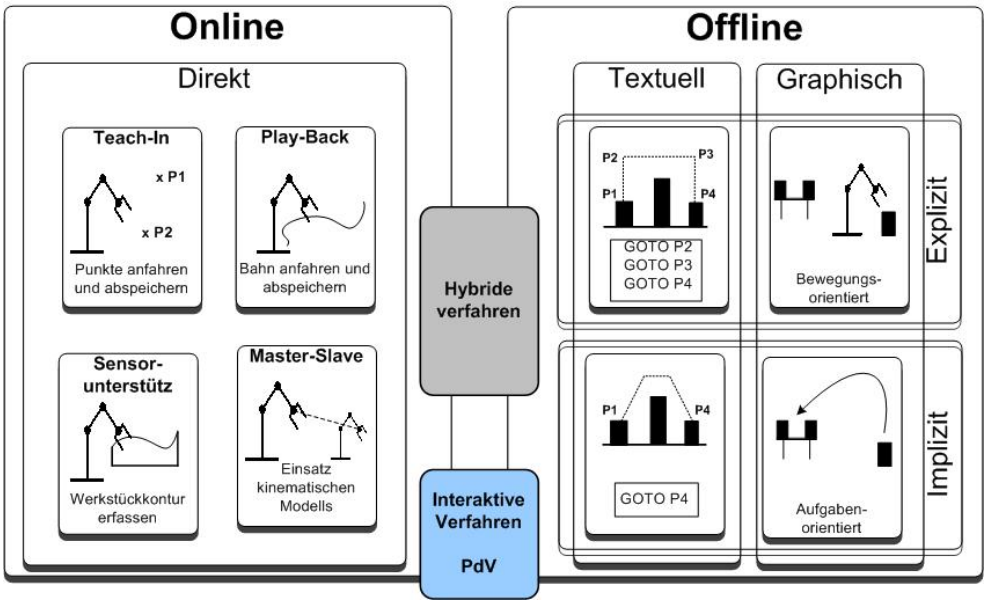


Abbildung 1.2: Roboterprogrammierverfahren

Aufgabe möglich woran sich dann die eigentliche Erzeugung des Programms anschließt. In manchen Fällen erfolgt vor der tatsächlichen Ausführung eine Überprüfung in Form einer Simulation, um mögliche Fehler rechtzeitig noch zu erkennen.

Abschließend sind nun noch einmal alle genannten Arten der Roboterprogrammierung graphisch dargestellt (Abb. 1.2). In den folgenden Kapiteln werden diese nun detaillierter behandelt, wobei diese graphische Veranschaulichung stets als Einordnung dienen sollte.

Kapitel 2

Direkte Programmierung

Zunächst sollen kurz die grudlegenden Verfahren der direkten Programmierung vorgestellt werden. Dies soll als Grundlage und dem Überblick dienen; der Fokus der Vorlesung Robotik 2 liegt nicht auf diesen Verfahren. Nichtsdestotrotz sind die Verfahren der direkten Programmierung ein wichtiges Hilfsmittel, das trotz (oder gerade wegen) seiner Simplizität nach wie vor in vielen Fällen Anwendung findet. Gerade für einfache Probleme ist eine solche einfache Programmiermethode durchaus sinnvoll.

2.1 Teach-In Programmierung

Bei der *Teach-In Programmierung* werden die Gelenke des Roboters durch manuelle Steuerung einzeln bewegt. So werden, für die vom Roboter auszuführende Arbeit, markante Punkte angefahren und die entsprechenden Gelenkstellungen abgespeichert. Wenn der Roboter z.B. eine gewisse Bahn abfahren soll werden einzelne Punkte der Bahn während des *Teach-In-Vorgangs* gespeichert, so dass sich diese dann als Folge von Zwischenpunkten darstellt.

Die manuelle Steuerung wie auch das Speichern oder Löschen von Anfahrpunkten erfolgt dabei über eine sog. *Teach-Box* oder *Teach-Panel*. Das Teach-Panel des VRS1 Roboters von VW zeigt Abb. 2.1.

Weitere Funktionen einer Teach-Box sind unter anderem die Eingabe von Geschwindigkeiten, die Eingabe von Befehlen zur Bedienung eines Greifers sowie das Starten bzw. Stoppen von ganzen Programmen. Außerdem ist es damit auch möglich im Nachhinein gespeicherte Werte um gewisse Parameter zu ergänzen, z.B. Geschwindigkeiten oder Beschleunigungen.

Als weiteres Eingabegerät bei der Teach-In Programmierung sei die Space-Mouse zu nennen (Abb. 2.2 zeigt die Spacemouse PlusXT von Concentric). Bei dieser wird ihre Kugel an einem Kraft-Momenten-Sensor befestigt, wodurch Kräfte als x,y,z Abweichungen und Momente als Drehungen interpretiert werden. Dadurch kann man im ganzen 6 Freiheits-

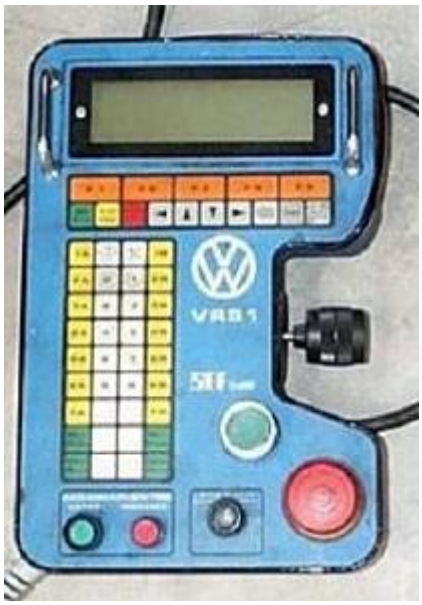


Abbildung 2.1: Teach-Panel des VW vrs1

grade erfassen und erhält eine, im Gegensatz zu einer normalen Maus oder eines Joysticks, hohe Positionsgenauigkeit.

Weitere Anwendungsgebiete der Teach-In Programmierung befinden sich hauptsächlich in der Fertigungsindustrie (z.B. Schweißen), oder auch in Handhabungsaufgaben (z.B. Pakete vom Fliessband nehmen).

Abbildung 2.2: Spacemouse PlusXT von Concentric

2.2 Play-Back Programmierung

Auch bei der *Play-Back Programmierung* werden die gewünschten Gelenkwerte durch manuelle Steuerung abgespeichert. Allerdings erfolgt dies hier nicht über eine Teach-Box oder Ähnliches, sondern der Roboter wird hier tatsächlich durch den Benutzer bewegt. Dabei wird dieser auf *Zero-Force-Control* eingestellt, was heißen soll, dass sich der Roboter ohne Widerstände durch den Benutzer bewegen lässt. Es wird also die zu befahrende Bahn abgefahren und während dessen die Gelenkwerte abgespeichert. Dies kann auf zwei Arten passieren:

- automatisch durch eine vordefinierte Abtastfrequenz oder
- manuell durch Tastendruck

Da hier also keine einzelnen Punkte mit ihren entsprechenden Gelenkstellungen eingestellt werden müssen, können dadurch relativ einfach mathematisch schwer zu beschreibende Bewegungen gemacht werden, im Gegensatz zur *Teach-In Programmierung*. Ein weiterer Vorteil dieser Art der Direkten Programmierung ist, dass der Bediener die Möglichkeit hat seine eigenen Erfahrungen in den Bewegungsvorgang zu integrieren. So weiß dieser z.B. aus Erfahrung am besten wie bestimmte komplexere Aufgaben oder Bewegungen zu erledigen sind.

Es treten bei der *Play-Back Programmierung* allerdings auch häufig Probleme auf. So bekommt der Bediener schnell Probleme wenn größere, und dadurch meist unbeweglichere, Roboter zu programmieren sind. Dies geschieht vor allem dann wenn wenig Platz zur Verfügung steht, und der Roboter dadurch schlechter zu rangieren ist. Dadurch steigt zwangsläufig auch das Sicherheitsrisiko, was diese Programmierart für manche Situationen unmöglich macht. Manchmal ist es auch unabdingbar eine sehr hohe Abtastrate zu verwenden, z.B. wenn die abzufahrende Bahn sehr genau einzuhalten ist (wegen möglichen Hindernissen etc.); dann kann es schnell passieren dass der benötigte Speicherbedarf sehr stark in die Höhe geht. Als letzter Nachteil sei hier nun noch genannt, dass sobald alle Werte gespeichert wurden, nur noch schlecht Korrekturen vorgenommen werden können; manchmal muss gar der Roboter komplett neu programmiert werden.

Typische Anwendungsgebiete sind hier z.B. Lackier- oder Klebearbeiten.

2.3 Master-Slave Programmierung

Bei der dritten hier vorgestellten Art der *Direkten Programmierung* erfolgt die Programmierung des Roboters über einen Master-Roboter (dieser entspricht einem kinematischen Modell des tatsächlichen Roboters). Dieser wird vom Bediener geführt und ist meist wesentlich kleiner und dadurch beweglicher als der zu programmierende Slave-Roboter.

Der Bediener führt also die zu machende Bewegung auf dem Master-Roboter aus, und diese wird dann auf den Slave-Roboter übertragen, wobei beide Bewegungen synchron ausgeführt werden. Da der Slave-Roboter normalerweise größer und dadurch stärker ist, wirkt dieser als Kraftverstärker, d.h. es können z.B. tatsächlich größere Lasten gehoben werden als mit dem Master-Roboter.

Man erkennt hier schnell das durch diese Vorgehensweise in erster Linie einen der größten Nachteile der *Play-Back Programmierung* behoben wird, nämlich die schlechten Möglichkeiten der Programmierung größerer und schwererer Roboter. Durch den Einsatz des beweglicheren Master-Roboters erhält man hier eine verbesserte Beweglichkeit und dadurch ein verringertes Sicherheitsrisiko für den Bediener. Allerdings ist die Tatsache nicht zu vernachlässigen, dass dafür zwei Roboter benötigt werden, was dieses Programmierverfahren sehr teuer werden lässt.

Eingesetzt wird die *Master-Slave Programmierung* hauptsächlich wenn größere Lasten oder auch größere, schwerere Roboter gehandhabt werden müssen.

2.4 Sensorunterstützte Programmierung

Bei der letzten hier beschriebenen Art der *Direkten Programmierung* ist der Bediener nicht direkt am Roboter (wie z.B. bei der *Play-Back Programmierung*), sondern markiert die abzufahrende Bahn mit Hilfe eines Programmiergriffels. Zum Einsatz kommen hierbei Leucht- oder auch Laserstifte. Um die damit beschriebene Bahn erfassen zu können, benötigt der Roboter externe Sensoren, z.B. Kameras oder Laserscanner, je nachdem welcher Art der vom Bediener benutzte Programmiergriffel ist. Die Bahn wird hierbei wieder als Folge von Gelenkwinkel abgespeichert; davor muss nun allerdings noch die inverse Kinematik berechnet werden, d.h. welches Gelenk wie bewegt werden muss damit der Roboter tatsächlich die vorgegebene Bahn anfährt.

Neben diesem manuell auszuführenden Punkten der *Sensorunterstützten Programmierung* erfolgen die Vorgabe des Start- bzw. Zielpunktes sowie die sensorische Erfassung der Sollkontur automatisch.

Anwendungen findet diese Art hauptsächlich im Schleifen sowie Entgraten von Werkstücken.

2.5 Zusammenfassung

Nachdem nun die wichtigsten Arten der *Direkten Programmierung* vorgestellt wurden, sollen nun noch einmal zusammenfassend die wichtigsten Merkmale dieser Programmierart aufgeführt werden.

Als Vorteile sind hier zu nennen, dass die Direkte Programmierung

- schnell bei einfachen Bewegungen ist.

- sofort anwendbar ist.
- eine geringe Fehleranfälligkeit hat.
- dem Bediener keine Programmierkenntnisse abverlangt.
- kein Modell der Umwelt benötigt.

Andererseits gibt es auch folgende Nachteile:

- Bei komplexen Bewegungen wird der benötigte Aufwand extrem hoch.
- Die Programmierung ist nur mit und am Roboter möglich.
- Jede Programmierung ist spezifisch nur für einen Robotertyp, also nicht übertragbar.
- Es gibt eine nicht unwesentliche Verletzungsgefahr durch den Roboter.

Die *Direkte Programmierung* findet demnach hauptsächlich bei relativ einfache, sich wiederholenden Aufgaben Anwendung. Vor allem der Vorteil der einfachen Bedienbarkeit, und die damit verbundene Möglichkeit einen Roboter ohne Programmierkenntnisse zu bedienen, ist, besonders für kleine und mittlere Unternehmen, entscheidend. Möglichkeiten komplexere Aufgaben auf Robotern zu realisieren werden nun in den Folgenden Kapiteln vorgestellt (Kapitel 3: Roboterorientierte Programmierung bzw. Kapitel 4: Aufgabenorientierte Programmierung).

Kapitel 3

Roboterorientierte Programmierung

In diesem Kapitel wird die *Roboterorientierte Programmierung* beschrieben, welche sich weniger am Anwender orientiert (siehe Kapitel 4 Aufgabenorientierte Programmierung), als an der zu programmierenden Maschine; natürlich aber nicht so stark wie die direkte Programmierung. Es besteht allerdings ein Unterschied zwischen roboterorientierter und expliziter Programmierung, bei welcher dem zu programmierenden Roboter genau (explizit) gesagt wird welche einzelnen Schritte er durchzuführen hat, um eine gewisse Aufgabe zu erledigen. Es ist allerdings leicht einzusehen dass eine roboterorientierte Programmierung meist auch gleichzeitig explizit ist. Wir befinden uns also auf der rechten Seite von Abb. 1.2 oben.

Um die *Roboterorientierte Programmierung* einordnen zu können, betrachte man eine komplexe Fertigungsaufgabe, welche, wenn sie so an einen Roboter gestellt wird, sicher als *aufgabenorientiert* einzuordnen ist. Zerlegt man diese komplexe Aufgabe nun über Einzelaufgaben, Einzeloperationen über primitive Bewegungen bis hin zu Koordinatentransformationen oder Servoregelungen, hat man sich von der aufgabenorientierten zur roboterorientierten Programmierung bewegt. Diese Art der Programmierung kapselt dabei folgende Eigenschaften zu einer Roboterorientierten Steuerung:

- Ansteuerung der internen und externen Hardware
- Bewegungsaktionen
- Lokale Modelle
- Elementare Operationen.

3.1 Anforderungen und Komponenten

Allerdings werden an den Roboter für diese Art der Programmierung schon einige Anforderungen gestellt, teilweise im Unterschied zur direkten Programmierung:

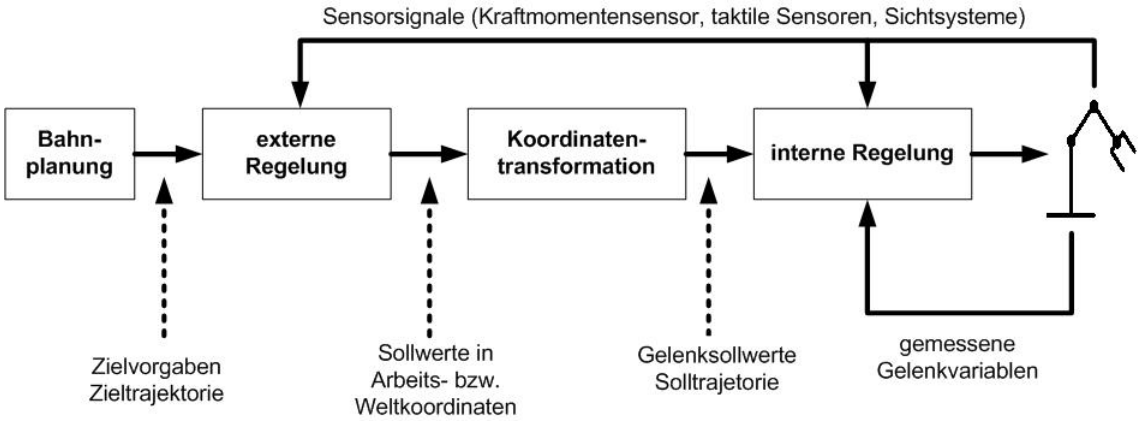


Abbildung 3.1: Regelungszyklus eines Roboters

So benötigt dieser zur Ausführung von, möglicherweise primitiven, Bewegungsaktionen eine Positions- und Geschwindigkeitsregelung der Räder, Gelenke, Greifer oder anderer aktiver Komponenten. Um z.B. erkennen zu können wo sich der Roboter auf einer abzufahrenden Bahn befindet, oder in welcher Stellung sich ein Greifer befindet, muss er in der Lage sein, seine externe und interne Sensorik auslesen und parametrieren zu können. Des Weiteren muss er in der Lage sein Koordinatentransformationen durchzuführen, bzw. die direkte/inverse Kinematik berechnen zu können. Dies ist nicht bei allen Arten dieser Programmierung notwendig, mehr dazu in Abschnitt 3.2.1 Bewegungsanweisungen. Außerdem muss er Trajektorien zu komplexen Bewegungen zusammensetzen können (z.B. zu einem Griff); sowie mehrere von solchen komplexen Bewegungen zu Elementaroperationen verketten können. Als Beispiel eines aus zwei komplexen Bewegungen zusammengesetzten Operators sei die Verkettung einer Armbewegung und eines Griffs genannt.

Die *Roboterorientierte Programmierung* lässt sich in mehrere größere Komponenten teilen:

Zu Beginn steht natürlich die Bahnplanung, nach welcher Zielvorgaben bzw. Zieltrajektorien an die **externe Regelung** übergeben werden. Diese ermittelt dann die Sollwerte in Arbeits- bzw. Weltkoordinaten. Um zu wissen wie z.B. einzelne Gelenke bewegt werden müssen, bedarf es noch einer Koordinatentransformation. Diese bestimmt die Sollwerte für **interne Regelung**. Diese regelt de Bewegung der Robotergelenke, wobei Sensorsignale während der Ausführung stets an die externe und interne Regelung zurückgegeben werden. Der Regelungszyklus eines Roboters ist in Abb. 3.1 graphisch dargestellt.

3.2 Sprachelemente

In einer Roboterprogrammiersprache sollten etliche Sprachelemente integriert sein, um Befehle an den Roboter stellen zu können. So werden z.B. Befehle benötigt für:

- die Bewegung des Roboters oder mehrerer Roboter
- den Betrieb von Werkzeugen / Greifern
- die Ein- und Ausgabe von Daten / Signalen über Schnittstellen
- die externe Sensorik
- die Synchronisation und Kommunikation zwischen Prozessen
- die logische Verkettung von Koordinatenanweisungen

Vor allem auf die ersten beiden Befehlsarten wird nun speziell genauer eingegangen:

3.2.1 Bewegungsanweisungen

Bei fast allen Aufgaben, die an einen Roboter gestellt werden können, werden Befehle zur Bewegung benötigt. Dabei kann man prinzipiell zwei Arten unterscheiden, wie diese Anweisungen erfolgen können:

Angabe der Gelenkwinkel
Bei dieser Art der Bewegungsanweisung wird den Gelenken gesagt um welchen Winkel sie sich drehen sollen, d.h. die Angabe erfolgt hier nicht in Weltkoordinaten. Dies kann entweder unkoordiniert, d.h. die Gelenke bewegen sich mit maximaler Geschwindigkeit bis sie die vorgegebene Stellung erreicht haben, oder koordiniert, d.h. mit Regelung der Geschwindigkeit geschehen. Letztere Variante hat den Vorteil, dass die resultierende Bahn näher an der idealen Bahn liegt als bei der unkoordinierten Bewegung.

Leicht einzusehen bei diesen Bewegungsanweisungen ist, dass man keine inverse Kinematik benötigt, da keine Koordinatentransformationen durchgeführt werden müssen. Als weiterer Vorteil hierbei gilt die Möglichkeit, dass alle Positionen sehr genau eingestellt werden können, und man außerdem eine sehr hohe Wiederholgenauigkeit erreicht. Allerdings sind die Angaben der Winkel immer abhängig vom jeweiligen Robotertyp, sodass man sie nicht auf andere Roboter übertragen kann. Auch besteht kein Bezug der Gelenkwinkel zur tatsächlichen Objektlage, da der Roboter keinerlei Informationen in Weltkoordinaten erhält.

Kartesische Bewegungen
Im Gegensatz zur Angabe jedes Gelenkwinkels, wird hier nun die gewünschte Stellung des Endeffektors in Weltkoordinaten dargestellt. Der Roboter muss nun selbst berechnen wie er dazu seine einzelnen Gelenke bewegen muss, d.h. es wird eine inverse Kinematik benötigt. Die Bewegung erfolgt nun entweder als lineare kartesische Bewegung oder als kartesische Bewegung auf einer Bahn.

Bei der *linearen Bewegung* wird Ausgangs- und (ein einziger) Zielpunkt sowie die Geschwindigkeit der Maschine vorgegeben, und die Bewegung erfolgt dann linear zum gewünschten Ziel. Der TCP bewegt sich dabei auf einer Geraden im kartesischen Raum.

$$MoveS(P_{ausgang}, P_{ziel}, Speed, WAIT/NOWAIT)$$

Zur *Bewegung auf einer Bahn* werden normalerweise mehrere Punkte bis hin zum Zielpunkt vorgegeben, und die Bewegung erfolgt dann abhängig vom *Bahntyp*.

$$MoveS(F_{ausgang}, F_{ziel}[n], PathType, Speed, WAIT/NOWAIT)$$

Solche Bahntypen sind unter anderem:

- Polygonzug / lineare Interpolation
- lineare Interpolation mit Überschleifen
- Splines, Bezier-Kurven
- Kreisbahnen

Beim *Polygonzug* werden die Zielpunkte linear, durch Geraden, mit einander verbunden, und ist somit als einfachster Bahntyp anzusehen. Dies hat den großen Nachteil dass an den einzelnen Punkten undifferenzierbare Stellen auftreten, d.h. der Polygonzug macht dort einen Knick. Dies macht diesen Bahntyp für viele Bewegungen bzw. Roboter nicht anwendbar, bzw. andere Bahntypen sind oftmals besser zu befahren. Allerdings werden die einzelnen, vorgegebenen, Bahnpunkte hier auch tatsächlich abgefahren, was nicht bei allen Bahnen der Fall ist.

Bei der *linearen Interpolation mit Überschleifen* wurde das Problem der undifferenzierbaren Stellen an den Stützstellen gelöst, indem an diesen Punkten entstehenden Kanten „abgerundet“ werden. Dadurch wird die Bahn zwar für viele Roboter besser befahrbar, allerdings wird eine weitere Überprüfung der Bahn notwendig. Theoretisch können sich nämlich Hindernisse auf der neu berechneten Bahn, liegen die nicht auf dem linearen Polygonzug liegen.

Eine *Bezier-Kurve* der Dimension n ist eine Kurve, welche nur an der ersten und n-ten Stützstelle diese tatsächlich auch annimmt. Alle weiteren Stützellen werden nur angenähert, wobei die Steigungen im Start- bzw. Zielpunkt mit denen des Polygonzuges übereinstimmen. Genauso wie beim "Überschleifenïst hier auch eine Überprüfung der Bahn auf Hindernisse etc. notwendig. *Splines* (der Dimension n) sind Funktionen, welche aus Polynomen zusammengesetzt sind, an den Stützstellen aber (n-1 mal) stetig differenzierbar bleiben. Es ist z.B. auch möglich Splines aus Bezier-Kurven zusammenzusetzen.

Die *Kreisbahn* ist im Prinzip selbsterklärend, wobei auch hier zu beachten ist, dass nur selten tatsächlich alle vorgegebnen Punkte erreicht werden können. So ist ein Kreis durch drei vorgegebene Punkte eindeutig bestimmt, d.h. sobald vier oder mehr Punkte vorgegeben werden ist es nur selten möglich einen Kreis exakt durch diese zu legen. Eine spätere Überprüfung der Bahn ist dann also auch hier notwendig; undifferenzierbare „Ecken“ entstehen hier natürlich, genauso wie bei den beiden zuletzt genannten Bahntypen, nicht.

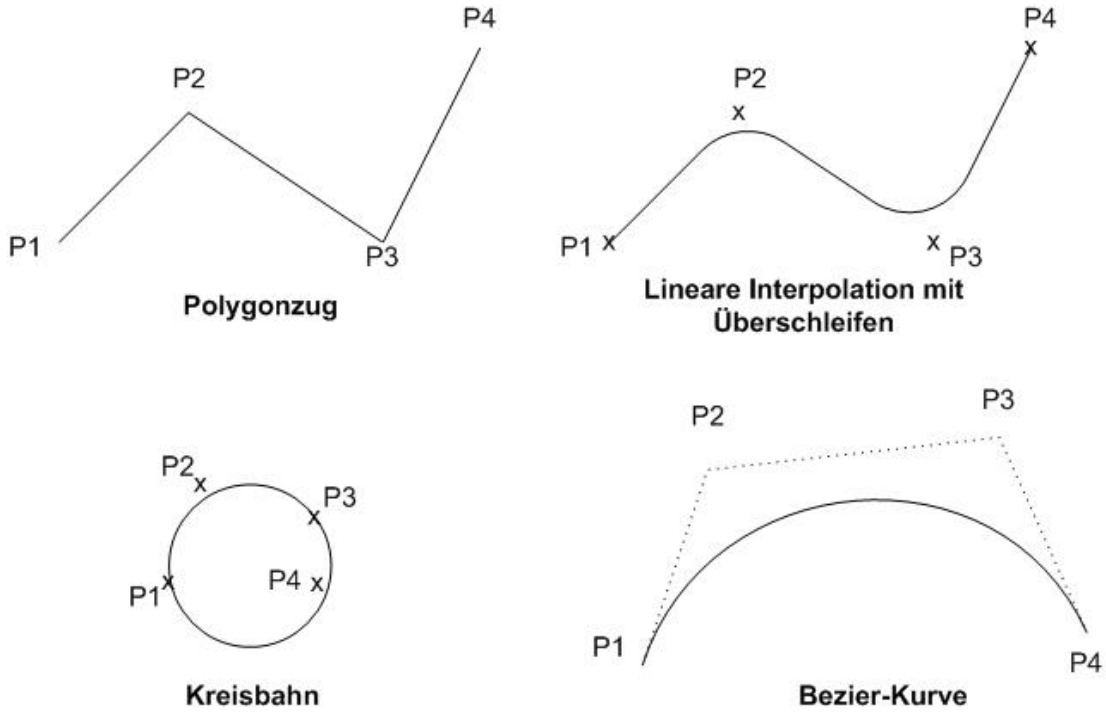


Abbildung 3.2: Bahntypen für kartesische Bewegungsanweisungen

Alle vier genannten Bahntypen sind nun noch einmal graphisch in Abb. 3.2 dargestellt.

3.2.2 Greiferbefehle

Für die Handhabung von Greifern kann man, genauso wie bei den Bewegungsanweisungen, verschiedene Abstraktionsgrade unterscheiden. So kann man die Steuerung auch hier im Gelenkwinkel-Raum vornehmen, oder im kartesischen (zylindrischen,...) Raum. Die dritte Stufe ist hier nun die semantische Steuerung.

Steuerung im Gelenkwinkel-Raum
Bei der Steuerung im *Gelenkwinkel-Raum* wird wieder jedem einzelnen Gelenk des Greifers ein bestimmter Wert zugewiesen. Dabei wird die Anzahl der übergebenen Parameter durch die Anzahl der Freiheitsgrade bestimmt. So verfügt ein simpler Backengreifer nur über einen Freiheitsgrad, ihm muss bei der Steuerung also auch nur ein einziger Parameter übergeben werden. (Zum Vergleich: Die menschliche Hand verfügt über 22 Freiheitsgrade.)

Genauso wie bei den Bewegungsanweisungen ist hierbei der Vorteil, dass man keine inverse Kinematik benötigt wird, da keine Koordinatentransformationen durchgeführt werden müssen. Allerdings bestehen auch die gleichen Probleme, nämlich dass eine derartige Steuerung nicht übertragbar ist, da die Konfigurationen abhängig vom jeweiligen Greifer sind.

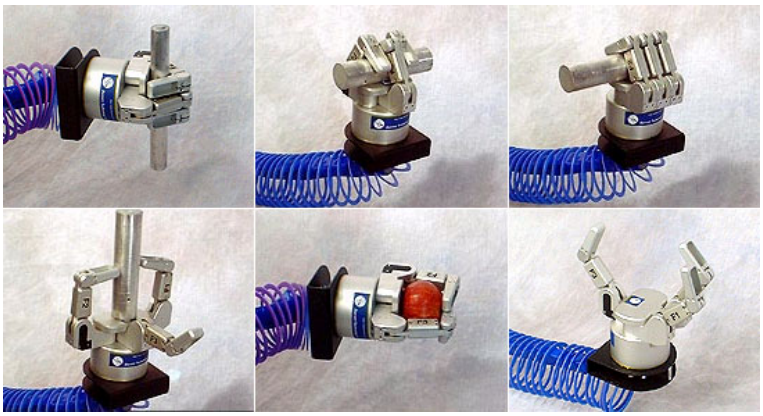


Abbildung 3.3: BarrettHand BH8-262

Außerdem besteht kein Bezug zwischen den einzelnen Fingerstellungen und der Stellung der gesamten Hand.

Als Beispiel sei die BarrettHand, eine 3-Finger Hand, betrachtet (Abb. 3.3): Ein Befehl für eine solche Hand kann z.B. so aussehen:

$$BarrettHand(Spread : 0, F1 : 15000, F2 : 12000, F3 : 15000)$$

Steuerung im kartesischen Raum
Als Parameter werden nun nicht mehr einzelne Gelenke, bzw. ihre Winkel oder Stellungen, angegeben, sondern die Position/Orientierung jedes Fingers/jeder Fingerspitze in kartesischen Koordinaten. Da dies in Handkoordinaten geschieht, wird nun aber eine inverse Kinematik benötigt, d.h. der Kapselungsaufwand ist im Vergleich zur Steuerung im Gelenkwinkel-Raum viel größer.

Auch hier ist ein Nachteil die fehlende Übertragbarkeit auf andere Hände, da jede Hand unterschiedliche Konfigurationen erlaubt (unterschiedliche Konfigurationsräume).

Betrachtet man als Beispiel ebenso die BarrettHand aus Abb. 3.3, so könnte ein Befehl folgendermaßen aussehen:

$$BarrettHand(F1 : [5, 5, 0], F2 : [5, 0, 5], F3[0, 0, 5])$$

Semantische Steuerung
Die dritte Abstraktionsstufe von Greiferbefehlen ist die *Semantische Steuerung*, bei der nun nicht mehr speziell jedem Finger ein Parameter übergeben wird. Vielmehr wird dem Roboter nun die zu benutzende Griff-Form, das zu greifende Objekt oder Eigenschaften von diesem, wie z.B. seine Größe oder Form, als Parameter übergeben. Um den gewünschten Griff auch durchführen zu können ist nun aber natürlich ein Mapping auf die Roboterhand nötig. Der Kapselungsaufwand ist im Vergleich zu den beiden schon genannten Arten der Greiferbefehle am größten (inverse Kinematik ebenfalls wieder nötig).

Allerdings hat man nun den großen Vorteil, dass man die Steuerung auf andere Roboterhände übertragen kann. Zu beachten ist noch, dass die möglichen Griffformen von der entsprechenden Roboterhand abhängen; genauso können unterschiedliche Roboterhände unterschiedliche Objekte greifen, abhängig z.B. von ihrer Größe.

Ein Befehl für die BarrettHand (Abb. 3.3) könnte nun z.B. so aussehen:

$$BarrettHand(circular - grasp, 12cm)$$

3.2.3 Ein- und Ausgabebefehle

Als letztes Sprachelement der roboterorientierte Programmierung wird nun die Ein- bzw. Ausgabe von Signalen über *Schnittstellen* betrachtet. Solche Schnittstellen verbinden einen Steuerrechner mit externer Hardware, wie z.B. Kameras oder auch Antriebe, und können durch die Art der Daten definiert werden, die bei der Ausführung zwischen den Modulen ausgetauscht werden. [DR91]

Am Steuerrechner können unterschiedliche Schnittstellen existieren (abhängig vom anzuschließenden Gerät). Solche Schnittstellen sind z.B. (incl. Beispiel für angeschlossene Hardware).

- FireWire (Kameras)
- CAN-Bus (Greifer, Roboterarme)
- seriell (Antriebe, Laserscanner)

Um die Daten, die über die unterschiedlichen Schnittstellen des Steuerrechners ausgetauscht werden, benötigt man auf diesem noch verschiedene Bibliotheken. So wird z.B. zur Steuerung eines Roboterarms (über CAN-Bus) Funktionen zur Berechnung der inversen Kinematik gebraucht, oder zur Steuerung eines Laserscanners (serieller Anschluss) Funktionen zum Starten bzw. Stoppen von diesem.

3.3 Realisierungen expliziter Programmiersprachen

Nachdem nun einige Basisanweisungen der expliziten Roboterprogrammierung vorgestellt wurden, wird nun beschrieben wie konkret textuelle Programmiersprachen realisiert werden können.

Man kann zu Beginn einige Grundlagentypen unterscheiden. So unterscheiden sich z.B.:

- Punkt zu Punkt Sprachen $\leftrightarrow$ Höhere Programmiersprachen
- Roboterabhängige Programmiersprachen $\leftrightarrow$ Roboterunabhängige Programmiersprachen

- Strukturierte Programmierung ↔ Objektorientierte Programmierung

Neuentwurf
Durch einen kompletten Neuentwurf einer Programmiersprache entsteht eine *roboterspezifische Programmiersprache*. In der Regel wird hier für jeden Robotertyp, bzw. von jedem Roboterhersteller, eine eigene Sprache entwickelt; dadurch sind solche Sprachen natürlich nicht auf andere Roboter übertragbar. Der Hersteller gibt dabei auch die Möglichkeiten der Ansteuerung des Roboters durch vorgegebene Schnittstellen vor, wodurch eine sehr gute Ansteuerung möglich wird. Ein großer Nachteil bei diesem Verfahren ist allerdings der sehr große Aufwand, da alle Programmteile neu geschrieben werden müssen.

Weiterentwicklung von Steuerungssprachen
Im Gegensatz zum Neuentwurf werden nun vorhandene Steuerungssprachen weiterentwickelt, was die Realisierung weniger aufwendig macht, da nicht mehr alle Programmteile neu geschrieben werden müssen. Schon vorhandene NC Sprachen, welche speziell zur Steuerung von Werkzeugmaschinen entworfen wurden, waren teilweise auch für Roboter konzipiert: so lies sich APT (NC-Sprache) zu RAPT für Roboter weiterentwickeln, oder EXAPT (NC-Sprache) zu ROBEX.

Erweiterung höherer Programmiersprachen
Auf der anderen Seite können aber auch höhere Programmiersprachen um Roboter-/Greiferbefehle erweitert werden, um sie für Roboter nutzen zu können. Aus der Programmiersprache Pascal wird so beispielsweise, durch Ergänzung solcher Befehle, PasRo (Pascal for Robots). In der Regel entstehen hier meistens Compilersprachen. Dieses Verfahren hat, gegenüber den vorher genannten Methoden der Realisierung expliziter Programmiersprachen, etliche Vorteile:

- gute Kontrollstrukturen, formatierte Ein-/ Ausgabe
- mögliche Nutzung von vorhandenen mathematischen Bibliotheken
- unterstützende Software vorhanden da höhere Programmiersprache bekannt

3.4 Die normierte Programmier-Schnittstelle IRDATA

Wie sich im letzten Abschnitt (3.3 Realisierungen expliziter Programmiersprachen) erkennen lässt, ist ein großer Nachteil der roboterorientierten Programmierung die sehr große Anzahl an unterschiedlichen Programmiersprachen. Dadurch lässt sich eine Sprache oftmals nur für genau einen Roboter nutzen; genauso ist es möglich, dass sich ein Roboter auch nur von genau einer Sprache steuern lässt.

Um genau diesen Nachteil zu beheben ist die Erzeugung eines roboterunabhängigen Codes von Vorteil. Ein solcher Code ist z.B. der *IRDATA-Code*. Durch seine standardisierten und somit festgeschriebenen Sprachelemente kann man IRDATA als neutrale Schnittstelle

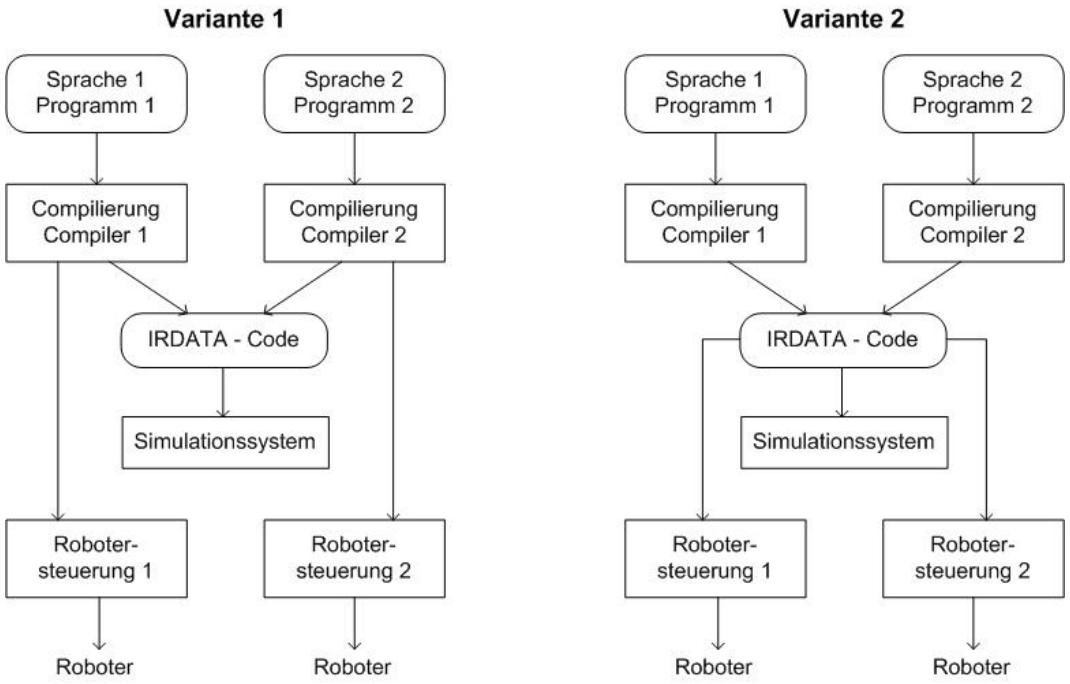


Abbildung 3.4: Schaubilder der Nutzung von IRDATA-Code

zwischen verschiedenen Robotersprachen auf der einen, und unterschiedlichen Robotersystemen auf der anderen Seite ansehen. Dazu wird durch die *Sprach-Compiler* IRDATA-Code generiert, welcher von den Robotersteuerungen verstanden werden muss. Dann ist es möglich einen Roboter durch mehrere Sprachen programmieren zu können, bzw. ein Programm mit unterschiedlichen Robotern auszuführen. [DR91]

IRDATA ist also eine interne, normierte Schnittstelle (VDI 2863) zwischen (mehreren) Anwendern und (mehreren) Roboterprogrammiersprachen. Bei der Nutzung dieses Codes kann man zwei Varianten unterscheiden, welche in Abb. 3.4 schematisch dargestellt sind.

Variante 1
Hier werden die unterschiedlichen Programmiersprachen durch die jeweiligen Compiler kompiliert und daraus IRDATA-Code gewonnen. Dieser Code wird dann an ein Simulationssystem übergeben, welches vor der tatsächlichen Robotersteuerung ausgeführt wird. Die Erzeugung des Maschinencodes erfolgt aber, genauso wie der IRDATA-Code, aus den verschiedenen Programmiersprachen. Die Validierung des IRDATA-Programms erfolgt dabei mit Standardwerkzeugen.

Variante 2
Genauso wie bei der ersten Variante wird nun der IRDATA-Code durch Kompilierung der unterschiedlichen Programmiersprachen gewonnen und an das Simulationssystem übergeben.
Der Unterschied besteht darin, dass der Maschinencode nun auch durch IRDATA-Repräsentationen

erzeugt und validiert wird, und nicht mehr durch die unterschiedlichen Programmiersprachen.

3.5 Graphische Programmierverfahren

Alle bis jetzt behandelten Sprachelemente, sowie auch die IRDATA-Schnittstelle, fallen unter den Bereich textuelle Verfahren (vgl. Abb. 1.2). Nun sollen abschließend noch einige Aspekte der *graphischen Programmierverfahren* beschrieben werden.

Die graphische Roboterprogrammierung ist gekennzeichnet durch die Verwendung von graphischen Eingabegeräte [DR91]. Dabei stellt man Kontrollstrukturen, wie z.B. if/else, Schleifen oder Marken, graphisch dar, und simuliert dadurch eine online-Programmierung, z.B. durch Simulations-Tools wie RobCAD. Trajektorien entstehen hier durch Interpolation aus Stützpunkten, Freihandzeichnungen oder analytisch.

Dieses Programmierverfahren hat verschieden Vorteile gegenüber den textuellen Verfahren:

- schnelles Erstellen von Prototypen dank graphischer Simulation (Rapid prototyping)
- Validierung das Programms durch Simulation
- komfortable Programmierumgebung

Allerdings entstehen dabei oftmals erheblich höhere Kosten als bei den textuellen Programmierverfahren; die 2-D Sicht des Anwenders ist auch oftmals ein Nachteil bei diesem Verfahren. [Güs92]

3.5.1 Beispiel

Als Beispiel sei hier noch abschließend in diesem Abschnitt das System **Lego Mindstorms** genannt. Für dieses Konzept gibt es zahlreiche graphische Programmierumgebungen, wie z.B. RoboLab oder RCX-Code. Diese Programmierverfahren sind in der Regel sehr einfach, da ikonisch, zu programmieren und man sieht in der Regel direkt was man gerade programmiert, z.B. Schleifen etc.

Im RCX-Code von MindStorms werden Programme durch Aneinanderfügen grafischer Blocks aufgebaut (Abb. 3.5). Durch simples *drag-n-drop* können so sehr schnell einfache Programme entstehen. Dadurch erreicht man, dass zur Steuerung eines Roboters keine hochqualifizierten Fachkräfte benötigt werden, sondern dass dies nach sehr kurzer Einarbeitungszeit jeder kann. Genau dies bedeutet oftmals den größten Vorteil im Vergleich zu den textuellen Verfahren. Dies gilt allerdings nur bis zu einer gewissen Komplexität der Programme, da die graphischen Programmierverfahren doch relativ früh an ihre Grenze stoßen können.

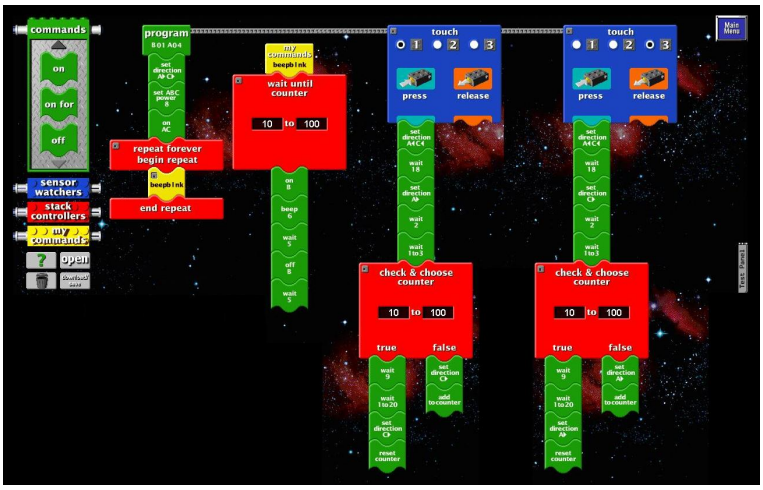


Abbildung 3.5: RCX-Code - graphische Programmierung für Lego Mindstorms

Kapitel 4

Aufgabenorientierte Programmierung

Die Aufgabenorientierte Programmierung wird dem Bereich der impliziten Programmierung zugerechnet (siehe Abb. 1.2 auf der rechten Seite unten). Ausgerichtet ist dieser Ansatz auf den Anwender und weniger auf die zu programmierende Maschine - im Gegensatz zur expliziten oder gar direkten Programmierung (vgl. Kapitel 3 bzw. 2): Der Anwender gibt nicht einzelne Lösungsschritte vor, sondern eine Beschreibung der Aufgabe selbst; es bleibt dann einem *Planungssystem* überlassen, die nötigen Abläufe zu ermitteln. Um seine Aufgabe erfüllen zu können, muss das Planungssystem über verschiedene Modelle verfügen: Erforderlich sind ein Umweltmodell, ein Modell der Aufgabe aber auch des Roboters selbst.

Zur Erstellung des fertigen Plans hat ein Planungssystem verschiedene Schritte abzuarbeiten: Darunter die Wissensauswertung, d.h. die Interpretation der vorgegebenen Modelle, die Aufgabenplanung, wozu auch die Identifizierung von Teilzielen gehört, die Bewegungsplanung und Sensorüberwachung. Wir werden hierauf bei der Besprechung von konkreten Planungssystemen zurückkommen. Zunächst soll aber die Erstellung von Modellen diskutiert werden, die die Wissensgrundlage für die Arbeit eines Planungssystem darstellen. Abbildung 4.1 zeigt die notwendigen Teile eines umfassenden Modells in einer hierarchischen Gliederung. Die einzelnen Bestandteile werden in den folgenden Abschnitten genauer erläutert.

4.1 Umweltmodell

Das Umweltmodell liefert dem Planungssystem eine Beschreibung der Umgebung, in der sich Aufgaben für den Roboter ergeben können. Es besteht seinerseits aus zwei Teilen: Dem *Objektmodell* und dem *Szenenmodell* (Abb. 4.1). Im Objektmodell werden die einzelnen Gegenstände beschrieben, die sich in der Umgebung des Roboters befinden und bei möglichen Aufgaben eine gewisse Rolle spielen könnten (z.B. Tische oder Werkstücke). Diese Beschreibung umfasst sowohl geometrische Formen, als auch weitere (problembezogene) Eigenschaften; man denke hier etwa an Griffpunkte oder die Beschaffenheit einer

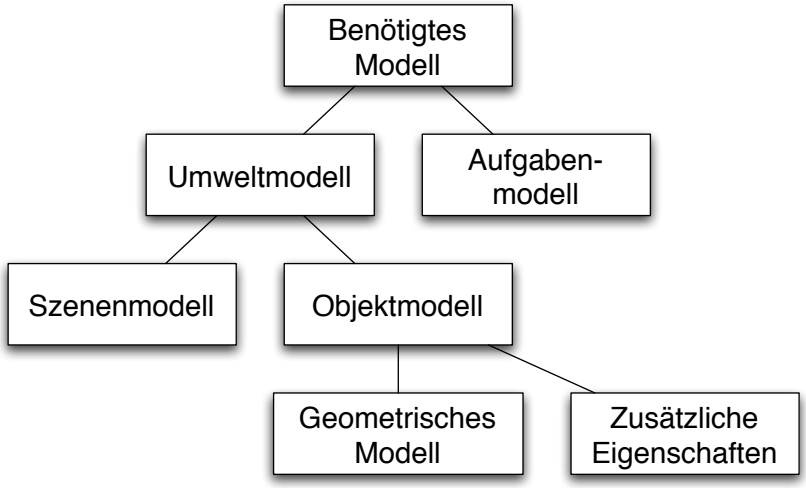


Abbildung 4.1: Bestandteile eines Modells zur aufgabenorientierten Programmierung

Oberfläche. Auf der anderen Seite bildet das Szenenmodell Zusammenhänge zwischen verschiedenen Objekten ab - wiederum geometrische („Werkstück liegt auf dem Tisch“), aber auch solche mit einer problembezogenen Semantik („Tasse kann mit Inhalt der Flasche gefüllt werden“).

Die Übergänge zwischen Objekt- und Szenenmodellierung sind aber keinesfalls scharf, sondern vielmehr fließend und in hohem Maße von der gestellten Aufgabe abhängig: Man stelle sich beispielsweise einen Serviceroboter vor, der in einem Büro verschiedenen Tätigkeiten nachgehen soll; die Einrichtungsgegenstände dieses Büros ließen sich als eigenständige Objekte modellieren, oder aber als feste Bestandteile der Raumgeometrie. Welche Art der Modellierung ist zu bevorzugen? Um diese Frage zu beantworten, sollten die Anforderungen an den Roboter bekannt sein: Bei einen Putzroboter wäre die Modellierung als Teil der Raumgeometrie bereits ausreichend - für einen solchen Helfer stellt das Mobiliar schließlich nichts anderes dar, als eine Menge von Hindernissen, die es zu umfahren gilt. Ganz anders sieht die Situation bei einen Roboter aus, der mit so komplexen Anweisungen umgehen können soll, wie „nimm die rote Tasse vom Tisch und bringe sie zur Kaffeemaschine zurück“: Bevor er eine Tasse vom Tisch aufzunehmen vermag, muss sich der Roboter erst einmal der Existenz eines solchen Möbelstücks bewusst sein.

Im Folgenden werden beide Teile der Umweltmodellierung besprochen; den Anfang macht die Objektmodellierung.

4.1.1 Objektmodell

Wie bereits bei der Einführung in die Umweltmodellierung angeklungen ist, teilt sich die Modellierung einzelner Objekte auf in die geometrische Modellierung und die Beschreibung struktureller, problembezogener Eigenschaften.

Der erste Teil dieses Abschnitts wird sich mit der geometrischen Modellierung beschäftigen, d.h. mit Techniken zur Formbeschreibung. Die genaue Kenntnis der Objektgeometrien spielt in der aufgabenorientierten Programmierung eine bedeutende Rolle, und zwar gleich aus mehreren Gründen: Meist besteht ein wesentlicher Teil bei der Umsetzung einer gestellten Aufgabe im Abfahren von Trajektorien, sei es zur Fortbewegung des gesamten Roboters oder zur Führung eines Endeffektors, mit dem Gegenstände gegriffen oder anderweitig manipuliert werden können; eine Bewegungsplanung zur Erstellung solcher Trajektorien setzt notwendigerweise auch ein Geometriemodell der Umgebung voraus, müssen doch mögliche Kollisionen mit potentiellen Hindernissen erkannt und vermieden werden. Verwendung für das Geometriemodell gibt es zudem auch nach Abschluss der Planungsphase: Üblicherweise erfolgt dann eine Simulation, verbunden mit einer Visualisierung, in der der erstellte Plan auf Fehler getestet wird - eine geometrische Beschreibung betroffener Objekte ist hierfür unabdingbar.

Zur Erstellung geometrischer Modelle kann man sich verschiedener Ansätze und Techniken bedienen; hier werden vorgestellt:

- Methoden zur Beschreibung von Kurven und von dreidimensionalen Objekten durch deren begrenzende Kanten
- Methoden zur Beschreibung von Flächen und wiederum von dreidimensionalen Objekten durch deren begrenzende Oberflächen
- Methoden, die direkt auf dreidimensionalen Grundkörpern basieren.

Flächen statt Kanten zu verwenden, Grundkörper statt Flächen, mag zwar jeweils Unzulänglichkeiten der vorigen Methode beseitigen, wirft aber umgekehrt auch neue Schwierigkeiten auf; daher ist keine dieser Methoden für alle Situationen gleichermaßen gut (oder schlecht) geeignet und ein wesentlicher Aspekt der Modellierung besteht zunächst darin, eine der konkreten Problemstellung angemessene Technik auszuwählen – verschiedene Beispiele werden diese Notwendigkeit im weiteren Verlauf des Kapitels deutlich machen.

Im zweiten und letzten Teil dieses Abschnitts wird die Modellierung struktureller Objektmerkmale diskutiert: Da die richtige Auswahl von Merkmalen jedoch von den zu erwartenden Aufgaben abhängt, werden wir es dabei belassen, typische Vertreter vorzustellen und anschließend noch auf Möglichkeiten zur Integration solcher Daten in das geometrische Modell eingehen.

In gewisser Weise darf auch die Beschreibung der Kinematik und Dynamik einzelner Objekte der Umgebung (oder des Roboters selbst) zur Objektmodellierung gezählt werden.

Auf diese Aspekte wird hier jedoch nicht eingegangen und es sei stattdessen auf die weiterführende Literatur und die Vorlesung „Robotik 1“ verwiesen.

4.1.1.1 Kurvenmodell

Kurven stellen die wohl grundlegendste Art der Formgebung dar, die über die Angabe reiner Punkte hinausgeht. Im Folgenden werden Verfahren beschrieben, die zum rechnergestützten Entwerfen von Kurven geeignet sind und ferner wie sich aus solchen Kurven dreidimensionale Körper aufbauen lassen.

Ein erster Ansatz zur rechnergestützten Kurvendarstellung könnte darin bestehen, die zu beschreibende Kurve als Bildmenge einer Funktion $f : \mathbb{R} \to \mathbb{R}^2$ für den zweidimensionalen bzw. $f : \mathbb{R} \to \mathbb{R}^3$ für den dreidimensionalen Fall aufzufassen, den gewünschte Verlauf also gleichsam analytisch und in geschlossener Form anzugeben. Dass dieser Ansatz in der Praxis jedoch höchst selten verwendet wird hat mehrere Gründe:

- Häufig ist es nur mit erheblichem Aufwand möglich eine Funktion zu bestimmen, deren Graph die gewünschte Form aufweist (man denke beispielsweise an den Umriss eines neuen Automobils).
- War es dennoch möglich eine Funktion mit den gewünschten Eigenschaften zu finden, stellt deren Auswertung das nächste Problem dar: Bei Polynomen mag dies noch rasch von statten gehen, doch bei „hinreichend“ komplexeren Funktionen könnten selbst moderne Rechner bei einer Echtzeitdarstellung nicht mehr über ausreichende Rechenleistung verfügen.
- Eine nachträgliche Änderung des Kurvenverlaufs ist nur durch Neubestimmung der gesamten Funktion möglich.

Stattdessen wird zumeist der Ansatz verfolgt eine Menge von *Stütz-* oder *Führungspunkten* vorzugeben und diese durch eine Kurve zu *interpolieren* bzw. zu *approximieren*. Diese Herangehensweise unterstützt zudem die interaktive Modellierung, bei der der Benutzer die Lage der Stützpunkte variieren, neue Stützpunkte hinzufügen und andere Kurvenparameter (wie etwa Tangential- oder Krümmungseigenschaften) anpassen kann, um so die Kurve nach und nach in die gewünschte Form zu bringen.

Polynominterpolation
Ein nahe liegender Ansatz zur Interpolation von $n+1$ vorgegebenen Stützpunkten (x_0, y_0) bis (x_n, y_n) mit $x_i \neq x_j$ für $i \neq j$ besteht in der Verwendung eines Polynoms P vom Grade $\leq n$. Ein solches Polynom hat die Form

$$P = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X^1 + a_0.$$

Durch die $n + 1$ Bedingungen $P(x_i) = y_i$ für $i = 0, \ldots, n$ lassen sich die Koeffizienten $a_0, \ldots a_n$ bestimmen. Das Interpolationspolynom ist darüber hinaus eindeutig (was die

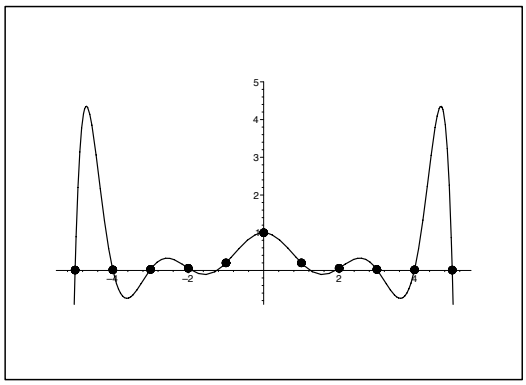


Abbildung 4.2: Starke Schwingungen bei der Polynominterpolation

Verwendung des bestimmten Artikels erst rechtfertigt): Ist nämlich Q ein weiteres Polynom mit $Q(x_i) = y_i = P(x_i)$ für $i = 0, \ldots, n$, so ist $P - Q$ ein Polynom ebenfalls vom Grad $\leq n$ mit $n + 1$ Nullstellen $x_0, \ldots, x_n$. Da jedes vom Nullpolynom verschiedene Polynom vom Grade $\leq n$ aber nicht mehr als n Nullstellen besitzen kann, muss es sich bei $P - Q$ um das Nullpolynom handeln. Damit ist $P = Q$. Methoden zur rechnerischen Bestimmung von P (beispielsweise mit Lagrange-Polynomen) werden in [Deu02] vorgestellt.

Die Polynominterpolation weist jedoch gewisse Eigenschaften auf, die sie für unsere Zwecke (und die gesamte Computergrafik) als ungeeignet erscheinen lassen:

- Für große n ist das Interpolationspolynom i.A. schlecht konditioniert.
- Durch Änderungen an einzelnen Knoten kann der gesamte Kurvenverlauf beeinträchtigt werden.
- Oftmals treten mit zunehmender Anzahl an Stützpunkten (und damit größerwerdendem Grad) zunehmend Oszillationen auf (vgl. Abbildung 4.2).

Die bei der Interpolation mittels Polynomen aufgetretenen Probleme wollen wir zum Anlass nehmen zu überlegen, welche Eigenschaften die eingesetzten Modellierungstechniken aufweisen sollten. Insbesondere wäre es wünschenswert, wenn

- sich Änderungen an einzelnen Punkten nur in einem begrenzten Bereich der Kurve auswirkten (Lokalität)
- sich bereits beim Setzen der Stützpunkte die Form der Kurve voraussagen ließe und
- neue Punkte jederzeit hinzugefügt und bestehende verändert werden könnten (Unterstützung der Interaktivität).

Splineinterpolation
Ein Ansatz, der die geforderten Eigenschaften aufweist, besteht in der lokalen Interpolation

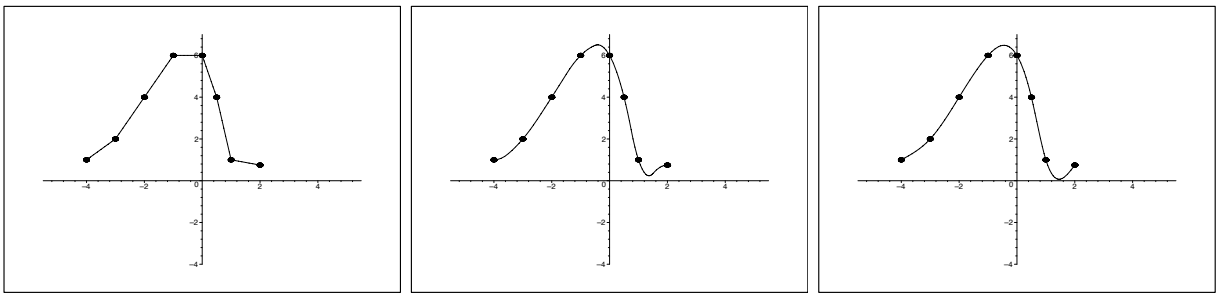


Abbildung 4.3: Lineare, quadratische und kubische Splineinterpolation

mit Polynomen niedrigen Grades und der „Verheftung“ dieser an den „Nahtstellen“. Zu $n+1$ Stützpunkte (x_i, y_i), $i = 0, \ldots, n$ auf einem Intervall $[a, b]$ mit

$$a = x_0 \leq x_1 \leq \cdots \leq x_{n-1} \leq x_n = b$$

ist ein *interpolierender Spline* s der Ordnung k definiert als eine $k-2$ mal stetig differenzierbare Funktion auf $[a, b]$, die die Stützpunkte (x_i, y_i) interpoliert, d.h. $s(x_i) = y_i$ erfüllt, und deren Einschränkungen auf die Teilintervalle $[x_i, x_{i+1}]$ Polynome vom Grad $\leq k-1$ sind. Durch s werden also je zwei aufeinander folgende Stützpunkte vermittels eines Polynoms vom Grad $\leq k-1$ verbunden, wobei sich der Übergang zwischen diesen Polynomen an den gemeinsamen Stützpunkten bis zur $(k-2)$-ten Ableitung „glatt“, d.h. stetig-differenzierbar, vollzieht. In Abbildung 4.3 sind Splines der Ordnungen $2, 3$ und 4 für eine feste Menge von Stützpunkten dargestellt.

Splines der Ordnung 2 heißen auch lineare Splines und ergeben sich gerade durch Verbinden der Stützpunkte mittels Geraden (Polynomen vom Grad 1). Wir wollen uns hier auf kubische Splines, also Splines der Ordnung 4, beschränken, da diese in der graphischen Datenverarbeitung weite Verbreitung gefunden haben und auch für unsere Zwecke bestens geeignet sind. Einer der Gründe für diese Entscheidung wird beim Vergleich der einzelnen Kurvenverläufe in Abbildung 4.3 deutlich: Erst ab der Ordnung 4 wird eine Splinekurve durch das Menschliche Auge als „glatt“ wahrgenommen: Lineare Splines weisen an den Stützpunkten ganz offensichtlich Ecken auf und auch bei quadratischen Splines fallen die Unstetigkeiten in der zweiten Ableitung noch auf (besonders deutlich zwischen der vierten und fünften, sowie der siebten und achten Stützstelle zu erkennen).

Sei nun $p_i = s|_{[x_i, x_{i+1}]}$, also die Einschränkung der Splinefunktion s auf das i-te Teilintervall $[x_i, x_{i+1}]$. Die p_i sind Polynome dritten Grades und können in der Form

$$p_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$$

dargestellt werden. Bei n Teilintervallen sind also insgesamt $4n$ Koeffizienten zu bestimmen.

Die Kriterien hierfür ergeben sich aus der Forderung nach der Interpolationseigenschaft des Splines und den Stetigkeitsbedingungen. So muss für $i = 1, \ldots, n-1$ gelten:

$$\begin{aligned} p_{i-1}(x_i) = y_i \ , \quad p_i(x_i) = y_i & \qquad \text{Interpolationseigenschaft} \\ p'_{i-1}(x_i) = f'_i(x_i) & \qquad \text{Stetigkeit in der ersten Ableitung} \\ p''_{i-1}(x_i) = f''_i(x_i) & \qquad \text{Stetigkeit in der zweiten Ableitung} \end{aligned}$$

Zusätzlich müssen noch (x_0, y_0) und (x_n, y_n) auf dem Spline liegen, also

$$p_0(x_0) = y_0 \quad \text{und} \quad p_{n-1}(x_n) = y_n$$

so dass sich insgesamt $4(n-1) + 2 = 4n - 2$ Gleichungen ergeben. Damit bleiben $4n - (4n-2) = 2$ Freiheitsgrade unbestimmt und können frei gewählt werden, um die Form des Splines festzulegen. Typische Randbedingungen sind:

- Vorgabe der Steigungen an den Randpunkten: $p'_0(x_0) = c$ und $p'_{n-1}(x_n) = d$. Dieser Splinetyp wird als *vollständige* kubische Splineinterpolation bezeichnet.
- Keine Krümmung an den Randpunkten: $p''_0(x_0) = p''_{n-1}(x_n) = 0$. Man spricht von der *natürlichen* Splineinterpolation.
- Für den Fall, dass der Verlauf als periodisch angenommen werden kann (insbesondere also $y_0 = y_n$ gilt) bietet es sich an zu fordern, dass Steigung und Krümmung an den Enden übereinstimmen, also $p'_0(x_0) = p'_{n-1}(x_n)$ und $p''_0(x_0) = p''_{n-1}(x_n)$ gilt. Naheliegenderweise wird dieser Ansatz als *periodische* Splineinterpolation bezeichnet.

Bezier-Kurven

Die Bezier-Technik wurde Ende der 1950er, Anfang der 1960er Jahre fast zeitgleich sowohl von Pierre Bezier bei Renault, als auch von Paul de Casteljau bei Citroën entwickelt. Zur Modellierung von Autokarosserien ersannen beide dieselbe Methode zur Kurvenbeschreibung, die heute, da es Pierre Bezier oblag die erste Veröffentlichung hierzu zu schreiben (bei Citroën wurden die Ergebnisse von Casteljau als Betriebsgeheimnis zurückgehalten), unter seinem Namen bekannt ist. Doch auch Paul de Casteljau kam noch zu Ehren, wurde doch der „Algorithmus von Casteljau“, ein sehr effizienter Algorithmus zur Auswertung von Bezier-Kurven, nach ihm benannt.

Im Gegensatz zu den bisher behandelten Techniken, der Polynom- und Splineinterpolation, handelt es sich bei der Beziertechnik nicht um eine Interpolations-, sondern vielmehr um eine Approximationstechnik: I.A. liegen nur der erste und der letzte der hier als *Führungspunkte* bezeichneten und vom Benutzer vorzugebenden Punkte auf der entstehenden Kurve. Auch in anderer Hinsicht unterscheidet sich dir Bezier-Technik von der Polynom- und Splineinterpolation: Waren dort Funktionen der Form $s : \mathbb{R} \to \mathbb{R}$ zu bestimmen (die also einem x- einen y-Wert zugeordnet haben), so haben wir es nun mit einer sog. *polynomiellen Kurve* vom Grad n in $\mathbb{R}^2$ zu tun. Dies ist eine Abbildung der Form

$$C : \mathbb{R} \to \mathbb{R}^2, \quad C(t) = \sum_{i=0}^{n} a_i t^i \quad \text{mit } a_0, \ldots, a_n \in \mathbb{R}^2$$

Exkurs zur Splineinterpolation:
Wir wollen unsere Aufmerksamkeit nun noch kurz auf die Herkunft des Namens lenken. Dazu betrachte man für eine zweimal stetig differenzierbare Funktion $f : [a, b] \to \mathbb{R}$ deren *Krümmung* $\kappa(t)$ in einer Stelle $t \in [a, b]$, die durch

$$\kappa(t) := \frac{f''(t)}{(1 + f'(t)^2)^{3/2}},$$

gegeben ist und sich für kleine $f'(t)$ durch die zweite Ableitung $f''(t)$ annähern lässt. Die Splinefunktion s hat nun unter allen Funktionen f, die die geforderten Bedingungen (natürlich, vollständig oder periodisch interpolierend, zwei mal stetig differenzierbar) erfüllen, die minimale Gesamtkrümmung, die durch

$$||f''(t)|| := \int_a^b f''(t)^2 \, dt$$

festgelegt wird. Beschreibt f den Verlauf einer dünnen Holzlatte, so misst

$$E := \int_a^b \kappa(t)^2 \, dt$$

die Biegeenergie dieser Latte. Die Latte wird sich so einstellen, dass E minimiert wird. Da aber

$$E \approx ||f''(t)||$$

gilt, beschreibt der kubische Spline damit annähernd die Lage einer solchen Holzlatte (engl. Spline), die an den Stützstellen z.B. durch Nägel fixiert ist – und tatsächlich wurden solche Holzlatten früher als Zeicheninstrumente eingesetzt (beispielsweise bei der Konstruktion von Schiffsrümpfen). Die vollständige Splineinterpolation entspricht dem Fall, dass die Steigung der Latte an den Endpunkten ebenfalls fest vorgegeben wird, während die Latte bei der natürlichen Interpolation außerhalb des festgelegten Bereiches gerade verläuft (da dies das natürliche Verhalten für eine Holzlatte darstellt, wäre damit auch Herkunft der Bezeichnung geklärt). Für die periodische Interpolation findet sich keine Entsprechung, da ein fortgesetzt periodischer Verlauf bei real existierenden Holzlatten bisher noch nicht beobachtet werden konnte.

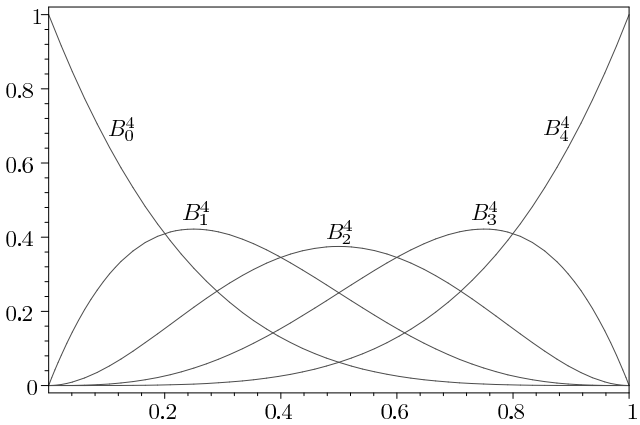


Abbildung 4.4: Bernsteinpolynome vom Grad 4

wobei wir hier t auf das Intervall $[0, 1]$ einschränken wollen.
Die Grundlage für die Bezier-Technik bilden die *Bernsteinpolynome*: Das i-te Bernsteinpolynom vom Grad n ist als die Abbildung

$$B_{i,n}(t) := \binom{n}{i} t^i (1-t)^{n-1} \quad \text{für } t \in [0,1]$$

definiert. Zusammen bilden die Bernsteinpolynome $\{B_{0,n}, B_{1,n}, \ldots, B_{n,n}\}$ eine Basis des Raumes der Polynome vom Grad $\leq n$, d.h. jedes solche Polynom lässt sich als Linearkombination der Bernsteinpolynome vom Grad n darstellen (Beweis in [Deu02]). Geschuldet dieser Tatsache lässt sich nun auch jede polynomielle Kurve $C(t)$ vom Grade n mit Hilfe der Bernsteinpolynome ausdrücken:

$$C(t) = \sum_{i=0}^{n} P_i B_{i,n}(t) \quad \text{für } t \in [0,1],$$

für geeignete *Bezierpunkte* P_i.

Bei der Bezier-Technik kann der Benutzer die Bezier-Punkte vorgeben und die Kurve ergibt sich dann nach obiger Formel. Dass die Form einer solchen Kurve eng mit ihren Bezierpunkten verknüpft ist, lässt Abbildung 4.5 bereits erahnen; wir wollen uns einige Aspekte dieser Korrespondenz im Folgenden noch genauer überlegen (oder wenigstens darauf hinweisen).
Zunächst fällt auf, dass erster und letzter Bezierpunkt auf der Kurve liegen. Der Grund dafür erschließt sich unmittelbar bei einem Blick auf Abbildung 4.4 und auf die Definition der Bernsteinpolynome: Für $t = 0$ (dies markiert den Anfang der Kurve) nimmt $B_{0,n}$ den Wert 1 an, während alle anderen Bernsteinpolynome den Wert 0 annehmen. Folglich ist $C(0) = P_0 \cdot 1 + 0$. Am anderen Ende der Kurve, also für $t = 1$, gilt entsprechend $C(1) = 0 + P_n \cdot 1$.
Darüber hinaus weisen die Bezierpunkte einer Kurve die folgenden geometrischen Eigenschaften auf (Beweise sind im schon häufig zitierten Werk [Deu02] zu finden):

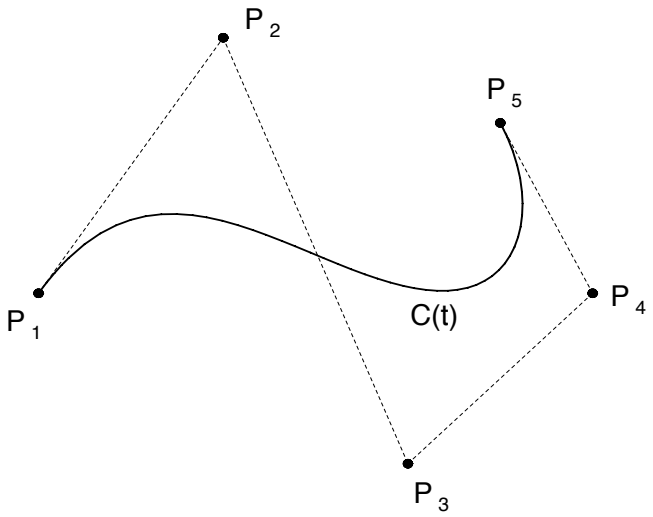


Abbildung 4.5: Bezier-Kurve mit Führungspunkten

- Die Tangenten in den beiden Endpunkten stimmen mit den Geraden zwischen dem ersten und zweiten bzw. dem letzten und vorletzten Bezierpunkt überein.
- Genauer: Die Kurve ist in den Randpunkten bis zur k-ten Ableitung durch die darauf folgenden (am Anfang) bzw. vorhergehenden (am Ende) k Bezierpunkte bestimmt.
- Die polynomielle Kurve verläuft vollständig in der konvexen Hülle ihrer Bezierpunkte.
- Eine Bezierkurve ist affin-invariant. Statt eine affine Abbildung direkt auf eine Bezierkurve anzuwenden, können also zunächst die Bilder ihrer Bezierpunkte bestimmt und diese anschließend durch eine neue Kurve verbunden werden; das Ergebnis ist dasselbe.

Die enge Beziehung zwischen dem Kurvenverlauf und den Bezierpunkten sowie die Existenz effizienter Algorithmen zum Zeichnen von Bernsteinpolynomen (basierend auf dem Algorithmus von Casteljau) haben die Bezier-Technik zu einem bedeutenden Werkzeug in der Computergrafik und geometrischen Modellierung werden lassen. In der bisher vorgestellten Form weist die Bezier-Technik jedoch einen gravierenden Nachteil auf, der den weiter oben angegebenen Vorstellungen hinsichtlich der gewünschten Eigenschaften von Modellierungstechniken widerspricht: Bereits die Änderung eines einzelnen Führungspunktes kann den Verlauf der gesamten Kurve beeinflussen (Verletzung der Lokalitätsforderung). Die im Folgenden vorgestellte B-Spline-Technik begegnet diesem Nachteil.

B-Splines

Die angesprochene globale Beeinflussung einer Bezierkurve durch Veränderung eines einzel-

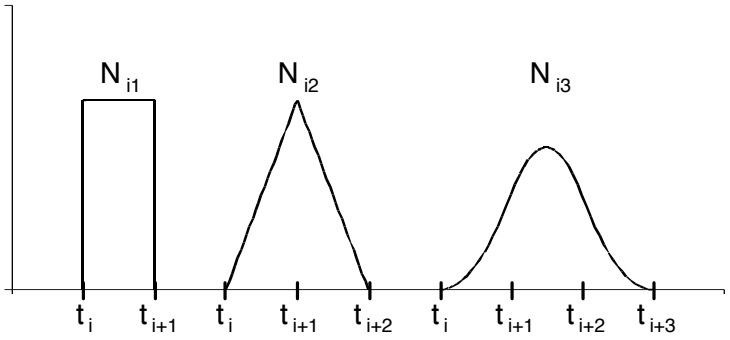


Abbildung 4.6: B-Splines der Ordnung $k = 1, 2, 3$

nen Führungspunktes liegt in den Eigenschaften der Bernsteinpolynome begründet: Abbildung 4.4 zeigt, dass diese zwar jeweils genau ein Maximum auf $[0, 1]$ annehmen (an dieser Stelle ist der Einfluss des entsprechenden Führungspunktes am größten) aber außer an den Randpunkten nie zu 0 auswerten. Jeder Führungspunkt beeinflusst daher den gesamten Verlauf der Kurve. Bei der Verwendung von B-Splines besteht dieses Problem nicht: Statt den Bernsteinpolynomen kommen hier Basisfunktionen mit lokalem *Träger* („Nichtnullstellenmenge“) zum Einsatz, wodurch Änderungen an einem Punkt auf die unmittelbare Umgebung beschränkt bleiben.

Sei $a = t_0 \leq \cdots \leq t_n = b$. Für den *Knotenvektor* $\Delta := (t_0, \ldots, t_n)$ sind die *B-Splines* der Ordnung k rekursiv definiert durch

$$N_{i1}(t) := \begin{cases} 1 & \text{falls } t_i \leq t \leq t_{i+1} \\ 0 & \text{sonst} \end{cases}$$

$$N_{ik}(t) := \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t)$$

Die N_{ik} sind stückweise Polynome vom Grad $\leq k-1$ und bilden gewissermaßen die Erweiterung der charakteristischen Funktion („Hutfunktion“) $\chi_{[t_i, t_{i+1})}$ auf das Intervall $[t_i, t_{i+k})$, wie Abbildung 4.6 illustriert. Der Parameter k legt demnach fest über wie viele Teilintervalle sich N_{ik} von t_i aus erstreckt; außerhalb dieses Bereichs ist N_{ik} stets 0. Durch die Konvention $\frac{0}{0} = 0$ wird ein Zusammenfallen von Knoten ermöglicht.

Für die nun folgenden Betrachtungen sollen einige Vereinfachungen getroffen werden: Fortan sei $k = 3$ gesetzt und außerdem seien die Knoten ganzzahlig und äquidistant gewählt. Kurzum: Zu den Führungspunkten $P_0, \ldots, P_n$ sei der Knotenvektor

$$\Delta = [0, 0, 0, 1, 2, \ldots, n-2, n-1, n-1, n-1]$$

betrachtet, und über diesem die B-Splines $N_{i,3}(t)$. Das $(k = 3)$-fache Zählen des ersten und letzten Knotens (man erinnere sich an die oben erwähnte Konvention für diesen Fall)

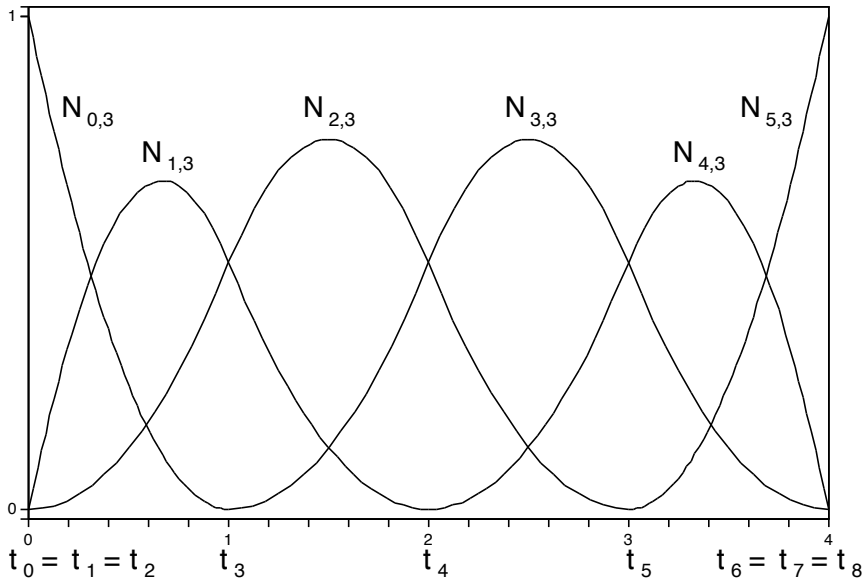


Abbildung 4.7: Quadratische B-Splines ($k = 3$) für $n = 5$

ist notwendig, um den $N_{i,3}$ eine nichtperiodische Form zu geben. Für $n = 5$ sind sie in Abbildung 4.7 dargestellt.

B-Spline-Kurven werden nun ganz analog zu den Bezierkurven über eine Konvexkombination von Führungspunkten definiert, nur dass statt der Bernsteinpolynome die B-Splines als Gewichtsfunktionen verwendet werden:

$$C(t) := \sum_{i=0}^{n} P_i N_{i,3}(t)$$

Die so definierten B-Spline-Kurven weisen u.a. die folgenden Eigenschaften auf:

- Erster und letzter Führungspunkt liegen exakt auf der Kurve (ein Blick auf Abbildung 4.7 genügt, um diese Eigenschaft einzusehen).
- An den Endpunkten Verlaufen die Linien P_0P_1 bzw. $P_{n-1}P_n$ tangential zur Kurve.
- Da bei B-Splines $N_{i,3}$ der Ordnung 3 nur im Teilintervall $[t_i, t_{i+3})$ von 0 verschiedene Werte angenommen werden, wirken sich Änderungen am Führungspunkt P_i auch nur in diesem Bereich aus.
- Anders ausgedrückt tragen für $t \in [t_i, t_{i+1})$ nur die B-Splinefunktionen $N_{i-3,3}, \ldots, N_{i,3}$ zum Verlauf der Kurve bei. Der Kurvenpunkt $C(t)$ liegt damit in der konvexen Hülle der Führungspunkte $P_{i-3}, \ldots, P_i$.

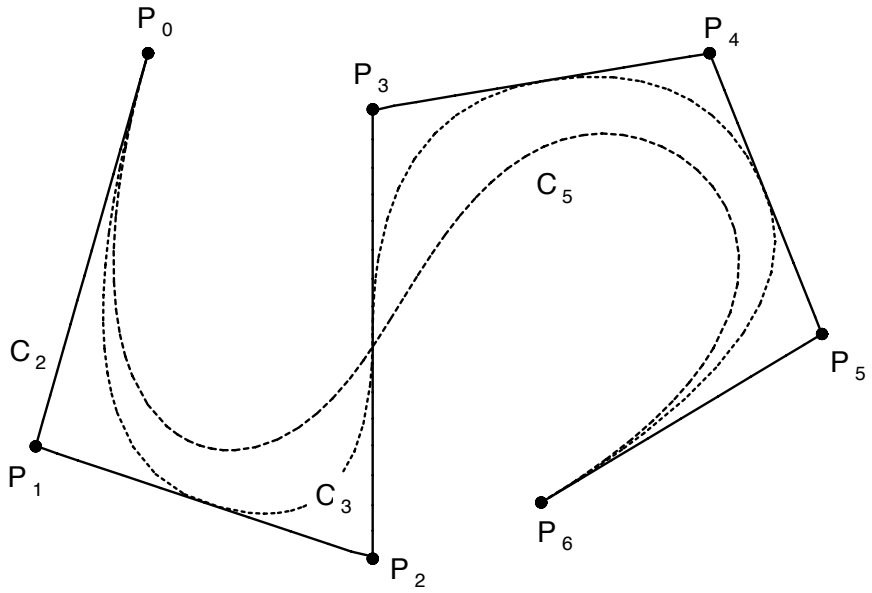


Abbildung 4.8: B-Spline-Kurven für $k = 2, 3, 5$

Der letztgenannte Punkt bedingt die guten Lokalitätseigenschaften der B-Spline-Technik und daher die Überlegenheit gegenüber der reinen Bezier-Technik. In Abbildung 4.8 sind zum Vergleich der Lokalitätseigenschaften für eine Menge von Führungspunkten B-Spline-Kurven verschiedener Ordnungen eingezeichnet. Man beachte wie für kleiner werdendes k die Kurve zunehmend enger an den Polygonzug durch die Führungspunkte heranrückt.

In der B-Spline-Technik vereinen sich also Konzepte aus der Bezier-Technik und der Splineinterpolation: Von der Bezier-Technik stammt die Idee der Modellierung durch Konvexkombination von Führungspunkten, aus der Splineinterpolation die der Verwendung stückweiser Polynome. Tatsächlich stellen Bezier-Kurven einen Spezialfall der B-Spline-Kurven dar: Für einen Knotenvektor, der nur aus k-fachem Anfangs- und Endknoten besteht sind nämlich die B-Splines der Ordnung k identisch mit den Bernsteinpolynomen vom Grad $k - 1$. Auch zur Splineinterpolation besteht ein direkter mathematischer Zusammenhang: Die B-Splines stellen gerade eine Basis des *Splineraumes* mit sehr guten Eigenschaften dar.

Auch für die Berechnung von B-Spline-Kurven existiert ein effizienter Algorithmus, nicht unähnlich dem Verfahren von Casteljau [Kop89]. Durch einfaches Zählen der Randpunkte lassen sich periodische B-Splines und damit geschlossene Kurven definieren. Darüberhinaus existieren Varianten und Erweiterungen der hier vorgestellten Methode, die es beispielsweise durch den Einsatz rationaler B-Spline-Funktionen (Quotienten zweier Polynome) erlauben, genaueren Einfluss auf die Gestalt und den Verlauf der Kurve zu nehmen.

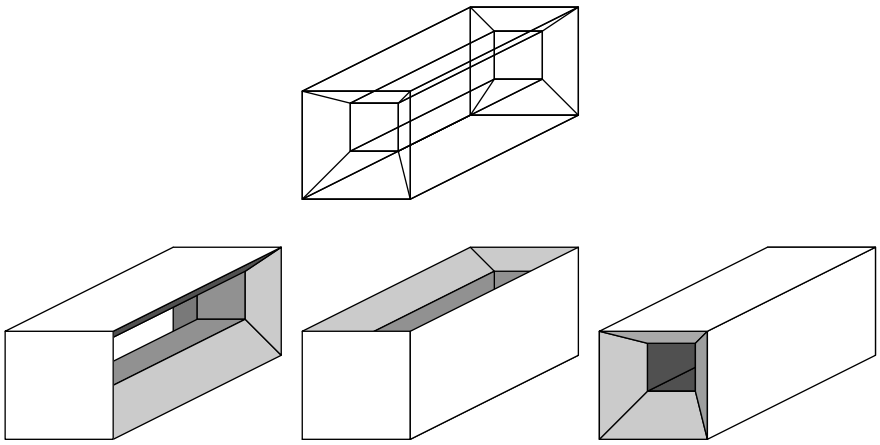


Abbildung 4.9: Mehrdeutigkeiten beim Kantenmodell

Körpermodellierung durch Kanten

Bisher wurden einige der gebräuchlichsten Darstellungsformen für Kurven vorgestellt. Im Folgenden soll geklärt werden, wie sich aus einzelnen Kurven, die selbst ja kein Volumen aufweisen, dreidimensionale Körper aufbauen lassen. Dazu kommt das *Kanten-* oder *Drahtmodell* zum Einsatz, bei dem die begrenzenden Kanten des zu modellierenden Objektes bestimmt, durch Kurven approximiert und diese dann zusammen mit Start- und Endpunkten gespeichert werden. Offensichtlich ist zur Speicherung bei dieser Darstellungsform nur eine geringe Datenmenge erforderlich und auch die Einfachheit der zu speichernden Daten lässt das Kantenmodell vorteilhaft erscheinen. Die Effizienz bei der Darstellung solcher Kantenmodelle war vor einigen Jahren ebenfalls ein gewichtiges Argument, das für die Verwendung dieses Ansatzes sprach, doch durch das Aufkommen schnellerer Rechner hat es an Bedeutung verloren. Dem stehen jedoch eine Reihe gravierender Nachteile gegenüber: Zunächst das der Mehrdeutigkeiten bei der Modellierung. So ist in Abbildung 4.9 aus dem reinen Kantenmodell (oben) nicht ersichtlich, welcher der in der unteren Zeile abgebildeten Körper tatsächlich beschrieben werden soll, da ja offene und geschlossene Flächen nicht unterschieden werden können.

Des Weiteren lässt das Kantenmodell prinzipiell die Konstruktion „unmöglicher“ Körper zu und Operationen, die die Volumeneigenschaften eines solchermaßen modellierten Körpers berühren, nicht sinnvoll durchführbar sind. Der zuletzt genannten Punkt soll anhand von Abbildung 4.10 erläutert werden: Modelliert wurde ein Quader anhand seiner begrenzenden Kanten; dieser Quader soll nun durch einen Schnitt im oberen Drittel aufgeteilt werden. Es ist aber anhand des reinen Kantenmodells nicht zu entscheiden, ob die entstehenden Teilkörper nun offen, geschlossen oder gar innen hohl sind.

Offensichtlich ist das reine Kantenmodell also nicht oder nur sehr eingeschränkt dazu geeignet, die eingangs formulierten Erfordernisse eines geometrischen Objektmodells im Rahmen der aufgabenorientierten Programmierung zu erfüllen: Weder können Schnittpunkte, -linien noch -flächen berechnet werden, eine Kollision mit anderen Objekten bliebe unentdeckt.

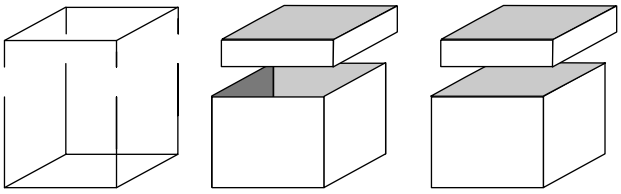


Abbildung 4.10: Schnittoperation beim Kantenmodell

Methoden zur Beschreibung der *Oberfläche* sind erforderlich und werden im nächsten Abschnitt vorgestellt.

4.1.1.2 Flächenmodell

Eine weitere Methode ein geometrisches Modell zu bilden besteht darin ein Flächenmodell einzusetzen. Dies bedeutet im Genauen dass hier nicht mehr eindimensionale Objekte die Grundlage der Modellierung bilden, sondern Zweidimensionale, also Flächen, diese Position einnehmen. Die Modellierung wird hierdurch etwas komplexer, dem gegenüber stehen aber einige Vorteile wie z.B. Eindeutige Objektdefinitionen (nicht wie bei Kantenmodell → „unmögliche“ Körper), Fläche muss nicht wie beim Kantenmodell angenommen planar sein (Flächen können Krümmungen – z.B. Bezierflächen – aufweisen.

Man unterscheidet bei der Flächenmodellierung grundsätzlich zwischen zwei Typen der Interpolation – finite Interpolationen und transfinite Interpolationen. Bei der finiten Interpolation sind nur endliche viele Stützstellen bzw. Kontrollpunkte vorgegeben. Im Gegensatz zu Transfiniten, bei denen eine unendlich Menge von Punkten vorgegeben ist (z.B. Randkurve bei Coons'schen Flächen).

In den meisten Fällen versucht man ein geometrisches Modell durch Zusammensetzen verschiedener einzelner Flächenelemente zu erstellen und die zu modellierende Umgebung ausreichend genau darzustellen.

Die Flächenelemente lassen sich durch verschiedene Methoden beschreiben, einige oft eingesetzte Methoden werden hier nun beschrieben:

analytisch gegebene Fläche
Hierbei wird versucht die gesamte zu modellierende Fläche durch eine Funktion zu beschreiben (es ist auch möglich Teilflächen analytisch zu modellieren, jedoch ist es schwierig diese dann später zu verbinden). Im 3-Dimensionalen wäre z.B. durch

$$z(x, y) = f(x, y) \quad \text{jedem } (x, y)\text{-Paar analytisch eine „Höhe“ zugewiesen (3D).}$$

Daraus ergibt sich ein exaktes Modellierungsverfahren, aus welchem für bestimmte Aufgabenbereiche entscheidende Vorteile resultieren:

- geschlossene Darstellung, d.h. jeder benötigte Punkt kann durch simples Einsetzen

in die Funktionsgleichung bestimmt werden. Aus der geschlossenen Darstellung ergibt sich auch eine gute Eignung für visuelle Darstellungsformen. Außerdem benötigt man nur sehr geringen Speicherplatz, da nur die Funktion und nicht einzelne Punkte abgespeichert werden müssen (siehe Freiformflächen)

- Analytische Darstellung, daraus resultieren einfache und beliebig exakte Rechenverfahren (z.B. Schnitt von Ebenen usw.)

Nachteile:

- nur wenige Flächen lassen sich analytisch darstellen, bzw. es muss viel Zeit investiert werden entsprechende Funktionen zu ermitteln.
  (z.B. gesucht: Funktion die eine Sitzfläche eines Stuhles genau beschreibt)

Bildung einer großen Fläche aus einem Netz von Einzelflächen
→ Freiformflächen, z.B. Dreiecke, Vierecke
Durch Approximation mit vielen kleinen Elementarflächen wird die gesamte zu modellierende Fläche zusammengesetzt. Man spricht hier von einer Freiformfläche, die sich analytisch nicht genau beschreiben lässt, sondern nur Approximation zulässt. Hier bieten sich verschiedene einfache (z.B. Viereckfläche), wie auch komplexere (z.B. Bezier-Flächen, Coons'sche Flächen) Flächen an.

Dreiecksfläche

Angefangen mit der einfachsten Fläche, der Dreiecksfläche, die nur aus 3 Eckpunkten besteht. Sie lässt sich wie folgt definieren: Gegeben seien 3 Punkte im Raum P_1, P_2, P_3. Damit hat die Fläche folgende Gleichung:

$$F(u,v) = u \cdot P_1 + v \cdot P_2 + (1-u-v) \cdot P_3 \quad \text{mit } u \leq u+v \leq 1$$

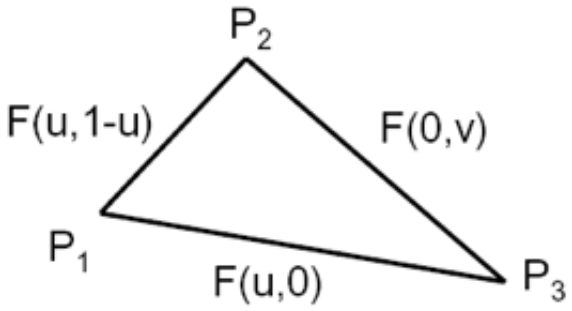


Abbildung 4.11: Dreiecksfläche mit Eckpunkten P_1, P_2 und P_3

Zur Erstellung von Dreiecksflächennetzen muss die nächste angrenzende Dreiecksfläche natürlich zwei derselben Eckpunkte besitzen wie die erste, um eine

geschlossene Fläche zu bilden. CAD-Programme besitzen auch meist Dreiecksflächennetze um Objekte zu modellieren, daher ist eine externe Verarbeitung damit leicht möglich (außerdem: direkt darstellbar mit OpenGL → Array von 3D Punkten).

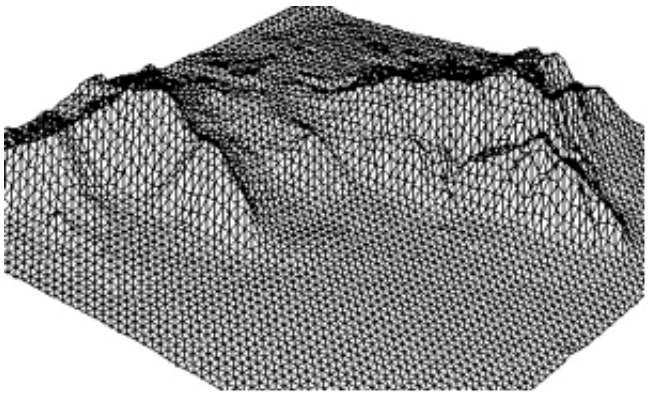


Abbildung 4.12: Dreiecksflächennetz

Bilineare Viereckselemente / Pflaster

Noch genauere Modellierungen werden möglich wenn als eingesetzte Elementarfläche eine Viereckige Fläche gewählt wird. Dazu interpoliert man in beide Parameterrichtungen u, v linear (bilineare Interpolation). Dies liefert dann ein bilineares Flächenstück (im Allgemeinen ist dies keine Ebene, sondern ein parabolisches Hyperboloid, was dem Ausschnitt aus - bzw. einer Sattelfläche entspricht. Den Interpolationsvorgang in Worten ausgedrückt: Wenn sich die Punkte P_1 und P_2 nach P_3 bzw. P_4 auf geradem Weg bewegen dann bilden die Verbindungsgeraden (zwischen P_1, P_2) die interpolierte Fläche (→ überstrichene Fläche der mitgeführten Verbindungsgeraden)

Diese wird durch folgende Gleichung beschrieben:

$$F(u,v) = (1-u)(1-v) \cdot P_1 + (1-u)v \cdot P_2 + u(1-v) \cdot P_3 + uv \cdot P_4$$
$$\text{mit } 0 \leq u \leq v \text{ und } 0 \leq v \leq 1$$

Vorteile:

- Flächen können gekrümmt sein (weniger Gitterpunkte, bei gleich guter Approximation)

Nachteile:

- Rechnen mit gekrümmten Flächen ist aufwendiger

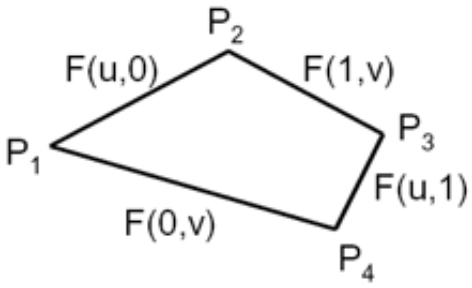


Abbildung 4.13: Viereckfläche mit Eckpunkten P_1, P_2, P_3 und P_4

Bezierflächen

Die schon im vorigen Abschnitt vorgestellten Bezierkurven werden nun dazu verwendet Flächen zu modellieren. Der Ansatz von Bezierkurven wird erweitert auf Flächen. Somit ergibt sich die Definition:

Gegeben ist ein Gitter von Führungspunkten P_{ij}

$$0 \leq i \leq N \quad \text{und} \quad 0 \leq j \leq M$$

Im Allgemeinen liegen nur die 4 Eckpunkte auf den Kurven, die restlichen Führungspunkte liegen außerhalb der Fläche.
Damit ist die Fläche beschrieben durch:

$$F(u,v) = \sum_{i=0}^{N} \sum_{j=0}^{M} P_{ij} \cdot B_{i,N}(u) \cdot B_{j,M}(v)$$

mit

$$B_{i,N} = (1-u)B_{i,N-1}(u) + uB_{i-1,N-1}(u)$$
$$B_{j,M} = (1-v)B_{j,M-1}(v) + vB_{j-1,M-1}(v)$$

Die Ausdrücke $B_{i,N}$ bzw. $B_{j,M}$ heißen auch Bernsteinpolynome. Wenn sich benachbarte Bezier-Patches die Kontrollpunkte jeweils „teilen“ dann ist ein stetiger Übergang zwischen beiden Flächenelementen gewährleistet. Zum Erzeugen eines glatten Übergangs müssen jedoch weitere Bedingungen eingehalten werden.

Coons'sche Flächen (auch „Coons-Patch“, „Coons-Pflaster“)/ Lofting:

Als Grundlage für diesen Typ dient die bilineare Interpolation, jedoch mit Erweiterung dass man an den Rändern 4 frei definierbare Funktionen verwenden kann. Es wird mit Hilfe der jeweils gegenüberliegenden Kurven bilinear interpoliert. Daraus folgt dass man den Teil der „standard“ bilinearen Interpolation (Randkurven = gerade Verbindung zwischen den Punkten) einmal zuviel addiert hat, zieht man nun diesen einmal ab erhält man die gewünschte Fläche.

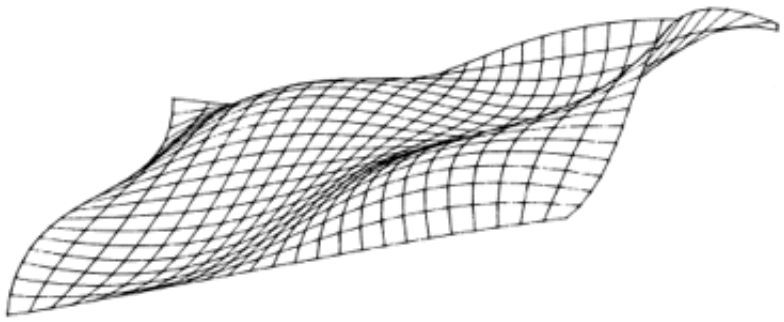


Abbildung 4.14: Bezierfläche

- Gegeben: 4 parametrisierte Randkurven des Pflasters
- Lineare Interpolation der Fläche für gegenüberliegende Flächen (lofting)
- Kombination der resultierenden Flächen durch Addition und Korrekturterm

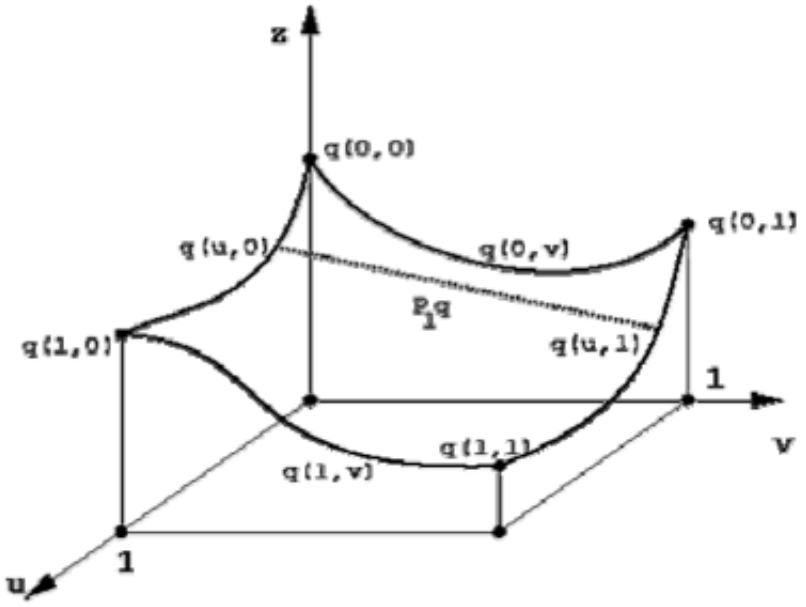


Abbildung 4.15: Coons'sche Flächen

Die beiden letzteren Interpolationsschemata bilden „glattere“ Flächen an den Übergangstellen der Grundelemente.

Eine weitere Form von gekrümmten Kurven sind NURBS (Non-Uniform Rational B-Splines). NURBS-Flächen sind die allgemeinsten Arten von Flächen, alle anderen Flächen sind Vereinfachungen und können mit diesen NURBS beschrieben werden. Die Berechnung von NURBS ist aber deutlich aufwendiger.

Körpermodellierung durch Flächen
Analog zum bereits vorgestellten Kantenmodell (siehe Abschnitt 4.1.1.1) werden beim *Flächen-* oder *Oberflächenmodell* die zu modellierenden Körper über deren begrenzende Flächen beschrieben. Gegenüber dem Kantenmodell liegen andere Vor- und Nachteile vor: Die Modellierung selbst gestaltet sich ungleich aufwendiger, doch lässt sich ein wesentlich genaueres Abbild schaffen (was sich auch bei der Visualisierung bemerkbar macht - hier natürlich erkauft durch wesentlich höheren Rechenaufwand). Der wohl bedeutendste Kritikpunkt an Kantenmodellen, die fehlende Möglichkeit Kollisionen zwischen Objekten sicher zu erkennen, wird durch den Einsatz begrenzender Flächen behoben: Sobald sich Objekte durchdringen führt dies auch unweigerlich zur Überschneidung von deren Oberflächen, wobei sich nun natürlich eine ganz neue Frage stellt, namentlich die nach dem Außen und Innen eines Körpers. Hierzu wurde die Idee der *Orientierbarkeit* einer Fläche ersonnen. In [HB96] ist ein Test auf Orientierbarkeit beschrieben, das Verfahren von Möbius (als Beispiel für eine nicht orientierbare Fläche könnte u.a. das wohlbekannte Möbiusband dienen).

Nach der nun folgenden Beschreibung einer Technik, die zur Beschreibung von Körpern Kanten und Flächen kombiniert und durch topologische Informationen anreichert, werden sodann zum Schluss des Abschnitts über die geometrische Modellierung noch Techniken basierend auf Volumina vorgestellt, die dann auch Operationen auf Körpern, wie das Zerteilen und Vereinigen, zulassen.

4.1.1.3 „Boundary-Representation“

Die bisher vorgestellten Techniken zu Modellierung von Objekten, namentlich das Kanten- und Oberflächenmodell, verfolgen einen ausschließlich geometrischen Ansatz: Beschrieben wird die räumliche Lage von Objekten, Flächen, Kanten und Punkten. Das Boundary-Representation-Modell („Begrenzungsmodell“) verbindet bei der Darstellung von Objekten solche geometrischen Informationen mit einer hierarchischen Einordnung dieser in eine topologische Struktur: Für sämtliche in der Darstellung auftretenden geometrischen Elemente werden zusätzlich deren Beziehungen untereinander gespeichert. Die topologische Struktur gibt dann in einfacher Weise Auskunft auf Fragen, wie

- Welche Flächen gehören zu einem Objekt?
- Welche Kanten gehören zu einer Fläche?
- Zu welchem Objekt gehört eine Fläche?
- Zu welchem Objekt gehört eine Kante? (hier wäre zuerst zu ermitteln, zu welcher Fläche die Kante und zu welchem Objekt die Fläche gehört)
- Welche Flächen stoßen aneinander (haben also eine gemeinsame Kante)?

Für einen einfachen, in Abbildung 4.16 dargestellten Quader Q sind in Abbildung 4.17 sämtliche Elemente in ihrer hierarchischen Struktur angegeben. Die oben gestellten Fragen ließen sich durch einfaches Verfolgen der Kanten beantworten.

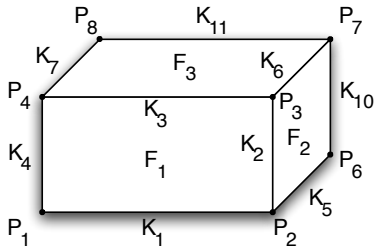


Abbildung 4.16: Quader mit eingezeichneten Seiten- und Kantenbezeichnungen

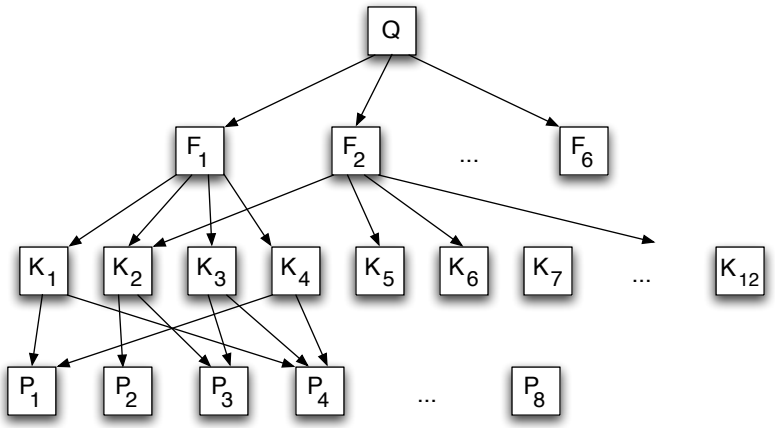


Abbildung 4.17: Hierarchische Darstellung des Quaders aus Abbildung 4.16 mit Hilfe der „Boundary-Representation“

4.1.1.4 Volumenbasierte Modelle

Bei den bisher vorgestellten Methoden basiert die Darstellung komplexer, dreidimensionaler Körper auf Kurven und Flächen, also volumenlosen Gebilden. Diese Ansätze waren mit gewissen Nachteilen behaftet: Es war prinzipiell möglich, aus einfachen Kurven unmögliche dreidimensionale Gebilde aufzubauen, und außerdem war bei Operationen auf Körpern, wie z.B. dem Teilen, nicht klar, welche genauen Ausprägungen die Ergebnisse solcher Operationen hatten.

Im Folgenden werden daher zwei Techniken präsentiert, bei denen komplexe Körper aus einzelnen Volumina aufgebaut werden: Aus unterschiedlichen parametrisierbaren Primitivkörpern wie Würfeln oder Kugeln bei der CSG-Technik bzw. aus einheitlichen, den Raum zerlegenden Elementarzellen bei der Zellenzerlegungsmethode.

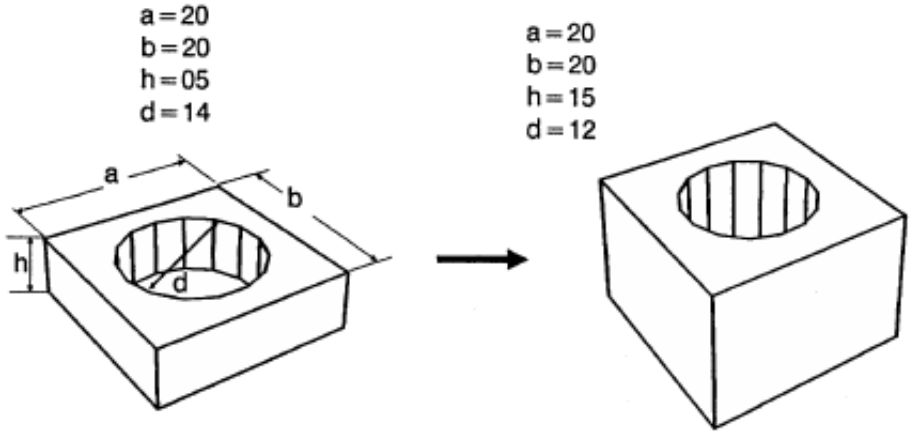


Abbildung 4.18: Beispiel für komplexere Grundform mit Parametern

Volumenmodelle und „Constructive Solid Geometry“ Grundlage für die volumenbasierte Modellierung bilden primitive, dreidimensionale Grundkörper und topologische Operationen auf diesen. Die Grundkörper selbst sind parametrisierbar, d.h. es lassen sich ausgehend von einigen feststehenden Grundformen (wie z.B. Würfel und Kugel, aber auch komplexeren Geometrien) durch Festlegen von Parameterwerten so genannte *Varianten* bilden: Für einen Würfel kann beispielsweise die Kantenlänge und für eine Kugel der Radius angegeben werden. Gegebenenfalls muss für die Parameterwerte eine Konsistenzprüfung stattfinden: Für den Körper in Abbildung 4.18 geben a und b die Breite bzw. Tiefe des äußeren Quaders und d den Durchmesser der Bohrung in der Mitte des Körpers an. Die natürliche Einschränkung für den inneren Radius d lautet hier: $d < \min\{a, b\}$.

Komplexere Geometrien können durch die Anwendung zweistelliger Verknüpfungsoperationen erzeugt werden. Beginnend mit den zur Verfügung stehenden Grundkörpern werden aus diesen zunächst neue Körper kombiniert, die dann ihrerseits wiederum entweder mit weiteren Grundkörpern oder anderen zusammengesetzten Objekten verknüpft werden können. Offensichtlich ist das Ergebnis bei der Verknüpfung zweier Körper wieder ein korrekter Körper.

Unter Verwendung des Volumenmodells beschreibt *CSG* (**C**onstructive **S**olid **G**eometry) eine Methode zur Darstellung dreidimensionaler Geometrien aus parametrierbaren Grundkörpern, Transformationen zur Positionierung von Grundkörpern im Raum und Verknüpfungen von Körpern. An Grundkörpern stehen die in Tabelle 4.1 aufgeführten Formen zur Verfügung, z.T. ergänzt um Profilkörper (eine Profilfläche wird entlang einer Geraden verschoben) oder Rotationskörper (eine Konturlinie wird um eine Achse gedreht). Aus der Tabelle können auch die möglichen Parameter entnommen werden. Schnitt ($\cap$), Vereinigung ($\cup$) und Subtraktion ($\setminus$) bilden die Menge der Verknüpfungsoperatoren; Translation (Verschiebung) und Rotation (Drehung) bezüglich des zugrunde liegenden Koordinatensystems sind die möglichen Transformationen.

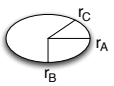
Grundkörper	Parameter	Skizze
Quader	Länge, Breite, Höhe	
regelmäßiges Prisma	Radius, Seitenzahl, Höhe	
Zylinder	Radius, Höhe	
Kegelstumpf	Radius unten, Radius oben, Höhe	
Ellipsoid	Radius A, Radius B, Radius C	

Tabelle 4.1: Grundkörper des CSG-Modells

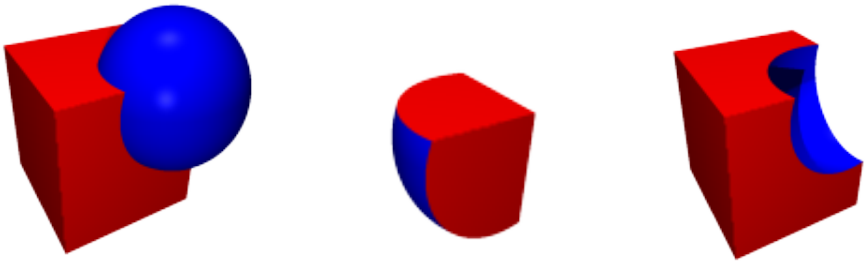


Abbildung 4.19: Vereinigung, Schnitt und Subtraktion eines Würfels und einer Kugel

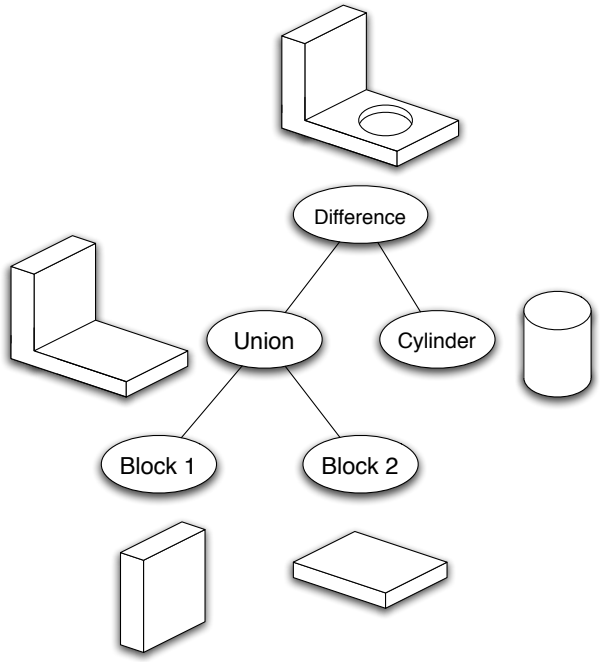


Abbildung 4.20: CSG-Baum mit resultierendem Körper

Bestandteil des CSG-Modells ist ferner das Konzept der Repräsentation zusammengesetzter Körper als Binärbaum: Transformations- und Verknüpfungsoperationen bilden die inneren Knoten, in den Blättern stehen Grundkörper und Transformationsparameter (Drehwinkel und Verschiebungsvektoren). Abbildung 4.20 zeigt einen solchen CSG-Baum und den sich ergebenden Körper (Transformationen wurden der Übersichtlichkeit halber weggelassen).

Eine Alternative zur Beschreibung durch Bäume ist durch die Verwendung der folgenden Grammatik gegeben:

```
<Operator>            ::= ∪ | ∩ | \
<Transformation>      ::= Translation | Rotation
<Transformationswert> ::= Vektor
<Objekt>              ::= <Primitiv> |
                          <Objekt><Operator><Objekt> |
                          <Objekt><Transformation><Wert>
```

Offensichtlich stimmt der Wortschatz dieser Grammatik mit der Menge der im CSG-Modell repräsentierbaren Körper überein.

In der bisher beschriebenen Form weisen diese Darstellungsmethoden, sei es als Baum oder als Wort einer Grammatik, einen gewissen Nachteil auf: Das gleiche Objekt kann aufgrund der Kommutativität und Assoziativität der verwendeten Operatoren durch unterschied-

liche Bäume (bzw. Worte) beschrieben werden. Durch die Einführung einer Normalform kann dies vermieden werden: Jeder Körper lässt sich durch einen Baum darstellen, in dem alle Schnitt- und Subtraktionsknoten einen linken Teilbaum ohne Vereinigungsoperator und einen rechten Teilbaum, der nur aus einem Primitivkörper besteht, besitzen. Durch diese Einschränkungen „wandern“ die Vereinigungsoperatoren in Richtung der Wurzel und Schnitt- sowie Subtraktionsoperatoren in Richtung der Blätter des Baumes. Die so gebildeten Bäume sind dann zwar eindeutig, doch kann die Gesamthöhe stark anwachsen und es wird gegebenenfalls das Hinzufügen von Primitiven erzwungen, die zum endgültigen Bild nicht beitragen.

Im Vergleich zu den anderen vorgestellten Methoden zur Körpermodellierung hält sich der Eingabeaufwand bei der CSG-Technik in Grenzen. Zudem können – wie bereits eingangs erwähnt – keine inkonsistenten Körper auftreten und Volumeneigenschaften wie Schwerpunkt, Masse oder auch Trägheit lassen sich verhältnismäßig einfach bestimmen, da die Auswertung der für solche Berechnungen notwendigen mehrfachen Integrale

$$\int_O = f(x,y,z)\ d(x,y,z)$$

aufgrund der Identitäten

$$\int_{A\cup B} f\ d(x,y,z) = \int_A f\ d(x,y,z) + \int_B f\ d(x,y,z) - \int_{A\cap B} f\ d(x,y,z)$$

und

$$\int_{A\setminus B} f\ d(x,y,z) = \int_A f\ d(x,y,z) - \int_{A\cap B} f\ d(x,y,z)$$

bei bekanntem Integral über den Schnittkörper sehr einfach ist [Sar82]. Dem steht ein hoher Implementierungs- und Darstellungsaufwand und eine Beschränkung auf einfache Oberflächen gegenüber: Freiformflächen, wie sie häufig zur Darstellung komplizierter Relief-Strukturen benötigt würden, lassen sich in dieses Modell nur sehr schwer integrieren. Die Herausforderung dieses Ansatzes besteht also in einer möglichst genauen *Approximation* der zu modellierenden Körpergeometrie mittels möglichst weniger, einfacher Grundkörper. Es ist hier stets abzuwägen zwischen dem benötigten Detailgrad einerseits und dem Umfangs des Modells andererseits.

Es soll noch erwähnt werden, dass CSG-Modelle durch die konstruktive Art der Modellierung leicht interaktiv zu erstellen und editieren sind. So kommt eine (gegenüber unserer Darstellung stark erweiterte) CSG-Technik z.B. auch bei dem Editor des populären 3D-Computerspiels „Unreal Tournament“ zum Einsatz: Abbildung 4.21 zeigt neben dem Ausschnitt einer fertigen Spielumgebung (Level) auch Teile der Werkzeugpalette des Editors mit Schaltflächen zur Auswahl von Volumen-Operationen und Grundkörpern (hier ist u.a. auch ein Treppen-Grundkörper verfügbar).

Zellzerlegungsverfahren Im Gegensatz zum CSG-Modell werden komplexe Objekte bei Zellteilungsverfahren nicht aus Varianten unterschiedlicher Grundkörper aufgebaut,

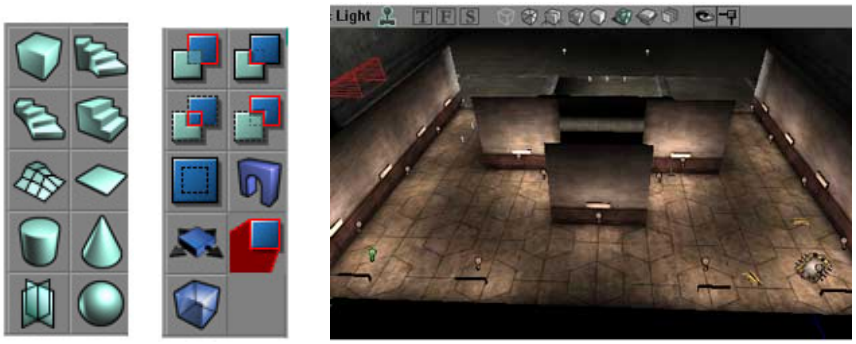


Abbildung 4.21: Ausschnitt aus einer Unreal-Spielumgebung und Teile der Werkzeugpalette des Level-Editors

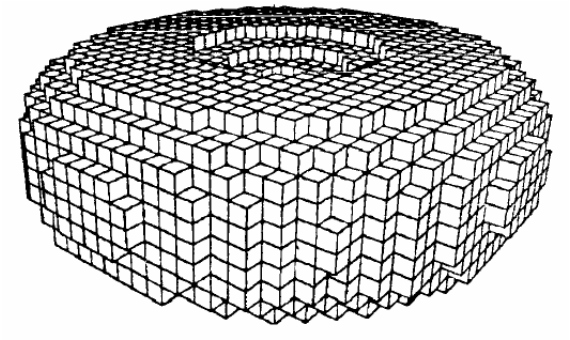


Abbildung 4.22: Darstellung eines Torus mit der Zellzerlegungsmethode

sondern aus disjunkten, einheitlichen *Elementarzellen*, die in gleichmäßiger Anordnung den Raum ausfüllen. Zum Einsatz kommen zumeist einfache geometrische Objekte wie Tetraeder oder Quader. Ein Körper wird dann durch die Menge der Zellen dargestellt, die in seinem Inneren liegen bzw. von seiner Oberfläche geschnitten werden. Dabei kann der zu modellierende Körper umso genauer approximiert werden, je feiner die Maschenweite gewählt wird. Theoretisch ließe dieses Verfahren also eine beliebig genaue Darstellung zu, doch wächst auch der benötigte Speicher mit fortschreitender Verfeinerung an. Abbildung 4.22 zeigt einen Torus, der aus Würfeln zusammengesetzt ist.

Um dem Nachteil des enormen Speicherbedarfs zu begegnen, kann das Zerlegungsverfahren auch rekursiv angewendet werden. Wir beschreiben hier das *Oktalbaumschema* (engl: Octree-Schema); das zweidimensionale Gegenstück ist das Quadtree-Schema.

Zunächst wird der darzustellende Körper in einen ihn umschließenden Würfel eingefasst. Dieser wird nun in kleinere Würfel halber Kantenlänge (insgesamt also acht Stück) unterteilt, für die ermittelt wird, ob sie von dem Körper vollständig belegt, teilweise belegt oder gänzlich frei gelassen werden. Für die teilweise belegten Zellen wird dieses Verfahren so lange rekursiv angewandt, bis eine vorher festzulegende Mindestgröße der Zellen erreicht ist. Als Datenstruktur kommt ein Baum mit achtfacher Verästelung (vierfach im zweidimensionalen Fall) zum Einsatz – daher auch der Name dieses Verfahrens. Abbildung 4.23 zeigt

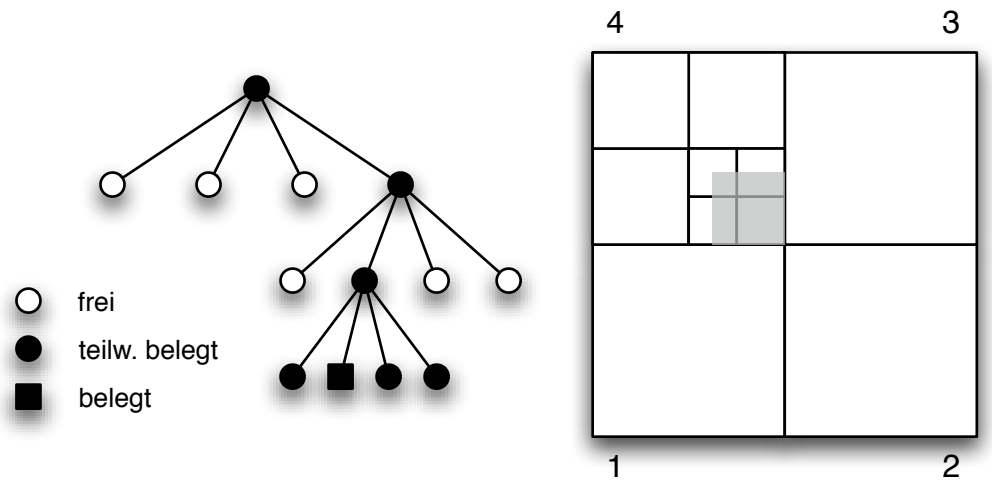


Abbildung 4.23: Quadtree der Tiefe 3 (Beispiel aus [HB96])

einen Quadtree der Tiefe 3. Mit jedem Rekursionsschritt verdoppelt sich die Genauigkeit, so dass in einem Baum der Höhe h Details in der Größenordnung von 2^{-h} des anfänglichen Würfels modelliert werden können. Beispielsweise ergibt sich für ein Anfangsvolumen von 1 m^3 mit $h = 10$ eine Auflösung von etwa einem Millimeter.

4.1.1.5 Modellierung zusätzlicher Eigenschaften

Die vorhergehenden Abschnitte haben sich ausschließlich mit geometrischen Aspekten der Modellierung beschäftigt. Oft bedarf es jedoch auch des Wissens um zusätzliche technologische Eigenschaften, auch *Strukturdaten* genannt, der zu modellierenden Objekte, um die gestellten Aufgaben lösen zu können.

Die Frage, welche zusätzlichen Eigenschaften im Rahmen der Modellierung erfasst werden müssen, kann nicht allgemein beantwortet werden: Zu stark hängt die Auswahl von der gestellten Aufgabe ab. Strukturdaten für zu manipulierende Objekte könnten beispielsweise Angaben über

- Greifpunkte
- Verbindungspunkte zur Montage
- Ablagepunkte bei Paletten
- Füllmengen bei Paletten
- Stellungen bezüglich eines Referenzobjektes
- Materialeigenschaften wie Festigkeit, Steifigkeit, Elastizität und Dichte

- Oberflächenbeschaffenheit wie Reibungs- und Reflexionseigenschaften oder Farbe

und mehr umfassen. Darüber hinaus können auch technologische Charakteristika des Roboters, wie etwa

- Tragkraft
- Positionierungsgenauigkeit
- Geschwindigkeits- und Beschleunigungswerte
- Greifkraft
- oder Materialeigenschaften der Greiferoberfläche (Reibungskoeffizient – interessant in Verbindung mit der Oberflächenbeschaffenheit des Werkstücks)

in das Modell aufgenommen werden. Abbildung 4.24 zeigt eine Auswahl zusätzlicher Eigenschaften, die für die Beschreibung eines Tisches von Interesse sein könnten.

Aufgrund der bereits erwähnten Abhängigkeiten der notwendigen Strukturdaten von den möglichen Aufgaben ist es ratsam, sich zunächst ein Bild über die Art der durchzuführenden Aufgaben zu machen, noch bevor mit der Modellierung zusätzlicher Eigenschaften begonnen wird. Beispielsweise gilt es bereits beim Festlegen der Greifpunkte eines Schraubenziehers den späteren Verwendungszweck dieses Objektes zu berücksichtigen: Soll der Schraubenzieher aktiv als Werkzeug in einem Montageprozess eingesetzt werden, sind ganz andere Greifpunkte erforderlich verglichen mit einem Szenario, in dem der Schraubenzieher lediglich als zu transportierendes Objekt auftritt.

Neben der Entscheidung, welche Eigenschaften überhaupt modelliert werden sollen, ist auch eine mögliche Integration dieser zusätzlichen Strukturdaten in das geometrische Modell zu berücksichtigen. Materialeigenschaften wie die oben genannten lassen sich häufig einem Objekt als Ganzes zuordnen, wohingegen es durchaus vorkommen kann, dass Unter- und Oberseite desselben Objektes ganz unterschiedliche Reibungseigenschaften aufweisen (man denke beispielsweise an ein Fußbodenelement, das an der Oberseite der besseren Standfestigkeit wegen aufgeraut ist).

4.1.2 Szenenmodell

Die nachfolgenden Modelle sollen die genaue Semantik der Objekte untereinander beschreiben. Die geometrischen Modelle hingegen beschreiben die exakten Abmaße und Lagen der Objekte, die folgenden Modelle stellen die Beziehungen zwischen den vorhandenen Objekten dar und abstrahieren von den genauen geometrischen Anordnungen im Raum. Es wird die Frage beantwortet, „Wie hängen die Objekte zusammen?“ Wenn man die Umwelt manipulieren will, dann reicht nur geometrische Modellierung nicht aus. Man muss über genügend Informationen über die Beziehung und Zusammenhänge der einzelnen Objekte im Raum verfügen.

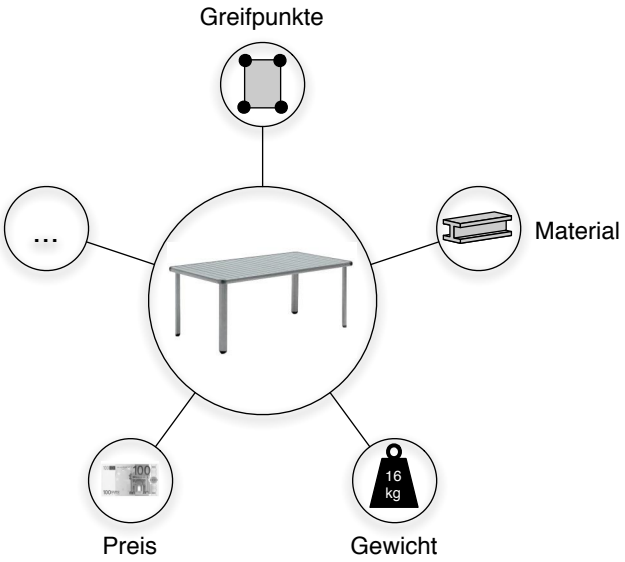


Abbildung 4.24: Eine Auswahl zusätzlicher Eigenschaften für einen Tisch

4.1.2.1 „Entity-Relationship-Modell“

Wie schon aus dem Modellnamen hervorgeht, besteht dieses Modell aus „Entities“ (Entitäten, Objekte) und aus „Relationships“ (Beziehungen, zwischen den Objekten). Um diese Beziehungen möglichst real zu beschreiben, werden diesen noch „Kardinalitäten“ (mögliche Anzahl beteiligter Objekte) zugeordnet. Die Art der Beziehung und den Zweck beschreiben „Rollen“. Sind z.B. zwei Objekte gegeben, Kunde und Artikel, dann kann eine Beziehung zwischen Kunde und Artikel bestehen, diese wird mit „kauft“ bezeichnet, die Rolle des Kunden könnte „Besteller“, die der Artikel „bestellte Artikel“ lauten. Das ER-Modell wird meist graphisch dargestellt, man nennt dies dann ein „ER-Diagramm“, dabei sind verschiedene Darstellungsformen und Notationen gebräuchlich, die hier vorgestellte ist nur eine von Vielen. Es ist Vorsicht geboten, da die unterschiedlichen Darstellungsformen teilweise genau umgekehrte Notationen besitzen (z.B. in Bezug auf die Kardinalitäten). In Tabelle 4.2 sind den einzelnen ER-Elementen graphische Symbole zugeordnet, in 4.3 sind weitere Beschreibungskonstrukte zu sehen. Diese dienen dazu zusätzliche Informationen wie Attribute den Objekten zuzuweisen. Weiter ist es in ER-Modellen möglich eine Generalisierungshierarchie zu erstellen, welche von Objekten abstrahiert und Objektklassen mit Unterobjekten bildet (siehe Bsp. Generalisierungshierarchie, Greifer und Werkzeug sind beides Effektoren, d.h. man kann beide zu der Klasse Effektor zusammenfassen, bzw. aus dieser Klasse ableiten.

In Abbildung 4.25 ist ein Beispiel eines ER-Diagramms zu sehen, hier wird die Umwelt und die „darin“ enthaltenen Objekte, Peripherie, usw. und deren Zusammenhang modelliert. ER-Modelle werden in vielen Bereichen der Informatik eingesetzt, hauptsächlich im Zusammenhang mit Anwendungsentwicklung und Datenbankdesign. Ein großer Nachteil beim

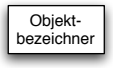
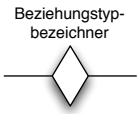
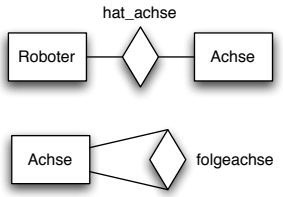
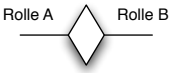
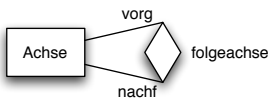
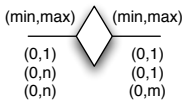
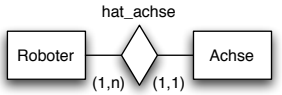
Modellierkonstrukt	Graphisches Symbol	Beispiel
Objekttyp (entity set)		
Beziehungstyp (zweiseitig oder rekursiv)		
Rollen		
Kardinalität		

Tabelle 4.2: Grundelemente des ER-Modells

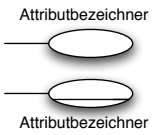
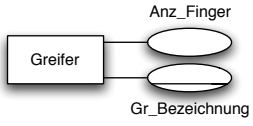
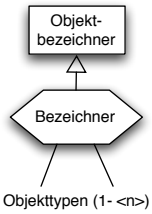
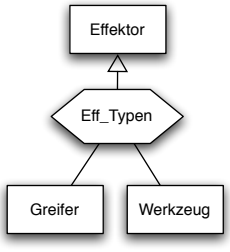
Modellierkonstrukt	Graphisches Symbol	Beispiel
Attribute: beschreibende und identifizierende		
Generalisierungs-hierarchie		

Tabelle 4.3: Weitere Elemente des ER-Modells

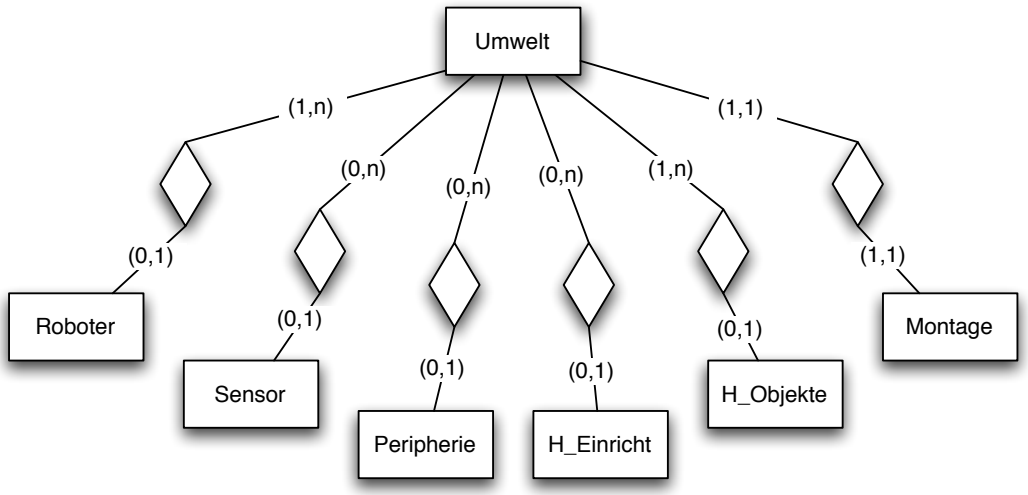


Abbildung 4.25: Beispiel für ein ER-Diagramm

Einsatz in der Robotik zur Szenenmodellierung besteht in der nicht trivialen Implementierung durch objektorientierte Programmiersprachen, dieser Nachteil besteht deswegen nur in der Robotik da in anderen Bereichen auf eine Implementierung verzichtet wird.

4.1.2.2 Semantische Netze

Semantische Netze wurden in den früheren 1960ern von dem Sprachwissenschaftler *Ross Quillian* als Repräsentationsform semantischen Wissens vorgeschlagen. Ein semantisches Netz ist ein formales Modell von Begriffen und ihren Beziehungen (=Relationen). Es wird in der Informatik und künstlichen Intelligenz zur Wissensrepräsentation genutzt. Meist wird semantisches Netz durch einen verallgemeinerten Graphen realisiert.
Diese Darstellungsform besteht wieder aus Objekten und Beziehungen, diese werden in einem Semantischen Netz als Knoten (= Objekte) und Beziehungen (= Kanten) dargestellt. Das Modell bietet keine weiteren Elemente, die Kanten werden benannt, um die Beziehung genauer zu spezifizieren. Es sind nur zweistellige, gerichtete Beziehungen zugelassen, werden mehrstellige Beziehungen benötigt müssen diese über eigens dazu eingeführte Objekte realisiert werden. Welche Beziehungen erlaubt sind, wird in unterschiedlichen Modellen sehr unterschiedlich festgelegt. Ähnlich verhält es sich mit Attributen, da kein eigenes Element für Attribute zur Verfügung steht, werden diese auch als Objekte dargestellt, genauer gesagt als „Wertobjekt“ (siehe Abbildung 4.26, „Amtec Roboter“ hat Farbe „blau“). Ein beachtlicher Vorteil gegenüber dem vorher vorgestellten ER-Modell besteht darin, dass sich Semantische Netze, sehr einfach und schnell in objektorientierten Programmiersprachen implementieren lassen. Von diesem Modell gibt es wieder zahlreiche Varianten und Ausbildungen auf die hier nicht näher eingegangen werden soll.

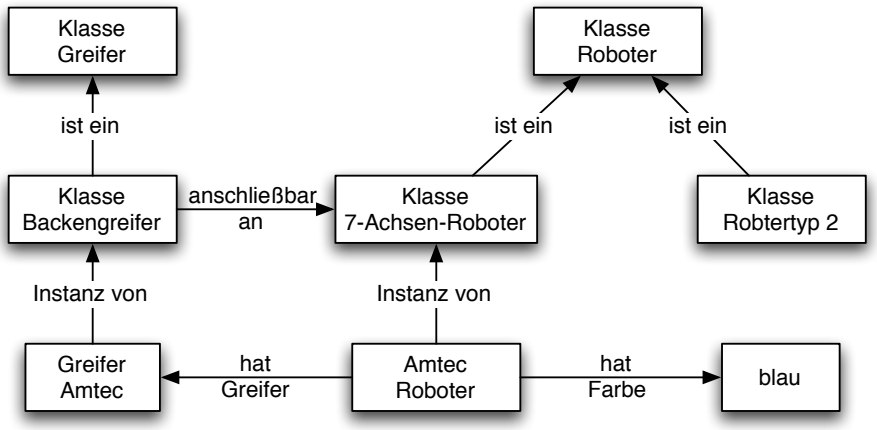


Abbildung 4.26: Semantisches Netz eines Roboters

4.1.2.3 Frame-Modell nach Minsky

Eine weitere Darstellungsform für Wissen wurde von Minsky eingeführt. Das Frame-Modell nach Minsky besteht grundlegend aus „Frames“, „Slots“ und Beziehungen zwischen Frames. Ein Frame beschreibt eine Art Schablone (nicht zu verwechseln mit Koordinatenframes), welche die Objekte darstellt. Diese können hierarchisch in Beziehung zueinander stehen. Dabei bilden die Kanten eine Instanziierung der Objekte von anderen Frames. Durch Frames werden die Eigenschaften von Objekten, ihre Einordnung in eine Hierarchie von Objektklassen und ihre Beziehungen zu anderen Objekten dargestellt. Die einzelnen Frames wiederum enthalten mehrere Slots, welche die Eigenschaften bzw. Attribute der Objekte bereitstellen. Außerdem enthalten Slots Verweise auf andere Frames, damit lassen sich Assoziationen und Vererbungen zu anderen Frames definieren. Dieses Modell benutzt eine ähnliche Semantik wie UML (Unified Modeling Language). Da auch dieses Modell offensichtlich ein objektorientiertes Beschreibungsschema verwendet, ist auch hier der Implementierungsaufwand bei der OOP (objektorientierte Programmierung) gering.

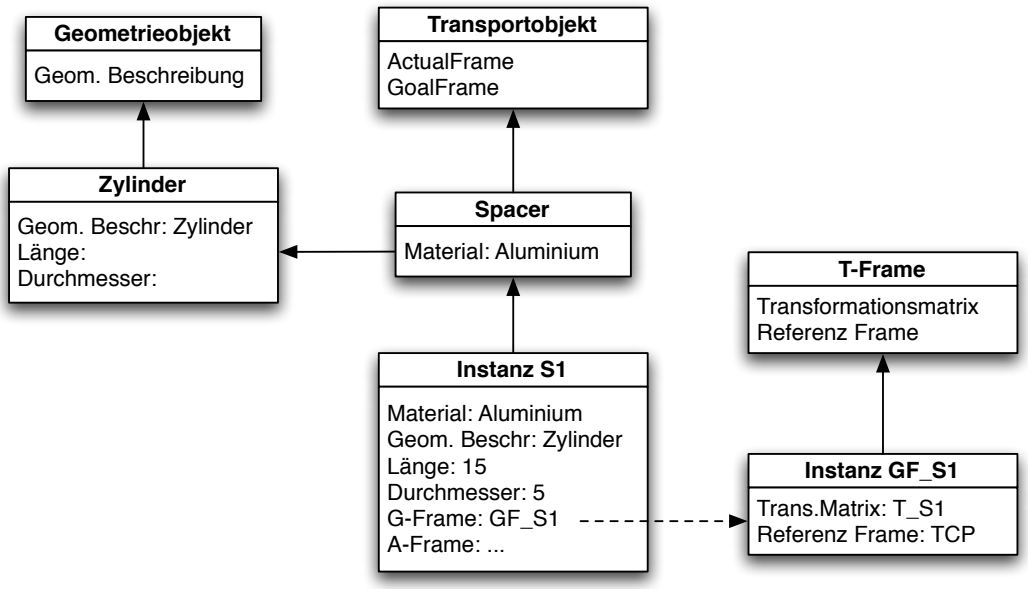


Abbildung 4.27: Beispiel eines Frame-Modells nach Minsky

4.2 Aufgabenmodell

Nachdem nun die Objektmodellierung genauer erläutert wurde, wenden wir uns nun der noch benötigten Modellierung der vom Roboter auszuführenden Aufgabe zu (siehe Abb. 4.1). In der Aufgabenmodellierung werden dabei Modelle für die gestellte Aufgabe sowie für deren Gliederung in Teilziele benötigt. Abbildung 4.28 gibt noch mal eine Übersicht über alle benötigten Modelle der Aufgabenorientierten Programmierung. Wir werden uns in den nun folgenden Abschnitten einen Überblick über die Struktur des Aufgabenmodells, der Notwendigkeit von unterschiedlichen Bedingungen und abschließend mit der Validierung der Modelle vor der tatsächlichen Ausführung beschäftigen.

Der Mensch gibt seine Befehle durch eine symbolische Eingabe an das System und der Roboter steuert dementsprechend seine Aktorik, um den Befehl auszuführen. Zwischen der symbolischen Eingabe und der Aktorsteuerung werden ein Aufgabenmodell und ein Umweltmodell benötigt. Ein Planer ist dafür zuständig, dass diese vier Komponenten zusammenarbeiten. Die symbolische Eingabe wird in das Aufgabenmodell umgesetzt und anhand der Informationen des Umweltmodells werden konkrete Robotersteuerungen erzeugt (siehe Abb. 4.29).

4.2.1 Struktur des Aufgabenmodells

Die allgemeine Vorstellung ist, dass Ziele auf verschiedenen Detaillierungsebenen beschrieben werden können und dementsprechend zu einer hierarchischen Gliederung in Teil- bzw.

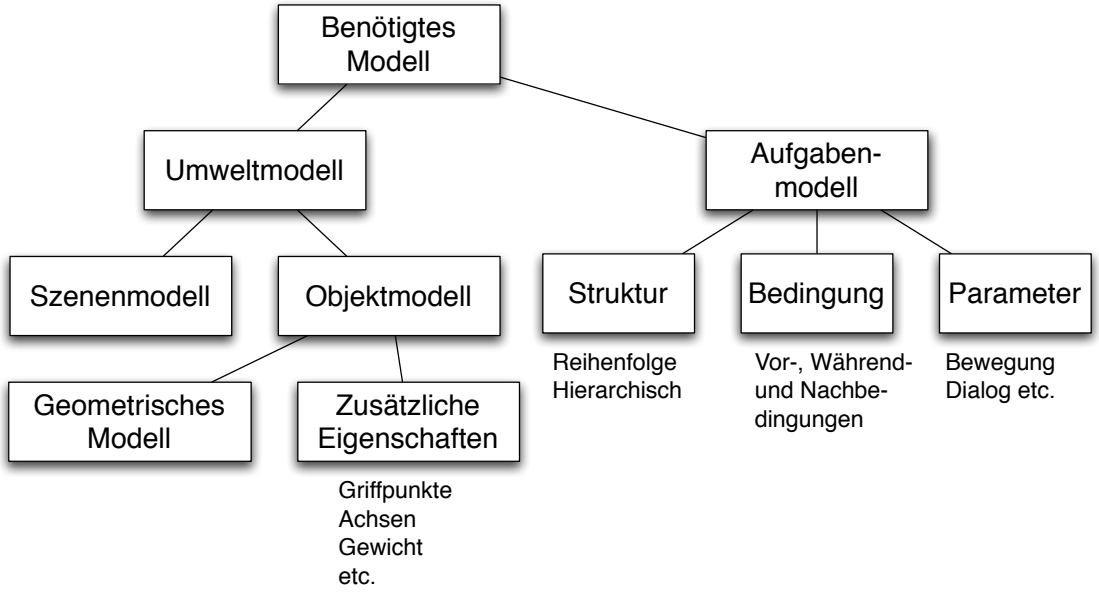


Abbildung 4.28: Klassifikation der benötigten Modelle für Umwelt und Aufgabe

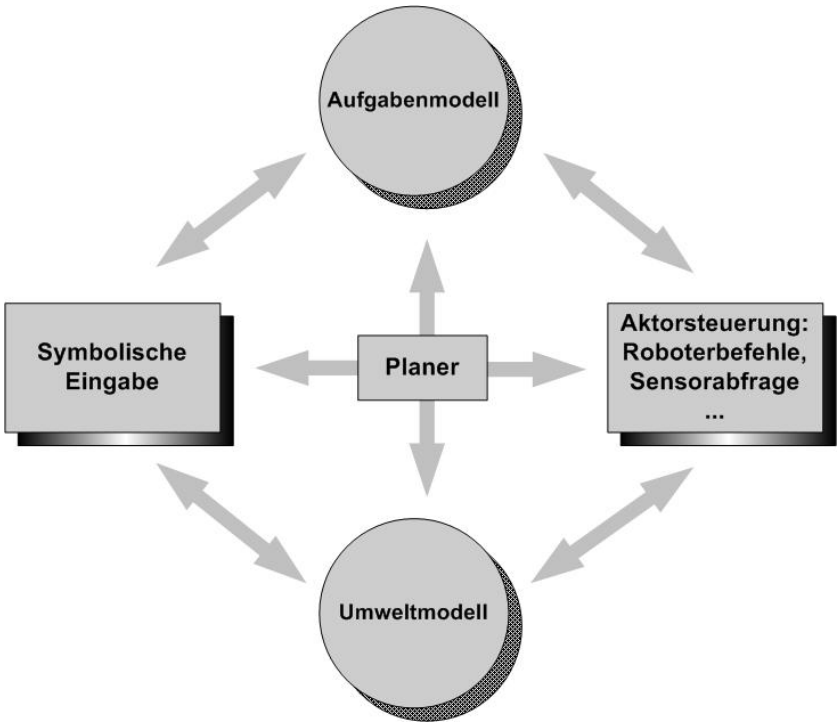


Abbildung 4.29: Einordnung des Aufgabenmodells

Unterziele führen. In der Aufgabenmodellierung ist es wichtig zu bestimmen, auf welcher Abstraktionsebene die Aufgaben vorgegeben werden.

Der Benutzer hat bei der Aufgabenorientierten Programmierung die Möglichkeit seine Aufgabe auf unterschiedlichen Abstraktionsebenen zu stellen. Als Beispiel betrachten wir kurz die Aufgabe, dem Benutzer eine Flasche Bier aus dem Kühlschrank zu bringen. Diese kann man auf der einen Seite sehr abstrakt formulieren: „Hol‘ mir ‘ne Flasche Bier!“ Leicht nachzuvollziehen ist, dass der Roboter nun, bis zur tatsächlichen Ausführung, eine Reihe von Zerlegungen dieser Aufgabenstellung vornehmen muss. Diese Arbeit kann in gewisser Weise der Benutzer dem Roboter abnehmen, indem er die Aufgabe ein Wenig detaillierter formuliert: „Fahre in die Küche, öffne den Kühlschrank, greife eine Flasche Bier, fahre zurück und reiche mir die Flasche.“

Nach der symbolischen Eingabe der Aufgabenstellung erfolgt also eine schrittweise Zerlegung von dieser, bis ausführbare Blöcke entstehen (sog. Elementaroperationen, siehe auch Abschnitt 4.2.1.2: Abstraktionsebenen).

Durch eine solche symbolische Abstraktion werden verschiedene Anforderungen notwendig:

- **Erweiterbarkeit**: Die Informationen bei dynamischen Modellen müssen jederzeit erweiterbar sein.
- **Wiederverwendbarkeit**: Einzelne Lösungsschritte sollen wiederholt ausführbar sein.
- **Erklärbarkeit**: Der Roboter muss dem Benutzer seine Handlung stets erklären können.

4.2.1.1 Vorranggraph

Bei der Verkettung von einzelnen elementaren Aufgaben zu komplexen Handlungen entstehen oft temporale und kausale Abhängigkeiten zwischen einzelnen Teilhandlungen. Bei der bereits genannten Aufgabe ein Bier zu holen ist es beispielsweise unumgänglich, dass der Kühlschrank geöffnet wird bevor ein Bier aus diesem gegriffen werden kann.

Eine gute Möglichkeit, eine solch notwendige Reihenfolge zu modellieren, ist der sog. *Vorranggraph*. Ein einfaches, anschauliches Beispiel zeigt Abb. 4.30. Man kann aus diesem folgende Abhängigkeiten ablesen:

vor „B“ muss „A“ ausgeführt sein
vor „C“ muss „A“ ausgeführt sein
vor „D“ muss „C“ ausgeführt sein
vor „E“ müssen „D“ und „B“ ausgeführt sein
keine Aussage zwischen „B“ und „C“ bzw. zwischen „B“ und „D“ .

Vorranggraph des Cranfield-Benchmarks Als etwas praktischere Anwendung eines Vorranggraphen wollen wir uns nun noch den *Cranfield-Benchmark* anschauen. Abbildung

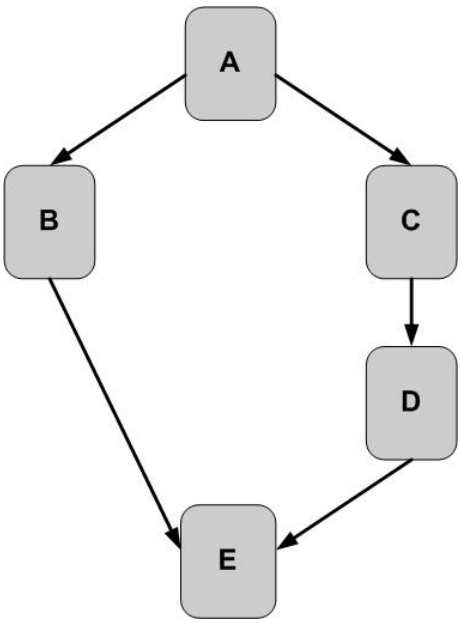


Abbildung 4.30: Beispiel eines Vorranggraphen

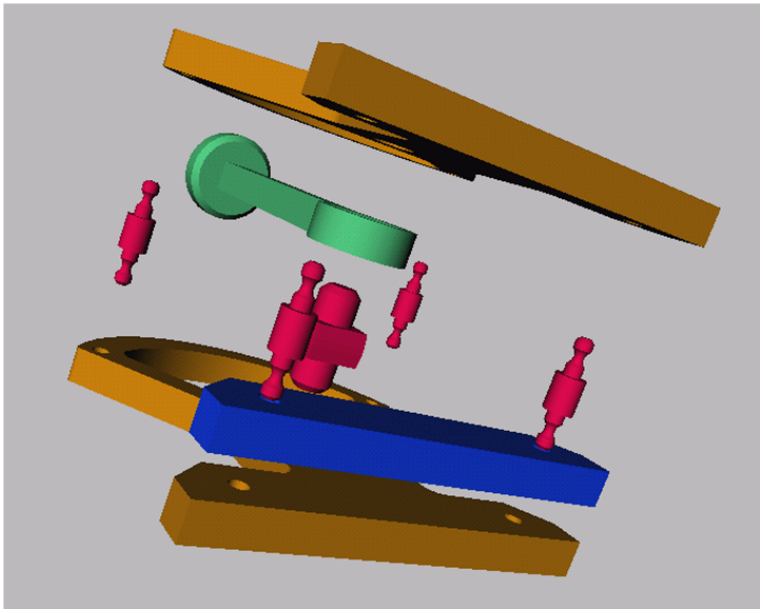


Abbildung 4.31: Der Cranfield-Benchmark

4.31 zeigt einige Montageschritte, wobei zu sehen ist, dass die Reihenfolge eine entscheidende Rolle spielen wird; bevor der Hebel platziert werden kann muss dieser einerseits gegriffen werden, wovor allerdings schon der Schaft eingesetzt sein muss.

Der komplette Vorranggraph zur Montage des Cranfield-Benchmarks zeigt Abb. 4.32, aus dem z.B. zu erkennen ist, dass zuerst immer eine Basisplatte auf Position 01 gelegt werden muss, bevor mit der weiter Montage der anderen Teile fortgefahren werden kann.

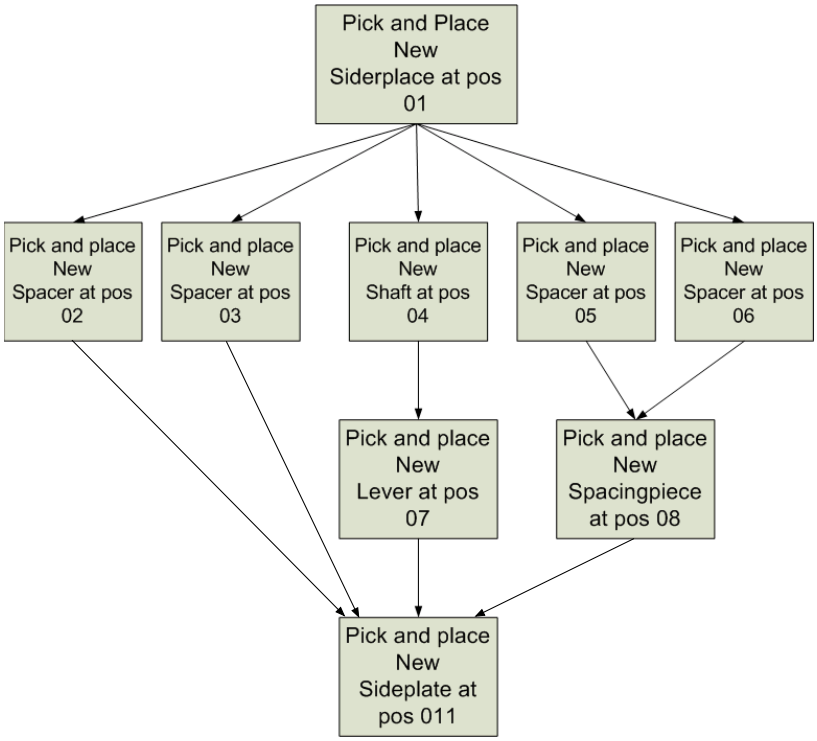


Abbildung 4.32: Vorranggraph zur Montage des Cranfield-Benchmarks

4.2.1.2 Abstraktionsebenen

Neben der strukturellen Reihenfolge bei der Aufgabenmodellierung, lässt sich diese auch in unterschiedliche hierarchische Abstraktionsebenen unterteilen. Wie schon beim genannten Beispiel des Bringens einer Flasche Bier zu erkennen war, kann eine Aufgabe sehr abstrakt bis sehr roboternah beschrieben werden. Eine solche semantische Abstraktion, wird im Allgemeinen in drei unterschiedliche Stufen unterteilt:

1. **Mission**
   Unter der Sprache oder Ebene *Mission* versteht man sehr komplexe Aufgabenstellungen, die vor der Ausführung zwangsläufig weiter unterteilt werden müssen. Diese Zerlegung in die nächste Abstraktionsebene ist im starken Maße von den jeweiligen Parametern abhängig und erfordert stets Informationen aus dem Umweltmodell

(Abb. 4.29).

2. **Task**
   Durch die Zerlegung einer *Mission* entstehen sog. *Tasks*, welche sowohl räumlich als auch funktional geschlossene Einheiten bilden. Dadurch entstehen Aufgaben welche von einer aktiven Einheit (z.B.: Roboter, Sensor, Fahrzeug) bearbeitet werden können, und aus denen sich wiederum umfangreiche Aufgaben zusammensetzen lassen.

3. **Action**
   Bei einer *Action* (auch *Elementaroperator* genannt) werden sehr einfache Roboterbefehle symbolisch beschrieben. Diese Elemente, die die unterste Stufe der Abstraktionshierarchie bilden, werden nach auftretender Umweltbeziehung unterteilt:

   - Bewegungen im freien Raum
   - Bewegungen im Objektnähe
   - Bewegungen mit Kontakt zu Objekten

   Aus dieser Einteilung lassen sich Informationen über gegebenenfalls benötigten Sensoreinsatz oder zu verwendende Bahntypen ermitteln. Bei Bewegungen im freien Raum existiert z.B. keine Gefahr einer Kollision, weswegen ein entsprechender Sensor nicht benötigt wird.

Modellierung von Aktionen und Tasks Zur Modellierung von den genannten Aktionen bzw. Tasks wird zuerst die Definition eines entsprechenden Operators (OP) benötigt:
OP = (N,O,A,Par,K) mit

- N = (eindeutiger) Name des Operators
- O = Liste der beteiligten Objekte
- A = Auswahlbedingung zur Bestimmung der Objekte
- Par = Liste der Parameter, z.B. Positionen, Kräfte, Beschleunigungen
- K = Körper des Operators

Die Unterscheidung, ob mit solch einem Operator eine *Action* oder ein *Task* modelliert wird, erfolgt nun durch die Wahl des Körpers K.

- Für eine Action ist dieser ein einfaches, ausführbares Programm, mit dem sich einfache Fähigkeiten wie *Move* realisieren lassen. Solch einen Operator nennt man auch *Elementaroperator* .
- Besteht der Körper K allerdings aus weiteren Operatoren ($K = (K_1, K_2, ..., K_n)$), wodurch sich komplexere Aktionen realisieren lassen, so spricht man von einem *Makro-Operator*.

Hierarchischer Aufbau Die beschriebene, hierarchische Unterteilung einer Aufgabe von einer *Mission* bis hin zu *Actions* zeigt beispielhaft Abbildung 4.33. Ganz oben ist die *Mission* "Tisch decken" als Beispiel einer höheren Beschreibung einer Aufgabe. Sie wird dann in kleinere Segmente (*Tasks*) unterteilt, aus welchen sich dann eine Reihe von Elementaroperatoren (EOs) ableiten lassen.

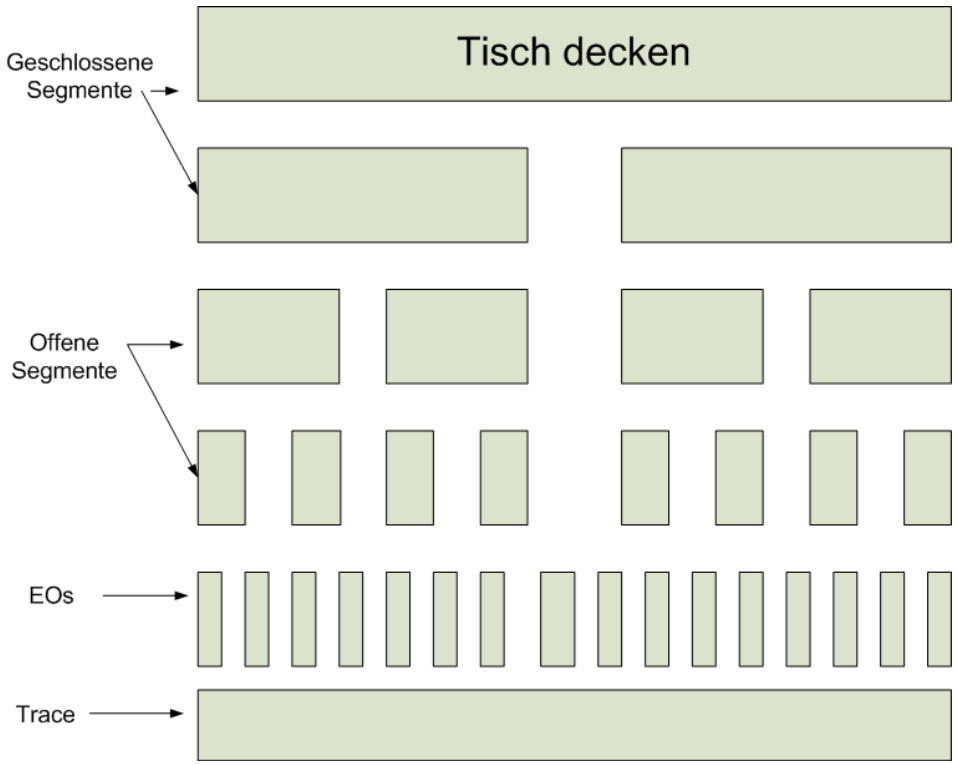


Abbildung 4.33: Hierarchischer Aufbau des Aufgabenmodells

4.2.2 Bedingungen

Ein wichtiger Aspekt der Aufgabenmodellierung stellt die Beschreibung von Bedingungen dar, welche eine reibungsfreie und erfolgreiche Ausführung der gestellten Aufgabe gewährleisten sollen. Diese Bedingungen lassen sich in *Vor- Während und Nachbedingungen* unterteilen, welche im Folgenden mit ihren jeweiligen Zwecken beschrieben werden sollen.

4.2.2.1 Vorbedingungen

Durch *Vorbedingungen* werden Anforderungen gestellt, die erfüllt sein müssen, bevor mit der Ausführung eines Operators (Elementar- oder Makrooperator) begonnen werden kann. Diese können entweder den Roboter selbst oder dessen Umwelt betreffen; einfache Beispiele sind: Zur Ausführung eines Pick-Up-Operators ist es z.B. notwendig zu überprüfen, dass:

1. der zu benutzende Greifer aktiv ist (betrifft Roboter)
2. der Greifer frei ist (betrifft Roboter)
3. der zu greifende Gegenstand frei zugänglich ist (betrifft Umwelt)

Vorbedingungen können auf allen bereits beschriebenen Abstraktionsebenen auftreten, und sind teilweise schon während der Planung überprüfbar. Zumeist bestehen sie aus einer Reihe von Einzelbedingungen, die konjunktiv miteinander verknüpft sind, also allesamt erfüllt sein müssen, bevor mit der Ausführung begonnen werden kann. Auf spezielle Anforderungen an ihre Form wird in Kapitel 6 noch eingegangen.

4.2.2.2 Währendbedingungen

Durch *Währendbedingungen* werden Einschränkungen beschrieben, die während der gesamten Ausführungsdauer eingehalten werden müssen, und haben deshalb den Zweck den Roboter während der Aktionsausführung in Echtzeit zu überwachen. Als triviales Beispiel kann hier genannt werden, dass während der Ausführung fast jeder Aktion ein ausreichender Ladestand der Batterie des Roboters vorausgesetzt werden muss. Allerdings ist es oftmals ebenso nötig bestimmte Kontaktkräfte oder Drehmomente zu beschränken, um beispielsweise eine Überbeanspruchung des Materials zu vermeiden.

4.2.2.3 Nachbedingungen

Im Gegensatz zu den Vor- und Währendbedingungen werden durch *Nachbedingungen* weniger Bedingungen für die Ausführung einer Aktion, als vielmehr an die sich daraus ergebenden *Effekte* beschrieben. Durch Überprüfung der Nachbedingungen im Anschluss an die Beendigung einer Handlung kann der Erfolg eben dieser Aktion festgestellt werden: Weicht der Umweltzustand (oder der Zustand des Roboters) von den beschriebenen Nachbedingungen ab, so ist während der Ausführung ein Fehler aufgetreten, der korrigiert werden muss. Bei einer Abfolge von mehreren Aktionen sollten die Nachbedingungen z.B. so gestaltet sein, dass durch sie die Einhaltung der Vorbedingungen der nachfolgenden Aktion zugesichert werden können. Genau wie die Vorbedingungen können die Nachbedingungen auf allen Abstraktionsebenen auftreten; sie sind allerdings erst während der Ausführung und noch nicht während der Planung überprüfbar.

4.2.2.4 Prädikatenlogik

Bisher wurden die drei für die aufgabenorientierte Programmierung wesentlichen Bedingungstypen incl. Beispiele vorgestellt. Die Formulierung dieser beispielhaften Bedingungen erfolgte dabei unter Verwendung normaler Umgangssprache. Für die automatische Auswertung von Bedingungen ist es jedoch erforderlich eine formalere Beschreibung zu finden, die auch von Rechenmaschinen verstanden und verarbeitet werden kann. Zu diesem Zwecke werden zunächst Syntax und Semantik der *Prädikatenlogik erster Ordnung* (PL1) eingeführt, woraufhin sich aber unmittelbar Betrachtungen anschließen werden, die aufzeigen, dass für die Formulierung von Bedingungen, wie sie im Bereich der aufgabenorientierten Programmierung anzutreffen sind, bereits eine Teilmenge der PL1 ausreichend ist.

Mit der als $\mathbb{V}$ bezeichneten Menge der *Variablen* und der Menge F der *Funktionssymbole* (auf die Bedeutung, d.h. die Semantik, dieser Mengen wird später eingegangen werden) wird die Menge $\mathbb{T}$ der Terme induktiv definiert:

- $\mathbb{V} \subseteq \mathbb{T}$
- Mit $f \in F$ und $t_1, \ldots, t_n \in \mathbb{T}$ ist auch $f(t_1, \ldots, t_n) \in \mathbb{T}$

In dieser Definition gibt n die *Stelligkeit* des Funktionssymbols f an, wobei auch $n = 0$ gelten kann: f heißt in diesem Fall *Konstante*. Zusätzlich benötigt man die Menge P der *Prädikatsymbole*, denen wie bei den Funktionen eine Stelligkeit zugewiesen wird; ein 0-stelliges Prädikat heißt *aussagenlogisches Atom*. Wiederum induktiv kann nun die Menge $\mathbb{F}$ der prädikatenlogischen *Formeln* (genauer: die Menge $\mathbb{F}_\Sigma$ der prädikatenlogischen Formeln über der *Signatur* $\Sigma = (F, P)$) definiert werden:

1. $\{\text{true}, \text{false}\} \subseteq \mathbb{F}$
2. $t = s \in \mathbb{F}$ für zwei Terme $t, s \in \mathbb{T}$
3. $p(t_1, \ldots, t_n) \in \mathbb{F}$ für Terme $t_1, \ldots, t_n \in \mathbb{T}$ und ein n-stelliges Prädikat $p \in P$ (insb. sind aussagenlogische Atome gültige Formeln)
4. Mit $x \in \mathbb{V}$ und $A, B \in \mathbb{V}$ sind ebenfalls in $\mathbb{V}$:

$$\neg A,\ A \vee B,\ A \wedge B,\ A \rightarrow B,\ A \leftrightarrow B,\ \forall x A,\ \exists x A$$

Einige Beispiele sollen dies klarer machen:

Für $\mathbb{V} = \{x, y, z\}$, $F = \{a, f, g\}$ und $P = \{Q, p, q\}$, wobei a eine Konstante, f eine zwei- und g eine dreistellige Funktion, Q ein aussagenlogisches Atom und p sowie q zwei- bzw. dreistellige Prädikate seien, gilt:

$$x,\ f(x,y),\ g(x,y,f(x,f(x,y))) \in \mathbb{T} \qquad y = a,\ Q,\ p(y,f(x,x)) \in \mathbb{F}$$
$$p(x,y),\ x = f(x,y),\ g(x = y, Q, x) \notin \mathbb{T} \qquad f(p(x,y)),\ q(p(\text{true}, x), x, a) \notin \mathbb{F}$$

4.2.3 Validierung der Modelle

Vor der tatsächlichen Ausführung in der realen Umwelt müssen die vorhandenen Aufgabenmodelle validiert werden, ob z.B. der Manipulator keine unerreichbaren Stellungen einzunehmen hat, die Bewegungsbahnen kollisionsfrei abgefahren werden können oder die auftretende Kräfte nicht die Grenzen des Manipulatorarms überschreiten. Die Gründe für solch eine Überprüfung der Modelle sind also im Wesentlichen eine erhöhte Sicherheit und niedrigere Kosten. Es gibt zwei unterschiedliche Validierungsmöglichkeiten:

1. **Simulation der Komponenten**
   Die Validierung erfolgt anhand gegebener Einschränkungen, wie Kollisionen oder Optimierungskriterien (Weg, Zeit, Energie, Kosten,...).
2. **Graphische Animation**
   Der Anwender überprüft visuell die erstellten Methoden.

4.2.3.1 Simulation der Komponenten

Durch eine Simulation der unterschiedlichen Roboterkomponenten soll deren Funktionalitäten überprüft werden. Demnach sind für unterschiedliche Komponenten sehr unterschiedliche Simulationsmethoden notwendig. Ein Manipulator kann so z.B. sowohl einer kinematischen als auch dynamischen Simulation unterzogen werden, ein Sensor hingegen eher auf seine Messungenauigkeit. Diese beiden Beispiele sollen nun stellvertretend für die Simulation von Roboterkomponenten etwas genauer beschrieben werden.

Simulation eines Manipulators Wie bereits angedeutet kann ein Manipulator sowohl in seiner Kinematik als auch in seiner Dynamik simuliert werden.

Bei der *Kinematiksimulation* soll die Ausführung der Aufgabe auf die Erreichbarkeit von Punkten, innere Kollisionen sowie auf alle auftretenden Konfigurationen hin überprüft werden, also wird, allgemein gesprochen, die Bewegung des Manipulators simuliert. Als Eingabewerte dienen dabei die zu erreichenden Zielstellungen, also meistens ein Positionsvektor (x, y, z) sowie ein Orientierungsvektor (α, β, γ). Durch Berechnung der inversen Kinematik (vergleiche Robotik 1) erhält man als Ausgabewerte einen Vektor der Gelenkwinkel $(\theta_1, \theta_2, ..., \theta_n)$, welche nötig sind um die Zielstellung einzunehmen. Aus diesem lassen sich dann z.B. Aussagen über Erreichbarkeit oder möglicherweise auftretenden inneren Kollisionen machen.

Im Gegensatz dazu berechnet die *Dynamiksimulation* eines Manipulators alle auftretenden Kräfte und Momente, welche während der Ausführung auf den Manipulator wirken. Dies ist z.B. durch die Gelenkwinkelgeschwindigkeiten, der Masseverteilung der einzelnen Gelenke sowie der Masse der gegriffenen Objekte möglich. Am Beispiel der Gelenkwinkelgeschwindigkeiten $\dot{\theta_1}, \dot{\theta_2}, ..., \dot{\theta_n}$, welche durch Ableiten des Vektors $(\theta_1, \theta_2, ..., \theta_n)$ formal berechnet werden können, sieht man, dass die Dynamik auf einer vollständig bekannten

Kinematik aufbaut. Mit den berechneten Kräften und Momenten lassen sich durch Vergleich mit den bekannten Maximalbeanspruchungen Überschreitungen der Nutzlastgrenze detektieren, oder auftretende Schwingungen bestimmen.

Simulation eines Sensors Durch die Simulation eines Sensors soll festgestellt werden, ob die reale Messung korrekt ausgeführt wurde. Die berechneten Werte einer *Sensorsimulation* dienen also als eine Vorgabe für die reale Messung. Durch die großen Unterschiede von unterschiedlichen Sensortypen, müssen die verwendeten Simulatoren für jeden Sensor neu parametrisiert werden. Als Ausgabe wird normalerweise kein exakter Wert geliefert, sondern ein Bereich in dem die reale Messung sich befinden sollte, oder der Ausgabewert wird mit einer Unschärfe versehen. Um die reale Messung auf eine korrekte Ausführung hin zu überprüfen, muss lediglich geprüft werden, ob der Wert in dem vom Simulator ermittelten Bereich liegt.

4.2.3.2 Graphische Animation

Oftmals ist es sinnvoll einer Simulation eine *Graphische Animation* vorzuziehen, gut zu erkennen an dem Beispiel einer Kollisionsüberprüfung. Bei einer Simulation würde das Problem bestehen, dass der Rechenaufwand extrem groß wäre, da jedes Objekt gegen jedes andere Objekt auf möglich Kollisionen überprüft werden muss. Dieses Problem wird behoben, indem der Mensch, wegen seiner unerreicht guten perzeptionellen Fähigkeiten, zum Überwachen von Bewegungen eingesetzt wird. Eine graphische Animation (Visualisierung) hat also die Aufgabe dem Benutzer einerseits eine visuelle Validierung zu ermöglichen, andererseits aber auch eine Optimierung des Roboterprogramms.

Abbildung 4.34 zeigt eine graphische Animation eines Roboters und seiner Arbeitsumgebung (Arbeitszelle). Die Validierung eines möglichen Modells in solch einer Umgebung geschieht in zwei Schritten: Zuerst erfolgt die graphische Animation der passiven Objekte (z.B. der Arbeitsplatte), aufbauend auf den Daten der Weltmodelle. Danach werden aktive Objekte (z.B. Tür, Manipulator) aufgrund ihrer Kinematik in die vorhandene Arbeitszelle integriert. Soll beispielsweise der in Abb. 4.34 zu sehende Manipulator integriert werden, so wird zuerst in einer Simulation überprüft, ob alle Stellungen erreichbar sind. Danach erst kann seine Bewegung graphisch dargestellt werden, wobei für den Einsatz mehrerer Manipulatoren auch mehrere Simulatoren notwendig sind. Dadurch ist es möglich zwischen unterschiedlichen Robotern

- eine Kooperation zu ermöglichen
- Bewegungen auf Kollisionen zu überprüfen.

Sind alle Roboter und deren Arbeitsumgebungen dargestellt, können unter Anderem verschiedene Distanzen ermittelt werden (z.B. zwischen Manipulator und zu greifendes Objekt). Diese Werte können dann für eine möglicherweise notwendige Kollisionsüberprüfung oder auch für Sensormodelle verwendet werden.

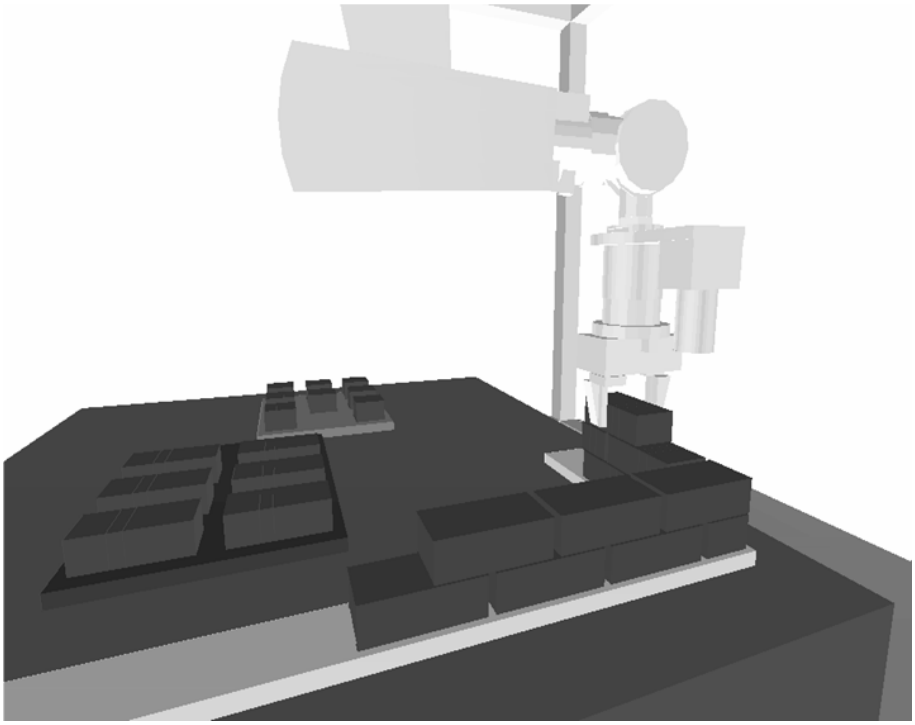


Abbildung 4.34: Graphische Animation (Arbeitszelle)

Ein Problem bei einer solchen graphischen Art der Validierung ist allerdings, dass sehr spezielle Rechner benötigt werden (CPUs, Grafikkarten etc.), wodurch die Kosten schnell in die Höhe steigen können.

Man kann unterschiedliche Arten der graphischen Animation unterscheiden, je nachdem welches geometrische Modell für die Repräsentation von Objekten gewählt wurde (siehe auch Abschnitt 4.1.1: Objektmodell). Die Wahl des geometrischen Modells hängt in starkem Maße von der entsprechenden Aufgabe ab, so ist z.B. für eine Kollisionsüberprüfung, oder auch eine Abstandsmessung, meist ein Flächenmodell am sinnvollsten. Nachfolgend sind einige der wichtigsten Darstellungsmodelle aufgeführt:

- Beim **Flächenmodell**, auch *surface model* genannt, werden die Objekte durch Nachbildung ihrer Oberfläche modelliert (siehe Abb. 4.35 a)). Dabei werden einzelne, ebene oder gekrümmte, Oberflächenelemente verwendet, mit denen durch Aneinandersetzten komplexe Objekte beschrieben werden können.

- Das **Kantenmodell**, auch *wireframe model* genannt, ist ebenfalls selbsterklärend: Dargestellt werden nun nicht ganze Flächen, sondern nur alle, auf der Oberfläche des Objekts existierenden Kanten (Abb. 4.35 b)). Dadurch verringert sich die Komplexität der einzugebenden Daten erheblich, der Eingabeaufwand kann allerdings, je nach Objekt, rapide ansteigen.

- Bei einer Darstellung auf Basis des **Hidden Line Modells** werden nur nun noch

Kanten graphisch dargestellt, welche auch tatsächlich sichtbar sind (Abb. 4.35 c)). Anders ausgedrückt: versteckte Kanten werden ausgeblendet, wodurch dieses Modellierungsverfahren weniger Rechenkapazität benötigt als das Kantenmodell.

- Das Modell der *Boundary Box* ist in Abbildung 4.35 d) zu sehen. Objekte werden hierbei durch Zusammensetzen von parametrisierbaren Quadern approximiert. Da hier das tatsächliche Volumen nur angenähert wird, können keine exakten Kollisionsberechnungen durchgeführt werden.

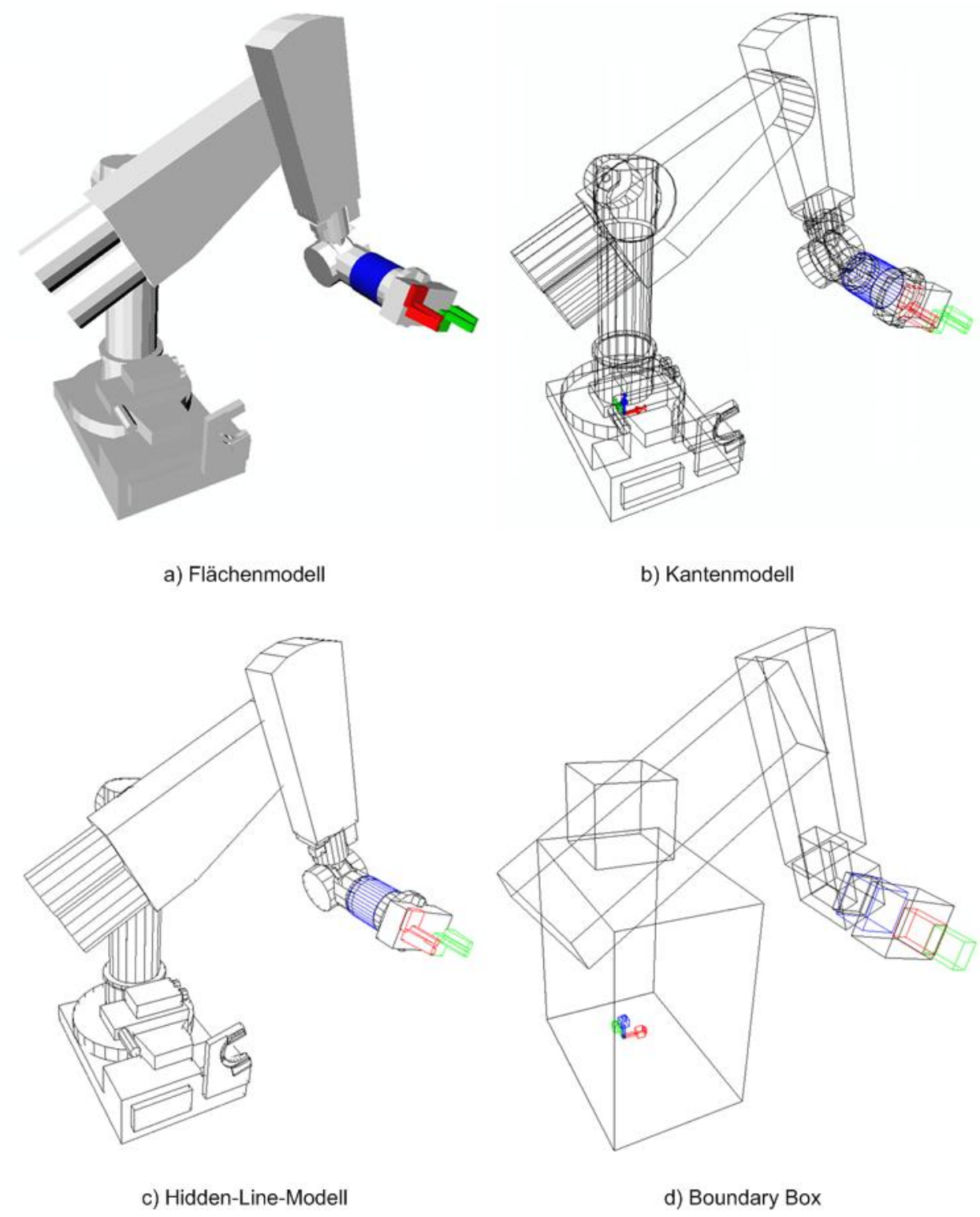


Abbildung 4.35: Verschiedene graphische Animationsmodelle

Kapitel 5

Programmieren durch Vormachen

Die Programmierung von Robotern ist zeitaufwendig und erfordert Expertenwissen. Da der Endbenutzer nicht über dieses verfügt, ist es ihm meist nicht möglich dem Roboter neue Aufgaben zu übergeben. Mit Hilfe von Programmieren durch Vormachen (PdV)[1] kann dem Endbenutzer die Möglichkeit gegeben werden, durch einfaches Vorführen der Lösung einer Aufgabe dem Roboter neue Aktionen und ganze Handlungsabläufe beizubringen. Die Vorführung wird vom Roboter mit Hilfe von Sensoren beobachtet, aufgezeichnet und interpretiert. Danach muß eine Kinematik gefunden werden, welche die beobachteten Bewegungen umsetzt. Anschließend ist der Roboter in der Lage, die vorgeführte Lösung zu wiederholen.

5.1 Neue Anforderungen an Robotersysteme

In der heutigen Zeit werden an Robotersysteme neue Anforderungen gestellt. Diese liegen bei der Produktion auf dem Gebiet der Klein-, Kleinst- und Unikatfertigung. Es sind Produkte mit vielen Ausstattungsvarianten und hoher Rekonfigurierbarkeit. Auch Serviceanwendungen finden in der Regel nicht in strukturierten Einsatzumgebungen statt. Die zu manipulierenden Objekte und mögliche Hindernisse sind variabel in der Arbeitsumgebung verteilt.

Die traditionellen Techniken der Roboterprogrammierung, wie das Teach-In oder explizite- und implizite Programmierung, können viele neue Anforderungen nicht erfüllen. Teach-In Verfahren sind meist unflexibel und nur begrenzt einsetzbar. Die explizite Programmierung erfordert Expertenwissen, und für die implizite Programmierung benötigt man viel spezielles Domänenwissen.

Die Lösung heißt interaktive Programmierung. Ein Ansatz hierbei ist PdV, das eine einfache, benutzerfreundliche und flexible Programmiermethode anbietet. Hierbei werden mit

[1] engl.: Programming by Demonstration (PbD)

Hilfe vom Benutzer durchgeführten Handlungssequenzen Programme generiert. So lernt der Roboter neue Aktionen. Neben der Robotik findet das Verfahren auch Anwendung in der Büroautomatisierung, dem Erstellen von graphischen Benutzeroberflächen und weiteren Bereichen [Cyp93].

An die interaktive Programmierung werden bestimmte Anforderungen gestellt:

1. Eine intuitive Interaktionsform zur einfachen Bedienung des Systems.

2. Weiterhin wird eine Transparenz der einzelnen Prozesse im Programmiersystem gefordert, um die Umsetzung von Handlungen des Bedieners nachvollziehen zu können.

3. Der Abgleich von Benutzerintention mit Systemhypothesen wird gefordert. So können zum Beispiel falsche Systemhypothesen korrigiert werden.

4. Auch eine gewisse Flexibilität und Optimierung der Programme ist wünschenswert. Ein Programm, das der Roboter erlernt hat, ist nützlicher, wenn es in viele Situationen anwendbar ist.

5. Zum Schluß ist noch zu nennen die Wiederverwendbarkeit von interaktiv erstellten Teillösungen für Manipulationsaufgaben. Die Nützlichkeit besteht darin, wie oft ein Funktionsbaustein in anderen Programmen wiederverwendbar ist.

Hinzu kommen vier Randbedingungen. Diese sind notwendig, wenn man leistungsfähige und automatische aber gleichzeitig auch sichere und komfortable Roboterprogramme erzeugen will:

1. Die Generierung eines Programms soll weitgehend automatisiert von statten gehen.

2. Die Interaktion mit dem Benutzer soll hinreichend für die Anwendung aber auf das Nötigste beschränkt sein.

3. Der Informationsgewinn und die Eindeutigkeit der Ergebnisse der Benutzerinteraktion für das System soll maximal sein.

4. Die gebotene Form der Mensch-Maschine Interaktion soll so benutzerfreundlich wie möglich sein.[Fri99]

Johannsen [Joh93] definiert die in Punkt 4 genannte Benutzerfreundlichkeit wie folgt: „Die Benutzerfreundlichkeit kann als das Bemühen interpretiert werden, die Kommunikation zwischen Menschen und der Maschine genauso „freundlich“ - also verständlich, transparent und flexibel - zu gestalten wie dies günstigenfalls bei der zwischenmenschlichen Kommunikation erreichbar ist.“

5.1.1 Programmieren durch Vormachen - ein generelles Framework

In [DRE+99] und [ERZD02] wird ein allgemeines Framework für PdV-Systeme vorgestellt. Als Basiskomponenten wurden hierbei folgende Phasen ausgemacht:

1. *Beobachtung* der Benutzerdemonstration mittels externer und interner Sensorik. Die am häufigsten genutzten Sensoren sind hier Kamerasysteme. Je nach zu erlernender Aufgabe lassen sich aber auch weitere Sensoren wie Datenhandschuhe oder Positionstracker verwenden. Mit den gewonnenen Sensordaten lassen sich dann Objekte klassifizieren, Trajektorien aufzeichnen und weitere Vorverarbeitungsschritte wie Grifferkennung durchführen.

2. *Segmentierung* in relevante Operationen oder Umweltzustände. Für diesen Schritt benötigt man die Trajektorien aus der Demonstration, eine Datenbasis mit den Sensordaten, einen Satz Aktionstypen (Griffe, Lageänderungen von Objekten), einen Satz von Elementaroperationen und das Umweltmodell. Durch eine geeignete Heuristik lässt sich damit die Segmentierung durchführen. Zusätzlich kann eine Interaktion mit dem Benutzer über verschiedene Kommunikationskanäle wie grafische Benutzerschnittstellen oder Sprache dabei helfen, eine sinnvolle Segmentierung zu finden und damit die Performanz des Systems zu verbessern. Durch Rückfragen lassen sich auch diejenigen Objekte ausfindig machen, die in die Lösung der Aufgabe involviert waren und damit unnötige Berechnungen aller Relationen zwischen allen Objekten einsparen. überdies sollte in diesem Schritt auch Rauschen ausgefiltert werden.

3. *Abstraktion* von der Demonstration um die Lösung der Aufgabe so allgemein wie möglich darzustellen. Um die generierte Sequenz später generalisieren zu können, müssen Instanzen falls möglich in Variablen umgewandelt werden. Dabei muss sichergestellt werden, dass in der Ausführungsphase nur solche Variablen instantiiert werden, die räumliche Vorbedingungen erfüllen. Es wird also für jeden "generalisierten" Operator ein Satz von Vorbedingungen in Form von wichtigen Relationen gespeichert. Generiert werden diese Vorbedingungen mit Hilfe von Hintergrundwissen und wiederum durch Rückfragen an den Benutzer. Man benötigt also eine deduktive Komponente, die die generalisierten Operatoren in Makro-Operatoren gruppiert. Da Makro-Operatoren weitere Makro-Operatoren als Kinder beinhalten können, lässt sich die gesamte Benutzervorführung auf verschiedenen Abstraktionsebenen darstellen. Durch die Generalisierung lässt sich die Lösung später auf ähnliche Problemklassen anwendbare Lösungsbeschreibungen abbilden. In dieser Phase lassen sich auch vom Demonstrator durchgeführte spontane und nicht zielorientierte Bewegungen ausfiltern.

4. *Transfer* der internen Wissensrepräsentation auf das Zielsystem. Aus den zuvor gewonnenen semantischen Informationen lässt sich jetzt ein ausführbares Roboterprogramm generieren. Dafür müssen die Operationen auf die Operationen des Zielsy-

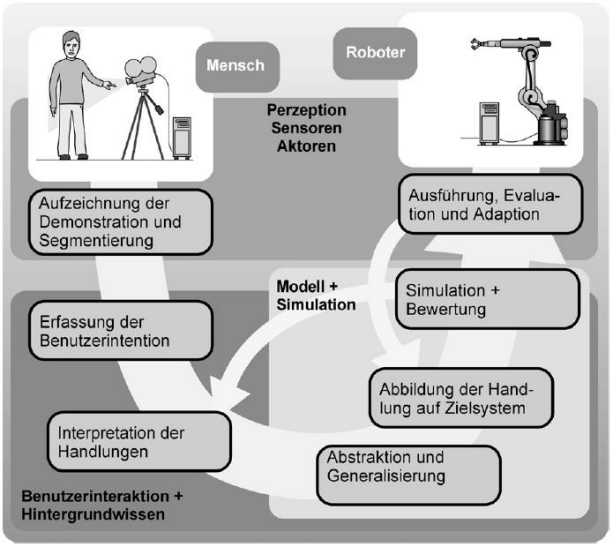


Abbildung 5.1: überblick des Abbildungsprozesses zwischen Mensch- und Roboterskills

stems abgebildet werden. Die Ausgabe dieser Phase ist eine Sequenz von Elementarbewegungen, die nur für das Zielsystem und das jeweilige Umweltmodell gültig sind. Diese kann direkt in das Simulationsmodul weitergeleitet werden.

5. *Simulation* des physikalischen Vorgangs zur Validierung der getroffenen Entscheidungen. In der Simulation wird die gelernte Aufgabe von einem virtuellen Modell des Roboters in einer virtuellen Umwelt an virtuellen Objekten ausgeführt. Anhand einer visuellen Ausgabe lässt sich vom Benutzer die korrekte Ausführung der Aufgabe überprüfen.

6. *Ausführung* auf dem Zielsystem. Die zuvor validierte Sequenz elementarer Roboterbewegungen wird an den Roboter-Controller weitergereicht. Wenn bei der Modellierung in der Simulationsphase keine Fehler gemacht wurden, ist es sehr wahrscheinlich, dass die Ausführung nicht fehlschlagen wird.

Am problematischsten sind hier die Schritte 2 und 3. Sie erfordern explizites Hintergrundwissen über die Umwelt und ein genaues Modell des Benutzers. Existierende Systeme beschränken sich allerdings meist auf nur eine der oben aufgelisteten Phasen. Einen überblick auf die globale PdV-Zyklus liefert Abbildung 5.1.

5.2 Klassifikation von PdV-Systemen

Beim Programmieren durch Vormachen gibt es mehrere Möglichkeiten die verwendeten Verfahren zu klassifizieren. In Abbildung 5.2 ist ein überblick über vier Klassifikationskriterien gegeben, welche im Folgenden genauer vorgestellt werden.

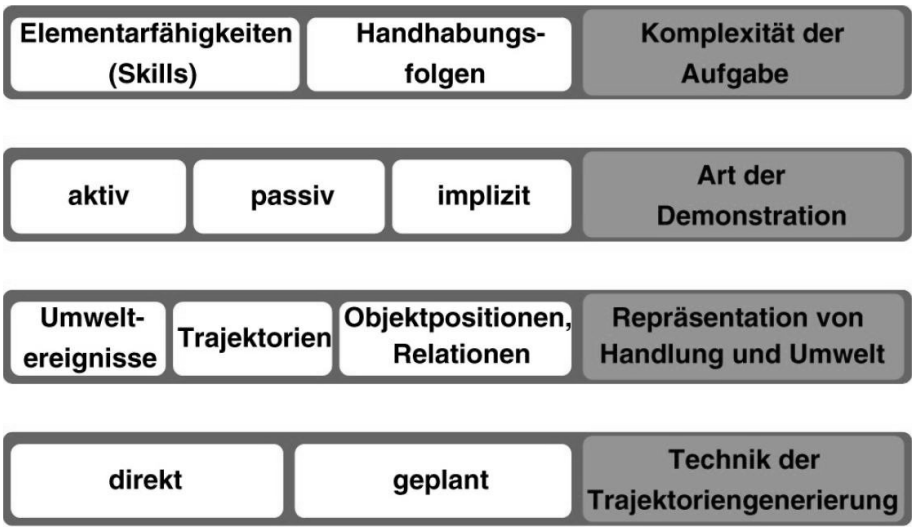


Abbildung 5.2: Klassifikation von PdV-Systemen [DRE+99]

5.2.1 Komplexität der Aufgabe

Hierbei sind zuerst die Elementarfähigkeiten[2] zu nennen. Sie sind einfache Reflexe oder einfache Bewegungen. Beide stellen einen direkten Sensor-Aktor Bezug her. Um das gewünschte Verhalten zu erlernen, werden neuronale Netze auf adaptiven Regelkreisen sowie Zustandsautomaten verwendet. Das erlernte Wissen ist nur für die trainierte Sensor-Aktor Kombination zu verwenden. Auch ist eine große Anzahl an Lernbeispielen nötig.

Um ganze Handlungsfolgen vom Benutzer zu lernen, muß ein breites Hintergrund-, Planungs- und Modellwissen vorhanden sein. Außerdem sollte die Möglichkeit bestehen, weiter im Dialog mit dem Benutzer zu bleiben, um eventuell auftretende Unklarheiten im Handlungsablauf zu klären[AL91, KH95, Kai97, Seg88, II90, KK94].

5.2.2 Art der Demonstration

Die aktive Demonstration wird direkt vom Benutzer ausgeführt, während er vom System überwacht wird. Dies kann durch Datenhandschuhe, Kameras, haptische Interfaces und andere Trackingsysteme geschehen. Aus den gesammelten Umwelt- und Benutzerdaten gilt es die relevanten Aktionen herauszufinden.

Um den Demonstrationsablauf passiv zu gestalten, können Master-Slave Systeme zum Einsatz kommen. Auch eine 3D Maus oder ein graphisches Interface ist möglich. Während der Vorführung zeichnet der Roboter einfach mit seinen internen und externen Sensoren alle Veränderungen auf. Da er anschließend mit Hilfe der aufgezeichneten Daten (Gelenkwinkel

[2]engl.: skills

u.s.w.) den genauen Ablauf wiederholen kann, ist das Ziel der Benutzerdemonstration ohne viel Intelligenz auf Roboterseite zu erreichen. Ein Nachteil ist, daß die Anwendung der gelernten Handlung auf ein Zielsystem beschränkt ist.

Bei der impliziten Demonstration ist dem Benutzer ein fester Satz von graphischen Befehlen gegeben, mit denen er den Zielzustand erreicht. Bei einer festen übersetzung der graphischen Anweisungen in Roboterbefehle kommt man schnell zu guten Ergebnissen. Andererseits ist für die beschränkte Anzahl von Befehlen, die dem Benutzer zur Verfügung stehen, die Eingabe von Beispielen nicht immer einfach. [FHD98, KII94, KII92, Kai97, KH95]

5.2.3 Repräsentation von Handlung und Umwelt

Hierbei können drei Bereiche voneinander abgetrennt werden:

- die Umweltergebnisse mit ihren Wirkungen und Reaktionen auf die Umwelt (das finden der Kausalitäten ist schwierig)
- die Trajektorien für eine explizite analytische Repräsentation (keine Generalisierung möglich)
- die Objektpositionen mit ihren Relationen und Operationen (es ist eine strukturierte Umgebung nötig)

Für eine vollständige Repräsentation sind zum Beispiel die Trajektorien der Hand und der Finger zu bestimmen. Zu jedem Zeitpunkt der Demonstration wird Wissen über Objekte (z.B. Position) gefordert. Ein kinematisches Modell kann auch weiterhelfen.

5.2.4 Technik der Trajektoriengenerierung

Ist eine bestimmte Benutzereingabe mit den Sensoren aufgenommen, muß sie für das Zielsystem aufbereitet werden. Bei der direkten Abbildung ist eine explizite Transformation gegeben oder erlernt. Das Problem hierbei ist, daß Zielsystem und Zielumgebung korrespondieren müssen. Bei verrauschten Daten kann eine Hidden Markov Repräsentation hilfreich sein.

Die geplante Abbildung benötigt Zwischenziele und Zustandsfolgen. Diese werden durch eine Analyse der Benutzereingabe gefunden. Viele Informationen wie einzelne Trajektoren werden nicht mehr benötigt. Danach ist eine gute Planung im Konfigurationsraum möglich[KI97, Kan97, DRE+99].

5.3 Prozeßkomponenten der interaktiven Roboterprogrammierung

Dieses Kapitel befaßt sich mit der Analyse der bei einer interaktiven Programmierung beteiligten Komponenten. In Abbildung 5.1 sind die vier Einheiten und ihre Beziehungen dargestellt.

1. Der Benutzer ist zu nennen, welcher das System verwendet.
2. Eine weitere Einheit bilden die Sensoren und Mensch-Maschine Interaktionsformen.
3. Eingebunden in das Programmiersystem ist ein Weltmodell sowie Planungs- und Entscheidungsmechanismen.
4. Die ausführenden Manipulatoren sind mit einer erweiterten Steuerung ausgestattet. Weiterhin dienen sie zur Ausführung von Elementarfähigkeiten und haben die Fähigkeit zur Kommunikation.

Jeder diese Komponenten trägt ihren spezifischen Anteil zum interaktiven Programmieren bei und wird im Folgenden genauer besprochen.

5.3.1 Der Benutzer

Die interaktive Programmierung geschieht durch den Menschen. Er ist in der Lage multimodal mit dem Programmiersystem zu interagieren.

Physische Demonstration: Sie entspricht dem Menschen am besten, da sie für ihn die natürliche Methode zur Vermittlung von Aktionswissen darstellt. Alle Handlungsabläufe werden normal vollzogen und vom Programmiersystem verwertet. Aber nicht alle physischen Demonstrationen können vom Menschen aufgrund seiner beschränkten sensorischen, motorischen und intellektuellen Fähigkeiten ausgeführt werden. So kann man keine absolute Positioniergenauigkeit erwarten und auch die Wiederholgenauigkeit ist begrenzt. Auch produzieren menschliche Benutzer schon bei einfachen Manipulationsaufgaben häufig ineffiziente oder überflüssige Handlungen [KFD95, Rea94]. Ebenfalls tragen der Umfang und die Komplexität der Demonstration der zu programmierenden Aufgabe dazu bei.

Graphische Demonstration: Diese ist für den menschlichen Benutzer mit den gleichen Problemen verbunden wie die physische. Hinzukommen Probleme der Navigation und Koordination in der simulierten Umgebung. Aufgrund der fehlenden Realität der virtuellen Welt kann es sogar zu Simulationsübelkeit[3] kommen. Dies geschieht durch den Widerspruch

[3] simulation sickness

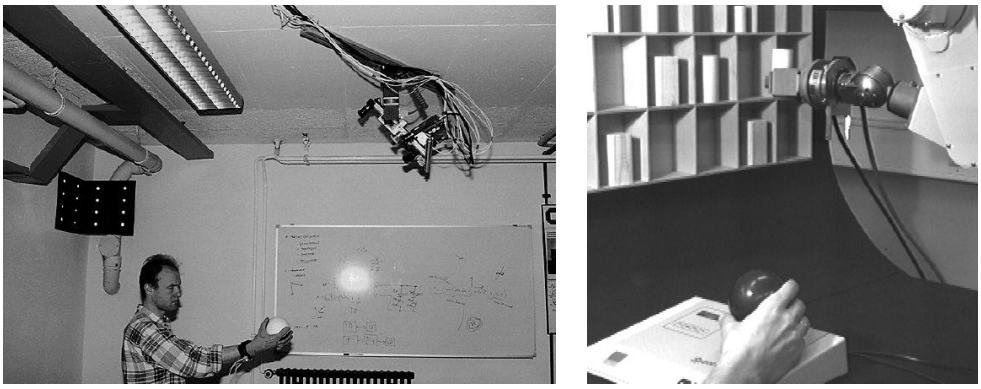
der Signale, welche vom Gleichgewichts-, Orientierungs-, Seh- und Gehörsinn aufgenommen werden [KLBL93].

Symbolische/Ikonische Demonstration: Bei ihr setzt der menschliche Benutzer aus vorhandenen Teillösungen eine Lösung für die Manipulationsaufgabe zusammen. Das Problem ist hierbei die Bestimmung der zu manipulierenden Objekte und die Parametrisierung der Bewegungsfolge.

Die Kommentierung von Systemhypothesen, die Vermittlung der Sensorik von Aktionen und der eigenen Intention ist für den Benutzer bei entsprechenden Benutzerschnittstellen (symbolische, graphische Darstellung) leichter.

5.3.2 Interaktionsformen und Sensoren

Die zur Kommunikation gewählte Interaktionsform mit dem Benutzer und die zu seiner Beobachtung benutzen Sensoren haben großen Einfluß auf die Qualität der Vorführung. Sowohl die Interaktionsarten als auch die verschiedenen Sensoren werden in diesem Teil genauer betrachtet. Die Interaktion kann von einfachem Kommentieren und überwachen von Hypothesen und Arbeitsschritten des Programmiersystems bis hin zu verschiedenen Formen der Vorführung reichen:



(a) (b)

Abbildung 5.3: Interaktionsform - Physische Demonstration (a): Vorführung mit Datenhandschuh und Kameras (b): Vorführen durch Steuerung eines Roboters mit einer 6D Kugel

Physikalische Demonstration: Hierbei vollzieht der Benutzer die Manipulationsaufgabe mit realen Objekten (s. Abb.: 5.3). Während der Vorführung wird er vom Programmiersystem durch Sensoren erfaßt und seine Handlungen aufgezeichnet. Die physikalische

Demonstration ist als die natürlichste Art zur Demonstration von Manipulationsaufgaben für den Menschen zu sehen. Objekte können direkt mit den Händen manipuliert werden [YMK96] oder mittels spezieller Vorführgeräte (z.B. Laserstift). In jedem Fall ist ein großes Maß an Konzentration auf Seiten des Benutzers gefordert. Er muß sich weiterhin über die Beschränkung und Konfiguration der ihn beobachtenden Sensoren bewußt sein (z.B. Verdeckung bei Beobachtung durch eine Kamera). Wegen der Abtastfrequenz der Sensoren ist auch die Geschwindigkeit, mit der die Demonstration vorgeführt wird, zu beachten. Oft ist es auch sinnvoll, die Manipulationsaufgabe in Teillösungen zu untergliedern, um gegebenenfalls Fehler leichter zu beheben.

(a) (b)

Abbildung 5.4: Interaktionsform - Graphische Demonstration (a): Datenhelm (b): Shutter Brille

Graphische Demonstration: Der Benutzer kann hierbei die Manipulationsaufgabe lösen, indem er 3D Objekte in einer simulierten Umgebung bewegt. Die Probleme der physischen Demonstration mit Sensoren (eingeschränkte Beobachtbarkeit, limitierte Genauigkeit, zeitlicher Aufwand) sind bei dieser Art der interaktiven Programmierung nicht vorhanden. Es wird versucht die Vorteile der intuitiven physischen Demonstration zu übernehmen und die angesprochenen Nachteile zu umgehen [EJ97]. Leider entstehen bei dieser Art von Demonstration andere Probleme. Auf einem Monitor ist die Wiedergabe einer 3D Szene mit ihren zu manipulierenden Objekten nur schwer zu erfassen. Sollen nur Translationen und Rotationen in einer Ebene ausgeführt werden (z.B Bestückung einer Platine) reicht ein Monitor aus. Besser eigenen sich für die Darstellung von 3D Umgebungen Datenhelme (Abb. 5.4a) [TH92] und Shutter Brillen (Abb. 5.4b) [MT90] sowie 3D Höhlen [GMF$^+$92]. Mit den bisher beschriebenen Hilfsmitteln zur interaktiven Programmierung ist keine Möglichkeit der Kraftrückkopplung gegeben. Die in Abbildung 5.5 aufgeführten haptischen Ein- und Ausgabegeräte (Datenhandschuh mit Exoskelett und Phantom) ermöglichen die Reaktionskräfte an den Benutzer weiterzugeben [Bur96]. Er kann besser auf die simulierte Welt einwirken, da er ein Feedback bekommt. Wegen der Echtzeitmodellaktualisierung ist aber

ein hoher Rechenaufwand nötig.

(a)

(b)

Abbildung 5.5: Interaktionsform - Graphische Demonstration Demonstration (a): Exoskelett für Datenhandschuh (b): Phantom Manipulator

Symbolische/Ikonische Demonstration: Es wird eine Aktionssequenz erzeugt, durch graphisches Aneinanderreihen vorhandener Teillösungen. Für diese Art der Demonstration reicht eine menügesteuerte, konventionelle graphische Schnittstelle [Rie97]. Es müssen nur entsprechende Operatoren vorhanden sein, welche alle Informationen zur Manipulation einzelner Objekte beinhalten. Die Ressourcenanforderungen an das System sind dementsprechend gering.

Kommentierung: Sie ist eine ergänzende Interaktionsform zur physischen oder graphischen Programmierung. Der Benutzer nimmt hierbei keine aktive Rolle ein, sondern reagiert auf Systemhypothesen. Nach einer Demonstration erhält man die Möglichkeit, Systemhypothesen zu präsentieren und durch Auswahl, Editierungen und Ablehnung mit der Benutzerintension abzugleichen.

In vielen Anwendungen (Textverarbeitung, Graphikanwendungen, Programmierung) sind textuelle oder menübasierte Schnittstellen ausreichend. In der Robotik kann es durch den räumlichen Bezug der Systeme zusätzlich nötig sein, über zweidimensionale graphische Interaktionsmuster hinaus eine dreidimensionale Schnittstelle zu bieten. Nur mit Kommentierung ist eine überwachung und gegebenenfalls eine Korrektur der systeminternen Programmgenerierung möglich.

Sensoren sind für die physische und graphische Demonstration unablässig. Es ist eine große Anzahl an verschiedenen Arten von Sensoren möglich. Deshalb werden hier nur einige Klassen von Sensoren mit großer Verbreitung in der interaktiven Programmierung angesprochen.

Bildgebende Sensoren: Diese sind meist in Form von Kameras zur Beobachtung anzutreffen [SK93, KI97]. Sie haben den Vorteil, daß zusätzlich zur Aufnahme und Analyse der Demonstration auch eine Modellierung der Umwelt möglich ist. Hierfür verwendet man Verfahren zur Objektmodellierung, -erkennung und Bestimmung von Objektlage [SF95a]. Mit den bildgebenden Sensoren ist leider ein hoher Rechenaufwand verbunden. Es existieren auch eine Anzahl ungelöster Probleme (im Bereich der Objekterkennung, Bildfolgenanalyse, Szeneninterpretation). Als letztes sind die Schwierigkeiten für den Benutzer zu benennen. Er muß darauf achten, im optimalen Bildbereich zu agieren und Verdeckungen zwischen Hand und zu manipulierendem Objekt vermeiden.

Magnetfeldbasierte Positionssensoren: Mit diesen Sensoren lassen sich direkt die Position und Orientierung der Benutzerhand bestimmen. Das Problem der Verdeckung relevanter Szeneninformationen entfällt. Auch verschiedene Lichtverhältnisse können die Erkennung nicht behindern. Darum werden magnetfeldbasierte Positionssensoren vielfach eingesetzt.

Probleme ergeben sich bei dieser Messmethode durch die quadratische Abnahme der Feldstärke. Hinzu kommen Störungen des Sensorsystems durch metallische Gegenstände, Monitore, Netzgeräte und andere elektronische Felder. Der Arbeitsbereich und die Genauigkeit wird deshalb stark eingeschränkt.

Interne Robotersensoren: Die aus internen Robotersensoren gewonnenen Werte sind zuverlässig. Außerdem ergibt sich der Vorteil, wenn für Demonstration und Ausführung der gleiche Roboter verwendet wird, eine erfolgreiche Demonstration der Ausführung (bei gleicher Umweltsituation) garantiert ist.

Die Nachteile sind hierbei der hohe Geräteaufwand und ein geringe Komfort für den Benutzer. Eine kompetente Steuerung des Roboters ist für Nichtexperten sehr schwierig [FK97]. Trotz der nicht vorhandenen Benutzerfreundlichkeit, die beim interaktiven Programmieren gefordert wird, verwendet man interne Robotersensoren bei klassischen Teach-In und im Bereich der Telerobotik.

Datenhandschuhe & -anzüge: Beide werden meist in Kombination mit magnetfeldbasierten Sensoren verwendet [FHD98, TK96]. Es werden Dehnmeßstreifen und Lichtleiter verwendet, um die Bewegungen zu messen. Diese führen zu jedem Zeitpunkt der Demonstration zu zuverlässigen Messungen der kinematischen Zustände.

Aufgrund des unterschiedlichen Körperbaus verschiedener Benutzer ist die Genauigkeit dieser Sensoren beschränkt. Selbst beim erneuten Anlegen beim selben Benutzer kommen schon Abweichungen zustande.

Exoskelette: Bei ihnen ist bei der Aufzeichnung der Gelenkwinkel eine höhere Genau-

igkeit zu erreichen als bei Datenhandschuhen und Datenanzügen. Die größere Genauigkeit ist durch ihre mechanische Struktur und Anpaßbarkeit begründet.

Dem entgegen stehen das Gewicht, die hohen Kosten und die Komplexität ihrer Handhabung, welche den Benutzer in seiner Demonstration einschränkt.

Da jeder der vorgestellten Sensortypen sowie der möglichen Demonstrationsarten Vor- und Nachteile hat, ist es sinnvoll sie verschieden zu kombinieren. So kann eine grobe Lokalisierung über magnetfeldbasierte Sensoren geschehen, und mit bildgebenden Sensoren würde die Lagebestimmung der Hand aufgezeichnet werden. Hinzu kommen dann aber Aufgaben der Sensordatenfusion.

Objektdetektion und Objektverfolgung: Sind alle Sensordaten gesammelt kann mit der Objektdetektion begonnen werden. Ihr Ziel ist es ein Weltmodell zu erstellen. Oft wird die Objektdetektion durch eine Deckenkammera beschleunigt. Um ein Objekt erkennen zu können, muß eine Korrespondenz zu den Sensordaten gefunden werden. Hierbei liegt ein Objekt mit seinen passenden Merkmalen (Punkte, Kanten, Flächen) im System vor. Aus den Sensordaten gilt es ebenfalls Merkmale zu extrahieren. Ein erster Ansatz ist meist das Differenzbild, zur weiteren Klassifikation schließen sich Verfahren wie die Hough Transformation an.

Wurde ein Objekt identifiziert, kann mit der Objektverfolgung begonnen werden. Hierbei soll die Position des Objektes vorausgesagt werden, da eine permanente Neuerkennung zu rechenaufwendig ist. Mit geeigneten kinematischen Bewegungsmodellen wird die neue Objektlage bestimmt. Die Glattheit von physikalischen Bewegungen erleichtert diese Arbeit [App95].

5.3.3 Programmiersystem

Ein weiterer Teil der in Abbbildung 5.1 beschriebenen Prozesskomponenten ist das Programmiersystem.
Die Aufgabe eines interaktiven Programmiersystems besteht aus mehreren Teilen, wie man aus Abbildung 5.6 entnehmen kann. Mit ihrer Hilfe wird für die Manipulationsaufgabe eine interaktive Lösung generiert und in geeigneter Repräsentation gespeichert. Diese muß sich später in Elementaroperationen zerlegen lassen. Das Programmiersystem muß auch in der Lage sein, Programme effizient an Manipulatorzielsysteme zu übergeben. Auch die Aufgabenlösungen auf konkrete Manipulatoren zu übertragen, fällt ihm zu.

Das Programm wird erstellt durch das Programmiersystem mit Methoden zur Datenanalyse, Transformation von Repräsentation und Ableitung von Wissen. Hierbei kann das System auf das Benutzerwissen und das Hintergrundwissen in Form einer Wissensbasis zurückgreifen.

Die Bearbeitungsschritte (Abb.: 5.6) beginnen mit Aufzeichnen der Demonstration und der Identifikation von Schlüsselaktionen (Wann, was oder wie wurde gegriffen und losge-

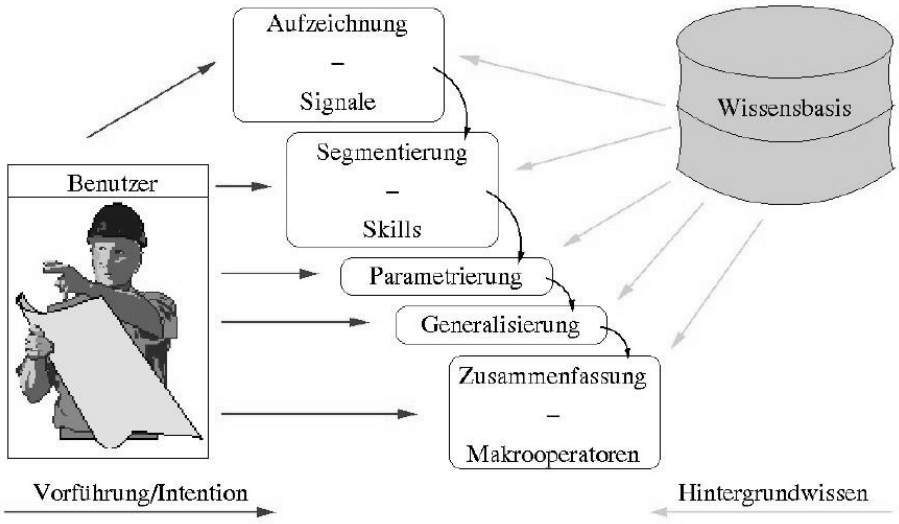


Abbildung 5.6: Arbeitsschritte eines Programmiersystems

lassen). Danach findet eine Korrektur der Schlüsselpunkte und des Trajektorienverlaufs statt. Weiter geht es mit der Interpretation der Transportphase (Objekt transportiert, aufgenommen, abgelegt), gefolgt von der semantische Analyse (Umweltveränderungen) und der semantischen Interpretation (welche Veränderungen sind für die Lösung wichtig). Der letzte Schritt ist dann die Codierung und Optimierung der Lösung.

Damit das interaktive Programmieren zum Erfolg führt, müssen ein paar Bedingungen erfüllt sein. Als erstes steht eine robuste Aufnahme (im Bezug auf potenzielle Fehler und Rauschen) und Analyse von Benutzerdemonstration. Am Ende der Verarbeitung muß die Benutzerintension durch eine richtige Transformation in Elementarfähigkeiten überführt sein. Weiterhin will man, daß die Operatorensequenz verallgemeinerbar ist und trotzdem robust, allgemein anwendbar und zudem die Benutzerintension immer noch exakt wiedergibt. So müssen alle Datenverarbeitungs- und Ableitungschritte bezüglich der Intentionen des Benutzers gerechtfertigt sein. So wird nicht nur die Richtigkeit der Umsetzung gewährleistet, sondern auch eine Gefährdung des Benutzers bei der Ausführung des Programms minimiert.

Für die Repräsentationstransformation gibt es außer dem Rückgriff auf Hintergrundwissen [THSy93] auch die Möglichkeit der Verwendung statistischer Verfahren. Korrelationsanalysen helfen Segmente konstanter Abhängigkeit während der Benutzervorführung zu ermitteln [Kai97]. So können leichter Elementarfähigkeiten und Reglungsstrategien erkannt werden. Es gibt auch Ansätze mit neuronalen Netzen (eine Folge von Elementarfähigkeiten mit den dazugehörigen vorsegmentierten Sensordaten trainiert diese).

Die Generalisierung von Operatorfolgen (Sequenz von Bewegungs- und Greifoperationen) kann geschehen durch änderung (generalisieren/spezifizieren) der Berechnungsbedingun-

gen und Gültigkeitsbereichen von Parametern (von Bewegungs- und Greifoperationen). Es kann aber auch der Kontrollfluß des generierten Programms verallgemeinert werden. Dies erreicht man durch Einbinden von Kontrollstrukturen wie Programmschleifen und Verzweigungen. Dadurch wird das Programm in einer größeren Anzahl von Umweltsituationen einsetzbar. Zur Programmgeneralisierung (Parameter- und Strukturgeneralisierung) kann man drei verschiedene maschinelle Lernverfahren nennen:

- interaktive Verfahren
- deduktive, analytische Verfahren
- erklärungsbasierte Verfahren

Leider gibt es gewisse Beschränkungen bei heutigen Systemen zur interaktiven Programmierung. So ist es schwierig, Benutzerfreundlichkeit und automatische Datenanalyse und -transformation sowie Generalisierung gegenüber den Sicherheitsanforderungen und dem Abgleich mit den Benutzerintensionen in Einklang zu bringen. Bei neuronalen Netzen und statistischen Verfahren kann die große erforderliche Menge an Beispielen zu einem Problem werden, da sie bei realen Anwendungen nicht zur Verfühgung stehen. Programme können noch nicht automatisch aus Benutzervorführungen generiert werden. Bisher existieren nur leistungsstarke Analyse- und Lernverfahren, die zwar eine effiziente Generalisierung und Verarbeitung der Aktionssequenz erlauben, aber noch nicht genau genug die Benutzerintension wiedergeben oder Korrektheit und Sicherheit garantieren.

5.3.4 Der Manipulator

Ein Manipulator ist ein Roboter, der sich aus einer Steuerung und mechanischen Komponenten (Manipulatorarm) zusammen setzt. Er ist in der Lage, die Position und Orientierung eines Objektes bezüglich eines Bezugskoordinatensystems zu ändern.

Manipulatorabhängige und -unabhängige Repräsentation: Die vollständige (manipulatorabhängige) Lösung von Manipulationsaufgaben besteht aus Aktionssequenzen oder aus Gelenkwinkel-, Kraft- und Momenttrajektorien. Ist eine Aktionssequenz vorhanden, kann diese direkt ausgeführt werden. Andernfalls kann mit Hilfe der Manipulatorsteuerung die zu den Trajektorien gehörende Sequenz erstellt werden (z.B. klassisches Teach-In).

Bei Serviceanwendungen erwies sie sich aber aus zwei Gründen als nicht flexibel genug. Zum einen erlauben invariante Trajektorien keine Veränderung der Lage zu den manipulierten Objekten in der Umwelt. Auch Variationen in Material, Form und Gewicht beim Manipulationsobjekt können nicht berücksichtigt werden. Bei einer Veränderung dieser Werte zur Programmausführung verlieren statische Trajektorien ihre Gültigkeit. Als zweites ist zu benennen, daß unterschiedliche Manipulatoren verschiedene Konfigurationsräume aufweisen aufgrund der Kinematik der Manipulatorarme und Endeffektoren. Des Weiteren zeigen sie auch eine verschiedene Sensorausstattung.

Da im Dienstleistungsbereich eine schwach strukturierte Umwelt anzutreffen ist und auch die Wiederverwendbarkeit gefordert wird, ist die manipulatorabhängige Programmrepräsentation nicht sinnvoll. Weiterhin benötigt man bei expliziter Trajektorienerfassung viel Speicher.

Elemantarfähigkeiten: Diese beschreiben durch Sensor-Aktor-Regeln eine Handlung vom Start bis zum Ende. Die Situation wird hierbei nur durch ein über lokale Sensorwerte erstellte Umweltinformation beschrieben.

Es werden komplexe Manipulationsaufgaben durch elementare Manipulations- und Bewegungsfähigkeiten gelöst. Sie überführen den Roboter von einem aktuellen Startzustand durch eine geeignete Aktionsfolge, wie in den Elementarfähigkeiten definiert, in einen Zielzustand. Da sehr schnelle Reaktionszeiten gefordert sind und die Umweltinformationen nur lokal vorliegen, sind Elementarfähigkeiten auf der Ebene der Regelungen eingebracht. Neue Elementarfähigkeiten können auch durch Benutzerdemonstrationen gelernt werden (Abb.: 5.7). Es können unüberwachte[4] sowie sensorrückgekoppelte[5] Elementarfähigkeiten erzeugt werden. Vorteile findet man in der flexiblen Repräsentation und der benutzer-

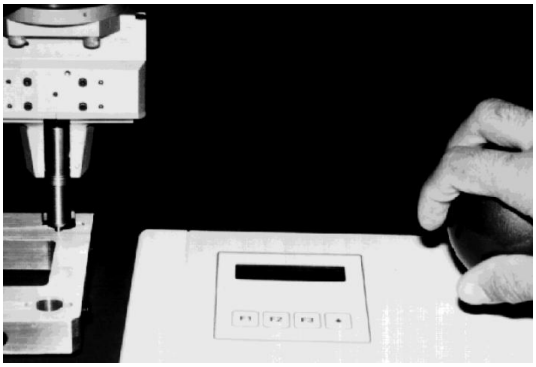


Abbildung 5.7: Erzeugung von Elementarfähigkeiten aus einer Demonstration mittels einer 6D Maus

freundlichen symbolischen Darstellung. Weiterhin zeigt die manipulatorunabhängige Repräsentation größere Flexibilität gegenüber Umweltveränderungen und unterschiedlichen Manipulatorentypen. Als Nachteil ist zu sehen, daß Elementarfähigkeiten für jeden Manipulator a-priori explizit erzeugt werden müssen. Somit ist die manipulatorunabhängige Repräsentation im gegebenen Umfeld sinnvoller.

Viele Arbeiten zur Gewinnung unüberwachter Elementarfähigkeiten für Manipulationen beschäftigen sich mit Rekonstruktion von Raumtrajektorien aus den Benutzervorführungen [SK93, NH94]. Es gibt aber auch Projekte, welche möglichst ähnliche Bahnsegmente finden und diese in abstrakter Form speichern [HT93].

[4]engl.: open-loop
[5]engl.: closed-loop

In der Tabelle 5.1 sind die Eigenschaften der manipulatorabhängigen und -unabhängigen Programmrepräsentation noch einmal gegenübergestellt.

Eigenschaft	**Manipulatorabhängig**	**Manipulatorunabhängig**
Kommandoebene	Signalebene	Symbolebene
Kontrollfluß	numerisch	symbolisch
Kontrollwissen	Solltrajektorien	Umweltzustandsfolge
Aktionseinheit	Gelenkwinkelsätze	Elementarfähigkeiten
Zieldefinition	manipulatorbezogen	umwelt- / aufgabenbezogen

Tabelle 5.1: manipulatorabhängigen und -unabhängigen Programmrepräsentation [Fri99]

Abschließend sei noch erwähnt, daß Elementarfunktionen einen hohen Wiederverwendbarkeitswert haben. So braucht ein Programm nur die Sequenz der Elementarfähigkeiten enthalten und ihre einzelnen Parameter bestimmen und repräsentieren. Oft weichen in einer dynamischen Welt die Parameter der Elementarfähigkeiten von denen der Benutzerdemonstation ab. Aus diesem Grund werden für variable Parameter Berechnungsfunktionen und Auswahlbedingungen statt konkreter Werte benutzt. [Fri99]

5.4 Beispiele für Programmieren durch Vormachen

5.4.1 Air Hockey

Air Hockey ist ein Spiel für zwei Spieler. Sie stehen sich am Spieltisch gegenüber. Mit Hilfe von speziellen runden Spielschiebern schlagen sie einen flachen Puck über das Spielfeld. Durch viele kleine Löcher auf dem Spielfeld wird Luft geblasen, so dass ein Luftkissen entsteht, auf dem der Puck nahezu reibungslos gleiten kann. Um zu verhindern, dass der Puck vom Spieltisch fällt, ist das Spielfeld durch eine Bande begrenzt, von welcher der Puck mit geringem Geschwindigkeitsverlust abprallt. Ziel des Spiels ist es, den Puck in das gegnerische Tor zu schlagen, das sich am gegenüberliegenden Ende des Tisches befindet.

Bentivegna et al. haben ein Framework entwickelt, mit dem ein Agent einen Satz vordefinierter Primitiven durch Beobachten initial eintrainiert und durch übung verbessert. Implementiert wurde das Framework u.A. in Form eines Agenten, der Air Hockey sowohl in einer Simulation als auch auf einem realen Tisch spielen kann ([BUAC02], [BA02], [BA00], [BAC04]). Dieses System wird ich im Folgenden genauer vorgestellt.

Die Entwickler haben sich aus mehreren Gründen für das Air Hockey Spiel entschieden: Zum einen nimmt die Hardware-Umgebung (Abbildung 5.8) nur so viel Platz ein, dass sie sich in einer Labor-Umgebung aufbauen lässt, und es kann eine mit der Maus spielbare Software-Version (Abbildung 5.9) entwickelt werden. Für die Hardware-Variante wurde der humanoide Roboter DB verwendet, gegen den man ganz normal im Air Hockey an-

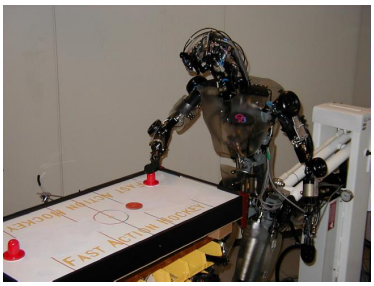


Abbildung 5.8: Hardware-Umgebung für Air Hockey

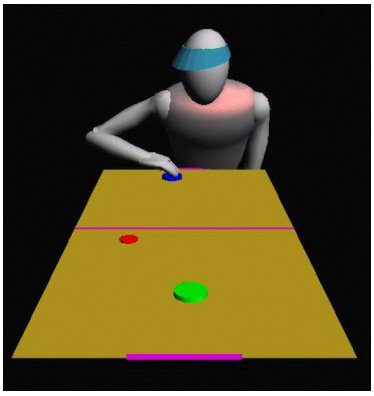


Abbildung 5.9: Software-Umgebung für Air Hockey

treten kann. Die Software-Version lässt sich mit der Maus gegen einen virtuellen Gegner spielen. Natürlich fehlen in der Software-Variante reale Umweltbedingungen, aber mit ihr lassen sich nützliche Trainingsdaten sammeln, ohne dabei die Kosten für das komplette Roboter-Setup tragen zu müssen. Außerdem lassen sich in der Simulation wiederholbare und leicht zu kontrollierende Experimente durchführen. Zum anderen entstehen relativ geringe Ansprüche an die Sensorik, da sich die Bewegungen aller Objekte in einer Ebene abspielen. Es genügt also eine Kamera, um das Spiel zu beobachten und zu steuern. Mit jedem Zeitschritt müssen die Positionen des Pucks, des eigenen und des gegnerischen Schlägers verfolgt werden. Zur Kalibrierung wurden hier zusätzlich noch sechs Marker am Rand des Spielfeldes angebracht. Da sich der Puck sehr schnell bewegen kann, ist es wichtig die Daten mit einer ausreichend hohen Wiederholrate zu erhalten. Sonst wird es sehr schwierig, die Bahn des Pucks rechtzeitig und genau genug vorherzusagen, um den Roboterarm in die gewünschte Position zu bringen. Hier wurde ein Kamerasystem mit 60Hz gewählt, und auch die Software-Variante arbeitet mit 60Hz. Auf die Probleme, die sich beim Kalibrieren durch Kopfbewegungen des Roboters und damit auch der Kamera und beim Tracken der Objekte, z.B. durch Verdeckung des Pucks durch den Roboterarm, ergeben, wird in [BUAC02] genauer eingegangen. Ich werde hier hauptsächlich auf die Lernverfahren eingehen, wie sie in [BA02], [BA00] und [BAC04] erläutert werden.

Roboter generieren Kommandos für ihre Aktuatoren in regelmäßigen Zeitintervallen. Hu-

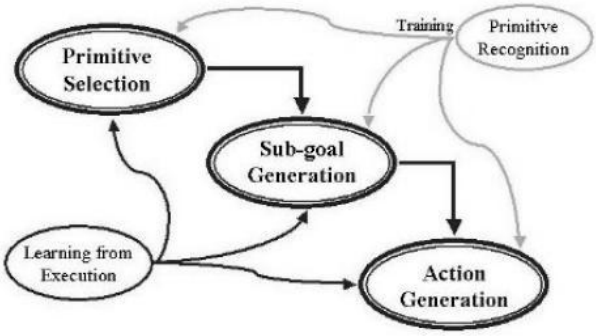


Abbildung 5.10: Framework für Lernen aus Beobachtung mittels Primitiven

manoide besitzen oft 30 oder mehr Freiheitsgrade. Lernen in diesem hochdimensionalen Raum ist sehr zeitaufwändig bis völlig unmöglich, und auch eine zufällige Suche kommt nicht in Frage. Um die Dimensionalität des Lernproblems in den Griff zu bekommen, wurden hier Primitiven verwendet. Unter Primitiven versteht man Teillösungen für eine Aufgabe, die sich zu einer Lösung für die komplette Aufgabe zusammensetzen lassen. Man braucht also einen Satz von Primitiven, der alle Aktionen beinhaltet, die zum Lösen der Aufgabe benötigt werden. Es gibt verschiedene Wege, diese Primitiven zu gewinnen. Man kann den Roboter automatisch Primitiven entweder durch Beobachten oder durch agieren in der Aufgabenumgebung generieren lassen, oder aber, wie auch in diesem Beispiel, die Primitiven mit Hilfe von Domänenwissen selber erstellen. Die Primitiven, die für das Air Hockey Spiel verwendet werden, sind:

- *Gerader Schuss*: Ein Spieler trifft den Puck so, dass er geradlinig in Richtung des gegnerischen Tors gleitet, ohne die Bande zu berühren.
- *Bandenschuss*: Ein Spieler trifft den Puck so, dass er zunächst an der Bande abprallt und sich dann in Richtung des gegnerischen Tors bewegt.
- *Tor verteidigen*: Ein Spieler bringt seinen Schläger so in Position, dass der Puck daran gehindert wird, in sein Tor zu gleiten.
- *Langsamer Puck*: Der Spieler trifft einen sich langsam bewegenden Puck, der sich in seiner Reichweite befindet.
- *Tue nichts*: Ein Spieler bringt seinen Schläger in Ruhelage, während der Puck sich in der gegnerischen Hälfte des Spielfeldes befindet.

Abbildung 5.10 liefert eine übersicht für das in diesem Beispiel verwendete Framework. Die beobachteten Daten müssen zunächst in die oben aufgelisteten Primitiven zerlegt werden. Dafür ist das "Primitive Recognition"-Modul verantwortlich. Für die Segmentierung benutzt es sogenannte kritische Ereignisse. Ein kritisches Ereignis tritt immer dann auf, wenn sich die Geschwindigkeit und die Richtung des Pucks schlagartig ändern. Die segmentierten

Daten liefern dann die Trainingsdaten für die anderen Module.

Basierend auf dem aktuellen Zustand und der Beobachtung zuvor ausgeführter Primitiven wählt das "Primitive Selection"-Modul den Typ der aktuellen Primitive aus. Dafür wird aus den beobachteten Daten eine Datenbank kreiert, welche die benutzte Primitive, die Umweltzustände vor Ausführung der Primitiven und die aus der Ausführung resultierenden Umweltzustände enthält. Anhand eines idealisierten physikalischen Modells wird berechnet, wo der Puck die Bande getroffen hätte, wenn der Gegner ihn nicht geblockt hätte. Durch eine einfache Nearest-Neighbor-Suche auf dieser Datenbank lässt sich der Zustand finden, der dem aktuellen Umweltzustand am nähesten kommt, und so die dazugehörige Primitive wählen. Eine deutliche Verbesserung der Performanz würde hier das K-Nearest-Neighbor-Verfahren bringen. Es wird - zumindest in der aktuellen Version des Frameworks - immer die Primitive des nähesten Datenpunktes gewählt.

Nachdem der Typ der Primitiven gewählt ist, können die Teilziele berechnet werden und daraus die Parameter der Primitive (wie z.B. der Winkel zwischen Schläger und Puck beim Schlagen, die gewünschte Geschwindigkeit und Richtung des Pucks nach dem Schlag). Da der näheste Datenpunk auch gleichzeitig die beobachtete Ausgabe der Demonstration enthält, kann diese Information als das gewünschte Teilziel gewählt werden. Wird wie oben schon erwähnt hier die K-Nearest-Neighbor-Suche angewandt, lässt sich das Verfahren robuster machen. Dazu wird aus den Ausgaben der k nähesten Datenpunkte das Teilziel anhand eines Locally-Weighted-Learning-Modells berechnet.

Programmiert wurde der Air Hockey Agent bis jetzt nur darauf, seinen Schläger mit gewünschter Geschwindigkeit an eine gewünschte Position zu bewegen. Um richtig Air Hockey spielen zu können, muss er allerdings wissen, wann er den Schläger wohin bewegen muss. Anhand der oben gewonnenen Parameter berechnet das "Action Generation"-Modul die benötigten Schlägerbewegungen. Hierfür wurden drei Ansätze ausprobiert. Der erste verwendet ein idealisiertes physikalisches Umweltmodell, anhand dessen die nötigen Schlägerbewegungen errechnet werden, um den Puck mit der gewünschten Geschwindigkeit in die gewünschte Richtung zu schießen. Dieser Ansatz ist stark abhängig von der Genauigkeit des Modells, funktioniert in dieser Anwendung aber sehr gut. Die beiden anderen Ansätze verwenden ein Lernverfahren mit Neuronalen Netzen bzw. mit einem Locally-Weighted-Regression-Modell. Im Gegensatz zum ersten Verfahren werden hier also die aufgezeichneten Daten verwendet um zu lernen, wie der menschliche Spieler den Schläger bewegt hat, um den Schuss durchzuführen. Das eintrainierte Neuronale Netz liefert die nötigen Informationen sehr zügig, das LWR-Modell hat den Nachteil, dass es bei zu großen Datenbanken oder zu langsamen Computern zu Problemen führen kann, da dieses Verfahren jeden Datenpunkt berücksichtigt.

Bis zu diesem Punkt hat das System maximal so gut zu spielen gelernt, wie der Mensch,

den es beobachtet hat. Allerdings wurden hierbei auch eventuelle Fehler eintrainiert, die der Demonstrator gemacht hat. Um die Performanz des Systems auch nach der Vorführung noch zu verbessern, wurde ein "Learning From Practise"-Modul eingeführt. Dieses beinhaltet Informationen zur Bewertung der Leistung des Agenten in Hinblick auf die Lösung der Aufgabe. Dieses Wissen wird dann benutzt, um die Datenbank für die anderen Module zu aktualisieren. Der virtuelle Air Hockey Spieler kann also seine eigene Vorführung beobachten und beim Spielen und üben weitere Daten sammeln.

5.4.2 Japanische Volkstänze

Shinichiro Nakaoka beschreibt in seiner Masterarbeit ([Nak03]) ein System, mit dem sich zuvor aufgezeichnete japanische Volkstänze von einem humanoiden Roboter vorführen lassen. Als Plattform wurde hierfür das HRP (Humanoid Robot Project) System verwendet. Die Tänze werden mit einem Motion-Capturing-System aufgezeichnet. Der Tänzer wird dafür mit optischen und magnetischen Markern versehen. Deren Position und Orientierung lässt sich so im 3-dimensionalen Raum verfolgen. Die Bewegungsdaten werden in einem Array aus Frames in zeitlicher Reihenfolge gespeichert, wobei ein Frame in Form eines Arrays aus Markerpositionen ist. Diese aufgenommenen Bewegungsdaten müssen in Bewegungsdaten für den Roboter gewandelt werden. Das liegt zum einen daran, dass Roboter nicht über Markerpositionen sondern über Gelenkwinkelstellungen und Winkelgeschwindigkeiten angesteuert werden, zum anderen aber auch daran, dass sich der Mensch und der Roboter im Körperaufbau unterscheiden. Darüber hinaus muss sichergestellt werden, dass der Roboter die Balance hält. Abbildung 5.11 zeigt den Konvertierungsprozess von den aufgenommenen Daten zu Daten für die Robotersteuerung. Dieses System werde ich im Weiteren genauer erklären.

Auch dieses System zerlegt die beobachtete Vorführung in eine Folge von Bewegungsprimitiven, also kleinstmögliche Einheiten der Choreographie. Eine Primitive kann sich durchaus mehrere Male im Tanz wiederholen. Nakaoka unterscheidet zwischen zwei Arten von Bewegungsprimitiven: Die erste ergibt sich aus deutlich ausgeprägten Posen, in denen der Tänzer passend zum Rhythmus kurzzeitig verweilt. Diese Posen werden hier "Schlüsselposen" genannt. Der übergang in eine bestimmte Schlüsselpose wird als Bewegungsprimitive aufgefasst. Diese Primitiven müssen nicht zwangsläufig vom ganzen Körper ausgeführt werden, es können auch Bewegungen einzelner Körperteile mit individuellem Timing sein. Synchronisierte Primitiven einzelner Körperteile lassen sich in einer Primitive zusammenfassen. Die andere Art von Primitiven ergibt sich aus der Interaktion zwischen dem Körper des Tänzers und der Umwelt, insbesondere zwischen den Beinen und dem Boden, wie z.B. bei Bewegungen wie Schritten, in die Hocke gehen, Drehen und Springen. Diese Bewegungen lassen sich als Funktionen gekennzeichnet durch Interaktionen betrachten, und unterliegen dadurch auch Einschränkungen, die sich beispielsweise durch dynamische Gesetze ergeben. Es wird also zwischen posenorientierten und funktionsorientierten Primitiven unterschieden, und eine Tanzbewegung setzt sich aus beiden Arten von Primitiven zusammen. Wie

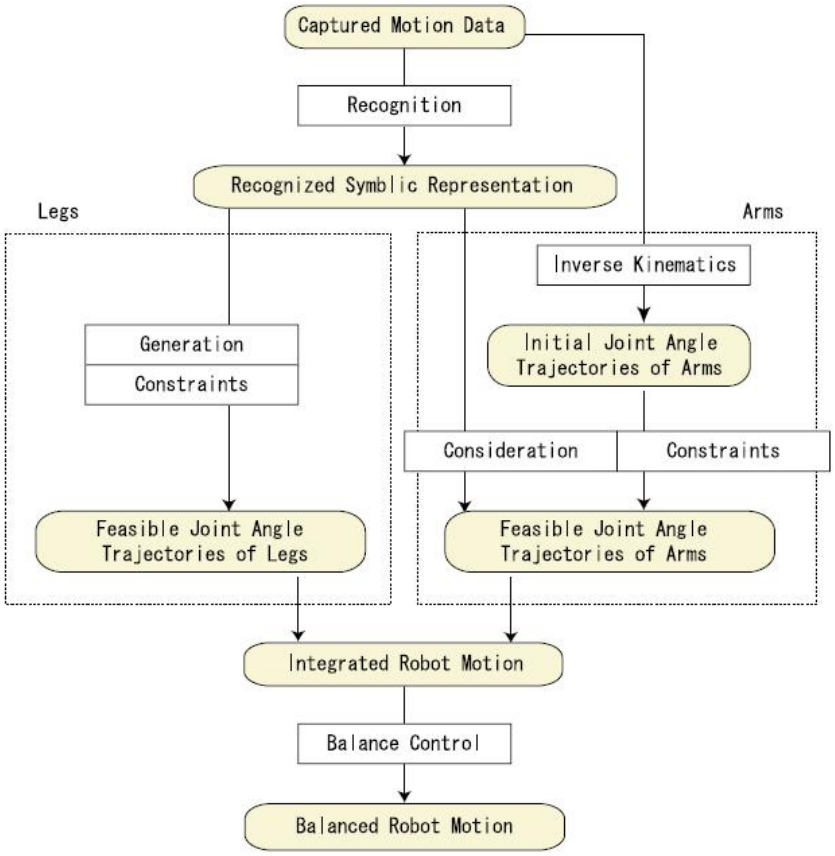


Abbildung 5.11: Übersicht über Konvertierungsprozess

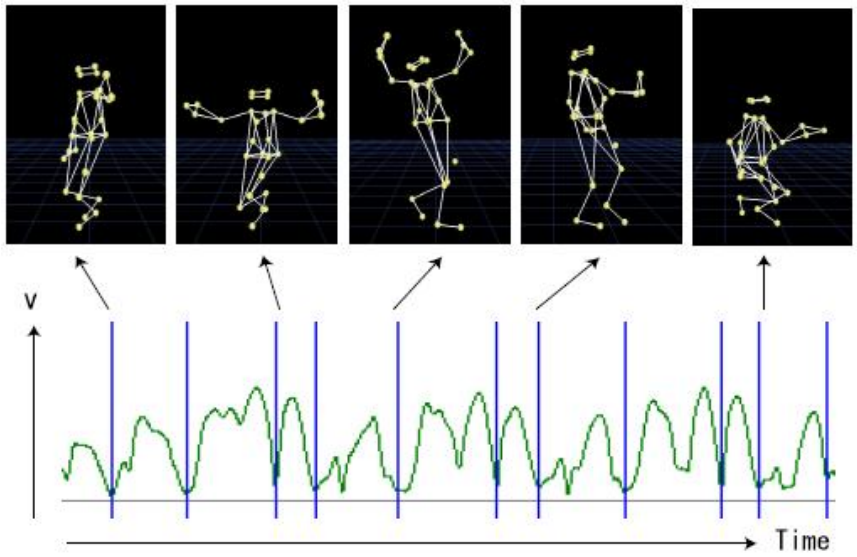


Abbildung 5.12: Geschwindigkeitsgraph einer Handbewegung und vorgenommene Segmentierung

auch schon in Abbildung 5.11 zu sehen ist, wird die Konvertierung für Bewegungen der Beine und Bewegungen des Oberkörpers bzw. der Arme getrennt behandelt. Die posenorientierten Primitiven werden für den Oberkörper verwendet, die funktionsorientierten für Beinbewegungen. Ich werde daher die weitere Betrachtung der Bewegungserkennung für Arm- und Beinbewegungen getrennt vornehmen.

Für die Segmentierung der Armbewegungen müssen zunächst die Schlüsselposen gefunden werden. Um den zu einer Schlüsselpose gehörenden Frame zu finden, sucht man nach Frames, bei denen die Geschwindigkeit einer Hand nahe Null ist. Abbildung 5.12 zeigt einen Auszug aus dem Geschwindigkeitsgraphen für eine Hand und die damit vorgenommene Segmenteinteilung. Die gefundenen Segmente werden die Basis für die Bewegungsprimitive. Als nächster Schritt müssen die Segmente als die jeweilige Primitive klassifiziert werden. Grundlage hierfür ist die Form der von der Hand zurückgelegten Trajektorie. Mit Hilfe des Dynamic-Programming-Matching-Distance-Algorithmus lassen sich zwei Segmente mit ähnlicher Bahn als eine Bewegungsprimitive klassifizieren. Nach der Klassifizierung der Primitiven wird versucht, nach Mustern in ihrer Abfolge zu suchen und evtl. sich wiederholende Folgen von Primitiven zu einer Einheit zusammenzufassen. Dafür wird der A-Priori-Algorithmus verwendet. Abbildung 5.13 zeigt ein Beispiel für eine solche Strukturierung. Darüber hinaus sucht dieses System nach synchronisierten Primitiven der rechten und der linken Hand. Treten solche Kombinationen wiederholt auf, werden sie zu zu einer Bewegungsprimitiven zusammengefasst, die aus Bewegungen beider Arme besteht. Gewonnen werden die posenbasierten Primitiven durch inverse Kinematik, die die Bewegung des Menschen auf den Roboter abbildet.

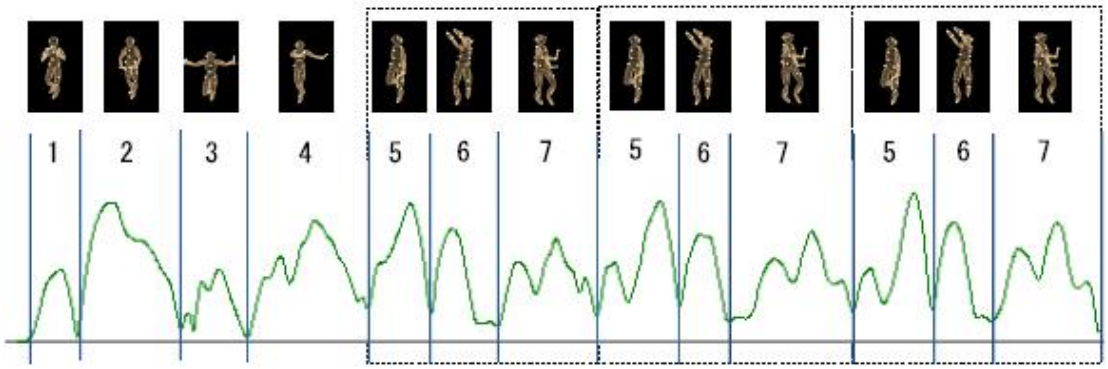


Abbildung 5.13: Strukturierung der Folge klassifizierter Primitiven

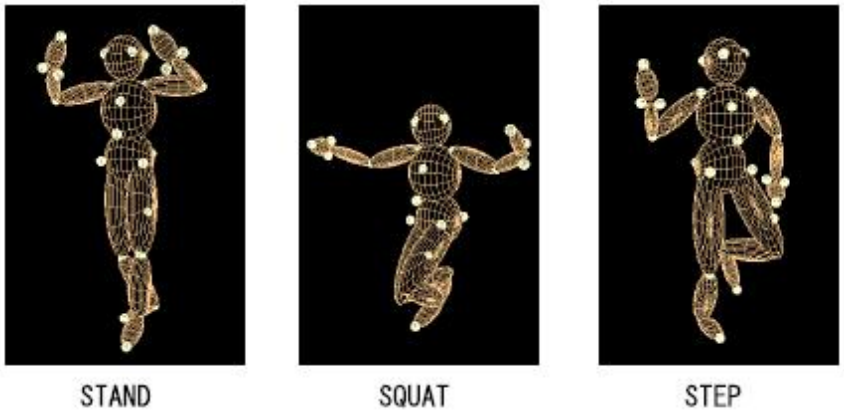


Abbildung 5.14: Bewegungsprimitiven für die Beine

Die Primitiven für die Beine dagegen können nicht direkt aus der Beobachtung gewonnen werden und müssen von Hand programmiert werden. Für die hier betrachteten japanischen Tänze (Jongara-Bushi und Aizu-Bandaisan) wurden drei Primitiven ausgemacht (siehe Abb. 5.14): STAND (Stehen), STEP (Schritt) und SQUAT (Hocken). Die STAND-Primitive repräsentiert Bewegungen, bei denen beide Beine Kontakt mit dem Boden haben und die Balance aufrechterhalten. Die beiden Parameter sind die Dauer der Standzeit und die Projektion des Masseschwerpunktes. Die STEP-Primitive steht für Bewegungen, bei denen ein Fuß angehoben wird, während der andere den Körper stützt. Um damit verschiedene Figuren des Tanzes nachbilden zu können, werden folgende Parameter benötigt: Die mittlere Zeit und die Gesamtzeit der Bewegung sowie Zustände zu diesen Zeitpunkten. Die mittlere Zeit ist der Zeitpunkt, bei dem sich der schwingende Fuß am höchsten Punkt befindet, die Gesamtzeit der Zeitpunkt, wenn der schwingende Fuß wieder auf dem Boden aufsetzt. Beide Zeiten werden vom Beginn der Bewegung ab gemessen. Als Zustände werden Position und Orientierung des schwingenden Fußes und zusätzlich die Orientierung der Hüfte zum Endzeitpunkt, jeweils in Relation zum Standfuß, benötigt. Die SQUAT-Primitive verkörpert Bewegungen, bei denen beide Kniegelenke gebeugt und wieder ge-

Abbildung 5.15: Geschwindigkeits-Graph eines Fußes

Abbildung 5.16: Geschwindigkeits-Graph der Hüfthöhe

streckt werden. Die Parameter sind die mittlere Zeit, d.h. der Zeitpunkt, in dem die Hüfte ihren tiefsten Punkt erreicht, und die Gesamtzeit. Zusätzlich wird als Zustand der mittleren Zeit noch die Höhe der Hüfte berücksichtigt. Die Segmente der STAND-Primitive korrespondieren mit den Frames, in denen die Geschwindigkeits-Graphen sowohl der Füße (Abb. 5.15) als auch der Hüfthöhe (Abb. 5.16) zur gleichen Zeit nahe Null bleiben. Im mittlere Frame eines Segments wird die Position des Masseschwerpunktes ermittelt und als Parameter gespeichert. Für die STEP-Primitive wird der Geschwindigkeits-Graph eines Fußes (Abb. 5.15) analysiert. Der Graph besteht größtenteils aus einer Folge glockenförmiger Kurven. Solch eine "Glocke" wird als Segment angesehen, wenn noch sichergestellt wurde, dass die Bewegung nicht zu klein für einen Schritt war. Jedes Segment wird repräsentiert durch den Startframe, den mittleren Frame und den Endframe, und anhand dieser Frames werden auch die Parameter ermittelt. Die Segmente der SQUAT-Primitive werden mit dem Geschwindigkeits-Graphen der Hüfte ermittelt. Eine Hock-Bewegung wird in diesem Graph durch eine konkave Kurve gefolgt von einer konvexen Kurve repräsentiert. Es muss also nach einem Paar solcher Kurven gesucht werden und zusätzlich kleine Schwingungen in der Hüfthöhe ausgefiltert werden.

Die erkannten Primitiven müssen jetzt in Bewegungsdaten für den Roboter umgewandelt werden. Dabei werden zunächst die Bewegungsdaten für Arme und Beine getrennt generiert, um dann unter Berücksichtigung der Balance zusammengefügt zu werden. Der interessierte Leser sei auf Kapitel 4 in [Nak03] verwiesen.

Nakaoka hat mit diesem System Bewegungen für den Humanoiden Roboter HRP-1S generiert und die Bewegungen getestet. Abbildung 5.17 zeigt den HRP-1S, wie er einen Jongara-Bushi tanzt. Numerisch lässt sich die Qualität der generierten Tanzbewegungen leider nur schwer erfassen, da sich die Körper von Roboter und Mensch zu sehr unterscheiden, aber wie auch in Abbildung 5.18 zu sehen ist, liefert das System eine gute Imitation der menschlichen Vorführung.

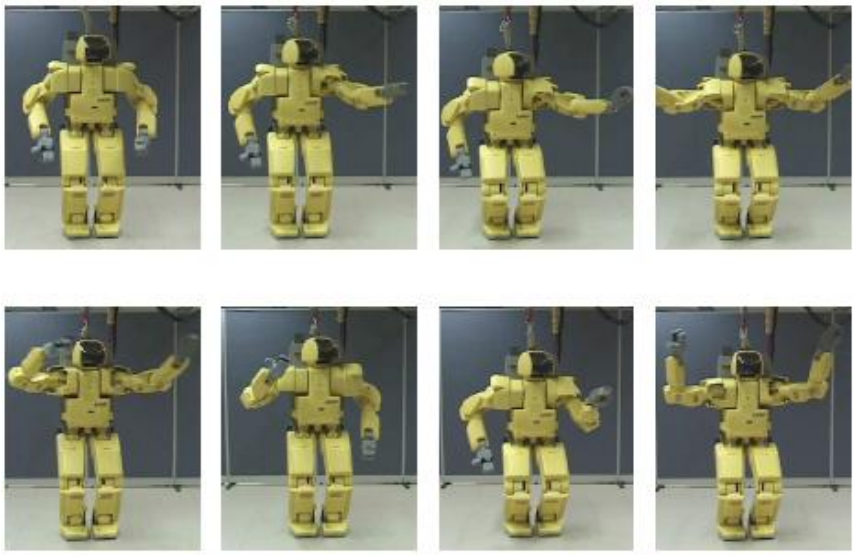


Abbildung 5.17: HRP-1S tanzt einen Jongara-Bushi

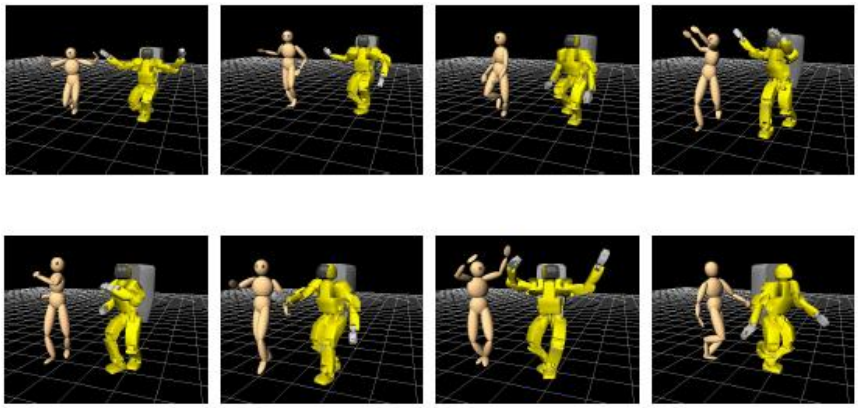


Abbildung 5.18: Vergleich der originalen Bewegung und der generierten Roboterbewegung

Abbildung 5.19: Leonardo: kosmetisch komplett (links), mit freigelegter Mechanik (Mitte) und das animierte Modell (rechts)

5.4.3 Gesichtsimitation

Breazeal et al. stellen in [BBG+05] ein System vor, das versucht, vorgeführte Mimiken in Anlehnung an den Lernprozess eines Kleinkindes zu imitieren. Als Experimentierplattform nutzen sie den humanoiden Roboter Leonardo (Abb. 5.19), von dem sie eigens eine Software-Version entwickelten. Er zeichnet sich unter Anderem durch ein ausdrucksstarkes Gesicht mit 24 Freiheitsgraden und ein aktives binokulares Kamerasystem aus. Das Framework lehnt sich an den 1997 von Meltzoff und Moore postulierten Active Intermodal Matching (AIM) Mechanismus an. Dieser soll angeboren sein und der frühen Imitation bei Neugeborenen zugrundeliegen. Einen überblick auf das entworfene Framework bietet Abbildung 5.20.

Das *Perception System* benutzt einen hierarchischen Mechanismus um Zustandsinformationen aus dem Sensorinput abzuleiten. In einer Baumstruktur werden atomare Erkennungseinheiten gespeichert, deren Aufgabe es ist, Merkmale aus den Rohsensordaten zu erkennen und zu extrahieren. Dabei befinden sich spezifischere Erkennungseinheiten im Baum näher an den Blättern. Für die Gesichtserkennung und die Merkmalsextraktion wurde die Software *Axiom ffT* der Firma Nevengineering verwendet. Eine dieser Erkennungseinheiten die Gesichtserkennungseinheit, die als Kindknoten Erkennungseinheiten für Augenbrauen, Augen, Nase und Mund hat. Es gibt noch Einheiten die erkennen, wenn sich die menschlichen Gesichtsmerkmale bewegen, und solche die erkennen, wenn das menschliche Gesicht auf Leonardos Bewegungen reagiert.

Das *Action System* ist verantwortlich für das Verhalten des Roboters. Individuelle Aktionsmuster werden in Form von sog. Aktions-Tupeln repräsentiert. Die Schlüsselkomponenten eines Aktions-Tupels sind die Aktion und ihr auslösender Kontext. Für dieses System werden zwei Aktionen benötigt: Die Motor-Babbel-Aktion und die Imitations-Aktion.

Das *Motor System* ist dafür zuständig, die Bewegungen vorzunehmen, die die vom Action System gewählten Aktionen durchführen. Motorbewegungen werden als Pfade durch einen direkt gewichteten Graphen repräsentiert, den sog. Posengrahp. Die Knoten im Graph sind

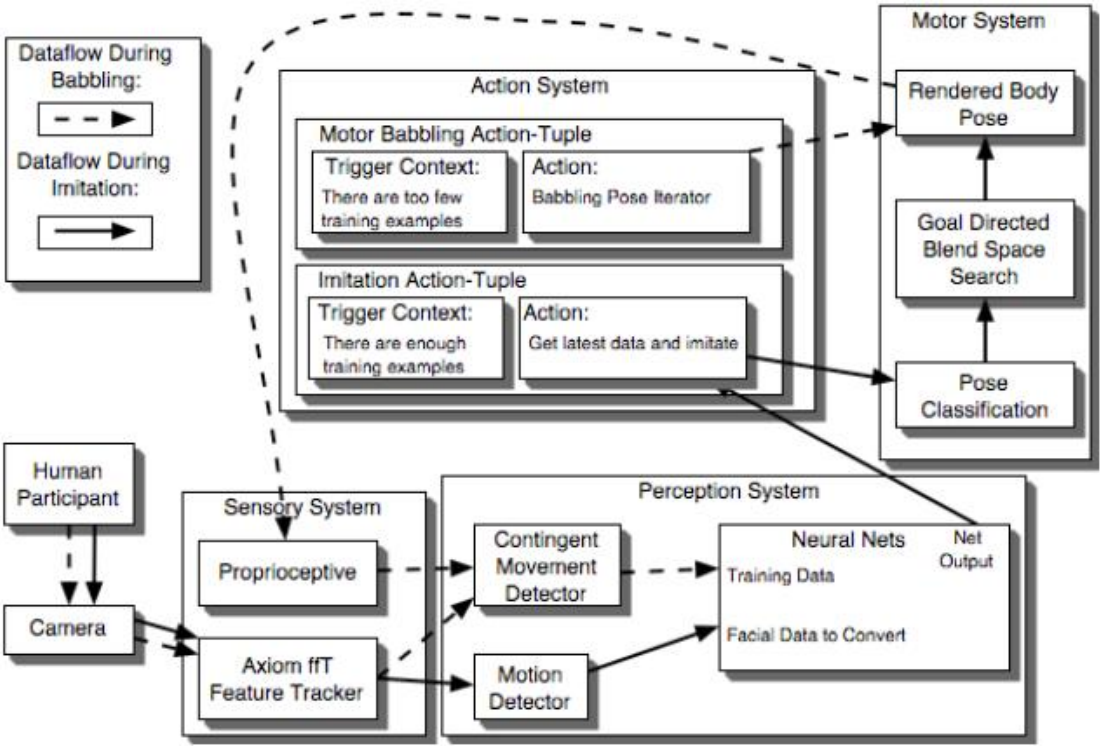


Abbildung 5.20: Leonardos Framework

die Posen, also eine bestimmte Gelenkwinkenkonfiguration. Leonardo ist mit einem kleinen Satz an Grundposen ausgestattet. Eine Verbindung zwischen zwei Posen steht für einen erlaubten übergang zwischen Gelenkwinkenkonfigurationen. Die Grundposen und die Pfade zwischen ihnen spannen den Raum aller möglichen Bewegungen auf, den Posenraum. Vollständige Bewegungstrajektorien exitstieren als Pfade durch diesen Raum. Außerdem erlaubt das Motorsystem, Posen zu gewichten und dann zu mischen. Die Grundposen die konvexe Hülle von Leonardos Posenraum. Um noch mehr Flexibilität in den Bewegungen zu erhalten, wurde das Motorsystem in drei Subsysteme unterteilt, eines für die Mundregion und jeweils eines für die rechte und die linke Augenregion.

Die Struktur einer imitativen Interaktion besteht aus zwei Phasen. In der ersten Phase imitiert ein menschlicher Teilnehmer Leonardos Gesichtsausdrücke. Wenn die *Axiom ffT* Software ein Gesicht im Blickfeld des Roboters erkennt, wird die Motor-Babbel-Aktion ausgeführt. Während diese aktiv ist, wählt Leonardo eine zufällige Pose aus dem Basissatz, übergibt diese an das Motorsystem und verweilt dort einen Augenblick. In dieser Zeit versucht der Mensch, Leos Gesichtsausdruck zu imitieren. Mit Hilfe des Babbelns kann der Roboter physikalisch seinen Posenraum erkunden und er lernt dabei, wie er die menschlichen Posen mit den seinen abgleicht. Für die intermodale Abbildung von seinen und den menschlichen Posen werden getrennte neuronale Netze für jede mit rechtem Auge, linken

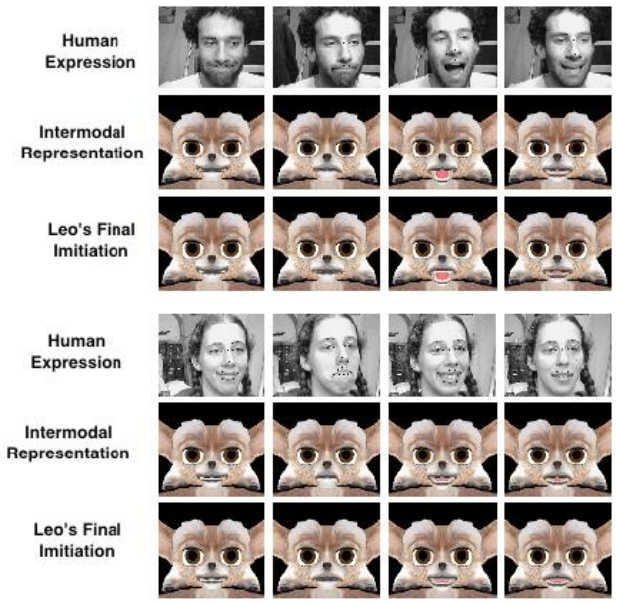


Abbildung 5.21: Leonardo imitiert zwei menschliche Teilnehmer. Die erste Reihe zeigt dabei immer das Kamerabild des menschlichen Gesichtsausdrucks, die zweite Reihe die intermodale Repräsentation des Ausdrucks, und die dritte Leos beste Approximation der intermodalen Repräsentation der menschlichen Pose nach der Suche.

Auge und Mund korrespondierende Gesichtsregion trainiert.

Nachdem Leo jetzt empfangene Gesichtsausdrücke im intermodalen Raum repräsentieren kann, beginnt er, den Menschen zu imitieren. Dafür sucht das Motorsystem nach einer Pose im Posengraph, das dem intermodalen äquivalent am nähesten kommt. Ausgehend von dieser Pose wird durch Mischen mit anderen Posen aus dem Posengraph nach einem Maximum in der übereinstimmung gesucht. Im Moment wird ein einfacher Hill-Climbing Algorithmus verwendet, um nach einem geeigneten Satz von Gewichten für die gemischten Posen zu suchen.

Die Implementierung wurde nur auf der simulierten Version von Leonardo getestet, da zu diesem Zeitpunkt das Silikongesicht des Roboters noch nicht montiert war. Wie man in Abbildung 5.21 sehen kann, fallen die Ergebnisse zufriedenstellend aus.

5.4.4 iPor

iPor (Interaktives Programmieren von Robotern) wurde von der Universität Karlsruhe entwickelt. Mit Hilfe dieses Systems wird anhand des Cranfield Benschmark beispielhaft das in den vorherigen Abschnitten dargestellte Programmieren durch Vormachen verdeutlicht.

Als Sensorik wird ein Datenhandschuh, ein magnetfeldbasierter Positionsmesser und ein Stereokamerakopf verwendet. Bei den Interaktionsformen wird sich auf die physische und ikonische Demonstration mit Kommentierung beschränkt. Die abschließende Programmrepräsentation geschieht über symbolische Operatoren (Elementar- oder Makrooperatoren). Der Ablauf des ganzen Prozesses ist in Abbildung 5.22 dargestellt.

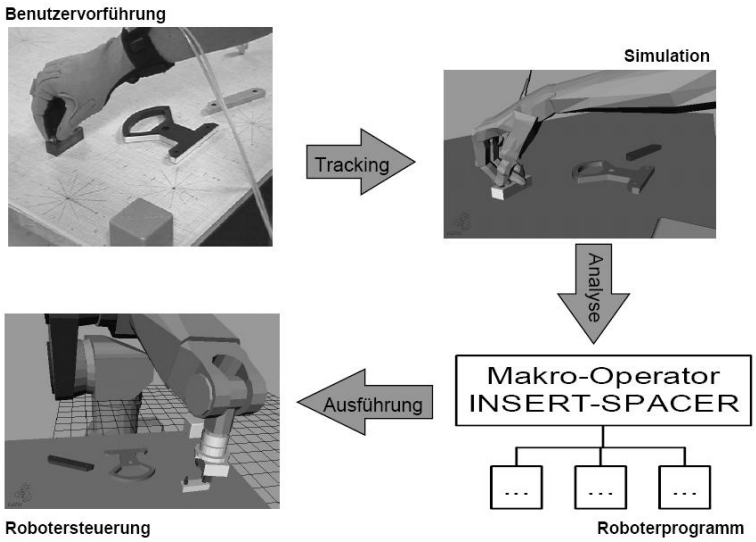


Abbildung 5.22: Ablauf des Programmierens durch Vormachen (IPoR)

Aus den vom Benutzer vorgeführten Aktionen müssen Schlüsselpunkte identifiziert werden, um elementare Einheiten (pick and place) zu bilden. Hierzu stellt man die Frage: „Was wurde gegriffen?“ und erstellt eine Objekthypothese, welche aus dem Umweltmodell abgeleitet wird. Sie könnte lauten: „Wähle das Objekt, welches der Hand am nächsten liegt!“.

Die nächste Frage, die es zu klären gilt, ist: „Wie wurde gegriffen?“ Um einzelne Griffe zu identifizieren, müssen zuerst die Fingerbeugungs- und Spreizwinkel (Abb. 5.23) sowie die Stellung des Handgelenks bestimmt werden. Insgesamt sind dies 20 zu bestimmende Parameter, die man durch den Datenhandschuh (in diesem Fall mit Dehnmessstreifen) erhält.

Prinzipiell gibt es beliebig viele Griffarten und der Mensch wählt sie unbewußt aus. Trotzdem ist jeder Griff abhängig von Parametern wie Objektform, Objektgröße und seiner Aufgabe. Durch diese Angaben kann man die Griffhierachie nach Cutkosky (Abb. 5.24) benutzen. Die Klassifikation geschieht über neuronale Netze mittels der mit dem Datenhandschuh erfaßten Werte der Fingergelenkstellungen. Die so erfassten Werte werden in einem Schichtennetz (unter Berücksichtigung der Hierarchieebenen im Cutkosky Baum) weiterverarbeitet. Jede der 10 Schichten wird mit einem eigenen Netz erlernt. Gute Resul-

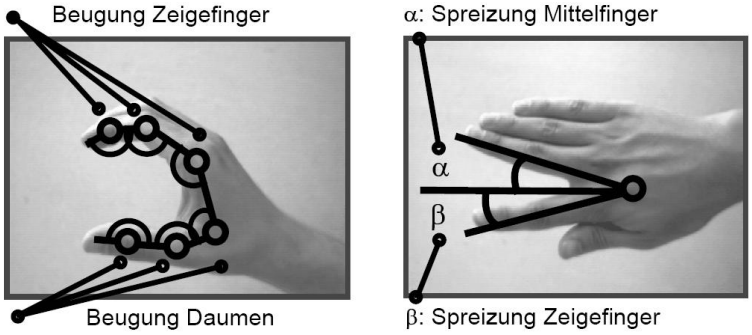


Abbildung 5.23: messen der Fingerbeugungs- und Spreizwinkel mit einem Datenhandschuh

tate erzielt man mit lokalen Aktivierungsfunktionen wie der radial basis funktion (n-dim Gauskurven) und rectangular basis (n-dim Trapezoide). Jedes Neuron erhält einen eigenen Referenzvektor und die Erstellung geschieht durch automatisches clustering. So erreicht man im Durchschnitt eine Erkennungsleistung von 92% bei singel-user Systemen, 89% bei multi-user mit bekannten Benutzern und noch 82% bei multi-user Systemen mit unbekannten Benutzern.

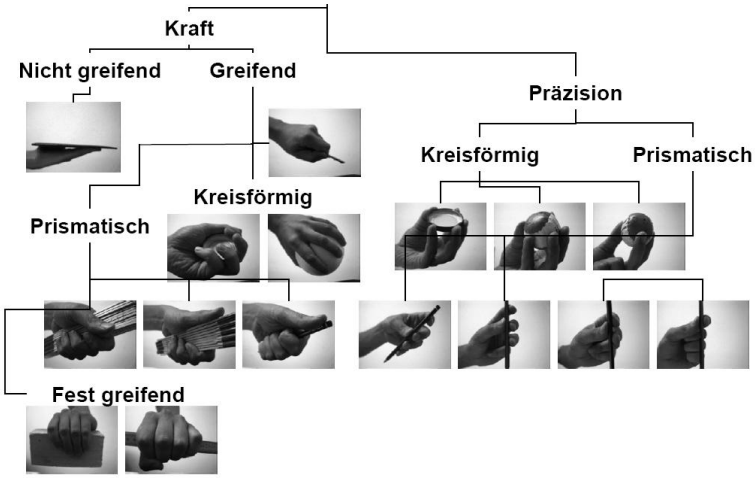
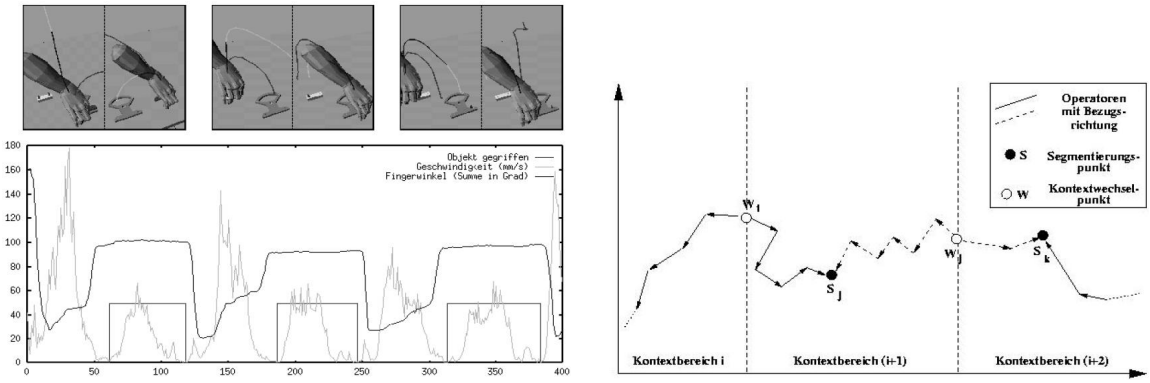


Abbildung 5.24: messen der Fingerbeugungs- und Spreizwinkel mit einem Datenhandschuh

Wann letztendlich ein Griff stattgefunden hat, wird über die Summe der Fingergelenkwinkel und die Geschwindigkeit der Hand geschlossen. Die Identifikation von Griffen ist in Abbildung 5.25 (a) dargestellt.

Nach der Grifferkennung wird mit der Segmentierung und Generalisierung begonnen. Der erste Schritt beim Segmentieren ist das Aufteilen der Vorführung in Kontexte. Die einzelnen Kontextbereiche mit den Kontextwechselpunkten sind in Abbildung 5.25 (b) zu sehen. Die Auswahl der Kontextzugehörigkeit und die damit verbundenen Segmentierungspunkte wer-



(a) (b)

Abbildung 5.25: Beispiel IPoR (a): Identifikation von Griffen (b): Kontextbereiche der Vorführung

den über eine Heuristik wie die maximale Distanz zwischen Operatoren (greifen, loslassen, fügen, ...) bestimmt. Ein Segmentierungspunkt beschreibt zum Beispiel eine „pick“ Elementaroperation. Der Kontextwechselpunkt liegt an der Stelle, wo der Wechsel zu „place“ Operation erfolgt. Diese wird auch durch einen Segmentierungspunkt makiert.

Weiter geht es mit der Interpretation der Transportphasen. Hierbei ergeben sich drei Möglichkeiten:

- Eine Möglichkeit ist die Linearisierung der Trajektorien mit „Iterative-Endpoint-Fit“.
- Es können auch Hypothesen bezüglich der Bewegungselementaroperationen benutzt werden. Diese basieren auf den Winkeln zwischen den Segmenten, der Abweichung mehrerer Segmente, der Segmentlänge und dynamischer Schwellwertadaption.
- Oder man beschränkt sich auf die elementaren Bewegungsoperatoren (lineare, splinefömige, freie und geplante Bewegungen).

Leider gibt es Probleme durch verrauschte und ungenaue Sensorwerte (Datenhandschuh +/-2cm, Stereokamera +/-1cm) bei der Verarbeitung der Trajektorien. Die Lösung besteht in der Korrektur relevanter Ausschnitte. Dies kann zum einen interaktiv durch Kommentierung oder automatisch mit Hintergrundwissen geschehen.

Danach wird eine Repräsentation der Semantik durch Relationen erstellt. Dies ist in Abbildung 5.26 zu sehen. Es sind die Relationen vor und nach der Anwendung des Operators zu bestimmen:

1. relevante Objekte:
   $O = \{o | (o\ \ manipulierendes\ \ Objekt) \vee (o\ \ Referenzobjekt)\}$

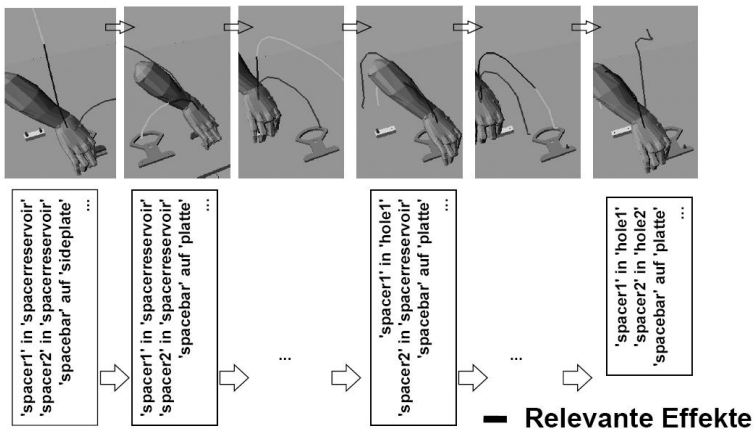


Abbildung 5.26: semantische Analyse und Interpretation

2. initiale Effektmenge:
   $R = \{r_l(o_i, o_k) | (r_l(o_i, o_k) \in Sta_{initial}) \wedge (r_l(o_i, o_k) \notin Sta_{final})$
   $\vee (r_l(o_i, o_k) \notin Sta_{initial}) \wedge (r_l(o_i, o_k) \in Sta_{final}), \text{mit} o_i, o_k \in O, i \neq k, r_l \in Rel\}$

3. Informationsbasierte Filterung von R:
   $R' = R \left\{ r_l | l(r_l) < l_{Schwelle},\ mit\ \ l(r_l) = \log(\frac{1}{p(r_l)})\ \ und\ \ p(r_l) = \frac{\#r_i}{\#R} \right\}$

Relationen werden vom Benutzer aus R' ausgewählt und kommentiert. In Abbildung 5.26 ist zu sehen, wie sich die Relationen ändern. So wird im ersten zum zweiten Schritt aus `'spacer1' in 'spacerreservoir'`, `'spacer1' in 'spacerreservoir'` und `'spacebar' auf 'sideplate'` nach der Elementaroperation `'spacer1' in 'spacerreservoir'`, `'spacer1' in 'spacerreservoir'` und `'spacebar' auf 'platte'`

Dann folgt die Codierung und Optimierung der Vorführung. Hierzu werden Makrokontexte (montiere Stift und Pendel) bestimmt und die Ausführungskontexte generalisiert. Auch Verzweigungs- und Redundanzanalyse gehören in diesen Abschnitt. Anschließend folgt die Löschung nicht relevanter Operatoren und Segmente. Auch eine Substitution der Objekte durch Variablen findet jetzt statt. Die Parameter der Makroperatioen (automatisch, interaktiv) werden bestimmt.

Nach der Abbildung der Benutzerdemonstration auf das Zielsystem erfolgt die Instanzierung des Makrooperators auf die Umwelt. Weiterhin werden die einzelnen Operatoren (Kinematik, Kraft- und Positionsreglung) im Zielsystem eingebracht. Hierbei kann Hintergrundwissen über das Zielsystem einfießen. Bei der Ausführung ist dann auf die Zuordnung der Regelgrößen zu den Elementaroperatoren zu achten.

5.5 Vergleich

Im Folgenden Abschnitt wird versucht, die Gemeinsamkeiten und Unterschiede der oben vorgestellten Beispiele herauszustellen.

Die ersten drei vorgestellten Systeme befassen sich mit dem Erlernen motorischer Fähigkeiten aus Benutzerdemonstrationen, das letzte mit dem Erlernen komplexerer Handlungsketten. Die Eingabedaten werden dafür unterschiedlich vorverarbeitet. Beim Air Hockey sind die Trajektorien der Schläger und des Pucks jeweils relativ zum Spieltisch von Interesse. Diese werden durch ein visuelles Tracking-System gewonnen. Das System zum Erlernen japanischer Volkstänze benutzt eine Motion-Capturing-Software, um die Positionen der visuellen und magnetischen Marker zu verfolgen. Leonardo erhält seine Daten von der Software *Axiom ffT*, die aus der visuellen Eingabe bereits Merkmalsvektoren extrahiert. iPor verwendet ein dediziertes System zum Erfassen der Handbewegungen, bestehend aus Datenhandschuhen und magnetischen Positionssensoren.

Die ersten beiden Beispiele segmentieren und generieren Bewegungen aus Bewegungsprimitiven. Die Primitiven in Beispiel 3.1 sowie im iPor-System zum Erlernen von Handlungsketten wurden alle von Hand vordefiniert, für die Segmentierung wurden kritische Ereignisse verwendet. Beispiel 3.2 unterscheidet Primitiven für den Oberkörper und für die Beine. Erstere werden durch inverse Kinematik von der Software generiert und anhand von Schlüsselposen segmentiert. Die Primitiven für die Beine lassen sich nicht automatisch erstellen und werden daher manuell programmiert. Die Segmentierung erfolgt anhand der Geschwindigkeitsgraphen für die Füße und die Hüfthöhe. Das System zum Imitieren von Mimiken erkundet seinen Posenraum in der ersten Phase der imitativen Imitation auf Basis weniger vordefinierter Grundposen mit Hilfe des Motor-Babbelns automatisch, und gleicht diese gleichzeitig mit den menschlichen Posen ab.

Alle vier Beispiele sind mit einer Simulationsumgebung ausgestattet, in der sich die erlernten Verfahren validieren lassen. Die Software-Variante für das Air Hockey Spiel bildet die Hardware-Version allerdings nicht ganz genau ab, da ihr reale Umweltbedingungen fehlen. Nichts desto trotz lässt sich das Erlernte mit dem "Learning From Practice"-Modul in der Software-Version noch verbessern.

5.6 Zusammenfassung und Ausblick

Angefangen mit den veränderten Anforderungen in der Robotik wurden neue Möglichkeiten des Programmierens durch Vormachen betrachtet. Auch eine Einteilung der verschiedenen interaktiven Programmiervarianten (Komplexität der Aufgabe, Art der Demonstration, Repräsentation von Handlung und Umwelt, Technik der Trajektorien) wurde vorgestellt. Nach der Klassifikation der PdV wurden die einzelnen Prozesskomponenten (Benutzer, Interaktionsform, Programmiersystem, Manipulator) mit ihren Abhängigkeiten unterein-

ander angesprochen. Zuletzt hat mit dem IPoR ein Beispiel das Beschiebene verdeutlicht und es wurde ein Anwendungsszenario skiziert.

Damit ist aber noch nicht das Ende der Programmierung durch Vormachen erreicht. In der Zukunft steht die Erweiterung von bereits vorhandenen Aktionstypen. In Arbeit ist auch die Integration taktiler Sensoren und eine zweiarmige Vorführung. Schön wären auch weitere Interaktionsmodi für den Benutzer. Es ist zum Beispiel eine sprachliche Kommentierung denkbar oder gestengesteuerte Kommandos. Die Integration von Verfahren zur Skill-Akquisition zur Onlineerzeugung elementarer Operatoren ist in Zukunft denkbar. Weiterhin sind auch Anwendungen auf anthropomorphen Zweiarmrobotern in der Entwicklung.

Kapitel 6

Aktionsplanung

6.1 Einleitung

Die allgemeine Tendenz der Roboterprogrammierung läuft darauf hinaus, dass Aufgaben für einen Roboter nicht mehr explizit (Wie ist etwas zu tun?, siehe Kapitel 3: Roboterorientierte Programmierung) formuliert werden sollten, sondern nur noch durch Angabe der zu lösenden Aufgabe (Was ist zu tun?, siehe Kapitel 4: Aufgabenorientierte Programmierung). Aus dieser muss der Roboter selbst in der Lage sein, unter Berücksichtigung seiner Umwelt, die nötigen Schritte zur Lösung der Aufgabe abzuleiten (siehe auch Abb. 4.29). Der Schritt von einer Aufgabenspezifikation zu einem Plan von elementaren Einzelaktionen wird Aktionsplanung genannt. Diese wird in diesem Kapitel näher vorgestellt.

6.1.1 Definition von Planung

In [SB96] wird Planung wie folgt definiert: *,,Planen heißt, zu einer definierten Aufgabe deren Durchführung zu bestimmen“*. Eine spezifischere, mehr in Richtung der Aktionsplanung gehende Definition findet sich in [RN03b]: *,,The task of coming up with a sequence of actions that will achieve a goal is called planning“*.

Bei der Aktionsplanung wird ein bekannter Anfangszustand angenommen und ein Zielzustand vorgegeben. Gesucht ist nun eine Folge von Aktionen, die den Anfangszustand in den Zielzustand überführen. Dabei wird hier davon ausgegangen, dass die Konsequenzen von Handlungen deterministisch sind. Ebenfalls wird die Umwelt vollständig, ohne Unsicherheiten oder Messrauschen wahrgenommen. Alle Ereignisse werden vom Roboter initiiert. In den erzeugten Plänen wird es daher keine Verzweigungen geben.

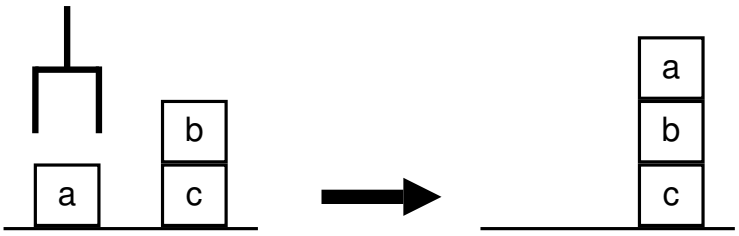


Abbildung 6.1: Anfangs- und Zielzustand in der Blockwelt

6.1.2 Überblick

In Kapitel 6.2 wird mit dem Situationskalkül ein rein auf Prädikatenlogik basierender Ansatz zur Planung beschrieben. Dort wird auch das Frame Problem behandelt. In Kapitel 6.3 wird auf Sprachen zur Darstellung von Planungsproblemen eingegangen. Dazu wird die Sprache des Planers STRIPS beschrieben sowie die ihr zugrunde liegende STRIPS Assumption eingeführt. Als Erweiterung zu STRIPS wird die Sprache ADL vorgestellt. In Kapitel 6.4 wird die Planung als ein Suchproblem in einem Zustandsraum betrachtet. Einige wichtige Suchalgorithmen werden kurz vorgestellt. Kapitel 6.5 beschäftigt sich mit linearer Planung, insbesondere mit dem STRIPS Planer und der Sussman Anomalie. Partial Order Planning wird in Kapitel 6.6 behandelt. In Kapitel 6.7 wird ein hierarchisches Planungsverfahren vorgestellt. Abschließend werden in Kapitel 6.8 die Inhalte zusammengefasst und ein kurzer Überblick über weitere, nicht behandelte Verfahren gegeben.

6.1.3 Die Blockwelt

Zur Erklärung der verschiedenen Aktionsplanungsverfahren wird die auch in der Literatur häufig herangezogene Blockwelt[1] als Beispielszenario verwendet. In dieser stehen quadratische Blöcke auf einem Tisch. Sie können durch einen Greifer gegriffen und bewegt werden. Die Möglichkeiten sind stark begrenzt: Blöcke können auf dem Tisch stehen oder auf einem anderen Block. Es können nicht mehrere Blöcke (versetzt) auf demselben Block stehen, ein Stapel von mehreren Blöcken ist allerdings möglich. Der Greifer kann nur einen Block auf einmal aufnehmen, und das auch nur, wenn auf diesem kein weiterer Block steht. In dieser Definition der Blockwelt wird der Tisch als hinreichend groß angenommen, d. h. es ist immer genug Platz, um einen Block auf dem Tisch abzulegen. Die Plätze auf dem Tisch werden als gleichwertig angenommen, es spielt also keine Rolle, wo ein Block auf dem Tisch liegt.

Schön an dieser Beispielwelt ist, dass sich Szenarien gut visualisieren lassen. Bilder im Stile von Abb. 6.1 werden die Beispiele in dieser Arbeit veranschaulichen.

[1] In der englischsprachigen Literatur „blocks-world“

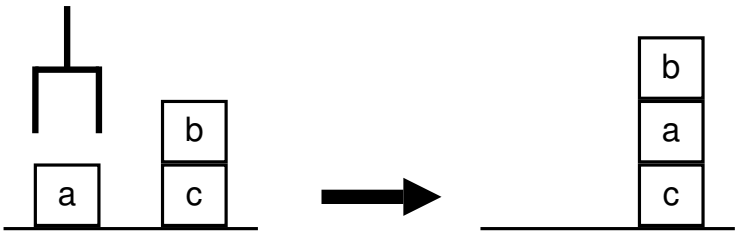


Abbildung 6.2: Aufgabe in der Blockwelt

6.2 Planen als Logik: Situationskalkül

Das Situationskalkül[2] [McC63] benutzt ausschließlich die Prädikatenlogik 1. Ordnung zur Planung (siehe auch Abschnitt 4.2.2.4). Sämtliche Annahmen, die von der Welt gemacht werden, müssen daher durch Axiome formuliert werden. Die Axiome werden nun anhand des Blockweltbeispiels in Abb. 6.2 erläutert.

Aktionen werden durch Funktionen dargestellt. Die einzige Aktion in der Blockwelt ist das Bewegen eines Blocks: $Move(block, to)$.

Situationen sind Zustände der Umwelt. Sie werden durch die Anwendung von Aktionen auf den Anfangszustand dargestellt. Dieser wird als s_0 bezeichnet. Die Funktion $Result(Action)$ liefert den Folgezustand einer Aktion. Damit lassen sich die Zustände in Abb. 6.3(a) bis 6.3(d) darstellen als:

$$\begin{aligned} s_1 &= Result(Move(b, table), s_0) \\ s_2 &= Result(Move(a, c), s_1) = Result(Move(a, c), Result(Move(b, table), s_0)) \\ s_3 &= Result(Move(b, a), s_2) = Result(\ldots) \end{aligned}$$

Die Beschreibung der Situation enthält also den Plan, wie sie erreicht werden kann. Aufgabe des Planers muss es daher sein, die Beschreibung der Zielsituation zu finden. Dazu muss noch beschrieben werden, wann eine Situation aus einer anderen hervorgehen kann.

Situationsabhängige Attribute werden im Situationskalkül *Fluents* genannt. Die Attribute selbst werden durch Funktionen oder Variablen dargestellt. Ihre Auswertung

[2]engl.: Situation Calculus

wird durch das Prädikat $holds(fluent, situation)$ vorgenommen. Im Anfangszustand des Blockweltbeispiels in Abb. 6.3(a) gilt beispielsweise:

$$\begin{aligned} Holds(On(a, table), s_0) \\ Holds(On(c, table), s_0) \\ Holds(On(b, c), s_0) \\ Holds(Clear(a), s_0) \\ Holds(Clear(b), s_0) \\ Holds(Clear(table), s_0) \end{aligned}$$

$On(b, c)$ bedeutet, dass b auf c liegt. $Clear$ gibt an, dass auf dem Objekt ein weiteres Objekt abgelegt werden kann. Der Tisch wird immer als frei angenommen.

Situationsunabhängige Attribute können einfach als Prädikate formuliert werden, z. B. $IsBlock(a), IsTable(table)$

Vorbedingungen von Aktionen Zur Beschreibung der Ausführbarkeit wird das Prädikat $Poss(action, s)$ eingeführt. Sein Name leitet sich vom englischen ,,possible“ ab. Es besagt, dass die Aktion $action$ in der Situation s ausgeführt werden kann. Die Vorbedingungen für eine Aktion werden als Axiome formuliert: $Vorbedingungen \Longrightarrow Poss(action, s)$. In der Blockwelt sieht dies für $Move$ wie folgt aus:

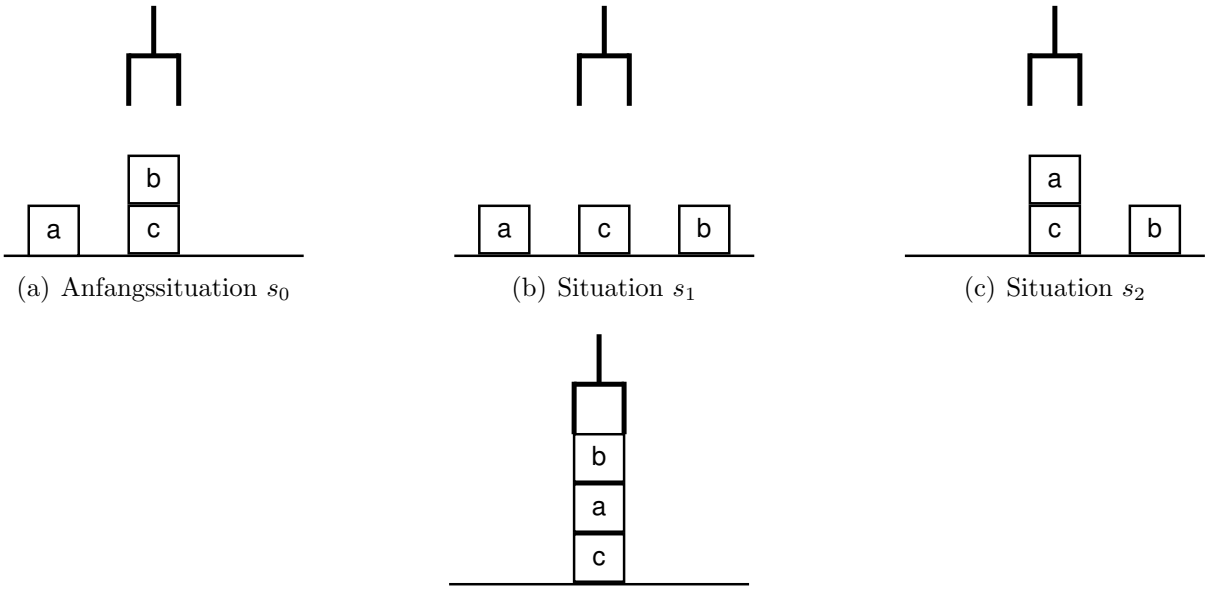
$$\begin{aligned} \forall x, y, s : \neg IsTable(x) \wedge holds(Clear(x), s) \wedge holds(Clear(y, s)) \wedge x \neq y \\ \Longrightarrow Poss(Move(x, y), s) \end{aligned}$$

Ergebnisse von Aktionen Als letzter Punkt müssen noch die Konsequenzen einer Aktion auf die folgende Situation beschrieben werden. Dies geschieht durch Axiome in der Form $Poss(action, s) \Longrightarrow holds(\text{fluents in neuer Situation}, result(action, s))$. Es muss für jede Aktion der gesamte Folgezustand spezifiziert werden, insbesondere müssen alle Fluents angegeben werden, auch jene, welche die Aktion nicht verändert. In der Blockwelt ergeben sich für die Aktion $Move$ folgende Axiome für die Effekte der Aktion:

$$\begin{aligned} \forall x, y, s : Poss(Move(x, y), s) \Longrightarrow \\ holds(On(x, y), result(Move(x, y), s)) \end{aligned}$$

$$\begin{aligned} \forall x, y, s : Poss(Move(x, y), s) \wedge y \neq Table \Longrightarrow \\ \neg holds(Clear(y), result(Move(x, y), s)) \end{aligned}$$

$$\begin{aligned} \forall x, y, z, s : Poss(Move(x, y), s) \wedge holds(On(x, z), s) \Longrightarrow \\ \neg holds(On(x, z), result(Move(x, y), s)) \wedge holds(Clear(z), result(Move(x, y), s)) \end{aligned}$$



(a) Anfangssituation s_0 (b) Situation s_1 (c) Situation s_2

(d) Zielsituation s_3

Abbildung 6.3: Lösung zum gegebenen Problem aus Abb. 6.2

Weitere Axiome werden für die unveränderten Fluents benötigt:

$$\forall u, v, x, y, s : Poss(Move(x, y), s) \wedge x \neq u \wedge y \neq v \Longrightarrow$$
$$holds(On(u, v), result(Move(x, y), s)) \Longleftrightarrow holds(On(u, v), s)$$

$$\forall u, x, y, s : Poss(Move(x, y), s) \wedge y \neq u \wedge \neg holds(On(x, u), s) \Longrightarrow$$
$$holds(Clear(u), result(Move(x, y), s)) \Longleftrightarrow holds(Clear(u), s)$$

$$\forall x, y, s : Poss(Move(x, y), s) \Longrightarrow$$
$$holds(Clear(Table), result(Move(x, y), s))$$

Diese Axiome werden Frameaxiome genannt.

Diese Definitionen sind noch nicht vollständig. Beispielsweise ist die Tatsache, dass keine zwei Blöcke an derselben Stelle stehen können ist nicht modelliert. Genauso wenig wie der Zusammenhang zwischen $Clear(x)$ und $On(y, x)$. Auf einem freien Block kann es weiteren Block y geben, der auf diesem steht.. Werden diese Zusammenhänge noch modelliert, kann ein Beweiser eine Suche nach einem Plan vornehmen. Dazu müssen noch die Fluents des Zielzustands als zu beweisender Satz angegeben werden:

$$\exists s : holds(On(a, c), s) \wedge holds(On(b, a), s)$$

Ein Beweiser[3], der s liefert, liefert damit auch die Lösung für das Planungsproblem.

Das Verfahren ist sehr mächtig, schließlich stehen zur Modellierung die gesamten Möglichkeiten der Prädikatenlogik zur Verfügung. Dies hat jedoch zur Konsequenz, das die Suche für den Beweiser äußerst aufwendig werden kann. Dies hängt nicht nur vom Umfang der Axiome ab, sondern auch vom Beweiser und wie die Axiome formuliert sind. Problematisch ist die Tatsache, dass die Formulierung der Axiome äußerst unintuitiv und damit fehleranfällig ist.

Bereits erwähnt wurde, dass für jede Aktion auch die Fluents spezifiziert werden müssen, die unverändert bleiben. Schon im einfachen Beispiel der Blockwelt war zu erkennen, dass dies im Normalfall die meisten Fluents sind, da Aktionen meist nur wenige Attribute der Situation verändern. Für jede Aktion und jedes unveränderte Fluent wird also ein Axiom benötigt, um einen eigentlich trivialen Zusammenhang zu beschreiben. Dieses Problem wird in der Literatur als *Frame Problem* bezeichnet [MH69].

6.3 Darstellung des Planungsraums

Um das Frame Problem zu umgehen, trennt man die Suche nach erfüllten Attributen in einer Zustandsbeschreibung und die Suche zwischen den Zuständen eines Weltmodells. Bei der Suche in den Zuständen kann weiter die Prädikatenlogik angewandt werden, um die Folgen von Aktionen zu bestimmen. Für die Transformation zwischen den Zuständen werden dagegen aus den Aktionen Operatoren bestimmt [FN71]. Dadurch erreicht man eine einfachere Darstellung der Folgen einer Aktion, da es nun möglich ist, dass Operatoren Eigenschaften implizit unverändert lassen. Dies vereinfacht das Erstellen der Axiome, die das Weltmodell beschreiben und beschleunigt ebenso die Suche in einem Zustand der Welt. Weiter ist es nun auch möglich, die Suche zwischen den Zuständen, also die Erzeugung der Aktionssequenz, durch Einbringen speziellen Wissens über das Problem zu beschleunigen.

Um Operatoren zu ermöglichen, welche die Attribute der Welt implizit unverändert lassen, wird ein Operator wie folgt aus drei Teilen definiert:

Vorbedingung[4] beschreibt wie gehabt die Bedingungen, die in einem Zustand erfüllt sein müssen, damit die Aktion ausgeführt werden kann.

Add-Liste Attribute, die dem neuen Zustand durch das Ausführen der Aktion hinzugefügt werden.

Delete-Liste Attribute, die aus dem neuen Zustand durch das Ausführen der Aktion negiert werden.

Dabei gilt die *STRIPS*[5]*-Assumption*, die besagt, dass Attribute, die in der Definition des Operators nicht erwähnt werden, unverändert bleiben.

[3] engl: theorem prover

[5] STanford Research Institute Problem Solver

Nach dieser Feststellung benötigt man eine Sprache, in der man unter Berücksichtigung der gewonnenen Erkenntnisse Planungsprobleme beschreiben kann. Dazu wird im folgenden die Sprache des STRIPS Planers vorgestellt. Im Anschluss daran wird noch kurz auf eine Erweiterung zu STRIPS eingegangen, die Sprache ADL[6].

6.3.1 STRIPS Repräsentation

In [FN71] wird der STRIPS Planer vorgestellt. Dazu wird eine, in der Beschreibung der Zustände vereinfachte Variation der Sprache aus [RN03b] verwendet. In diesem Abschnitt wird nur die Sprache zu STRIPS vorgestellt, der STRIPS Planer wird in Kap. 6.5.1 erläutert.

Zur Beschreibung des Planungsproblems muss ein Anfangszustand und ein Zielzustand sowie die Beschreibung der Operatoren angegeben werden. Der Anfangszustand wird wie folgt notiert:

$$Init(P(a,b) \wedge Q(c,d) \wedge \ldots)$$

Dabei sind nur positive, konjunktiv verknüpfte Prädikate erlaubt. Es gilt allerdings die Annahme einer geschlossenen Welt[7], d. h. nicht aufgeführte Attribute sind falsch. Weiter dürfen auch keine Funktionen verwendet werden.

Die Festlegung eines Ziels erfolgt analog zum Anfangszustand:

$$Goal(P(a,b) \wedge Q(c,d) \wedge \ldots)$$

Ein Zustand erfüllt das Ziel genau dann, wenn in ihm mindestens alle Attribute erfüllt sind, die im Ziel aufgeführt werden.

Aktionen werden mittels des oben vorgestellten Konzeptes der Add- und Delete-Listen beschrieben:

$$\begin{aligned} Action(Name(x,y), & \\ \text{PRECOND} &: P(x,y) \wedge \ldots \\ \text{ADD} &: Q(y) \wedge \ldots \\ \text{DELETE} &: R(x) \wedge \ldots \\ ) & \end{aligned}$$

Auch hier ist wieder nur die konjunktive Verknüpfung erlaubt. Alle Variablen der drei Listen müssen auch in der Parameterliste erscheinen.

Nun soll die Situation in Abb. 6.2 in der STRIPS Repräsentation dargestellt werden. Zuerst wird die Aktion $Move(x,y)$ beschrieben. Die Vorbedingungen sind wie gehabt, es muss gelten: $Clear(x) \wedge Clear(y) \wedge x \neq y \wedge \neg isTable$. Nun stellen sich allerdings zwei Probleme:

[6] Action Description Language
[7] engl.: closed-world assumption

Angenommen, der zu bewegende Block x stünde auf einem Block b. Nach dem Bewegen von x wäre b frei. Man benötigt also b, um $Clear(b)$ zur neuen Situation hinzuzufügen. Dieser steht nicht in der Parameterliste und es gibt auch keine Möglichkeit, ihn aus der Add- oder Delete-Liste zu ermitteln. Dies ist leicht gelöst, indem man b in die Parameterliste einfügt und die Suche nach b so in die Suche nach Vorbedingungen in der Situation verlagert. Ein größeres Problem ist, dass y der Tisch sein könnte. In diesem Fall dürfte $Clear(Table)$ nicht aus den Situationsmerkmalen entfernt werden. STRIPS liefert jedoch zuerst einmal keine Möglichkeit, dies bedingt geschehen zu lassen.

Um das Problem zu lösen, kann man $Move$ in zwei verschiedene Aktionen aufteilen, von denen eine nur Blöcke auf Blöcke bewegen kann.

$$
\begin{aligned}
&Action(Move(x, y, b),\\
&\qquad \text{PRECOND} \;:\; IsBlock(x) \wedge IsBlock(y) \wedge On(x, b) \wedge Clear(x) \wedge Clear(y) \wedge\\
&\qquad\qquad neq(x, y)\\
&\qquad \text{ADD} \;:\; On(x, y) \wedge Clear(b)\\
&\qquad \text{DELETE} \;:\; On(x, b) \wedge Clear(y)\\
&\qquad)
\end{aligned}
$$

Dazu muss ein neues Prädikat $IsBlock(x)$ eingeführt werden, mit dem festgestellt werden kann, ob es sich bei dem Argument um einen Block oder den Tisch handelt. Da STRIPS keine $\neq$ Relation unterstützt, muss diese als Prädikat $neq(x, y)$ modelliert werden. Die zweite Aktion kann nur Blöcke auf den Tisch verschieben:

$$
\begin{aligned}
&Action(MoveTable(x, b),\\
&\qquad \text{PRECOND} \;:\; IsBlock(b) \wedge IsBlock(x) \wedge On(x, b) \wedge Clear(x)\\
&\qquad \text{ADD} \;:\; On(x, Table) \wedge Clear(b)\\
&\qquad \text{DELETE} \;:\; On(x, b)\\
&\qquad)
\end{aligned}
$$

Nachdem die Aktionen nun spezifiziert sind, kann man mit dem Wissen über die benötigten Prädikate noch leicht den Startzustand formulieren:

$$
\begin{aligned}
Init(\quad & IsBlock(a) \wedge IsBlock(b) \wedge IsBlock(c) \wedge On(a, table) \wedge On(c, table) \wedge On(b, c) \wedge\\
& Clear(Table) \wedge Clear(a) \wedge Clear(b) \wedge neq(a, b) \wedge neq(b, c) \wedge neq(a, c) \wedge\\
& neq(table, a) \wedge neq(table, b) \wedge neq(table, c))
\end{aligned}
$$

Und ebenso das Ziel:

$$Goal(On(a, c) \wedge On(b, a))$$

STRIPS Sprache	**ADL Sprache**
Nur positive Literale in den Zuständen	Positive und negative Literale in den Zuständen
Geschlossene Welt Nicht aufgeführte Literale sind falsch	Offene Welt Nicht aufgeführte Literale sind unbekannt
P in Add-List $\Longrightarrow$ P zum Zustand hinzufügen	P in Add-List $\Longrightarrow$ zum Zustand hinzufügen und $\neg P$ löschen
P in Delete-List $\Longrightarrow$ P aus Zustand löschen	P in Delete-List $\Longrightarrow$ aus Zustand löschen und P hinzufügen
Nur Konstanten als Ziel	Quantifizierte Variablen : $\exists x : \ldots$
Nur Konjunktionen als Ziel	Konjunktion, Disjunktion und Negation
Keine bedingten Effekte	Bedingte Effekte: Falls P dann E
Keine Unterstützung für Gleichheit	Eingebautes Äquivalenzprädikat $x = y$
Keine Unterstützung für Typen	Typisierte Variablen, x : Block

Tabelle 6.1: Gegenüberstellung der Sprachen STRIPS und ADL. Aus [RN03b].

6.3.2 Action Description Language

Obwohl STRIPS die Beschreibung von Planungsproblemen gegenüber der Prädikatenlogik deutlich vereinfacht, bleiben immer noch einige Verbesserungsmöglichkeiten offen. Im vorigen Beispiel wäre eine Typisierung von Variablen wünschenswert gewesen. Bedingte Effekte einer Aktion hätten es erspart, die Aktion *Move* aufzuteilen. In [Ped86] wird die ,,Action Description Language“ (ADL) als Erweiterung von STRIPS vorgeschlagen. Sie wird in Tabelle 6.1 mit STRIPS verglichen.

6.4 Planung als Suche

Durch die STRIPS Repräsentation aus Kapitel 6.3.1 wird der Planungsraum zu einem Graphen. Knoten in diesem sind Zustände der Welt, beschrieben durch eine Liste von erfüllten Attributen. Die Kanten werden durch Anwendung der Aktionen auf die Zustände definiert. Ist nun ein Start- und ein Zielzustand gegeben, so besteht das Planungsproblem darin, einen Pfad vom Startzustand zum Zielzustand zu finden.

Besondere Bedeutung hat die Richtung der Suche. Bei der *Progression* (Vorwärtssuche) wird vom Startzustand ausgegangen. Von dort werden Aktionen ausgewählt, deren Vorbedingungen erfüllt sind. Mit ihnen werden neue Zustände erzeugt und diese (gemäß des verwendeten Suchalgorithmus) weiter durchsucht. Der Nachteil dieser Methode ist, dass sehr viele Aktionen verwendet werden, die keinen Bezug zur Lösung des Problems haben und die Suchbäume dadurch sehr breit werden.

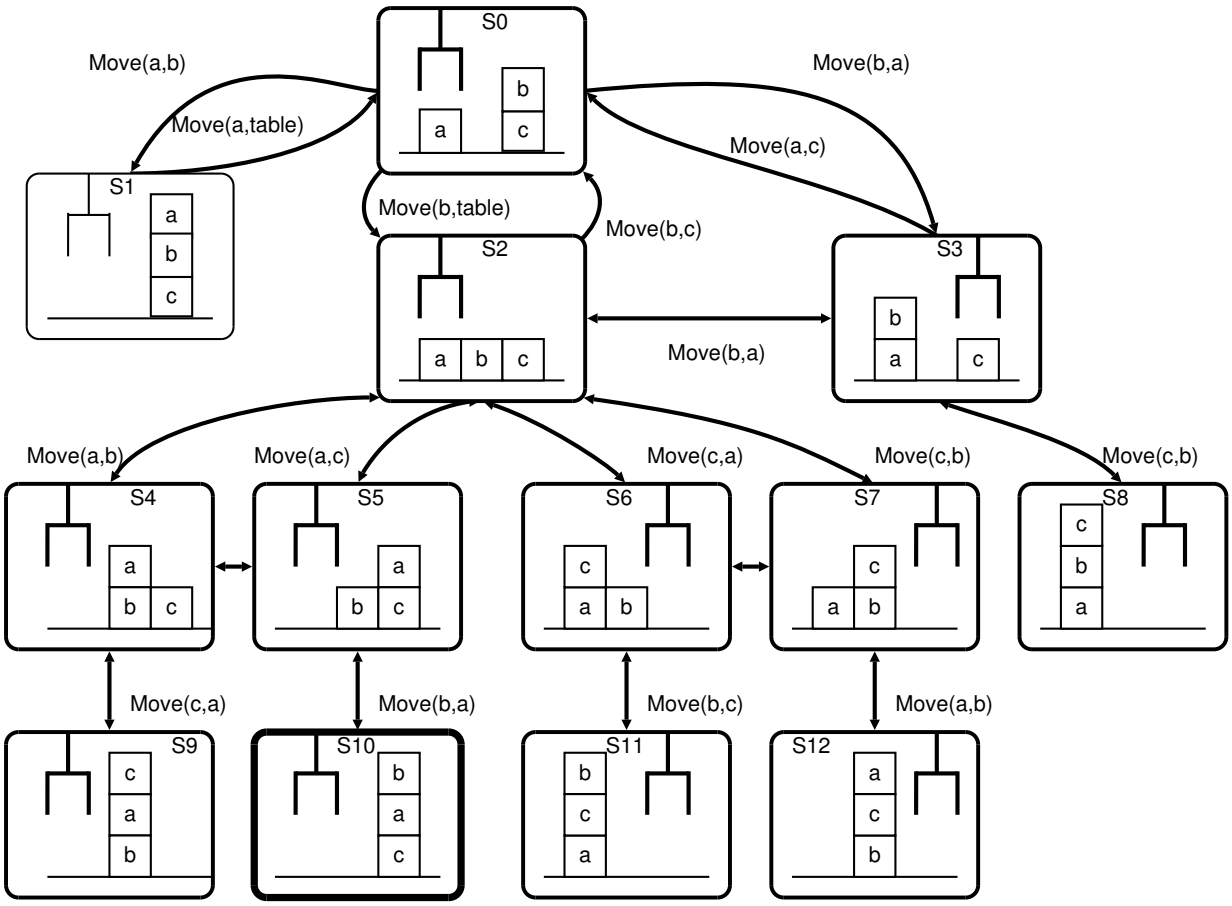


Abbildung 6.4: Suchraum zum Beispiel aus Abb. 6.2. Alle Kanten sind reflexiv, da alle Aktionen in der Blockwelt rückgängig gemacht werden können. Außer bei Zustand S0 sind nur Aktionen, die nach unten führen beschriftet.

Als Alternative bietet sich die *Regression* (Rückwärtssuche) an. Bei ihr wird vom Zielzustand ausgegangen. Von dort aus werden Vorgängerknoten erzeugt, indem Aktionen ausgewählt werden, die ein Teilziel des Knotens erfüllen. Der neue Knoten enthält alle Teilziele des alten Knotens, bis auf die von der Aktion erfüllten. Die Vorbedingungen der Aktion werden als Teilziele hinzugefügt. Die Suche endet, sobald ein Zustand gefunden wird, dessen Teilziele vom Startzustand erfüllt werden.

6.4.1 Suchalgorithmen

In diesem Abschnitt wird kurz auf verschiedene Suchalgorithmen eingegangen, die zur Planung verwendet werden. Von besonderer Bedeutung sind dabei die im Folgenden definierten Eigenschaften:

Definition Ein Suchalgorithmus ist *optimal*, wenn er den kürzesten Pfad vom Start- zum Zielknoten liefert.

Definition Ein Suchalgorithmus ist *vollständig*[8] wenn er, falls es einen Pfad vom Start- zum Zielknoten gibt, diesen findet.

Vollständigkeit ist besonders bei unendlichen Zustandsräumen relevant. Diese könnten beispielsweise in der Blockwelt entstehen, wenn es eine Aktion gäbe, die einen neuen Block erstellt und diesen auf den Tisch legt.

6.4.1.1 Tiefensuche

Die Tiefensuche beginnt am Startzustand. Von dort aus wird immer der tiefste bekannte Knoten weiter erforscht. Im Beispiel aus Abb. 6.4 muss beachtet werden, dass die Suche bereits besuchte Knoten auch wirklich erkennt, da der Graph nicht zyklenfrei ist. Ist dies gewährleistet, wird die Tiefensuche die Zustände in folgender Reihenfolge bearbeiten: S0, S1, S2, S4, S9, S5, **S10**. Sie ist weder optimal noch vollständig.

Für unendlich große Zustandsräume ist die Tiefensuche ungeeignet. Sie lässt sich jedoch um eine Tiefenbeschränkung erweitern. Wird diese erreicht, wird der Knoten nicht weiter exploriert. Die Tiefenbeschränkung wird dann sukzessive erhöht, bis eine Lösung gefunden ist. Mehr zur Tiefensuche findet sich in [Goo05].

6.4.1.2 Breitensuche

Die Breitensuche durchsucht die unerforschten Knoten gleicher Tiefe, beim Startzustand der die Tiefe 0 hat, beginnend. Eine besondere Behandlung von Zyklen ist nicht notwendig, die Breitensuche ist vollständig und optimal. Im Beispiel aus Abb. 6.4 werden die Knoten in der Reihenfolge S0, S1, ..., **S10** besucht. Mehr zur Breitensuche siehe [Goo05].

Eine Erweiterung zur Breitensuche ist die *Bidirektionale Suche*. Dabei wird eine Breitensuche vom Startzustand und eine weitere vom Zielzustand aus gestartet. Ein Pfad ist gefunden, wenn es einen Knoten gibt, der von beiden Suchen gefunden wurde. Die Lösung ergibt sich durch aneinanderhängen der beiden Pfade zum gemeinsamen Knoten. Die bidirektionale Suche ist ebenfalls optimal und vollständig. Mehr zur bidirektionalen Suche findet sich in [RN03b].

6.4.1.3 A*-Suche

Während die Tiefen- und Breitensuche nur die Topologie des zu durchsuchenden Graphens berücksichtigen, wird bei der A* Suche weiteres Wissen über die Struktur des Suchraums

[8] engl.: complete

eingebracht. Die geschieht durch eine Heuristik $h(S_n)$, die die Kosten (Entfernung, Anzahl Aktionen oder ähnlich) vom Knoten S_n zum Zielknoten schätzt. Weiter können für jeden Knoten die tatsächlichen Kosten $g(S_n)$ berechnet werden, die entstehen, um ihn vom Startzustand aus zu erreichen. Die eigentliche Suche läuft dann wie folgt ab:

Zuerst wird für den Startzustand $f(S_0) = h(S_0)$ gesetzt. Nun wird aus allen bekannten Knoten der Knoten mit dem niedrigsten $f(S_n)$ gewählt. Für die von diesem aus erreichbaren Nachfolgerknoten wird $f(S_n) = g(S_n) + h(S_n)$ gesetzt. Dies wird wiederholt, bis der Zielzustand gefunden ist.

Es wird also immer der Knoten weiter durchsucht, für den der vermutete Weg vom Start zum Ziel minimal ist. Die Heuristik $h(S_n)$ zieht die Suche zum Ziel hin. Bei der Suche einer Route auf einer Landkarte kann als Heuristik zum Beispiel die euklidische Distanz gewählt werden. Im Falle des Planungsproblems ist es nicht so einfach, eine gute Heuristik zu bestimmen. Eine einfache Heuristik ist die Zahl der offenen Teilziele. Damit die A* Suche optimal ist, darf die Heuristik die Kosten nicht überschätzen. Die vorgeschlagene Heuristik erfüllt dies nur, wenn eine Aktion nicht mehrere Teilziele erfüllen kann. Um bessere Heuristiken zu finden, benutzt man Algorithmen, die vereinfachte Planungsprobleme lösen, z. B. indem ignoriert wird, dass Aktionen Ergebnisse von vorherigen Aktionen beeinflussen können.

Bei Verwendung einer geeigneten Heuristik ist die A* Suche vollständig und optimal. In Tabelle 6.2 wurde die A* Suche auf das Beispiel aus Abb. 6.4 angewandt. Die Kosten sind dabei die Länge des Pfades, die Heuristik die Anzahl der offenen Teilziele ($On(b, a)$, $On(a, c)$, $On(c, table)$).

6.5 Lineare Planer

Bei einer rein auf Suche beschränkten Planung wird die Größe des Planungsraumes schnell zum Problem. Ein einfaches Beispiel: n unterscheidbare Pakete sollen in ein Flugzeug geladen werden und zu einem bestimmten Flughafen geflogen werden. Alleine zum vollständigen Beladen des Flugzeuges gibt es in diesem Beispiel $n!$ mögliche Pläne. Dieses Wachsen des Zustandsraums wird als *Kombinatorische Explosion* bezeichnet. Um diese zu verhindern gibt es zwei Ansätze. Zum einen kann man dem Planer eine bestimmte Reihenfolge der Erfüllung der Teilziele aufzwingen, zum anderen kann man Planer entwerfen, welche die Reihenfolge von Aktionen nur dann planen, wenn diese tatsächlich von Bedeutung ist. Der zweite Ansatz wird in Abschnitt 6.6 beschrieben.

Ein Planer, bei dem die Reihenfolge der Teilziele fest ist, heißt *linear*. Die zu erreichenden Teilziele werden bei so einem Planer als Stapel angelegt, dessen einzelne Elemente der Reihe nach abgearbeitet werden müssen. Im Gegensatz dazu arbeitet ein *nicht linearer* Planer mit einer Menge von Zielen. Von diesen wird (nicht deterministisch) eines ausgewählt und bearbeitet.

Bei linearen Planern wird ebenfalls ausgenutzt, dass die Teilziele voneinander unabhängig

Schritt	Besuchter Knoten	Bekannte Knoten ($f(n) = g(n) + h(n)$)	Kommentar
0		S0($2 = 0 + 2$)	Startschritt, S0 wird mit 2 offenen Teilzielen initialisiert
1	S0	S1($3 = 1 + 2$), S2($3 = 1 + 2$), S3($2 = 1 + 1$)	Die Nachfolgerknoten von S0 werden berechnet. Die Kosten um die Knoten zu erreichen sind für alle drei Knoten 1, S1 und S2 haben weiter 2 offen e Teilziele, lediglich in S3 wird das Teilziel $On(b, a)$ erfüllt und hat damit nur noch ein offenes Teilziel. Mit $f(S3) = 2$ erhält dieser Knoten die niedrigste Bewertung und wird daher im nächsten Schritt durchsucht.
2	S3	S1(3), S2(3), S8 ($4 = 2 + 2$)	Die von S3 aus erreichbaren Knoten S0 und S2 sind billiger zu erreichen als über S3, daher wird ihre Bewertung nicht verändert. Lediglich der Knoten S8 wird neu erforscht.
3	S1	S2(3), S8 (4)	Die Knoten S1 und S2 sind nun mit einer Bewertung von 3 gleichwertig, der Algorithmus entscheidet sich willkürlich für S1. Dort kann jedoch kein neuer Knoten entdeckt werden, die einzige Kante führt zurück zum Vorgänger. Im nächsten Schritt wird daher S2 durchsucht.
4	S2	S8(4), S4($4 = 2 + 2$), S5($3 = 2 + 1$), S6($5 = 2 + 3$), S7($5 = 2 + 3$)	Von S2 aus werden 4 neue Nachfolger gefunden. In S5 und S6 liegt c nicht mehr auf dem Tisch, daher sind wieder 3 Teilziele offen. In S4 sind weiter 2 Teilziele offen, der Zustand S5 hat nur ein offenes Teilziel. Daher wird er am niedrigsten bewertet und als nächstes durchsucht.
5	S5	S4(4), S5(3), S6(5), S7(5), **S10(**$3 = 3 + 0$**)**	Mit dem Zustand S10 wird das Ziel gefunden.

Tabelle 6.2: Beispiel zur A* Suche mit Anzahl der offenen Teilzielen als Heuristik

sind, d. h. die Erfüllung eines Ziels nicht die eines anderen beeinflusst. Dies ist häufig der Fall. Beeinflussen sich Teilziele, so ist zwischen *positiver Interaktion*, d. h. die Lösung eines Teilziels löst gleichzeitig auch ein anderes und *negativer Interaktion*, d. h. die Lösung eines Teilziels wird durch die eines anderen zerstört, zu unterscheiden. Lineare Planer können oft auch bei nicht unabhängigen Teilzielen Lösungen finden, diese können jedoch suboptimal sein (siehe Abschnitt 6.5.2).

6.5.1 Der STRIPS Planer

Der wohl bekannteste lineare Planer ist der STRIPS Planer nach [FN71]. Seine Planungsraumrepräsentation wurde bereits in Abschnitt 6.3.1 erläutert, im folgenden wird daher nur auf seine Suchstrategie eingegangen. Die zu durchsuchenden Zustände enthalten neben der Situationsbeschreibung nun ebenfalls einen Stapel mit unerfüllten Teilzielen $< g_n, g_{n-1}, \ldots, g_0 >$. Dabei ist g_n das nächste zu bearbeitende Ziel, g_0 das Endziel. Zum Bestimmen der Reihenfolge der Aktionen soll hier eine Liste verwendet werden. Der Originalalgorithmus baut stattdessen einen Baum aus Zuständen, Zielen und Aktionen auf, was nötig ist, um nach einer falschen Aktionswahl ein Backtracking durchzuführen, also die Suche mit einer anderen Aktionswahl fortzusetzen. Hier soll der Einfachheit halber angenommen werden, der Algorithmus würde sich immer für die richtige Aktion entscheiden, d. h. wir betrachten die Wahl einer Aktion als nicht deterministisch.

Die Planung läuft nun wie folgt ab:

STRIPS(Situation s, Ziele $g =< g_n, ... >$, Aktionsliste al)

do forever:

- Sei g_n das oberste Element des Stapels g
- Falls s g_n erfüllt:
  - Falls g_n ist Endziel: return fertig.
  - Falls g_n als Vorbedingung von Aktion a entstanden ist (nicht durch Aufteilen einer Vorbedinung in Teilziele): hänge a an al an, lösche g_n aus g und füge Nachbedingung von a zu s hinzu. continue.
  - sonst: lösche g_n aus g. continue.
- sonst (s erfüllt g_n nicht):
  - Falls g_n in Teilziele aufteilbar, tue dies und füge diese zu g hinzu. continue.
  - Wähle Aktion a, die g_n erfüllt. Füge ein Ziel mit nicht erfüllten Vorbedingungen von a zu g hinzu. Falls es keine solche Aktion gibt[9], return Fehlschlag

[9] an dieser Stelle müsste bei einer echten Implementierung das Backtracking stattfinden

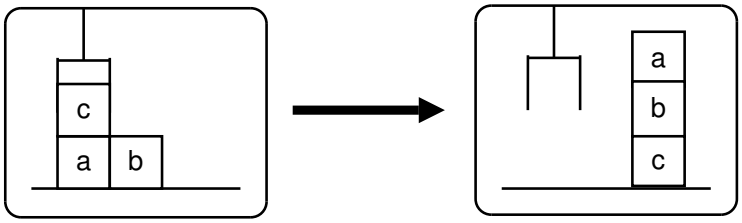


Abbildung 6.5: Beispielproblem zum STRIPS Planer

Die Anweisung continue setzt die Schleife in der Zeile do forever fort. Ein Beispiel zur Ausführung des Algorithmus findet sich im folgenden Abschnitt.

6.5.2 Die Sussman Anomalie

In Abb. 6.5 ist ein einfaches Problem in der Blockwelt dargestellt. Die für den Menschen offensichtliche optimale Lösung für dieses Problem ist die Aktionsfolge $MoveTable(c)$, $Move(b, c)$ und $Move(a, b)$. STRIPS ist jedoch nicht in der Lage, diese Lösung zu finden. Die vom Planer durchgeführte Suche ist in Abb. 6.6 aufgezeigt. Durch die Formulierung des Ziels $(On(a, b), On(b, c))$ wird die Aktion $Move(a, b)$ vor $Move(b, c)$ erzwungen. Der Effekt dieses Schritts wird jedoch beim Erfüllen des Teilziel $On(b, c)$ wieder zerstört (negative Teilzielinteraktion) und muss am Ende des Plans repariert werden. Dadurch ergibt sich der folgende Plan: $MoveTable(c)$, $Move(a, b)$, $MoveTable(a)$, $Move(b, c)$, $Move(a, b)$. Dieser Plan ist zwar korrekt, doch sind zwei Aktionen überflüssig. Auch die umgekehrte Ordnung der Teilziele hätte einen suboptimalen Plan erzeugt. Dieses Phänomen ist in der Literatur als die *Sussman Anomalie*[10] bekannt. Ihr Name entstand, weil sie im Zusammenhang mit Gerald Sussmans „HACKER“ Planer [Sus73] entdeckt wurde.

6.5.3 Unlösbare Probleme

Ein bekanntes, für einen linearen Planer nicht lösbares Problem ist das *UPS Problem*, auch unter dem Namen *One Way Rocket* bekannt. Dieses soll hier nur kurz skizziert werden. Gegeben ist ein Startzustand mit 2 Objekten O_1 und O_2 und einem betankten Flugzeug P an einem Flughafen A_1: $HaveFuel(P)$, $At(A_1, P)$, $At(A_1, O_1)$, $At(A_1, O_2)$. Weiter gibt es die Aktion $Load(a, o, p)$, die das Verladen eines Objektes in das Flugzeug modelliert

[10] Sussman Anomaly

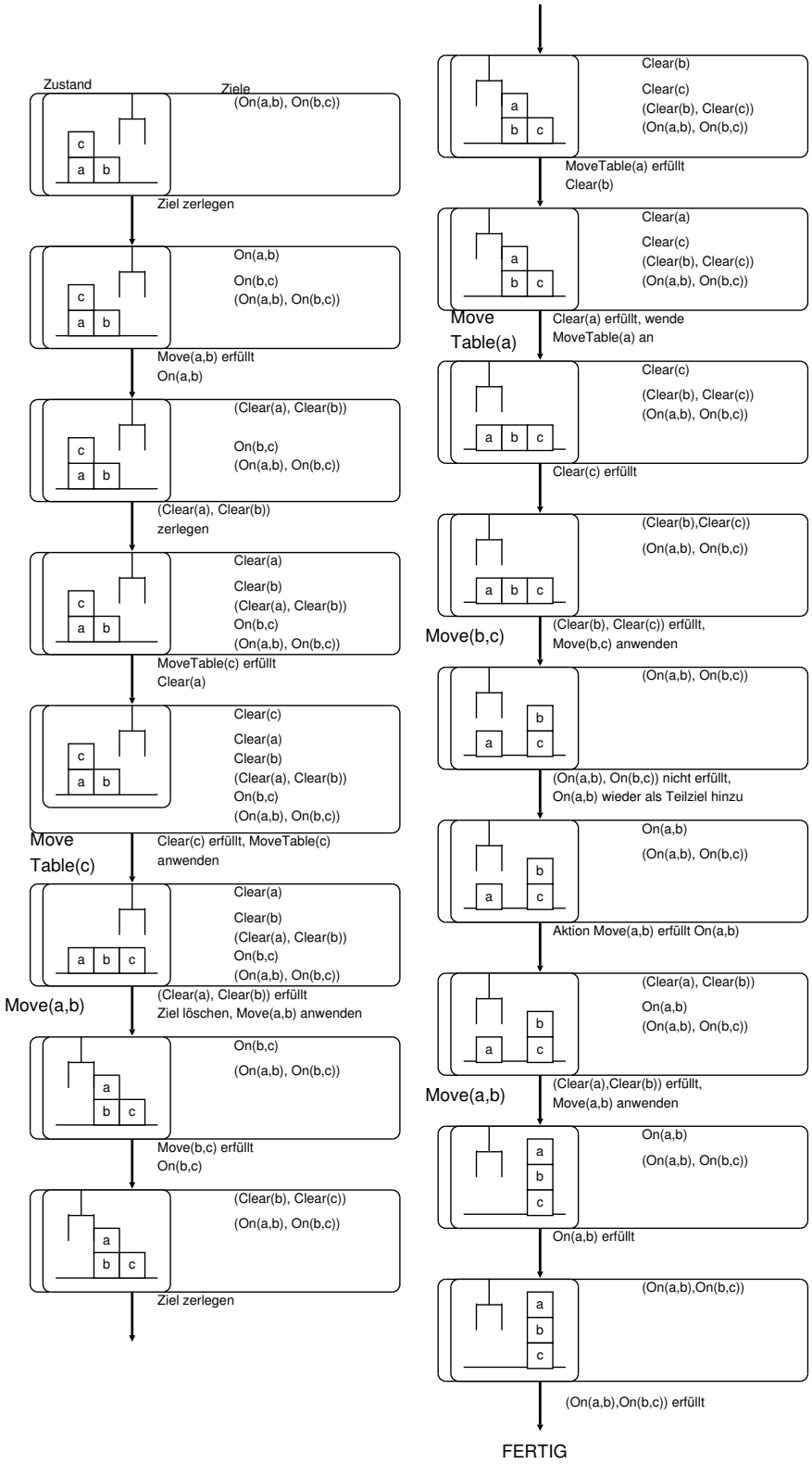


Abbildung 6.6: Lösungsweg der Sussman Anomalie

und analog dazu die Aktion $Unload(a, o, p)$. Die einzige Aktion, die hier genau spezifiziert werden muss, ist Fly:

$$
\begin{aligned}
&Action(Fly(p, a_1, a_2),\\
&\qquad \text{PRECOND} \;:\; Plane(p) \wedge At(p, a_1) \wedge HaveFuel(p)\\
&\qquad \text{ADD} \;:\; At(p, a_2)\\
&\qquad \text{DELETE} \;:\; At(p, a_1), HaveFuel(p)\\
&\qquad)
\end{aligned}
$$

Das Ziel ist es, die beiden Objekte an einen Flughafen A_2 zu bringen: $At(A_2, O_1)$, $At(A_2, O_2)$. Dies kann einem linearen Planer nicht gelingen, da immer ein Teilziel vollständig erfüllt wird und damit die Vorbedingung für das zweite Ziel nicht mehr gegeben ist.

6.6 Partial Order Planning

Da lineare Planer unvollständig sind, ist es notwendig, sich für nicht-lineare Planer Strukturen zu überlegen, mit denen die Anzahl der möglichen Pläne reduziert werden kann. Dazu eignen sich partiell geordnete[11] Pläne[12]. Das hier vorgestellte Partial Order Planning (POP) basiert auf [Wel94] und dessen Beschreibung in [RN03b].

POP basiert auf einer Suche im Raum der möglichen Pläne, nicht auf einer direkten Suche im Zustandsraum. Die Pläne werden durch eine Menge Aktionen $\mathcal{A}$ und einer Ordnung $\mathcal{O}$ beschrieben. Die Ordnung besteht aus Paaren der Form A $\prec$ B, die besagen, dass Aktion A vor B ausgeführt werden muss. Dabei wird A und B nur dann geordnet, wenn A tatsächlich zwingend vor B ausgeführt werden muss. Ein solcher Plan wird *partiell geordnet* genannt. Im Gegensatz dazu waren die in den vorigen Abschnitten vorgestellten Pläne *vollständig geordnet*[13].

In Abb. 6.7 ist beispielhaft ein Plan zum Anziehen eines Paars Schuhe gezeigt. Da die Socken vor den Schuhen angezogen werden müssen, ist zwischen diesen Aktionen jeweils ein Pfeil, der eine Ordnungsbedingung darstellt. Die Aktionen an unterschiedlichen Füßen sind voneinander unabhängig, zwischen ihnen gibt es keinen Pfeil. Diese Darstellung ermöglicht nicht nur eine Verkleinerung des Planungsraums, sie hält gleichzeitig die Möglichkeit offen, Teilziele (im Beispiel rechter bzw. linker Schuh an) unabhängig zu Planen.

Um auf POP planen zu können, muss ihnen jedoch noch eine Menge $\mathcal{O}$ mit *kausalen Bedingungen* hinzugefügt werden. $A \xrightarrow{c} B$ bedeutet, dass A c bewirkt, welches von B als Vorbedingung benötigt wird. Im Beispiel ist „linke Socke $\xrightarrow{\text{linke Socke an}}$ linker Schuh“ eine kausale Bedingung. Sie ist durch den Text an den Pfeilspitzen dargestellt (jede kausale Bedingung ist auch eine Ordnungsbedingung). Ähnlich wie in STRIPS wird eine Menge

[11] Die partielle Ordnung wird auch Halbordnung genannt.
[12] engl.: partial ordered plan
[13] engl.: total ordered

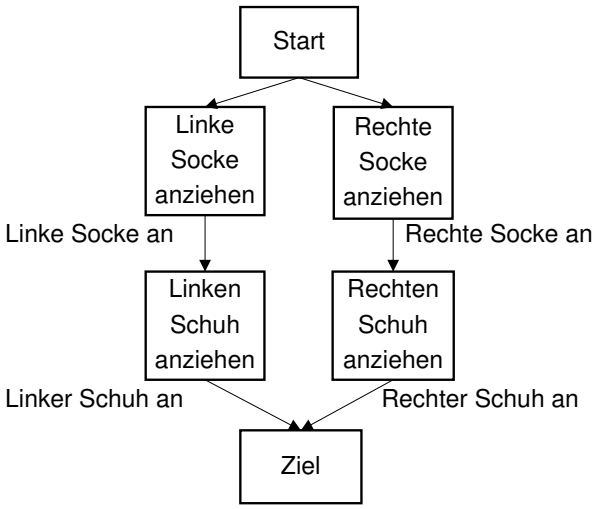


Abbildung 6.7: Beispiel für einen partiell geordneten Plan nach [RN03b]. Pfeile entsprechen Ordnungsbedingungen.

mit offenen Vorbedingungen $\mathcal{P}$ benutzt, sie wird nun jedoch wirklich als Menge, nicht als Stapel betrachtet.

Zur Beschreibung des Algorithmus wird die nichtdeterministische Funktion *Wähle Aktion* benutzt. Um positive Teilzielinteraktion berücksichtigen zu können, kann diese nun auch Aktionen liefern, die bereits im Plan vorkommen. Wäre im Beispiel des Schuheanziehens *linker Schuh anziehen* auch von der rechten Socke abhängig, so muss dies durch die bereits im Plan vorhandene Aktion *rechte Socke anziehen* erfüllt werden können. Ein zweites *rechte Socke anziehen* wäre dagegen nicht wünschenswert. Eine weitere Funktion *Ordne* ordnet einen Plan mit in Konflikt stehenden Aktionen so, dass die Konflikte verhindert werden. Eine Aktion kann in einem Konflikt mit einer kausalen Bedingung stehen, wenn sie die von ihr gesicherte Eigenschaft zerstört. Bedroht etwa die Aktion D die Eigenschaft c aus $A \xrightarrow{c} B$, so kann D entweder vor A (C $\prec$ A) oder nach B (B $\prec$ C) eingeordnet werden. Daher ist auch diese Funktion nicht deterministisch.

Zu Beginn der Planung wird ein Plan erstellt, der die Aktionen Start und Ziel sowie die Ordnungsbedingung Start $\prec$ Ziel enthält. Start hat keine Vorbedingungen. Durch seine Effekte wird der Startzustand modelliert. Ziel hat die gewünschten Eigenschaften des Zielzustandes als Vorbedingung. Neue Pläne werden wie folgt erzeugt:

- Wähle eine Aktion A, die eine offene Vorbedingung p einer Aktion B erfüllt.
- Füge A $\prec$ B und $A \xrightarrow{p} B$ zum Plan hinzu. Lösche p aus der Menge der offenen Vorbedingungen.
- Falls A noch nicht im Plan vorhanden, füge A hinzu, ebenso wie Start $\prec$ A und $A \prec$ Ziel

- Suche nach Konflikten zwischen vorhandenen Kausalbedingungen und A bzw. zwischen $A \xrightarrow{p} B$ und anderen Aktionen. Falls Konflikte bestehen, wende ,,Ordne“ auf den Plan an.
- Falls es keine offenen Zielbedingungen mehr gibt, gib diesen Plan als Lösung zurück

Lösen der Sussman Anomalie mittels POP Zur Verdeutlichung des Algorithmus soll im Beispiel in Abb. 6.8 gezeigt werden, wie der POP Planer das Problem der Sussman Anomalie (Abb. 6.5) löst.

Im ersten Schritt wird der Plan mit den Start- und Zielaktionen initialisiert (Abb. 6.8(a)). Die Ordnungsbedingungen zwischen Start und anderen Aktionen bzw. Ziel und anderen Aktionen werden in Abb. 6.8 nicht dargestellt. Im nächsten Schritt wird die Vorbedingung $On(b, c)$ ausgewählt und mittels der Aktion $Move(b, c)$ erfüllt. Die Vorbedingung $Clear(b)$ wird im folgenden Schritt durch die schon vorhandene Aktion Start erfüllt, es ergibt sich der Plan in Abb. 6.8(b).

Nun wird das offene Ziel $On(a, b)$ gewählt. Dieses kann durch die Aktion $Move(a, b)$ erfüllt werden, die dem Plan hinzugefügt wird. Allerdings hat diese $\neg Clear(b)$ zur Folge. Da $Clear(b)$ Vorbedingung von $Move(b, c)$ ist, bedroht $Move(a, b)$ also die Kausalbedingung $Start \xrightarrow{Clear(b)} Move(b, c)$, in Abb. 6.8(c) durch einen gepunkteten Pfeil dargestellt. Weil ein Vorziehen der Aktion $Move(a, b)$ vor Start nicht möglich ist, bleibt als einzige Lösung, $Move(a, b)$ hinter $Move(b, c)$ einzuordnen, also $Move(b, c) \prec Move(a, b)$ zum Plan hinzuzufügen. Der soweit entstandene Plan ist in Abb. 6.8(d) zu sehen.

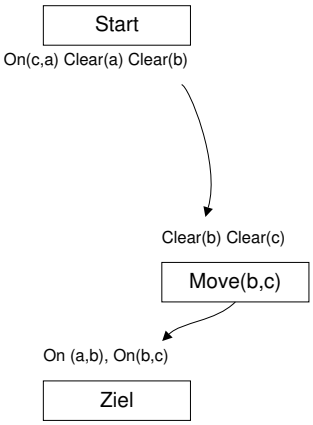
In den beiden folgenden Schritten wird mittels der bereits vorhandenen Aktion Start die offene Bedingung $Clear(a)$ bzw. $Clear(b)$ erfüllt, sodass sich der Plan aus Abb. 6.8(e) ergibt. Die letzte offene Vorbedingung $Clear(c)$ kann mittels der Aktion $MoveTable(a)$ erfüllt werden. Sie bedroht keine Kausalbedingungen. Ihre Vorbedingungen können im letzten Schritt durch die Startaktion erfüllt werden.

Der fertige Plan ist in Abb. 6.8(f) dargestellt. Aus ihm folgt der total geordnete Plan $MoveTable(a)$, $Move(b, c)$ und $Move(a, b)$, der das Problem optimal löst.

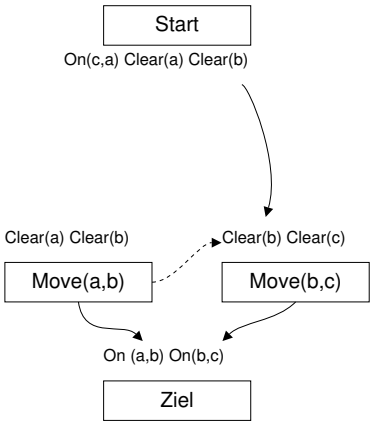
6.7 Hierarchische Planung

Die bisher vorgestellten Planungsverfahren beruhten alle auf einer recht willkürlichen Aneinanderreihung von ausführbaren Aktionen. Durch Heuristiken ließ sich die Richtung der Suche in Richtung des Ziels beeinflussen, prinzipiell ist jede Aktionsreihenfolge möglich. Menschen gehen dagegen bei der Planung anders vor. Planung findet zuerst auf einer hohen, abstrakten Ebene statt. Erst wenn dort ein Plan besteht, wird dieser weiter in einfachere Teilschritte zerlegt. Will man sich zum Beispiel eine Tasse Kaffee aus der Küche holen, wird man nicht darüber nachdenken, ob man zuerst in den Schrank greifen soll oder doch erst vom Schreibtischstuhl aufstehen soll. Stattdessen wird man sofort auf den einfachen

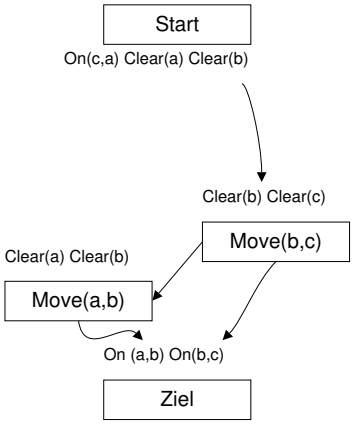
(a) Anfangssituation



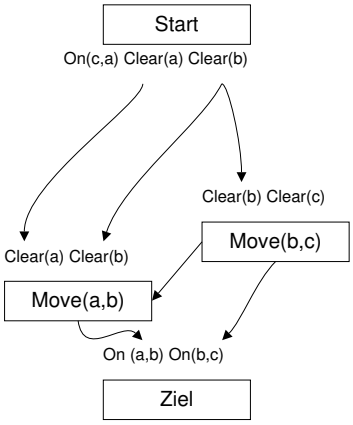
(b) Move(b,c) hinzugefügt, Start verwendet (2 Schritte zusammengefasst)



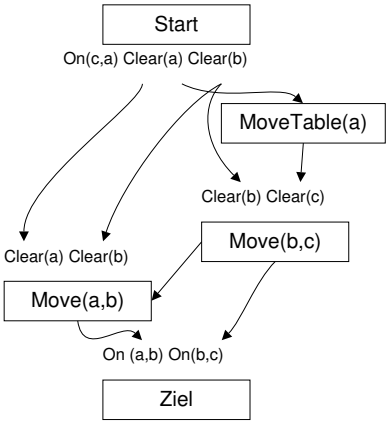
(c) Move(a,b) hinzugefügt



(d) Plan geordnet



(e) Startaktion genutzt, um Vorbedingungen zu erfüllen (2 Schritte zusammengefasst)



(f) MoveTable(a) hinzugefügt und Vorbedingungen durch Start erfüllt (2 Schritte zusammengefasst)

Abbildung 6.8: POP auf das Beispiel der Sussman Anomalie angewandt. Quelle:[Wel94]

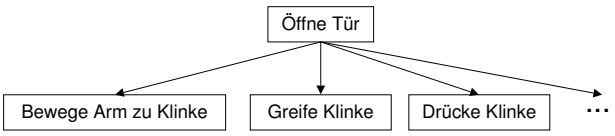


Abbildung 6.9: Beispiel für einen Makrooperator

Plan ,,gehe in die Küche, besorge Tasse, schenke Kaffee ein, gehe zurück an den Schreibtisch“ kommen. Dieser wird dann (beim Menschen während der Ausführung) in weitere, konkretere Aktionen zerlegt. ,,gehe in die Küche“ lässt sich beispielsweise in ,,stehe vom Schreibtischstuhl auf, gehe zur Tür, öffne Tür, ...“ zerlegen, öffne Tür weiter in ,,bewege Arm zu Türklinke, greife Türklinke, drücke Türklinke nach unten, ...“. Letztlich entsteht so ein komplexer Plan durch rekursives Lösen einfacher Teilprobleme.

Aber musste eine Lösung der Teilprobleme tatsächlich gesucht werden? Eigentlich nicht, denn man wusste ja schon vor Beginn der Planung wie man zum Beispiel die Aufgabe ,,gehe in die Küche“ lösen würde. Es wird also auf zwei verschiedene Weisen zusätzliches Wissen in die Planung eingebracht: zum einen führen die Unterteilungen auf den höheren Ebenen dazu, dass Planung auf die durch sie spezifizierten Teilprobleme beschränkt werden kann, die sich gegenseitig nicht beeinflussen. Es findet also eine Teilzielzerlegung statt. Zum anderen ist für viele häufig vorkommenden Teilprobleme ihre Lösung von vornherein bekannt, es muss keine Suche mehr stattfinden.

Um hierarchisches Planen auf dem Computer zu ermöglichen, führt man so genannte Makrooperatoren ein(siehe auch Abschnitt 4.2.1.2: Abstraktionsebenen). Wie Aktionen besitzen diese Vor- und Nachbedingungen. Neben diesen enthalten sie jedoch noch einen *Körper*. Dieser ist eine Liste mit *Subtasks*, also Aufgaben, die abgearbeitet werden müssen, um den Makrooperator auszuführen. In Abb. 6.9 ist der Operator ,,Öffne Tür“ dargestellt. Expandiert man einen Operator, so ergibt sich ein hierarchisches Netz von Teilaufgaben. Daher wird diese Art der Planung als Planung mit *hierarchial task network* (HTN) bezeichnet.

Ein einfacher Makrooperator *Schuh anziehen(seite)* könnte z. B. zur Lösung des Schuhproblems aus Abb. 6.7 verwendet werden. Als Körper hat er die Aktionen *seite Socke anziehen* und *seite Schuh anziehen*. Davon ausgehend, dass in diesem sehr einfachem Beispiel Schuhe und Socken immer vorhanden sind, benötigt er keine Vorbedingungen. Der Planer kann nun einfach für jede Seite *Schuh anziehen* anwenden, und hat nun auf der höchsten Ebene einen vollständigen Plan. Wird nun im nächsten Schritt *Schuh anziehen* expandiert, so ergibt sich der selbe Plan, wie in Abb. 6.7.

Ein Makrooperator kann auch mehrere Definitionen haben, falls ein gegebenes Ziel auf verschiedenen Wegen gelöst werden kann. Ebenso können Makrooperatoren rekursiv definiert sein. Ein in der Blockwelt sinnvoller Makrooperator könnte beispielsweise $ClearBlock$ sein, der dafür sorgt, dass ein Block frei wird:

$$
\begin{aligned}
&Action(ClearBlock(b),\\
&\qquad \text{PRECOND} &&: Clear(b)\\
&\qquad \text{ADD} &&: (leer)\\
&\qquad \text{DELETE} &&: (leer)\\
&\qquad \text{BODY} &&: (leer)\\
&\qquad)\\
&Action(ClearBlock(b),\\
&\qquad \text{PRECOND} &&: On(x,b)\\
&\qquad \text{ADD} &&: Clear(b)\\
&\qquad \text{DELETE} &&: (leer)\\
&\qquad \text{BODY} &&: ClearBlock(x), MoveTable(x)\\
&\qquad)
\end{aligned}
$$

Die erste Definition bearbeitet einen bereits freien Block, die zweite entfernt Blöcke von einem Stapel, bis der angegebene Block frei ist.

Ein Algorithmus zur Planung auf HTNs wird beispielsweise in [NAI+03] beschrieben. Seine Vorgehensweise soll hier nur skizziert werden, tatsächlich ist der vorgestellte Algorithmus deutlich mächtiger, so beherrscht er auch zeitliche Planung.

Pläne werden partiell geordnet dargestellt. Die Planung ähnelt daher stark dem im vorigen Abschnitt beschriebenen Partial Order Planning. Allerdings ist es um die Möglichkeit, Makrooperatoren zu verwenden und zu expandieren erweitert. Besonders zu berücksichtigen ist die positive Teilzielinteraktion. In [RN03b] ist dazu das folgende Beispiel zu finden. Die Aufgabe ist es, die Flitterwochen zu genießen und danach eine Familie zu gründen. Ein HTN Planer könnte diese Ziele wie folgt aufteilen: Um die Flitterwochen zu genießen sollte man zuerst heiraten und dann nach Hawaii fliegen. Um eine Familie zu gründen könnte der Planer vorschlagen, zu heiraten und danach zwei Kinder groß zu ziehen. Der generierte Plan enthält dann zwei Mal die Aktion Heiraten, was allerdings kaum gewünscht sein wird.

Ein weiteres Problem ist die Ordnung der Aktionen. Zwar kann der Plan auf einer hohen Ebene verlangen, dass zwei Makrooperatoren hintereinander geordnet sein müssen. Aber daraus folgt nicht, dass jede Aktion des zweiten Operators nach jeder des ersten folgen muss. Stattdessen wird es auf der tieferen Ebene nur einige Aktionen geben, die geordnet sein müssen. Diese Abhängigkeiten müssen extra bestimmt werden.

6.8 Zusammenfassung und Ausblick

Es wurden verschiedene Verfahren zur Aktionsplanung vorgestellt. Das Situationskalkül basiert vollständig auf einer prädikatenlogischen Darstellung des Planungsproblems. Dies

hat den Vorteil, dass diese Darstellungsform sehr mächtig ist, also auch sehr komplexe Probleme formuliert werden können. Dies ist jedoch auch gleichzeitig ein Nachteil, da das Formulieren sehr schwierig wird.

Im Situationskalkül müssen von Aktionen unveränderte Attribute explizit in den nächsten Zustand übernommen werden. Andere Planer verwenden dagegen oft die sog. STRIPS Assumption, d. h. alles, was in einer Aktion nicht verändert wird, wird aus dem alten Zustand übernommen. Die STRIPS Repräsentation ist eine Sprache, die Planungsprobleme unter dieser Annahme beschreibt. Ist ein Planungsproblem in ihr gegeben, lässt sich die Planung als eine Suche im Zustandsraum betrachten. Jedoch werden die Zustandsräume für eine Suche ohne weitere Annahmen zu groß. Daher wird versucht, weiteres Wissen in die Planung einzubringen. Auf der Ebene der Suche ist dies über eine Heuristik möglich, welche die Richtung der Suche beeinflusst.

Eine andere Möglichkeit ist die lineare Planung. Dabei werden Ziele in Teilziele zerlegt, die in einer festen Reihenfolge bearbeitet werden. Lineare Planung ist allerdings unvollständig.

Partial Order Planer arbeiten auf partiell geordneten Plänen. In diesen müssen nicht alle Aktionen geordnet sein, wodurch die Anzahl der zu durchsuchenden Pläne reduziert und gleichzeitig die Planung von unabhängigen Teilzielen ermöglicht wird.

Planung auf Hierarchischen Task Netzwerken ermöglicht es, für bekannte Teilprobleme Standardlösungen als Wissen in die Planung einzubringen. Gleichzeitig können auch Methoden des Partial Order Plannings genutzt werden.

Als Ausblick bleibt noch auf weitere, hier nicht behandelte Verfahren hinzuweisen. Besonders zu erwähnen sind Verfahren, die auf Planungsgraphen basieren. Dazu wird zuerst vom Startzustand ausgehend ein Graph mit Aktionsfolgen generiert, die keine offensichtlichen Widersprüche erzeugen [RN03b]. Danach wird in diesem Graph nach einer korrekten Lösung gesucht. Bei der SAT Planung (von **sat**isfiability, Erfüllbarkeit) wird das Planungsproblem auf eine aussagenlogische Formel abgebildet. Eine Lösung wird gesucht, indem ein Modell für diese gesucht wird.

Planung ist traditionell ein Kerngebiet der Künstlichen Intelligenz und als solches immer noch ein sehr aktives Forschungsgebiet. Daher ist damit zu rechnen, dass zu den bekannten Algorithmen in Zukunft noch performantere hinzukommen werden.

Kapitel 7

Probabilistische Entscheidungsverfahren

7.1 Motivation und Einführung

Die Anforderung des *adaptiven Reasoning*[1] an eine Robotereinheit ist in der Vergangenheit immer stärker gewachsen, z.B. im Bereich der Serviceroboter. Die allgemeine Aufgabe für einen Roboter besteht oftmals darin, eine sehr komplexe Aufgabe in einer sich ständig ändernden Umgebung, mit einer Unmenge an Eigenschaften, zu erfüllen (und damit stoßen die meisten *klassischen* Planungsverfahren an ihre Grenzen, siehe Kapitel 6). Da solche Aufgaben, sowie auch die Umwelt und der Aufbau des Roboters, sehr unterschiedlich sein können, kann man solch ein System nur sehr allgemein beschreiben.

Die Idee eines *rationalen Agenten* spielt hierbei eine zentrale Rolle. Solch eine Einheit kann über Sensoren ihre jeweilige Umgebung wahrnehmen, und anschließend, aufgrund der Umweltbeobachtungen und internen *Policy* (siehe Abschnitt 7.2.3), entsprechende Aktionen durchführen. Der interne reasoning process[2], nach welchem erst Aktionen ausgeführt werden, kann sehr leicht oder auch sehr schwer sein, abhängig vom jeweiligen Szenario. Im folgenden einführenden Kapitel soll nun zuerst die Arbeitsumgebung eines Roboters allgemein charakterisiert werden, und danach das *reasoning system*[3], durch welches Aktionen ausgewählt werden, beschrieben werden.

[1] dt.: anpassungsfähige Beweisführung, Argumentation
[2] dt.: Entscheidungsprozess
[3] dt.: Entscheidungssystem

7.1.1 Umgebungseigenschaften

Es ist wichtig, bevor die eigentliche Aufgabe an den Roboter gestellt wird, seine Arbeitsumgebung systematisch zu beschreiben. Für rationale Agenten existieren nach aktueller Definition [RN03a] folgende Dimensionen:

1. vollständig beobachtbar ↔ teilweise beobachtbar
2. deterministisch ↔ stochastisch
3. episodisch ↔ sequentiell
4. statisch ↔ dynamisch
5. diskret ↔ kontinuierlich
6. ein Agent ↔ mehrere Agenten

Hierbei beschreibt der erste Begriff jeweils eine relativ einfache Umgebung, der Zweite eine komplexere. Reale Umgebungen werden normalerweise ausschließlich durch die zweiten Eigenschaften beschrieben, nur vereinfachte oder abstrakte Arbeitsgebiete werden durch die vereinfachten Eigenschaften charakterisiert.

7.1.2 Reasoning system

Zuerst muss nun der Ausdruck *reasoning system* genau definiert werden, was im Rahmen eines rationalen Agenten erfolgen kann: *Ein reasoning system ist eine Prozedur, welche für einen Agenten Aktionen auswählt, abhängig von der Wahrnehmung und dem Hintergrundwissen, um eine vorher bestimmte Mission auszuführen.*

Um die im vorherigen Abschnitt genannten komplexen Merkmale einer realen, physischen Umgebung in einem reasoning system handhaben zu können, braucht es algorithmische Technologien, die jene explizit berücksichtigt. Es gibt hierbei verschiedene *Meta-Klassen* von Agentenstrukturen: einfache Agenten beschränken sich auf simple Reflexe (und sind damit ungeeignet, um in komplexen Umgebungen eingesetzt zu werden), andere orientieren sich mehr an der, an sie gestellten, Mission, oder an der von ihnen erwarteten Nützlichkeit. Der nachfolgende Abschnitt wird nun Vorgehensweisen und Technologien für letztgenannte Klasse beschreiben.

7.2 Der Markov-Entscheidungsprozess

Im folgenden Abschnitt soll nun eine Möglichkeit beschrieben werden, wie das Problem behoben werden kann, dass Planungssysteme, wie in Kapitel 6 vorgestellt, nicht mit Umgebungseigenschaften wie *dynamisch* oder *stochastisch* umgehen können. Prinzipiell gibt es

hierfür eine große Anzahl an Möglichkeiten, welche hauptsächlich auf Wahrscheinlichkeitsmodellen basieren. Eine sehr nützliche sind dabei die sog. Markov Entscheidungsprozesse[4] (MDPs) [Bel57].

7.2.1 Die Struktur von MDPs

Ein MDP besteht aus:

- Einer Menge $S = \{s_0, s_1, s_2, ...\}$ von Zuständen, welche das Szenario beschreiben (die Umgebung ist zu einem bestimmten Zeitpunkt in genau einem Zustand).
- Einer Menge $U = \{u_0, u_1, u_2, ...\}$ von möglichen Aktionen, die der Agent ausführen kann.
- Einem Übergangsmodell $T(s', u, s)$.
- Einem Belohungsmodell $R(s, u)$.
- Einem Startzustand s_0.

An der Existenz einer Zustandsmenge S kann bereits gesehen werden, dass die Umwelt bei einem MDP diskret (nicht kontinuierlich) ist. Sowohl ein Zustand als auch eine Aktion muss dabei jeweils immer einzigartig sein. Meistens werden dabei logische oder numerische Eigenschaften zur Beschreibung verwendet, ein Zustand z.B. kann so als eine Momentaufnahme der Welt mit bestimmten Eigenschaften gesehen werden. Das Übergangsmodell T ist der wichtigste Teil eines MDP, da dieses den Markov-Prozess modelliert. Es enthält die Wahrscheinlichkeiten für jeden neuen Zustand s', als Folge der Handlung u aus dem Zustand s heraus (daher enthält T eine Wahrscheinlichkeitsmatrix für jede Aktion s). Das Übergangsmodell macht es daher möglich mit einer stochastischen Umgebung zu arbeiten: unsichere Folgen von Handlungen oder auch vollständig zufällige Ereignisse.

Desweitern gilt die sog. *Markov-Eigenschaft* (die den MDPs ihren Namen gibt), welche besagt, dass alle Wahrscheinlichkeiten immer nur vom aktuellen Zustand abhängen und nicht von den bereits vergangenen. Also braucht ein Agent immer nur den aktuellen Zustand und das Übergangsmodell zu kennen, um die Folgen seiner Handlungen abschätzen zu können. Um tatsächlich einzelne Aktionen auszuwählen, bedarf es des Belohungsmodells R, welches angibt, welche Belohung der Agent bekommt während er sich in s befindet und u ausführt. Die in jedem Schritt aufaddierten Belohnungen, welche positiv, negativ oder auch gleich Null sein können, werden *Utility*[5] genannt, wobei die Aufgabe des Agenten nun darin besteht, Aktionen mit hohen Belohnungen auszuwählen, um die Utility zu maximieren.

[4] engl.: markov decision processes
[5] dt.: der Nutzen

7.2.2 Der rationale Agent in einem MDP

Ein MDP arbeitet in diskreter Zeit (also nicht kontinuierlich) und sequentiell, weswegen man einen Agenten einfach in einen MDP einbauen kann. Ein Wahrnehmen - Schlussfolgern - Handeln[6] Zyklus eines MDP-Agenten lässt sich allgemein in folgende Schritte unterteilen:

1. Der Agent nimmt wahr in welchem Zustand er sich zur Zeit befindet.
2. Er sucht diejenige Aktion aus, mit der sich die erwartete, zukünftigen Utility maximieren lässt.
3. Er führt die Aktion aus und die Umwelt geht stochastisch in einen neuen Zustand über. Die Wahrscheinlichkeiten möglicher Folgen hängen vom Übergangsmodell der ausgewählten Aktion ab.
4. Er bekommt, abhängig vom ursprünglichen Zustand und der ausgewählten Aktion, eine Belohnung (oder Strafe) und der Zyklus startet von vorne.

Nun sollen diese vier Schritte etwas genauer beschrieben werden:

Schritt 1: Bei normalen MDPs nimmt der Agent mit seinen Sensoren immer den korrekten Zustand wahr, in dem er sich befindet. Dies setzt voraus, dass die Umgebung stets vollständig beobachtbar ist (vgl. Abschnitt 7.1.1). Für den komplexeren Fall (eine nur teilweise beobachtbare Umgebung) wird eine Erweiterung des MDP in Abschnitt 7.3 vorgestellt.

Schritt 2: Die Aufgabe des Agenten in Schritt zwei, nämlich die zukünftige Utility zu maximieren, ist der interessanteste aber auch komplexeste Schritt. In einer deterministischen Welt, mit einer endlichen Anzahl an Schritten, kann das Maximierungsproblem durch Suchen gelöst werden (siehe Abschnitt 6.4: Planung als Suche): Da jeder Zustand durch eine bestimmte Aktion in genau einen Folgezustand übergeht, kann eine Folge von Aktionen gefunden werden, welche die größte aufaddierte Belohnung hat. Da bei MDPs eine Aktion aber die Umgebung mit unterschiedlichen Wahrscheinlichkeiten in unterschiedliche Zustände überführt, gestaltet sich das Problem hier wesentlich schwieriger und wird im nächsten Abschnitt genauer behandelt.

Schritt 3: Nachdem die Aktion ausgewählt wurde, wird diese nun ausgeführt und die Umgebung macht einen diskreten(!) Zeitschritt. Der nächste Zustand s' hängt vom vorherigen Zustand s und von der ausgewählten Aktion u ab, und ist in einer nicht-deterministischen Umgebung nicht vorherbestimmt. Die Abhängigkeit des Folgezustandes s' wird dabei durch

[6] engl.: perceive - reason - act

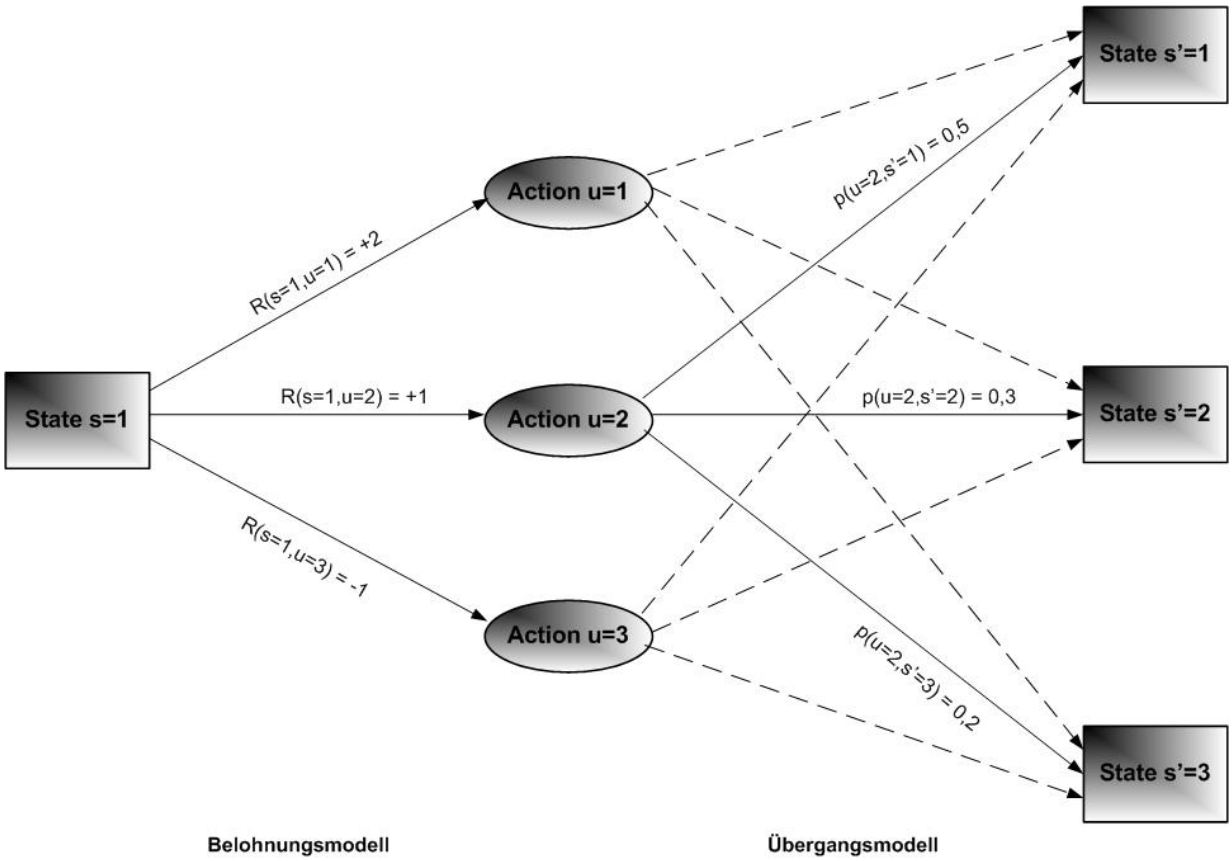


Abbildung 7.1: Schritt 2 und 3 eines MDPs: Die Wahl der nächsten Handlung ist abhängig von der Policy; der Übergang in den nächsten Zustand ist abhängig vom Übergangsmodell.

die entsprechende Spalte $T(*, u, s)$ des Übergangsmodells dargestellt. Hierbei können alle möglichen s' mit einem Wert $T(s', u, s) > 0$ als Folgezustand auftreten, wobei natürlich alle Wahrscheinlichkeiten in der Spalte aufaddiert 1 ergeben müssen.

Schritt 4: Schließlich wird die Belohnung (oder Strafe), u während des Zustandes s ausgeführt zu haben, zur Gesamtbelohnung addiert. Der Agent befindet sich nun im Zustand s'.

Abbildung 7.1 zeigt nun noch einmal graphisch speziell die Schritte 2 und 3.

7.2.3 Reasoning mit MDPs

Nun soll der Prozess eines rationalen Agenten eine Aktion auszuwählen, um eine Aufgabe möglichst gut auszuführen, genauer untersucht werden (dieser Prozess wird *Reasoning* genannt). Es wurde bereits gesagt, dass dies in deterministischen Umgebungen relativ einfach

ist, allerdings wird der Prozess in stochastischen Umgebungen viel komplizierter, da hier die meisten Folgen einer Handlung mit einer Unsicherheit belastet sind.

Zuerst muss untersucht werden, warum die Markov-Eigenschaft von MDPs eine so zentrale Rolle spielt: Bei einem Zustand, welcher alle Informationen enthält, spielt die Vergangenheit keinerlei Rolle, d.h.: Egal was der Agent in der Vergangenheit getan hat, um zum aktuellen Zustand zu gelangen, die nächste, beste Aktion wird immer die gleiche sein! Deswegen ist es für den Agenten nur von Bedeutung, für jeden Zustand eine optimale Aktion zu kennen; solch ein Wissen wird *Policy*[7] genannt. Wichtig ist nun, dass solch eine Policy zu jeder Zeit gültig ist, solange natürlich grundlegende Eigenschaften der Umgebung existent bleiben, sie also nur einmal berechnet werden muss (auch wenn dies kompliziert sein kann). Wenn ein Agent zu jedem Zustand eine Policy hat, weiß er also stets, was als nächstes zu tun ist!

Das Prinzip eines MDPs besagt, dass ein Agent nicht nur wissen muss, was als nächstes zu tun ist, sondern mit welcher Aktion er die erwartete Utility maximieren kann. Eine optimale Policy ist also eine Policy, mit welcher sich die gesamten Belohnungen maximieren lassen. Allerdings kann dies immer nur vermutet werden, da keine Belohnung garantiert werden kann. Es wird also eine Aktion ausgewählt, welche mit größtmöglicher Wahrscheinlichkeit Belohnungen nach sich zieht, wobei mögliche Belohnungen genauso wie ihre dazugehörenden Wahrscheinlichkeiten berücksichtigt werden müssen. Normalerweise werden weit in der Zukunft liegende Belohnungen als unwichtiger betrachtet als nahe liegende, was zu einem Discount-Faktor γ führt (normalerweise knapp unter 1.0). Die Definition einer optimalen Policy π^* ist damit dann wie folgt:

$$\pi^* = argmax_\pi E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, u_t)|\pi\right] \tag{7.1}$$

Wobei t die zukünftigen Zeitschritte darstellt und $R(s_t, u_t)$ die Belohnung zum Zeitpunkt t.

Es existieren nun unterschiedliche Ansätze um diese optimale Policy zu berechnen, wobei, da dies schnell ein sehr komplexes Problem darstellen kann, oftmals nur Näherungen, dessen Fehler beschränkt sind, gesucht werden. Ein gut anwendbarer Ansatz ist die *Value-Iteration* [How60], welche sich gut erweitern lässt, um mit vielen MDP-Erweiterungen arbeiten zu können. Die grundsätzliche Idee ist dabei, dass jeder Zustand einen bestimmten Nutzwert[8] besitzt, welcher der erwarteten Utility entspricht, die durch die folgende Zustandsequenz erreicht werden kann. Wenn ein Agent also die optimale Policy π^* befolgt, entspricht dieser Nutzwert genau der maximal zu erwarteten Utility, vom jetzigen Zustand aus in die Zukunft.

Mit so einem iterativen Ansatz kann diese Definition, unter Annahme einer optimalen Policy, erweitert werden: Die Utility eines bestimmten Zustandes ist die Belohnung für die Ausführung der besten Aktion (in diesem Zustand) plus die Utilities von allen Folge-

[7] dt.: Verfahrensweise, Taktik

[8] engl.: utility value

zuständen multipliziert mit ihren Auftretenswahrscheinlichkeiten. Daraus ergibt sich folgende Formel:

$$U(s) = R(s) + \gamma max_u \sum_{s'} T(s', u, s)U(s') \tag{7.2}$$

Der Algorithmus solch einer Value-Iteration kann also die Utilities aller Zustände berechnen, indem er zuerst alle Werte zufällig belegt, und die Gleichung dann lokal für jeden Zustand iterativ löst (bis die Änderungen unter einer vorgegebenen Konvergenzschranke liegen). Wenn die Utilities für alle Zustände ausreichend gut bekannt sind, lässt sich die beste Aktion max_u leicht aus der rechten Seite von Gleichung 7.2 herleiten.

7.2.4 Anwendbarkeit von MDPs in realen Umgebungen

Es wurde bis jetzt gezeigt, dass sich MDPs gut von Agenten in stochastischen Umgebungen einsetzten lassen. Nun sollen noch andere Umgebungseigenschaften im Hinblick auf ihre Anwendbarkeit bei MDPs untersucht werden. Rationale Agenten, welche einen MDP benutzen, können gut in Umgebungen handeln, die stochastisch, dynamisch und sequentiell sind:

- **dynamisch:** Der typische Wahrnehmen - Schlussfolgern - Handeln Zyklus erlaubt es dem Agenten die Folgen einer Handlung zu beobachten, welche ausgeführt wurde, obwohl die Folgen ungewiss waren.
- **sequentiell:** Das Aufaddieren aller in der Vergangenheit erhaltenden Belohnungen und das Entscheiden über alle möglichen zukünftigen Folgen kommen mit der Eigenschaft *sequentiell* klar.

Damit haben MDPs zwei weitere Vorteile gegenüber klassischen Planungsverfahren. Allerdings gibt es auch zwei Umgebungseigenschaften, mit denen auch MDPs nicht zurechtkommen: nur teilweise beobachtbar und kontinuierlich:

- **kontinuierlich:** Der endliche Zustandsraum gliedert die Welt in einen diskreten Raum, und die diskreten Zeitschritte eines MDPs benötigen ein diskretes Wahrnehmen der Zeit.
- **teilweise beobachtbar:** Der Agent muss mit seinen Sensoren immer exakt wahrnehmen können, in welchem Zustand er sich zurzeit befindet. Dies schließt nur teilweise beobachtbare Umgebungen aus.

Für Umgebungen, welche nur teilweise beobachtbar sind, stellt der nächste Abschnitt eine Erweiterung der MDPs vor, die mit dieser Eigenschaft umgehen können.

7.3 Teilweise beobachtbare MDPs

Es gibt eine Erweiterung für MDPs, welche in nur teilweise beobachtbaren Umgebungen einsetzbar ist: teilweise beobachtbare Markov- Entscheidungsprozesse[9] (POMDPs) [Son71]. Mit POMDPs können zwar unsichere Zustandswahrnehmungen gehandhabt werden, sie sind allerdings, wie wir sehen werden, auch schwerer zu benutzen und haben zwangsläufig eine komplexere Struktur.

7.3.1 Die Struktur von POMDPs

Ein POMDP besitzt einige, notwendige Erweiterungen im Gegensatz zu klassischen MDPs, um in nur teilweise beobachtbaren Umgebungen arbeiten zu können. Ein POMDP besteht demnach aus:

- Einer Menge $S = \{s_0, s_1, s_2, ...\}$ von Zuständen, welche das Szenario beschreiben (die Umgebung ist zu einem bestimmten Zeitpunkt in genau einem Zustand).
- Einer Menge $U = \{u_0, u_1, u_2, ...\}$ von möglichen Aktionen, die der Agent ausführen kann.
- Einer Menge $M = \{m_0, m_1, m_2, ...\}$ von Messgrößen, welche der Agent wahrnehmen kann.
- Einem Übergangsmodell $T(s', u, s)$.
- Einem Beobachtungsmodell $O(m, s)$.
- Einem Belohungsmodell $R(s, u)$.
- Einem Startzustand s_0.

Also besitzt ein POMDP alle strukturellen Elemente eines MDPs, allerdings noch zwei zusätzliche, um mit dem Nachteil zurechtzukommen, nicht jeden Zustand korrekt wahrnehmen zu können. Die Idee eines POMDPs ist nun die, dass die Zustände eines MDPs weiterhin als tatsächliche Gegebenheiten der Umgebung existieren, der Agent diese allerdings nicht mehr direkt wahrnehmen kann. Anstelle der bei MDPs wahrnehmbaren Zustände, treten nun die sog. Messgrößen M, welche über das Beobachtungsmodell $O(m, s)$ mit den tatsächlichen Zuständen in Beziehung stehen. Dieses Beobachtungsmodell ist eine weitere Matrix, in der die Wahrscheinlichkeiten stehen, dass der tatsächliche Zustand der Umgebung s ist, wenn der Agent die Messgröße m wahrnimmt. Mit dieser Definition ist es möglich, alle Unsicherheiten einer teilweise beobachtbaren Umgebung, wie z.B. Sensorfehler, nicht wahrnehmbare Umgebungseigenschaften oder allgemeine Wissensbeschränkungen des Agenten, zu modellieren.

[9] engl.: partially observable markov decision processes

Diese Fähigkeit macht den POMDP auf der einen Seite flexibler einsetzbar als einen klassischen MDP, auf der anderen Seite geht aber auch die zentrale Eigenschaft verloren, dass der Agent stets weiß in welchem Zustand er ist, und daraus Handlungen ableiten kann. Da ein Agent mit einem POMDP allerdings auch stets Wissen über seinen aktuellen Zustand benötigt, der tatsächliche Zustand aber nicht vollständig wahrnehmbar ist, benötigt er einen Ersatz dafür: den *vermuteten Zustand*[10]. Dieser ist eine diskrete Wahrscheinlichkeitsverteilung (auch *Histogramm* genannt) und eine sekundäre Eigenschaft eines POMDPs. Jede Wahrscheinlichkeit in dem Histogramm eines bestimmten vermuteten Zustandes beschreibt wie stark der Agent glaubt, der entsprechende tatsächliche (reale) Zustand sei wahr. Bei einem POMDP besteht also weiterhin die fundamentale Eigenschaft, dass sich die Umgebung zu jeder Zeit in genau einem Zustand befindet, der Agent weiß allerdings normalerweise nicht mehr sicher, welcher das ist.

7.3.2 Die Berechnung des vermuteten Zustandes

Nun wollen wir uns der, im Vergleich zur trivialen Zustandswahrnehmung eines MDPs, komplexen Berechnung des vermuteten Zustandes widmen. Dieser hängt natürlich einerseits von der aktuellen Beobachtung des Agenten ab, allerdings ebenso von dem zuletzt vermuteten Zustand und von der zuletzt ausgeführten Aktion. Dies führt zum Konzept des sog. *bayesschen Filters*, dessen Idee es ist, zwei Arten des Informationsgewinnes zu vereinen, um eine Genauigkeit zu erzielen, welche jede der beiden Arten für sich nicht erreichen kann. Die zwei Arten sind:

- Das **Wahrnehmen** der Messgrößen und das direkte Ableiten des vermuteten Zustandes aus diesen.
- Eine **Vorhersage** des Umgebungszustandes aufgrund des alten vermuteten Zustandes, indem die Wahrscheinlichkeiten des Übergangsmodells betrachtet werden.

Die Vorhersage erfolgt dabei iterativ, beruhend auf dem letzten vermuteten Zustand, das Wahrnehmen dagegen direkt. Der bayessche Filter vereint nun durch folgende Formel diese beiden Arten für jedes Element der Wahrscheinlichkeitsverteilung des vermuteten Zustandes:

$$b'(s') = \alpha O(m, s') \sum_s T(s', s, s) b(s) \tag{7.3}$$

Diese Art von bayesschen Filtern werden auch Vorwärts-Filter[11] genannt, so dass die Formel 7.3 auch $forward(b, u, m)$ genannt werden kann. In dieser ist α eine Normalisierungskonstante, und der vermutete Zustand wird in jedem Prozessschritt neu berechnet. Mit

[10] engl.: belief state
[11] engl.: forward filtering

diesem Wissen kann der Informationsfluss eines POMDPs als ähnlich wie der eines MDPs angesehen werden.

7.3.3 Der rationale Agent in einem POMDP

Bei einem MDP war die optimale Aktion für den Agenten nur vom aktuellen Zustand abhängig, bei einem POMDP tritt nun an diese Stelle der vermutete Zustand. Daher ist einerseits die Markov-Eigenschaft (siehe Abschnitt 7.2.1) immer noch gültig, andererseits besteht der Wahrnehmen - Schlussfolgern - Handeln Zyklus aus ähnlichen Schritten wie bei einem MDP-Agenten:

1. Während der Wahrnehmung erfasst der Agent die Messgröße m.
2. Der Agent berechnet den neuen vermuteten Zustand aus dem alten vermuteten Zustand, der letzten Aktion sowie den neuen Messgrößen.
3. Der Agent schlussfolgert, mit welcher Aktion sich die erwartete, zukünftige Utility maximieren lässt, abhängig vom neuen vermuteten Zustand.
4. Der Agent führt die Aktion aus und die Umgebung geht stochastisch in einen neuen, tatsächlichen Zustand über, abhängig vom Übergangsmodell der entsprechenden Aktion.
5. Der Agent bekommt eine Belohnung (oder Bestrafung), abhängig vom letzten, tatsächlichen Zustand und der Aktion. Der Zyklus startet von vorne.

Man kann bereits hier eine wichtige Eigenschaft dieser Art des Reasonings erkennen: der Agent macht seine nächste Aktion immer nur von seinem vermuteten Zustand abhängig. Wenn also dieser in Wahrheit sehr stark vom tatsächlichen Umgebungszustand abweicht, kann seine ausgewählte Aktion sehr schlecht sein und nicht die erwünschten Folgen haben.

Ein weiterer Unterschied zu klassischen MDPs, der den Umgang mit einem POMDP komplizierter macht, ist folgender: Die Policy eines MDPs besagt, dass es für jeden Zustand der endlichen Zustandsmenge S genau eine optimale Aktion u gibt. Bei POMDPs dagegen ist der Raum der vermuteten Zustände kontinuierlich, welcher so viele Dimensionen besitzt, wie es tatsächliche Zustände gibt. Die Werte in jeder dieser Dimensionen liegen zwischen 0 und 1, da jede mögliche Wahrscheinlichkeitsverteilung ein Punkt im Vermutungsraum ist. Deswegen können die vermuteten Zustände auch als „Vermutungspunkte“ im Vermutungsraum interpretiert werden. Wegen diesem kontinuierlichen Vermutungsraum kann es keine Policy geben, die bestimmten Zuständen eine fest vorgeschriebenen Aktion zuweist.

Desweiteren kann man einen Unterschied zu einem MDP erkennen, wenn der Agent von seinem aktuell vermuteten Zustand in die Zukunft blickt: Er muss nun nicht nur die Wahrscheinlichkeiten des Übergangmodells beachten, sondern auch die Tatsache, dass es im Zustandraum Zustände geben kann, welche er besser wahrnehmen kann als andere.

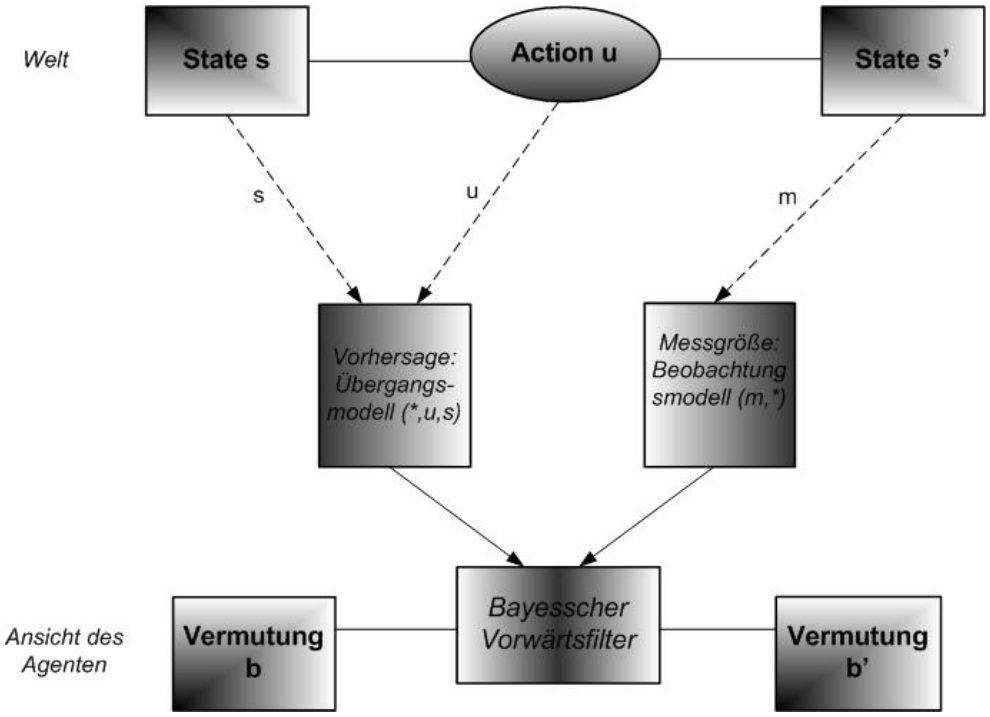


Abbildung 7.2: Der Informationsfluss eines POMDPs (ohne Entscheidung über nächste Aktion). Man kann den Zusammenhang zwischen dem tatsächlichen und dem vermuteten Zustand erkennen (Bayessscher Vorwärtsfilter).

Man kann also erkennen, dass ein POMDP, obwohl er ähnliche Strukturen und gleiche Grundideen wie ein MDP verwendet, sehr viel komplexer ist. Abbildung 7.2 zeigt nun graphisch die einzelnen Schritte eines POMDP.

Der nächste Abschnitt soll nun im Detail den Reasoning-Prozess eines POMDP erklären.

7.3.4 Reasoning mit POMDPs

Genau wie bei MDPs bedeutet Reasoning wieder, eine optimale Policy zu finden und zu verwenden, und ebenfalls wie bei MDPs existiert eine Value-Iteration für POMDPs (siehe Abschnitt 7.2.3). Allerdings muss die Idee etwas abgeändert werden, und die Berechnung gestaltet sich wegen dem kontinuierlichen Vermutungsraum schwieriger. Dieser führt nämlich zu einer Policy, welche einen kontinuierlichen Raum abdecken muss, und die durch eine stückweise lineare Value-Funktion über dem Vermutungsraum dargestellt wird. Stückweise ist sie, weil es verschiedene Aktionen für den Agenten gibt, und linear, wegen dem linearen Erwartungsoperator (eine mathematische Herleitung hierfür kann man in [TBF05] finden).

Nun wollen wir diese Value-Funktion etwas genauer betrachten: Diese setzt sich aus Teilstücken von linearen Funktionen, definiert über dem Vermutungsraum mit $|Zustaende|$ Dimensio-

nen, zusammen. Jede lineare Funktion gehört hierbei zu genau einer Aktion; die ganze, zusammengesetzte Value-Funktion stellt immer das Ergebnis des Algorithmus der Value-Iteration dar, welcher eine gewisse Anzahl an Zeitschritten (auch *Horizont*[12] genannt) in die Zukunft berechnet hat. Also sind POMDPs, im Gegensatz zu MDPs, nicht unabhängig von der Anzahl an betrachteten Zeitschritten. Der Funktionswert, an einem bestimmten Vermutungspunkt, ist die erwartete, zukünftige Utility, wenn der Agent im entsprechenden, vermuteten Zustand ist und die Aktion ausführt, welche zur linearen Funktion gehört (es werden dabei immer dem Horizont entsprechend viele Zeitschritte in die Zukunft betrachtet). Zusammenfassend lässt sich sagen: Die bestmögliche Aktion, die der Agent in einem bestimmten vermuteten Zustand ausführen kann, ist die, die zu derjenigen linearen Funktion gehört, welche an dieser Stelle den größten Funktionswert hat. Durch diese Definition der Value-Funktion, welche alle Teilstücke linearer Funktionen - die sich als Hyperebenen visualisieren lassen - enthält, wobei diese Teilstücke wiederum das Maximum über bestimmten Bereichen des Vermutungsraumes sind, entsteht eine konvexe Funktion.

Am Besten lässt sich das Prinzip solch einer Value-Funktion an einem graphischen Beispiel verstehen, so ist in Abbildung 7.3 ein Beispiel mit nur zwei Zuständen zu sehen. Hierbei

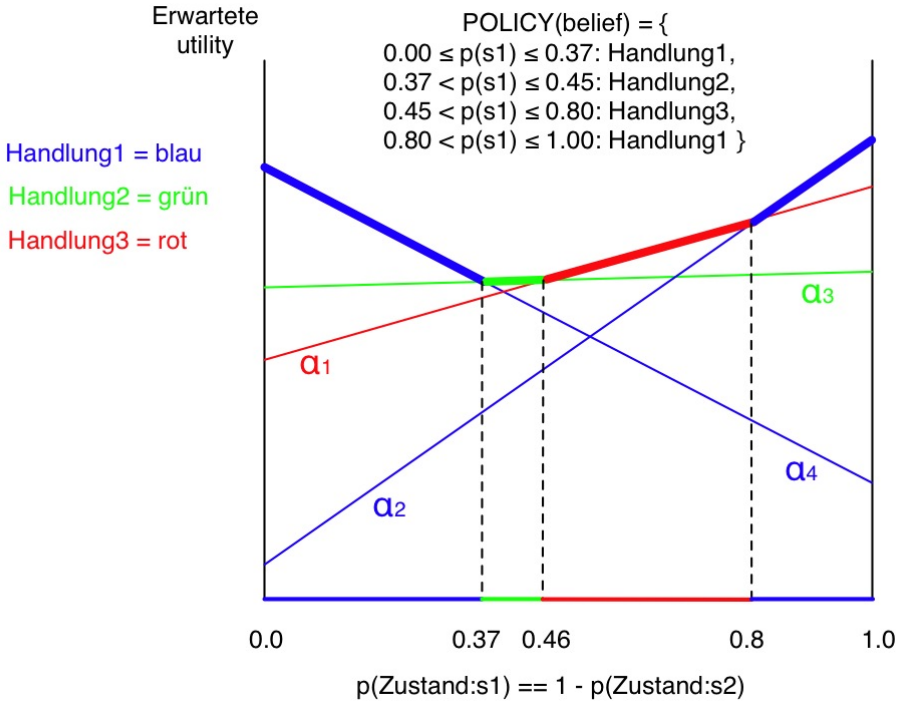


Abbildung 7.3: Eine Policy für einen POMDP mit zwei Zuständen und drei Aktionen. Vier Hyperebenen (alphas) sind zu sehen.

wird der einfache Zusammenhang $p(state1) + p(state2) = 1$ ausgenutzt, um den zweidi-

[12]engl.: horizon

mensionalen Vermutungsraum ($dim(Vermutungsraum) = |Zustaende| = 2$) in nur einer Dimension darzustellen. Sobald die Anzahl der Zustände allerdings größer ist, wird es nahezu unmöglich dies noch übersichtlich zu visualisieren. Abbildung 7.3 zeigt hierbei gut, wie der $max()$ Operator (fett gedruckte Kurve) eine konvexe Menge aus Teilstücken der Hyperebenen bildet.

Also lässt sich bei solch einem Value-Iterationsansatz die Policy des Agenten durch eine stückweise lineare, konvexe Funktion darstellen. Wenn solch eine Policy existiert, lässt sich die optimale Aktion für einen Agenten in einem bestimmten vermuteten Zustand leicht finden: Zuerst wird der Funktionswert an diesem vermuteten Zustand für jede lineare Funktion, welche Teil der Value-Funktion ist, berechnet. Die Aktion, welche zur Hyperebene mit dem höchsten Wert gehört, hat die größte erwartete Utility innerhalb des Horizonts, und ist deshalb optimal. Weitaus schwerer als diese Policy anzuwenden, ist allerdings das Aufstellen der Value-Funktion, was im nun folgenden Abschnitt beschrieben werden soll.

7.3.5 Exakte Value-Iteration für POMDPs

Der erste Schritt der Value-Iteration für POMDPs [CKL94] ist, für jede vom Agenten ausführbare Aktion eine lineare Funktion zu setzen. Der „Eckwert“ von solch einer initialisierenden, linearen Funktion (also der Wert an der Stelle $(0_1, ..., 0_{i-1}, 1_i, 0_{i+1}, ...0_n)$), gibt die sofortige Belohnung an, wenn in Zustand i die Aktion u ausgeführt wird, welche zur Hyperebene gehört. Die Hyperebene erstreckt sich zwischen diesen Ecken, da das Risiko von dazwischen liegenden Wahrscheinlichkeitsverteilungen (Vermutungspunkte) gegen alle möglichen Belohnungen einer Aktion gewichtet wird. Alle initialisierenden Funktionen entsprechen hierbei jeweils einem Horizont von 1, und die Policy ist dann der $max()$ Operator, angewandt auf die gesamte Menge.

Der Algorithmus der Value-Iteration schaut nun iterativ je einen Zeitschritt in die Zukunft, indem die Wahrscheinlichkeiten für alle Aktionen und für die Beobachtungen von diesen berechnet werden. Da der Vermutungsraum kontinuierlich ist, werden alle linearen Funktionen G mit den Wahrscheinlichkeiten von jedem möglichen Übergangs-Beobachtungspaar multipliziert. Dadurch entsteht eine lineare Hilfsfunktion $V_{k,u,m}$ für jede lineare Funktion k, Aktion u und Messgröße m. Es muss beachtet werden, dass bei jedem Schritt nach der Initialisierung, obwohl jede Funktion zu genau einer Aktion gehört, mehrere Funktionen entstehen können, wegen verschiedener Zeitlinien der Events. Die lineare Hilfsfunktion $V_{k,u,m}$ lässt sich durch folgende Formel berechnen:

$$V_{k,u,m} = \sum_{j=1}^{N} \sum_{i=1}^{N} v_{k,i} p(m|x_i) p(x_i|u, x_j) \tag{7.4}$$

N entspricht dabei der Anzahl der Zustände, $p(m|x_i)$ der entsprechende Beobachtung und $p(x_i|u, x_j)$ der gemachten Vorhersage. Also wird die lineare Funktion $V_{k,u,m}$ durch Vorwärtsfiltern aus der alten linearen Funktion k gewonnen (mathematisch also eine Pro-

jektion). Da dieser Schritt, wie gesehen, nicht nur eine lineare Funktion sondern mehrere berechnet, scheint der Rechenaufwand dadurch drastisch zu steigen. Allerdings zeigt die reale Anwendung dieses Algorithmus, dass das Aufstellen der endgültigen Menge linearer Funktionen das weit aus zeitintensivere Problem darstellt. Deshalb muss für jede Messgröße $m \in M$ eine Hilfsvariable $k(m)$ definiert werden, für dessen Werte gilt: $1 \leq k(m) \leq |G_{alt}|$. Für jede mögliche Kombination der Werte der $k(m)$ (also des Vektors $(k(1), ..., k(|M|))$ wird nun eine neue lineare Funktion G' berechnet, wobei alle Belohnungen bis zu diesem Zeitpunkt berücksichtigt werden, plus die Belohnung für diesen Schritt:

$$\forall i \leq N : G'_{u,i} = \left[r(x_i, u) + \sum_z v_{k(m),u,m,i} \right] \tag{7.5}$$

Diese neue lineare Funktion G' wird nun der alten Menge an linearen Funktion zugefügt. Dabei kann diese Vergrößerung der Funktionenmenge die Policy natürlich nur verbessern, niemals aber verschlechtern (anders ausgedrückt: die neue Menge an linearen Funktionen ist besser oder mindestens gleichgut wie die alte). Zum besseren Verständnis nun wieder eine graphische Veranschaulichung (Abbildung 7.4): Im Vergleich zu Abbildung 7.3

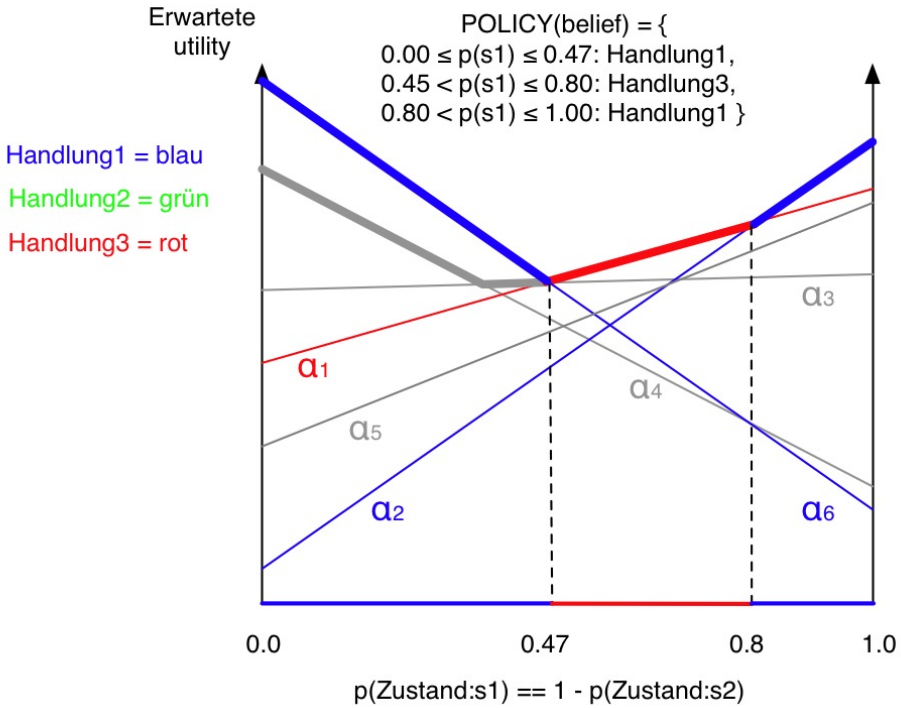


Abbildung 7.4: Ein Schritt der Value-Iteration: Zwei neue Hyperebenen sind zu sehen.

sieht man nun zwei neue, durch den Value-Iterationsschritt entstandene, Hyperebenen α_5 und α_6. Während α_5, wie zu sehen, nirgends das Maximum von allen Funktionen besitzt

(und deshalb weggelassen werden kann), verändert α_6 die Policy (und macht α_3 und α_4 überflüssig).

Das Problem solch einer exakten Value-Iteration wurde bereits angesprochen: die sehr große Anzahl an neuen Funktionen G, welche in jedem Schritt zur Funktionenmenge dazukommen. Die Anzahl an neuen Funktionen in jedem Iterationsschritt beträgt hierbei $|G_{alt}|^m$. Die Anzahl der linearen Funktionen $|G|$ nach t Schritten für Aktion u und Beobachtung m beträgt:

$$|G_t| = |u|^{(\sum_{i=0}^{t-1} |m|^i)} \tag{7.6}$$

Daraus folgt, dass die exakte Value-Iteration zur Komplexitätsklasse 2-EXPSPACE gehört.

Dieser viel zu große Aufwand macht die exakte Value-Iteration komplett unbrauchbar für reale Anwendungen eines POMDP-Agenten. Auch wenn viele Funktionen in jedem Iterationsschritt vollständig unbeachtet bleiben können, da sie für die Policy keinerlei Rolle spielen, bleibt dieser Algorithmus immer noch viel zu langsam. Lange Zeit wurden POMDPs deswegen nicht für rationale Agenten verwendet, allerdings wurde in den letzten Jahren eine neue Art der Value-Iteration entwickelt, welche POMDPs für rationale Agenten anwendbar macht: die approximativen Verfahren. Der nächste Abschnitt stellt nun den wichtigsten Approximationsalgorithmus vor.

7.3.6 Punktbasierte Value-Iteration für POMDPs

Die punktbasierte Value-Iteration[13] (PBVI) [PGT04] vereinfacht die Value-Iteration eines POMDPs, indem die erwartet Utility nur noch an bestimmten Punkten berechnet wird (ähnlich wie bei MDPs). Dadurch ist es nicht mehr nötig aus linearen Funktion so viele neue Funktionen zu projizieren, wie es Kombinationen an Messgrößen gibt. Die prinzipielle Eigenschaft der Value-Funktion, wie im vorherigen Abschnitt, nämlich eine stetige, stückweise lineare, konvexe Funktion zu sein, bleibt hierbei erhalten, es wird allerdings nur noch eine Stichprobenmenge aller Vermutungspunkte berücksichtigt. Die nun entstehende Funktion kann wiederum als Policy für alle Punkte angesehen werden, nicht nur für die aus der Stichprobe. Da dies ein approximativer Ansatz ist, kann man auch kein optimales Ergebnis erwarten, wie bei der exakten Value-Iteration. Allerdings hat er einen beschränkten Fehler und als großen Vorteil ist er in der Praxis gut einsetzbar, da der Rechenaufwand erheblich verkleinert wurde.

Der erste Schritt bei dieser Iteration besteht also nun darin, eine Untermenge der Vermutungspunkte zu wählen, in welchen wieder lineare Funktionen berechnet werden, welche an diesen Vermutungspunkten die beste Utility (größten Funktionswert) haben. Danach wird die lineare Hilfsfunktionen $V_{k,u,m}$ durch die gleiche Formel 7.4 wie bei der exakten Value-Iteration berechnet. Allerdings müssen nun nicht mehr alle Projektionen, also alle Kombinationen der Messgrößen, berücksichtigt werden. Es wird zwar nun die Utility

[13] engl.: point based value iteration

der Hilfsfunktionen für jede Aktion und Messgröße in jedem Vermutungspunkt berechnet, allerdings wird jeweils nur diejenige lineare Funktion mit der größten Utility an einem Vermutungspunkt gespeichert. Ein Vorteil davon ist z.B.: Oftmals haben benachbarte Punkte die gleiche lineare Funktion, wodurch sich die Anzahl der benötigten, linearen Funktionen drastisch reduzieren lässt. Man kann diesen Algorithmus nun wieder iterativ in die Zukunft fortsetzen, allerdings besteht bei der PBVI nun noch eine weitere Möglichkeit die Qualität der entstandenen Policy zu verbessern: man erhöht die Anzahl der Punkte in der gewählten Untermenge.

Diese Punkte werden dabei allerdings nicht wahllos, und auch nicht nach einer einfachen, geometrischen Vorschrift gezogen. Es ist nämlich oft der Fall, dass bestimmte Punkte im Vermutungsraum in der Ausführung nie erreicht werden, und diese Tatsache wird bei der Wahl der Punkte mitberücksichtigt. Daher wird ein anderer Ansatz gewählt, indem für jede mögliche Aktion eine Stichprobenentnahme auf die Vorwärtsfilterung durch Übergangs- und Beobachtungsmodell angewendet wird. Dadurch wird ein Übergang von einem alten Vermutungspunkt in einen neuen simuliert. Dies geschieht durch folgende Schritte:

1. Eine Stichprobenentnahme eines wirklichen Zustandes s aus der aktuellen Vermutungszustandverteilung, durch einen multinomiellen Stichprobenzieher $b(*)$.

2. Eine Stichprobenentnahme eines neuen, wirklichen Zustandes s' aus dem Übergangsmodell, unter Beachtung der Aktion und des gezogenen Zustandes s, durch einen multinomiellen Stichprobenzieher $T(*, u, s)$.

3. Eine Stichprobenentnahme einer Beobachtung m, des neuen, wirklichen Zustandes s', aus dem Beobachtungsmodell, durch einen multinomiellen Stichprobenzieher $O(*, s'$.

4. Durch Vorwärtsfilterung der drei Stichproben entsteht aus dem alten Vermutungszustand ein neuer $b'(s') = forward(b, u, m)$.

Es werden nun nicht alle neuen Vermutungspunkte (einer für jede Aktion und jeden alten Vermutungspunkt) gespeichert, sondern für jede alten Punkt nur derjenige, der den größten Abstand bezüglich der standard Euklidnorm im Vermutungsraum hat. Dadurch erreicht man, dass die Vermutungspunkte gleichmäßig im Raum verteilt werden, aber gleichzeitig auf Regionen beschränkt, welche vom Szenario überhaupt erreicht werden.

Man kann also bei der PBVI einerseits durch fortführende Iteration immer weiter die entstehende Policy verbessern, andererseits auch durch Vergrößern der berücksichtigten Vermutungspunktmenge. Dadurch entsteht ein sehr gut anwendbares Verfahren, welches die Policy stetig verbessert. Beweise zur Abschätzung des Fehlers und der Komplexität der Value- und Vermutungspunktiteration kann man zusätzlich in [PGT04] finden.

7.4 Zusammenfassung

Wie man gesehen hat, sind klassische Planungsverfahren meistens ungeeignet, um in komplexen Umgebungen eingesetzt werden zu können. Mit den POMDPs existiert ein Ansatz, um rationale Agenten in teilweise beobachtbaren, zufälligen, sequentiellen und dynamischen Arbeitsumgebungen einzusetzen. Allerdings müssen diese auch immer diskret handhabbar sein, d.h. sobald die zeitliche Wahrnehmung kontinuierlich sein muss, stoßen auch POMDPs an ihre Grenzen. Es existieren zwar Erweiterungen für POMDPs um auch diesen Nachteil auszugleichen, allerdings verschlechtert sich die Anwendbarkeit drastisch, sobald man sich von den diskreten Modellen löst. POMDPs sind also zurzeit ein weit verbreitetes Forschungsgebiet, in welchem auch in nächster Zeit noch etliche Neuerungen erscheinen dürften, allerdings sind die tatsächlichen Anwendungen von ihnen in der Robotik noch stark begrenzt.

Kapitel 8

Graphische Simualtionstechniken

Bei der Roboterprogrammierung ist es oftmals sehr hilfreich, vor dem eigentlichen Ausführen des Programms eine *Simulation* durchzuführen. Betrachtet man z.B. den Zweck eines Simulators zum Testen neuer Flugzeuge oder neuer Flugzeugtechniken, merkt man schnell, dass in manchen Anwendungsgebieten Simulationen sogar unabdingbar sind; in diesem Fall aus Sicherheitsgründen.

Im folgenden Kapitel sollen graphische Simulationstechniken mit Anwendungen in der Robotik betrachtet werden.

8.1 Definition

Nach VDI-Richtlinie 3633 ist Simulation die Nachbildung eines dynamischen Prozesses in einem Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind.

Prinzipiell muss man zwischen der *Simulation* und der *Emulation* unterscheiden:

- **Simulation:** Nachbildung von Vorgängen auf einer Rechneranlage auf der Basis von Modellen.
- **Emulation:** Ein Spezialfall der Simulation, bei der das Verhalten einer Maschine durch ein anderes System *komplett* nachgebildet wird.

Der entscheidende Unterschied hierbei ist, dass bei der Emulation das System **komplett** nachgebildet wird; bei der Simulation werden oftmals nur relevante Teile des Systems nachgebildet, oder es werden, um den Aufwand zu reduzieren, komplexe Bereiche vereinfacht (wie z.B. komplexe physikalische Gesetze).

In der Realität werden Simulationen zu verschiedenen Zwecken verwendet:

Der Simpelste ist das einfache Verstehen des zu simulierenden Systems. Dies geschieht z.B. bei der Analyse von physikalischen oder chemischen Prozessen. Einen Schritt weiter geht

die Prädiktion von Systemverhalten, was bei der Entwicklung von technischen Systemen oftmals eine Rolle spielt. Eine Kopplung dieser beiden Zwecke (Verstehen & Prädiktion) kommt bei der Simulation von Finanzmärkten, oder bei der Analyse und Vorhersage von Wirtschaftsentwicklungen zum Tragen. Als letzter Punkt sei hier noch die Reproduktion realer Systeme genannt, zu sehen beispielsweise bei Flugsimulatoren oder medizinischen Trainingscentern.

Man kann bei Simulationen verschiedene *Simulationsarten* unterscheiden:

- Bei der **Diskreten Simulation** sind die Zustände des zu simulierenden Systems diskret.
- Im Gegensatz dazu sind bei der **Kontinuierlichen Simulation** die Zustandsänderungen kontinuierlich (unendlich viele Zustandsänderungen, wie z.B. in naturwissenschaftlichen Modellen).
- Bei **Dynamischen Simulationen** spielen die zeitlichen Verläufe und Rückkopplungseffekte eine relevante Rolle (Weltmodelle).
- Bei der **Statischen Simulation** schließlich wird ein zeitunabhängiges Verhalten der Modelle beobachtet.

8.2 Simulation von Robotersystemen

Für die Robotik ist es oft unabdingbar, Robotersysteme zu simulieren. Dabei gibt es, im Vergleich zu anderen Simulationen, einige Besonderheiten:

- *Eingeschränkte Domäne:* Da, bei impliziter Programmierung, oft die gesamte Umgebung des Roboters modelliert werden muss, versucht man bei einer Simulation die Komplexität gering zu halten; so wird die Umwelt oftmals nur eingeschränkt modelliert.
- Es werden oft *Roboteremulatoren* verwendet, um komplette Robotersysteme originalgetreu simulieren zu können.
- Abermals um die Komplexität gering zu halten kommen bei den *Sensoren* des Roboters meistens Simulationen zum Einsatz, und keine Emulationen.

Ziel dabei ist eine Validierung des Programms vor seiner eigentlichen Ausführung, was gegebenenfalls die Kosten senken kann. Ebenfalls wird versucht, eine möglichst einfache Simulation zu erreichen, um unter anderem den Rechneraufwand gering zu halten.

Die Gründe für Simulationen von Robotersystemen sind folgende:

- **Kosten:** Oft ist es zu teuer ein Robotersystem in der Realität zu testen, z.B. wegen möglichem Verschleiß der Hardware.
- **Zeit:** Simulationen sind meistens schneller durchzuführen als reale Systeme.
- **Gefahren:** Wie am Beispiel des Flugsimulators zu sehen ist, sind Simulationen oft gut um Gefahren für Mensch und Maschine zu erkennen und zu vermeiden.
- **Validierung:** Durch Simulationen lässt sich oft erkennen, ob richtig programmiert wurde, bzw. ob das Programm auch das Gewünschte leistet.

8.2.1 Komponenten

Die Simulation eines kompletten Robotersystems ist ein sehr komplexer Vorgang, an dem sehr viele unterschiedliche Komponenten beteiligt sind. Es gibt unterschiedliche Möglichkeiten diese in verschiedene Bereiche einzuteilen; es wird nun versucht, einen Überblick über alle Komponenten einer Simulation zu geben. Je nach Art des zu simulierenden Systems werden eventuell nicht alle Komponenten benötigt.

Eine große Gruppe der Simulationskomponenten sind die **Physikalischen Komponenten**, welche man weiter in *aktive* und *passive Komponenten* unterteilen kann. Zu den *Aktiven Komponenten* zählen unter anderem jegliche Manipulatoren, Effektoren, Fahrzeuge und Sensoren. Im Gegensatz dazu sind *Passive Komponenten* nicht-programmierbare Gegenstände, welche durch Geometrie und Position in der Simulation/Emulation repräsentiert werden. Beispiele davon sind z.B. Werkstücke, Magazine, stationäre Hindernisse und Montagetische.

Besonders bei den aktiven Komponenten wird meistens noch eine *Zeitgeber-Komponente* benötigt, besonders dann, wenn mehrere von solchen aktiven Komponenten verteilt installiert sind. Diese sorgt für eine Taktung zwischen diesen Komponenten, und regelt dadurch den zeitlichen Ablauf der Simulation.

Logische Komponenten sind dagegen, wie der Name schon sagt keine physikalisch existierenden Objekte. Dazu zählen unter anderem *Planungskomponenten*, wie die Bahnplanung, die Kollisionsvermeidung oder auch die Planung von Trajektorien und Griffen. Ebenfalls passiv sind *Steuerungskomponenten*, wie z.B. Programmierschnittstellen.

Oftmals ist es sehr hilfreich während der Simulation den aktuellen Zustand auszugeben, um dem Anwender eine Kontrollmöglichkeit zu geben. Solch eine *Animationskomponente* sorgt für eine graphische Visualisierung des Zustandes, so dass während der Simulation jederzeit zu sehen ist, was für Ergebnisse die Simulation liefert.

Notwendig bei einer Robotersimulation (oder Emulation) ist auch fast immer eine *Interaktion* zwischen Anwender und Simulationssystem. Dazu zählt z.B. eine Eingabeschnittstelle für den Anwender, die es ihm erlaubt die Robotersimulation jederzeit zu steuern. Außerdem sollte es möglich sein Status- und Fehlermeldungen empfangen und ausgeben zu können.

Da vor allem komplexe, aufwendige Simulationen sehr viel Rechnerleistung benötigen, ist es oft sinnvoll, die Arbeit auf mehrere Prozessoren zu verteilen. Dann ist eine *parallele Kopplung* zwischen den Computern notwendig.

8.2.2 Anforderungen

Wie bereits bei den möglichen Komponenten einer Simulation angedeutet, wird an ein Simulationssystem eine Anzahl von Anforderungen gestellt.

So wird für die Interaktion des Benutzers mit dem Simulationssystems eine *interaktive Benutzerschnittstelle* benötigt. An die *aktiven Komponenten* werden normalerweise ebenfalls diverse Anforderungen gestellt: so sollte es möglich sein beliebige Robotertypen interaktiv genieren und validieren zu können, d.h. der Anwender sollte während der Simulation in der Lage sein neue Roboterkomponenten erzeugen, oder alte entfernen bzw. ändern zu können. Des Weiteren müssen Bewegungen erzeugt werden können, z.B. für Roboter oder andere aktive Komponenten. Für allgemeine *Physikalische Komponenten* werden Funktionen benötigt, um eine möglichst reale Umwelt nachbilden zu können.

An die *Planungskomponenten* werden weiterhin folgende Anforderungen gestellt: Unter anderem werden Funktionen zur Planung und Ausführung von Roboteraktionen benötigt, unter anderem zur Berechnung und Planung von Trajektorien.

Da am Anfang einer Simulation der wahre Umfang dieser nicht immer schon bekannt ist, wird von der Systemstruktur eine gewisse Erweiterbarkeit vorausgesetzt, um auf mögliche Änderungen bzw. Erweiterungen der Simulation reagieren zu können.

8.2.3 Simulation einer Robotersteuerung

Bei der Simulation (oder Emulation) einer Robotersteuerung wird das Robotersteuerungsprogramm mit all seinen Befehlen nicht mehr an die Hardware-Steuerung übergeben, sondern an einen Simulator (oder Emulator). Danach wird die Simulation ausgeführt.

Es ist dabei ein entscheidender Unterschied, ob die Simulation und das reale System unabhängig voneinander ausgeführt werden, oder ob ein gewisser Abgleich zwischen diesen stattfindet. 8.1 zeigt schematisch die beiden Verfahren.

Bei der *unabhängigen* Ausführung von Simulation und dem realen System (linkes Bild in 8.1) wird, der vom Programmiermodul erzeugte, Programmcode entweder an die Robotersteuerung *oder* an den Roboteremulator übergeben. Um beide ausführen zu können ist es nun erforderlich dass die Robotersteuerung und der Emulator (bzw. Simulator) die gleiche Sprache verstehen. Danach erfolgt entweder die Off-Line Ausführung des virtuellen Roboters, oder die tatsächliche On-Line Ausführung des realen Roboters.

Auf der rechten Seite der 8.1 sieht man, dass hier an den Emulator/Simulator nicht der gleiche Programmcode wie and die Steuerung übergeben wird. Dafür ist jetzt ein separates

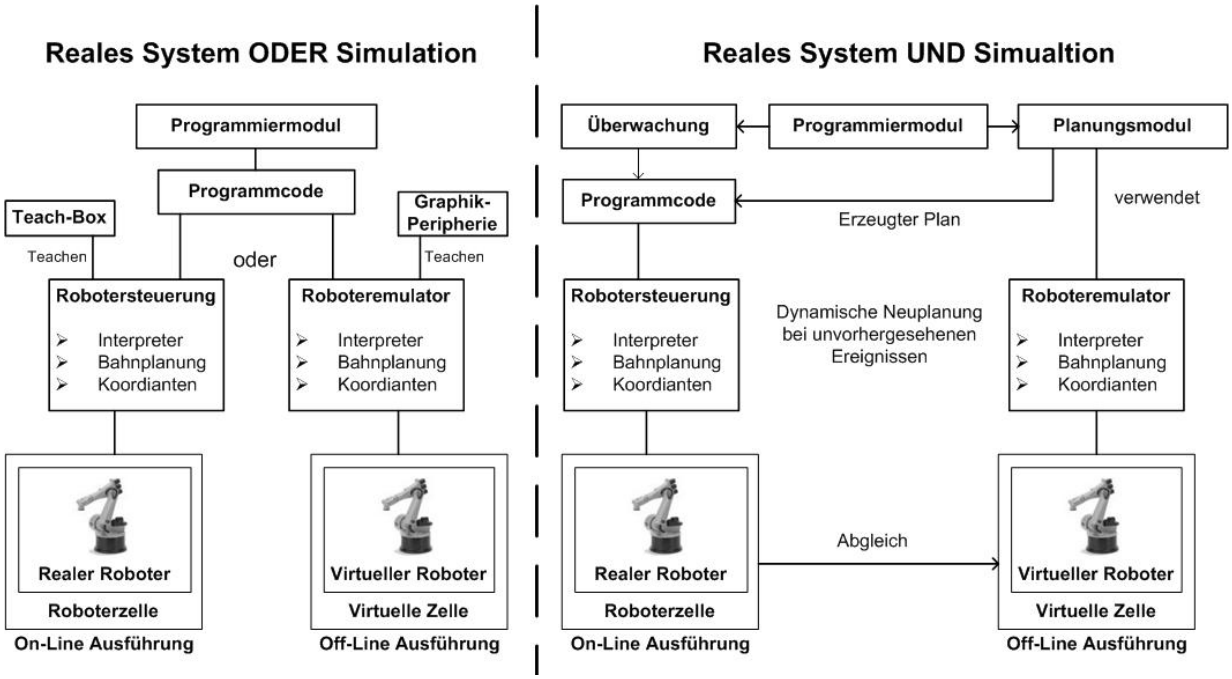


Abbildung 8.1: Simulation einer Robotersteuerung

Planungsmodul zuständig, durch dessen Plan auch erst der Programmcode erzeugt wird. Wenn dieser dann an die Robotersteuerung übergeben wurde, erfolgt während der On-Line Ausführung ein ständiger Abgleich mit der Simulationsausführung, um bei unvorhergesehenen Ereignissen eine Neuplanung durchführen zu können.

Noch einen Schritt weiter geht die *gleichzeitige* Ausführung von Simulation und realem System: Dabei wird der Programmcode wieder an die Robotersteuerung und den Emulator/Simulator übergeben, und nach derer (gleichzeitiger) Ausführung wird eine mögliche Differenz berechnet (Abweichungen der tatsächlichen Ausführung von der Simulation). Durch diese Differenz erfolgt dann gegebenenfalls eine Modelladaption des Emulators, bzw. es erfolgt eine neue Verifikation des Programms.

8.3 Beispiel: GraspIt!

GraspIt! ist ein Simulator zur Bewertung und Planung von Robotergriffen. Roboterhände besitzen oft eine hohe Anzahl von Freiheitsgraden, und der Suchraum für Griffe an einem Objekt wird dadurch sehr groß: Die Dimensionalität des Suchraums ergibt sich aus den Dimensionen für Position und Lage der Hand bezüglich des Objekts (3+3) und der Anzahl der Freiheitsgrade der Hand. Analytische Planung von Griffen ist also im Normalfall nicht möglich, so dass wissensbasierte und heuristische Ansätze zum Einsatz kommen.

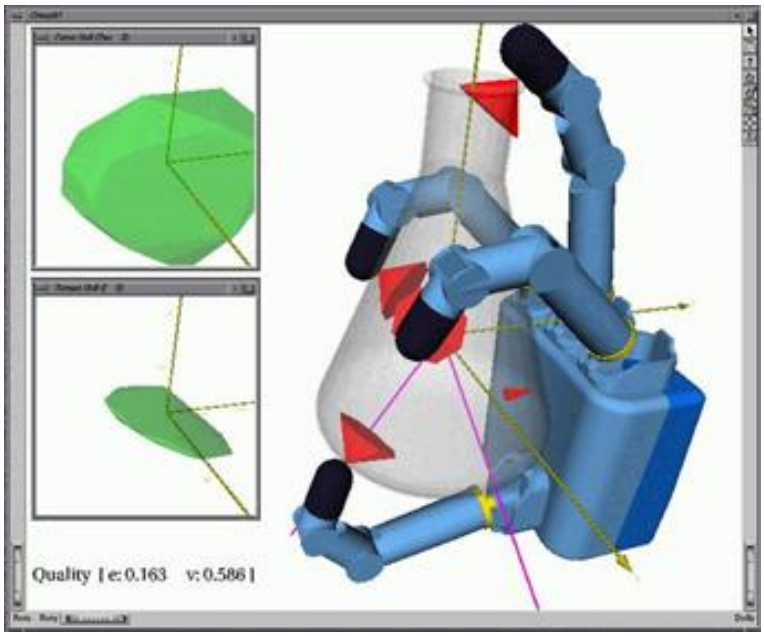


Abbildung 8.2: GraspIt! - komfortable Simulation von Robotergriffen

GraspIt! dient in erster Linie zum konfigurieren und bewerten von Griffen mit einer Roboterhand an einem Objekt. Abb. 8.2 zeigt einen solchen Griff mit der DLR Hand an einem Erlenmeyer-Kolben.

GraspIt! lässt sich intern in mehrere Komponenten gliedern [MA01]:

- Die Erstellung bzw. Implementierung einer Vielzahl von Roboterhänden und Objekten.
- Ein Benutzerinterface zur Visualisierung und zum manuellen Manipulieren der Roboterhand.
- Ein System zur Erkennung von Kollisionen, welches ebenfalls die Kontaktpunkte/flächen erkennt und die auftretenden Reibungskegel bestimmt.
- Die Qualitätsberechnung des Griffes, wobei unterschiedliche Qualitätsmerkmale zur Verfügung stehen.
- Eine Visualisierungsmöglichkeit des wrench spaces (dt: Stoßraum), mit welchem es möglich ist, zu erkennen unter welchen äußeren Kräften ein Griff stabil bleibt.

8.3.1 Simulation unter GraspIt!

Mit GraspIt! kann der Benutzer unterschiedliche Objekte simulieren:

Bei den Modellen von **Roboterhänden** werden die aus Robotik 1 bekannten DH-Parameter benutzt. Dabei wird eine kinematische Kette (z.B. der Finger einer Roboterhand mit mehreren Gelenken) von Gelenk zu Gelenk beschrieben, indem in die einzelnen Gelenke lokale Koordinatensysteme gelegt werden. Eine weitere wichtige Eigenschaft für die Simulation von Roboterhänden sind deren Freiheitsgrade sowie ihre Materialbeschaffenheit. Letztere ist besonders zur Berechnung von auftretenden Reibungen zwischen Objekten wichtig.

Objekte werden unter GraspIt! in erster Linie durch ihre Geometrie beschrieben. Allerdings sind bei der Simulation noch weitere Eigenschaften von Bedeutung, z.B., genauso wie bei den Roboterhänden, Materialeigenschaften (Reibungskoeffizienten), das Gewicht des Objektes sowie dessen Massenschwerpunkt und Trägheitsmomente bzw. Trägheitstensoren.

8.3.2 Kollisionserkennung und Griffanalyse

Um zu vermeiden, dass sich Objekte während der Simualtion durch einander bewegen, benutzt GraspIt! das PQP-System (Proximity Query Package). Nachdem ein Objekt in den Simulator geladen wurde, werden all seine Winkel an das PQP übergeben und werden zu den Blätter von hierarchischen Bäumen aus Grundkörpern. Rekursive Algorithmen können dann schnell bestimmen, ob ein Winkel eines Objektes sich mit einem anderen Winkel eines anderen Objektes schneidet, oder ob ein bestimmter Mindestabstand eingehalten wird. Eine wichtige Eigenschaft der Objekte unter GraspIt! vereinfacht das Verfahren der Kollisionserkennung: alle Objekte werden als unverformbar angenommen! Eine Kollision kann als minimaler Abstand von zwei Objekten von PQP berechnet werden. Besteht die Kontaktstelle nur aus einem Punkt, spricht man von einem Punktkontakt, mehrdimensionale Kontaktflächen können durch Punktkontakte approximiert werden. [MC03]

Nachdem nun die Kontaktpunkte (oder Kontaktflächen) bestimmt sind, müssen nun die Reibungskegel in diesen bestimmt werden. Ein Reibungskegel gibt an, welche Kräfte an der Kontaktstelle wirken dürfen ohne dass sich der Kontakt löst. Zuerst benötigt man hierfür die Kontaktnormalenebene an das zu greifende Objekt. Im Kontaktpunkt wird nun die Normale zu dieser Tangentenebene berechnet. Wendet man nun als Nährung das *Coulombsche Reibungsmodell* an, erhält man, dass die Kraft, welche an dem Kontaktpunkt auf das Objekt wirken darf, innerhalb eines Kegel liegen muss. Seine Spitze liegt im Kontaktpunkt, seine Drehachse liegt entlang der Kontaktnormalen und sein Öffnungswinkel beträgt

$$\varphi = \tan^{-1} \mu_s$$

. Dabei ist μ_s der Reibungskoeffizient zwischen den beiden Objekten. Zur Vereinfachung wird dieser Kegel nun noch durch eine geeignet gewählte Pyramide angenähert. In Abb. 8.3 a) sieht man eine Barretthand mit gegriffenem Objekt sowie die rot gekennzeichneten Reibungskegel in den vier Kontaktpunkten, bzw. in Abb. 8.2 eine andere Roboterhand incl. Reibungskegel.

Der nächste Schritt der Griffanalyse ist die Analyse jedes einzelnen Kontaktpunktes, und die Bestimmung, welchen Anteil diese an der gesamten Griffstabilität hat. Nun folgt die

Berechnung des sog. *grasp wrench space*, der Raum der angibt unter welchen einwirkenden Kräften und Momenten der Griff noch stabil bleibt. Wichtig: Liegt der Ursprung innerhalb dieses Raumes, ist der Griff stabil, d.h. das Objekt wird ohne äußere Kräfte festgehalten. Mit diesem Raum hat man ein gutes Werkzeug gefunden, um Aussagen über die Griffgüte machen zu können. Man unterscheidet nun zwei unterschiedliche Verfahren:

Das nächstliegende Verfahren ist die Berechnung des Volumens des *grasp wrench space*, wobei je größer das Volumen als desto besser (stabiler) wird der Griff angenommen. Dieses Verfahren berücksichtigt jedoch nicht die Anfälligkeit des Griffes auf einzelne Kräfte oder Momente. So kann es vorkommen, dass das Volumen zwar relativ groß ist, jedoch bei kleinsten Kräften entlang einer bestimmten Richtung der Griff instabil wird. Deswegen wird meistens das zweite Verfahren verwendet: Hierbei wird der minimale Abstand des *grasp wrench space* zum Ursprung berechnet.

Zusammenfassend noch einmal die wichtigsten Punkte der Griffanalyse unter GraspIt!:

- Bestimmung der Kontaktpunkte des Griffes
- Berechnung der Reibungskegel in diesen Punkten
- Berechnung des *grasp wrench space*
- Bestimmung der Griffgüte (Volumen oder kleinster Abstand)

Es sei noch erwähnt dass bei der, oftmals notwendigen und sinnvollen, Visualisierung des *grasp wrench space* 3 Kräfte oder Momente fixiert werden müssen, da der 6-dimensionale (3 Kräfte und 3 Momente) Raum sonst graphisch nicht darstellbar wäre. (3-D Fall vorausgesetzt). Eine solche Visualisierung kann man in 8.3 b) sehen. [MC03]

8.3.3 Automatischer Greif-Planer

GraspIt! verfügt über einen automatischen Greif-Planer, mit welchem für ein vorgegebenes Objekt und eine vorgegebene Roboterhand automatisch unterschiedliche Griffe erstellt werden können. Unter allen entstehenden Griffen kann dann durch Bestimmung der Griffgüte der stabilste ausgewählt werden.

Die Planung von Griffen besteht dabei aus mehreren Schritten und basiert auf einer Objektbeschreibung durch Grundkörper, ähnlich einer vereinfachten CSG-Darstellung.

Für ein Objekt muss also zunächst eine Darstellung aus Grundkörpern existieren. Diese besteht aus einem oder mehreren Grundkörpern vom Typ Kugel, Zylinder, Kegel und Quader, den jeweiligen Parametern der Grundkörper (Radien, Seitenlängen etc.) sowie ihren Positionen bezüglich des Originalobjekts.

Diese Grundkörper werden zur Erzeugung von Griffhypothesen verwendet. Die entstehenden Hypothesen bestehen aus Handpositionen bezüglich des Objekts, einer assoziierten

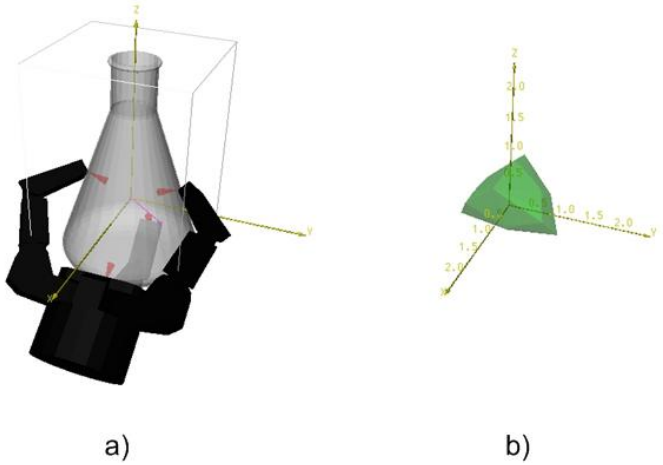


Abbildung 8.3: a) Barretthand mit gegriffenem Objekt sowie Reibungskegel b) zugehöriger Grasp-Wrench-Space bei 3 fixierten Momenten

Orientierung sowie einer Handpose. Diese Startposen für Griffe werden evaluiert, indem die Hand in der Simulation auf das Objekt zubewegt wird und die Finger nach unterschiedlichen Strategien geschlossen werden. Der entstehende Griff kann mit Hilfe der beschriebenen Verfahren auf Stabilität untersucht und bewertet werden.

Eine detaillierte Beschreibung hierzu findet sich in [MKCA03].

Kapitel 9

Telerobotik

9.1 Motivation

Es stellt sich die Frage, wozu man Telerobotik benötigt und warum Einsätze nicht allein von einem ferngesteuerten Gerät oder einem autonomen Roboter verrichtet werden können. Dies soll im Folgenden genauer erläutert werden.

Der Mensch ist nicht in jeder Umgebung einsetzbar, z.B. kann

- ein Einsatz in radioaktiv verseuchten Gebieten zu gefährlich werden.
- der benötigte Aufwand für die auszuführende Aufgabe zu groß werden (z.B. in der Raumfahrt - Entwicklungen von Raumfähren, Schutzanzügen, Werkzeugen etc.).
- der Mensch allein von seiner Größe und seinem Gewicht nach einem Erdbeben nicht in Trümmerbergen die Suche vornehmen.

Daher wird sehr stark am Einsatz von Robotern in diesen Gebieten gearbeitet.

Autonome Systeme mit Manipulationsmöglichkeit müssen sich nicht nur um ihren internen Zustand kümmern, sondern auch um die Auswirkungen ihres Handelns in der Umgebung. Dabei könnte eine, für die mitgeführten Ressourcen, zu komplexe Sensorauswertung notwendig werden, und das System keine oder nicht die richtige Reaktion auf unbekannte oder unvorhergesehene Ereignisse liefern. Daher ist aus einiger Entfernung ein gewisses Maß an Kontrolle und Unterstützung durch den Menschen nötig.

Um den Benutzer zu entlasten, erfolgt eine größtmögliche Integration von Komponenten der autonomen Systeme (z.B.: Planer). Des Weiteren wäre es sehr mühselig jeden Routinearbeitsschritt einzeln zu steuern [Say99]. Hinzu kommt, dass durch bestimmte Übertragungsmedien oder große Distanzen Verzögerungen beim Signalaustausch entstehen, wobei ab einer Übertragungszeit von mehr als 0,3 Sekunden eine direkte Steuerung nicht mehr möglich ist. Dies tritt gerade bei Weltraumanwendungen häufig auf [Fun91].

Daher muss das System über eine gewisse Eigenintelligenz verfügen, wie z.B. Grundreflexe, damit es sofort auf Probleme reagieren kann [Say99].

Der Mensch wird wegen seiner überlegenen perzeptiven und reaktiven Fähigkeiten in den Kontrollkreislauf eingebunden.

→ **Daher ist der Einsatz von Teilautonomen Systemen erforderlich.**

9.2 Definitionen

In diesem Abschnitt sollen kurz einige wichtige, im Weiteren verwendete, Begriffe erklärt werden.

9.2.1 Telepräsenz/Teleexistenz

Als Telepräsenz wird das Gefühl des Bedieners des Systems bezeichnet, sich in der entfernten Remote-Umgebung präsent zu fühlen. Dies wird über den Teleoperator durch sensorielle Erfassung relevanter Informationen aus der Remote-Umgebung erreicht, die dann in einer für den menschlichen Bediener natürlichen und wirklichkeitsnahen Weise präsentiert werden. Ideale Telepräsenz ist die totale sensorielle "Immersion "(Eintauchen) des menschlichen Bedieners mit all seinen sensorischen Fähigkeiten in die Remote-Umgebung.

9.2.2 Telemanipulation

Bei der Telemanipulation steuert der Bediener direkt den Roboter, d.h. ohne Sensorik und ohne Planungskomponenten.

9.2.3 Teleoperation

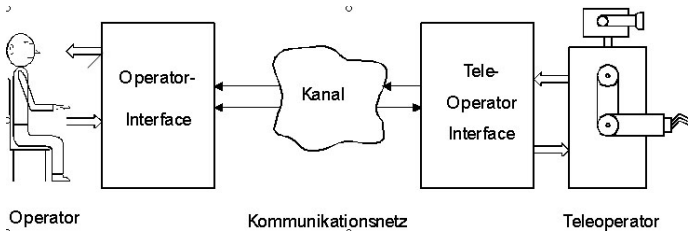
Teleoperation ist die Erweiterung der sensoriellen und manipulatorischen Fähigkeiten einer Person für das Wirken an einem entfernten Ort (remote location). Ein Teleoperator muss Sensoren, Arme und/oder Hände, (u. U. auch ein Fahrzeug oder einen Gehapparat zu deren Beförderung,) und multimodale Kommunikationskanäle von und zum menschlichen Bediener haben. Teleoperation bezeichnet auch die zeitlich direkte und kontinuierliche Regelung durch den menschlichen Bediener.

9.2.4 Telerobotik

Telerobotik ist die weiterentwickelte Form der Teleoperation, bei der der menschliche Bediener als Überwacher fungiert; der Mensch formuliert Ziele, Nebenbedingungen, Pläne,

Annahmen, Vorschläge und Kommandos, die durch einen zwischengeschalteten Computer an den Teleoperator übermittelt werden. Dabei erhält der Bediener Informationen vom System zurück, z.B. über den Stand der durch den Teleoperator erfüllten Aufgaben, über Schwierigkeiten, Bedenken und schließlich sensorielle Daten über den Teleoperator selbst und die Remote-Umgebung.

9.3 Komponenten



Bei Telerobotiksystemen ist der Operator durch ein Operator-Interface über einen Kanal (beispielsweise eine Funkverbindung) mit einem Teleoperator-Interface verbunden. Durch dieses erhält er Werte über den Roboter/Manipulator und dessen Umgebung, kann Aufgabenstellungen planen, simulieren und an den Teleroboter übermitteln.

Benutzerschnittstelle: Die Benutzerschnittstelle nimmt durch Sensorik die Eingaben des Benutzers entgegen und bietet die Möglichkeit Voraussagen über das Verhalten der Roboters bzw. der Umwelt an Hand von lokalen Modellen zu treffen. Dem Benutzer werden die letzten Daten des Roboters präsentiert, sowie dessen Reaktionen in lokalen Modellen. Diese Präsentation kann sowohl visuell, auditorisch, olfaktorisch als auch haptisch erfolgen [Kam04]. Des weiteren verfügt die Benutzerschnittstelle über eine Kommunikationsschnittstelle, durch die sie über einen Kanal Kontakt mit dem Teleoperator-Interface aufnehmen kann.

Teleoperator-Interface: Das Teleoperator-Interface nimmt Signale vom Operator-Interface entgegen und erstellt mit Hilfe von lokalen Modellen in einem Vorverarbeitungsschritt die Ansteuerung für die Aktorik des Roboters. Das Interface nimmt die Sensordaten des Roboters entgegen, verarbeitet diese intern weiter und schickt eine Rückmeldung an das Operator Interface. Des Weiteren werden ständig die berechneten Daten mit den reellen abgeglichen um möglichst gut auf Fehler und unvorhergesehene Ereignisse reagieren zu können.

9.3.1 Benutzerschnittstellen

9.3.1.1 Eingabe

Früher war die Eingabe einer Benutzerschnittstelle alphanumerisch und basierte auf Master-Slave Prinzipien (siehe Abschnitt 2.3), d.h. der Benutzer gab mit Hilfe des Masterarms die auszuführende Aktion in einem Beispielraum vor und diese wurde dann vom Slave im Original ausgeführt [T.B86]. Heute geschieht dies über eine Grafische Oberfläche (engl.: grafical user interface, GUI) und "Pulldown-Menüs". Die Eingabe erfolgt über verschiedene Geräte, deren Verwendung stark vom jeweiligen System abhängt. Es gibt Geräte, die vom Benutzer lediglich Steuerungsbefehle entgegen nehmen und solche die ihm ein physikalisches Feedback in Form von Gelenkswiderständen oder Oberflächenveränderung geben. Eine andere Möglichkeit ist die Programmierung durch Vormachen (PdV, siehe Kapitel 5).

Zu den Geräten ohne ein direktes Feedback gehört der Daten-Handschuh (DataGlove, siehe Abb. 9.1). Die Bewegungen der Finger und Hand werden in Form von Handstellung und Position im Raum gemessen. So können Gesten erkannt werden und, gekoppelt mit einem Tracking Modul, können auch ganze Bewegungsabläufe erfasst werden.

Abbildung 9.1: Datenhandschuh

Das Force-Feedback-Device (FFD) Phantom ähnelt entfernt einem Roboterarm; die Bedienung erfolgt über das Bewegen der "Hand". Die Bewegung wird in eine virtuelle Umgebung in den Computer übertragen. Wenn beispielsweise ein Objekt in einer virtuellen Welt bewegt wird und dieses auf ein Hindernis stößt, gibt das Phantom eine Kraft an den Benutzer aus, so dass dieser ein Gefühl für die Kollision erhält.

Bei der Methode des Programmieren durch Vormachens (PdV) soll einem Roboter durch mehrfaches Zeigen ein Handlungsablauf beigebracht werden. Der Roboter oder das beob-

Abbildung 9.2: verschiedene Phantom-Modelle

achtende System muss erkennen, welche Aktionen wie ausgeführt werden müssen und diese dann in Bewegungen des Roboters übersetzen.

Die *Ausgabe* kann visuell beispielsweise stereoskopisch oder über Raytracing erfolgen. Ein beliebtes Ausgabegerät in der Telepräsenz sind Head-Mounted-Displays (siehe Abbildung 9.3), sie werden auf dem Kopf getragen. Durch zwei eingebaute Bildschirme wird vor den Augen des Benutzers ein virtuelles 3D-Bild erzeugt. Ein im HMD eingebauter Tracker ermittelt die Kopfposition und -orientierung. Aus diesen Daten berechnet der Computer die Blickrichtung des Benutzers und zeigt entsprechende Bilder an.

Abbildung 9.3: Head-Mounted Display HMD

Die Ausgabe kann aber auch via Kraftrückkopplung erfolgen. Ein Beispiel hierfür ist das Exoskelett (Abbildung 9.4). Es kann über einen Daten-Handschuh gezogen werden und

gibt eine vom Computer gesteuerte Kräfte-Rückkopplung an verschiedene Teile der Hand und der Finger wieder.

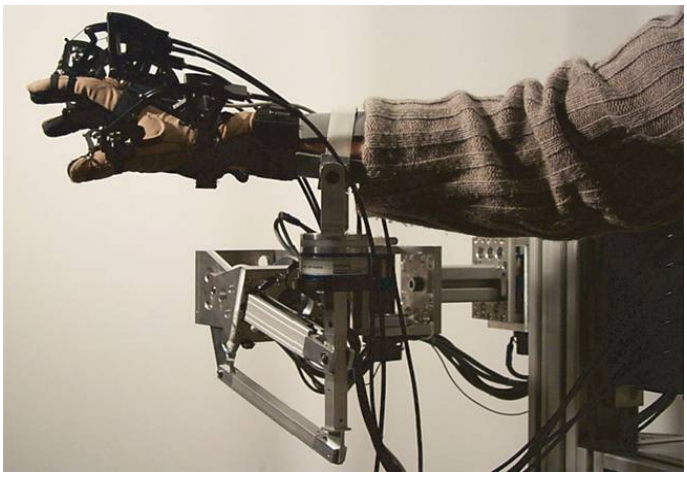


Abbildung 9.4: Hand-Kraftexoskelett

Der Einsatz dieser Komponenten hängt sehr stark von der Bandbreite und Zuverlässigkeit ab, mit der Informationen zwischen dem Benutzer und dem Roboter ausgetauscht werden können [Say99].

Im Nahbereich, also bei guter und zuverlässiger Anbindung ohne Datenverlust, kann der Benutzer in Echtzeit über einen Telepräsenz-Modus den Roboter steuern, als wäre er vor Ort. Momentan konzentriert sich die Forschung stark darauf, dem Benutzer haptische Eindrücke zu vermitteln [Kam04].

Wenn der Abstand zwischen Benutzer und Roboter so groß wird, dass keine Verbindung ohne Zeitverzögerungen mehr aufgebaut werden kann, ist die direkte Steuerung via Kraft-Rückkopplung nicht mehr möglich [Say99]. In diesem Fall greift man auf Systeme zurück, die keine direkte Kraft-Rückkopplung zum Benutzer mehr besitzen, sondern ihm lediglich Veränderungen z.B. in grafischer Form anzeigen. Der Benutzer kann weiterhin den Roboter direkt steuern. Bei einer Verbindung über einen sehr begrenzten und mit Verzögerungen belasteten Kanal gibt es die Möglichkeiten einer “move and wait“ Strategie, um das Risiko einer inkorrekten Handlung zu vermeiden. Hierbei wird nach einer kleinen Bewegung gewartet, um zu überprüfen, dass alles wie erwartet funktioniert hat, um dann die nächste kleine Bewegung durchzuführen. Diese Methode ist zu vermeiden, da beispielsweise eine unachtsame Bewegung des Roboterarms nicht rechtzeitig erkannt und verhindert werden kann.

Um die Leistungsfähigkeit zu steigern, muss der Benutzer eine sofortige Rückmeldung erhalten. Wegen der Kommunikationsverzögerung kann die Rückmeldung aber nicht von dem entfernten Einsatzort kommen. Daher wird ihm die Möglichkeit geschaffen in einem

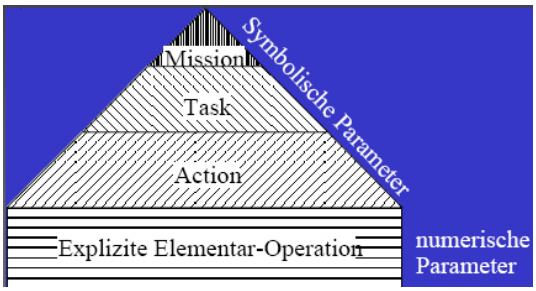


Abbildung 9.5: Hierarchischer Aufbau von TELOS

Weltmodell das Verhalten des Roboters und dessen Umwelt zu testen und erst dann die Befehle an den Roboter zu übermitteln.

9.3.2 Aktionsplaner

Unter einem Planungssystem für Roboter versteht man allgemein ein System, das Ausgehend von einem Startzustand und der Beschreibung eines gewünschten Zielzustandes eine Folge von Aktionen generiert, die das betrachtete System schrittweise in den Zielzustand überführt (siehe Kapitel 6). Somit wird vom Benutzer zumeist nur noch das Spezifizieren des Zielzustandes verlangt und das Planungssystem löst das Problem, wie der Zielzustand erreicht werden kann [DR91]. Ein Aktionsplaner ist in der Regel hierarchisch aufgebaut (Abbildung 9.5). Ein zum Ziel führender Plan wird zunächst auf höchster Ebene generiert und dann auf der nächst niedrigeren Stufe verfeinert.

Um planen zu können muss das System wissen, welche expliziten Elementar-Operationen (EEO, siehe Abschnitt 4.2.1.2) auf der untersten Ebene ausgeführt werden können, was diese bewirken und wann sie eingesetzt werden können. Für die Planungsprozesse der untersten Ebene wird ein Weltmodell benötigt.

Ein Aktionsplanungssystem, wie es bei den Mars Rover Missionen Opertunity und Spirit der NASA 2004 zum Einsatz kam, ist MAPGEN ("Mixed Initiative Activity Plan Generator") [ACBC+04]. Mapgen ist eine Kombination zweier Planungssysteme APGEN (Activity Plan Generator) und EUROPA. APGEN ist ein manuell betriebenes Planungssystem, das schon bei den Missionen Cassini und Deep Impact zum Einsatz kam. EUROPA ist ein automatisches Planungs- und Zeitplanungssystem und war als Teil eines Experimentes zur Demonstration eines geschlossenen KI-basierenden Systems an Bord von Deep Space 1.

Der Bediener gibt die Missionsziele vor, Mapgen generiert einen Plan und überprüft diesen immer wieder, ob er mit den mitgeführten Ressourcen durchführbar ist und keine Regelverletzungen aufweist. Erkennt Mapgen beispielsweise, dass die durchzuführende Aktion die Batterie entleeren würde, dann versucht es die Tasks neu zu verteilen um dies zu umgehen. Im Notfall kann das System des Rovers Tasks auch abbrechen, dies macht Sinn, wenn es beispielsweise sonst zu einer Kollision kommen würde, die im vorherigen Planungsprozess

auf der Erde nicht bemerkt worden ist. Daher kann der Benutzer den Tasks verschiedene Prioritäten zuteilen, um bei Abbrechen eines Tasks sicherzustellen, welche Tasks in jedem Fall durchgeführt werden sollen.

Zusätzlich erstellt MAPGEN aus den Aufgabenstellungen einen Zeitplan, welche Aufgabenstellung ausgeführt werden kann und zu welchem Zeitpunkt das am besten geschehen sollte. Das Ziel ist, das Maximum aus den zur Verfügung stehenden Ressourcen und der vorhanden Zeit herauszuholen.

Nach Fertigstellung des Plans für den nächsten Tag wird dieser als komplette Befehlsfolge an den Rover gesendet.

9.3.3 Weltmodell

Bei der Anwendung von Telerobotern kommt es häufig zu Verzögerungen in der Datenübertragung zwischen Operator und Teleroboter, weswegen es wünschenswert ist die Befehle vor der Datenübermittlung überprüfen zu können. Daher werden auf Basis eines Weltmodells Simulationen auf Seiten des Operators eingesetzt, in welchem er Aktionen und Algorithmen testen kann.

Um ein geeignetes Weltmodell zu erlangen, müssen zunächst alle wichtigen Informationen der realen Welt vom Roboter erfasst und an den Operator übermittelt werden [DR91]. Diese Daten werden einer Analyse unterzogen um die Eigenschaften der Objekte im Raum darzustellen. Analysiert wird, in welcher Beziehung Objekte zueinander stehen (z.B.: 'Das ist Objekt A und es liegt auf Objekt B'), die Orientierung der Objekte im Raum, kinematische Informationen, also die spezifische Bewegungsfähigkeit mechanischer Systeme (Scharnier einer Tür), die geometrische Modellierung (z.B. Abmessungen), die Struktur in Form von Zuständen von Objekten (z.B. Tür ist zu) und Dynamische Eigenschaften (z.B. maximale Beschleunigung oder Elastizität).

9.3.4 Bahn- / Greifplanung

An Hand der Informationen im Weltmodell können nun Bewegungen des Roboters in der Umgebung geplant werden. Die Planung erfolgt in Abhängigkeit der Startstellung des Systems, der Zielstellung und der Geometrie der Umwelt (Hindernisse). Gesucht ist eine mögliche kollisionsfreie Bahn bzw. die Distanz der Greifer zu einem Objekt für einen sicheren Griff. Diese Werte können nun in einer Simulation überprüft werden.

9.3.5 Simulation / Animation

Die Simulation läuft auf dem in Abschnitt 9.3.3 besprochenen Weltmodell. Durch die Simulation soll die Planung, Offline-Programmierung und Test ohne Beteiligung von Hardware

möglich sein. Auch soll eine flexible Planung unterstützt werden, damit der Planer Alternativen erproben und bewerten kann. Die Sicherheit für die Hardware wird durch Testen auf Kollisionen und übermäßige Belastungen erhöht.

Die von der Bahn- und Greifplanung ermittelte Bewegung wird auf Erreichbarkeit (Kinematik), Kollisionsfreiheit (Geometrie), die auftretenden Kräfte und Momente (Dynamik) hin überprüft.

Die Simulation kann zur Unterstützung des Benutzers auf einer grafischen Oberfläche dargestellt werden (Animation). Für die vorhandenen Sensoren werden Werte berechnet, welche als Muster (Patterns) für die reale Ausführung benutzt werden.

Da das Weltbild aber kein exaktes Abbild der realen Einsatzumgebung und der realen Objekte darstellt, ist eine Überwachung der realen Ausführung nötig.

9.3.6 Überwachung

Der Teleroboter muss in der Lage sein seine Aktionen auf Korrektheit zu überprüfen. Bei einem korrekten Verlauf soll er mit dem nächsten Auftrag beginnen, ansonsten startet er mit einer Fehlerbehandlung. Hierzu muss er über ein gewisses Maß an Autonomie verfügen [Dep91].

9.3.6.1 Teleroboter

Zunächst werden die Bewegungskommandos in direkte Ansteuerungsbefehle der Gelenkmotoren umgewandelt. Nun werden die Motoren in der gewünschten Art bewegt. Das (Nicht-)Erreichen der Zielstellung wird an die Überwachung zurückgemeldet. Die Sensor- und Manipulatorbefehle werden an die entsprechenden Systeme weitergeleitet.

9.3.6.2 Sensorik

Die Steuerung des Teleroboters leitet die Sensorkommandos zur Sensorsteuerung weiter. Die aktuellen Sensorwerte werden mit den Mustern aus der Simulation verglichen, damit bei möglicherweise auftretenden Abweichungen der abweichende Sensor mit Ist- und Sollwert als Fehlermeldung zurückgeben werden kann.

9.3.6.3 Fehlerbehandlung

Bei der Diagnose eines Fehlers bestimmt die Fehlerbehandlung, welche Art von Fehler vorliegt. Die Entscheidung fällt auf Grund des aktuellen Roboterkommandos und dessen Stadium oder auf Grund des Sensors und dessen Ist- zu Sollwert Abweichung. Unter Umständen kann ein Abbruch auch durch Aktionen von höherer Priorität verursacht worden sein.

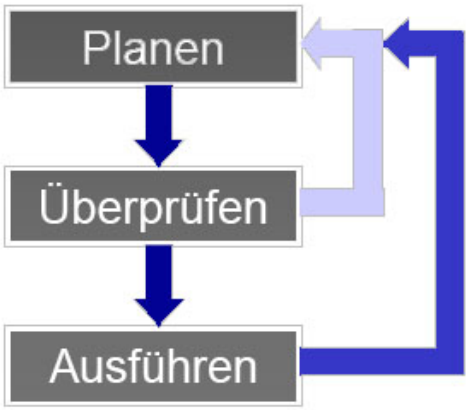


Abbildung 9.6: Das Drei Phasen-Modell von TELOS

Aufgrund der Diagnosedaten schlägt das System verschiedene zur Auswahl stehende Möglichkeiten vor, den Fehler zu beheben.

9.4 Kontrollfluss

Der Kotrollfluss soll im Weiteren am Beispiel des Teleoperations-Systems TELOS des IPR erklärt werden [DF93]. Die Ausführung erfolgt nach einem drei Phasen-Modell, welches sich in eine Planphase, Überprüfungsphase und die Ausführung aufteilt (Abbildung 9.6). Nach dem Planen werden die erzeugten Werte in einer Simulation überprüft, sollten Fehler auftreten erfolgt eine Neuplanung. Nach erfolgreicher Überprüfung wird die Aktion an den Teleroboter gesendet und dort ausgeführt. Bei einem Fehler während der Ausführung auf dem Teleroboter erfolgt über die Fehlerbehandlung eine teilweise Neuplanung, und nach dem Beheben des Fehlers wird der alte Plan fortgesetzt.

9.4.1 Planphase

Bei der Planphase wählt der Benutzer zunächst eine Mission auf der obersten Ebene aus und parametrisiert diese. Verschiedene Aktionsplaner zerlegen diese Schritt für Schritt:

1. Der Aktionsplaner für Missionen zerlegt die Mission in Tasks und parametrisiert diese automatisch.
2. Der Aktionsplaner für Tasks zerlegt die Tasks in Aktionen und parametrisiert diese automatisch.

3. Der Aktionsplaner für Aktionen wandelt die symbolischen Werte der Aktionen in numerische Parameter um und erzeugt damit Explizite Elementar Operationen (EEO).

Der Bediener kann mit einer speziellen EEO über ein 6D-Eingabegerät den Roboter direkt steuern (siehe auch Abschnitt 4.2.1.2).

9.4.2 Überprüfungsphase

Nachdem die Umwelt in einer grafischen Simulation aufgebaut wurde, wird ein Roboter- und Sensorkommando in dieser überprüft. Die Aktion wir in der Grafik dargestellt (z.B. Bewegung des Roboters, Schließen einer Tür). Wenn die Aktion erfolgreich durchgeführt wurde, wird die nächste Aktion überprüft. Wenn ein Fehler auftritt, wird die Planungsphase aufgerufen.

9.4.3 Ausführung

Zur Kontrolle wird ein Expliziter Elementar Operator (EEO) auf der grafischen Oberfläche dargestellt, danach wird dieser zum Teleroboter gesendet und dort ausgeführt. Die Sensoren kontrollieren das Ausführen des Roboterkommandos, um bei einer erfolgreicher Ausführung den nächste EEO an den Roboter senden zu können. Wenn ein Fehler auftritt, wird die Fehlerbehandlung aufgerufen.

9.5 Beispiele

9.5.1 Da-Vinci System

Teleroboter gewinnen in der Medizin immer weiter an Bedeutung, so könnten in Zukunft Operationen beispielsweise auf einem Kreuzfahrtschiff oder einer Raumstation ohne die physische Anwesenheit eines Chirurgen durchgeführt werden.

In der Regel arbeiten Roboter in der Chirurgie nach dem Master-Slave-Prinzip oder als einfaches Trägersystem. Sie stehen zu jedem Zeitpunkt unter der Kontrolle des Operateurs, wobei durch Filter unerwünschte Bewegungen, wie das natürliche Zittern der Hand, herausgefiltert werden. Durch Skalierung kann die Bewegung in beliebigem Verhältnis auf die endoskopischen Instrumente übersetzt werden, damit mikrochirurgische Eingriffe mit sehr hoher Präzision durchgeführt werden können. Momentan existieren zwei Manipulator-Systeme auf dem Markt, das *System Zeus* und das *ystem da-Vinci*. Der Einsatz liegt in der laparoskopischen und der minimal-invasiven Chirurgie (MIC) [FSP01].

Das da-Vinci Surgical System besteht aus 3 Teilen (Abbildung 9.7): der Konsole des Chir-

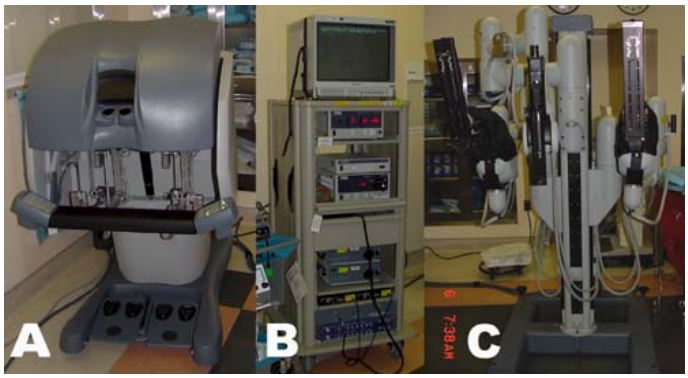


Abbildung 9.7: Das da Vinci System

urgen (A), dem Video-Elektronik-Turm [1](B) und dem Roboterturm [2](C), der mit drei bis vier Roboterarmen ausgestattet werden kann.

Als Benutzerschnittstelle dient hier die Konsole mit ihrem binokularem 3D Bildsystem, mit diesem wird dem Benutzer eine virtuelle 3D Operationsumgebung angezeigt. Die Bedienung erfolgt über zwei Master-Arme die unterhalb des Displays angeordnet sind und über Fußpedale, mit diesen verschiedene Ausrichtungen der Roboterarme und Instrumente gesteuert werden.

Abbildung 9.8: Ein Master-Arm des da Vinci Systems

Die Hände werden in die Master-Arm-Vorrichtungen gesteckt (Abbildung 9.8) und das

[1]electronic tower
[2]robotic's tower

Abbildung 9.9: Robonaut

System übersetzt die Bewegungen der Hand, des Handgelenks und der Finger in präzise Echtzeit-Bewegungen der Roboterarme und Operationsinstrumente im Patienten.

Am Roboterturm befinden sich drei bis vier Roboterarme, mit denen die Slave Bewegungen ausgeführt werden: zwei bis drei für Instrumente und einer für das Endoskop. Bei *da-Vinci* werden Bewegungen nur direkt übersetzt und angepasst, das System arbeitet niemals eigenständig.

9.5.2 Robonaut

Robonaut ist ein humanoider Roboter, der von der NASA in Zusammenarbeit mit der DARPA im Johnson Space Center entwickelt wurde [otNJSC06]. Das Ziel des Robonaut-Projektes ist die Entwicklung und Demonstration eines Roboter Systems, welches als EVA (Astronauten Ersatz) eingesetzt werden kann. Er soll mit Menschen zusammenarbeiten oder an Stellen eingesetzt werden können, an denen die Gefährdung für Menschen zu hoch ist. Um in den Gegebenheiten in einer Raumfähre arbeiten zu können wurde der Robonaut dem Menschen im Design nachempfunden (Abbildung 9.9).

Er verfügt über Hände mit fünf Fingern und menschenähnliche Arme. Robonaut ist mit einem breiten Spektrum von Sensoren bestückt, unter anderem zur Registrierung von Position, Temperatur, Tastempfindungen, Kräfteeinwirkung und von Drehmomenten. Allein an einem Arm befinden sich mehr als 150 Sensoren.

Robonaut verfügt über ein Simulationsprogramm, dass beispielsweise zur Überprüfung von

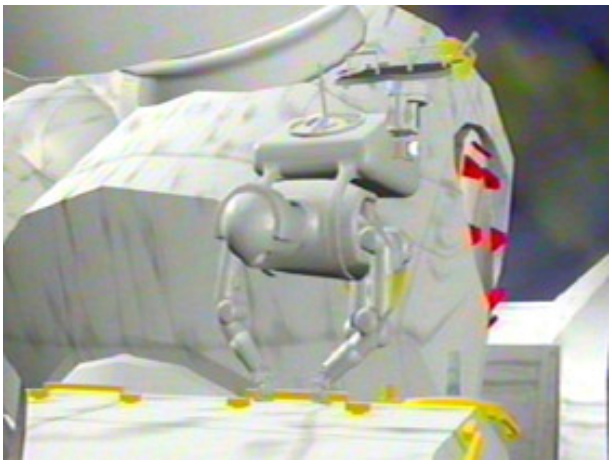


Abbildung 9.10: Robonaut in der Simulation

Steuerungs-, Greifalgorithmen und Wegeplanungen etc. verwendet kann. Die Simulation stellt das Erscheinungsbild, die Kinematik und die Dynamik von Robonaut da (siehe Abbildung 9.10). Benutzer können so Bewegungsabläufe bei schwierigen Anwendungen testen, bevor sie die Hardware direkt ansteuern, und so die Gefahren für den Roboter minimieren.

Robonaut ist in der Lage unter Anweisung eines Menschen in natürlicher Sprache und Gestik Bolzen mit Hilfe eines Schraubwerkzeuges fest zu ziehen (siehe Abbildung 9.11).

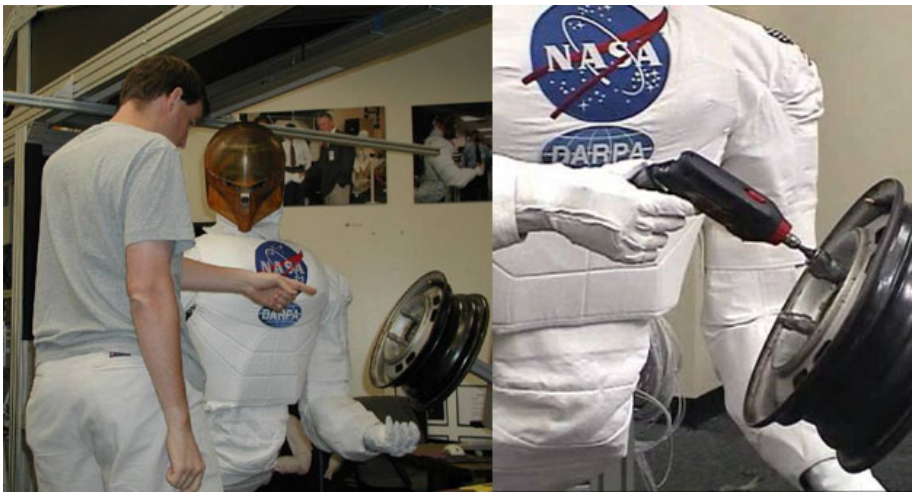


Abbildung 9.11: Robonaut findet und zieht Bolzen autonom fest

Über das Application Programmer's Interface (API) ist jeglicher Freiheitsgrad des Robonauts steuerbar. Durch sein Design ist eine intuitive Steuerung via Telepräsenz möglich.

Literaturverzeichnis

[ACBC+04] Mitchell Ai-Chang, John L. Bresina, Leonard Charest, Adam Chase, Jennifer Cheng jung Hsu, Ari K. Jónsson, Bob Kanefsky, Paul H. Morris, Kanna Rajan, Jeffrey Yglesias, Brian G. Chafin, William C. Dias, and Pierre F. Maldague. MAPGEN: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8–12, 2004.

[AL91] H. Asada and S. Liu. Transfer of human skills to neural net robot controllers. *IEEE International Conference on Robotics and Automation*, 1991.

[App95] Steffen Apprich. *Programmieren durch Vormachen unter Verwendung eines 3D-Sichtsystems.* 1995.

[Ars04] A. M. Arsenio. Learning task sequences from scratch: applications to the control of tools and toys by a humanoid robot. In *Proceedings of the 2004 IEEE International Conference on Control Applications*, volume 1, pages 400–405, 2004.

[BA00] D. Bentivegna and C. Atkeson. Using primitives in learning from observation. In *First IEEE-RAS International Conference on Humanoid Robotics (Humanoids-2000).* 2000.

[BA02] D. Bentivegna and C. Atkeson. Learning how to behave from observing others. In *SAB02-Workshop on Motor Control in Humans and Robots: on the interplay of real brains and artificial devices.* Edinburgh, UK, 2002.

[BAC04] D. Bentivegna, C. Atkeson, and G. Cheng. Learning tasks from observation and practice. In *Journal of Robotics and Autonomous Systems*, pages 163–169. 2004.

[BBG+05] C. Breazeal, D. Buchsbaum, J. Gray, D. Gatenby, and B. Blumberg. Learning from and about others: Towards using imitation to bootstrap the social understanding of others by robots. In *L. Rocha and F. Almedia e Costa (Eds.), Artificial Life*, volume 11(1-2), pages 111–130. MIT Press, Cambridge, 2005.

[Bel57] R. E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BUAC02] D. Bentivegna, A. Ude, C. Atkeson, and G. Cheng. Humanoid robot learning and game playing using pc-based vision. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2449–2454. Lausanne, Switzerland, October 2002.

[Bur96] Grigore Burdea. *Force and touch feedback for virtual reality.* Wiley, 1996.

[CKL94] A. R. Cassandra, L. P. Kaelbling, and M. L. Littman. Acting optimally in partially observable stochastic domains. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[CM00] Jason Chen and B. J. McCarragher. Programming by demonstration - constructing task level plans in a hybrid dynamic fram. In *Proc. 2000 IEEE Intl. Conf. on Robotics & Automation, San Francisco, CA*, 2000.

[Cyp93] Allen Cypher. *Watch what I do.* MIT Press, 1993.

[Dep91] Dept. of Comput. and Inf. Sci., Pennsylvania Univ., Philadelphia, PA. *Efficient Control of a Robotic System for Time-Delayed Environments.* IEEE, 1991.

[Deu02] P. Deuflhard. *Numerische Mathematik 1.* de Gruyter, 2002.

[DF93] Rüdiger Dillmann and Jürgen Fröhlich. Interaktives, benutzerüberwachtes programmieren von telemanipulatoren, 1993.

[DH91] Rüdiger Dillmann and Martin Huck. *Informationsverarbeitung in der Robotik.* Springer, 1991.

[DKZ05] R. Dillmann, S. Knoop, and R. Zöllner. Vorlesungsfolien zu: Robotik ii - programmierung von robotersystemen. Institut für Rechnerentwurf und Fehlertoleranz (IRF), Universität Karlsruhe, Fakultät für Informatik, SS 2005.

[DLR91] DLR. *Control Structures in Sensor-Based Telerobotic Systems*, volume 1. IEEE, June 1991.

[DR91] Huck M. Dillmann R. *Informationsverarbeitung in der Robotik.* Springer Verlag, 1991.

[DRE+99] R. Dillmann, O. Rogalla, M. Ehrenmann, R. Zöllner, and M. Bordegoni. Learning robot behaviour and skills based on human demonstration and advice: the machine learning paradigm. In *9th International Symposium of Robotics Research (ISSR '99), Snowbird, UT, USA*, pages 229–238, October 1999.

[EJ97] Freund E. and Rossmann J. How to control a multi-robot system by means of projective virtualreality. *Advanced Robotics, 1997. ICAR '97. Proceedings., 8th International Conference on*, 759-764, 1997.

[ERZD02] M. Ehrenmann, O. Rogalla, R. Zöllner, and R. Dillmann. Belehrung komplexer aufgaben für serviceroboter: Programmieren durch vormachen im werkstätten- und haushaltsbereich. In *ROBOTIK 2002: Leistungsstand - Anwendungen - Visionen - Trends*. VDI Verlag, Ludwigsburg, Germany, June 2002.

[FHD98] H. Friedrich, J. Holle, and R. Dillmann. Interactive generation of flexible robot programs. *IEEE International Conference on Robotics and Automation*, 1998.

[FK97] Holger Friedrich and Michael Kaiser. What can robots learn from humans? *IFAC Workshop on Human-Oriented Design of Advanced Robotics Systems (DARS '95)*, 1997.

[FN71] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fri99] Holger Friedrich. *Interaktive Programmierung von Manipulationssequenzen*. GCA-Verl., als ms. gedr. edition, 1999.

[FSP01] Philipp A. Federspil, Jan Stallkamp, and Peter K. Plinkert. Robotik ein evolutionssprung in der operativen medizin? *Deutsches Ärzteblatt 2001*, 44:A 2879–2884, November 2001.

[Fun91] Janez Funda. Teleprogramming: Towards delay-invariant remote manipulation, Januar 1991.

[Geo90] Michael P. Georgeff. Planning. In James Hendler, Austin Tate, and Mark Drummond, editors, *Readings on Planning*. Morgan Kaufmann, 1990.

[GMF+92] Eckel G., Gobel M., Hasenbrink F., Heiden W., Lechner U., Tramberend H., Wesche G., and Wind J. Benches and caves. *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, 1083-1088 vol.2, 1992.

[Goo05] Gerhard Goos. *Vorlesungen über Informatik*, volume 1. Springer, 4 edition, 2005.

[Güs92] B. Güsmann. *Einfürung in die Roboterprogrammierung*. Vieweg Verlag, 1992.

[HB96] C. Zenger H.J. Bungartz, M. Griebel. *Einführung in die Computergrafik*. Vieweg Verlag, 1996.

[How60] R. A. Howard. *Dynamic Programming and Markov Processes*. PhD thesis, M.I.T., 1960.

[HT93] Ogata H. and Takahashi T. A geometric approach to task understanding for robotic assemblyoperations. *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, 1993.

[HTD90] James Hendler, Austin Tate, and Mark Drummond. A review of ai planing techniques. In James Hendler, Austin Tate, and Mark Drummond, editors, *Readings on Planning*. Morgan Kaufmann, 1990.

[II90] M. Inaba and H. Inoue. Vision based robot programming. *Robotics Research 5*, 1990.

[IPK04] Soshi Iba, Chris Paredis, and Pradeep Khosla. Interactive multi-modal robot programming. In *9th International Symposium on Experimental Robotics*, June 2004.

[Joh93] Gunnar Johannsen. *Mensch-Maschine-Systeme*. Springer, 1993.

[Kai97] Michael Kaiser. *Interaktive Akquisition elementarer Roboterfähigkeiten*. Infix, 1997.

[Kam04] Peter Kammermeier. *Verteilte taktile Stimulation zur Vermittlung mechanischer Berührungsinformation in Telepräsenzanwendungen*. Düsseldorf, VDI Verlag, 2004.

[Kan97] Makoto Kaneko. Scale-dependent grasps. *Eighth international symposium of Robotics Research*, 1997.

[KFD95] M. Kaiser, H. Friedrich, and R. Dillmann. Obtaining good performance from a bad teacher, 1995.

[KH95] R. Koeppe and G Hirzinger. Learning compliant motions by task-demonstration in virtual environments. *Fourth International Symposium on Experimental Robotics*, 1995.

[KI97] Sing Bing Kang and K. Ikeuchi. Toward automatic robot instruction from perception-mapping humangrasps to manipulator grasps. *Robotics and Automation, IEEE Transactions on*, 1997.

[KII92] Kuniyoshi, Inaba, and Inoue. Seeing, understanding and doing human task. *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, 1992.

[KII94] Kuniyoshi, Inaba, and Inoue. Learning by watching: extracting reusable task knowledge fromvisual observation of human performance. *Robotics and Automation, IEEE Transactions on*, 1994.

[KK94] Sing Bing Kang and Ikeuchi K. Determination of motion breakpoints in a task sequence from humanhand motion. *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, 1994.

[KLBL93] Robert S. Kennedy, Norman E. Lane, Kevin S. Berbaum, and Michael G. Lilienthal. Simulator sickness questionnaire: An enhanced method for quantifying simulator sickness. *International Journal of Aviation Psychology*, Vol. 3, No. 3, Pages 203-220, 1993.

[Kop89] H. Kopp. *Graphische Datenverarbeitung*. Hanser Studienbücher, 1989.

[MA01] Andrew T. Miller and Peter K. Allen. Graspit!: A versatile simulator for grasp analysis. Technical report, Department of Computer Science, Columbia University, 2001.

[MC03] Andrew T. Miller and Henrik I. Christensen. Implementation of multi-rigid-body dynamics within a robotic grasping simulator. Technical report, Royal Institute of Technology, Stockholm, Schweden, 2003.

[McC63] John McCarthy. Situations and actions and causal laws. *Stanford Artificial Intelligence Project, Memo 2*, 1963.

[MH69] John McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer, D. Michie, and M. Swann, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[MKCA03] Andrew T. Miller, Steffen Knoop, Henrik I. Christensen, and Peter K. Allen. Automatic grasp planing using shape primitives. Technical report, Dept. of Computer Science, Columbia University, New York; IAIM, University of Karlsruhe, Germany; Centre for Autonomous Systems, Royal Institute of Technology, Stockholm, Schweden, 2003.

[MT90] Toshihiro Matsui and Michiharu Tsukamoto. An integrated robot teleoperation method using multi-media display. In *The fifth international symposium on Robotics research*, pages 145–152, Cambridge, MA, USA, 1990. MIT Press.

[NAI+03] D.S. Nau, T.C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, and F. Yaman. Shop2: An htn planing system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[Nak03] S. Nakaoka. Generating whole body motions for a biped humanoid robot from captured human dances. Master of information science and technology in computer science, Graduate Schol of Information Science and Technoligy - University of Tokyo, Tokyo, January 2003.

[NH94] Delson N. and West H. Robot programming by human demonstration: the use of humaninconsistency in improving 3d robot trajectories. *Intelligent Robots and Systems '94. 'Advanced Robotic Systems and the Real World', IROS '94. Proceedings of the IEEE/RSJ/GI International Conference on*, 1994.

[NM04] Monica Nicolescu and Maja Mataric. Natural methods for robot task learning: Instructive demonstrations, generalization and practice. *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 241–248, 2004.

[NNIY02] A. Nakazawa, S. Nakaoka, K. Ikeuchi, and K. Yokoi. Imitating human dance motions through motion structure analysis. In *Proceedings of the 2002 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*, pages 2539–2544. Lausanne, Switzerland, October 2002.

[NS63] Allen Newell and H. A. Simon. Gps, a program that simulates human thought. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. MvGraw-Hill, 1963.

[otNJSC06] Official Robonaut Webpage of the NASA Johnson Space Center. Robonaut projekt homepage. `http://vesuvius.jsc.nasa.gov/erer/html/robonaut/robonaut.html`, January 2006.

[Ped86] E. P. D. Pednault. Formulating multiagent dynamic-world problems in the classical planing framework. In Michale P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 47–82, 1986.

[PGT04] Joelle Pineau, Geoffrey J. Gordon, and Sebastian Thrun. Applying metric-trees to belief-point pomdps. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[RD91] M. Huck R. Dillmann. *Informationsverarbeitung in der Robotik*. Springer, 1991.

[Rea94] James Reason. *Menschliches Versagen*. Spektrum, Akad. Verl., 1994.

[Rie97] Markus Riepp. *Wissensbasierte Parametrisierung von Operatorsequenzen*. 1997.

[RN03a] S. Russell and P. Norvig. *Artifcial Intelligence - A Modern Approach*. Pearson Education, 2 edition, 2003.

[RN03b] Stuart J. Russell and Peter Norvig. *Artificial intelligence*. Prentice Hall, 2. ed., internat. ed. edition, 2003.

[Sar82] R.F. Sarraga. *Computation of Surface Areas in GMSolid*. IEEE Computer Graphics and Applications, 1982.

[Say99] Craig Sayers. *Remote control robotics*. Nueva York, EUA : Springer, 1999.

[SB96] Hans-Jürgen Siegert and Siegfried Bocionek. *Robotik: Programmierung intelligenter Roboter*. Springer, 1996.

[Seg88] Alberto M. Segre. *Machine learning of robot assembly plans*. Kluwer, 1988.

[SF95a] W. Brent Seales and Olivier D. Faugeras. Building three-dimensional object models from image sequences. *Comput. Vis. Image Underst.*, 61(3):308–324, 1995.

[SF95b] W. Brent Seales and Olivier D. Faugeras. Building three-dimensional object models from image sequences. *Computer Vision and Image Understanding*, 1995.

[SF95c] W. Brent Seales and Olivier D. Faugeras. Building three-dimensional object models from image sequences. *Computer Vision and Image Understanding*, 1995.

[SK93] Tso S.K. and Liu K.P. Visual programming for capturing of human manipulation skill. *Intelligent Robots and Systems '93, IROS '93. Proceedings of the 1993 IEEE/RSJ International Conference on*, 1993.

[Son71] E. J. Sondik. *The optimal control of partially observable Markov Decision Processes*. PhD thesis, Stanford university, 1971.

[Sus73] G. A. Sussman. A computational model of skill acquisition. *M.I.T. AI Lab. Memo no. AI-TR-297*, 1973.

[T.B86] T.B.Sheridan. Human supervisory control of robot systems. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 808–812. IEEE, April 1986.

[TBF05] S. Thrun, W. Burghard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

[TH92] Takahashi T. and Ogata H. Robotic assembly operation based on task-level teaching in virtualreality. *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, 1083-1088 vol.2, 1992.

[THSy93] Takahashi T., Ogata H., and Muto S.-y. A method for analyzing human assembly operations for use inautomatically generating robot commands. *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, 695-700 vol.2, 1993.

[TK96] Chao-Ping Tung and A.C. Kak. Integrating sensing, task planning, and execution for roboticassembly. *Robotics and Automation, IEEE Transactions on*, 187-2018 vol.12, 1996.

[Wel94] Daniel Weld. An introduction to least-commitment planing. *Artificial Intelligence Magazine*, pages 27–61, 1994.

[Wik] Wikipedia. www.wikipedia.org.

[YMK96] Jiar Y., Wheeler M., and Ikeuchi K. Hand action perception for robot programming. *Intelligent Robots and Systems '96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on*, 1586-1593 vol.3, 1996.

Index