Lernen von Montagestrategien in einer verteilten Multiroboterumgebung

Inauguraldissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

vorgelegt von

Markus C. Ferch

Dez. 2001

Universität Bielefeld
Technische Fakultät
AG Technische Informatik
Postfach 100131
D–33501 Bielefeld

gedruckt auf alterungsbeständigem Papier nach DIN-ISO 9706

Bedanken möchte ich mich bei:

First of all Torsten Scherer, Yorck von Collani, Joachim Ferch, Christian Scheering, Thorsten Graf, Peter Meinicke, Bernd Rössler, Dirk Müller, Frank Röben, Jianwei Zhang, Gerhard Sagerer, Helge Ritter, Jochen Steil, Alois Knoll und – last but not least Angelika Deister.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Einleitung

Forschungen im Bereich der Montage können auf eine lange Tradition zurückblicken. Klassische Verfahren beschreiten meist den Weg der Demontage, wobei sie zunächst ein bereits bestehendes Aggregat zerlegen und dabei alle möglichen Zerlegungssequenzen generieren, um aus ihnen mittels einer Heuristik die optimale Montagesequenz auszuwählen. Diese Methoden garantieren das Auffinden einer Lösung auch bei sehr komplexen Konstruktionen. Dafür werden dann allerdings bereits recht ausgeklügelte Suchverfahren (A^* oder AA^* (BANDER AND WHITE, 1998)) verwendet. Hinzu kommt, daß die Einzelbauteile gewisse geometrische Bedingungen erfüllen müssen.

Beobachtet man jedoch die Vorgehensweise eines Menschen bei der Montage von Aggregaten, stellt man fest, daß hier weitestgehend vorwärts gerichtete Strategien verfolgt werden. Damit ist gemeint, daß eine Person, ein endgültiges Ziel vor Augen, schrittweise beginnt, die Einzelteile zusammenzufügen, und wenn sie feststellt, daß sie einen Fehler gemacht hat, diesen zu korrigieren versucht. Es wird also in abgewandelter Form das Prinzip des Versuchs und Irrtums verfolgt, wobei sich der Versuch meist an dem gewünschten Ziel orientiert und nicht völlig dem Zufall überlassen wird.

Jeder einzelne Montageschritt wird von einer Vielzahl unbewußter Handlungen wie Umgreifen, Umorientieren, Fixieren etc. begleitet, die durch langjähriges Training erworben wurden. In technischen Systemen, deren haptische Fähigkeiten weit eingeschränkter sind als die des Menschen, muß deshalb beim Erlernen von Montagestrategien mit langen Lernphasen gerechnet werden, besonders wenn man nicht nur einen Aspekt, z.B. das Finden einer Montagesequenz mit Lernmechanismen, realisieren möchte, sondern wenn sich das gesamte System, wie in dieser Arbeit, auf lernende Methoden abstützt. In solchen Fällen bieten sich besondere Architekturen an wie z.B. Layered-Learning (STONE, 1998) oder AA-learning (WENG ET AL., 1998).

In der vorliegenden Arbeit wurden verschiedene Aspekte lernender Systeme auf ein reales Mehrrobotersystem (Abb. 3.1) übertragen. Dabei wurden sowohl Einzelfähigkeiten mit Hilfe von Lernmechanismen antrainiert als auch das globale Verhalten, hier die Montage von Aggregaten aus Spielzeug, durch ein vorwärtsgerichtetes, zielorientiertes Verstärkungslernen untersucht.

Zur Realisation wurden unterschiedlichste Verfahren wie Funktionsapproximation, PCA, Markovsche Entscheidungsprozesse, Graphrepräsentation und -vergleich etc. verwendet. Darüber hinaus wurde ein Objektmodell zur mentalen Simulation der zu konstruierenden Bauteile realisiert und eine neue Repräsentation unendlicher Zustands-/Aktionsräume entwickelt. Neben der Fragestellung des Lernens wurde eine verteilte Roboteransteuerung verwirklicht, die es erlaubt, mehrere Manipulatoren gleichzeitig in einem sich überschneidenden Arbeitsbereich unabhängig voneinander zu betreiben. Besonderer Wert wurde dabei auf die Kooperation gelegt.

Struktur der Arbeit

Die Arbeit gliedert sich in zwei aufeinander aufbauende Teile. Der erste Teil umfaßt die Kapitel 1-3 und beschäftigt sich mit der Struktur des entwickelten Multirobotersystems. Dabei werden Kommunikationsmechanismen zwischen den realen Robotern / Robotersteuerungsprogrammen sowie Basisfertigkeiten der einzelnen Roboter thematisiert. Teil II, mit den Kapiteln 4-7, beschäftigt sich mit dem Erlernen von Handlungssequenzen in einem Multirobotersystem. Die im zweiten Teil entwickelten Lernverfahren basieren auf den im ersten Teil beschriebenen Mechanismen und behandelt u.a. die besonderen Schwierigkeiten eines verteilten Lerners. Andererseits werden die erweiterten Möglichkeiten eines Mehrrobotersystems, wie z.B. kooperatives Greifen, genutzt.

Kapitel 1

In diesem Kapitel wird ein Überblick über vorhandene Robotersteuerungssysteme und verteilte Systeme gegeben. Dabei stehen die Möglichkeiten zur Ansteuerung mehrerer Roboter im Vordergrund.

Kapitel 2

Das Kapitel beschreibt das im Rahmen dieser Arbeit entwickelte Kommunikationsprotokoll zum Austausch von Datenobjekten im Sinne objektorientierter Programmierung und erklärt den Aufbau eines verteilten Robotersystems.

Kapitel 3

Hier werden die für das spätere Lernverfahren implementierten Anwendungen näher erläutert. Darunter fallen die eigentlichen Roboteranwendungen, Programme zur Feinpositionierung und Objekterkennung sowie eine Anwendung zur Verwaltung des Arbeitsraumes, die in Zusammenarbeit mit den eigentlichen Roboteranwendungen Kollisionen verhindert.

Kapitel 4

Gibt einen Überblick über die gängigsten zustandsbasierten, unüberwachten Lernverfahren.

Kapitel 5

Beschreibt das im Rahmen dieser Arbeit entwickelte Objektmodell. Dies wird verwendet, um Raumrelationen zu bestimmen und um festzustellen, ob ein Montageschritt überhaupt sinnvoll ist.

Kapitel 6

Beschäftigt sich mit der Repräsentation von Zuständen und Aktionen in einem gemeinsamen Graphen.

Kapitel 7

Geht detaillierter auf einige der verwendeten Lernverfahren und die dabei zu beachtenden Details ein.

Teil I

Verteilte Multirobotersteuerung

1

Überblick über Robotersteuerungen und verteilte Syteme

1.1 Robotersteuerungen

In diesem Kapitel sollen verschiedene Robotersteuerungssysteme vorgestellt werden, insbesondere unter dem Gesichtspunkt ihrer Verwendbarkeit mit mehreren Robotern. Dazu sind zwei Aspekte zu betrachten: die Programmiersprache und das Betriebssystem.

Da die Anzahl an Robotersteuerungen und Programmiersprachen sehr groß ist, kann hier nur ein unvollständiger Abriß gegeben werden. Nahezu jeder Roboterhersteller hat eine oder mehrere eigene Steuerungssprachen entwickelt. Wenn man bedenkt, daß Unimation 1962 gegründet wurde, der erste offizielle Fortran Standard Fortran II, IV, 66 aber erst 1972 verabschiedet wurde, so ist das nicht weiter verwunderlich. Die ältesten Sprachen lehnen sich an Basic an und werden interpretiert, einige Systeme verwenden eine Pascal/Algol ähnliche Syntax und bieten durch die Verwendung von Compilern etwas mehr Performance. Systeme wie die PA-Library von Mitsubishi sind eher selten. Hier wird keine neue Programmiersprache definiert, sondern vorhandene Sprachen werden um die nötige Funktionalität zur Robotersteuerung erweitert, ohne den Programmierer einzuschränken.

Die zum Teil sehr alten Systeme wie z.B. V^+ (lag bereits 1998 in der Version 11 vor) haben natürlich einen gewissen Wandel erfahren, so daß heute fast alle Systeme Kommunikationsschnittstellen wie z.B. TCP/IP unterstützen. Da die Entwicklung auf diesem Gebiet jedoch sehr rasant verläuft, können die proprietären Systeme nicht mit den sich schnell wandelnden Standards mithalten, so daß die Schnittstellen nur rudimentäre Funktionalität bieten.

Grundsätzlich werden zur Beschreibung von Positionen verschiedene Datentypen benötigt. Im kartesischen Raum benutzt man üblicherweise homogene 4×4 Matrizen (PAUL, 1982), im Konfigurationsraum n-dim. Vektoren, wobei n die Anzahl der Gelenke ist. Bewegungsbefehle werden meist in einem separaten Task in eine Abfolge von Gelenkwinkeln (Trajektorie) umgerechnet (Kinematik) und mittels Gelenkreglern an die Leistungselektronik weitergegeben. Die Gelenkregelung findet bei den aktuellen Systemen auf separater Hardware statt. Systeme für redundante Roboterarme regeln nicht die Gelenkpositionen, sondern

die Geschwindigkeit der Gelenke (Jakobimatrix) wie z.B. die PA-Library (s.u.). Aber auch hier wird auf der Programmierebene kartesisch positioniert.

Die Trajektoriengenerierung muß sich nicht nur um das Erreichen des Zielpunktes kümmern, sondern auch Geschwindigkeitsprofile berechnen sowie zwischen aufeinanderfolgenden Bewegungen überblenden können. Manche Systeme bieten nur die Möglichkeit, geradlinige Trajektorien zu generieren, sei es im kartesischen, sei es im Konfigurationsraum. Andere erlauben es dem Anwender, komplexe Bahnen zu spezifizieren, die dann mittels Splinealgorithmen (meist B-Splines) interpoliert werden.

Um komplexere Positionsbeziehungen auszudrücken, fehlen den meisten Sprachen die nötigen Konstrukte wie z.B. Positionsgleichungen (PAUL, 1982). Stattdessen wird eine große Anzahl von Bewegungsbefehlen angeboten, z.B. zur Bewegung in Toolkoordinaten etc. . Viele Sprachen bieten auch Rechenoperationen mit Transformationen und Vektoren an, um Positionsgleichungen selbst zu berechnen. Es gibt allerdings auch Implementationen, die noch nicht einmal das ermöglichen. Wenn man keine Möglichkeiten hat, Positionsgleichungen zu berechnen, so finden sich doch zumindest Befehle, die es erlauben, Werkzeugtransformationen (Tool) und Referenzkoordinatensysteme (Basistransformation) zu definieren, die dann in den oben erwähnten spezialisierten Funktionen berücksichtigt werden.

Die Verwendung externer Sensorik hat den Bedarf nach Eingriffsmöglichkeiten in die Trajektorienplanung geweckt. Manche Systeme unterstützen den Entwickler dahingehend, daß er im sogenannten Interpolationstakt Einfluß auf die Trajektorie nehmen kann. Die vom Trajektoriengenerator in festen Zeitabständen berechneten Positionen lassen sich vor Umrechnung in die Gelenkwerte/Gelenkgeschwindigkeiten vom Benutzer nochmals modifizieren. Systeme, die solche Mechanismen nicht vorsehen, sind jedoch zumindest in der Lage, die Trajektorie mit einer zweiten Bewegung zu überlagern.

Die Ausführung spekulativer Bewegungen hat bisher noch keines der betrachteten Lösungen thematisiert, was die Realisation gewisser Konzepte der Kollisionsvermeidung erschwert.

Im Anschluß sollen einige Systeme (ohne Anspruch auf Vollständigkeit) kurz beschrieben werden.

1.1.1 Programmiersprachen

VAL/V$^+$ (Adept)

Die erste Roboter-Programmiersprache, die jemals entwickelt wurde, ist VAL/V$^+$ (ADEPT TECHNOLOGY, 1993). Sie basiert auf den Konzepten von BASIC und läuft nur auf den speziell dafür vorgesehenen Rechnern der Roboterhersteller Unimation/Adept. Die Programme/Skripts werden interpretiert, was es erlaubt, sie zur Laufzeit zu modifizieren. Sie sind in einer nicht mehr eindeutig nachvollziehbaren Weise mit dem Betriebssystem verzahnt. So spricht Adept selbst vom V$^+$-Betriebssystem und von V$^+$-Programmen. Es können mehrere Programme gleichzeitig, aber nur mit unterschiedlichen Prioritäten (Tasks) gestartet werden, sie sind reentrant. Positionen werden über Transformationen beschrieben, die aber nur über Roll, Pitch und Yaw-Winkel angegeben werden können. Es gibt keine Möglichkeiten, mit Positionen zu rechnen, es gibt keine Positionsgleichungen. Damit entfällt auch die Möglichkeit der engen Koppelung mehrerer Roboter. Peripherie, die nicht vom Hersteller unterstützt wird, z.B. Netzwerke, serielle Schnittstellen, Busse wie CAN-BUS, Inter-BUS etc., kann nicht eingebunden werden.

Zur Einbindung unterstützter Sensorik läßt sich die kartesische Position im Interpolationstakt modifizieren; auf Grund des geschlossenen (proprietären) Systems ist man aber auf die von Adept unterstützte Sensorik beschränkt. Konstrukte wie z.B. unendliche Bewegungen fehlen ebenfalls. Bei dem Konzept der unendlichen Bewegungen verfolgt die Robotersteuerung permanent eine Zielposition, auch wenn sie bereits erreicht ist. Die Zielposition läßt sich seitens des Benutzers, aus einer anderen Task, zu jedem beliebigen Zeitpunkt modifizieren worauf hin das System einen neuen Zielpunkt verfolgt. Es gibt eine Reihe von Varianten dieses Konzepts die z.B. nur erlauben kleine Änderungen in der Zielposition durch zu führen, oder nicht die gesammte Zielposition sondern nur eine Teiltransformation einer Positionsgleichung zu verändern. In solchen Fällen müssen andere Wege beschritten werden. Dem Basic ähnlichen Dialekt fehlen sämtliche Konzepte moderner Programmiersprachen.

Um mehrere Roboter zu steuern, muß für jeden Roboter ein eigener Task (ein eigenes Programm) gestartet werden. Die Steuerung mehrerer Roboter aus einem Programm ist nicht vorgesehen.

KRC 1 (Kuka)

Mit der PC-Robotersteuerung KRC1 (SCHNEIDER, 1998) realisierte Kuka die Ansteuerung ihrer Roboter von einem PC aus. Begründet wird dieses Vorgehen zum einen mit den geringeren Kosten, zum anderen möchte man dem industriellen Benutzer eine ansprechende graphische Führung bieten. Dies ist aus Sicht von Kuka nur durch den Einsatz zweier Betriebssysteme, VxWorks für den echtzeitfähigen Teil und WIN95 für die grafische Präsentation möglich. Um diese vom PC-Standard nicht vorgesehene Lösung zu bieten, wird genau wie bei Mitsubishis Lösung mit der PA-Library eine spezielle Steckkarte benötigt, die neben der Trennung der beiden Betriebssysteme durch Generieren eines nicht maskierbaren Interrupts auch die Gelenkregelung übernimmt, deren Echtzeitanforderungen selbst die Möglichkeiten des VxWorks-Betriebssystems überfordert. Programmiert werden Roboter über die Kuka-eigene, proprietäre Programmiersprache KRL.

Die Anzahl der von einem Rechner ansteuerbaren Roboter wird durch diese Architektur limitiert, da sich die Gelenkregler auf derselben Karte wie der sogenannte „LP-Realtime Chip“ befindet, der für die Trennung der beiden Betriebssysteme zuständig ist.

Zur Kommunikation mit externer Sensorik verfügt Kukas Lösung über eine Vielzahl von Kommunikationskanälen, angefangen bei seriellen Schnittstellen über Ethernet bis hin zu Inter-Bus. Durch Verwendung von handelsüblichen PC-Komponenten läßt sie sich theoretisch beliebig erweitern, wenn man bereit ist, die nötigen Treiber für VxWorks selbst zu entwickeln.

KAREL (Fanuc Robotics)

Karel heißt die Roboterprogrammiersprache von Fanuc/GMF. Sie lehnt sich in ihrer Syntax stark an Pascal an und besitzt auch deren prozeduralen Charakter. Positionen lassen sich über homogene Transformationen und Vektoren beschreiben, mit denen auch gerechnet werden kann. Positionsgleichungen fehlen allerdings. In jedem Task kann nur ein Roboter programmiert werden, was die enge Koppelung mehrerer Roboter unmöglich macht; maximal können 16 Achsen angesteuert werden. Kommunikation mit anderen Rechnern ist nicht vorgesehen.

Karel ist wie die meisten kommerziellen Systeme proprietär und wird nur für/mit der dazugehörigen Hardware ausgeliefert. Etliche Komponenten der Sprache werden nur optional angeboten.

BAPS (Bosch)

Von der Firma Bosch wurde die an Pascal angelehnte Steuersprache Baps entwickelt. Sie ist im wesentlichen für Roboter gedacht. Durch die Verwendung eines Compilers (im Gegensatz zu einem Interpreter wie bei vielen anderen Sprachen) ist eine Remoteentwicklung z.B. vom PC aus möglich. Baps bietet die Möglichkeit, Programmteile in „shared library“ zu verlagern, um sie an Dritte weitergeben zu können, ohne seinen Quellcode offenlegen zu müssen.

Theoretisch lassen sich mit Baps mehrere Roboter aus einem Programm heraus ansteuern. Es bestehen ebenfalls Kommunikationskanäle (serielle Leitung, TCP/IP). Positionsgleichungen werden nicht unterstützt. Baps ist proprietär und läuft auf der von Bosch dafür vorgesehenen Hardware.

1.1.2 Libraries

PA-Library (Mitsubishi Heavy Industries)

Die PA-Library von Mitsubishi ist neueren Datums (1998). Es handelt sich um eine C - Library zur Ansteuerung des PA10-Roboters. Der Quellcode ist verfügbar, so daß sie sich auf beliebige Betriebssysteme portieren läßt. Das liegt auch daran, daß die Library nichts weiter tut als den Funktionsaufruf und die Parameter an eine ISA-Karte weiter zu reichen.

Um mehrere Roboter anzusteuern ist die Library in der Lage, mehrere ISA-Karten in einem Rechner anzusprechen (`pa_open_arm(ArmNo1)...`). Die Position des Roboters läßt sich in Gelenk- und kartesischen Koordinaten steuern. Es gibt keine Funktionen zum Rechnen mit Matrizen bzw. Vektoren. Es ist auch nicht möglich, Positionsgleichungen zu erstellen, weshalb sich eine enge Koppelung mehrerer Manipulatoren (in einer gemeinsamen kinematischen Kette) nicht realisieren läßt. Durch den Einsatz eines beliebigen, möglichst echtzeitfähigen OS und der Programmiersprache C stehen alle nur erdenklichen Kommunikationskanäle mit anderen Rechnern/Robotern zur Verfügung.

Der Sensoreinsatz wird durch die Möglichkeit unterstützt, die Position im Interpolationstakt zu manipulieren.

RCCL

RCCL (Robot Control C Library) wurde Mitte der 80er Jahre an der Mc Gill Universität in Kanada entwickelt (HAYWARD AND PAUL, 1986). Wie die PA-Library hat sie den Vorteil, Library einer ANSI-genormten Programmiersprache zu sein und besitzt alle damit verbundenen Vorteile. Im Gegensatz zur PA-Library von Mitsubishi verlagert sie aber die eigentliche Arbeit nicht auf ein zusätzliches Prozessorboard, sondern führt alle Berechnungen bis zur Trajektoriengenerierung auf dem Hostrechner durch. Dazu benötigt sie ein multithreading/tasking fähiges Betriebssystem (vorzugsweise Unix), das jedoch gewisse Echtzeitanforderungen erfüllen muß. Die Gelenkregelung wird von RCCL nicht thematisiert; hier wird weiterhin roboterspezifische Hardware benötigt.

Es ist zumindest theoretisch möglich, beliebig viele Roboter anzusteuern. Positionen lassen sich über Gelenkwinkel und kartesisch (homogene 4×4 Matrizen) definieren. Mit letzteren kann gerechnet werden. Roboterbewegungen werden über Positionsgleichungen spezifiziert.

Sensordaten können über Callback-Funktionen im Interpolationstakt Einfluß auf die Trajektorie nehmen, darüber hinaus stehen alle gegebenen Möglichkeiten des jeweils verwendeten Betriebssystems offen.

Nachteil von RCCL ist der nach außen hin monolithische Aufbau. Ohne detaillierte Kenntnisse der (nicht dokumentierten) Interna ist es nicht möglich, andere als die vorgesehene Hardware (Gelenkregler) anzusteuern. Einige der internen Lösungsansätze sind sehr umständlich und begründen sich durch den Versuch, auch Hardware wie die VAX Architektur zu unterstützen. Weiterhin setzt RCCL voraus, einen in Grenzen echtzeitfähigen Zugang zur Hardware des Betriebssystems zu bekommen. Unter den verwendeten Unix-Varianten limitiert das die Zyklusraten des Trajektoriengenerators auf 10ms. Deshalb führt die Library auch keine Gelenkregelung durch, sondern erwartet eine roboterspezifische Lösung. Ebenfalls nicht verfügbar sind Lösungen für redundante (7-achsige) Roboter mittels Jakobimatrix.

ARCL

ARCL (Advanced Robot Control Library) (CORKE AND KIRKHAM, 1995) wurde an der CISRO in Preston, Australien entwickelt und ist eine Reimplementierung von RCCL. Sie bietet exakt die Funktionalität von RCCL, besitzt ein leicht verändertes API und einige wenige neue Konzepte im Umgang mit Callback-Funktionen auf Interpolationsebene. Der Aufbau von ARCL wurde jedoch deutlich modularer gestaltet, was eine leichtere Anpassung an verschiedene Robotertypen erlaubt. Um eine bessere Echtzeitfähigkeit zu gewährleisten, hat man sich bei ARCL auf die Verwendung des Betriebssystems VxWorks konzentriert. Unter anderen nicht echtzeitfähigen UNIX Varianten existiert aber eine Simulationslösung.

1.2 Verteilte Systeme

Um kooperativ mit mehreren Robotern arbeiten zu können, die sich nicht nur von einem Programm aus ansteuern lassen, ist es notwendig, Informationen auszutauschen. Die indirekte Methode über externe Sensoren (Kraftsensoren, Kameras) ist sehr mühsam, rechenintensiv und zeitaufwendig, weshalb sie hier nicht weiter diskutiert werden soll.

Die weite Verbreitung des Internets bzw. seines Protokolls (TCP/IP) hat in den letzten Jahren dazu geführt, daß eine ganze Reihe von Bibliotheken entstanden sind, die dem Programmierer Kommunikationsaufgaben erleichtern sollen, ohne daß er sich mit den Interna wie z.B. Socketprogrammierung oder Byteorder auseinandersetzen muß. Alle Systeme haben dabei eine gewisse Zielsetzung wie z.B. verteiltes Rechnen oder verteilte Steuerung. Der Höhepunkt dieser Entwicklung war bis jetzt die Spezifikation der Programmiersprache Java, die die nötigen Kommunikationsmechanismen wie z.B. Serialisierung bereits in die Sprache integriert und sich erstmals Gedanken um den Sicherheitsaspekt von verteilten Anwendungen macht. So werden Java-Applets in einer sogenannten Sandbox gestartet, wo sie beliebigen Schaden anrichten können, ohne wirklich etwas zu zerstören. Um Java auf allen Hardware-Plattformen und unter allen Betriebssystemen lauffähig zu bekommen, wendete

man sich erstmals von der Verwendung eines Compilers ab und ging zurück zu interpretierbarem Code.

Neben Java sind PVM, CORBA und ACE, drei Meilensteine im Bereich der verteilten Systeme die hier kurz umrissen werden sollen.

1.2.1 PVM

PVM (Parallele Virtuelle Maschine) (GEIST ET AL., 2000) ist eines der am weitesten verbreiteten Software-Pakete zum Entwickeln verteilter Systeme. Es ermöglicht die Verteilung einer Berechnung auf eine große Zahl von Rechnern, ist frei erhältlich und wurde auf eine große Zahl von Plattformen portiert: Win95, NT 3.5.1, NT 4.0 , Linux, Solaris 2.x, SCO, NetBSD, BSDI, FreeBSD, AIX 3.x, AIX 4.x, HPux, OSF, NT-Alpha , IRIS 5.x, IRIS 6.x, Cray YMP, T3D, T3E, Cray2, Convex Exemplar, IBM SP2, 3090, NEC SX-3, Fujitsu, Amdahl, TMC CM5, Intel Paragon, Sequent Symmetry, Balance.

Zunächst muß von Benutzerseite festgelegt werden, welche Hostrechner an dem System teilnehmen sollen. Dies geschieht manuell und kann während des Systemlaufs verändert werden. Anschließend können Applikationen auf den verschiedenen Rechnern plaziert werden. Zur optimalen Nutzung der eingesetzen Hardware lassen sich deren Fähigkeiten (Rechenleistung) ermitteln. Die Granularität, in der sich PVM-Anwendungen parallelisieren lassen, sind sogenannte Tasks, die zumindest auf Unix-Rechnern mit Prozessen gleichzusetzen sind. Die Anzahl der Prozesse pro Rechner wird von PVM nicht beschränkt. Die verteilten Tasks lassen sich in sogenannte Collections gruppieren, unter denen Botschaften ausgetauscht werden können. Um trotz der Vielfalt der Architekturen (hier insbesondere Hardware) beliebig Daten austauschen zu können, werden alle Botschaften zunächst in das Netzwerkformat (big-endian) gewandelt und auf dem Empfangsrechner wenn nötig zurückkonvertiert. Um die Kommunikation zwischen lokalen PVM-Tasks – also solchen, die auf einem Rechner aber auf z.B. unterschiedlichen Prozessoren laufen – zu optimieren, werden besondere Kommunikationswege geschaffen (Pipes, shared Memory, ...). Hier müssen jedoch speziell angepaßte PVM-Versionen verwendet werden.

PVM ist in C geschrieben, kann inzwischen aber auch aus C++ und Fortran-Programmen benutzt werden. Es wurde für das Verteilen rechenintensiver Probleme auf mehrere Rechner geschrieben und ist in diesem Bereich eines der ältesten Systeme. Eine der prominentesten Anwendungen ist das make tool zum Übersetzen von Programmen. Mittels PVM ist es in der Lage, den Übersetzungsprozeß auf theoretisch beliebig viele Rechner zu verteilen. Das Problem globaler Systemvariablen wie z.B. `errno` wurde durch das Einführen eigener Variablen `pvm_errno` realisiert. Dahinter verstecken sich Makros, die im Moment der Abfrage den betreffenden Wert ermitteln. Dadurch lassen sich bereits bestehende Programme, die nachträglich parallelisiert/verteilt werden sollen, nicht ohne weiteres in PVM übertragen.

Die einzelnen Tasks einer virtuellen Maschine werden durch Task-Identifikationsnummern (TID's) gekennzeichnet. Schreib- und Lesebefehle richten sich an die TID´s, die folglich eindeutig sein müssen. Um eine fehlerhafte Vergabe von TID´s auszuschließen, übernehmen die PVM-Dämonen diese Arbeit. Durch diese Vorgehensweise stellt sich das Problem des gleichzeitigen Versendens von Botschaften an mehrere Tasks. Es muß seriell geschehen, Mechanismen wie IP-Multicasting sind nicht vorhanden. Um dem Benutzer jedoch die Arbeit zu erleichtern, gibt es die Möglichkeit, Taskgruppen zu definieren und an alle Mitglieder einer Gruppe Botschaften zu senden, die dann vom PVM-System nacheinander an jeden einzelnen versendet werden.

Der PVM-Dämon dient nicht nur der Vergabe von Identifikatoren, sondern ist auch in der Lage, Tasks zu starten. Es ist nicht notwendig, jeden Task von Hand zu initialisieren. Üblicherweise wird ein Mastertask (auch Initialisierungstask genannt) geschrieben, der die PVM-Dämonen veranlaßt, die eigentlichen Programmtasks zu starten. Dazu müssen diese aber Zugriff auf den Binärcode der jeweiligen Tasks besitzen. PVM geht hier davon aus, daß entweder ein Netzwerk-File-System (NFS) vorliegt oder die Programme vom Benutzer per Hand auf den jeweiligen Rechner transferiert wurden.

1.2.2 CORBA

Die Common Object Request Broker Architecture ist eine herstellerunabhängige, also genormte Architektur[1], die von Rechnern genutzt werden kann, um über ein Netzwerk miteinander zu kooperieren. Das verwendete Protokoll wird IIOP genannt. Es gibt CORBA-Implementationen auf so vielen Rechnerarchitekturen, daß eine Aufzählung den Rahmen sprengen würde. Die meisten CORBA-Anwendungen betreffen große Web-Server-Systeme mit großem Anfragevolumen. Es existieren auch schlanke CORBA-Implementationen für embedded Systems, die jedoch nur Teile der CORBA-Spezifikation implementiert haben und sich deshalb nicht wirklich CORBA nennen dürfen.

Neben der großen Zahl an Rechnerarchitekturen und Betriebssystemen, für die CORBA existiert, werden auch mehrere Programmiersprachen unterstützt (C, C++, Java, COBOL, SmalTalk, Ada, Lisp und Pyton). Java hat sich auf Grund der besonders gelungenen API jedoch durchgesetzt, was eine Ursache für die meist sehr geringe Geschwindigkeit von CORBA-Anwendungen ist.

Damit sich die Objektrepräsentationen der verschiedenen Sprachen untereinander verstehen können, wurde eine weitere Objektrepräsentation (IDL: Interface Description Language) definiert. Für jedes Objekt muß zusätzlich noch eine IDL-Beschreibung generiert werden, bzw. wird aus der IDL-Beschreibung ein Objektgerüst generiert. Es müssen nur solche Operationen und Operanden im Interface definiert werden, die nach außen sichtbar sein sollen. Die IDL ist gewissermaßen der kleinste gemeinsame Nenner zwischen den unterschiedlichen Programmiersprachen, um Anfragen in einer Sprache und von einer CORBA-Implementation von Programme in einer anderen Sprache und Implementation beantworten zu können.

Jede Objektinstanz besitzt eine eindeutige Identifikationsnummer. Sie wird verwendet, um Dienste von dieser Instanz aufzurufen. Das aufrufende Programm muß dabei nicht wissen, ob die Instanz auf dem eigenen Rechner existiert oder auf einem anderen. Sogenannte ORBs sorgen dafür, daß die nötigen Informationen gegebenenfalls vom anfragenden Rechner zum Zielrechner geleitet werden und die Ergebnisdaten wieder zurückfließen.

CORBA entstand als Komponentenmodul für Programme, die Dienste anderer Applikationen nutzen wollten und hat sich inzwischen zu einem sehr mächtigen Tool entwickelt, das überwiegend im Web-Bereich Internetshopping (e-Business), Onlineretreavel etc. eingesetzt wird. Als Komponentenmodul für unvernetzte Rechner ist es inzwischen schon zu mächtig und wird daher in neuen Desktopentwicklungen nur noch ungern als Komponentenmodul verwendet.

[1] Die Normung wird von der OAG Objekt Architectur Group durchgeführt.

1.2.3 ACE

Das Adaptive Communication Environment (SCHMIDT, 2001),(SYYID, 2000) ist ein objektorientiertes Kommunikations-Werkzeug. Offiziell wurde es im Frühjahr 2000 herausgegeben. Es basiert auf C++[2], ist frei erhältlich und läuft auf einer breiten Palette von Betriebssystemen: Win32 (WinNT 3.5.x, 4.x, 2000, Win95/98, und WinCE mit MSVC++, Borland C++ Builder, und IBM's Visual Age auf Intel und Alpha Plattformen), den meisten UNIX Versionen (Solaris 1.x und 2.x, SPARC und Intel, SGI IRIX 5.x und 6.x, DG/UX, HP-UX 9.x, 10.x und 11.x, DEC/Compaq UNIX 3.x und 4.x, AIX 3.x und 4.x, DG/UX, UnixWare, SCO und frei erhältlichen UNIX Implementationen wie Debian Linux 2.x, RedHat Linux 5.2 und 6.0, FreeBSD und NetBSD), Echtzeit-Betriebssystemen (LynxOS, VxWorks, Chorus ClassiX 4.0, QnX Neutrino, und PSoS) sowie MVS OpenEdition und CRAY UNICOS.

Neben der reinen Kommunikation bietet es Lösungen für Ereignis-Demultiplexing, Dispatching von Eventhandlern, Verwaltung und Bearbeitung sogenannter Service Message Routinen, das dynamische (Re)Konfigurieren von verteilten Services sowie Synchronisationswerkzeuge für konkurrierende Prozesse. Dabei zielt es auf Echtzeitanwendungen und Systeme mit hohen Performanceanforderungen ab.

Die Verbreitung auf einer großen Anzahl von Plattformen ist nur durch das Entwickeln eines *OS Adapter Layers* möglich. Gekapselt werden: *Concurrency* und *Synchronization*, *Interprocess Communication (IPC)* und *shared Memory, Event-Demultiplexing* Mechanismen, explizites dynamisches Linken und Filesystem Mechanismen. Der OS Adapter Layer wurde in C geschrieben. Um ihn besser in objektorientierten Anwendungen einsetzen zu können, wird er nochmals von C++-Klassen gekapselt, die auf ihm aufbauen. Die folgenden Aspekte wurden gekapselt: *Concurrency* und *Synchronization*, IPC und Filesystemfunktionalität sowie Speichermanagement (shared Memory).

Auf dieser C++-Kapselungs-Fassade baut das eigentliche ACE-Framework zum Entwickeln verteilter Systeme auf:

- **Event-Demultiplexing-Komponenten**
  ACE-Reactor und Proactor sind mit den Mitteln der Vererbung zu erweiternde Demultiplexer. Sie dispatchen die von der jeweiligen Anwendung benötigten Ereignisbehandlungsroutinen auf der Grundlage von I/O, Timern, Signalen und Synchronisationsmechanismen (Conditions, Barriers etc.).

- **Service-Initialization-Komponenten**
  Die ACE-Acceptor und Connector-Komponenten entkoppeln die aktiven und passiven Rollen einer Anwendung beim Verbindungsaufbau, wie er bei Anwendungsspezifischen Aufgaben initial einmal durchgeführt werden muß.

- **Service-Konfigurations-Komponenten**
  Die ACE-Service-Configurator-Komponenten ermöglichen das Konfigurieren von Anwendungen, deren sogenannte Services dynamisch konstruiert wurden, sei es zum Zeitpunkt des Programmstarts, sei es zur Laufzeit.

- **Stream-Komponenten**
  Die ACE-Stream-Komponenten vereinfachen die Entwicklung von Kommunikations-Software wie z.B. Protokoll-Stacks auf Benutzerebene.

[2] Es existiert inzwischen auch eine Java Version.

- **ORB-Adapter-Components**
  ACE läßt sich mit Hilfe des sogenannten ORB‘s unsichtbar in CORBA-Implementationen integrieren. Eine auf ACE basierende CORBA-Implementation nennt sich TAO und behauptet von sich, echtzeitfähig zu sein (V.FAY-WOLF ET AL., 2000).

ACE kommt bereits mit einer Reihe prototypischer, verteilter Dienste, die kurz aufgelistet werden sollen:

- **Naming-Service**
  Diese Anwendung assoziiert Werte mit Namen. Die Assoziation ist eindeutig, ein Programm kann neue Assoziationen anlegen, abfragen und verändern.

- **Time-Service**
  Time-Services dienen zur Synchronisation von Uhren innerhalb eines lokalen oder globalen(wide area) Netzwerkes.

- **Token-Service**
  Token-Services sind Interprozeß-Mutexe bzw. Leser/Schreiber-Mutexe.

- **Server-Logging-Service**
  Unter dem Server-Logging-Service wird eine Anwendung verstanden, die es ermöglicht, benutzerdefinierte Logs zu erstellen, um systemglobale Ressourcen zu verwalten, die sich nicht mit einem Mutex beschreiben lassen.

- **Client-Logging-Service**
  Client-Logging-Services dienen dazu, Logginganfragen von einem lokalen Rechner zu multiplexen und an einen globalen Server-Logging-Service weiterzuleiten.

- **Logging-Strategy-Service**
  Ein Überwachungs-Modul, um die Ausgaben der einzelnen Services zu protokollieren.

ACE findet im Verhältnis zu CORBA nur geringe Beachtung im Internetbereich, wird jedoch durch seine hohe Geschwindigkeit im Bereich der Systemsteuerung z.B. bei Flugzeugen oder bei Telekommunikationssystemen (SIEMENS) eingesetzt.

2

Struktur eines verteilten Systems zur Multiroboteransteuerung

2.1 Kommunikationsprotokoll

In diesem Kapitel wird ein Konzept zur objektorientierten Kommunikation zwischen zwei Anwendungen vorgestellt. Es bildet die Grundlage für das im weiteren skizzierte und realisierte verteilte Multirobotersystem, ist aber nicht auf diesen Anwendungskreis begrenzt. Es handelt sich hierbei ausschließlich um die Beschreibung des Objekttransfers und nicht um das Design einer Client-Server-Struktur mit Dienstanbietern etc. Näheres dazu findet sich in den anschließenden Abschnitten. Dieser Teil ist vergleichbar mit dem Konzept der *Factorys* von CORBA, ohne ihren hohen Aufwand zu besitzen.

2.1.1 Grundlegendes Konzept

Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt, das es erlaubt, beliebige Objekte – hier in C++-Klassennotation[1] – über ein verbundenes Netzprotokoll zu übertragen. Verbunden wird hierbei im Gegensatz zu paketorientiert verstanden; z.B. ist TCP ein verbundenes Protokoll, während UDP eine paketorientierte Kommunikation darstellt (STEVENS, 1998). Um die Probleme verbindungsloser Protokolle zu umgehen, wird ausschließlich mit verbindungsorientierten Kommunikationsmedien/Protokollen gearbeitet.

Verwendet werden die Mechanismen der Vererbung und der Virtualität. Alle objektorientierten Sprachen, die über diese Mechanismen verfügen, sind prinzipiell geeignet, auf diese Weise Objekte zu versenden. Bei Spezifikation einer einheitlichen Datenrepräsentation ist sogar ein Austausch zwischen verschiedenen Programmiersprachen möglich.

Zum Versenden werden zwei Klassen definiert:

- **Objektkanäle**: Sie wickeln das Protokoll ab und sind für die eigentliche Versendung der Daten zuständig.

[1] keine notwendige Bedingung.

- **Datenobjekte**: Sei werden von einem Basisobjekt abgeleitet und stellen gewisse Unterstützungsroutinen zur Verfügung. Hierbei handelt es sich zum einen um die Datenserialisierung und zum anderen um die virtuelle Konstruktion.

2.1.2 Objektkanäle

Objektkanäle sind Punkt-zu-Punkt-Verbindungen. Sie sind nicht typisiert, das heißt alle Objekte, die von dem oben erwähnten Basisobjekt abhängen, sollen über sie versendet werden können. Hierin setzt sich das im folgenden beschriebene Konzept von ähnlichen Kommunikationstools wie z.B. DACS (FINK ET AL., 1998) ab. Konkret ist mit nicht typisiert gemeint, daß die Empfangsseite a priori nicht wissen muß, was für ein Objekt als nächstes von der sendenden Seite kommen wird. Dadurch ist es möglich, auf diesem Kommunikationskonzept ein Client-Server-Framework aufzusetzen, das ebenfalls untypisiert ist. Um diese Bedingungen realisieren zu können, werden Mechanismen zur virtuellen Konstruktion, also zur Instantiierung von Objekten, deren Typ erst zur Laufzeit feststeht, benötigt. Da C++ keine virtuelle Konstruktion kennt, wurde dieser Mechanismus in der vorliegenden Arbeit nachgebildet. Dazu werden alle virtuell konstruierbaren Objekte in einer Datenstruktur[2] registriert. Sie stellt

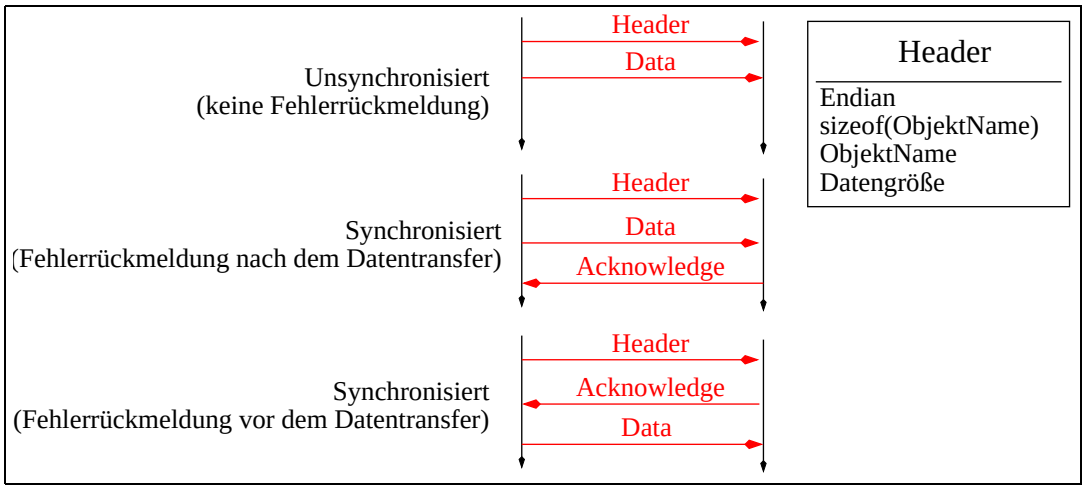


Abbildung 2.1: Die Daten werden in den Header und die eigentlichen Nutzdaten zerlegt. Wenn das Objekt unbekannt ist, müssen die Daten aus dem Kanal gelöscht werden, und möglicherweise muß eine Fehlermeldung an den Sender geschickt werden. Daraus entstehen die hier gezeigten Kommunikationsmodi.

eine Assoziation zwischen dem Objekt bzw. einem eindeutigen Objektbezeichner[3] (OB), wie er z.B. durch das run-time-type-information (RTTI) von C++ gegeben ist oder der zur Übersetzungszeit festgelegt werden kann[4], und einer Konstruktorfunktion, die ein leeres Objekt instantiiert, her. Vor dem Versenden der eigentlichen Nutzdaten eines Objektes wird vom Kanal zunächst der sogenannte Objektkopf (Header), bestehend aus dem OB und vorweg dessen Länge sowie der Nutzdatengröße, also der Anzahl zu sendender Nutzdatenbytes,

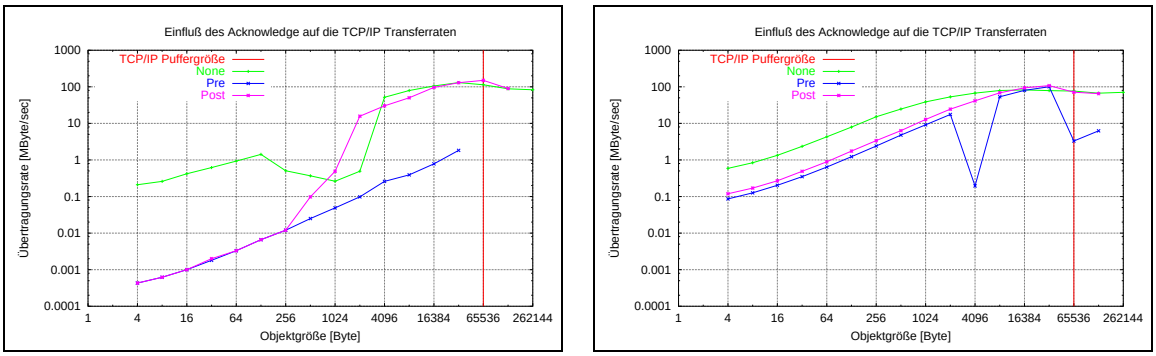
[2] Hashtabelle, Binärbaum.

[3] nicht zu verwechseln mit einer Instanz-ID wie bei CORBA.

[4] Um von den unterschiedlichen Namenskonventionen der Compiler unabhängig zu werden, bietet sich die zweite Methode an.

in den Datenstrom eingefügt. Der Empfänger sucht mit dem OB in der Datenbank die zugehörige Konstruktorfunktion aus, konstruiert (instantiiert) das Objekt und ruft eine Methode des Objektes auf, die das Zuweisen der Daten aus dem Datenstrom an die Klassenvariablen übernimmt.

Ein Problem entsteht, wenn kein Objekt des gewünschten Typs in der Datenbank registriert ist. In diesem Falle müssen die zum Objekt gehörenden Daten aus dem Datenstrom entfernt werden, da sonst keine weitere Kommunikation möglich ist. Für dieses Problem gibt es mehrere Lösungsmöglichkeiten:



(a) Ohne Objektpufferung

(b) Mit Objektpufferung

Abbildung 2.2: Diese zwei Grafiken veranschaulichen den Einfluß einer Bestätigung auf die theoretischen Nutzdatentransferraten (die Headerinformation und mögliche Bestätigungen werden nicht zu den Nutzdaten gezählt, die tatsächlich übertragenen/gepufferten Daten sind also größer). Dabei steht *None* für Übertragungen ohne Bestätigung, *Pre* für die Übermittlung einer Bestätigung nach den Protokolldaten und vor den Nutzdaten und *Post* für das Versenden von Bestätigungen nach Erhalt des gesamten Objektes, also hinter den Nutzdaten. Transfers ohne Bestätigung haben bei kleinen Objekten die größte Performance. Sie läßt sich durch Objektpufferung nochmals deutlich steigern. Die Post-Bestätigung ist der Pre-Bestätigung im Normalfall (nicht Fehlerfall) überlegen, kann bei Objekten größer 1024 Byte sogar zum Modus ohne Bestätigung aufschließen. Die Objektpufferung (nicht zu verwechseln mit den Send/Receive-Puffern des Betriebssystems) wirkt sich auf alle drei Modi positiv aus. Bei ungünstigen Konstellationen wie z.B. bei 4096 Byte und Pre-Bestätigung bricht die blaue Kurve jedoch auf den Wert der ungepufferten Übertragung zusammen. Die ebenfalls eingezeichneten senkrechten Striche bei 64K geben in diesem Fall sowohl die Größe der Send/Receiver-Puffer des Betriebssystems als auch die Puffergröße der Objektpufferung an. Die maximalen Raten werden in etwa bei der Hälfte der Puffergrößen erreicht.

Start/End - Marken: Vor und hinter einem Objekt können Start- und Stop-Bytes eingefügt werden. Im Falle eines unbekannten Objektes müssen alle Daten bis zur nächsten Endmarkierung ignoriert werden. Das Problem dieser Framing genannten Technik, wie sie z.B. in (SIMPSON, 1994) beschrieben wird, ist die Tatsache, daß die Marke nicht in den Nutzdaten auftreten darf, was beim Empfangen und Senden eine byteweise Filterung der Daten zur Folge hat. Die Framingtechnik ist weit verbreitet, z.B. beim PPP Protokoll (SIMPSON, 1994). Frame-Anfänge werden mit `0x7e` [5], Enden mit `0x7c` markiert. Wenn eins dieser Bytes im Datenstrom auftritt, wird ihm eine `0x7d` vorangestellt

[5] 0x steht für hexadezimal.

und das Zeichen selbst durch ein `XOR` mit dem Wert `0x20` ausmarkiert. Dadurch darf auch das Zeichen `0x7d` selbst nicht in den Daten stehen und muß ebenfalls markiert und maskiert werden. Da die sendende Stelle nicht weiß, ob das Objekt bekannt ist, müssen die Start- u. Stop-Bytes im Datenstrom prinzipiell ausmaskiert werden.

Vorteil dieser Methode: Es kann mitten in einem Datenstrom aufgesetzt werden. Alle Daten bis zum Anfang des nächsten Frames werden ignoriert, und ein korrekter Aufsetzpunkt kann gefunden werden.

Nachteil: Der Datenstrom muß ständig byteweise analysiert werden, nicht nur im Fehlerfalle.

Trennung von Kopf und Nutzdaten: Zunächst wird nur der Objektkopf gesendet und dann auf eine Bestätigung gewartet. Erst dann werden die Nutzdaten übertragen; der Empfänger bestätigt dem Sender, daß er das Objekt kennt, was bedeutet, daß eine Rückkommunikation benötigt wird. In Abb. 2.1 entspricht das Verfahren der untersten Altanative (Fehlerrückmeldung vor dem Datentransfer).

Vorteil dieser Methode: Die Empfangsseite weiß, daß das zuletzt gesendete Objekt nicht erkannt wurde und kann diesen Fehler an das Benutzerprogramm weiterleiten. Es ist nicht nötig, im Header die Datengröße anzugeben, da im Fehlerfalle keine Daten versendet werden.

Nachteile: Man legt sich erstens auf ein bidirektionales Kommunikationsmedium fest, und zweitens entstehen Probleme in nebenläufigen Anwendungen. In solchen Programmen ist es möglich, daß bereits ein nebenläufiger Thread[6] über die Rückleitung kommuniziert. Hinzu kommt der Verlust an Bandbreite, was sich in einer deutlich geringeren Transferrate ausdrückt (Abb. 2.2). Es hat sich gezeigt, daß der Gewinn durch eine automatische Rückkommunikation, also das Weiterleiten des Fehlerzustandes, gegenüber dem Aufwand und den Einschränkungen einer Rückkommunikation nur gering ist, da man nötigenfalls auch auf Benutzerebene eine Rückkommunikation implementieren kann.

Übermitteln der Datengröße: Wie bereits oben erwähnt, kann die Datengröße in den Header mit aufgenommen werden. Ist das Objekt auf der Empfangsseite nicht registriert, weiß man, wieviel Daten übersprungen werden müssen. Es ist weder eine byteweise Interpretation der Daten noch eine Rückkommunikation notwendig (Abb. 2.1 oberste Variante (Unsynchronisiert)).

Möchte man zusätzlich eine automatische Bestätigung (Acknowledge) wie im obigen Fall, so kann man sie nach dem Empfang der Nutzdaten versenden. Die Nutzdaten werden allerdings immer übertragen, auch wenn der Empfänger sie nicht korrekt interpretieren kann (Abb. 2.1 mittlere Variante (Fehlerrückmeldung nach dem Datentransfer)).

Die im Rahmen dieser Arbeit erstellte Software unterstützt alle drei Lösungsansatz, verwendet wird in der Regel aber Ansatz drei, weshalb es notwendig ist, jeder Klasse eine Funktion mitzugeben (`unsigned long int DataSize(void)`), die beschreibt, wie groß die tatsächlichen Nutzdaten sind, siehe auch List. B.1.1.1 und List. B.1.2.1.

[6] Threads sind verkürzt dargestellt nebenläufige Prozesse, die denselben Adressraum teilen.

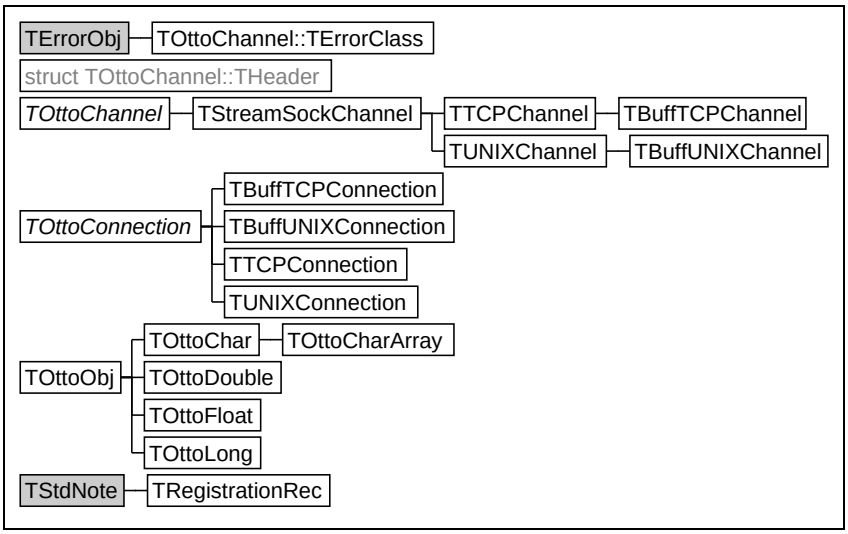
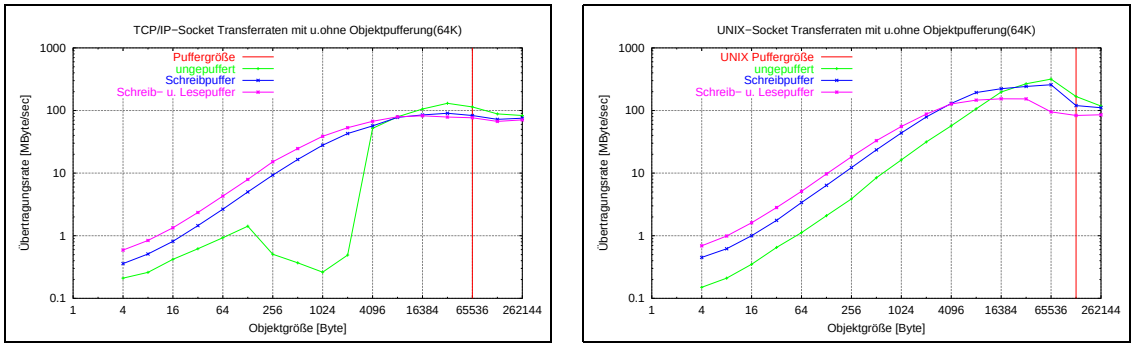


Abbildung 2.3: Ein möglicher Vererbungsbaum zum Versenden von C++-Klassen kann folgendermaßen aussehen: Wird wie in der vorliegenden Implementation Gebrauch von Exceptionhandling gemacht, definiert man sinnvollerweise eine eigene Fehlerklasse, hier `TOttoChannel::TErrorClass`. Der im Text erwähnte Objektkopf (Header) kann durch eine eigene Struktur gekapselt werden, hier `TOttoChannel::THeader`. Die Basisklasse für Objektkanäle wird von `TOttoChannel` gebildet, von der sich diverse Unterklassen verzweigen. Die eigentlichen Datenobjekte leiten sich von der Klasse `TOttoObj` ab; exemplarisch wurden die wichtigsten Grunddatentypen bereits definiert. Zum Registrieren der Objekte, hier in einem ausgeglichenen Binärbaum, wird die Klasse `TRegistrationRec` verwendet, die von einem Standardknoten der Binärbaumbibliothek abgeleitet wird. Bei der Klasse `TOttoConnection` bzw. ihren Nachfahren handelt es sich um eine Kombination aus jeweils zwei Instanzen des Typs `TOttoChannel`: eine zum Senden, die andere zum Empfangen. Dadurch wird bidirektionale Kommunikation mit standardisierter Rückkommunikation auch in nebenläufigen Anwendungen möglich.

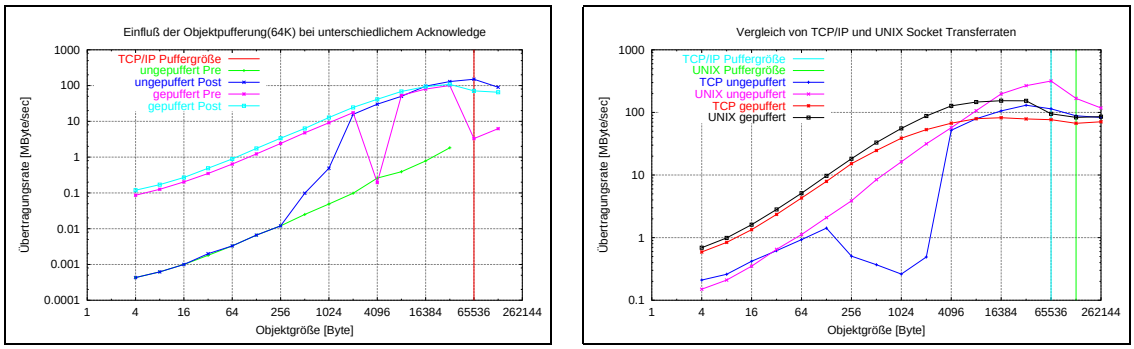
Datenpufferung

Ein Aspekt von Datentransferkanälen ist die Datenpufferung. Sie kann den Datentransfer deutlich beschleunigen und die Nutzung der vorhandenen Bandbreite optimieren, indem sie die maximalen Paketgrößen optimal ausnutzt. Welchen Einfluß Datenpufferung innerhalb von Objekten und über Objekte hinweg auf das Übertragungsverhalten hat, soll in diesem Abschnitt betrachtet werden. Eine Klassenhierarchie wie in Abb. 2.3 ermöglicht es, über beliebige, verbundene Kommunikationskanäle (TCP-,UNIX-Sockets, serielle Schnittstelle, etc.) gepuffert oder ungepuffert Objekte zu übertragen. In der Praxis hat sich gezeigt, daß besonders bei komplexen Objekten und unbedarfter Benutzung die Übertragungsrate deutlich absinkt (Siehe Abb. 2.7). Diesem Effekt kann entgegengewirkt werden, wenn gepufferte Verbindungen verwendet werden. Sowohl der Sender(Schreibpuffer) als auch der Empfänger(Lesepuffer) kann Pufferung verwenden. Bei Kanälen, die eine automatische Bestätigung erzeugen bzw. auf Senderseite erwarten, kann die Pufferung nur über die Daten eines einzelnen Objektes durchgeführt werden. Bei Kanälen ohne Bestätigung ist es möglich, über mehrere Objekte hinweg zu puffern. Die Puffergröße sollte dann deutlich größer sein als die Objektgröße, da sie sich sonst in Bezug auf die Performance negativ bemerkbar machen kann, siehe Abb. 2.4 und Abb. 2.5.



(a) TCP/IP-Sockets (b) UNIX-Domainsockets

Abbildung 2.4: Diese beiden Grafiken veranschaulichen den Einfluß der Pufferungsmethoden auf die Transferrate bei Verbindungen ohne Bestätigung,(a) bei der Verwendung des TCP/IP-Protokolls, (b) bei der Verwendung von UNIX-Domainsockets. In **(a)** zeigt sich bei ungepufferten Übertragungen sehr einfacher Objekte, die lediglich aus einem Datenblock und dessen Größe bestehen, ein Einbruch der Übertragungsrate zwischen 128KB und 2048KB. Bei **(b)** ist dies nicht zu erkennen. Die Verwendung eines Schreibpuffers hebt die Transferraten kleiner Objekte insgesamt, und der erwähnte Einbruch verschwindet völlig. Man beachte, daß es sich hier um eine doppeltlogarithmische Darstellung handelt. Zusätzliche Lesepufferung verbessert die Rate nochmals geringfügig. Bei Objekten größer als 4KB überholt der ungepufferte Transfer bereits den gepufferten, was daran liegt, daß sich das verwendete Objekt mit zwei Schreiboperationen versenden läßt, also nicht besonders komplex ist. Zum Einfluß der Komplexität auf die Übertragungsraten siehe Abb. 2.7.



(a) (b)

Abbildung 2.5: Abbildung **(a)** zeigt den Einfluß der Pufferung auf Übertragungen mit Bestätigung. Es zeigt sich, daß die Transferraten mit Post-Acknowledge für große Objekte höher liegen als die Transferraten mit Pre-Acknowledge und auch stärker von der Pufferung profitieren. Bei ungünstiger Konstellation brechen die Raten der Kommunikation mit Pre-Acknowledge auf die Werte der ungepufferten Übertragung ein. In **(b)** zeigt sich die höhere Performance der UNIX-Domainsockets auf Grund ihres geringeren Protokollaufwands. Sehr deutlich ist auch zu sehen, daß die Übertragungsraten großer Objekte (insbesondere größer als die Objektpuffer) gegenüber ungepuffertem Transfer wieder abnehmen.

2.1.3 Datenobjekte

Wie bereits erwähnt, müssen alle über obige Kanäle zu versendenden Objekte von einem Basisobjekt abgeleitet werden. Aufgabe dieses Basisobjektes ist zum einen die Übermittlung der Byteorder, zum anderen definiert es eine Reihe abstrakter Funktionen, die überdefiniert werden müssen: Eine Funktion zum Serialisieren der Daten, eine für den umgekehrten Zweck und eine Funktion, die die Größe der tatsächlich versendeten Daten beschreibt, denn diese Zahl ist nicht zwangsweise identisch mit der Objektgröße. Wenn z.B. das Objekt Zeigervariablen beinhaltet, schlagen diese in der Objektgröße nur mit der Größe des Zeigers selbst zu Buche und nicht mit der Größe der Daten, auf die der Zeiger zeigt. Beim Versenden von Daten wird aber genau diese Größe benötigt.

Serialisieren Um Daten über eine Schnittstelle bzw. ein Netzwerk zu versenden, müssen Puffer gefüllt werden, die dann vom Betriebssystem geschrieben werden. Die oben genannten Kanäle kapseln diese OS-spezifische Aufgabe ein und stellen stattdessen eine Schreibroutine zur Verfügung. Die Serialisierungsfunktion bekommt eine Referenz auf den Kanal, mit dem die Daten zu versenden sind und benutzt dessen Schreibroutine, um alle gewünschten Daten zu versenden. Zeiger auf Daten lassen sich dereferenzieren, Listen können durchlaufen und die Einzelobjekte versendet werden. Dem Benutzer sind also alle Freiheiten gegeben, er muß sich insbesondere beim Design seiner Klassen nicht an Vorgaben wie z.B. die Verwendung besonderer Datentypen halten. Bei Vererbungshirarchien wird zunächst die Serialisierungsfunktion des Vorgängers aufgerufen (Siehe List. B.1.1.1 und List. B.1.2.1).

Verteilen Das Verteilen der Daten aus einem Kanal auf die objektlokalen Variablen stellt quasi die inverse Operation zum Serialisieren dar. Serialisieren und Verteilen müssen also passend zueinander geschrieben werden. Ein zusätzliches Problem stellt die Byteorder dar. Auf den unterschiedlichen Architekturen werden die Bytes bekanntlich in unterschiedlicher Reihenfolge abgespeichert (little- vs. big-endian). Üblicherweise findet die Konvertierung in das sogenannte Netzwerkformat (big-endian) auf Seiten des Senders statt und muß vom Empfänger gegebenenfalls zurück konvertiert werden. Sind Sender und Empfänger Rechner im little-endian Format, führt das zu einem überflüssigen und bei großen Datenmengen durchaus signifikanten Rechenaufwand. Aus diesem Grunde versendet das Basisobjekt als einziges Datum die Byteorder, in der die Daten zu interpretieren sind. Beim Verteilen muß dann jedoch, abhängig von der lokalen Byteorder, ein Umordnen der Bytes stattfinden. Dieser Prozeß läßt sich nicht automatisieren, da exakte Informationen über die Datentypen benötigt werden, es sei denn, man schreibt einen Codegenerator, der die Serialisierungs- und Verteilungsroutinen weitgehend automatisch generiert.

Datengröße Die Datengröße beschreibt die Anzahl gesendeter Datenbytes, bei Objekten mit konstanter Größe eine konstante Zahl. Bei Objekten, die dynamische Listen, Bäume oder ganz allgemein Graphenstrukturen beinhalten, ist eine Berechnung im Moment des Versendens notwendig. Problematisch wird es bei polymorphen Graphen. In solchen Fällen wird in der Serialisierungsfunktion nicht die Schreibroutine des Kanals aufgerufen, sondern das betreffende Objekt als solches mit Hilfe des Kanals gesendet. Hier wird also ein rekursiver Aufruf der Kanalsendefunktion durchgeführt. Dadurch wird die Headerinformation des betreffenden Objektes in den Datenstrom des umfas-

senden Objektes mit aufgenommen; diese zusätzlichen Daten müssen ebenfalls zur Datengröße des umfassenden Objektes hinzugezählt werden, siehe Abb. 2.6.

Die Performance des Versendens hängt stark von der Anzahl der Schreib-/ Leseoperationen in der Serialisierungs- bzw. Verteilungsfunktion ab. Besteht das Objekt aus vielen einzelnen Variablen, ist der Aufwand beim Versenden/Empfangen größer als bei einem Objekt derselben Größe, das nur aus einem einzigen Puffer besteht, siehe Abb. 2.7.

Besondere Aufmerksamkeit muß dem Versenden polymorpher Objekte gewidmet werden. Sie stellen besondere Ansprüche an die Datenkanäle und an die Mechanismen zum Serialisieren, Verteilen sowie die Ermittlung der tatsächlichen Datengröße.

Polymorphe Objekte

Unter polymorphen Objekten versteht man Klassen, deren exakter Typ erst zur Laufzeit feststeht. Damit steht auch die Datengröße erst zur Laufzeit fest. Ist ein polymorphes Objekt O_1 Bestandteil eines anderen Objektes O_2, das versendet werden soll, so ist auch dessen exakte Datengröße erst zur Laufzeit bekannt, auch wenn O_2 unter Umständen gar nicht polymorph ist. Deshalb wird aus der Senderoutine von O_2 zum Versenden von O_1 nicht dessen Sendefunktion verwendet, sondern die Sendefunktion des Kanals, der dann wiederum einen neuen Header mit den gesamten Größeninformationen in den Datenstrom einbettet. Dazu muß die Versende- und Empfangsroutine des Kanals reentrant sein. Zu beachten ist weiterhin, daß die zusätzlichen Headerdaten ebenfalls zur Objektgröße addiert werden müssen. Die Objektdatengröße des polymorphen Objektes liefert das zu sendende Objekt selbst. Die zu beachtenden Details finden sich in Abb. 2.6 und in List. B.1.3.1, das ein Beispiel für die drei Funktionen `Read`, `Write` und `DataSize` im Falle einer polymorphen Liste angibt.

2.1.4 Zusammenfassung

Es ist möglich, mit den bestehenden Konzepten einer Programmiersprache wie C++ Datenobjekte bis hin zu polymorphen Klassen über ein Netzwerk zu übertragen, ohne daß die Empfangsseite a priori wissen muß, welche Daten zu erwarten sind. Dazu muß der Sprachumfang nicht erweitert werden, und es bleiben dem Anwender alle Freiheiten zur Gestaltung seiner Klassen. Die Bedingungen dafür sind, daß alle versendbaren Objekte von einer gemeinsamen Basisklasse abgeleitet werden müssen (Mehrfachvererbung ist möglich (siehe Abb. B.1)), jedes Objekt über eine Funktion zum Serialisieren, Verteilen und zur Ermittlung der Datengröße verfügt und die Objekte in der sendenden und empfangenden Anwendung bekannt sein müssen. Letzterer Aspekt stellt den einzigen tatsächlichen Nachteil dar.

Neben der für diese Arbeit durchgeführten Implementation wurden die hier beschriebenen Methoden durch Implementationen anderer Personen verifiziert. Sie unterscheiden sich in gewissen Details wie z.B. der Registrierung, der Behandlung polymorpher Objekte und der Vererbungshierarchie, beruhen aber im wesentlichen auf den hier beschriebenen Konzepten.

2.2 Bildung eines Programmverbundes ohne zentralen Server

In einem verteilten Multirobotersystem wird jeder Roboter aus einem eigenen Programm heraus gesteuert. Neben den eigentlichen Robotern werden weitere Anwendungen verwen-

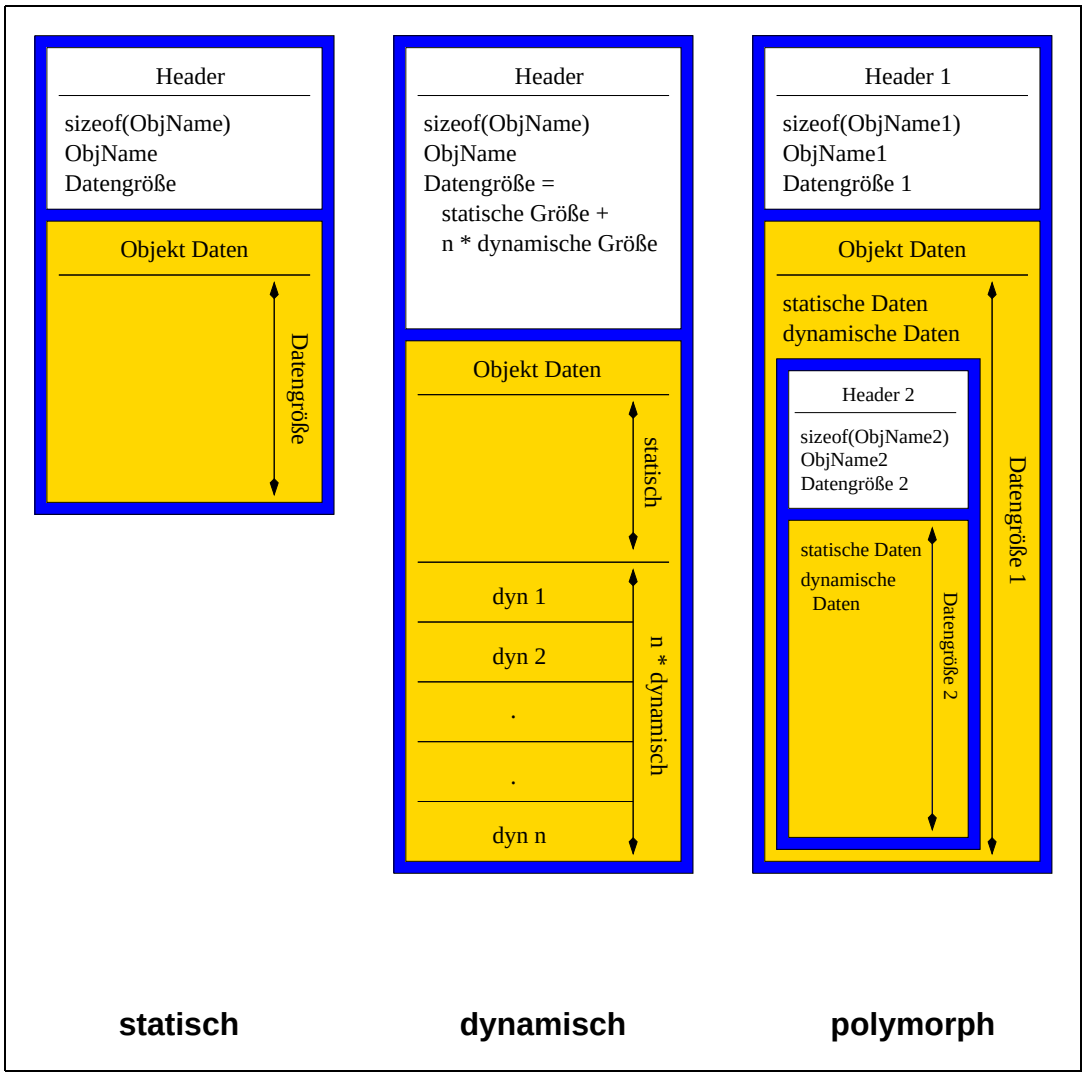


Abbildung 2.6: Beim Versenden von Objekten müssen drei Fälle unterschieden werden.
Der statische Fall: Die Datengröße steht zum Zeitpunkt des Übersetzens fest.
Der dynamische Fall: Die Datengröße ist erst zur Laufzeit bekannt, wenn z.B. eine dynamische Liste Bestandteil des Objektes ist.
Der polymorphe Fall: Hier ist ein polymorphes Objekt, also ein Objekt, das erst zur Laufzeit bekannt ist, in den Daten enthalten. Damit ist nicht nur die Datengröße laufzeitabhängig, sondern auch die gesamte Struktur des Objektes. Eine elegante Lösung für dieses Problem ist schwierig. In dieser Implementation wurde die Funktion des Kanals zum Versenden ganzer Objekte reentrant gestaltet. Dadurch läßt sie sich auch aus der Serialisierungsfunktion eines Objektes heraus aufrufen. In einem solchen Fall werden die Headerdaten erneut in das Objekt geschrieben, um auf der Senderseite das Instantiieren eines passenden Objektes zu ermöglichen. Sendet der Kanal Bestätigungen, müssen diese noch während des Versendens des eigentlichen Objektes bearbeitet werden. Problematisch ist jedoch die Ermittlung der richtigen Objektgröße, da neben den Daten zusätzliche Header mit in den Datenstrom aufgenommen werden. Die Funktionen zum Berechnen der Objektgröße lassen sich aber mit geringem Aufwand entsprechend anpassen. Die nötige Funktionalität zum Bestimmen der zusätzlichen Headergröße läßt sich im Basisobjekt verstecken, wodurch der Anwender lediglich eine andere Funktion zum Berechnen der Größe des untersten Basisobjektes verwenden, sich aber nicht mit den Details des Headers auseinandersetzen muß.

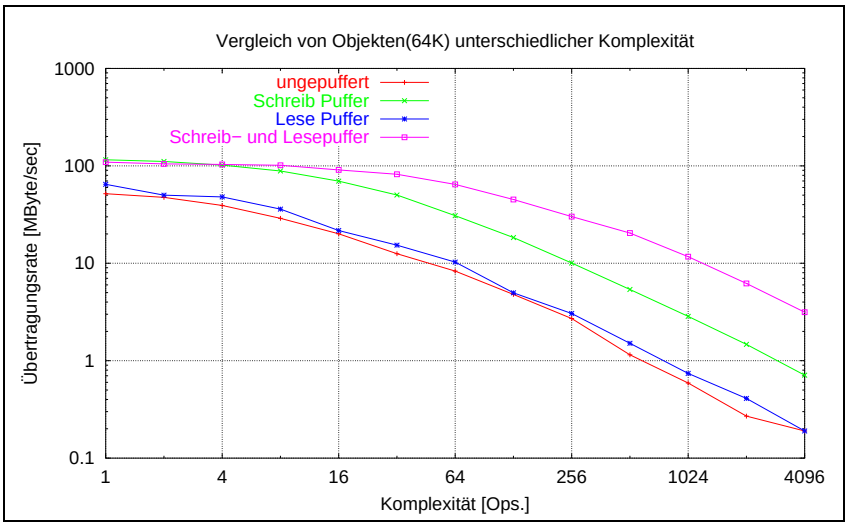


Abbildung 2.7: Um den Einfluß der Komplexität eines Objektes auf die Transferrate messen zu können, wird ein Objekt, bestehend aus einem 4KByte großen Datenblock, in mehreren Schritten versendet. Bei einem Wert von eins wird der Datenblock mit einer Schreiboperation übertragen. Das steht für ein Objekt niedriger Komplexität. Bei einem Wert von 4096 wird jedes Byte des Objektes mit einer eigenen Schreiboperation versendet, wodurch ein sehr komplexes Objekt simuliert wird.
Die Übertragungsrate sinkt bei komplexen Objekten stark ab, was sich aber durch Verwendung von Schreib- und Lesepuffern abmildern läßt. Bemerkenswert ist auch die Tatsache, daß der Lesepuffer allein kaum einen Performancegewinn bringt, zusammen mit einem Schreibpuffer die Transferrate aber im günstigsten Falle versechzehnfacht; dabei entfällt etwa die Hälfte des Zuwachses auf den Lesepuffer. Je einfacher die Objekte werden, um so geringer ist der Einfluß des Schreibpuffers auf die Transferrate; er kann sogar negativen Einfluß haben.

det, die z.B. globale Lockingmechanismen ausführen oder für Bildverarbeitungsaufgaben zuständig sind. Auf die einzelnen Anwendungen wird in Kapitel 3 im Detail eingegangen. Um zwischen allen Komponenten kommunizieren zu können, müssen die beteiligten Programminstanzen einen Verbund bilden. Dazu ist es notwendig zu wissen, welche Programminstanzen überhaupt im Verbund existieren. Jede Programminstanz braucht eine eindeutige Adresse, unter der sie zu erreichen ist, bzw. zu der eine Netzwerkverbindung aufgebaut werden kann. Bei dem Design eines entsprechenden Systems müssen unter anderem folgende Entscheidungen getroffen werden: Soll eine paketorientierte oder eine verbindungsorientierte Kommunikation aufgebaut werden, soll sie flacher oder hierarchischer Struktur sein. Die Vor- und Nachteile der einzelnen Mechanismen sollen im folgenden kurz erläutert werden.

Verbindungsorientierte Strukturen: Ein Vorteil solcher Systeme ergibt sich daraus, daß man sich nicht um die speziellen Probleme paketorientierter Systeme kümmern muß. Außerdem läßt sich leicht erkennen, wenn eine Verbindung geschlossen wird. Dadurch kann der Verbindungsabbau ereignisgesteuert realisiert werden. Wenn also eine Programminstanz p, die sich im Verbund befindet, beendet wird, werden alle Programme, die eine Verbindung zu p halten, aufgeweckt und über den Zusammenbruch der Verbindung informiert.
Nachteile einer solchen Vorgehensweise sind der höhere Protokollaufwand und der gesteigerte Ressourcenbedarf. Bei einer (n zu n) Kommunikation mit verbindungsorientierter Kommunikation werden im günstigsten Falle für jede beteiligte Programminstanz

n-1 Verbindungen[7] benötigt.

Paketorientierte Strukturen: Die Vorteile dieser Klasse von Systemen sind die Schwächen der verbindungsorientierten Systeme und umgekehrt. Solange ein Datum Platz in einem Paket findet und man eventuell sogar den Verlust eines Paketes akzeptieren kann[8], ist ein paketorientiertes System schneller und Ressourcen sparender. Nachteile ergeben sich jedoch genau dann, wenn die Datenmenge über die maximale Paketgröße wächst und/oder Paketverluste nicht toleriert werden können. Dann wird genau der Protokollaufwand benötigt, der in verbindungsorientierten Protokollen bereits existiert. Weiterhin existiert das prinzipielle Problem, daß im Verbund mit anderen Programmen nicht erkannt wird, ob eine Instanz ausgeschieden ist. Soll ein derartiger Mechanismus implementiert werden, weil es sich z.B. um sicherheitsrelevante Software handelt, die den Ausfall gewisser Komponenten auf gar keinen Fall tolerieren kann, so muß zusätzlicher Aufwand betrieben werden.

Hierarchische Netzstrukturen: Die prominentesten Vertreter einer hierarchischen Netzstruktur sind klassische Client - Server - Architekturen mit lediglich einer Hierarchiestufe. Mehrstufige Systeme, wie sie z.B. in PVM (GEIST ET AL., 2000) verwendet werden, sind seltener.

Der große Vorteil dieser Systeme ist ihre relativ einfache Implementierbarkeit und gute Überschaubarkeit. Für die Entwicklung von Client-Server Anwendungen existieren eine große Menge Werkzeuge, die benötigten Mechanismen sind gut dokumentiert und weithin bekannt. Der Client benötigt im günstigsten Falle nur eine einzige Verbindung, nämlich mit seinem Server.

Um eine (n zu n)-Kommunikation aufzubauen, kann entweder eine (n zu 1) - (1 zu n) Kommunikation stattfinden, dabei tauscht die einzelne Anwendung immer nur Daten mit einem Server aus, oder es kann ein Nameserver-Konzept implementiert werden. Alle Programme melden sich beim Nameserver an und erhalten die Adressen der bereits im Verbund befindlichen Programminstanzen.

Nachteil eines hierarchischen Systems ist die Tatsache, daß ein Server leicht zum Flaschenhals wird und zusätzlich ein *single point of failure* ist. Alle Anwendungen, die in einer Hierarchiestufe unter einem Server stehen, fallen aus, wenn der Server ausfällt. Dabei reicht es aus, daß die Verbindung gekappt wird. Wenn die ausgefallenen Programminstanzen nur mit ihrem Server verbunden sind, entstehen weiterhin Probleme für die im Verbund verbliebenen Anwendungen, da sie feststellen müssen, welche Komponenten ausgefallen sind, um entsprechende Gegenmaßnahmen zu ergreifen. Hier schließt sich ein weites Feld von möglichen Fehlerbehandlungsmechanismen an.

Flache Netzstrukturen: In einer flachen Netzstruktur ist man vor dem Ausfall eines Servers sicher, da kein solcher existiert. Ein möglicher Flaschenhals ist ebenfalls nicht gegeben, da jede Anwendung mit jeder anderen auf direktem Wege kommunizieren kann. Ein derartiges, strukturloses Design hat aber auch Nachteile. Die Implementation ist möglicherweise schwerer zu durchschauen, da mit mehreren Anwendungen gleichzeitig kommuniziert werden kann/muß und sich der Verbindungsaufbau deutlich

[7] Dateihandle.
[8] z.B. bei der Online- Übertragung von Signalen.

komplexer gestaltet. Strukturen, die sich auch in einem solchen System etablieren lassen, sind auf Netzebene nicht zu erkennen. Des weiteren muß in jeder Anwendung eine Vielzahl von Informationen gespeichert werden, die sich platzsparend auf einem einzelnen Server hätte unterbringen lassen. Der Ressourcenverbrauch ist daher höher. Im Prinzip ist jede Anwendung gleichzeitig Server. Der Verbindungsaufbau zu den anderen im Verbund befindlichen Anwendungen ist deutlich komplizierter und bedarf einer grundsätzlicheren Auseinandersetzung mit den technischen Möglichkeiten heutiger Netzwerke, da die benötigten Mechanismen nicht durch gängige Werkzeuge bzw. die gängigen API's zur Verfügung gestellt werden.

In dieser Arbeit wird das Konzept einer flachen, verbindungsorientierten Netzstruktur weiter verfolgt, da die Anzahl der verwendeten Komponenten nicht zu hoch ist und die Vorteile, insbesondere im Bereich der Stabilität, die Nachteile bei weitem überwiegen. Als Verbindungskanäle werden die in Kapitel 2.1 beschriebenen Objektkanäle verwendet. Im folgenden soll dargelegt werden, wie ein flacher Programmverbund gebildet werden kann, ohne daß ein Server benötigt wird und ohne daß a priori bekannt ist, wer zu dem Verbund gehört.

2.2.1 Multicasting

Bei der Etablierung der Verbindung ohne Server wird ein weitgehend unbeachteter Mechanismus des Ethernetstandards, das sogenannte Multicasting, genutzt. Es wurde nachträglich in den TCP V4 Standard mit aufgenommen und ersetzt unter dem neuen TCP V6 den Broadkast.
Multicasts sind grundsätzlich paketorientiert. Sie stellen eine Möglichkeit dar, ein Paket an mehrere Empfänger unter einer Adresse zu versenden. Der TCP/IP-Adreßraum ist in mehrere Bereiche eingeteilt, die Adressen von 0.0.0.0 bis 223.255.255.255 stehen für Rechner im Internet, während die Adressen 224.0.0.0 bis 239.255.255.255 sogenannte Multicastadressen darstellen (STEVENS, 1998). Jede Anwendung kann sich auf einer oder mehreren Multicastadressen registrieren und Daten von anderen bzw. an andere Teilnehmer auf dieser Multicastadresse versenden. Zu der Multicastadresse selbst kommt noch die Portnummer hinzu, wodurch eine weitere Aufteilungsmöglichkeit gegeben ist. Ursprünglich gedacht waren Multicasts für Streaminganwendungen mit einem Sender und vielen Empfängern (Internetradio). Dabei bietet das Konzept nicht nur für den Sender, der ja mit einem Paket viele Empfänger erreichen kann, einen Vorteil, sondern auch für den Empfänger[9]. Da Multicasts bereits auf der Ebene des Ethernetprotokolls implementiert sind, werden sie nur dann an den TCP/IP-Layer des Betriebssystems weitergereicht, wenn sich vorher tatsächlich ein Programm auf der entsprechenden Multicastadresse registriert hat. Wenn also ein Multicastpaket einen Rechner erreicht, auf dem es von keiner Anwendung abonniert wurde, verursacht es im Rechner keinerlei Rechenaufwand, da es bereits von der Netzwerkhardware ausgefiltert wurde, im Gegensatz zu einem klassischen Broadcast, das prinzipiell vom Betriebssystem analysiert werden muß, bevor entschieden werden kann, ob das Paket überhaupt von Interesse ist.

[9] um genau zu sein für den nicht Empfänger.

2.2.2 Struktur eines Programms im Verbund

Jede Anwendung im Verbund besitzt eine Reihe von Komponenten, die als Threads realisiert werden und dazu dienen, die Verbindung zu allen im Verbund existierenden Programmen zu halten bzw. zu etablieren, Abb. 2.8. Das eigentliche Programm befindet sich im Ereignis-

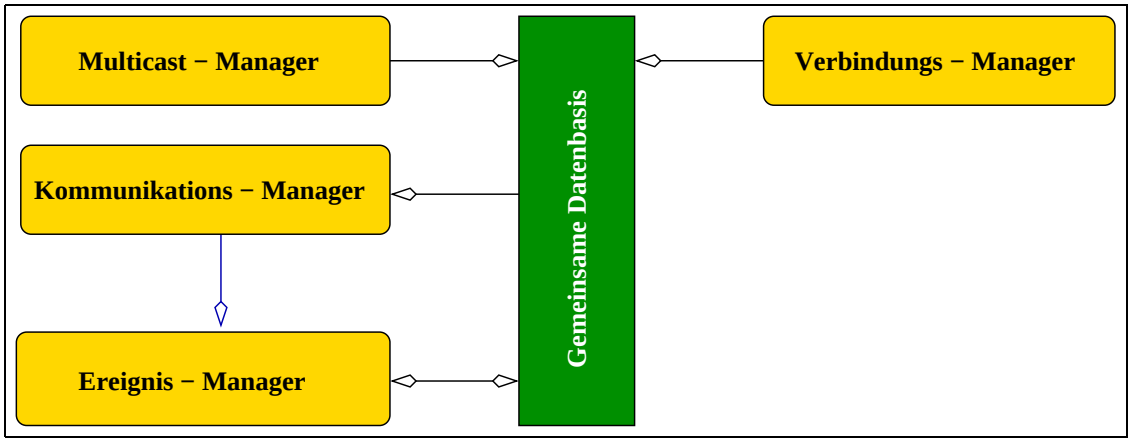


Abbildung 2.8: Eine Programminstanz im Netzverbund verfügt über die folgenden Komponenten: **(a)** Multicast-Manager, **(b)** Verbindungs-Manager, **(c)** Kommunikations-Manager und **(d)** Ereignis-Manager **(a)** Der Multicast-Manager nimmt Multicastpakete neuer Kandidaten auf und realisiert den Verbindungsaufbau. **(b)** Der Verbindungs-Manager initiiert den Verbindungsaufbau und spielt den Counterpart zum Multicast-Manager. **(c)** Der Kommunikations-Manager wartet auf genormte Messages anderer Netzteilnehmer und leitet sie an den Ereignis-Manager weiter. **(d)** Der Ereignis-Manager stellt das eigentliche Programm dar, er reagiert auf eingehende Ereignisse. Die Datenbasis speichert globale Daten aller Komponenten.

Manager. Er analysiert die einlaufenden Botschaften anderer Programme und ruft gegebenenfalls die entsprechenden Funktionen auf. Die Botschaften erhält er vom Kommunikations-Manager über eine interne Fifo.

Der Kommunikations-Manager verwaltet alle eingehenden Datenverbindungen. Dazu bedient er sich einer Liste von Objektkanälen, die in der gemeinsamen Datenbasis abgelegt ist. Liegen Daten an, wird zunächst ein Objekt vom Typ `TOttoObj` gelesen. Dabei handelt es sich um den Basistyp aller über einen Objektkanal versendbaren Objekte (Kapitel 2.1). Der Thread geht allerdings davon aus, daß die eingehenden Daten von einem *Message-Objekt* abgeleitet wurden. Ist das nicht der Fall, werden sie verworfen. Andernfalls wird der Typ dynamisch angepaßt und das Objekt an den Ereignis-Manager weitergeleitet (siehe Kapitel 2.3).

Der eigentliche Verbindungsaufbau zwischen den Anwendungen wird durch den Multicast-Manager und den Verbindungs-Manager realisiert. Dabei ist letzterer in seiner Laufzeit begrenzt. Wird eine Anwendung neu gestartet, versendet der Verbindungs-Manager auf einer festen Adresse ein Multicastpaket mit einem definierten Aufbau. Wesentlicher Bestandteil sind die Adressen, unter denen diese neue Anwendungsinstanz zu erreichen ist. Das Multicast erreicht alle bereits aktiven Anwendungen und die sendende Anwendung selbst. Auf eingehende Multicasts reagiert der Multicast-Manager. Er analysiert das eingehende Paket und wählt die für ihn günstigste Adresse aus[10], um anschließend einen Verbindungsauf-

[10] z.B. eine UNIX-Domainadresse, wenn sich beide Anwendungen auf demselben Rechner befinden, oder eine

bau durchzuführen. Die dabei entstehenden Objektkanäle werden dann in die gemeinsame Datenbasis gestellt und der Kommunikations-Manager wird aktiviert, um von der Änderung Notiz zu nehmen. Dadurch, daß die Anwendung ihr eigenes Multicast empfängt, wird eine Verbindung zur eigenen Anwendung hergestellt.

Im Rahmen des Verbindungsaufbaus lassen sich anwendungsspezifische Daten abfragen. Dazu senden alle bereits im Verbund bestehenden Anwendungen eine Art Featureliste über die neu etablierte Verbindung. Ist die neue Anwendung an einem bestimmten Feature interessiert, fordert sie weitere Informationen. Auf Grund der Verwendung von Objektkanälen und einem objektorientierten Design lassen sich neue Features leicht hinzufügen.
Der Verbindungs-Manager wird nur so lange benötigt, wie noch nicht alle Antworten auf das von ihm generierte Multicast eingegangen sind. Er kann aber nicht wissen, wieviele Antworten kommen werden. Hier läßt sich mit einem Timeout im Bereich einiger Minuten arbeiten. Immer wenn eine Rückmeldung auf das Multicast eingeht, wird das Timeout zurückgesetzt. Nach Ablauf des Timeouts beendet sich der Verbindungs-Manager selbständig.

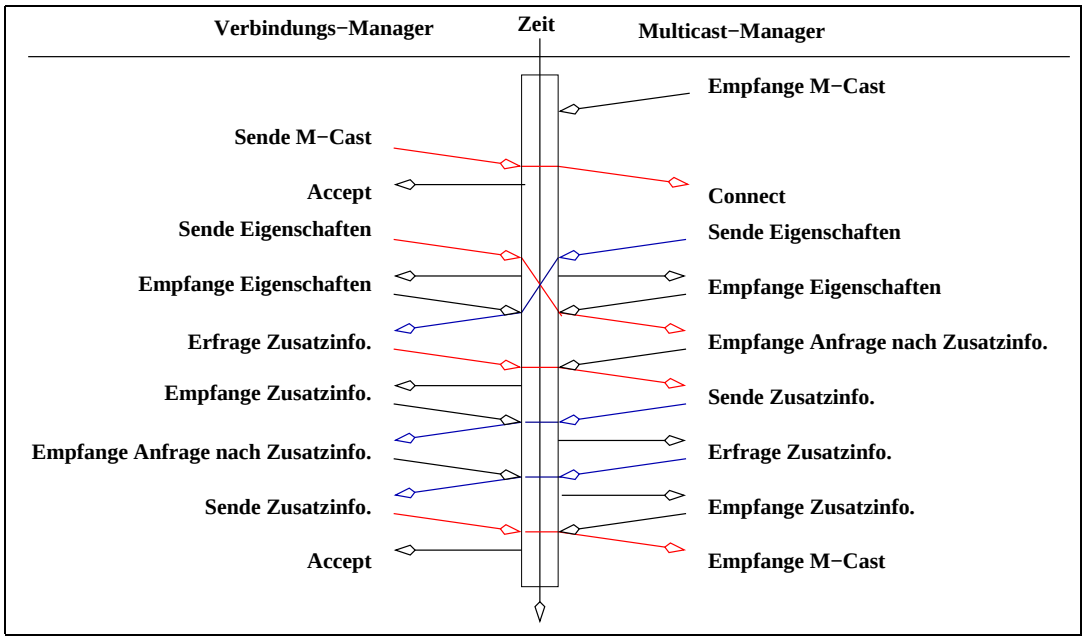


Abbildung 2.9: Zeitverlauf bei einem Verbindungsaufbau. Der Multicast-Manager muß bereits aktiv sein. Er wartet auf eingehende Pakete, auch vom eigenen Verbindungs-Manager, der solange wartet, bis der eigene Multicast-Manager aktiv ist und dann ein Multicast sendet. Danach kommt es zum Verbindungsaufbau (Accept bzw Connect). Es schließt sich der Austausch der Programm-Eigenschaften an, was synchron erfolgen kann. Das Nachfordern von Detailinformationen geschieht hingegen asynchron, da der jeweils andere Kommunikationspartner nicht weiß, welche Zusatzinformationen nachgefordert werden.

2.3 Austausch von Botschaften

In diesem Abschnitt wird auf das integrierte *Messagepassing-Konzept* eingegangen. Sofern keine separaten Verbindungen geöffnet werden, müssen zwei Anwendungen über die

IP-Adresse, falls es sich um unterschiedliche Rechner handelt.

beim Programmstart etablierten Kommunikationskanäle kommunizieren. Der Kommunikationspartner wird dabei über einen eindeutigen Namen angesprochen. Er setzt sich aus dem Rechnernamen, der Prozeßnummer und der Threadnummer des Ereignis-Managers zusammen.

Es gibt eine Vielzahl von existierenden Konzepten und Standards zum Austausch von Botschaften aus dem Bereich der Agententechnologie. Dabei werden oft nicht nur sehr konkrete Vorgaben darüber gemacht, wie ein Austausch von Botschaften auszusehen hat, sondern auch, wie die Semantik der Botschaften zu gestalten ist. Um sich darin zunächst nicht festlegen zu müssen, ist das hier gewählte Konzept weitgehend frei von Interpretationen der transportierten Inhalte. Lediglich das Generieren von Antworten definiert eine gewisse Semantik, die am Ende dieses Abschnitts beschrieben wird.

Botschaften sind Objekte, die von einer gemeinsamen Basisklasse abgeleitet werden. Der Ereignis-Manager muß so programmiert werden, daß er auf ihm bekannte Objekte entsprechend reagiert. Was konkret in einem Objekt transportiert wird, ob es nur Nummern sind, textuelle Beschreibungen oder Lisp-Code, wird zunächst nicht festgelegt; es hängt von dem verwendeten Ereignis-Manager ab. Unbekannte Objekte, die zwar empfangen werden können, deren Semantik und/oder Syntax vom Ereignis-Manager aber nicht verstanden wird, werden nicht sofort verworfen, sondern gesondert bearbeitet. Dadurch kann auch für solche Fälle ein einheitliches Verhalten realisiert werden.

Problematischer wird es, wenn eine Aufgabe aus mehreren Kommunikationschritten besteht, bzw. wenn verhindert werden soll, daß die Botschaften vom Kommunikations-Manager immer in die Eingangsfifo des Ereignis-Managers geleitet werden. Man kann sich z.B. vorstellen, daß eine Anwendung mehrere nebenläufige Ereignis-Manager verwenden möchte, oder sie Ereignis-Manager dynamisch startet und beendet. Zur Lösung dieses Problems wurde ein Konzept entwickelt, das es erlaubt, Daten nicht nur an den Standard-Ereignismanager zu *routen*. Die Daten, die über die Standardkommunikationskanäle geleitet werden, sollen vom Kommunikations-Manager an den richtigen Empfänger gelangen.

Zunächst wird in der gemeinsamen Datenbasis eine Liste von Fifos (Empfängern) verwaltet. Threads können hier eigene Fifos ein- und austragen. Damit der Kommunikations-Manager weiß, in welcher der Fifos er die Daten ablegen muß, werden die Botschaften um entsprechende Felder erweitert. Ein typischer Verlauf einer Kommunikation sieht dann folgendermaßen aus:

Als erstes wird von einer Anwendung eine Botschaft empfangen, die an den Standard--Ereignis-Manager weitergereicht wird. Diese Botschaft beinhaltet in einem ihrer Felder eine Nummer, die einen Verweis auf eine Fifo in der Senderanwendung darstellt. Wenn der Ereignis-Manager auf diese Botschaft reagiert, trägt er in seine eigene Botschaft nicht die Standardfifo[11], sondern die mit der soeben erhaltenen Botschaft gelieferte Nummer ein. Wenn diese Botschaft dann den ursprünglichen Initiator erreicht, wird im Kommunikations-Manager eine andere Fifo als die standardmäßig vorgesehene verwendet, und die Daten können an einen anderen Thread oder eine andere Programmstelle gelangen.

Programmiertechnisch ist dieses Konzept aufwendig, da eine Reihe von Bedingungen beachtet werden müssen. So muß z.B. eine Fifo eingetragen werden, bevor eine Botschaft abgesendet werden darf, und diese Fifo ist auch wieder auszutragen (zu löschen), wenn z.B. der entsprechende Thread beendet wird. Außerdem muß auf der Empfangsseite beim Versenden einer Antwort auf diese Fifo Bezug genommen werden.

[11] z.B. Handle 0.

Die Lösung besteht also darin, daß man eine Botschaft an den Standard-Ereignis-Manager sendet und für die Antwort eine eigene Antwortfifo definiert und den Verweis (Handle) auf diese Fifo mitsendet. Kommt eine Antwort zurück, und wurde vom Antwortenden die Zielfifo in seine Antwort übertragen, leitet der Kommunikations-Manager die Daten automatisch in eine andere als die Standardfifo. Eine Variante dieses Konzepts besteht darin, je-

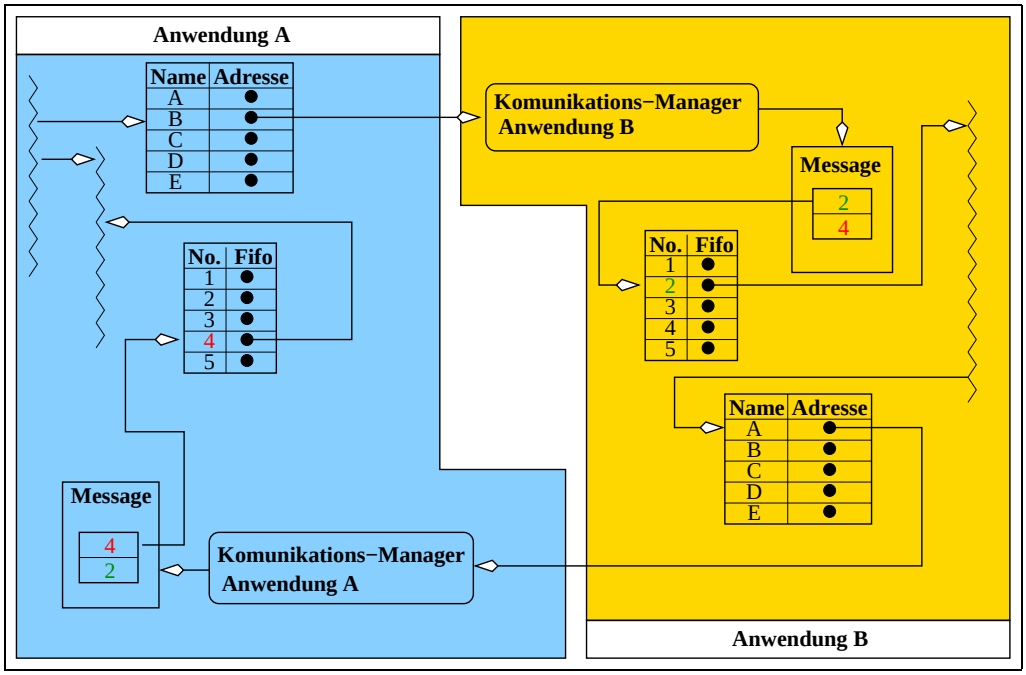


Abbildung 2.10: Austausch einer Botschaft und das Versenden einer entsprechenden Rückantwort: Anwendung A sendet eine Message an Anwendung B, die von deren Kommunikations-Manager gelesen wird. Anschließend startet A einen neuen Thread, der die Antwort bearbeiten soll. In der Message stehen zwei Nummern (2,4), die als Verweise in eine Liste von Fifos zu interpretieren sind. Die erste Zahl (2) gilt für den Empfänger, hier Anwendung B, die zweite für die Antwort. Wenn B eine Antwort zurücksenden will, dreht er z.B. die beiden Zahlen um (4,2). Der Kommunikations-Manager von A interpretiert wiederum die erste der beiden Zahlen und leitet die Daten mit Hilfe einer Tabelle an einen bestimmten Empfänger, z.B. den neuen Thread.

der Botschaft zwei Adressen mitzugeben: die der Standard-Fifo und die einer *Antwort*-Fifo. Weiterhin wird eine Flagge verwaltet, die angibt, ob es sich um eine Antwort handelt oder nicht. Der Kommunikations-Manager wertet diese Flagge aus und wählt die entsprechende Adresse. Dieses Vorgehen hat Vorteile, wenn die empfangene Botschaft modifiziert und auf direktem Wege zurückgesendet werden soll. Es reicht dann aus, die Flagge auf Antwort zu schalten, und die Botschaft wird im Empfänger an die richtige Adresse geleitet.

2.4 Gruppierungsmöglichkeiten

Abschließend soll auf die Möglichkeit der Bildung von Programmgruppierungen hingewiesen werden. Es gibt vielfältige Möglichkeiten, Gruppierungen zu realisieren. Sie spielen sich auf den verschiedenen Ebenen der Software ab. Allen Mechanismen gemein ist jedoch, daß es sich um statische Gruppierungsmechanismen handelt. Damit ist gemeint, daß der Entwickler

einer Anwendung festlegt, zu welcher Gruppe/Gruppen eine Anwendung gehört und diese Festlegung nicht mehr geändert wird. Dynamische Gruppierungskonzepte müssen auf der Anwendungsebene selbst realisiert werden.

- **Gruppierung auf Netzwerkebene:** Durch die Wahl unterschiedlicher Multicastadressen können mehrere getrennte Verbünde geschaffen werden. Den Anwendungen in einem Verbund bleibt die Existenz eines zweiten Verbundes verborgen. Neue Agenten im Verbund A lösen keinerlei Rechenaufwand in Anwendungen des Verbundes B aus.

- **Anwendungsgesellschaften:** Jede Anwendung muß sich einer Gesellschaft (Society) zuordnen. Beim Versenden eines Multicasts wird ein Text (Name der Gesellschaft) mit versendet. Der Empfänger vergleicht den Namen mit seinem eigenen und baut nur dann eine Verbindung auf, wenn Übereinstimmung herrscht.

- **Funktionsgruppen:** Wenn es in einem Programmverbund Anwendungen mit ähnlicher Funktionalität gibt, z.B. Roboteranwendungen, Bildverarbeitungsanwendungen, Anwendungen zur Berechnung komplexer Algorithmen,... ,dann macht es Sinn, diese zu einer Gruppe zusammenzufassen. Das bedeutet nicht, daß Agenten unterschiedlicher Gruppen nicht miteinander kommunizieren können. Sie gehören im Gegensatz zu den beiden obigen Konzepten immer noch zum selben Verbund. Anwendungen sind so in der Lage, ganze Gruppen von Programmen mit ähnlicher Funktionalität anzusprechen, was es erlaubt, die Kommunikation zu optimieren. Um die Gruppierung zu realisieren, versendet jede Anwendung in der initialen Kommunikationsphase eine dynamische Liste von Gruppen, zu denen sie gehören möchte. Der Empfänger baut aus dieser Information eine Datenstruktur gemäß Abb. 2.11 auf.

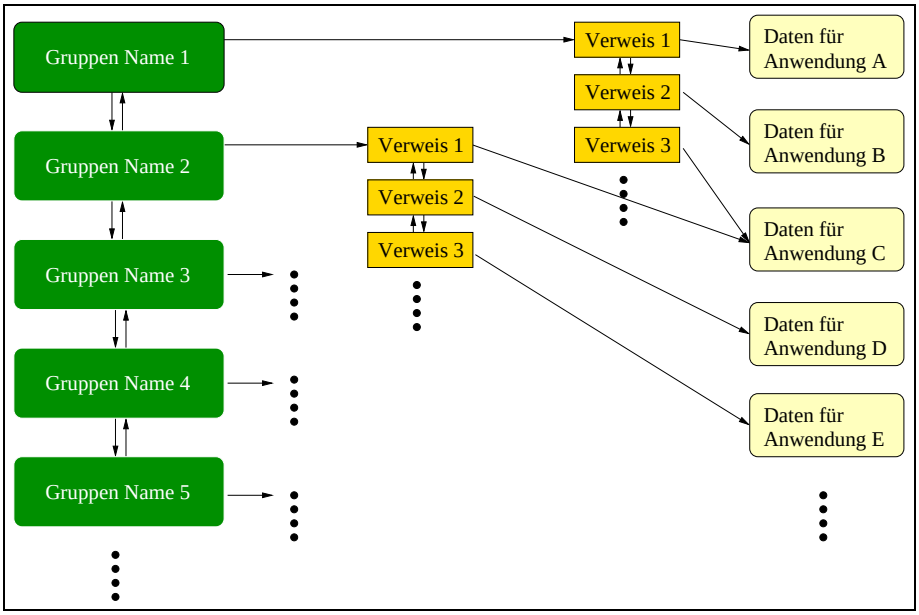


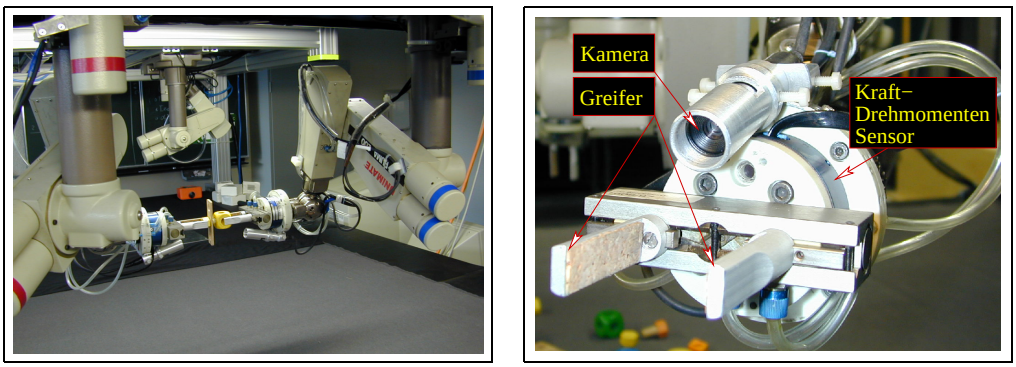
Abbildung 2.11: Jede Anwendungsinstanz verwaltet eine Struktur, die es ihr erlaubt, jedes Programm im Verbund einer Gruppe zuzuordnen. Dabei muß berücksichtigt werden, daß ein Programm mehreren Gruppen angehören kann. Die Daten sind so angeordnet, daß der Gruppenname den primären Suchindex darstellt. Anwendung C gehört in diesem Fall zu ersten und zweiten Gruppe.

3

Einheiten zur Ansteuerung mehrerer Roboterarme

3.1 Robotereinheit(en)

Für das hier behandelte Szenario stehen drei Puma 260 Roboterarme zur Verfügung. Neben den Gelenkencodern ist jeder Roboter mit einer Handkamera und einem Kraftsensor bestückt (siehe Abb. 3.1). Jeder Roboter wird von einer eigenen Roboteranwendung ge-



(a) Puma260 Roboter (b) Endeffektor

Abbildung 3.1: In einer Montagezelle befinden sich mehrere Roboter (Puma260) in hängender Konfiguration. Jeder Roboter ist mit einem Kraftsensor und einer Handkamera ausgerüstet, deren Blickrichtung zwischen die Greifer gerichtet ist.

steuert. Sie ist hierarchisch aufgebaut. Konzeption, Realisation und ausführliche Tests der notwendigen Roboterprogramme sind Gegenstand dieser Arbeit.

Für die in dieser Arbeit konzipierten Roboterprogramme werden drei Ebenen verwendet, die sich als Vererbungshierarchie im Programmcode wiederfinden.

- ❏ Die unterste Ebene erlaubt eine Art Fernsteuerbarkeit der Roboter aus anderen Anwendungen heraus ohne autonomes Handeln. Der Funktionsumfang erstreckt sich von den grundlegenden Bewegungsbefehlen bis hin zu komplexen Bewegungen zum Übergang in Kontaktsituationen und deren Aufrechterhaltung unter Zwangsbedingungen. Diese Möglichkeiten werden z.B. vom OPERA-System (VON COLLANI, 2001) zur Ansteuerung einer beliebigen Anzahl von Robotern genutzt.

- ❏ Auf der nächsten Ebene werden kollisionsfreie Bewegungen eingeführt. Dazu wird eine eigene Bahnplanung vorgenommen und ein globaler Lockingmechanismus verwendet, der Bereiche im kartesischen Raum sperren kann.

- ❏ Auf der obersten Ebene wird eine konkrete Aufgabe, nämlich das Erlernen einer kooperativen Montage von Baufixteilen, in die Roboteranwendung eingebettet. Einmal angestoßen, beginnt eine solche Roboteranwendung zusammen mit allen anderen momentan verfügbaren Robotern, ein vorgegebenes Zielaggregat zu montieren. Dabei verwendet bzw. entwickelt die Anwendung eine eigene Strategie.

3.1.1 Gefügige Bewegungen

Gefügige Bewegungen sind Bewegungen unter Randbedingungen, in diesem Falle die Einhaltung von vorab definierten Kräften. Wie oben erwähnt, werden auf der untersten Ebene bereits Bewegungen mit derartigen Bedingungen realisiert. Im Rahmen dieser Arbeit wurde eine Reihe von Konzepten untersucht und schließlich eine befriedigende Lösung gefunden, die im folgenden dargelegt wird.

Die größte Herausforderung stellt der Übergang von einer freien zu einer gefügigen Bewegung unter Zwangsbedingungen dar. Auf Grund der technischen Gegebenheiten des verwendeten Aufbaues wurden Lösungsansätze untersucht, die das Hookesche Gesetz zu Grunde legen: $F = -Cs$ bzw. im 6-dimensionalen Falle $\vec{F} = -\mathcal{C}\vec{s}$

$$\begin{bmatrix} \vec{f} \\ \vec{m} \end{bmatrix} = -\mathcal{C} \begin{bmatrix} \Delta\vec{p} \\ \Delta\vec{\phi}, \end{bmatrix} \tag{3.1}$$

wobei $\vec{f}$ der dreidimensionale Kraft- und $\vec{m}$ der Drehmomentenvector ist, $\Delta\vec{p}$ die Positions- und $\Delta\vec{\phi}$ die Orientierungsänderung beschreibt. $\mathcal{C}$ ist eine 6×6 Steifigkeitsmatrix.

Bei der Verwendung eines PI-Reglers ergibt sich das in Abb. 3.3(a) gezeigte Verhalten. Dabei wurde der Roboter in einer definierten Höhe über einem Hindernis (hier ein Stahlblock) positioniert und der Regler mit einer Sollkraft von $5N$ aktiviert.

Der Übergang zwischen freiem und Kontaktzustand kann mit einem solchen Ansatz nur dann gelöst werden, wenn die Proportionalitätskonstanten so klein gewählt werden, daß das System unakzeptabel träge wird. Auf einen D-Anteil muß auf Grund des enormen Sensorrauschens verzichtet werden. Verwendet man größere Proportionalitätskonstanten, gerät das System in Schwingungen, wobei es immer wieder den Kontakt mit der Oberfläche verliert. Das Hindernis ist bei den hier gegebenen Zyklusraten (14ms) für diesen Ansatz einfach zu steif.

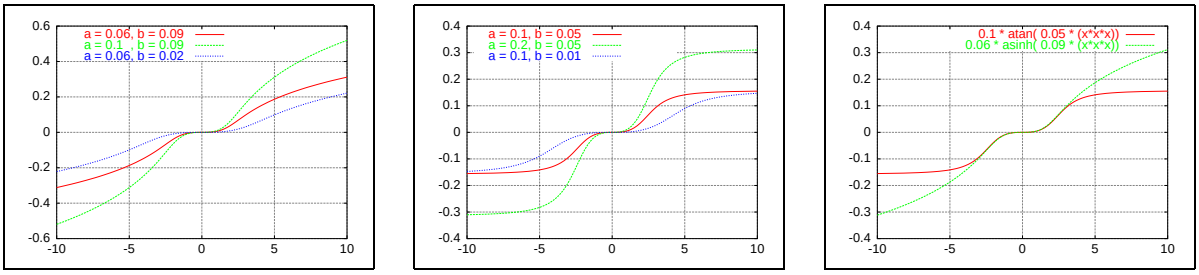
Deutlich verbesserte Eigenschaften ergeben sich bei der Verwendung eines nichtlinearen Reglers. Die Regelabweichung wird zusätzlich mit einer geeigneten Funktion multipliziert und das Ergebnis als Eingabe eines PI-Reglers verwendet. Um das Verhalten des Reglers

bei besonders großen und kleinen Regelabweichungen abzuschwächen, haben sich die folgenden Funktionen als geeignet erwiesen:

$$e_1 = a \cdot \operatorname{asinh}(b \cdot e^3) \tag{3.2}$$

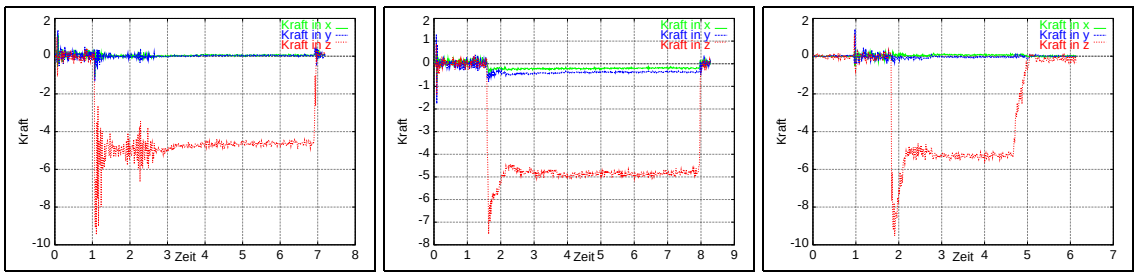
$$e_2 = a \cdot \operatorname{atan}(b \cdot e^3), \tag{3.3}$$

wobei e die ursprüngliche Regelabweichung ist und a, b zwei frei wählbare Parameter zur Veränderung des Funktionsverlaufs sind, siehe Abb. 3.2.



(a) $a \cdot \operatorname{asinh}(b \cdot e^3)$ (b) $a \cdot \operatorname{atan}(b \cdot e^3)$ (c) Vergleich

Abbildung 3.2: Nichtlineare Korrekturfunktionen für die PI-Regler zum *compliance control*. Sehr große und sehr kleine Regelabweichungen e werden abgeschwächt. Gegenüber asinh bietet atan den Vorteil einer definierten Asymptote. Dadurch läßt sich garantieren, daß die Reglerausgabe nicht über alle Maßen steigen kann. In der Nähe des Nullpunktes zeigen beide Funktionen nahezu den gleichen Verlauf. Die Regelabweichung e wird abgeschwächt, um dadurch ein Verhalten ähnlich einer Totzone zu realisieren und starkes Sensorrauschen wirksam zu unterdrücken.



(a) Kräfte bei der Verwendung eines PI-Reglers

(b) Kräfte bei der Verwendung eines nichtlinearen PI-Reglers

(c) Kräfte bei der Verwendung eines Funktionsapproximators nach 18 Lernepochen

Abbildung 3.3: Kraftverlauf beim Herstellen eines Kontaktes zwischen einem Roboter und seiner Umgebung. Als Umgebung wurde in diesem Beispiel ein Stahlblock verwendet, der eine besonders hohe Steifigkeit aufweist. Es zeigt sich, daß die Ergebnisse des PI-Reglers nicht verwendbar sind, da das System zu heftigen Schwingungen neigt. Der nichtlineare PI-Regler zeigt ein besseres Verhalten, ist gegenüber der Variante mit Funktionsapproximator aber träger.

Abb. 3.3(b) zeigt das Verhalten eines entsprechenden Reglers beim Übergang in die

Kontaktsituation. Für $\sinh$ gilt:

$$\begin{aligned} \text{Funktion} &: \operatorname{asinh}(x) = ln\left(x + \sqrt{x^2+1}\right) \\ \text{Definitionsbereich} &: -\infty < x < \infty \\ \text{Wertebereich} &: -\infty < \operatorname{asinh}(x) < \infty \end{aligned}$$

asinh hat keine Asymptote und damit auch $a \cdot \operatorname{asinh}(bx^3)$ nicht. Nach

$$e_1' = \frac{3abx^2}{1+(bx^3)^2} \qquad e_1'' = \frac{6abx\left[1+\left(bx^3\right)^2\right] - 3abx^2 \cdot 6b^2x^5}{\left[1+(bx^3)^2\right]^2} \tag{3.4}$$

folgen Wendepunkte bei:

$$\pm\frac{1}{2b}\sqrt{2} \tag{3.5}$$

Für atan gilt:

$$\begin{aligned} \text{Funktion} &: \operatorname{atan}(x) \\ \text{Definitionsbereich} &: -\infty < x < \infty \\ \text{Wertebereich} &: -\frac{\pi}{2} < \operatorname{atan}(x) < \frac{\pi}{2} \end{aligned}$$

atan hat eine Asymptote und damit auch $a \cdot \operatorname{atan}(bx^3)$. Diese Funktion ist deshalb für die Begrenzung der Stellgröße besser geeignet als asinh. Nach

$$e_2' = \frac{3abx^2}{(1+b^2x^6)^{\frac{1}{2}}} \qquad e_2'' = \frac{6abx(1+b^2x^6)^{\frac{1}{2}} - 3abx^2 \cdot \frac{1}{2}(1+b^2x^6)^{-\frac{1}{2}} \cdot 6b^2x^5}{1+b^2x^6} \tag{3.6}$$

folgen Wendepunkte bei:

$$\pm\frac{1}{3b}\sqrt{6} \tag{3.7}$$

Das Regelverhalten als Ganzes läßt sich auch durch einen Funktionsapproximator (Anhang C) lernen. Daraus ergeben sich die in Abb. 3.3(c) gezeigten Kurven. Weitere Details lassen sich Kapitel 7.2 entnehmen. Die Verläufe der Kraft in Abb. 3.3(a) und Abb. 3.3(b) spiegeln wieder, daß der Funktionsapproximator schneller die Sollkraft einregelt (0.5s gegenüber 1s), obwohl die Geschwindigkeit, mit der er auf das Hindernis trifft, höher ist. Die Tiefe des ersten Überschwingens wird im wesentlichen durch die Geschwindigkeit bestimmt, mit der das Hindernis getroffen wird. Da die Totzeit, also die Zeit, bis sich Reglerausgaben an der Strecke bemerkbar machen, ca. 5 Regelzyklen, also 70ms beträgt, baut sich bis zu diesem Zeitpunkt bereits eine signifikante Kraft auf.

3.1.2 Kollisionsvermeidung

Kollisionsfreie Bewegungen, wie sie auf der mittleren Ebene definiert sind, arbeiten mit einem globalen Locking-Mechanismus (Kapitel 3.2.1). Er erlaubt es, Bereiche im kartesischen

Raum exklusiv zu sperren. Dazu werden zwei Typen von Regionen verwendet und an die globale Überwachungsinstanz übermittelt: Kugeln und eine Liste von Strecken mit dazugehörigen Sicherheitsabständen. Die Parameter der Regionen sind so zu wählen, daß sie das gesamte Volumen des jeweiligen Robotersegmentes einschließen (konvexe Hülle). Diese Methode ist zwar relativ verschwenderisch, da sie die tatsächliche Form eines Segmentes großzügig einhüllt, sie erlaubt aber eine einfache und schnelle Berechnung der Schnittpunkte zwischen verschiedenen Segmenten. Die Lage der Segmente läßt sich durch eine Vorwärtskinematik berechnen, die hier exemplarisch für den verwendeten Puma 260 gezeigt werden soll:

Erweiterte DH - Parameter

Abmessungen eines Puma 260Abmessungen eines Puma 260. Alle Angaben in inchPuma-Maasse Die Denavit Hartenberg - Parameter, die für eine Vorwärtskinematik sinnvollerweise benutzt werden, dienen im Normalfall zur Berechnung der Position des Endeffektors. In unserem Falle werden aber auch die Positionen der Segmentanfangs- und Endpunkte benötigt. Dazu lassen sich die DH-Parameter wie folgt erweitern:

① Translation entlang der z-Achse um d [mm].

② Rotation um die (z)-Achse um θ [DEG].

③ Translation entlang einer der Koordinatenachsen (x, y, z) um (a, b, c) [mm] respektive.

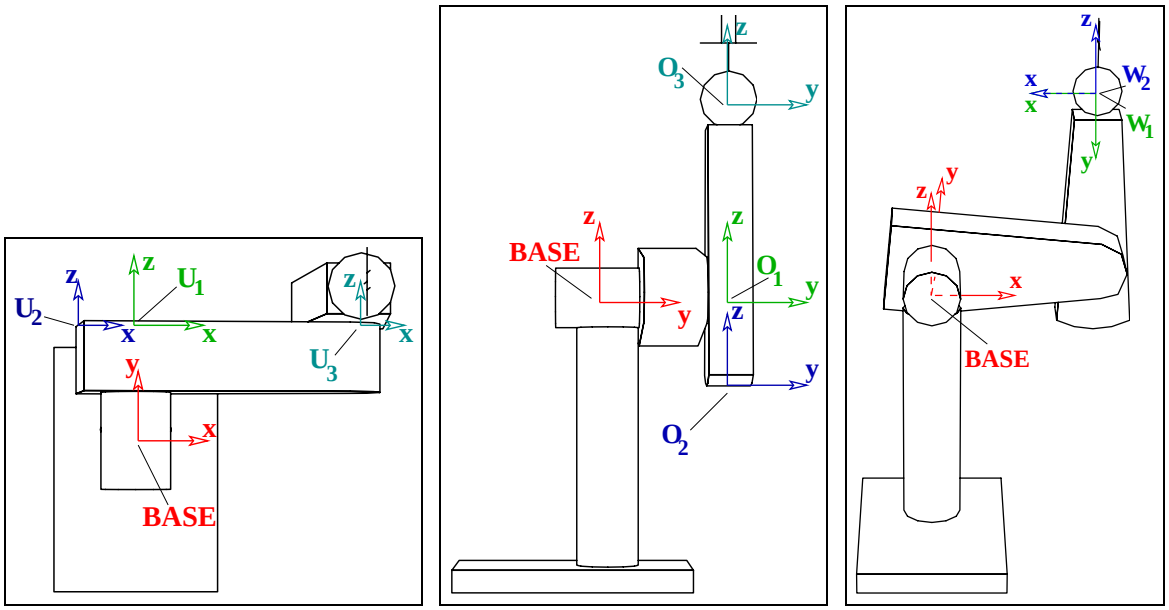
④ Rotation um die (x)-Achse um α [DEG].

Für einen Puma 260, wie er in diesem Setup verwendet wird, ergeben sich die benötigten Parameter wie folgt:

		mm a_x	mm b_y	mm c_z	mm d	deg α	deg θ
Base $\rightarrow U_1$	A_1	0.00	98.74	0.0	0.0	-90.0	j_1
$U_1 \rightarrow U_2$	A_2	-144.78	0.00	0.0	0.0	0.0	j_2
$U_1 \rightarrow U_3$	A_3	203.20	0.00	0.0	0.0	0.0	j_2
$U_3 \rightarrow O_1$	A_4	0.00	0.00	0.0	27.5	90.0	j_3
$O_1 \rightarrow O_2$	A_5	0.00	0.00	0.0	-100.0	0.0	0.0
$O_1 \rightarrow O_3$	A_6	0.00	0.00	0.0	203.2	0.0	0.0
$O_3 \rightarrow W_1$	A_7	0.00	0.00	0.0	0.0	90.0	j_4
$W_1 \rightarrow W_2$	A_8	0.00	0.00	0.0	0.0	-90.0	j_5
$W_2 \rightarrow$ TCP	A_9	0.00	0.00	0.0	208.0	0.0	j_6

Die sich daraus ergebenden Matrizen A_1 bis A_9 zum Berechnen der Position des jeweiligen Koordinatensystems werden in folgender Form gebildet:

$$A = \begin{bmatrix} \cos(\theta) & -\sin(\theta)\cdot\cos(\alpha) & \sin(\theta)\cdot\sin(\alpha) & \cos(\theta)\cdot a_x - \sin(\theta)\cdot b_y \\ \sin(\theta) & \cos(\theta)\cdot\cos(\alpha) & -\cos(\theta)\cdot\sin(\alpha) & \sin(\theta)\cdot a_x + \cos(\theta)\cdot b_y \\ 0 & \sin(\alpha) & \cos(\alpha) & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$



(a) Ansicht von oben, mit Koordinatensystemen des Unterarms.

(b) Frontalansicht mit Koordinatensystemen des Oberarms.

(c) Ansicht von der Seite mit Koordinatensystemen des Handgelenks.

Abbildung 3.4: Koordinatensysteme eines Puma 260 zur Berechnung der Segmentpositionen mittels Vorwärtskinematik.

Berechnung der Segmentanfangs- und Endpunkte

Die Segmentanfänge und -enden ergeben sich dann durch folgende Transformationen:

$$A_1 = \begin{bmatrix} \cos(j_1) & 0 & -\sin(j_1) & -98.74 \cdot \sin(j_1) \\ \sin(j_1) & 0 & \cos(j_1) & 98.74 \cdot \cos(j_1) \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_2 = \begin{bmatrix} \cos(j_2) & -\sin(j_2) & 0 & -144.0 \cdot \cos(j_2) \\ \sin(j_2) & \cos(j_2) & 0 & -144.0 \cdot \sin(j_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} \cos(j_2) & -\sin(j_2) & 0 & 203.2 \cdot \cos(j_2) + 19.0 \cdot \sin(j_2) \\ \sin(j_2) & \cos(j_2) & 0 & 203.2 \cdot \sin(j_2) \cdot \cos(j_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_4 = \begin{bmatrix} \cos(j_3) & 0 & \sin(j_3) & 0 \\ \sin(j_3) & 0 & -\cos(j_3) & 0 \\ 0 & 1 & 0 & 27.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -100.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_6 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 203.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_7 = \begin{bmatrix} \cos(j_4) & 0 & \sin(j_4) & 0 \\ \sin(j_4) & 0 & -\cos(j_4) & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_8 = \begin{bmatrix} \cos(j_5) & 0 & \sin(j_5) & 0.0 \\ -\sin(j_5) & 0 & \cos(j_5) & 0.0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_9 = \begin{bmatrix} \cos(j_6) & -\sin(j_6) & 0 & 0 \\ \sin(j_6) & \cos(j_6) & 0 & 0 \\ 0 & 0 & 1 & 208.0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$j_1 \ldots j_6$ stehen für den jeweiligen Gelenkwert.

3.2 Weitere Programme im Verbund

3.2.1 Verwaltung von Raumbereichen

Zur Kollisionsvermeidung lassen sich kartesische Raumbereiche sperren. Ihre Verwaltung wird von einer Anwendung im Verbund realisiert. Die Existenz einer solchen Programminstanz ist für die kollisionsfreie Bewegung unabdingbar. Existiert keine solche Programminstanz, lassen sich die Roboteranwendungen nicht starten. Falls sich das Verwaltungsprogramm während des Betriebs beenden sollte, stoppen die Roboteranwendungen selbständig.
Folgende Funktionalität wird von einer Anwendung zur Verwaltung kartesischer Raumbereiche zur Verfügung gestellt:

TRYLOCKORB: Versucht, eine Kugel zu sperren. Dazu wird der Schnittbereich mit allen existierenden Regionen berechnet. Existieren keine Überschneidungen, wird die Kugel in die Menge der Regionen aufgenommen und eine positive Antwort zurückgesendet. Es wird zusätzlich ein Verweis (Handle) auf die Region zurückgeliefert. Der Verweis ist beim Freigeben der Region anzugeben. Gibt es eine Überschneidung, wird eine ablehnende Antwort gesendet und die Region verworfen.

TRYLOCK: Wie *TRYLOCKORB*; die Region besteht aber nicht aus einer Kugel, sondern aus einer Liste von Strecken mit einem dazugehörigen Sicherheitsabstand. Sie beschreiben die Lage der Robotersegmente im Raum. Der Sicherheitsabstand ist dann so zu wählen, daß es zu keiner Kollision kommt, da das eigentliche Segment ja eine räumliche Ausdehnung besitzt.

LOCKORB: Sperrt eine Kugel r im kartesischen Raum. Dazu werden die Schnittbereiche mit allen existierenden Regionen $r_1 \ldots r_n$ berechnet. Existiert kein Schnittbereich, wird die Region in die Liste der gesperrten Bereiche r_{n+1} aufgenommen und eine positive Rückantwort gegeben. Sobald die erste Kollision (z.B. mit der Region r_k) festgestellt wird, bricht der Vergleichsvorgang ab und die Region wird in eine Liste von wartenden Regionen aufgenommen. De facto wird die ursprüngliche Anfrage mit der Region und dem Anfragetyp, hier also *LOCKORB* abgespeichert. Hinzu kommt noch ein Verweis (*Handle*) auf r_k. Diese Information wird später benötigt, wenn r_k freigegeben wird.

LOCK: Wie *LOCKORB*, die Region besteht aber nicht aus einer Kugel, sondern aus einer Liste von Strecken mit einem dazugehörigen Sicherheitsabstand.

TRYADDLOCKORB: Wie *TRYLOCKORB*, nur daß die Region mit einer bestimmten Region kollidieren muß, mit allen anderen aber nicht kollidieren darf, um in die Liste der akzeptierten Regionen aufgenommen zu werden. Die Region, mit der eine Kollision erzwungen wird, muß von derselben Anwendung stammen wie die neue Region.

TRYADDLOCK: Wie *TRYADDLOCKORB*, nur handelt es sich bei der Region nicht um eine Kugel, sondern um eine Liste von Strecken und deren Sicherheitsabständen. Die erzwungene Kollision ist nur dann gültig, wenn es korrespondierende Strecken gibt und diese jeweils miteinander kollidieren. Korrespondieren heißt, daß eine Strecke aus der anderen durch Translation und Rotation hervorgehen kann.

ADDLOCKORB: Wie *TRYADDLOCKORB*, nur daß bei einer Kollision mit einer anderen als der vorgesehenen keine Antwort versendet wird, sondern die Region in die Liste der Wartenden aufgenommen wird.

ADDLOCK: Wie *ADDLOCKORB*, nur daß es sich nicht um eine Kugel, sondern um eine Liste von Strecken und den zugehörigen Sicherheitsabständen handelt.

UNLOCK: Freigeben einer Region r, die über einen *Handle* anzugeben ist. Anschließend wird getestet, ob eine andere Region r_w auf die Freigabe von r wartet. Wenn ja, wird r_w aus der Liste der Wartenden entfernt und über die *Loopbackverbindung* der Anwendung eine künstliche Anfrage simuliert.

3.2.2 Feinpositionierungsmodul

Das Feinpositionierungsmodul stellt eine eigene Anwendung im Verbund dar. Es versteht ausschließlich initiale Anfragen auf Start eines Feinpositionierungszyklusses mit Angabe des Roboters und des erwarteten Bauteiltyps. Daraufhin wird ein eigener Prozeß (Thread) abgesetzt, der einen separaten Kommunikationskanal mit dem Anfragesteller etabliert. Der Robotername dient dazu, die passende Handkamera anzusprechen. Die Angabe des richtigen Objekttyps ist unabdingbar, um die korrekten, a priori gelernten Daten zu laden. Es wird ein ansichtenbasierter Ansatz verwendet (Kapitel 7.3.3), der keine Erkennungsleistungen bietet; liegt ein anderes als das übermittelte Objekt unter dem Greifer, kann das verwendete Verfahren dies nicht erkennen und wird fehlerhafte Ergebnisse liefern.

Durch das Abspalten eines separaten Threads wird es möglich, mehrere Roboter gleichzeitig zu bedienen. Nachdem die Verbindung aufgebaut ist, wird ein Bild vom passenden Bilder-Server[1] angefordert und der nötige Roboterkorrekturschritt $(\Delta x, \Delta y, \Delta\rho)$ berechnet. Die Daten werden an den Roboter übermittelt, und es wird darauf gewartet, daß dieser die Korrektur durchführt. Ist das geschehen, wird erneut ein Bild der Handkamera angefordert usw., bis die Korrekturen einen vorgegebenen Wert unterschreiten. Ist die Korrektur nur noch minimal, wird das in der letzten Botschaft an den Roboter vermerkt und die Verbindung geschlossen. Der Prozeß beendet sich daraufhin selbständig.

3.2.3 Objekterkennungsmodul

Das Objekterkennungsmodul arbeitet im Prinzip ähnlich wie das Feinpositionierungsmodul. Da eine Objekterkennung aber nicht zyklisch erfolgen muß, wird kein gesonderter Prozeß abgespalten, vielmehr können die im Kapitel 2.3 beschriebenen Mechanismen zur Kommunikation verwendet werden. Der Algorithmus berechnet nach Anforderung eines Bildes für die passende Handkamera ein Maß, wie gut das angeforderte Bild mit dem ihm bekannten Objekt übereinstimmt. Darüber hinaus kann der anfragende Roboter eine Hypothese abgeben, was seiner Meinung nach unter dem Geifer liegen müßte. Anhand eines Schwellwertes bestätigt der Objekterkenner diese Hypothese oder verwirft sie.

Der Objekterkenner wurde implementiert, um die genaue Lage von Schrauben vor dem Feinpositionieren/Greifen zu ermitteln. Schrauben verändern stark ihre Ansicht, wenn sie umkippen. Daraus resultiert ein anderes Greifverhalten und auch eine andere Feinpositionierung. Details zu dem verwendeten Verfahren werden in Kapitel 7.3.2 beschrieben.

[1] Diese Anwendung gehört nicht zum Verbund, da sie für andere Zwecke entwickelt wurde.

3.2.4 Programmstatusmeldungen

Bei der Verwendung vieler Programme verliert man leicht den Überblick. Außerdem ist es unpraktisch, die Ausgaben aller Programme in einer Vielzahl von Fenstern zu beobachten. Abhilfe schafft ein Programm, das alle aktuell im Verbund befindlichen Anwendungen auflistet und ihre Ausgaben protokolliert (Abb. 3.5). Programminstanzen desselben Typs werden zur besseren Übersichtlichkeit zusammen gruppiert. Da diese Protokollinstanz ein ganz normales Programm im Verbund ist, verwendet sie die üblichen Kommunikationsmechanismen. Neue Programme erscheinen automatisch in den Listen, Programme, die sich beenden oder abstürzen, verschwinden automatisch. Dazu bedarf es nahezu keiner Neuentwicklungen, da bereits jedes Programm im Verbund die Fähigkeit besitzt, Verbindungen automatisch auf- und abzubauen. Eine weitere Fähigkeit, die bereits in der Basisklasse implementiert wurde, ist das Antworten auf sogenannte Ping-Messages. Damit läßt sich feststellen, ob der Ereignis-Manager einer Anwendung noch empfangsbereit ist.

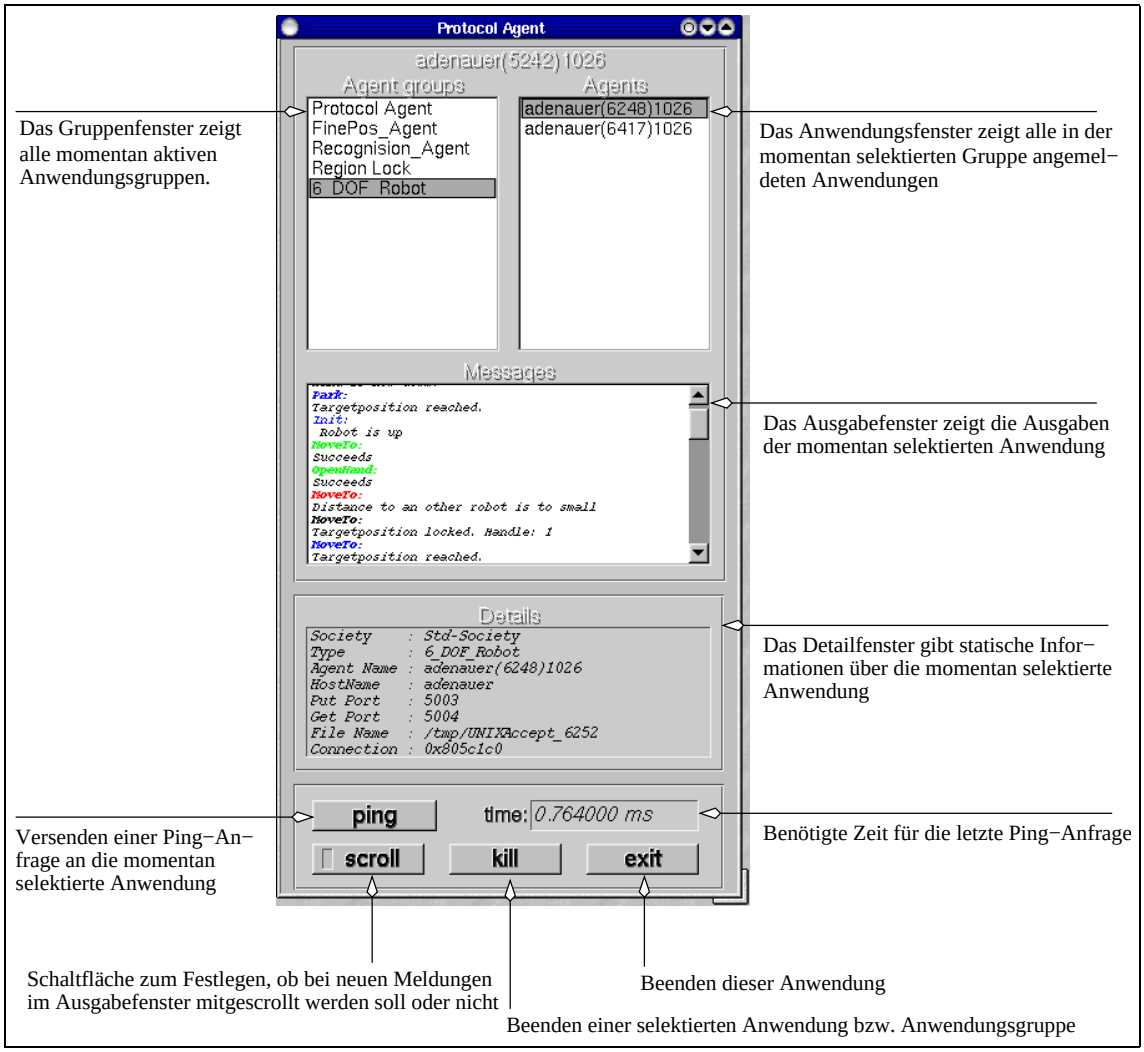


Abbildung 3.5: Ausgabefenster für Programmstatusmeldungen. Es kann gleichzeitig benutzt werden, um Anwendungen zu beenden oder zu überprüfen, ob eine Anwendung noch empfangsbereit ist.

Teil II

Erlernen einer Montagestrategie

4

Grundlegende Verfahren des Verstärkungslernens

Verstärkungslernen (Reinforcement Learning) (SUTTON AND BARTO, 1999) ist in den letzten fünf bis zehn Jahren populär geworden. Die vielen Verfahren, die unter dieser Bezeichnung firmieren, haben gemein, daß sie ein System, häufig einen Agenten, so programmieren oder sein Verhalten durch Belohnung und Bestrafung so modifizieren, daß er eine bestimmte Aufgabe erfüllt, ohne ihm genau zu erklären, wie er das schafft. Dazu ist jedoch ein enormer Aufwand notwendig, denn alle derartigen Verfahren arbeiten nach dem Konzept des Versuchs und Irrtums. Zwei Gruppen von Verfahren sind zu unterscheiden:

Genetische Algorithmen: Bei diesen Verfahren wird versucht, einen Raum von Strategien abzusuchen, in dem man, ausgehend von einer oder mehreren Strategien, Veränderungen oder Kreuzungen an/zwischen ihnen vornimmt und untersucht, ob die neue/neuen Strategien besser oder schlechter sind als die ursprünglichen (HU BER, 1996). Ein Verfahren aus diesem Bereich wird in Abschnitt 4.6 näher beschrieben.

Zustandsbasierende Entscheidungsverfahren: Verfahren, die in diesem Bereich angesiedelt sind, verwenden statistische Methoden, um für eine Menge von Zuständen die Aktionen zu ermitteln, die den gewünschten Effekt erzielen.

Darüber hinaus können auch Neuronale Netze wie z.B. in (WENGEREK, 1995) durch Reinforcementsignale trainiert werden. Hier sollen im weiteren Verfahren der zweiten Klasse behandelt werden: Aufgabe eines lernenden Systems, also z.B. eines Agenten, ist es, das Verhalten π (behavior,policy) so einzurichten, daß das System in jeder Situation s eine Aktion a wählt, die in den Zustand s' und auf lange Sicht zu dem gewünschten Ziel führt. Bei jeder Aktion, die das System durchführt, verändert es den Zustand der Welt, bringt sich also in eine andere Situation. Dafür bekommt es ein Verstärkungssignal r (reward). Dieses r wird durch eine Funktion δ berechnet, die vom Zustand und der Aktion abhängt. Sie muß nicht zwangsweise deterministisch sein und ist dem Lernverfahren als solches nicht unbedingt bekannt. Alle bekannten Verfahren des Verstärkungslernens haben Probleme, wenn δ sich zeitlich schnell ändert. Was anfangs noch richtig war und belohnt wurde, ist später falsch

und wird bestraft. Solche Effekte treten z.B. auf, wenn zwei Mannschaften gegeneinander spielen und eine davon nach einer gewissen Zeit abrupt ihre Strategie ändert. Ein anderes, prominentes Beispiel ist die Fahrstuhlsteuerung eines Hochhauses. Hier müssen sich die Strategien mehrmals am Tag ändern (je nach Tageszeit), um den Durchsatz zu optimieren.

Umgebungen, in denen sich der Zustand ohne Ausführung einer Aktion ändert, werden nichtdeterministisch genannt. Ebenfalls als nichtdeterministisch bezeichnet man Systeme, bei denen eine Aktion nicht immer in denselben Zielzustand führt.

Weiterhin gehen die Verfahren in diesem Abschnitt davon aus, daß der gesamte Zustandsraum $\mathcal{S}$ abgezählt und im Speicher eines Rechners gehalten werden kann.

4.1 Markovsche Entscheidungsprozesse

Um Verfahren des Verstärkungslernens besser formalisieren zu können, beschreibt man sie als Markovsche Entscheidungsprozesse (MEP). Die folgenden Größen lassen sich dabei formalisieren:

- $s \in \mathcal{S}$, eine diskrete Menge von Weltzuständen.
- $a \in \mathcal{A}$, eine diskrete Menge von Aktionen, die von einem System – z.B. einem Agenten – ausgeführt werden können. Die Ausführung einer Aktion a in einem gegebenen Zustand s überführt das System in einen Zustand s'. Möglicherweise ist $s = s'$.
- $v \in \mathcal{V}$, eine diskrete Menge von sogenannten Merkmalen (Features).
  Wenn der Zustandsraum $\mathcal{S}$ zu groß ist, um auch nur annähernd abgedeckt zu werden, muß man ihn einschränken. Es gibt eine Reihe von Ansätzen dazu, u.a. den Übergang zur Merkmals-Menge $\mathcal{V}$. Dabei wird die Einschränkung der Menge von der Aktion abhängig gemacht. Der eingeschränkte Zustandsraum $\mathcal{S}$ wird dann Merkmalsraum (Feature-Space) $\mathcal{V}$ genannt. $\mathcal{V}$ läßt sich als Eingabemenge für die zu lernende Wertefunktion benutzen. Wenn man den Aktionsraum $\mathcal{A}$ verwenden will, um für jede Aktion $a_i \in \mathcal{A}$ einen eigenen Merkmalsraum zu benutzen, so verwendet man eine Abbildung $e : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{U}$. Sie wird auch aktionsabhängige Merkmalsfunktion genannt. $\mathcal{U}$ ist eine diskrete Menge von Merkmalen, die die zu erwartenden kurzfristigen Effekte der gewählten Aktion widerspiegeln. $f(s)$ ist der Feature-Vektor. Er sieht z.B. folgendermaßen aus:

  $$f(s) = \{e(s, a_0), e(s, a_1), \ldots, e(s, a_n)\} = \mathcal{V} \tag{4.1}$$

  Jedes Element im Merkmalsraum beschreibt, ob die Aktion a_i in Situation s eventuell erfolgreich ist oder nicht. Bei einer Montageoperation a_m läßt sich mittels Objektmodell bestimmen, ob das entstehende Aggregat Teil des Zielaggregates ist, um damit eine Vorhersage für die Montageaktionen zu treffen. Details zur Verwendung der Merkmalsmenge lassen sich aus (STONE, 1998) entnehmen, im folgenden soll aber nicht weiter auf den Merkmalsraum, wie ihn Stone verwendet, eingegangen werden.
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, eine Verstärkungsfunktion, wird auch als Reward bezeichnet, weist jeder Kombination aus Zustand und Aktion einen reellen Wert (Verstärkungssignal bzw. Reward) zu. Der Reward $r(s, a_i)$ ist unabhängig vom Reward $r(s, a_j)\ \ \forall i \neq j$

- $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, eine Zustandsübergangsfunktion; Sie legt für jeden Zustand $s \in \mathcal{S}$ fest, bei welcher Aktion $a \in \mathcal{A}$ welcher Folgezustand $s' \in \mathcal{S}$ betreten wird.
- $\pi : \mathcal{S} \to \mathcal{A}$, eine Funktion, die die Menge der Zustände $\mathcal{S}$ auf die Menge der Aktionen $\mathcal{A}$ abbildet. Sie wird Policy bzw. Strategie genannt und bestimmt das Verhalten des lernenden Systems.
  Bei Verwendung des Feature-Spaces $\mathcal{V}$ kann sie folgendermaßen aussehen:
  1. Überführung in den Feature-Space: $f : \mathcal{S} \mapsto \mathcal{V}$
  2. Lernen der Wertefunktion: Die Wertefunktion weist jedem Zustand eine Bewertung zu und wird in diesem Falle mittels Feature-Space definiert. $Q : \mathcal{V} \times \mathcal{A} \mapsto \mathbb{R}$. Q schätzt den erwarteten Reward für jede der möglichen Aktionen.
  3. Auswahl einer neuen Aktion: Es wird eine Aktion a ausgewählt und der mögliche Reward verwendet, um Q zu trainieren.
- $V^\pi : \mathcal{S} \to \mathbb{R}$, eine Wertefunktion, die jedem Zustand $s \in \mathcal{S}$ eine Bewertung zuweist, wenn er die Strategie π verfolgt. Es gibt unterschiedliche Möglichkeiten, V^π zu berechnen. Sei t der momentane Zeitpunkt und s_t der Zustand zum Zeitpunkt t, dann kann man V^π folgendermaßen definieren:
  - Verfahren des akkumulierten Rewards bei einem endlichen Horizont:

    $$V^\pi(s_t) = E\left[\sum_{d=0}^{h} r_{t+d}\right] \tag{4.2}$$

    Bei dieser Methode soll der erwartete Reward für die nächsten h Schritte maximiert werden; das System ist also nur bis zu einem bestimmten Punkt (Horizont) vorausschauend und kümmert sich nicht um die weitere Zukunft.
    In nichtdeterministischen Systemen kann man nicht mit Bestimmtheit sagen, ob bei Ausführung einer Aktion a im Zustand s der Folgezustand s' oder s'' sein wird. Deshalb arbeitet man nicht mit den Größen $\left[\sum_{d=0}^{h} r_{t+d}\right]$, sondern mit deren Erwartungswerten $E\left[\sum_{d=0}^{h} r_{t+d}\right]$.
  - Verfahren des abgeschwächten, akkumulierten Rewards bei einem unendlichen Horizont:

    $$V^\pi(s_t) = E\left[\sum_{d=0}^{\infty} \gamma^d r_{t+d}\right] \quad \text{mit } 0 \leq \gamma < 1 \tag{4.3}$$

    Bei dieser Methode wird die Summation der zukünftigen, erwarteten Rewards nicht nach einer bestimmten Anzahl abgebrochen, stattdessen werden die Einflüsse der Rewards, die weiter in der Zukunft liegen, systematisch abgeschwächt (γ^d mit $0 \leq \gamma < 1$). Es entsteht tatsächlich ein endlicher Horizont, da man sonst ein Problem mit der unendlichen Summe bekäme.
  - Verfahren des gemittelten Rewards bei endlichem Horizont:

    $$V^\pi(s_t) = \lim_{h \to \infty} \frac{1}{h} E\left[\sum_{d=0}^{h} r_{t+d}\right] \tag{4.4}$$

Es wird angenommen, daß das lernende System Aktionen wählt, die den mittleren, erwarteten Langzeit-Reward maximieren. Es bildet damit die obere Schranke von Gleichung 4.3, wenn γ gegen 1 geht.

4.2 Exploration des Zustandsraumes

Wie bereits erwähnt, gehen alle Verfahren des Verstärkungslernens nach dem Prinzip des Versuchs und Irrtums vor. Angenommen, das System hat eine gewisse Strategie π gelernt; dann stellt sich die Frage, wie gut die Strategie ist. Wählt man z.B. eine bisher noch nie verwendete Aktion aus, könnte das schneller zum Ziel führen, es könnte aber auch Zeitverschwendung sein, da die bisherige Strategie an diesem Punkt bereits gut/optimal ist. In der Mathematik ist dies als das Problem der n-armigen Banditen bekannt. Eine Reihe von Automaten soll h-mal bedient werden. Das Problem ist, welcher Spielautomat in welcher Reihenfolge bedient werden soll, um den maximalen Gewinn zu erzielen. Angenommen, alle Automaten sind unabhängig, dann besitzt jeder Automat eine unbekannte aber unabhängige Wahrscheinlichkeit dafür, daß man gewinnt oder nicht. Andererseits existiert für jeden Automaten eine Wahrscheinlichkeit dafür, daß, wenn er gewonnen hat, er nochmal gewinnt...
In diesem Abschnitt sollen die gängigsten Strategien beschrieben werden; weitere Details findet man unter anderem in (D.A.BERRY AND B.FRISTEDT, 1985) und (THRUN, 1992).

4.2.1 Dynamische Programmierung

Dieser Ansatz geht davon aus, daß das System lediglich eine bekannte endliche Anzahl h von Aktionen ausführen darf. Dieser Sachverhalt wird u.a. dadurch beschrieben, daß in Gleichung 4.3 γ für die ersten n Schritte auf 1 gesetzt wird und für alle weiteren auf 0. Unter dem Oberbegriff *Dynamische Programmierung* versteht man Algorithmen, die eine bestimmte Klasse von Problemen möglicherweise beschleunigen.

Immer dann, wenn umfangreiche Berechnungen in kleinere, unabhängige Teilberechnungen zerlegt werden können, die möglicherweise wiederum zerlegt werden usw., lassen sich die Berechnungen durch Verwendung Dynamischer Programmierung eventuell beschleunigen. Es wird versucht, den Algorithmus so anzulegen, daß beim Lösen eines Teilproblems Ergebnisse verwendet werden, die bereits bei der Berechnung anderer Teilprobleme gefunden wurden. Dadurch vermeidet man es, Berechnungen mehrfach durchzuführen, wie es bei den einfachsten, rekursiven Ansätzen meist der Fall ist. Ein Beispiel dafür ist der De Bohr-Algorithmus zur Berechnung von B-Splines (ENGELN-MÜLLGES AND REUTTE, 1993) oder der Algorithmus von Warschall zur Berechnung der transitiven Hülle eines Graphen (SEDGEWICK, 1992). Leider ist die Dynamische Programmierung kein Allheilmittel. Wenn für die Lösung eines komplexen Problems stets andere Teilprobleme benötigt werden, ohne auf bereits existierende Lösungen zugreifen zu können, die Anzahl der Teilprobleme also sehr groß ist, oder ein Problem schlecht aus Teilproblemen zusammengesetzt werden kann, erhält man durch Dynamische Programmierung keinen Gewinn. Auch einige Probleme im Bereich der *Bayes-Netze* lassen sich mit Methoden der Dynamischen Programmierung behandeln. Ein Bayes-Netz besteht aus einer Menge von Zuständen, die mit gerichteten Kanten untereinander verbunden sind. Knoten, in die keine Kanten einlaufen, müssen mit initialen Wahrscheinlichkeiten für ihr Zutreffen bzw. Nicht-Zutreffen belegt werden. Knoten mit einlaufenden Kanten benötigen die qualitativen Einflüsse, also bedingten Wahrscheinlichkeiten

der Vorgänger-Knoten (D.A.BERRY AND B.FRISTEDT, 1985) (PRESMAN AND SONON, 1990). Zu bestimmen ist eine Abbildung von Zuständen (belief states) auf Aktionen. Ein Zustand kann als eine Tabelle von Aktionen und den dadurch erzeugten Reward modelliert werden. $V^*(z_h)$ sei der noch zu erwartende Reward, wenn weitere h Schritte auszuführen sind und davon ausgegangen wird, daß diese h Schritte optimal gewählt werden. Wenn alle Züge ausgeführt sind, ist $V^*(z_0)$ = 0. Das ist der Ansatzpunkt der Dynamischen Programmierung (kleinstes Teilproblem): Wenn für alle Zustände z, in denen noch t Schritte bis zum Ende verbleiben, die Werte für V^* bekannt sind, lassen sich die V^*-Werte für alle Zustände mit einer verbleibenden Zeit von $t+1$ wie folgt berechnen:

$$
\begin{aligned}
n_j &= \text{Aktionswahl} \\
w_j &= \text{erwarteter Reward bei Aktion } n_j \\
z &= (n_1, w_1; n_2, w_2; \ldots, n_k, w_k) \\
V^*(z) &= \max_{i=0\ldots k} \left(\sigma V^*(z^1) + (1-\sigma) V^*(z^2)\right) \\
z^1 &= (n_1, w_1; \ldots; n_i + 1, w_i + 1; \ldots;, n_k, w_k) \\
z^2 &= (n_1, w_1; \ldots; n_i + 1, w_i; \ldots;, n_k, w_k)
\end{aligned}
$$

mit der Varianz σ.
Wenn es also möglich ist, ausgehend vom Zielzustand alle optimalen Aktionen unter Verwendung der vorher bereits berechneten, bis dahin optimalen Aktionsfolgen zu berechnen, spricht man vom dynamischen Programmieransatz. Voraussetzung ist allerdings die Kenntnis der Funktionen r und δ. (BARTO ET AL., 1995) haben sich ausgiebig mit der Beziehung zwischen Dynamischer Programmierung und dem Verstärkungslernen auseinandergesetzt.

4.2.2 Gittins Allocation Indizes

Gittens (GITTINS, 1989) beschreibt eine Methode, die die optimale Aktionsauswahl für k-armige Banditen berechnet. Sie setzt Gleichung 4.3 voraus. Angenommen, eine Aktion sei bereits n mal ausgeführt worden und habe dabei w mal gewonnen. Mit einem bekannten Wert γ existieren Tabellen sogenannter Indices $I(n, w)$. Sie beschreiben eine Kombination aus dem zu erwartenden Gewinn bei einem bestimmten n u. w und einem Wert, der den Informationsgehalt beschreibt, der aus besagter Aktion folgt.

Gittens hat bewiesen, daß bei Wahl der Aktionen mit dem größten Index sich immer ein optimales Verhältnis zwischen Exploration und Exploitation ergibt. Diese Methode funktioniert aber nur, wenn man einen sofortigen Reward bekommt.

4.2.3 Greedy-Strategien

Diese Strategie kann man als die wohl offensichtlichste Methode bezeichnen: Es wird immer die Aktion gewählt, die den höchsten Gewinn (größten Reward) verspricht. Das hat zur Folge, daß suboptimale Lösungen, wenn sie zufällig vor der optimalen gefunden wurden, immer wieder ausgewählt werden und die optimale Lösung nicht gefunden werden kann.

Andere Strategien verwenden Heuristiken, die durch das Ausführen und den dadurch erhaltenen Reward verändert bzw. verbessert werden. Derartige Methoden sind u.a. in (SUTTON, 1990), (KAELBLING, 1993), (SCHMIDHUBER, 1991) untersucht bzw. verwendet worden.

4.2.4 Intervallbasierte Methoden

Bei dieser Klasse von Explorationstechniken wird versucht, während der Lernphase mehr und mehr statistische Informationen zu sammeln. Im sogenannten *interval estimation* Algorithmus (KAELBLING, 1993) wird zu jeder Aktion die Anzahl ihrer Verwendung und der dabei gesammelte Reward gespeichert. Gewählt wird immer die Aktion mit der größten oberen Grenze des Konfidenz-Intervalls $100 \cdot (1 - \alpha)\%$ mit $0 \leq \alpha \leq 1$. Je größer α, umso größer ist die Exploration.

4.2.5 Mit Zufallszahlen arbeitende Strategien

Eine ausschließlich zufällige Wahl der Aktionen kann höchstens in der sehr frühen Phase eines Lernprozesses sinnvoll sein, wenn keinerlei Kenntnis über den zu erwartenden Reward vorliegt. Etwas besser ist es da, im Normalfall die Aktion mit dem besten Reward zu wählen und lediglich mit einer Wahrscheinlichkeit p zufällig eine der schlechter bewerteten Aktionen auszuwählen.

Deutlich besser erscheint es jedoch, eine Kombination aus zufälliger Aktionswahl und gezieltem Verfolgen bereits erkundeter Bereiche im Zustandsraum zu realisieren (*Boltzmann exploration*). Es wird für jede Aktion eine vom erwarteten Reward abhängige Wahrscheinlichkeit berechnet. Je größer der erwartete Reward, desto größer die Wahrscheinlichkeit. Die zufällige Aktionswahl wird dann von diesen Wahrscheinlichkeiten beeinflußt (siehe Tab. 7.4 in Abschnitt 7.5.4). Die Wahrscheinlichkeiten für jede Aktion $P(s, a)$ werden nach Boltzmanns Energiefunktion berechnet:

$$\mathrm{P}(s,a) = \frac{e^{\frac{R(s,a)}{T}}}{\sum_{a'} \left(e^{\frac{R(s,a')}{T}} \right)} \tag{4.5}$$

4.3 Verzögerte Rewards

Ein System soll grundsätzlich das erklärte Ziel erreichen. Es sollte also nicht nur auf den momentanen Reward schauen, sondern auch auf den der folgenden Zustände. Im Extremfall produziert jedoch nur der Zielzustand einen Reward. Es sollen hier einige Techniken Erwähnung finden, die mit dem sogenannten *Verzögerten Reward* umgehen können. Das prominenteste Beispiel sind die Markovschen Entscheidungsprozesse, die bereits oben erwähnt wurden, siehe auch (BELLMAN, 1957), (BERTSEKAS, 1987), (HOWARD, 1960), (PUTERMAN, 1994).

4.3.1 Modellbasierte Strategien

Unter dem optimalen Wert eines Zustands $s \in \mathcal{S}$ soll Gleichung 4.3 verstanden werden, wenn die Strategie π optimal ist.

$$
\begin{aligned}
V^*(s_t) &= \max_\pi E\left(\sum_{d=0}^{\infty} \gamma^d r_{t+d}\right) \\
&= \max_\pi E\left(r_t + \sum_{d=1}^{\infty} \gamma^d r_{t+d}\right) \\
&= \max_\pi E\left(r_t + \gamma \sum_{d=0}^{\infty} \gamma^d r_{(t+1)+d}\right) \\
&= \max_\pi E\left(r_t + \gamma V^*(s_{t+1})\right) \qquad (4.6) \\
&= \max_{a \in \mathcal{A}} \left(r(s_t, a) + \gamma \sum_{s' \in \mathcal{S}} \delta(s_t, a) V^*(s')\right), \forall s_t \in \mathcal{S} \qquad (4.7)
\end{aligned}
$$

Gleichung 4.7 besagt, daß der Wert $V^*(s_t)$ die Summe aus dem erwarteten Verstärkungssignal $r(s_t, a)$ und dem erwarteten, abgeschwächten Wert, also ($\gamma \sum_{s' \in \mathcal{S}} \delta(s_t, a) V^*(s')$) des Folgezustandes s' bei Verwendung der optimalen Aktion a ist. Die optimale Strategie π^* definiert sich als die Funktion, die V^π gegenüber allen Zuständen maximiert:

$$\pi^* = \arg\max_\pi V^\pi(s) \quad \forall s \in \mathcal{S} \qquad (4.8)$$

Setzt man für V^π das entsprechende V^*, ergibt sich:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \left(r(s, a) + \gamma \sum_{s' \in \mathcal{S}} \delta(s, a) V^*(s')\right). \qquad (4.9)$$

Um π^* zu finden, sind mehrere mögliche Ansätze denkbar:

- Suchen der optimalen Wertefunktion V^* durch einen iterativen Algorithmus (value iteration).
- Direktes Manipulieren der Funktion π (policy iteration), so daß sie gegen π^* konvergiert.

Beide Ansätze wurden mehrfach diskutiert und unterschiedlichste Iterationsverfahren untersucht.

4.4 Erlernen einer optimalen Strategie π^*

Zwei Gruppen von Ansätzen werden hier diskutiert: die modellfreien Ansätze und die Ansätze, die versuchen, ein Modell zu lernen. Das eigentliche Problem beider ist die Frage, wie man Rewards vergeben soll. Wie kann man wissen, welche Belohnung eine Aktion zu bekommen hat, insbesondere wenn sie weit in die Zukunft reichende Konsequenzen für das gesamte System hat. Deshalb wird in vielen Ansätzen gewartet, bis der Zielzustand erreicht

ist, um dann nachträglich alle durchgeführten Aktionen zu belohnen. Das setzt aber voraus, daß es einen solchen Zielzustand gibt. Bei fortlaufenden Aufgaben wie z.B. Roboterfußball kann ein Zielzustand nur schwer bestimmt werden, insbesondere dann, wenn sich die Strategien der beiden Teams mit der Zeit verschieben.

In solchen Fällen wird eine *zeitliche Differenzmethode (temporal difference method)* verwendet. Bei diesen Methoden wird der Wert eines Zustandes auf der Grundlage einer sofortigen Belohnung und den Werten der nachfolgenden Zustände berechnet (SUTTON, 1988).

4.5 Q-Lernen

Folgende zusätzliche Größen werden definiert:

$Q(s,a)$**:** Die Wertefunktion beim Q-Lernen.

Q^***:** $Q^*(s,a)$ ist die Schetzung des Lerners für $Q(s,a)$.

V^***:** $V^*(s,a) = \max_a \ Q^*(s,a)$

π^***:** $\pi^*(s) = \arg\max_a \ Q^*(s,a)$

Ziel ist es, eine Wertefunktion $Q(s,a)$ zu erlernen, die jedem Tupel aus einem Zustand $s \in S$ und einer Aktion $a \in A$ eine Zahl (Bewertung) zuordnet. Wenn die Durchführung einer Aktion a im Zustand s in den Zustand s' führt, in dem es eine Belohnung gibt, dann wird der Wert der Funktion $Q^*(s,a)$ wie folgt erhöht:

$$Q^*(s,a) := Q^*(s,a) + \epsilon \cdot r(s') \tag{4.10}$$

mit der Lernschrittrate ϵ und der Belohnung $r(s')$. Sind viele der Rewards Null, sollte man noch den Wert des Folgezustands berücksichtigen. Da im Folgezustand s' mehrere Aktionen möglich sind, wählt man den größtmöglichen Folgewert $V(s')$:

$$V(s') = \max_a Q^*(s',a) \tag{4.11}$$

Die Verbesserung von Q^* (Gleichung 4.10) ändert sich dann zu:

$$Q^*(s,a) := Q(s,a)^* + \epsilon(r(s',a) + \gamma V(s')) \tag{4.12}$$

Schließlich wird noch ein Abklingterm eingeführt, um zu verhindern, daß $Q*$ über alle Maßen wächst:

$$Q^*(s,a) := Q^*(s,a) + \epsilon((r(s',a) + \gamma V(s')) - Q^*(s,a)) \tag{4.13}$$

Wenn Features gemäß Gleichung 4.1 verwendet werden, wird an Stelle des Zustands s der Feature-Vektor $f(s)$ in die Bewertungsfunktion Q^* eingesetzt.
Q^* wird wie folgt definiert:

$$Q^*(s,a) := r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \delta(s,a) \max_{a'} Q^*(s',a'), \tag{4.14}$$

und die Lernregel lautet:

$$Q^*(s,a) := Q^*(s,a) + \epsilon(r(s,a) + \gamma \max_{a'} Q^*(s',a') - Q^*(s,a)). \tag{4.15}$$

Häufig wird für ϵ ”1” gewählt, so daß die Gleichung zu

$$Q^*(s,a) := r(s,a) + \gamma \max_{a'} Q^*(s',a'). \tag{4.16}$$

wird.

In nichtdeterministischen Umgebungen ist die Konvergenz von Gleichung 4.15 respektive Gleichung 4.16 aber nicht gegeben. Man stelle sich z.B. eine nichtdeterministische Belohnungsfunktion $r(s,a)$ vor, die jedes Mal wenn der Übergang (s,a) vollzogen wird, eine andere Belohnung vergibt. Das führt dann dazu, daß Q^* sich immer wieder verändert, selbst wenn es bereits die perfekte Q Funktion darstellt. Um dieses Problem zu umgehen, läßt sich z.B. Gleichung 4.16 so verändern, daß sie mit einem sich abschwächenden gewichteten Mittelwert zwischen der aktuellen Schätzung Q^* und dem revidierten Wert arbeitet:

$$Q_i^*(s,a) \leftarrow (1-\alpha_i) Q_{i-1}^*(s,a) + \alpha_i [r + \max_{a'} Q_{i-1}^*(s',a')]$$

mit α_n:

$$\alpha_i = \frac{1}{1 + \mathsf{visits}_i(s,a)}$$

Der Index i soll anzeigen, daß es sich um die i-te Iteration des jeweiligen Wertes handelt. Die Größe $\mathsf{visits}_i(s,a)$ gibt an, wie oft der Übergang (s,a) zum Zeitpunkt der i-ten Iteration durchgeführt wurde. Im initialen Zustand, also beim ersten Ausführen eines Überganges (s,a) wird übrigens Gleichung 4.5 zu ”1” und damit Gleichung 4.5 zu Gleichung 4.16

Durch diese Modifikation werden die Änderungen an Q^* mit zunehmender Lerndauer immer kleiner und die Konvergenz kann sichergestellt werden. Diese Wahl für α_n ist allerdings nicht die einzige, die eine Konvergenz im nichtdeterministischen Falle garantiert.

Weitere Details finden sich in (WATKINS, 1989) (MITCHELL, 1997). Der große Vorteil von Q-Lernen gegenüber anderen Verfahren aus diesem Bereich ist die Unabhängigkeit der Konvergenz vom Explorationsverfahren. Die Q-Werte konvergieren unabhängig vom Verhalten der Agenten zu den optimalen Werten.

4.5.1 Adaptive Heuristic Critic

Bei diesem Ansatz werden im wesentlichen zwei Komponenten implementiert: Die eigentliche Lernkomponente (*RL*), die eine Abbildung $\mathcal{S} \rightarrow \mathcal{A}$ lernt, und eine als (*AHC*) bezeichnete Komponente, die der (*RL*)-Komponente bei jedem Lernschritt einen heuristischen Wert ν übermittelt. Der (*RL*)-Algorithmus versucht, die zu erhaltenden ν's zu maximieren, ν ist also für die (*RL*)-Komponente der Reward. Um die Rewards zu maximieren, legt der Algorithmus

nur den aktuell erhaltenen Wert für ν, den Zustand s und die ausgewählte Aktion a zugrunde. Es kann z.B. einer der Ansätze für das Problem der k-armigen Banditen verwendet werden. Die (*AHC*)-Komponente versucht, geeignete Werte für ν zu generieren, was wiederum durch einen Lernprozeß geschieht. Sie verwendet die Zustände vor s und nach s' der Aktionsauswahl, das tatsächliche Verstärkungssignal r und die von der (*RL*)-Komponente generierte Aktion a, um die momentane Strategie der (*RL*)-Komponente zu bewerten. Um die Wertefunktion zu berechnen, verwendet die (*AHC*)-Komponente den sogenannten $TD(0)$-Algorithmus (SUTTON, 1988).

$$V(s) := V(s) + \alpha(r + \gamma V(s') - V(s)). \tag{4.17}$$

Das Ganze kann als zweistufiger, sich immer wiederholender Prozeß verstanden werden. Zunächst wird die von der (*RL*)-Komponente definierte Strategie π als konstant angesehen. Der (*AHC*)-Algorithmus lernt darauf die Wertefunktion V^π. Dann wird die Wertefunktion als bekannt angenommen, und der (*RL*)-Algorithmus lernt eine neue Strategie π'. Details über das Konvergenzverhalten einiger (*AHC*)-Ansätze finden sich in (WILLIAMS AND BAIRD, 1993). Nachteile gegenüber Q-Lernen:

- ❏ AHC ist deutlich komplizierter zu implementieren.
- ❏ Die Konvergenz von AHC ist von der Art der Exploration des Zustandsraumes abhängig.
- ❏ Die Lernraten der beiden Komponenten müssen aufeinander abgestimmt werden, damit das Verfahren konvergiert; die langsamste Komponente bestimmt das Tempo.

4.5.2 Der $TD(\lambda)$-Algorithmus

Der $TD(\lambda)$-Algorithmus ist eine Verallgemeinerung des $TD(0)$-Algorithmus mit $\lambda = 0$.

$$V(u) := V(u) + \alpha(r + \gamma V(s') - V(s))e(u). \tag{4.18}$$

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{s,s_k}, \text{ mit } \begin{cases} 1 & \text{wenn } s = s_k \\ 0 & \text{sonst} \end{cases} \tag{4.19}$$

Eine ganze Reihe von Untersuchungen haben sich mit der Effizienz des $TD(\lambda)$-Ansatzes beschäftigt und Verbesserungsvorschläge gemacht (CHICHOSZ AND MULAWKA, 1995) (DAYAN, 1992) (SINGH, 1996).

4.6 „Guided Table Fill In“ - Lernen in hochdimensionalen Aktionsräumen

„Guided Table Fill In“ (SCHNEIDER, 1994) ist ein Verfahren zur intelligenten Aktionsauswahl in hochdimensionalen Aktionsräumen. Schneider behandelt in (SCHNEIDER, 1994) die Frage, wie sich eine Wurfbewegung mit einem Zweigelenkroboter erlernen läßt. Jedes der zwei Gelenke benötigt eine Abfolge von sechs Gelenkgeschwindigkeiten. Werden sie hintereinander an einen Gelenkregler gesendet, sollen sie zu einer Wurfbewegung führen.

Schneider interpretiert die zwölf Werte als zwölfdimensionalen, stetigen Aktionsraum. Das eigentliche Problem stellt sich als Suche eines geeigneten Punktes innerhalb des zwölfdimensionalen Raumes dar. *(Schneider sagt nicht, warum er das Problem nicht als Suche eines Pfades innerhalb eines zweidimensionalen Raumes betrachtet, obwohl die aufeinanderfolgenden Gelenkgeschwindigkeiten aus physikalischen Gründen nicht unabhängig voneinander sein können.)* In (MOOR, 1990) wird ein „Randomwalk“ - Verfahren beschrieben, um einen möglichen Punkt im Raum zu finden, jedoch nur in Aktionsräumen mit geringer Dimensionalität. „Guided Table Fill In“ ist eine Methode, die sich an die Idee der genetischen Algorithmen anlehnt. Ausgehend von einem initialen Satz von Gelenkgeschwindigkeiten (Aktionsabfolgen) werden diese anhand einer Bewertung (hier die Wurfweite) verändert/verbessert. Sie dient dazu, die Güte zweier Aktionsfolgen zu beurteilen. Weitere Änderungen werden nur an der Aktionsfolge mit der besten Bewertung vorgenommen.

4.6.1 Unterschiede zum Problem des Erlernens einer Montagesequenz.

- Es werden Sequenzen *gleichartiger* Aktionen ausgeführt (es wird jeweils eine Gelenkgeschwindigkeit an die Gelenkregler übergeben).
  Montagesequenzen bestehen aus einer Abfolge nichtgleichartiger Aktionen (greifen, stechen, warten, loslassen, ...)

- Eine Montagesequenz spannt nicht wie die behandelten Problemklassen einen kontinuierlichen tensorischen Raum auf, sondern besteht aus einem Aktionsbaum mit diskreten Größen. Die Äste können unter Umständen eine unendliche Tiefe besitzen, Abb. 7.12.

- Das von Schneider verwendete Verfahren geht von einer kontinuierlichen Bewertung für jede Aktionsfolge aus. Die Länge der Aktionsfolge wird a priori festgelegt.
  Die Anzahl an Einzelaktionen in einer Montagesequenz ist variabel und a priori unbekannt. Eine kontinuierliche Bewertung heißt, daß es einer Bewertung des gebauten Objekts bedarf (Der Zusammenbau war zu 0.6 erfolgreich), bzw. der Aktionsfolge (Die Aktion war zwar erfolgreich, es wurden aber *viele* Umwege gemacht).
  Häufig liegt aber nur eine binäre Bewertung vor: Ziel wurde erreicht oder Ziel wurde verfehlt.

- „Guided Table Fill In“ zieht keinerlei Nutzen aus Informationen, die es während der Ausführung einer Sequenz, also zwischen zwei einzelnen Aktionen, aus möglichem Modellwissen beziehen kann.

5

Spezifikation von Bauteilen und Aggregaten (Objektmodell)

5.1 Objektrepräsentationen

Mit dem Begriff *Objektrepräsentationen* ist eine Datenstruktur gemeint, die gewisse Aspekte mechanischer Aggregate (auch häufig als Baugruppen bezeichnet) beschreibt. Mehrere Artikel beschäftigen sich mit der Definition des Begriffes „mechanisches Aggregat“ (JONES AND WILSON, 1996), (RABEMANANTSOA AND PIERRE, 1996), (REQUICHA AND WHALEN, 1991), (WOLTER., 2000). In der englischsprachigen Literatur wird anstelle von Aggregat meist das Wort *assembly* verwendet, womit dann entweder der Vorgang des Zusammenbaus oder das dabei entstehende Artefakt gemeint ist. Hier soll es um die zweite Bedeutungsform gehen.

Im wesentlichen läßt sich ein Aggregat als Menge miteinander verbundener Einzelbauteile beschreiben. Mit *verbunden* ist die Einschränkung gewisser Freiheitsgrade durch das Zusammenfügen der einzelnen Komponenten gemeint (RABEMANANTSOA AND PIERRE, 1996). Auf unterstem mathematischen Niveau läßt sich ein Aggregat aber auch als Menge von Bauteilen mit festen Positionen definieren (REQUICHA AND WHALEN, 1991).

Das wesentliche Verwendungsgebiet für Objektrepräsentationen sind Systeme zur automatischen Generierung von Montagesequenzen. Diese Sequenzgenerierung ist ein seit langem untersuchtes Gebiet mit zahllosen Ansätzen. Sie benötigen neben den Algorithmen zur Erzeugung einer Sequenz Datenstrukturen, auf denen diese Algorithmen arbeiten. Deshalb ist der Begriff des Objektmodells etwas weiter zu fassen als lediglich die oben angegebene Beschreibung. Im weiteren sollen einige der gängigsten Modelle beschrieben werden; dabei lassen sich zwei Aspekte herausstellen:

① der geometrische Aspekt; dabei werden im wesentlichen Lagebeziehungen der Objekte modelliert und häufig mit anderen physikalischen Größen verknüpft, und

② topologische/strukturelle Modelle, die die Struktur der Aggregate modellieren, meist als hierarchische Graphen, ohne genauer auf die physikalische Struktur einzugehen.

Viele der in der Praxis angewandten Modelle sind nicht streng einer der beiden Klassen zuzuordnen, sie besitzen beide Aspekte.

5.1.1 Oberflächengraphen

Unter einem Aggregat wird hier eine Menge von Oberflächen (nicht notwendigerweise Ebenen) und eine Beschreibung ihrer Beziehungen verstanden. Z.B. kann diese Beschreibung als BNF definiert werden, was in diesem Falle nichts anderes ist als eine textuelle, genormte Beschreibung eines Graphen. (THOMAS AND NISSANKE, 1996) stellen eine solche BNF für zylindrische Körper vor. Aus einer derartigen textuellen Aggregatrepräsentation läßt sich im Rechner leicht ein Graph generieren. Er beschreibt ausschließlich die Struktur des Aggregates durch die Verbindungen von Oberflächen. Ein Knoten repräsentiert eine Oberfläche, eine Kante die Verbindungsbeziehung der beiden beteiligten Knoten (Oberflächen). Um zwei Oberflächen zu verknüpfen, müssen sie entweder adjacent zueinander sein oder eine gemeinsame Kante besitzen.

Eine solche Darstellung definiert ausschließlich die Form, nicht aber die Eigenschaften der einzelnen Flächen; eine Information, wie sie nicht nur im Bereich der Sequenzgenerierung für Montagepläne von Interesse ist, sondern auch im Bereich der Computergrafik. Deshalb wird der reine Oberflächengraph so erweitert, daß jeder Kante und jedem Knoten eine Menge von Attributen zugewiesen werden. Man kann sogar dazu übergehen, die einzelnen Flächenparameter wie Kantenlängen, Radien, Ort und Orientierung als Attribute zu beschreiben. In (THOMAS AND NISSANKE, 1996) wird für den dort entwickelten Oberflächengraphen auf Basis einer BNF (s.o.) eine ganze Algebra aufgestellt.

5.1.2 Boundary-Representation

Eine *Boundary Representation*, kurz B-Rep. ist eine topologische und geometrische Beschreibung der Hülle (Begrenzung) eines Objekts/Bauteils mit Hilfe eines Graphen. Die Knoten stehen dabei für Flächen, Kanten und Punkte, die Verbindungen zwischen den einzelnen Knoten für Nachbarschaftsbeziehungen. *Boundary Representations* eignen sich daher gut, um Oberflächeneigenschaften zu beschreiben. Weiterhin gibt es in vielen B-Reps. zwei weitere Konzepte:

Loops (Schleifen): Damit ist eine geschlossene Folge von Kanten gemeint, die eine Reihe von Flächen (faces) zu einem Ring verbindet.

Shells (Muscheln): Eine Menge maximal verbundener Objekte.

Beispiele für die Implementation von B-Rep. Graphen finden sich in (BAUMGART, 1972), (ANSALDI ET AL., 1985), (WOO, 1985), (WEILER, 1986).

5.1.3 Volumetrische Modelle

Volumetrische Modelle werden auch als Constructive Solid Geometries (CSG) bezeichnet. Sie beschreiben nicht die Hülle eines Objekts, sondern das gesamte Volumen; sie werden in zwei Gruppen unterteilt:

Zerlegungsmodelle: Ein Objekt wird als eine Menge von Basisobjekten beschrieben, die quasi „zusammengeklebt“ werden.
Es existieren wiederum zwei mögliche Untergruppen: die Objektschemata und die Raumschemata.

Objektschemata: Hier werden die Objekte als paarweise Kombination aus quasi unzusammenhängenden Elementarzellen beschrieben, deren Vereinigung das Objekt umschließt (BOISSONNAT, 1984). Sie werden weniger für die Analyse von Montageprozessen als für die Objektrekonstruktion und für Finite-Element-Netze verwendet.

Raumschemata: Der vom Objekt ausgefüllte Raum wird in regelmäßige Volumenelemente (Voxels) zerteilt (SAMET, 1990). Auf Grund der Verwendung einer bestimmten Sorte von Volumenelementen, z.B. Würfel, kann dieser Raum nur näherungsweise überdeckt werden. Je feiner man die Volumenelemente macht, umso genauer läßt sich das Objekt beschreiben. Die Beschreibung wird allerdings auch speicherintensiver und rechenaufwendiger.

Einige dieser Modelle werden als Zusatzbeschreibungen zu anderen Modellen (B-Rep. oder Solid-Geometries) verwendet, um deren Schwächen bei der Berechnung statischer bzw. dynamischer Eigenschaften auszugleichen (SAMET, 1990).

Konstruktionsmodelle: Ein Objekt wird als eine boolesche Kombination von Basisobjekten beschrieben (REQUICHA, 1980).

Ziel ist die Erzeugung baumartiger Graphen mit einem Wurzelknoten, gerichteten Kanten, aber keinen Zyklen. Die inneren Knoten stehen für boolesche Mengen-Operatoren auf den darunterliegenden Knoten oder für starre Bewegungen, die Lagebeziehungen ausdrücken. Ein Knoten kann einen, zwei oder keinen Folgeknoten besitzen. Diese Blätter korrespondieren mit Basisformen wie z.B. Zylindern, Flächen, Würfeln etc. und einem Satz von Parametern, mit deren Hilfe die Basisformen skaliert und positioniert werden können. Beliebige Erweiterungen bis hin zu Funktionen sind denkbar und auch bereits getestet worden. Diese Darstellungen sind zwar eindeutig, leider können aber mehrere Darstellungen das gleiche Bauteil beschreiben, weshalb man sie nicht zum Graphmatching verwenden kann.

Um die Vorteile von B-Rep.'s und CSG's zu vereinen, werden wie z.B. in (PRINZ ET AL., 1996) beide Modellierungsformen parallel eingesetzt.

5.1.4 Relationales Baugruppenmodell

Das von (DE MELLO, 1989) eingeführte und von (MOSEMANN, 2000) erweiterte relationale Baugruppen-Modell ist ein Tripel, bestehend aus einer Menge $\mathcal{C}$ von Bauteilen, einer Menge $\mathcal{R}$ von Relationen und einer optionalen Hierarchie H auf den Bauteilen aus $\mathcal{C}$.

Bauteile: Bauteile werden durch ihre absolute Weltposition, ihre Montagefeatures und eine Constructive Solid Geometry beschrieben. Montagefeatures sind relativ zum Bauteilursprung positionierte Montagepunkte. Sie besitzen einen bestimmten Typ, z.B. Loch, Auflagefläche, Stift, ...

Relationen: Zwischen allen Bauteilen einer Baugruppe besteht eine Relation. Bei n Bauteilen bedeutet das $\binom{n}{2}$ Relationen. Jede Relation besteht aus den beteiligten Bauteilen, der relativen Lage der beiden Bauteile zueinander und einer Liste aller symbolisch räumlichen Relationen und aller physikalischen Kontakte.

Mit Hilfe einer solchen Beschreibung lassen sich nicht nur alle Montagesequenzen generieren, sondern auch Optimalitätskriterien einführen. Durch die Speicherung physikalischer Relationen wie z.B. Reibung kann die Stabilität einer Baugruppe während der Montage in die Bewegungsplanung der montierenden Roboter einbezogen werden (MATTIKALLI ET AL., 1994), (MOSEMANN ET AL., 1997), (MOSEMANN ET AL., 1998).

5.1.5 Strukturorientierte Modelle

In diese Klasse der Objektrepräsentationen fallen eine Vielzahl von Modellen, so daß es unmöglich ist, alle im Detail zu beschreiben. Deshalb soll hier nur auf die bekanntesten und einige neuere Vertreter dieser Klasse eingegangen werden.

Hierarchische Modelle

In dieser Klasse von Modellen wird versucht, eine gewisse Ordnung in die Baugruppen eines Aggregates zu bringen, indem man ihnen z.B. funktionale Rollen zuschreibt (CHAKRABARTY AND WOLTER, 1995),(HOFFHENKE AND WACHSMUTH, 1997) sowie (BAUCKHAGE ET AL., 1999). Auch (MOSEMANN, 2000) integriert eine optionale Hierarchisierung in sein relationales Baugruppenmodell. Üblicherweise werden bei diesen Ansätzen Baumstrukturen aufgebaut (Abb. 5.4), die bestimmte Aspekte des Aggregates darstellen. Großer Nachteil dieser Ansätze: Ein Aggregat kann oft auf unterschiedliche Arten repräsentiert werden.

AND/OR-Graphen

AND/OR-Graphen sind ein allgemeines Konzept, das nicht zwangsweise auf das Gebiet der Montageplanung beschränkt sein muß. De facto besitzen sie aber nur hier eine Bedeutung. Ihre allgemeine Verwendbarkeit zeigt sich auch darin, daß sie sich immer in ein Petri-Netz umwandeln lassen (CAO AND SANDERSON, 1998). AND/OR-Graphen gelten als die kompakteste Darstellung aller möglichen Montagesequenzen eines Aggregates.

Sei $\mathcal{P}$ die Menge aller Bauteile und $\mathcal{C}$ die Menge aller Verbindungen zwischen zwei Bauteilen. Dann läßt sich ein Aggregat im einfachsten Fall durch das Paar $\langle \mathcal{P}, \mathcal{C} \rangle$ beschreiben. Jeder Knoten eines AND/OR-Graphen ist eine Untermenge $\mathcal{P}'$ von $\mathcal{P}$ unter der Bedingung, daß die Bauteile von $\mathcal{P}'$ ein stabiles Aggregat bilden. Jeder Knoten θ_k läßt sich in zwei Teilbaugruppen θ_i, θ_j zerlegen, so daß $\theta_i \cup \theta_j = \theta_k$ ist, vorausgesetzt, die Zerlegung in θ_i und θ_j ist mechanisch tatsächlich durchführbar. Die beiden Kanten, die die Unterbaugruppen θ_i, θ_j mit dem aus ihnen entstehenden Aggregat θ_k verbinden, werden als *AND-Arc* bezeichnet. Er wird dann als Paar $\langle \theta_k, \theta_i, \theta_j \rangle$ dargestellt, Abb. 5.1. Es gibt außerdem noch die sogenannten *IST-Arc*'s[1] zwischen zwei Zuständen λ_1 und λ_2. Sie repräsentieren interne Änderungen eines Aggregates, also Operationen, bei denen das Aggregat nicht auseinander genommen wird, sondern z.B. seine Lage verändert. Für das Aufstellen aller möglichen Montagesequenzen werden die *IST-Arc*'s λ_1, λ_2 nicht benötigt. Um einen konkreten Montagevorgang mit einem realen Roboter planen zu können, sind sie allerdings unabdingbar.

Formale Definitionen eines AND/OR-Graphen gibt es viele; in (CAO AND SANDERSON, 1998) werden allein drei mögliche Definitionen angeboten. Hier die Definition von de Mello

[1] Internal State Transition.

selbst: Der *AND/OR-Graph* aller durchführbaren Montagesequenzen eines Aggregates, das aus den Bauteilen $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ besteht, ist das Paar $\langle \mathcal{S}_P, \mathcal{D}_P \rangle$ mit

(5.1)
$$\mathcal{S}_P = \left\{ \theta \in \prod(\mathcal{P}) \mid sa(\theta) \wedge st(\theta) \right\},$$

der Menge stabiler Unterbaugruppen und

(5.2)
$$\mathcal{D}_P = \{(\theta_k, \{\theta_i, \theta_j\}) \mid [\theta_i, \theta_j, \theta_k \in S_P] \ \wedge \ [\mathcal{U}(\{\theta_i, \theta_j\})] \ \wedge \ [mf(\{\theta_i, \theta_j\})] \ \wedge \ [gf(\{\theta_i, \theta_j\})]\},$$

der Menge der durchführbaren Montageoperationen.

$\prod(\mathcal{P})$ ist die Menge aller Untermengen von $\mathcal{P}$ (Potenzmenge),

$sa(\theta)$ ist wahr, wenn θ ein existierendes Teilaggregat ist,

$st(\theta)$ ist wahr, wenn θ ein stabiles Aggregat ist,

$\mathcal{U}\{(\mathbf{A_1}, \mathbf{A_2}, \ldots, \mathbf{A_n})\}$ ist die Menge $A_1 \cup A_2 \cup \cdots \cup A_n$,

$mf(\{\theta_i, \theta_j\})$ ist wahr, wenn die Montage von θ_i u. θ_j mechanisch möglich ist,

$gf(\{\theta_i, \theta_j\})$ ist wahr, wenn die Montage von θ_i u. θ_j geometrisch möglich ist.

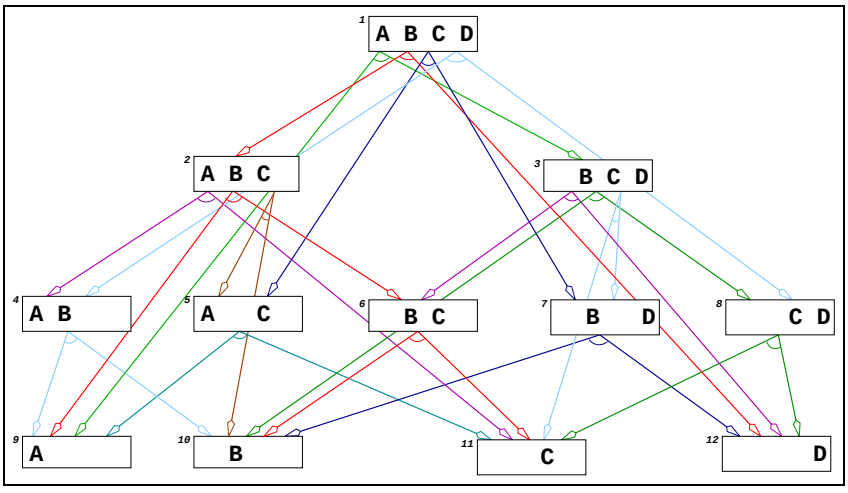


Abbildung 5.1: AND/OR-Graph-Repräsentation eines Aggregates mit 4 Bauteilen $\{A, B, C, D\}$. Mit einem AND-Arc wird ein Knoten mit jeweils zwei Folgeknoten (subassemblies) verbunden. Der Graph beinhaltet alle möglichen Montagesequenzen, die man erhält, wenn man von den Blättern zur Wurzel aufsteigt.

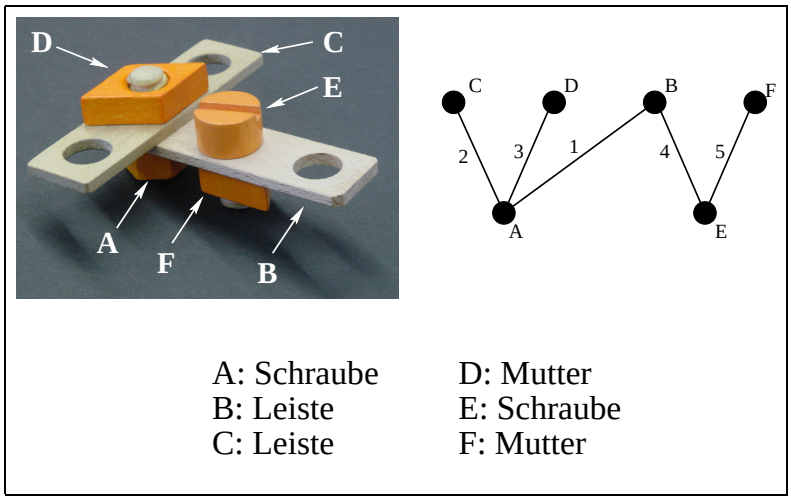


Abbildung 5.2: Das auf der linken Seite gezeigte Baufix-Aggregat läßt sich mittels Liaison-Graph beschreiben (rechts). Die Bauteile A-F werden dabei mit Hilfe von fünf Liaisons verbunden. Die Tatsache, daß die Bauteile A, B, C u. D. bzw. B, E u. F. einen gemeinsamen Verbindungspunkt haben, wird durch den Liaison-Graphen nicht repräsentiert.

Liaison-Graphen

Sie sind eines der ältesten Konzepte im Bereich der Objektmodellierung. Eine aktuelle Beschreibung des Konzepts findet sich in (ABELL ET AL., 1991). Durch Aufstellen eines Graphen werden die Verbindungen der Baugruppen eines Aggregats beschrieben, Abb. 5.2. Jeder Knoten steht für ein Bauteil, jede Kante für eine Verbindungsrelation, auch *liaison* genannt. Was man unter einer Liaison versteht, ist nicht genau definiert. Bei einer Darstellung reicht bereits ein Kontakt der Bauteile, bei anderen muß eine Form von Verbindung hergestellt werden, z.B. durch ein Gewinde.

Die Anzahl der Liaisons kann verwendet werden, um ein Komplexitätsmaß für Aggregate zu definieren (FAZIO ET AL., 1999). Sei L die Anzahl der Liaisons in einem Aggregat und N die Anzahl der Bauteile, dann ist

$$C = \frac{(L - L_{\text{min}})}{L_{\text{max}} - L_{\text{min}}} = \frac{(L - N + 1)}{(N - 1)} \cdot \left(\frac{N}{2} - 1\right)^{-1} \tag{5.3}$$

die sogenannte Konnektivität. L_{min} ist die minimale Anzahl von Verbindungen in einem Aggregat, L_{max} die maximale:

$$L_{\text{min}} = N - 1 \tag{5.4}$$

$$L_{\text{max}} = \frac{N}{2} \cdot (N - 1) \tag{5.5}$$

C ist ein intuitives Maß für die Montagekomplexität. Es liegt zwischen 0 und 1, wobei 0 für ein minimal verbundenes Aggregat steht und 1 für ein sehr komplexes Aggregat (maximale Konnektivität). Die einzelnen Knoten eines Liaison-Graphen lassen sich z.B. durch

Hinzunahme von Positions/Orientierungsinformationen und anderen Werten erweitern. Baugruppen können durch eigene Liaison-Graphen beschrieben werden, die dann in einem Gesamtgraphen nur noch als einzelner Knoten auftauchen. Durch die rekursive Dekomposition von Liaison-Graphen in Teilaggregate (*subassemblies*) lassen sich Montagesequenzen automatisch generieren (LEE, 1994).

Zustandsübergangsdiagramm (*State transition diagram*)

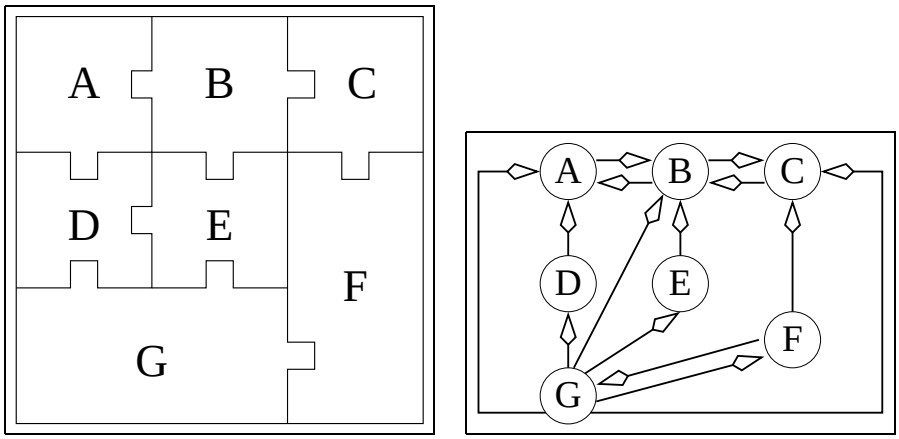
Dieses in (WYNNE ET AL., 1998) vorgestellte Konzept ist für die Beschreibung von Montageprozessen entwickelt worden. Es handelt sich allerdings um ein sehr allgemein gefaßtes Konzept, mit dem sich jede Form von Ablaufprozessen beschreiben läßt. Bezogen auf die Beschreibung von mechanischen Aggregaten wird ein solches Aggregat nicht als eine Ansammlung bereits verbundener Einzelteile oder Flächen bzw. Volumenelemente betrachtet, sondern als dessen Montageprozeß, also ein Vorgang (Prozeß), der aus einer Menge von Eingaben (Bauteile) ein bestimmtes Ergebnis (Aggregat) erzeugt. Dabei durchläuft er eine Reihe von Zuständen (States). Jeder Zustand wird durch seine Eingangs- und Ausgangsgrößen beschrieben. Der einfachste Prozeß besteht demnach aus genau einem Zustand, nämlich dem, dessen Eingangsbauteile in einem Schritt in ein Aggregat umgewandelt werden. Ein Beispiel ist eine Waschmaschine, die schmutzige Wäsche in saubere verwandelt, wobei sie mehrere Zustände durchläuft {Vorwäsche, Hauptwäsche, Spülen, Schleudern}.

Sei $\mathcal{I}$ eine Menge von Eingangsgrößen, $\mathcal{O}$ eine Menge von Ausgangsgrößen und $\mathcal{C}$ eine Menge von Nebenbedingungen, dann wird ein Zustand S wie folgt definiert: $S = \langle I, O, C \rangle$. Eine geordnete Abfolge von Zuständen oder ein Graph, in dem die Zustände Knoten sind und die Zustandsübergänge gerichtete Kanten, wird unter der Voraussetzung, daß die Übergänge tatsächlich durchführbar sind, Zustandsübergangsdiagramm *(State transition diagram)* genannt. Es ist nicht gesagt, daß ein Zustand S nur genau eine eingehende und eine ausgehende Kante besitzt.

Präzedenz-Graph (Precedence-Graphs)

Präzedenz-Graphen sind ein von (PRENTING AND BATTAGLIN, 1964) vorgestelltes Konzept, das hauptsächlich im Bereich des *line balancing* Einsatz findet. Es hat große Bedeutung für alle Verfahren, die sich mit irgendeiner Art von *Scheduling* Problemen befassen. Man stelle sich ein Aggregat aus mehreren Teilen vor wie z.B. in Abb. 5.3(a). Nehmen wir weiterhin an, daß eine vorgegebene Montagerichtung definiert ist, z.B. von oben, dann kann Bauteil *F* nicht mehr eingebaut werden, wenn *C, B* und *G* schon eingebaut sind, da es sonst mit ihnen zusammenstoßen würde. Einen *Präzedenz-Graphen*, der diese Problematik beschreiben soll: Montage des Aggregats aus Abb. 5.3(a), zeigt Abbildung Abb. 5.3(b). Jeder Knoten repräsentiert ein Bauteil. Eine Kante von Bauteil p zu q' liegt genau dann vor, wenn Bauteil q vor Bauteil q' eingebaut werden muß[2]. Es gibt, wie man leicht erkennt, mehrere Präzedenz-Graphen, die den Zusammenbau des Aggregats Abb. 5.3(a) beschreiben. Wird eine andere Montageorientierung gewählt, erhält man auch einen anderen Graphen (z.B. unten heißt, daß alle Pfeile in Abb. 5.3(b) umgedreht werden müssen). Um das gezeigte Aggregat montieren zu können, muß man mehrmals die Montagerichtung ändern. Diese Tatsache läßt sich aus dem Graphen ableiten. Man sieht z.B., daß A und B , B und Z sowie G und F paarweise

[2] Immer unter der Prämisse einer vorgegebenen Einbaurichtung.



(a) Ein Aggregat mit 7 Bauteilen

(b) Präzedenz-Graph für einen Einbau von oben

Abbildung 5.3: Beispiel für einen Präzedenz-Graphen (b). Die Einzelteile des Aggregats (a) hängen in ihrer Montagereihenfolge von einander ab. Um den passenden Graphen aufstellen zu können, muß man eine Montagerichtung wählen: hier z.B. von oben.

aufeinander zeigen. Diese Bauteile sind also bei der gegebenen Montagerichtung paarweise voneinander abhängig und können demzufolge nicht aus der entsprechenden Richtung montiert werden. Viele Arbeiten verwenden Präzedenz-Graphen, um die möglichen Montagereihenfolgen eines Aggregats zu ermitteln. Dabei geht es unter anderem (CHEN AND HENRIOUD, 1994) darum, wie man bei einem gegebenen, meist sehr komplexen Aggregat möglichst wenig Graphen generieren muß.

Montagemodelle *(assembly-models)*

In dieser Klasse von Repräsentationen werden Aggregate dadurch beschrieben, daß man die möglichen Kombinationen, relativen Positionen und Beziehungen ihrer Einzelbestandteile näher spezifiziert. Die verschiedenen Realisationen unterscheiden sich im wesentlichen darin, wie sie die Beziehung zwischen zwei Bauteilen modellieren. In einem sehr frühen Modell von 1977 wurden Baumstrukturen generiert, deren Blätter auf die einzelnen Komponenten referenzierten und die Knoten Montageoperationen beschrieben. Aus derselben Zeit stammt IBM´s AUTOPASS Projekt (LIEBERMANN AND WESLEY, 1977). Hier beschreiben die Knoten geometrische Objekte, und die Kanten stehen für eine von vier Relationen: *{teil-von, Anfügung, Einschränkung, Verknüpfung}*. (DE MELLO AND SANDERSON, 1986) u. (WOLTER, 1989) stellten später ein Relationsschema vor, mit dessen Hilfe aus *assembly-models* alle durchführbaren Montagesequenzen errechnet und beschrieben werden können (AND/OR-Graphen).

Eigenschaftsmodelle *(feature-models)*

Ein ebenfalls wichtiger Aspekt bei Montageoperationen ist die explizite Beschreibung der Form-Features. Dabei werden Aggregate als Verbindungen von Teilen beschrieben, wobei

die Verbindungen mit den einzelnen Fügeoperationen korrespondieren. Üblicherweise werden keine geometrischen Eigenschaften mitbeschrieben. Das Hauptproblem besteht darin, aus einer geometrischen Beschreibung die einzelnen Features zu extrahieren, ein Problem, das von vielen Wissenschaftlern untersucht wurde. Aus neuerer Zeit stammen die Arbeiten von (VAN HOLLAND ET AL., 1995) und (CARRACIOLO ET AL., 1995).

5.1.6 Lernen von Montagesequenzen anhand von funktionalen Modellen

In der von C. Bauckhage vorgestellten Methode (BAUCKHAGE ET AL., 1999) wird aus einem gegebenen Aggregat eine Montagesequenz generiert. Das Aggregat soll in Form eines CAD - Modells oder als Bild vorliegen.

Aus der vorliegenden Repräsentation wird das „funktionale Modell“ generiert, das aus sogenannten funktionalen Komponenten besteht. Als Beispiele für solche funktionalen Komponenten führt Bauckhage Muttern-Teile (NUT-Parts), Bolzen-Teile (BOLT-Parts) und weitere Teile (MISC-Parts) an. Mit letzteren sind Teile gemeint, die nur lose Verbindungen eingehen können. Es wird ein einfacher Graph vorgestellt, der sich theoretisch beliebig reproduzieren läßt. Es entsteht daraus ein Baum aus sich immer wiederholenden, gleichartigen Subbäumen (Abb. 5.4), in dem die funktionale Bedeutung eines Subbaumes erhalten bleibt. Aus dem

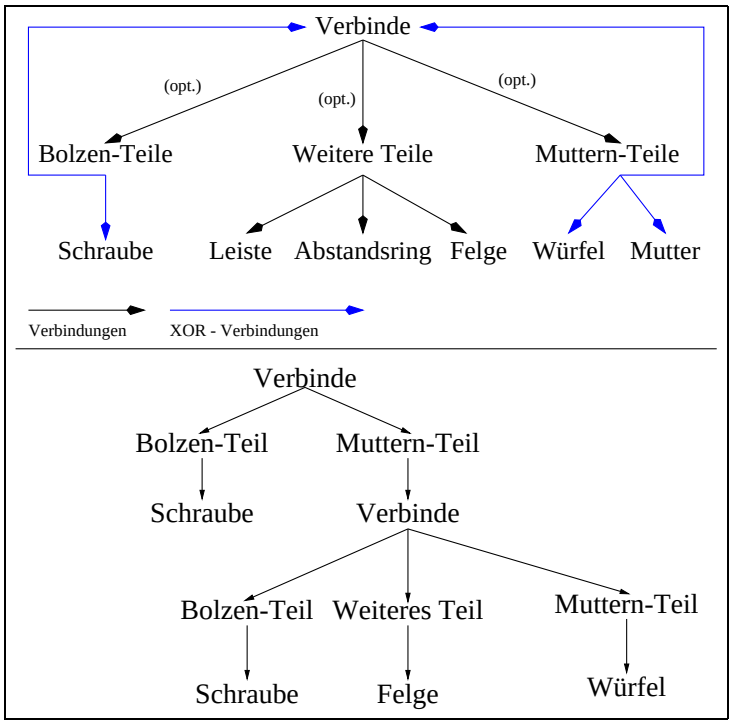


Abbildung 5.4: Ein funktionales Modell als Assembly Tree von Teilen aus dem Baufixszenario. Jedem Bauteil wird eine Funktion zugewiesen, die es während einer Montageoperation übernimmt. So entstehen aus Einzelteilen Aggregate, die wiederum Funktionen zugewiesen bekommen. Eine solche Repräsentation ist nicht eindeutig, mehrere Bäume beschreiben dasselbe Aggregat. Deshalb ist es nicht sinnvoll, zwei Bäume miteinander zu vergleichen.

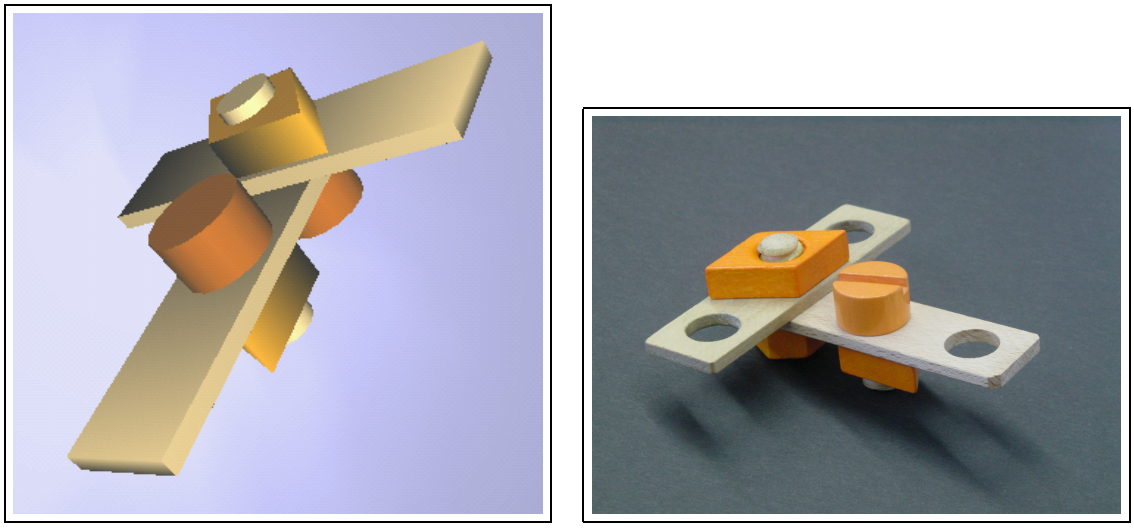
funktionalen Modell und einem Bild des Zielaggregates wird eine Montagesequenz generiert. Die Optimalität der Sequenz hängt von der Reihenfolge der Clusteranalyse im verarbeiteten

Bild ab. Unterschiedliche Bilder können daher zu unterschiedlichen Montagesequenzen mit unterschiedlicher Optimalität führen.

Die wesentlichen Probleme dieses funktionalen Modells liegen zum einen in der Tatsache, daß ein funktionaler Graph nicht eindeutig ist. Damit ist gemeint, daß ein Aggregat durch mehrere funktionale Graphen beschrieben werden kann. Auf Grund dieser Eigenschaft können die Graphen nicht verwendet werden, um zwei Aggregate miteinander zu vergleichen.

5.2 Graphenmodell zur Beschreibung von Baufixaggregaten

Um Montagevorgänge beschreiben zu können, werden im täglichen Leben Bauanleitungen verwendet. Dabei handelt es sich üblicherweise um ikonische und/oder textuelle Beschreibungen des Montageablaufes. Die ikonischen Repräsentationen sind oft mit Pfeilen versehen, um Montagerichtungen anzudeuten (Explosionszeichnungen, siehe als Beispiel (D.S.HONG AND H.S.CHO, 1995)). Die Abfolge von Bildern mit verbindenden Texten veranschaulicht das Fortschreiten innerhalb einer Montagesequenz, Abb. 5.6.
Eine andere, eher technische Art der Beschreibung sind die sogenannten *Assembly trees*, die z.B. als AND/OR-Graphen oder Petrinetze (CAO AND SANDERSON, 1998) (VALLE AND CAMACHO, 1996), (ZHA, 1999), Constraint networks (TUNG AND KARK, 1996), Assembly Graphs (BUKCHIN ET AL., 1997), (TZAFESTAS AND STAMOU, 1997) und Flußdiagramme (D.S.HONG AND H.S.CHO, 1995) modelliert werden können. Der Vorteil insbesondere von AND/OR-Graphen und Petrinetzen ist die Vielzahl an bekannten Suchalgorithmen wie z.B. A^* und (AA^*) (BANDER AND WHITE, 1998), (HART ET AL., 1968), die gewissen Optimalitätskriterien gehorchen. Um Montagesequenzen virtuell im Rechner nachvollziehen zu können,



(a) Simuliertes Baufixaggregat

(b) Reales Baufixaggregat

Abbildung 5.5: Ein Baufix-Aggregat, bestehend aus sechs Einzelbauteilen (zwei Leisten, zwei Schrauben und zwei Muttern). Die Darstellung in (a) wurde aus dem in diesem Abschnitt beschriebenen Objektmodell generiert, um dessen Korrektheit zu überprüfen.

wird eine für den Montagevorgang möglichst prägnante Repräsentation der montierten Teile benötigt. Eine Abfolge solcher Repräsentationen ergibt eine Sequenz, siehe Abb. 5.6. Die in den folgenden Abschnitten beschriebene Objektmodellierung lehnt sich sowohl an das Konzept der Liaison/Connection-Graphen (FAZIO AND WHITNEY, 1987), (JENTSCH AND KADEN, 1984) als auch an das Relationale Baugruppen-Modell (DE MELLO, 1989) u. (MOSEMANN, 2000) an. Dabei gehen zwei oder mehrere unterschiedliche Bauteile Verknüpfungen (*Liaisons*) miteinander ein, die zu einer graphenähnlichen Struktur führen; es werden nur die im Baufixszenario vorkommenden Relationen zwischen Bauteilen berücksichtigt.

Neben einigen anderen Aspekten ist der Hauptverwendungszweck des hier entwickelten Modells der Vergleich zwischen Aggregaten. Ist Aggregat A exakt gleich Aggregat B oder ist A in B enthalten, und wenn ja, wie oft? Dazu wird eine eindeutige Beschreibung benötigt, die einen einfachen Graphenvergleich ermöglicht. Deshalb mußten die oben erwähnten Modelle abgeändert werden. So verwenden die von Mosemann beschriebenen Graphen absolute Bauteilpositionen, des weiteren werden auch zwischen unverbundenen Bauteilen Relationen beschrieben. Diese Punkte erschweren den Vergleich zwischen Aggregaten.
Liaisongraphen beschreiben zwar die Verbindungsstruktur, jedoch nicht genau genug; es fehlen auch jegliche Informationen über die Lagebeziehungen der Bauteile.

5.2.1 Objektmodell zur mentalen Simulation

In dem im weiteren noch zu beschreibenden Objektmodell werden einzelne Bauteile an ihren *Ports* mit Hilfe von *Liaisons* verknüpft. Die Bauteile werden dann in sogenannten Aggregaten zusammengefaßt. Insbesondere ist es möglich, mehrere (> 2) Bauteile an einer *Liaison* miteinander zu verknüpfen. Als Beispiel stelle man sich eine Achse vor, auf die mehrere Zahnräder montiert werden. Ein Zahnrad besitzt nicht nur eine Beziehung zur Achse, auf die es montiert wird, sondern auch zu dem Zahnrad, das vor ihm und das nach ihm auf die Achse gesteckt wird, denn genau diese Reihenfolge ist es, die bei Montage-/Demontageprozessen von Bedeutung ist. Im folgenden werden die einzelnen Größen (Bauteile, Port, Liaison, Aggregat) näher beschrieben.

Bauteile

Jedes Bauteil wird durch die folgenden Größen spezifiziert:

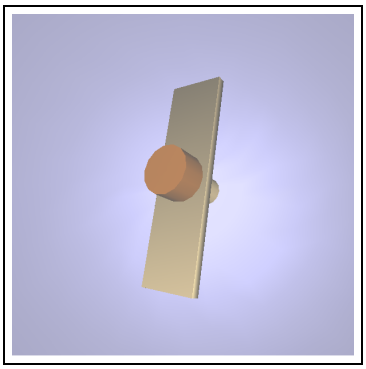
Position $\mathbf{T}$: Jedes Bauteil besitzt ein festzulegendes Koordinatensystem; für die hier verwendeten Baufixteile siehe Abb. 5.8. Die Lage des Bauteilkoordinatensystems bezüglich des umgebenden Koordinatensystems wird durch T beschrieben. Das umgebende Koordinatensystem kann z.B. ein Weltkoordinatensystem sein oder ein Aggregatkoordinatensystem.

Masse $\mathbf{m}$: Die Masse des Bauteils.

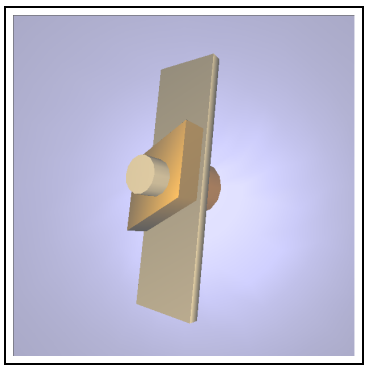
Massenschwerpunkt $\vec{r}$: Der Vektor vom Bauteilkoordinatensystem zum Massenschwerpunkt.

Name: Ein eindeutiger Bezeichner des Bauteils innerhalb eines möglichen Aggregats.

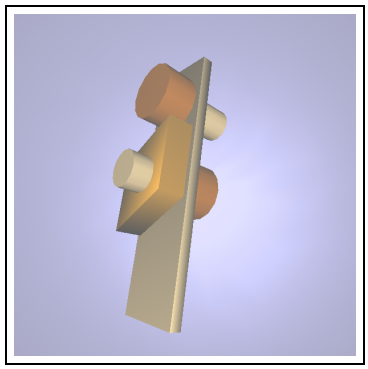
Menge der Verbindungsstellen: $\mathcal{P} = \{\text{Port}_1, \text{Port}_2, \ldots, \text{Port}_p\}$ Ein Bauteil kann nur an einer konkreten Position und auf eine konkrete Weise mit anderen Bauteilen verbunden



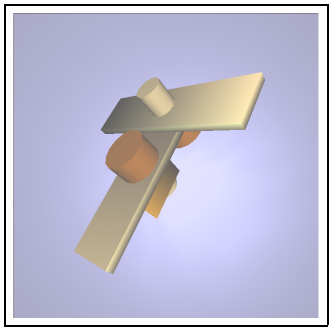
(a) Durch das mittlere Loch einer Dreilochleiste eine orange Schraube stecken



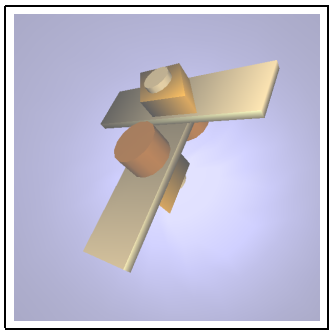
(b) Befestigen der Schraube mit einer Mutter



(c) Eine zweite Schraube von der gegenüberliegenden Seite durch eines der äußeren Löcher der Leiste stecken.



(d) Eine weitere Leiste im 90^0-Winkel auf die Schraube stecken



(e) Festziehen der zweiten Schraube mit einer Mutter.

Abbildung 5.6: *Eine Montagesequenz als Abfolge von Montagezuständen und beschreibenden Texten.*

werden. Die Menge der möglichen Verbindungsstellen ist bauteilspezifisch. Hier sind nur potentielle Verbindungsstellen gemeint. Näheres im Anschluß an dieses Kapitel.

Anzahl der Verbindungsstellen $p = \text{card}(\mathcal{P})$ Die Anzahl der Verbindungsstellen ist bauteilspezifisch, man denke z.B. an ein Puzzle: Eckteile haben nur zwei, Randteile drei und alle anderen vier Verbindungsstellen.

Attribut(e): Unter Attributen sollen allgemein beschreibende Größen eines Bauteils verstanden werden, z.B.:

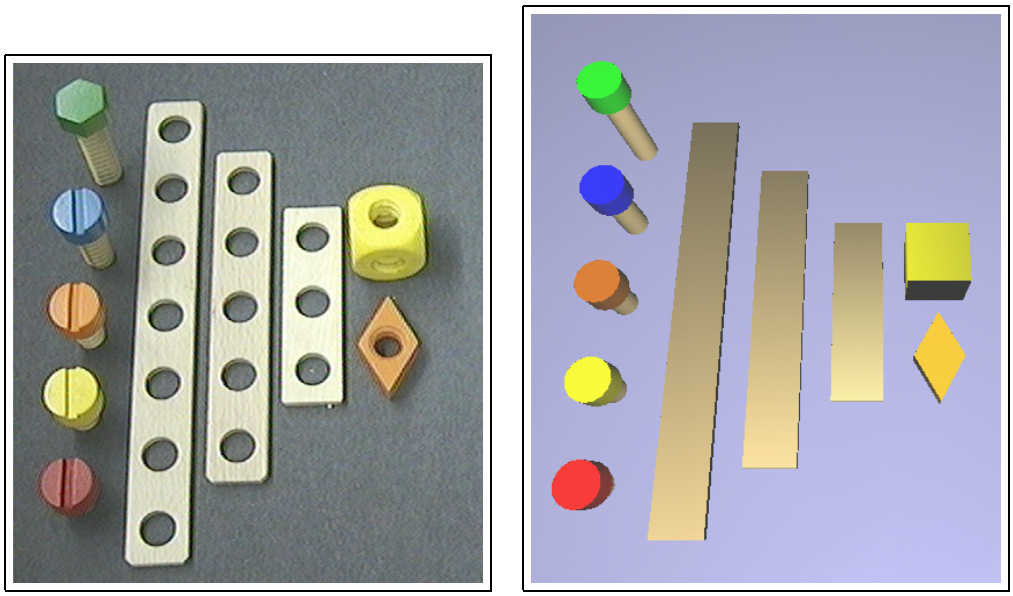
Farbe $f \in \mathcal{F} = \{\text{rot, grün, blau, orange, gelb, weiß,}\dots\}$,

Material $o \in \mathcal{O} = \{\text{Metall, Holz, Pappe, Plastik,}\dots\}$,

...

Typ t: $t \in \mathcal{K}$, wobei K eine Menge von Kategorien ist, z.B.:
$\mathcal{K} = \{$Schrauben, Muttern, Leisten, Ringe ...$\}$.

Symmetriefunktion $S_i : \mathcal{T} \rightarrow \mathbb{R}$: Jedes Bauteil besitzt Symmetrieeigenschaften. Für jedes Bauteil wird die Symmetrie in Form eines Winkels um die (x,y)- und z-Achse des Bauteils angegeben. S_i ist eine Abbildung, die dem Bauteil des Typs $t \in T$ einen Winkel ω um die i-te Achse zuweist. Die Rotation des Bauteils um diese Achse und den entsprechenden Winkel führt zu einer Lage des Bauteils, die symmetrisch zur vorherigen ist. Für eine konkrete Implementation siehe List. B.2.1.1.



(a) reale Baufixteile

(b) Visualisierung der virtuellen Baufixteile

Abbildung 5.7: Die folgenden Baufixteile wurden mit der obigen Beschreibung modelliert: Schrauben (grün, blau, orange, gelb und rot), Leisten (7-, 5- und 3-Loch), Würfel und *Rauten*muttern. Da die Simulation der Bauteile nicht zum primären Zweck des Objektmodells gehört, wurden nur die Umrisse modelliert.

Abb. 5.7 zeigt eine Visualisierung von Baufixteilen, die mit Hilfe des obigen Objektmodells modelliert wurden. Die Abmessungen der Bauteile wurden einer statistischen Untersuchung entnommen, siehe Anhang A.

Verbindungsstellen (*Ports*)

Wie bereits erwähnt, bezieht sich das Objektmodell auf Bauteile mit konkreten Verbindungsstellen. Werden die Ports zweier Bauteile miteinander verbunden, entsteht ein Graph, näheres dazu weiter unten. Jeder Port verfügt über eine Reihe von Parametern, die für die Montage benötigt werden und gewisse Plausibilitätsüberprüfungen ermöglichen:

Offset $\mathbf{O}$: Eine Transformation vom Bauteilkoordinatensystem zur Portposition. Die Lage der Portkoordinatensysteme wird in Abb. 5.9 gezeigt.

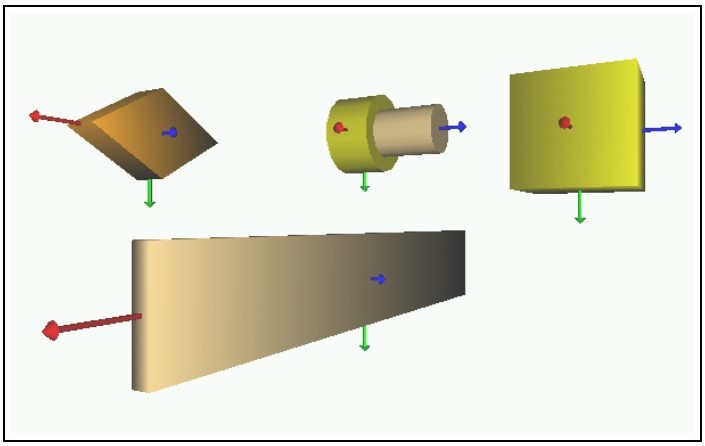


Abbildung 5.8: Lage der Bauteilkoordinatensysteme T, X-Achse= rot, Y-Achse = grün, Z-Achse = blau.

Typ $\mathbf{u}$: $u \in \mathcal{U} = \{$nehmend, gebend$\}$ Verbindungsstellen können entweder gebend (Gewinde, Bolzen,...) oder nehmend (Mutter, Loch, Rinne, ...) sein. Diese Information wird nicht nur zur Überprüfung von Plausibilitätsbedingungen (Schraube auf Schraube ist nicht möglich), sondern auch zur Interpretation der Tiefeninformation l und zur Vergabe einer funktionalen Rolle im Sinne von (BAUCKHAGE ET AL., 1999) verwendet. Siehe List. B.3.2.1 als Beispiel für den Gebrauch von s bei der Montage.

Tiefe $\mathbf{l}$: Sie wird benötigt, um die Position der Bauteile bei einer Montage berechnen zu können und um abzuschätzen, ob eine Montage überhaupt möglich ist. Die Angabe einer solchen Tiefen-, Längeninformation macht nicht nur bei gebenden Ports, also Schrauben, einen Sinn, sondern auch bei nehmenden Verbindungen. Hier ersetzt die Tiefen- bzw. Längenangabe der Verbindungsstücke die Berechnung der genauen Montageinformation bzw. der Abstände, die beim Anfahren der Montagepositionen eingehalten werden müssen. Sie dient also auch zur Trajektorienplanung.

Nummer $\mathbf{n}$: Laufende Nummer des Ports am Bauteil.

Liaison: Verweis auf eine Liaison[3], falls vorhanden. Auf diesem Weg wird ein Graph, bestehend aus *Ports* mehrerer Bauteile, die alle auf dieselbe Liaison zeigen, gebildet.

Besitzer: Verweis auf das Bauteil, zu dem der Port gehört (Doppelverzeigerung).

Die Lage und Orientierung der Ports an den realisierten Baufixteilen ist festgelegt, sie läßt sich aus Abb. 5.9 entnehmen.

Verbindungen zwischen Bauteilen (*Liaisons*)

Liaisons sind konkrete Verbindungen zwischen einem oder mehreren Ports, die wiederum auf ein konkretes Bauteil verweisen. Folgende Größen beschreiben eine Liaison:

[3] Abschnitt 5.2.1.

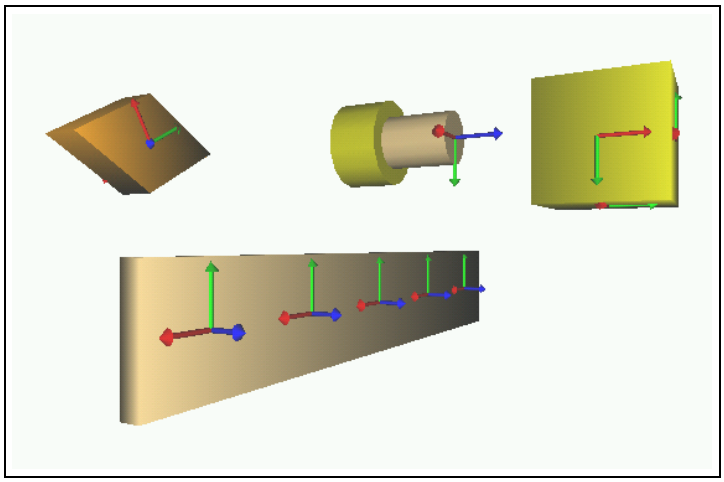


Abbildung 5.9: Lage der Portkoordinatensysteme O.
X-Achse = rot, Y-Achse = grün, Z-Achse = blau. Bei gebenden Ports zeigt das Koordinatensystem vom Bauteil weg, bei nehmenden auf das Bauteil hin.

$\mathcal{V}$**:** Menge der Ports, die miteinander verbunden sind. Durch Definition einer Funktion r läßt sich auf $\mathcal{V}$ eine Ordnung definieren, wodurch Beziehungen der Bauteile innerhalb der Liaison beschrieben werden können.

$r : \mathcal{V} \rightarrow \mathbb{N}$**:** Eine bijektive Funktion r, die jedem Element aus $\mathcal{V}$ genau eine Ordnungszahl zuweist.

card$(\mathcal{V})$**:** Die Anzahl der Elemente in $\mathcal{V}$.

j**:** Die Stabilitätseigenschaft $j \in \{\text{stabil, instabil}\}$ beschreibt, ob eine Liaison aufgrund der Schwerkraft zerfallen kann oder nicht.

Liaisons sind also die Bindeglieder zwischen den Bauteilen und stellen damit automatisch einen Artikulationspunkt dar. Aus den Daten der Einzelbauteile und der Lage der Ports läßt sich die neue Lage der montierten Bauteile berechnen. Dabei ist zu beachten, daß die Bauteile sich nicht durchdringen. Sei $\mathcal{E}$ eine Menge von Bauteilen, dann muß gelten:

$$\forall i, j (i \neq j) \Rightarrow \mathcal{E}_i \cap^* \mathcal{E}_j = \emptyset \tag{5.6}$$

wobei $\cap^*$ die „regularized intersections“ (REQUICHA, 1977), (REQUICHA, 1980) zweier Bauteile beschreibt.
Gegeben seien zwei Bauteile a und b; b soll an a angefügt werden. Die Position (und Orientierung) von b wird neu berechnet unter Verwendung von Bauteil a, den spezifizierten Ports und einem Winkel θ. Die Orientierung von b wird der von a angeglichen, dabei bleibt im Baufixszenario jedoch ein Freiheitsgrad offen. Dieser wird durch den Winkel θ festgelegt. Es sind:

P_G = Position des Bauteils mit dem *gebenden* Port

P_T = Position des Bauteils mit dem *nehmenden* Port

O_G = Offset innerhalb des Bauteils zum *gebenden Port*

O_T = Offset innerhalb des Bauteils zum *nehmenden Port*

C = Matrix, die eine Rotation von θ Grad um die $z-$Achse und eine Translation entlang der $z-$Achse um die noch freie Länge l des Bauteils mit dem *gebenden* Port beschreibt.

$$C = \mathrm{Rot}(\theta, z) \cdot \mathrm{Trans}(l, z) \tag{5.7}$$

Daraus läßt sich folgende Positionsgleichung aufstellen:

$$P_G \cdot O_G \cdot C = P_T \cdot O_T, \tag{5.8}$$

woraus sich die Positionen für P_G bzw. P_T wie folgt ableiten:

$$P_T = P_G \cdot O_G \cdot C \cdot O_T^{-1} \tag{5.9}$$

$$P_G = P_T \cdot O_T \cdot C^{-1} \cdot O_G^{-1} \tag{5.10}$$

Für den Fall, daß beide Ports nehmende Ports sind, gilt Gleichung 5.10 ohne den Translationsanteil in C, denn hier wird nichts aufgesteckt, die Ports liegen flach aufeinander.

Um die Position von b entsprechend zu modifizieren, muß festgestellt werden, welche Rolle (gebend oder nehmend) die Ports besitzen, die miteinander verbunden werden sollen, siehe List. B.3.2.1. Zwischen a u. b wird zum Aufbau eines Graphen eine *Liaison* benötigt. Entweder besitzt eines der Bauteile bereits eine *Liaison* (mit einem anderen Bauteil) oder es muß eine neue *Liaison* angelegt werden. Jeder neu in eine *Liaison* aufzunehmende Port p_{neu} bekommt eine neue Ordnungszahl:

$$\mathcal{V}_{\mathrm{neu}} = \mathcal{V}_{\mathrm{alt}} \cup p_{\mathrm{neu}} \tag{5.11}$$

$$p_{\mathrm{max}} = \max_{p \in \mathcal{V}_{alt}} r(p) \tag{5.12}$$

$$p_{\mathrm{min}} = \min_{p \in \mathcal{V}_{alt}} r(p) \tag{5.13}$$

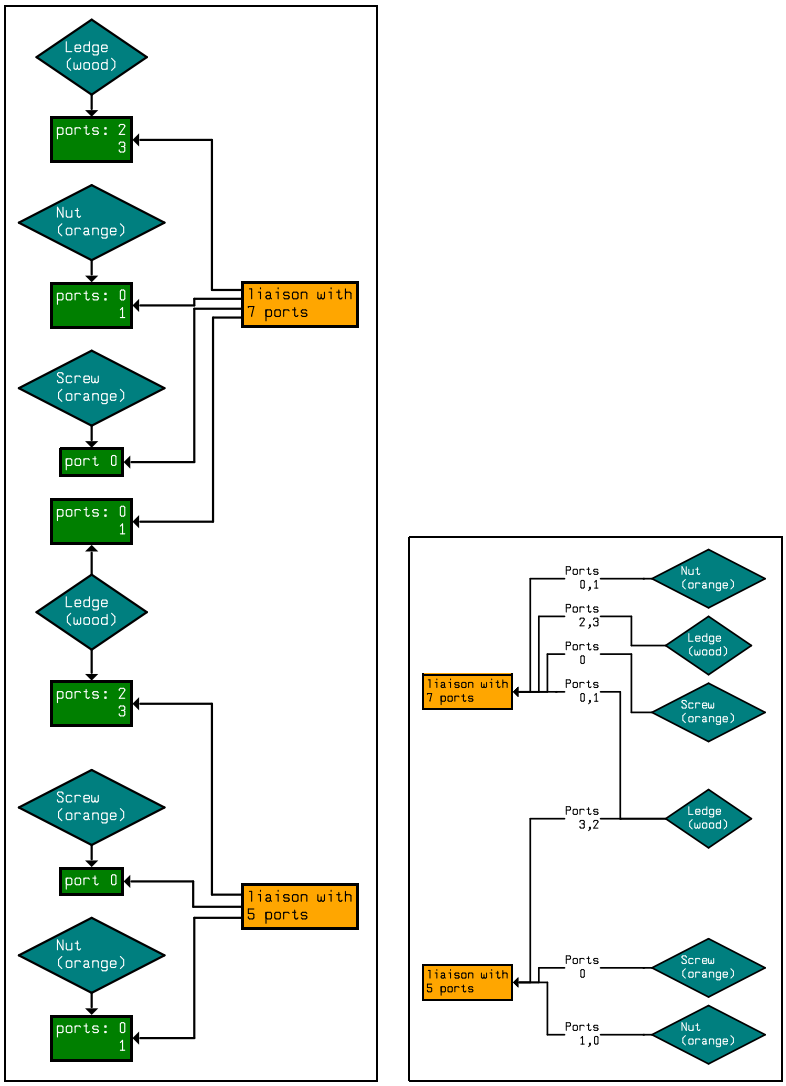
$$r(p_{\mathrm{neu}}) = \begin{cases} 0 & \text{wenn } \mathcal{V}_{\mathrm{alt}} = \emptyset \\ r(p_{\mathrm{max}}) + 1 & \text{sonst} \end{cases} \tag{5.14}$$

Demontageoperationen können nur mit p_{max} und p_{min} durchgeführt werden, da sie die äußersten Ports einer *Liaison* sind. Mit der Einführung einer solchen virtuellen Größe – *Liaisons* sind physikalisch nicht präsent, und theoretisch läßt sich auch ein Objektmodell spezifizieren, das ohne *Liaisons* auskommt – ist es auf einfachem Wege möglich, Mehrfachverbindungen, wie sie z.B. bei Schrauben häufig auftreten, zu modellieren. Um einen Eindruck von dieser Struktur zu bekommen, betrachte man Abb. 5.10(a) u. Abb. 5.10(b). Sie visualisieren den Graphen der das in Abb. 5.5 gezeigten Aggregat beschreibt.

Aggregat(e)

Aggregate werden hier als eine Menge $\mathcal{E}$ von Bauteilen verstanden, die mittels einer Menge $\mathcal{L}$ von Liaisons verbunden sind. Damit ist auch eine nicht stabile Verbindung zwischen zwei Bauteilen erlaubt. Im Detail wird ein Aggregat durch folgende Größen beschrieben:

$\mathcal{E}$**:** Menge aller Bauteile des Aggregats.



(a) Die tatsächliche Graphenstruktur. Sie besteht aus: Bauteilen, Ports und Liaisons.

(b) Eine vereinfachte Darstellung. Sie ist kompakter, trägt aber denselben Informationsgehalt.

Abbildung 5.10: Die Struktur eines Aggregates läßt sich als Graph verstehen. Die hier dargestellten Beispiele sind Beschreibungen des in Abb. 5.5 gezeigten Aggregates. Die orangen Kästchen repräsentieren Liaisons zwischen mehreren Ports, die rautenförmigen Kästchen symbolisieren die einzelnen Bauteile und die grünen Kästchen in Abbildung (a) stehen für die Ports, an denen die einzelnen Teile verbunden werden. In der vereinfachten Darstellung werden die Portnummern an die Verbindungen zwischen Liaison und Bauteil geschrieben.
Es tritt häufig auf, daß bei der Montage nicht nur ein, sondern gleich zwei sich gegenüberliegende Ports eines Bauteils benutzt/belegt werden, z.B. bei Leisten. Um diesem Effekt Rechnung zu tragen, wurden die jeweiligen Ports zusammengefaßt.

$\mathcal{L}$**:** Menge aller Liaisons, mit denen die Bauteile in $\mathcal{E}$ verbunden sind.

card$(\mathcal{E})$**:** Anzahl der Bauteile in $\mathcal{E}$.

card$(\mathcal{L})$**:** Anzahl der Liaisons in $\mathcal{L}$.

D**:** Transformation vom umgebenden Weltkoordinatensystem zum Aggregatnullpunkt.

d**:** $d \in \mathcal{E}$ ist das Bauteil, dessen Nullpunkt mit dem Nullpunkt des Aggregats übereinstimmt.

Bezeichner: Unter einem Bezeichner soll ein Element $b \in \mathcal{B}$ aus einer Menge von Kategorien wie z.B. {Motor, Fahrgestell, Leitwerk, Mast, Ruder, ...} verstanden werden. Damit ist es möglich, Aggregate über ihren Bezeichner zu benennen und aus einer Menge von Aggregaten auszuwählen. Allerdings ergibt sich das Problem, wie man Aggregate behandelt, für die es keinen Bezeichner in der Menge der Kategorien gibt. Entweder die Menge wird in einem solchen Falle dynamisch erweitert oder man definiert eine Art Plazebo-Kategorie, die auf alle unbekannten Aggregate zutrifft. Wie sich eine umfassende Konzeptualisierung mechanischer Baugruppen realisieren läßt, kann man in (HOFFHENKE AND WACHSMUTH, 1997) lesen.

Die durch Bauteile und Liaisons aufgespannte Struktur zeigt Abb. 5.10. Jedes Aggregat definiert ein eigenes Koordinatensystem. Im Gegensatz zum Bauteil kann man dessen Lage aber nicht festlegen, es gibt mehrere Möglichkeiten:

- **Frei definierbar:** Die Lage der Bauteile und die des Aggregatkoordinatensystems D haben prinzipiell nichts miteinander zu tun. Eine Drehung des Aggregatkoordinatensystems kann zu einer Translation und Rotation aller Bauteile im Raum führen. Zwei Instanzen desselben Aggregats können unterschiedliche Nullpunkte besitzen. Die freie Wahl erlaubt es, eine intrinsische Aggregatorientierung einzuführen. Damit ist gemeint, daß z.B. zwei Kommunikatoren, die sich über ein und dasselbe Aggregat unterhalten, eine gemeinshame Bedeutung für Ortsangaben wie z.B. „vorne“ verwenden, unabhängig davon, wo das Aggregat bezüglich der Kommunikatoren im Raum liegt. Sie müssen sich lediglich auf eine gemeinsame Interpretation der Koordinatensysteme einigen.[4]

- **Im Schwerpunkt:** Damit ist die Position des Koordinatensystems eindeutig festgelegt, nicht aber die Orientierung. Alle Instanzen haben dasselbe Koordinatensystem. Es ändert sich, wenn ein neues Bauteil angebaut wird. Für die Orientierung bieten sich die Trägheitsachen an.

  Im Hinblick auf die Interpretation als intrinsisches Aggregat im Sinne zweier Kommunikatoren ist dies zwar eine mögliche Festlegung, für menschliche Kommunikatoren allerdings eher unüblich, da sie sich von anderen Kriterien wie z.B. Funktionsweise oder Form leiten lassen.

- **In einem Bauteil:** Das Aggregatkoordinatensystem korrespondiert immer mit einem der Bauteilkoordinatensysteme. Vom Programmierer kann das entsprechende Bauteil festgelegt werden. Sonst wird das erste ins Aggregat aufgenommene Bauteil gewählt. Nachteil dieser Methode: Gleichartige Aggregate können verschiedene Koordinatensysteme D besitzen; Vorteil: Das Aggregat kann leicht um eins seiner Bauteile rotiert

[4] Die Wahl von intrinsischen und extrinsischen Koordinaten im Sprachgebrauch ist allgemein vom Kulturkreis abhängig.

werden. Bei der Manipulation von Aggregaten ist ein solches Bezugssystem von besonderem Interesse; man denke z.B. an Schrauboperationen oder an das Übergeben an einen anderen Roboter.

In der hier verwendeten Implementation wird sowohl der ersten als auch der dritten Ansatz verwendet.
Die Algorithmen, die auf den Aggregaten durchgeführt werden, erlauben es, Montageoperationen zu simulieren, die tatsächliche Durchführbarkeit zu überprüfen, Schwerpunktberechnungen durchzuführen, die Gleichheit von Aggregaten und das Enthaltensein eines Aggregats in einem anderen zu testen.
Im Hinblick auf das Sequenzlernverfahren sind der $=$ - und der $\subset$ - Operator von besonderem Interesse.
Ein Aggregat a ist gleich einem Aggregat b, wenn:

(5.15) $$\mathcal{E}_a = \mathcal{E}_b \ , \ \mathcal{L}_a = \mathcal{L}_b$$

Ein Aggregat a ist Teil eines Aggregats b, wenn:

(5.16) $$\mathcal{E}_a \subseteq \mathcal{E}_b \ , \ \mathcal{L}_a \subseteq \mathcal{L}_b$$

Sei $\mathcal{E}_a$ die Menge aller Bauteile des Aggregats a. Die offensichtlichen Bedingungen für diese Beziehungen sind :

$$\mathcal{E}_a = \mathcal{E}_b, \text{ wenn } \text{card}(\mathcal{E}_a) = \text{card}(\mathcal{E}_b) = 0$$
$$\mathcal{E}_a \neq \mathcal{E}_b, \text{ wenn } \text{card}(\mathcal{E}_a) \neq \text{card}(\mathcal{E}_b)$$
$$\mathcal{E}_a \subseteq \mathcal{E}_b, \text{ wenn } \text{card}(\mathcal{E}_a) \leq \text{card}(\mathcal{E}_b)$$

$$\mathcal{L}_a = \mathcal{L}_b, \text{ wenn } \text{card}(\mathcal{L}_a) = \text{card}(\mathcal{L}_b) = 0$$
$$\mathcal{L}_a \neq \mathcal{L}_b, \text{ wenn } \text{card}(\mathcal{L}_a) \neq \text{card}(\mathcal{L}_b)$$
$$\mathcal{L}_a \subseteq \mathcal{L}_b, \text{ wenn } \text{card}(\mathcal{L}_a) \leq \text{card}(\mathcal{L}_b)$$

Weiterhin gilt:

(5.17) $$\mathcal{E}_a = \mathcal{E}_b \ \Leftrightarrow \ \forall x (x \in \mathcal{E}_a \ \Leftrightarrow \ x \in \mathcal{E}_b)$$

Sei T_x die relative Position des Bauteils x bezüglich des Aggregats, zu dem x gehört und

(5.18) $$f(x,y) = T_x^{-1} * T_y \text{ mit } x \in \mathcal{E}_a \text{ und } y \in \mathcal{E}_b;$$

(5.19) wenn $p_a = p_b$ und $q_a = q_b$, dann muß $f(p_a, p_b) = f(q_a, q_b)$ sein.

Damit ist gemeint, daß zu jedem Element p_a des Aggregats a genau ein äquivalentes Element p_b im Aggregat b existiert und die relative Raumbeziehung zwischen allen gleichen Paaren identisch ist. Da es sich um die Struktur eines realen Baufixaggregates handelt, kann ein Teil nicht zweimal im Graphen auftreten, denn das hieße, daß beide Teile denselben Raum einnehmen. Damit ist Gleichung 5.17 eine allgemeine Bedingung für Graphen, die Aggregate beschreiben.
Zwei Bauteile zweier Aggregate sind äquivalent, wenn sämtliche Größen wie Typ, Farbe,

Masse, ..., die die Bauteile betreffen, gleich sind und die unmittelbare Graphenumgebung übereinstimmt, also die Anzahl und Struktur der mit den Ports der Bauteile verknüpften Liaisons identisch ist. Ausgenommen ist die Position der Bauteile, da sie sich auf ein Aggregatkoordinatensystem bezieht, das nicht zwangsweise gleich sein muß. Damit zwei Aggregate gleich sind, müssen alle Bauteile äquivalent und die relative Lage aller gleichartigen Bauteilpaare innerhalb des Aggregats identisch sein.

Beispiel: Angenommen, zwei einfache Aggregate a u. b, bestehend aus einem Würfel (p_1) und einer Schraube (p_2), sollen auf Gleichheit überprüft werden; der einzige Unterschied zwischen den Aggregaten ist das Bauteil c, auf das sich das Aggregatkoordinatensystem C bezieht (siehe Aggregatdefinition). Im Falle von a bezieht sich D^a auf den Würfel p_1^a, im Falle von b (D^b) auf die Schraube p_2^b. Das heißt, die Bauteiltransformation $T_(p_1^a) = I$ und $T_(p_2^b) = I$, da sie im jeweiligen Nullpunkt des Aggregats liegen, während $T_(p_2^a) \neq I$ und $T_(p_1^b) \neq I$. Es muß aber für alle Paare (p^a, p^b) von äquivalenten Bauteilen aus a u. b, also in diesem Beispiel $((p_1^a, p_1^b), (p_2^a, p_2^b))$, die Transformation $T_{p^a}^{-1} * T_{p^b}$ gleich sein. Also

$$T_{p_1^a}^{-1} * T_{p_1^b} = T_{p_2^a}^{-1} * T_{p_2^b} = (D^a)^{-1} * D^b \tag{5.20}$$

Weil die relativen Lagebeziehungen der Bauteile überprüft werden, reicht es aus, die lokalen Verknüpfungsbeziehungen der Bauteile untereinander zu prüfen.

Für den $\subset$ – Operator gilt dasselbe wie für den $=$ – Operator mit der Einschränkung, daß die Anzahl der Bauteile in $\mathcal{E}_a \leq \mathcal{E}_b$ und die durch L_a beschriebene Verzeigerung nur eine Teilmenge von L_b sein muß.

Der für das hier beschriebene Aggregatmodell entworfene Algorithmus, mit dessen Hilfe festgestellt werden kann, wie oft ein Aggregat a in einem Aggregat b enthalten ist, wird in List. B.3.3.1 Anhang B.3.3 im Detail gezeigt. Hier die prinzipielle Beschreibung des ihm zugrundeliegenden iterativen Verfahrens: Festgestellt werden soll, wie oft a in b enthalten ist.

① Auswählen eines Bauteils p_a aus a, üblicherweise das erste im Aggregat gespeicherte.

② Auswählen eines Bauteils p_b aus b unter Verwendung von a und einer Liste L bereits analysierter Bauteile. p_b soll denselben Bauteiltyp wie p_a besitzen.

③ Berechnung der Differenztransformation zwischen p_a und p_b. $D_1 = T_{p_a}^{-1} * T_{p_b}$. Damit ist die initiale Differenztransformation berechnet, sie muß für alle übereinstimmenden Bauteilkombinationen aus a u. b gelten.

④ Durchlaufe alle Bauteile in a, beginnend bei dem zweiten und suche das dazu passende Aggregat in b, wiederum unter Verwendung der Liste L.

⑤ Wenn zu jedem Bauteil ein passender Partner gefunden werden kann und alle Differenztransformationen gleich D_1 sind, ist a mindestens einmal in b enthalten. Alle Bauteilzuordnungen werden in L vermerkt, damit sie bei einem zweiten Durchlauf nicht mehr berücksichtigt werden. Anschließend beginnt man wieder bei (1).

⑥ Wenn mindestens zu einem Bauteil in a kein passendes Bauteil in b gefunden werden konnte, bzw. deren Differenztransformation nicht mit D_1 übereinstimmt, heißt das noch nicht, daß keine Übereinstimmung existiert. Es könnte ja sein, daß die initiale Zuordnung zwischen p_a u. p_b falsch war, das heißt, das D_1 eine Differenztransformation zwischen zwei verschiedenen Bauteilen ist. In diesem Falle zurück zu (2).

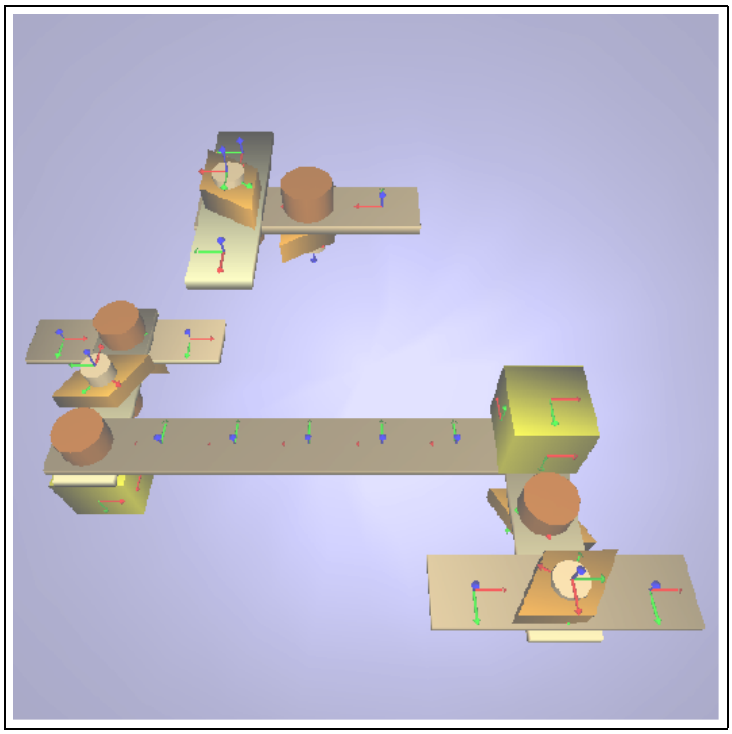


Abbildung 5.11: Zwei Aggregate; das obere ist zweimal in dem unteren enthalten.

⑦ Läßt sich in (2) keine weitere Bauteilzuordnung finden, ist a nicht bzw. kein weiteres Mal in b enthalten.

Es bleibt festzulegen, wann zwei Bauteile einander zugeordnet werden können. Dafür müssen die Attribute (Typ u. Farbe) sowie die Anzahl der Ports übereinstimmen, und die lokalen Verknüpfungen des Bauteils aus a müssen Teilmenge der lokalen Verknüpfungen des Bauteils in b sein. Dazu werden alle Ports abgesucht und die an ihnen befindlichen Liaisons analysiert. Der Berechnungsaufwand dieses Verfahrens ist zwar sehr hoch, bei den betrachteten Aggregaten ist die Anzahl gleicher Bauteile aber gering, was die Vorgehensweise vertretbar macht.

Mit Hilfe dieses iterativen Algorithmus ist es möglich, Teilaggregate zu identifizieren. Dies wiederum ermöglicht es festzustellen, ob man in einem Montageprozeß von dem zu bauenden Zielaggregat abweicht.

6

Dynamische Zustands-/Aktionsgraphen

6.1 Q-Tabellen

Üblicherweise werden die gelernten Werte der Funktion $Q(s, a)$ in einer Q-Tabelle bzw. Adjazenzmatrix gespeichert. In ihr werden für alle Kombinationen aus Zuständen und Aktionen entsprechende Q-Werte abgelegt. Tab. 6.1 ist ein Beispiel dafür. Sie bezieht sich auf das wohl prominenteste Beispiel für Q-Lernen: die *Grid-World*. Dabei wird eine Ebene in ein diskretes Raster unterteilt. Jeder Rasterpunkt entspricht einem Zustand. Gesucht wird ein Weg von einem beliebigen Startpunkt in der Welt zu einem festen Zielpunkt. Abb. 6.1 zeigt die zu Tab. 6.1 gehörende Raumdiskretisierung. Der Zielzustand (1,3) ist mit GS markiert. Als Aktionen sind die folgenden Bewegungen vorgesehen: rechts, links, oben oder unten. Wie man bereits an diesem einfachen Beispiel sehen kann, lassen sich 15 der 32 Kombinationen a priori ausschließen, da sie in nicht existente Zustände führen, bzw. vom Zielzustand wegführen.

	(1,1)	(1,2)	(1,3)	(1,4)	(2,1)	(2,2)	(2,3)	(2,4)
hoch	-	-	-	-	81	90	100	90
runter	72	81	-	81	-	-	-	-
links	-	81	-	100	-	72	81	90
rechts	90	100	-	-	81	90	81	-

Tabelle 6.1: Beispiel für eine Q-Tabelle, hier für eine 4×2 *Grid-World*

Dennoch werden von der Q-Tabelle 32(28) Speicherplätze belegt. Da die gesamte Tabelle im Speicher des Rechners Platz finden muß (KAELBLING AND LITTMAN, 1996), entsteht bei komplexen diskreten Systemen, die mit Q-Tabellen arbeiten, ein enormer Speicherbedarf noch bevor ein einziger Lernschritt durchgeführt werden kann. Die Argumentation für eine solche Vorgehensweise ist, daß das System nur dann vollständig gelernt ist, wenn zu allen Zuständen ein richtiger Q-Wert existiert.

Geht man jedoch von sehr komplexen Systemen aus, so ist an ein vollständiges Lernen der Tabelle gar nicht zu denken. Es muß vielleicht auch nicht immer die beste Lösung, bzw.

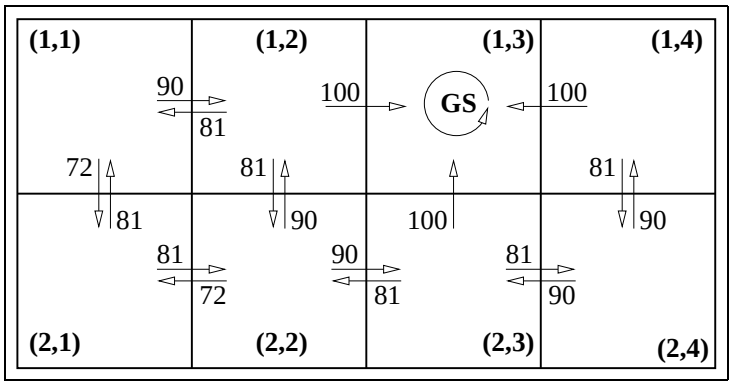


Abbildung 6.1: 4×2 *Grid-World* mit den entsprechenden Werten für $Q(s, a)$

es müssen nicht immer alle Lösungen gefunden werden, unter Umständen reicht schon irgendeine Lösung. Wenn es zwei Wege gibt, von denen der eine optimaler als der andere ist, muß man sich die suboptimale Lösung nicht mehr merken, da man sowieso immer den optimaleren Weg wählen würde. Im Falle einer Q-Tabelle ist das jedoch egal, da man durch Löschen der Q-Werte keinen Speicherplatz gewinnt. Ließe sich der Speicher zum Beschreiben des suboptimalen Weges wieder freigeben, könnte viel Speicherplatz gespart werden.

Viele Zustände lassen sich möglicherweise bereits durch Modellwissen ausschließen wie im Grid-World Beispiel Abb. 6.1. Allgemein spricht man davon, daß die Aktionen zustandsabhängig sind[1]. In der Summe können diese Effekte den benötigten Speicherbedarf signifikant senken, wenn man nicht mit einer statischen Tabelle, sondern mit einem dynamischen Graphen arbeiten würde.

6.2 Definition eines Zustands-/Aktionsgraphen

Das Konzept der Markovschen Entscheidungsprozesse (MDP) legt es nahe, für die Repräsentation der Zustände und Aktionen einen Graphen zu wählen. Dabei werden Zustände als Knoten und Aktionen als Kanten modelliert, siehe z.B. Abb. 6.2. Werden die Q-Werte in den Kanten abgespeichert, ist man nicht mehr gezwungen, den gesamten, theoretisch möglichen Graphen von Beginn an im Speicher aufzubauen. Man führt nur Zustände ein, die das System tatsächlich betritt. In der Lernphase wächst der Graph und mit ihm der benötigte Speicherbedarf. Weiterhin ist es bei einer solchen Darstellung möglich festzustellen, in welchen Folgezustand das System übergehen wird, wenn eine bestimmte Aktion ausgeführt wird. Das ist eine Information, die in einer Q-Tabelle völlig fehlt. Damit lassen sich auch weiter in der Zukunft liegende Belohnungen berücksichtigen.

6.2.1 Initialisierung

Wie jedoch soll man zu Beginn, wenn der letztendliche Graph noch nicht existiert, die ebenfalls nicht existenten Q-Werte behandeln, die zur Auswahl der nächsten Aktion benötigt werden (Gleichung 7.2). Im Falle der Q-Tabelle werden alle Einträge vor der Lernphase initialisiert, häufig mit Null, gelegentlich auch mit Zufallsgrößen.

[1] Peter Stone behandelt dieses Problem in (STONE, 1998) durch Einführen eines Feature-Spaces.

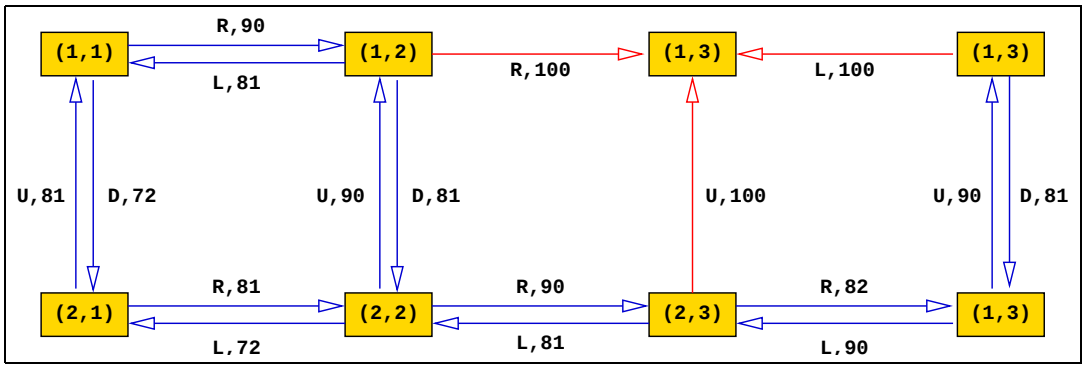


Abbildung 6.2: Zustandsgraph für das in Abb. 6.1 gezeigte *Grid-World* Beispiel. Zustände sind mit orange hinterlegt, Kanten symbolisieren Aktionen. Die Aktionen sind mit U=hoch, D=runter, R=rechts und L=links bezeichnet.

Im Falle eines Graphen kann man die initialen Werte nicht abspeichern, sondern muß sie berechnen. Die einfachste Lösung, die auch in den für diese Arbeit entwickelten Programmen benutzt wird, ist die Wahl einer konstanten Größe, also ebenfalls Null. Es ist aber auch denkbar, andere oder zufällige Werte zu berechnen. Im letzteren Fall muß aus den Parametern der entsprechenden Aktion und dem momentanen Zustand eine Pseudozufallszahl so generiert werden, daß bei mehrmaligem Berechnen immer dasselbe Ergebnis gebildet wird, bei unterschiedlichen Aktionsparametern jedoch unterschiedliche Werte entstehen.

6.2.2 Aktionen als Knoten

In manchen Umgebungen besteht die Möglichkeit, daß eine Aktion a, die in einem Zustand s ausgeführt wird, zu unterschiedlichen Zeiten in unterschiedliche Zielzustände führt. Werden die Aktionen als Kanten modelliert, wie es der MDP nahelegt, gibt es keine Möglichkeit, diesem Sachverhalt Rechnung zu tragen (Abb. 6.3). Deshalb ist es in solchen Umgebungen besser, auch Aktionen als Knoten eines neuen Typs zu verstehen. Werden Zustandsknoten und Aktionsknoten in einem (bipartiter) Graphen zusammengebracht, entsteht ein sogenannter Zustands-/Aktionsgraph. Für das Grid-World Beispiel ergibt sich dann die in Abb. 6.4 dargestellte Struktur, die allerdings gegenüber Abb. 6.2 keine Vorteile bietet, da die Grid-World eine deterministische Umgebung ist. Im Vergleich zur Q-Tabelle werden aber nur die 16 tatsächlich durchführbaren Aktionen, respektive Q-Werte, gespeichert, dazu kommen noch die sechs Zustände. Abb. 6.5 zeigt ein anderes, hypothetisches Beispiel, jedoch mit nichtdeterministischen Zustandsübergängen. In einem solchen Graphen sind die beiden Knoten-Typen auf wohldefinierte Weise miteinander zu verbinden.

Ein Zustands-/Aktionsgraph läßt sich durch folgende Größen beschreiben:

6.2.3 Definition

Ein Zustands-/Aktionsgraph ist ein Quadrupel $G = \langle \mathcal{S}, \mathcal{A}, \mathcal{V}_s, \mathcal{V}_a \rangle$, das durch die folgenden Mengen beschrieben wird:

$\mathcal{S}$ ist die Menge der Zustandsknoten. Jeder Zustandsknoten ist ein Vektor, der eine bestimmte Sicht der Welt repräsentiert.

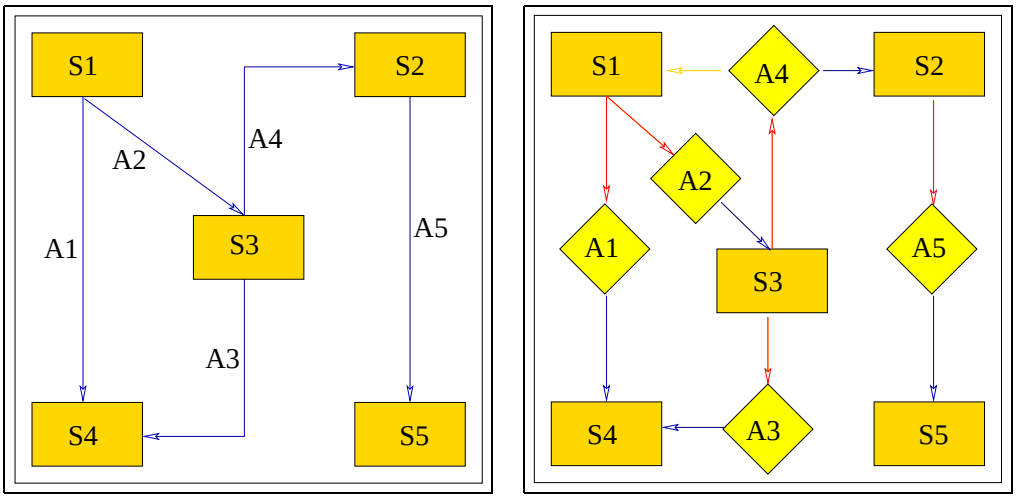


Abbildung 6.3: Es gibt zwei Möglichkeiten, Aktionen zu modellieren: als Kanten (a) oder als Knoten (b) Nur die rechte Darstellung erlaubt es, nichtdeterministische Zustandsübergänge (A4) korrekt zu behandeln.

$\mathcal{A}$ ist die Menge der Aktionsknoten. Ein Aktionsknoten ist ebenfalls ein Vektor. Er besteht aus Größen, die die Aktion beschreiben. Zusätzlich werden Informationen über den Q-Wert ($Q(s,a)$) abgelegt.

$\mathcal{V}_s$ ist die Menge der gerichteten Kanten $\mathcal{V}_s \subset \mathcal{S} \times \mathcal{A}$, die von Zuständen ausgehen und auf Aktionen zeigen. Die Kanten sind geordnete Paare (a,b) (aus $(a,b) = \{\{a\},\{a,b\}\}$ folgt: $(a,b) \Rightarrow (a,b) = (c,d) \Leftrightarrow a = c \wedge b = d$). Mit Hilfe eines solchen geordneten Paares wird die Richtung der Kante definiert.

$\mathcal{V}_a$ ist die Menge der gerichteten Kanten $\mathcal{V}_a \subset \mathcal{A} \times \mathcal{S}$, die von Aktionen ausgehen und in Zuständen enden.

Alle Zustandsknoten $s \in \mathcal{S}$ sind paarweise verschieden:

$$\forall (s,s') : (s \in \mathcal{S}, s' \in \mathcal{S} \Rightarrow s \neq s'), \tag{6.1}$$

Bei Aktionen ist die Situation komplexer Dazu betrachte man Abb. 6.5; hier existieren zwei Knoten (K2 u. K5), die dieselbe Aktion verkörpern sollen. Deshalb wird eine Funktion t eingeführt, die jedem Aktionsknoten im Graphen (und damit in $\mathcal{A}$) einen Aktionstyp zuweist:

$$t : \mathcal{A} \rightarrow \mathcal{T}, \tag{6.2}$$

wobei $\mathcal{T}$ die Menge der Aktionstypen ist, z.B. {vor, zurück, rechts, links}. Wie man sich leicht vorstellen kann, sind nicht alle beliebigen Verbindungen $\mathcal{S} \times \mathcal{A}$ in V_s erlaubt. Sei $\mathcal{V}'$ die Menge aller erlaubten Mengen von Kanten $\mathcal{A} \times \mathcal{S}$. Dann ist

$$\mathcal{V}' \subseteq \left\{ \left(\prod \mathcal{V} \right) \setminus \emptyset \right\} \tag{6.3}$$

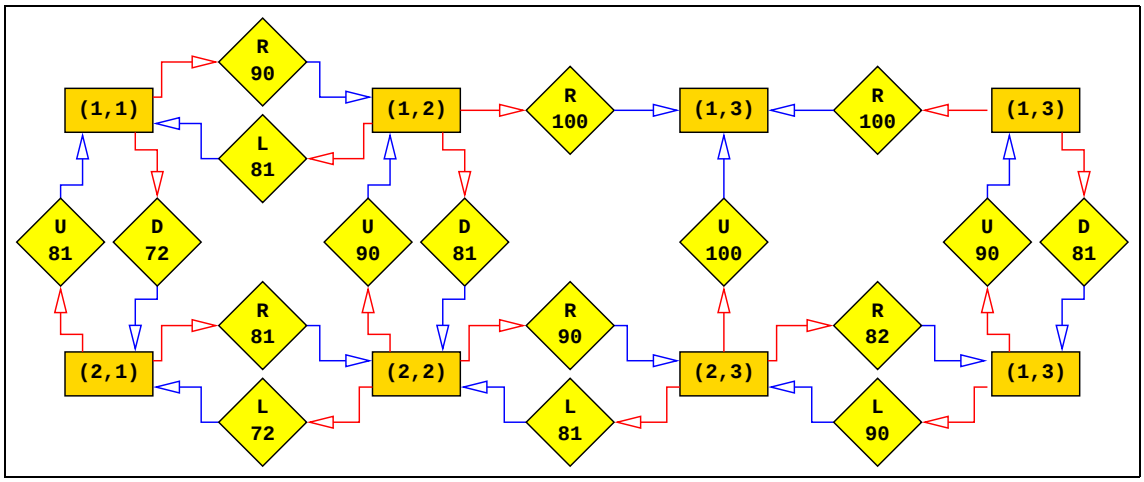


Abbildung 6.4: Zustands-/Aktionsgraph für das in Abb. 6.1 gezeigte *Grid-World* Beispiel. Zustände sind mit orange hinterlegt, Aktionen mit gelb. Die Aktionen sind mit U=hoch, D=runter, R=rechts und L=links bezeichnet. Kanten der Menge $\mathcal{V}_s$ sind rot, die der Menge $\mathcal{V}_a$ sind blau.

mit $\prod \mathcal{V}$ der Potenzmenge von $\mathcal{V}$. Die Einschränkungen der in $\mathcal{V}$' enthaltenen Mengen lassen sich wie folgt beschreiben:

$$
\begin{aligned}
\mathcal{V}' = \{x \mid & \left(x \in \left\{\left(\prod \mathcal{V}\right) \setminus \emptyset\right\}\right) \wedge \\
& (\exists (s_1, a_1) : ((s_1, a_1) \in x \wedge \\
& \exists (s_2, a_2) : ((s_2, a_2) \in x \wedge \\
& ([s_1 = s_2 \wedge a_1 \neq a_2] \rightarrow [t(a_1) \neq t(a_2)]) \vee \\
& ([s_1 \neq s_2] \rightarrow [a_1 \neq a_2]\ )\ )\ )\ )\}.
\end{aligned}
\tag{6.4}
$$

Weiterhin lassen sich die Mengen W und W' definieren:

$$\mathcal{W} = \mathcal{A} \times \mathcal{S} \tag{6.5}$$

$$\mathcal{W}' \subseteq \left\{\left(\prod \mathcal{W}\right) \setminus \emptyset\right\}, \tag{6.6}$$

so daß $\mathcal{V}_s \in \mathcal{V}'$ und $\mathcal{V}_a \in \mathcal{W}'$ ist. Die Kanten, die von einem Zustand s ausgehen, laufen also jeweils zu Aktionen unterschiedlichen Typs; die Kanten, die von Aktionen a ausgehen, laufen in unterschiedliche Zustände.

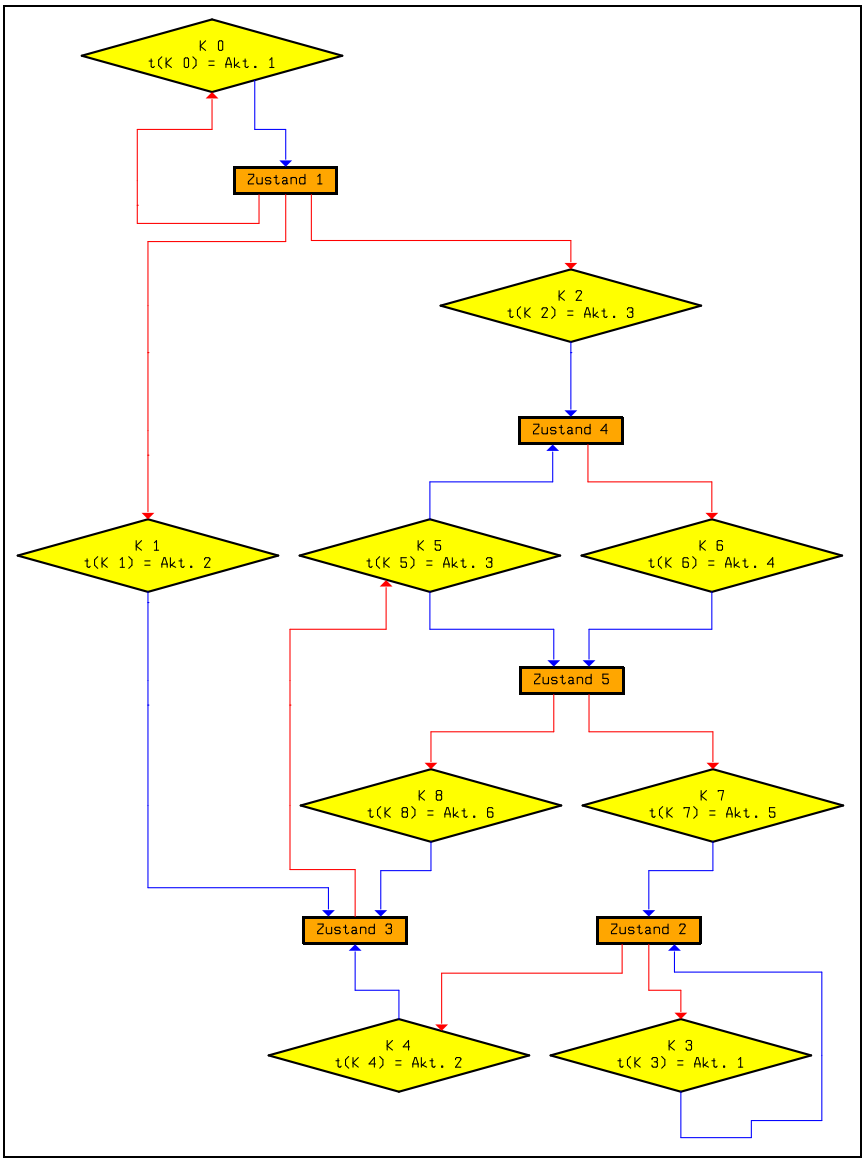


Abbildung 6.5: Beispiel eines Zustands-/Aktionsgraphen, bestehend aus fünf Zuständen (orange Kästen), die mit insgesamt neun Aktionen (gelbe Kästen) verbunden sind. Man beachte, daß die Aktion $K5$ zwei Folgezustände besitzt, während die Aktionen $K0$ und $K3$ in ihren Ursprungszustand zurückführen. Die roten Kanten führen immer von einem Zustand in eine Aktion ($\mathcal{V}_s$), die blauen von einer Aktion in einen Zustand ($\mathcal{V}_a$). Besitzt ein Zustand mehrere rote Kanten, so sind die Typen der Aktionen, auf die diese Kanten zeigen, unterschiedlich.

In der initialen Lernphase besteht ein Graph lediglich aus einem einzigen Knoten, nämlich dem aktuell beobachteten Zustand s_i. Um den Graphen zu expandieren, ermittelt eine Funktion $l : \mathcal{S} \rightarrow \mathcal{A}'$ mit $\mathcal{A}' \subset \mathcal{A}$ alle möglichen Aktionen in Abhängigkeit vom aktuellen Zustand s_i . Um eine konkrete Aktion auszuwählen, wird Gleichung 7.2 verwendet, die jedoch die Werte $Q'(s_i, a_j)$ für alle potentiellen Zustandsübergänge benötigt. Im Normalfall finden sie sich in der Q-Tabelle. Im Falle des Zustands-/Aktionsgraphen sollten sie an den Aktionsknoten gespeichert sein. Existiert der entsprechende Knoten für a_j nicht, kann man davon ausgehen, daß $Q'(s_i, a_j)$ den initialen Wert (hier Null) besitzt. Ist eine Aktion ausgewählt

worden, wird sie und die entsprechende Kante ($\mathcal{V}_s$) instantiiert. Ist die Aktion durchführbar, gelangt das System in einen neuen Zustand, der ebenfalls in den Graphen aufgenommen und mit der Aktion verbunden wird (Kante in $\mathcal{V}_a$). Ist die Aktion nicht ausführbar, was möglicherweise erst bei der Durchführung mittels Sensorik erkannt wird, ist der Folgezustand gleich dem Ursprungszustand.

Die Berechnung der neuen Q-Werte erfolgt nach Gleichung 4.15: für die die Werte $Q(s', a')$ benötigt werden. Bei der Verwendung eines Graphen muß man lediglich alle Aktionszweige des Folgezustandes s' absuchen. Im Falle des Q-Lernens ist dafür eine Rekursionstiefe von 1 zu verwenden.

6.3 Nichtdeterministische Zustands-/Aktionsgraphen

Im nichtdeterministischen Falle ändert sich am beschriebenen Prinzip nichts. Zwei mögliche Fälle müssen jedoch unterschieden werden:

① Nach dem Ausführen einer Aktion landet das System nicht immer im selben Zustand. Jede Aktion muß also mehrere Folgezustände aufnehmen können. Die Berechnung des maximalen Q-Wertes

$$\max_{a \in \mathcal{A}} Q(\delta(s, a), a'), \tag{6.7}$$

muß dieser Tatsache Rechnung tragen und die Q-Werte der Aktionen aller Folgezustände berücksichtigen. Demnach liefert $\delta(s, a)$ nicht mehr einen einzelnen Folgezustand, sondern eine Menge von Folgezuständen.

② Ohne Ausführung einer Aktion ändert sich der Weltzustand. Dieser Effekt tritt immer dann auf, wenn mehrere Aktoren den Weltzustand beeinflussen und die Abtastrate, mit der man den aktuellen Weltzustand beobachtet, höher ist als die Rate, mit der man neue Aktionen auswählt/ausführt.

Wenn man den neuen Zustand in den Graphen aufnehmen will, stellt sich die Frage nach den Verbindungen. Es gilt zu vermeiden, daß mehrere unverbundene Teilgraphen entstehen, da man in einem solchen Fall die Q-Werte möglicherweise nicht mehr korrekt berechnen kann (s.o.). Zwei Lösungsstrategien sind denkbar:

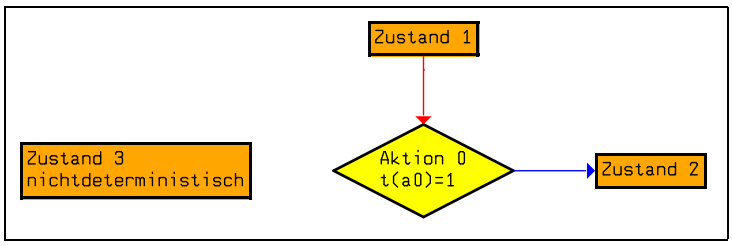
(a) Der Zustand wird mit der zuletzt ausgeführten Aktion verbunden, da es offensichtlich möglich ist, ohne weiteres in diesen Zustand zu gelangen. Nachteil: Wenn der Zeitpunkt des nichtdeterministischen Überganges nicht genau bekannt ist, weiß man nicht, welcher Zustand eigentlich Voraussetzung für den Übergang ist.

(b) Es wird eine Warteaktion eingefügt, die von dem Zustand vor dem nichtdeterministischen Übergang in den Zustand danach führt. Da man bei einem dynamischen System vermutlich sowieso über eine Warteaktion verfügt, ist diese Vorgehensweise sinnvoller als (a).

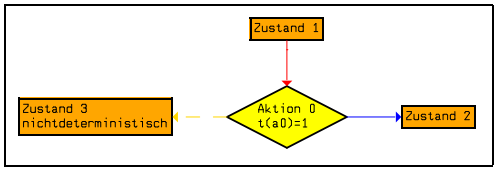
Die meisten realen Systeme und fast alle in der Robotik untersuchten Systeme, die mit MDP's arbeiten[2], müssen mit nichtdeterministischen Umgebungen umgehen. Entweder wird

[2] Sie stammen in der Regel aus der mobilen Robotik(Robot Socker).

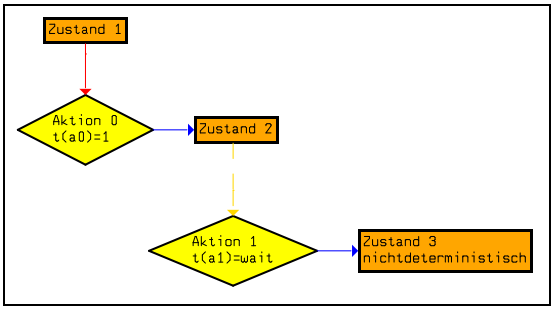
der Nichtdeterminismus durch andere Aktoren verursacht, egal ob künstlich oder natürlich, oder er entsteht durch Unsicherheiten, von denen es in realen umgebung viele gibt. Deshalb ist es von großer Wichtigkeit, daß Modelle mit solchen Prozessen umgehen können. Zu den Prozessen gehört neben dem eigentlichen Lernverfahren die Organisation des Wissens. Sie hat sich als ein grundsätzliches Problem bei diskreten MDP's herausgestellt, da der benötigte Speicherbedarf zu groß ist. Über die Möglichkeit, ein solches System erfolgreich zu trainieren, muß man sich dann, zumindest für ein reales System, keine Gedanken mehr machen.



(a) Das System wandert ohne Ausführung einer Aktion in einen anderen Zustand (*Zustand 3*). Wenn keine Maßnahmen ergriffen werden, um *Zustand 3* mit den anderen Zuständen zu verbinden, verliert der Graph, den Zusammenhang. Ein unzusammenhängender Graph ist für die Berechnung von $Q(s,a)$ nach Gleichung 4.15 nicht mehr zu gebrauchen, da sich $\max_{a'} Q(s',a')$ nicht mehr korrekt berechnen läßt.



(b) Der neue Zustand wird als Folgezustand der letzten Aktion betrachtet. Bei dieser Vorgehensweise argumentiert man so, daß der nichtdeterministische Zustandsübergang offensichtlich nicht vom Handeln des Roboters abhängt und deshalb auch während der Ausführung der letzten Aktion hätte auftreten können. Diese Argumentation greift aber zu kurz, da der Übergang zu *Zustand 3* durchaus vom Erreichen des *Zustandes 2* abhängig sein kann.



(c) Es wird eine Warteaktion eingefügt, mit der der neue Zustand verbunden wird. Nach dem Motto: Warten ist immer gut, trifft diese Vorgehensweise den Sachverhalt besser. Wenn man bedenkt, daß der Zustandsübergang geschah, während keine Aktion durchgeführt wurde, entspricht die Warteaktion der Realität, auch wenn die eigentliche Anwendung gar nicht gewartet hat, sondern mit internen Berechnungen beschäftigt war.

Abbildung 6.6: Durch nichtdeterministische Zustandsübergänge kann es dazu kommen, daß das System in Zustände übergeht, ohne daß eine Aktion ausgeführt wurde. Dennoch muß in einem solchen Fall der neue Zustand mit dem restlichen Graphen verbunden werden.

7

Lernen auf mehreren Ebenen

Komplexe Systeme wie z.B. Roboter/Multiroboter-Systeme werden aus unterschiedlichen Gründen (Kapselung, Komplexität, Parallelisierbarkeit, ...) in Ebenen zerlegt, die aufeinander aufbauen. So ist bereits ein klassisches Robotersystem wie z.B. „RCCL“ in drei Ebenen aufgeteilt: die Gelenksteuerungsebene, die Bahnsteuerungsebene und die Planungsebene. Auf der untersten Ebene werden kartesische Positionen in Gelenkwinkel umgerechnet und diese wiederum über einen Regler oder eine Reglerkette in Motorströme umgewandelt. Die kartesischen Positionen stammen aus der mittleren Ebene, in der Positionsgleichungen in eine Abfolge von kartesischen Positionen umgerechnet werden müssen. Hier lassen sich z.B. Singularitäten behandeln oder aus Sensoren gewonnene Informationen zur Bahnkorrektur anbringen. Die Positionsgleichungen stammen aus der obersten, für die Planung verantwortlichen, Ebene. Auf dieser werden die Handlungen des Roboters durch Bestimmung einer Abfolge von Bewegungen geplant.

Zusätzlich können Algorithmen einer Ebene Einfluß auf die Arbeitsweise der darunter liegenden Ebenen nehmen, indem sie Parameter verändern, Regler auswählen oder das Interpolationsverfahren für die kartesischen Punkte zwischen Start- und Zielpunkt festlegen.

Es wird von Lernen auf mehreren Ebenen[1] gesprochen, wenn auf den unterschiedlichen Ebenen eines softwaregesteuerten Systems Lernverfahren eingesetzt werden. Wenn die Ebenen voneinander abhängen, beeinflussen sich die Lernverfahren unter Umständen gegenseitig. Oft sind komplexe Lernprobleme erst lösbar, wenn sie in mehrere Lernebenen zerlegt werden, auf denen unterschiedlichste Lernmethoden zum Einsatz kommen. Deshalb kann die Wahl eines lernenden Ansatzes selbst schon Grund für einen hierarchisierten Aufbau sein.

Lernverfahren werden immer dann zu komplex, wenn eine direkte Abbildung der Eingangsdaten (Sensordaten) auf die (Ausgangsdaten) Aktorik nicht mehr möglich ist. Bildet man eine hierarchische Struktur, empfiehlt es sich, das System von den unteren zu den oberen Ebenen hin zu entwickeln. Je höher eine Ebene liegt, um so komplexer ist die zu behandelnde Problematik, da die jeweils höhergelegene Ebene auf das Funktionieren der darunter liegenden Schicht(en) angewiesen ist.

[1] Layered - Learning.

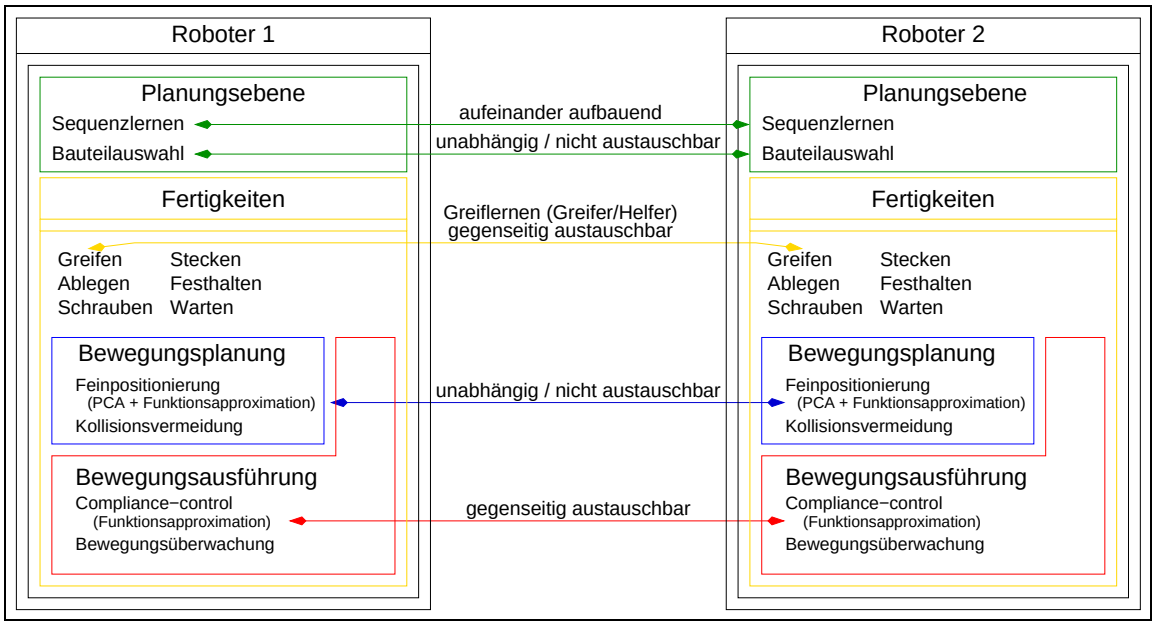


Abbildung 7.1: Die Struktur eines Agenten läßt sich in mehrere Ebenen zerlegen. Jede Ebene kann Lernverfahren beinhalten, so daß die darüberliegenden Ebenen von diesen Verfahren abhängen. Die gelernten Werte sind nicht prinzipiell zwischen den Agenten austauschbar. Beinhalten sie hardwarespezifische Werte (Kalibrierung), sind sie roboterspezifisch. Bei anderen Verfahren wie z.B. dem Sequenzlernen zum Greifen und zur Montage verteilt sich das Wissen auf die beteiligten Roboter.

Es kann unter Umständen notwendig sein, den Lernvorgang auf den einzelnen Ebenen zu entkoppeln, um dynamische Effekte zu vermeiden. In diesen Fällen wird nur in einer Schicht gelernt, während die anderen auf ihren bereits gelernten Werten arbeiten, ohne weiter hinzuzulernen. Natürlich läßt sich auf jeder Ebene ein anderes, für die jeweilige Problematik angepaßtes Lernverfahren verwenden.

Die Ebenenstruktur selbst ist fest vorgegeben und wird nicht mitgelernt. Sie kann als evolutionär gegeben betrachtet werden. Einen Ansatz zur Selbstorganisation eines hierarchisch organisierten Lerners wird in (TAKAHASCHI AND ASADA, 2000) beschrieben.

Die einzelnen Ebenen werden zum Teil auch als „Verhaltensmuster[2]“ interpretiert. Verwendet ein Verhaltensmuster ein zweites, liegt eine Hierarchisierung vor. So teilt Stone (STONE, 1998) seine Roboteragenten in folgende Verhaltensebenen auf: Individuelles Verhalten, Multi-Agenten-Verhalten und Mannschaftsverhalten. Im konkreten Fall des lernenden Multiroboterarm-Systems besteht jeder Roboter aus mehreren Schichten, siehe Abb. 7.1. Der in dieser Abbildung mit Fertigkeiten bezeichnete Bereich besteht zu einem Teil aus Lernverfahren, zu einem anderen aus fest einprogrammierten Fähigkeiten.

[2] Behaviors.

7.1 Lernebenen zur Steuerung eines Roboterarms

7.1.1 Lernverfahren unterhalb der Fertigkeiten

In dem hier benutzten Konstruktionsszenario werden zur Realisation der unterschiedlichen Fähigkeiten nicht immer voneinander unabhängige Lernverfahren verwendet:

- Lernverfahren für gefügige Bewegungen:
  - Herstellen einer Kontaktsituation.
  - Einregeln einer Sollkraft / eines Solldrehmoments.
  - Minimieren der Kräfte/Drehmomente während des Einführens einer Schraube in ein Loch.
  - Minimieren der Kräfte/Drehmomente während eines Schraubvorganges.
- Lernverfahren zur Armpositionierung:
  Lernen von Korrekturbewegungen zur Feinpositionierung über einem a priori bekannten Objekt.
- Lernverfahren zur Objekterkennung mit einer Handkamera, um festzustellen welches Objekt unter der Handkamera bzw. unter dem Greifer liegt. Einige Objekte ändern stark ihre Gestalt in Abhängigkeit vom Blickwinkel. Wenn dieser mit der Greiforientierung korrespondiert, ist nicht nur der Bauteiltyp, der unter dem Greifer liegt, sondern auch die Orientierung von Interesse. Daraus lassen sich Orientierungsklassen ableiten, z.B. stehend, liegend, die wiederum Einfluß auf das Feinpositionieren und Greifen haben.

Weitere Lernverfahren auf dieser Ebene bis hin zum Erlernen der Kinematik sind untersucht worden. Da sie in der konkreten Anwendung nicht realisiert wurden, soll hier nicht weiter darauf eingegangen werden.

7.1.2 Lernverfahren auf der Ebene der Fertigkeiten

Die folgenden Fertigkeiten werden für die einfachen Montageoperationen benötigt: Greifen, Stecken, Schrauben, Ablegen, Loslassen, Festhalten. Sie wurden größtenteils fest programmiert, benutzen aber für Teilhandlungen Lernverfahren. So wird die gefügige Bewegung der Steckfertigkeit von einem Funktionsapproximator erlernt. Problematisch wird es jedoch, wenn eine Fertigkeit mit essentiellen Unsicherheiten umzugehen hat. Damit sind Fertigkeiten gemeint, deren Durchführung – abhängig von ihrer Umgebung – völlig unterschiedlicher Handlungen bedarf. Programmiert man eine derartige Fähigkeit fest ein, wird das Programm sehr umfangreich und komplex, da alle möglichen Voraussetzungen und Verzweigungen innerhalb der Fertigkeit berücksichtigt werden müssen. In solchen Fällen können Lernverfahren eingesetzt werden, die für alle tatsächlich auftretenden Situationen das richtige Handeln erlernen.

Das Aufnehmen eines einzelnen Bauteils vom Tisch fällt in diese Klasse von Fertigkeiten und läßt sich, wie später gezeigt wird, mit Hilfe eines Zustands-/Aktions-Graphen lernen.

Problematik des Greifens

Eine Greifoperation ist solange unproblematisch, wie der Ort und die Orientierung, an der gegriffen werden soll, erreicht werden kann. Ist das nicht der Fall, muß man entweder das Greifen insgesamt aufgeben, oder es muß eine Greifstrategie unter Beteiligung anderer Roboter erarbeitet werden. Der Ablauf des Greifens kann in Abhängigkeit von der Lage des Bauteils unterschiedliche Teilhandlungen erforderlich machen.

In dem hier gegebenen Szenario aus Schrauben, Würfeln und Leisten sind insbesondere liegende Schrauben, aber auch Würfel von dem Problem der unerreichbaren Greiforientierung betroffen. Die Unerreichbarkeit in der Position gilt für alle Bauteile gleichermaßen. Besonders Schrauben neigen dazu umzufallen, also nicht auf dem Gewinde stehenzubleiben wie in Abb. 5.7. Die für das Greifen günstige, auf dem Gewinde stehende Orientierung ist sehr instabil, und bereits die leichtesten Störungen beim Ablegen können ein Umfallen verursachen. Die Orientierungsangaben, die man z.B. von einer globalen Ansicht bekommt, sind also mit einer gewissen Unsicherheit behaftet. Die stehende Orientierung (Schraube steht auf dem Gewinde) ist in einem natürlichen Szenario eher unwahrscheinlich.

Die Orientierung des Bauteils im Greifer hat direkten Einfluß auf übergeordnete Montageprozesse. Die Bauteilorientierung ist deshalb Parameter der Greiffertigkeit und nicht dessen zufälliges Ergebnis. Sonst existiert nämlich keinerlei Planungssicherheit über diese Größe auf der darüberliegenden Schicht.

Zunächst ist das Greifen eines Bauteils ein Problem des einzelnen Roboters. Stellt er fest, daß er das Problem nicht selbst lösen kann, sucht er sich einen Helfer, um mit ihm zusammen die Aufgabe zu lösen. Das verwendete Lernverfahren findet zwischen diesen zwei Robotern mit vordefinierten Rollen statt. Je nach Aufgabenstellung wird ein gewisser Satz von Erfahrungen, in diesem Fall ein Zustands-/Aktionsgraph, selektiert.

Der helfende Roboter läßt sich bei seiner momentanen Tätigkeit an definierten Stellen unterbrechen. Er muß dann einen Zustand herstellen, in dem er helfen, z.B. das aktuell im Greifer befindliche Bauteil ablegen kann, und zwar möglichst so, daß er es später ohne fremde Hilfe wieder aufnehmen kann. Er führt dann die nötigen Hilfshandlungen durch, und nach deren Beendigung kehrt er wieder in seinen ursprünglichen Zustand zurück. Es kann notwendig werden, daß ein helfender Agent sich selbst helfen lassen muß. Daraus kann sich das Problem des rekursiven Aufrufs ergeben. Ein Agent kommt jedoch – zumindest beim Greifen eines einzelnen Bauteils – nur äußerst selten in eine solche Situation. Deshalb soll dieser Aspekt nicht weiter behandelt werden.

Wählt man die internen Zustände sorgfältig aus, lassen sich die erlernten Strategien auf andere Roboter übertragen. Dazu müssen die Zustände roboterunabhängig gestaltet werden. Die Details des Lernverfahrens finden sich in Kapitel 7.4.

7.1.3 Lernverfahren auf der Planungsebene

Die Planungsebene bestimmt das Handeln des Roboters im großen. Hier soll ein einfaches Montageszenario erlernt werden. Dabei liegen verstreut auf der Arbeitsfläche einige Bauteile, die von den Robotern zu einem vorgegebenen, einfachen Aggregat montiert werden. Es soll keine zentrale Planungsinstanz existieren, die auf Basis des Zielaggregats im klassischen Sinn einen Montagegraphen und daraus Roboterhandlungen generiert. Vielmehr sollen die einzelnen Roboter durch zielgerichtetes Probieren einen Weg zum vorgegebenen

Aggregat finden. Es soll dabei keinerlei zentrale Instanz geben, die das Gelernte über alle Roboter verteilt, vielmehr handelt jeder Roboter für sich und findet dabei seine Rolle.

Es wird auch hier ein zustandsorientiertes Lernverfahren verwendet, das in der Lage sein muß, in einem potentiell unendlichen Zustandsraum zu operieren. Die Zustands-/Aktionsgraphen bieten dafür das geeignete Hilfsmittel.

7.2 Lernende Kraftregelung

In diesem Abschnitt soll dargelegt werden, wie ein Funktionsapproximator(Anhang C) verwendet werden kann, um die für eine Kraftregelung nötigen Korrekturen zu berechnen. Die Entscheidung für das B-Spline Modell als Approximator beruht auf zwei herausragenden Eigenschaften desselben, die besonders beim Onlinelernen mit hohen Zyklusraten vorteilhaft sind: Zum einen lassen sich leicht mehrdimensionale Funktionen approximieren, zum anderen – und das ist der entscheidende Aspekt – wirken sich Lernschritte nur lokal aus. Damit ist gemeint, daß Änderungen an den Singletons nicht gleich die gesamte Form der Funktion beeinflussen, sondern nur deren unmittelbare Umgebung.

Das Hookesche Gesetz (3.1) dient dazu, den Korrekturschritt zu berechnen. Deshalb kann ein überwachtes Lernverfahren zum Training des Approximators benutzt werden. Um die richtige Funktion approximieren zu können, muß der Regler allerdings die richtigen Eingangsdaten erhalten. In dem hier realisierten Approximator werden folgende Eingangsdimensionen verwendet:

$F_{\mathbf{error}}$ Regelabweichung (Kraftfehler)

$T_{\mathbf{error}}$ Regelabweichung in dem mit F verbundenen Drehmoment

F Absoluter Kraftwert

$\dot{F}$ Kraftänderung

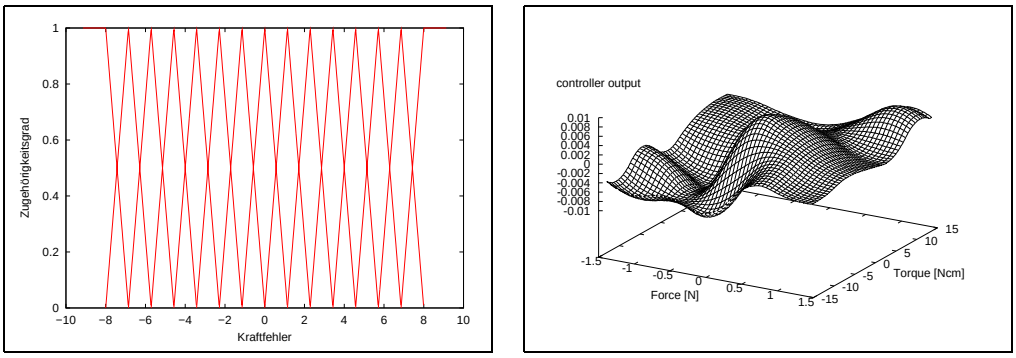
x Position

$\dot{x}$ Positionsänderung

Jede Dimension wird durch linguistische Terme in Form von B-Splines beschrieben. Sie werden homogen über den Wertebereich verteilt wie z.B. in Abb. 7.2(a). Um den Approximator zu trainieren, wird er in das Regelsystem eingefügt und bekommt in jedem Regelschritt (14ms) ein Trainingsbeispiel. Die dafür benötigte Regelschleife zeigt Abb. 7.3. Die entscheidende Größe stellt dabei die Regelabweichung (F_{error}) dar. Zusammen mit den anderen oben genannten Eingangsgrößen geht sie sowohl in den eigentlichen Regler als auch in den Lernalgorithmus ein. Während der Regler aus diesen Daten und den Werten der Regelbasis nach Gleichung C.3 eine Stellgröße, hier die Positionsänderung berechnet, modifiziert das Lernverfahren die Werte in der Regelbasis (Gleichung C.9). Dazu werden die Positionskorrekturen benötigt, die zu dem aktuellen Kraftfehler geführt haben. Sie zu bestimmen ist genauso wichtig wie schwierig, da sie von den Totzeitgliedern der Regelstrecke (roter Kasten) abhängen. Für die verwendeten Puma 260 Roboter mit Gelenkreglern im Proportionalmodus[3] wurde experimentell[4] ein Wert von 1+4 Zyklen bestimmt. Das zusätzliche

[3] Es ist möglich, einen Integralregler innerhalb der Gelenkregelung einzuschalten.

[4] Auswertung der Sprungantwort.



(a) Partitionierung der ersten Eingangsdimension mit B-Splines 2^{ter} Ordnung.

(b) Zwei Dimensionen der online gelernten Funktion zum compliance control

Abbildung 7.2: Dimensionsbeschreibung und approximierte Funktion zum compliance control

Totzeitglied entsteht durch die Robotersteuerung[5] selbst, die die Daten[6] verzögert an die Roboter weiterleitet. Abb. 7.2(b) zeigt zwei Dimensionen eines online - trainierten Reglers, Abb. 3.3(c) die Kraftwerte, die auftreten, wenn der Roboter aus einer kontaktfreien Situation auf ein Hindernis auffährt.

7.3 Objektklassifikation und Feinpositionierung

Zur exakten Positionierung über einem Bauteil wird in dieser Arbeit eine am Roboter befestigte Handkamera Abb. 3.1(a) benutzt.

Es gibt mehrere Möglichkeiten, die Problematik des Greifens mit einem Zweibackengreifer zu lernen. In (WENGEREK, 1995) wird ein Verstärkungslernverfahren zur Positionierung eines Greifers mit einer externen Kamera beschrieben. Dabei wird das Problem der Dimensionsreduktion durch Überlagerung der Bilder mit einem/mehreren Rastern (*Virtuelle Sensorfelder*) realisiert. Als Lerner fungiert ein Perzeptron.

In dieser Arbeit wird, um keinerlei Kalibration vornehmen zu müssen, eine Kombination aus einem ansichtenbasierten Verfahren und einem Funktionsapproximator (AnhangC) Eingesetzt, das im folgenden kurz beschrieben wird. Die in dieser Arbeit verwendeten Implementationen stammen aus Arbeiten von R. Schmidt (SCHMIDT, 1998), C. Berger (BERGER, 2000) und T. Scherer (SCHERER, 2000).

Zunächst soll auf die Idee des Verfahrens und die darin enthaltene Dimensionsreduktion kurz eingegangen werden. Die verschiedenen Schritte, die benötigt werden, um Objekte wiederzuerkennen, bzw. um sie unter einem Greifer in eine genormte Lage zu bringen, werden in den beiden anschließenden Unterabschnitten erläutert.

[5] RCCL.
[6] Gelenk-setpoints.

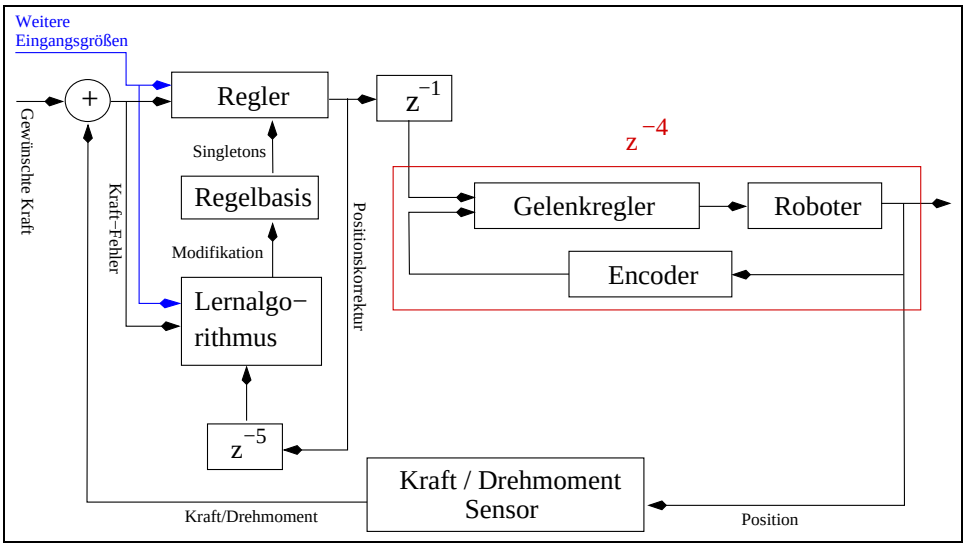


Abbildung 7.3: Die Regelschleife, die eingesetzt wird, um die Abweichungen der Kräfte von einer Sollkraft mit Hilfe eines Funktionsapproximators zu minimieren. Sie beinhaltet die Totzeit des Gelenkreglers, die hier fünf Zyklen entspricht.

7.3.1 PCA

Das Problem bei der Analyse von Bildern ist ihre hohe Dimensionalität. Wenn man jeden Pixel als eine Dimension auffaßt, ergeben sich bei einem Bild in halber PAL-Auflösung[7] 110592 Dimensionen. Ein derartiger Vektor beinhaltet eine große Menge an redundanter Information Mit Hilfe der PCA lassen sich die Bereiche eines Bildvektors finden, die die größten Änderungen im Bild bezogen auf eine Menge von Bildern haben.

Das Verfahren besteht aus zwei Teilen: Vor der eigentlichen Anwendungsphase wird eine offline-Trainingsphase benötigt. In dieser wird die eigentliche PCA durchgeführt. In der Anwendungsphase werden die bei der PCA berechneten Hauptkomponenten verwendet.

Anwendungsphase

In der Anwendungsphase werden die Bilder zunächst vorverarbeitet. Hierzu gibt es die unterschiedlichsten Methoden; wichtigste Komponente ist jedoch die Energienormierung (Helligkeitsnormierung) des Bildes, damit es mit anderen Bildern vergleichbar wird.
Der nächste Schritt ist die sogenannte Projektion. Dabei wird das Bild $\vec{b}$ (n-Dimensionen) mit einer aus den k Hauptkomponenten bestehenden Matrix C so multipliziert, daß ein k-dimensionaler Projektionsvektor $\vec{p}$ entsteht:

$$\begin{pmatrix} c_{11} & \dots & c_{1n} \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ c_{k1} & \dots & c_{kn} \end{pmatrix} \begin{pmatrix} b_1 \\ \dots \\ \dots \\ b_n \end{pmatrix} = \begin{pmatrix} p_1 \\ \dots \\ \dots \\ p_k \end{pmatrix}, \tag{7.1}$$

wobei n die Anzahl der Bildpixel ist und k die Anzahl an verwendeten Hauptkomponenten. Dabei ist $k \ll n$, was eine entsprechende Dimensionsreduktion darstellt. Benötigt wird dazu

[7] 384×288 Pixel.

eine Matrix C, die aus den k Eigenvektoren besteht, die mit den k größten Eigenwerten korrespondieren (Hauptkomponenten). C wird in der Trainingsphase berechnet.

Trainingsphase

Zunächst wird eine Menge von Trainingsbildern aufgenommen, die das Objekt in möglichst vielen Orientierungen und Translationen zeigen. Jedes Bild wird mit der entsprechenden Vorverarbeitung normiert und in eine Bildmatrix geschrieben. In einem weiteren Schritt wird nochmals von allen Bildvektoren der über alle Bilder gebildete Durchschnittsvektor abgezogen. Wenn man sich eine Punktewolke vorstellt, die sich in einem n-dimensionalen Raum befindet und man von jedem Punkt der Wolke den Mittelwert über alle Punkte abzieht, so kommt das einer Verlagerung der Wolke in den Nullpunkt gleich. In einem nächsten Schritt wird die implizite Kovarianzmatrix $B' = \mathrm{icov}(B)$ gebildet. B' ist im Gegensatz zu B quadratisch, was zur Berechnung der Eigenvektoren u. Eigenwerte (ENGELN-MUELLGES AND REUTTER, 1996) notwendig ist. Mathematisch korrekter wäre die Verwendung der Kovarianzmatrix, diese ist aber auf Grund ihrer Größe nicht handhabbar.

7.3.2 Objektklassifikation aus Handkameraperspektive

Spezifisch für diese Anwendung der Hauptkomponenten ist die verwendete Vorverarbeitung und die Erkennung selbst. Bei der Vorverarbeitung wird eine Farbklassifikation durchgeführt, die im wesentlichen darin besteht, die Bilder im YVC-Raum zu betrachten und alle Punkte auf prototypische Farbachsen zu projizieren. Dadurch ist es möglich, Regionen weitestgehend unabhängig von den Beleuchtungsverhältnissen zu selektieren. Die größte erkannte Region wird im Bildmittelpunkt zentriert und mit der entsprechend gelernten Matrix C projiziert.

Das Erkennungsverfahren besteht darin, den projizierten Vektor mit einer Menge anderer Vektoren zu vergleichen und die höchste Übereinstimmung zu ermitteln. Die Trainingsbilder liefern dabei die Menge der Vergleichsvektoren. Da bekannt ist, zu welchem Objekt das Trainingsbild gehört, kann auf diesem Wege auf die Objektklasse geschlossen werden.

Durch den letzten Schritt handelt es sich insgesamt um ein überwachtes Verfahren, auch wenn die PCA zu den unüberwachten Verfahren gezählt wird. Es ist deshalb sinnvoll, an Stelle der PCA die ORF[8] zu verwenden.

7.3.3 Feinpositionierung mit der Handkamera

Im Unterschied zur Objekterkennung wird bei diesem Verfahren das Objekt in der Vorverarbeitung nicht zentriert, des weiteren wird der projizierte Vektor nicht mit den Testbildern verglichen, sondern als Eingang eines Funktionsapproximators (Kapitel C) verwendet. In der Trainingsphase sind die Translations- und Rotationsfehler bekannt, was es erlaubt, den Funktionsapproximator mit den projizierten Testbildern zu trainieren (da auch dieser Teil überwacht ist, gilt das gleiche wie oben).

Eine Besonderheit des Verfahrens ist die Entkoppelung von Rotation und Translation. Es hat sich herausgestellt, daß Translationsfehler wesentlich größere Varianzen erzeugen als Rotationsfehler. Deshalb muß zunächst der Translationsfehler weitgehend ausgeregelt werden, bevor der Rotationsfehler behoben werden kann.

[8] Output-relevant-features.

7.4 Erlernen einer Greifstrategie

Um ein komplexeres Aggregat zuammenbauen zu können, müssen die einzelnen Komponenten/Bauteile von einem Tisch aufgenommen werden. Dabei spielt die Orientierung, in der sich das Bauteil nach dem Greifen in der Hand des Roboters befindet, eine wesentliche Rolle. Sie ist maßgeblich dafür verantwortlich, ob und wie anschließende Montageoperationen durchzuführen sind. Um alle Bauteile gleichermaßen behandeln zu können, muß ein Bauteil immer so gegriffen werden, daß der Port für die sich anschließende Montageoperation parallel bzw. antiparallel zur Greifer-z-Achse verläuft. Diese Forderung kann von einem einzelnen Roboter nicht immer erfüllt werden.

Bei symmetrischen Objekten wie den Leisten und Muttern und partiell auch den Würfeln ist eine gedankliche (virtuelle) Rotation des entsprechenden Bauteils möglich. Dadurch kann die gewünschte Lage im Greifer oder auf dem Tisch erzielt werden, ohne daß tatsächlich eine Umorientierung stattgefunden hat. Bei Schrauben ist eine derartige symmetrische Rotation lediglich entlang des Gewindes möglich, was im allgemeinen von wenig Interesse ist, da dadurch die Lage im Greifer nicht auf die gewünschte Weise verändert werden kann.

In einem realen Szenario ist die typische Lage einer Schraube auf dem Tisch liegend, im Gegensatz zu Abb. 5.7. Für einen einzelnen Roboter, der zum Greifen lediglich über einen Zweibackengreifer verfügt, ist dies aber genau die Orientierung, mit der eine nachfolgende Montageoperation (Stecken,Schrauben) nicht zurechtkommt.

Entweder limitiert sich das System auf das Greifen stehender Schrauben oder es wird der Einsatz mehrerer Roboter genutzt. Wenn der Untergrund (hier Tischplatte) uneben[9] ist, führt eine Beschränkung auf stehende Schrauben schnell in eine Situation, in der alle Schrauben liegen. Stellt der Roboter eine Schraube ab, kann und wird diese auf Grund der Unebenheiten umfallen. (Das Umwerfen der Schraube kann auch durch andere Effekte verursacht werden, z.B. Bewegungen beim Greifen anderer Bauteile oder verbleibende Kräfte vor dem Öffnen der Hand beim Abstellen der Schraube.) Um längere Montagestrategien erlernen zu können, ist es daher unabdingbar, Schrauben auch in liegender Orientierung greifen zu können.

Zum Greifen sollen deshalb prinzipiell zwei Roboter verwendet werden. Sei M der Roboter, der das Bauteil letztendlich halten soll und S der dabei unterstützende Roboter. Folgende Ausgangssituationen sind dann möglich: Das Bauteil steht, das Bauteil liegt, das Bauteil befindet sich in Reichweite von S, in Reichweite von M oder in Reichweite von S und M.

Angenommen, es stehen eine Reihe von primitiven Operationen $\mathcal{A}$ für jeden Roboter zur Verfügung. Wie müssen sie dann kombiniert werden, um das gewünschte Ziel zu erreichen? Dabei ist zu berücksichtigen, daß die beiden kooperierenden Roboter prinzipiell unabhängig agieren. Es werden Operationssequenzen für beide Roboter benötigt, die synchronisiert sind.

7.4.1 Aktionsraum

Die Menge der Aktionen $\mathcal{A}$ soll aus den folgenden Operationen bestehen:

① **Greifen 0^o:** Bauteil greifen, unabhängig von der Bauteilorientierung. Dabei wird der Greifer immer senkrecht zur Tischoberfläche positioniert. Bei Würfeln gibt es zwei mögliche Orientierungen. Deshalb werden zwei Greifaktionen (0^o, 90^o) unterschieden.

[9] wie in diesem Szenario.

② **Greifen 90^o:** Wie oben, aber um den Annäherungsvektor mit 90^0 gedreht. Bei Bauteilen, die nur in 0^o gegriffen werden können[10], wird ein Fehlerstatus generiert, der von dem Lernverfahren als Reinforcementsignal genutzt wird.

③ **Ablegen:**Abgelegt wird immer zwischen den beiden beteiligten Robotern, so daß danach beide Roboter Zugriff auf das Bauteil bekommen. Liegt bereits ein Bauteil an der entsprechenden Position, wird die Nachbarschaft in diskreten Radien abgesucht.

④ **Übergeben 180^o:** Beim Übergeben eines Bauteils an den jeweils anderen Roboter bestehen bei Würfeln mehrere mögliche Übergabeorientierungen: 180^o heißt, beide Roboter haben genau eine entgegengesetzte Orientierung (Annäherungsrichtung). In die Übergabefunktionen fließen Modellinformationen ein, um unmögliche Übergabeorientierungen[11] auszuschließen. Tritt ein solcher Fall ein, wird ein Fehler zurückgeliefert, der als Reinforcementsignal genutzt wird.

⑤ **Übergeben 90^o:** Bei dieser Übergabefunktion steht das Tool des übergebenden Roboters im Winkel von 90^o plus einem bauteilabhängigen Wert zum übernehmenden Roboter. In diese Übergabefunktion fließen ebenfalls Modellinformationen ein, um unmögliche Übergabeorientierungen[12] auszuschließen. In einem solchen Fall wird ebenfalls ein Fehler zurückgeliefert, der als Reinforcementsignal genutzt wird.

⑥ **Übernehmen :** Gegenstück zum Übergeben. Das Übernehmen erfordert eine etwas andere Abfolge von Bewegungen als das Übergeben, verläuft im Großen und Ganzen aber symmetrisch. Übernommen wird immer nur in einer Orientierung.

⑦ **Warten :** Nichts tun, bis der andere Roboter durch eine Aktion den Weltzustand geändert hat. Mit Hilfe dieser Operation lassen sich die Roboter auf einfache Weise synchronisieren. Gesondert behandelt werden muß der Fall, in dem beide Agenten gleichzeitig in den Wartezustand übergehen. Es ist zwar richtig, daß einer der beiden Roboter nicht in den Wartezustand hätte übergehen dürfen. Welcher von beiden der richtige gewesen wäre, ist aber nicht entscheidbar, deshalb werden beide Roboter aufgeweckt und es wird nicht bestraft.

(Siehe Abb. 7.4 – Abb. 7.7)

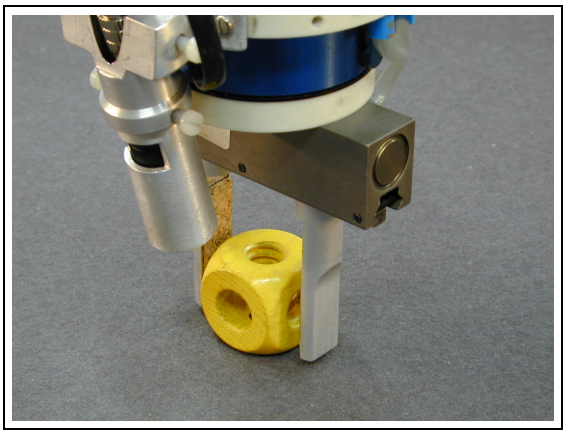
7.4.2 Zustandsraum

Die Abfolge von Operationen wird durch ein Q-Lernverfahren ermittelt, das schrittweise einen Zustands-/Aktionsgraphen im Sinne von Kapitel 6 entwickelt. Der Zustandsvektor beinhaltet folgende Größen:

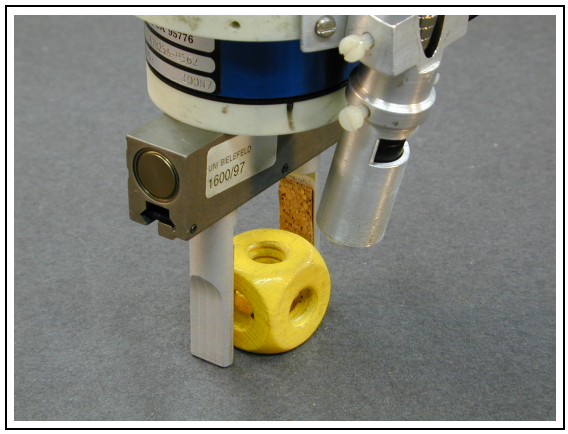
Greifbarkeit: $o \in \mathcal{O}$. Beschreibt für welchen der Roboter das Objekt greifbar ist. o kann einen der folgenden Werte annehmen:

-1: Von keinem der beiden Roboter greifbar,
1: in der Reichweite des jeweils anderen Roboters,
2: in der eigenen Reichweite,
3: in der Reichweite beider Roboter.

[10] alle außer Würfel.
[11] z.B. Schrauben mit Gewinde voran.
[12] z.B. Rautenmutter mit Gewinde parallel zum Tisch.

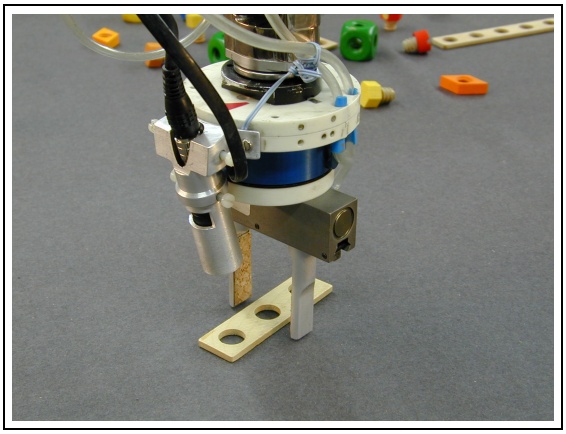


(a) Greifen in der Orientierung 0^0.

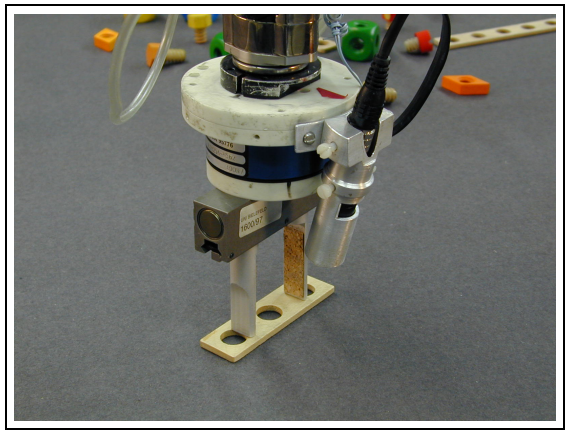


(b) Greifen in der Orientierung 90^0.

Abbildung 7.4: Es existieren zwei Greifaktionen mit einer Winkeldifferenz im Roll-Winkel von 90^0. Für Würfel folgt daraus, daß das gewindelose Loch einmal in Schließrichtung verläuft, im anderen Fall in Normalenrichtung.

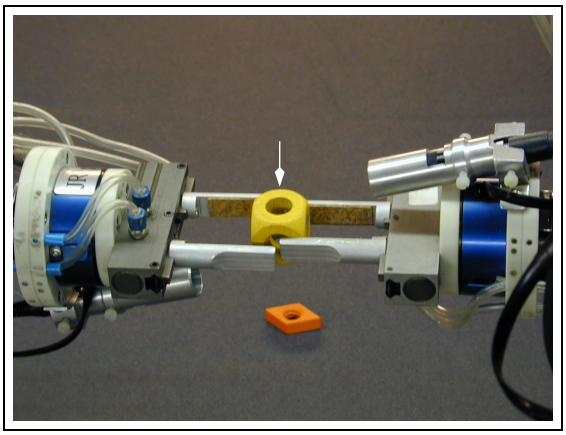


(a) Greifen einer Leiste in der Orientierung 0^0

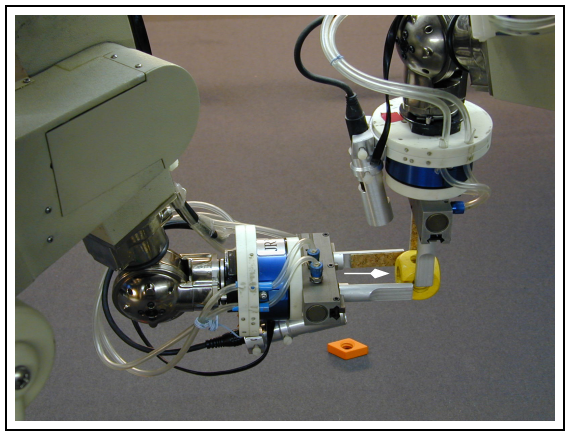


(b) Der Versuch, eine Leiste in der Orientierung 90^0 zu greifen.

Abbildung 7.5: Bei Leisten macht nur eine der beiden Orientierungen Sinn. Durch Modellwissen wird die falsche Variante von vornherein ausgeschlossen.

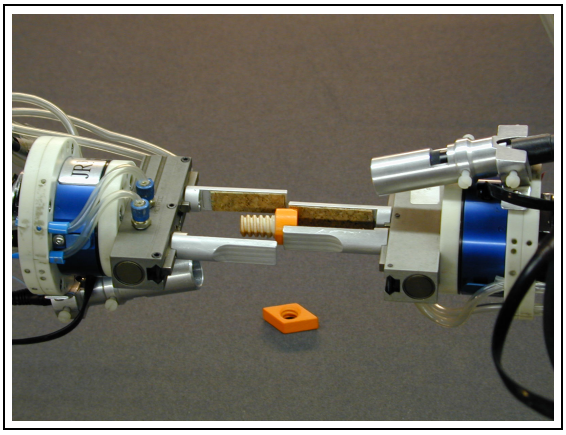


(a) Übergeben eines Würfels mit einer Orientierung von 180^0.

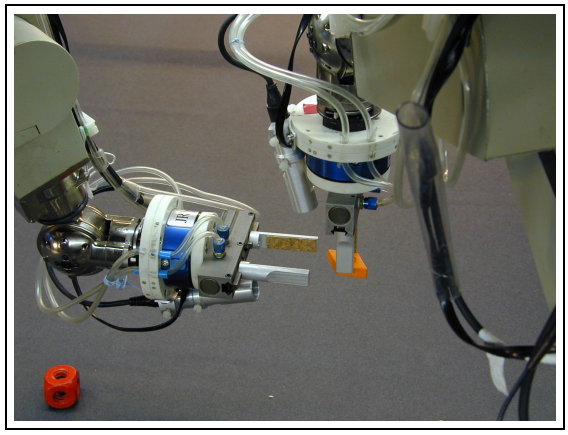


(b) Übergeben eines Würfels mit einer Orientierung von 90^0.

Abbildung 7.6: Würfel lassen sich in den beiden möglichen Orientierungen 180^0 u. 90^0 übergeben. Man beachte wiederum die Lage des gewindelosen Loches (Pfeil).



(a) Übergeben einer Schraube in 180^0 ist nicht möglich.



(b) Übergeben einer Rautenmutter in 90^0 ist nicht möglich.

Abbildung 7.7: Für andere Bauteile als Würfel funktioniert nur die eine oder andere Methode. Um den Lernprozeß zu beschleunigen, wurde das Wissen darüber in die entsprechenden Aktionen einprogrammiert, ist aber nicht zwingend notwendig. Die Handkamera kann benutzt werden, um festzustellen, ob das Übergeben geglückt ist (b); die Schraube mit dem Gewinde nach vorn (a) kann durch ein *Ablegen* und erneutes *Umgreifen* umgedreht werden. Das setzt aber voraus, daß die *Greiffunktion* damit umgehen kann.

Greiferzustand: $g \in \mathcal{G}$**.** Diskreter Wert, der beschreibt, ob sich das Bauteil im Greifer des Zielroboters, des helfenden Roboters oder auf dem Tisch befindet.

Aktueller Port: $P_{\textbf{ist}}$**.** Der Port, an dem das Bauteil momentan festgehalten wird, vorausgesetzt es wird ein Bauteil gehalten.

Aktueller Port: $P_{\textbf{soll}}$**.** Der Port, an dem das Bauteil vom Zielroboter festgehalten werden soll.

Orientierung im Greifer: T**.** Eine (4×4)-Transformation, die vom TCP zum Koordinatensystem des Ports P führt. Ist kein Bauteil gegriffen, ist T undefiniert und wird bei Vergleichsoperationen zwischen Zuständen nicht berücksichtigt.

Bauteil-Typ: $t \in \mathcal{T}$**.** $t \in$ {Schraube, Leiste, Würfel, ...}.

Der Zielzustand ist erreicht, wenn Roboter M das Zielbauteil $t_z \in \mathcal{T}$ am richtigen Port ($P_{\text{ist}} = P_{\text{soll}}$) hält und die Z-Achsen des TCP- und des Port-Koordinatensystems parallel bzw. antiparallel sind.

Tab. 7.1 zeigt die rechnerisch mögliche Anzahl an Weltzuständen in Abhängigkeit vom Bauteiltyp. Wenn die Symmetrieeigenschaften der einzelnen Baufixteile (Kapitel 5.2.1) berücksichtigt werden, lassen sich die Zustände reduzieren. In der Praxis fallen weitere Kombinationen weg. Ist z.B. das Objekt außerhalb der Reichweite beider Roboter, verlieren die anderen Dimensionen ihre Bedeutung.

Bauteile	Zustände	
	ohne Symmetrie	mit Symmetrie
Schraube	24	24
Mutter	48	12
Würfel	432	48
3-Loch-Leisten	432	108
5-Loch-Leisten	1200	300
7-Loch-Leisten	2352	588

Tabelle 7.1: Anzahl rechnerisch möglicher Zustände

7.4.3 Lernverfahren

Wie bereits mehrfach erwähnt, wird ein Zustands-/Aktionsgraph mit Hilfe eines Q-Lernverfahrens aufgebaut. Das Lernverfahren bezieht sich auf jeweils einen der zwei beteiligten Roboter. Es werden zwei unabhängige Zustands-/Aktionsgraphen gelernt, die aber zwischen den Robotern ausgetauscht werden können, wenn sich die Rollenverteilung umkehrt. Gelernt wird in Epochen. Eine Epoche ist beendet, wenn entweder das Ziel erreicht ist oder eine maximale Anzahl n von Aktionen durchgeführt wurde. In der konkreten Anwendung muß n deutlich größer sein als die Anzahl an Aktionen, die benötigt werden, um die Gesamtaufgabe zu lösen.

Wie man Tab. 7.1 entnehmen kann, ist die Menge der möglichen Zustände endlich. Zustandsübergänge innerhalb des Verfahrens können aber nichtdeterministisch sein. Zwei Effekte können auftreten:

(a) Da die Orientierungen der Bauteile[13] unerwarteten Veränderungen unterliegen, ändert sich auch der Zustand ohne ersichtlichen Grund. Das wird aber erst bei einem erneuten Greifen festgestellt. Ein möglicher Zeitpunkt, zu dem sich die Orientierung des Bauteils verändern kann, ist allerdings bekannt: das Ablegen. Falls eine Schraube unmittelbar nach dem Abstellen umfällt, was sich mittels Objekterkennung analysieren läßt, kann der Folgezustand des Ablegens korrekt ermittelt werden.

(b) Durch Aktivitäten des jeweils anderen Roboters kann sich ebenfalls der Weltzustand verändern.

Wie oben beschrieben, können auf Grund von Modellannahmen die einzelnen Aktionen entscheiden, ob sie durchführbar sind. In solchen Fällen werden künstliche, negative Rewards der Größe -10 vergeben. Ansonsten werden keine sofortigen Belohnungen verteilt. Erst bei Erreichen des Zielzustandes bekommt das System ein Verstärkungssignal von +100.

7.4.4 Beispiel

In diesem Abschnitt soll anhand eines Beispiels das Greifen (Umgreifen) einer Schraube beschrieben werden, die sich im Greifbereich des Zielroboters befindet, jedoch in liegender Orientierung. Die beiden beteiligten Roboter haben bereits eine Reihe von Lernepochen absolviert, siehe Tab. 7.2. Danach ergeben sich die vier in Abb. 7.8 u. Abb. 7.9 gezeigten

Epoche	**Anzahl der Aktionen des**	
	Zielroboters	helfenden Roboters
1	15	15
2	15	15
3	12	12
4	9	7
5	4	7
6	4	4
7	4	3

Tabelle 7.2: Beim kooperativen Greifen wurden in dem hier beschriebenen Beispiel 7 Lernepochen benötigt, um die in Abb. 7.8 u. Abb. 7.9 gezeigten Zustands-/Aktionsgraphen zu erlernen.

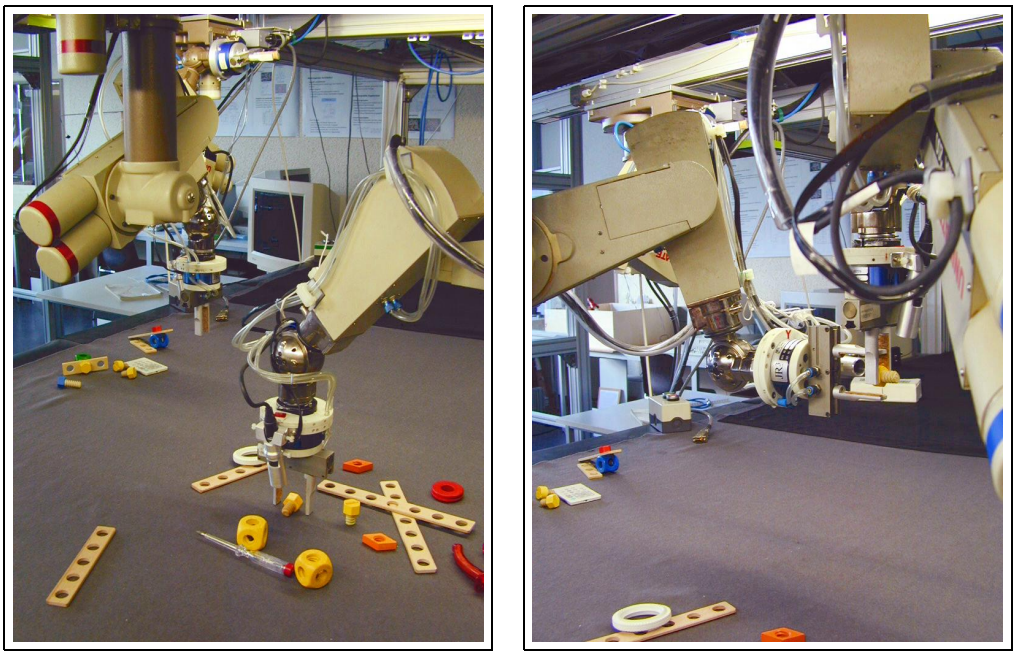
Schritte einer Handlungssequenz. Die folgenden Schritte werden der Reihe nach durchgeführt: **1.** Greifen der Schraube durch den Zielroboter, **2.** Übergeben an den helfenden Roboter, **3.** auf dem Gewinde abstellen und **4.** Greifen in der Sollorientierung durch den Zielroboter.

Aus der Sicht des einzelnen Roboters sieht der gelernte Handlungsablauf allerdings nicht ganz so geradlinig aus. Abb. 7.10 u. Abb. 7.11 zeigen den gelernten Zustands/-Aktionsgraphen. Für den Zielroboter ergibt sich daraus die folgende Abfolge von Aktionen:

(1) Greifen mit einem Winkel von 0^0.

(2) Übergeben an den Helfer mit einem Winkel von 90^0.

[13] insbesondere Schrauben.



(a) Der Zielroboter greift die liegende Schraube.

(b) Beim Übergeben der Schraube vom Zielroboter an den helfenden Roboter wird ein Winkel von 90^0 plus einem bauteilabhängigen Anteil zwischen den Robotern eingenommen.

Abbildung 7.8: In der initialen Situation ist die Schraube nur für den Zielroboter greifbar, befindet sich aber in liegender Orientierung. Um die Schraube im Greifer zu reorientieren, wird sie an den zweiten Roboter übergeben. Dieser hält sie jetzt in der richtigen Orientierung, ist aber nicht der Zielroboter.

(3a) Wenn die Schraube vom helfenden Roboter gehalten wird, warten.

(3b) Wenn die Schraube zwischen den beiden Robotern auf dem Tisch steht, greifen.

Für den helfenden Roboter ergibt sich daraus die folgende Abfolge von Aktionen:

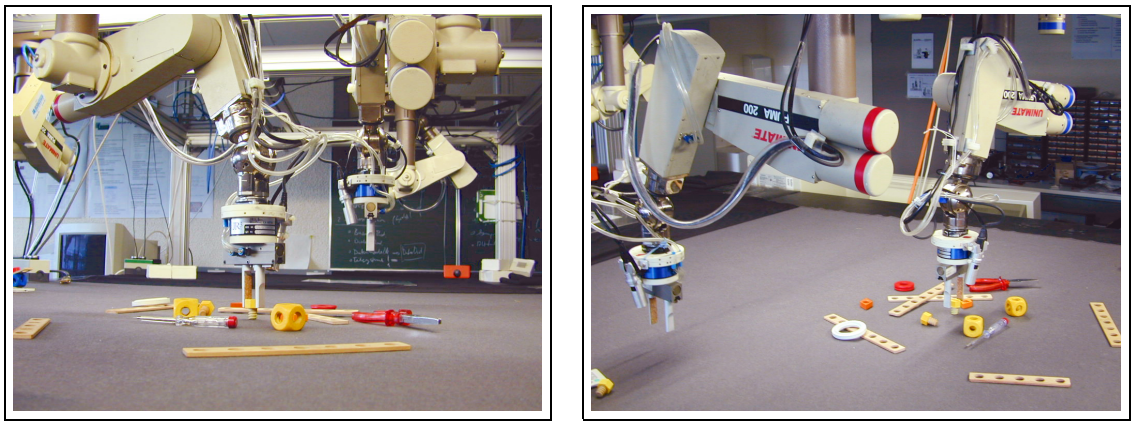
(1) Warten.

(2) Wenn der andere Roboter die Schraube in der (falschen) Orientierung hält, Übernahmeoperation starten.

(3) Wenn die Schraube gegriffen wurde, gleich wieder abstellen, was entweder in den Zielzustand führt, oder aber, solange die Schraube noch auf dem Tisch steht, in einen noch nicht optimal trainierten Zustand.

Optimalität

Optimalität ist immer eine Frage des Kriteriums. Üblicherweise wird die Frage nach dem schnellsten oder kürzesten Weg zum Ziel gesucht. Beim Greifen spielt der Aspekt der Robustheit eine große Rolle.



(a) Der helfende Roboter stellt die Schraube zwischen sich und dem Zielroboter ab.

(b) Der Zielroboter greift die Schraube vom Tisch

Abbildung 7.9: Die zweite Hälfte des Umgreifens besteht darin, die Schraube mit Hilfe des Tisches an den Zielroboter zu übergeben. Dabei stellt der helfende Roboter die Schraube auf dem Gewinde ab und ermöglicht so dem Zielroboter das Greifen.

Robustheit läßt sich hier so definieren: Eine Greifoperation ist um so robuster, je seltener das Bauteil in einer labilen Lage auf dem Tisch steht und je einfacher die durchzuführenden Aktionen sind. Es bleibt also zu definieren, wann eine Operation einfacher als eine andere ist. Dafür gibt es kein allgemeingültiges Maß. Für die hier implementierte Anwendung kann gelten, daß eine kooperative Teilhandlung, also eine, an der mehrere Roboter beteiligt sind, kritischer und damit schwieriger ist als eine mit nur einem Roboter.

Die in obigem Beispiel gefundene Lösung ist in Hinsicht auf Robustheit suboptimal, da die Schraube auf das Gewinde gestellt wird, eine bekanntlich instabile Orientierung. Der unter diesem Gesichtspunkt optimalere Weg wäre, die Schraube zunächst in den gemeinsamen Greifbereich zwischen Zielroboter und helfendem Roboter zu legen (nicht auf das Gewinde), wo sie dann vom Helfer aufgenommen und an den Zielroboter übergeben wird. Wenn man zur Berechnung der Optimalität die Anzahl der an jeder Aktion beteiligten Roboter addiert und für das Stellen einer Schraube aus Gründen der Labilität eine zusätzliche Bestrafung von 1 vergibt, erhält man jeweils für beide Roboter zusammen im ersten Fall einen Wert von 8, im zweiten einen von 7, wobei der kleinere Wert die robustere Handlungssequenz beschreibt.

Um den Lernalgorithmus beim Erkennen der optimalen Lösung zu unterstützen, muß das Verfahren geringfügig abgeändert werden: Die Ablegefunktion muß einen negativen Reward liefern, wenn sich die Orientierung des abgelegten Bauteils unmittelbar nach dem Ablegen gegenüber der berechneten Orientierung ändert. Da die Schraube tatsächlich im Mittel häufiger umfällt als stehen bleibt, wird das Abstellen auf dem Gewinde insgesamt niedriger bewertet, was auf lange Sicht zu einer Präferenzierung des hier als optimaler beschriebenen Handlungsstranges führt.

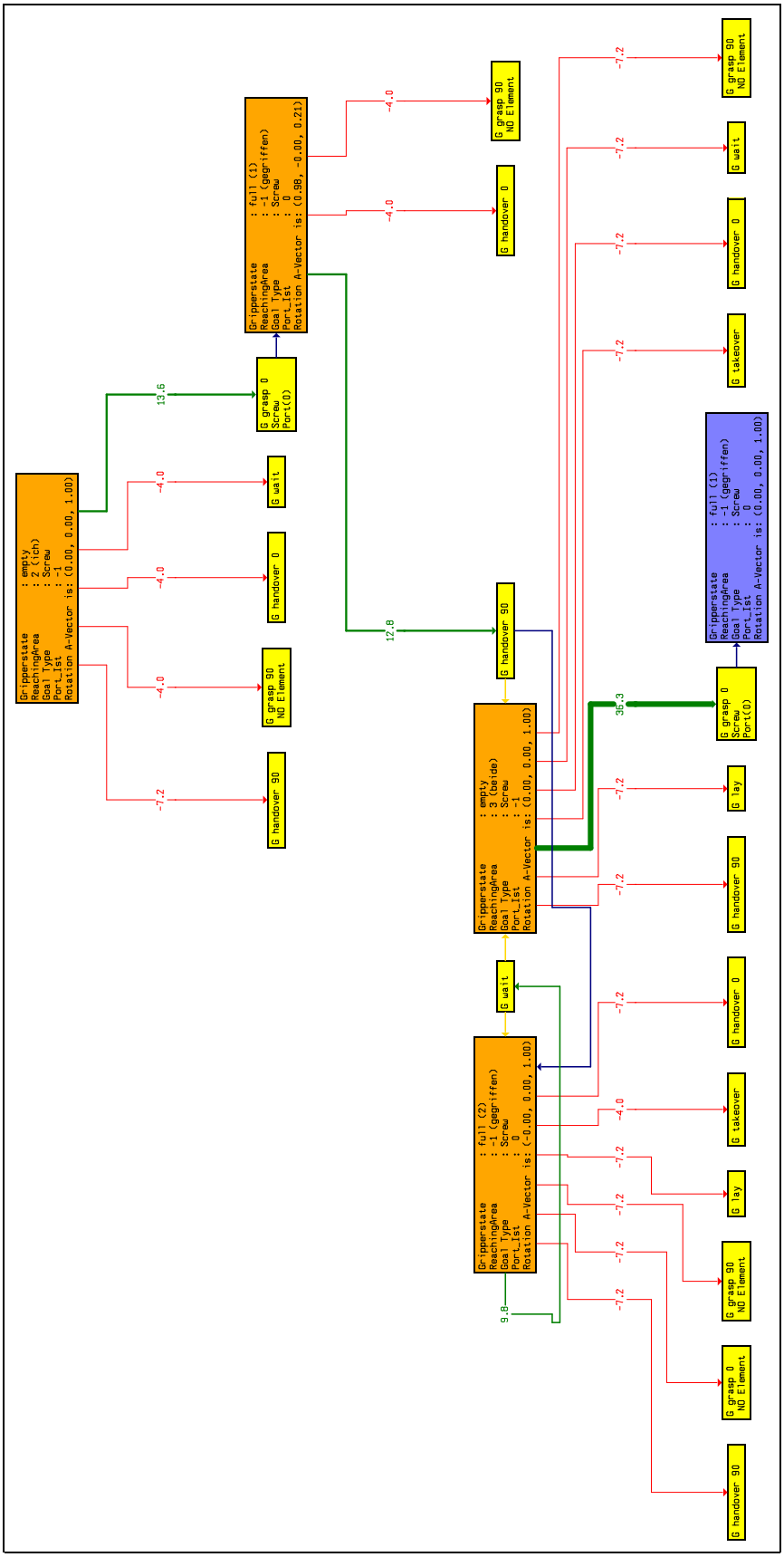


Abbildung 7.10: Automatisch generierte Darstellung des Zustands-/Aktionsgraphen für den Zielroboter. Der Startzustand befindet sich ganz oben, der Zielzustand ist blau hinterlegt.
Aus Gründen der Übersichtlichkeit wurden Kanten, die von einer Aktion zurück in den ursprünglichen Zustand zeigen, nicht dargestellt. Die Bewertungen der Aktionen befinden sich an den jeweiligen Kanten. Rote Kanten besitzen negative, grüne positive Bewertungen. Kanten aus der Menge $\mathcal{V}_a$ sind blau eingefärbt, Kanten, die aus nichtdeterministischen Übergängen stammen, gelb.
Es ist zu beachten, daß es sich hierbei nicht um den vollständigen Zustands-/Aktionsgraphen handelt, sondern nur um den für diese Situation notwendigen Teil.

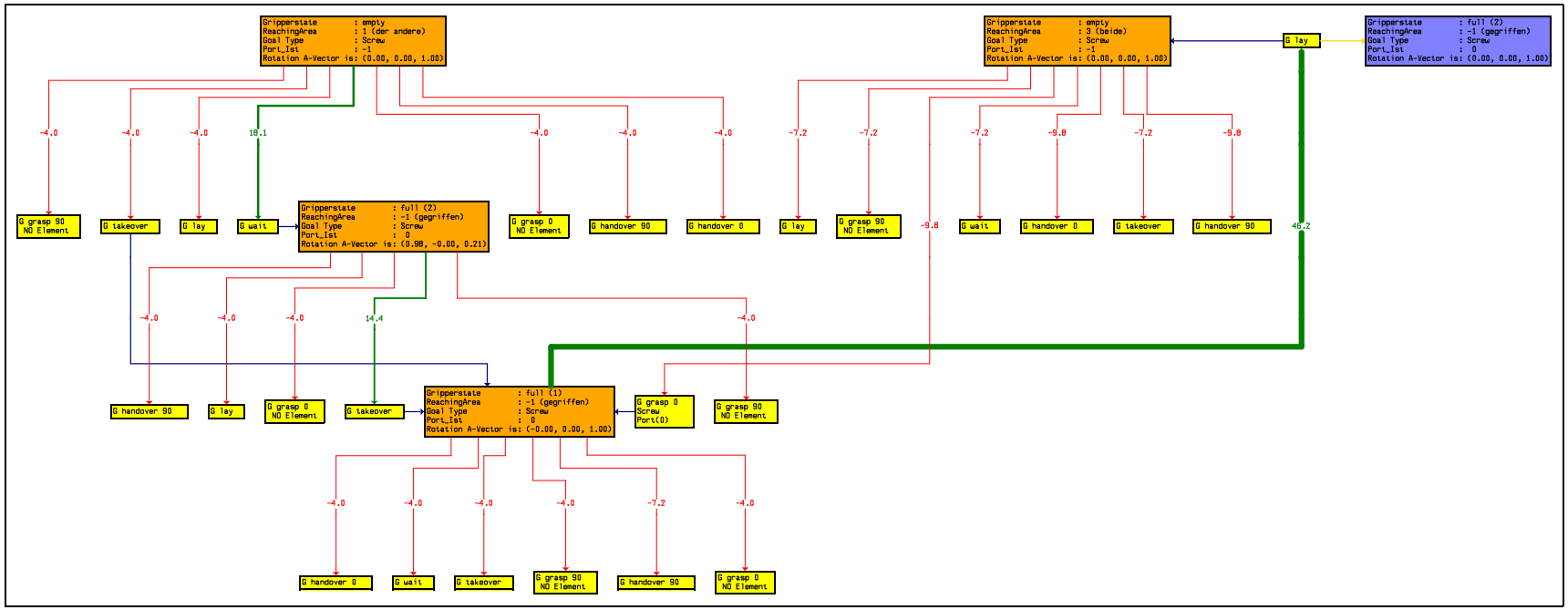


Abbildung 7.11: Automatisch generierte Darstellung des Zustands-/Aktionsgraphen für den helfenden Roboter am Beispiel des kooperativen Greifens einer liegenden Schraube. Der Startzustand befindet sich links oben, der Zielzustand ist blau hinterlegt.
Aus Gründen der Übersichtlichkeit wurden Kanten, die von einer Aktion zurück in den ursprünglichen Zustand zeigen, nicht dargestellt. Die Bewertungen der Aktionen befinden sich an den jeweiligen Kanten. Rote Kanten besitzen negative, grüne positive Bewertungen. Kanten aus der Menge $\mathcal{V}_a$ sind in blau eingefärbt, Kanten, die aus nichtdeterministischen Übergängen stammen, in gelb.
Es ist zu beachten, daß es sich hierbei nicht um den vollständigen Zustands-/Aktionsgraphen handelt, sondern nur um einen kleinen Ausschnitt.

Backstep-punishment

Unter *Backstep-punishment* versteht man eine Methode zum Beschleunigen des Q-Lernverfahrens. Die Idee, die hinter dieser Methode steckt, ist die Überlegung, daß sich ein Lernverfahren schneller weiterentwickeln würde, wenn es Aktionen, die es zu einem Zeitpunkt t durchgeführt hat, zum Zeitpunkt $t+1$ nicht zurücknehmen würde. Wenn man also feststellt, daß das System unmittelbar in den vorangegangenen Zustand zurückkehren will, könnte man diese Operation verbieten. Dadurch entstehen aber Probleme, wenn sich das System in eine Sackgasse manövriert hat. Dann gäbe es keinen Weg zurück. Um das zu vermeiden, könnte man in solchen Fällen eine Ausnahme machen, man würde aber nichts dabei lernen. Damit ist gemeint, daß das System immer wieder in die Sackgasse laufen würde. Das Backstep-punishment löst dieses Problem, indem es für alle unmittelbaren Rückschritte ein künstliches, negatives Reinforcementsignal generiert. Im allgemeinen wird dafür lediglich der zuletzt besuchte Zustand gespeichert. Der Lerner muß jedoch nicht wissen, welche Aktion eine andere zurücknimmt, also quasi invers ist.

Diese Vorgehensweise funktioniert nicht mehr, wenn das Verhalten der Umgebung nichtdeterministisch ist. In diesem Falle könnte es nämlich sein, daß man trotz Ausführung der Umkehroperation nicht wieder im vorherigen Zustand landet, da z.B. ein anderer Roboter etwas in der Welt geändert hat. Um dennoch Backstep-punishment einsetzen zu können, soll hier eine kleine Erweiterung eingeführt werden:

Definiert man für alle Operationen die dazu möglichen inversen Aktionen, kann man den Rückschritt nicht am Zustand, sondern an der Abfolge der Aktionen selbst festmachen. Diese Methode funktioniert im deterministischen wie im nichtdeterministischen Falle um den Preis, daß codiert werden muß, welche Aktionen invers zueinander sind. In Abb. 7.11 kann man den Effekt des Backstep-punishments im Zustands-/Aktionsgraphen sehen. Der Zustand rechts vom Zielzustand beschreibt die Situation, in der die Schraube zwischen den beiden Robotern auf dem Tisch steht, weil der helfende Roboter sie dort hingestellt hat. Wenn der Helfer sie jetzt wieder aufnimmt (*take* 0^0), wird das System erneut in den untersten Zustand gebracht usw. Durch Backstep-punishment wird das erneute Greifen aber bestraft, da es einem Ablegen[14] in besagtem Zustand folgt. Greifen ist die inverse Operation zu Ablegen, Ablegen ist aber nur dann invers zum Greifen, wenn das Greifen an derselben Stelle stattfindet wie das Ablegen, da ja wie bereits erwähnt hier Ablegen heißt, daß man etwas zwischen den Kooperationspartnern ablegt.

7.5 Erlernen einer Montagestrategie

7.5.1 Charakteristika

- ❏ Mehrere Roboter arbeiten zusammen an einem gemeinsamen Ziel; hier der Bau eines Aggregates.
- ❏ Der einzelne Roboter kann nicht den gesamten Zustandsraum beobachten. Es gibt Zustandsänderungen, die nicht von allen Agenten bemerkt werden.
- ❏ Die Menge der möglichen Zustände ist unendlich, und die Anzahl von Lernbeispielen ist verhältnismäßig gering.

[14] Siehe Bewertung der Aktion (take 0^0).

- ❏ Wird das Zielaggregat erreicht, kann ein Langzeitreward gegeben werden. Jeder Agent ist in der Lage festzustellen, ob eines der in Bearbeitung befindlichen Aggregate das Zielaggregat ist.

- ❏ Durch Verwendung des Objektmodells (Kapitel 5) ist es möglich, Vorhersagen über den Sinn gewisser Aktionen zu machen; das ermöglicht, Kurzzeitrewards bzw. Penalties zu generieren. Sie werden als aktionsabhängige Features bezeichnet, da sie lediglich eine Aussage darüber treffen, ob ihrer Meinung nach eine Aktion erfolgversprechend ist oder nicht.

7.5.2 Repräsentation des Weltzustandes

Der Zustandsvektor beschreibt die Sicht des jeweiligen Roboters zu einem bestimmten Zeitpunkt auf die Welt. Einige der Größen sind roboterspezifisch, andere global, werden also von allen Robotern betrachtet und möglicherweise modifiziert. Daraus folgt, daß es sich aus Sicht des einzelnen Roboters um eine nichtdeterministische Welt handelt.

Ein Zustandsvektor s besteht aus drei Größen:

① das zu bauende Zielaggregat;

② das vom Agenten gegriffene Aggregat und die dazugehörigen Greifinformationen (Bauteil, Port);

③ Aggregate/Baugruppen, die von anderen Agenten gegriffen wurden und die dazugehörigen Greifinformationen.

Es soll hier noch auf den dritten Wert des Zustandsvektors eingegangen werden, denn in der Menge der gegriffenen Aggregate wird im Verlauf der Montage das Zielaggregat entstehen. Ein Aggregat besteht aus Bauteilen, die nicht notwendigerweise zusammengesetzt sind. Alle Teile, die gegriffen werden, können an das Aggregat angebaut werden und gehören deshalb bereits nach dem Greifen zum Aggregat. Werden sie wieder abgelegt, ohne montiert zu werden, verschwinden sie wieder aus dem Aggregat. **Begründung:** Nur so ist es möglich, zwei oder mehrere Teilaggregate zu bauen u. anschließend zusammenzufügen, ohne den Zustandsvektor in seiner Dimensionalität zu verändern .

Um das gewünschte Verhalten eines/mehrerer Roboter zu erlernen (Bau des Zielaggregats, Greifen eines Bauteils), muß der Weltzustand alle Variationsmöglichkeiten beinhalten, die das Handeln der Roboter beeinflussen könnten. Wie oben bereits angedeutet, werden Greifinformationen zu dem gegriffenen Aggregat mit abgespeichert. Es reicht selbstverständlich nicht aus zu wissen, welches Aggregat ein Roboter in der Hand hält, es muß auch bekannt sein, wie er es hält, denn davon hängt ab,welche weiteren Montageschritte in welcher Reihenfolge durchgeführt werden können. Die benötigte Information läßt sich im Falle des Baufixszenarios auf das Bauteil und den Port, an dem gegriffen wird, reduzieren. Dadurch wächst der Zustandsvektor jedoch auf fünf Dimensionen. Siehe auch das Beispiel in Tab. 7.3.

Ziel-Aggregat	Vom Agenten gehaltenes Aggregat	Bauteil	Portnummer	Liste gegriffener Aggregate
Leitwerk	leer	ungültig	ungültig	Leiste (5)
Leitwerk	Schraube (gelb)	Schraube (gelb)	0	Leiste (5), Würfel (rot), Schraube (gelb),
Leitwerk	Schraube (gelb)+ Leiste (5)	Schraube (gelb)	0	Schraube (gelb)+ Leiste (5), Würfel (rot)
Propeller	leer	ungültig	ungültig	Schraube (gelb)+ Leiste (3) , Leiste (3)
Propeller	Würfel (rot)	Würfel (rot)	2	Schraube (gelb)+ zwei Leisten (3) , Würfel (rot)
Propeller	leer	ungültig	ungültig	Schraube (gelb)+ zwei Leisten (3)+ Würfel (rot)

Tabelle 7.3: Beispiele für unterschiedliche Weltzustände. Die ersten drei Zustände beschreiben Schritte beim Bau eines Leitwerks, die letzten drei beim Bau eines Propellers („**+**“ bedeutet, daß die damit verbundenen Bauteile zu demselben Aggregat gehören, „ **,**“ steht zwischen Bauteilen unterschiedlicher Aggregate. Die in Klammern gesetzten Größen hinter den Bauteilen sind Attribute der Bauteile, also z.B. Leiste (5) ist eine Leiste mit fünf Löchern).

Aus dieser Definition eines Zustandsvektors s ergibt sich, daß die verschiedenen Roboter nicht die gleiche Weltsicht haben, da sie sich in den Dimensionen zwei u. drei unterscheiden.

7.5.3 Repräsentation von Aktionen

Aktionen sind, von außen betrachtet, Handlungsprimitiva eines einzelnen Roboters, z.B. Greifen, Ablegen, Warten, Anschrauben, Aufstecken, Festhalten, Loslassen. Ziel des Lernverfahrens ist es, für jeden Roboteragenten eine Abfolge von Aktionen zu finden, die als Gesamtergebnis das gewünschte Zielaggregat liefert, siehe Abb. 7.12. Dazu wird jeder mögliche Zustand s mit einer oder mehreren bewerteten Aktionen a verknüpft (Zustands-/Aktionsgraph).

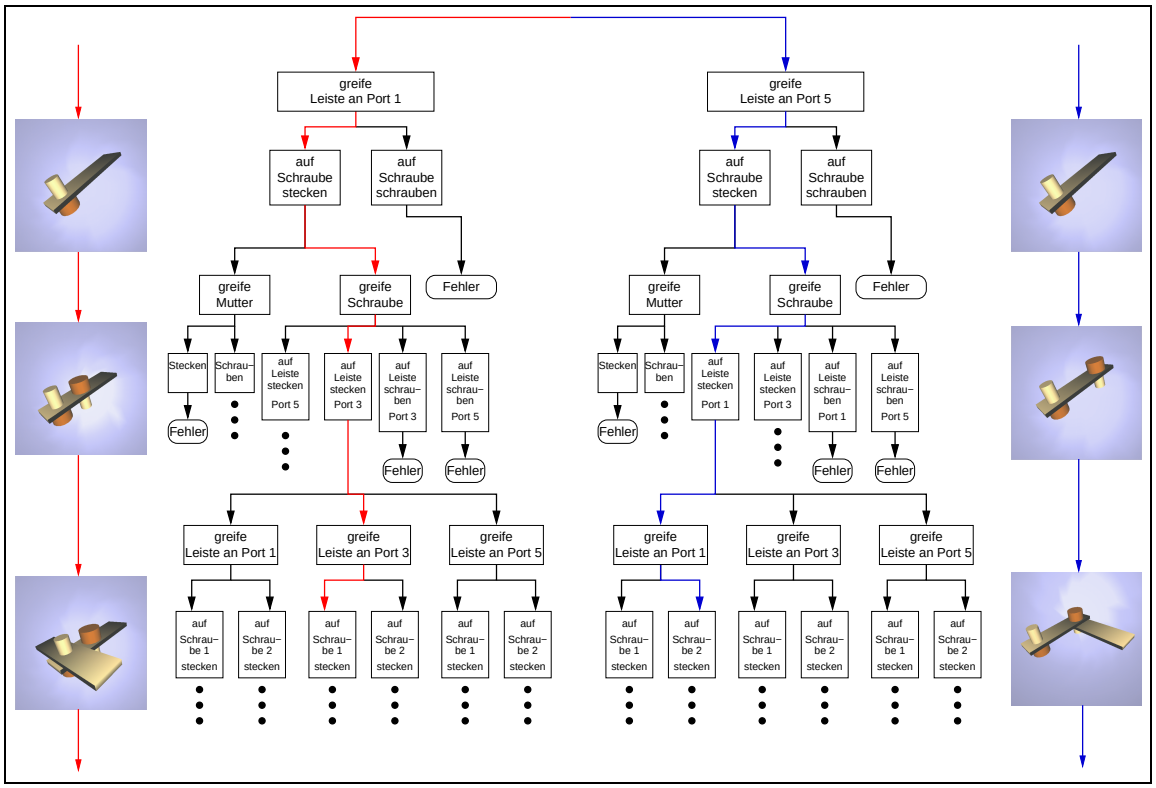


Abbildung 7.12: Theoretischer Aktionsbaum für einen Roboter. Um das Beispiel überschaubar zu halten, ist nur ein Teilbaum dargestellt. Die Aktionen anderer Roboter sind ebenfalls nicht kodiert. Links und rechts werden die den farbigen Pfaden entsprechenden Aggregate gezeigt.

Parametrisierung

Die Aktionen sind jedoch zum Teil mit Parametern versehen, wodurch weitere Aktionen entstehen, die bei der Auswahl berücksichtigt werden müssen. In einem ersten Schritt wird aus der Menge der Aktionsklassen eine Aktion ausgewählt und erst dann werden die konkreten Parameter festgelegt. Die Bewertung der Aktionsklassen entspricht der höchsten Bewertung innerhalb der Klasse. (Hinter einer Aktionsklasse verbergen sich, wie bereits erwähnt, mehrere mögliche Aktionen mit stark abweichenden Bewertungen.)

Wenn-Dann Regeln

Um das Lernverfahren zu beschleunigen, wird nach der Auswahl einer Aktionsklasse zunächst bewertet, ob eine Aktion aus dieser Klasse in der vorliegenden Situation überhaupt durchführbar ist (precondition). Dazu wird Weltwissen verwendet, das in Form von Wenn-Dann Regeln vorliegt. Wenn die folgenden Bedingungen erfüllt sind, dann wird die gewählte Aktion nicht ausgeführt, sondern gleich die gesamte Aktionsklasse bestraft:

WENN Bauteil im Greifer UND Aktion = grasp
WENN Bauteil im Greifer UND die letzte durchgeführte Aktion wahr (screw ODER plug ODER hold) UND Aktion = lay
WENN Bauteil im Greifer UND die letzte durchgeführte Aktion wahr (screw ODER plug ODER hold) UND (Aktion = plug ODER screw)
WENN Bauteil im Greifer UND Aktion = hold
WENN Bauteil im Greifer UND die letzte durchgeführte Aktion wahr nicht (screw ODER plug ODER hold) UND Aktion = release
WENN kein Bauteil im Greifer UND Aktion = grasp UND das Bauteil ist nicht im Zielaggregat enthalten
WENN kein Bauteil im Greifer UND Aktion = wait
WENN kein Bauteil im Greifer UND Aktion = lay
WENN kein Bauteil im Greifer UND (Aktion = plug ODER screw)
WENN kein Bauteil gegriffen wurde UND die letzte durchgeführte Aktion wahr (grasp ODER hold) UND Aktion = hold
WENN kein Bauteil im Greifer UND Aktion = release

Man betrachte dazu Abb. 7.15. Der oberste Zustand ist mit *„No Aggregat“* bezeichnet, was besagt, daß der entsprechende Roboter kein Bauteil hält. Er ist mit einer Reihe von Aktionen verbunden, unter anderem einer *screw*-Aktion (ganz links). Der Parameter ist hier auf Aggregat gesetzt, was kein konkretes Aggregat ist, sondern einen Platzhalter für alle existierenden Aggregate darstellt.

Die Verwendung solcher Regeln entspricht einer vereinfachten Form der in (LEVESQUE ET AL., 1994) vorgestellten Axiomatisierung von Aktionen und ihrem Effekt auf das Situationskalkül (MCCARTHY AND HAYES, 1969).

Weiteres Modellwissen

Weiteres Modellwissen ist in den jeweiligen Aktionsprimitiva integriert, das Q-Lernverfahren hat sich z.B. für die Aktion „*verschrauben*“ entschieden. Im momentanen Zustand wurde bereits in einem der früheren Lerndurchläufe eine Greifoperation mit einer Dreilochleiste an Port (3) durchgeführt und natürlich bewertet. Angenommen, das Lernverfahren ist in einem Explorationsmodus und möchte die neue Aktion (*verschrauben*) ausprobieren. Es muß entschieden werden, welches Aggregat/Bauteil und welcher Port gewählt werden sollen – damit ist nicht das Bauteil gemeint, das bereits vom Roboter gehalten wird, sondern das Bauteil, an das montiert werden soll. Die Antwort darauf kann im Zustandsvektor gefunden werden: Nur wenn ein anderer Roboter z.B. eine Schraube an Port 0 hält, macht es Sinn, die Montageaktion auszuwählen. Der Aktionsraum ist gewissermaßen dynamisch, er verändert sich mit den Handlungen des Roboters, insbesondere können die Handlungen anderer Roboter die möglichen Aktionen beeinflussen.In dieser Arbeit wurde auf die Bildung des jeweiligen Aktionsunterraumes verzichtet. Stattdessen werden Aktionen, die sich nicht im Unterraum befinden, künstlich bestraft. Diese Methode ist zwar, sowohl in Hinsicht auf die benötigte Zeit wie auch auf den benötigten Speicherplatz, etwas ineffizienter, sie ist aber weniger komplex in der Realisierung.

Automatisch generierte Darstellung von Zustands-/Aktionsgraphen

Während des Lernprozesses kann der Zustands-/Aktionsgraph visualisiert werden (siehe Abb. 7.13). Da immer mehrere Roboter an einem Montageprozeß beteiligt sind, müssen mehrere Graphen parallel betrachtet werden. Die Zustände sind orange, die Aktionen gelb hinterlegt. Der momentane Zustand des Systems ist in blau eingefärbt. Die Kanten aus der Menge $\mathcal{V}_s$ sind schwarz (Bewertung = 0), rot (Bewertung negativ) und grün (Bewertung positiv) eingefärbt. Kanten aus der Menge $\mathcal{V}_a$ sind blau oder, wenn sie aus einem nichtdeterministischen Übergang stammen, gelb dargestellt. Bei Aktionen, die unmittelbar in den Zustand zurückführen, aus dem sie kommen, werden die rückführenden Kanten ($\mathcal{V}_a$) nicht dargestellt, um den Graphen nicht unnötig zu verkomplizieren.

In Abb. 7.13(b) ist der Effekt einer WENN - DANN Regel zu erkennen: Der zweite Zustand von oben entsteht durch das Greifen einer Leiste. Danach versucht der Lerner, erneut eine Greifoperation durchzuführen, was durch eine der Regeln (s.o.) als unsinnig erkannt wird und damit zu einer negativen Bewertung der Aktion führt, ohne daß die Aktion tatsächlich ausprobiert werden muß.

In Abb. 7.13(a) ist zu sehen, wie der Roboter aus dem zweiten Zustand von oben heraus eine Aktion *Plug*-90^0 ausprobiert. Nachdem sich die beiden Roboter über die beteiligten Bauteile und Montagestellen verständigt haben, stellen sie fest, daß das resultierende Bauteil nicht Bestandteil des Zielaggregates ist und brechen die Aktionsabarbeitung frühzeitig ab.

7.5.4 Aktionsauswahl

Die Aktionen werden im Explorationsmodus nach einer gewichteten Wahrscheinlichkeit bestimmt, siehe auch Abschnitt 4.2.5. Im einzelnen kann für das Q-Lernen eine der folgenden Funktionen verwendet werden:

(7.2) $$\text{Prob}(z,a) = \frac{k^{Q'(z,a)}}{\sum_i k^{Q'(z,a)}}$$

(7.3) $$\text{Prob'}(z,a) = \frac{e^{\frac{Q'(z,a)}{T}}}{\sum_i e^{\frac{Q'(z,a)}{T}}}$$

Für diese Arbeit wird ein Verfahren gemäß Gleichung 7.2 verwendet. Mit dieser Gleichung läßt sich jeder Aktion eine Wahrscheinlichkeit zuweisen.

no action	grasp	lay	plug-0^0	plug-90^0	screw	hold	release
10%	5%	20%	6%	4%	10%	10%	35%

Tabelle 7.4: Beispiel für die Verteilung von Wahrscheinlichkeiten auf Aktionen

Das Auswahlverfahren sieht dann wie in List. B.3.1.1 aus, wobei `CalculateProbability(i,K)` die Berechnung der Wahrscheinlichkeit für die i-te Aktion[15] gemäß Gleichung 7.2 oder Gleichung 7.3 ist.

[15] Laut Tab. 7.4.

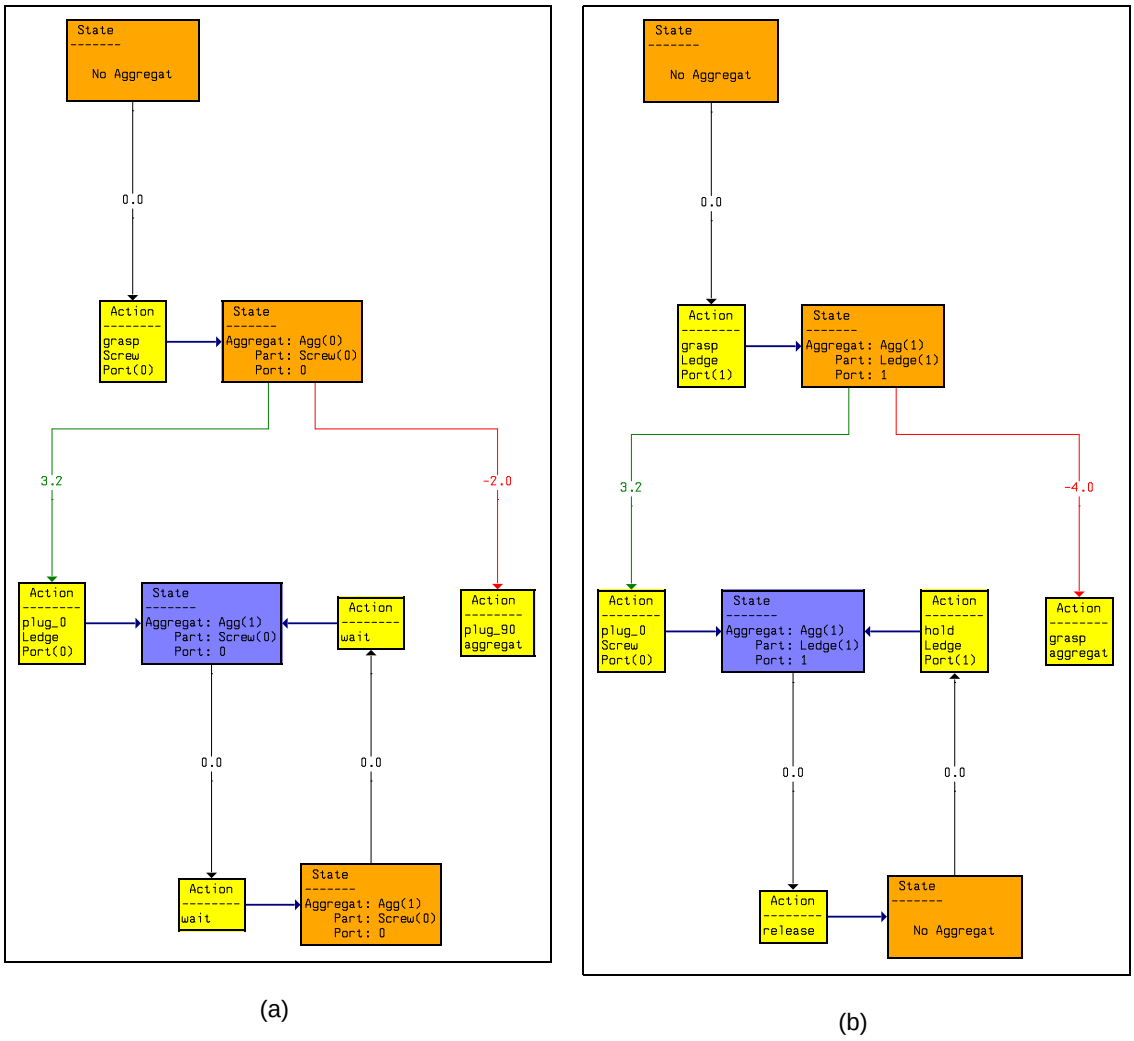


Abbildung 7.13: Die zwei hier abgebildeten Zustands-/Aktionsgraphen wurden automatisch während eines Lernprozesses zur Montage des in Abb. 5.5 dargestellten Aggregates generiert.

7.5.5 Aktionen mit mehreren Robotern

Es gibt einige Aktionen, an denen nur ein Roboter beteiligt ist (*Ablegen, Loslassen, Warten und, solange das Bauteil greifbar ist, auch Greifen.*). Bei den Montageoperationen müssen jedoch mehrere (2) Roboter kooperieren. Wie kommen solche Kooperationen zustande, und was für Konsequenzen für das Lernverfahren hat der Zwang zur Kooperation? Aktionen, an denen nur ein Manipulator beteiligt ist, sind relativ unproblematisch, da nur der betroffene Arm aus seiner Aktion etwas lernen kann. Sind mehrere Manipulatoren an einer Aktion beteiligt, so muß es nicht nur zu einem räumlichen Rendezvous, sondern auch zu einem Rendezvous der Handlungen kommen. Die Teilhandlungen benötigen eine gewisse Form von Synchronisation. Neben dem Aufbau eines Rendezvous ist auch das Auflösen einer synchronen Handlungssequenz zu spezifiziert. Um ein Gefühl dafür zu bekommen, sollen die betroffenen Operationen im einzelnen betrachtet werden:

Montageaktionen: Hier gibt es immer einen aktiven und einen passiven Agenten. Der eine steckt oder schraubt, der andere hält sein Aggregat an einer ausgehandelten Position fest und ist bestenfalls nachgiebig gegenüber großen Krafteinwirkungen. Der aktive Agent prüft den Erfolg der Aktion und beendet das Rendezvous entweder mit Erfolg oder mit einem Fehler. Wer die Aktion initiiert, ist prinzipiell egal, die Rolle, die die jeweiligen Roboter übernehmen (aktiv, passiv), ist situationsabhängig. Folgende Möglichkeiten bestehen:

- ❏ Der Roboter mit dem nehmenden Port ist stets aktiv, der mit dem gebenden Port ist passiv.
- ❏ Der Roboter mit dem gebenden Port ist stets aktiv, der mit dem nehmenden Port ist passiv.
- ❏ Der Roboter mit dem schwereren Aggregat ist passiv.
- ❏ Wird ein Aggregat von mehreren Roboteragenten gehalten, z.B. nach einer Montageoperation, und es soll eine weitere Montageoperation durchgeführt werden, dann sind diejenigen Roboter passiv, die gemeinsam ein Aggregat halten.

Aktionen zwischen Agenten (Rendezvous)

In diesem Abschnitt sollen die Schritte erläutert werden, die man benötigt, um Aktionen auszuführen, an denen mehrere Roboter beteiligt sind. Da sie im Normalfall unabhängig (unsynchronisiert) arbeiten und lernen, sind dazu einige Maßnahmen notwendig.

Problemdefinition

Unter einem Rendezvous soll der Mechanismus verstanden werden, mit dessen Hilfe zwei Roboter ihre Aktionen aufeinander abstimmen. Dieser Abschnitt ist sehr technisch und basiert auf Details des verwendeten Agentensystems.

Das Problem bei der Initiierung einer Kooperation zwischen zwei Robotern ist der bis dahin zeitlich unkoordinierte Verlauf der Aktivitäten. Um ein Rendezvous zu realisieren, muß eine zeitliche Synchronisation erreicht werden. Zusätzlich sollen Roboter, die um ein Rendezvous bitten, von dem Gebetenen bevorzugt behandelt werden, um das Lernverfahren nicht zu stark auszubremsen. Ein Rendezvous käme sonst nur zustande, wenn beide Lerner zum selben Zeitpunkt dieselbe Aktion auswählen.

Eine weitere Schwierigkeit besteht darin, daß die Anwendungen unter Umständen gleichzeitig um ein Rendezvous bitten und keine Antwort bekommen, da jeder auf die Antwort des anderen wartet. Eine solche Situation muß erkannt und umgangen werden.

Lösungsansatz mittels nebenläufigem Prozeß

Der Lern-Prozeß ist ein Unterprozeß des Agenten, der solange die Hauptlernschleife durchläuft, bis das Zielaggregat erreicht ist. Dabei werden Aktionen ausgewählt, durchgeführt, das Ergebnis wird bewertet, gegebenenfalls ein Reward bestimmt und der Zustands-/Aktionsgraph aktualisiert.

Da Rendezvous immer im Zusammenhang mit Aktionen auftreten, muß der Lernprozeß in der Lage sein, sie zu handhaben. Folgender Ablauf wird benötigt:

① Vor Auswahl einer neuen Aktion wird geprüft, ob ein Rendezvous Request eingegangen ist. Dazu wird eine spezielle Fifo (`RVFifo`) verwendet. Wenn nicht, weiter bei (2.) , sonst weiter bei (8.) .

② Nach dem Standardverfahren des Q-Lernens wird eine Aktion ausgewählt, z.B. **Montage** (2a.) oder **Warten** (2b.) .

(a) Es wird eine Montageaktion initiiert, die zunächst einen Rendezvous Request an einen sinnvoll erscheinenden Partner versendet und bei (3.) weitermacht. Existiert kein solcher Rendezvous Partner, so wird die Aktion als mißlungen betrachtet, entsprechend bewertet und zu (1.) gesprungen.

(b) Es wird die Warteaktion ausgeführt. Es soll solange gewartet werden, bis sich etwas am Weltzustand ändert. Dazu blockiert die Aktion den Programmfluß auf einer sogenannten Condition (`WorldStateChanged`). Hat sich der Zustandsvektor durch Aktionen anderer Agenten verändert, wird die Warteaktion beendet und zu (1.) gesprungen. Auch Rendezvous Requests führen zum Verlassen der `WorldStateChanged` Condition. Die Condition wird vom Ereignismanager (Kapitel 2.2.2) gesetzt.

③ Es wird auf eine Antwort gewartet und wiederum die `RVFifo` verwendet. Kommt in der `RVFifo` eine Mitteilung an, so kann es sich um die Antwort auf obigen Request (2a.) handeln, dann weiter bei (4.) , oder es handelt sich um den Request eines anderen Agenten, dann weiter bei (5.) .

④ Die Antwort wird analysiert. Ist sie positiv, wird eine Verbindung zu dem in der Antwort vermerkten Kanal aufgebaut (Client), sonst wird die Aktion als fehlgeschlagen betrachtet, entsprechend bewertet, dann weiter bei (1.) .

⑤ Entweder handelt es sich um einen Request des Roboters, an den man seinen eigenen Request in (2a.) gerichtet hatte, dann weiter bei (7.) , sonst weiter bei (6.) .

⑥ Dem Requester wird eine Ablehnung zugesendet. Sie weist darauf hin, daß man momentan selbst in Verhandlung ist, was wiederum den Reward des Empfängers dieser Ablehnung beeinflussen soll; dann zurück zu (3.) .

⑦ Offensichtlich haben sich die Requests der beiden Agenten zeitlich überschnitten. Auch der andere Roboter empfängt soeben einen Request, obwohl er eine Antwort erwartet hat. Es kann ein Verbindungsaufbau initiiert werden, aber es muß entschieden werden, welcher der Agenten die Rolle des Servers übernimmt und wer Client ist. Eine eindeutige Lösung ist die Verwendung der Agentennamen. Derjenige Name, der gemäß ASCII - Norm der größere ist, wird zum Server (10.) , der andere Roboter übernimmt die Rolle des Clients und springt zurück zu (3.) .

⑧ Entweder handelt es sich um einen Request, dann weiter bei (9.) , oder es liegt eine Fehlersituation vor (PANIC).

⑨ Der Request wird analysiert und positiv (10.) oder negativ beantwortet. In letzterem Falle weiter bei (1.) .

⑩ Es wird ein Kanal geöffnet (Server) und eine Antwortmessage versendet, in der die Adresse des Kanals steht. Danach wird auf den Verbindungsaufbau gewartet (`accept`).

Um einen wartenden Lern-Prozeß wieder aufzuwecken, wird die Mitarbeit des Ereignis-Managers benötigt: Alle Rendezvous Requests und Rendezvous Antworten sind Standardbotschaften (Kapitel 2.3), landen also im Ereignis-Manager. Führt der Lern-Prozeß momentan eine Aktion (ausgenommen die Warteaktion) aus, so kann der Request gleich abgelehnt werden. Sonst wird die Message an die `RVFifo` übergeben und ein Signal auf die `WorldStateChanged` Condition gegeben, um eine möglicherweise laufende Warteaktion zu beenden.

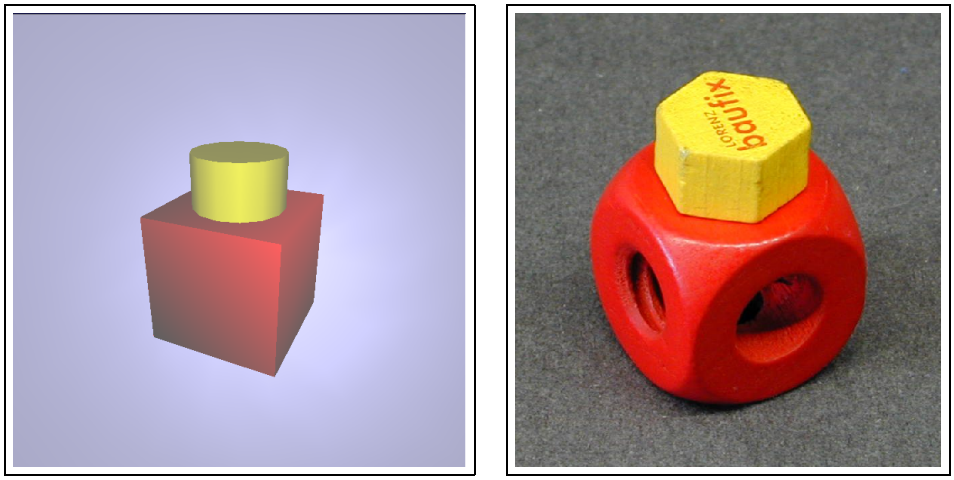
7.5.6 Zustandsübergänge und Rewards

Die dritte Dimension innerhalb des Zustandsvektors (Kapitel 7.5.2) kann sich jederzeit ändern. Nach dem Greifen eines Objektes ändern sich Greiferzustand u. Objektorientierung entsprechend. Nach dem Übergeben oder Ablegen geht der Status in (leer,leer,?) über. Nach einer Fügeoperation (Stecken, Schrauben) ist der Übergang unklar, entweder es wird gelernt, wer das Bauteil los läßt und wer es weiterhin hält, oder es wird festgelegt, daß immer der mit dem nehmenden Port das Bauteil los läßt. Die zweite Vorgehensweise stellt jedoch eine recht zweifelhafte Heuristik dar, weshalb die erste verwendet wird.

Der maximale Reward wird vergeben, wenn einer der beteiligten Roboter das Zielaggregat konstruiert hat. Bei der Montage gibt es nur dann einen Reward, wenn etwas zusammengefügt wurde, das auch ein Teil des Zielaggregates ist. Da auf Demontageoperationen verzichtet werden soll, wird vor einer Montageoperation getestet, ob sie zu einem Teil des Zielaggregates führen wird. Ist dem nicht so, wird die Aktion bestraft und nicht weiter ausgeführt. Alternativ könnte man das Lernverfahren an dieser Stelle beenden und von vorn (mit einer neuen Epoche) beginnen. In diesem Falle hätte man sich in eine Sackgasse manövriert. Beim Greifen gibt es negativen Reward, wenn das Bauteil nicht im Zielaggregat enthalten ist oder die entsprechende Anzahl bereits im entstehenden Aggregat verbaut ist, bzw. von anderen Robotern gehalten wird. Ablegen, Festhalten und Loslassen gibt im Normalfall keinen Reward, es sei denn, die Aktionen werden auf Grund von backstep-punishment bestraft.

7.5.7 Ergebnisse

Die einfachste denkbare Form einer Montage: Verschrauben zweier Bauteile



(a) Simuliertes Zielaggregat (b) Reales Zielaggregat

Abbildung 7.14: Einfaches Zielaggregat, bestehend aus Schraube und Würfel

Ein sehr einfaches Zielaggregat stellt Abb. 7.14 dar. Die Aktionssequenz besteht im günstigsten Falle aus nur zwei Aktionen pro Roboter: 1. *Greifen*, 2. *Schrauben*. Da das Zielaggregat sehr einfach strukturiert ist, werden zum Erlernen lediglich fünf Epochen benötigt. Jede Epoche besteht aus maximal 50 Aktionen, die aber in keinem Fall benötigt werden (Tab. 7.5). Die bei diesem Lernprozeß entwickelten Zustands-/Aktionsgraphen werden in Abb. 7.15 und

Roboter	1.	2.	3.	4.	5.
1	15	19	3	3	3
2	28	25	3	3	3

Tabelle 7.5: Anzahl der Lernschritte beim Erlernen einer sehr einfachen Montage.

Abb. 7.16 gezeigt. Wie (Tab. 7.5) erkennen läßt, benötigen die Roboter drei statt der im optimalen Falle notwendigen zwei Aktionsschritte bis zum Erreichen des Zielzustandes. Aus den Graphen läßt sich die Ursache dafür erkennen. In Abb. 7.15 wird nach dem Greifen einer Schraube zunächst eine Warteaktion ausgeführt, während in Abb. 7.16 der Roboter gleich mit einer Warteaktion beginnt und erst, wenn der andere Roboter eine Schraube gegriffen hat, seinerseits mit dem Greifen eines Würfels beginnt.

Montage eines Aggregates aus drei Bauteilen ohne Umgreifen

Eine bereits etwas schwierigere Aufgabe ist der Bau eines Aggregates mit drei Bauteilen (Abb. 7.17), da die kombinatorischen Möglichkeiten bei der Auswahl des zu greifenden Bauteils steigen. Die Aktionssequenz besteht im günstigsten Falle aus fünf Aktionen pro Robo-

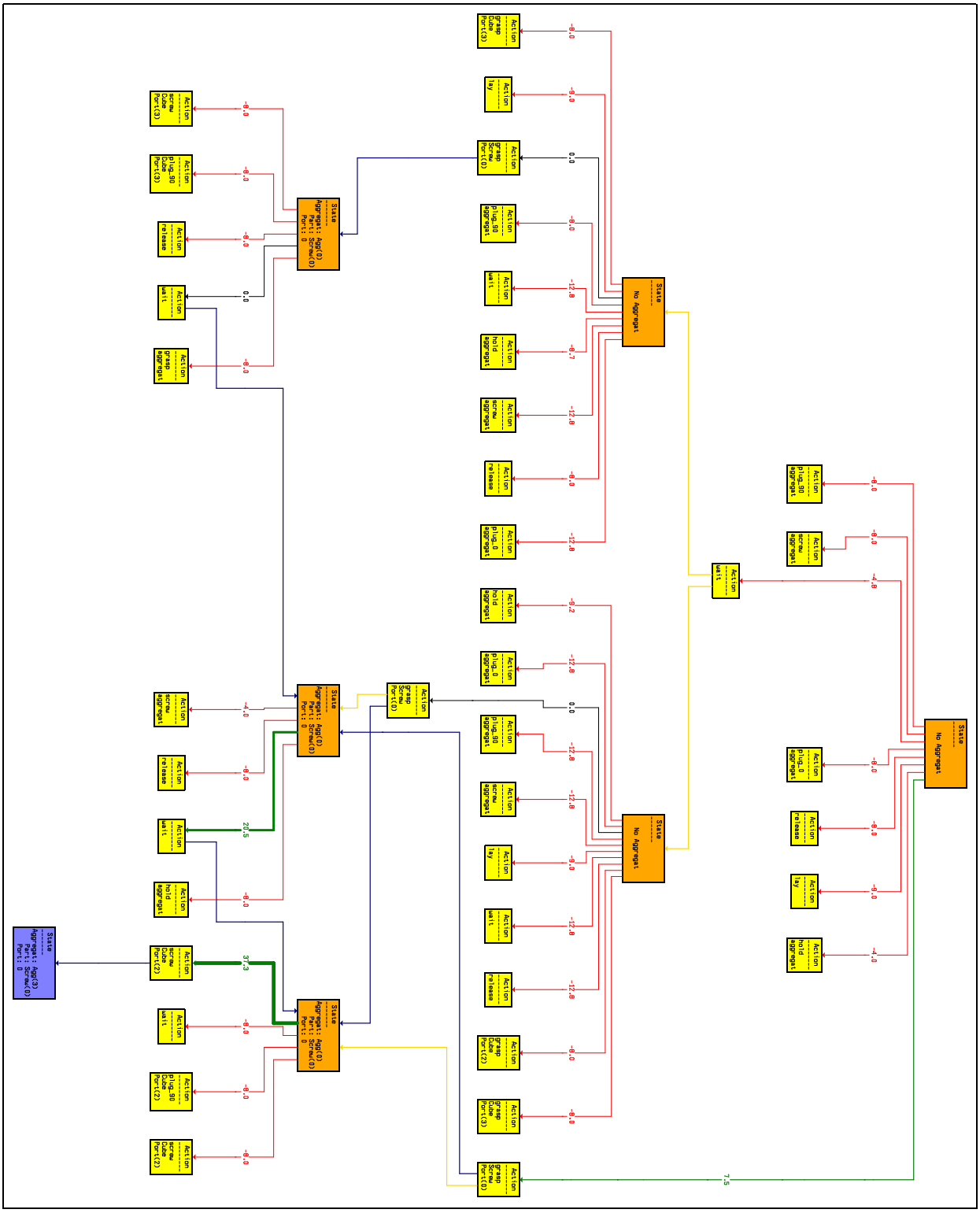


Abbildung 7.15: Zustands-/Aktionsgraph des roten Roboters

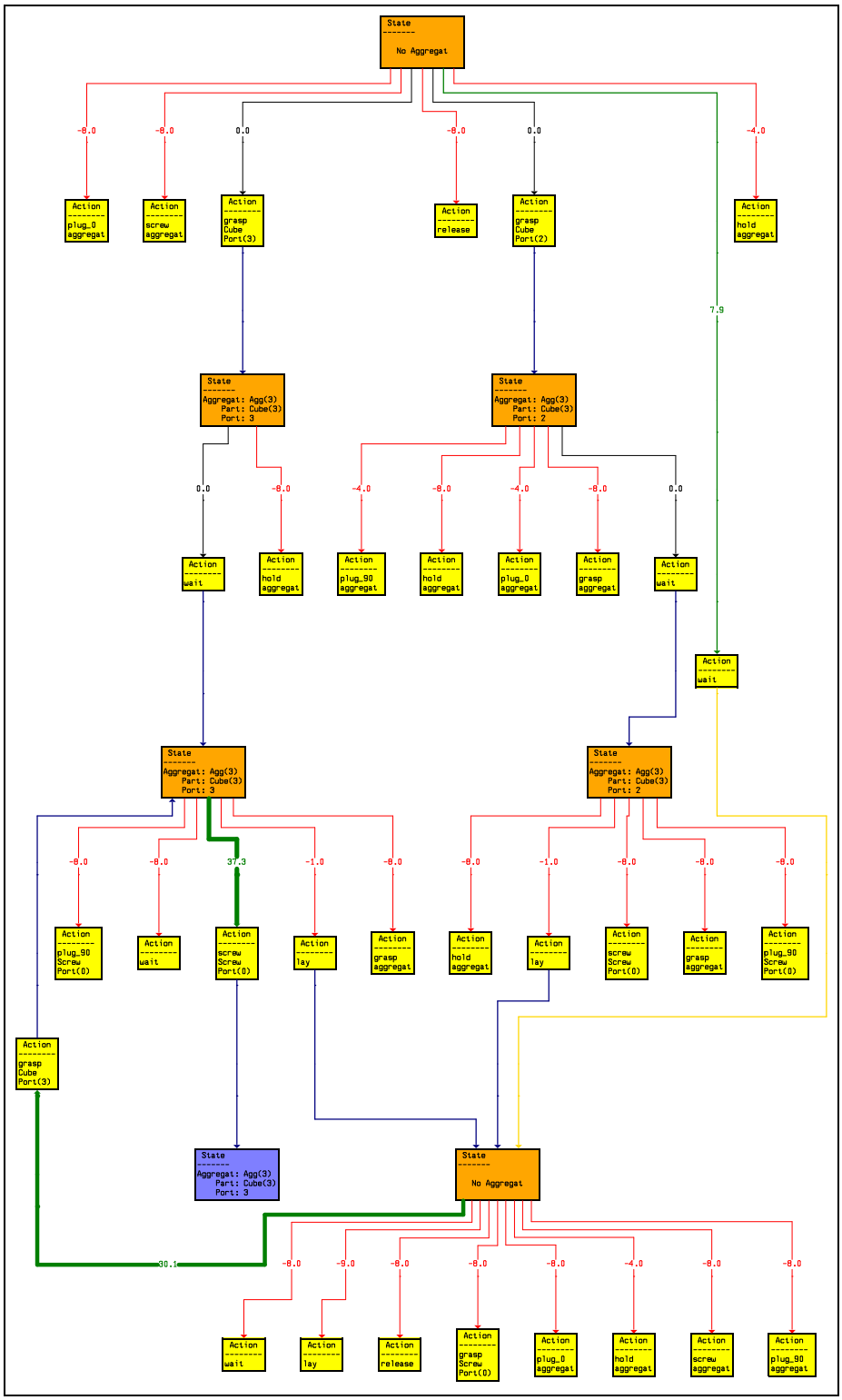
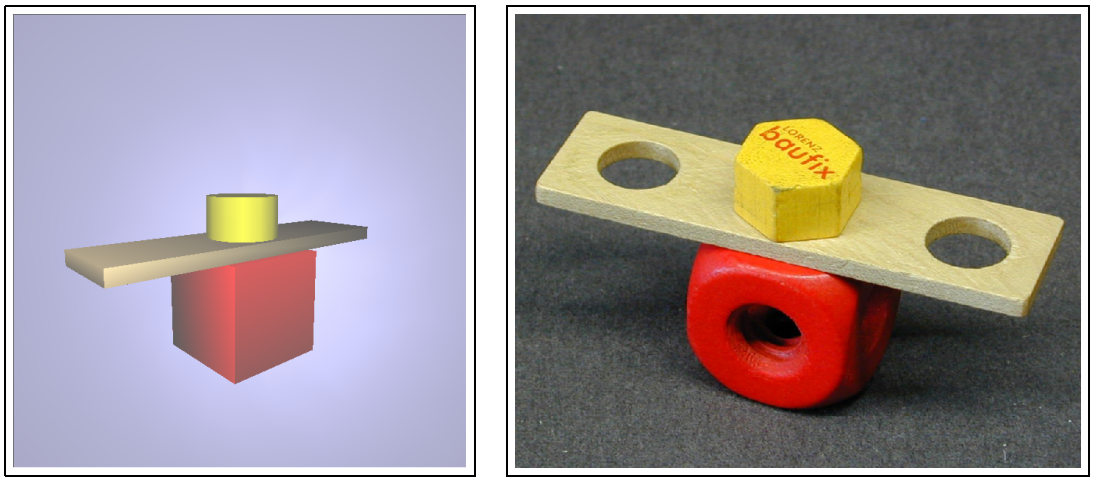


Abbildung 7.16: Zustands-/Aktionsgraph des blauen Roboters



(a) Simuliertes Zielaggregat (b) Reales Zielaggregat

Abbildung 7.17: Zielaggregat, bestehend aus Schraube. Leiste und Würfel

ter (Tab. 7.7). Für das Erlernen einer derartigen Montage werden neun Epochen mit bis zu 100 Aktionen benötigt (Tab. 7.6). Die dieser Tabelle zugrundeliegenden Graphen werden in Abb. 7.19 und Abb. 7.18 gezeigt.

Agent	1.	2.	3.	4.	5.	6.	7.	8.	9.
rot	100	100	50	100	7	5	4	4	4
blau	100	100	57	100	8	11	6	5	5

Tabelle 7.6: Anzahl der Lernschritte beim Erlernen einer sehr einfachen Montage

Aktions-nummer	erster Roboter	Bewer-tung	zweiter Roboter	Bewer-tung
1.	greifen (Leiste, Port 3)	12.6	greifen (Schraube,Port 0)	8.5
2.	stecken - 90^0	19.3	stecken - 90^0	21.5
3.	loslassen	21.7	warten	24.2
4.	greifen (Würfel, Port 3)	33.0	warten	33.7
5.	schrauben	37.4	schrauben	37.4

Tabelle 7.7: Die Aktionsfolge bei der Montage des Aggregates Abb. 7.17. Es werden für die Montage 5 Aktionen benötigt.

Montage des Aggregates Abb. 5.5

Das in Abb. 5.5 gezeigte Aggregat besteht aus sechs Bauteilen, die durch zwei Liaisons miteinander verbunden sind (Abb. 5.10). Für eine erfolgreiche Montage ist mindestens eine Umgreifoperation notwendig. Da Umgreifen keine explizite Aktion ist, muß sie aus *release*

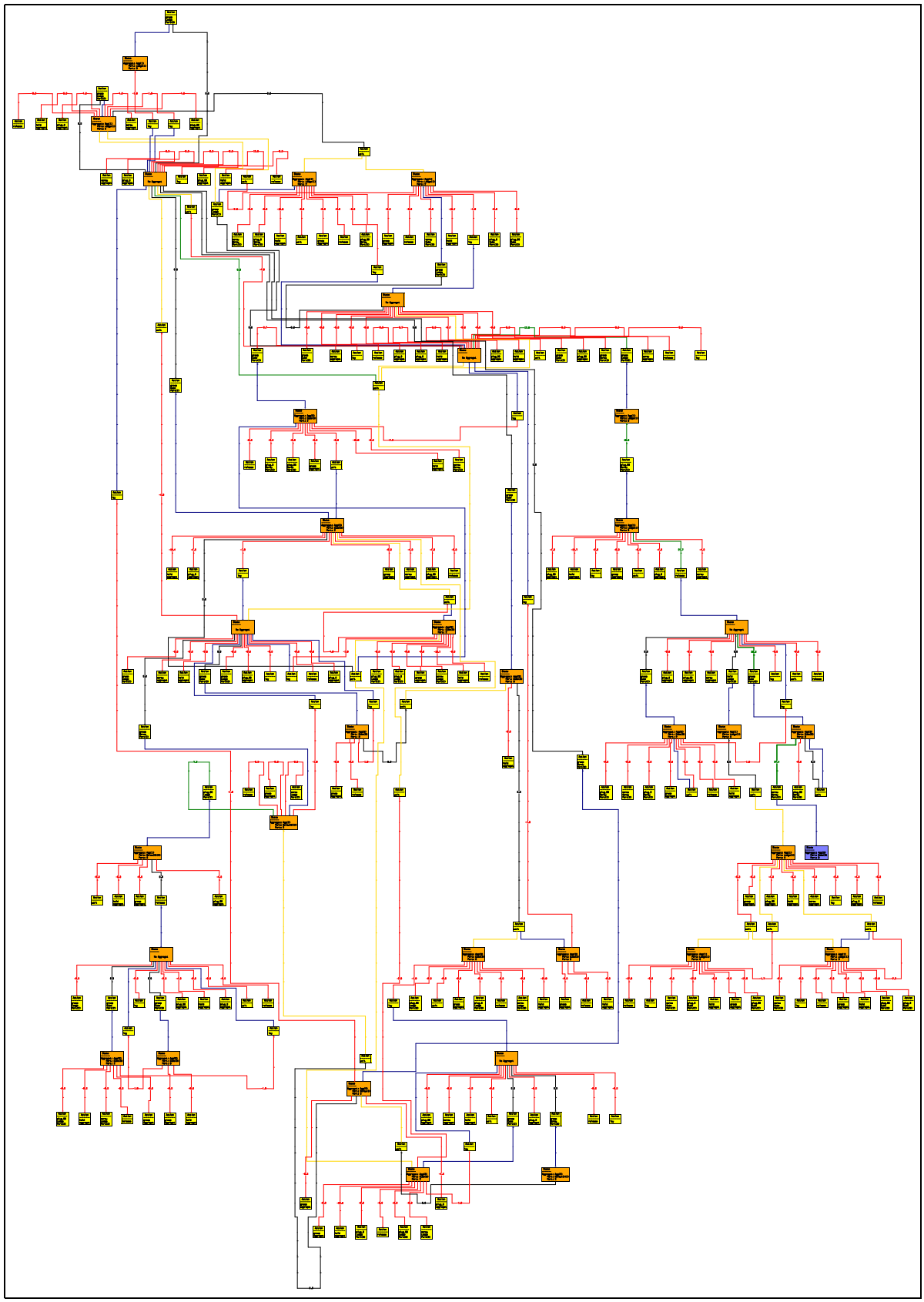


Abbildung 7.18: Der Zustands-/Aktionsgraph des ersten Roboters zur Montage des Aggregates aus Abb. 7.17.

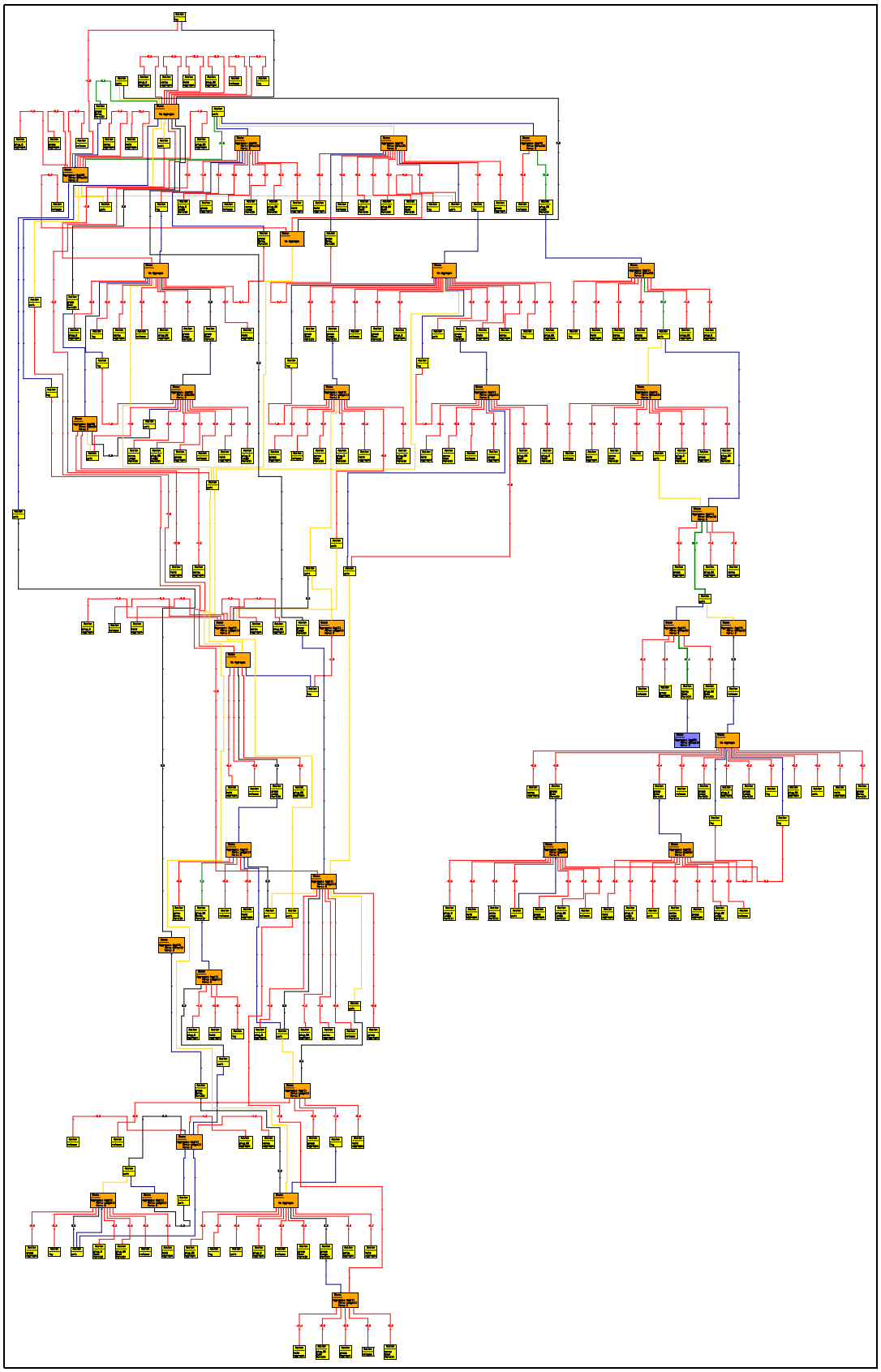


Abbildung 7.19: Der Zustands-/Aktionsgraph des zweiten Roboters zur Montage des Aggregates aus Abb. 7.17.

und *hold* zusammengesetzt werden. Tab. 7.8 gibt die nach dreißig Epochen gefundenen Aktionsfolgen wieder, wobei jede Epoche aus 150 Einzelaktionen besteht. Die Aktionsfolgen

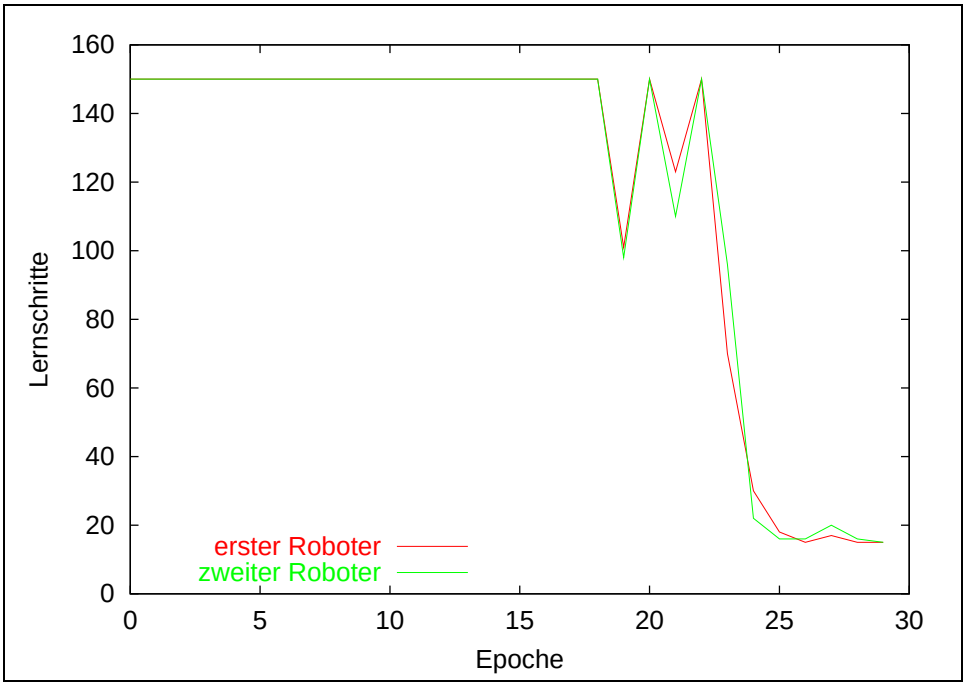


Abbildung 7.20: Für die Montage des in Abb. 5.5 gezeigten Aggregates werden mindestens dreißig Epochen benötigt, bis sich die durchzuführende Aktionsfolge auf Grund der erlernten Q-Werte gefestigt hat. Die maximale Anzahl an Lernschritten pro Epoche beträgt 150. Das Zielaggregat wird das erste Mal nach 19 Epochen vollständig konstruiert.

sind den Graphen Abb. 7.22 und Abb. 7.21 entnommen. Sie bestehen aus insgesamt fünfzehn Einzelaktionen.
Die für das Lernverfahren größten Probleme entstehen beim initialen Greifen der ersten beiden Bauteile und bei der Umgreifaktion. Das Greifen der ersten beiden Bauteile ist deshalb so lernaufwendig, weil zu der Wahl der passenden Bauteile am richtigen Port noch eine zeitliche Komponente hinzukommt. Es kann nur dann zu einer erfolgreichen Montage kommen, wenn beide Roboter gleichzeitig das richtige Bauteil halten und einer oder beide sich für die richtige Montageoperation entscheiden. Da das Zielaggregat aber nicht nur aus zwei unterschiedlichen Bauteilen besteht, sinkt die Wahrscheinlichkeit, daß eine derartige Situation eintritt, gegenüber dem ersten Beispiel (Schraube, Würfel) deutlich ab.

Das Erlernen der notwendigen Schritte für den Umgreifvorgang ist, wie bereits erwähnt, ebenfalls sehr zeitaufwendig. Der Grund dafür liegt an den fehlenden zwischenzeitlichen Belohnungen. Es sind bis zu drei Aktionen hintereinander auszuführen, ohne daß das System einen Reward erhält. Das zeigt sich auch an der deutlich geringeren Bewertung für Aktion 9.

Aktions-nummer	erster Roboter	$Q(s,a)$	zweiter Roboter	$Q(s,a)$
1.	greifen (Schraube, Port (0))	12.1	greifen (Leiste, Port (1))	16.0
2.	stecken - 0^0	13.9	stecken - 0^0	19.2
3.	warten	3.9	loslassen	12.9
4.	warten	7.6	greifen (Leiste, Port (3))	15.0
5.	stecken - 90^0	14.6	stecken - 90^0	16.7
6.	warten	6.1	loslassen	10.2
7.	warten	11.8	greifen (Mutter, Port (1))	12.2
8.	schrauben	13.1	schrauben	13.0
9.	loslassen	5.3	warten	7.0
10.	halten (Leiste, Port (2))	8.1	loslassen	11.8
11.	warten	10.2	greifen (Schraube, Port (0))	13.2
12.	stecken - 0^0	17.4	stecken - 0^0	15.4
13.	loslassen	20.9	warten	6.9
14.	greifen (Mutter, Port (0))	32.1	schrauben (bestraft)	33.4
15.	schrauben	37.3	schrauben	33.4

Tabelle 7.8: Das Aggregat Abb. 5.5 wird mit den in Abb. 7.22 und Abb. 7.21 gezeigten Graphen auf die in dieser Tabelle gezeigte Weise zusammengebaut.

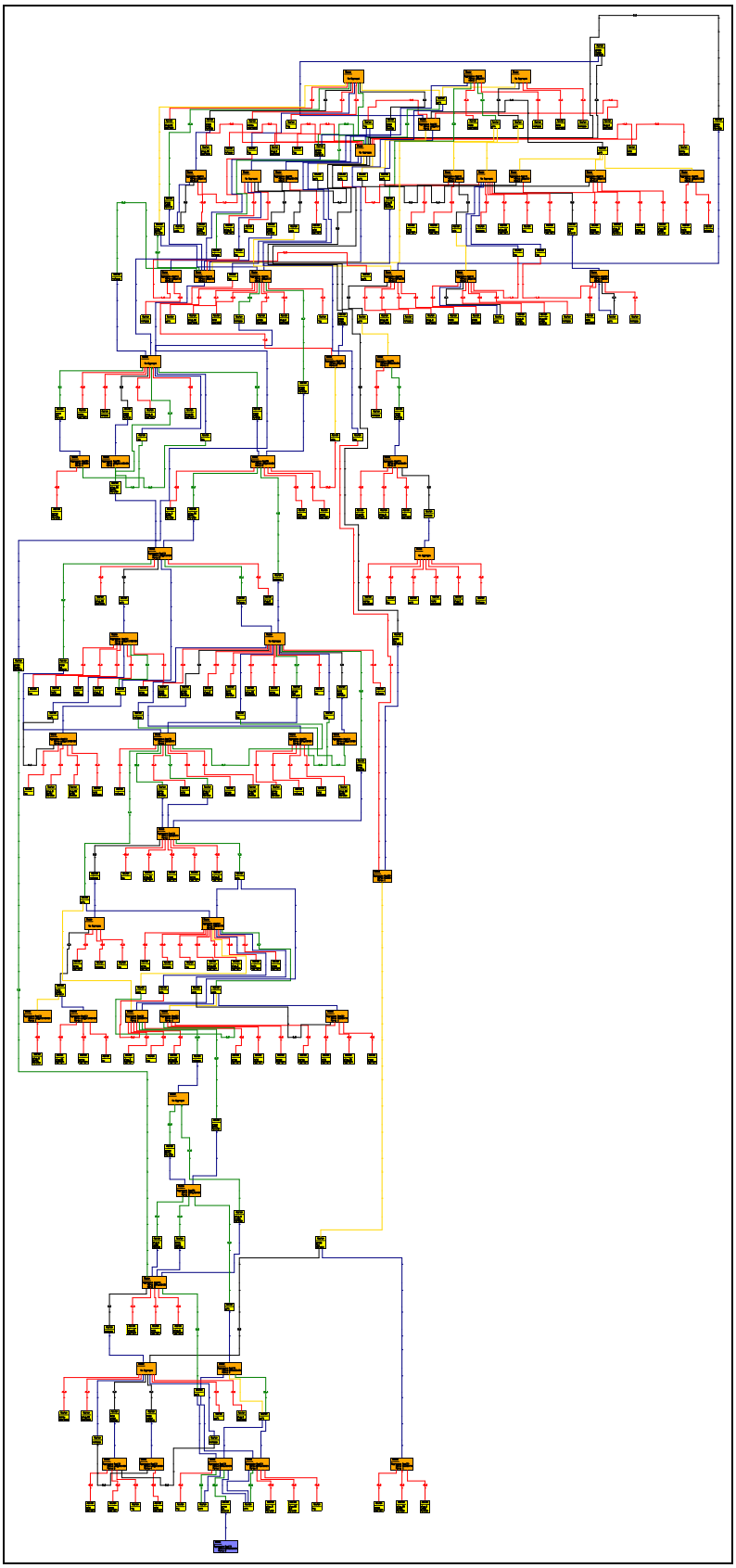


Abbildung 7.21: Zustands-/Aktionsgraph des ersten Roboters für die Montage des in Abb. 5.5 gezeigten Aggregates.

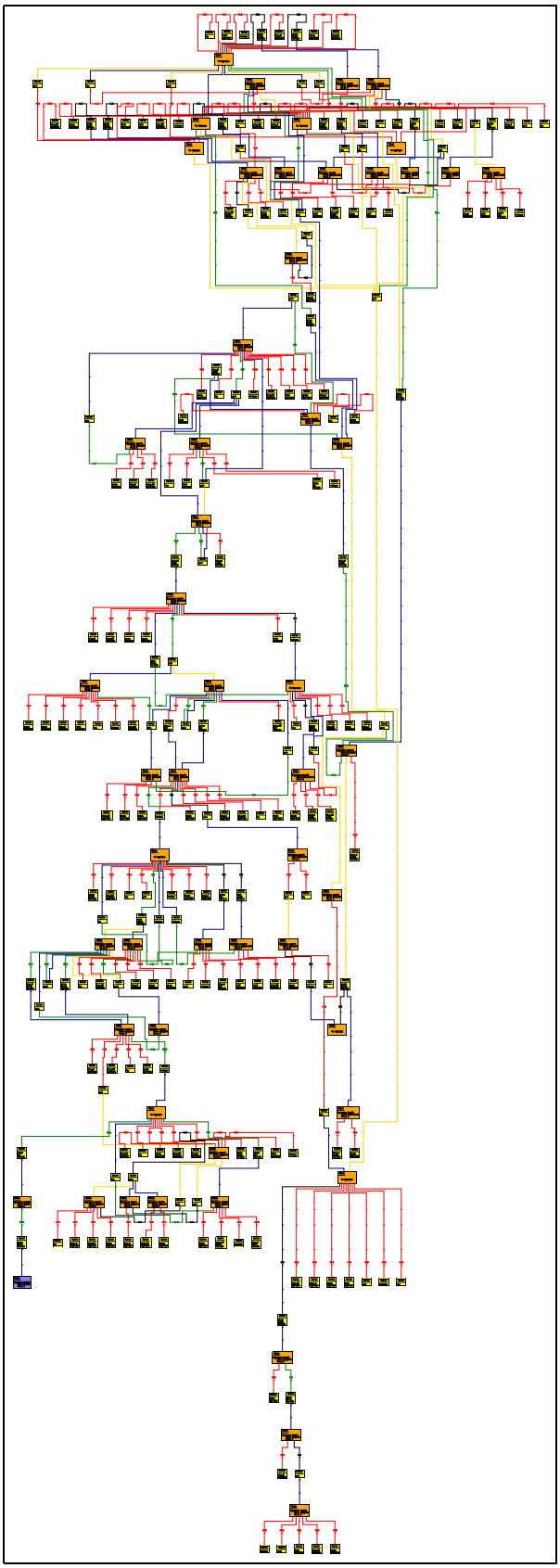
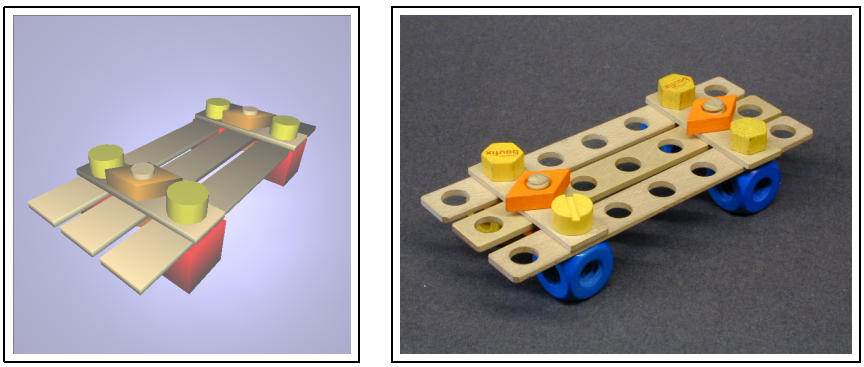


Abbildung 7.22: Zustands-/Aktionsgraph des zweiten Roboters für die Montage des in Abb. 5.5 gezeigten Aggregates.

Montage des Aggregates Abb. 7.23

Dieses aus 17 Bauteilen bestehende Aggregat benötigt insgesamt 57 Montageschritte.



(a) Simuliertes Zielaggregat (b) Reales Zielaggregat

Abbildung 7.23: Zielaggregat, bestehend aus 17 Bauteilen

Aktions-nummer	erster Roboter	zweiter Roboter
1.	greifen (Schraube (orange), Port (0))	greifen (Leiste (7), Port (7))
2.	stecken - 0^0	stecken - 0^0
3.	warten	loslassen
4.	warten	greifen (Leiste (3), Port (3))
5.	stecken - 90^0	stecken - 90^0
6.	warten	loslassen
7.	warten	greifen (Mutter, Port (0))
8.	schrauben	schrauben
9.	warten	loslassen
10.	warten	halten (Leiste (7), Port (11))
11.	loslassen	warten
12.	greifen (Schraube (orange), Port (0))	warten
13.	stecken - 0^0	stecken - 0^0
14.	warten	loslassen
15.	warten	greifen (Leiste (3), Port (3))
16.	stecken - 90^0	stecken - 90^0
17.	warten	loslassen
18.	warten	greifen (Mutter, Port (0))
19.	schrauben	schrauben
20.	loslassen	warten
21.	halten (Leiste (3), Port (0))	warten
22.	warten	loslassen

Fortsetzung auf der nächsten Seite

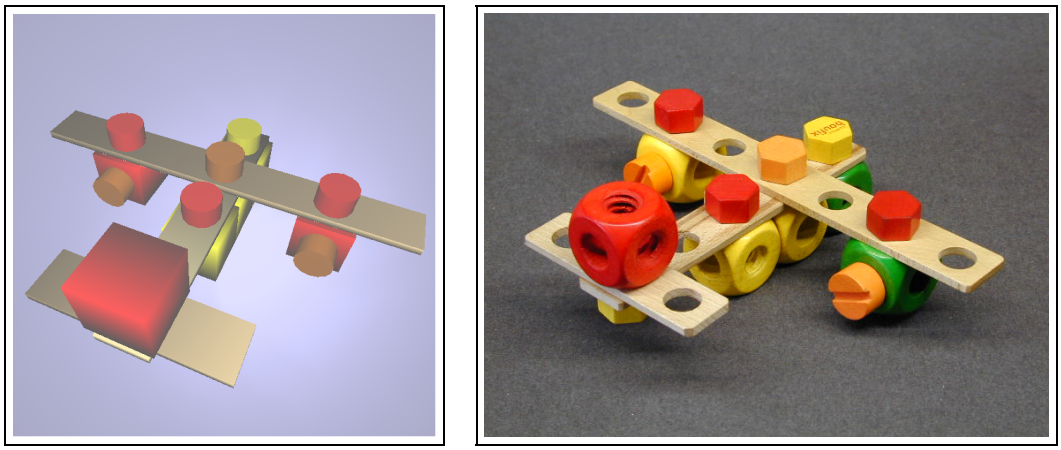
Fortsetzung der vorherigen Seite

Aktions-nummer	erster Roboter	zweiter Roboter
23.	warten	greifen (Schraube (gelb), Port (0))
24.	stecken - 0^0	stecken - 0^0
25.	loslassen	warten
26.	greifen (Leiste (7), Port (3))	warten
27.	stecken - 0^0	stecken - 0^0
28.	loslassen	warten
29.	greifen (Würfel (blau), Port (3))	warten
30.	schrauben	schrauben
31.	loslassen	warten
32.	halten (Leiste (3), Port (4))	warten
33.	warten	loslassen
34.	warten	greifen (Schraube (gelb), Port (0))
35.	stecken - 0^0	stecken - 0^0
36.	loslassen	warten
37.	greifen (Leiste (7), Port (3))	warten
38.	stecken - 0^0	stecken - 0^0
39.	loslassen	warten
40.	greifen (Würfel (blau), Port (3))	warten
41.	schrauben	schrauben
42.	loslassen	warten
43.	halten (Leiste (3), Port (0))	warten
44.	warten	loslassen
45.	warten	greifen (Schraube (gelb), Port (0))
46.	stecken - 0^0	stecken - 0^0
47.	loslassen	warten
48.	greifen (Würfel (blau), Port (3))	warten
49.	schrauben	schrauben
50.	loslassen	warten
51.	halten (Leiste (3), Port (4))	warten
52.	warten	loslassen
53.	warten	greifen (Schraube (gelb), Port (0))
54.	stecken - 0^0	stecken - 0^0
55.	loslassen	warten
56.	greifen (Würfel (blau), Port (3))	warten
57.	schrauben	schrauben

Tabelle 7.9: Das Aggregat Abb. 7.24 läßt sich nach einer mehrtägigen Trainingsphase auf die in dieser Tabelle gezeigte Weise zusammenbauen.

Montage eines Flugzeuges (Abb. 7.24)

Dieses aus 17 Einzelteilen bestehende Aggregat benötigt 7 Umgreifaktionen, die das Lernen besonders langwierig machen.



(a) Simuliertes Zielaggregat (b) Reales Zielaggregat

Abbildung 7.24: Flugzeug als Zielaggregat

Aktions-nummer	erster Roboter	zweiter Roboter
1.	greifen (Schraube (orange), Port (0))	greifen (Leiste (7), Port (7))
2.	stecken - 90^0	stecken - 90^0
3.	warten	loslassen
4.	warten	greifen (Leiste (5), Port (3))
5.	stecken - 0^0	stecken - 0^0
6.	warten	loslassen
7.	warten	greifen (Würfel (gelb), Port (3))
8.	schrauben	schrauben
9.	warten	loslassen
10.	warten	halten (Leiste (5), Port (1))
11.	loslassen	warten
12.	greifen (Schraube (gelb), Port (0))	warten
13.	stecken - 0^0	stecken - 0^0
14.	warten	loslassen
15.	warten	halten (Leiste (5), Port (5))
16.	loslassen	warten
17.	greifen (Schraube (rot), Port (0))	warten
18.	stecken - 0^0	stecken - 0^0
19.	warten	loslassen
20.	warten	greifen (Würfel (gelb), Port (3))
21.	schrauben	schrauben
22.	loslassen	warten
23.	halten (Leiste (5), Port (8))	warten

Fortsetzung auf der nächsten Seite

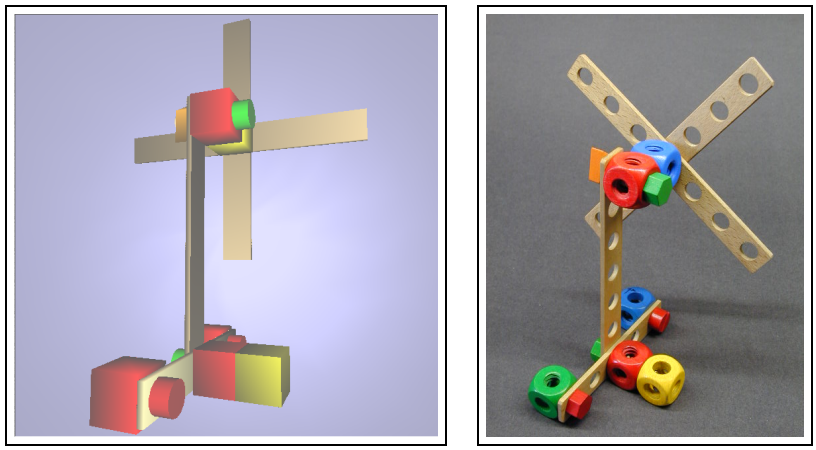
Fortsetzung der vorherigen Seite

Aktions-nummer	erster Roboter	zweiter Roboter
24.	warten	loslassen
25.	warten	greifen (Schraube (gelb), Port (0))
26.	stecken - 0^0	stecken - 0^0
27.	loslassen	warten
28.	greifen (Leiste (3), Port (2))	warten
29.	stecken - 90^0	stecken - 90^0
30.	loslassen	warten
31.	greifen (Würfel (rot), Port (2))	warten
32.	stecken - 0^0	stecken - 0^0
33.	warten	loslassen
34.	warten	halten (Leiste (7), Port (3))
35.	loslassen	warten
36.	greifen (Schraube (rot), Port (0))	warten
37.	stecken - 90^0	stecken - 90^0
38.	warten	loslassen
39.	warten	greifen (Würfel (rot), Port (3))
40.	schrauben	schrauben
41.	warten	loslassen
42.	warten	halten (Würfel (rot), Port (4))
43.	loslassen	warten
44.	greifen (Schraube (orange), Port (0))	warten
45.	schrauben	schrauben
46.	warten	loslassen
47.	warten	halten (Leiste (7), Port (11))
48.	loslassen	warten
49.	greifen (Schraube (rot), Port (0))	warten
50.	stecken - 90^0	stecken - 90^0
51.	warten	loslassen
52.	warten	greifen (Würfel (rot), Port (3))
53.	schrauben	schrauben
54.	warten	loslassen
55.	warten	halten (Würfel (rot), Port (4))
56.	loslassen	warten
57.	greifen (Schraube (orange), Port (0))	warten
58.	schrauben	schrauben

Tabelle 7.10: Das Aggregat Abb. 7.24 läßt sich nach einer mehrtägigen Trainingsphase auf die in dieser Tabelle gezeigte Weise zusammenbauen.

Montage eines Ventilators (Abb. 7.25)

Dieses aus 16 Bauteilen bestehende Aggregat benötigt insgesamt 52 Montageschritte, dabei wird insgesamt viermal umgegriffen.



(a) Simuliertes Zielaggregat (b) Reales Zielaggregat

Abbildung 7.25: Lüfter als Zielaggregat

Aktions-nummer	erster Roboter	zweiter Roboter
1.	greifen (Schraube (grün), Port (0))	greifen (Leiste (7), Port (7))
2.	stecken - 0^0	stecken - 0^0
3.	warten	loslassen
4.	warten	greifen (Leiste (7), Port (7))
5.	stecken - 90^0	stecken - 90^0
6.	warten	loslassen
7.	warten	greifen (Würfel (blau), Port (1))
8.	stecken - 0^0	stecken - 0^0
9.	warten	loslassen
10.	warten	greifen (Würfel (rot), Port (3))
11.	schrauben	schrauben
12.	loslassen	warten
13.	halten (Würfel (rot), Port (2))	warten
14.	warten	loslassen
15.	warten	greifen (Schraube (grün), Port (0))
16.	stecken - 0^0	stecken - 0^0
17.	loslassen	warten
18.	greifen (Leiste (7), Port (1))	warten
19.	stecken - 90^0	stecken - 0^0
20.	loslassen	warten
21.	greifen (Mutter, Port (1))	warten
22.	schrauben	schrauben
23.	warten	loslassen

Fortsetzung auf der nächsten Seite

Fortsetzung der vorherigen Seite

Aktions-nummer	erster Roboter	zweiter Roboter
24.	warten	halten (Leiste (7), Port (12))
25.	loslassen	warten
26.	greifen (Schraube (grün), Port (0))	warten
27.	stecken - 0^0	stecken - 0^0
28.	warten	loslassen
29.	warten	greifen (Leiste (5), Port (5))
30.	stecken - 90^0	stecken - 90^0
31.	warten	loslassen
32.	warten	greifen (Würfel (rot), Port (1))
33.	stecken - 0^0	stecken - 0^0
34.	warten	loslassen
35.	warten	greifen (Würfel (gelb), Port (1))
36.	schrauben	schrauben
37.	loslassen	warten
38.	halten (Leiste (5), Port (0))	warten
39.	warten	loslassen
40.	warten	greifen (Schraube (rot), Port (0))
41.	stecken - 0^0	stecken - 0^0
42.	loslassen	warten
43.	greifen (Würfel (blau), Port (3))	warten
44.	schrauben	schrauben
45.	loslassen	warten
46.	halten (Leiste (5), Port (8))	warten
47.	warten	loslassen
48.	warten	greifen (Schraube (rot), Port (0))
49.	stecken - 0^0	stecken - 0^0
50.	loslassen	warten
51.	greifen (Würfel (grün), Port (3))	warten
52.	schrauben	schrauben

Tabelle 7.11: Das Aggregat Abb. 7.24 läßt sich nach einer mehrtägigen Trainingsphase auf die in dieser Tabelle gezeigte Weise zusammenbauen.

7.6 Lernen durch Instruktion

Unter Lernen durch Instruktion versteht man die Fähigkeit eines Systems, eine einmal instruierte Handlungsabfolge (Aktionsfolge) zu erlernen. Das hier verwendete Verfahren kann mit minimalem Aufwand so umstrukturiert werden, daß es in der Lage ist, eine instruierte Aktionsfolge zu erlernen. Dazu bedarf es lediglich zweier Maßnahmen: Zum einen muß während der Lernphase die automatische Aktionsauswahl mittels Gewichten abgeschaltet werden. Stattdessen kommen die vom Instrukteur vorgegebenen Aktionen zur Anwendung. Zweitens muß nach dem instruierten Ablauf, bei dem der Zustands-/Aktionsgraph aufgebaut wird, ein *Experience replay* durchgeführt werden, bei dem die Handlungsabfolge mehrmals

virtuell durchlaufen wird. Dabei werden die Gewichte der instruierten Handlungsabfolge entsprechend verstärkt. Die in Abb. 7.26 und Abb. 7.27 gezeigten Zustands-/Aktionsgraphen

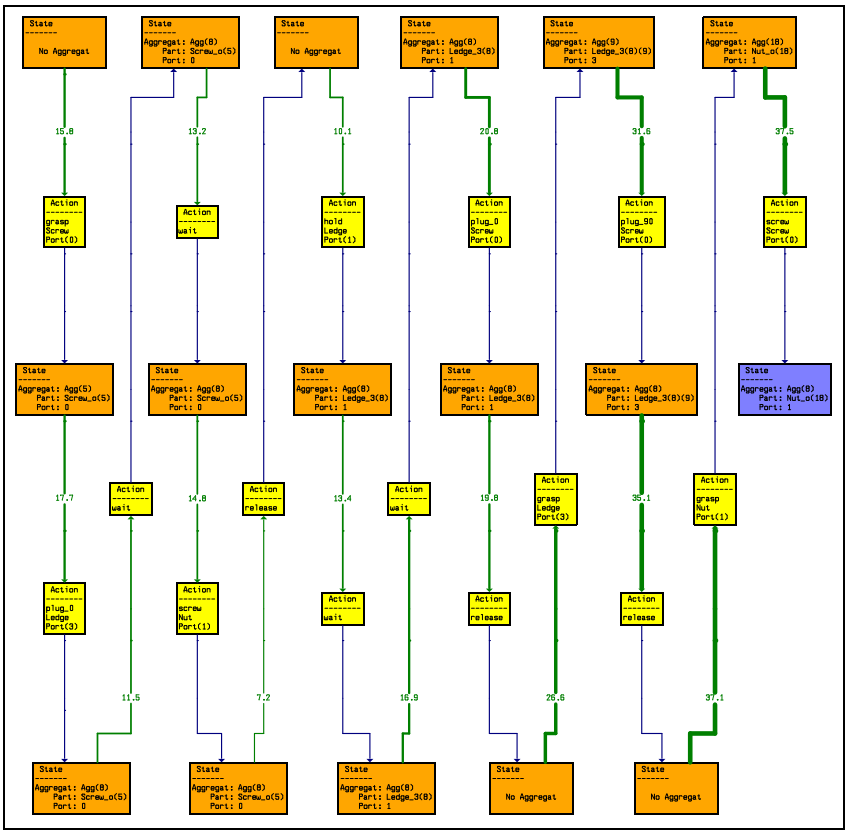


Abbildung 7.26: Zustands-/Aktionsgraph einer instruierten Aktionsfolge für den ersten Roboter. Es handelt sich dabei weniger um einen Graphen als um eine Kette hintereinander auszuführender Aktionen. Die Instruktionskette erzeugt das in Abb. 5.5 gezeigte Aggregat. Die resultierende Aktionsfolge läßt sich direkt aus dem Graphen entnehmen, sie unterscheidet sich deutlich von der in Tab. 7.8

spiegeln weniger einen Graphen als eine Sequenz wieder. Das *Experience replay* wurde solange durchgeführt, bis der Q-Wert, der in den Zielzustand führt, konstant war, Reward und Abklingterm sich also die Waage halten (Gleichung 4.15).

7.7 Lernverfahren zur Bauteilauswahl

Auf der übergeordneten Planungsebene werden, wie bereits in den vorangegangenen Kapiteln beschrieben, Aktionen ausgewählt, analysiert, gegebenenfalls ausgeführt und bewertet. Dazu gehören auch die Greifoperationen. Die Planungsebene muß neben der Aktionsklasse auch den dazugehörenden Bauteiltyp und Port ermitteln. In der realen Welt muß sich der Roboter aber zusätzlich noch für eine konkrete Instanz des zu greifenden Bauteils entscheiden. Es gibt mehrere Strategien, ein konkretes Bauteil auszuwählen:

① sequentiell,

② zufällig,

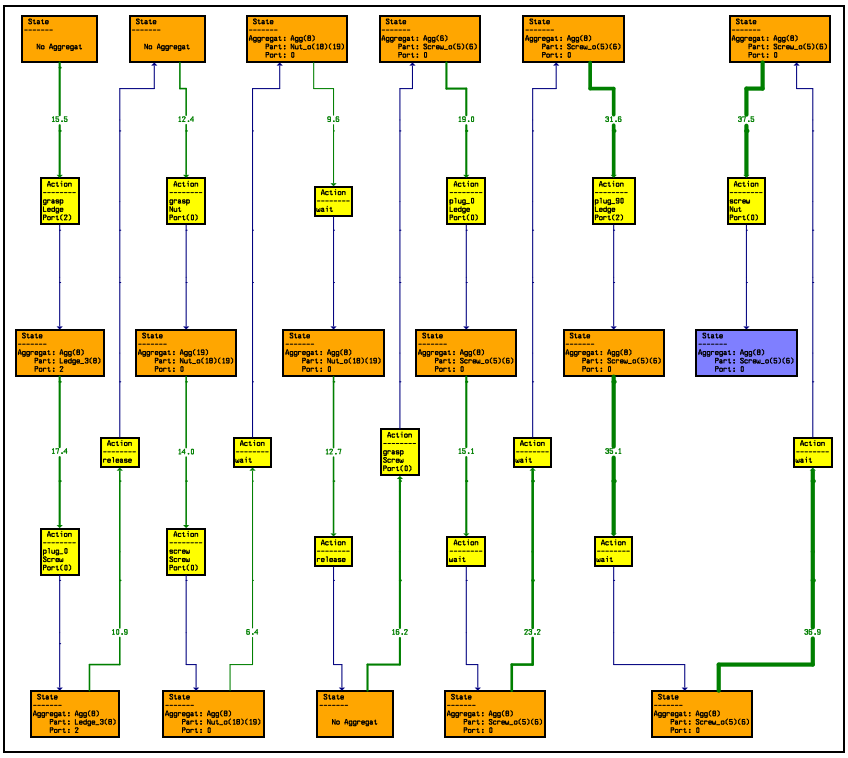


Abbildung 7.27: Zustands-/Aktionsgraph einer instruierten Aktionsfolge für den zweiten Roboter. Es handelt sich dabei weniger um einen Graphen als um eine Kette hintereinander auszuführender Aktionen. Die Instruktionskette erzeugt das in Abb. 5.5 gezeigte Aggregat.

③ nach einer Bewertungsfunktion:

(a) Je geringer der räumliche Abstand zwischen Bauteil und TCP unter Berücksichtigung von Singularitäten, um so höher ist die Bewertung.

(b) Je näher sich das Bauteil im Konfigurationsraum an den momentanen Robotergelenkwinkeln befindet, ebenfalls unter Berücksichtigung von Singularitäten, um so höher ist die Bewertung.

Diese Strategien sind entweder völlig ad hoc oder verwenden die Strecke von der aktuellen Roboterposition zum Bauteil als Bewertung. Der tatsächliche Aufwand, um einen Gegenstand aufzunehmen, fließt jedoch nicht ein. Der geringste Aufwand liegt vor, wenn sich der entsprechende Gegenstand bequem greifen läßt. Dazu sollte er möglichst nahe an der aktuellen TCP-Position liegen, sich weit weg von einer Singularität befinden, keinen Konfigurationswechsel erfordern, die gewünschte Orientierung besitzen und mit hoher Wahrscheinlichkeit über eine zutreffende Orientierungsangabe verfügen. Wie bereits in Abschnitt 7.1.2 erwähnt, sind die Orientierungsangaben zumindest bei Schrauben nicht unbedingt richtig. Gesucht wird eine Kostenfunktion c, die den benötigten Aufwand schätzt. Die Funktion läßt sich mit einem überwachten Lernverfahren finden, da beim Durchführen einer konkreten Greifoperation der tatsächliche Aufwand ermittelt werden kann. Für jedes Bauteil läßt sich eine eigene Kostenfunktion c_t mit $t \in \mathcal{K}$, der Menge aller Bauteile, lernen. Die Parameter von c_t bzw. c sind problemspezifisch. Im Falle des Baufixszenarios und der verwendeten Montageumgebung werden der Ort x, y, die Orientierung o_{ist} und die Ziel-Orientierung o_{soll}

verwendet.

(7.4) $$\text{Kostenfunktion} = c(x, y, z, o_{\text{ist}}, o_{\text{soll}})$$

Zur Bestimmung von c läßt sich ein Funktionsapproximator verwenden, z.B. der in Anhang C beschriebene B-Spline Fuzzy Controller. Die Parameter x, y, z, o_{ist} und o_{soll} sind dabei für eine konkrete Greifoperation bekannt, die Kosten lassen sich durch eine Summe aus Einzelposten bestimmen, z.B.:

(7.5) $$c_t(x, y, z, o_{\text{ist}}, o_{\text{soll}}) = \alpha(c_t^s(x, y, z) + c_t^d(x, y) + c_t^o(o_{\text{ist}}, o_{\text{soll}}) - c(x, y, z, o_{\text{ist}}, o_{\text{soll}}))$$

mit

(7.6) $$c_t^s(x, y, z) = \begin{cases} 0 & \text{wenn } j_5 \geq 8^0 \\ \sigma * 100 & \text{wenn } j_5 < 8^0 \end{cases}$$

(7.7) $$c_t^d(x, y) = \rho * \sqrt{x^2 + y^2}$$

(7.8) $$c_t^o(o_{\text{ist}}, o_{\text{soll}}) = \begin{cases} 0 & \text{wenn } o_{\text{ist}} = o_{\text{soll}} \\ \tau * 10 & \text{wenn } o_{\text{ist}} \neq o_{\text{soll}} \end{cases}$$

wobei j_5 der Gelenkwinkel des fünften Gelenkes an der Greifposition, berechnet durch die inverse Kinematik, ist und $c(x, y, z, o_{\text{ist}}, o_{\text{soll}})$ als Abklingterm fungiert. Die Kostenfunktion kann ebenfalls benutzt werden, um einen günstigen Ablageort zu ermitteln.

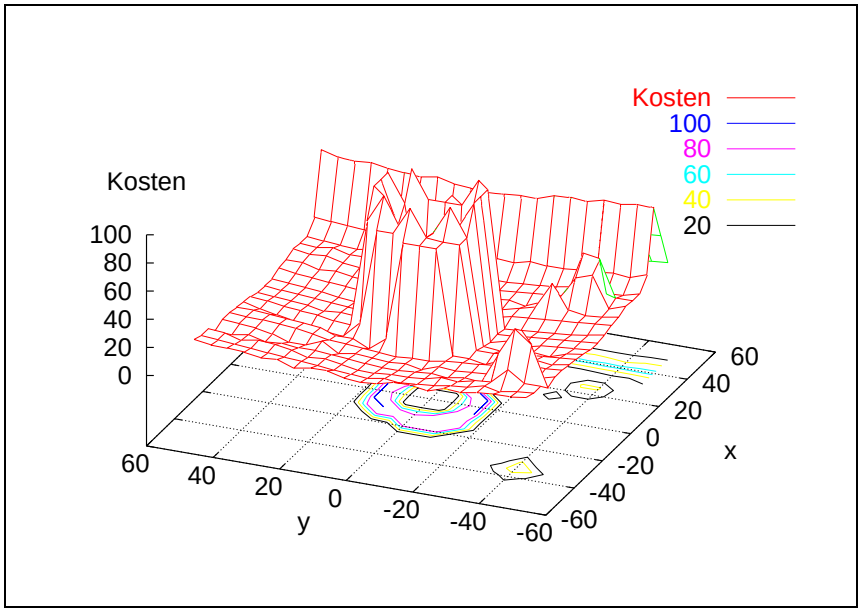


Abbildung 7.28: Kostenfunktion für das Greifen einer stehenden, gelben Schraube in Abhängigkeit vom Ort. Die Greifhöhe z ergibt sich dabei aus der Bauteillänge, codiert durch die Farbe und den Abstand zwischen Roboter und Tisch. Der Ring um den Nullpunkt wird durch den Bereich singulärer Stellungen verursacht (Gleichung 7.6). Mit größerer Entfernung zum Nullpunkt steigen die Kosten leicht an (Gleichung 7.7). In einem Streifen bei etwa $x = 40$ befindet sich eine Welle in der Tischoberfläche, die bewirkt, daß Schrauben beim Abstellen umfallen, was die Kosten entsprechend erhöht (Gleichung 7.8).

Zusammenfassung und Ausblick

In dieser Arbeit wurde eine verteilte Roboterarchitektur und ein darauf aufbauendes Verfahren zum Erlernen einer Montagestrategie entwickelt. Die folgenden Problemstellungen wurden bearbeitet:

- ❏ Prinzipien objektorientierter Kommunikation über ein Netzwerk.
- ❏ Aufbau eines Anwendungsnetzwerkes ohne zentrale Kontrollinstanz.
- ❏ Verwaltung von gemeinsam genutzten Ressourcen in einem verteilten System. Im Falle der Robotik handelt es sich z.B. um den Arbeitsraum, in dem sich die Roboter bewegen.
- ❏ Entwicklung eines Aggregat-Modells und darauf basierender Algorithmen zur Identifikation von Teilaggregaten.
- ❏ Entwicklung eines Konzeptes zur Beschreibung unendlicher Zustands-/Aktionsräume.
- ❏ Realisation von Verstärkungslernverfahren zum kooperativen Greifen von Bauteilen und zur kooperativen Montage von Aggregaten mit einem realen Multirobotersystem.

Verteilte Multiroboterumgebung

Die verteilte Multirobotersteuerung besteht aus einer Reihe von Anwendungen, die auf Grund ihres objektorientierten Designs auf derselben Softwaregrundlage aufbauen. Die Anwendungen detektieren sich innerhalb eines Netzwerkes selbständig und bilden einen gemeinsamen Anwendungsverbund. Informationen zwischen den Anwendungen werden mit Hilfe von Objekten ausgetauscht. Diese können vom Anwender beliebig gestaltet werden. Es werden dynamische Strukturen, Vererbung und Polymorphismus unterstützt. Die Kommunikation wird über Objektkanäle abgewickelt, die sicherstellen, daß ein Empfänger nur solche Anweisungen interpretiert, die ihm auch bekannt sind und der Versender über fehlerhaft zugestellte Objekte informiert wird. Es steht ein definiertes Verfahren zur Generierung von Anfrage-/Antwortpaaren zur Verfügung, das in der Lage ist, mehrere Antwortwarteschlangen zu bedienen.

Als Erweiterung des Kommunikationssystems sind die folgenden Punkte denkbar:

- ❏ Zur Vereinfachung des Kommunikationsprotokolls (Kapitel 2.1) ist ein Konzept zu realisieren, das ohne `DataSize` Funktion auskommt.
  Bei der Entwicklung einer API gilt es, die richtige Mischung aus Benutzerkomfort und Anwendungsperformance zu finden. Die vorgestellte Lösung versucht, dem Anwender

möglichst große Freiheiten in der persönlichen Gestaltung seiner Anwendung zu bieten und das bei hoher Performance, wobei jedoch zum Teil der Benutzungskomfort auf der Strecke bleibt. Ein solcher Aspekt sind die abstrakten Funktionen, die zwangsweise vom Anwender überladen werden müssen. Diese zu minimieren, ohne das restliche Konzept maßgeblich zu ändern, ist nicht trivial.

- ❏ Umgestaltung des Kommunikationsmanagers (Abschnitt 2.2.2), so daß für jede Verbindung ein eigener Thread verwendet wird.
  Der Kommunikationsmanager als zentrale Empfangseinheit einer Anwendung verwaltet eine Vielzahl von Verbindungen. Er ist damit anfällig für Fehler, die beim Lesen von Daten aus einer Verbindung auftreten. Die Konsequenz ist, daß ein Fehler in einem einzelnen Kommunikationskanal die gesamte Kommunikation auch auf anderen Verbindungen, die vom Kommunikationsmanager behandelt werden, zum Erliegen bringen kann.

- ❏ Anpassung des Messageprotokolls an bestehende Abfrage-Konzepte aus dem Bereich der Agentenkommunikation.
  Es gibt eine Reihe von Agentenkommunikationssprachen wie z.B. KQML (FININ ET AL., 1994). Sie definieren einen sinnvollen Satz an Basiskommunikations- und Serviceabfragemechanismen. Es erscheint vernünftig, das verwendete Messageprotokoll zwischen den einzelnen Anwendungen zu systematisieren und sich bei der Abfrage an bestehenden Standards zu orientieren.

- ❏ Verwendung eines separaten Kommunikationskanals für Frage-/Antwortsequenzen.
  In der Praxis hat sich gezeigt, daß das in Abschnitt 2.3 vorgestellte Konzept zum Durchführen von Frage-/Antwortsequenzen bei unbedarfter Verwendung zu Deadlocks neigt. Deshalb ist es vorteilhafter, eine Kategorie von Anfragen zu definieren, die automatisch einen nebenläufigen Prozeß abspalten, der eigenständig eine separate Kommunikation aufbaut.

- ❏ Verfeinerung der Kollisionsvermeidung.
  Je größer und ungenauer die von der Software gesperrten Arbeitsbereiche des Roboters sind, um so größer ist die Wahrscheinlichkeit einer Kollisionsdetektion. Durch die Verwendung von 3D-Polygonen ist es möglich, die exakte Form des Roboters zu beschreiben, was zu einer feineren Kollisionsvermeidung führt. Weiterhin sollten spekulative Bewegungen in die verwendete Trajektorienplanung integriert werden, um exakte Bahnen im kartesischen Raum sperren zu können. Dadurch minimiert man ebenfalls die zu sperrenden Volumina.

Lernen von Montagestrategien

Das Lernverfahren zur kooperativen Montage beruht auf einem Aggregat-Modell, das im Rahmen dieser Arbeit entwickelt wurde. Es spiegelt die Verbindungstopologie der Aggregate wieder und erlaubt die Berechnung von Montagepunkten/Orientierungen. Es beschreibt ein Aggregat als Graphen, der im wesentlichen aus drei unterschiedlichen Knotentypen (Parts, Ports u. Liaisons) besteht. Mit Hilfe von Graphensuche lassen sich Aggregate miteinander

vergleichen und Teilaggregate identifizieren. Darüber hinaus lassen sich die Aggregatbeschreibungen für eine mentale Simulation von Montageaktionen verwenden, um festzustellen, ob gewisse Operationen durchführbar sind und welche Roboterorientierungen dabei eingenommen werden müssen.

Um Montagesequenzen mit Hilfe von Verstärkungslernmethoden interaktiv zu erlernen, werden Aggregatmodelle zum Bestandteil eines Weltzustandes. Dadurch entstehen unendlich große Zustandsräume, die mit den bisherigen Ansätzen zur Speicherung diskreter Zustandsräume nicht behandelt werden können. Zur Repräsentation solcher Zustandsräume wurde ein dynamisches Modell, bestehend aus Zustands- und Aktionsknoten, entworfen und realisiert. Es werden nur solche Zustände und Aktionen modelliert, die auch tatsächlich vom Lerner besucht werden. Ein zusätzlicher Vorteil gegenüber den bisherigen Repräsentationen ist die einfache Berechnung des erwarteten Rewards, nicht nur für den unmittelbar folgenden Schritt, sondern auch für beliebig viele Schritte in die Zukunft. Neben der Behandlung von unendlich großen Zustandsräumen können Lernfragestellungen in nichtdeterministischen Umgebungen behandelt werden.

Die Zustands-/Aktionsgraphen wurden verwendet, um Sequenzen zum kooperativen Greifen und zur Montage von Baufixaggregaten selbständig zu erlernen. Dabei wird ein Q-Lernverfahren eingesetzt, das für jeden der beteiligten Roboter einen eigenen Graphen entwickelt. Die Einzelaktionen bestehen dabei unter Umständen wiederum aus erlernten Operationen, wie z.B. dem Greifen oder überwachten Verfahren zur Objekterkennung und der Kraft-/Positionsregelung, die zunächst am realen Mehrarmsystem realisiert wurden.

Als Erweiterung des Lernsystems sind die folgenden Punkte denkbar:

- Verallgemeinerung der Zustands-/Aktionsgraphen zur besseren Übertragbarkeit auf andere Anwendungsgebiete.

- Verkleinerung der gelernten Graphen durch Verwendung von Pruning-Strategien.
  Eine weitere interessante Fragestellung ist das sogenannte *Graph Pruning*. Hier geht es darum, nach einer gewissen Lernphase den Zustands-/Aktionsgraphen durch das Löschen unnötiger Zweige zu verkleinern. Dabei muß jedoch beachtet werden, daß durch das Löschen eines Zweiges indirekt dessen Bewertung auf den initialen Wert, also z.B. Null zurückgesetzt wird. Deshalb müssen die Bewertungen benachbarter Zweige/Knoten angepaßt werden.

- Simulation kreativen Verhaltens durch *Approximate Matching*.
  Appoximate Matching von Baufixaggregaten ermöglicht es, Variationen in den Aggregaten zu erzeugen. Dazu wird ein Verfahren benötigt, das die Ähnlichkeit zweier Baufixaggregate berechnet. Eine mögliche Vorgehensweise ist die Übertragung der *Edit Distance* (KURTZ, 1996) auf Graphenstrukturen.

- Verbesserung der Generalisierungseigenschaften und Beschleunigung des Lernprozesses durch Erlernen von Baugruppen.
  Je länger die Sequenz vom Start zum Zielzustand ist, um so länger dauert der Lernprozeß, um sie zu finden. Eine Möglichkeit zur Beschleunigung besteht darin, ein Aggregat in sinnvolle Baugruppen zu unterteilen z.B. Leitwerk, Propeller, Rumpf, Fahrwerk. Die Sequenz zum Erlernen einer Baugruppe ist kürzer und damit schneller zu lernen. Darauf aufsetzend wird eine Sequenz benötigt, die die Einzelbaugruppen wiederum zu einem Gesamtaggregat zusammensetzt. Dazu muß man jedoch Baugruppen greifen und handhaben können.

A

Baufix Statistik

A.1 Allgemeine Bemerkungen

Da die Baufixteile in ihren Abmessungen sehr stark streuen, wurde eine statistische Erfassung der Maße durchgeführt. Die Standardabweichung s einer Stichprobe berechnet sich nach:

$$s = \sqrt{v^2}$$

mit der Stichprobenvarianz v^2. Es gilt:

$$v^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)$$

wobei x_i die gemessenen Längen bezeichnet und $\bar{x}$ den Mittelwert; alle Längenangaben in mm. Bei Angabe von s hat die Aussage eine 67%'ige Wahrscheinlichkeit, bei $2s$ sind es 95%.

Rundkopfschrauben Mangels ausreichender Anzahl an Rundkopfschrauben konnte die mittlere Schraubenlänge nicht bestimmt werden. Gemessen wurden der Kopfdurchmesser, die Kopfhöhe und der Gewindeaußendurchmesser, Tab. A.2. Die Stichprobe umfaßte 13 Schrauben unterschiedlicher Längen.

Sechskantschrauben Es wurde die Höhe des Schraubenkopfes, die Länge der Schrauben (inklusive Kopfhöhe), der Kopfdurchmesser und der Gewindedurchmesser vermessen Tab. A.1.
Der Kopfdurchmesser wurde dreimal, jeweils zwischen den gegenüberliegenden Seiten gemessen. Die Kopflänge wurde sechsmal gemessen, jeweils von Ecke zu Ecke des Sechsecks.

Würfel Die Stichprobe umfaßt 16 Würfel; es wurden die Längen aller Kanten und die Durchmesser der Löcher gemessen, Tab. A.3.

Leisten Die Stichprobe umfaßt 11 Lochleisten; es wurden die Länge, Breite, Höhe sowie die Lochdurchmesser gemessen, Tab. A.4.

A.1.1 Tabellen

Tabelle A.1: 12 rote Sechskantschrauben. Fehlerangaben ($2s$)

	Längen		Fehler
Schraubenlänge	28.3	$\pm$	0.5
Gewindeaußendurchmesser	12.98	$\pm$	0.4
Kopfdurchmesser	20.0	$\pm$	0.6
Kopfhöhe	12.1	$\pm$	0.1

Tabelle A.2: 13 Rundkopfschrauben. Fehlerangaben ($2s$)

	Längen		Fehler
Gewindeaußendurchmesser	12.98	$\pm$	0.4
Kopfdurchmesser	21.1	$\pm$	0.53
Kopfhöhe	12.1	$\pm$	0.3

Tabelle A.3: 16 Würfel. Fehlerangaben ($2s$)

	Längen		Fehler
Gewindelose Löcher	14.9	$\pm$	0.26
Gewindelöcher	11.5	$\pm$	0.6
Kantenlänge	30.9	$\pm$	0.5

Tabelle A.4: 11 Dreilochleisten. Fehlerangaben ($2s$)

	Längen		Fehler
Länge	91.7	$\pm$	1
Breite	25.1	$\pm$	0.2
Höhe	4.1	$\pm$	0.2
Lochdurchmesser	14.95	$\pm$	0.14

B

Quellcodeauszüge

B.1 Objekttransfer

B.1.1 Beispiele mit einem char

List. B.1.1.1: Versenden eines char

Begin Of TOttoChar

```
1  int TOttoChar::Write(POttoChannel Channel) const
2  {
3    if (inherited::Write(Channel) < 0)
4      return -1;
5
6    return Channel->Write(c,sizeof(char));
7  }
8
9  // ------------------------------------------
10
11 int TOttoChar::Read (POttoChannel Channel)
12 {
13   int  convert = inherited::Read(Channel);
14   Channel->Read(c,sizeof(char));
15
16   return convert;
17 }
18
19 // ---------------------------------------------
20
21 unsigned long int TOttoChar::DataSize(void)const
22 {
23   return (inherited::DataSize() + sizeof(char));
24 }
```

End Of TOttoChar

Das einfachste statische Objekt, das man versenden kann, ist ein `char`. Es sind die drei Methoden `Write`, `Read` und `DataSize` zu definieren. Zunächst wird immer die Vorgängermethode aufgerufen (`inherited::Write` (Zeile 3), `inherited::Read` (Zeile 13) und `inherited::DataSize` (Zeile 23). Der Rückgabewert von `inherited::Read` wird in der Variablen `convert` zwischengespeichert (Zeile 13), da er zum Rückgabewert unserer Funktion wird. Danach werden die für dieses Objekt spezifischen Aktionen ausgeführt. Beim Versenden wird `Channel->Write`, die Sendemethode des Kanals, verwendet (Zeile 6), beim Empfangen `Channel->Read`, seine Empfangsmethode (Zeile 14). Die Methode `DataSize` addiert zu der Datengröße des Vorgängerobjektes lediglich `sizeof(char)`, die Anzahl tatsächlich versendeter Daten.

B.1.2 Beispiele mit einem char*

List. B.1.2.1: Objekt zum Versenden eines char*

Begin Of TOttoCharArray

```
1  int TOttoCharArray::Write(POttoChannel Channel) const
2  {
3    unsigned long int s = Size;     // don't send int values over a network
4
5    if (TOttoObj::Write(Channel) < 0)
6      return -1;
7
8    Channel->Write(&s,sizeof(s));
9    if (s)
10     Channel->Write(c,s);
11
12   return 1;
13
14 }
15
16 // -------------------------------------------
17
18 int TOttoCharArray::Read (POttoChannel Channel)
19 {
20   int               convert; // Gibt an, ob sich die Byteorder verändert hat.
21   unsigned long int s;
22
23
24   convert = TOttoObj::Read(Channel);
25   Channel->Read(&s,sizeof(s));
26
27   if (convert) Swap(&s,&Size);   // Byteorder tauschen
28   else         Size = s;
29
30   if(c) delete[] c;
31   if (Size)
32     {
33       c = new char[Size];
34       Channel->Read(c,Size);
35     }
36   else
37     c = NULL;
```

```
38
39    return convert;
40  }
41
42  // ------------------------------------------------------
43
44  unsigned long int TOttoCharArray::DataSize(void)const
45  {
46    return (TOttoObj::DataSize() + Size + sizeof(unsigned long int));
47  }
```

End Of TOttoCharArray

Beim Versenden eines Strings (`char*`) ist die Datenmenge dynamisch, das heißt sie steht zur Übersetzungszeit des Objektes nicht fest. Damit dennoch die passende Länge an Daten empfangen werden kann, wird zunächst deren Länge (`s`) versendet. Da man keine Integer versenden sollte[1], wird `s` in der Sendefunktion (`TOttoCharArray::Write`) als `unsigned long int` definiert. Nach Aufruf der Vorgängermethode (Zeile 5) wird zunächst `s` versendet (Zeile 8) und, wenn `s > 0` ist, auch die eigentlichen Daten (Zeile 10).

Beim Empfang ist die Vorgehensweise identisch: Zunächst wird die Empfangsmethode des Vorgängers aufgerufen (Zeile 24) und dann die Datenlänge `s` gelesen (Zeile 25). Bevor die entsprechende Anzahl Bytes gelesen werden kann, muß sichergestellt werden, daß `s` die richtige Byteorder besitzt. Diese Information entnimmt man der Variablen `convert` (Zeile 27). Anschließend kann der benötigte Speicherplatz belegt werden (Zeile 33) und die Daten aus dem Kanal gelesen werden (Zeile 34).

Die Funktion `DataSize` besteht aus der Addition der Vorgängerdatengröße, der Länge des zu versendenden Strings, hier in der Objektvariablen `Size` abgelegt, und der Größe des zusätzlichen `long`s, der die Datengröße angibt.

B.1.3 Beispiel für polymorphe Objekte

Das Versenden und Empfangen polymorpher Objekte bedarf einiger weniger, besonderer Schritte, wie sie in Kapitel 2.1 Seite 32 beschrieben werden. Das Vorgehen anhand der im Rahmen dieser Arbeit entwickelten Software soll hier an einem Beispiel skizziert werden. Es soll ein Aggregat (siehe Kapitel 5 bzw. Abb. B.1) mit allen enthaltenen Bauteilen, ports und Liaisons versendet bzw. empfangen werden. Wie müssen die drei Funktionen `Read`, `Write` und `DateSize` aufgebaut werden? Das List. B.1.3.1 besteht aus zwei Teilen: in Zeile 1 – 11 wird die Klasse `TSendableAggregat` deklariert (hier wird nur der relevante Teil gezeigt), in den Zeilen 12 – 167 findet sich die Implementation. `TSendableAggregat` ist von `TDrawableAggregat` und `TOttoObj`, dem Basisobjekt aller versendbaren Objekte, abgeleitet. Auf die Details von `TDrawableAggregat` soll nicht weiter eingegangen werden. Es sei nur soviel gesagt: `TDrawableAggregat` verfügt über eine Liste von Bauteilen `TDynArray<PBaufixElement> Parts` und Liaisons `TDynArray<PLiaison> Liaisons`, die das Objekt beschreiben. Da ein Aggregat aus mehreren unterschiedlichen Bauteilen bestehen kann, ist `Parts` eine Liste von Zeigern auf Bauteile (`PBaufixElement`), hinter denen sich jedoch unterschiedliche Bauteile (`TLedge, TScrew, TNut, TCube`) verbergen können. Der tatsächliche Typ der Daten ist also erst zur Laufzeit bekannt.

[1] die Größe, die eine `int` Variable im Speicher belegt, ist abhängig von der Rechnerarchitektur.

Sendefunktion (`Write`)

Die Sendefunktion findet sich in den Zeilen 13 – 71. Die Zeilen 18 - 30 verlaufen wie in den bisherigen Beispielen. In Zeile 31 wird kein Basisdatentyp versendet, sondern ein Objekt, das bereits über eine `Write` Funktion verfügt. Um den Code nicht nochmals schreiben zu müssen, kann einfach `Name.Write(Channel)` aufgerufen werden. In den Zeilen 33 – 49 schließt sich das Durchlaufen der Liste mit den polymorphen Bauteilen an. Von besonderem Interesse sind die Zeilen 42 und 43. In ihnen wird die Funkton `Channel->Put` bzw. bei gepufferten Kanälen `Channel->BPut` aufgerufen. Diese Funktionen dienen dazu, ganze Objekte zu versenden; sie erzeugen einen eigenen Objektheader und rufen ihrerseits die `Write`-Methode des entsprechenden Objektes auf.

Empfangsfunktion (`Read`)

Die Empfangsfunktion befindet sich in den Zeilen 73 – 123. Die wesentlichen Unterschiede zu den vorherigen Beispielen finden sich in Zeile 88. Um das in `TSendableAggregat` enthaltene Objekt `Name` vom Typ `TOttoCharArray` zu empfangen, wird dessen `Read`-Methode verwendet, was ein Umrechnen der Byteorder mit beinhaltet. Für die übrigen Elemente der Klasse geschieht das in den Zeilen 90 – 98. Das Empfangen der polymorphen Elemente vollzieht sich in den Zeilen 102 – 109. Dabei wird die Methode `Channel->Get()` verwendet, die ein neues Objekt des entsprechenden Typs konstruiert und dessen `Read`-Methode aufruft. Der Rückgabetyp von `Channel->Get()` ist `TOttoObj*`, muß also noch in den passenden Typ konvertiert werden. Dabei kann sinnvollerweise eine Typüberprüfung (`dynamic_cast`) stattfinden.

Funktion zur Angabe der Objektdatengröße (`DataSize`)

In den Zeilen 124 bis 167 findet sich die Implementation der Funktion `TSendableAggregat:: DataSize`. Im Unterschied zu den bisher gezeigten Beispielen wird in Zeile 137 zur Bestimmung der Datengröße des Objektes `Name` dessen `DataSize`-Methode aufgerufen. Weiterhin müssen in den Zeilen 140 – 151 bzw. 154 – 165 die beiden Listen mit den polymorphen Objekten durchlaufen werden, um deren Größen zu bestimmen. In Zeile 147 respektive 161 wird die Größe eines polymorphen Objektes zur Gesamtgröße addiert, dabei muß die Methode `ObjSize` und nicht `DataSize` verwendet werden. Diese Methode addiert zu der Objektdatengröße noch die Headergröße hinzu. Die Methode `ObjSize` wird in `TOttoObj` deklariert und implementiert und muß vom Benutzer nicht überladen werden, da sie sich der `DataSize`-Methode bedient, um die Datengröße zu ermitteln.

List. B.1.3.1: Versenden/Empfangen eines polymorphen Objektes.

Begin Of TSendableAggregat

```
class TSendableAggregat : public TDrawableAggregat, public TOttoObj
{
  TOttoCharArray         Name;

  void RecalcLiaisons(void);

 public:
  virtual int               Write    (POttoChannel Channel)const;
  virtual int               Read     (POttoChannel Channel);
  virtual unsigned long int DataSize (void)     const;
};

// ---------------------------------------------------------
// - int TSendableAggregat::Write(POttoChannel Channel)const
// ---------------------------------------------------------
int TSendableAggregat::Write(POttoChannel Channel)const
{
  int   rv;
  int   i = 0;
  float buff[12];

  if ((rv = TOttoObj::Write(Channel)) < 0)
    return -1;

  rv += Channel->Write(&CenterPartIndex,sizeof(CenterPartIndex));
  rv += Channel->Write(&PartCount,      sizeof(PartCount));
  rv += Channel->Write(&LiaisonCount,   sizeof(LiaisonCount));

  memcpy(buff, AggregatPosition.GetData(),sizeof(buff));
  rv += Channel->Write(buff,              sizeof(buff));
  rv += Name.Write(Channel);

  while(i < GetPartCount())
    {
      PBaufixElement P = GetPart(i);
      if (P)
        {
          i++;
          if (dynamic_cast<PSendableBaufixElement>(P))
            {
              if(Channel->IsBuffered() && GetUOHM(Channel) == UOHM_NONE)
                rv += Channel->BPut(*(dynamic_cast<PSendableBaufixElement>(P)));
              else
                rv += Channel->Put(*(dynamic_cast<PSendableBaufixElement>(P)));
            }
          else
           abort();
        }
    }

  i = 0;
  while(i < GetLiaisonCount())
    {
      PLiaison L = GetLiaison(i);
```

```
55        if (L)
56          {
57            i++;
58            if (dynamic_cast<PSendableLiaison>(L))
59              {
60                if(Channel->IsBuffered() && GetUOHM(Channel) == UOHM_NONE)
61                  rv += Channel->BPut(*(dynamic_cast<PSendableLiaison>(L)));
62                else
63                  rv += Channel->Put(*(dynamic_cast<PSendableLiaison>(L)));
64              }
65            else
66             abort();
67          }
68      }
69
70    return rv;
71  }
72
73  // ------------------------------------------------------
74  // - int TSendableAggregat::Read (POttoChannel Channel)
75  // ------------------------------------------------------
76  int TSendableAggregat::Read (POttoChannel Channel)
77  {
78    int   convert;
79    float buff[12];
80
81    if ((convert = TOttoObj::Read(Channel)) < 0)
82      return -1;
83
84    Channel->Read(&CenterPartIndex,sizeof(CenterPartIndex));
85    Channel->Read(&PartCount,      sizeof(PartCount));
86    Channel->Read(&LiaisonCount,   sizeof(LiaisonCount));
87    Channel->Read(buff,            sizeof(buff));
88    Name.Read(Channel);
89
90    if (convert)
91      {
92        Swap(CenterPartIndex);
93        Swap(PartCount);
94        Swap(LiaisonCount);
95
96        for(int i = 0; i < 12; i++)
97          Swap(buff[i]);
98      }
99
100   memcpy(AggregatPosition.GetData(),buff,sizeof(buff));
101
102   for(int i= 0;i < GetPartCount();i++)
103     {
104       PSendableBaufixElement P = dynamic_cast<PSendableBaufixElement>(Channel->Get());
105       if (P)
106         Parts[i] = P;
107       else
108        abort();
109     }
110
111   for(int i= 0;i < GetLiaisonCount();i++)
```

```
112     {
113       PSendableLiaison L = dynamic_cast<PSendableLiaison>(Channel->Get());
114       if (L)
115         Liaisons[i] = L;
116       else
117        abort();
118     }
119
120   RecalcLiaisons();
121   return convert;
122 }
123
124 // ------------------------------------------------------------
125 // - unsigned long int TSendableAggregat::DataSize(void)const
126 // ------------------------------------------------------------
127 unsigned long int TSendableAggregat::DataSize(void)const
128 {
129   unsigned long int rv = 0;
130   int               i;
131
132   rv = TOttoObj::DataSize() +
133     sizeof(CenterPartIndex) +
134     sizeof(PartCount)       +
135     sizeof(LiaisonCount)    +
136     12 * sizeof(float)      +
137     Name.DataSize();
138
139   i = 0;
140   while(i < GetPartCount())
141    {
142     PBaufixElement P = GetPart(i);
143     if (P)
144      {
145       i++;
146       if (dynamic_cast<PSendableBaufixElement>(P))
147         rv += dynamic_cast<PSendableBaufixElement>(P)->ObjSize();
148       else
149         abort();
150      }
151    }
152
153   i = 0;
154   while(i < GetLiaisonCount())
155    {
156      PLiaison L = GetLiaison(i);
157      if (L)
158       {
159        i++;
160        if (dynamic_cast<PSendableLiaison>(L))
161          rv += dynamic_cast<PSendableLiaison>(L)->ObjSize();
162        else
163         abort();
164       }
165    }
166   return rv;
167 }
```

B.2 Klassenbeispiele

B.2.1 Das Bauteile-/Baufix-Modell

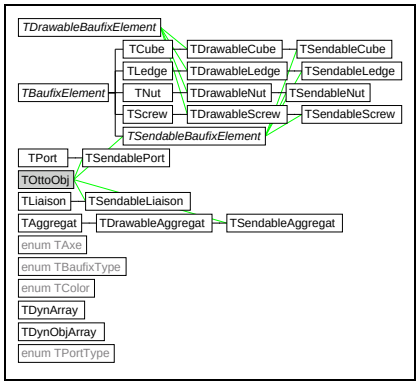


Abbildung B.1: Klassenhierarchie des Baufixmodells mit Bauteilen, Aggregaten und den verwendeten Aufzählungstypen.

List. B.2.1.1: Allgemeine Basisklasse für Verbindungsstellen zwischen Bauteilen.

Begin Of Port Description

```
1  class TPort
2  {
3    TTransform    Offset;       // Lage  des Ports bezüglich
4                                // des  Objekts
5    float         Depth;        // Länge des Ports (in z-Richtung
6                                // des Ports).
7    float         InitialDepth; // Initiale Länge des Ports (in
8                                // z-Richtung des Ports).
9    TPortType     Type;         // Typ des Ports (nehmend,
10                               // gebend)
11   unsigned char Nr;           // Laufende Nummer des Ports am
12                               // Objekt
13   PLiaison      Liaison;      // Wenn verbunden mit anderen
14                               // Ports, Zeiger auf die Liaison
15   PBaufixElement Owner;       // Zeiger auf das Objekt, zu dem
16                               // der Port gehört
17 };
```

End Of Port Description

List. B.2.1.2: Allgemeine Basisklasse für Verbindungen zwischen Bauteilen bzw. deren Ports.

Begin Of Liaison Description

```
class TLiaison
{
  TDynArray<TPort*>  Ports;// Array von Zeigern auf Ports,
                           // die von dieser Liaison verbunden
                           // werden
  unsigned short PortCount;// Anzahl Ports, die durch diese
                           // Liaison verbunden werden
};
```

End Of Liaison Description

List. B.2.1.3: Allgemeine Basisklasse für Bauteile jeglicher Art.

Begin Of Part Description

```
class TBaufixElement
{
  TTransform     Pos;// Position des Objekts in der Welt u. Obj.
                     // Nullpunkt. Ist das Obj. in einem Aggregat,
                     // beschreibt Pos die Lage zum Aggregatko-
                     // ordinatensystem.

  // ------------------------------------------------------------

  TVector        CenterOfMass;// Vektor von Pos zum
                              // Massenschwerpunkt.
  float          Mass;        // Masse des Bauteils.

  // ------------------------------------------------------------

  unsigned short PortCount;// Anzahl Ports eines Objekts
  PPort          PortList; // Array mit "PortCount" Ports

  // ------------------------------------------------------------

  char           Name[ELEMENT_NAME_LENGTH];// Objektname
  TColor         Color;                    // Objektfarbe
  TBaufixType    Type;                     // Objekttyp

  // ------------------------------------------------------------
  unsigned short DelFlag;// Information, die von Aggregaten
                         // genutzt wird
};
```

End Of Part Description

B.3 Codebeispiele

B.3.1 Aktionsauswahl

List. B.3.1.1: Aktionsauswahl

Begin Of Action Selection

```
TActionType SelectAction(PQKnot ActIS)
{
  float        f = random() / RAND_MAX;
  TActionType  i;

  for (i = wait; i < MAXACTIONES; i++)
   {
    f -= ActIS->CalculateProbability(i,K);
    if(f <= 0.0)
     return i;
   }
  return noaction;
}
```

End Of Action Selection

B.3.2 Virtuelle Montageoperation

List. B.3.2.1: Berechnung der neuen Bauteilposition bei einer Fügeoperation

Begin Of Mount Parts

```

// ------------------------------------------------------------------------
// - int  Mate(unsigned short P1,
//              TBaufixElement *BE,
//              unsigned short P2,
// -            TAngle Ang=TAngle(0.0,ANGLETYPE_DEG_CONST))
// ------------------------------------------------------------------------
int TBaufixElement::Mate(unsigned short P1,
                         TBaufixElement *BE,
                         unsigned short P2,
                         PLiaison       &Liaison,
                         TAngle         Ang)
{
  int      ts1,ts2,rv;
  float    OD = 0;

  if (!BE)                              return -1;// Null pointer übergeben
  if ((ts1 =     TestLiaison(P1)) < 0) return -2;// Portnr. existiert nicht
  if ((ts2 = BE->TestLiaison(P2)) < 0) return -3;// Portnr. existiert nicht
  if (ts1 && ts2)                       return -4;// Beide Ports haben bereits
                                                  // eine Liaison

  PPort Port1 =     GetPort(P1);
  PPort Port2 = BE->GetPort(P2);

  if((Port1->GetType() >= give_screw) &&
     (Port2->GetType() >= give_screw))
   return -5;                             // Beide Ports sind gebende Ports

```

```
30    //
31    // Port1 ist gebender Port  (Schraube)
32    //
33    if(Port1->GetType() >= give_screw)
34     {
35      //
36      // Alte Portlänge in "OD" merken, da "MateGT" die Portlänge
37      // verändert, die alte aber noch gebraucht wird.
38      //
39       OD = -Port1->GetDepth();
40
41       if((rv=MateGT(Port1, Port2, Liaison)) != 0)
42        return rv;
43
44       TTransform C(TVector(0.0, 0.0, OD));
45       C.RotateSelf(zvector,Ang);
46       BE->SetPos(Pos *                 //                    -1
47                  Port1->GetOffset() *  //   PG * OG * C  * OT  = PT
48                  C *
49                  Port2->GetOffset().Invert());
50     }
51
52    //
53    // Port2 ist der gebende Port (Schraube)
54    //
55    else if(Port2->GetType() >= give_screw)
56     {
57      //
58      // Alte Portlänge in "OD" merken, da "MateGT" die Portlänge
59      // verändert, die alte aber noch gebraucht wird.
60      //
61       OD = -Port2->GetDepth();
62
63       if((rv = MateGT(Port2,Port1,Liaison)) != 0)
64        return rv;
65
66       TTransform C (TVector(0.0, 0.0, OD));
67       C.RotateSelf(zvector,Ang);
68       BE->SetPos(Pos *                  //             -1    -1
69                  Port1->GetOffset() *   //  PT * OT * C  * OG  = PG
70                  C.Invert() *
71                  Port2->GetOffset().Invert());
72     }
73    //
74    // beide Ports sind nehmende Ports  (sehr selten)
75    //
76    else
77     {
78      Liaison=Port1->Add(Port2);
79      BE->SetPos(Pos * Port1->GetOffset().Rotate(zvector,Ang) *
80                 Port2->GetOffset().Invert());
81     }
82   return 0;
83  }
```

B.3.3 Bestimmung der Teilaggregate

List. B.3.3.1: Suche Aggregat a in Aggregat b.

Begin Of PartOf

```
1
2  // ------------------------------------------------------------------
3  // - int  TAggregat::PartOf(const TAggregat &right)
4  // ------------------------------------------------------------------
5  // - Ist dieses (this) Aggregat Teilaggregat eines größeren (right)?
6  // - Wenn ja, wie oft.
7  // ------------------------------------------------------------------
8
9  int  TAggregat::PartOf(const TAggregat &right)
10 {
11   //
12   // Geringere Anzahl Bauteile
13   //
14   if(right.PartCount < PartCount)    return 0;
15
16   //
17   // Beide Aggregate sind leer
18   //
19   if(right.PartCount == 0 && PartCount == 0) return 1;
20
21   //
22   // Geringere Anzahl Liaisons
23   //
24   if(right.LiaisonCount < LiaisonCount) return 0;
25
26
27   int rv = 0;
28   int entrys = 0;
29   PBaufixElement *TestedElements;
30
31   //
32   // Liste bereits getesteter Bauteile auf den Stack legen.
33   //
34   TestedElements = (PBaufixElement *) alloca(sizeof(PBaufixElement) *
35                    right.GetPartCount());
36   memset(TestedElements,0,sizeof(PBaufixElement) * right.GetPartCount());
37
38   //
39   // Suche "this" einmal in right
40   //
41   while(SinglePartOf(right,TestedElements,entrys))
42    {
43      rv++;
44    }
45
46   return rv;
47 }
48
```

End Of PartOf

List. B.3.3.2: Suche des einmaligen Auftretens eines Bauteils in einem Aggregat.

Begin Of Single Part Of

```
int  TAggregat::SinglePartOf(const TAggregat       &right,
                                   PBaufixElement  *GL,
                                   int             &GE)const
{
  TTransform       DiffA("DIFF-A");
  TTransform       DiffB("DIFF-B");
  PBaufixElement  *LL;
  PBaufixElement  *IL;
  PBaufixElement  *SL;
  int              LE,IE,SE;
  int              j,i = 0;
  int              flag;
  PBaufixElement   First, IP, SP, TP;


  //
  // Diverse Kopien von GL werden benötigt
  //
  LL = (PBaufixElement *) alloca(sizeof(PBaufixElement) * right.GetPartCount());
  IL = (PBaufixElement *) alloca(sizeof(PBaufixElement) * right.GetPartCount());
  SL = (PBaufixElement *) alloca(sizeof(PBaufixElement) * right.GetPartCount());

  //
  // Suche das erste, global noch nicht getestete Bauteil
  //
  do
   {
    First = Parts.GetElement(i++);
   }
  while ( i < GetPartCount() && !First);

  //
  // Initialisiere IL mit GL
  //
  memcpy(IL, GL,sizeof(PBaufixElement) * right.GetPartCount());
  IE = GE;

  do
   {
    //
    // Suche das erste Bauteil aus right, das mit "First" korrespondieren könnte
    //
    IP = right.PartOf(First,IL,IE);

    if(IP)
     {
      if (IE >= right.GetPartCount())
       {
         printf("PANIC how did we get here (1) !!! \n");
         abort();
       }

      //
      // Speichere IP in IL, da es bereits getestet wurde.
```

```
55        //
56        IL[IE++] = IP;
57
58        //
59        // Berechnen der Differenztransformation
60        //
61        DiffA = First->GetPos().Invert() * IP->GetPos();
62
63        //
64        // Initialisiere SL mit GL
65        //
66        memcpy(SL, GL,sizeof(PBaufixElement) * right.GetPartCount());
67        SE = GE;
68
69        //
70        // Durchlaufe alle Bauteile in this, beginnend bei dem zweiten.
71        //
72        j = i;
73
74        do
75         {
76          do
77           {
78            SP = Parts.GetElement(j++);
79           }
80          while ( j < GetPartCount() && !SP);
81
82          //
83          // Initialisiere LL mit SL
84          //
85          memcpy(LL, SL,sizeof(PBaufixElement) * right.GetPartCount());
86          LE = SE;
87
88          if (LE >= right.GetPartCount())
89           {
90             printf("PANIC how did we get here (2) !!! \n");
91             abort();
92           }
93
94          //
95          // speichere IP in LL, da es bereits getestet wurde.
96          //
97          LL[LE++] = IP;
98
99          if (SP)
100          {
101           flag = 0;
102           do
103            {
104
105             //
106             // Suche ein zu SP korrespondierendes Bauteil in right
107             //
108             TP = right.PartOf(SP,LL,LE);
109             if (TP)
110              {
111               if (LE >= right.GetPartCount())
```

```
112                {
113                  printf("PANIC how did we get here (3) !!! \n");
114                  abort();
115                }
116               LL[LE++] = TP;
117               DiffB = TP->GetPos() * SP->GetPos().Invert();
118               flag = 1;
119             }
120           } while ( TP && flag && !(DiffB == DiffA) );
121
122          if (TP && flag && DiffB == DiffA)
123           {
124            if (SE >= right.GetPartCount())
125             {
126              printf("PANIC how did we get here (4) !!! \n");
127              abort();
128             }
129            SL[SE++] = TP;
130           }
131         }
132       } while ( (j < GetPartCount()) && flag && (DiffB == DiffA) );
133
134      if (j == GetPartCount() && flag && (DiffB == DiffA))
135       {
136        memcpy(GL, SL,sizeof(PBaufixElement) * right.GetPartCount());
137        GE = SE;
138
139        if (GE >= right.GetPartCount())
140         {
141          printf("PANIC how did we get here (5) !!! \n");
142          abort();
143         }
144
145        GL[GE++] = IP;
146        return 1;
147       }
148     }
149    else
150     {
151      return 0;
152     }
153   } while (IP);
154
155  return 0;
156 }
```

List. B.3.3.3: Suche nach einem korrespondierenden Bauteil.

Begin Of Search corresponding Part

```
PBaufixElement TAggregat::PartOf(const PBaufixElement Part,
                                       PBaufixElement Tested[],
                                       ULONG entrys) const
{
  ULONG i;
  int c;

  for(i=0, c=0; (i < GetMaxPartCount() &&
                 c < GetPartCount()); i++)
   {

     PBaufixElement BE = Parts.GetElement(i);
     if(BE != NULL)
      {
        c++;
        if (BE->Compare(Part))
         {
          if (!IsInList(BE,Tested,entrys))
           return BE;
         }
      }
   }
  return NULL;
}
```

End Of Search corresponding Part

C

Funktionsapproximation mit B-Splines

In Anhang C wird die im Rahmen dieser Arbeit verwendete C++-Klassenbibliothek zur Funktionsapproximation mit B-Splines im Detail beschrieben. Das erste Kapitel gibt eine kurze Übersicht über die zugrundeliegende Theorie. Details lassen sich in (FERCH, 1997) u. (ZHANG AND KNOLL, 1999) nachlesen. Danach werden die Aufgaben der einzelnen Klassen in der Klassenbibliothek kurz beschrieben und die wichtigsten Funktionen besprochen. Es schließen sich einige Beispielprogramme an.

C.1 B-Splines (Grundlagen)

B-Splines und B-Spline-Kurven sind ein altes und gut verstandenes Teilgebiet innerhalb der Numerik (ABRAMOWSKI AND MÜLLER, 1991), (D.A.BERRY AND B.FRISTEDT, 1985). Ihre günstigen Eigenschaften bei der Approximation von Kurven und insbesondere bei Oberflächen haben sie im Bereich des Computer Aided Design und Manufacturing sehr beliebt gemacht (BEACH, 1991), (LI, 1995).

Bei der B-Spline-Technik werden komplexe Formen aus einfachen Teilen zusammengesetzt, d.h. aus Polynomen niedrigen Grades. Darüber hinaus verläuft sie nahezu analog zur Bézier-Technik, die ein Spezialfall der B-Spline-Technik ist. Näheres zur Bezier-Technik siehe (ABRAMOWSKI AND MÜLLER, 1991), (SACHS, 1969) und (D.A.BERRY AND B.FRISTEDT, 1985).

C.1.1 Definitionen

Sei: $\xi = (x_0, x_1, \ldots, x_k)$ ein k-Tupel

bestehend aus den Komponenten x_i, mit $i = 0 \ldots k$. Sie werden Knoten und ξ Knotenvektor

genannt. Ein B-Spline wird dann folgendermaßen definiert:

(C.1) $$N_i^1(x) = \begin{cases} 1 & \text{für } x \in [x_i, x_{i+1}), \\ 0 & \text{sonst .} \end{cases} \qquad i = 0, \ldots, k-1.$$

(C.2) $$N_i^n(x) = \frac{x - x_i}{x_{i+n-1} - x_i} \cdot N_i^{n-1}(x) + \frac{x_{i+n} - x}{x_{i+n} - x_{i+1}} \cdot N_{i+1}^{n-1}(x).$$

$$0 \le i \le k-n-1, \quad 1 \le n \le k-1, \quad \frac{0}{0} = 0$$

mit n **Ordnung (order)**. Es gilt: Die Ordnung ist gleich $1 + d$. d ist der Grad (degree); er gibt die Größe des Exponenten des verwendeten Polynoms an. Grad 2 ist also ein quadratisches Polynom. Die Ordnung einer B-Spline-Funktion gibt die Breite des Intervalls an, in dem die Funktion größer Null ist. Die B-Spline-Funktion N_i^n hat nach obiger Definition den Grad n-1 und damit die Ordnung n und ist somit im Intervall $[x_i, x_{i+n}) > 0$.

C.1.2 Berechnung

Gleichung C.1 und Gleichung C.2 legen es nahe, eine rekursive Implementation zur Berechnung der B-Splines zu verwenden. Dabei ist Gleichung C.1 der Rekursionsanker und Gleichung C.2 die Rekursionsfortschreibung. Der hochgestellte Index wird bei jedem Rekursionsschritt um eins verringert, wodurch der Algorithmus in jedem Fall terminiert. Die Anzahl der Rechenoperationen k in Abhängigkeit von n ergibt sich zu $24 \cdot n - 11 = k$. Der Algorithmus hat also lineare Komplexität. Im Hinblick auf Echtzeitfähigkeit ist jedoch die hohe Steigung (24) beunruhigend. Hinzu kommt noch, daß es sich bei einigen der Operationen um Divisionen und Funktionsaufrufe handelt. Sie benötigen unverhältnismäßig viel mehr Zeit als Additionen oder Multiplikationen.

Wenn die Ordnung der zu berechnenden B-Splines feststeht, bietet sich zumindest bis zur Ordnung 4 die explizite Berechnung an, wie die folgenden Gleichungen zeigen. Genauere Details siehe (Lı, 1995).

B-Splines erster Ordnung berechnen sich wie der Rekursionsanker in Gleichung C.1:

$$N_i^1(x) = \begin{cases} 1 & \text{für } x_i \le t < x_{i+1}, \\ 0 & \text{sonst .} \end{cases}$$

Bei B-Splines der Ordnung zwei wird die Funktion in drei Abschnitte zerteilt:

$$N_i^2(x) = \begin{cases} \dfrac{x - x_i}{x_{i+1} - x_i} & \text{für } x_i \le x < x_{i+1}, \\ \dfrac{x_{i+2} - x}{x_{i+2} - x_{i+1}} & \text{für } x_{i+1} \le x < x_{i+2}, \\ 0 & \text{sonst .} \end{cases}$$

Die stückweise Definition setzt sich bei B-Splines der Ordnung 3 und 4 fort:

$$N_i^3(x) = \begin{cases} \dfrac{(x - x_i)^2}{(x_{i+2} - x_i)(x_{i+1} - x_i)} & \text{für } x_i \le x < x_{i+1}, \\ \dfrac{(x - x_i)(x_{i+2} - x)}{(x_{i+2} - x_i)(x_{i+2} - x_{i+1})} + \dfrac{(x_{i+3} - x)(x - x_{i+1})}{(x_{i+3} - x_{i+1})(x_{i+2} - x_{i+1})} & \text{für } x_{i+1} \le x < x_{i+2}, \\ \dfrac{(x_{i+3} - x)^2}{(x_{i+3} - x_{i+1})(x_{i+3} - x_{i+2})} & \text{für } x_{i+2} \le x < x_{i+3}, \\ 0 & \text{sonst .} \end{cases}$$

$$
N_i^4(x) = \begin{cases}
\dfrac{(x-x_i)^3}{(x_{i+3}-x_i)(x_{i+2}-x_i)(x_{i+1}-x_i)} & \text{für } x_i \le x < x_{i+1}, \\
\left\{ \begin{array}{l} \dfrac{(x-x_i)}{(x_{i+3}-x_i)} \cdot \\ \left\{ \dfrac{(x-x_i)(x_{i+2}-x)}{(x_{i+2}-x_i)(x_{i+2}-x_{i+1})} + \dfrac{(x_{i+3}-x)(x-x_{i+1})}{(x_{i+3}-x_{i+1})(x_{i+2}-x_{i+1})} \right\} + \\ \dfrac{(x_{i+4}-x)(x-x_{i+1})^2}{(x_{i+4}-x_i)(x_{i+3}-x_{i+1})(x_{i+2}-x_{i+1})} \end{array} \right. & \text{für } x_{i+1} \le x < x_{i+2}, \\
\left\{ \begin{array}{l} \dfrac{(x-x_i)(x_{i+3}-x)^2}{(x_{i+3}-x_i)(x_{i+3}-x_{i+1})(x_{i+3}-x_{i+2})} + \dfrac{(x_{i+4}-x)}{(x_{i+4}-x_{i+1})} \cdot \\ \left\{ \dfrac{(x-x_{i+1})(x_{i+3}-x)}{(x_{i+3}-x_{i+1})(x_{i+3}-x_{i+2})} + \dfrac{(x_{i+4}-x)(x-x_{i+2})}{(x_{i+4}-x_{i+2})(x_{i+3}-x_{i+2})} \right\} \end{array} \right. & \text{für } x_{i+2} \le x < x_{i+3}, \\
\dfrac{(x_{i+4}-x)^3}{(x_{i+4}-x_{i+1})(x_{i+4}-x_{i+2})(x_{i+4}-x_{i+3})} & \text{für } x_{i+3} \le x < x_{i+4}, \\
0 & \text{sonst.}
\end{cases}
$$

C.1.3 Berechnung der Ausgabe

j : Die Nummer des Eingangs

x_j : Der Eingangswert des j-ten Eingangs

k_j : Die Ordnung der B-Splines, die auf dem j-ten Eingang definiert sind

m_j : Die Anzahl der auf dem j-ten Eingang definierten linguistischen Terme

i_j : Der Index des i-ten B-Splines auf dem j-ten Eingang

$N_{i_j,j}^{k_j}$: Der i-te linguistische Term des j-ten Eingangs mit der Ordnung k_j

$Y_{i_1,i_2,..,i_n}$: Die Kontrollpunkte (de Boor-Punkte) der Regel $(i_1, i_2, \ldots, i_n)$

Der Reglerausgang wird als sogenannte B-Spline Fläche oder *general NUBS*[1] *hypersurface*

[1] **N**on **U**niform **B**asis **S**plines.

gebildet. Für einen Eingangsvektor $\vec{x} = (x_1, \dots x_n)^T$ ergibt sich:

(C.3)
$$y(\vec{x}) = \frac{\sum_{i_1=1}^{m_1} \cdots \sum_{i_n=1}^{m_n} \left(Y_{i_1,\dots,i_n} \prod_{j=1}^{n} N_{i_j,j}^{k_j}(x_j) \right)}{\sum_{i_1=1}^{m_1} \cdots \sum_{i_n=1}^{m_n} \left(\prod_{j=1}^{n} N_{i_j,j}^{k_j}(x_j) \right)}$$

da der Nenner gleich eins ist:

$$= \sum_{i_1=1}^{m_1} \cdots \sum_{i_n=1}^{m_n} \left(Y_{i_1,\dots,i_n} \prod_{j=1}^{n} N_{i_j,j}^{k_j}(x_j) \right)$$

Ein Vergleich von Gleichung C.3 mit gängigen Defuzzyfikationsmethoden ergibt, daß es sich bei dieser Formel im Grunde um die *center of area* bzw. *center of gravity*-Defuzzyfikationsmethode handelt.

C.1.4 Der Knotenvektor

Der Knotenvektor legt die Lage und Form[2] der B-Splines fest. Ein fehlerhafter Knotenvektor kann somit zu schlechtem Konvergenzverhalten führen. Sollen die Knoten auf dem Wertebereich gleichmäßig, also äquidistant verteilt werden, empfiehlt sich ein schematisches Vorgehen. Mehrere Ansätze sind möglich. Bei der hier benutzten Variante liegt auf den Grenzen des Wertebereichs das Maximum eines B-Splines. Das hat den Nebeneffekt, daß die *partition of unity* unter Umständen geringfügig über den Wertebereich hinausreicht. Auf diese Weise ist es aber möglich, eindeutig zwischen Randregeln und Kernregeln zu unterscheiden. Kernregeln sind diejenigen Regeln, deren B-Splines mit ihrem Maximum innerhalb oder auf den Grenzen des Wertebereichs liegen, Randregeln liegen mit ihrem Maximum außerhalb. Folgende schrittweise Vorgehensweise ergibt sich:

① Gesucht wird der Knotenvektor $\xi = x_0, \dots, x_k$ mit k Knoten.

② Wertebereich $[MinW, MaxW]$ festlegen.

③ Anzahl der im Wertebereich liegenden B-Splines: R festlegen. Die Regeln, die genau auf den Intervallgrenzen liegen, werden mitgezählt.

④ Ordnung n der verwendeten B-Splines festlegen.

⑤ Berechnung von

(C.4)
$$dk = \frac{MaxW - MinW}{b - 1}$$

⑥ Die Anzahl der Knoten für die sogenannten Kern-B-Splines R, also solche innerhalb des Wertebereichs, berechnen sich dann wie folgt:

(C.5)
$$k' = R + n$$

[2] Auch die Ordnung legt die Form fest, aber nicht allein.

⑦ Um die Eigenschaft der *partition of unity* zu gewährleisten, werden für B-Splines mit Ordnung > 2 weitere Knotenpunkte benötigt. Die Anzahl ist in folgender Weise abhängig von der Ordnung:

$$k = \begin{cases} k' + \frac{n}{2} & \forall\, n \mod 2 = 0 \\ k' + \frac{n+1}{2} & \forall\, n \mod 2 \neq 0 \end{cases} \tag{C.6}$$

⑧ Die Lage des ersten Knotens x_0 berechnet sich folgendermaßen:

$$x_0 = \begin{cases} MinW - dk * (n-1) & \forall\, n \mod 2 = 0 \\ MinW - dk * (n-1) - \frac{dk}{2} & \forall\, n \mod 2 \neq 0 \end{cases} \tag{C.7}$$

⑨ jeder weitere Knoten $x_i = x_{i-1} + dk \;\; \forall i = 1 \ldots k-1$

C.1.5 Die Lernregel

Um eine gewünschte Ausgangsgröße zu erhalten, wird ein Verfahren zur Optimierung der Kontrollpunkte benötigt. Dazu wird eine Kostenfunktion aufgestellt, die es zu minimieren gilt. Die wohl beliebteste Kostenfunktion ist die quadratische Abweichung:

$$E = \frac{1}{2} \left(y_r - y_d\right)^2, \tag{C.8}$$

wobei y_d der Sollwert und y_r der aktuelle Ausgabewert des Reglers ist. Die Kontrollpunkte $Y_{i_1,i_2,\ldots,i_n}$ müssen so verändert werden, daß die Kostenfunktion E ein Minimum annimmt. Die Änderung $\Delta Y_{i_1,i_2,\ldots,i_n}$ der Kontrollpunkte ergibt sich aus der Ableitung der Kostenfunktion nach allen Kontrollpunkten:

$$\Delta Y_{i_1,i_2,\ldots,i_n} = \epsilon \frac{\partial E}{\partial Y_{i_1,i_2,\ldots,i_n}} = \epsilon \left(y_r - y_d\right) \prod_{j=1}^{n} N_{i_j,j}^{k_j}(x_j), \quad \text{mit } 0 < \epsilon \leq 1 \tag{C.9}$$

Die Verwendung dieser Kostenfunktion garantiert das Auffinden eines lokalen Minimums, da die zweite partielle Ableitung nach $Y_{i_1,i_2,\ldots,i_n}$ grundsätzlich größer Null ist (ZHANG AND KNOLL, 1997).

Eine einfache Anwendung ist die Approximation von Funktionen mit z.B. einer oder zwei Variablen. Sei $f(x)$ eine solche Funktion, dann vereinfacht sich Gleichung C.9 wie folgt:

$$\Delta Y_i = \epsilon \frac{\partial E}{\partial Y_i} = \epsilon \left(y_r(x) - f(x)\right) N_i^k(x), \quad \text{mit } 0 < \epsilon \leq 1 \tag{C.10}$$

und für $f(x_1, x_2)$ folgt:

$$\Delta Y_{i_1,i_2} = \epsilon \frac{\partial E}{\partial Y_{i_1,i_2}} = \epsilon \left(y_r(x_1,x_2) - f(x_1,x_2)\right) N_{i_1,1}^{k_1}(x_1) \cdot N_{i_2,2}^{k_2}(x_2), \quad \text{mit } 0 < \epsilon \leq 1 \tag{C.11}$$

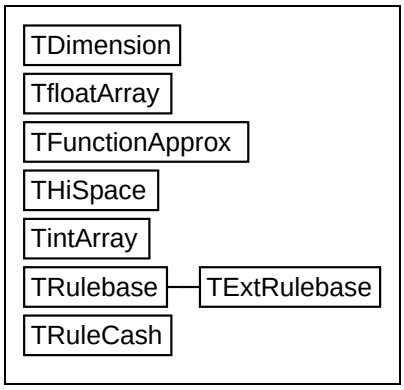


Abbildung C.1: Die C++-Klassen zur Funktionsapproximation mit B-Splines sind sehr flach. Aus Geschwindigkeitsgründen wurde eine möglichst einfache Struktur gewählt. Deshalb ist auch auf Mehrfachvererbung und den Gebrauch von Virtualität verzichtet worden.

C.2 Die Klassenbibliothek

Abb. C.1 zeigt den sehr flachen Vererbungsbaum der hier beschriebenen Klassenbibliothek. Es handelt sich um acht Klassen, von denen im Normalfall aber nur fünf, nämlich `TintArray`, `TfloatArray`, `TDimension`, `THiSpace` und `TRuleBase` zum Einsatz kommen. Die anderen Klassen dienen zur Erweiterung (`TExtRuleBase`), zur Beschleunigung bestimmter Berechnungen wie z.B. der Ableitung (`TRuleCashe`) und zum vereinfachten Verwenden in Standardsituationen (`TFunctionApprox`). Es folgt die Beschreibung der einzelnen Klassen:

`TfloatArray`: Diese Klasse wird als Kapselung für C-Arrays des Typs `float`, also z.B. `float Vector[100]`, einem hundertdimensionalen Vektor verwendet. Da die anderen Klassen intensiven Gebrauch von Arrays unbekannter Länge machen und die Gefahr einer Bereichsüberschreitung sehr groß ist, kennt `TfloatArray` die Dimensionalität und führt Bereichsüberprüfungen durch. Sie läßt sich mit dem `#define PARAMETER_VERIFICATION` an- und abschalten. Da die Klasse auch in multithreaded Anwendungen eingesetzt werden kann, wird für die Speicherung der Daten ein hybrider Ansatz gewählt. Arrays mit bis zu `STATIC_SIZE` Elementen werden auf dem Stack angelegt, größere Datenmengen werden dynamisch alloziert. Dadurch spart man sich beim Kopieren einer Variablen vom Typ `TfloatArray` die dynamische Speicher-Anforderung, die nicht nur zeitaufwendig, sondern intern auch noch durch Mutexe geschützt ist. Auf die Verwendung von Templates wurde verzichtet, da viele Compiler mit diesen Mechanismen immer noch Probleme haben.

- ❏ Quellcodedatei: `tarray.cc`
- ❏ Header: `#include "tarray.hh"`
- ❏ Abhängig von `Convert.hh`
- ❏ Konstruktoren : Es existieren eine Vielzahl von Konstruktoren, deren Parameterlisten selbsterklärend sein sollten und die deshalb hier nicht weiter erwähnt werden.

- **`float &operator[] (const long d);`**
  Die wohl wichtigste Funktion; sie ermöglicht den Zugriff auf ein beliebiges Arrayelement nach C-Standard beginnend bei Index 0.
- `float GetValue (const long d) const;`
  Die const-Version des Operators [] liefert keine Referenz, sondern nur den Wert zurück.
- **`void SetValue (const long d, const float v);`**
  Anstelle einer Zuweisung `a[19] = 10.0;` mit Hilfe des obigen Operators kann auch diese Funktion verwendet werden: `a.SetValue(19, 10.0);`
- **`long GetDim (void) const`**
  Mit dieser Funktion läßt sich die Dimension oder Länge des Arrays ermitteln.
- **`int Store (FILE *Handle);`**
  Speichert die Daten des Arrays in einer bereits geöffneten Datei im Big-Endian Format ab. `Handle` muß also ein Zeiger auf eine gültige Variable vom Typ `FILE` sein.
- **`int Load(FILE *Handle);`**
  Lädt die Daten aus einer bereits geöffneten Datei in das Array `Handle`, muß also ein Zeiger auf eine gültige Variable vom Typ `FILE` sein. Die Arraygröße wird gegebenenfalls angepaßt. Die Daten werden in dem von `Store` verwendeten Format erwartet.

  Durch die prinzipielle Verwendung von Big-Endian können die Daten auch zwischen Rechnern unterschiedlicher Architektur ausgetauscht werden.

`TintArray`: Siehe `TfloatArray`

- Quellcodedatei: `tarray.cc`
- Header: `#include "tarray.hh"`
- Abhängig von `Convert.hh`

`TDimension`: Diese Klasse beschreibt eine Dimension des möglicherweise mehrdimensionalen B-Spline Controllers. Jede Dimension besteht aus einem Knotenvektor und der Ordnung der auf ihm definierten B-Splines sowie deren Anzahl. Diese Klasse wird von der Regelbasis (`TRuleBase`) verwendet, um den Wert der B-Splines zu berechnen. Der Benutzer kommt mit dieser Klasse nur in soweit in Berührung, als daß er sie zum Konstruieren einer Regelbasis benötigt. Ihre Funktionalität ist für ihn von untergeordneter Bedeutung.

- Quellcodedatei: `tdimension.cc`
- Header: `#include "tdimension.hh"`
- Abhängig von `Convert.hh, tarray.hh`
- Konstruktoren :
  neben dem void Konstruktor und dem Copy Konstruktor existieren:
  - **`TDimension(int le, int de, TfloatArray fa);`**
  - **`TDimension(int le, int de, float *fa);`**

Dabei steht `le` für die Anzahl der B-Splines, `de` für die Ordnung der B-Splines und `fa` für den zu verwendenden Knotenvektor, einmal als Objekt vom Typ `TfloatArray`, einmal als C Array von `floats`

- **`int Store (FILE *Handle);`**
  Speichern einer Dimensionsbeschreibung. Siehe dazu auch weiter oben die Beschreibung von `TfloatArray::Store(File *Handle)`
- **`int Load (FILE *Handle);`**
  Laden einer Dimensionsbeschreibung. Siehe dazu auch die Beschreibung von `TfloatArray::Load(File *Handle)`

`THiSpace`: In dieser Klasse wird ein n-dimensionales Feld gekapselt. Im Rahmen der B-Spline Fuzzy Kontrollpunkte nimmt es die Singletons oder Controllvertices auf, ist aber nicht auf diese Funktionalität beschränkt. Um effizient auf die Daten zugreifen zu können, werden sie in einem großen Datenblock abgelegt und die Adressen gemäß folgender Vorschrift berechnet :

```
1  size_t offset = 0;
2
3  for(int i = 0; i < dimensions; i++)
4    offset += Indexes[i] * DimOffsets[i];
```

`Indexes` ist vom Typ `TintArray` und beschreibt, welches Element ausgewählt werden soll. `DimmOffset` ist ebenfalls vom Typ `TintArray` und beinhaltet die vorab berechneten Offsets der einzelnen Dimensionen.

- Quellcodedatei: `thispace.cc`
- Header: `#include "thispace.hh"`
- Abhängig von `tarray.hh`
- Konstruktoren sind:
  - **`THiSpace(const TintArray &ds);`**
  - **`THiSpace(const char *FileName);`**
  - **`THiSpace(const THiSpace &source);`**
- **`float &operator[](TintArray Indexes);`**
- **`int Load (char *FileName);`**
- **`int Store (char *FileName);`**
- **`int Load (FILE *Handle);`**
- **`int Store (FILE *Handle);`**

Da der normale Benutzer der Klassenbibliothek mit diesem Objekt überhaupt nicht in Berührung kommt, werden die Details hier nicht weiter ausgebreitet.

`TRuleBase`: In diesem Objekt laufen alle Fäden zusammen. Es benötigt für jede Dimension eine Dimensionsbeschreibung in Form eines `TDimension` Objektes, legt ein `THiSpace` Objekt an und kann auf diesem die nötigen Funktionen zur Funktionsapproximation bzw. zum Lernen durchführen.

- Quellcodedatei: `trulebase.cc`
- Header: `#include "trulebase.hh"`
- Abhängig von `tarray.hh, tdimension.hh` und `thispace.hh`
- Konstruktoren:
  - **`TRuleBase(void);`** der void - Constructor;
  - **`TRuleBase(char d, PDimension *Dims, char *FileName);`**
    Dieser Konstruktor ist in der Lage, eine Regelbasis unter Verwendung der im Array `Dims` angegebenen Dimensionsbeschreibungen und der Anzahl an Dimensionen d anzulegen. Gibt man einen Dateinamen an, wird versucht, die Singletons aus der entsprechenden Datei zu laden.
- Funktionen zum Berechnen und Trainieren:
  - **`float GetOutPut(const TfloatArray &Input);`**
    Berechnet den approximierten Funktionswert an der Stelle `Input`
  - **`TfloatArray GetGradient(TfloatArray Input, const int d);`**
    Berechnet die d´te Ableitung des approximierten Funktionswerts an der Stelle `Input`. Vorsicht! Der Grad der Ableitung, die sich berechnen läßt, ist vom Grad der B-Splines abhängig. B-Splines des Grades zwei (Ordnung drei) können nur einmal abgeleitet werden, solche des Grades drei zweimal ...
  - **`void Learn(TfloatArray Input, float Error,float epsilon);`**
    Durchführen eines Lernschrittes. `Input` steht für den Eingangsvektor, an dem gelernt werden soll, `Error` gibt den Fehler an, den die bis jetzt approximierte Funktion an der Stelle `Input` hat und `epsilon` ist die zu verwendende Lernschrittrate. Werte außerhalb des von der Rulebase definierten Bereiches werden auf die Ränder der *partition of unity* kopiert.
- Funktionen zum Laden und Speichern:
  - **`int StoreSingletons(char *FileName);`**
    Ruft die entsprechende Store-Funktion des verwendeten `THiSpace` Objektes auf.
  - **`int LoadSingletons(char *FileName);`**
    Ruft die entsprechende Load-Funktion des verwendeten `THiSpace` Objektes auf.
  - **`int Store(const char *FileName);`**
    Speichert die gesamte Regelbasis mit allen dazugehörigen Dimensionsbeschreibungen und den Singletons im Big-Endian Format in der Datei `FileName` ab.
  - **`int Load(const char *FileName);`**
    Lädt die gesamte Regelbasis mit allen dazugehörigen Dimensionsbeschreibungen und den Singletons aus einer von Store erzeugten Datei mit Namen `FileName`.

Auf eine Vielzahl von Routinen zur Ausgabe der gelernten Funktion, der Dimensionsbeschreibungen oder der Singletons im ASCII-Format soll hier im Detail nicht eingegangen werden.

C.3 Beispiele zur Verwendung der Klassenbibliothek

C.3.1 Beispiel I: Approximation einer eindimensionalen Funktion

List. C.3.1.1: Lernen einer eindimensionalen Funktion

Begin Of learn1dfunction.cc

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "trulebase.hh"
#include "tfunctionapprox.hh"
#include "tarray.hh"
#include "tdimension.hh"

// ------------------------------------------------------------------------
float Sin(float x)
{
  return (float) sin((float) x);
}
// ------------------------------------------------------------------------
float Sin_2_pi_x(float x)
{
  return (float) sin((float) x * 2 * M_PI);
}
// ------------------------------------------------------------------------

main(int argc, char *argv[])
{
  PFunctionApprox Approximator;
  PRulebase       RuleBase;

  char RBaseFile[80];
  char PlotFile[80];
  char ErrorPlotFile[80];
  char SplinesPlotFile[80];

  if (argc < 3)
    {
      return (-1);
    }

  //
  // Dateinamen generieren
  //
  strcpy(RBaseFile,argv[2]);
  strcpy(PlotFile,argv[2]);
  strcpy(ErrorPlotFile,argv[2]);
  strcpy(SplinesPlotFile,argv[2]);

  strcat(RBaseFile,".spa");
  strcat(PlotFile,".plt");
  strcat(ErrorPlotFile,"_error.plt");
  strcat(SplinesPlotFile,"_splines.plt");
```

```
50
51    //
52    // Regelbasis erzeugen
53    //
54    PDimension Dimensionen[1];
55    TfloatArray KvektorA (21,
56                          -0.213, -0.142, -0.071, 0.0,   0.071, 0.142, 0.213,
57                           0.284,  0.355,  0.426, 0.497, 0.568, 0.639, 0.71 ,
58                           0.781,  0.852,  0.923,  0.994, 1.065, 1.136,1.207);
59
60    Dimensionen[0] = new TDimension(18,3,KvektorA);
61    RuleBase       = new TRulebase (1,Dimensionen);
62
63    if (RuleBase->Load(argv[1]) < 0)
64      cout << "No RuleBase " << argv[1] <<". Singletons treated to zero\n";
65
66    //
67    // Funktionsapproximator anlegen
68    //
69    Approximator = new TFunctionApprox (RuleBase);
70    Approximator->Approx1D(Sin_2_pi_x,-0.05,1.05,1);
71
72    // Plotten der Kurve
73    RuleBase->Plot (PlotFile, 100.0);
74
75    RuleBase->PlotDimension(SplinesPlotFile, 0.01, 0);
76
77    //  RuleBase abspeichern
78    RuleBase->Store(RBaseFile);
79
80    //  Fehlerkurve plotten
81    cout <<"Maximaler Fehler: "
82         <<Approximator->ErrorCurve1D(Sin_2_pi_x,ErrorPlotFile,0.0,1.0)<<endl;
83
84    delete Approximator;
85    delete RuleBase;
86    delete Dimensionen[0];
87  }
```

End Of learn1dfunction.cc

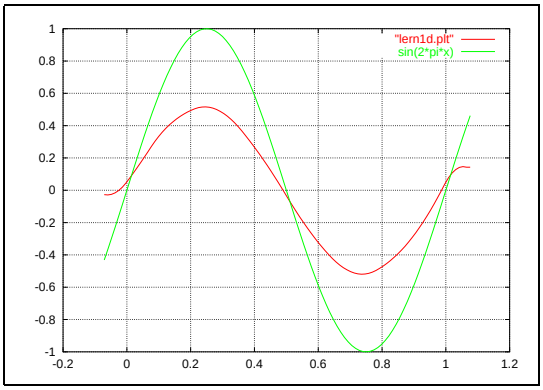
In Zeile 6-9 stehen die benötigten Header-Dateien. Sie beinhalten die in diesem Beispielprogramm benötigten Klassen. In den Zeilen 11 bis 21 werden zwei zu approximierende Beispielfunktionen definiert. In den Zeilen 25 und 26 werden zwei Zeiger definiert. Sie sollen später einmal auf eine Regelbasis `RuleBase` und ein Objekt vom Typ `TFunctionApprox` zeigen. Zunächst werden aber eine Reihe von Dateinamen (Zeilen 28 bis 49) definiert und initialisiert. In Zeile 54 wird ein Array der Länge eins von Zeigern auf Dimensionen angelegt. Da in diesem Beispiel nur eindimensionale Funktionen approximiert werden sollen, brauchen wir nur ein Array mit einem Eintrag. Im nächsten Beispiel mit zwei Dimensionen wird der Sinn dieser Konstruktion deutlicher.

Zeile 55 bis 58 beinhaltet die Konstruktion des Knotenvektors. Er hat 21 Knoten, beginnend bei -0.213 bis 1.207. Er wird in Zeile 60 verwendet, in der die Dimensionsbeschreibung instantiiert wird, siehe Abb. C.2(d). Der erste Parameter gibt die Anzahl der B-Splines an, hier also 18. Der zweite Parameter definiert die Ordnung der B-Splines, also 3, was nach

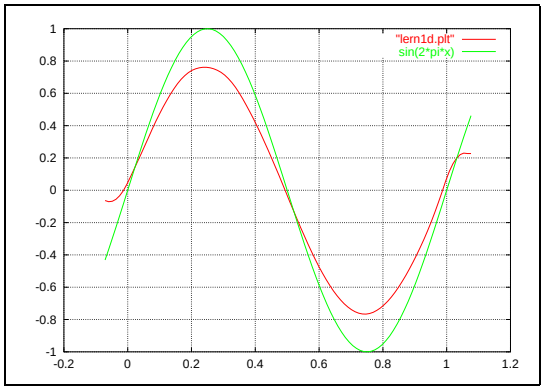
obiger Definition einen Knotenvektor der Länge 21 erforderlich macht.

In Zeile 61 wird der Konstruktor der Regelbasis aufgerufen. Es werden nur zwei Parameter, nämlich die Anzahl der Dimensionen und das Array mit Dimensionsbeschreibungen übergeben. Der dritte optionale Filename entfällt. Stattdessen wird die Load-Methode in Zeile 63 verwendet. Sie lädt nicht nur die Singletons. Zur einfacheren Verwendung wurde in der Datei `tfunctionapprox.cc` ein Objekt implementiert, das das Lernen vordefinierter, also a priori bekannter Funktionen vereinfachen soll. In 69 wird ein solches Objekt instantiiert und in Zeile 71 benutzt. Hier die Methode `Approx1D(Sin_2_pi_x, -0.05, 1.05, 1)`, mit deren Hilfe die Funktion $\sin(2\pi)$ in den Grenzen von -0.5 bis 1.05 gelernt wird. Der letzte Parameter (hier 1) gibt die Anzahl an Lernepochen an, d.h. es wird einmal durch den gesamten Wertebereich gelaufen und ein Lernschritt durchgeführt.

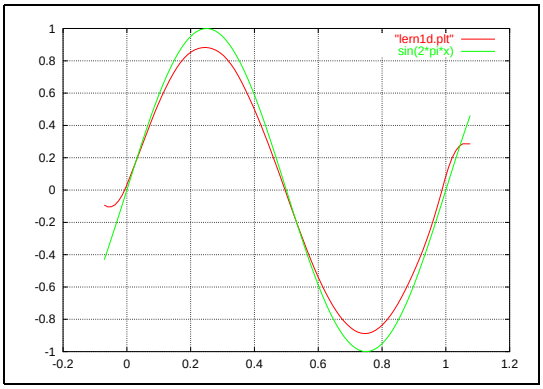
Es schließen sich diverse Funktionen zum Plotten des Ergebnisses an. In Zeile 81 wird dann die Regelbasis selbst gespeichert, und nach Berechnung der Fehlerkurve Zeile 81 u. 82 wird das Programm beendet. Abb. C.2(a) bis Abb. C.2(d) zeigen die approximierte Funktion und die zu approximierende Funktion nach 1 bis 100 Lernepochen.



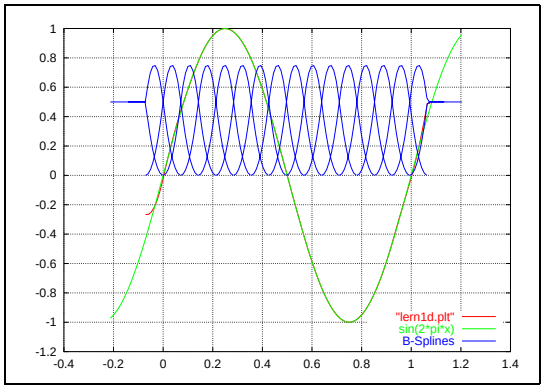
(a) Nach einer Lernepoche



(b) Nach zwei Lernepochen



(c) Nach drei Lernepochen



(d) Nach 100 Lernepochen

Abbildung C.2: *Drei Stadien der Approximation einer eindimensionalen Funktion, hier $sin(2\pi x)$, mit 18 B-Splines bzw. Singletons. Die letzte Grafik enthält zusätzlich die B-Splines.*

C.3.2 Beispiel II: Approximation einer zweidimensionalen Funktion

Als nächstes ein Beispiel, das in einigen Punkten verändert wurde, um eine Funktion zu approximieren, die von zwei Eingangsgrößen abhängt:

List. C.3.2.1: Approximation der Funktion $sin(2\pi x) \cdot \cos(\pi y)$

Begin Of learn2dfunction.cc

```
1  #include <iostream.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <math.h>
6  #include <time.h>
7  #include <sys/times.h>
8
9  #include "textrulebase.H"
10 #include "tfunctionapprox.H"
11 #include "tarray.H"
12 #include "tdimension.H"
13
14 // ------------------------------------------------------------------------
15
16 float sincos(float x, float y)
17 {
18   return (float) sin(2.0*M_PI*x) * cos(M_PI*y) ;
19 }
20 // ------------------------------------------------------------------------
21
22 int main(int argc, char *argv[])
23 {
24   PFunctionApprox Approximator;
25   PExtRulebase       RuleBase;
26
27   char RBaseFile[80];
28   char PlotFile[80];
29   char D_PlotFile[80];
30   char SplinesPlotFile[80];
31
32   if (argc < 4)
33    return (-1);
34
35   //
36   // Dateinamen generieren
37   //
38   strcpy(RBaseFile,argv[2]);
39   strcpy(PlotFile,argv[2]);
40   strcpy(D_PlotFile,"d_");
41   strcpy(SplinesPlotFile,argv[2]);
42
43   strcat(RBaseFile,".rba");
44   strcat(PlotFile,".plt");
45   strcat(D_PlotFile,argv[2]);
46   strcat(D_PlotFile,".plt");
47   strcat(SplinesPlotFile,"_splines.plt");
48
```

```
49    //
50    // Regelbasis erzeugen
51    //
52    PDimension Dimensionen[2];
53    TfloatArray KnotenvektorA (29,
54                               -3.1111, -2.8888, -2.6667, -2.4444, -2.2222,
55                               -1.9999, -1.7778, -1.5555, -1.3333, -1.1111,
56                               -0.8888, -0.6666, -0.4444 ,-0.2222,  0.0,
57                                0.2222,  0.4444,  0.6666,  0.8888,  1.1111,
58                                1.3333,  1.5555,  1.7778,  1.9999,  2.2222,
59                                2.4444,  2.6667,  2.8888,  3.1111);
60
61    Dimensionen[0] = new TDimension(26,3,KnotenvektorA);
62    Dimensionen[1] = new TDimension(26,3,KnotenvektorA);
63
64    RuleBase       = new TExtRulebase (2,Dimensionen);
65
66    if (RuleBase->Load(argv[1]) < 0)
67      cout << "No RuleBase " << argv[1] <<". Singletons treated to zero\n";
68
69    //
70    // Funktionsapproximator anlegen
71    //
72    Approximator = new TFunctionApprox(RuleBase,0.1);
73
74    //
75    // Approximation durchführen
76    //
77
78    //Approximator->Approx2D(sincos,-2.50,2.50,-2.50,2.50,atoi(argv[3]));
79    Approximator->Approx2DRand(sincos,-2.5, 2.5,-2.5,2.5,atoi(argv[3]),NULL);
80
81    //
82    //Plotten der Funktion
83    //
84
85    //RuleBase->Plot(PlotFile);
86    RuleBase->Plot(PlotFile, 0.1, -3.0, 3.0,
87                             0.1, -3.0, 3.0);
88
89    //
90    //Plotten der Ableitungsvektoren
91    //
92    RuleBase->PlotDerivation(D_PlotFile,50, 1);
93
94    //
95    // Splines plotten
96    //
97    Dimensionen[0]->Plot(SplinesPlotFile,0.01);
98
99    //
100   //RuleBase abspeichern
101   //
102   RuleBase->Store(RBaseFile);
103
104   delete Approximator;
105   delete RuleBase;
```

```
106   delete Dimensionen[0];
107   delete Dimensionen[1];
108   return 0;
109 }
```

End Of learn2dfunction.cc

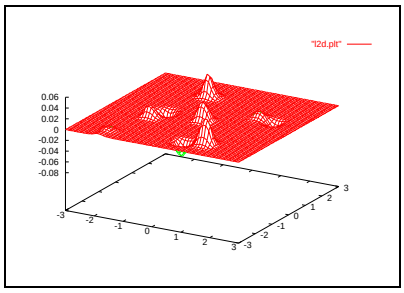
In Zeile 16 bis 19 wird die zu approximierende Funktion $sin(2\pi x) \cdot \cos(\pi y)$ definiert, um in Zeile 78 bzw. 79 zur Approximation verwendet zu werden. Zunächst aber werden in den Zeilen 38 bis 47 die Ausgabedateinamen zusammengesetzt. Ähnlich wie im ersten Beispiel müssen die Dimensionsbeschreibungen erstellt werden. Zeile 53 bis 59 definieren einen Knotenvektor, in Zeile 61, 62 werden die Dimensionsbeschreibungen instantiiert. In Zeile 72 wird wiederum ein Funktionsapproximatorobjekt definiert. Die Approximation kann entweder mit der Methode `TFunctionApproximator::Approx2D` aus Zeile 78 oder mit `TFunctionApproximator::Approx2DRand` aus Zeile 79 geschehen. Es sind neben der zu approximierenden Funktion die Wertebereiche und die Anzahl der Lernschritte anzugeben. Um die Anzahl der Lernschritte variabel zu gestalten, wird `atoi(argv[3])`, der in eine Integer-Zahl gewandelte dritte Programmparameter (siehe Zeile 22) verwendet. Während `TFunctionApproximator::Approx2D` eine systematische Abtastung des Lernraumes durchführt – die Granularität, mit der das geschieht, läßt sich z.B. beim Anlegen des Funktionsapproximatorobjektes Zeile 72 angeben – werden die Lernpunkte bei `TFunctionApproximator::Approx2DRand` zufällig ausgewählt. In diesem Falle bedeuten zehn Lernschritte tatsächlich zehn Punkte, an denen gelernt wird, während bei der ersten Funktion zehn Lernepochen gemeint sind. Dabei durchläuft die Methode den gesamten Wertebereich. Um nicht bei jeder Epoche exakt dieselben Werte zu trainieren, wird der Startwert mittels Zufallszahl innerhalb des ersten Intervalls verschoben. Es schließen sich diverse Ausgabefunktionen an. Zu bemerken ist in Zeile 92 `RuleBase->PlotDerivation(D_PlotFile,50, 1);`. Mit ihrer Hilfe läßt sich die Ableitung der approximierten Funktion berechnen. Der erste Parameter gibt die Ausgabedatei an, der zweite die Anzahl an Samples und der dritte den Grad der Ableitung. Da zur Approximation B-Splines dritter Ordnung Verwendung finden, kann in diesem Fall nur die erste Ableitung berechnet werden. Die erzeugten Ausgaben sind übrigens dazu gedacht, von Gnuplot gelesen zu werden. Der sukzessive Aufruf des Programms mit :

```
lern2dfunktion l2d.rba l2d 10 ,
lern2dfunktion l2d.rba l2d 90 ,
lern2dfunktion l2d.rba l2d 900,
                ...
```

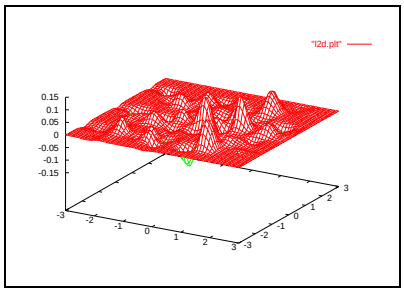
erzeugt jedesmal eine Datei, die Abb. C.2(a) bis Abb. C.2(d) entspricht.

C.3.3 Beispiel III: Approximation eines Punktes im 3-D-Raum.

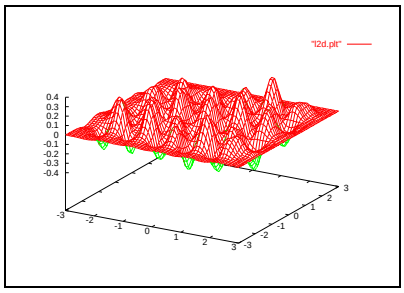
Die Erstellung des Knotenvektors wie in Zeile 55 (Beispiel I) und Zeile 53 (Beispiel II) ist eine recht mühsame Angelegenheit, bei der leicht Fehler auftreten, die oft nur durch eine Visualisierung der tatsächlich generierten B-Splines gefunden werden. Da jedoch in den meisten Fällen äquidistante Knotenvektoren verwendet werden, ist es besser, den Knotenvektor automatisch zu generieren. Das folgende Listing zeigt, wie gemäß Gleichung C.4 bis



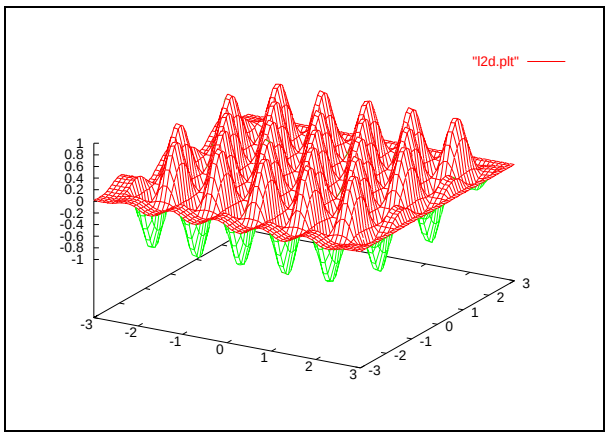
(a) Nach 10 Lernschritten



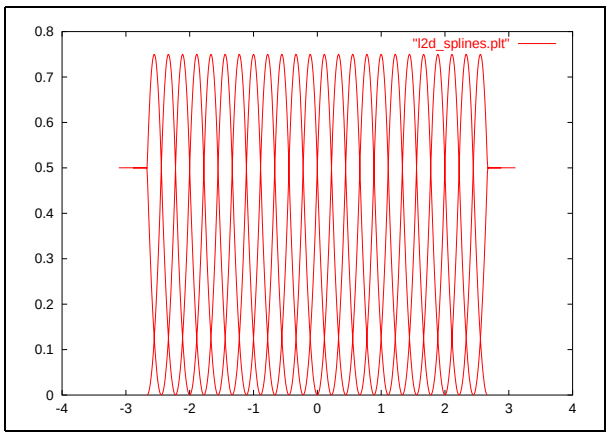
(b) Nach 100 Lernschritten



(c) Nach 1000 Lernschritten



(d) Nach 100000 Lernschritten



(e) B-Splines u. Knotenvektor

Abbildung C.3: Die Abbildungen (a) bis (d) zeigen vier Stadien der Approximation einer zweidimensionalen Funktion ($sin(2\pi x) \cdot \cos(\pi y)$) mit 26 B-Splines auf jeder Achse. Daraus ergeben sich 676 Singletons. Die Punkte, an denen gelernt wurde, sind mit einem Zufallszahlengenerator ausgewählt worden. In der letzten Grafik sieht man die 26 über dem Knotenvektor im Beispielprogramm generierten B-Splines (Zeile 53-59).

Gleichung C.7 ein äquidistanter Knotenvektor berechnet werden kann, hier am Beispiel einer Regelbasis mit drei Dimensionen.

List. C.3.3.1: Generieren eines äquidistanten Knotenvektors

Begin Of 3dcontroller.cc

```
1
2  #define SPLINE_DEG 3
3
4  // ---------------------------------------------------------------------
5  // -  PRuleBase NewRuleBase(float min1, float max1, int Splines1, ...
6  // ---------------------------------------------------------------------
7  PRulebase NewRulebase(float min1, float max1, int Splines1,
8                        float min2, float max2, int Splines2,
9                        float min3, float max3, int Splines3,
10                       char *FileName = NULL)
11 {
12   PRulebase rv;
13
14   PDimension Dim[3];
15   long knoten1, knoten2, knoten3 ;
16   float x01, x02, x03;
17   float dk1 = (max1 - min1) / ((float) Splines1 - 1.0);
18   float dk2 = (max2 - min2) / ((float) Splines2 - 1.0);
19   float dk3 = (max3 - min3) / ((float) Splines3 - 1.0);
20
21   if ((SPLINE_DEG & 1) != 0)      // durch zwei teilbar ?
22     {
23       knoten1 = Splines1 + SPLINE_DEG + ((SPLINE_DEG + 1) / 2);
24       knoten2 = Splines2 + SPLINE_DEG + ((SPLINE_DEG + 1) / 2);
25       knoten3 = Splines3 + SPLINE_DEG + ((SPLINE_DEG + 1) / 2);
26
27       x01 = min1 - dk1 * (SPLINE_DEG  - 1.0) - (dk1 / 2.0);
28       x02 = min2 - dk2 * (SPLINE_DEG  - 1.0) - (dk2 / 2.0);
29       x03 = min3 - dk3 * (SPLINE_DEG  - 1.0) - (dk3 / 2.0);
30     }
31   else
32     {
33       knoten1 = Splines1 + SPLINE_DEG + (SPLINE_DEG / 2);
34       knoten2 = Splines2 + SPLINE_DEG + (SPLINE_DEG / 2);
35       knoten3 = Splines3 + SPLINE_DEG + (SPLINE_DEG / 2);
36
37       x01 = min1 - dk1 * (SPLINE_DEG - 1.0);
38       x02 = min2 - dk2 * (SPLINE_DEG - 1.0);
39       x03 = min3 - dk3 * (SPLINE_DEG - 1.0);
40     }
41
42   TfloatArray K1(knoten1);
43   TfloatArray K2(knoten2);
44   TfloatArray K3(knoten3);
45
46
47   K1[0] = x01;
48   for (int i = 1; i < knoten1; i++)
49     K1[i] = K1[i-1] + dk1;
50
51   K2[0] = x02;
52   for (int i = 1; i < knoten2; i++)
53     K2[i] = K2[i-1] + dk2;
54
```

```
55    K3[0] = x03;
56    for (int i = 1; i < knoten3; i++)
57      K3[i] = K3[i-1] + dk3;
58
59
60    Dim[0] = new TDimension(knoten1-SPLINE_DEG,SPLINE_DEG,K1);
61    Dim[1] = new TDimension(knoten2-SPLINE_DEG,SPLINE_DEG,K2);
62    Dim[2] = new TDimension(knoten3-SPLINE_DEG,SPLINE_DEG,K3);
63
64    rv = new TRulebase(3,Dim);
65
66    delete Dim[0];
67    delete Dim[1];
68    delete Dim[2];
69
70
71    if(FileName)
72      rv->Load(FileName);
73
74    return rv;
75  }
```

End Of 3dcontroller.cc

Eingabeparameter sind die jeweiligen Wertebereiche `min1` bis `max3` sowie die Anzahl der B-Splines. Die Funktion geht nach der im einführenden Kapitel beschriebenen Methode zum Definieren eines Knotenvektors vor. Eine nähere Erläuterung soll hier nicht gegeben werden, vielmehr soll mittels dieser Funktion ein 3-D-Controller instantiiert und trainiert werden ohne zu Hilfenahme des `TFunctionApprox` Objektes aus den vorherigen Beispielen. Dazu betrachte man folgendes Programm:

List. C.3.3.2: Lernen eines bestimmten Wertes in einem 3-D-Raum

Begin Of learn2dfunction.cc

```
1   #include <iostream.h>
2   #include "3dcontroller.hh"
3   // ------------------------------------------------------------------------
4   // ------------------------------------------------------------------------
5   void Learn(PRulebase RB, float in1, float in2, float in3, float soll)
6   {
7     TfloatArray In(3,in1,in2,in3);
8     float       out = RB->GetOutPut(&In);
9
10    RB->Learn(In, soll - out, LSR);
11  }
12
13  // ------------------------------------------------------------------------
14  // ------------------------------------------------------------------------
15  int main(void)
16  {
17    PRulebase Rulebase = NewRulebase(-127.0, 128.0 , 10,
18                                     -127.0, 128.0 , 10,
19                                        0.0, 255.0 , 10,
20                                     "MyBase.rba");
```

```
21
22    cout<< " 10, 10, 10 = "<< GetOutput(Rulebase,10.0,10.0,10.0)   << endl;
23    cout<< "110,110,110 = "<< GetOutput(Rulebase,110.0,110.0,110.0)<< endl;
24
25    for(int i = 0; i < 1000; i++)
26     {
27       Learn(Rulebase, 10.0, 10.0, 10.0,  1.0);
28       Learn(Rulebase, 110.0, 110.0, 110.0, -1.0);
29       cout<< " 10, 10, 10 = "<< GetOutput(Rulebase,10.0,10.0,10.0)   << endl;
30       cout<< "110,110,110 = "<< GetOutput(Rulebase,110.0,110.0,110.0)<< endl;
31     }
32
33    Rulebase->Store(FileName);
34    delete Rulebase;
35  }
36
```

End Of learn2dfunction.cc

Das Anlegen der Regelbasis (17-20) geschieht mit der oben beschriebenen Funktion `NewRulebase`. Die erste und zweite Dimension läuft von -127 bis 128 und wird von 10 Splines abgedeckt, die dritte Dimension hat einen Wertebereich von 0 bis 255, ebenfalls mit zehn B-Splines. Existiert noch keine Regelbasisdatei `"MyBase.rba"`, sind alle Singletons mit Null initialisiert. In Zeile 22 und 23 wird die Controller ausgabe an den exemplarischen Stellen $(10, 10, 10)$ u. $(110, 110, 110)$ einmal berechnet und ausgegeben. In Zeile 25 bis 31 werden an genau diesen Stellen 1000 Lernschritte durchgeführt. Der Punkt $(10, 10, 10)$ soll die Ausgabe 1 produzieren, der Punkt $(110, 110, 110)$ den Wert -1. Um das zu erreichen, wird die in den Zeilen 6 bis 12 gelistete Funktion verwendet. In Zeile 8 werden die drei Eingangswerte in einem Objekt vom Typ `TfloatArray` verpackt. Um den Fehler berechnen zu können, wird in Zeile 9 zunächst die momentane Controller ausgabe berechnet. In Zeile 11 kann dann der eigentliche Lernschritt durchgeführt werden.

Literaturverzeichnis

ABELL, T., AMBLARD, G., BALDWIN, D., FAZIO, T. D., LUI, M.-C. M., AND WHITNEY, D. (1991). *Computer-Aided Mechanical Assembly Planning*, chapter Computer aids for finding, representing, choosing amongst, and evaluating the assembly sequences of mechanical products, pages 383–435. Kluwer Academic Press Boston.

ABRAMOWSKI, S. AND MÜLLER, H. (1991). *Geometrisches Modellieren*. Wissenschaftsverlag Mannheim.

ADEPT TECHNOLOGY (1993). *V^+ User's Guide v.11*. Adept Technology, 11 Auflage.

ANSALDI, S., FLORIANI, L. D., AND FALCIDIENO, B. (1985). Geometric modelling of solid objects by using a face adjanceny graph representation. In *Computer Graphics (Proceedings of Siggraph 1985)*, volume 19, pages 131–139.

BANDER, J. L. AND WHITE, C. C. (1998). A Heuristic Search Algorithm for Path Determination with Learning. *IEEE Transactions on Systems, Man, and Cybernetics - Part A*, 28(1):1998.

BARTO, A., BRADKE, S., AND SINGH, S. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, Special volume : Computational research on interaction and agency 72(1):81–138.

BAUCKHAGE, C., KUMMERT, F., AND SAGERER, G. (1999). Learning Assembly Sequence Plans Using Functional Models. In *International Symposium on Assembly and Task Planning*, ISART, pages 1–7. IEEE.

BAUMGART, B. G. (1972). Winged Edge Polyhedron Representation. Technical Report STAN-CS-320, Stanford University, Palo Alto, CA.

BEACH, R. C. (1991). *An introduction to the Curves and Surfaces of Computer - Aided Design*. Van Nostrand Reinhold New York.

BELLMAN, R. (1957). *Dynamic Programming*. Princeton, NY.

BERGER, C. (2000). Ein robuster, farbbasierter Objekterkenner zur Roboterfeinpositionierung in einem realen Laborumfeld. Master's thesis, Universität Bielefeld, Technische Fakultät, AG Technische Informatik.

BERTSEKAS, D. P. (1987). *Dynamic Programming (Deterministic and Stochastik Models)*. Prentice-Hall, Englewood Cliffs, NJ.

BOISSONNAT, J.-D. (1984). Geometric structures for three-dimensional shape representation. *ACM Transactions on Graphics*, 3(4):266–286.

BUKCHIN, J., DAREL, E., AND RUBINOVITZ, J. (1997). Team-oriented assembly system design: A new approach. *International Journal of Production Economics*, 51(1-2):47–57.

CAO, T. AND SANDERSON, A. C. (1998). AND/OR Net Representation for Robotic Task Sequence Planning. *IEEE Transactions on Systems, Man, and Cybernetics-Part C:*, 28(2):204–217.

CARRACIOLO, R., CERESOLE, E., MARTINO, T. D., AND GIANNINI (1995). *Graphics and Robotics*, chapter From CAD models to Assembly Planning, pages 115–129. Springer-Verlag, Berlin.

CHAKRABARTY, S. AND WOLTER, J. (1995). A Structure-Oriented Approach to Assembly Sequence Planning. Technical Report TR95-006.

CHEN, K. AND HENRIOUD, J.-M. (1994). Systematic Generation of Assembly Precedence Graphs. In *IEEE International Conference on robotics and Automation*, volume 2, pages 1476–1482. IEEE.

CHICHOSZ, P. AND MULAWKA, J. J. (1995). Fast and efficient reinforcement learning with truncated temporal differences. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 99–107. Morgen Kaufmann.

CORKE, P. AND KIRKHAM, R. (1995). The ARCL Robot Programming System. `http://www.cat.csiro.au/cmst/staff/pic/robot.html`.

D.A.BERRY AND B.FRISTEDT (1985). *Bandid Problems (Sequential Allocation of Experiments*. Chapman and Hall.

DAYAN, P. (1992). The Convergence of $TD(\lambda)$ for general λ . *Machine Learning*, 8(3):341–362.

DE MELLO, L. H. (1989). *Task Sequence Planning for Robotic Assembly*. PhD thesis, Carnegie Mellon.

DE MELLO, L. S. H. AND SANDERSON, A. C. (1986). AND/OR graph representation of assembly plans. Technical Report CMU-RI-TR-86-8, Robotics Institute - Carnegie-Mellon Univ.

D.S.HONG AND H.S.CHO (1995). A Neuronal-network-based Computational Schema for Generating Optimised Robotic Assembly Sequences. *Eng. Applic. Artif. Intell.*, 8(2):129–145.

ENGELN-MÜLLGES, G. AND REUTTE, F. (1993). *Numerik-Algorithmen mit Fortran 77-Programmen*. Wissenschaftsverlag Mannheim, 7. Auflage.

ENGELN-MUELLGES AND REUTTER (1996). *Numerik- Algorithmen*. VDI Verlag.

FAZIO, T. D. AND WHITNEY, D. (1987). Simplified generation of all mechanical assembly sequences. *IEEE Trans. an Robotics and Automation*, RA-3(6):640–658.

FAZIO, T. L. D., RHEE, S. J., AND WHITNEY, D. E. (1999). Design-Specific Approach to Design for Assembly (DFA) for Complex Mechanical Assemblies. *IEEE Transactions on Robotics and Automation*, 15(5):869–881.

FERCH, M. (1997). Lernende, kraftsensorbasierte Regelung eines Zweiarmrobotersystems. Master's thesis, Universität Bielefeld.

FERCH, M. AND ZHANG, J. (2001). Learning cooperative grasping with the graph representation of a state action space. In *9th EUROPEAN WORKSHOP ON LEARNING ROBOTS* .

FERCH, M., ZHANG, J., AND KNOLL, A. (1999). Robot skill transfer based on B-spline fuzzy controller for force-control tasks . In *In Proceedings of the IEEE International Conference on Robotics and Automation, Detroit*, ICRA.

FININ, T., FRITZSON, R., MCKAY, D., AND MCENTIRE, R. (1994). KQML as an Agent Communication Language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, Maryland. ACM Press.

FINK, G. A., JUNGCLAUS, N., RITTER, H., AND SAGERER, G. (1998). Entwicklung verteilter Musteranalysesysteme mit DACS. In Levi, P., Ahlers, R.-J., May, F., and Schanz, M., editors, *Mustererkennung 98, 20. DAGM-Symposium Stuttgart, Informatik aktuell*, pages 429–436, Berlin. Springer-Verlag.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. (2000). PVM A Parallel Virtual Machine . `http://www.epm.ornl.gov/pvm/pvm_home.html`.

GITTINS, J. C. (1989). *Multi-armed bandit allocation indices*, volume XII of *Wiley-Interscience series in systems and optimization*. Willey, Chicester, NY.

HART, P., NILSON, N., AND RAPHAEL, B. (1968). A formal basis for the heuristic determination of minimum cost path. *IEEE Trans. Syst., Sci., Cyber.*, SSC-4(2):100–107.

HAYWARD, V. AND PAUL, R. (1986). Robot manipulator control under Unix RCCL: A robot control "C" library. *Int. Journal of Robotics Research*.

HOFFHENKE, M. AND WACHSMUTH, I. (1997). Dynamische Konzeptualisierung mit imaginalen Prototypen. Technical Report Report 97/9, SFB 360 / Universität Bielefeld.

HOWARD, R. A. (1960). *Dynamic Programming and Markov Prozesses*. The MIT Press, Cambridge, MA.

HU BER, J. S. (1996). *Evolutionary Computation: Theory and Applications*, chapter A general method for multi-agent learning and incremental self-improvement in unrestricted environment. Scientific Publ. Co., Singapore.

JENTSCH, W. AND KADEN, F. (1984). Automatic generation of assembly sequences. *Artificial Intelligence and Information-Control Systems of Robots*, pages 197–200.

JONES, R. AND WILSON, R. (1996). A Survey of Constraints in Automated Assembly Planning. In *In Proc. IEEE int. Conf. Robotics and Automation*, pages 1525–1532.

KAELBLING, L. P. (1993). *Learning in Embedded Systems*. The MIT Press, Cambridge, MA.

KAELBLING, L. P. AND LITTMAN, M. L. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, pages 237–285.

KURTZ, S. (1996). Approximate String Searching under Weighted Edit Distance. In *Proceedings of Third South American Workshop on String Processing*.

LEE, S. (1994). Subassembly Identification and Evaluation for Assembly Planning. *IEEE Transaction on Systems, Man and Cybernetics*, 24(3):493–503.

LEVESQUE, H. J., REITER, R., LESPÉRANCE, Y., LIN, F., AND SCHERL, R. B. (1994). GOLOG: A Lerning Programmin Language For Dynamic Domains. *The Journal of Logic Programming*, 19(20):1–25.

LI, X. (1995). *Repräsentation und exakte Konvertierung von Freiformflächen auf der Basis eines integrierten Produktmodellierers*. PhD thesis, Forschungsberichte aus dem Institut für Rechneranwendung in Planung und Konstruktion der Universität Karlsruhe.

LIEBERMANN, L. AND WESLEY, M. (1977). Autopass: An automatic programming system for computercontrolled mechanical assembly. *IBM Journal of Research and Development*, 21:321–333.

MATTIKALLI, R., BARAFF, D., AND KHOSLA, P. (1994). Finding all gravitationally stable orientations of assemblies. In *IEEE Int. Conf. on Robotics and Automation*, pages 251–257.

MCCARTHY, J. AND HAYES, P. (1969). *Machine Intelligence*, volume 4, chapter Some philosophical problems from the standpoint of artificial intelligence, pages 463–502. Edinburgh University Press, Edinburgh, Scotland.

MITCHELL, T. M. (1997). *Machine Learning*. Mc Grawhill New York.

MOOR, A. (1990). *Efficient Memory-Based Learning of Robot Control*. PhD thesis, University of Cambridge.

MOSEMANN, H. (2000). *Beiträge zur Planung, Dekomposition und Ausführung von automatischen generierten Roboteraufgaben.* PhD thesis, Tech. Univ. Braunschweig.

MOSEMANN, H., RÖHRDANZ, F., AND WAHL, F. (1997). Stability Analysis of Assemblies Considering Friction. *IEEE Transaction on Robotics and Automation*, 13(6).

MOSEMANN, H., RÖHRDANZ, F., AND WAHL, F. (1998). Assembly Stability as a Constraint for Assembly Sequence Planning. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, ICRA.

PAUL, R. P. (1982). *Robot manipulators: mathematics, programming, and control; the computer control of robot manipulators*. MIT Pr.

PRENTING, T. O. AND BATTAGLIN, R. M. (1964). The precedence diagram: A tool for analysis in assembly line balancing. *The Journal of Industrial Engeneering*, XV(4):208–213.

PRESMAN, E. AND SONON, I. (1990). *Sequential Control with Incomplete Information*. Academic Press.

PRINZ, M., LIU, H.-C., O. NNAJI, B., AND LUERTH, T. (1996). From CAD-Based Kinematic Modelling to Automated Robot Programming. *Robotics and Computer-Integrated Manufacturing*, 12(1):00–109.

PUTERMAN, M. L. (1994). *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc, New York, NY.

RABEMANANTSOA, M. AND PIERRE, S. (1996). An artificial intelligence approach for generating assembly sequences in CAD/CAM. *Artificial Intelligence in Engineering*, pages 97–107.

REQUICHA, A. (1980). Representations for rigid solids: Theory, methods, and systems. *ACM Computing Survey*, 12(4):437–464.

REQUICHA, A. AND WHALEN, T. (1991). *Computer-aided mechanical assembly planning*, chapter Representations for Assemblies, pages 15–39. Kluwer Academic Press Boston.

REQUICHA, A. G. (1977). Mathematical models of rigid solid objects. Technical Report 28, Univ. of Rochester.

SACHS, L. (1969). *Statistische Auswertungsmethoden*. Springer-Verlag, Berlin, 2. Auflage.

SAMET, H. (1990). *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley.

SCHERER, T. (2000). Roboterisierung des Probenmanagements in der Biotechnologie. GVV/DECHEMA Prozessmesstechnik in der Biotechnologie Diskussionstagung Bamberg.

SCHMIDHUBER, J. (1991). Curious model-building control system. In *International Joint Conference on Neural Networks, Singapore*, volume 2, pages 1458–1463. IEEE.

SCHMIDT, D. C. (2001). The ADAPTIVE Communication Environment (ACE). An Object Oriented Network Programming Toolkit in C++ . `http://www.cs.wustl.edu/~schmidt/ACE.html`.

SCHMIDT, R. (1998). Ein robustes Verfahren zur Roboter- Feinpositionierung durch visuelles Lernen. Master's thesis, University of Bielefeld, Fakulty of Technology.

SCHNEIDER, G. (1998). Steuerungstechnik für Industrieroboter. KUKA Roboter GmbH, `http://www.kuka-roboter.de/webc/rd_deut/index.html`.

SCHNEIDER, J. G. (1994). High Dimension Action Space in Robot Skill Learning. Technical report, The University of Rochester.

SEDGEWICK, R. (1992). *Alorithmen*. Addison-Wesley.

SIMPSON, W. (1994). PPP in HDLC-like Framing. STD 51, RFC 1662, Daydreamer, `http://www.ietf.org/rfc/rfc1662.txt?number=1662`.

SINGH, S. P. (1996). Reinforcement Learning with replacing eligibility traces. *Machine Learning*, 22(1).

STEVENS, W. (1998). *Networking APIs, Sockets and XTI*, volume Vol. 1 of *UNIX Network Programming*. Prentice Hall International.

STONE, P. (1998). *Layered Learning in Multi-Agent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University.

SUTTON, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44.

SUTTON, R. S. (1990). Integrated architectures for learning, planning and reacting based on approximating dynamic programming. In *International Conference on Machine Learning*. Morgen Kaufmann.

SUTTON, R. S. AND BARTO, A. G. (1999). *Reinforcement Learning (An Introduction)*. A Bradford Book, MIT Press Cambridge, London, 2 Auflage.

SYYID, U. (2000). The Adaptive Communication Environment: ACE (A Tutorial) . `http:://www.cs.wustl.edu/~schmidt/PDF/ACE-tutorial.pdf`.

TAKAHASCHI, Y. AND ASADA, M. (2000). Vision-guided behavior acquisition of a mobile robot by multi-layer reinforcement learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 395–402. IEEE.

THOMAS, J. AND NISSANKE, N. (1996). An algebra for modelling assembly tasks. *Mathematics and Computers in Simulation*, pages 639–659.

THRUN, S. B. (1992). *Handbook of Intelligent Control (Neuro, Fuzzy and Adaptive Aproaches)*, chapter The role of exploration in learning control. Van Nostrand Reinhold, New York.

TUNG, C.-P. AND KARK, A. C. (1996). Integrating Sensing, Task Planning, and Executing for Robotic Assembly. *IEEE Transactions on Robotics and Automation*, 12(2):187–201.

TZAFESTAS, S. G. AND STAMOU, G. B. (1997). Concerning automated assembly: knowledge-based issues and a fuzzy system for assembly under uncertainty. *Computer Integrated Manufacturing Systems*, 10(3):183–192.

VALLE, C. D. AND CAMACHO, E. (1996). Automatic Assembly Task Assignment for a Multi Robot Environment. *Control Eng. Practice*, 4(7):915–921.

VAN HOLLAND, W., BRONSVOORT, W., AND JANSEN, F. (1995). *Graphics and Robotics*, chapter Feature Modelling for Assembly, pages 131–148. Springer-Verlag, Berlin.

V.FAY-WOLF, DIPIPPO, L., COOPER, G., JOHNSTON, R., KORTMANN, P., AND THURAISINGHAM, B. (2000). Real-Time CORBA. *Parallel And Distributed Systems*, 11(10):1073–1089.

VON COLLANI, Y. (2001). *Repräsentation und Generalisierung von diskreten Ereignisabläufen in Abhängigkeit von Multisensormustern*. PhD thesis, Universität Bielefeld - Technische Fakultät.

VON COLLANI, Y., FERCH, M., ZHANG, J., AND KNOLL, A. (2000). A General Learning Approach to Multi sensor Based Control using Statistical Indices. In *Proc. of the IEEE Int. Conf. on Robotics and Automation, San Francisco, California*.

WATKINS, C. J. C. H. (1989). *Learning from Delayed Reward*. PhD thesis, King's College, Cambridge, UK.

WEILER, K. (1986). *Topological Structures for Geometric Modeling*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY.

WENG, J., EVENS, C. H., HWANG, W. S., AND LEE, Y.-B. (1998). The Development Approach to Artificial Intelligence: Concepts, Developmental Algorithms and Experimental Results. Technical Report MSU-CPS-98-25, Department of Computer Science and Engineering, Michigan State University, Est Lansing, MI 48824.

WENGEREK, T. (1995). *Reinforcement-Lernen in der Robotik*. PhD thesis, Universität Bielefeld - Technische Fakultät.

WILLIAMS, R. J. AND BAIRD, L. C. (1993). III. Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning system. Technical Report NU-ccS-93-11, Northeastern University, College of Computer Science, Boston, MA.

WOLTER., J. (2000). Assembly Sequence Planning Terminology . `http://www.cs.tamu.edu/research/robotics/Woter/asp/terms.html`.

WOLTER, J. D. (1989). On the Automatic Generation of Assembly Plans. In *Proc. IEEE Intl. Conf. on Robotics & Automation, Scottsdale, AZ*, pages 14 – 19.

WOO, T. C. (1985). A Combinatorial Analysis of Boundary Data Structure Schemata. *IEEE Computer Graphics and Applications*, 5(3):21–40.

WYNNE, W. H., FUH, J., AND ZHANG, Y. (1998). Synthesis of design concepts from a design for assembly perspective. *Computer Integrated Manufacturing Systems*, 11(1-2):1–13.

ZHA, X. (1999). An Integrated Intelligent Approach and System for Rapid Robotic Assembly Prototyping, Planning and Control. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation*.

ZHANG, J. AND FERCH, M. (1998). Rapid On-Line Learning of Compliant Motion for Two-Arm Coordination. In *Proceedings of the IEEE International Conference on Robotics and Automation, Leuven, Belgium*, ICRA.

ZHANG, J. AND FERCH, M. (2001). Extraction and Transfer of Fuzzy Control Rules for Sensor-Based Robotic Operations. *Fuzzy Sets and Systems*. (accepted).

ZHANG, J., FERCH, M., AND KNOLL, A. (1999a). Extraction and Transfer of Fuzzy Control Rules for Sensor-Based Robotic Operations. In *International Fuzzy Systems Association World Congress*, Taipei.

ZHANG, J., FERCH, M., AND KNOLL, A. (1999b). Two-Arm Coordination Control by Learning on a B-Spline Model. In *16th IAARC/IFAC/IEEE International Symposium on Automation and Robotics in Construction*, Madrid, Spain.

ZHANG, J., FERCH, M., AND KNOLL., A. (2000.). Carrying heavy objects by multiple manipulators with self-adapting force control. In *Proc. of Intelligent Autonomous Systems*, pages 204–211.

ZHANG, J. AND KNOLL, A. (1997). Incrementally Constructing Sensor-Based Behaviours with B-Spline Basis Functions. In *Robotics a. Automation*.

ZHANG, J. AND KNOLL, A. (1999). Designing fuzzy controllers by rapid learning. *Fuzzy Sets and Systems*, 101(2).