Werkzeuggestützte Entwicklung kooperativer Agenten im Dienstkontext

Stefan Fricke

Doktorarbeit

Berlin 2000

Werkzeuggestützte Entwicklung

kooperativer Agenten im Dienstkontext

vorgelegt von
Diplom-Informatiker
Stefan Fricke
aus Berlin

Vom Fachbereich 13 - Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurswissenschaften
- Dr.-Ing. -
genehmigte Dissertation

Promotionsausschuß:

Vorsitzender: Prof. Dr. K. Obermayer
Berichter: Prof. Dr. H. Krallmann
Berichter: Prof: Dr. E. Konrad

Tag der wissenschaftlichen Aussprache: 18. April 2000

Berlin 2000

D 83

Vorwort / Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit in Drittmittel-geförderten Forschungsprojekten als wissenschaftlicher Mitarbeiter am DAI-Labor im Fachgebiet Systemanalyse und EDV des Fachbereichs Informatik an der Technischen Universität Berlin.

In der Zeit von 1995 bis 1996 war die Realisierung von Werkzeugen zur Entwicklung von kooperierenden Diensten für kommunikationsbasierte Systeme (REkoS) Gegenstand meiner Projektarbeit am DAI-Labor. Dieses von der Deutschen Telekom Berkom GmbH geförderte Projekt hatte die Entwicklung einer agentenbasierten Diensteplattform zum Ziel, im Rahmen des Projekts kam ich zum ersten Mal mit den Agententechnologien in Berührung. Die Ergebnisse von REkoS stellen die wesentliche Grundlage für den praktischen Teil der Dissertation dar.

Später arbeitete ich in weiteren Forschungs- und Anwendungsprojekten am DAI-Labor. Mit der Entwicklung weiterer Agentenarchitekturen und insbesondere auch durch die Erstellung vergleichender Studien komplettierte sich mein Wissen über den Stand der Technik in diesem Forschungszweig der Informatik. Die hierbei gesammelten theoretischen und praktischen Erfahrungen prägen den theoretischen Teil meiner Doktorarbeit.

Mein besonderer Dank gilt Herrn Prof. Dr. Krallmann, der als wissenschaftlicher Leiter der Projekte meine Arbeit über den gesamten Zeitraum unterstützt und gefördert hat und mir im Rahmen meiner Projekttätigkeit den notwendigen Freiraum zur Entwicklung und Reifung der Dissertation gewährte. Ebenso bedanke ich mich bei Prof. Dr. Konrad für die Übernahme des Koreferats.

Diese Arbeit wäre ohne den Rückhalt der Kollegen vom REkoS-Projekt nicht möglich gewesen; mein Dank gilt Hermann Többen, Bernhard Bamberg und Ulrich Meyer. Besonderen Dank verdient mein direkter Vorgesetzter Dr. Albayrak, der mich inhaltlich unterstützte und antrieb.

Ich widme meine Arbeit dem Weltfrieden, denn es gibt wichtigere, existentielle Dinge als das eigene berufliche Fortkommen.

Stefan Fricke

Kurzfassung

Agentenorientierte Techniken (AOT) befassen sich mit der Realisierung verteilter, kommunizierender, selbst-koordinierender Systeme, deren Elemente Software-Agenten sind. Viele Untersuchungsgegenstände und Ziele der AOT finden sich auch in der Welt der Telematikdienste wieder: Hier fordert die Vielzahl von Anbietern und zunehmende Komplexität der Dienste offene Diensteplattformen, auf denen Dienste bereitgestellt und genutzt werden können. Für das Funktionieren solcher Plattformen sind intelligente und flexible Abläufe sowie Kontrollmöglichkeiten notwendig.

Diese Arbeit beschreibt die Konzeption und Implementierung einer agentenbasierten Entwicklungsumgebung für diensterbringende Agenten, bestehend aus Agentenprogrammiersprache, Agentenarchitektur und Testbett. Die Programmierung von Agenten findet werkzeugunterstützt mit Hilfe von vier Spezifikationssprachen statt: Dienste werden als Fähigkeiten deklarativ in Form von Tasks und prozedural durch eine modulare Skriptsprache beschrieben. Zielgerichtetes Verhalten wird durch die Beschreibung der Intentionalität eines Agenten mittels Motivationen, Zielen und Sensing-Alarms ausgedrückt. Sprechakte dienen für einfache soziale Interaktionen wie Dienstnutzungen. Schließlich lassen sich mit Hilfe von rollenbasierten Interaktionsprotokollen komplexere Verhandlungen mit mehreren Teilnehmern definieren.

Eine im Kern reaktive, komponentenbasierte Agentenarchitektur mit mächtiger Managementfunktionalität bildet das Grundgerüst der generischen Fähigkeiten eines jeden Agenten. In die Architektur sind Features wie thread-parallele Verarbeitung, dynamisches priorisiertes Scheduling, Laufzeitdatenprotokollierung, intelligente Handlungsauswahl, Datenpersistenz und ein einfacher Lernmechanismus integriert. Fehlersituationen werden durch Überwachung von Laufzeitdaten wie Kosten und Zeit schnell erkannt und durch Auswahl alternativer Handlungsskripte oder Ausschreibung des Dienstes an andere Agenten flexibel behandelt. Durch die Architektur wird jeder Agent zu einem flexiblen, koordinierten und kommunizierenden Informationsverarbeiter.

Vervollständigt wird die Entwicklungsumgebung durch ein Testbett zum empirischen Testen und Debuggen. Es besteht aus einer Menge graphischer Monitore, die die Abläufe auf der Ebene der Agentenprogrammiersprache sichtbar machen und kontrollierte Eingriffe erlauben. Das Testbett erlaubt sowohl das Debugging einzelner Agenten, die Untersuchung kompletter Szenarien als auch die Generierung und Auswertung von Laufzeitprotokollen.

Abstract

Agent-oriented techniques aim at the realization of distributed, communicating, and self-coordinating systems on the basis of so-called software agents. Many of the scientific topics do also apply to the field of telecommunication applications: The presence of competing service providers and the growing complexity of the services being delivered raise the need for open service platforms supporting the creation as well as the delivering of services. Intelligent and flexible control mechanisms are characteristic features for such platforms.

This thesis describes the design and implementation of a development environment for the realization of agent-based services comprising an agent programming language, an agent architecture, and a testbed. The agent programming process is supported by graphical editors on top of four specification languages: Services are specified declaratively by the concept of tasks and procedurally with a modular scripting language. The mental state of an agent constitutes goal-directed behaviour and is described by means of motivations, goals, and sensing alarms. Simple social interactions can be expressed by speech acts whereas rather complex negotiations between several parties are to be specified with a role-based interaction protocol language.

A component-based reactive agent architecture enhances agents with powerful generic management capabilities: Integrated into the architecture are capabilities of thread-parallel execution, dynamic prioritized scheduling, logging of runtime data, intelligent action selection, a persistence mechanism, and a learning facility. The occurence of errors is detected automatically and handled in a flexible manner, either by selecting alternative scripts or delegating tasks to other agents. Based on the architecture´s generic management capabilities agents are flexible, self-coordinating, communicating information processors.

Being distributed, autonomous systems, the intended behaviour of agent-based systems is a difficult task to ensure. It is the purpose of the agent testbed to allow empirical testing and debugging. Several monitor applications for the particular aspects of the agent programming language constitute the testbed. With these tools causal relationships inside the agents as well as in an agent society can be monitored, and modified by controlled experimentation. Furthermore, runtime protocols can be generated and evaluated.

Inhaltsverzeichnis

Vorwort / Danksagung **1**

Kurzfassung **2**

Abstract **3**

Inhaltsverzeichnis **4**

Kapitel 1 Einleitung **14**

1.1 Hintergrund 15
1.2 Problemstellung 17
1.3 Lösungsansatz 19
1.4 Beitrag dieser Arbeit 20
1.4.1 Abgrenzung 21
1.5 Aufbau der Dissertation 22

Teil A
Techniken zur Implementierung von Agenten **26**

Kapitel 2 Fähigkeiten, Methoden und Repräsentation 28
2.1 Charakteristische Merkmale von Softwareagenten ...29
2.1.1 Notwendige Eigenschaften ...29
2.1.2 Optionale Eigenschaften ...31
2.1.3 Klassifizierung von Agenten ...32
2.1.4 Distributed problem solving und multi-agent systems ...35
2.1.5 Eigenschaften von Multiagentensystemen ...36
2.2 Kommunikation ...37
2.2.1 Sprechakte ...39
2.2.2 Interaktionsprotokolle ...41
2.2.3 Zusammenfassung ...41
2.3 Koordination ...42
2.3.1 Normen und Rollen ...44
2.3.2 Spieltheorie ...45
2.3.3 Scheduling ...46
2.3.4 Contract-Net-Protokoll ...46
2.3.5 Auktionen ...48
2.3.6 Andere Koordinationsprotokolle ...49
2.3.7 Vermittlung und Organisation durch Faciltiators ...50
2.3.8 Marktmechanismen ...51
2.3.9 Zusammenfassung ...52
2.4 Aktorische und kognitive Fähigkeiten ...54
2.4.1 Prozedurale Repräsentationsformen ...54
2.4.2 Modallogiken ...56
2.4.3 Zeitliches Wissen ...58
2.4.4 Planung und Planausführung ...59
2.4.5 Lernen und Adaption ...61
2.4.6 Zusammenfassung ...62
2.5 Zusammenfassung und Bewertung ...63

Kapitel 3 Agentenarchitekturen 66
3.1 Informationsfluß und -verarbeitung ...67
3.2 Komponentenarchitekturen ...69
3.3 Geschichtete Architekturen ...70
3.4 Reaktive Architekturen ...73
3.5 Kognitive Architekturen ...75
3.6 BDI-Architekturen ...78
3.7 Zusammenfassung und Bewertung ...80

Kapitel 4 Agentenprogrammiersprachen 84

4.1 Agent-0 ... 85
4.1.1 Methodik ... 89
4.1.2 Bewertung ... 90
4.2 dMARS ... 91
4.2.1 Methodik ... 94
4.2.2 Bewertung ... 96
4.3 Agent Building Shell / COOL ... 97
4.3.1 Methodik ... 99
4.3.2 Bewertung ... 100
4.4 TÆMS / GPGP ... 101
4.4.1 Methodik ... 105
4.4.2 Bewertung ... 105
4.5 Weitere Ansätze zur Agentenprogrammierung ... 106
4.6 Testen und Debugging ... 108
4.7 Agentenorientiertes Programmieren als neues Programmierparadigma? ... 110
4.8 Der REkoS-Beitrag zum agentenorientierten Programmieren ... 112
4.9 Zusammenfassung und Bewertung ... 113

Teil B

REkoS 116

Kapitel 5 Sprachen und Definitionswerkzeuge in REkoS 118

5.1 Der REkoS-Ansatz ... 118
5.2 REkoS-Glossar ... 120
5.2.1 Intentionalität ... 120
5.2.2 Interaktivität ... 121
5.2.3 Aktorische Fähigkeiten ... 121
5.2.4 Globaler und lokaler Wissenskontext ... 121
5.3 Definition von Sprechakten ... 123
5.3.1 Das Sprechaktmodell ... 123
5.3.2 Attribute eines Sprechakts ... 125
5.3.3 Begriffswelt ... 126
5.3.4 Werkzeuge zur Definition von Sprechaktmodellen ... 127
5.4 Beschreibung des mentalen Zustands ... 130

5.4.1 Variablen 131
5.4.2 Motivationen....132
5.4.3 Ziele....133
5.4.4 Normative Ziele....133
5.4.5 Tasks....134
5.4.6 Werkzeuge für die Modellierung des mentalen Modells....136
5.5 Programmierung von Fähigkeiten....137
5.5.1 Das Skriptmodell....137
5.5.2 Skriptsprache....138
5.5.3 Werkzeuge für die Spezifikation von Fähigkeiten....138
5.6 Spezifikation von Interaktionsprotokollen....140
5.6.1 Protokollklassen....140
5.6.2 Rollen....141
5.6.3 Sprache zur Beschreibung von Interaktionsprotokollen....141
5.6.4 Spezifikationswerkzeug für Interaktionsprotokolle....143
5.7 Integration....145
5.7.1 Codegenerierung für Sprechaktdefinitionen....145
5.7.2 Codegenerierung für die Intentionalitätsbeschreibungssprache....147
5.7.3 Codegenerierung für die Skriptsprache....147
5.7.4 Kreierung von Agenten....150
5.8 Zusammenfassung und Bewertung....151

Kapitel 6 Eine Agentenarchitektur für Dienste 154

6.1 Anforderungen an die Architektur....155
6.2 Das Architekturmodell....157
6.3 Interpreter....159
6.4 Intentionalitätsmodul....162
6.4.1 Verwaltung von Instanzen....163
6.4.2 Vorwärtspropagierung von Motivationen....164
6.4.3 Skriptauswahl....165
6.4.4 Rückwärtspropagierung von Skriptergebnissen....166
6.5 Wissensbasis....168
6.5.1 Sammlung von Erfahrungswissen....168
6.5.2 Persistente Datenhaltung....170
6.6 Scheduler....171
6.6.1 Priorisierung von Skripten....171
6.6.2 Aktivitätenmanagement....172
6.7 Skriptmanager....174

6.8 Kooperationsmanager ... 177
6.9 Kommunikationsmodul ... 178
6.9.1 Agentendämon ... 179
6.10 Zusammenfassung und Bewertung ... 179

Kapitel 7 Debugging 182

7.1 Konzept für das Debugging verteilter Systeme ... 183
7.1.1 Debugging einzelner Agenten ... 183
7.1.2 Debugging von Agentensystemen ... 184
7.1.3 Architektur des Debuggers ... 184
7.2 Agentendebugging ... 186
7.3 Die Testbettapplikation ... 187
7.4 Überwachung des mentalen Zustands ... 188
7.5 Debugging von Skripten ... 189
7.6 Kommunikationsmonitor ... 192
7.7 Laufzeitprotokolle ... 194
7.7.1 Auswertung von Protokollen ... 195
7.8 Weitere Debugging-Werkzeuge ... 196
7.9 Zusammenfassung und Bewertung ... 197

Kapitel 8 Zusammenfassung und Ausblick 200

8.1 Zusammenfassung ... 200
8.1.1 Der REkoS-Ansatz zum agentenorientierten Programmieren ... 201
8.1.2 REkoS-Agentenarchitektur ... 202
8.1.3 Bewertung ... 203
8.2 Vergleich mit anderen Ansätzen ... 204
8.3 Erweiterungen und Verbesserungen ... 206
8.3.1 Dienstvermittlung ... 207
8.3.2 Dienstontologie ... 207
8.3.3 Verbesserungen der Architektur ... 208
8.3.4 Security ... 209
8.3.5 Testbett ... 209

Anhang 212

A1 Grammatik für Skripte 212
A2 Ausschnitt einer generierten Step-Implementierungsdatei 212
A3 Ausschnitt aus einer generierten Skript-Headerdatei 213

Literaturverzeichnis 216

Lebenslauf 236

Abbildungsverzeichnis

Dienstnutzung mit Agenten 19
Struktureller Aufbau der Dissertation 23
Contract-Net-Protokoll 47
FIPA-Protokoll einer englischen Auktion 49
KQML-Agenten in einem föderierten System 51
Prinzipieller Aufbau einer Komponentenarchitektur 70
Prinzipieller Aufbau einer Schichtenarchitektur 71
TouringMachines-Agentenarchitektur 72
Strukturarmer Aufbau einer Agentenarchitektur 74
SOAR-Agentenarchitektur 76
BDI-Architektur nach Rao & Georgeff 79
BDI-Verarbeitungszyklus für BDI-Agenten 79
Agent-0-Interpreter 88
Agentenentwicklungsprozeß in AgentBuilder 90
Ausführung eines Plans in dMARS 93
Agenten-Klassendiagramm in dMARS 95
COOL-Konversation als endlicher Automat 98
Modellierung einer Task-Struktur in TÆMS 102
Sprechaktdefinitionstool 128
Aktivierungshierarchie der mentalen Begriffe 130
Werkzeuge zur Beschreibung der Intentionalität 136
Skript-Beziehungen 137
Bildschirmformulargestützte Spezifikation von Fähigkeiten 139
Graphische Repräsentation von Kooperationsrollen 143
Struktur und Verkettung von Steps (vereinfachte Darstellung) 148
Generierter Code eines Skripts (Ausschnitt) 149
Die REkoS-Agentenarchitektur 158
Interpreterschleife 160
Begründungshierarchie durch Reason-Pointer 163
Beziehungen zwischen generierten und Architekturklassen 175
Ausschnitt aus der Realisierung der Script-Klasse 176
Kommunikation zwischen Testbett und Agenten 185
Bedienelemente zur Kontrolle eines Agenten 186
Einstiegsmenü des Testbetts 188
Testbettwerkzeuge zur Kontrolle des mentalen Zustands 189
Skript-Debugger 190
Script-Klasse mit integriertem Testbett-Code 191

Kommunikationsmonitor 193
Werkzeug zur Auswertung von Laufzeitprotokollen 195
Präsentation von aggregierten Laufzeitdaten 196

Tabellenverzeichnis

TABELLE 1. Klassifikation von Agenten 33
TABELLE 2. Sichtbarkeit der REkoS-Begriffswelt 122
TABELLE 3. Basis-Sprechaktmodell für REkoS-Agenten 127
TABELLE 4. Synchronisationsprimitive in Kooperationsskripten 144
TABELLE 5. Übersicht über generierten Code für Handlungsskripte ... 149
TABELLE 6. verarbeitbare Nachrichten des Interpreters 161
TABELLE 7. verarbeitbare Nachrichten des Intentionalitätsmoduls 167
TABELLE 8. verarbeitbare Nachrichten des Schedulers 173
TABELLE 9. verarbeitbare Nachrichten des Skriptmanagers 176
TABELLE 10. Agentenentwicklungssysteme im Vergleich 206

KAPITEL 1 Einleitung

Der Telekommunikationsmarkt unterliegt einem ständigen Wandel. Mit der Privatisierung dieses Marktes konkurrieren zahlreiche Netz- und Dienstanbieter um die Gunst der Kunden. Der Konkurrenzkampf läßt die Gewinnmargen beim Anbieten von Kommunikationsbandbreite und einfachen Diensten sinken, so daß die Telekommunikationsanbieter nach neuen Ertragsquellen Ausschau halten müssen. Die Lösung liegt in auf den Kunden zugeschnittenen Inhalten. Dabei muß es sich nicht um neuartige Dienste handeln, auch in der Kombination bestehender Dienste und Informationsquellen liegen Mehrwertpotentiale.

Aus technologischer Sicht steht die Eintrittspforte in die Informations- und Dienstleistungsgesellschaft weit offen: Kommunikationskanäle sind bis in die Haushalte hinein verfügbar, die Leistungsfähigkeit marktüblicher PCs ermöglicht komplexe Anwendungen, und nicht zuletzt steht mit dem Internet ein netzwerktransparentes Medium zur Informationsdistribution zur Verfügung. Dennoch sind einige Hindernisse auf dem Weg in die Telematikwelt[1] zu überwinden: Vor dem Hintergrund eines stetig wachsenden Informationsdschungels, charaktierisiert durch eine zunehmende Informationsmenge bei gleichzeitig abnehmender Halbwertzeit der Gültigkeit, stellen das Auffinden, Zusammenstellen und bequeme Nutzen von Inhalten und Informationsdiensten große Herausforderungen dar.

1. Telematik ist ein Kunstwort, das aus der Verschmelzugn von *Tele*kommunikation und Infor*matik* entstanden ist [Flusser 1985], S.86.

1.1 Hintergrund

Der Dienstbegriff sperrt sich einer klaren, eindeutigen Definition. Im Kontext dieser Arbeit wird unter einem Dienst eine zu erbringende Leistung verstanden, die in unterschiedlichen Ausprägungen erbracht werden kann. Von besonderer Relevanz ist der deklarative Aspekt der Dienstbeschreibung: Nicht *wie* der Dienst erbracht wird ist wichtig, sondern *was* ein Dienst leistet.

Im folgenden werden Dienste aus Sicht der verschiedenen Beteiligten untersucht. Es lassen sich drei Rollen identifizieren: Dienstnutzer, -anbieter und -vermittler. Eine Dienstnutzung gliedert sich in drei Phasen. Am Anfang steht die Suche nach einem Dienstanbieter. Ist dieser gefunden, erfolgt die Spezifikation der Aufgabe, bevor in der letzten Phase der Dienst erbracht wird. Aus Sicht eines Dienstanbieters stellen die Realisierung, das Anbieten und Warten von Diensten wichtige Aspekte dar. Dienstvermittler stellen als Anbieter von Infrastruktur das Bindeglied zwischen Anbietern und Nutzer dar.

Dienstsuche

Dienstleister und Nachfrager können auf unterschiedlichen Wegen zueinander finden. Die Hauptinteressen der Anbieter liegen hierbei in guter Erreichbarkeit und der Möglichkeit Kunden längerfristig zu binden. Als Lösung bietet sich die Präsenz auf einer Portalseite im Internet in Zusammenhang mit kundenfreundlichen Zusatzleistungen wie Warenkörben, Benutzerprofilen und Active Trading an.

Auf Seiten des Dienstnachfragers sind die Interessen anders gelagert: Er möchte, in möglichst kurzer Zeit und mit geringem Aufwand, den passenden Anbieter für seine Aufgabenbeschreibung finden. Für das Auffinden von Anbietern oder Produkten kommen daher Suchdienste in Frage, die auf Content-Sprachen beruhen. Derartige Sprachen erlauben eine gezielte Beschreibung von Diensten unter Berücksichtigung von Begriffszusammenhängen und heben sich somit von den rein textbasierten Internet-Suchmaschinen ab[2].

Dienstspezifikation

Nachdem ein Dienstanbieter gefunden ist, kann der Dienstvertrag ausgehandelt werden. Hierbei werden über eine Dienstnutzungsschnittstelle die konkreten Dienstparameter festgelegt. Zu unterscheiden sind Schnittstellen, die ausschließlich von anderen Diensten genutzt werden und solche, die die Endbenutzer verwenden. Erstere zeichnen sich durch ein knappes, formales Interface aus, das die wesentlichen Nutzungsmerkmale - das sind Dienstname und die zugehörigen Parameter -

2. Beispielsweise das auf XML beruhende Resource Description Framework [Bray 1998].

enthält. Menschliche Nutzer hingegen benötigen einen komfortableren Zugang, typischerweise über grafische Schnittstellen mit zusätzlichen Informationen wie Hilfetexten.

In dieser Arbeit werden anwendungsunabhängige, d.h. generische, von domänenspezifischen Parametern unterschieden. *Generische Dienstparameter* sind Kosten, Zeit und Qualität, sie sind für die Beschreibung eines jeden Dienstes obligatorisch. Dabei sind Zeit und Kosten quantitative Größen, die bei einer Dienstanforderung als Obergrenzen angegeben werden. Durch den Zeitparameter wird ein Termin festgesetzt, zu dem der Dienst vollständig zu erbringen ist bzw. die Ergebnisse vorliegen müssen. Die Qualität sagt etwas über die Güte der Diensterbringung aus. Sie ist für eine Dienstanforderung optional, dient aber für Dienstanbieter zur besseren Unterscheidung ähnlicher Leistungen. Alle weiteren Leistungsmerkmale, die für die Nutzung eines Dienstes zu spezifizieren sind, werden unter dem Begriff *spezifische Dienstparameter* aufgeführt. Hierunter falle beispielsweise räumliche Koordinaten für eine Layoutberechnung.

Diensterbringung

In der letzten Phase erfolgt die Diensterbringung entsprechend den festgelegten Parametern. Die Dienstausführung kann lokal durch den Dienstbereitsteller oder aber verteilt durch Nutzung von Subdiensten anderer Anbieter erfolgen. Insbesondere für den letztgenannten Fall, in dem der Dienstbereitsteller die Verarbeitung teilweise aus der Hand gibt, läßt sich die Notwendigkeit eines Dienstmanagements herleiten. Zur funktionalen Unterscheidung der Managementaufgaben bedient man sich üblicherweise des aus dem Bereich des Netzwerkmanagement stammenden FCAPS-Modells[3]. Dort werden Aktivitäten zum Fehler-, Konfigurations-, Accounting-, Leistungs- und Sicherheitsmanagement aufgeführt.

Für die Diensterbringung von besonderem Interesse sind die Bereiche Accounting und Fehlermanagement: Damit erbrachte Dienste abgerechnet werden können, müssen entstandene Kosten protokolliert werden. Hierbei ist zu beachten, daß ein vorgegebener Rahmen nicht gesprengt wird. Außerdem kann für den Dienstnachfrager Kostentransparenz ein wichtiger Aspekt sein. In diesem Fall ist aufzuschlüsseln, welche Kosten wo angefallen sind. Analoges gilt für das Management der anderen Parameter. Unter das Fehlermanagement fallen alle Aktivitäten zur Erkennung von Störungen sowie anschließende behandelnde Maßnahmen. Dazu gehört das Monitoring des Systemverhaltens, die Generierung und das Weiterleiten von Fehlermeldungen wie auch Funktionalität zur Erhöhung der Fehlertoleranz. Mit dem Auftreten einer Störung sind alle davon betroffenen Aktivitäten zu identifizie-

3. FCAPS steht für Fault-, Configuration-, Accounting-, Performance- und Security-Management und ist Bestandteil des OSI-Managementframeworks [ISO 7498-4].

ren und neu zu bewerten. Maßnahmen können in der Unterbrechung oder Beendigung aktiver bzw. in der Verlagerung oder Verhinderung geplanter Handlungen liegen.

Dienstbereitstellung

Vor der möglichen Nutzung eines Dienstes steht dessen Realisierung und Bereitstellung. Zunächst ist bei der Implementierung auf Konformität der Dienstnutzungsschnittstellen zu achten, damit sich ein neuer Dienst nahtlos in ein bereits bestehendes Angebotsspektrum einfügen läßt. Neben der eigentlichen Funktionalität sind FCAPS-Managementfunktionen zu integrieren und entsprechende Schnittstellen für Überwachung, Kontrolle und Reports zu realisieren.

Hilfreich für die Bereitstellung und Wartung sind Dienstplattformen oder -architekturen wie TINA-C[4]. Sie bieten einen allgemeinen Rahmen für das Kreieren, Auffinden und Nutzen von Diensten, geben Managementfunktionalität bzw. entsprechende Schnittstellen vor und bieten u.U. weitere Infrastruktur wie Verschlüsselungs- und Authentifikationsmechanismsn an.

1.2 Problemstellung

Bevor im nächsten Abschnitt die thematischen Schwerpunkte und Lösunsgansätze der Dissertation vorgestellt werden, lohnt sich ein Blick auf die mit der Bereitstellung und Nutzung von Diensten verknüpften Probleme.

- Entwicklungsaufwand: Neben der eigentlichen Dienstfunktionalität sind Nutzungsschnittstellen und FCAPS-Managementaufgaben zu implementieren, die den Entwicklungsaufwand und damit den kritischen time-to-market deutlich erhöhen. Idealerweise würde eine Dienstarchitektur die notwendigen Mechanismen zum Erfassen von Laufzeitdaten, Monitoring und Kontrolle bereits als integrale Bestandteile beinhalten und damit den Programmierer entlasten. Analoges gilt für die Realisierung der Schnittstellen.
- Dienstspezifikation: Bei der Kreierung eines neuen komplexen Services sollten bereits existierende Basisdienste genutzt werden können. Wünschenswert ist eine Abkehr vom Prinzip der hart verdrahteten, detailliert beschriebenen Dienste hin zu einer deklarativen Dienstspezifikation. Voraussetzung hierfür ist eine Spezifikationssprache, mit der Dienst-Subdienst-Strukturen mit möglichst weitgehender Abstraktion von konkreten Abläufen, Nutzungsschnittstellen, Raum und Zeit ausdrückbar sind.

4. Telecommunications *I*nformation Networking Architecture Consortium [Chapman, Montesi 1995].

- Wartbarkeit: Das Angebotsspektrum einer Diensteplattform ist dynamisch: Neue Dienste kommen hinzu, alte werden entfernt, andere werden durch neue Versionen ersetzt. Für das Management einer solchen Plattform läßt sich daraus folgern, daß das Einfügen, Löschen und Austauschen von Diensten zur Laufzeit, möglichst ohne Leistungsabfälle des Gesamtsystems, möglich sein muß. Außerdem werden Test- und Analysewerkzeuge, wie sie im Netzwerkmanagement zum Einsatz kommen[5], zum Erkennen und Beheben von Performanceproblemen und Fehlersituationen benötigt.
- Dienstdelegation: Wenn Dienste wie im 2. Punkt beschrieben spezifiziert werden, kann die Verteilung von Subdiensten, anstatt fest verdrahtet zur Designzeit, flexibel zur Laufzeit erfolgen. Den größten Nutzen bieten dynamische Beauftragungen, die die aktuelle Situation wie Auslastung oder abgelaufene Zeit mit berücksichtigen. Hierfür werden Koordinierungsmechanismen benötigt, die ebenfalls integraler Bestandteil einer Dienstarchitektur sein sollten. Ein weiterer Vorteil dynamischer Delegation liegt in der stärkeren Offenheit derartiger Systeme: Neu hinzugekommene Dienste sind sofort erreichbar, entfernte oder temporär nicht erreichbare Services können flexibel umgangen werden.
- Kontrolle: Nicht nur die Nutzung von Subdiensten, sondern auch die internen Abläufe der Diensterbringung sollten einer flexiblen Kontrolle unterliegen. Bei starren Abläufen besteht beispielsweise die Gefahr, daß lange Aktionssequenzen zu entsprechend langen Antwortzeiten bzw. zur Überlastung oder gar zur temporären Nichterreichbarkeit eines Dienstanbieters führen. Wünschenswert ist ein intelligentes internes Aktivitätenmanagement, das auf paralleler Verarbeitung beruht oder mit Unterbrechungsmechanismen arbeitet. Auf diese Weise ist fokussiertes Bearbeiten priorisierter Aufgaben möglich; ferner sind Synergiepotentiale, beispielsweise durch Vermeidung von Mehrfachaktivierungen, nutzbar.

Für jedes der genannten Probleme existieren Lösungsansätze, die zumeist Insellösungen darstellen. Am weitesten reichen die im Bereich der Telekommunikationsanwendungen angesiedelten Ansätze TINA-C und IN[6], die Dienststeuerung und -vermittlung trennen. Die Stärken von IN liegen in der Interoperabilität; Basisdienste lassen sich auf einfache Weise zu neuen Services aggregieren. TINA beschreibt ein umfassendes Rahmenwerk zur Erstellung von Dienstszenarien und beruht dabei auf der objektbasierten Kommunikations-Middleware CORBA. Dort werden Dienste entsprechend als Objekte angesehen und mit einer Objektdefinitionssprache spezifiziert. Keiner der Ansätze deckt jedoch alle Problemfelder zufriedenstellend ab, insbesondere der Aspekt der internen Kontrolle, sowie bei TINA der hohe Entwicklungsaufwand und bei IN die intelligente Dienstdelegation finden nicht ausreichend Berücksichtigung.

5. Beispielsweise HP OpenView [HP 1998].

6. Intelligente Netze [Magedanz, Popescu-Zeletin 1996].

1.3 Lösungsansatz

Die Distributed Artificial Intelligence (DAI) ist ein relativ junges Teilgebiet der Informatik. Sie beschäftigt sich mit der Realisierung verteilter, intelligenter, kooperativer Systeme, deren Bausteine Software-Agenten sind. Ziel der vorliegenden Arbeit ist es, die Anforderungen der Dienstewelt mit den Vorteilen der Agententechnologien zu verschmelzen.

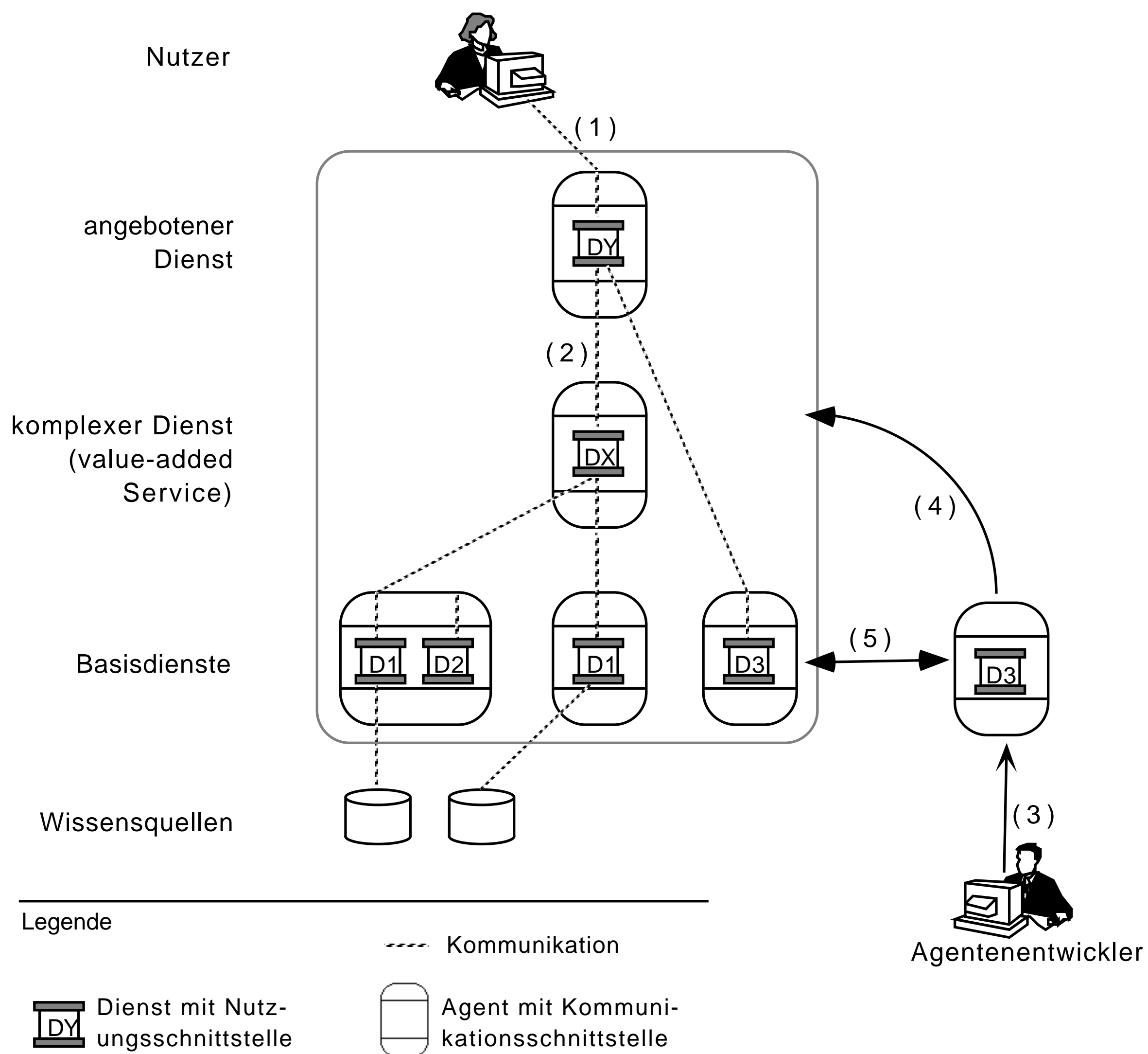

Abbildung 1-1: Dienstnutzung mit Agenten

Agenten realisieren Dienste, die sie menschlichen Benutzern über graphische Oberflächen (siehe Abb. 1-2 (1)) anbieten. Außerdem können Agenten über ihre interaktiven Fähigkeiten die Dienste anderer Agenten automatisch nutzen (2). Dies ermöglicht die Aggregation einfacher Basisdienste zu komplexen Mehrwertdiensten, sogenannten value-added services. Die Definition eines Dienstes besteht in der Entwicklung eines entsprechenden Agenten, der den Service als Fähigkeit nach außen anbietet (3). Ein Bestandteil dieser Arbeit ist die Entwicklung einer Agentenprogrammiersprache, die sich auf die Beschreibung der Dienstfunktionalität konzentriert.

Jeder Agent ist ein Informationsverarbeiter, der mit intelligenten Steuerungsmechanismen ausgestattet ist. Diese generischen Fähigkeiten beinhalten das Erfassen relevanter Laufzeitdaten, flexible Fehlerbehandlung, zielorientierte, fokussierte Verarbeitung, Aktivitätenkoordination sowie grundlegende Kommunikationsmechanismen. Mit Hilfe der zusätzlichen Fähigkeiten ist der Agent in der Lage zu erkennen, wann er einen Dienst nicht rein lokal, sondern in Kooperation mit anderen Anbietern abarbeiten muß. Gründe für die Auslagerung von Teilaktivitäten können in fehlender lokaler Funktionalität, Zeit-, Kosten- oder Qualitätsanforderungen begründet sein. Er kann dann nach passenden Anbietern von Subdiensten suchen und Aufgaben entsprechend delegieren, so daß ein dynamischer Verarbeitungsfluß entsteht.

Die Bereitstellung eines Dienstes erfolgt durch Integration des Agenten in ein Agentensystem (4). Weil Agenten als eigenständige, kommunikationsfähige Prozesse laufen, ist dieser Schritt trivial: Die Kommunikationsmechanismen beinhalten Informationsanfragen nach Diensterbringern, wodurch auch nach dem Hinzufügen, Entfernen oder Ersetzen eines Agenten (5) das aktuell angebotene Funktionsspektrum jederzeit systemtransparent und damit vollständig nutzbar ist.

1.4 Beitrag dieser Arbeit

Gegenstand dieser Arbeit ist die Konzeption und Implementierung einer Programmierumgebung für die Entwicklung kooperativer Agenten. Thematisch ist diese Arbeit damit im Bereich des agentenorientierten Programmierens angesiedelt. Der Schwerpunkt liegt weniger auf einer theoretischen Ebene, sondern es wird vielmehr ein pragmatischer Ansatz verfolgt. Dabei stehen softwaretechnische Aspekte wie Programmiersprache, Programmausführung und Performance im Zentrum des Interesses. Die wichtigsten Beiträge und Ergebnisse dieser Dissertation lassen sich wie folgt zusammenfassen:

- Kritische Betrachtung des Stands der Dinge im Bereich der agentenorientierten Techniken: Agenten werden häufig als Softwaretechnologie des nächsten Jahrzehnts beschworen und sollen die immer noch aktuelle Softwarekrise und die

zunehmende Komplexität informationstechnischer Anwendungen meistern helfen. Diese Arbeit zeigt auf, wo Stärken und Schwächen der Agententechnologie liegen und welche Anstrengungen noch unternommen werden müssen, um die vorhandenen Techniken zu einem neuen Programmierparadigma zu erheben.

- Bereitstellung einer Programmierumgebung: Der Tatsache, daß Agenten Fähigkeiten in verschiedenen Bereichen besitzen, wird dadurch Rechnung getragen, daß vier Spezifikationssprachen zur Beschreibung von Agenten entwickelt werden. Mit einer Sprache zur Beschreibung von aktorischen Fähigkeiten werden Dienste realisiert. Zur Beschreibung der einfachen Interaktionen zwischen Agenten dient die Kommunikationssprache. Mit ihr lassen sich Informationsaustausch oder Dienstanforderungen ausdrücken. Komplexere Kommunikationsformen wie Verhandlungen und Kooperationen werden durch die Interaktionsprotokollsprache abgedeckt. Zur Beschreibung des sogenannten mentalen Zustands kommt schließlich die Intentionalitätsbeschreibungssprache zum Einsatz, sie stellt die Basis für zielgerichtetes Handeln dar. Graphische Definitionswerkzeuge unterstützen eine konsistente Modellierung.
- Entwicklung einer Architektur für diensterbringende Agenten: Für die Ausführung des von den Agentenprogrammiersprachen generierten Codes wird eine komponentenbasierte Agentenarchitektur konzipiert und realisiert. Diese Ausführungsumgebung übernimmt neben einer effizienten und parallelen Verarbeitung wichtige dienstspezifische Aufgaben wie das Scheduling anstehender Aktivitäten und die Überwachung und Registrierung von Leistungsparametern. Auch ein einfacher Lernmechanismus ist in die Architektur integriert.
- Debuggingkonzept und Testwerkzeuge: Verteilte Systeme, wie sie Agentensysteme darstellen, bringen neue Fehlerklassen mit sich. Die Parallelität der Abläufe, eintretende und das System verlassende Agenten machen das Systemverhalten schwer nachvollziehbar. Aus diesem Grund wird eine werkzeuggestützte Test- und Debuggingumgebung bereitgestellt. Mit ihr kann das Verhalten einzelner Agenten auf der Ebene der Agentenprogrammiersprache verfolgt und modifiziert werden. Auch die Beobachtung und Eingriffe in das Systemgeschehen sind möglich, ferner können die Aktivitäten einer post-mortem-Analyse unterzogen werden.

1.4.1 Abgrenzung

Der in dieser Dissertation verfolgte Ansatz beruht auf komplexen, funktionsmächtigen, kommunizierenden Agenten. Aspekte wie Mobilität aber auch Intelligenz durch kognitive Methoden sind hier nicht relevant. Ebensowenig Gemeinsamkeiten bestehen zu „agentifizierter“ intelligenter Software wie E-Mail-Filter oder persönliche digitale Assistenten und zu tendenziell einfach gestrickten „Internetagenten“ oder den Namensvettern aus dem Bereich Netzwerkmanagement.

Vielfach wurde die Beziehung zwischen objektorientierten und Agentenprogrammiersprachen diskutiert[7]. Diese Arbeit vertritt die Ansicht, daß beide Ansätze voneinander verschieden sind; die hier entwickelten Sprachen sind deklarativ und modular. Es liegt nicht in der Intention der Dissertation, eine Software-Engineering-Methodologie für den Bereich der agentenorientierten Dienste zu erarbeiten, hierfür ist der verwendete Dienstbegriff zu allgemein gefaßt. Außerdem fußt die Realisierung von Anwendungen stark auf der hier implementierten Agentenarchitektur und Spezifikationssprachen, so daß die Entwicklungsmethodik schwerlich in andere Bereiche übertragbar ist. Ansätze, die in Richtung Erarbeitung einer agentenorientierten Softwareentwicklungsmethodik zielen, basieren entweder auf bekannten objektorientierten Verfahren[8], von denen sich diese Arbeit, wie anfangs des Absatzes erwähnt, abgrenzt, oder sind, gleich dem hier verfolgten Ansatz, auf ein bestimmtes Architekturmodell zugeschnitten[9].

Dadurch, daß der Schwerpunkt auf *Agenten*techniken liegt, treten *Dienst*aspekte in einigen Punkten in den Hintergrund: Schwerpunkte der Dissertation umfassen die Bereitstellung und flexible Erbringung von Diensten durch Softwareagenten; den Phasen Dienstsuche und Dienstspezifikation wird dagegen weniger Aufmerksamkeit geschenkt. Die deklarative Beschreibungssprache für Fähigkeiten stellt nur ein beschränktes Begriffsvokabular bereit, Ansätze, die sich mit dem Aufbau von Ontologien beschäftigen[10], bieten in bezug auf Interoperabilität und Ausdrucksmächtigkeit umfassendere Konzepte. Auch Sicherheitskonzepte, wie sie für kommerzielle Dienste unerläßlich sind, werden in der Dissertation nicht behandelt.

Stattdessen werden bestehende Techniken ausgewertet und in einem neuen Zusammenhang zur Erschließung des weiten Problemfeldes der kooperativen Telematikdienste integriert.

1.5 Aufbau der Dissertation

Die Dissertation gliedert sich in zwei Teile. Der erste Teil behandelt agentenorientierte Techniken, Agentenprogrammiersprachen und -architekturen. Hierbei werden existierende Ansätze untersucht und in Bezug zur Thematik dieser Arbeit gesetzt. Mit einer kritischen Abhandlung zum Stand der Dinge und der Vorstellung eines ei-

7. [Shoham 1993], [Genesereth, Ketchpel 1994], [Kinny, Georgeff 1996].

8. [Kinny, et al. 1996], [Chauhan 1997], [Wooldridge, et al 1999].

9. [Shoham 1993], [Smith, et al. 1994], [Barbuceanu, Fox 1996], [Reticular 1998].

10. Beispielsweise die wissensbasierte Ontologiebeschreibungssprache Ontolingua [Gruber 1992].

genen Ansatzes der agentenorientierten Programmierung schließt der erste Teil. Der zweite Teil beschreibt mit der REkoS[11]-Agentenentwicklungsumgebung den eigenen entwickelten Ansatz. Aufbauend auf den Erkenntnissen des ersten Teils in den Bereichen Agentenprogrammiersprachen und Techniken wird eine werkzeugunterstützte Agentenbeschreibungssprache entwickelt. Um eine möglichst nahtlose Integration zu gewährleisten wurde eine Agentenarchitektur realisiert, die den von den Entwicklungstools generierten Code ausführen kann. Sie integriert einerseits generische Techniken und beruht andererseits auf den Erfahrungen mit anderen Architekturen. Vervollständigt wird die Entwicklungsumgebung durch durch das REkoS-Testbett, das Tools zum Monitoring und Debugging bereitstellt. Das Testbett setzt auf der REkoS-Agentenprogrammiersprache auf und ist funktional in die Architektur eingebettet. Eingefaßt werden beide Teile durch einen einleitenden Abschnitt, der in die Domäne einführt und den Ausblick, wo die Ergebnisse dieser Arbeit zusammengefaßt und einer kritische Bestandsaufnahme unterzogen werden.

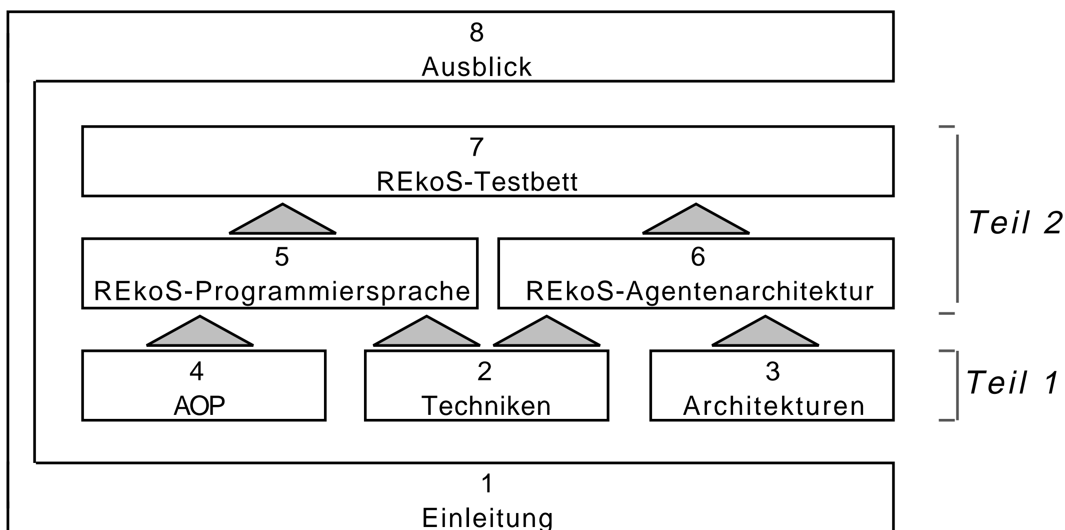

Abbildung 1-2: Struktureller Aufbau der Dissertation

Das nachfolgende 2. Kapitel hat agentenorientierte Techniken zum Thema. Nach einer Auseinandersetzung mit dem Agentenbegriff in Abschnitt 2.1 werden die mit bestimmten Attributen verknüpften Fähigkeiten identifiziert und Techniken zu deren Realisierung vorstellt. Hierbei werden die Themen Kommunikation, Koordination und Handlungsausführung (Abschnitte 2.2 bis 2.4) untersucht. Unter dem letzt-

11. Realisierung von Werkzeugen für die Entwicklung kooperativer Dienste in kommunikationsbasierten Systemen.

genannten Punkt werden auch kognitive Fähigkeiten, die die Wirkungsweise eines Agenten flexibler und intelligenter gestalten, behandelt. Zum Abschluß werden die Ergebnisse aus Sicht der agentenbasierten Diensterbringung betrachtet und bewertet.

Das 3. Kapitel behandelt Agentenarchitekturen, die das Bindeglied zwischen den im 2. Kapitel beschriebenen Techniken und der Agentenprogrammierung bilden (Abschnitt 3.1). Eine Agentenarchitektur integriert verschiedene der vorher vorgestellten Fertigkeiten zu einem einheitlichen Verarbeitungsmodell. Ausgehend von der Erfahrung, daß der universelle, intelligente Agent nicht realisierbar ist und Fähigkeiten und Techniken nicht beliebig kombinierbar sind, wurden viele unterschiedliche Agentenarchitekturen für verschiedene Einsatzzwecke entwickelt. Verschiedene Architekturtypen, die sich in ihrem strukturellem Aufbau oder der Art der Informationsverarbeitung unterscheiden, werden untersucht: Komponentenarchitekturen (Abschnitt 3.2), in denen Funktionsmodule miteinander interagieren, Schichtenarchitekturen mit unterschiedlichen Verarbeitungsebenen (Abschnitt 3.3), strukturarme reaktive Architekturen (Abschnitt 3.4), intelligente kognitive Architekturen (Abschnitt 3.5) und reaktive Architekturen, die dem Belief-Desire-Intention-Verarbeitungsmodell verpflichtet sind (Abschnitt 3.6). Abschließend findet eine Bewertung der untersuchten Architekturtypen im Hinblick auf die Eignung für Dienstagentenszenarien statt.

Programmiersprachen für Agenten sind Gegenstand des 4. Kapitels. Zunächst werden Sprachen untersucht, die sich zum Ziel gesetzt haben, einen Beitrag zum Thema agentenorientierte Softwareentwicklung zu leisten (Abschnitte 4.1 bis 4.4). Hier werden die programmiersprachlichen Konzepte, das dahinterliegende Architekturmodell und die Entwicklungsmethodologie diskutiert. Neben diesen „ganzheitlichen“ Ansätzen existiert eine Reihe von Sprachen, die sich auf Teilaspekte der Agentenprogrammierung konzentrieren; diese werden in Abschnitt 4.5 zusammengefaßt, bevor das für verteilte Systeme kritische Thema des Testens und Debuggings in Abschnitt 4.6 angesprochen wird. Abschnitt 7 setzt sich kritisch mit der Fragestellung auseinander, inwieweit „agentenorientiertes Programmieren“ als neues Programmierparadigma aufgefaßt werden kann. Danach wird im 8. Abschnitt der eigene Ansatz motiviert und kurz skizziert, bevor abschließend die vorgestellten Sprachen miteinander verglichen und die abgeleiteten Erkenntnisse mit dem Dienstkontext in Bezug gesetzt werden.

Das 5. Kapitel beschreibt die REkoS-Programmiermethodologie. Die ersten beiden Abschnitte geben einen Überblick und führen in die REkoS-Begriffswelt ein. In den folgenden vier Abschnitten werden die Beschreibungssprachen und Eingabewerkzeuge für die Belange der Kommunikation, der Kooperation, der prozeduralen Fähigkeiten sowie der steuernden Zielstrukturen vorgestellt. Abschnitt 5.7 „Integration“ behandelt Aspekte der Codegenerierung für die später vorgestellte Architektur beschreibt, wie die Werkzeuge in eine Toolbox zur Synchronisation und Überwa-

chung der einzelnen Arbeitsschritte eingebunden sind. Abschließend erfolgt eine kritische Betrachtung der REkoS-Programmiermethodologie anhand der bei der Realisierung zweier Anwendungen gemachten Erfahrungen.

Im 6. Kapitel werden die besonderen Anforderungen der REkoS-Domäne beschrieben (Abschnitt 6.1) und daraus eine komponentenbasierte Agentenarchitektur entwickelt. Das besondere Augenmerk bei der Konzeptionierung der REkoS-Agentenarchitektur gilt der Entlastung der Softwareentwickler bei der Erstellung von Agentensystemen. Dies wurde durch den Einbau mächtiger generischer Komponenten zur internen flexiblen Steuerung und Verarbeitung bei einer gleichzeitig möglichst klaren und kleinen Programmierschnittstelle berücksichtigt. Der zweite Abschnitt dieses Kapitels gibt einen groben Überblick über die realisierte Architektur bevor in den nachfolgenden Abschnitten 6.3 bis 6.9 die einzelnen Komponenten mit ihren Funktionalitäten und Schnittstellen eingehend beschrieben werden. Am Ende des Kapitels steht ein Resümee, in dem Entwurfsentscheidungen, bereitgestellte und fehlende Funktionalität kritisch beleuchtet werden.

Das 7. Kapitel widmet sich dem in der Agentengemeinde recht stiefmütterlich behandelten Thema des Debuggens von Agentensystemen. Ausgehend aus den Erfahrungen anderer Ansätze (Abschnitt 4.6) wird ein grobes Konzept zur empirischen Untersuchung der Korrektheit von Agenten und Mehragentensystemen entwickelt. Auf Basis dieses methodischen Unterbaus werden eine Reihe von Debuggingwerkzeugen konzipiert und implementiet, mit denen sich die Quellen fehlerhaften Systemverhaltens auf Ebene der Agentenbeschreibungssprache aufspüren lassen. Die Testbettapplikation (Abschnitt 7.3) integriert die unterschiedlichen Werkzeuge für agentenlokales (Abschnitte 7.4 und 7.5) und systemweites Debugging (Abschnitt 7.6) sowie die Laufzeitprotokollgenerierung und -auswertung (Abschnitt 7.7). Weitere Werkzeuge und direkt aus dem Agenten heraus aktivierbare Debug-Level und einfache Konfigurationsmöglichkeiten (Abschnitte 7.2 und 7.8) vervollständigen den Überblick über das REkoS-Testbett. Abschließend erfolgt eine Zusammenfassung und Bewertung des realisierten Ansatzes.

Die Arbeit schließt mit einer Zusammenfassung der erzielten Ergebniss und einer kritischen Erörterung der Stärken und Schwächen der gewählten Ansätze. Weiterhin wird diskutiert, wie andere, bislang nicht berücksichtigte Konzepte zwecks Erweiterung der Funktionalität sinnvoll integriert werden können.

Teil A

Techniken zur Implementierung von Agenten

KAPITEL 2 Fähigkeiten, Methoden und Repräsentation

Am Anfang einer Arbeit über agentenorientierte Softwareentwicklung steht sinnvollerweise der Begriff des Agenten. Dieses Kapitel beschreibt Eigenschaften und Fähigkeiten, die mit dem Begriff von Agenten im Zusammenhang stehen und stellt Methoden zu deren Realisierung vor. Dabei werden die verschiedenen Einflußgebiete der DAI sichtbar, die vor allen Dingen aus der Künstlichen Intelligenz und der Sprechakttheorie sowie auch aus Organisationstheorie, Ökonomie und Entscheidungstheorie stammen.

Zunächst werden die charakteristischen Merkmale von Agenten untersucht. Hieraus lassen sich konkrete Anforderungen an notwendige Fähigkeiten ableiten. Die weiteren Abschnitte dieses Kapitels beschreiben Methoden und Implementierungstechniken zur Realisierung dieser Fähigkeiten. Im zweiten Abschnitt dieses Kapitels werden die Interaktionsformen kommunikationsbasierter Agenten untersucht. Der dritte Abschnitt widmet sich den Aspekten der Koordination, wobei zwischen agentenlokalen und systemweiten Aspekten unterschieden wird. Handeln und Reasoning ist Gegenstand des vierten Abschnittts, bevor im fünften Abschnitt eine kritische Betrachtung und Zusammenfassung folgt.

2.1 Charakteristische Merkmale von Softwareagenten

Zur Beantwortung der Frage, was einen Agenten, insbesondere im Vergleich zu nicht-agentischen Programmen, auszeichnet, sind von verschiedener Seite Definitionsversuche unternommen worden. Einer einheitlichen Sicht auf das Konzept von Agenten steht jedoch die weitläufige Nutzung des Agentenbegriffs im Wege, angefangen von einfachen motorischen Steuerungseinheiten über kommunikationsfähige und Dialoge führende Programme, sensorische in einer Umgebung agierende Wesen bis hin zu selbständig handelnden wissensbasierten, teilweise auch mobilen Systemen[1]. Häufig wird Agent auch nicht als alleinstehender Begriff, sondern zusammen mit Attributen wie intelligent oder autonom verwendet[2].

2.1.1 Notwendige Eigenschaften

Nachstehend folgt eine Aufzählung von typischen Agenteneigenschaften, die üblicherweise zur Unterscheidung nicht-agentischer Systeme herangezogen werden. Hierbei handelt es sich um Eigenschaften, die das „Wesen" eines Software-Agenten ausmachen und die in diesem Sinne auch als *generisch* bezeichnet werden können. Als Referenzen dienen im wesentlichen die Arbeiten von Franklin & Graesser und Wooldridge & Jennings[3].

- **Autonomie:** Motivierend für die Nutzung dieses Begriffs ist die intuitive Vorstellung eines Agenten als ein im Auftrag eines anderen Handelnden. Agenten agieren weitgehend selbständig, d.h. ohne ständige Kontrolle eines menschlichen Auftraggebers. Diese Abkehr vom Paradigma der direkten Manipulation[4] wird häufig durch Hervorhebung der „Kontrolle über die eigenen Aktionen und den internen Zustand" unterstrichen[5].

  Die Kontrollfähigkeit über das eigene Handeln impliziert, daß ein Agent eine Metaebene besitzt, die ihm erlaubt Aktionen zu planen, auszuwählen oder abzubrechen. Hierfür muß ein Agent über mehrere Handlungsalternativen verfügen und zur Einschätzung der Handlungskonsequenzen in der Lage sein. Techniken

1. [Minsky 1985], [Genesereth, Ketchpel 1994], [Coen 1994], [Hayes-Roth 1995].
2. *Intelligent* beispielsweise in [Albayrak, et al. 1996] oder [Crowston, Malone 1988], bzw. *autonom* in [Ferguson 1992a] , [Brustoloni 1991]. Der Name der wichtigsten europäischen Konferenz zum Thema Agenten, die MAAMAW, enthält den Begriff *autonomous* im Namen, während das Standardisierungsgremium FIPA [FIPA 1997] das Attribut *intelligent* nutzt.
3. [Franklin, Graesser 1996], [Wooldridge, Jennings 1995b].
4. [Schneiderman 1983].
5. [Castelfranchi 1995].

zur Stärkung der Autonomieeigenschaft beinhalten Planungsfähigkeiten (Kapitel 2.4.4), Koordinierungstechniken (Kapitel 2.3) und auch Kommunikationsfähigkeit zum Beschaffen unbekannter Informationen (Kapitel 2.2).

- **Intelligenz:** Dieses Attribut wird im Zusammenhang mit Agenten zumeist wahllos benutzt ohne darauf einzugehen, wo die Intelligenz lokalisiert ist oder wie sie in Erscheinung tritt[6]. In der DAI existieren drei Ansätze zur Erlangung intelligenten Verhaltens: die Sichtweise der KI durch Verwendung kognitiver Methoden, Intelligenz als Resultat vieler einfacher interagierender Agenten (*emergent intelligence*) und Intelligenz ohne Repräsentation in physikalischen Robotern.

  An KI-Techniten kommen in Agenten am häufigsten Fähigkeiten des Lernens bzw. der Adaption und des Planens (Kapitel 2.4.4 und Kapitel 2.4.5) zum Einsatz. Intelligentes Verhalten durch wohlkoordinierte Aktivitäten wird bei Software-Agenten durch unterschiedliche Mechanismen erreicht; diese werden in Kapitel 2.3 behandelt. Schließlich ist intelligentes Verhalten auch mit reaktiven, strukturarmen Agentenarchitekturen (siehe Kapitel 3.4) zu erzielen.

- **Interaktivität:** Agenten, die in einem Verbund mit anderen Agenten oder mit Menschen arbeiten, benötigen Mechanismen zur Interaktion mit ihrer Umgebung[7]. In der Informatik wurde der Kommunikationsbegriff bislang entweder mit Datenaustausch oder Funktionsaufrufen nach dem Client-Server-Prinzip assoziiert. Anders verhält es sich mit der Agentenkommunikation, in der die *Informations*übermittlung, zumeist auf einer linguistischen Ebene, im Zentrum des Interesses steht.

  Üblicherweise wird die Sprechakttheorie als Grundlage herangezogen, auf die Dialoge und Kooperationsprotokolle aufsetzen. Kapitel 2.2 behandelt die unterschiedlichen Kommunikationsmodelle, die in Agentensystemen zum Einsatz kommen.

- **Reaktivität:** Aus der Dynamik der Umgebung, hervorgerufen durch Aktionen anderer Agenten oder sonstiger Umwelteinflüsse, ergibt sich als Anforderung an Agenten, auf Ereignisse unmittelbar reagieren zu können.

  Reaktivität kann durch ein Aktivitätenmanagement erreicht werden, das in der Architektur (Kapitel 3) verankert ist. Hier ist festgelegt, wie Informationen empfangen und verarbeitet werden. Daneben unterstützen prozedural realisierte Fähigkeiten (Kapitel 2.4.1) die Reaktivität, weil diese im Gegensatz zu deklarativer Wissensverarbeitung ohne Wissensrepräsentation und der damit verknüpften Wartung eines Weltmodells auskommen.

6. Beispielsweise bei [Wooldridge, Jennings 1995b].

7. [Genesereth, Ketchpel 1994] führen die Fähigkeit eine Agentenkommunikationssprache zu beherrschen als *die* zentrale Eigenschaft von Agenten an.

- **Intentionalität und zielgerichtetes Verhalten:** Agenten werden auch als intentionale Systeme verstanden und unter Zuhilfenahme „mentaler“ Begriffe wie Glauben, Wünsche oder Ziele modelliert. Die Zuschreibung mentaler Attribute ist nach Dennet[8] zur Erklärung des Verhaltens komplexer artifizieller Systeme, wie sie Agenten zumeist darstellen, angemessen. Der wesentliche Unterschied zu anderen Systmen liegt darin, daß Agenten intentionale Begriffe *explizit* darstellen und nutzen[9].

  Aspekte der Intentionalität bei der Programmierung von Agenten werden in Kapitel 2.4.2 sowie bei den BDI-Agentenarchitekturen (Kapitel 3.6) behandelt.
- **Rationalität und optimales Handeln:** Zielgerichtetes Handeln wird häufig mit dem Rationalitätsbegriff verknüpft, der auf Nutzenoptimierung abzielt. Agenten sind nach Simon aufgrund von Ressourcenknappheit und unvollständiger Information begrenzt rationale Systeme[10].

  Neben dem schon erwähnten zielgerichteten Verhalten sind kommen andere Methoden der Nutzenoptimierung wie Spieltheorie (Kapitel 2.3.2), Entscheidungstheorie und Heuristiken zur Aktionsauswahl zum Einsatz. Die Umsetzung rationalen Verhaltens wird in Kapitel 2.4.4 behandelt.

2.1.2 Optionale Eigenschaften

Während die oben beschriebenen Eigenschaften von Agenten als grundlegend angesehen werden, existieren häufig weitere Merkmale, die sich aus bestimmten Aspekten des Einsatzgebietes ergeben.

- **Soziale Attitüden:** Kalenka und Jennings prägten den Begriff *social level*[11], auf dem soziale Normen, die das Handeln eines Agenten bestimmen, modelliert sind. Diese Verhaltensebene ist über dem knowledge level[12] angesiedelt und kann somit als Unterscheidungsmerkmal von Agenten zu anderer wissensbasierter Software dienen. Es werden Attribute wie Persönlichkeit, Charakter,

---

8. [Dennet 1987] führt in diesem Zusammenhang den *intentional stance* ein. Mentale Zustände stehen bei Shoham [Shoham 1993], einer der Begründer des Gebiets der agentenorientierten Programmierung, im Mittelpunkt: "An agent is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments."

9. [Wagner 1997].

10. „Bounded rationality“ [Simon 1957]. Vergleiche dazu den „maximum rationality hypothesis“-Ansatz von [Newell 1982], der eine direkte Verbindung zwischen Zielen, Wissen und nachfolgenden Aktionen herstellt und die Optimalitätsannahme in [Anderson 1991].

11. [Kalenka, Jennings 1997], [Jennings, Campos 1997].

12. [Newell 1982].

Glaubwürdigkeit oder emotionaler Zustand in Verbindung mit Agenten benutzt. Aus der Sicht eines kommunizierenden informationsverarbeitenden Systems sind „Wahrheitsliebe“ und „Wohlwollenheit“ gewünschte Eigenschaften. Was die Verpflichtungspolitik anderen Agenten gegenüber angeht, werden Begriffe wie „kooperativ“, "hilfsbereit", „selbstverliebt“ oder „egoistisch“ benutzt. Zumeist dienen diese Antropomorphismen allerdings dazu, dem Verhalten eines Agenten bei dessen Beobachtung menschliche Züge zuzubilligen, als daß sie Konzepte mit darunterliegenden Implementierungsmethoden darstellen würden. Erste Ansätze zur Modellierung derartiger Attribute bieten modale Logiken, wie sie in Kapitel 2.4.2 vorgestellt werden.

- **Realzeitfähigkeit:** In einigen Anwendungsbereichen besteht die Notwendigkeit der Zusicherung von Antwortzeiten innerhalb eines Toleranzrahmens. Zwar ist die Performance eines Agenten gewöhnlich schlechter als die eines auf ein Problem zugeschnitten Programms. Der Grund dafür liegt in der höheren Komplexität der einzelnen Softwarekomponenten und der unter dem Punkt „Autonomie“ aufgeführten Meta-Reasoning-Ebene, die durch die Einführung einer neuen Interpretationsschicht in starkem Maße Prozessorressourcen verschlingt. Aber gerade die Fähigkeit zur Introspektion bietet auch Möglichkeiten für die Verbesserung des Realzeitverhaltens: Beim Vorhandensein mehrerer Handlungsalternativen kann durch Aktionsselektionsregeln diejenige gewählt werden, die in der aktuellen Situation die beste Performance verspricht. Auch kann ein Agent, der mit einer komplexen Aufgabe beschäftigt ist, Teile dieser Arbeit auf andere Agenten übertragen (Kapitel 2.3.4). Einen anderen Ansatz beschreiten Realzeitalgorithmen, die in in Kapitel 2.4.1 vorgestellt werden.
- **Mobilität:** Unter Mobilität wird die Fähigkeit eines Agenten verstanden, über ein Netzwerk zu einem anderen Rechner zu migrieren und dort mit der Abarbeitung fortzufahren. Derartiges remote processing ist sinnvoll, wenn der ferne Rechner leistungsfähiger ist als der lokale, und auf diese Weise Rechenzeit gespart werden kann. Im Bereich des distributed processing und im Netzwerkmanagement wird Codemobilität zum Zweck des load balancing genutzt. Mobile Agenten bieten mit ihrer Vor-Ort-Verarbeitung Offlinefähigkeit und können Kommunikationskosten einsparen. Für den Kontext dieser Arbeit ist die Mobilitätseigenschaft wegen der Konzentration auf schwergewichtige Agenten jedoch uninteressant.

2.1.3 Klassifizierung von Agenten

Die im vorigen Abschnitt beschriebenen Eigenschaften sind für sich genommen zu schwach, um zur Unterscheidung von anderer, nicht-agentischer Software zu dienen. Beispielsweise trifft die Autonomieeigenschaft auch auf einen Hintergrundprozeß zur Verwaltung der Auftragsschlange eines Druckers zu, oder es lassen sich selbst Thermostaten intentional beschreiben[13]. Auch sind die meisten Begriffe zu schwammig, was sich als problematisch für eine Umsetzung im Rahmen des De-

signs und der Programmierung erweist. Gilbert erkennt bereits in der Ausführung von Skripten zielorientiertes Handeln[14], was den Agentenbegriff sehr weit faßt. Nach Simon[15] ist die Annahme, alle Ziele eines Agenten zu kennen, illusorisch. Ziele müssen nicht einmal explizit repräsentiert sein. Beispielsweise kann hinter einer „zielstrebigen" Suche ein einfacher Backtrackingalgorithmus stehen.

So legen die unterschiedlichen Sichtweisen auf das Konzept von Agenten die Erarbeitung einer Klassifikation nahe. Ziel ist es, durch Bündelung von Eigenschaften oder Anwendungsmerkmalen zu einer stärkeren Konkretisierung zu gelangen. Ansätze hierzu finden sich bei Brustoloni sowie Franklin & Graesser[16]. Die untenstehende Tabelle definiert Agententypen anhand hervorstechender Merkmale und setzt diese in Bezug zu Anwendungsbereichen. Die Typen basieren weitgehend auf einem Vorschlag von Nwana[17]..

TABELLE 1. Klassifikation von Agenten

Agententyp	Eigenschaften	Einsatzgebiete
kognitive Agenten	kognitive Fähigkeiten	Probleme lösen, Psychologie
persönliche Assistenten, Interfaceagenten	Adaptivität, Benutzerschnittstelle	Sekretariatsdienste, Informationsdienste
mobile Agenten	flexibel, klein	keine spezifischen Aufgaben
kooperative Agenten	Kommunikation und Kooperation	CSCW, Workflow Management
konkurrierende Agenten	Rationalität, Kommunikation	electronic commerce
reaktive Agenten	Realzeitfähigkeit	Prozeßsteuerung

- **Kognitive Agenten** verdienen am ehesten das Attribut „intelligent". Sie sind Problemlöser und zusätzlich mit Planungsfähigkeiten oder Lernmodulen ausge-

13. [McCarthy, Hayes 1969].

14. [Gilbert 1997].

15. [Simon 1991].

16. Beispielsweise schlägt [Brustoloni 1991] eine Dreiteilung in regulierende, planende und adaptive Agenten vor, während [Franklin, Graesser 1996] eine offene Taxonomie mit Viren, aufgabenspezifischen und unterhaltenden Agenten auf der obersten Unterscheidungsebene wählen.

stattet. Die Grundlagen kognitiver Agenten und gleichnamiger Architekturen liegen in der Künstlichen Intelligenz. Inferenzmechanismen und Wissensrepräsentationsformalismen sind in Expertensystemschalen integriert; die Programmierung reduziert sich somit weitgehend auf die Wissensmodellierung. Kognitive Agenten sind monolithische Einagentensysteme, bei denen Reaktivität und die Interaktion mit anderen Agenten nur eine geringe Rolle spielen.

- **Persönliche Assistenten** erleichtern tägliche Routinetätigkeiten. Sie filtern E-Mails, durchsuchen das World-Wide-Web nach interessanten Informationen und präsentieren diese. Ein solcher Agent sollte mit den Anforderungen seines Besitzers mitwachsen. Deshalb ist Lernfähigkeit eine wichtige Eigenschaft. Häufig sind persönliche Assistenten auch nur intelligente Schnittstellen zu proprietärer Software, d.h. sie „wrappen“ ein Programm mit dem Ziel besserer Bedienbarkeit. In diesen Fällen werden auch sie auch als Interface-Agenten bezeichnet, wodurch die Gewichtung der Nutzerschnittstelle zum Ausdruck kommt. Typischerweise sind auch persönliche Assistenten Einagentensysteme, so daß Agentenkommunikation nicht von Belang ist. Sie werden ohne eine Agentenarchitektur oder Agentenprogrammiersprache entwickelt und sind damit für den Bereich der agentenorientierten Programmierung uninteressant.
- **Mobile Agenten** stellen eher eine Technik als einen anwendungsorientierten Agententyp dar. Sie benötigen eine Laufzeitumgebung, die den Migrationsprozeß unterstützt, Plattformunabhängigkeit gewährleistet und Sicherheitsmechanismen sowohl für den Host wie auch die migrierenden Agenten bietet. Weil heutzutage Kommunikation im Vergleich zu Rechnerleistung sowohl teurer als auch langsamer ist, müssen mobile Agenten von möglichst geringer Codekomplexität sein. Andernfalls sind kommunikationsbasierte Lösungen vorzuziehen, die darüberhinaus auch weniger Sicherheitsprobleme mit sich bringen.
- **Kooperative Agenten** erarbeiten komplexe Aufgabenstellungen gemeinsam in einem Verbund. Kommunikation, Kooperation und Koordination zwischen mehreren Agenten stehen dabei im Vordergrund. Auf diese Weise lassen sich Synergien erzielen, wenn die beschränkten bereitgestellten Fähigkeiten der einzelnen Agenten zu komplexen value-added Services aggregiert werden.
- **Konkurrierende Agenten** kooperieren nur zum Zweck der eigenen Gewinnmaximierung. Anders als bei kooperativen Szenarien können hier die Absichten und Ziele einzelner Agenten divergieren, wie beispielsweise in Käufer-Verkäufer-Situationen. Ein globales Systemziel ist nicht vorhanden, kann aber häufig indirekt in Form ökonomischer Größen wie Umsatzmaximierung oder Pareto-

17. [Nwana 1996]. Dort werden zusätzlich Informations- bzw. Internetagenten sowie hybride und smarte Agenten genannt. Aber auch andere Unterteilungen sind möglich. Beispielsweise schlagen [Wooldridge, Jennings 1995b] eine Unterscheidung reaktiver, deliberativer und hybrider Agenten vor, [Wagner 1997] klassifiziert anhand der in den Agenten verwendeten Repräsentationsformen *knowledge*, *perception*, *task* und *intention*.

Optimalität ausgedrückt werden. Schwerpunkt beim Design von Agenten für konkurrierende Situationen liegt in der Ausgestaltung der Rationalität.

- **Reaktive Agenten** kommen in zeitkritischen Anwendungen zum Einsatz. Statt komplexer kognitiver Fähigkeiten zeichnet diese Agenten eine schnelle Reaktionsfähigkeit aus. Fest verdrahtete Verhaltensmuster und schlanke, strukturarme Architekturen sind typische Merkmale dieser Art von Agenten. Entsprechend schwach sind Intentionalität und Intelligenz ausgeprägt, Kommunikation findet häufig datenorientiert statt durch Sprechakte statt.

2.1.4 Distributed problem solving und multi-agent systems

Betrachtet man Systeme, die aus mehreren kollaborativ arbeitenden Agenten bestehen, so ergeben sich zwei grundsätzlich unterschiedliche Sicht- und Herangehensweisen. Erfolgt die Modellierung problemzentriert top-down[18], beginnend mit einer Zerlegung des Problems in Subprobleme und nachfolgender Identifikation und Implementierung von Spezialagenten für die Teilaufgaben, sprechen wir von einem DPS[19]-System. Es existiert ein zentrales Systemdesign, und das System ist insoweit relativ geschlossen, als daß die teilnehmenden Agenten a priori bekannt sind. Sie verhalten sich im Sinne der zu erzielenden Problemlösung kooperativ und benötigen, da sie Teil eines Gesamtkonzepts sind, nur eingeschränkte Autonomie. In DPS-Systemen ist der Untersuchungsgegenstand der Rationalität oder des Verfolgens von Zielen von eher nebensächlichem Rang, zumeist sind diese Aspekte implizit in das Systemdesign eingebaut und nicht Gegenstand der Agentenmodellierung. DPS-Systeme werden üblicherweise zur Realisierung verteilter Problemstellungen verwendet, wobei die Verteilung räumlicher, funktionaler oder zeitlicher Natur sein kann.

Im Gegensatz zum reduktionistischen Ansatz des DPS ist die Vorgehensweise im Bereich *multi-agent systems* (MAS) konstruktivistisch: Hier steht die Modellierung einzelner Agenten im Vordergrund. Weder ist ein globales Ziel vorgegeben noch ein Problem zu lösen, anstelle dessen verfolgen die Agenten eigene, persönliche Ziele. Während DPS-Systeme von ihrer Konzeption her grundsätzlich kooperativ sind, zeichnen sich MAS eher durch Eigeninteresse der Agenten oder sogar Konkurrenzsituationen aus. Aufgrund der Fokussierung auf die Modellierung einzelner Agenten eignet sich dieser Ansatz trefflich für offene Domänen[20]. MAS-Agenten benötigen ein höheres Maß an Autonomie, Flexibilität und Intelligenz, da sie in unbe-

18. Wooldridge und Jennings nennen dies reduktionistisch [Wooldridge, Jennings 1995b].

19. Distributed Problem Solving. Nach [Davis, Smith 1983] besteht das verteilte Problemlösen aus den 4 Phasen *Problemzerlegung*, *Problemverteilung*, *Abarbeitung der Teilprobleme* und *Lösungssynthese*.

20. [Hewitt 1991].

kannten Umgebungen wirken. Das „Funktionieren“ eines Mehragentensystems, das aus „selbst interessierten“[21] Agenten besteht, beruht oft auf Synergieeffekten, die durch dynamische Koordination und Kooperation entstehen können.

Stand der Technik sind DPS-Lösungen, die entweder auf etablierten Techniken wie der Blackboard-Kommunikation beruhen[22] oder mit Hilfe sogenannter agentenorientierter Designmethoden im Geiste objektorientierter Softwareentwicklung erstellt werden[23]. Im Bereich MAS hingegen sind viele Aspekte noch Forschungsgegenstand und werden erst in zukünftigen Anwendungen zum Einsatz kommen. Hier sind vor allen Dingen elektronische Märkte zu nennen, auf denen Agenten als Stellvertreter mit unterschiedlichen Zielen (als Anbieter und Nachfrager) handelseinig werden sollen. Erste prototypische Systeme existieren[24].

2.1.5 Eigenschaften von Multiagentensystemen

In Szenarien mit mehreren autonomen Agenten treten Merkmale verteilter Systeme auf, die sich im Verhalten eines isoliert arbeitenden Agenten nicht zeigen. Dennoch haben diese Systemeigenschaften Einfluß auf die Gestaltung der einzelnen Agenten.

- **Komplexe und dynamische Umgebung:** Multiagentensysteme sind komplex und zeichnen sich durch nichtdeterministisches Verhalten aus. Dynamik entsteht über längere Sicht durch das System verlassende oder neu das System betretende Agenten. Im Unterschied zu monolithischen Systemen sind die Informationen in einem Multiagentensystem typischerweise intransparent über die einzelnen Agenten verteilt. Jeder Agent kennt nur einen für ihn interessanten Ausschnitt der Welt, der nicht notwendigerweise exakt und korrekt sein muß. Durch das gleichzeitige Wirken mehrerer autonomer Einheiten ist Wissen über den Zustand der Umgebung generell temporärer Natur.

  Aus diesen Betrachtungen ergeben sich Konsequenzen für das Design einzelner Agenten: Die Informationsfülle und deren Dynamik machen es notwendig, daß jeder Agent unter der Prämisse unvollständigen, möglicherweise inkorrekten

21. Freie Übersetzung des Terminus *self interested*, der üblicherweise als Gegensatz von altruistisch benutzt wird.

22. Frühe DPS-Lösungen, die auf dem Blackboardmodell (siehe Abschnitt 2.2) beruhen, sind HEARSAY [Erman, Lesser 1975] und das Distributed Vehicle Monitoring Testbed [Lesser, Corkill 1983].

23. Hierunter fallen Anwendungen aus den Bereichen Luftverkehrsmanagement [Ljungberg, Lucas 1992] oder Netzwerkkontrolle [Rao, Georgeff 1990].

24. Zum Beispiel KASBAH als automatisierter Handelsplatz [Chavez, Maes 1996]. Ein anderes Beispiel stellt die Verteilung von Transportaufträgen zwischen konkurrierenden Speditionen dar [Fischer, et al. 1995].

Wissens arbeiten kann. Mechanismen zum Umgang mit temporalem und nichtmonotonem Wissen bieten Logikerweiterungen (Abschnitt 2.4.2 ff) und Informationsfilterungs- und -speicherungstechniken (Abschnitt 2.4.5). Der Umgang mit dynamischem Wissen erzwingt ein gewisses Maß an Reaktivität, das entweder in der Architektur (siehe Abschnitte 3.1, 3.4 und 3.6) verankert ist oder sich in Fähigkeiten zum Umplanen (Abschnitt 2.4.4) ausdrücken kann.

- **Synergieeffekte durch Kooperation und Koordination:** Wenn Agenten ihre Aktionen mit denen anderer Agenten abstimmen, kann die Gefahr des Auftretens unerwünschter Situationen wie Deadlocks oder Ressourcenengpässe vermindert werden. Wohlkoordiniertes Handeln kann sogar zu Performance-Verbesserungen führen, wenn redundante Handlungen nicht mehrfach durchgeführt werden oder zeitkritische Aufgaben zerlegt und von mehreren Agenten bearbeitet werden. Auch Synergieeffekte durch Kombination einfacher Basisfunktionalitäten zu neuartigen, sog. "value-added services" sind möglich.

  Diese Potentiale erschließen sich durch Rollenverhalten, Wissensaustausch oder durch längerfristige Kooperationen. Häufig wird nicht einmal domänenspezifisches Wissen benötigt, so daß Koordinierungs- und Kooperationsmechanismen als generische Bestandteile in Agenten eingepflanzt werden können. In den Unterabschnitten zu Abschnitt 2.3 werden verschiedene Herangehensweisen zur Verbesserung der Systemkohärenz vorgestellt.

- **Flexibilität durch Offenheit:** Agentensysteme können bewußt redundant ausgelegt sein, um die Ausfallsicherheit zu erhöhen. Wenn ein System derart auch noch bei Ausfall einzelner Komponenten funktioniert, spricht man von „graceful degradation“. Üblicherweise wird Redundanz benutzt, um Performanceprobleme zu lösen oder kritische Flaschenhälse zu „verbreitern“. Ein anderer Aspekt der Flexibilität liegt in der Offenheit von Agentensystemen: Agenten sind eigenständige Prozesse, die über eine einheitliche Schnittstelle, die Agentenkommunikationssprache, verfügen. Dadurch können Agenten zur Laufzeit dynamisch ersetzt und hinzugefügt werden, ohne das Gesamtsystem zum Stillstand zu bringen.

2.2 Kommunikation

In der Informatik existieren viele etablierte Kommunikationstechniken für unterschiedliche Einsatzzwecke. Protokolle sorgen für eine verlustfreie Übermittlung; Datenaustausch erfolgt entweder verbindungsorientiert oder verbindungslos, synchron oder asynchron, als Datenstrom oder in Pakete geschnürt und eventuell noch komprimiert und verschlüsselt. In verteilten Systemen sorgt Kommunikations-Middleware wie CORBA oder DCE[25] mit Brokering-Diensten für eine einheitliche Sicht auf die in einem Netzwerk verteilten Programmelemente und deren einfache Nutzung.

Agentenkommunikationssprachen bringen als Neuerung eine linguistische Ebene ein. Die Relevanz einer derartigen, speziell auf Agenten zugeschnittenen Sprache wird durch ein Zitat von Genesereth verdeutlicht, der Softwareagenten als

> *„software components that communicate with their peers by exchanging messages in an expressive agent communication language“*

bezeichnet[26]. Beim Design einer Agentenkommunikationssprache sind drei Aspekte von besonderer Relevanz:

- **Inhalt und Bedeutung:** Agentenkommunikationssprachen sind ausdruckskräftig, sie ermöglichen einen Informationsaustausch auf der Wissensebene. Auch übermitteln Agenten nicht einfach nur Daten, deren Interpretation dem Empfänger überlassen ist; vielmehr spielt auch die Bedeutungsebene eine Rolle: Agenten *erbitten* die Erbringung von Diensten, *wünschen* die Erreichung eines Zieles, *erfragen* die Pläne anderer Agenten oder *schlagen* Wege zur Behebung von Konflikten *vor.*
- **Universalität:** Anwendungen bestehen häufig aus Agenten, die von unterschiedlichen Entwicklern stammen. Diese Heterogenität macht eine einheitliche Kommunikationssprache wünschenswert, die unabhängig von der konkreten Impementierung der Agenten ist. Erst diese Allgemeingültigkeit führt zu offenen Systemen, in denen Agenten ausgetauscht, hinzugefügt oder entfernt werden können.
- **Dialoge:** Im Unterschied zu „klassischen“ Kommunikationsanwendungen funktioniert die Agentenkommunikation nicht nach dem Client-Server-Prinzip, sondern findet zwischen gleichberechtigten Partnern statt. Agenten sollten in der Lage sein zielgerichtet Dialoge zu führen, um Konfliktsituationen zu bereinigen oder gemeinsam komplexe Aufgaben zu erbringen.

In frühen Anwendungen der DAI kamen vorwiegend *Blackboards* als Kommunikationsform zum Einsatz[27]. Sie stellen eine abstrakte Form des *shared memory* dar. Agenten kommunizieren miteinander, indem sie Daten auf das Blackboard schreiben bzw. von ihm lesen. Ein weiteres Merkmal liegt in der indirekten Kommunikation; für das direkte Adressieren einer Nachricht an einen bestimmten Agenten bietet das Blackboard keine Unterstützung. Mit der zunehmenden Realisierung echt verteilter Systeme gerieten Blackboards zugunsten direkter, sprechaktbasierter Kommunikationsformen zunehmend in den Hintergrund[28].

25. Common Object Request Broker Architecture [Yang, Duddy 1996], bzw. Distributed Computing Environment [Brando 1995].

26. [Genesereth, Ketchpel 1994].

27. [Carver, Lesser 1992]. Beispiele stellen die DPS-Agentensysteme HEARSAY-II [Nii 1986] und DVMT [Lesser, Corkill 1983] dar.

2.2.1 Sprechakte

Im Bereich der DAI kommen überwiegend an die menschliche Kommunikation angelehnte, sprechaktbasierte Sprachen zum Einsatz. Wesentliches Element eines Sprech*akt*s ist neben der übermittelten Information der Aspekt des Handelns. Austin[29] identifizierte drei unterschiedliche Aspekte der Kommunikation: Der lokutionäre Akt beschreibt den Sprechakt im Hinblick auf Artikulation, Konstruktion und Logik, die illokutionäre Ebene drückt eine beabsichtigte Wirkung und damit eine kommunikative Funktion aus, während der perlokutionäre Aspekt auf Konsequenzen beim Empfänger fokussiert. Für Agenten spielt ausschließlich die illokutionäre Ebene eine Rolle. Lokutionäre Aspekte werden entweder durch die Übertragung mittels des Kommunikationsmediums abgedeckt und sind daher uninteressant, oder können - wie beispielsweise Betonung oder Gestik - prinzipiell nicht übermittelt werden. Auf der anderen Seite liegt es außerhalb der Kontrolle eines Agenten, die beabsichtigte Wirkung bei einem anderen Agenten sicherzustellen.

Gegenwärtig dominieren zwei Vertreter der sprechaktbasierten Sprachen den Bereich der Agentenkommunikation: KQML und FIPA-ACL[30]. Beide Sprachen fußen auf einer Reihe illokutionärer Performatives und trennen Syntax, Nachrichteninhalt und Bedeutung. Für die Darstellung der eigentlichen Botschaft kommen logikbasierte Wissensrepräsentationssprachen[31] zum Einsatz, die wiederum ein in einer sogenannten Ontologie beschriebenes Vokabular nutzen.

```
(tell
    :content price( bid( 150 ))
    :language Prolog
    :ontology markets
    :in-reply-to
    :force
    :sender buyer
    :receiver auctioneer))
```

Die Sprachbeschreibung von KQML enthält 43 Performatives, die sämtlich den Kategorien assertives und directives[32] zuzurechnen sind. Demgegenüber kommt ACL

28. Eine Ausnahme stellt die „open agent architecture“ [Cohen, et al. 1994] dar, die auf hierarchisch verteilten Blackboards beruht.

29. [Austin 1962].

30. Knowledge Query and Manipulation Language, [Finin, et al. 1994a], [Finin, et al. 1994b], bzw. Agent Communication Language der Foundation for Intelligent Physical Agents, [FIPA 1997].

31. KIF (Knowledge Interchange Format) [Genesereth, et al. 1991] im Falle von KQML und SL [FIPA 1997] bei FIPA. Hierbei handelt es jeweils um Vorschläge; die Sprechakte besitzen jeweils einen Slot, in dem die Content-Sprache anzugeben ist.

mit 20 Nachrichtentypen aus, die aus 5 grundlegenden Kommunikationsakten[33] aufgebaut werden. Der wichtigste Unterschied zwischen beiden Sprachen liegt in der semantischen Fundierung. Während für KQML-Sprechakte nur prosaische Annotierungen existieren, werden Bedeutung und Anwendbarkeit von FIPA-Nachrichten für die Senderseite modallogisch mittels der mentalen Operatoren belief, uncertainty und intention beschrieben.

Den stärksten semantischen Unterbau besitzt die planbasierte Kommunikation[34]. Sie macht im Gegensatz zu den vorher beschriebenen Ansätzen auch Annahmen über die erzielte Wirkung beim Empfänger, berücksichtigt also zusätzlich zur illokutionären Ebene perloktionäre Aspekte.

```
Vorbedingung:
     cando( belief( speaker, fact )),
     want(  belief( speaker, belief(hearer, fact))).
Aktion:
     INFORM(  speaker, hearer, fact ).
Nachbedingung:
     belief(  hearer,   belief( speaker, fact )),
     belief(  speaker,  belief( hearer, belief(
              speaker, fact ))).
```

Der Name dieser Kommunikationsform ist dem klassischen Planungsmodell der KI entlehnt[35]. Dort repräsentieren Planoperatoren elementare Aktionen, die mit Vor- und Nachbedingungen umschrieben werden. Analog dazu beinhalten auch planbasierte Sprechakte eine Anwendbarkeitsbedingung sowie eine Beschreibung der Effekte bei den Kommunikationspartnern. Die Vorbedingung drückt einen Zustand beim Sprecher aus, die Nachbedingungen beschreiben Konsequenzen für Sprecher und Empfänger. Dadurch können beide Parteien sich ein Bild vom mentalen Zustand des jeweiligen Kommunikationspartners machen.

32. Die Kategorisierung von Sprechakten geht auf [Searle 1969] zurück. Searle identifizierte 5 Basiskategorien illokutionärer Akte: assertives, directives, commisives, declaratives und expressives.

33. Hierbei handelt es sich um inform, request, query, confirm, disconfirm.

34. Zu den planbasierten Kommunikationsmodellen gehören die Arbeiten von [Cohen, Perrault 1979], [Cohen, Levesque 1990] und [Allen, Perrault 1980] sowie der „Situated Conversation Theory“-Ansatz von [Numaoka, Tokoro 1990].

35. Siehe Abschnitt 2.4.4.

2.2.2 Interaktionsprotokolle

Mit den oben behandelten Sprechakten können Informationen ausgetauscht und einfache Frage-Antwort-Dialoge durchgeführt werden. Damit Agenten auch zu längerfristigen Interaktionen wie etwa Verhandlungen und Kooperationen in der Lage sind, benötigt man Interaktionsprotokolle. Ein solches Protokoll regelt den Ablauf einer Konversation. Es legt für alle Beteiligten fest, wann sie welche Sprechakte austauschen und welche Inhalte übermittelt werden. In diesem Sinne stellen Interaktionsprotokolle eine Verbindung von Kommunikation und Handeln dar, wozu die Sprechakttheorie allein nicht ausdrucksmächtig genug ist:

> *„...dialogue has no syntax, speech act types are not the relevant categories over which to define the regularities of conversation; there exists no other finite alphabet over which to define the regularities; ...“*[36]

Zustandsgraphen stellen eine mögliche Spezifikationssprache für Interaktionsprotokollen dar. Sie beschreiben die möglichen Interaktionspfade einer Konversation zweier Agenten. Jeder Knoten repräsentiert den Zustand eines Agenten im Protokoll, der durch Senden einer Nachricht den Zustand des anderen, empfangenden Agenten ändert[37]. Derartige Graphen bieten zwar eine Sicht auf die stattfindende Kommunikation, die Handlungen, die ein Agent vornimmt werden jedoch nicht dargestellt. Um an einem Interaktionsprotokoll teilzunehmen, benötigt ein Agent eine Repräsentation zumindest seines Teils des Zustandsgraphen

2.2.3 Zusammenfassung

Kommunikation ist bei Softwareagenten weitgehend gleichbedeutend mit der Fähigkeit zum Senden und Empfangen von Sprechakten. Zwar benötigt ein Agent häufig auch „klassische“ Kommunikationsformen, um beispielsweise mit konventioneller Software zu interagieren, diese Sprachen und Protokolle sind jedoch nicht Gegenstand der Agentenkommunikation. Die Haupttriebkraft bei der Gestaltung von Agentenkommunikationssprachen liegt dabei in der Anforderung, Agentensysteme offen zu gestalten.

Von verschiedener Seite wurde Kritik am Sprechaktmodell geübt. Auf eine fehlende Semantik wurde von Werner und Singh[38] hingewiesen und Lösungen diesbezüglich angeboten. Auch bereitet Searles Kategorisierung Probleme: Viele Nach-

36. [Levinson 1981].

37. Ein Beispiel eines Interaktionsgraphen findet sich in Kapitel 4 (siehe Abbildung 4-5 auf Seite 99).

38. [Werner 1987], [Singh 1991].

richtentypen, die in der Agentenkommunikation benutzt werden, sind nicht *eindeutig* in die Klassifikation einzuordnen[39]. Die Hauptschwäche von KQML ist deren fehlende deklarative Semantik, die einen zu großen Interpretationsspielraum zuläßt[40]. Diese Semantik wird von der FIPA-ACL vorgegeben, basiert jedoch auf Annahmen über die Wissensrepräsentation der Agenten.

Für die Modellierung komplexerer Konversationen finden Interaktionsprotokollsprachen Verwendung. Sie legen das kommunikative Verhalten aller an einer Konversation beteiligten Agenten fest. Mithin können die Zielgerichtetheit und Effizienz der Interaktionen per Design sichergestellt bzw. durch die explizite Bezugnahme auf ein Protokoll die Gefahr abweichenden Verhaltens durch „paranoide“ Agenten vermindert werden. Jedoch ist es schwierig Protokolle zwischen mehr als zwei Parteien zu modellieren. Auch bereitet die Verknüpfung von Kommunikation und Aktion Probleme; zumeist beschränken sich die Protokollsprachen auf die auszutauschenden Nachrichten.

Für einige Typen von Agenten spielt die Sprechaktkommunikation keine oder nur eine untergeordnete Rolle. Dazu gehören die kognitiven Agenten, bei denen die Informationsverarbeitung und -gewinnung im Zentrum des Interesses steht (siehe Kapitel 3.5). Bei persönlichen Assistenten und Interfaceagenten stehen Schnittstellen zum Benutzer bzw. zu anderer Software im Vordergrund; Inter-Agentenkommunikation findet nicht statt.

2.3 Koordination

Nach Malone[41] kann Koordination als Prozeß verstanden werden, Abhängigkeiten zwischen Aktivitäten zu managen. In der DAI gibt es zwei Sichten auf Koordinierungsproblematiken: eine lokale Sicht, bei der es um das Aktivitätenmanagement innerhalb eines Agenten geht und eine Systemsicht, die auf Abstimmung von Handlungen zwischen mehreren Agenten abzielt.

Ein lokales Koordinationsproblem liegt vor, wenn

- ein Agent zwischen alternativen Aktionen wählen kann, die in unterschiedlicher Weise Einfluß auf die Umgebung haben oder

39. Dazu gehören command, query, demand, announce, accept, refuse und propose. [Ballmer, Brennenstuhl 1981] kritisieren die Willkürlichkeit von Searles Klassifizierung beim Versuch, sämtliche englische Verben in die 5 Kategorien einzuordnen.

40. [Cohen, Levesque 1995].

41. [Malone, Crowston 1991].

- die Reihenfolge und Zeit der Ausführung von Aktionen den Agenten und seine Umgebung unterschiedlich beeinflussen.

Entsprechend beschäftigen sich lokale Koordinierungstechniken mit der Auswahl geeigneter Aktionen und deren zeitliche Anordnung. In diesem Sinne findet die Koordinierung vor der Ausführung statt. Vorgeschaltet - und an anderer Stelle behandelt[42] - ist die Fragestellung nach der Generierung von Aktionsalternativen.

Demgegenüber beschäftigt sich globale Koordinierung mit dem Erreichen eines kohärenten Systemverhaltens. In Szenarien mit autonomen, parallel arbeitenden Agenten können viele Situationen auftreten, die die Kohärenz beeinträchtigen. Dazu zählen:

- Redundanzen: Planen zwei Agenten dieselben Aktivitäten, kann ein Synergieeffekt dadurch erzielt werden, daß nur ein Agent diese ausführt und den anderen am Ergebnis partizipieren läßt.
- Mangelnde Kompetenz: Ein Agent, der eine ihm übertragene Aufgabe nicht oder nur unzureichend erfüllen kann sollte versuchen, diese Aufgabe an einen besser geeigneten Agenten zu delegieren.
- Überlastsituationen: Ähnlich wie beim load balancing im Bereich des Netzwerkmanagements erhöht eine gleichmäßige Auslastung der gemeinsam an einer Aufgabe arbeitenden Agenten die Systemperformance.
- Behinderungen: Die Aktionen eines Agenten können die Handlungen anderer Agenten behindern. Derartige Situationen können in Form von Zielkonflikten frühzeitig erkannt und durch Verhandlungen bereinigt werden.
- Ressourcenengpässe: Agenten verbrauchen Ressourcen, z.B. in Form von Rechnerzeit oder der Belegung von Kommunikationskanälen. Dort, wo mehrere Agenten auf dieselben Betriebsmittel zugreifen ist der Zugriff oder die Verteilung zu regeln.
- Unterschiedliche Ansichten: Unterschiedliche sensorische Fähigkeiten, verschiedene Filterungsstrategien oder Repräsentationsformen können dazu führen, daß zwei Agenten ein unterschiedliches Bild derselben Umgebung haben. Die Erarbeitung einer gemeinsamen Weltsicht kann zu beiderseitigem Erkenntnisgewinn führen.

Die genannten Problemsituationen zeichnen sich dadurch aus, daß ein gemeinsames Vorgehen der beteiligten Agenten notwendig ist. Wegen der schwer vorherzusehbaren Systemdynamik und der Autonomieeigenschaft der Agenten sollten Lösungen

42. An dieser Stelle genügt es anzunehmen, daß einem Agenten zu einem Zeitpunkt mehrere Aktionen zur Auswahl stehen. Der vorgeschaltete Schritt zur Generierung von Aktionen besteht zumeist im Zugriff auf eine Handlungsbibliothek von und ist direkt in die Agentenarchitektur verankert. Alternative Ansätze finden sich in BDI-Logiken (Kapitel 2.4.2), regelbasierten Ansätzen und Plankalkülen (Kapitel 2.4.5).

flexibel durch Einigungsprozesse erarbeitet werden. Derartige zweckorientierte Konversationen werden durch die oben vorgestellten Interaktionsprotokolle modelliert, die, je nach dem Konversationsgegenstand, auch Verhandlungs-, Kooperations- oder Koordinationsprotokolle genannt werden.

2.3.1 Normen und Rollen

Eine Verankerung von sozialen Normen wie *hilfsbereit* oder *ehrlich* in Agenten erscheint auf den ersten Blick unnötig anthromorphisierend - welchen Sinn machen unehrliche oder nicht hilfsbereite Agenten, ist eine solche Attributierung überhaupt statthaft? Gründe für eine explizite Modellierung derartiger Verhaltenssteuerungen können Anwendungen bieten, in denen sich Agenten nicht per se wohlwollend oder kooperativ verhalten. Im Zusammenhang mit einem Rollenkonzept ist dann beispielsweise ausdrückbar, daß ein Agent sich einem *gleichgesinnten* oder *abhängigen* Agenten gegenüber *mitteilsam* verhält, während er gegenüber *Vorgesetzten* eher *reserviert* ist. Normen und Rollen sind so in der Lage, das Verhalten zu steuern, können aber umgekehrt auch zur Erklärung des Agentenverhaltens dienen.

Beispielsweise sind auf der Basis eines belief-Operators[43], der Wissen eines Agenten ausdrückt, Verhaltensweisen durch einfache Formeln beschreibbar:

$$tell_{ab}(p) \supset Belief_a(p)$$

$$tell_{ab}(p) \supset Belief_b(Belief_a(p))$$

$$tell_{ab}(p) \supset Belief_b(p)$$

Die erste Formel drückt „Ehrlichkeit" aus. Ein Agent a kommuniziert ein Fakt p (mittels tell) an einen anderen Agenten b nur dann, wenn er auch an die Wahrheit von p glaubt. Im 2. Ausdruck ist die Sicht des empfangenden Agenten b mit Hilfe von Meta-Wissen beschrieben. Eine Vorstellung von „Zuverlässigkeit" liegt dem drittten Ausdruck zugrunde: Unter der Voraussetzung, daß der Sender einer Information als zuverlässig gilt, kann der Empfänger das übermittelte Fakt in seine Datenbasis aufnehmen.

Mit ähnlichen Axiomatisierungen lassen sich auch dem Rationalitätsbegriff verwandte Eigenschaften modellieren. Ein Beispiel bildet die Stärke der Beziehungen zwischen den modalen Begriffen Glauben, Wunsch und Intention, die, antropomorphisierend, Realismus genannt wird. Cohen und Levesque[44] schlagen eine „enthusiastische" Realismuseigenschaft vor, nach der aus einer Intention logisch der

43. Der Belief-Operator wird im Abschnitt über Wissensrepräsentation (Abschnitt 2.4.2) behandelt.

44. [Cohen, Levesque 1987].

Wunsch und aus dem Wunsch logisch der Glaube ableitbar ist. Rao und Georgeff[45] formulieren einen schwächeren Realismus, indem sie aus dem Vorhandensein einer Intention lediglich schließen, daß gleichzeitig kein widersprüchlicher Wunsch existieren darf. Kalenka und Jennings beschäftigen sich mit der Modellierung von *social norms*[46] und modellieren auf ähnliche Weise „hilfsbereite" und „kooperative" Agenten.

2.3.2 Spieltheorie

Die Spieltheorie bietet ein formales und theoretisches Gerüst zur rationalen Entscheidungsfindung in konkurrierenden, d.h. nicht kooperativen Situationen. Üblicherweise werden mit der Spieltheorie zwei Gegenspieler betrachtet. Jeder repräsentiert in einer Payoff-Matrix eine vollständige Nutzenrelation zwischen den eigenen möglichen Aktionen und denen des Gegners. Für jede Aktion kann nicht nur der eigene Nutzen sondern auch der des Gegners aus der Matrix abgelesen werden. Somit besitzt jeder Agent ein genaues Bild vom Opponenten, wodurch beide in der Lage sind, begründete Entscheidungen zu fällen.

So gut sich die Spieltheorie zur Darstellung und Untersuchung von Konkurrenzsituationen eignet, ist sie jedoch generell in Agentensystemen nur bedingt einsetzbar. Nach Nwana[47] liegen die Hauptschwächen der Spieltheorie in einer Reihe von unrealistischen Grundannahmen. Zum einen sei die starke Rationalitätsannahme, Agenten als Nutzenoptimierer mit vordefinierten Strategien zu sehen, problematisch. Noch bedenklicher wiegt die Bedingung, daß alle Agenten durch Verwendung einer gemeinsamen Payoff-Matrix vollständiges Wissen über die anderen Agenten besitzen - Multiagentensysteme sind gewöhnlich durch unvollständiges und temporäres Wissen charakterisiert.

Überdies ist es schwierig, den Nutzen einer Situation in Zahlen auszudrücken. Die Verwendung einer metrischen Bewertung führt zur paarweisen Vergleichbarkeit von Situationen, was im allgemeinen nicht möglich oder erwünscht ist. Bei mehr als zwei Handelnden vergrößert sich zudem die Komplexität der Matrix erheblich. Für jede Person muß eine eigene Dimension existieren. Dies bedeutet kombinatorischen Aufwand bei der Repräsentation. So wird die Spieltheorie auch vornehmlich in Simulationen eingesetzt, deren Auswertung Rückschlüsse auf allgemeine Aspekte der Entscheidungsfindung und Strategien im Kontext mehrerer Handelnder oder Sinn oder Unsinn von Kooperationen in unterschiedlichen Situationen geben können[48].

45. [Rao, Georgeff 1991a].

46. [Kalenka, Jennings 1995], [Kalenka, Jennings 1997].

47. [Nwana 1996].

2.3.3 Scheduling

Mit Schedulingverfahren beschäftigt sich ein Teilbereich des Operations Research. Auch in Agenten kommt Reihenfolgeplanung zum Einsatz, wenn längerfristige Ziele verfolgt werden und somit in die Zukunft geplant wird. Aktivitätenmanagement im Sinne des Scheduling beinhaltet das Aufstellen und Verwalten einer Agenda für beabsichtigte Aktionen. Bei der Erstellung eines Schedules sind dabei die bekannten Abhängigkeiten, die zwischen Aktivitäten herrschen, zu berücksichtigen. Zumeist handelt es sich hierbei um zeitliche Restriktionen wie vorher-nachher-Beziehungen, es kann aber auch anderes Wissen wie Ressourcenverbrauch oder Prioritäten mit einfließen[49].

Abgesehen vom zusätzlichen Verwaltungsaufwand bei der Erstellung eines Zeitplans kann Scheduling großen Nutzen in Bezug auf flexibles Handeln eines Agenten bringen: Ein Agent, der sich einem anderen Agenten zur termingerechten Erledigung einer Aufgabe verpflichtet hat, repräsentiert diese Aktivität mit einem spätesten Anfangszeitpunkt in seinem Schedule. Anschließend kann er weitere Aktivitäten in vorhandene Lücken einbauen, ohne daß ein vollständiges Neuplanen aller Aktionen mit nachfolgendem Neuverhandeln mit dem auftraggebenden Agenten notwendig wäre.

Bei der Entwicklung von Agenten wird explizites Zeitmanagement von Aktivitäten jedoch kaum angewendet. Dies liegt zum einen an der NP-Komplexität dieser Problemklasse und dem schwierigen Design von Algorithmen, die den Anforderungen der Echtzeit, inkrementellen Arbeitsweise und der Berücksichtigung auch anderer Dimensionen als der Zeit gerecht werden. Zumeist findet sich Reihenfolgeplanung oder ein anderes Zeitmanagement implizit in der Architektur verankert.

2.3.4 Contract-Net-Protokoll

Unter den Interaktionsprotokollen ist das Contract-Net-Protokoll[50] der prominenteste Vertreter. Sein Zweck ist das Delegieren von Aufgaben an andere Agenten, aber auch die Vergabe von Ressourcen kann mit diesem Mechanismus gesteuert werden. Das Contract-Net-Protokoll kennt drei Rollen: einen Manager, mehrere Bidder und einen Contractor. Der Manager schreibt per Broadcast-Nachricht eine Aufgabe aus, die er erledigt haben möchte (task announcement). Spezifiziert werden Name, Pa-

48. [Rosenschein, Genesereth 1985], [Rosenschein 1986].

49. [Garvey, Lesser 1993], [Sycara, et al. 1990].

50. [Smith 1980]. Dieses Protokoll kommt in vielen Agentenanwendungen in unterschiedlichen Realisierungen (z.B. in [Sandholm 1993]) zum Einsatz.

rameter, Qualität sowie eine Zeitspanne (Timeout), innerhalb der Angebote zu machen sind.

Ein Agent, der eine Aufgabenausschreibung empfängt, kann mit einem Angebot (bid) antworten und schlüpft damit in die Rolle eines Bieters. Das Angebot ist dabei eine Konkretisierung der Ausschreibung. Der Manager wartet bis zum spezifizierten Timeout und wertet danach die eingetroffenen Angebote aus (evaluate bids). Anschließend erteilt er einem Bieter den Zuschlag (announce award), wodurch dieser zum Contractor wird. Bis zur vollständigen Erledigung der Aufgabe durch den Contractor mit abschließender Ergebnismitteilung per answer-result-Sprechakt tauschen Manager und Contractor Kontrollnachrichten aus, die über den Fortschritt der Abarbeitung Auskunft geben.

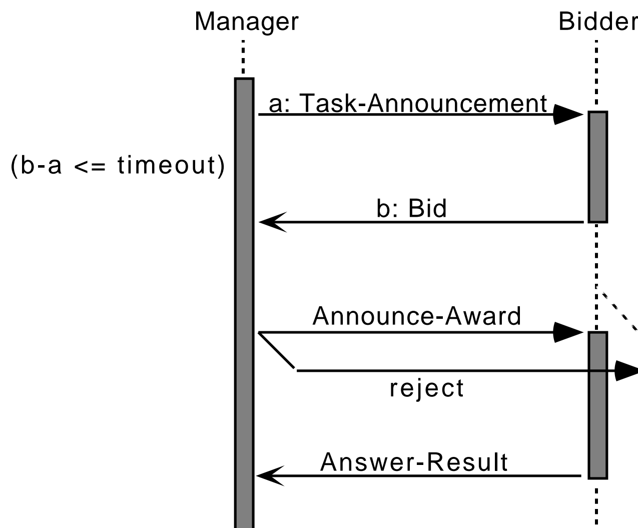

Abbildung 2-1: Contract-Net-Protokoll

Das Contract-Net-Protokoll deckt die zweite Phase des Distributed Problem Solving[51] ab. Allerdings bietet das Protokoll keine Unterstützung für den Prozeß der Problemzerlegung und läßt auch offen, wie die Bewertung der Angebote auszusehen hat. Außerdem ist für die Bieter weder festgelegt, wann sie Gebote abzugeben haben noch wann sie einen Zuschlag ablehnen sollen. Vielmehr beschränkt sich das Contract-Net-Protokoll auf die Beschreibung der auszutauschenden Nachrichten. Das ursprünglich von Davis und Smith entwickelte Protokoll kommt in verschiede-

51. Siehe Abschnitt 2.1.4 auf Seite 35.

nen Varianten zum Einsatz, beispielsweise in Form des Multistage-Negotiation-Protokolls[52], das eine Verallgemeinerung des Contract-Net-Protokolls darstellt. Es wurde für die Ressourcenzuteilung konzipiert und beinhaltet zusätzlich einen Mechanismus zur Auflösung von auftretenden Konflikten durch Constraintrelaxierungsmechanismen. In einer zyklischen Abfolge findet ein Informationsaustausch über Ressourcenkonflikte statt, wobei der Ausschreibungsgegenstand sukzessive verfeinert bzw. modifiziert wird.

2.3.5 Auktionen

Auktionen sind Protokolle, in denen die Preisbildung zwischen Anbietern und Nachfragern effizient und fair gestaltet wird. Im FIPA-Proposal für eine Agentenkommunikationssprache werden zwei Auktionsprotokolle beschrieben, um den Marktpreis einer Ware herauszufinden. Dabei beginnt man sich bei einer "englischen Auktion" (siehe Abb. 2-2) mit einem niedrigen Preisniveau und erhöht solange, bis keine neuen Gebote mehr eingehen. Mit dem Startsprechakt, der beispielsweise per Broadcast an alle Agenten geht, wird die Auktion eröffnet. Das Bieten erfolgt anonym, d.h. kein Bieter kann sich ein Bild von den Konkurrenten machen. Am Ende erhält der Meistbietende den Zuschlag, falls dessen Gebot über einem festgelegten Minimalpreis liegt. Die Auktion wird entsprechend durch Bekanntgabe beendet und der Bestbietende mit dem perform(action)-Sprechakt um Bezahlung gebeten.

52. [Conry, et al. 1986].

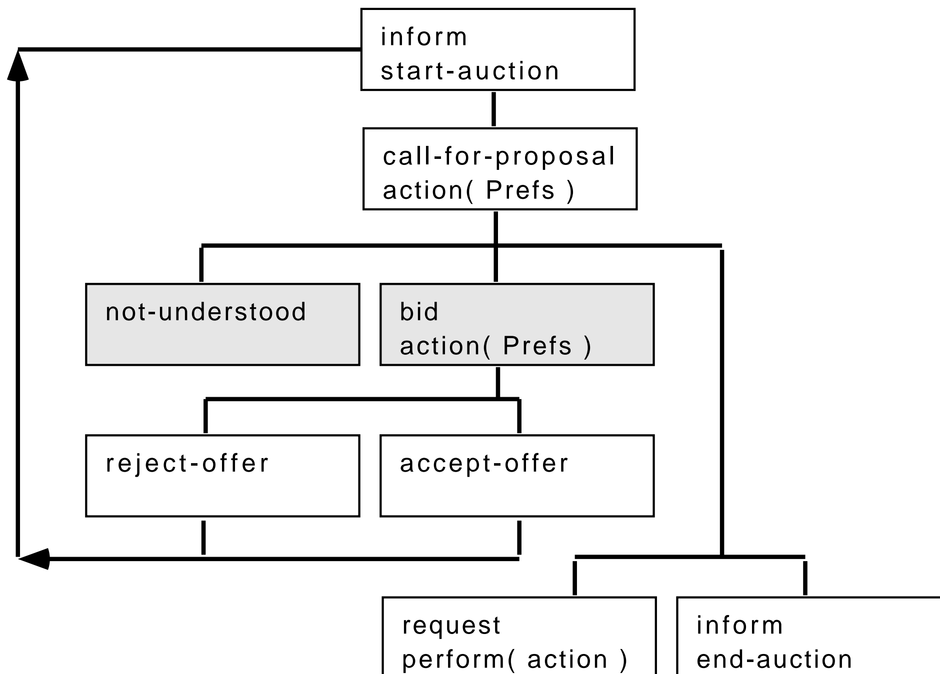

Abbildung 2-2: FIPA-Protokoll einer englischen Auktion[53]

Der umgekehrte Weg wird durch die sogenannte "holländische Auktion" beschritten. Hier startet der Auktionator mit einem offensichtlich über dem Marktwert liegenden Preis und reduziert solange, bis der erste Bieter akzeptiert. In beiden Modellen hat ein ausgezeichneter Auktionator das Heft des Handelns in seiner Hand. Andere in der Ökonomie häufig vorkommende Preisbildungsmechanismen wie Ruf-Auktionen, Maklervermittlung oder Vickery-Mechanismen sind bislang noch nicht in Agentenszenarien eingesetzt worden, lassen sich aber problemlos mit Zustandsgraphen modellieren.

2.3.6 Andere Koordinationsprotokolle

Für den Aufgabenbereich der Konfliktlösung haben Chang und Woo mit dem SANP das sogenannte Kampfmodell nachmodelliert[54]. Es wird von einem Koordi-

53. Aus [FIPA 1997].

54. Speech Act-based Negotiation Protocol [Chang, Woo 1992]. Die theoritische Grundlage findet sich in [Gulliver 1979], wo ein 8-stufiges Verhandlungsprotokoll vorgestellt wird.

nationsbedarf zwischen Agenten ausgegangen, der dadurch entsteht, daß diese Agenten unterschiedliche Ziele verfolgen oder die aktuelle Situation unterschiedlich interpretieren. Der Protokollablauf beginnt mit der Festlegung des Verhandlungsgegenstands, gefolgt von der Darlegung der Positionen, einer Minderung der Differenzen und dem eigentlichen Feilschen („bargaining“). Zur Verhandlungsstrategie gehören auch Entscheidungen darüber, wann ein Schlichter hinzgezogen werden muß. Vereinfachte Realisierungen dieser Idee stellen die auf dem Austausch von propose-, revise-, accept- und refuse-Sprechakten beruhenden Protokolle in Anwendungen von IMAGINE und Cool dar[55].

Ein anderer Ansatz zur Konfliktbehandlung von Durfee und Montgomery basiert auf der Repräsentation von Verhaltenshierarchien[56]. Das Verhandlungsprotokoll nutzt diese Hierarchien, um zunächst abstrakte Informationen über beabsichtigte Aktionen zwischen den Agenten auszutauschen. Wird eine Übereinstimmung festgestellt, liegt ein potentieller Konfliktfall vor. In diesem Fall kann ein Agent entscheiden, entweder die Situation durch Wahl einer anderen Aktion zu bereinigen oder sukzessive weitere, weniger abstrakte Informationen auszutauschen.

PERSUADER[57] verknüpft Verhandlungen mit fallbasiertem Schließen. Damit können zurückliegende Verhandlungen zur Gestaltung aktueller oder zukünftiger Konversationen hinzugezogen werden. Zwischen zwei Parteien werden auf Basis einer Menge konfliktierender Ziele und eines Kontextes zunächst initiale Vorschläge ausgetauscht, die danach wechselseitig zyklisch verändert bzw. verbessert werden, bis eine Einigung erzielt werden kann. Zusammen mit den übermittelten Kompromißangeboten werden Argumente mitgeliefert, die aus zurückliegenden Verhandlungsergebnissen abgeleitet und dem aktuellen Gegenstand angepaßt werden. Führt die Verhandlung nach einer vorgegebenen Zahl von Runden nicht zum Erfolg, wird ein Schlichter zu Rate gezogen.

2.3.7 Vermittlung und Organisation durch Faciltiators

Damit es zum Wissensaustausch zwischen zwei Agenten kommen kann, müssen sich die Kommunikationspartner adressieren können. Ein Agent, der der Allgemeinheit einen Dienst oder eine Information zur Verfügung stellen möchte oder bei der Suche nach einer Problemlösung auf fremde Hilfe angewiesen ist, verwendet dafür typischerweise eine Broadcast-Nachricht. Dieses „Rufen in den Wald“ kann

55. [Lux, et al. 1992] bzw. [Barbuceanu, Fox 1996].

56. [Durfee, Montgomery 1990].

57. [Sycara 1988].

jedoch zu einer erheblichen Belastung der Kommunikationsmedien und der Agenten führen.

Zur Minderung des „information overload“ können sogenannte Facilitator-Agenten[58] eingesetzt werden. Diese Spezialagenten fungieren als Informations-Broker für jeweils eine Gruppe von Agenten. Jeder Agent meldet „seinem“ Facilitator sämtliche Sprechakte, die er versteht und korrekt zu bearbeiten imstande ist. Durch die Vernetzung untereinander wirken die Facilitators als Dispatcher. Eine andere Funktionalität liegt im Bereitstellen aller Agentenadressen, die eine bestimmte Nachricht bearbeiten können. Daraufhin kann ein anfragender Agent direkt mit den fraglichen Agenten kommunizieren.

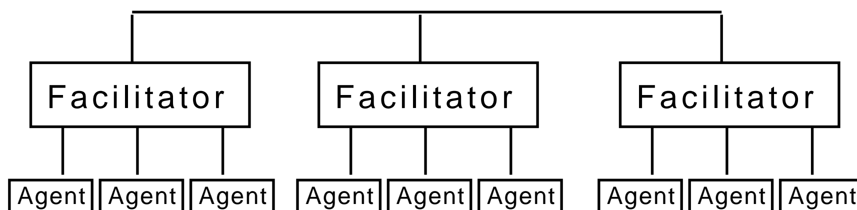

Abbildung 2-3: KQML-Agenten in einem föderierten System

Durch die mittelbare, assistierte Kommunikation und die Zuordnung von Agenten zu Facilitators entsteht ein föderiertes System, wie Abbildung 2-3 illustriert. Die Agenten geben ihre Autonomie zugunsten des sie vertretenden Vermittlers preis, indem sie jenem ihre Fähigkeiten und Wünsche mitteilen. Der Facilitator übernimmt die Verantwortung für die Erfüllung der Wünsche. Im Verhältnis zu Proxy-Agenten, Name-Servern oder Informations-Brokern sind Facilitators erheblich mächtiger. So sind sie in der Lage, Übersetzungen zwischen verschiedenen Sprachdialekten vorzunehmen, Nachrichten zu bündeln oder zu splitten[59].

2.3.8 Marktmechanismen

Die Mechanismen des Marktes, die auf Angebot und Nachfrage und auf persönlichem Nutzen beruhen, werden auch in Agentenszenarien eingesetzt. Ihre Attraktivität beziehen diese wohlverstandenen Techniken aus ihrer Effizienz, „Fairneß“ ge-

58. Facilitator-Agenten stammen von Vermittler-Agenten (mediators) ab [Genesereth 1992]. Der Sprachgebrauch des Facilitators stammt aus der KQML-Welt, in FIPA werden Nachrichtenvermittlung und Informationsdienste durch den Agent Resource Broker und das Agent Management System erbracht.

59. [Genesereth, Ketchpel 1994].

mäß den Prinzipien bzw. Gesetzen des Marktes und der universellen Verwendbarkeit bei Problemen der Ressourcenverteilung[60].

Agenten, die nach ökonomischen Prinzipien handeln, sind Nutzenoptimierer, verhalten sich also rational (vergleiche Seite 31). Mit dem Einsatz von Marktmechanismen werden zwei Ziele verfolgt: Einerseits wird ein auf Ausgleich abzielender, allgemeingültiger Rahmen vorgegeben, innerhalb dessen ein Agentensystem funktioniert. Auf der anderen Seite können auch zwischen konkurrierenden Agenten Kooperationen sinnvoll sein. Eine Möglichkeit besteht im dynamischen Bilden von Koalitionen. Hierbei geht es um das Aufdecken von Situationen, in denen Agenten bereit sind, mit anderen Agenten zusammenzuarbeiten. Die Möglichkeit einer Koalition ist dann gegeben, wenn der erwartete Nutzen für alle beteiligten Agenten bei einem gemeinsamen Vorgehen höher einzuschätzen ist als ohne Zusammenarbeit[61]. Die Etablierung einer Koalition kann iterativ durch gegenseitigen Austausch präferierter Aktionen, lokaler Berechnung des eigenen Nutzens, Vergleich und Berechnung eines Gesamtnutzens erfolgen[62]. Auch Abwandlungen des Contract-Net-Protokolls wurden zur Koalitionsformierung eingesetzt[63].

Ein einfacher und effizienter Mechanismus für die Preisverhandlung zweier Agenten - einer ist Anbieter und der andere Nachfrager einer Ware - wurde in KASBAH[64] integriert. Der Anbieter beginnt mit der Forderung eines Wunschpreises, den er mit der Zeit bis zu einem gesetzten Limit sukzessive herabsetzt. Umgekehrt bietet der Nachfrager zunächst ein kleines Entgelt, das er nachfolgend erhöht. Dabei sind linearen, quadratischen oder kubischen Preisbildungsstrategien über die Zeit möglich.

2.3.9 Zusammenfassung

Eine wichtige Anforderung an verteilte Systeme ist das koordinierte Handeln der einzelnen Glieder. In Agentensystemen sind diese Glieder weitgehend autonom, so daß Koordination per Systemdesign schwierig zu erreichen ist. Daher kommen unterschiedliche Techniken zur dynamischen Koordinierung zur Anwendung: Abgestimmtes Handeln zwischen mehreren Agenten findet zumeist in Form von Interaktionsprotokollen statt. Lokales Aktivitätenmanagement ist eng mit dem Begriff der

60. [Wellman 1995].

61. [Sandholm, Lesser 1997].

62. [Ketchpel 1994].

63. Beispielsweise das TRACONET von [Sandholm 1993].

64. [Chavez, Maes 1996].

„bounded rationality“ (siehe Seite 31) verbunden. Der Agent handelt bzw. entscheidet nach dem Prinzip der Nutzenoptimierung auf Basis seines aktuellen Wissens.

Vereinzelt wurden auch Möglichkeiten zur Erreichung global koordinierten Verhaltens ohne Interaktionsprotokolle untersucht. Eine Möglichkeit besteht darin, Agenten mit der Fähigkeit zur Selbstorganisation auszustatten. Parunak pflanzte nach dem Vorbild der Natur einfache, stochastisch gesteuerte Verhaltensmuster in Agenten ein[65] und zeigte damit, daß komplexes und gleichwohl koordiniertes Verhalten auch für einfache Agenten ohne Gedächtnis und mit nur primitiv ausgeprägter Wahrnehmung realisierbar ist. Einen anderen Weg beschreiten Ansätze, die die möglichen auftretenden Koordinationsfragen im Systemdesign berücksichtigen. Jeder Agent arbeitet dann nach fest vorgegebenen lokalen Präferenzen, in denen das gewünschte globale Systemverhalten kodiert ist. Im Prinzip lösen hierbei die Agenten Teile eines verteilten Algorithmus[66]. In der Praxis ist es jedoch schwierig, alle möglichen auftretenden Koordinationsfragen im Systemdesign zu berücksichtigen, da sie schwerlich vollständig zu erfassen sind. Schließlich können Agenten unter bestimmten Bedingungen auch durch Rationalitätsannahmen und Schlußfolgerungen aufgrund des Verhaltens anderer Agenten ohne Kommunikation miteinander kooperieren[67].

Wenn Agenten nur unvollständiges Wissen über ihre Umgebung besitzen und sich jene darüberhinaus dynamisch verändert, werden Koordinationsprobleme am adäquatesten durch Protokolle geregelt; kommunikationslose Verfahren scheitern bei wachsender Komplexität und Unsicherheit. Die wichtige Bedeutung einer expliziten Modellierung von Koordinationswissen kommt in vielen Ansätzen zum Ausdruck. Jennings schlägt einen cooperation knowledge level vor[68], in der INTERRAP-Agentenarchitektur findet sich ein cooperation level[69], Abichiche repräsentiert den Kooperationsprozeß symbolisch[70].

Häufig findet das Aktivitätenmanagement eines Agenten implizit statt, d.h. es ist nicht Gegenstand der Agentenmodellierung oder -programmierung, sondern ist im Design versteckt. Beispielsweise regelt die Ablaufsteuerung, welche die Kontrolle

65. [Parunak 1997]. Derart implizit koordiniertes Verhalten findet man bei staatenbildenden Insekten, Fischschwärmen oder im Jagdverhalten von Löwen und Wölfen.

66. Informationen zu diesem Ansatz finden sich bei [Baker 1998].

67. [Genesereth, et al. 1986].

68. [Jennings, Wittig 1992].

69. [Müller, Pischel 1994], [Müller, et al. 1995].

70. [Abchiche, et al. 1992].

ausübt und durch die Agentenarchitektur vorgegeben ist, häufig Rang- und Reihenfolge der Aktionen (siehe Abschnitt 3.1 auf Seite 68).

2.4 Aktorische und kognitive Fähigkeiten

Softwareagenten verfügen im Gegensatz zu Industrierobotern nicht über physikalisch wirkende Effektoren, stattdessen ist die Aktorik algorithmischer Natur. In der einschlägigen Literatur finden sich hierfür auch die Begriffe Fähigkeiten, Aktionen, Behaviours, Pläne, Prozeduren und Methoden. Für die Modellierung und Programmierung der agentenspezifischen Fähigkeiten gibt es folgende Dinge zu beachten:

- Agenten benötigen Wissen über ihre Fähigkeiten, damit sie diese als Dienste der übrigen Welt anbieten können. Auch für die lokalen Koordinationsaufgaben (siehe Abschnitt 2.3) ist derartiges Meta-Wissen notwendig.
- Abhängig vom Wissensstand des Agenten kann es sinnvoll oder notwendig sein, von geplanten Aktivitäten abzulassen oder in Ausführung befindliche Aktionen abzubrechen. Eine Sprache zur Beschreibung von Fähigkeiten sollte daher vom Agenten interpretiert werden und möglichst eine modulare Struktur mit atomaren Aktionen geringer Komplexität besitzen.
- Ein höheres Maß an Flexibilität und Intelligenz bekommen Agenten, wenn sie mit kognitiven Fähigkeiten ausgestattet werden. Derartige Fertigkeiten realisieren die im ersten Punkt angesprochene Meta-Ebene; sie ermöglichen begründetes, in die Zukunft gerichtetes Handeln. Kognitive Fähigkeiten beruhen auf Symbolverarbeitung und damit auf auf Wissensrepräsentationsformalismen.

Die Fähigkeiten eines Agenten sind also nicht aus rein funktionaler Sicht zu betrachten sondern beinhalten auch Modellierungsaspekte wie Unterbrechbarkeit, Deklarativität und Wissensrepräsentation.

2.4.1 Prozedurale Repräsentationsformen

Prozedurale Repräsentationen besitzen gegenüber deklarativen Darstellungsformen den Vorteil der besserne Performance, da sie direkt ausführbar sind. Neben einer direkten Codierung kommen vor allen Dingen vier Ansätze in Frage: interpretierte Sprachen, Realzeitalgorithmen, Produktionssysteme und Planoperatoren.

- Interpretierte Sprachen bieten gegenüber compilierten dem Prozessor, d.h. dem Agenten, eine bessere Kontrolle über die Ausführung: Die Verarbeitung kann jederzeit abgebrochen werden, jedoch mit dem Nachteil ohne Ergebnis zu terminieren.
- Anytime-Algorithmen machen Agenten realzeitfähig. Sie zeichnen sich dadurch aus, daß sie schnellstmöglich eine erste approximative Lösung berechnen und

diese dann iterativ bis zum optimalen Ergebnis verfeinern. Das Verfahren ist jederzeit abbrechbar und liefert die Lösung der aktuellen Iteration. Jedoch sind Anytime-Algorithmen schwierig zu entwerfen und existieren bislang nur für wenige Domänen[71].

- Ein anderer Ansatz in Richtung Real-Time-Computing liegt in der Verwendung *multipler Methoden*, welche eine Anzahl unterschiedlicher Algorithmen zu einer Problemklasse zusammenfassen. Dazu muß Wissen verfügbar sein, für welche Probleminstanzen welcher Algorithmus am schnellsten terminiert oder das beste Resultat liefert. Eine Monitorkomponente überwacht die Termintreue und reagiert entsprechend bei Zeitknappheit durch Aktivierung einer anderen Methode. Dabei ist es wesentlich, daß die Algorithmen strukturell ähnlich sind und Zwischenergebnisse anderer Algorithmen verwerten können. Nur so ist sichergestellt, daß beim Wechsel von einem Algorithmus zu einem anderen die Rechenzeit des ersten nicht verschwendet ist[72].
- Produktionssysteme zählen zu den sogenannten schwachen Problemlösemethoden der KI[73]. Handlungsmöglichkeiten werden durch Regeln beschrieben, die aus einer Vorbedingung und einem Aktionsteil bestehen. Ein Regelinterpreter steuert die Ausführung, indem er jedesmal, wenn ein neues Fakt in die Wissensbasis eintritt, alle Vorbedingungen des Regelsystems auf Anwendbarkeit überprüft (Konfliktmengenberechnung). Je nach Design des Regelinterpreters können auch nichtdeterministische Arbeitsweisen realisiert werden: Wenn jeweils nur eine Regel zur Anwendung kommen darf, enthält die Konfliktmenge alternative Aktionen, die im Falle eines Scheiterns oder Widerrufens verwendet werden können. Prolog-Inferenzmaschinen funktionieren auf diese Weise.

  Regeln bieten einen einfachen und universellen Mechanismus zur Verknüpfung von Wissen - bzw Zuständen - und Handlungen. Sie sind eher strukturschwach und machen keine Einschränkungen in Bezug auf die Granularität des modellierten Wissens. Prinzipiell kann der Regelinterpreter derart gestaltet werden, daß er seine Arbeit mit vertretbarem Aufwand gegebenenfalls unterbrechen kann.
- Handlungspläne stelle eine modulare Form komplexerer Handlungsbeschreibungen für Agenten dar. Üblicherweise werden derartige Pläne durch baumarti-

71. Anytime-Scheduling-Algorithmen finden sich bei [Boddy, Dean 1994], [Horvitz 1987], [Zilberstein, Russell 1996]. Zu den iterativen Verfeinerungsalgorithmen zählen ferner im Bereich der KI heuristische Suchmethoden und Monte-Carlo-Techniken, aus dem Operations Research die dynamische Programmierung sowie numerische Approximationsverfahren.

72. [Garvey, Lesser 1993], [Zilberstein, Russell 1995], [Zilberstein, Russell 1996]. Einen Überblick über die Forschung in der real-time-KI geben [Garvey, Lesser 1994].

73. Bekanntester Vertreter der Produktionssysteme ist Ops5 [Forgy 1981]. Neben der Anwendung in zahlreichen Expertensystemshells kommt Ops5 beispielsweise auch in Soar (siehe Abschnitt „SOAR“ auf Seite 76) oder der Agentenbeschreibungssprache Magsy [Fischer 1993] zum Einsatz.

ge Strukturen beschrieben, die den Kontrollfluß auf abstraktem Niveau widerspiegeln. Verzweigungen stellen Alternativen dar, die der Agent beschreiten kann. Häufig finden sich in den Knoten derartiger Planbeschreibungen nicht einfach nur prozedurale Aktionen sondern auch Subpläne und Ziele, zu deren Erreichung wiederum Pläne auszuwählen sind.

2.4.2 Modallogiken

Die Prädikatenlogik erster Stufe definiert eine formale Sprache zur Repräsentation mathematischer Wahrheiten (Axiome) und ermöglicht durch Anwendung einer endlichen Menge wahrheitsbewahrender Regeln die Ableitung aller gemäß der axiomatischen Vorgaben wahren Aussagen. Was die Prädikatenlogik für die Wissensverarbeitung interessant macht, ist die Verfügbarkeit mächtiger Inferenzmechanismen sowie die strikte Trennung von Syntax und Semantik.

Ursprünglich ist die Prädikatenlogik für statische, mathematische Welten konzipiert. Mit der Geburt der KI wurde dann das Interesse geweckt, auch die reale Welt mit diesem Formalismus zu repräsentieren und das Wissen entsprechend zu verarbeiten. Hier weist die Logik jedoch sowohl als Formalismus als auch als Reasoning-Mechanismus gravierende Schwächen auf: Einerseits ist die Monotonieannahme für die reale Welt inpraktikabel, andererseits lassen sich weder dynamische Aspekte noch unklare und widersprüchliche Sachverhalte ausdrücken. Diese Diskrepanzen haben zu einer Reihe von Logikerweiterungen geführt, von denen im Bereich der Agenten hauptsächlich Modallogiken eingesetzt werden.

Modallogiken gehören zu den nichtmonotonen Logiken[74], die von der *logischen* Schlußfolgerung abkehren und *sinnvolle* Schlußfolgerungen erlauben. Dabei kann das Wissen mit der Zeit durchaus abnehmen. Modale Logiken realisieren nichtmonotones Schließen mittels sogenannter modaler Operatoren. Epistemische Logiken drücken Wissen und Glauben aus[75]. In der autoepistemischen Logik[76] wird der modale Belief-Operator benutzt, um einerseits Wissen über die Welt und andererseits selbstbezüglichen Glauben bzgl. des eigenen Wissens zu modellieren. Als Ergebnis ist der Agent in der Lage, Schlüsse über die Welt mit dem Bewußtsein eigener Informationsdefizite zu ziehen. Ein Agent glaubt ein Fakt genau dann, wenn es in sei-

74. Andere Vertreter nichtmonotonen Logiken, die auch mit kontradiktorischem Wissen umgehen können, sind Circumscription [McCarthy 1980] und Default-Logik [Reiter 1980]. Sie wurden in Agenten jedoch bislang nicht verwendet.

75. [Pap 1957]. Nach [Wright 1951] (in [Struth 1994] zusammengefaßt) existiert ein ganzer Katalog unterschiedlicher Modalitäten, wie temporale, assertive, auswertende, kausale und weitere mehr. Einen guten Überblick über epistemische Logiken geben Halpern und Moses [Halpern, Moses 1992].

76. [Moore 1985].

ner Belief-Datenbasis enthalten ist. Auf diese Weise sind autoepistemische Logiken korrekt und vollständig in Bezug auf die Beliefs eines Agenten.

Die Lücke zwischen Wissen und Handeln überbrücken modale Belief-Desire-Intention-Logiken, kurz BDI-Logiken genannt. Nach Dennet sind BDI-Logiken intensionale Systeme[77] und eignen sich daher gut für die Beschreibung von Agenten. Beliefs drücken das Wissen eines Agenten zu einem gegebenen Zeitpunkt aus; Desires beschreiben aktuell nicht vorhandene, aber erwünschte Zustände; Intentions charakterisieren Bedingungen (Situationen oder Zustände), die während der Verarbeitung zur Erreichung aktueller Ziele eingehalten oder erreicht werden müssen[78].

In der Agentengemeinde herrscht Uneinigkeit darüber, welche Modalitäten zu nutzen sind: Cohen und Levesque benötigen nur Beliefs und Goals und leiten daraus eine Formalisierung für Intentionen ab[79]. Rao und Georgeff dagegen verwenden auch Intention als modalen Begriff[80], und noch einen Schritt weiter geht Singh durch Verwendung von Wissen, Know-How und Kommunikation[81]. Weitere mentale Begriffe sind Goals, Joint Goals (gemeinsame Absichten), (Joint) Commitments oder (Joint) Actions[82].

Axiomatisierungen

Auf der Basis der philosophischen Arbeiten von Bratman wurden BDI-Logiken in verschiedenen Arbeiten formallogisch fundiert[83]. Der Zusammenhang zwischen den verschiedenen Modalitäten wird durch eine Menge von Axiomen beschrieben. Hier bietet sich eine Fülle von Möglichkeiten, die unterschiedliche Sichtweisen auf das Verhalten der Agenten widerspiegeln.

77. [Dennet 1987].

78. [Singh, et al. 1998], S. 13f.

79. [Cohen, Levesque 1990], [Cohen, et al. 1990], [Cohen, Levesque 1987]. Sie verwenden das Konzept des "persistenten Ziels", das in der Zukunft wahr werden muß und solange aktiv ist, bis der Agent glaubt, daß er es erfüllt hat oder es nicht erfüllbar ist. Intention zu handeln beruht auf einem persistenten Ziel und einer Aktion, mit der das Ziel erreicht wird.

80. [Rao, Georgeff 1991a], [Rao, Georgeff 1991b].

81. [Singh 1994].

82. [Cohen, Levesque 1987], [Cohen, Levesque 1995].

83. [Bratman 1987]. Siehe dazu die Arbeiten von [Cohen, Levesque 1990], [Rao, Georgeff 1992] und [Singh 1994].

Daß Axiome nicht beliebig formuliert werden sollten, weil sonst die intendierte Bedeutung der Modaloperatoren ad abdurdum geführt werden könnte, ist unmittelbar einsichtig. So widerspricht die logische Folgerbarkeit dem Sinn der Absicht. Wenn ich beispielsweise die Intention habe, zum Zahnarzt zu gehen und ein Zahnarztbesuch logisch mit Schmerzen verknüpft ist, dann ist es nicht sinnvoll zu folgern, daß ich beabsichtige, mir Schmerzen zufügen zu lassen. Auch ist es wenig sinnvoll, eine Aussage, die für alle Ewigkeit als wahr betrachtet wird, als Ziel oder Intention zu übernehmen[84].

$$Goal(p) \supset Belief(Goal(p)) \qquad Intention(p) \supset Belief(Intention(p))$$
$$Goal(p) \supset Belief(p)$$
$$Goal(p) \supset not(Belief(not(p))) \qquad Intention(p) \supset not(Belief(not(p)))$$

Das erste der drei obigen Axiome stellt einen eher trivialen Zusammenhang zwischen einem Ziel und dem Glauben an den Zielzustand dar. Rao und Georgeff nennen dies „strengen Realismus"[85], da die Zielewelt eine Submenge der Belief-Welt darstellt. Selbiges gilt für die Beziehung zwischen Intentionen und Beliefs. Alternativ formulieren sie „schwachen Realismus" mittels negierter Ausdrücke (in der dritten Zeile). Die zweite Aussage ist dagegen problematisch, drückt sie doch aus, daß ein Agent bereits an einen noch nicht erreichten Zielzustand glaubt. Cohen und Levesque modellieren einen „überenthusiastischen" Realismus, bei dem sich aus Beliefs Ziele und aus Zielen Intentionen ableiten lassen[86].

Neben der expliziten Repräsentation des internen mentalen Zustands erfüllen die intentionalen Attitüden einen weiteren Zweck. Die Kommunikation von Beliefs und Zielen hilft Agenten sich ein Bild vom Zustand anderer Agenten zu machen. Dies kann als Basis für Verhandlungen, gemeinsame Planung oder koordinierte Aktivitäten dienen. Wenn Agenten Pläne austauschen, können potentielle Konfliktsituationen frühzeitig erkannt und entsprechend ausgeräumt werden. Daß auch Kommunikation in das Belief-Modell eingebunden werden kann, wurde bereits in Abschnitt 2.3.1 aufgezeigt.

2.4.3 Zeitliches Wissen

Zeitliche Zusammenhänge spielen in vielen Agentenszenarien eine wichtige Rolle[87]. Ein Indiz hierfür liefert die zentrale Bedeutung der Agentenkoordination: Ak-

84. [Cohen, Levesque 1987].

85. [Rao, Georgeff 1991b].

86. [Cohen, Levesque 1987].

tionen unterschiedlicher Agenten stehen häufig in zeitlichem Zusammenhang derart, daß zur Erreichung eines Zieles mehrere Agenten entweder gleichzeitig handeln müssen, oder aufgrund gemeinsam genutzter Ressourcen geplante Aktivitäten andersartig zu synchronisieren sind.

Die Repräsentation zeitlichen Wissens basiert entweder auf Intervalldarstellungen oder auf Zeitpunkten. Relationen zwischen zeitlichen Intervallen wurden von Allen[88] eingeführt. Im Gegensatz zu Allens Intervallalgebra verwenden Zeitpunktansätze wie der Situationskalkül[89] eine explizite Notation von Zeit. Andere metrische Ansätze benutzen lineare Ungleichungen oder zeitliche Constraints[90]. Auch Kombinationen beider Verfahren sind möglich[91]. Zur Darstellung zeitabhängiger Beliefs gelangt man durch Einführung neuer Operatoren wie „Vergangenheit" oder „Zukunft", die auf Belief-Ausdrücke anwendbar sind. Erweiterungen dieser Art werden modale temporale Logiken genannt. Die Einführung expliziter Zeitnotationen kann auch durch Erweiterung der Belief-Ausdrücke um Konstanten und Variablen, die Zeitpunkte repräsentieren, erfolgen[92].

2.4.4 Planung und Planausführung

Aktionsplanung ist ein viel bearbeitetes Teilgebiet der KI. Um ein vorgegebenes Ziel zu erreichen, muß ein Agent eine Menge von Aktivitäten (Planoperatoren) in geeigneter Reihenfolge durchführen. Es ist Aufgabe eines Planers, den - oft unendlich großen - Planungsraum effizient zu bearbeiten, so daß Lösungen in Form einer Sequenz von Aktionen in akzeptabler Zeit errechnet werden können. Dabei stellt nicht nur die zielgerichtete Suche sondern auch die Schwierigkeit einer geeigneten Repräsentation der Planoperatoren ein Problem dar[93]. Sowohl in der Beschreibung der Anwendbarkeit eines Operators (Qualification-Problem) wie auch der Auswir-

87. Franklin und Graesser [Franklin, Graesser 1996] betonen in ihrer Definition eines Agenten ein zeitliches Bewußtsein: *„An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future"*.

88. [Allen 1983], [Allen 1984].

89. [McCarthy, Hayes 1969].

90. [Malik, Binford 1983] bzw. [Dechter, et al. 1991].

91. [McDermott 1982], [Kautz, Ladkin 1992].

92. Diese Art der Zeitdarstellung wird in Agent-0 (siehe Abbildung 4.1 auf Seite 86) verwendet.

93. Planoperatoren gehen auf STRIPS [Fikes, Nilsson 1971] zurück und verwenden üblicherweise wohldefinierte Formeln aus der Prädikatenlogik erster Stufe.

kungen (Ramification-Problem) und unveränderten Merkmale (Frame-Problem) der Umwelt liegen Komplexitätsfallen[94].

Der Nutzen von Aktionsplanung für Agenten hängt stark vom Einsatzgebiet ab. So ist detailliertes Planen in schnell veränderlichen Umgebungen wenig sinnvoll, weil nicht sichergestellt werden kann, daß ein Plan nach seiner Fertigstellung noch ausführbar ist. Wichtig ist, Planung und Ausführung miteinander zu verzahnen, so daß Inkonsistenzen schnell festgestellt werden können.

- **Wiederverwendbare Pläne:** Hat der Agent Zugriff auf eine Bibliothek von allgemeinen oder abstrakten Plänen, besteht wie beim case-based-Reasoning die Aufgabe darin, ein Mapping von einem aktuellen Problem auf Pläne der Bibliothek zu finden, die entsprechenden Pläne mit den aktuellen Parametern zu instantiieren und dann mit der Planabarbeitung zu beginnen.
- **Partielle Pläne:** Zunächst wird ein generischer oder abstrakter Plan gebildet, der zum einen schnell zu erstellen ist und zum andern eine gewisse Flexibilität bezüglich der konkreten Ausformulierung offen läßt. Derartige Planskelette werden beim hierarchischen Planen verwendet[95]. Die Feinplanung erfolgt dann jeweils für das nächste überschaubare Teilproblem, so daß Planen und Handeln eng verzahnt ablaufen. Außerdem kann das grobe Planskelett auch bei geänderten Situationen weiter verwendet werden; eine Neuplanung betrifft u.U. nur einen Teilbereich des gesamten Plans.

Andere Ansätze für komplexe und dynamische Umgebungen integrieren Planung, Planausführung, Beobachten (Monitoring) von ausgeführten Aktionen zusammen mit Mechanismen zur Um- oder Neuplanung[96]. Umgang mit rapide wechselnden Umgebungen versprechen Verfahren zur Planreparatur, bei der ein initialer Plan iterativ verändert wird[97]. Kurzfristiges Planen in Zusammenhang mit least-commitment-Strategien kann die notwendige Reaktivität berücksichtigen[98]. Untersucht wurden auch probabilistische Verfahren zur Planung mit unsicherem Wissen[99].

94. [McCarthy, Hayes 1969], [McCarthy 1977], [Shoham, McDermott 1988], [Hertzberg 1994].

95. Z.B. in NOAH [Sacerdoti 1974].

96. NASL [McDermott 1990] verknüpft die Erstellung eines Plans mit dessen Ausführung; darüberhinaus integrieren die Planungssysteme SIPE [Wilkins 1985] und IPEM [Ambros-Ingerson, Steel 1990] Monitoring und Replanning.

97. [Gent, et al. 1997].

98. [Weld 1994]. Ein prominentes Beispiel für den Erfolg dieses Ansatzes stellt der Roboter Shakey [Nilsson 1984] dar.

99. Beispiele für probabilistische Planer sind die Buridan-Systeme von [Boutilier, et al. 1995]; [Kass, Raferty 1994] und [Lavine, Schervish 1997] verwendeten Bayes-Netze.

2.4.5 Lernen und Adaption

Lernen und Adaption sind kognitive Fähigkeiten, die auf die Zukunft gerichtet sind. Dabei beschreibt Lernen die Fähigkeit, neues *Wissen* zu erlangen und nutzbar zu machen, während Adaption das an veränderte Situationen angepaßte *Verhalten* betont. Beide Begriffe zielen auf die Verbesserung der Leistungsfähigkeit ab und werden im folgenden synonym verwendet.

Lernfähigkeit ist generell eine wünschenswerte Eigenschaft von Systemen, stellt sie doch ein Merkmal von Intelligenz dar. Darüberhinaus sind adaptive Fähigkeiten für Agenten wichtig, weil sie langlebige Prozesse darstellen. Wenn ein Agent dieselben Aktionen wiederholt ausführt sollte er entsprechend vom gesammelten Erfahrungswissen früherer Aktivierungen profitieren können. Außerdem macht es die oft komplexe und nicht vollständig bekannte Einsatzumgebung in der Praxis unmöglich, einen Agenten mit ausreichenden Fähigkeiten und Wissen auszustatten.

Dem Nutzen von Lernfähigkeit steht die Schwierigkeit bei der Entwicklung von Lernmethoden und deren Komplexität gegenüber. Eine Reihe von Entwurfsentscheidungen sind zu treffen, die den Tradeoff zwischen Kosten des Lernens und Nutzen des Erlernten beeinflussen: Was soll gelernt werden: Fakten oder Metawissen? Wie wird gelernt: reflexiv oder deliberativ, assistiert oder automatisch? Wie wird mit kontradiktorischem Wissen umgegangen? Was wird wann vergessen?

Wie schon beim Planen bestimmen die Umgebungsmerkmale die verwendbaren Lernverfahren. Über den Wert des Gelernten kann a priori keine Aussage getroffen werden, so daß aufwendige Lernverfahren ungeeignet sind. Darüberhinaus fehlt für erklärungsbasiertes Lernen oder Generalisieren im allgemeinen das notwendige Wissen bzw. die Kontrolle. Auch assistiertes Lernen ist aufgrund der Entkopplung des Agenten vom Menschen zumeist nicht möglich. Die am häufigsten in Agenten eingesetzten Konzepte beruhen daher auf einfachen Mechanismen:

- **Caching:** Eine sehr einfache und effiziente, reflexartige Form des Lernens stellt Caching dar. In einer Art Kurzzeitgedächtnis speichert der Agent Wissen, das er bei Bedarf wieder abrufen kann. Da Caching nur das Abspeichern von Informationen, nicht aber Aggregation oder Abstraktion beinhaltet, ist diese Form des Lernens zumeist auf das Speichern und Abrufen von Ergebnissen lokaler Berechnungen oder Informationsanfragen beschränkt. Nur wenn die Welt - bzw deren Repräsentation - von geringer Komplexität ist, macht auch das Zwischenspeichern von situationsbedingten Handlungen einen Sinn.
- **Lernen durch Feedback:** Durch *Reinforcement Learning* (RL) lernt ein Agent Verhalten durch trial-and-error-Interaktionen mit einer dynamischen Umgebung. Bei diesem Verfahren sind Handeln und Lernen eng gekoppelt. Im Gegensatz zu den meisten anderen Lernverfahren arbeitet RL nicht auf symbolichen Repräsentationen. Stattdessen beruht die Anwendbarkeit von RL-Mechanismen auf drei Annahmen: einer diskreten Menge von Zuständen der Umgebung, einer

diskreten Menge von Agentenaktionen und einer Menge skalarer Reinforcement-Signale[100]. RL verarbeitet das unmittelbare Feedback der Umgebung als Antwort auf eine durchgeführte Aktion, indem er die Anwendbarkeitsbedingung der eben ausgeführten Handlung entsprechend dem Reinforcement-Signal anpaßt.

2.4.6 Zusammenfassung

Weil autonome Agenten ohne ständige Kontrolle eines Benutzers arbeiten, benötigen sie Fähigkeiten zur Steuerung des eigenen Handelns. Die Entscheidung, was als nächstes zu tun ist, beruht auf Wissen, das auf einer Meta-Ebene angesiedelt ist. Dieses Wissen dient nicht zum direkten Handeln, sondern ist Input kognitiver Prozesse, die ihrerseits in Aktionen münden. Deklarative Repräsentationen eignen sich für die Beschreibung dieser Meta-Ebene. Vorteile dieser Darstellungsart gegenüber prozeduraler Repräsentation liegen in der Dynamik des Wissens, die durch Schlußfolgerungen erzielt wird. Im Vergleich zu fest verdrahteten Handlungsweisen bringt die Verarbeitung deklarativen Wissens jedoch eine zusätzliche Verarbeitungsebene ins Spiel. Auch darf der hohe Aufwand, den ein Agent zur Verwaltung seines Weltmodells in dynamischen Umgebungen betreiben muß, nicht vernachlässigt werden. Auf der anderen Seite sind viele komplexe Aktivitäten prozeduraler Natur. Ein Beispiel stellt das Fahrradfahren dar, das auf erlernten motorischen Fähigkeiten beruht und ohne Wissensebene funktioniert. Jedoch führt die ausschließliche Verwendung prozeduralen Wissens in einem Agenten zur Frage, wie der Agent „wissen kann, was er weiß“.

Um temporale und modale Aspekte erweiterte Logiken spiegeln den Bedarf einerseits nach zeitlich bedingten Aussagen und andererseits nach intentionalen Aspekten und der Behandlung unsicheren Wissens wider. Während über die Verwendung von mentalen Begriffen zur Beschreibung von Agentenzuständen weitgehende Einigkeit in der DAI-Gemeinde herrscht, wird die Verwendung von modalen Logiken kontrovers diskutiert. Weder über die zu verwendenden Modaloperatoren[101] noch über deren Verarbeitung herrscht Klarheit. Darüberhinaus bereitet die Verarbeitung der Modallogiken aufgrund der Unentscheidbarkeit der Prädikatenlogik - und damit insbesondere auch der hier beschriebenen erweiterten Formalismen - Schwierigkeiten. Gängige Axiomatisierungen tendieren dazu, entweder wenig aussagekräftig, oder aber zu stark und damit nicht gut auf die reale

100.[Kaelbling, et al. 1996].

101.Neben dem namensgebenden Belief-, Desire-, Intention-Tripel existieren Ansätze, die entweder weniger, zusätzliche oder andere Begriffe verwenden. So kommt Shoham in seiner Agent-0-Beschreibungssprache mit Beliefs und aus und ergänzt noch einen nicht-mentalen Capability-Operator.

Welt übertragbar zu sein. Von ihrem Wesen her stellen BDI-Logiken eher einen Constraintformalismus als ein Verarbeitungsmodell dar.

Allgemein verspricht die Integration kognitiver Fähigkeiten intelligenteres, in die Zukunft gerichtetes Handeln. Planziele oder teilausgearbeitete Pläne können zwischen Agenten kommuniziert werden und Rückschlüsse für das Handeln anderer Agenten ermöglichen; erlerntes Wissen hilft Fehler zu vermeiden und dient der Performance-Steigerung. Auf der anderen Seite ist der Nutzen von Handlungsplanung in stark dynamischen Umgebungen generell fraglich[102]: Unvorhergesehene Ereignisse können die Ausführung eines Plans verhindern, so daß zeitaufwendiges Replanning notwendig wird. Deshalb kommen nahezu ausschließlich wiederverwendbare Pläne zur Anwendung[103], d.h. Agenten besitzen abstrakte Pläne, die situationsabhängig abgerufen, instantiiert und ausgeführt werden. Auch der Einfluß des maschinellen Lernens ist aufgrund von charakteristischen Umgebungsmerkmalen eher gering. Assistiertes Lernen ist nicht möglich, außerdem ist der Wert des Erlernten aufgrund der unsteten Umgebung schwer eingeschätzbar. So kommen zumeist nur primitive Methoden wie Caching oder Feedback-Lernen zum Einsatz.

2.5 Zusammenfassung und Bewertung

Ein allgemein akzeptierter Agentenbegriff existiert nicht. Stattdessen werden charakterisierende Eigenschaften zur Unterscheidung von nicht-agentischer Software herangezogen. Verschiedene Einsatzgebiete und damit verbundene unterschiedliche Anforderungen führen schließlich zur Identifizierung von Agententypen. Die markantesten Merkmale von Agenten umfassen sprechaktbasierte Kommunikation, Fähigkeiten zur Koordination auf lokaler wie globaler Ebene und die explizite Repräsentation mentaler Zustände, mit denen zielgerichtetes Verhalten modelliert wird. Zu jedem dieser Aspekte existiert eine Vielzahl unterschiedlicher Modellierungs- und Implementierungsansätze, wodurch sich die Gefahr der „Inkompatibilität“ erhöht: So scheitert die Kommunikation zwischen einem FIPA-konformen und einem KQML-basierten Agenten schon an der unterschiedlichen Syntax; Agenten, die nicht dasselbe Interaktionsprotokoll beherrschen, können nicht kooperieren, und verschiedene Repräsentationen mentaler Zustände führen dazu, daß Begriffe wie Ziele zwar kommunizierbar, aber nicht verarbeitetbar sind.

Der Anwendungsbereich Diensterbringung vereinigt Elemente der beiden Disziplinen Distributed Problem Solving und Multi-Agent Systems. DPS liefert einen

102.[Ginsberg 1989].

103.Beispielsweise in den planbasierten BDI-Architekturen [Pollack 1992], [Rao, Georgeff 1991b] (siehe Abschnitt 3.6 auf Seite 79).

Beitrag zum verteilten, kooperativen Problemlösen durch Aufgabenzerlegung, -delegation und Lösungssynthese, während MAS Ansätze bereitstellt, die die Dynamik eines nicht zentral designten Agentensystems verstehen und kontrollieren helfen. Die im Rahmen dieser Arbeit betrachteten diensterbringenden Agenten sind kooperativ, d.h. sie bieten ihre Leistungen „bereitwillig“ und nicht in Konkurrenz an. Um komplexe, kombinierte Dienste anbieten zu können wird ein hohes Maß an Interaktivität und Koordination benötigt. Einfache Dienste können mittels Sprechaktkommunikation angefragt und angeboten werden. Für die Zusammenstellung, Aushandlung und Erbringung komplexerer Services hingegen sind Interaktionsprotokolle notwendig. Die Beschreibungssprache für Dienste muß modular und deklarativ sein: Modularität unterstützt den Aufbau komplexer Dienste mit Hilfe einfacherer Subdienste, während Deklarativität das Verhandeln, Anbieten und Nutzen von Diensten erleichtert. Aufgrund der Langlebigkeit ist auch Lernfähigkeit sinnvoll.

KAPITEL 3 Agentenarchitekturen

Im vorherigen Kapitel wurden Fähigkeiten von Agenten untersucht und Methoden zu deren Realisierung vorgestellt. Die eingangs in der Klassifikation vorgestellten Agententypen vereinigen jeweils spezifische Bündel dieser Merkmale. Soweit es sich hierbei um generische Fähigkeiten handelt, ist es wenig sinnvoll, sie zum Gegenstand der Agentenprogrammierung zu machen. Vielmehr sollte eine „Agentenprogrammiersprache“ oder „Agentenenwicklungsumgebung“ derartige „agentische“ Eigenschaften automatisch bereitstellen, um so den Programmieraufwand zu verringern. Diesem Zweck dienen die sogenannten Agentenarchitekturen, die in diesem Kapitel behandelt werden. Eine Agentenarchitektur ist laut Maes[1] eine Methodologie für die Programmierung von Agenten:

> *„It specifies how ... the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of an agent determine the actions ... and future internal state of the agent. An architecture encompases techniques and algoithms that support this methodology.“*

1. [Maes 1991].

Eine Agentenarchitektur beschreibt demnach den strukturellen Aufbau eines Agenten durch eine Menge von Komponenten und definiert den Informationsfluß zwischen diesen. In den Komponenten sind die im vorigen Kapitel behandelten generischen Fähigkeiten zu finden. Durch die vorgegebene Informationsspeicherung und -verarbeitung ist gleichzeitig die Programmierschnittstelle festgelegt, hierbei sind die von der Architektur bereitgestellten und genutzten Wissensrepräsentationsformen zu nutzen.

Mit der strukturellen Beschreibung der Agentenarchitektur sind die Grundfunktionen des Agenten sowie der Datenfluß und damit auch die grundlegende Kontrolle festgelegt. Nun müssen die einzelnen Aktivitäten noch aufeinander abgestimmt werden. Diese interne Koordination beeinflußt, neben den externen Kooperationen, maßgeblich Effizienz und Qualität des Agentenverhaltens und wird im nächsten Abschnitt angesprochen. In der Praxis treten drei unterschiedliche Strukturvarianten auf: komponentenbasierte, geschichtete sowie strukturarme. Von diesen handeln die nachfolgenden Abschnitte. Zwei weitere Architekturtypen, die wichtige Beiträge zur Agentenprogrammierung liefern, werden anschließend behandelt. Es handelt sich hierbei um kognitive und Belief-Desire-Intention-Architekturen, die üblicherweise in Komponentenbauweise realisiert sind.

3.1 Informationsfluß und -verarbeitung

Das klassische, auf dem Von-Neumann-Rechnermodell beruhende Prinzip der seriellen Verarbeitung findet sich in vielen Agentenarchitekturen wieder: Sämtliche Abläufe finden in einer festgelegten Reihenfolge statt, die Komponenten oder Module eines Agenten sind in einem Prozeß zusammengefaßt. Es sprechen jedoch einige Gründe gegen eine rein serielle Informationsverarbeitung: Sequentielle Abläufe sind oft statisch und unflexibel, da sie auf einer zentralen Steuerung beruhen. Eine derartige globale Kontrollfunktion ist für komplexe Systeme, wie Agenten sie darstellen, oft unzweckmäßig. Für eine flexiblere Steuerung der internen Abläufe bieten sich verschiedene Mechanismen an:

- **Meta-Reasoning:** Das Dilemma, zwischen Handeln und Nachdenken zu entscheiden hat zwei extreme Ausprägungen: Wenn ein Agent zuviel räsonniert, z.B. plant oder lernt, kommt das Handeln zu kurz. Auf der anderen Seite führt blinder Aktionismus auch nur selten zum Ziel. Entsprechend sollte die Intensität des Räsonnierens den Gegebenheiten der Umgebung angepaßt sein. So sind in einer stark veränderlichen Welt komplexe kognitive Methoden relativ nutzlos. Auch die intuitive Idee, Wahrnehmung, Räsonnieren und Agieren parallel ablaufen zu lassen befreit den Entwickler nicht von der Kontrollproblematik: Die

einzelnen Tätigkeiten haben kombinatorische Komplexität, so daß die Verteilung der Aufgaben auf mehrere Prozesse keine grundsätzliche Lösung darstellt.

Die Ablaufsteuerung kann selbst Gegenstand des Räsonnierens sein. Dies ist der Fall, wenn der Agent als Meta-Interpreter in der Lage ist, seine eigenen Tätigkeiten zu beobachten und zu kontrollieren. Meta-Reasoning gestattet einem Agenten, seine Situation zu beurteilen, Sachzwänge und mögliche Freiheitsgrade zu identifizieren und die entsprechenden Maßnahmen dynamisch zu treffen. Meta-Reasoning ermöglicht facettenreiche Handlungspolitiken, angefangen von der fokussierten Bearbeitung durch bevorzugte Ressourcenzuteilung bis hin zur stärkeren Berücksichtigung der reaktiven Fähigkeiten im Falle vieler zu verarbeitender Events. Allerdings verschlingt die Einführung einer neuen Interpreterebene erhebliche Prozessorressourcen, so daß im ungünstigen Fall das Problem der Abstimmung zwischen Handeln und Räsonnieren nur eine Ebene nach oben verschoben wird.

- **Interrupts:** Mit Unterbrechungsmechanismen kann die Reaktivität und Effizienz eines Agenten gesteigert werden. Externe Ereignisse lösen einen Interrupt aus und triggern eine fest verdrahtete Verarbeitungsroutine im Agenten. Im Unterschied zum Polling-Ansatz, bei dem der Agent zyklisch Ereignisse abfragt, nehmen hier die Ereignisse eine aktive Rolle ein, sie kontrollieren den Agenten. Diese Verarbeitungsweise hat den Vorteil, daß der Ablauf eines Agenten in Form einer einfachen *sense-reason-act*-Schleife statisch modelliert werden kann.

  Da Interrupts asynchron auftreten, muß ein Agent in der Lage sein, seinen aktuellen Zustand effizient zu speichern, damit er nach der Interruptbehandlung schnell wieder den aktuellen Kontext wiederherstellen kann. Treten Interrupts in großer Zahl auf, besteht die Gefahr, daß der Agent quasi nur noch von externen Ereignissen kontrolliert wird und die Effizienz stark herabgesetzt wird. Priorisierungen von Ereignissen oder Filterungsmechanismen im Agenten können hier Abhilfe schaffen.

- **Asynchrone Verarbeitung** ist eine schwache Form der Parallelität. Man findet sie häufig in modularen Architekturen, in denen eine Entkopplung der Verarbeitungsmodule erreicht werden soll. Interaktionen zwischen den Modulen finden mittels Nachrichtenaustausch oder Anfragen statt. Jede Komponente läuft als eigenständiger Prozeß und ist für die Pufferung eingehender Informationen mit einer Queue ausgestattet.

  Durch die Nebenläufigkeit asynchron arbeitender Systeme und die damit verknüpfte nichtdeterministische Arbeitsweise treten jedoch Kontrollprobleme auf: Das Verhalten derartiger Systeme ist nur schwer berechenbar und oft auch im Nachhinein nicht nachvollziehbar.

- **Produktionssysteme** realisieren eine ereignisorientierte Informationsverarbeitung. Das Verhalten eines Agenten wird durch eine Menge von Regeln beschrieben, die durch Erfüllung ihrer jeweiligen Vorbedingungen feuern und entweder

Handlungen auslösen oder den mentalen Zustand verändern, woraufhin weitere Regeln zur Ausführung kommen können

Jedoch bieten Produktionssysteme kaum Strukturierungsmöglichkeiten, so daß komplexe Systeme durch große Regelbasen schwer zu erstellen und zu verwalten sind. Auch erschwert sich die Kontrolle und das Verständnis der Abläufe in Regelsystemen mit wachsender Zahl parallel feuernder Produktionen (Schneeballeffekt). Ansatzpunkte zu stärkerer Kontrolle und Strukturierung bieten Metaregeln. Beispiele für Metaregeln stellen Auswahlregeln dar, die bei Anwendbarkeit mehrerer Regeln entscheiden, welche zur Ausführung kommt oder Lernregeln, die Produktionen löschen, verändern und ggfs. hinzufügen.

Eine weitere Maßnahme in Richtung einer effektiven Informationsverarbeitung liegt in selektiver Wahrnehmung. Viele Agentenarchitekturen verfügen über eine Kommunikationskomponente, über die die Wahrnehmung abgewickelt wird. In diese Eintrittspforte werden üblicherweise Filter eingebaut, die irrelevante oder zu komplexe Informationen herausfiltern oder verdichten. Auf diese Weise gelangen weniger Informationen in den Agenten, wodurch sich sich der Aufwand zur Verwaltung des mentalen Zustands verringert.

3.2 Komponentenarchitekturen

Wenn unterschiedliche funktionale Komponenten vorliegen, spricht man von einer modularen Struktur. Modulare Zerlegung ist eine im Software Engineering weit verbreitete Methode des Systemdesigns. Bei einer Komponentenarchitektur realisiert jedes Modul eine Funktionalität und arbeitet typischerweise mit einem eigenen Datenmodell. Charakteristisch für derart strukturierte Agenten ist die Anwesenheit einer Kommunikationskomponente, über die der gesamte Nachrichtenverkehr abgewickelt wird. Häufig anzutreffen sind auch ein Reasoning-Modul, eine Komponente zur Steuerung und Ausführung der lokalen Fähigkeiten und eine Wissensbasis, die ein einheitliches Datenmodell für alle Komponenten bietet.

Neben der Herausstellung unterschiedlicher Funktionalitäten wird bei diesem horizontalen Design auch der Daten- oder Kontrollfluß mit modelliert. Alle Inputliefernden, d.h. vorgeschalteten Komponenten stellen Daten im vorgeschriebenen Format bereit. Die Daten werden entweder in eine komponenteneigene Wissensbasis eingetragen oder als Events in eine vorgesehene Input-Queue geschrieben. Derart als nebenläufiges Objekt gestaltet, kann jede Komponente in einem eigenen Prozeß laufen und durch die Pufferung einkommender Daten auch simultan von mehreren vorgeschalteten Komponenten bedient werden. Komponenten, die diese Prozeßeigenständigkeit nicht besitzen, müssen dagegen von außen direkt und synchron über ihre Funktionsschnittstelle aktiviert werden.

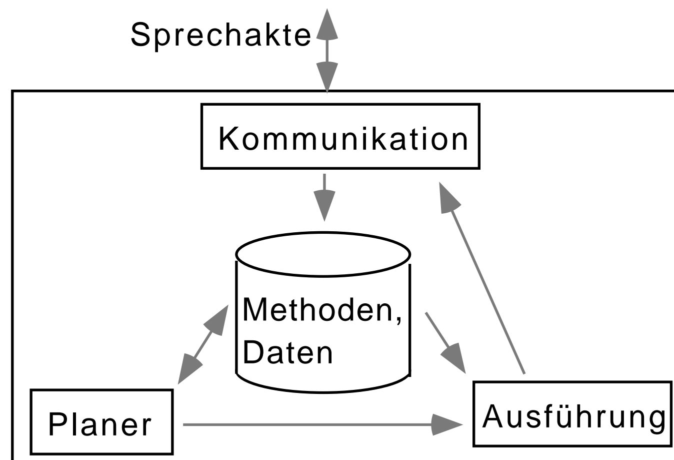

Abbildung 3-1: Prinzipieller Aufbau einer Komponentenarchitektur

Verschiedene komponentenbasierte Agentenarchitekturen zu vergleichen fällt schwer. Unterscheidungsmerkmale liegen sowohl in Zahl und Funktion der verwendeten Module wie auch in der auf dieser Ebene der Beschreibung nicht sichtbaren Art der Informationsspeicherung und -verarbeitung. Üblicherweise werden Agenten mit einer Entscheidungs- oder Kognitionskomponente als deliberative Agenten bezeichnet. Unterklassen bilden planende und lernende Architekturen, die entsprechend mit Plan- bzw. Lernmodulen ausgestattet sind. Fehlen kognitive Module, handelt es sich um eine reaktive Agentenarchitektur.

3.3 Geschichtete Architekturen

Altenativ zur modulorientierten Sicht kann ein Agent auch in unterschiedliche Ebenen der Wissensverarbeitung hierarchisch zerlegt werden. Ein Agent besteht dann aus einer Menge übereinanderliegender Schichten (Layers), die, mit zunehmender Höhe, immer komplexere Aufgaben übernehmen[2]. Gemäß dem Modell der verschiedenen Wissensarten bietet sich eine Unterteilung in die vier Abstraktionsschichten *physical level, symbol level, knowledge level* und *social level* an[3]. Auf jeder Ebene befinden sich Methoden zur Verarbeitung des dort verwalteten Wissens.

2. Eine die Verwendung hierarchischer Architekturen stützende These stammt von [Simon 1962], der hierarchische Dekompositionen für die Konstruktion jeglicher komplexer Gebilde vorschlug.

3. Die ersten drei Schichten nach [Newell 1982], die soziale Ebene gemäß [Jennings, Campos 1997].

Mit zunehmender Abstraktion werden die Verarbeitungsmethoden mächtiger, aber auch komplexer: Findet auf der physikalischen Schicht ausschließlich sensorischer Austausch statt, so wird auf der darüberliegenden Ebene bereits Wissen repräsentiert und verarbeitet. Auf dem knowledge level ist die Rationalität angesiedelt; hier werden Ziele verfolgt. Schließlich finden sich auf der sozialen Ebene Mechanismen für Kooperationen, soziale Normen usw.

Der Informationsfluß geht von unten nach oben. Wahrnehmungen gelangen über die physikalische Ebene in den Agenten. Jede Ebene funktioniert dabei als Filter, indem sie alle diejenigen Informationen verarbeitet, für die sie konzipiert ist und die nicht verarbeitbaren an die nächst höhere Schicht weiterleitet. Zum Beispiel wird übermitteltes Wissen von der symbolischen Ebene abgespeichert, während Sprechakte, die Bestandteil eines Kooperationsprotokolls sind, bis in die oberste Ebene durchgereicht werden. Entsprechend besitzt jede Schicht ihre eigene Wissensbasis mit assoziierten wissensverarbeitenden Methoden.

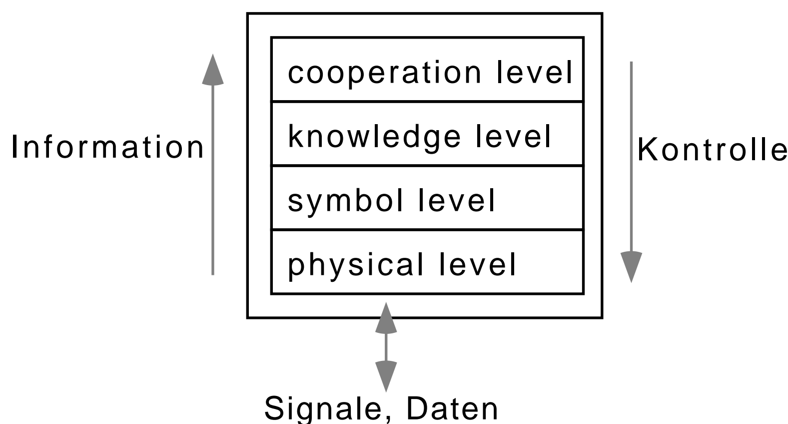

Abbildung 3-2: Prinzipieller Aufbau einer Schichtenarchitektur

In entgegengesetzter Richtung zum Informationsfluß findet die Kontrolle statt, weil jede Schicht zur Realisierung der Konsequenzen ihrer Wissensverarbeitungsmechanismen Fähigkeiten der darunterliegenden Ebenen benötigt. So arbeitet der knowledge-level direkt auf den Repräsentationen der Symbolebene, während für das Aussenden eines Sprechakts aus einem Kooperationsprotokoll heraus sowohl die symbolische Ebene zur Repräsentation des Nachrichteninhalts wie auch die physikaliche Schicht für das Sendeprotokoll benötigt wird.

Die Strukturierung einer Ebene kann durch funktionelle Dekomposition, wie im vorigen Abschnitt beschrieben, oder oder durch Komposition kleiner Einheiten erfolgen. Im ersten Fall entspricht eine Schichtenarchitektur praktisch einer Agenten-

hierarchie, wobei jede Schicht ein „Agent“ ist, der in direkter Abhängigkeit zu den angrenzenden Schichten steht. Bei einer Organisation durch Komposition wird jede Ebene durch eine Menge kleiner, relativ unabhängiger Subsysteme beschrieben. Subsysteme mit gleichen oder ähnlichen Eigenschaften können als signalverarbeitende „Kompetenzmodule“ miteinander konkurrieren (Abschnitt 3.4). Findet sich für ein Signal kein kompetenter Verarbeiter, wird es zur nächsten Schicht weitergeleitet.

Beispiele für Schichtenarchitekturen sind InteRRaP und TouringMachines. InteRRaP[4] besteht aus 3 Ebenen, wobei jede Schicht eine eigene Wissensbasis und diverse Kontrollkomponenten besitzt, also komponentenbasiert aufgebaut ist. Die untere Ebene ist für die Ausführung von Aktionen, Kommunikation und Wahrnehmung verantwortlich; die mittlere Schicht implementiert und kontrolliert die reaktiven Basisaktivitäten des Agenten mittels sogenannter „patterns of behaviour“, die Planelementen ähneln. In der oberen Schicht ist eine Planbibliothek angesiedelt, außerdem werden hier Kooperationen in Form von gemeinsamen Plänen durchgeführt.

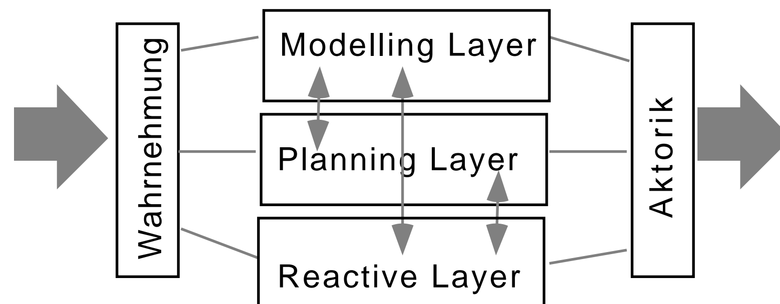

Abbildung 3-3: TouringMachines-Agentenarchitektur

Im Unterschied zu InteRRaP ist jede Schicht der TouringMachines[5]-Architektur mit den Sensoren und Effektoren direkt verknüpft. Die untere Schicht besteht aus Situations-Aktions-Regeln. Eine darüberliegende Planungsschicht verwaltet die Intentionalität des Agenten in Form von Zielen. Es wird eine hierarchische, partielle Planung unter Verwendung einer Bibliothek von Teilplänen durchgeführt, wobei Planerstellung und -ausführung miteinander verwoben sind. In der obersten Schicht, dem Modelling Layer, wird die Umgebung des Agenten symbolisch repräsentiert.

4. Integration of Reactive behaviour and Rational Planning [Müller, Pischel 1993].

5. [Ferguson 1992b], [Ferguson 1994].

Hier findet ein Monitoring der anderen Komponenten statt, werden Erklärungen via abduktiver Inferenz generiert und Vorhersagen durch zeitliches Reasoning getroffen. Damit können Zielkonflikte mit anderen Agenten aufgedeckt und aufgelöst werden. Durch Zensorregeln und Unterdrückungsregeln wird die für Schichtenarchitekturen typische Kontrolle von oben nach unten realisiert.

Weitere Beispiele für Schichtenarchitekturen existieren. GLAIR[6] ist ein dreischichtiges Architekturmodell mit wissensbasiertem Knowledge-Level mit Reasoning- und Lernfähigkeiten und der Möglichkeit des Wissentransfers zwischen den Ebenen. In GRATE[7] kontrolliert die Kooperationsschicht durch ein Joint-Responsibilities-Modell die Aktionen des Domain-Levels; ein Control-Layer stimmt die Aktionen mit anderen Agenten ab.

3.4 Reaktive Architekturen

Geschichtete und komponentenartige Architekturen haben gemeinsam, daß sie Agenten als Informationsverarbeiter modellieren. Für beide Ansätze ist die interne Repräsentation eines Welt- oder Wissensmodells charakteristisch. Als Antwort auf die oft problembehaftete Wissensrepräsentation[8] wurden rein reaktive Agentenarchitekturen entwickelt, die ohne ein Modell der Umwelt funktionieren, da die Agenten direkt in der Welt verankert sind[9].

Die Wahrnehmung der Umwelt erfolgt über Reize oder Signale empfangende Sensoren. Die Verarbeitung findet üblicherweise durch einen einheitlichen, einfachen Mechanismus statt, der sämtliche Wahrnehmungen direkt, d.h. ohne eine Klassifikation oder symbolische Repräsentation vorzunehmen, verarbeitet. Möglichkeiten für derart uniforme Mechanismen auf Signalebene bieten Zustandsautomaten, stimulus-response-artige Produktionssysteme oder konkurrierende Kompetenzmodule[10].

---

6. Grounded Layer Architecture with Integrated Reasoning [Hexmoor et al 1993a], [Hexmoor et al 1993b].

7. [Jennings 1992].

8. Hier sind das symbol grounding-Problem [Harnad 1990] und die Schwierigkeiten bei der Wissensverwaltung (schnell ändernde Umwelt, widersprüchliche oder unsichere Informationen) zu nennen.

9. Man spricht hier auch von „situated agents“ oder „embodied agents“. Befürworter reaktiver Architekturen zur Erreichung intelligenten Verhaltens sind Dreyfuß [Dreyfus 1979], Minsky [Minsky 1985] und Brooks [Brooks 1991a], [Brooks 1986].

10. [Maes 1994a]. [Brooks 1986] benutzt stattdessen den Begriff „behaviours“.

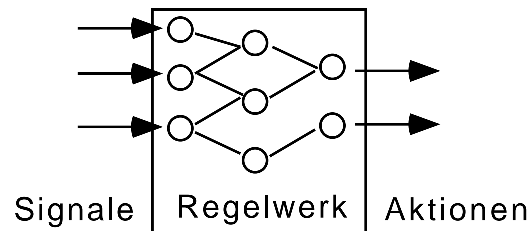

Abbildung 3-4: Strukturarmer Aufbau einer Agentenarchitektur

Das Einsatzgebiet für strukturarme Agenten liegt im Bereich der Robotik, wo harte Echtzeitanforderungen in Verbindung mit einer unbekannten, schwer vorhersehbaren Umwelt Wissensrepräsentation und den Einsatz kognitiver Methoden von vornherein aussichtslos erscheinen läßt. Prominentester Vertreter dieser Art von Architekturen ist die Subsumption-Architektur[11], die aus einem fest verschalteten Netzwerk endlicher Automaten besteht. Die Automaten verarbeiten einfache Signale, die entweder direkt von Sensoren stammen oder Ausgangssignale anderer Automaten sind. Überschreitet das anliegende Signal einen Schwellenwert, wird das Verhaltensmuster des Automaten aktiviert. Resultat ist wiederum ein Signal, das an anliegende Zustandsautomaten oder Handlungen auslösende Actuators geschickt wird. Diese Art der Informationsverarbeitung ähnelt stark den neuronalen Netzen. Ähnlich wie dort ist außerdem eine Schichtenstrukturierung gegeben, wobei höhere Schichten tiefer liegende Ebenen durch Unterdrückungsregeln steuern können. Das Beispiel eines navigierenden Roboters zeigt die Modellierung und Interaktionen der einzelnen Schichten: In einer dreischichtigen Architektur ist die untere Schicht auf Vermeidung von Kontakt mit Objekten ausgerichtet, während die oberste Ebene den Agenten in eine bestimmte Richtung ohne Berücksichtigung von Hindernissen steuert. Mit der mittleren Schicht führt der Agent zufällige Wanderungen durch. So gelangt der Roboter durch Kombination aller Schichten ohne explizite Wegeplanung unter Umgehung von Hindernissen zu jedem beliebigen Ziel.

Von Matriac[12] wurde die Subsumption-Architektur durch Hinzunahme weiterer repräsentationsbasierter Verhaltensebenen in Richtung Lernen und Planen erweitert. Ein weiterer Abkömmling der Subsumption-Architektur, jedoch mit einer symbolverarbeitenden obersten Ebene, ist Atlantis[13]. Maes[14] beschreibt eine ähnliche

11. [Brooks 1986].

12. [Mataric 1991], [Mataric 1992].

13. A Three-Layer Architecture for Navigating Through Intricate Situations [Gat 1992], [Gat 1993].

Architektur, die aus zusammengeschalteten „Kompetenzmodulen“ besteht. Hier wird jedes Modul durch Strips-ähnliche Vor- und Nachbedingungen beschrieben. Die Module stehen in Konkurrenz zueinander, zur Auswahl im Falle mehrerer in Frage kommender Module wird ein Aktivierungslevel als Relevanzfaktor herangezogen. Zusätzliche Intelligenz durch Lernen der Regelvorbedingungen wurde in die Architektur integriert[15].

3.5 Kognitive Architekturen

Kognitive Architekturen zielen auf allgemein intelligentes Verhalten ab. Sie werden im Sinne der KI als wissensbasierte, informationsverarbeitende Systeme realisiert und sind ihrer Struktur nach den komponentenbasierten Architekturen zuzuordnen. Wie bei allen wissensbasierten Systemen ist Wissen von dessen Verarbeitung getrennt. Dies kommt besonders bei der Verwendung des Zielbegriffs zum Vorschein, der in nahezu jeder kognitiven Architektur von zentraler Bedeutung ist.

In deliberative Agentenarchitekturen sind oftmals „weiche“ Interruptmechanismen eingebaut. Während wahrgenommene Informationen komplexen Bearbeitungsfunktionen zum Update des Weltmodells und zum Erstellen von Handlungsplänen zugeführt werden, führen andere Ereignisse direkt zur Aktivierung von Handlungsmustern. So erlauben die ursprünglich als monolithische Einagentensysteme konzipieren Architekturen teilweise auch den Einsatz in dynamischen Umgebungen.

SOAR

SOAR[16] basiert auf den frühen KI-Systemen GPS und OPS5. Intelligenz wird gemäß des Rationalitätsprinzips als das optimale Erreichen von Zielen verstanden. In SOAR findet zielorientiertes Problemlösen als heuristische Suche in Problemräumen statt. Im Prinzip wird dabei klassische Handlungsplanung, wie in Abschnitt 2.4.4 beschrieben, durchgeführt: Die Suche erfolgt durch sukzessives Anwenden von Operatoren, bis der Zielzustand erreicht ist. In Erweiterung klassischer Planungssysteme ist die Problemraumsuche in einen komplexen Entscheidungszyklus eingebaut. Zur Repräsentation von Wissen stellt SOAR zwei Konzepte in Form eines Langzeit- und eines Kurzzeitgedächtnisses bereit. Das Langzeitgedächtnis spei-

14. [Maes 1990].

15. [Maes, Brooks 1990] nutzten eine Form des Reinforcement Learnings (siehe Seite 64).

16. State, Operator, and Result [Laird, et al. 1987]. Die hier gewonnenen Erkenntnisse kumulierte [Newell 1990] später zu einer integrierten Theorie der menschlichen Kognition.

chert operatives Wissen zum Aufbau der Problemräume und Kontrollwissen zur Steuerung der Suchprozesse einheitlich in Form von Produktionen. Kurzzeitliches Wissen wird durch Attribut-Wert-Listen dargestellt, die zu Objekten zusammengefaßt werden. Das Kurzzeitgedächtnis bildet den Arbeitsspeicher, hier findet die gesamte Informationsverarbeitung statt. Aus dem einheitlichen Repräsentations- und Zugriffsmechanismus sowie der Möglichkeit, den Arbeitsspeicher in Bereiche zu strukturieren ergibt sich eine starke Ähnlichkeit zu Blackboards. Die offene Gestaltung des Arbeitsspeichers erlaubt das Hinzufügen beliebiger Module[17], die diesen Speicher oder ein zugewiesenes Segment zum Informationsaustausch und zur Koordinierung benutzen können.

Die Informationsverarbeitung findet in zwei Phasen statt. In der ersten Phase der Wissenssuche feuern anwendbare Produktionen des Langzeitgedächtnisses, welche auf dem Arbeitsspeicher operieren. Einerseits führt dieser Prozeß zur Generierung neuer Objekte, was wiederum andere Produktionen zur Anwendung kommen lassen kann, andererseits entstehen und ändern sich Präferenzen, die in der zweiten Phase zur Steuerung für die weitere Verwendung von Zielen, Problemräumen, Zuständen und Operatoren benutzt werden.

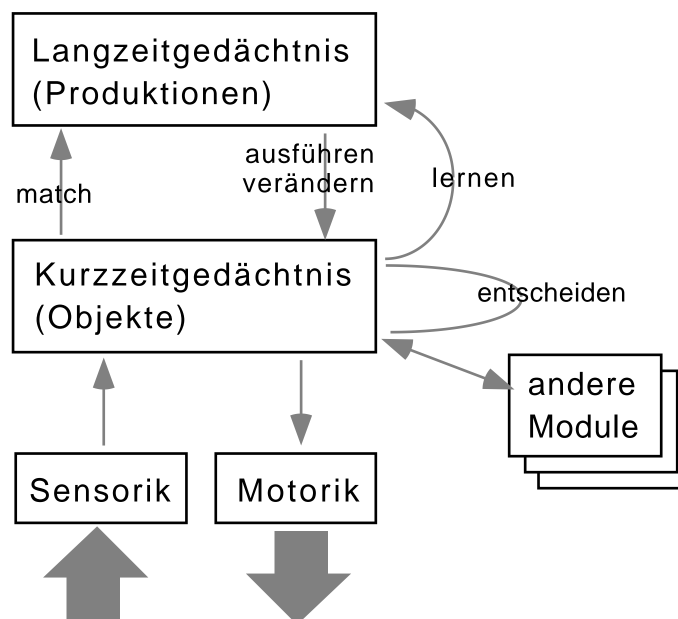

Abbildung 3-5: SOAR-Agentenarchitektur[18]

17. Neben dem Langzeitgedächtnis und den Modulen für Sensorik und Aktorik.

In der zweiten Phase wählt die Entscheidungsprozedur anhand des aktuellen Wissens im Kurzzeitgedächtnis unter Zuhilfenahme vorhandener Präferenzen einen Operator und wendet diesen auf den zugehörigen Problemraum an. Durch sukzessive Anwendung von Operatoren wird entweder irgendwann das Ziel erreicht, oder man landet in einer Sackgasse. In diesem Fall wird ein Unterziel generiert, dessen Aufgabe es ist, den Suchprozeß aus der Sackgasse zu lenken. Kann eine Sackgasse auf diese Weise nicht aufgelöst werden, kommen problemraumunabhängige Mechanismen wie Backtracking zur Anwendung. Zur Vermeidung von Sackgassen greift ein Chunking-Lernmechanismus, der jedesmal aktiviert wird, wenn ein erfolgreicher Weg aus einer Sackgasse gefunden wurde. Es wird eine Produktionsregel generiert, deren Vorbedingung den „Eingang“ der Sackgasse, d.h. die Ausgangssituation und deren Aktionsteil das Wissen zur Auflösung der Sackgasse beschreibt. Trifft ein Agent später auf eine ähnliche Situation, feuert die erlernte Regel und der Weg in die Sackgasse wird vermieden.

Prodigy

Einen anderen Ansatz zur Verknüpfung von Planen mit Lernen beschreitet Prodigy[19]. Herzstück dieser Architektur ist ein nichtlinearer Planer, der den Zustandsraum durch rückwärtsverkettete Suche aufspannt. Die Suchprozedur ist dabei in der Lage, mehrere Zielzustände in einem Plan zu integrieren. Planoperatoren basieren auf STRIPS, erweitern dessen Formalismus jedoch um Merkmale wie veroderte und negierte Vorbedingungen in Verbindung mit Quantoren und bedingte Effekte. Die Operatorauswahl wurd durch Kontrollregeln heuristisch gesteuert: Selektions- und Ausschlußregeln verkleinern die Menge der anwendbaren Operatoren und schränken so den Suchraum ein; Präferenzregeln definieren eine partielle Ordnung auf den Operatoren.

Während die Kontrollregeln den Planungsprozeß aktiv unterstützen, dienen die in Prodigy integrierten Lernkonzepte zur Effizienzsteigerung a posteriori. Automatisch wird Kontrollwissen zur Steigerung der Effizienz des Planungsprozesses mit Hilfe von *explanation-based learning* (siehe Abschnitt 2.4.5 auf Seite 64) generiert. Zum andern existiert eine Reihe von Lernmodulen, die auf Assistenz eines Experten angewiesen sind und zwecks Steigerung der Planqualität sowie zur Akquise von Domänenwissen zum Einsatz kommen.

---

18. Nach [Newell, et al. 1989], S.111.

19. [Carbonell, et al. 1991], [Veloso, et al. 1995].

Theo

Theo[20] integriert klassisches Problemlösen mit Lernen und Selbstreflektion. Das Wissen eines Agenten wird einheitlich in Frames verwaltet. Ein Slot, der mit einem konkreten Wert belegt ist, wird als Belief bezeichnet und kann als Teil des aktuellen Weltmodells angesehen werden. Slots ohne Werte stellen Probleme dar, die die Aktivierung von Problemlösungsmethoden zur Berechnung des Wertes nach sich ziehen. Hierfür steht eine Menge von Methoden bereit, die die Verwendung von Defaultwerten, das Auswerten von Funktionen, die Übernahme von bekannten Werten aus der Framehierarchie und *explanation-based learning* umfassen. Ein Lernmodul, das auf statistischer Inferenz beruht, sortiert die Methoden, so daß fehlende Werte in kürzester Zeit gefunden werden können. Das Paradigma der einheitlichen Repräsentation wird soweit getrieben, daß auch Metawissen über Frames in Bezug auf die Beobachtung der Effizienz, Schwierigkeit und Erfolg der Frameverarbeitung durch Frames dargestellt wird. Stimulus-Response-Regeln bilden den reaktiven Part eines Theo-Agenten. Mit zunehmender Zahl dieser Regeln steigt die Reaktivität, so daß hierin ein Mittel zur Skalierung gegeben ist.

3.6 BDI-Architekturen

Agenten, die mit mentalistischen Notationen wie in Abschnitt 2.4.2 beschrieben arbeiten, werden unter dem Sammelbegriff BDI-Architekturen geführt. Dadurch, daß diese Architekturen ein Verarbeitungsmodell für modale Logiken realisieren, ist eine enge Verzahnung von Architektur und Programmiersprache gegeben. In der Praxis sind BDI-Architekturen stark reaktiver Natur.

Die „klassische“ BDI-Agentenarchitektur nach Rao und Georgeff[21] ist in Abbildung 3-6 zu sehen. Ein Agent besteht aus vier Komponenten mit je einer Wissensbasis plus einem Interpreter (Reasoner), der Kontrollfunktionalität hat. Die Belief-Wissensbasis enthält das Weltmodell bzw. -wissen des Agenten, das durch Wahrnehmungen aktualisiert wird. In der Desire-Komponente werden die Ziele des Agenten verwaltet. Neue Ziele oder das Verwerfen aktueller Ziele entstehen als Konsequenz veränderter Beliefs. Die Planbibliothek enthält die Basisfähigkeiten in Form von einfachen Aktionen oder abstrakten Plänen. Das Intention-Modul verknüpft die - mitunter widersprüchlichen - Desires mit auszuführenden Aktionen und Plänen. Eine Intention ist somit die Selbstverpflichtung, eine Aktion aufgrund eines oder mehrerer Desires durchzuführen.

20. [Mitchell, et al. 1991].

21. [Rao, Georgeff 1991b].

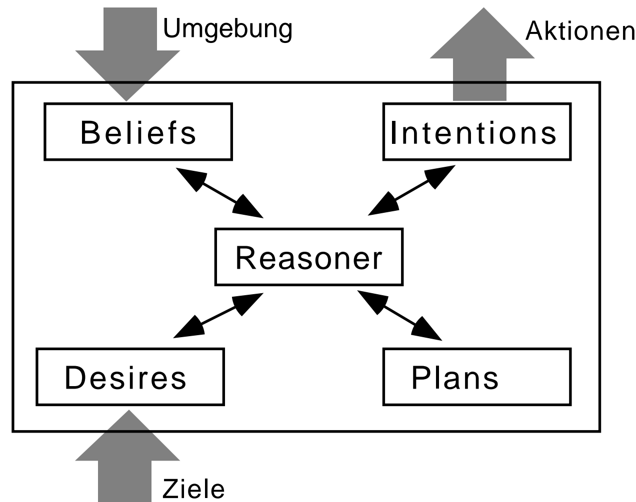

Abbildung 3-6: BDI-Architektur nach Rao & Georgeff

Der Interpreter besteht aus einer Schleife, die zyklisch zwischen Wahrnehmung mit Aktualisierung der Belief-Datenbasis, Generierung von Desires aus den Beliefs, Generierung von Intentions aus den aktuellen Desires und Ausführung der eingeplanten Aktionen wechselt. Damit während der Informationsverarbeitung keine Informationen über Änderungen der Umwelt verloren gehen besitzt ein BDI-Agent eine Event-Queue, in der Ereignisse automatisch eingetragen werden. Üblicherweise sind die verwendeten Modallogiken um Zeitrepräsentationen erweitert, so daß Aktionen erst gescheduled werden und fristgerecht zur Ausführung kommen.

```
1 initialize-state();
2 repeat
3   options = option-generator( event-queue );
4   selected-options = deliberate( options );
5   update-intentions( selected-options );
6   execute();
7   get-new-external-events();
8   drop-successful-attitudes();
9   drop-impossible-attitudes();
10 end repeat
```

Abbildung 3-7: BDI-Verarbeitungszyklus für BDI-Agenten[22]

22. Nach [Rao, Georgeff 1992].

Der erste Ansatz war der von Shoham vorgestellte Interpreter der Agent-0-Sprache (siehe Abschnitt 4.1). Weiterentwickelte, mittlerweise in der Praxis eingesetzte Abkömmlinge existieren. Es handelt sich hierbei um AgentBuilder[23], eine Architektur, die auch Sprechaktkommunikation mit einschließt, sowie die am AAIS in Australien realisierten BDI-Architekturen PRS und dMARS[24].

IRMA[25] ist eine Architektur mit einem Planer als zentrale Komponente. Eine Reihe von Mechanismen trimmen den Planungsprozeß zum Einsatz in komplexen, dynamischen Umgebungen. Aktionsoptionen werden durch Zielabgleich (*means-end-reasoning*) und aufgrund äußerer Einflüsse generiert. Sie durchlaufen einen Filterungsprozeß zwecks Kompatibilitätsprüfung mit bereits aktiven Plänen, bevor eine *Deliberation*-Komponente eine Aktion als Intention auswählt. Zusätzliche Kontrolle bietet ein Filter-Überschreibmechanismus, mit dem die Sensitivität des Agenten gegenüber Änderungen in der Umgebung eingestellt werden kann. Mit Hilfe dieses Filters können Optionen, die bestimmte Charakteristika der Umgebung matchen, ohne Berücksichtigung möglicher Kompatibilitätsprobleme der *Deliberation*-Komponente zugeführt werden.

3.7 Zusammenfassung und Bewertung

Die Architektur beschreibt Aufbau und Funktionsweise eines einzelnen Agenten. Von ihr bereitgestellte generische Funktionalitäten im Bereich der Informationsfilterung, -speicherung und -verarbeitung bilden die „architectural commitments“, welche die Verwendungsmöglichkeiten der Architektur einschränken. Entsprechend den vielen möglichen Anforderungen in bezug auf die Aufgabenstellungen und Ausprägungen der Einsatzumgebung existieren zahlreiche Architekturen mit unterschiedlichen Schwerpunkten.

Kognitive Architekturen besitzen als besonderes Feature Reasoning-Mechanismen, sind aber mangels Kommunikationsfähigkeit und Reaktivität nicht für Mehragentensysteme konzipiert. Auf ebendiese Echtzeitanforderungen in unsteten Umgebungen zielen die strukturarmen Architekturen ab, die in hybrider (mehrschichtiger) Form sogar mit kognitiven Fähigkeiten ausgestattet werden können. Jedoch liefern diese Ansätze keine befriedigende Antwort auf die Frage, was agentenorientiertes Programmieren ist. Die Gründe liegen in der fehlenden internen Struktur und dem Verzicht auf symbolische Repräsentation und Interagentenkommunikation.

23. [Reticular 1998].

24. [Georgeff, Lansky 1987], [d´Inverno, et al. 1997], siehe auch Abschnitt 4.2.

25. Intelligent Resource-Bounded Machine Architecture [Bratman, et al. 1988], [Pollack, Ringuette 1990].

Dieses Problem stellt sich auch für Schichtenarchitekturen im allgemeinen: Sie vereinigen zwar die Vorteile beider Welten, jedoch stellen sie eher lose Konzepte dar; generische Fähigkeiten, ein konkretes Verarbeitungsmodell oder Kontrollfluß sind kaum vorhanden, und die Programmierschnittstelle ist aufgrund des sowohl vertikalen wie auch horizontalen Informationsflusses umfangreich und - im Falle redundanter, konkurrierender Informationsverarbeitungseinheiten - relativ intransparent. Hingegen sind in modularen Architekturen die generischen Fähigkeiten funktional getrennt und der Informationsfluß verläuft linear durch die einzelnen Komponenten. Diese transparente, deterministische Art der Verarbeitung kann die Programmierung von spezifischen Fähigkeiten bzw. Ausgestaltung von Komponenten erleichtern, denn die Programmierschnittstelle ist übersichtlicher. Die Stärke von BDI-Architekturen liegt in der Repräsentation des mentalen Zustands eines Agenten und der nachvollziehbaren und dennoch flexiblen Verarbeitung von abstrakten Zielen bis hin zu Aktionen. Eine logische Fundierung existiert jedoch in der Praxis nicht[26]. Außerdem vernachlässigen BDI-Modelle die Aspekte der Interaktion und bieten keine Lösungen für die Probleme der Wissensverwaltung und des Belief-Updates[27], was ihren Einsatz in kooperativen und deliberativen Problemfeldern erschwert.

Viele der untersuchten Architekturen besitzen den Status von Forschungsprototypen; sie interpretieren entweder alle Agenten innerhalb eines Prozesses und erlauben keine verteilte Ausführung, oder sie stellen nur ein loses Rahmenwerk bereit und bieten wenig Unterstützung zur Realisierung von Anwendungen. Zur ersten Gruppe zählen TouringMachines, IRMA, Agent-0, zur zweiten Gruppe Interrap und GLAIR. In der Praxis treten häufig Agenten auf, die direkt, d.h. ohne zugrundeliegende Architektur, implementiert sind. Hierbei handelt es sich zumeist um Einagentensysteme[28].

Eine Architektur für Dienste erbringende Agenten sollte auf einem zugleich deklarativen wie auch modularen Repräsentationsmechanismus beruhen, um Dienste in unterschiedlicher Komplexität beschreiben zu können. Das Wissen - und damit die Informationsverarbeitung - im Dienstkontext ist eher einfacher Natur und flach strukturiert, weshalb auf mächtige KI-Wissensrepräsentationsformalismen und Inferenzmechanismen verzichtet werden kann. Zur kooperativen Diensterbringung bedarf es einer Unterstützung sprechaktbasierter Interaktionsprotokolle. Weil ko-

26. Siehe dazu die Problematik der Axiomatisierungen in Abschnitt 2.4.2.

27. [Wagner 1997].

28. Hierzu zählen vor allen Dingen Agenten der Kategorien persönliche Assistenten und Web-basierte Informationsagenten (z.B. Letizia [Lieberman 1995] oder der Remembrance-Agent [Rhodes, Starner 1996]). Auch Julia [Foner 1997], ein Eliza-ähnliches Programm, wird als intelligenter Agent bezeichnet.

operative Diensterbringung durch gegenseitiges Nutzen bereitgestellter Funktionalitäten realisiert wird, steht Reaktivität im Vordergrund. Jedoch sollte die Architektur Basisfunktionalitäten für das lokale Aktivitätenmanagement bereitstellen, um entsprechend den Anforderungen aus Abschnitt 1.1 die Diensterbringung überwachen und im Fehlerfall gegebenenfalls unterbrechen zu können.

Durch die starke Ausrichtung auf Interaktivität und geringe Verwendung von Weltwissen unterscheidet sich eine Architektur für Dienste ebenso von kognitiven wie von BDI-Architekturen. Nutzen kann eine Dienstarchitektur hingegen aus der Reaktivität und dem einfachen Verarbeitungsmodell des BDI-Ansatzes ziehen, während kognitive Ansätze die deklarativen Wissensbeschreibungen und Lernkonzepte beisteuern können. Ein strukturarmer Aufbau kommt schon wegen der geringen Modularität der Programmierung nicht in Frage. Aufgrund der flachen Wissensmodellierung und -verarbeitung liegt daher ein komponentenbasierter Aufbau nahe.

KAPITEL 4 Agentenprogrammiersprachen

Dieses Kapitel widmet sich der Programmierung von Agenten und agentenbasierten Anwendungen. Mit der Verwendung des Begriffs „Agentenprogrammiersprache“ stellt sich die Frage nach der Abgrenzung zu anderen Programmiersprachen. Dasselbe gilt für den in der DAI häufig gebrauchten Terminus „agentenorientiertes Programmieren“ (AOP). Hier ist zu klären, inwieweit AOP eine eigene Methodologie beschreibt und wo die Unterschiede und Gemeinsamkeiten zu anderen Methodologien wie beispielsweise der objektorientierten liegen.

Bis heute ist nicht abschließend geklärt, was genau unter dem Begriff „agentenorientiertes Programmieren“ zu verstehen ist. Offensichtlich heben Techniken, wie sie im 2. Kapitel vorgestellt wurden das agentenorientierte Programmieren auf eine höhere Ebene als „klassisches“ Programmieren. Kommunikation und Koordination auf hohem Niveau und mentalistische Notationen zur Beschreibung interner Zustände stellen neuartige Herangehensweisen im Softwaredesign und der Programmierung dar. Allerdings sind derart mächtige Konzepte nur schwierig als Primitiven einer Programmiersprache zu realisieren, der große konzeptuelle Abstand zwischen der Semantik derartiger Konstrukte und der maschinellen Ausführungsebene ist nicht einfach zu überbrücken. Noch schwieriger gestaltet sich die Kombination der unterschiedlichen Konzepte.

An dieser Stelle kommen Agentenarchitekturen ins Spiel. Sie geben einen Rahmen für die interne Funktionsweise eines Agenten vor und stellen Basisfähigkeiten wie Wahrnehmung, Kommunikation und Reasoning bereit. Wie im letzten Kapitel

beschrieben wurde, legen die Repräsentations- und Verarbeitungsmechanismen der Architektur die Programmierschnittstelle fest. Der Entwickler kann auf bereitgestellte Funktionalität zurückgreifen und wird dadurch entlastet. Bietet eine Architektur beispielsweise eine Kommunikationsinfrastruktur, die das Versenden, Empfangen, Puffern und Parsieren von Sprechakten umfaßt, kann sich die Programmierung auf die Ebene der Nachrichtenverarbeitung und -semantik konzentrieren. Auf der anderen Seite schränken die auch „architectural commitments" genannten Struktur- und Verarbeitungsmerkmale einer Architektur die Freiheit des Programmierens ein.

Die meisten der existierenden Agentenarchitekturen bieten jedoch kein geschlossenes Konzept zur Agentenprogrammierung. So erfolgt die Implementierung im Falle kognitiver Architekturen durch den Aufbau einer Wissensbasis und der Formulierung von Kontrollwissen. Andere Architekturen wie InteRRaP bieten ein loses Gerüst von Mechanismen zur Gestaltung von Agenten an, analog zu Interfaces ohne konkrete Implementierung. Aus Sicht der Basisfähigkeiten eines Agenten ergeben sich drei Bereiche, die eine Agentenprogrammiersprache abzudecken hat: Interaktion, die eigentlichen Fähigkeiten und das dazwischengeschaltete Reasoning, das den „Leim" zwischen Wahrnehmen und Handeln bildet. Je nach dem Grad der Ausgestaltung dieser Basisfunktionalitäten in der Architektur kann die Programmierung in der Parametrierung, Anpassung, Erweiterung vorhandener oder gar vollständigen Realisierung nicht bereitgestellter Fähigkeiten liegen.

In den nächsten 4 Abschnitten werden die wichtigsten existierenden Beiträge zur agentenorientierten Programmierung vorgestellt. Hierbei handelt es sich um „ganzheitliche" Ansätze, die Antworten auf die oben gestellten Fragen nach dem Wesensinhalt von Agentenprogrammiersprachen und Methodologien geben. Anschließend werden im 5. Abschnitt Sprachen und Formalismen behandelt, die nur Teilbereiche der Agentenprogrammierung abdecken. Nach einer Diskussion der wichtigsten Aspekte des Testens und Debuggens von Agentensystemen in Abschnitt 6 werden die unterschiedlichen Ansätze zum agentenorientierten Programmieren kritisch betrachtet. Mit dem Resümee dieser Betrachtungen wird im 8. Abschnitt die Realisierung einer Entwicklungsplattform für den Dienstbereich begründet. Eine Zusammenfassung und Bewertung schließen dieses Kapitel ab.

4.1 Agent-0

Agent-0[1] und die Derivate PLACA, AgentBuilder, Concurrent MetateM und Agent-K[2] beruhen auf BDI-Konzepten (siehe Abschnitte 2.4.2 und 3.6) und verfol-

1. [Shoham 1990], [Shoham 1993].

gen einen agentenzentrierten Ansatz. Der Schwerpunkt bei der Modellierung konzentriert sich auf die Beschreibung des mentalen Zustands eines Agenten. Agentenkommunikation und andere Systemaspekte spielen dagegen eine untergeordnete Rolle. Shohams Verständnis von AOP beruht auf drei Säulen: einem logischen System zur Definition des mentalen Zustands, einer darauf aufgesetzten interpretierten Programmiersprache, und einem „Agentifizierung" genannten Prozeß der automatischen Transformation eines Agentenprogramms in ausführbare Systeme. Zur Beschreibung mentaler Zustände wird Prädikatenlogik genutzt, die um die Modaloperatoren Belief (BEL), Commitment (CMT) und Capability (CAN) und um eine Notation von Zeitpunkten erweitert wurde[3].

```
BEL( me, t1, BEL( you, t2, like( t3,me,you ))).
CAN( me, t1, open( t2, door )).
```

Beliefs, commitments und capabilities machen Aussagen über Zustände. Korrekt interpretiert drückt die obige CAN-Aussage aus, daß der Agent in der Lage ist sicherzustellen, daß die Tür zum gegebenen Zeitpunkt offen ist und nicht etwa, daß er die Fähigkeit besitzt, die Tür zu öffnen. Atomare Bestandteile der Programmiersprache sind Konstrukte für Aktionen, wobei zwischen lokalen („private") und kommunikativen unterschieden wird sowie mentale Beschreibungen in Form von beliefs und capabilities. Die Kommunikation wird vollständig durch die vier Sprechakte inform, request, unrequest und refrain abgedeckt. Terme können existentielle („?x") und universelle („?!x") Variablen enthalten.

```
%% bedingte Aktion
%% IF mentalCond THEN action
REQUEST( t1, you,
          IF      BEL( t2, fact )
          THEN    INFORM( t2, me, fact )).

%% Commitment-Regel
%% COMMIT(msgcond, mentalCond, agent, action)
COMMIT( REQUEST( ?t, ?agent, ?action ),
        BEL( ?!t, myfriend( ?a )),
        ?a,
        ?action ).
```

Darauf aufbauend lassen sich bedingte Aktionen ausdrücken. Obiges Beispiel zeigt die Verwendung eines IF-THEN-Konstrukts innerhalb eines request-Sprechakts. Wichtigstes Sprachkonstrukt von Agent-0 ist die Commitment-Regel: Sie verknüpft

2. PLACA: [Thomas 1994], AgentBuilder: [Reticular 1998], Concurrent MetateM: [Fisher 1994b], Agent-K: [Davies, Edwards 1994].

3. Die hier verwendete Syntax wurde aus Gründen der besseren Übersichtlichkeit vereinfacht.

Wahrnehmung und mentalen Zustand mit einem Handlungsmuster in Form einer Verpflichtung gegenüber einem anderen Agenten.

Ein Agent-0-Programm besteht aus einer Menge von initialen Beliefs, Capabilities und Commitment-Regeln. Capability-Beschreibungen bestehen aus dem Namen einer Aktion und einer mentalen Bedingung, die angibt, unter welchen Voraussetzungen die Aktion für den Agenten anwendbar ist. In Verbindung mit einem dem BDI-Modell angelehnten Interpreter (Abb. 4-1, vergleiche mit Abb. 3-6) ergibt sich folgender Ablauf:

1 Beim Belief-Update werden alle durch empfangene tell-Sprechakte eingegangenen neuen Informationen in die Belief-Wissensbasis eingetragen. Zu den hereingekommenen Informationen eventuell inkonsistente Beliefs werden aus der Datenbasis entfernt.

2 Beim Commitment-Update führen empfangene unrequest-Sprechakte zur Entfernung entsprechender Einträge aus der Commitment-Datenbasis. Refrain-Sprechakte bewirken, daß sich der Agent dazu verpflichtet, die im Content enthaltene Aktion nicht durchzuführen. Anschließend werden alle Commitment-Regeln auf Anwendbarkeit getestet und neue Commitments in die entsprechende Datenbasis geschrieben. Der Agent verpflichtet sich gegenüber dem im vorletzten Argument angegebenen Agenten zu der im letzten Argument spezifizierten Aktion genau dann, wenn

   - die Nachrichtenbedingung erfüllt ist, d.h. das erste Argument der Regel mit einem der empfangenen Sprechakte unifizierbar ist,
   - die mentale Bedingung (zweites Argument) erfüllt ist,
   - die Aktion eine aktuell gültige Capability bezeichnet und
   - der Agent sich nicht durch ein empfangenes refrain zur Nichtausführung der Aktion verpflichtet hat.

3 Im letzten Schritt des Zyklus werden alle aufgrund der aktuellen Systemzeit fälligen Commitments ausgeführt.

   - Inform-Sprechakte werden an die angegebenen Adressaten geschickt. Weiterhin wird in der Belief-Datenbasis vermerkt, daß der Adressat nun an den ihm mitgeteilten Sachverhalt glaubt oder, mit schwächerer Semantik, daß dem Adressaten die Information zugestellt wurde.
   - Andere Sprechakte werden versendet, ohne daß Änderungen in der internen Wissensbasis vorgenommen werden.
   - Private Aktionen werden direkt ausgeführt. Allerdings stellt hierfür der Interpreter keine Mechanismen zur Steuerung, Kontrolle, etc. bereit.

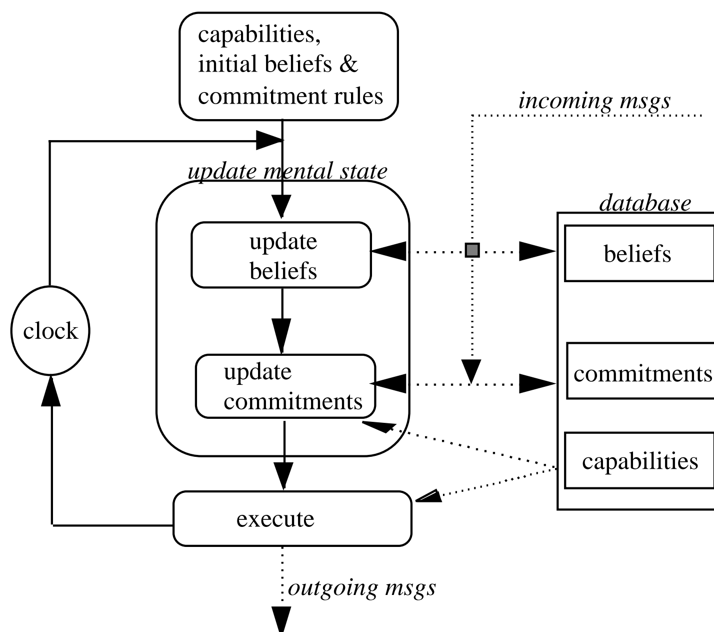

Abbildung 4-1: Agent-0-Interpreter

Eine Reihe von einschränkenden Annahmen sind zur Vereinfachung des Ablaufs und aus Komplexitätsgründen getroffen worden:

- Beliefs und Commitments unterliegen der Konsistenzannahme. Dies bedeutet, daß keine Verpflichtungen „aktiv“ sein können, die nicht entsprechend durch die aktuellen Beliefs unterstützt werden. So verpflichtet sich ein Agent zu einer Aktion x nur dann, wenn der durch x zu erreichende Zustand nicht bereits erfüllt ist. Außerdem gilt: (Genau dann) wenn ein Agent sich zu einer Aktion verpflichtet hat, glaubt er auch an deren Erfüllbarkeit. Ebenfalls gilt die Negation: Hat er sich nicht verpflichtet, dann glaubt er genau dieses.
- Aussagen sind einfache Terme; Verknüpfungen in Form von Konjuktionen oder Disjunktionen sind nicht erlaubt. Mentale Bedingungen hingegen dürfen verUNDete und verODERte Terme enthalten.
- Die Ebene der Handlungsmodellierung wird von Agent-0 nicht abgedeckt. Commitments beinhalten nur primitive Aktionen, bieten also nur eine sehr einfache Granularität. Wie komplexe Handlungsstränge oder Planungsaufgaben modelliert werden läßt die Sprache offen.
- Es existieren keine Konzepte für das Management von Beliefs. Per tell-Sprechakt empfangene Information wird einfach als neue Erkenntnis akzeptiert. Dabei wird nicht die Vertrauenswürdigkeit des Senders oder eigenes Kontextwissen

mit in Betracht gezogen. Die Behandlung von auftretenden Widersprüchen wird trivial dadurch gehandhabt, daß neue Fakten alte überschreiben.

4.1.1 Methodik

Während Agent-0 und PLACA Sprachen für die Programmierung reaktiver Agenten darstellen geht AgentBuilder einen Schritt weiter und kleidet die Programmiersprache in eine Entwicklungsumgebung ein. Die Entwickler von AgentBuilder beschreiten mit ihrem Ansatzes den klassischen Weg der phasenorientierten Softwareentwicklung. Sie ordnen agentenorientiertes Programmieren in die Nähe des objektorientierten Programmierens ein, mit dem Unterschied, daß die Agenten eine höhere Abstraktionsebene darstellen.

Die AgentBuilder-Sprache heißt RADL (Retucular Agent Definition Language) und ist in ein Toolkit integriert, das die unterschiedlichen Aspekte des BDI-Agentenmodells einzeln herausstellt. So existieren grafische Eingabewerkzeuge für die Beschreibung von Beliefs, Commitment-Regeln, Fähigkeiten, Commitments und Intentionen.

Der Programmierung vorgeschaltet sind Prozesse zur Projektorganisation, Ontologiedefinition und der Problemdekomposition (Abbildung 4-2). Im ersten Schritt wird ein Agentenprojekt kreiert. Dabei kann auf bereits existierende Projekte zurückgegriffen werden. So können bereits implementierte Agenten und Ontologien wiederverwendet werden. Bei der Analyse der Problemdomäne wird eine Ontologie in Form von losen Konzepten und Relationen zwischen diesen aufgebaut. Ergebnis ist eine konsistente Repräsentation der Begriffe der Problemdomäne. Die Problemdekomposition besteht aus der Identifikation der beteiligten Agenten und Agententypen. Hier werden Agentenskelette generiert, die im späteren Programmierzyklus mit den oben genannten Editoren weiter auszuprogrammieren sind.

Die prozeduralen Fähigkeiten eines Agenten werden als Aktionen in einer Bibliothek verwaltet und gemeinsam mit der generierten Agentendefinition und grafischen Bedienoberflächen zum Agentenprogramm zusammengebunden. Dieses wird dann von der Laufzeitumgebung entsprechend den BDI-Verarbeitungsmechanismen (siehe Abbildung 4-1) ausgeführt.

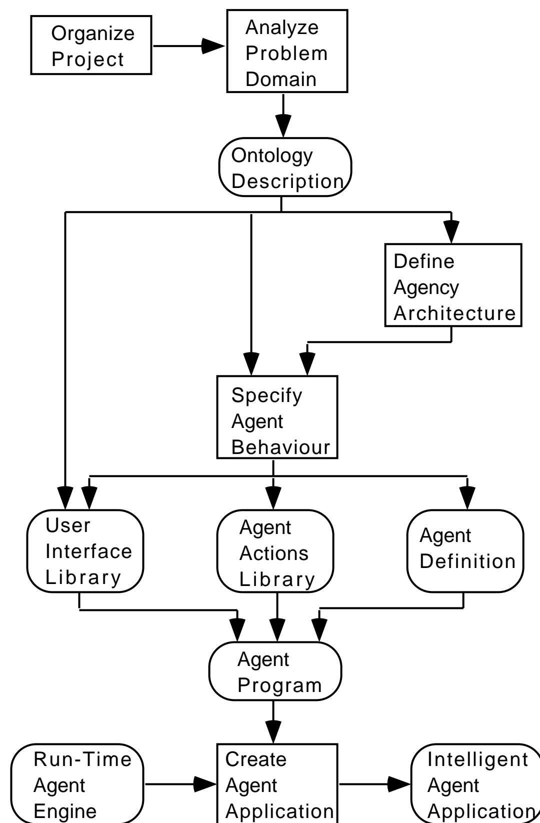

Abbildung 4-2: Agentenentwicklungsprozeß in AgentBuilder

Grafische Entwicklungswerkzeuge für Projektverwaltung, Ontologien, Protokolldefinition sowie Editoren für Commitments, Aktionen und Regeln vervollständigen die Entwicklungsumgebung.

4.1.2 Bewertung

Agent-0 ist eine Agentenprogrammiersprache, die auf dem reaktiven BDI-Agentenmodell basiert. Sie bietet einen regelbasieten Ansatz, um Kommunikation und Handeln durch Programmierung des mentalen Zustands miteinander zu verbinden. Agenten sind wenig autonom, sie handeln ausschließlich aufgrund von Handlungsanfragen bzw. -anweisungen.

PLACA[4] erweitert den Agent-0-Ansatz um Pläne und Intentionen. Agenten sind damit in der Lage, anstelle von einfachen Aktionen auch das Erreichen von abstrakteren Zielen per request-Sprechakt nachzufragen. Das Intentionskonzept dient zur

flexibleren Verwaltung des mentalen Zustands. Mit seiner Hilfe ist die Erreichbarkeit von Zielen, Gültigkeit und Ausführbarkeit von Teilplänen kontrollierbar. AgentBuilder integriert den vollständigen KQML-Sprachumfang in das Agentenmodell. Außerdem wird ein werkzeugunterstütztes Konzept für die Implementierung und Verwaltung von ganzen agentenbasierten Anwendungen mitgeliefert. Die KQML-Integration ist allerdings schwach: Die Interpretation von Sprechakten, sowohl auf Seiten des Senders wie des Empfängers, obliegt allein der Verantwortung des Agentenprogrammierers. In den Commitment-Regeln, wo die Verarbeitung von Sprechakten stattfindet, existieren keine Restriktionen bezüglich einer unterschiedlichen Behandlung der einzelnen Nachrichtentypen. Performatives werden zwar durch Pattern-Matching erkannt, sind aber beliebig interpretierbar. Das kann dazu führen, daß die intendierte Bedeutung ad absurdum geführt wird. Nirgendwo ist beispielsweise festgeschrieben, daß ein Agent ein ask-one tatsächlich als Anfrage interpretiert.

4.2 dMARS

dMARS[5] ist eine BDI-orientierte Entwicklungsumgebung. Im Vergleich zu Agent-0 wird hier ein stärker prozedural ausgerichteter Ansatz gewählt. Beliefs, Desires und Intentions stellen Datenstrukturen und nicht etwa Modaloperatoren dar. Ein weiterer Unterschied zu Agent-0 manifestiert sich in der zentralen Bedeutung von Plänen.

Aus der Event-Queue eines Agenten werden alle wahrgenommenen Ereignisse gelesen und daraus neue Desires generiert. Die Auswahl eines Plans aufgrund von Ereignissen erfolgt in einem Unifikationsschritt, der auch die Planvorbedingung mit berücksichtigt. Von u.U. mehreren zur Auswahl stehenden Plänen wird einer ausgewählt und instantiiert. Die Planinstanz wird als Intention auf den dafür vorgesehenen Intention-Stack zur Abarbeitung abgelegt.

Auf dem Intention-Stack befinden sich alle in aktiven Pläne und Teilpläne. Intentionen sind Sequenzen von Planinstanzen, die der Agent auszuführen beabsichtigt. Sequenzen entstehen immer dann, wenn bei der Abarbeitung des Rumpfes Unterziele aktiv werden, die ihrerseits zur Hinzunahme weiterer Pläne führen. Wenn mehrere Pläne gleichzeitig aktiv sind, enthält der Intention-Stack entsprechend vie-

4. PLAnning Communicating Agents.

5. dMARS (distributed Multi-Agent Reasoning System) [d´Inverno, et al. 1997] ist eine Weiterentwicklung des Procedural Reasoning Systems (PRS) [Georgeff, Lansky 1987]. Für eine eingeschränkte Version des PRS wurde mit AgentSpeak(L) [Weerasooriya, et al. 1995], [Rao 1996] eine logikbasierte Programmiersprache entwickelt.

le Einträge. Intentionen werden ausgeführt, sobald die Event-Queue leer ist. Die Planabarbeitung erfolgt entlang von Verzweigungen bis zu einem erfolgreichen Zielzustand oder bis keine Zweigalternativen mehr vorhanden sind. In diesem Fall gilt der Plan als gescheitert.

Repräsentation

- Beliefs werden als Terme ohne Variablen notiert. Die Verwendung eines not-Operators erlaubt negierte Beliefs.
- Goals bauen auf Beliefs auf. Mittels der Operatoren and und or sind komplexere Situationen (Belief-Zustände) beschreibbar, die dann zu einem von zwei möglichen Goal-Typen zusammengefaßt werden können:
  - Achievment-Goals bewirken, daß der Agent eine Handlungsfolge mit dem Ziel, den beschriebenen Zustand zu erreichen, durchführt. Entsprechend wird die Ausführung des aktuellen Plans unterbrochen.
  - Query-Goals veranlassen den Agenten ebenfalls zu einer Aktionssequenz, jedoch mit dem Ziel zu beweisen, ob das Query-Goal wahr ist oder nicht.
- Aktionen sind entweder „externe“ Prozeduraufrufe oder „interne“ Modifikationen der Beliefs. Bei letzteren ist das Hinzufügen oder Entfernen von Elementen möglich.
- Pläne beschreiben die Fähigkeiten eines Agenten. Jeder Plan ist ein 6-Tupel, bestehend aus einer Triggerbedingung, einem optionalen Kontext, dem Planrumpf, einer Bewahrungsbedingung und zwei Mengen von internen Aktionen.
  - Die Triggerbedingung spezifiziert, wann der Plan zu einer Intention wird, d.h. zur Ausführung kommt. Als Ereignis kann eine empfangene Nachricht, eine Änderung in der Belief-Wissensbasis oder ein neues Ziel angegeben werden.
  - Der Kontext beschreibt, welche Situation neben der Triggerbedingung gegeben sein muß, damit der Plan ausgewählt wird.
  - Planrumpf: Der Inhalt des Plans wird durch den Planrumpf beschrieben. Dieser liegt als Baumstruktur vor, in dem Knoten Zustände beschreiben. Die Blätter des Baums sind Endzustände, die nach erfolgreicher Abarbeitung erreicht werden. Verzweigungen sind mit Zielen oder Aktionen verbunden. Sie dienen als Auswahlpunkte, beschreiben also alternative Wege zur Beendigung des Plans.
  - Bewahrungsbedingung: Die sogenannte maintenance condition beschreibt die unveränderlichen Aspekte des Planens in Form von Situationsbeschreibungen, die während der Planausführung gelten müssen. Finden hier Änderungen statt, so ist eine erfolgreiche Planausführung nicht mehr gewährleistet. Die Planabarbeitung wird abgebrochen und ein Fehlerzustand gesetzt.

- Interne Aktionen: Zwei Mengen von Aktionen beschreiben die Auswirkungen des Plans nach dessen Beendigung auf den mentalen Zustand des Agenten. Die eine Menge wird ausgeführt, wenn die Planausführung erfolgreich, d.h. in einem Endzustand endete. Im Falle eines Scheiterns kommt die andere Menge interner Aktionen zur Ausführung.

Planausführung

Durch Vergleich eines neu registrierten Ereignisses in der Event-Queue mit den Triggerbedingungen der Pläne und gleichzeitigem Abgleich der aktuellen Beliefs mit den Kontextbeschreibungen der Pläne ergibt sich eine Menge von Plankandidaten. Von den möglichen Kandidaten wird einer ausgewählt. Es wird eine Planinstanz gezogen und mit einer Planumgebung auf den Intentionenstack gelegt.

```
IF triggercond AND precond
THEN
    WHILE not FAILSTATE AND maintenancecond
    DO
        evaluate_plan_body( FAILSTATE )
    ENDWHILE
    execute_internal_actions( FAILSTATE )
ENDIF
```

Abbildung 4-3: Ausführung eines Plans in dMARS

Ein Plan wird abgearbeitet, indem der Rumpf von der Wurzel zu den Blättern hin traversiert wird. In jedem Zustand der Planung gibt die Planstruktur durch die Verzweigungen eine Menge möglicher Handlungsalternativen vor. Durch Unifikation dieser Alternativen mit dem aktuellen mentalen Zustand des Agenten werden die tatsächlich anwendbaren Alternativen bestimmt. Von diesen wird eine ausgewählt und entsprechend durchgeführt, wodurch der nächste Zustand erreicht wird. Ein Fehlerzustand ist gegeben, wenn im aktuellen Zustand keine Aktion anwendbar und ist dieser Zustand kein Endzustand ist. Dann wird per Backtracking zum letzten Auswahlpunkt zurückgekehrt und eine andere Handlungsalternative gewählt. Existiert kein Backtrackpunkt, dann gilt der Plan als gescheitert.

Die Mächtigkeit des planbestimmten Handelns kommt sowohl durch die Repräsentationsstruktur als auch durch die Verarbeitung zum Ausdruck: Im einfachsten Fall besteht ein Plan aus einer Sequenz von Handlungsschritten und wird linear abgearbeitet. Durch die Baumstruktur in Verbindung mit Backtracking wird bereits kombinatorische Vielfalt erreicht. Die Flexibilität des Handelns - und damit die Komplexität - wird noch durch das Konzept der Sub-Pläne erhöht: Immer wenn die gewählte Alternative ein neues Goal ist, führt dies zur Auswahl eines neuen Plans,

der als Unterplan zunächst abgearbeitet wird, bevor mit der Verarbeitung auf der höheren Ebene fortgeschritten wird.

Drei optionale Attribute steuern die Ausführung der Pläne innerhalb eines Agenten: Mit der Priorität wird festgelegt, in welcher Reihenfolge zwei gleichzeitig aktive Pläne ausgeführt werden. Eine Vorzugsrelation definiert die Rangfolge alternativer Pläne. Dabei wird ein weit hinten eingeordneter Plan nur dann aktiv, wenn alle vor ihm liegenden Pläne bereits gescheitert sind. Schließlich kann durch ein no retry-Attribut verhindert werden, daß, nachdem die Abarbeitung innerhalb eines Planrumpfes in einer Sackgasse endete, auf einem anderen Auswahlpunkt wieder aufgesetzt wird. Dadurch wird die Vollständigkeit der Suche zum Vorteil einer schnellen Ergebnisfindung aufgegeben.

4.2.1 Methodik

Kinny, Georgeff und Rao[6] haben eine Modellierungstechnik für BDI-Agentensysteme entwickelt, die sich lose an objektorientierten Methodologien orientiert. Sie unterscheiden zwei Sichten, eine *externe* auf das Gesamtproblem und eine *interne*, die den einzelnen Agenten als Betrachtungsgegenstand hat. Dabei sieht die Methodik vor, zunächst die externe Sicht durch Problemdekomposition in eine Menge von Agenten zu beschreiben, um danach in drei Schritten die einzelnen Agenten aus interner Sicht zu modellieren. Mit der externen Sicht wird ein Systemmodell konstruiert, wobei die Modellierung durch etablierte Techniken objektorientierter Analyse und objektorientiertem Design erfolgt. Erst in der internen Sicht spielt die konkrete BDI-Architektur mit den verwendeten mentalen Konzepten eine Rolle.

Modellierung der externen Sicht

Die Modellierung der externen Sicht besteht im Aufbau einer Agentenklassenhierarchie, einer Zuordnung von Diensten an die einzelnen Klassen und einer Formulierung der Kontrollbeziehungen zwischen den einzelnen Klassen.

6. [Kinny, et al. 1996].

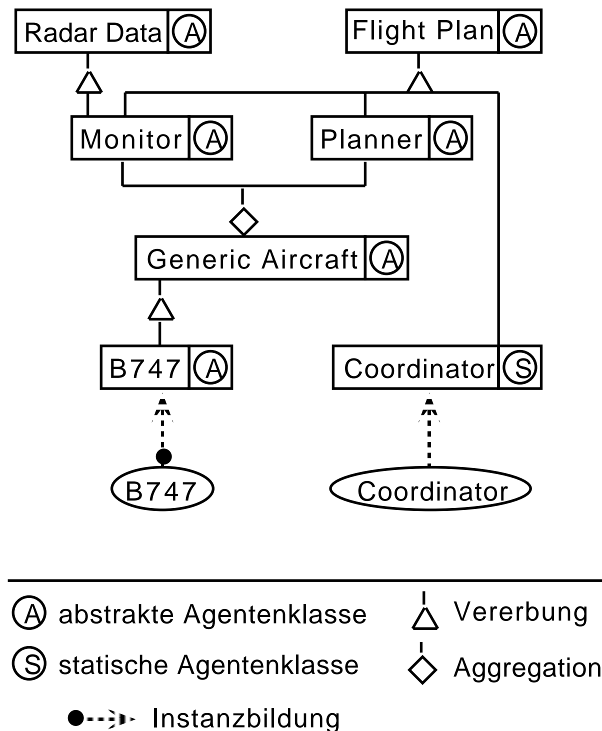

Abbildung 4-4: Agenten-Klassendiagramm in dMARS[7]

Der Aufbau der Klassenhierarchie soll aus einer Analyse des Problems entlang identifizierter *Rollen* abgeleitet werden. Ergebnis ist eine Menge abstrakter und konkreter Klassendiagramme mit Vererbungs- und Aggregationsrelationen. Vererbung bezieht sich auf die Attribute der Agentenklassen, namentlich Beliefs, Desires und Intentions. Mehrfachvererbung ist erlaubt, wie die Beziehungen zwischen Monitor, Radar Data und Flight Plan in Abbildung 4-4 zeigen. Aggregation drückt eine has-a-Beziehung aus. Semantisch bedeutet dies Subagentenbildung. Im Beispiel enthält die abstrakte Agentenklasse Generic Aircraft die Agenten Monitor und Planner. Aggregierte Agenten haben keinen gegenseitigen Zugriff auf ihre Attribute. So sind Monitor und Planner jeweils Flight Plan-Agenten, teilen jedoch nicht ihr Wissen über Flugpläne.

Danach werden für jede Rolle die assoziierten Verantwortlichkeiten beschrieben und entsprechenden Diensten zugeordnet. Auf Dienstebene werden anschließend

7. Die Grafik aus [Kinny, et al. 1996] (S.8) zeigt einen Ausschnitt aus einer Modellierung für ein Flugverkehrs-Managementsystem.

die Kontrollbeziehungen zwischen Agenten hergeleitet. Hierfür findet eine Beschreibung der Interaktionen statt, die Syntax und Semantik von Nachrichten und Kommunikation mit Systemkomponenten wie Benutzerschnittstellen beinhaltet. Abschließend ist ein Reengineering-Schritt vorgesehen, in dem neue Klassen gebildet werden. Hierbei werden gemeinsame Strukturen ähnlicher Agentenklassen zu abstrakten Klassen aggregiert und Agentenklassen nach den Kriterien Lebenszeit, Information und Schnittstellen neu abgeleitet.

Modellierung der internen Sicht

Im zweiten Schritt findet für jeden Agenten die BDI-spezifische interne Modellierung statt, indem ein Belief-, ein Ziel- und ein Planmodell erarbeitet wird. Weil die Funktionalität des BDI-Interpreters die Generierung von Beliefs aufgrund empfangener Nachrichten, die Ableitung von Zielen aus Beliefs, die weitere Aktivierung von Plänen und schließlich die korrekte Ausführung der Pläne vollständig und konsistent abdeckt, können die einzelnen Modelle relativ unabhängig voneinander erstellt werden.

Das Belief-Modell eines Agenten umfaßt alle relevanten Informationen der Umgebung sowie die elementaren Aktionen, die der Agent beherrscht. Zusätzliche Eigenschaften beschreiben, ob ein Belief dauerhaft ist oder verändert werden kann, ob er ein abstraktes Konzept darstellt, in der Wissensbasis gespeichert oder berechnet wird. Ein oder mehrere Elemente der Belief-Menge können herausgestellt werden um den initialen mentalen Zustand zu beschreiben. Auf ähnliche Weise wird das Zielmodell beschrieben. Im Planmodell schließlich erfolgt die Spezifikation der Pläne zur Erfüllung der Ziele oder aufgetretener Ereignisse. Entwicklungswerkzeuge zur Plankonstruktion und für TCP/IP-basierte Kommunikation zwischen Agenten unterstützen den Entwicklungsprozeß.

4.2.2 Bewertung

Die dMARS-Variante des BDI-Agentenmodells verknüpft, zusammen mit der vorgestellten objekt-orientierten Methodologie, globale Systemsicht und lokale Sicht auf einzelne Agenten.

Die objektorientierte Vorgehensweise beim Design zeigt, daß Agenten als komplexe, spezialisierte Objekte betrachtet werden. Kommunikation findet zwar auf Sprechaktebene statt; welcher Agent mit wem zu welchem Zeitpunkt interagiert, ist jedoch bereits fest im hierarchischen Systemdesign verankert. Kooperationsprotokolle werden nicht unterstützt. Weil darüberhinaus die Architektur keine Konzepte für die Verwaltung der Belief-Wissensbasis oder „intelligente“, flexible Generierung von Zielen, Intentionen oder Plänen in Form eines Resaoners bereitstellt, sind die Attribute Autonomie und Intelligenz sehr schwach ausgeprägt.

Stattdessen liegt die Stärke von dMARS-Agenten in ihrer Reaktivität und einfachen Struktur, gepaart mit einem sehr mächtigen Modellierungswerkzeug für Fähigkeiten. Einfache Regelmechanismen bewirken die Aktivierung von Plänen, die auch knowledge areas genannt werden - was ihrer Bedeutung wesentlich näher kommt. Hier ist die Flexibilität des Handelns in Form alternativer Handlungsstränge angesiedelt.

Kritisch anzumerken ist die fehlende Verbindung zwischen den in der Literatur ausführlich behandelten BDI-Modellen und den gleichnamigen Architekturen. So finden sich die Begriffe Belief, Desire und Intention zwar in Architektur und Sprache wieder, nicht jedoch die dahinterliegenden Theorien.

4.3 Agent Building Shell / COOL

Die Agent Building Shell[8] ist eine Expertensystem-Schale für Agentensysteme. Sie enthält eine Reihe von Sprachen und Funktionalitäten sowie eine toolbasierte Entwicklungs- und Laufzeitumgebung. Die Sicht auf die Agentenprogrammierung ist stark kommunikationszentriert: Agenten erarbeiten Lösungen, indem sie Kooperationsprotokolle, die hier Konversationen genannt werden, abarbeiten.

Konversationen werden als endliche Automaten mit einem ausgezeichnetem Anfangs und einer Menge von Endzuständen modelliert. Zustandswechsel finden durch Nachrichtenaustausch in Form von Sprechakten statt. Abbildung 4-5 zeigt ein Verhandlungsprotokoll aus Sicht des initiierenden Agenten. Vom Anfangszustand 1 geht der Agent durch Versenden eines Vorschlags (als *propose/* noptiert) in Zustand 2 über. Dort wechselt er, je nachdem wie die Antwort ausfällt, entweder nach 3, 4 oder 5. Derartige Konversationsgraphen werden mit Hilfe der Koordinationssprache COOL[9] beschrieben.

8. [Barbuceanu, Fox 1996].

9. COOrdination Language [Barbuceanu, Fox 1995].

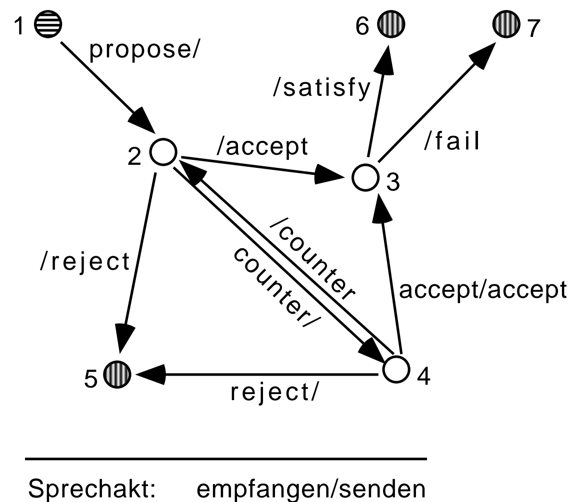

Abbildung 4-5: COOL-Konversation als endlicher Automat[10]

- Konversationsregeln spezifizieren das Verhalten eines Agenten innerhalb eines Kooperationsprotokolls. Für jeden Zustandsübergang im Konversationsgraphen muß eine Regel existieren, die das Verhalten des Agenten beschreibt. Die Vorbedingung einer Konversationsregel besteht aus dem Zustandsnamen und einem erwarteten Sprechakt. Im Aktionsteil der Regel ist neben der Handlungsanweisung eine Spezifikation der zu sendenden Nachricht und den Nachfolgezustand enthalten. Um auch Konversationen mit mehreren Agenten zu ermöglichen existieren Konstrukte, um den Empfang mehrerer Nachrichten zu testen.

```
(def-conversation-rule r1
    :current-state 2
    :received (proposed
        :sender ?initiator
        :content (produce ?what ?amount ?time))
    :such-that (can-do ?what ?amount ?time)
    :next-state 3
    :transmit (accept
        :content (produce ?what ?amount ?time)))
```

- Fehlerbehandlungsregeln feuern, wenn eine empfangene Nachicht nicht behandelt werden kann. Eine solche Situation kann durch Kommunikationsfehler, Verzögerungen oder fehlerhaft (bzw. unvollständig) implementierte Protokolle

10. Das Senden einer Nachricht wird durch einen nachgestellten, der Empfang durch einen vorangestellten Schrägstrich gekennzeichnet

eintreten. Durch Fehlerbehandlung kann nun flexibel reagiert werden - vom Start einer Klärungskonversation über das Vernachlässigen des Fehlers bis hin zur Fehlermeldung auf dem Terminal des Benutzers.

- Konversationsklassen spezifizieren die Zustände, Konversations- und Fehlerbehandlungsregeln, Kooperationspartner, Variablen und Kontrollmechanismen von Konversationen. Das Klassenkonzept erlaubt es, daß ein Agent mehrere Protokolle desselben Typs gleichzeitig fahren kann.

```
(def-conversation-class Cnv-1
     :initiator ?initiator
     :respondent ?respondent
     :variables (?v1 ?v2)
     :initial-state s0
     :final-states (s5 s7 s6)
     :conversation-rules ((s0 r1 r2), ...)
     :error-rules (e1, ...)
     :error-rule-applier ERA-1)
```

Ein Kooperationsprotokoll wird durch Instanzbildung einer Konversationsklasse gestartet. Dabei wird eine Konversationsumgebung generiert, die den Zustand der Protokolldurchführung verwaltet. Außerdem bietet sie einen Speicher für globale Variablen, mit dem beispielsweise Zwischenergebnisse persistent über mehrere Zustände hinweg verwaltet werden können.

4.3.1 Methodik

Agentenprogrammierung in COOL ist gleichbedeutend mit der Konstruktion von Konversationen. Hierfür existiert eine komfortable Entwicklungsumgebung, die Rapid Prototyping unterstützt. Konversationsklassen und -regeln können Schritt für Schritt definiert und getestet werden. Trifft der Agenteninterpreter auf eine nicht vorhandene Konversationsregel, wird der Benutzer aufgefordert, entsprechende Maßnahmen zu ergreifen. So können Protokolle inkrementell kreiert und verbessert werden. Eingebettet in die Entwicklungsumgebung sind Debuggingwerkzeuge, mit denen zur Lauf- bzw. Testzeit aktiv in das Systemgeschehen eingegriffen werden kann.

Zu dem interaktionszentrierten Modellierungsansatz stellt JAFMAS[11] eine Entwicklungsmethodik bereit. Danach besteht der Prozeß der Agentenprogrammierung aus 5 Schritten:

11. Java-based Agent Framework for Multi-Agent Systems [Chauhan 1997]. JAFMAS stellt im wesentlichen eine Re-Implementierung von COOL in Java dar.

1 Identifikation der Agenten: Eine vorgegebene Aufgabenstellung wird anhand von globalen Zielen in Agenten zerlegt. Dabei werden agentenlokale Ziele und Dienste identifiziert. Es folgt eine Einordnung ähnlicher Agenten zu Agentenkategorien. So gelangt man zu Agentenklassen.

2 Definition der Konversationen: Für jeden Agenten werden die notwendigen Konversationen mittels endlicher Automaten beschrieben.

3 Bestimmung der Konversationsregeln: Hierbei werden die Konversationen „mit Leben gefüllt“, indem die Zustandsübergänge der Automatenmodelle durch Regeln beschrieben werden.

4 Analyse der Konversationen: Die Vollständigkeit und logische Konsistenz aller Konversationen ist zu untersuchen. Es wird vorgeschlagen, sich hierbei automatischer Unterstützung z.B. in Form von Petrinetzwerkzeugen zu bedienen. Im Erfolgsfall ist die Systemkohärenz sichergestellt, ansonsten muß zum letzten Schritt zurückgekehrt werden.

5 Implementierung des Agentensystems: Mit Hilfe eines passenden Werkzeugs (JAFMAS-Klassenbibliotheken) werden die Konversationsregeln, Sprechakte und Agenten implementiert.

4.3.2 Bewertung

Agenten bestehen aus einer Menge von Konversationen. Eine Architektur, die Wissen speichert, verarbeitet oder räsonniert ist nicht vorhanden. Weil sich alles Handeln in den Konversationen abspielt, können Agenten als strikt reaktiv bezeichnet werden. Die Modellierung und Programmierung mit COOL/JAFMAS entspricht einer problemzentrierten Top-Down-Vorgehensweise. Letztendlich handelt es sich um eine Erweiterung bekannter objektorientierter Ansätze. Agenten sind verteilte Objekte, die anstelle von Methodenaufrufen zu komplexen Interaktionen in der Lage sind.

Die Kommunikation orientiert sich an KQML. Dabei wurde die Nachrichtensyntax um einen :conversation-Eintrag erweitert, der den Bezug eines Sprechaktes zur dazugehörigen Konversationsinstanz herstellt. Erst dadurch ist es möglich, auf die Historie einer Kooperation Bezug zu nehmen. Die Semantik von Sprechakten wird vollständig in den Konversationsregeln festgelegt und ist alleine vom Zustand der Konversation und dem Nachrichteninhalt abhängig. Eine linguistische Schicht ist nicht vorhanden, Sprechakte werden per pattern matching erkannt. Die Performative ist somit „syntaktischer Zucker“ ohne eigentliche Bedeutung. Da die Wahl einer Konversationsregel durch einen allgemeinen Unifikationsschritt entschieden wird, ist es beispielsweise möglich, sämtliche Kommunikation über einen einheitlichen tell-Sprechakt abzuhandeln.

4.4 TÆMS / GPGP

Der Beitrag von TÆMS / GPGP[12] für die agentenorientierte Programmierung liegt in der Modellierung und Nutzung von Koordinationsbeziehungen. Anders als bei COOL findet Koordination nicht in globalen Kooperationsprotokollen Ausdruck, sondern wird lokal behandelt. Hierfür bietet TÆMS ein „Rahmenwerk“ zur Repräsentation domänenunabhängiger Koordinationsprobleme und einer Zuordnung von Koordinationsmechanismen an. Darüberhinaus enthält TÆMS eine Simulationsumgebung, in der verschiedene Koordinationsmechanismen in unterschiedlichen Kontexten gegeneinander getestet werden können.

Die Spezifikation findet auf zwei Ebenen statt: Zunächst werden hierarchische Task-Strukturen beschrieben und zu organisatorischen Einheiten zusammengefaßt. Danach werden zwischen Elementen innerhalb einer Struktur die Koordinationsbeziehungen herausgearbeitet. Nach abgeschlossener Modellierung greifen die GPGP[13]-Mechanismen in den Agenten zur wohlkoordinierten Auswahl und Durchführung von Aktionen gemäß vorgegebener Ziele.

Task-Strukturen

Eine Szenariobeschreibung besteht aus einer Menge von Task-Strukturen. Diese werden als gerichtete, azyklische Graphen dargestellt. Jeder Knoten im Graph stellt eine Task dar, verbundene Knoten drücken Task-Subtask-Beziehungen aus. Knoten ohne Nachfolger werden auch Methoden genannt, das sind ausführbare, elementare Aktionen. Alle Task-Strukturen, zwischen denen Abhängigkeiten bestehen, werden zu einer Task-Gruppe zusammengefaßt. Auf diese Weise definiert jede Task-Gruppe eine Menge von Aktionen, die - unmittelbar oder mittelbar - voneinander abhängig sind. Umgekehrt sind Tasks, die unterschiedlichen Task-Gruppen angehören, voneinander unabhängig. Vervollständigt wird eine Task-Gruppe durch eine Zeitspezifikation, die angibt, wann spätestens die Gruppe abgearbeitet sein muß.

Jede Task besitzt ein Qualitätsattribut. Die Qualität einer Task wird durch die Qualität der Subtasks bestimmt. TÆMS bietet zur Aggregation die Möglichkeit der Summenbildung, Minimum, Maximum sowie arithmetisches Mittel (siehe die unter den Knoten in Abbildung 4-6 stehenden Namen).

12. Task Analysis, Environment Modeling, and Simulation [Decker 1995] und Generalized Partial Global Planning [Decker, Lesser 1991].

13. GPGP ist historisch aus der verteilten Sensoranwendung DVMT [Lesser, Corkill 1983], [Durfee, Lesser 1989], [Durfee, Lesser 1991] entstanden. Es besteht aus einer Reihe von Datenstrukturen und Algorithmen zur Behandlung der nachfolgend beschriebenen Koordinationsbeziehungen.

Methoden werden mit den generischen Attributen Qualität, Zeit und Kosten feinspezifiziert. Der Tatsache, daß diese Quantitäten gewöhnlich nicht exakt vorhersagbar sind, wird einer Zuordnung von Ungewißheitsfaktoren in Form von Wahrscheinlichkeiten Rechnung getragen. Koordinationsbeziehungen zwischen Paaren von Zielen oder Aktionen bringen zusätzliches Wissen ein. Wenn mehrere konkurrierende Methoden existieren, können die Attribute für die Methodenauswahl herangezogen werden.

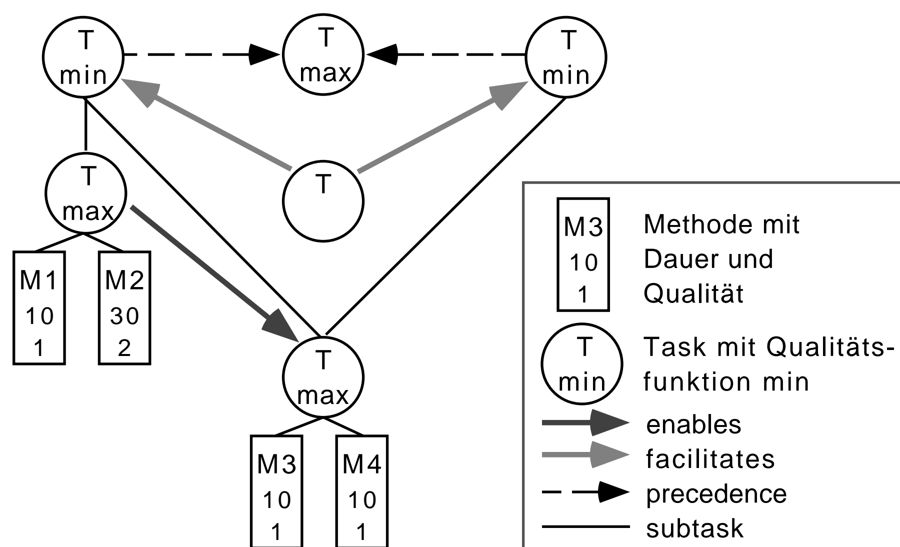

Abbildung 4-6: Modellierung einer Task-Struktur in TÆMS[14]

Koordinationsbeziehungen

Die elementare Subtask-Beziehung wurde bereits oben erklärt. Neben dieser agentenlokalen Relation[15] existieren Koordinationsbeziehungen zwischen Tasks, die globale Effekte haben. Diese Effekte beschränken sich im TÆMS-Kontext bei den betroffenen Agenten auf Zeit- und Qualitätsänderungen.

14. Die Repräsentation sagt nichts darüber aus, ob es sich bei Subtasks um UND oder ODER-Beziehungen handelt. [Decker, Lesser 1995] (S.10) sprechen von einer ODER-Beziehung im Falle einer MAX-Aggregationsvorschrift.

15. Task-Subtask-Beziehungen sind deshalb lokaler Natur, weil gewöhnlich die gesamte Task-Struktur (im Gegensatz zur Task-Gruppe) in einem Agenten angesiedelt ist.

- Die enables-Beziehung drückt aus, daß eine Task beendet sein muß, bevor die andere starten darf.
- Mit facilitates wird beschrieben, daß eine Task eine andere unterstützt. Wenn die unterstützende Task ausgeführt wurde, bewirkt dies, weil deren Ergebnis genutzt werden kann entweder eine Steigerung der Qualität oder Verbesserung der Performance bei der unterstützten Task.
- Hinders ist das Gegenteil von facilitates. Die Qualität der abhängigen Task wird herabgesetzt oder deren notwendige Laufzeit verlängert.
- Causes drückt aus, daß die erfolgreiche Abarbeitung einer Task automatisch die Vollendung der abhängigen Task bewirkt.
- Cancels bewirkt durch Herabsetzen der Qualität auf 0, daß eine Task nach Beendigung einer anderen nicht mehr durchgeführt wird.

Andere Beziehungen wie precedence oder shares-result lassen sich aus den oben genannten ableiten.

Ein weiteres Mittel zur Aktivitätenkoordinierung in TÆMS bieten Ressourcen. Die Verknüpfung einer Task mit einer Ressource bewirkt zwei Effekte: Einerseits wird der Zustand der Ressource durch eine der Methoden uses, replenishes, consumes, reads oder writes verändert, andererseits bewirkt deren Nutzung Änderungen in bezug auf die Qualität und Dauer der Task. Allerdings ist die Ressourcenmodellierung sehr einfach gehalten; mehr als zwei Tasks können nicht dieselbe Ressource nutzen.

Koordinationsmechanismen

Unter dem Begriff *Generalized Partial Global Planning* ist eine Serie von Mechanismen zur Verarbeitung von Koordinationsbeziehungen zusammengefaßt. GPGP erfordert keine spezielle Agentenarchitektur, gefordert werden bloß eine Belief-Wissensbasis, ein Scheduler und ein Koordinationsmodul. Die Wissensbasis enthält die oben beschriebenen Task-Strukturen anderer Agenten. Im Koordinationsmodul sind die nachfolgend beschriebenen Mechanismen enthalten. Dort findet auch die Steuerung eines Agenten statt: Zyklisch, jedoch mit einstellbarem Aufwand wird „Informationssammlung“, Kommunikation, Scheduling und Ausführung der lokalen Methoden betrieben. Informationssammlung betrifft das Eintragen aller empfangener Informationen in die Belief-Wissensbasis; Kommunikation beschränkt sich auf das Versenden von Task-Strukturen und Ergebnissen.

Herzstück eines jeden GPGP-Agenten ist ein nutzenoptimiert arbeitender Scheduler. Dieser verarbeitet die Informationen der Task-Hierarchien in Abhängigkeit von benutzerdefinierten Kriterien zu einer Task-Reihenfolge. Als Anforderungen können Gewichtungen der Qualitäts- und Kostenparameter angegeben werden. Die Vorgehensweise des Schedulers ist inkrementell; er berücksichtigt bereits geplante

Tasks aufgrund eigener lokaler oder globaler Verpflichtungen. Dabei werden die Berechnungsaufwandsinformationen genutzt, um Vorhersagen über den Zeitpunkt von vorliegenden Teilergebnissen zu machen.

Sowohl die lokalen wie auch die nicht-lokalen Commitments sind Resultat der nachfolgend vorgestellten Koordinationsmechanismen:

- Austausch privater Sichten auf Task-Strukturen: Agenten können einander über ihre Task-Strukturen informieren. Dabei können gesamte Strukturen, aber auch Teilstrukturen und abstrahierte, unvollständige Strukturen kommuniziert werden. Ziel ist es, anderen Agenten eine globalere Sicht zu verschaffen. Ein Agent, der eine Task-Struktur eines anderen Agenten empfängt, kann neue Koordinationsbeziehungen erkennen und nutzen und so seine oder die globale Performance verbessern. Zwischen dem möglichen Nutzen kommunizierter Strukturen und deren Umfang besteht jedoch eine relativ starke Wechselbeziehung. Um den Kommunikationsaufwand gering zu halten macht es daher üblicherweise Sinn, nur abstrakte Strukturen zu versenden.
- Kommunikation von Ergebnissen: Auch das Mitteilen ermittelter Resultate nach erfolgreicher Abarbeitung lokaler Methoden dient primär zur Vervollständigung der Weltsicht anderer Agenten. Hierbei können unterschiedliche Strategien gefahren werden: Entweder ist ein Agent wenig mitteilsam und gibt nur Ergebnisse, die aus Verpflichtungen anderen Agenten gegenüber beruhen heraus, oder er kommuniziert sämtliche Ergebnisse per Broadcast an alle Agenten. Auch ein Mittelweg kann beschritten werden, indem Ergebnisse, zu deren Übermittlung sich ein Agent verpflichtet hat, zusammen mit anderen Ergebnissen derselben Task-Struktur - eventuell per Broadcast - kommuniziert werden.
- Behandlung von Redundanzen: Redundanzen treten auf, wenn mehrere Agenten an denselben Methoden oder Task-Strukturen arbeiten oder wenn Unterstrukturen innerhalb einer Task-Hierarchie als Alternativen zueinander gelten. Ziel ist es, durch Redundanzvermeidung Mehrfacharbeit zu verhindern und so zu einer besseren Gesamtperformance zu gelangen. Die einfachste Methode sieht vor, daß ein beliebiger Agent zur Verarbeitung ausgewählt wird und das Resultat dem (bzw. den) anderen mitteilt. Dazu geht der Agent eine nicht-lokale Verpflichtung ein. Auch ist es möglich, die Beauftragung eines Agenten zum Gegenstand von Verhandlungen zu machen. Hierbei gilt es, den Agenten mit der besten Lösungsqualität, der schnellsten Lösungsfindung oder der geringsten Auslastung zu finden.
- Verarbeitung von Koordinationsbeziehungen: „Harte“ Beziehungen wie enables wirken direkt auf die Reihenfolge von Methoden. Sind für eine Methode eine Menge von vorher zu erledigenden Aufgaben bekannt und wurde für die Methode bereits eine zeitliche Zusage gemacht, müssen die Ergebnisse der vorangestellten Aufgaben rechtzeitig geliefert werden. Daraus ergeben sich Constraints für die spätesten Endtermine anderer Aufgaben, die den betreffenden Agenten kommuniziert werden müssen und dort beim Scheduling, zum Beispiel

durch Zeitminimierungsstrategien oder Umsortierung bereits eingeplanter Methoden, entsprechend zu berücksichtigen sind. In Verbindung mit Redundanz können zeitliche Nebenbedingungen auch relativ einfach durch Zuordnung der Aufgabe an den „eiligsten“ Agenten berücksichtigt werden.

Andere, „weiche“ Beziehungen sind verhandelbar. Beispielsweise kann eine erkannte facilitates-Relation lokal ohne Kommunikation genutzt werden, um die Vorgänger-Methode möglichst früh einzuschedulen. Dann ist die Wahrscheinlichkeit, daß der qualitätssteigernde oder laufzeitverringernde Effekt tatsächlich eintritt größer als bei späterer Bearbeitung.

4.4.1 Methodik

Anders als bei den vorher beschriebenen Ansätzen steht hier nicht der Agent im Mittelpunkt des Intersses. Stattdessen gibt TÆMS eine Modellierungssprache für komplexe Aufgaben vor. Der Programmierer spezifiziert Tasks baumartig top-down bis zur Ebene von Methoden und reichert die entstandene abstrakte Struktur mit Koordinationsbeziehungen zwischen Knoten an.

Anschließend sind die Methoden zu implementieren und in Agenten zu integrieren. Entscheidungsunterstützung für die Zuordnung von Methoden an Agenten, die Gestaltung der Schätzwerte für die generischen Parameter, den Umfang der kommunizierten Strukturen und die für die Systemperformance wichtigen Entscheidungen in bezug auf redundante Auslegung von Methoden gibt TÆMS jedoch nicht. Dafür kann eine Simulationsumgebung genutzt werden, um Aufschluß über das Systemverhalten bei unterschiedlicher Ausegung der genannten Parameter zu bekommen. Aus analytischen und durch Simulation belegten Betrachtungen wurden eine Reihe von generellen Vorhersagen bezüglich der Ausgestaltung *einzelner* Parameter abgeleitet.

Im Gegensatz zu den bisher vorgestellten Ansätzen spielt die Modellierung von typischen agentischen Fähigkeiten wie Kommunikation oder Kooperationsprotokollen keine Rolle.

4.4.2 Bewertung

TÆMS beschreitet mittels einer high-level-Spezifikationssprache einen problemzentrierten Ansatz. Der Beitrag zur agentenorientierten Programmierung liegt in der expliziten Herausarbeitung von Koordinationsbeziehungen zwischen Aspekten der Problemlösung. TÆMS/GPGP wurde für Anwendungen im Bereich verteilter Sensor-Netzwerke konzipiert und erfolgreich eingesetzt[16].

Von den Agenten wird im wesentlichen eine verteilte Scheduling-Leistung erbracht. Dabei nutzen sie global verfügbares, abstraktes Problemwissen in Form der

ihnen bekannten Taskstrukturen und Koodinationsbeziehungen sowie die während der Problemlösung übermittelten Informationen und eigene Schlußfolgerungen über die Aktivitäten anderer Agenten zur Erstellung lokaler Reihenfolgepläne. Durch die Konzentration auf das Erkennen und Ausnutzen von Koordinationsbeziehungen skalieren derartige Systeme gut.

Jedoch werden wesentliche Eigenschaften, die andere Agentenmodelle auszeichnen, nicht unterstützt. Insbesondere die fehlende sprechaktbasierte Kommunikation führt dazu, daß mit TÆMS realisierte Anwendungen geschlossene Systeme sind. Durch die Konzentration auf lokale Koordinierungsfähigkeiten und das Fehlen eines flexiblen Kooperationsmodells eignet sich dieser Ansatz nicht für dynamische Szenarien mit wechselnden, nicht im vorhinein bekannten Aufgabenstellungen. Im übrigen wird von den Autoren der Status von TÆMS/GPGP als Multiagenten-Testbett herausgestellt. Den Forschungscharakter dieses Ansatzes unterstreicht ein Simulator, der zwecks Gewinnung von Erkenntnissen in Bezug auf Skalierung von Agenten und Systemen eingesetzt wird.

4.5 Weitere Ansätze zur Agentenprogrammierung

Neben den oben vorgestellten Agentenprogrammiersprachen existieren Modellierungssprachen, die von konkreten Architekturen abstrahieren. Desire stellt ein Task-zentriertes Spezifikations-Rahmenwerk bereit[17]. In 5 Schritten werden nacheinander Task-Hierarchien aufgebaut, der Informationsaustausch zwischen Tasks: in Form von Informations-Links festgelegt, Aufgabenkontrollwissen über Task-Sequenzen formuliert, Tasks Agenten zugeorndet und schließlich Wissensstrukturen aufgebaut. Diese Wissensmodellierung erfolgt durch Konzepte und Relationen und basiert auf Prädikatenlogik. Wooldridge et al. übertragen die aus der objektorientierten Programmierung stammende FUSION-Methode auf die Agentensystemmodellierung[18]. Mittel zur Systementwicklung sind Rollen, Befugnisse, Protokolle, Verantwortlichkeiten und Invarianten. Während der Analysephase werden zunächst ein Rollen- und ein Interaktionsmodell getrennt entwickelt. In der anschließenden Designphase werden hieraus ein Agenten-, Service- und Acquaintances-Modell abgeleitet.

16. Bekanntestes Szenario ist das Distributed Vehicle Monitoring Testbed [Lesser, Corkill 1983], [Decker, et al. 1990], [Decker, Lesser 1991], in dem Agenten akustische Signale Fahrzeugtypen zuordnen und deren Routen berechnen.

17. [Brazier, et al. 1997], [Mulder, et al. 1997], siehe dazu auch den ähnlichen Ansatz von TÆMS in Abschnitt 4.4.

18. [Wooldridge, et al 1999].

Sogenannte „Agentenprogrammiersprachen“, die ohne darunterliegendes Agentenmodell auskommen bieten wenig Unterstützung zur Systemrealisierung. Zumeist handelt es sich hierbei um herkömmliche Programmiersprachen, die um mächtigere Kommunikationskonzepte erweitert wurden. Beispiele hierfür stellen Agent-Tcl und diverse Java-Erweiterungen dar. Bei letzteren handelt es sich um Bibliotheken, die Templates zur vollständigen Ausprogrammierung bereitstellen. Keine dieser Sprachen deckt jedoch Fähigkeiten oder Techniken ab, die in Kapitel 2 beschrieben wurden, so daß Agenten „from scratch“ zu implementieren sind.

Andere Herangehensweisen zeichnen sich dadurch aus, daß sie sich auf einzelne Aspekte der Agentenmodellierung konzentrieren. Diese lassen sich, entsprechend der Basisaktivitäten eines Agenten, in die drei Kategorien Kommunikation, Reasoning und Handlung einteilen. Daß die spezifischen Fähigkeiten mit Hilfe der bereitgestellten Repräsentationsformalismen modelliert werden, wurde bereits im 2. Kapitel gesagt; dort wurden auch die wichtigsten Beschreibungssprachen, namentlich KI-Repräsentationsformen und Planstrukturen, behandelt. Kommunikationsprotokoll-Beschreibungssprachen wie die Interaktionsmoleküle des IMAGINE-Projekts[19] oder die generischen Kooperationsobjekte[20] ermöglichen den Aufbau komplexer Dialoge auf der Basis einiger weniger Sprechakte.

Die Frage nach der Herkunft von Zielen eines Agenten wird in den meisten Fällen durch empfangene Sprechakte oder Fakten der Wissensbasis beantwortet. Einen anderen Weg beschreiten Norman und Long, indem sie eine Motivationstheorie in die Agenten verankern[21]: Agenten kreieren und priorisieren ihre Ziele als Reaktion und in Erwartung einer Situation. Bates untersucht Möglichkeiten, zielgerichtetes Verhalten durch Integration eines Konzepts von Emotionen in Agenten zu erreichen[22]. Von Kalenka und Jennings stammt die Idee für die Modellierung sozialer Normen. Sie definieren einen *social level*[23], auf dem sie beispielsweise hilfsbereite, kooperative und selbst-interessierte Agenten anhand ihrer Commitment-Strategien beschreiben können.

19. [Haugeneder, Steiner 1991], [Lux, et al. 1992].

20. Generic configurable cooperation protocols (GCCP) [Burmeister, et al. 1993].

21. [Norman, Long 1995], [Norman, Long 1996].

22. [Bates, et al. 1992], [Bates 1994].

23. [Kalenka, Jennings 1997], [Jennings, Campos 1997]. Der social level ist über Newells knowledg level [Newell 1982] angesiedelt.

4.6 Testen und Debugging

Programmierung ist mühevolle Hand- und Kopfarbeit Zwar versprechen agentenorientierte Techniken den konzeptuellen Abstand zwischen Spezifikation und Realisierung zu verringern. Dennoch bleibt die semantik- und wahrheitsbewahrende Übersetzung von Spezifikationen in Programmcode eine schwierige, fehlerbehaftete Angelegenheit, die Unterstützung durch Test- und Debuggingwerkzeuge verlangt. Verteiltheit, Parallelität, Nichtdeterminismus, eine schwer kontrollier- und vorhersagbare Umgebung erschweren jedoch das Verständnis und die Kontrolle der Abläufe innerhalb einer Agentenanwendung.

- Ursache-Wirkung-Muster sind schwer zu erkennen. Jede Aktion oder Ereignis kann mehrere zeitliche Vorgänger besitzen - doch welcher von diesen steht in einem kausalem Zusammenhang? Fehlerquellen sind aufgrund vieler möglicher Ursachen oft nur schwer lokalisierbar.
- Es ist schwierig wegen der nicht synchronen und verteilten Verarbeitung den Zustand eines Agentensystems zu beschreiben. Dies erschwert eine kontrollierte schrittweise Ausführung.
- Verteilte Systeme bringen neue Fehlerklassen mit sich. Verklemmungen, Ausfall und Überlastung von Prozessoren und Kommunikationsressourcen, Timeouts oder Kommunikationsfehler können auftreten.
- Der Kontrollbegriff in Mehragentensystemen hat zwei Dimensionen: Jeder Agent unterliegt als autonomer Problemlöser seiner eigenen, lokalen Kontrolle; andererseits existiert eine globale Kontrollsemantik, die sich durch das Systemverhalten, d.h. die Koordination der Agenten definiert.
- Der Korrektheitsbegriff ist durch Spezifikationen auf Verhaltensebene und einen schwammigen Koordiniertheitsbegriff schwer zu fassen. Auch sind Aussagen über das Verhalten eines Agenten nicht einfach zu treffen, wenn er in einer unbekannten Umgebung eingesetzt wird.

Prinzipiell kann das Testen durch direkten Eingriff in ein laufendes System oder durch Simulationstechniken erfolgen. Für den erstgenannten Ansatz müssen Agentenprogramme Debuggingfunktionalität bereitstellen, die ein Inspizieren und Verändern der Abläufe gestattet. Debugger sind Interpreter mit einer Schnittstelle zum Benutzer, die Eingriffsmöglichkeiten in den Programmablauf bietet. Typische Funktionen klassischer Debugger umfassen das Ansehen und Verändern von Variablenwerten, das Setzen von Breakpoints und eine kontrollierte schrittweise Ausführung.

Debugging von Agentensystemen findet auf der Ebene der Agentenprogrammierung statt: Untersucht werden die Kommunikation, der mentale Zustand und das Handeln der Agenten. In AgentBuilder (siehe Abschnitt 4.1.1) kann der Entwickler in die Interagenten-Kommunikation eingreifen und die Enwicklung der mentalen

Zustände im Verlauf des Uhrenzyklus beobachten. Er kann ferner Breakpoints setzen und die Programmabarbeitung schrittweise verfolgen. dMARS (siehe Abschnitt 4.2) speichert die Agentenzustände in Datenbasen ab und ermöglicht so Introspektion und Eingriffe; ferner kann die Planabarbeitung anhand der Ablaufgraphen beobachtet werden. Nahezu universelle Eingriffsmöglichkeiten bietet COOL: Protokolle können während der Abarbeitung verändert werden. Begünstigt wird die Bereitstellung derart mächtiger Werkzeuge durch den Verzicht auf parallele Abläufe und die Verwendung einer Interpretersprache (LISP).

Während Debugging in der realen Umgebung in Echtzeit stattfindet, sind bei einer Simulation alle Agenten und auch deren Umwelt vollständig in eine Testumgebung eingebettet. Dadurch ergibt sich ein deterministisches System, was den Vorteil mit sich bringt, daß Testläufe wiederholt werden können. Eingriffe in das Systemgeschehen sind unkompliziert: Nach einem Stoppen des Interpreters können Agenten- oder Umgebungsvariablen angesehen und manipuliert werden. Simulationsverfahren werden häufig dann eingesetzt, wenn das Testen in der realen Welt schwierig oder teuer ist. Allerdings hängt die Qualität der Ergebnisse von einer exakten Modellierung aller relevanten Umweltparameter und ihrer Zusammenhänge ab. Aus diesem Grund sind Simulationsansätze für reale Probleme meist ungeeignet, weil die Weltmodelle zu stark vereinfacht sind.

Simulationstestbetten für Agenten basieren zumeist auf abstrakten, zweidimensionalen Welten und sind für empirische Untersuchungen konzipiert. Beispiele sind Tileworld und MICE[24]. Beide sind domänenunabhängig, machen nur wenige Annahmen über den Aufbau eines Agenten und werden über diskrete Ereignisse gesteuert. Die Verwendung derartiger Testbetten zeigt zugleich deren Schwächen auf: Trotz einer großen Fülle von Freiheitsgraden können immer nur zwei Parameter bei der statistischen Auswertung gegeneinander geprüft werden[25]. Zudem ist die Interpretierbarkeit der Ergebnisse und deren Übertragung auf reale Anwendungen schwierig[26]. Umgekehrt sind domänenabhängige Testbetten mit einer komplexeren

24. Tileworld: [Pollack, Ringuette 1990]; MICE (Michigan Intelligent Coordination Experiment): [Montgomery, Durfee 1990], [Durfee, Montgomery 1989].

25. Tileworld wurde zum Testen unterschiedlicher Filterungsstrategien der IRMA-Architektur nur mit *einem* Agenten und wenigen Filterparametern eingesetzt [Pollack, et al. 1994]. In einem anderen Projekt wurde ein 2-Agenten-Szenario aufgesetzt, um den Einfluß verschiedener Parameter (Prozessorressourcen, Mitteilungsfreudigkeit, Fehlertoleranz) auf eine kooperative Problemlösung zu untersuchen [Walker, Jordan 1995].

26. TRUCKWORLD: [Hanks, et al. 1993a]; PHOENIX: [Hanks, et al. 1993b].

Weltmodellierung wie TRUCKWORLD oder PHOENIX[27] stark auf bestimmte Anwendungen und Architekturen zugeschnitten.

4.7 Agentenorientiertes Programmieren als neues Programmierparadigma?

Die vorausgegangene Erörterung von Agentenprogrammiersprachen und Entwicklungsmethodologien bietet ein diffuses Bild. Anders als etwa bei den objektorientierten Techniken fehlen durchgängige Konzepte von Analyse über Design bis hin zur Implementierung. Eine wichtige Ursache dafür liegt im unklaren, aber sehr mächtigen Agentenbegriff und dem nicht einheitlichen Verständnis was „agentenorientierte Techniken“ sind. Die aktuelle Situation im Bereich der Agentenentwicklung wird durch ein Zitat von Wooldridge und Jennings[28] verdeutlicht:

> *„At the time of writing, the development of any agent system - however trivial - is essentially a process of experimentation. There are no tried and trusted techniques available to assist the developer.“*

Neben den mangelhaften Begriffskonzeptualisierungen tun sich weitere Schwierigkeiten auf, das agentenorientierte Programmieren als *eine* Methodologie zu begreifen.

- **Verteiltes Problemlösen vs. Multiagentensysteme:** Beim distributed problem solving steht ein zu lösendes Problem und damit ein zentrales, geschlossenes Systemdesign, also eine reduktionistische Sicht im Vordergrund, während multi-agent systems auf offene Systeme abzielen und einen konstruktivistischen Ansatz darstellen. Der Schwerpunkt liegt hierbei auf der Modellierung und Programmierung einzelner Agenten mit einem höheren Grad an Autonomie. DPS-Agenten sind tendenziell kooperativ, während MAS-Agenten eher konkurrieren.
- **Wahl einer geeigneten Architektur:** Spezifische Anwendungsgebiete erfordern spezielle Agentenarchitekturen. Es fehlt jedoch an Metriken oder Benchmarks, um für ein Anwendungsgebiet eine passende Architektur zu finden. Kaum eine Architektur ist in der Praxis für mehrere Anwendungen wiederverwendet worden, stattdessen werden immer neue Architekturen - teils auch durch Anpassung bestehender - entwickelt.
- **Es fehlen Benchmarks**, mit denen das Verhalten eines Agentensystems beurteit oder unterschiedliche Ansätze verglichen werden könnten. Auch existieren kei-

27. [Cohen, et al. 1989], [Hart, Cohen 1990].

28. [Wooldridge, Jennings 1998], pitfall 4.3: You forget you are developing software.

ne Konzepte zum Verbessern oder Trimmen der Performance oder Systemkohärenz: Zu groß ist die Zahl der möglichen Einflußfaktoren wie Zahl der Agenten, Art und Dauerhaftigkeit des Wissens, Kommunikationsinfrastruktur, etc.

Desweiteren sind viele Probleme der Integration einzelner Techniken bis heute ungelöst. Diese lassen sich im wesentlichen durch folgende Dichotomien ausdrücken:

- **Agentensicht vs. Systemsicht:** Makrolevel-Aspekte wie Systemdekomposition und globale Koordination lassen sich nur schwer mit den Mikrolevel-Aspekten wie lokales Handeln und einfache Sprechaktkommunikation verknüpfen. Wie kann ein Agent globale Ziele erkennen, bzw. wie sichere ich zu, daß alle Agenten auf ein spezifiziertes Ziel hin arbeiten?
- **Handeln vs. räsonnieren:** Reaktive Agenten mit prozeduralen Fähigkeiten haben Echtzeitfähigkeit unter realen Bedingungen erwiesen, jedoch führt blindes Handeln keineswegs immer zum Ziel. Für flexibles, intelligentes Handeln ist im allgemeinen Wissensverarbeitung notwendig. Allerdings stellt Wissensmanagement im Kontext wechselhafter, offener und unscharfer Umgebungen eine unlösbare Aufgabe dar. Auch die Frage, wann - und in welchem Maße - ein Agent, anstatt ein bekanntes Handlungsmuster zu aktivieren, planen, lernen oder Entscheidungsprozeduren anstoßen soll, ist nicht einfach zu beantworten.
- **Handeln vs. kommunizieren:** Kommunikation ist langsam im Vergleich zur CPU-Nutzung, jedoch kann eine Aufgabendelegierung bei hoher Eigenlast Zeitersparnisse bringen. Wie soll ein Agent ohne genaue Kenntnis der Kommunikations- und CPU-Nutzungskosten und Leistungsfähigkeit anderer Agenten entscheiden, wann er eine Aufgabe delegiert?
- **Theorie vs. Praxis bei der Modellierung mentaler Zustände:** Theoretische Grundlagen für die Repräsentation mentaler Zustande in Form nichtmonotoner, zumeist epistemischer Logiken existieren zwar, jedoch ohne ein akzeptables Verarbeitungsmodell. Konkrete BDI-Architekturen realisieren nicht einmal diese einfachen Axiomatisierungen, sondern stellen „pragmatische“, reaktive Ansätze dar. Ein einheitliches Rahmenwerk, das mentale Begriffe, Zeit, unsicheres und konfliktierendes Wissen, Kommunikation und Handeln vereint, ist nicht in Sicht.

Die hier diskutierten Punkte und die vorher behandelten unterschiedlichen Ansätze zur Agenten- bzw. Systemmodellierung zeigen auf, daß Anspruch und Wirklichkeit der agentenorientierten Techniken auseinander klaffen. Von einem neuen Programmierparadigma sollte wegen der fehlenden Durchgängigkeit von Analyse, Design und Implementierung besser nicht gesprochen werden.

4.8 Der REkoS-Beitrag zum agentenorientierten Programmieren

Gegenstand dieser Arbeit ist die Konzeption und Implementierung einer integrierten Entwicklungsumgebung für diensterbringende Agenten. Das Anwendungsgebiet trägt sowohl Merkmale der Multiagentensysteme als auch des verteilten Problemlösens: Von den multi-agent systems kommen konstruktivistische Verfahren zur *dynamischen* Bildung komplexer Dienste auf Basis einfacher Fähigkeiten zur Anwendung, DPS steuert Methoden zur *statischen* Dienstegenerierung nach dem top-down-Prinzip zu. Für den erstgenannten Fall sind Agenten mit dem Schwerpunkt auf Autonomie, losgelöst von einem globalen Systemgedanken, zu modellieren, während für den anderen Ansatz Mechanismen für globale Zielverfolgung und Koordination benötigt werden.

Basis des gewählten Ansatzes bildet eine Agentenarchitektur, die charakteristische dienstspezifische Anforderungen in Form bereitgestellter generischer Fähigkeiten abdeckt. Über der Architektur liegt die Agentenprogrammiersprache. Sie enthält Konzepte zur Spezifikation von Diensten, zur Definition des mentalen Modells und der interaktiven Fähigkeiten eines Agenten. Graphische Definitionswerkzeuge bilden die Schnittstelle zum Programmierer, weitere Integrationswerkzeuge unterstützen den Prozeß der Agenten- und Systemkreierung. Eine ebenfalls werkzeuggestützte Test- und Debuggingumgebung hilft bei der Aufdeckung von Fehlern und dem Monitoring wichtiger Laufzeitdaten.

Neu am REkoS-Ansatz ist die Idee, eine im Vergleich zu existierenden Entwicklungsumgebungen mächtige Architektur mit einer Agentenprogrammiersprache zu paaren und damit die Agentenentwicklung auf eine noch höhere Ebene anzuheben. Die Verwendung von Werkzeugen bei der Programmierung, Generierung und Überwachung von Agenten führt zu einer Reduzierung des Gestaltungsaufwands auf die wesentlichen anwendungsspezifischen Aspekte. Darüberhinaus stellen die Integrations-Tools ein Rahmenwerk zur Vorgehensweise bei der Erstellung einzelner Agenten und Agentenverbänden bereit.

Die Herausforderungen an eine derartige Agentenentwicklungsumgebung sind vielfältig: Zum einen ist eine tendenziell reaktive Architektur zu entwickeln, die bereits generische Dienstmanagementfunktionalität bereitstellt. Eine derartige Architektur ist erheblich komplexer als die bislang in Verbindung mit Agentenprogrammiersprachen verwendeten Modelle. Zum andern sind lokale Kompetenz und Kooperationsformen geeignet miteinander zu verknüpfen. Keiner der beschriebenen Programmierumgebungen leistet dies zufriedenstellend. Schließlich müssen Dienste sowohl deklarativ beschreibbar als auch effizient ausführbar und dabei managebar sein.

4.9 Zusammenfassung und Bewertung

Agentenprogrammiersprachen basieren im Vergleich zu herkömmlichen Programmiersprachen auf einer mächtigeren Verarbeitungsmaschine, der Agentenarchitektur. Die Verarbeitung von Informationen findet auf höherer Ebene statt; anstelle von Speicherplatz, einfachen und abstrakten Datentypen sind Konstrukte wie Ziele, Sprechakte und Fähigkeiten Gegenstand der Programmierung. Die in diesem Kapitel behandelten Sprachen sind ausnahmslos systemzentriert und damit dem Prinzip des verteilten Problemlösens verpflichtet, jedoch betonen sie unterschiedliche Aspekte der agentenbasierten Systementwicklung und bieten divergierende Sichten auf das Konzept von Agenten.

Agent-0 und dMARS basieren auf reaktiven BDI-Architekturen. Agent-0 bietet logische Repräsentation, einen einfachen Sprechaktmechanismus und regelbasierte Programmierung. Demgegenüber ist dMARS stärker prozedural ausgerichtet: Agentenprogrammierung wird hauptsächlich als Planspezifikation verstanden, sprechaktbasierte Kommunikation wird jedoch nicht unterstützt. Beide Sprachfamilien verstehen Agenten als nachrichtenverarbeitende aktive Objekte. Entsprechend definiert dMARS einen stark an objektorientierte Softwarentwicklung angelehnten Analyse- und Designprozeß.

Während in den BDI-Sprachen - trotz eines top-down-Systemdesigns - die Realisierung der lokalen Abläufe innerhalb eines Agenten im Vordergrund stehen, konzentriert sich die Agentenentwicklung in COOL auf die Modellierung des Systemverhaltens. Agentenprogrammierung wird hier als inkrementeller Prozeß zur Gestaltung von Interaktionsprotokollen verstanden. Agenten bestehen aus einer Menge von Konversationsregeln und sind durch die fehlende mentale Verarbeitungsebene noch schlanker als BDI-Agenten.

Der Beitrag von TÆMS zur AOP besteht in der Bereitstellung einer Beschreibungssprache zum Ausdruck von Koordinationsbeziehungen zwischen Taskstrukturen. Der Ansatz ist aufgabenzentriert und beinhaltet keine Agentenprogrammierung im eigentlichen Sinne. Agenten sind die ausführenden Instanzen der Taskstrukturen, dazu beherrschen sie Mechanismen zur Nutzung der Koordinationsbeziehungen und zur Auflösung auftretender Ressourcenkonflikte. Auch hier findet keine sprechaktbasierte Kommunikation statt, anstelle dessen wird taskspezifisches Wissen ausgetauscht.

Offensichtlich beruhen verfügbare Agentenprogrammiersprachen auf sehr einfachen Agentenmodellen. Die Ursache hierfür liegt in der schwierigen Integration so unterschiedlicher Konzepte wie Reaktivität, Deliberativität, Interaktionsprotokolle und lokales Handeln. Einer agentenorientierten Programmiermethodologie stehen die fehlende Operationalisierbarkeit von Agentenspezifikationssprachen sowie mangelnder theoretischer Unterbau derzeit im Wege.

Bislang verfügbare Agentenprogrammiersprachen eignen sich nicht für den Anwendungsbereich der kooperativen Dienste. Die Architekturen sind generell zu schwach, die Methodologien und Sprachen unterstützen die benötigte agentenzentrierte Sichtweise zu wenig. Umgekehrt bieten kognitive Agentenarchitekturen, deren Schwerpunkt auf Intelligenz und Autonomie liegt, wenig Unterstützung in Bezug auf das Software-Design und die Programmierung - zumeist kommen schwache KI-Repräsentationsformen zum Einsatz. Diese Gründe rechtfertigen die Realisierung einer neuen Architektur, Programmiersprache und Entwicklungsmethodik für den Anwendungsbereich agentenbasierter Dienste.

Teil B

REkoS

Realisierung von Werkzeugen zur Entwicklung von kooperierenden Diensten für kommunikationsbasierte Systeme

KAPITEL 5 Sprachen und Definitionswerkzeuge in REkoS

Dieses Kapitel beschreibt den REkoS-Ansatz zur agentenorientierten Programmierung. Die verschiedenen Aspekte agentischen Handelns werden durch vier Beschreibungssprachen abgedeckt, die durch grafische Definitionswerkzeuge und Editoren zu einer Entwicklungsumgebung integriert werden.

Im ersten Abschnitt werden dienstspezifische Merkmale erläutert. Danach werden die für die Programmierung von REkoS-Agenten wichtigen Begriffe beschrieben. Die Abschnitte 3 bis 6 behandeln die Beschreibungssprachen. Ferner werden dort graphische Eingabewerkzeuge vorgestellt, die zur Unterstützung der Programmierung entwickelt wurden. Abschnitt 7 ist der Codegenerierung und Integration der Werkzeuge zu einer Entwicklungsumgebung gewidmet, bevor im letzten Abschnitt die Ergebnisse zusammengefaßt und bewertet werden.

5.1 Der REkoS-Ansatz

Wie das vorige Kapitel aufgezeigt hat, sind Architektur und Programmiersprache nicht unabhängig voneinander zu sehen. Vielmehr setzt die Programmierung auf der von der Architektur vorgegebenen Informationsverarbeitung und den bereitgestellten generischen Fähigkeiten auf, beziehungsweise wirken sich die Aspekte der Programmiersprache auf Designentscheidungen der Architektur aus. Aufgrund dieser Verflechtung können Anforderungen entweder als grundlegende Fähigkeiten von

der Architektur realisiert, oder durch die Programmiersprache abgedeckt werden. Die wesentlichen Anforderungen an die REkoS-Agentenprogrammiersprache betreffen Dienstbeschreibung, Interaktionen und zielgerichtetes Verhalten:

- Dienstbeschreibung: Die Programmiersprache soll die Beschreibung komplexer Dienste ermöglichen. Wichtig ist auch die Spezifikation der Nutzungsschnittstelle und Parameter, mit der eine kooperative Diensterbringung erst möglich wird. Damit ein Aktivitätenmanagement möglich wird muß die Sprache modular aufgebaut sein. Effiziente Verarbeitung kann durch Codekompilierung anstelle einer quellcodebasierten Interpretation erreicht werden.
- Interaktionen: Das Anbieten und Anfordern von Diensten sollte wegen der Offenheitsanforderung sprechaktbasiert erfolgen. Darüberhinaus müssen Agenten Dienste auch in Kooperation erbringen können. Zur Koordination dieser gemeinsamen Tätigkeiten sind Interaktionsprotokolle notwendig.
- Zielgerichtetes Verhalten: Das dynamische Verhalten eines Agenten soll von Zielen gesteuert sein. Intentionale Attitüden dienen der Transparenz und erlauben eine bessere interne Koordination. Diese zusätzliche Repräsentationsebene sollte ein verständliches Verarbeitungsmodell anbieten, das von der Architektur effizient umsetzbar ist[1].

Die Agentenprogrammiersprache muß ferner ein Konzept zur Integration von Diensten, Sprechakten und Interaktionsprotokollen bieten. Managementaufgaben sollen sind nicht Gegenstand der Programmierung sein sondern gehen als Anforderungen in das Architekturdesign ein, das im nächsten Kapitel beschrieben wird. Zur Berücksichtigung der oben genannten Anforderungen stellt REkoS vier Beschreibungssprachen bereit:

- Mit der *Kommunikationssprache* werden Sprechakte definiert, die die Basis-Interaktionsfähigkeiten der Agenten ausmachen. Mit ihrer Hilfe können Agenten Dienste anfragen und anbieten sowie Ergebnisse übermitteln.
- Mit einer sprechaktbasierten *Beschreibungssprache für Interaktionsprotokolle* lassen sich Verhandlungen und Kooperationen modellieren. So können Agenten mit Contract-Net-ähnlichen Protokollen Dienste kooperativ und flexibel durch Aufgabenverteilung erbringen; anwendungsspezifische Protokolle ermöglichen die Lösung beliebiger globaler Koordinationsprobleme.
- Eine *Sprache zur Beschreibung der Intentionalität* definiert mentale Zustände und leitet daraus zielgerichtetes Verhalten ab. Mit der Gestaltung der intentionalen Verhaltensebene steuert der Programmierer die Abläufe innerhalb eines Agenten und legt die Verknüpfung von Wahrnehmung und Handeln fest.
- Eine *Skriptsprache zur Beschreibung von Fähigkeiten* wurde entwickelt, um auf abstraktem Niveau die spezifischen Fähigkeiten eines Agenten programmieren

1. Zur Problematik der Verarbeitung mentaler Modelle siehe Abschnitt 2.4.6.

zu können. Sie enthält deklarative Beschreibungsmerkmale, mit denen Dienste verglichen werden können, unterstützt die Dekomposition von Diensten in Teildienste und bietet verschiedene Mechanismen zur flexiblen und dynamischen Diensterbringung an.

Für alle vier Beschreibungssprachen ist Werkzeugunterstützung sinnvoll, um einerseits ein schnelles und fehlerarmes Erfassen zu gewährleisten und andererseits Modellwartung und Codegenerierung durchzuführen. Weiterhin werden Tools zur Integration der einzelnen Sprachen und zur Erzeugung von Agenten mit grafischen Oberflächen benötigt.

5.2 REkoS-Glossar

In diesem Abschnitt werden die wichtigsten Begriffe der REkoS-Dienstewelt eingeführt, die Gegenstand des agentenorientierten Programmierens sind. Sie lassen sich in zwei Kategorien einteilen: Interaktionsprotokolle, Skripte, Methoden, Aktionen und Sprechakte sind ausführbare Elemente und werden auch als *Executables* bezeichnet. Demgegenüber zählen die intentionalen Attitüden Variable, Motivation, Ziel, Task und Wahrnehmung zu den passiven, aktivierbaren *Behaviours*.

5.2.1 Intentionalität

Die Intentionalität dient zur Modellierung deliberativen Verhaltens. Der hier eingeschlagene Weg einer Beschreibung mentaler Zustände mit nur wenigen Begriffen und ohne darunterliegende mächtige Repräsentationsformalismen kommt den BDI-Architekturen recht nahe.

- Ziele, auch Zustandsziele oder goals genannt, sind die Basis für zielgerichtetes Verhalten. Ein Ziel beschreibt einen Zustand, der erreicht werden soll. Dies geschieht üblicherweise durch die erfolgreiche Abarbeitung von Diensten (tasks). Ist der Zielzustand erreicht, erlischt das Ziel.
- Motivationen[2] (motivations) steuern die Zielgenerierung. Ihre Aktivierung und Deaktivierung hängt vom Zustand der Wissensbasis ab. Im Gegensatz zu Zielen erlöschen sie nicht automatisch durch Erreichen der abhängigen Ziele. Damit kann ein Agent zu wiederholten ähnlichen Verhaltensmustern angeleitet werden.

2. Motivationen wurden im REkoS-Kontext ursprünglich „Intentionen“ genannt. Weil letztgenannter Begriff im übrigen Agentenumfeld mit einer anderen Semantik belegt ist (siehe Abschnitt 2.4.2 „Modallogiken“ auf Seite 59), würde diese Verwendung zu Irritationen führen.

- Normative Ziele (normative goals) lösen keine Handlungen aus, sondern haben steuernden Charakter. Ein normative goal übt Einfluß über die Art und Weise aus, mit der ein Agent Dienste erbringt. So können Agenten zu kostengünstiger oder schneller Diensterbringung gebracht werden.

5.2.2 Interaktivität

- Sprechakte (speech acts) sind KQML-ähnliche (siehe Abschnitt 2.2.1) Nachrichtenprimitive. Mit ihnen kommunizieren Agenten untereinander.
- Wahrnehmungen (sensings) beschreiben Ressourcenänderungen, die in der Umwelt stattgefunden haben. Damit ist dem Agenten eine Möglichkeit zur Kommunikation mit nicht-agentischen Entitäten, wie beispielsweise Datenbanken gegeben.
- Aktionen (actions) sind das Gegenstück zu sensings. Sie bewirken Änderungen an Ressourcen, die außerhalb des Agenten liegen.
- Interaktionsprotokolle beschreiben zielgerichtete, längerfristige Interaktionen zwischen mehreren Agenten (siehe Abschnitt 2.3.6). Sie erweitern das einfache, auf Nachrichtenaustausch reduzierte Sprechaktmodell um komplexere Dialogformen.

5.2.3 Aktorische Fähigkeiten

- Aufgaben (tasks) sind der REkoS-Begriff für Dienste. Sie stellen deklarative Beschreibungen von Leistungen dar, drücken also aus, *was* erbracht werden kann.
- Skripte, auch scripts oder Handlungsskripte genannt, definieren die prozeduralen Fähigkeiten eines Agenten. Sie sind modulare, einfach strukturierte Handlungsbeschreibungen zur Implementierung von Tasks, beschreiben also *wie* etwas getan wird.
- Methoden (methods) realisieren elementare Fähigkeiten von Agenten. Anders als Skripte sind Methoden nicht weiter verfeinerbar.

5.2.4 Globaler und lokaler Wissenskontext

Für die Verständigung ist es notwendig, daß die Agenten auf derselben Begriffswelt operieren. Dies gilt nicht nur für die Nutzung derselben generellen Konzepte, sondern teilweise auch für deren konkrete Ausprägungen inklusive Semantik. So nützt es einem Agenten wenig zu wissen, daß ein anderer gerade ein Ziel x verfolgt, wenn er nicht eine Vorstellung von x hat. Auf der anderen Seite besitzen Agenten auch lokales Wissen und Fähigkeiten, das anderen Agenten verborgen bleibt.

Eine Differenzierung von lokalen und globalen Wissensstrukturen hat Einfluß auf die Werkzeuggestaltung für die Programmierung. Von mehreren Agenten geteiltes Wissen und gemeinsam genutzte Konzepte sollten entsprechend global, d.h. nur einmal erfaßt werden. Demgegenüber sind agentenlokale Fähigkeiten für jeden Agenten einzeln zu spezifizieren. Die folgende Tabelle zeigt auf, welche Instanzen der Rekos-Begriffswelt von allen Agenten einer Domäne oder Anwendung geteilt werden müssen und welche Terme rein lokale Bedeutung besitzen.

TABELLE 2. Sichtbarkeit der REkoS-Begriffswelt

Begriff	**lokale Verwendung**	**globale Verwendung**
Motivation	X	
Ziel	X	X
Task	X	X
Skript	X	
Methode	X	
Wahrnehmung	X	
Aktion		X
Sprechakt		X
Interaktionsprotokoll		X

Von zentraler Bedeutung für REkoS-Dienstanwendungen sind die Begriffe Task, Sprechakt und Interaktionsprotokoll; ihre Bedeutung und Verwendung muß agentenübergreifend eindeutig und klar sein. Im Falle des Task-Konzepts ergibt sich die Notwendigkeit einer allgemeinen Semantik aus dem dahinterstehenden Dienstgedanken. Dienste können nur sinnvoll zwischen Agenten vermittelt oder von Agenten angeboten werden, wenn Einigkeit in Bezug auf die Bedeutung des Dienstes herrscht. Entsprechend wird nicht für jeden Agenten ein eigenes Task-Modell gebildet, sondern Task-Spezifikationen werden von mehreren Agenten geteilt. Analoges gilt für Sprechakte und Interaktionsprotokolle. Beide Konzepte dienen dem Austausch und der Koordination zwischen Agenten und müssen von allen Beteiligten gleichermaßen verstanden werden. Hingegen stellen Skripte, Methoden, Aktionen und Motivationen prozedurales bzw. lokales Wissen dar und sind nicht Gegenstand der Kommunikation. Dementsprechend findet die Modellierung dieser Konzepte für jeden Agenten einzeln statt.

Einen Sonderfall bilden die Ziele. Oft ist es sinnvoll, wenn sich Agenten gegenseitig über ihre aktiven oder möglichen Ziele informieren. Ziele stellen gegenüber den Tasks abstraktere und mehr in die Zukunft gerichtete Informationen dar und können zur Erkennung von Konflikten und Synergiepotentialen auf hohem Level herangezogen werden. Auf der anderen Seite bilden Ziele das Bindeglied zwischen lokalem Wissen eines Agenten und dem Handeln. Aktivierung von Zielen und Ableiten von Maßnahmen zu deren Erfüllung sind dementsprechend inhärent lokaler Natur. Diese Dualität zwischen eindeutigem, globalem Namensraum und lokalen, intransparenten semantischen Fundierungen schafft ein potentielles Interpretationsproblem: Zwar können Agenten Informationen über ihre Ziele austauschen, nicht jedoch über deren Konsequenzen. Hierbei handelt es sich um eine Ausprägung der bekannten Schnittstellenproblematik: Aus Gründen der notwendigen Kapselung, Modularität oder auch nur Übersichtlichkeit erfolgt die Beschreibung einer Schnittstelle nur abstrakt auf syntaktischer Ebene und ist von der konkreten Ausgestaltung entkoppelt.

5.3 Definition von Sprechakten

Agentenkommunikation in REkoS orientiert sich an der Theorie der Sprechakte. Da sich die Kommunikationssprache in Teilen jedoch signifikant von existierenden Agentenprogrammiersprachen unterscheidet, wird im folgenden Unterabschnitt zunächst das grundlegende Gedankenmodell vorgestellt. Anschließend erfolgt eine Beschreibung der Struktur von Sprechakten, gefolgt von einer Vorstellung der verwendeten Ontologie und schließlich die Beschreibung des Sprechakt-Definitionswerkzeugs. Die Verwendung von Sprechakten folgt aus Skripten heraus, in Interaktionsprotokollen und in Ausnahmesituationen, beispielsweise wenn ein Agent eine benötigte Funktionalität selbst nicht bereitstellt. Technische Informationen zur Übermittlung und Verarbeitung von Sprechakten finden sich in der Beschreibung des Kommunikationsmoduls in Abschnitt 6.9.

5.3.1 Das Sprechaktmodell

Sprechaktbasierte Kommunikationssprachen wie KQML oder FIPA-ACL sind für offene Systeme konzipiert. Die Ebene der Performatives hat den Anspruch den gesamten Bereich der menschlichen (illokutionären) Kommunikation abzudecken. Inhalte können durch Verwendung mächtiger Wissensrepräsentationssprachen wie KIF oder Prädikatenlogik in beliebiger Komplexität und Umfang formuliert werden; eine Einschränkung findet nur über die verwendbaren Begriffe, die die Ontologie vorgibt, statt. Eine Sprache für Dienste, die diesen Generalitätsanspruch nicht

hat, kann entsprechend auf die notwendigen Aspekte hin maßgeschneidert und von unnötigem Ballast befreit werden.

Sprechakte in REkoS haben, anders als KQML, eine wohldefinierte Semantik, die auch noch über die prädikatenlogischen Beschreibungen der Effekte des FIPA-Ansatzes hinausgehen. Es werden sowohl die Auswirkungen auf den sendenden Agenten hat wie auch die Wirkung beim Empfänger beschrieben. Insofern ähnelt der hier beschrittene Weg der planbasierten Kommunikation, die in Abschnitt 2.2.1 beschrieben wurde. Die Beschreibung der Semantik sowohl für die Sender- als auch die Empfängerseite bietet den Vorteil einer transparenten, eindeutigen Interpretation für beide Parteien. Dem Sender eines Sprechakts ist im Groben klar, welche Auswirkungen der Akt auf den Empfänger hat. Umgekehrt kennt auch der Empfänger einer Nachricht die Konsequenzen, die sich für den sendenden Agenten aus der Übermittlung ergeben.

Auf der anderen Seite kann auch durch das vorgegebene Konzept eine "korrekte" (in bezug auf die intendierte Semantik) Verarbeitung von Sprechakten nicht garantiert werden. Es ist jedem Agenten - oder besser gesagt, jedem Agentenprogrammierer - überlassen, wie er die Sende- und Empfangsprozeduren implementiert. Problematisch ist die Kommunikation mit nicht REkoS-konformen Agenten. Aufgrund der operationalen Semantik legt ein REkoS-Agent u.U. mehr „Bedeutung“ in eine Nachricht, als dies sein Kommunikationspartner tut. Daher können Sende- und Empfangssemantik als optionale Attribute von Sprechakten angesehen werden und bilden deshalb auch kein generelles Hindernis zur Kommunikation mit anderen Agentenwelten.

Die REkoS-Kommunikationssprache ist wie KQML offen für die Ergänzung um neue Sprechakte. Es liegt in der Natur der Domäne, daß diensterbringende Agenten mit einer relativ kleinen Menge von Sprechakten auskommen. Für alle Dienstszenarien typisch sind Nachrichtenprimitive, die Anfragen nach und das Bekanntgeben von Informationen sowie Dienstanfragen und -beauftragungen abdecken. Auf der anderen Seite können spezielle Anwendungen sehr wohl weitergehendere kommunikative Fähigkeiten verlangen. Beispielhaft seien hier Netzwerkmanagement- und Datenbankanwendungen genannt, für die KQML eigene Performative-Klassen vorsieht. Weil die Welt der Dienste vielfältig und schwer überschaubar ist, liegt es nah, für relativ abgeschlossene und homogene Teilbereiche eigene Kommunikationsmodelle aufzubauen. In REkoS wird dies durch die Möglichkeit, Sprechaktbibliotheken zu erstellen, unterstützt. Auf diese Weise können die kommunikativen Fähigkeiten für einzelne Agenten oder auch ganze Anwendungsbereiche maßgeschneidert werden.

5.3.2 Attribute eines Sprechakts

Bei der Definition eines Sprechakts werden dessen Aufbau und Interpretierbarkeit beschrieben; Nachrichteninhalt sowie Adressierungsinformationen werden erst bei der Nutzung entsprechend spezifiziert.

Inhaltliche Ebene

Typ und Inhalt einer Nachricht werden durch die vier Attribute Performative, Operator, Content-Typ und Content beschrieben. Die Performative ist deckungsgleich mit dem gleichnamigen Konzept in KQML. Ein wesentlicher Unterschied liegt jedoch in den drei letztgenannten Attributen, die einerseits die Inhaltsebene abdecken und andererseits die Ontologie beinhalten.

Operator und Content-Typ dienen als Strukturierungsmittel für eine weitere Typisierung der Nachricht. Der Operator wird wie ein (Hilfs-) Verb gebraucht und bestimmt die Verwendung des Content-Typen, welches ein Wort aus der REkoS-Terminologie ist. Als Content werden schließlich konkrete Instanzen des Content-Typen verwendet. So entstehen Quasi-Sätze wie „ask_if can task backup(work.txt, size(400, k))“ oder „tell is_active goal information_gathering(topic(X))“[3]. Die Terminologie wird unten in Abschnitt 5.3.3 näher beschrieben.

Semantische Ebene

Zwei Attribute, sender_function und receiver_function, beschreiben, wie ein Sprechakt vom Sender bzw. vom Empfänger zu verarbeiten ist. Dadurch wird der Handlungsaspekt der Kommunikation modelliert[4]. Diese Attribute können unspezifiziert sein, womit ausgedrückt wird, daß keine Semantik vorgegeben wird. Die semantische Ebene ist nicht Bestandteil der übermittelten Daten eines Sprechaktes, sondern ist in der Wissensbasis eines Agenten angesiedelt.

Transport- bzw. Kommunikationsebene

Beim Versenden eines Sprechaktes wird der sender-Slot automatisch mit dem Namen des sendenden Agenten ausgefüllt. Der Adressat (receiver) ist entweder ein zu benennender Agent, eine Liste von Agentennamen oder *broadcast*, womit ausgedrückt wird, daß die Nachricht an alle erreichbaren Agenten geht[5].

---

3. Oder, um die Beziehung zwischen den Attributen besser auszudrücken, in Prädikatenschreibweise: ask_if(can(task(X))) bzw tell(is_active(goal(Y))).

4. Dieselbe Sichtweise findet sich in der planbasierten Kommunikation (Abschnitt 2.2.1) und bei [Cohen, Levesque 1995].

Eine Reihe von Slots regeln die (optionale) Ergebnisrückgabe. Prinzipiell besteht die Möglichkeit mit und ohne erwartete Rückantwort zu versenden, wobei Antwortmuster vorgegeben werden können. Auch eine Pufferung von Antwortströmen wird unterstützt; blockierende und nicht-blockierende Mini-Dialoge sind demnach möglich.

Sprechakten kann ein Timeout mitgegeben werden, der angibt, bis wann der sendende Agent eine Rückantwort spätestens erwartet. Mit diesem Mechanismus wird verhindert, daß fehlerhafte blockierende Kommunikationsformen zur dauernden Unterbrechung eines Dienstes führen.

Weitere Attribute

Sprechakte, die Bestandteil eines Interaktionsprotokolls (Abschnitt 5.6) sind, tragen eine Identifizierung, um der entsprechenden Protokollinstanz eindeutig zugeordnet werden können. Weitere, nur für die Übermittlung relevante Attribute werden in Abschnitt 6.9 beschrieben.

5.3.3 Begriffswelt

Die Terminologie für REkoS-Agenten ist sehr einfach gehalten. Sie besteht aus einer kleinen Menge feststehender Konzepte, die von allen Agenten verstanden wird und weiteren frei definierbaren Begriffen. Eine umfangreiche Konzeptualisierung des (Dienst-)Gegenstandsbereichs, bestehend aus dem Aufbau eines umfassenden Wörterbuchs und der Definition von Relationen zwischen einzelnen Begriffen, wurde nicht durchgeführt. Daher ist der Begriff „Ontologie“ im REkoS-Kontext nicht angebracht.

Fest vorgegebenen ist der Content-Typ, der eine Menge von Begriffen umfaßt, die auch intern von der Architektur als Entitäten zur Informationsdarstellung genutzt werden. Es handelt sich hierbei um die oben in Abschnitt 5.2 eingeführten deklarativen Begriffe task, goal, action und sensing, wobei goal sowohl die zustandsorientierten als auch die normativen Ziele umfaßt. Hinzu kommt noch knowledge für beliebige Einträge der Wissensbasis. Damit ist jeder Agent in der Lage, die genannten Begriffe zum Gegenstand der Kommunikation zu machen. Die gleichartige Verwendung dieser generischen Behaviours hilft aufgrund derselben Architektur bzw. Informationsverarbeitung Mißverständnisse in bezug auf die Interpretation auszuschließen.

5. Broadcast wird also zur Adressierung verwendet und tritt nicht als separater Sprechakt, wie etwa in KQML auf.

Operatoren sind im Prinzip frei wählbar. Ihre Aufgabe ist es zwischen der Performative und dem Behaviour eine weitere Bedeutungsebene einzuführen. Während die Performative den illokutionären Typ eines Sprechakts beschreibt, definiert der Operator die Verwendung oder Interpretation des Nachrichteninhalts. Im Vergleich zu KQML, wo alles außer der Performative in den Content gelegt wird, werden Verwendungsinformationen aus dem Nachrichteninhalt herausgezogen.

TABELLE 3. Basis-Sprechaktmodell für REkoS-Agenten

	tell	**ask**	**command**
task	can isActive isScheduled	can isActive isScheduled	do
goal	know isActive	know isActive	activate
knowledge	know	know	know
action	can	can	do
sensing	know	know	--

Tabelle 3 zeigt die grundlegenden kommunikativen Fähigkeiten von REkoS-Agenten. Dort sind jedem Paar aus Performative und Content-Typ alle wählbaren Operatoren zugeordnet. Beispielsweise können tell und task mit den drei Operatoren can, isActive und isScheduled verwendet werden. Diese Zuordnung drückt aus, daß ein Agent einen anderen Agenten über drei Aspekte eines Task-Behaviours informieren kann: daß er eine Task beherrscht (can), gerade ausführt (isActive) oder für spätere Aktivierung geplant hat (isScheduled). Der leere Eintrag unten rechts besagt, daß es keine sinnvolle Verknüpfung der command-Performative mit dem Sensing-Behaviour gibt, m.a.W. daß Sensings nicht befohlen werden können. Zusätzlich existiert noch ein sorry-Sprechakt, der als Antwort auf Fragen, die nicht beantwortet werden können und für Fehlermeldungen zur Anwendung kommt.

5.3.4 Werkzeuge zur Definition von Sprechaktmodellen

Mit dem Sprechakteditor werden Kommunikationsmodelle für Agenten erstellt und verwaltet. Auf Spezifikationsebene ist ein Sprechakt ein 6-Tupel, bestehend aus Performative, Operator, Content-Typ, ReplyBehaviour, Sender_function und Receiver_function. Die übrigen Attribute kommen erst bei der Nutzung eines Sprechaktes ins Spiel und spielen daher bei der Definition keine Rolle. Ein Kom-

munikationsmodell besteht aus einer Menge von Sprechaktdefinitionen gemäß der oben genannten Attributierung und beschreibt damit vollständig die kommunikativen Fähigkeiten eines Agenten.

Das Werkzeug erlaubt die Verwaltung mehrerer Sprechaktmodelle, so daß für verschiedene Domänen oder Agententypen eigne Kommunikationsmodelle erstellt werden können. Die frei definierbaren Attribute performative und operator werden ebenfalls von diesem Tool in separaten Editoren gemanaged. Nach dem Laden eines Sprechaktmodells werden die definierten Sprechakte, Operatoren und Performatives in separaten Fenstern angezeigt. Durch Löschen, Verändern und Hinzufügen von Einträgen können bestehende Modelle so an neue Anforderungen angepaßt oder neue Modelle generiert werden.

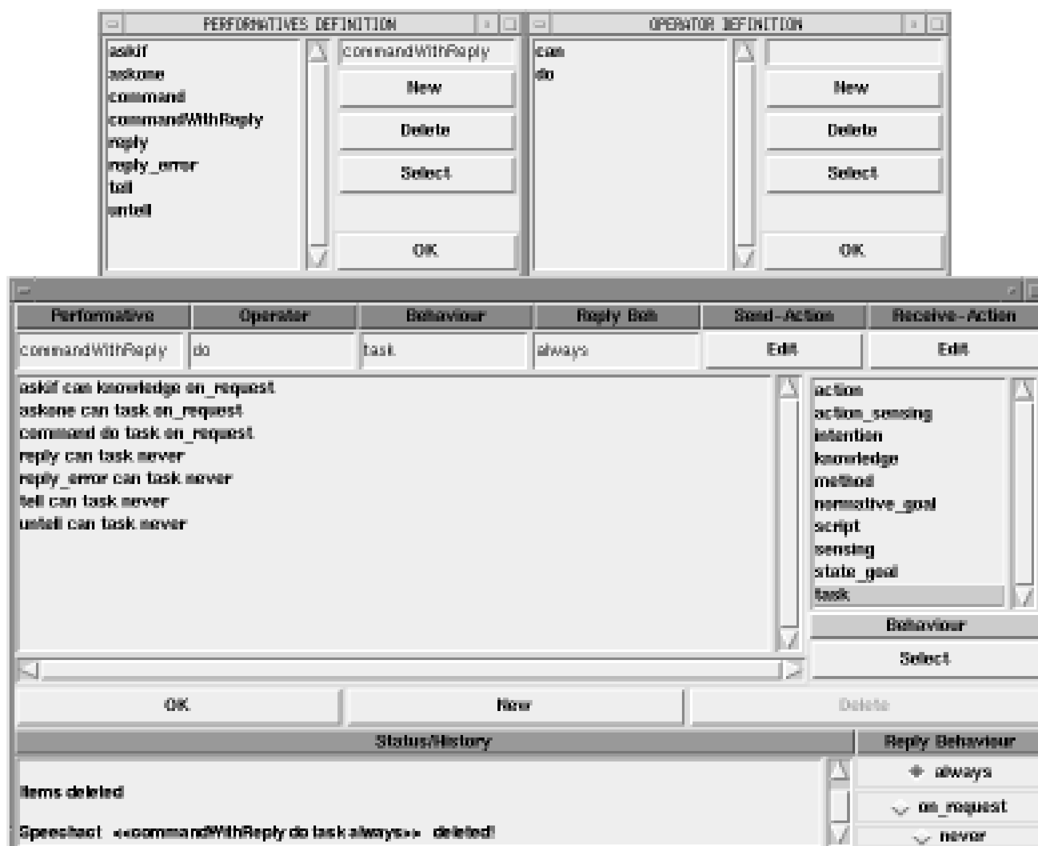

Abbildung 5-1: **Sprechaktdefinitionstool**

Ein neuer Sprechakt wird in zwei Schritten definiert. Zunächst wird dessen syntaktische Hülle durch Auswahl einer Performative, eines Operators und eines Content-

Typs erstellt. Die Auswahl des replyBehaviour bestimmt, ob eine Antwort nicht erwartet (never), gefordert (always) oder möglich (on_request) ist. Anschließend können die sender_function und die receiver_function in separaten Editoren programmiert werden.

Sprechaktsemantik

Die Programmierung der Sprechaktsemantik in Form der sender_function und receiver_function hat in Prolog zu erfolgen. Der Editor gibt ein Template vor, das den Prozedurkopf enthält, dessen Rumpf noch implementiert werden muß. Beispiele für Implementierungen liefern die folgenden Code-Extrakte:

```
command(in, ID, do, task(SPEC), RESULT) :-
    !,
    call(SPEC).

tell(out, ID, can, task(SPEC), true) :-
    !,
    getAgentName(ID,Receiver),
    getAgentName(me,Sender),
    assert( know(Receiver,know(Sender,
            can(task(SPEC))))).
tell(in, ID, can, task(SPEC), true) :-
    !,
    getAgentName(ID,Sender),
    assert( know(Sender,can(task(SPEC)))).
tell(A,B,C,D,E) :-
    errorMsg( tell(A,B,C,D,E), 'not understood' ),
    fail.
```

Das erste Argument eines Prädikats (Prolog-Sprachgebrauch für Funktion) zeigt an, ob es sich um die Sende- (out) oder Empfangsfunktionalität (in) handelt. Im zweiten Argument ist der Kommunikationspartner enthalten, er wird automatisch eingetragen. Argument drei und vier enhalten die Signatur und den Nachrichteninhalt, das fünfte Argument ist für die aufrufende Stelle zur Ergebnisübergabe vorgesehen. Im obigen Beispiel wird die Verarbeitung eines „command do task“ auf Empfängerseite einfach durch Aufruf der Task realisiert. Beim tell-Sprechakt vermerkt der Sender, daß der Empfäger über die eigene übermittelte Task-Fähigkeit Bescheid weiß. Auf der anderen Seite speichert auch der Empfänger die kommunizierte Fähigkeit des Absenders. Eine dritte Klausel dient zur Behandlung von Fehlerfällen. Statt der Ausgabe einer Fehlermeldung wäre hier auch eine Antwort (im RESULT-Parameter) in Form eines sorry-Sprechakts mit dem empfangenen tell-Content möglich.

5.4 Beschreibung des mentalen Zustands

Der mentale Zustand eines REkoS-Agenten wird durch Variablen, Motivationen, Zustandsziele, normative Ziele und Tasks beschrieben. Dabei werden die Zusammenhänge zwischen den einzelnen Entitäten in sogenannten Aktivierungshierarchien spezifiziert. Damit besitzt jeder Agent ein in seiner Struktur fest vorgegebenes mentales Modell. Vom Grundgedanken her ähneln Zweck und Verarbeitung des mentalen Zustands den BDI-Agenten aus Abschnitt 3.6: Seine Aufgabe ist es, aus dem aktuellen Wissen und empfangenen Informationen sinnvolle neue Tätigkeiten abzuleiten; für die Realisierung sind schnelle Mechanismen wichtig.

Die Implementierungssprache der intentionalen Attitüden ist Prolog. Diese Sprache wurde gewählt, weil sie als Wissensrepräsentationssprache mit integriertem Inferenzmechanismus und datenbank-ähnlichen Wissensmanagementfähigkeiten Vorteile gegenüber anderen Programmiersprachen bietet.

Abbildung 5-2 zeigt die intentionalen REkoS-Begriffe anhand einer Aktivierungshierarchie. Motivationen werden durch Variablen gesteuert und sorgen für die Aktivierung von Zielen, die wiederum zur Auswahl von Aufgaben führen. Aktive Tasks werden durch Skripte realisiert. Bei allen Abhängigkeiten handelt es sich um 1:n-Beziehungen, wodurch eine Motivation einen Baum von Goals, Tasks und Skripten aufspannen kann.

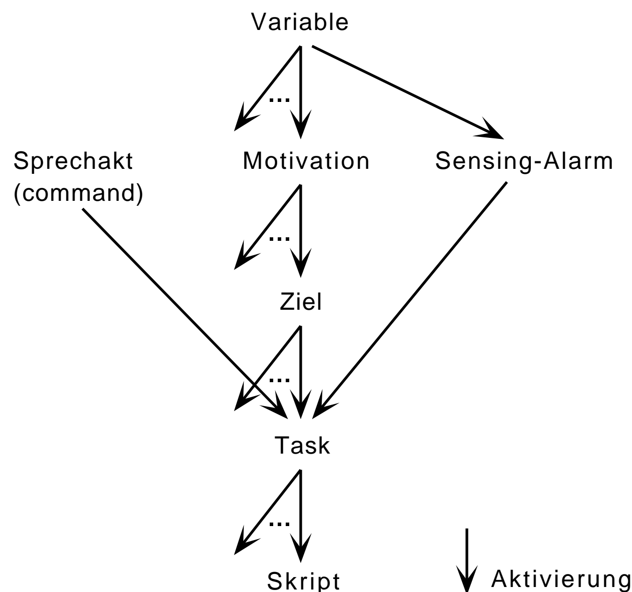

Abbildung 5-2: Aktivierungshierarchie der mentalen Begriffe

Mit Ausnahme der Task-Skript-Beziehungen handelt es sich um UND-Verknüpfungen. Dies bedeutet, daß ein Ziel sämtliche abhängigen Tasks aktiviert. Demgegenüber stellt jedes Skript eine Lösungsalternative für die übergeordnete Task dar. Die hier nicht aufgeführten normative Ziele haben eine steuernde Funktion, sie sind für die Auswahl von Skripten zuständig.

Neben diesen das zielorientierte Handeln eines Agenten modellierenden mentalen Begriffen existieren Möglichkeiten, einen Agenten unmittelbar zu Aktionen zu bewegen. Eine solche auf reaktives Verhalten abzielende Maßnahme ist der durch eine Wahrnehmung mittels einer Variablen ausgelöste sensing-alarm. Er führt zur direkten Aktivierung einer Task. Eine weitere Möglichkeit bieten Sprechakte wie command, deren Empfangsroutine - eventuell nach Überprüfungen der Art „bin ich dem Sender zur Annahme von Befehlen verpflichtet?“ - die Ausführung der im Nachrichteninhalt spezifizierten Task veranlaßt.

5.4.1 Variablen

Sowohl die Aktivierung einer Motivation als auch das Auslösen eines sensing-alarms und die Steuerung normativer Ziele hängen von Zustandsvariablen ab. Diese Variablen erfüllen mehrere Zwecke: Zum einen modellieren sie die interessanten Aspekte der Umwelt eines Agenten, zum andern dienen sie als interne „Schalter“. Jede Zustandsvariable besitzt mindestens einen Namen und einen Wert und optional einen Typ und Spezifikationen von unterem und oberem Schwellenwert. Modelliert werden Variablen als Prolog-Terme mit 5 Argumenten folgendermaßen:

```
%% variable( Name, Type, Value, Min, Max ).
variable(termin, date, time(14:00), _,
         time(16:59) ).
variable(kosten, integer, 0, 0, 100 ).
variable(anzahlTasks, integer, 0, _, _ ).
variable(needed(info(_)),_,false,_,_).
```

Nicht berücksichtigte Argumente werden als freie Variablen (Unterstriche) notiert. Unter- oder überschreitet der Wert einer Variablen einen definierten Schwellenwert, wird eine Ausnahmebehandlung in Form eines sensing-alarms ausgelöst. Welche Maßnahmen aufgrund einer Bereichsüberschreitung auszuführen sind, wird durch Regeln beschrieben:

```
%% sensing_alarm( VarName, Bound ) :- consequence.
sensing_alarm( kosten, lower_bound ) :-
    activate( task, fehler('negative Kosten') ).
sensing_alarm( kosten, upper_bound ) :-
    activate( normative_goal, kosten_minimieren ).
```

Die erste Regel aktiviert eine Fehlerbehandlungs-Task im Falle der Unterschreitung der Null-Kosten-Grenze. Demgegenüber wird in der zweiten Regel beim Überschreiten der gesetzten oberen Kostengrenze ein normatives Ziel zur Kosteneinsparung aktiviert. Üblicherweise ändert sich der Wert einer Variablen durch Wahrnehmungen (siehe Abschnitt 5.2.2)[6]. Für Variablen vom Typ integer kann eine Wahrnehmung den Wert einer Variablen setzen, erhöhen oder vermindern; bei anderen, nicht-kardinalen Typen ist bloß das Setzen auf einen Wert vorgesehen.

5.4.2 Motivationen

Motivationen dienen dazu, die Zielsetzungen eines Agenten über eine längere Zeit zu beeinflussen. Im Gegensatz zu den anderen mentalen Begriffen werden sie durch allgemeine Bedingungen aktiviert und nicht durch den Zustand anderer Behaviours. Eine Motivation wird durch eine Menge von Prolog-Regeln beschrieben, wobei jede Regel eine andere Aktivierungsbedingung ausdrückt.

```
%% motivation( name ) :- activation_condition.
motivation( hungry_for_information( Info )) :-
        needed( Info ).
motivation( hungry_for_information( Info )) :-
        requested( Info ),
        number_of_requestors( Num ),
        Num > 1.
motivation( hungry_for_information( Info )) :-
        requested( Info ),
        estimated_costs( Info, C ),
        offer( _Requestor, X ),
        X >= C.

needed( Info ) :-
        variable( needed(info(Info)),_,true,_,_),
        not_known( Info ).
```

Solange mindestens eine Aktivierungsbedingung zutrifft, ist die Motivation gültig. Sie erlischt automatisch, nachdem das letzte verbliebene Stimulans nicht mehr wahr ist. Somit beschreiben die Aktivierungsbedingungen auch gleichzeitig die Situation, in der die Deaktivierung vorgenommen werden muß. Das Management der Motivationen findet im Intentionalitätsmodul automatisch bei Änderungen der Wissensbasis statt, so daß die Kontrolle nicht von Hand implementiert werden muß (siehe Abschnitt 6.4 auf Seite 162).

6. Prinzipiell ist es auch möglich, eine Variable durch einen tell-Sprechakt zu setzen. Dies liegt in der Tatsache begründet, daß Variablen in der Wissensbasis verwaltet und von anderem Wissen nicht unterschieden werden.

5.4.3 Ziele

Zustandsziele werden durch Motivationen oder aus einem Skript heraus aktiviert. Wie aus dem Codebeispiel hervorgeht, werden Ziele durch beliebige Terme dargestellt. Die Termargumente sind allgemeine Parameter, die bei der Aktivierungsbedingung genutzt werden können. Jede weitere Aktivierung verstärkt ein aktives Ziel[7].

```
%% goal activation
%% activate( Goal_name, Motivation_name ).
activate(infoSeek( Item ) :-
          hungry_for_information( Item )).
activate(infoSeek( Item ) :-
          should_know_information( Item )).
```

Zusätzlich zum Aktivierungszustand existiert für jedes Ziel ein Attribut. das anzeigt, ob das Ziel erfüllt ist oder nicht. Operationalisiert wird dieses Attribut durch eine Menge von Erfüllungsbedingungen. Ist eine der Bedingungen wahr, gilt das Ziel als erfüllt und wird deaktiviert.

```
%% goal_name :- fulfilling_condition.
infoSeek( Item ) :-
         variable( needed(info(Info)),_,false,_,_).
```

Erfährt ein Ziel eine Zustandsänderung von aktiv auf inaktiv, werden die mit dem Ziel verknüpften Aufgaben beendet. Triftige Gründe für den vorzeitigen Abbruch eines Dienstes können in der Stornierung des beauftragenden Agenten oder Nutzers oder der Übermittlung des zu berechnenden Ergebnisses durch einen anderen Agenten liegen. Jedes Ziel besitzt zudem eine statische Priorität. Wird eine Task aufgrund eines Ziels aktiviert, so erbt sie dessen Priorität. Dienste mit höherer Priorität werden gegenüber solchen mit einem niedrigeren Wert bevorzugt abgehandelt. Die genaue Bedeutung der Prioritäten wird bei der Beschreibung des Schedulers in Abschnitt 6.6 erklärt. Ziel-Teilziel-Beziehungen werden in REkoS nicht modelliert. Eine ähnliche Art der Modularisierung ist stattdessen auf Skriptebene gegeben (siehe Abschnitt 5.5.1).

5.4.4 Normative Ziele

Das Konzept normativer Ziele realisiert Beschränkungen der Handlungsweise des Agenten, indem es selektiv auf die Menge ausführbarer Skripte wirkt. Ein normatives Ziel fügt zu den in der Wissensbasis existierenden Regeln, welche die Zuordnung von Handlungsskripten zu Tasks beschreiben, weitere Kontrollregeln hinzu.

7. Zielaktivierungen werden über sogenannte Begründungslisten (siehe Abschnitt 6.4.1) realisiert.

Diese haben den Zweck, die Ausführung bestimmter Skripte entweder zu verhindern oder zu veranlassen. Die Skriptauswahl kann durch Domänen- oder Strukturwissen - z.B. ob ein Skript ein Interaktionsprotokoll nutzt oder nicht - vom Programmierer fest vorgegeben werden oder auch dynamisch, beispielsweise durch Verwendung statistischer Daten über durchschnittliche Laufzeit oder Kosten berechnet werden.

Normative Ziele werden genauso wie Motivationen durch formulierbare Bedingungen aktiviert.

```
%% normative_goal( name ) :- activation_condition.
normative_goal( minimize_costs ) :-
        variable( selectionPolicy,_,minCosts,_,_ ).
normative_goal( minimize_time ) :-
        variable( selectionPolicy,_,minTime,_,_ ).
```

Für jedes normative Ziel realisiert ein gleichnamiges zweistelliges Prädikat dessen Semantik. Operativ wirkt diese mentale Attitüde auf die Auswahl von in Frage kommenden Skripten bei der Neuaktivierung einer Task. Das Prädikat besitzt zwei Argumente, die jeweils Listen von Skriptnamen sind. Das erste Argument stellt die in Frage kommenden Skripte als Menge dar, die durch den Prozedurrumpf gefiltert und/oder umsortiert werden. Das zweite Argument hält das Ergebnis dieses Filterungsprozesses. Sind mehrere normative Ziele gleichzeitig aktiv, kommen die entsprechenden Sortier- bzw. Filterprozeduren nacheinander zum Einsatz, wobei sie jeweils das Ergebnis der vorherigen Prozedur als Input verwenden.

```
%% normative_goal_name( +InSkripts, ?OutSkripts ) :-
%%      normative_goal_semantic.
minimize_costs( In, Out ) :-
        sortSkripts( In, costs, <, Out ).
minimize_time( In, [Best] ) :-
        findFastest( In, Best ).
```

5.4.5 Tasks

Tasks stellen das Bindeglied zwischen der deklarativ beschriebenen Intentionalität und der prozeduralen Ausführungsebene der Skripte dar. Es existieren vier Möglichkeiten eine Task zu starten: Aus Sicht der Intentionalität werden Tasks durch Ziele aktiviert und, wie oben beschrieben, auch von diesen gesteuert. Reaktives Handeln geschieht durch Auslösen von sensing-alarms. Der Start eines Dienstes aus einem Skript heraus wurde aus Gründen der Modularität zugelassen und ist dem Bereich der Programmierung von Fähigkeiten zugeordnet (siehe Abschnitt 5.5.2 „Skriptsprache“ auf Seite 138). Die letzte Aktivierungsmöglichkeit durch Empfang einer command-Nachricht findet im Rahmen der Kommunikation mit anderen

Agenten statt (Abschnitt 5.3) und wird im oben vorgestellten Kommunikationsdefinitionswerkzeug beschrieben.

```
%% activate( +TaskName, +GoalName, ?TaskArguments )
activate( dbAccess, infoSeek( Data ),
        [(search_string, Data)]).
```

Das dritte Argument des activate-Fakts ist eine Liste von Argumentname-Wert-Tupeln. Mit diesen werden entsprechende Argumente der Task belegt. Fehlende Parameter werden bei der Aktivierung eines Dienstes durch Default-Werte ersetzt.

Vor einem Einordnen von Tasks in die Aktivierungshierarchie kommt die Definition. Dienste werden über Namen und Argumente identifiziert. Daraus folgt, daß zur Beschreibung einer Task deren Name und Parameter zu erfassen sind. Ein Parameter besteht aus einem Bezeichner, einem Typ und optionalem Defaultwert:

```
%% task( +Name, +ParameterTripleList ).
task( dbAccess, [(search_string, string),
                (num_results, int, 1)] ).

%% capability( +TaskName, +Parameter_Value_List ) :-
%%       restrictions for / compatibility between
%%       parameters
capability( dbAccess, P_V_List ):-
        completeParamList( P_V_List, ParamList ),
        getParam( costs, Cs ), integer( Cs ),
        getParam( num_results, Ns ), integer( Ns ),
        task_constraint( dbAccess, Cs, Ns ).
task_constraint( dbAccess, Costs, NumResults ) :-
        NumResults < 20,         % few results..
        P is NumResults * 10,    % ..linear growth
        Costs >= P.
task_constraint( dbAccess, Costs, NumResults ) :-
        NumResults >= 20,        % many results..
        Costs >= 200.            % ..constant price
```

Mit dem capability-Prädikat checkt ein Agent, ob er in der Lage ist, einen angeforderten Dienst zu den in der Argumentliste beschriebenen Bedingungen auszuführen. Das obige Beispiel drückt ein Constraint zwischen den Parametern Kosten und Ergebnismenge aus, nämlich daß im Bereich von weniger als 20 Antworten der Preis für den dbAccess-Dienst linear wächst und darüber hinaus pauschal zum Preis von 200 abgerechnet wird. Gemäß dieser Definition würde eine Dienstanfrage mit einem Budget von 100 bei 12 verlangten Antworten scheitern.

Anders als bei Zielen oder Motivationen kann eine Task unter Umständen mehrfach zur selben Zeit gestartet werden. Dies ist sinnvoll, wenn es sich bei einer Mehrfachaktivierung nicht um den identischen Dienst handelt. Abbuchungsdienste sind typische Vertreter dieser Art von Services; Mehrfachabbuchungen desselben Be-

trags vom selben Konto müssen entsprechend auch mehrfach durchgeführt werden. Anders verhält es sich mit Datenbankanfragen: Ist eine Anfrage gestartet und kommt eine gleiche (oder eventuell eine von der aktiven Anfrage subsumierte) Anfrage herein, kann diese am Ergebnis der ersten Anfrage partizipieren. Ein weiteres Fakt umschreibt diesen Sachverhalt.

5.4.6 Werkzeuge für die Modellierung des mentalen Modells

Die Beschreibung der Intentionalität eines Agenten wird durch eine Reihe von Werkzeugen unterstützt. Es existieren Editoren zur Deklaration und Verwaltung der mentalen Attribute, zur Programmierung der Semantik und zur Definition der Aktivierungsbeziehungen der einzelnen Entitäten (siehe Abb. 5-3).

Bei der Definition eines neuen Begriffs wird das vollständige Erfassen aller notwendigen Attibute durch das Eingabewerkzeug gesteuert. Dabei wird weitgehend von der Programmierebene abstrahiert; das Erfassen und Beschreiben von Konzepten steht im Vordergrund. Die Aktivierungshierarchien zwischen Motivationen, Zielen und Tasks werden graphisch spezifiziert. Für die notwendigen Programmierarbeiten stellen Texteditoren Code-Templates zur Verfügung, auf diese Weise wird fehlerfreies Programmieren erleichtert.

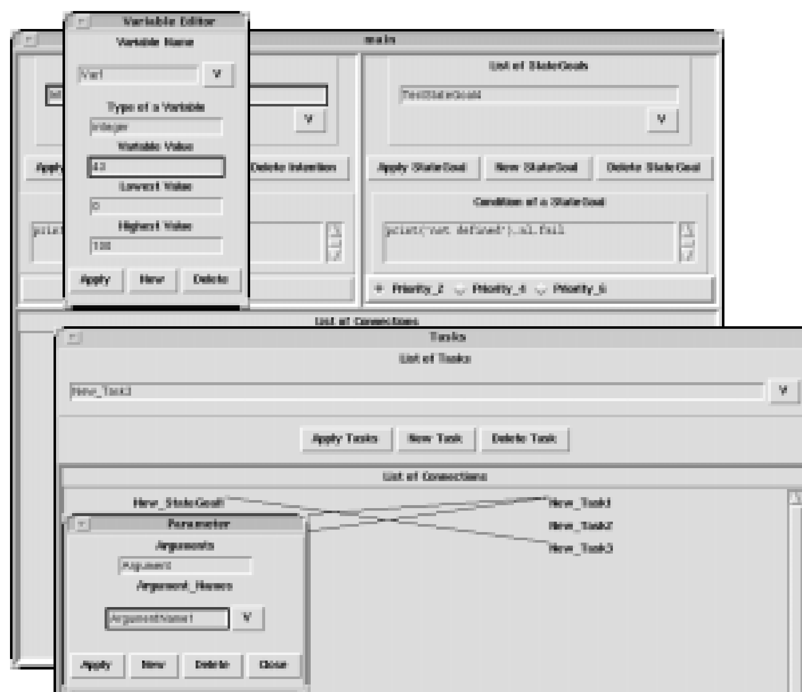

Abbildung 5-3: Werkzeuge zur Beschreibung der Intentionalität

5.5 Programmierung von Fähigkeiten

Die Implementierung der Fähigkeiten eines Agenten erfolgt auf zwei Ebenen. Auf relativ abstraktem Niveau werden Handlungsskripte spezifiziert. Dies geschieht mittels einer prozeduralen Skriptsprache und wird von Werkzeugen unterstützt. REkoS-Skripte besitzen eine modulare Struktur und ähneln entfernt den in dMARS verwendeten Handlungsplänen (siehe Abschnitt 4.2). Nach der Skriptbeschreibung erfolgt die eigentliche Programmierung in C++. Dabei hilft ein Programmgenerator bei der Transformation der Skripte in C++-Klassenbeschreibungen.

5.5.1 Das Skriptmodell

Die nachfolgend vorgestellte Programmiersprache orientiert sich an den wesentlichen Merkmalen anderer Skriptsprachen wie Tcl oder Perl: Sie ist sehr einfach gehalten, modular aufgebaut und wird interpretiert. Durch Beschränkung auf wenige programmiersprachliche Konzepte gestaltet sich der Umgang mit dieser Sprache relativ unkompliziert. Auf der anderen Seite sorgt die Modularität dafür, daß auch komplexere Sachverhalte modellierbar sind. Der Aspekt der Interpretation ermöglicht einem Agenten, Kontrolle über seine Aktionen auszuüben.

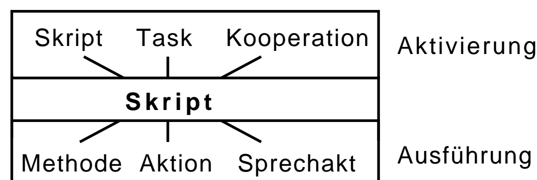

Abbildung 5-4: Skript-Beziehungen

Skripte bestehen aus Aktionen, Sprechakten und Methoden. Außerdem können sie auf andere Skripte, Aufgaben oder Kooperationsformen verweisen. Aktionen und Sprechakte werden genutzt, um außerhalb des Agenten liegende Ressourcen zu beeinflussen bzw. um mit anderen Agenten zu kommunizieren. Methoden hingegen sind lokale, nicht weiter verfeinerbare Fähigkeiten. Modularität ist durch die drei Konzepte Subskriptbildung, Task-Aktivierung und Anstoß einer Kooperation gegeben. Die Aktivierung eines Skripts aus einem Skript heraus ist äquivalent zum klassischen Prozedur- oder Unterprogrammaufruf. Task-Aktivierung stellt einen mächtigeren Mechanismus bereit: Mit der Aktivierung einer Task wird die prozedurale Skriptebene verlassen und die Erbringung eines (deklarativ beschriebenen) Dienstes veranlaßt. Die Entscheidung, wie die Task erbracht wird, wird dem Agenten überlassen. Er kann ein geeignetes Skript auswählen (entsprechend der in Abbil-

dung 5-2 (Seite 130) ersichtlichen 1:n-Beziehung zwischen Tasks und Skripten) oder die Aufgabe an einen anderen Agenten delegieren. Durch Starten einer Kooperation schließlich werden andere Agenten in die Problemlösung mit einbezogen. So können Teile der Problemlösung mit dem Ziel eines qualitativ besseren oder schnelleren Ergebnisses ausgelagert werden.

5.5.2 Skriptsprache

Skripte haben die Struktur von Funktionen. Sie besitzen einen Namen, Argumente und einen Rumpf, der aus einer Menge von Anweisungen besteht. Jedes Skript kann außerdem ein Ergebnis zurückliefern.

```
IF VerdachtUndSpezialistenBekannt( Verdacht,
                                   Spezialisten<= )
THEN
    DO
     Naechster( Spezialisten, Spezialist<= )
     BeauftrageSpezialisten(  Spezialist, Verdacht,
                              Diagnose<= )
    UNTIL DiagnoseErmittelt( Diagnose, Spezialist )
FI
RETURN Diagnose
```

Die einzig verfügbaren Kontrollkonstrukte zur Verknüpfung von Anweisungen sind die Sequenz, Verzweigung (IF-THEN-ELSE-FI) und Schleife (DO-UNTIL). Wie aus obigem Codebeispiel hervorgeht, werden Executables innerhalb eines Skripts wie Funktions- oder Prozeduraufrufe notiert. Aufrufargumente sind, wenn sie nicht explizit anders getypt sind, Strings. Ein „<=“-Zeichen hinter einem Argument identifiziert dieses als Rückgabeparameter, m.a.W., das Argument wird durch Abarbeitung der Anweisung mit einem Wert belegt. Die Skriptsprache kennt kein Konzept für lokale Variablen, so daß alle Arten von Zwischenergebnissen mit Aufrufargumenten modelliert werden müssen.

5.5.3 Werkzeuge für die Spezifikation von Fähigkeiten

Mit einer Reihe von Eingabewerkzeugen wird die Programmierung der spezifischen Fähigkeiten eines Agenten auf Skriptbeschreibungsebene unterstützt. Neben der eigentlichen Skriptprogrammierung findet hier auch die Zuordnung von Skripten zu Tasks statt. Abschließend erzeugt ein Programmgenerator C++-Klassenhüllen, die in einem letzten Schritt auszuprogrammieren sind. Dies ist klassische Programmiertätigkeit und liegt außerhalb der REkoS-Sicht auf die Agentenprogrammierung.

Abbildung 5-5 zeigt im Vordergrund das Formular für die Spezifikation von Skripten. Im oberern Bereich werden Name, Resultatstyp und Parameter beschrieben; für Dokumentationszwecke ist zudem ein Kommentarfeld vorgesehen. Das untere Eingabefeld dient der textuellen Eingabe des Skritprumpfes.

Ähnliche Masken exsitieren für die übrigen Executables. Allen gemeinsam sind ein eindeutiger Name, eine Argumentliste und ein Resultatstyp. Wenn nicht anders angegeben, werden Argumente und Ergebnisse als Strings behandelt. Für Tasks findet zusätzlich die Zuordnung der die Task realisierenden Skripte statt. Im Falle mehrerer zu Auswahl stehender Skripte können Auswahlregeln in Abhängigkeit der aktuellen Task-Parameter angegeben werden. Bei Sprechakten werden vorgegebene Argumente (Empfänger, Inhalt, Variable als Speicher für erwartete Antworten, etc.) belegt.

Abbildung 5-5: Bildschirmformulargestützte Spezifikation von Fähigkeiten

Bei der Beschreibung eines Skripts kann eine Inkompatibilitätsliste, bestehend aus anderen Skripten, angegeben werden. Die Liste beschreibt, welche anderen Skripte nicht gleichzeitig aktiv sein dürfen. Mit diesem einfachen Mittel kann der Skriptprogrammierer Hintergrundwissen über mögliche Ressourcenkonflikte, -engpässe oder andere mögliche Konfliktsituationen direkt in die Skriptausführung umsetzen[8]. Das Fenster im Hintergrund des Screenshots ist das Hauptfenster der Anwendung. Es gibt einen Überblick über alle bislang definierten Executables. Von hier aus können Tasks, Skripte u.s.w. hinzugefügt, bearbeitet und entfernt werden.

5.6 Spezifikation von Interaktionsprotokollen

REkoS bietet ein sehr mächtiges Interaktionsmodell, das die Leistungsfähigkeit der meisten anderen Agentenprogrammiersprachen übersteigt. Der wesentliche Unterschied zu anderen Ansätzen liegt in der Gestaltungsmöglichkeit von rollenbasierten Protokollen, an denen beliebig viele Agenten partizipieren können. Herkömmliche Interaktionsprotokollsprachen für Agenten berücksichtigen nur zwei Teilnehmer bzw. Rollen[9].

5.6.1 Protokollklassen

Die Diskussion globaler Koordinierungstechniken in Abschnitt 2.3.6 hat gezeigt, daß Interaktionsprotokolle einen allgemeinen Rahmen für die Lösung unterschiedlicher Koordinationsprobleme darstellen. Entsprechend werden in REkoS definierte Interaktionsprotokolle sogenannten *Protokollklassen* zugeordnet. Beispielsweise ist das Contract-Net ein Vertreter der Klasse „Aufgabenverteilung“, während SANP in die Rubrik „Konfliktbehandlung“ gehört.

Hinter dem Konzept der Protokollklassen stecken mehrere Ideen. Wenn eine Klasse ein Problemfeld oder Ziel abdeckt, kann ein passendes Protokoll automatisch beim Auftreten einer Problemsituation aktiviert werden. Es braucht dazu nur eine Zuordnung von Problemen auf Protokollklassen definiert sein. Sind mehrere Vertreter in einer Klasse enthalten, besteht eine Auswahlmöglichkeit, in die Kriterien wie voraussichtlicher Aufwand oder Lösungsqualität einfließen können.

8. Ein allgemeinerer Mechanismus zum Erkennen und Auflösen interner Konflikte ist nicht vorgesehen. Dazu fehlt es an einem schlüssigen Modellierungskonzept für Ressourcen oder Koordinationsbeziehungen wie beispielsweise in TÆMS (siehe Abschnitt 4.4 „TÆMS / GPGP“ auf Seite 102).

9. Beispielsweise COOL (siehe Abschnitt 4.3), GCCP oder IMAGINE (Abschnitt 4.5).

5.6.2 Rollen

Jedes Protokoll besteht aus einer Menge von Rollen, die jeweils durch einen Protokollgraphen beschrieben werden. Eine ausgezeichnete Rolle spielt die des Kooperationsmanagers. Sie wird automatisch von dem Agenten ausgefüllt, der die Kooperation startet. Mit der Managerrolle sind weitreichende Kontrollkompetenzen verbunden, so ist nur er in der Lage, ein Protokoll vorzeitig zu beenden. Jeder Agent in einer Kooperation füllt immer genau eine Rolle aus, die sein Verhalten bestimmt. Die Zuorndung von Rolle zu Agent ist eine 1:n-Beziehung. Auf diese Weise kann auf einfache Art z.B. das Verhalten eines Teams beschrieben werden.

Agenten müssen nicht über die gesamte Zeit einer Kooperation dieselbe Rolle ausfüllen, sondern können aus der Kooperation aussteigen oder die Rolle wechseln. Kommentarloses Beenden einer Kooperation ist einerseits der Autonomie geschuldet, kann auf der anderen Seite aber auch unbeabsichtigt durch Termination des Agenten oder Ausfall von Kommunikationsinfrastruktur geschehen. Ein Beispiel für einen Rollenwechsel bietet die unten beschriebene Realisierung eines Contract-Net-Protokolls. Dort schlüpft der Agent, der den Zuschlag bekommt, von der Rolle des Anbieters zu der des Contractors.

Die Kommunikation innerhalb eines Protokolls findet zwischen Rollen statt. Alle Sprechakte, die an eine Rolle adressiert sind, werden automatisch an sämtliche Agenten, die aktuell „in der Rolle“ sind, weitervermittelt.

5.6.3 Sprache zur Beschreibung von Interaktionsprotokollen

Die zwei wesentlichen Elemente von Interaktionsprotokollen, Kommunikation und lokales Handeln, werden bereits von den in Abschnitt 5.5 behandelten Skripten abgedeckt. Das dort beschriebene Skriptmodell wurde daher für die besonderen Anforderungen der Interaktionsprotokolle erweitert.

Kooperationsskripte

Jede Rolle besteht aus einer Menge von Kooperationsskripten, die eine Spezialform der Klasse der oben beschriebenen Handlungsskripte darstellen. Das eigentliche Skript, das den Handlungsablauf in einem Kooperationszustand durch eine Sequenz von Steps beschreibt, ist in eine Vor- und eine Nachbedingung eingekleidet.

Die Vorbedingung bestimmt, wann das Skript aktiviert wird. Anders als bei Handlungsskripten hängt die Aktivierung nicht von einer Task oder einem Ziel ab, sondern vom Verlauf der Kooperation. Prinzipiell kann an dieser Stelle ein beliebiger boolescher Ausdruck stehen. Die Aufgabe der Vorbedingung liegt aber in der Sicherstellung der Ablaufreihenfolge innerhalb der Rolle und in der Synchronisation

mit den anderern Kooperationsrollen. Daher werden hier typischerweise Zwischenergebnisse von vorherigen Skripten oder das Eintreffen von Sprechakten anderer an der Kooperation beteiligter Agenten abgefragt.

Die Nachbedingung erfüllt zwei Funktionen. Zum einen findet aktive Synchronisation durch Versenden von Sprechakten an andere Rollen statt. Zum andern legt sie fest, welches Skript der Rolle als nächstes aktiviert wird. Verlassen einen Knoten mehrere Kanten, so wird dies durch bedingte Verzweigungen realisiert. Entspricht das Kooperationsskript einem Endknoten, existiert kein Nachfolgeskript, sondern die Mitwirkung des Agenten an der Kooperation ist beendet. Einen Sonderfall stellt der mögliche Rollenwechsel dar, auch dieser wird in der Nachbedingung beschrieben.

Kommunikation

Kommunikation findet über den gewohnten Mechanismus der Sprechakte statt. Weil die Verarbeitung von Nachrichten innerhalb von Interaktionsprotokollen frei definierbar ist, hat die illokutionäre Ebene der Sprechakte hier keine Bedeutung, d.h. die send_function bzw. receive_function wird nicht ausgeführt. Im Prinzip handelt es sich demnach um reine Daten- oder Synchronisationskommunikation[10].

Beim Start eines Interaktionsprotokolls wird neben einer Protokollinstanz eine eindeutige Kooperations-ID generiert. Auf der Suche nach Partnern (bei der Initialisierung, s.o.) schickt der Kooperationsmanager an die potentiellen Kandidaten einen Sprechakt, der den Namen des Protokolls, die ID und eventuell die einzunehmende Rolle enthält. Nimmt ein angesprochener Agent teil, so generiert er sich eine Protokollinstanz der ihm zugewiesenen Rolle und ordnet ihr die zugehörige ID zu. Jeder Sprechakt, der aus einem Kooperationsskript abgeschickt wird, wird automatisch mit der Kooperations-ID und dem Rollennamen versehen. Auf diese Weise können eingehende „normale“ von zu Protokollen gehörenden Sprechakten unterschieden werden.

Neben der gerichteten Kommunikation von Agent zu Agent werden auch Broadcast- und Multicast-Nachrichten unterstützt. Broadcastnachrichten erreichen alle Agenten, die am Interaktionsprotokoll teilnehmen. Als Adressat kann aber auch ein Rollenname angegeben werden, entsprechend wird der Sprechakt allen Agenten der Rolle zugestellt.

10. Laut [Levinson 1981] (siehe Zitat in Abschnitt 2.2.2) sind Sprechakte und Interaktionsprotokolle (Dialoge) prinzipiell nicht miteinander vereinbar.

5.6.4 Spezifikationswerkzeug für Interaktionsprotokolle

Zur Spezifikation von Interaktionsprotokollen wurde ein Grapheneditor entwickelt. Er bietet Funktionalität zur Eingabe und Verwaltung von Protokollklassen, Protokollen und Rollen. Das wesentliche Element bei der Spezifikation ist die Beschreibung des synchronisierten Ablaufs zwischen den einzelnen Rollen.

Jede Rolle eines Interaktionsprotokolls wird durch einen gerichteten Graphen mit einem ausgezeichneten Anfangs- und - eventuell mehreren - Endknoten modelliert (siehe Abbildung 5-6 links). Dabei definieren Knoten Zustände, in denen ein Agent Berechnungen durchführt, während die Kanten Zustandsübergänge markieren. Bei der Abarbeitung eines Protokolls „wandert“ ein Agent durch den Graphen, bis er einen Endknoten erreicht.

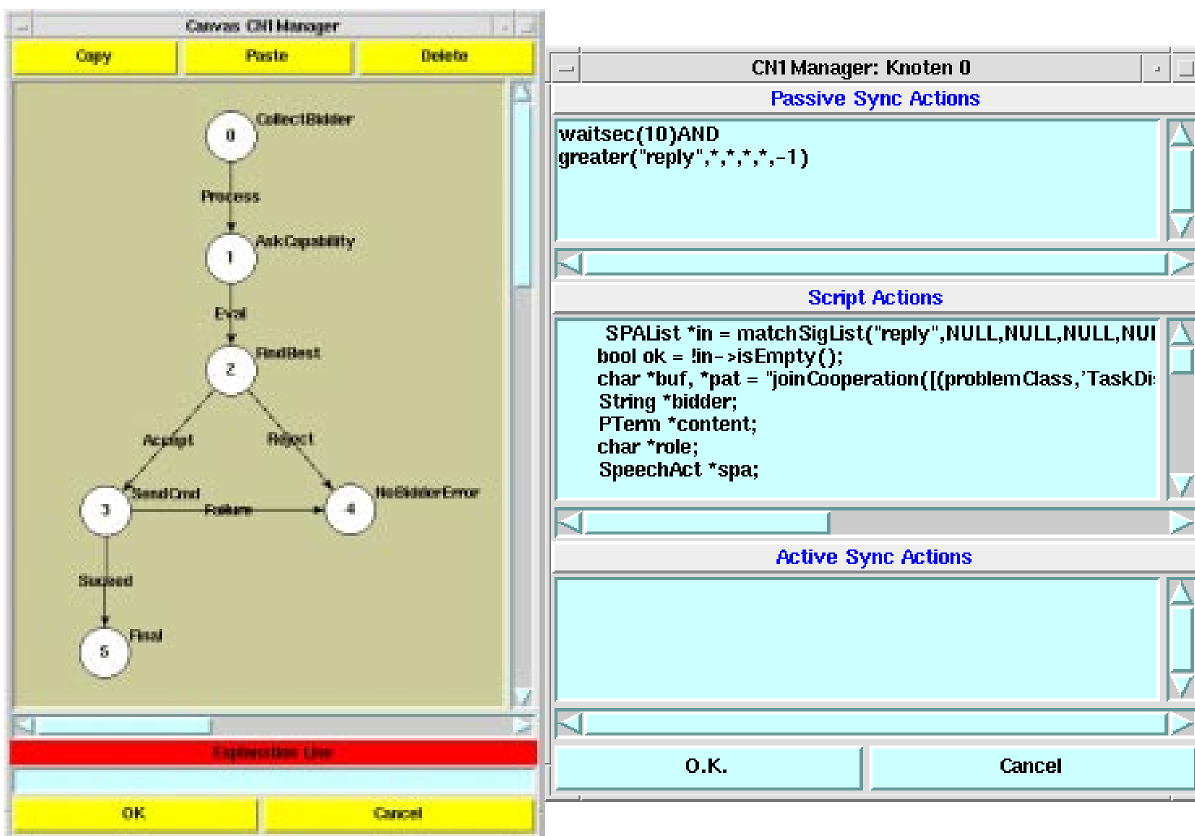

Abbildung 5-6: Graphische Repräsentation von Kooperationsrollen

Mit jedem Knoten ist ein Kooperationsskript assoziiert, das über einen Knoteneditor direkt aus dem Graphen heraus spezifiziert werden kann (siehe Abbildung 5-6

rechts). Der Aufbau eines derartigen Skripts ähnelt den Konversationsregeln in COOL (siehe Abschnitt 4.3). Es besteht aus einer Aktivierungsbedingung (passive sync actions), dem Handlungsrumpf (script actions) und einem Verzweigungsteil (active sync actions). Dabei wird die Handlungsausführung eines Knotens solange verzögert, bis die Aktivierungsbedingung wahr wird. Zum Abschluß der Skriptabarbeitung wird im Verzweigungsteil der Nachfolgeknoten bestimmt.

Die Aktivierungsbedingung eines Kooperationsskripts dient zur Synchronisation mit den Rollen anderer an der Kooperation beteiligter Agenten. Zur Unterstützung des Programmierers wurden einige oft benötigte Synchronisationsprimitive vordefiniert. Diese können, zusammen mit weiteren von Hand zu programmierenden Tests, mit Hilfe boolescher Operatoren zu logischen Ausdrücken zusammengefaßt werden.

TABELLE 4. Synchronisationsprimitive in Kooperationsskripten

Synchronisations-primitive	Bedeutung
sleep(X)	X Sekunden warten
know(Term)	warten bis der Prolog-Ausdruck Term in der Wissensbasis erfüllbar ist
wait_until(X)	bis zum Zeitpunkt X warten
s_a(X)	auf einen Sprechakt X warten
s_a(From, X)	auf Sprechakt X eines Kooperationspartners der Rolle From warten
s_a_set(Num, X)	auf mindestens Num Sprechakte der Art X warten
s_a_set(Num, From, X)	auf mindestens Num Sprechakte der Art X von Rolle From warten

Der Skriptrumpf wird mittels der Skriptsprache zur Programmierung der prozeduralen Fähigkeiten (siehe Abschnitt 5.5.2) beschrieben. Zusätzlich muß noch der Name des Nachfolgeknotens, sofern es sich nicht um einen Endknoten handelt, bestimmt werden. In der graphischen Spezifikation sind die möglichen Nachfolgeknoten bereits definiert; sie finden sich als „Sprunglabels“ im Verzweigungsteil des Kooperationsskripts wieder. Die Verzweigung zum Nachfolger geschieht dann aus dem Skript heraus mittels eines goto-Konstrukts, in das ein gültiges Label eingetragen wird.

5.7 Integration

Mit den vier vorgestellten Sprachen lassen sich Agenten in Bezug auf ihre interaktiven, prozeduralen und intentionalen Fähigkeiten vollständig beschreiben. Weil die Spezifikationsebene für Zusammenhänge und Abläufe auf hohem Abstraktionsgrad stattfindet, bedarf es weiterer Schritte um zu maschinenverarbeitbaren Code zu gelangen. Shoham nennt diesen Prozeß der automatischen Überführung einer Agentenbeschreibungssprache in lauffähige Agenten „Agentifizierung“[11].

Für jede Sprache wurde ein Programmgenerator entwickelt und in das jeweilige Spezifikationswerkzeug integriert. Der Agentifizierungsprozeß funktioniert für die Kommunikations- und die Intentionalitätsbeschreibungssprache vollständig, so daß außer der Werkzeugbedienung keine weiteren Programmier- oder sonstige Anpassungsarbeiten notwendig sind. Daß die Integration hierbei zu 100 Prozent gelang liegt in der Entscheidung, Prolog zu verwenden. Einerseits eignet sich diese deklarative, typfreie Sprache gut für Programmgenerierung, andererseits findet in den Spezifikationseditoren bereits Prolog-Programmierung statt. Erheblich schwieriger gestaltet sich die Codegenerierung für die skriptsprachenbasierten Spezifikationen der prozeduralen Fähigkeiten und Interaktionsprotokolle. Hier ist eine direkte Überführung in ausführbaren (C++-)Code nicht möglich; der konzeptuelle Abstand zwischen der typlosen, rein prozeduralen Skriptsprache und dem typisierten, objektorientierten Implementierungssprache ist einfach zu groß. Daher werden Klassenbeschreibungen generiert, die weiter von Hand auszuprogrammieren sind.

5.7.1 Codegenerierung für Sprechaktdefinitionen

Der Sprechakteditor generiert mehrere Dateien, die Ausschnitte eines Sprechaktmodells den Werkzeugen zur Erfassung der prozeduralen und kooperativen Fähigkeiten sowie dem später beschriebenen Debugger zur Verfügung stellen. So wird dem Skript-Editor eine Liste aller definierten Sprechakte in einer Datei zur Verfügung gestellt. Außerdem wird eine Prolog-Wissensbasis generiert, die jeder das Kommunikationsmodell nutzende Agent automatisch beim Programmstart einliest. Diese Datei enthält die Regeln für die Interpretation empfangener und Nachbearbeitung verschickter Sprechakte und weiteres Meta-Wissen über die bekannten Performatives und verstandenen Sprechakte.

Zwar werden Sprechakte durch den mit dem Editor definierten Prolog-Code interpretiert, ihre Nutzung hat jedoch aufgrund der gewählten Architekturimplementierungssprache in C++ zu erfolgen. Deshalb wird für jeden Sprechakt eine korre-

11. [Shoham 1993].

spondierende Klassenbeschreibung generiert, die eine Reihe von Konstruktoren zur Kreierung von zu sendenden Nachrichten bereitstellt.

```
1 class command :public SpeechAct {
2    ...
3 public:
4    void command(String sender;
                  String[] receiver;
                  unsigned speechActID;
                  MsgContent content;
                  unsigned coopProtocolID = 0);
5    void command(String sender;
                  String[] receiver;
                  unsigned speechActID;
                  MsgContent content;
                  unsigned timeout = 1;  // seconds
                  unsigned coopProtocolID = 0);
6    Agenda* send(boolean blocked);     // w. result
7    void send();                       // w/o result
8 }
```

In Abhängigkeit des spezifizierten Antwortverhaltens (on_demand) ergeben sich mehrere Nutzungsmöglichkeiten für ein command: Ohne eine erwartete Rückantwort wird der Sprechakt mit dem ersten Konstruktor aufgebaut[12] und mittels der zweiten send()-Methode in Zeile Zeile 7 abgeschickt. Für den Fall, daß vom empfangenden Agenten eine Reaktion erwartet wird, wird der zweite Konstruktor (Zeile 5) benutzt. Hierbei ist zusätzlich ein timeout zu spezifizieren, der angibt, wie lange der Sender maximal auf eine Antwort wartet. Zum Versenden kommt die erste send-Methode in Zeile 6 zum Einsatz, sie liefert als Ergebnis einen Verweis auf eine Agenda, aus der eingetroffene Antworten ausgelesen werden können. Die Kommunikation kann für den sendenden Agenten entweder blockierend oder nicht blokkierend erfolgen. Bei blockierendem Senden wartet der Agent die im timeout angegebene Zeitspanne, bevor er mit der Auswertung der Antwort(en) weiter fortfahren kann. Ist hingegen das blocked-Flag der Sendemethode auf false gesetzt, arbeitet der Agent unmittelbar weiter. In diesem Fall liegt es in der Verantwortung des Agentenprogrammierers, für die Verwaltung der Agenda und der in ihr enthaltenen Antworten Sorge zu tragen.

12. MsgContent beschreibt eine generische Klasse, die Operator, Content-Typ und Nachrichteninhalt enthält.

5.7.2 Codegenerierung für die Intentionalitätsbeschreibungssprache

Die Werkzeuge zur Beschreibung der Intentionalität generieren eine Prolog-Datei, welche zur Startzeit vom Agenten eingelesen wird und damit dessen mentalen Zustand initialisiert. Aus den Erfassungslisten, Texteditoren und graphischen Repräsentationen wird Code erzeugt, der den illustrierenden Beispielen in Abschnitt 5.4 entspricht. Darüberhinaus werden weitere Hilfsprädikate generiert, die den Zugriff auf die definierten intentionalen Begriffe ermöglichen. Außerdem wird der initiale mentale Zustand in die Datenbasis geschrieben. Er besteht für jedes Behaviour aus einer Aufzählung sämtlicher modellierter mentaler Begriffe, wobei sämtliche Behaviours mit dem Zustand „inaktiv“ attributiert sind. Variablen werden gemäß ihrer Definition mit ihren Anfangswerten initialisiert.

5.7.3 Codegenerierung für die Skriptsprache

Für jedes mit dem Werkzeug definierte Executable werden eine C++-Header- und Implementierungsdatei[13] generiert. Vor der Codegenerierung finden Syntaxchecks in bezug auf Skriptsprache und Typüberprüfungen für Skriptparameter, Variablen und Rückgabewerte statt. Der Programmgenerator gibt außerdem Fehlermeldungen für deklarierte, aber nicht ausprogrammierte Executables aus. Eine automatische Umsetzung der Skriptbeschreibungssprache in ausführbaren C++-Code stellt aufgrund der Deckungsgleichheit der verwendeten Konzepte (Sequenzen, Schleifen und Verzweigungen) kein Problem dar. Jedoch erfolgt die Umsetzung eines Handlungsskripts nicht direkt in eine Klassenmethode, sondern es findet eine weitere Aufteilung in sogenannte Steps statt.

Steps spannen den Syntaxbaum des zugehörigen Skripts auf: Jeder Step enthält eine elementare Anweisung und einen Verweis auf den nächsten auszuführenden Step. Die Ausführung eines Skripts findet dann durch Aktivierung der Steps in der in ihnen verdrahteten Reihenfolge statt. Obwohl es sich bei Steps um konkrete C++-Klassen - und damit um direkt ausführbaren Code - handelt, kann bei der Abarbeitung des Syntaxbaums von einer interpretierten Arbeitsweise gesprochen werden.

Zwar bringt jede zusätzliche Interpretationsschicht Performance-Einbußen mit sich, bietet auf der anderen Seite aber erhebliche Vorteile in bezug auf kontrollierte Ausführung. Aus eben diesem Grund wurde die Aufsplittung eines Skripts in Steps vorgenommen. So kann die Ausführung eines Skripts nach jedem Step unterbrochen werden. Ein späteres Wiederaufsetzen ist quasi ohne zusätzlichen Verwal-

13. Für Nichtkenner von C++: Eine Headerdatei enthält *Deklarationen* (Signaturen) von Klassen und Methoden. Deren *Definitionen* (Implementierungen) werden üblicherweise in separaten Implementierungsdateien verwaltet.

tungsaufwand möglich, es muß dazu nur der nächste Step aktiviert werden. Die Übersetzung des oben beschriebenen Skriptbeispiels (siehe page 137) in Steps sieht vereinfacht[14] wie in Abbildung 5-7 aus:

```
1 step_1::eval() {
2   if VerdachtUndSpezialistenBekannt( Verdacht,
3                                     Spezialisten ) {
4        nextStep = step_2;
5   else
6        nextStep = step_6;
7   }
8 step_2::eval() {
9   Naechster( Spezialisten, Spezialist );
10  nextStep = step_3;
11  }
12       // ...
13 step_5::eval() {
14  nextStep = NULL;
15  result = Diagnose;
16  }
```

Abbildung 5-7: Struktur und Verkettung von Steps (vereinfachte Darstellung)

Der Step step_1 zeigt die Umsetzung einer IF-THEN-ELSE-Verzweigung an, wobei in Abhängigkeit des Tests auf den nächsten Step verzweigt wird (Zeilen 1 - 7)[15]. Einen Aufruf der Methode Naechster realisiert step_2 (Zeilen 8 - 11), während step_5 den Abschluß des Skripts mit Resultatsübergabe implementiert (13 - 16).

Abbildung 5-8 zeigt die generierte Klassenbeschreibung für ein Skript VerifiziereVerdacht, das ein String-Argument V besitzt. Skriptklassen stellen Ausführungsumgebungen für die sie implementierenden Steps dar. Sowohl die Parameter des SKripts wie auch die Aufrufargumente der Steps werden als Membervariablen im Skript gehalten (Zeilen 3 und 4). Für den Zugriff auf diese Daten werden Setter- und Getter-Funktionen generiert (Zeilen 11 - 14). Die eval-Methode (Zeilen 15 - 18) sorgt dafür, daß der erste Step initialisiert wird und leitet die weitere Verarbeitung an eine gleichnamige Methode der Oberklasse weiter[16].

```
1 class VerifiziereVerdacht : public Script {
```

14. Der vollständige Code einer generierten Step-Klasse findet sich in Anhang A.2.

15. nextStep ist als Member-Variable in einer Oberklasse deklariert.

16. Detailliertere Informationen zum Ablauf finden sich in Abschnitt 6.7.

```
2 private:
3   String* V; // ...
4   String* result;
5 public:
6   VerifiziereVerdacht(Agent* agent, String* x1)
7           : Script(myAgent, new String(
8           "VerifiziereVerdacht")) {
9       V = x1;
10  }
11  String* getV() {return (V)};
12  void setV(String* x1) { // w/ memory management
13      if(V) V->detach(); V=x1; if(V) V->attach()
14  };
15  virtual void eval() {;
16      setNextStep(new step1());
17      Script::eval();
18  }
```

Abbildung 5-8: Generierter Code eines Skripts (Ausschnitt)

Wissen über die mit dem Werkzeug definierten Fähigkeiten und deren Zuordnung zu Klassennamen wird als Fakten für die Prolog-Wissensbasis generiert. Mit ihnen kann ein Agent Auskunft über seine prozeduralen Fähigkeiten geben. Auch die Zuordnung von zu aktivierenden Executables zu den generierten C++-Klassen ist mittels Fakten in der Wissensbasis geregelt.

TABELLE 5. Übersicht über generierten Code für Handlungsskripte

Tool-Spezifikation	generierter C++-Code	generierter Prolog-Code	Implementierungsarbeiten
Task T	T.h, T.cc	can(task, T). selectScript(T, ParamList, S).	keine
Skript S	S.h, S.cc, S_step1.h, ... S_step1.cc, ...	can(script, S). class(S, S).	keine
Methode M	M.h, M.cc	can(method, M).	eval-Methode

TABELLE 5. Übersicht über generierten Code für Handlungsskripte

Tool-Spezifikation	generierter C++-Code	generierter Prolog-Code	Implementierungsarbeiten
Sprechakt C	C.h, C.cc	siehe Sprechakt-definitionstool	keine
Aktion A	A.h, A.cc	can(action, A).	eval-Methode

Mit wenigen Einschränkungen ähnelt die Programmgenerierung für Interaktionsprotokolle dem oben beschriebenen Verfahren. Kooperationsskripte werden auf dieselbe Weise in Klassenbeschreibungen transformiert, nur daß es sich um von cooperationScript abgeleitete Klassen handelt. Die Graphenstruktur entsteht automatisch, indem die spezifizierten Verzweigungen zu Nachfolgeknoten in die eval-Methoden integriert werden. Weitere Klassen cooperationClass, cooperationProtocol und cooperationRole, die zueinander in Vererbungsrelation stehen, bilden den modularen Aufbau des Kooperationsmodells nach.

5.7.4 Kreierung von Agenten

Zum Start eines Agenten müssen zuvor die generierten und selbst programmierten C++-Quelldateien compiliert und mit den Architekturbibliotheken und generierten Prologprogrammen zusammengebunden werden. Für die Verwaltung der mit den einzelnen Editoren erstellten Dateien und Directory-Bäume stellt die Toolbox ein integrierendes Werkzeug bereit. Die Toolbox ist eine grafische Oberfläche, aus der heraus die einzelnen Definitionswerkzeuge aktiviert werden können. Eigentliche Aufgabe dieser Anwendung ist die Generierung von Makefiles[17], welche die in den einzelnen Editoren verwendeten Dateien und Suchpfade beinhalten. Durch Ausführung des Makefiles werden alle notwendigen Dateien übersetzt und zu einem ausführbaren Agenten zusammengebunden.

Zu jedem Agenten gehört eine einfache grafische Oberfläche mit Grundfunktionalitäten wie Speicherung persistenter Daten, Unterbrechen und Fortsetzen der Aktivitäten, Beenden oder Zurücksetzen des Agenten[18]. Außerdem kann ein Debugmodus ein- und ausgeschaltet werden.

17. Makefiles dienen zur Applikationserstellung unter UNIX.

18. Die graphische Bedienoberfläche ist in Form eines Tcl/Tk-Programms, das mit dem Agenten über Callback-Funktionen kommuniziert, realisiert. Eine Abbildung der Oberflächenelemente findet sich auf Seite 188 in Abbildung 7-2.

5.8 Zusammenfassung und Bewertung

Anders als herkömmliche Agentenprogrammiersprachen bietet REkoS nicht eine sondern vier Beschreibungssprachen für die Bereiche Kommunikation, Interaktionsprotokolle, Fähigkeiten und Intentionalität. Graphische Eingabewerkzeuge und die REkoS-Toolbox unterstützen die Programmierarbeit und bieten ein loses Integrationskonzept.

Zur Beschreibung und Implementierung von (lokalen) Diensten wurde eine *Skriptsprache* entwickelt, deren Mächtigkeit an die Ausdrucksstärke der Planbeschreibungssprache von dMARS (siehe Abschnitt 4.2) heranreicht. Ihr modularer Aufbau ermöglicht die Darstellung von Dienst-Subdienst-Beziehungen und unterstützt durch das Starten von Tasks, Initiieren von Interaktionsprotokollen und Versenden von Nachrichten einen flexiblen und dynamischen Ablauf.

Informationsaustausch findet über *Sprechakte* oder *Interaktionsprotokolle* statt. Das Sprechaktmodell legt, anders als KQML oder FIPA (siehe Abschnitt 2.2.1), auch das Verhalten des Empfängers fest. Die Nachrichteninhalte beziehen sich auf die REkoS-Begriffswelt der verarbeitbaren und mentalen Attitüden, sind aber, um Konformität mit den oben erwähnten Quasi-Standards zu erreichen, erweiterbar. Content-Sprache ist Standard-Prolog. Mit der Sprache zur Definition von Interaktionsprotokollen werden Verhandlungen oder Kooperationen rollenbasiert modelliert. Die Verknüpfung von Handeln und Kommunizieren drückt sich in den Rollengraphen aus, zur Synchronisation mit anderen Rollen werden Synchronisationsprimitive angeboten. Daneben existiert ein auf Reaktivität ausgerichteter, ereignisbasierter Wahrnehmungsmechanismus. Mit *Sensings* - und ihrem Gegenpart, den *Actions* - ist ein Basismechanismus für Interaktionen mit der nicht-agentischen Umbebung gegeben.

Zur Steuerung und Kontrolle der Aktivitäten eines Agenten wird ein hierarchisches mentales Modell aufgebaut. Wiederkehrende oder dauerhafte Aktivitäten werden über *Motivationen* modelliert, die Steuerung der Aufgaben erfolgt über *Ziele*, reaktive Verhaltensformen durch *sensing-alarms* unter Verwendung von *Variablen* programmiert. Situationsbedingte Auswahl geeigneter Skripte kann durch *normative Ziele* spezifiziert werden.

In der Praxis[19] zeigte sich, daß diensterbringende Agenten zumeist ohne komplexe mentale Zustandsbeschreibungen auskommen. Für einfach gestrickte Agenten, die nur einen Dienst anbieten, findet die Aktivierung besser über direktes An-

19. Zwei größere Anwendungsprototypen wurden mit REkoS realisiert: ein wissensbasiertes, kooperatives Diagnosesystem, bestehend aus 5 Agenten mit unterschiedlichem Expertenwissen und eine Anwendung aus dem Gebiet der intelligenten Netze für dynamisches Dienstkreierung und -erbringung [Fricke, et al. 1998], [Albayrak, et al. 1996].

sprechen statt oder erfolgt automatisch aus Skripten heraus. Der Weg über das Setzen einer Variablen mit nachfolgendem Erwachen einer Motivation hat sich in den meisten Fällen als zu umständlich erwiesen. Damit soll jedoch das generelle Konzept der von Stimuli kontrollierten Motivationen nicht in Abrede gestellt werden; die Anwendungen bzw. die programmierten Agenten waren in den realisierten Szenarien einfach nicht komplex genug. Generell ist das Variablenkonzept etwas ausdrucksschwach, es stellt eine Konzession aufgrund der fehlenden Wissensrepräsentationssprache dar. Die Aktivierungshierarchie hingegen ist einfach zu verstehen und ermöglicht die für reaktives Verhalten so wichtige effiziente Verarbeitung.

Der kooperativen Diensterbringung sind Grenzen aufgesetzt. Damit ein Agent einen fremden Dienst nutzen kann muß er dessen genaue Beschreibung, also den Namen der Task und deren Parameter, kennen. Zwei Tasks sind nur dann vergleichbar, wenn sie dieselbe syntaktische Struktur besitzen. Zur Abhilfe dieser Einschränkungungen kommen üblicherweise Ontologien zum Einsatz, die sehr viel weiter reichende Beschreibungsmöglichkeiten bieten. Die REkoS-“Ontologie“, bestehend aus den in Abschnitt 5.2 eingeführten generischen Begriffen, wäre um allgemeinere Konzeptualisierungen zu erweitern.

Insgesamt hat sich gezeigt, das die Verwendung der Programmierwerkzeuge erhebliche Zeitersparnis bringt. Die von Programmierungsdetails abstrahierende Sicht erleichtert die Generierung fehlerarmer Programmdateien.

KAPITEL 6 Eine Agentenarchitektur für Dienste

In diesem Kapitel wird die REkoS-Agentenarchitektur vorgestellt. Sie stellt die Ablaufumgebung für Agenten dar, die mit den im letzten Kapitel vorgestellten Beschreibungssprachen programmiert wurden. Es handelt sich um eine Komponentenarchitektur mit drei Verarbeitungsschichten, die die Basisfähigkeiten Interaktion, Handeln und Reasoning widerspiegeln.

Zunächst werden Anforderungen an die Architektur, wie sie sich aus Sicht des Anwendungsgebiets ergeben, erörtert. Sie fließen als Designentscheidungen in die Struktur und angebotene Funktionalität der Architektur ein. Anschließend wird ein Überblick über den generellen Aufbau und Funktionsweise der Architektur gegeben. Mit dem dritten Abschnitt beginnt eine detaillierte Beschreibung der einzelnen Komponenten. Als erstes wird der Interpreter vorgestellt, der den allgemeinen Ablauf der Informationsverarbeitung steuert. Es folgen Beschreibungen des Intentionalitätsmoduls und der Wissensbasis, wo die programmierten Fähigkeiten und mentalen Zustände repräsentiert und verarbeitet werden. Die Abschnitte 6.6 und 6.7 erläutern die für das Aktivitätenmanagement zuständigen Komponenten: den Scheduler und die handlungsausführenden Module Skript- und Kooperationsmanager. Im Abschnitt über das Kommunikationsmodul wird dessen Funktionsweise beschrieben und dabei auf die Realisierung eines Directory-Dienstes eingegangen. Abschließend werden die Ergebnisse und Erfahrungen zusammengefaßt und bewertet.

6.1 Anforderungen an die Architektur

Architekturen spielen für die Agentenprogrammierung eine vergleichbare Rolle wie Betriebssysteme für „klassische" Programmierung. Ihre Aufgabe liegt darin, eine Ausführungsumgebung mit Schnittstellen zur Steuerung und zum Zugriff auf Ressourcen bereitzustellen. Die Architektur muß also fähig sein, Programme, die in der Agentenprogrammiersprache formuliert sind, zu interpretieren. Welche zusätzlichen „generischen" Funktionalitäten als integraler Bestandteil der Architektur zu realisieren sind, ergibt sich aus einer Betrachtung der Anforderungen kooperativer Diensterbringung sowie allgemeinen Betrachtungen in bezug auf die Ablaufsteuerung.

Bei der kooperativen Problemlösung treten Agenten sowohl als Anbieter wie auch als Nachfrager von Leistungen auf. Dasselbe gilt für die kooperative Diensterbringung: Komplexe Dienste entstehen durch dynamische Nutzung einfacherer, von Agenten angebotener Basisdienste. Wenn ein Dienst nun von mehreren, u.U. nicht a priori bekannten Agenten erbracht wird, hat dies Konsequenzen, die nachfolgend kurz erörtert werden.

- **Überwachung:** Durch kooperative Diensterbringung entstehen temporäre Abhängigkeiten zwischen Agenten. Wenn Teilaspekte einer komplexen Aufgabe an andere Agenten ausgelagert werden, müssen dem beauftragenden Agenten Mechanismen zur Überwachung der ihm verpflichteten Agenten zur Verfügung stehen, die ihm erlauben, Engpässe und Probleme zu erkennen und entsprechende Maßnahmen zur Beseitigung zu ergreifen. Mögliche Ansätze bieten Verhandlungs- und Kooperationsprotokolle, die automatisch im Falle einer kritischen Situation starten. Überwachung besitzt auch eine lokale Dimension: Jeder Agent muß in der Lage sein, die Leistungsparameter seiner eigenen Aktivitäten zu kontrollieren, um verpflichtete Zusagen einhalten zu können.
- **Aktivitätenmanagement:** Die in Abschnitt 2.3 vorgestellten Koordinierungsprobleme treten auch in kooperativen Dienstumgebungen auf. Zeitliche Vorgaben müssen zusicherbar sein. Auch soll ein Agent bei fehlenden oder knappen Ressourcen nicht einfach terminieren oder die Verarbeitung abbrechen, sondern zunächst nach Lösungsmöglichkeiten suchen. Hier bieten die interaktiven Fähigkeiten der Agenten Lösungsansätze; beispielsweise in Form von Informationsbeschaffung, Verhandlung oder Aufgabendelegation. Weiterhin zum Aktivitätenmanagement gehört ein Rahmenwerk zum kontrollierten Unterbrechen, Wiederaufnehmen und Beenden von Aktivitäten.
- **Leistungs- und Qualitätsmanagement:** Aus Sicht des Systemdesigners oder Administrators sind Performanceaspekte interessant. Langfristige Beobachtungen des Systems sollen Aufschluß über Häufungen bestimmter Fehlertypen geben oder auf Leistungsengpässe schließen lassen können. Als Grundlage für die Beobachtung von Leistungsmerkmalen des Gesamtsystems sind umfangreiche Laufzeitdaten zu sammeln. Hierunter fällt z.B. das Überwachen von Leistungs-

parametern, die die Anzahl empfangener oder versandter Nachrichten beschreiben oder das Volumen der übertragenen Daten bzw. die Übertragungszeiten messen.

- **Accounting:** Das Sammeln von mit Diensten im Zusammenhang stehenden Laufzeitdaten wie Kosten und verbrauchte Zeit ist notwendig, um erbrachte Leistungen in Rechnung stellen zu können.
- **Transparenz:** Die Dienstverarbeitung muß nachvollziehbar sein. Wurde ein Dienst durch mehrere Agenten erbracht, sollte dies dem Auftraggeber ersichtlich gemacht werden. Auch nicht eingeschlagene alternative Lösungspfade und Fehlerquellen, die zu Verzögerungen oder Qualtiätsminderungen führen, sollten angezeigt werden. Letztere Informationen sind insbesondere für Systemadministratoren interessant, können sie doch Aufschluß über Flaschenhalssituationen und schlechtes Design bringen.
- **Präferenzen:** Die generischen Dienstparameter Zeit, Kosten und Qualität sind gewöhnlich nicht voneinander unabhängig. Dienste mit höchster Güte in kürzester Zeit kosten üblicherweise mehr als Dienste mit geringeren Anforderungen an Schnelligkeit und Qualität. Der Dienstnutzer sollte die Möglichkeit besitzen, die ihm wesentlichen Merkmale herauszustellen. Dies kann beispielsweise durch eine partielle Ordnung der Parameter oder durch Vorgabe einer Optimierungsfunktion geschehen.

Die Repräsentation von Fähigkeiten und Wissen ist durch die im vorigen Kapitel vorgestellten Beschreibungssprachen - bzw. den von den Übersetzern generierten Code - bereits vorgegeben. Auch Aspekte der internen Ablaufkontrolle werden durch die Intentionalitätsbeschreibungssprache bereits abgedeckt. Darauf aufbauend lassen sich weitere, die Informationsverarbeitung betreffende Anforderungen an die zu konzipierende Agentenarchitektur formulieren.

- **Kommunikation:** Die Architektur sollte sowohl synchrone als auch asynchrone Nachrichtenübermittlung unterstützen. Im Falle von synchroner Kommunikation sollte die auf eine Antwort wartende Aktivität nicht durch busy-waiting die Performance herabsetzen.
- **Parallele Verarbeitung:** Sowohl das lokale Aktivitätenmanagement als auch die angebotenen diensterbringenden Fähigkeiten sollen parallel ablaufen. Dadurch kann ein Agent mehrere Dienste gleichzeitig erbringen und die laufende Arbeit muß nicht für andere Aktivitäten, z.B. Managementaufgaben, unterbrochen werden. Nach außen hin ergibt sich das Bild eines kontinuierlich mit Diensten beschäftigten Agenten, bei dem die Koordinierungs- und Managementtätigkeiten im Hintergrund ablaufen. Im Umfeld knapper Prozessorressourcen sollte das Maß an Parallelität allerdings auf einen vernünftigen Wert eingeschränkbar sein, da die Prozeßverwaltung im allgemeinen aufwendig ist[1].
- **Lernfähigkeit:** Über Dienste, die kooperativ von mehreren Agenten erbracht werden, können im allgemeinen a priori keine zuverlässigen Aussagen in bezug

auf Zeit, Kosten und Qualität gemacht werden. Daher sollte ein Agent Erfahrungen, die aus der Zusammenarbeit mit anderen Agenten resultieren nutzen, um über die Zeit zu besseren Vorhersagen zu kommen. Auch bei rein lokal erbrachten Diensten macht Datensammlung und -auswertung Sinn, ist doch die Performance eines Agenten von seiner Rechnerumgebung und der Beanspruchung durch andere Agenten abhängig.

- **Persistenz:** Persistente Datenhaltung ermöglicht ein Wiederaufsetzen nach einem Systemabsturz. Bereits ausgeführte Berechnungen müssen dann nicht wiederholt werden, und auf erlerntes Wissen ist weiterhin abgreifbar. Der Idealfall, daß ein Agent bei der zuletzt durchgeführten, unterbrochenen Tätigkeit wieder aufsetzen kann, läßt sich im allgemeinen jedoch nicht realisieren, weil dazu sämtliche Laufzeitdaten gespeichert werden müßten[2]. Hier ist eine Abwägung zwischen Sicherheitsanspruch und effizientem Laufzeitverhalten vorzunehmen.

Sicherheitsmanagement ist nicht Gegenstand dieser Arbeit. In diesen Bereich fallen elektronische Sicherheitsmechanismen zur Authentisierung, Autorisierung, Verschlüsselung und auch datenschutzrechtliche Aspekte. Desweiteren sind Mechanismen zur Einbruchserkennung und -verfolgung zu erwähnen.

6.2 Das Architekturmodell

Zum besseren Verständnis der in den nachfolgenden Abschnitten beschriebenen komplexen Zusammenhänge der entwickelten Agentenarchitektur wird an dieser Stelle ein grober Überblick über ihren Aufbau und die prinzipielle Funktionsweise gegeben.

Die REkoS-Agentenarchitektur besteht aus drei Modulen, die die Basisaktivitäten Kommunikation, Handeln und Räsonnieren kapseln (Abbildung 6-1).

- Das Kommunikationsmodul ist die Schnittstelle eines Agenten zu seiner Umwelt. Wichtigste Aufgabe ist die Entkopplung der Wahrnehmung von den übrigen Tätigkeiten des Agenten. Denn nur kontinuierliche Aufmerksamkeit kann garantieren, daß der Agent sämtliche relevanten Ereignisse erkennt und nicht etwa wichtige Informationen durch „temporäre Unaufmerksamkeit“ verloren gehen. Die Kommunikation mit anderen Agenten wird über den Austausch von Sprechakten abgewickelt und bietet mit Wahrnehmungen und Aktionen einen allgemeinen Mechanismus zur Interaktion mit nicht-agentischen Entitäten.

1. Mehrere Aktivitäten können „quasi-parallel“ als unabhängige Prozesse auf einem Prozessor oder als „Leichtgewichtsprozesse“ (Threads) innerhalb einer Prozeßumgebung ablaufen. Das Scheduling der Aktivitäten wird vom Betriebssystem bzw. der Prozeßumgebung automatisch gesteuert.

2. Die besten Möglichkeiten zum Wiederaufsetzen bieten Interpreter, aber auch nur solange wie der die Interpreterumgebung selbst nicht abstürzt.

Hierbei handelt es sich nicht um echte Signale oder physische Aktivitäten, sondern um softwareseitige Simulationen, die durch Objektzustände repräsentiert werden: Änderungen von Objektzuständen können wahrgenommen oder durch Aktionen veranlaßt werden. Auf diese Weise können beliebige Programme, wie z.B. Datenbanken angesprochen werden.

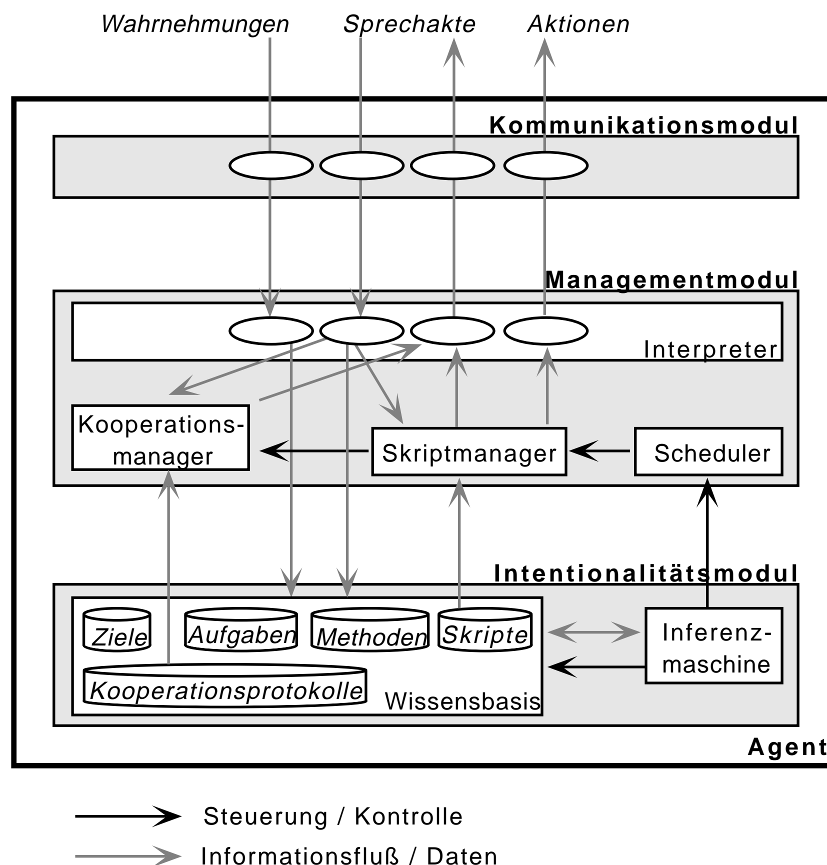

Abbildung 6-1: Die REkoS-Agentenarchitektur

- Im Managementmodul findet die Abarbeitung und Kontrolle der Dienste statt. Vier Komponenten realisieren die unterschiedlichen Aufgaben: Der Interpreter fungiert als zentrale Steuerungskomponente des Agenten, die den gesamten Ablauf der Informationsverarbeitung kontrolliert. Er leitet ankommende Nachrichten des Kommunikationsmoduls an die verantwortlichen Komponenten weiter, stößt Verarbeitungsprozesse an und stellt ausgehende Botschaften dem Kommunikationsmodul zu. Der Skriptmanager ist die für die Ausführung und Kontrolle der aktiven Skripte verantwortliche Komponente. Vorgeschaltet ist ein Scheduler, welcher eine Reihenfolgeplanung vornimmt, freigegebene Skripte dem Skriptmanager übergibt und noch nicht auszuführende Skripte puffert. Länger-

fristige Interaktionen mit anderen Agenten werden durch Kooperationsprotokolle realisiert. Sie werden von einer eigenen Komponente, dem Kooperationsmanager verwaltet.

- Das Intentionalitätsmodul enthält die Wissensbasis und eine darauf operierende Inferenzmaschine. In der Wissensbasis sind zum einen die prozeduralen Fähigkeiten in Form von Skript- und Kooperationsprotokoll-Klassenbibliotheken untergebracht als auch die Repräsentation der mentalen Attitüden. Außerdem dient die Wissensbasis als Speicher für Laufzeitdaten und Zwischenergebnisse. Aufgabe der Inferenzmaschine ist es, aus den durch Wahrnehmungen und eingehende Sprechakte resultierenden Änderungen der Wissensbasis sinnvolle Maßnahmen abzuleiten und diese dem Scheduler zu übermitteln.

Die einzelnen Komponenten arbeiten unabhängig voneinander, jede in einem eigenen Thread. Dadurch wird die Problematik der Koordinierung der einzelnen Tätigkeiten weitgehend entschärft. Kommunikation zwischen Komponenten findet durch asynchronen Nachrichtenaustausch statt. Dazu besitzt jeder Thread einen Nachrichtenpuffer, den er zyklisch abarbeitet. Das Kommunikationsmodul und die Komponenten des Managementmoduls sind in C++ implementiert; für das Intentionalitätsmodul wurde eigens ein Prolog-Interpreter[3] integriert, so daß die Wissensrepräsentation und Inferenzmechanismen deklarativ und logikbasiert durchgeführt werden.

6.3 Interpreter

Der Interpreter realisiert mit seiner Hauptarbeitsschleife die zentrale Steuerung eines REkoS-Agenten. Von hier aus werden die neu eingetroffenen Informationen verarbeitet, Motivationen und Zielsetzungen aktualisiert und daraus Aktivitäten für den Agenten abgeleitet.

In Abbildung 6-2 beschreibt die obere Schleife 1 den Verarbeitungszyklus des Interpreters. Im ersten Schritt werden alle seit dem letzten Durchlauf vom Kommunikationsmodul empfangenen Sprechakte und Wahrnehmungen aus dem Eingangspuffer gelesen und dem Intentionalitätsmodul übergeben. Dort wird, entsprechend den dort definierten Verhaltensregeln, die Wissensbasis des Agenten aktualisiert. Aufgrund der neuen Weltsicht sind Änderungen in den Motivationen möglich, die wiederum Einfluß auf die gültigen Ziele des Agenten haben. Im Ergebnis liefern die Berechnungen des Intentionalitätsmoduls eine Menge von Maßnahmen, die die Aktivierung und Deaktivierung von Aufgaben betreffen. Diese Maßnahmenliste wird dem Interpreter durch Eintrag in dessen Input-Puffer zugestellt.

3. Es handelt sich hierbei um C-Prolog, dessen C-Quellcode frei verfügbar ist.

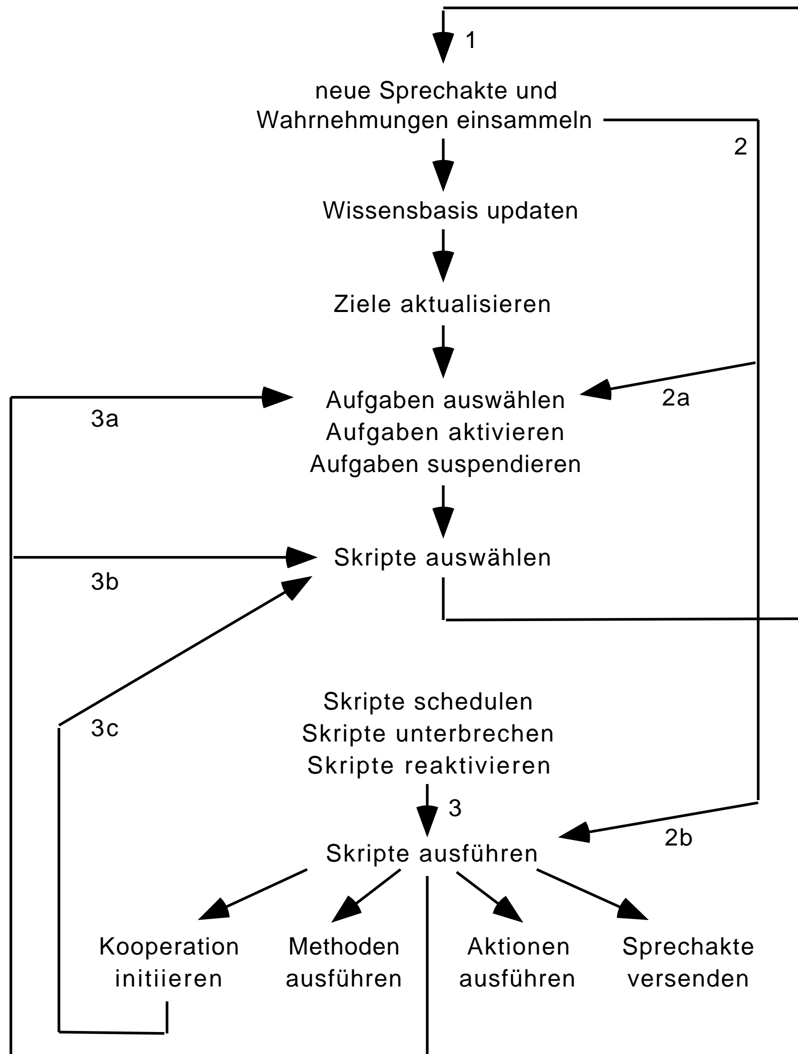

Abbildung 6-2: Interpreterschleife

Bei einer Aufgabenaktivierung prüft der Interpreter, ob die Task bereits aktiv ist. Dies geschieht durch Zugriff auf eine Hash-Tabelle, in der der Interpreter alle aktiven Aufgaben verwaltet. Existiert in der Tabelle eine entsprechende Task-Instanz mit der „Einfachaktivierungseigenschaft“, so wird ihr eine Begründung hinzugefügt, deren Inhalt das zur Aktivierung führende Ziel ist. Ansonsten wird eine neue Task-Instanz erzeugt und das Intentionalitätsmodul beauftragt, ein passendes Skript zu finden. Auch dieses wird als Ergebnis in den Eingangspuffer eingetragen, im nächsten Zyklus ausgelesen, daraus eine Skriptinstanz gebildet und dem Scheduler zur Einplanung übergeben.

Während der unter 1) beschriebene Weg die reflektive Arbeitsweise des Agenten beschreibt, drücken die Zweige 2a) und b) reaktive Abläufe aus. Sensing-Alarms führen zur direkten Aktivierung von Tasks (2a), erwartete Antworten oder Sprechakte, die Bestandteil von Kooperationsprotokollen sind, werden direkt an die entsprechende Stelle weitergeleitet (2b).

In Schleife 3 werden ausgewählte Skripte abgearbeitet. Vorgeschaltet ist ein Scheduler, der die Anzahl gleichzeitig laufender Skripte in Grenzen hält und das Unterbrechen und Wiederaufsetzen aktiver Skripte verwaltet. Skripte laufen in eigenen Prozessen, so daß der Interpreter weiterarbeiten kann, ohne auf ihre Beendigung warten zu müssen. Die Ausführung eines Skripts findet durch Interpretation statt und wird durch den Skriptmanager kontrolliert. Hierbei können mehrere Sonderfälle eintreten:

a) Wenn in einem Skript auf ein weiteres Skript verwiesen wird, so wird letzteres in die Menge der ausgewählten Skripte aufgenommen und bei nächster Gelegenheit vom Agenteninterpreter aktiviert.

b) Wird auf eine Aufgabe verwiesen, so stellt dies eine Aufgaben- Subaufgaben-Beziehung dar. Sie besteht zwischen der ursprünglichen Aufgabe, für die das Skript aktiviert wurde und der Aufgabe im Skript. Solche Aufgaben werden in die Menge der ausgewählten Aufgaben aufgenommen und vom Agenteninterpreter aktiviert.

c) Wird auf ein Interaktionsprotokoll verwiesen, so wird die entsprechende Instanz initialisiert und das in ihr enthaltene Protokoll aktiviert. Die im betreffenden Rollenplan enthaltenen Kooperationsskripte werden in die Menge der ausgewählten Skripte übernommen und aktiviert.

TABELLE 6. verarbeitbare Nachrichten des Interpreters

Ursprung	Inhalt	Verarbeitung
Kommunikationsmodul	Anfrage-Sprechakt	Antwortpuffer vorbereiten; Empfangsprozedur im Intentionalitätsmodul aktivieren
	Antwort-Sprechakt	in Antwortpuffer eintragen; Empfangsprozedur im Intentionalitätsmodul aktivieren
	anderer Sprechakt	Empfangsprozedur im Intentionalitätsmodul aktivieren

TABELLE 6. verarbeitbare Nachrichten des Interpreters

Ursprung	Inhalt	Verarbeitung
Intentio-nalitäts-modul	aktiviere Task	wenn Task aktiv: Begründung hinzufügen sonst: Task-Instanz ziehen; Skriptauswahl im Intentionali-tätsmodul veranlassen
	deaktivie-re Task	Begründung entfernen; wenn Task ohne Begründung: Task-Instanz löschen und Skriptabbruch beim Scheduler veranlassen
	aktiviere Skript	Skriptstart im Scheduler veranlassen
Scheduler	starte Skript	mit der Ausführung eines Skripts beginnen
	stoppe Skript	die Ausführung eines Skripts beenden
	unterbre-che Skript	zwecks späteren Wiederaufsetzens die Ausführung eines Skripts unterbrechen
Skriptma-nager	Skripten-de	das Skriptergebnis dem Intentionalitätsmodul zuführen den Scheduler über das Skriptende informieren
Skript	sende Sprechakt	Sprechakt an das Kommunikationsmodul weiterleiten

6.4 Intentionalitätsmodul

Das Intentionalitätsmodul kontrolliert die Wissensbasis. Es führt Aktualisierungen aufgrund empfangener Sprechakte und Sensings durch und leitet daraus Maßnahmen für zukünftiges Handeln ab. Teile der Funktionalität dieses Moduls sind agentenspezifisch und werden durch die Agentenprogrammierung in den Bereichen der Sprechaktverarbeitung und der Intentionalität, wie im vorigen Kapitel beschrieben, erzeugt.

Wichtigste Aufgabe des Intentionalitätsmoduls ist die Verwaltung des aus Variablen, Motivationen, Zielen und Aufgaben bestehenden vierstufigen Netzes mentaler Attitüden. Aus der Verarbeitungsschleife des Interpreters heraus werden zyklisch die Änderungen der Wissensbasis in das Netz eingespeist und entlang der Aktivie-

rungshierarchien durchpropagiert. Dieses inkrementelle Wissens-Update entspricht der Funktion eines *Truth Maintenance System*[4].

6.4.1 Verwaltung von Instanzen

Die 1:n-Beziehungen in der Aktivierungshierarchie (siehe Abbildung 5-2 auf Seite 130) zwischen Motivationen, Zielen, Tasks und Skripten können zu Mehrfachaktivierungen führen. Es macht jedoch keinen Sinn, eine Motivation oder ein Ziel mehrfach zu verfolgen. Stattdessen sollte ein Behaviour, das mehrfach aktiviert wird, eine höhere Wichtigkeit besitzen bzw. fokussiert verfolgt werden. Analoges gilt in den meisten Fällen auch für Tasks und Skripte: Eine von mehreren Seiten verlangte Tätigkeit wird am besten nur einmal durchgeführt, wobei alle auftraggebenden Parteien am Ergebnis partizipieren.

Motivationen, normative Ziele, Zustandsziele, Tasks und Skripte nehmen zu jedem Zeitpunkt einen der Zustände *aktiv* oder *inaktiv* ein. Die Steuerung dieser Zustände erfolgt für die beiden erstgenannten Behaviours über ihre Aktivierungsbedingungen (siehe Abschnitt 5.4.2 und 5.4.4).

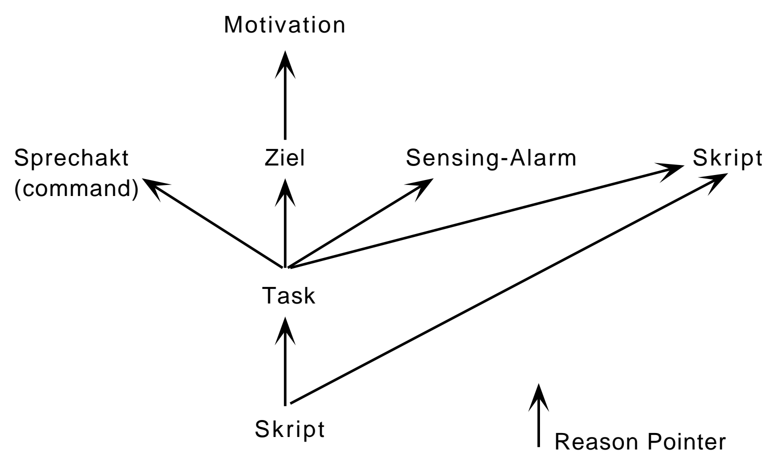

Abbildung 6-3: Begründungshierarchie durch Reason-Pointer

Mehrfachaktivierungen von Zielen, Tasks und Skripten werden durch *Begründungslisten* verwaltet. Mit jedem dieser mentalen Begriffe ist eine solche Liste assoziiert, in die die Aktivierungsursachen eingetragen werden. Skripte werden durch

4. [Doyle 1979].

Tasks aktiviert, Tasks haben als Ursachen Ziele, die wiederum durch Motivationen begründet sind. Die Begründungen werden durch sogenannte Reason-Pointer ausgedrückt und bilden eine Hierarchie, die entgegengesetzt zur Aktivierungshierarchie gerichtet ist. Inaktive Elemente besitzen leere Begründungslisten. Eine erste Aktivierung sorgt für den ersten Eintrag in die Liste, wodurch der Zustand auf aktiv wechselt. Werden weitere Begründungen eingetragen, so ändern diese nichts am aktiven Zustand. Behaviours werden erst dann inaktiv, wenn der letzte Reason-Pointer aus der Begründungsliste entfernt wurde.

Zwei Mechanismen sind zur Aktualisierung des mentalen Zustands notwendig. Einerseits beeinflußt neues Wissen die Motivationen des Agenten. Hierfür werden, angefangen bei den Motivationen, Maßnahmen bis hin zur Skriptebene durchpropagiert. Andererseits werden Aufgaben, Ziele und Motivationen durch abgearbeitete Skripte erfüllt. Diese Abhängigkeiten entsprechen einer Rückwärtspropagierung von Skriptergebnissen über Tasks bis zu den Zielen.

6.4.2 Vorwärtspropagierung von Motivationen

Die Vorwärtspropagierung wird zyklisch vom Interpreter angestoßen. Dieser Prozeß beginnt damit, daß die Aktivierungsregeln aller Motivationen getestet werden. Nach diesem Schritt stehen die neuen Aktivierungszustände der Motivationen fest. Diese werden abgespeichert. Weil nur Zustandsänderungen Auswirkungen auf die unteren Ebenen haben können, findet zunächst ein Abgleich der aktualisierten mit den bisherigen Zustandsinformationen statt. Ergebnis des ersten Schritts ist eine Liste, bestehend aus neuen und entfernten Motivationen. Motivationen, deren Zustand derselbe geblieben ist, werden nicht weiter betrachtet.

Im zweiten Schritt findet die Aktualisierung der Ziele anhand der veränderten Motivationen statt. Für jede neu aktivierte Motivation werden alle abhängigen Ziele verstärkt, indem die Motivation in die Begründungsliste eingetragen wird. Umgekehrt führen deaktivierte Motivationen zur Entfernung aus den Begründungslisten und damit zur Abschwächung aktiver Ziele. Für den nächsten Verarbeitungsschritt werden nun diejenigen Ziele weiter verwendet, die eine Zustandsänderung erfahren haben.

Der dritte Schritt behandelt Maßnahmen für die Ausführungsebene, die aufgrund der Änderungen im Zielmodell des Agenten ausgelöst werden. Hierbei werden die Konsequenzen für den Ausführungsstatus von Tasks und Skripten berechnet. Starten, Verstärken, Abschwächen und Stoppen von Aufgaben erfolgt wie in Schritt zwei über die Begründungslisten. Während die Behandlung von einmal ausführbaren Aufgaben identisch mit der von Zielen funktioniert, gilt für mehrfach ausführbare Tasks die Regel, daß jede Aktivierung zu einer zusätzlichen Aktivierung führt[5]. Es können dann durchaus mehrere Instanzen einer Aufgabe gleichzeitig vor-

handen sein. Nach dem Start einer Task wird die Auswahl eines geeigneten Skripts, mit der Task als Reason, initiiert. Analog führt das Stoppen einer Aufgabe zur Beendigung des assoziierten Skripts. Diese Skriptmaßnahmen werden als Ergebnis der Vorwärtspropagierung des Intentionalitätsmoduls dem Interpreter übergeben und anschließend vom Scheduler ausgeführt.

6.4.3 Skriptauswahl

Zur Ausführung eines Handlungsskripts kommt es entweder durch einen Aufruf aus einem Skript heraus oder aufgrund einer Taskaktivierung. Im ersten Fall ist das Skript benannt und wird direkt ausgeführt. Ist der Reason eine Task, können mehrere Skripte zur Auswahl stehen. Die Bestimmung der letztendlich auszuführenden Handlung erfolgt dann durch eine Reihe von Auswahl- und Filteroperationen.

- **Skriptauswahlregeln:** Falls für die zu aktivierende Aufgabe Skriptauswahlregeln definiert sind, werden diese ausgewertet. In den Regeln kommt domänenspezifisches Wissen zum Einsatz, mit dem anhand der vorgegebenen Task-Parameter eine Vorselektion der in Frage kommenden Skripte vorgenommen wird.
- **Erfahrungswissen:** Im Rahmen des Lernvorgangs angesammeltes Wissen über Kosten und Dauer der Skriptausführung kann zum Ausschluß weiterer Kandidaten führen. Dies ist möglich, wenn die Aktivierung durch einen Sprechakt oder aus einem Skript heraus stammt, weil in diesen Fällen üblicherweise Sollvorgaben für die genannten Argumente vorliegen.
- **Normative Ziele** wie Kostenminimierung oder Qualitätsmaximierung werden zum Schluß berücksichtigt, falls noch mehrere Skripte zur Auswahl stehen. Durch die normativen Ziele findet eine Sortierung nach Eignung statt. Sind mehrere Ziele aktiv, werden entsprechend mehrere Sortierungen durchgeführt und anschließend zu einer Liste integriert. Hierbei werden die Mittelwerte der Positionierungen in den einzelnen Sortierungen zur Ermittlung der Position in der Ergebnisliste benutzt.

Durch die Filteroperationen können drei Situationen eintreten, die unterschiedlich behandelt werden. Entweder ist die Menge der übrig gebliebenen Skripte auf 0, 1 oder mehrere Elemente reduziert:

a) Kein Skript: Der Agent die ihm übertragene Aufgabe nicht ausführen. In diesem Fall wird ein Kooperationsprotokoll der Klasse „Aufgabenverteilung“ mit der aktivierten Task als Verhandlungsgegenstand gestartet. Die Task bleibt bis zur Beendigung des Protokolls aktiv mit dem azzoziierten Kooperationsskript als

5. Einmal ausführbare Tasks sind, genauso wie Ziele, auch bei mehrfachen Aktivierungen nur einmal aktiv. Mehrfach ausführbare Aufgaben hingegen werden jedesmal neu gestartet (siehe Abschnitt 5.4.5 „Tasks“ auf Seite 134).

Begründung. Wenn die Protokollausführung scheitert oder kein Kooperationsskript vorhanden ist, liegt ein Fehlerfall vor, dessen Behandlung in Abschnitt 6.4.4 behandelt wird.

b) Ein Skript: Das Skript wechselt mit der Task als Reason in den Zustand *gestartet*. Zur Ausführung wird es mit seinen Parametern dem Interpreter zwecks Weitergabe an den Scheduler übergeben.

c) Mehrere Skripte: Gewöhnlich kommt das am höchsten priorisierte Skript, wie unter b) beschrieben, zur Ausführung. Weil die Priorisierung normalerweise auf Erfahrungswissen beruht, das wiederum durch Skriptausführung gesammelt wird, kann es sinnvoll sein, mitunter auch minder priorisierten Skripten eine Chance zu geben, um deren durchschnittliche Performancedaten zu aktualisieren. Hierfür dient das normative Ziel „Lernen“, mit dem die Skriptauswahlliste umsortiert werden kann. Die Realisierung dieser Neuanordnung ist dem Programmierer überlassen. Existieren mehrere gleich priorisierte Skripte, wird das erste in der Liste gewählt. Auch hier kann das genannte normative Ziel eine Änderung der Auswahl vornehmen.

6.4.4 Rückwärtspropagierung von Skriptergebnissen

Die Ausführung eines Handlungsskripts kann entweder gelingen oder scheitern. Der Erfolg oder Mißerfolg wird vom Skriptmanager über den Interpreter dem Intentionalitätsmodul mitgeteilt, wodurch die Rückwärtspropagierung angestoßen wird. Hierbei werden die Konsequenzen von der Skriptebene bis hinauf zu den Motivationen berechnet. Eingeleitet wird dieser Prozeß durch einen Zustandswechsel des Skripts auf *inaktiv*.

Nach erfolgreicher Abarbeitung eines Skripts wird die verantwortliche Task über den Reason-Pointer ausfindig gemacht und ebenfalls deaktiviert. Dies zieht eine Abschwächung aller Ziele nach sich, die für die Taskaktivierung verantwortlich zeichnen. Dort endet die Propagierung.

Scheiterte jedoch die Ausführung des Skripts, so wird mit einem Backtracking-ähnlichen Mechanismus nach alternativen Lösungen gesucht. Zunächst wird erneut das Ergebnis der Skriptauswahl betrachtet. Standen dort alternative Skripte zur Auswahl, so werden diese noch einmal, unter Berücksichtigung des aktuellen Weltzustands[6], in Betracht gezogen. Falls möglich kommt erneut der aussichtsreichste Kandidat zur Auswahl, ansonsten wird eine Kooperation mit dem Ziel der Aufgabenverteilung gestartet.

6. Fortgeschrittene Zeit, akkumulierte Kosten und anderes Domänenwissen können die Ausführung bisheriger Kandidaten verhindern. Daher müssen die beiden ersten Filteroperationen aus Abschnitt 6.4.3 erneut ausgeführt werden.

Fehlerbehandlung

Nur wenn alle diese Mittel versagen, ist das Scheitern unwiderruflich. Die Task wird deaktiviert und die Fehlersituation an alle unterstützenden Behaviours weiterpropagiert[7]. Dort findet eine typspezifische Fehlerbehandlung statt:

- **Ziel:** Das Ziel wird in den inaktiven Zustand zurückgesetzt und eine Vorwärtspropagierung wie in Abschnitt 6.4.2 durchgeführt. Dies führt zur Deaktivierung aller vom Ziel abhängigen Tasks, sofern diese keine weiteren Aktivierungsursachen besitzen.
- **Sensing-alarm:** Eine Fehlermeldung wird ausgegeben, weil offensichtlich ein Modellierungsfehler vorliegt: Die programmierte - und damit vorgesehene - Ausnahmebehandlung konnte nicht durchgeführt werden.
- **Sprechakt:** Der Dienst wurde auf Anfrage oder Anforderung eines anderen Agenten gestartet. An den verpflichtenden Agenten wird ein sorry-Sprechakt anstelle des Ergebnisses zurückgeschickt.
- **Skript:** Eine gescheiterte Subtask bewirkt das Scheitern des umliegenden Skripts, auf das sich sich die in diesem Abschnitt beschriebene Fehlerbehandlung verlagert.
- **Kooperationsskript:** Ein Fehler innerhalb eines Kooperationsskripts ist gleichbedeutend mit dem Scheitern der Kooperationsrolle des Agenten. Der Agent steigt aus der Kooperation aus und teilt dies dem Kooperationsmanager-Agenten mit. Ist er selbst der Protokollinitiator, sendet er allen beteiligten Rollen die Abbruchnachricht.

Rückwärtspropagierung endet immer auf der Zielebene. Motivationen werden nicht tangiert, weil ihre Aktivität schließlich nicht von Zielen, sondern von gesonderten Stimulanzen (siehe Abschnitt 5.4.2 „Motivationen“) abhängt.

TABELLE 7. verarbeitbare Nachrichten des Intentionalitätsmoduls

Ursprung	Inhalt	Verarbeitung
Interpreter	Wissens-basis-Update	Fakten in Wissensbasis eintragen; eventuell: Belief revision; Task-Maßnahmen berechnen und Interpreter übergeben

7. Zur Identifikation der Task-Aktivierungsursachen dient wieder die Begründungsliste.

TABELLE 7. verarbeitbare Nachrichten des Intentionalitätsmoduls

Ursprung	Inhalt	Verarbeitung
	Skriptauswahl	Skript anhand aktueller Task-Parameter und aktueller normativer Ziele selektieren und Interpreter übergeben; Wenn kein Skript auswählbar, Kooperation starten oder Fehlermeldung
	empfangener Sprechakt	Empfangsprozedur ausführen, falls vorhanden, Fehlermeldung, falls Sprechakt oder Inhalt unbekannt; eventuelles Ergebnis in Form instantiierter Variablen zurück an Interpreter geben
	sende Sprechakt	Sendeprozedur ausführen
Skriptmanager	Skript beendet	Zugehörige Task deaktivieren und Taskbeendigung nach oben durch das Intentionalitätsnetz propagieren
	Skript abgebrochen	lokale Fehlermaßnahmen einleiten (anderes Skript auswählen), andernfalls Fehlerbehandlung auf Task- und Zielebene durchführen
Scheduler	Task-Priorität	Priorität einer aktiven Task aufgrund der Anzahl der Begründungen (Aktivierungen) berechnen

6.5 Wissensbasis

Die Wisssensbasis ist als einfache Prolog-Datenbasis organisiert, in die beliebiges Faktenwissen eingetragen werden kann. Stärker strukturierte Repräsentationsformen, wie sie in kognitiven Architekturen zum Einsatz kommen, werden nicht unterstützt.

6.5.1 Sammlung von Erfahrungswissen

In die Architektur wurde ein einfaches Lernkonzept integriert. Es findet eine Aufzeichnung von Leistungsdaten statt, die den Zweck hat, Handlungsentscheidungen eines Agenten mit der Zeit zu verbessern. Dabei wird die Kontrolle des Skriptmanagers über die verarbeiteten Handlungsskripte genutzt, um statistische Werte zu erheben. Durch die Reduktion auf einfache quantitative Aussagen gehen jedoch Detailinformationen verloren. So werden die unterschiedlichen Variationen eines

Skripts, wie sie durch das Auftreten von Tasks oder Kooperationsformen im Skriptrumpf auftreten können, nicht unterschieden. Insbesondere in Kooperationsprotokollen hängen Ergebnisse und Performance von der Anzahl und Qualität der beteiligten Agenten ab - mithin Parmetern, die weder statisch noch a priori bekannt sind. Aus diesen Gründen werden jeweils zwei statistische Werte ermittelt: der Mittelwert als Durchschnittsgröße und die Standardabweichung (Varianz) als Aussage zur Schwankungsbreite.

- **Zeit:** Der Skriptmanager mißt für jedes Skript die verbrauchte Zeit. Im Falle einer Suspendierung der Ausführung wird die Zeitnahme entsprechend unterbrochen. Nur für erfolgreich abgearbeitete Skripte werden durch die Zeitmessung der Durchschnittswert und die Standardabweichung ermittelt. Diese Werte finden bei der Skriptauswahl im Falle eines aktiven normativen Ziels „Zeitminimierung“ Berücksichtigung. Standardmäßig findet die Effizienzberechnung ohne Berücksichtigung der aktuellen Parameter statt. Alternativn kann im Skripteditor veranlaßt werden, daß die Verarbeitungsdauern auch einzeln für jede Parameterbelegung durchgeführt werden. Diese Maßnahme kann sinnvoll sein, wenn ein Skript nur wenige Aufrufvarianten besitzt.
- **Kosten:** Auch der generische Kostenparameter wird zur Informationssammlung genutzt. Kosten sind für alle elementaren Skriptelemente definiert und summieren sich im Zuge der Verarbeitung. Sie kommen bei der Skriptauswahl zum Tragen, wenn das vordefinierte normative Ziel „Kostenminimierung“ aktiv ist.
- **Ergebnisse:** Skriptresultate können wahlweise gespeichert werden - dies ist über ein Flag im Skripteditor einstellbar. Nutzen bringt dieses Caching wenn ein Skript komplexe Berechnungen durchführt, weil dann im Wiederholungsfall auf das bereits vorhandene Ergebnis zurückgegriffen werden kann. Entsprechend findet das Speichern für jede Parameterkombination statt. Defaultmäßig ist diese Funktionalität nicht aktiviert, weil nicht a priori ausgeschlossen werden kann, daß Skripte Seiteneffekte haben (z.B. beim Speichern einer Datei) und der Nutzen der Ergebnisspeicherung vom der Wahrscheinlichkeit der Wiederverwendbarkeit abhängt und somit anwendungsabhängig ist.
- **Fehler:** Treten Fehler bei der Skriptverarbeitung auf, werden diese protokolliert. Als Fehlerquellen kommen Überschreitungen vorgegebener Zeit- oder Kostenlimits, nicht vorhandene Fähigkeiten und Fehler in der Kommunikation vor. Die Ausfallwahrscheinlichkeiten und Fehlerursachen können für eine post-mortem-Analyse herangezogen werden und zu Aufschlüssen für besseres System- bzw. Agentendesign führen.

Auch die im Intentionalitätsmodul durchgeführten Berechnungen bieten sinnvolle Möglichkeiten zur Sammlung und Nutzung von Verarbeitungsdaten.

- **Aktivierungen:** Jede Skriptaktivierung wird mitgezählt. Das Wissen um die Häufigkeit ausgeführter Skripte wird bei der Skriptauswahl (Abschnitt 6.4.3) genutzt, um in der Lernphase auch für häufig vernachlässigte Skripte aussagekräftige Laufzeitdaten zu erfassen[8].

- **Nichtberücksichtigung:** Wenn ein Skript, das einer Task zugeordnet ist, nicht zur Ausführung kommt, wird auch dies protokolliert. Die Ursache liegt hierbei in den Filter- und Auswahlmechanismen der in Abschnitt 6.4.3 beschriebenen Skritpauswahl. Dieses Wissen kann jedoch nicht vom Agenten selbst verwendet werden, sondern dient der post-mortem-Analyse seitens des Agentenprogrammierers.

Neben diesen Wissenserhebungen, die von Agenten selbständig durchgeführt werden, bietet die Architektur zahlreiche weitere Möglichkeiten zur Datensammlung. Ermöglicht wird dies durch den Debug-Modus von Agenten, der im nachfolgenden Kapitel behandelt wird.

6.5.2 Persistente Datenhaltung

Zumeist ist es wünschenswert, den Zustand eines Agenten über dessen Lebenszeit hinaus zu speichern, damit bei einem Neustart auf dem Erfahrungswissen vor dem letzten Abbruch wieder aufgesetzt werden kann. Derartige Neustarts können beim Austausch eines Agenten durch eine neue Version oder auch nach einem Absturz notwendig werden.

Beim kontrollierten Herunterfahren eines Agenten werden die aktiven Tasks und Interaktionsprotokolle noch vollständig abgearbeitet, so daß sich der Agent am Ende im Ruhezustand befindet. Der Agent läßt sich dann vollständig durch den Zustand der Wissensbasis beschreiben. Weil über die Wiederverwendungsmöglichkeiten von Wissen generell keine zuverlässigen Aussagen getroffen werden können, speichern Agenten vor ihrem Ableben nur das im Rahmen der Leistungsdatenaufzeichnungen erlernte Erfahrungswissen. Beim Neustart wird dann automatisch dieses Wissen, zusammen mit dem initialen mentalen Zustand des Agenten, in das Intentionalitätsmodul eingelesen. Für weitergehende persistente Datenhaltung ist der Programmierer verantwortlich. Die Architektur bietet hierfür Unterstützung in Form eines vordefinierten einstelligen persistent-Prädikats, mit dem Wissensmuster in Form von Prolog-Fakten für die spätere Speicherung spezifiziert werden können.

Der Persistenzmechanismus kann auch während der Laufzeit eines Agenten über die Benutzungsoberfläche angestoßen werden. Auf diese Weise lassen sich Backups ziehen, auf denen im Falle eines unkontrollierten Programmabbruchs wieder aufgesetzt werden kann. Ein Abspeichern des Zustands aktiver Skripte ist aufgrund der möglichen Seiteneffekte des von Hand programmierten Codes nicht möglich. In

8. Die „Lernphase“ erstreckt sich über den Zeitraum, in dem alle normativen Ziele, die die Skriptauswahl beeinflussen, inaktiv sind. Damit wird eine Lernstrategie wie beim reinforcement learning (Abschnitt 2.4.5 auf Seite 64) verfolgt, gemäß der auch schwach bewertete Aktionen zur Ausführung kommen sollten, weil die schlechte Bewertung auch auf mangelnde Datenerhebung zurückzuführen sein könnte.

den meisten Fällen abgebrochener Handlungen würden ohnehin Kommunikations-Timeouts auftreten[9], so daß deren Fortsetzung oder Wiederausführung nicht mehr in Frage kommt. Außerdem erkennen die Kommunikationspartner eines Agenten dessen Nichterreichbarkeit und reagieren entsprechend durch Abbruch von Protokollen oder Suche nach anderen Informationsquellen.

6.6 Scheduler

Prinzipiell kann jedes Handlungsskript quasi-parallel in einem eigenen Thread ablaufen. Einschränkungen ergeben sich nur, wenn logische Abhängigkeiten zwischen den Skripten bestehen, die eine Parallelität verbieten - diese Situationen können bei der Beschreibung der agentenspezifischen Fähigkeiten deklariert werden, (siehe Abschnitt 5.5.3). Andererseits ist es aus Effizienzgründen wenig sinnvoll, zu viele Skripte parallel zu starten, weil damit die Performance durch den hohen Aufwand des Thread-Handlings abfällt.

Aus diesen Gründen findet zwischen der Handlungsplanung des Intentionalitätsmoduls und der Handlungdurchführung des Skriptmanagers ein Scheduling statt. Der Scheduler verwaltet die zur Ausführung freigegebenen Skriptinstanzen in einer Liste und ordnet ihnen Prioritäten zu. Er leitet die am höchsten priorisierten Instanzen dem Skriptmanager zur Abarbeitung zu und berücksichtigt dabei die festgelegte Obergrenze ausführbarer Skripte[10].

Weil die Aktivierung und Deaktivierung von Handlungen im Intentionalitätsmodul dynamisch erfolgt, wird die Schedulingfunktionalität entsprechend häufig benötigt. Bei der Konzeption des Schedulingverfahrens standen daher Effizienz und inkrementelles Arbeiten im Vordergrund.

6.6.1 Priorisierung von Skripten

Jedem Skript wird initial eine Prioritätsstufe zugeordnet, die von der Aktivierungsursache abhängt. Daneben finden zur Laufzeit Neuberechnungen statt, wenn weitere Begründungen vorliegen oder Unterbrechungen stattgefunden haben. Die Prioritätsskala reicht von 0 bis 10, wobei 10 die höchste Priorität ausdrückt.

- Stufe 10 - Skriptkreierung durch sensing-alarm. Um die Reaktivität zu gewährleisten, werden sensing-alarms mit der höchsten Priorität behandelt.

9. Bei Verwendung von Sprechakten oder aufgrund zeitlicher Zusagen für andere Agenten.

10. Die Zahl maximal parallel laufender Handlungsskripte kann dynamisch über die Agentenoberfläche eingestellt werden (siehe Abschnitt 5.7.4).

- Stufe 9 - Unterbrochene Skripte. Ein Skript, das bereits in Ausführung war, dann aber unterbrochen wurde, wird bei der Reaktivierung fokussiert weiterverfolgt. Damit ist sichergestellt, daß auch niedrig priorisierte Handlungen, sofern sie einmal gestartet wurden, zum Abschluß gebracht werden.
- Stufe 8 - Skript - Subskriptbeziehungen. Wenn ein Skript ein anderes aufruft, dann bekommt das Subskript eine hohe Priorität. Dies ist deshalb sinnvoll, weil das Subskript nicht direkt aus dem Skript ausgeführt, sondern über den Scheduler aktiviert wird. Damit ist ein ähnlicher Fall wie bei unterbrochenen Skripten gegeben. Das „Mutterskript" behält seine ursprüngliche Priorität.
- Stufen 7 bis 1 - durch Sprechakte und Ziele aktivierte Skripte. Im Falle eines Sprechakts beträgt der Defaultwert 3. Bei Zielen werden deren Prioritäten, welche vom Benutzer bei der Definition angegeben wurden, übernommen. Innerhalb des Bereichs von 1 bis 7 können Prioritäten durch das Intentionalitätsmodul geändert werden.

Jedes vom Interpreter zur Verarbeitung freigegebene Skript wird zunächst anhand der Begründung priorisiert. Wenn ein Skript durch eine neue Unterstützung verstärkt oder durch eine weggefallene Begründung abgeschwächt wird, findet eine Neupriorisierung statt. Die neue Priorität berechnet sich nach der Anzahl der Unterstützungen i und der höchsten statischen Priorität der Reasons $P_{R_x}, (1 \le x \le i)$ gemäß der folgenden Formeln:

$$Prio_{hoch}(R_i) = min(8, max(P_{R_1}, \ldots, P_{R_i})) + i - 1)$$

$$Prio_{tief}(R_j) = max(0, max(P_{R_1}, \ldots, P_{R_{j-1}}, P_{R_{j+1}}, \ldots, P_{R_i}))) + i - 2)$$

Ein Skript trägt also die Priorität seines stärksten Reasons plus eins für jede weitere Begründung. Zusätzlich wird sichergestellt, daß der Bereich von 0 bis 7 nicht über- oder unterschritten wird.

6.6.2 Aktivitätenmanagement

Interpreter und Skriptmanager tragen zu schedulende Aktivitäten in eine Agenda des Schedulers ein. Vom Interpreter kommen Informationen über zu startende und abzubrechende Skripte herein, während der Skriptmanager Benachrichtigungen zu beendeten, unterbrochenen oder abgebrochenen Handlungen sendet. Durch Verarbeitung der Agendaeinträge aktualisiert der Scheduler eine Datenstruktur, in der alle aktiven Skripte mit ihren Verarbeitungszuständen und Prioritäten enthalten sind. In seiner Rolle als Kontrollinstanz des Skriptmanagers verfolgt der Scheduler zwei Strategien: Zum einen maximiert er die Auslastung in bezug auf parallel laufende Skripte, zum anderen werden in Ausführung befindliche Skripte möglichst nicht un-

terbrochen. Ein Pünktlichkeit berücksichtigendes Zeitmanagement findet hingegen nicht statt.

Vor dem Start eines Skripts wird dessen Inkompatibilitätsliste (siehe Abschnitt 5.5.3 auf Seite 138) betrachtet. Es darf nur dann gestartet werden, wenn keines der in der Liste enthaltenen Skripte gerade in Ausführung oder unterbrochen ist. Solange die Obergrenze für parallel laufende Skripte nicht erreicht ist, leitet der Scheduler ausführbare Skriptinstanzen, nachdem er sie priorisiert und als „laufend“ vermerkt hat, an das ausführende Modul weiter. Wenn die Kapazität des Skriptmanagers ausgelastet ist, werden Skripte mit dem Vermerk „nicht laufend“ gescheduled. Eine Ausnahme bilden die sensing-alarms: Um dem Reaktivitätsanspruch zu genügen, werden die mit ihnen assoziierten Skripte sofort ausgeführt. Sollte im Skriptmanager kein Platz für ein weiteres Skript vorhanden sein, so wird dieser geschaffen, indem das am geringsten priorisierte Skript unterbrochen wird. Dieses wird als „nicht laufend“ vermerkt und bekommt die Priorität 8 zugewiesen, womit es einen vorderen Platz in der Warteliste auszuführender Skripte einnimmt.

Ein in der Warteschlange ruhendes Skript wird erst dann dem Skriptmanager zugeführt, wenn dort ein Platz frei wird und außerdem keine höher priorisierten Skripte auf Ausführung warten. Skripte, die niedrig eingestuft sind, haben also in Lastsituationen nur geringe Chancen auf Aktivierung, können aber durch Mehrfachaktivierungen in der Warteschlange nach vorne rücken.

TABELLE 8. verarbeitbare Nachrichten des Schedulers

Ursprung	Inhalt	Verarbeitung
Interpreter	schedule Skript	Eintrag der Skriptinstanz in die Prioritätenliste; wenn sensing-alarm: SensingAlarm ausführen, sonst: wenn Anzahl laufender Skripte < Obergrenze: Skriptinstanz dem Skriptmanager zur Ausführung übergeben
	beende Skript	Skripteintrag aus Schedule entfernen; wenn Skript laufend oder unterbrochen, dann Beendigung des Skripts beim Skriptmanager veranlassen
	verstärke Skript	erhöhe Priorität unter Berücksichtigung des neuen Reasons
	schwäche Skript ab	vermindere Priorität unter Berücksichtigung des weggefallenen Reasons

TABELLE 8. verarbeitbare Nachrichten des Schedulers

Ursprung	Inhalt	Verarbeitung
Skript-interpreter	Skript unterbrochen	Skriptzustand = unterbrochen; Skriptpriorität = 8; Skript mit höchster Priorität akt
	Skript beendet	Skripteintrag aus Schedule entfernen und Ergebnis dem Interpreter übermitteln; höchstpriorisierte Skriptinstanz dem Skriptmanager zur Ausführung übergeben

6.7 Skriptmanager

Der Skriptmanager verwaltet die laufenden Handlungsskripte. Er wird vom Scheduler mit Skriptinstanzen getriggert und übernimmt deren Ausführung. Ihm obliegt liegt es auch, in Fehlerfällen Skripte zu unterbrechen oder unterbrochene Aktivitäten wieder aufzunehmen. Dies kann zu einem erneuten Scheduling der Aktivitäten führen. Außerdem organisiert er die Parameterübergabe zwischen Aufgaben und Skripten beim Start bzw. Ende eines Skripts. Alle aktiven Skriptinstanzen werden mit dem dazugehörigen Prozeß, der die Skriptinstanz abarbeitet, in einer Liste verwaltet.

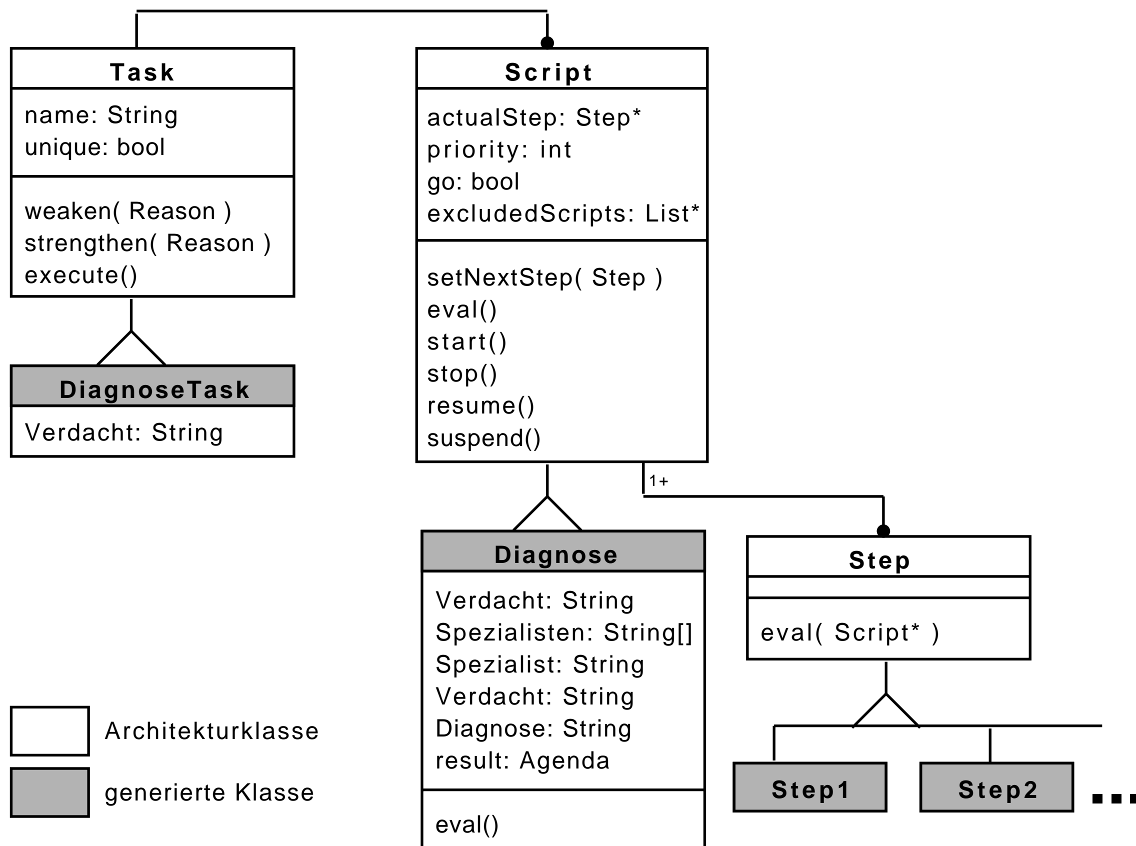

Abbildung 6-4: Beziehungen zwischen generierten und Architekturklassen

Abbildung 6-4 zeigt einen Ausschnitt aus einer Klassenbeschreibung für ein Skript. Zum Start eines Skripts wird eine spezifische Skriptinstanz (Diagnose) gezogen und deren eval-Methode angestoßen - sie setzt den ersten Step, wie bereits in Abschnitt 5.7.3 beschrieben, und ruft anschließend die eval-Methode der Vaterklasse Script auf. Script::eval() realisiert eine Schleife, die im ersten Schritt den aktuellen Step referenziert und danach dessen eval-Methode ruft (siehe Abbildung 6-5). Zwei Abbruchbedingungen beenden die Scheife: Wenn kein Nachfolge-Step existiert, wurde das Script erfolgreich abgearbeitet. Auf der anderen Seite wird die Ausführung abgebrochen, sobald das go-Flag auf false steht. In diesem Fall liegt eine äußere Ursache wie beispielsweise die Deaktivierung des für die Skriptaktivierung verantwortlichen Ziels vor.

```
1 class Script  : public Executable {
2 protected:
3   Step  *actualStep;        // the actual Step
```

```
4    REKOS_Thread myThread;     // Script's thread
5    bool go;                   // let the thread go
6    bool running;
7  public:
8  void Script::eval() {
9    Step *step;
10   while(go&&(step = getActualStep()))
11        step->eval(this);
12   // store actual costs in database ...
13   // notify scheduler of termination
14   myThread.exit(0);
15 }
```

Abbildung 6-5: Ausschnitt aus der Realisierung der Script-Klasse

Nach erfolgreichem Abschluß eines Skripts werden dessen Laufzeitdaten dem Lernprozeß des Intentionalitätsmoduls übergeben und der Thread des Skripts beendet. Der SkriptManager informiert den Interpreter über das Skriptende und liefert ihm das Ergebnis. Im Interpreter wird das Skriptresultat an alle Reasons der verantwortlichen Task weitergeleitet. Im Falle eines Sprechakts (z.B. command) wird das Ergebnis als Antwort an den Absender zurückgeschickt. Handelte es sich um ein Skript (Subskript-Beziehung), wird diesem das Ergebnis übergeben.

Einfluß auf die Skriptabarbeitung nimmt der Skriptmanager über das go-Flag und die Thread-handling-Methoden der Skriptklasse. Durch Setzen des Flags auf false wird ein Skript terminiert, Anhalten und Fortführen der Verarbeitung läuft über die Thread-Funktionalitäten suspend und resume.

TABELLE 9. verarbeitbare Nachrichten des Skriptmanagers

Ursprung	Inhalt	Verarbeitung
Scheduler	starte Skript	Ausführungs-Thread starten; ersten Step initialisieren und eval-Methode des Skripts aufrufen
	beende Skript	go-Flag des Skripts auf false setzen und Thread beenden
	unterbreche Skript	go-Flag des Skripts auf false setzen

6.8 Kooperationsmanager

Das Kooperationsmanagementmodul kontrolliert die Ausführung von Interaktionsprotokollen. Analog zum Skriptmanager verwaltet es die aktiven Protokollrolleninstanzen. Jedoch findet hier keine Thread-parallele Verarbeitung statt, stattdessen übernimmt das Managementmodul vollständig die Kontrolle und Ausführung der Kooperationsskripte. Die Ursache liegt in der besonderen Form der Interaktionsskripte begründet: Zur Ausführung kommt es erst, nachdem seine Vorbedingung erfüllt ist. Müßten für das notwendige Polling jeweils ein Thread aufgeweckt und danach (falls die Bedingung weiterhin nicht erfüllt ist) wieder schlafen gelegt werden, würde dies zu hohem Verwaltungs-Overhead führen.

Der Ablauf eines Interaktionsprotokolls gliedert sich in drei Phasen:

- Erkennen des Kooperationsbedarfs: Kooperationsbedarf wird automatisch festgestellt, wenn Situationen eintreten, die auf die Beschreibung einer dem Agenten bekannten Protokollklasse passen. Normalerweise geschieht dies bei Ausführung eines Skripts, wenn eine erforderliche Subtask vom Agenten selbst nicht (bzw. nicht in der vorgeschriebenen Qualität oder Zeit) erbracht werden kann. In diesem Fall wird eine Kooperation mit dem Ziel der Aufgabendelegation gestartet. Eine andere Situation liegt vor, wenn ein Fakt von einem anderen Agenten per tell-Sprechakt mitgeteilt wird, das im Widerspruch zum eigenen aktuellen Wissen steht. In diesem Fall könnte - falls vorhanden - ein Protokoll der Klasse „Konfliktbehandlung“ aktiviert werden. Eine weitere Möglichkeit besteht darin, ein Protokoll direkt aus einem Skript heraus zu starten. Schließlich kann die Aktivierung eines Interaktionsprotokolls auch direkt in einem Handlungsskript festgeschrieben sein.
- Initialisierung der Kooperation: Ein Agent, der Kooperationsbedarf festgestellt hat, wird zum Manager des Protokolls. Er wählt eine geeignete Protokollinstanz und informiert alle potentiell am Interaktionsprozeß beteiligten Agenten über das gewählte Protokoll. Die initiale Rollenzuordnung kann explizit per Delegation erfolgen, oder aber die angesprochenen Agenten schlüpfen selbständig in eine der angebotenen Rollen und teilen dies dem Manager mit. Das Protokoll kann gestartet werden, wenn alle initialen Rollen ausgefüllt sind. In diesem Fall sendet der Manager an alle Rollen ein Startsignal. Kann die Kooperation nicht starten, weil beispielsweise ein angesprochener Agent die Mitarbeit verweigert, steht dem Manager frei, ein anderes Protokoll zu wählen, mit dem betreffenden Agenten eine Verhandlung (wiederum in Form eines Interaktionsprotokolls) zu führen, oder das Scheitern seines Kooperationsversuchs festzustellen.
- Abarbeitung der Kooperation: Das Protokoll wird abgearbeitet, indem die beteiligten Agenten gemäß der in den Rollengraphen festgelegten Abfolge Handlungen durchführen und Nachrichten austauschen.

Zur Verwaltung der Rollenskripte existiert im Kooperationsmanager eine Nachrichtenagenda, in die alle Interaktionsprotokoll-Sprechakte vom Kommunikationsmo-

dul eingetragen werden. Die Komponente pollt nacheinander jede aktive Protokollrolle und überprüft, ob für die Ausführungsvorbedingung des aktuellen Rollenskripts gültig ist. Dazu hat sie direkten Zugriff auf alle dem Interaktionsprotokoll zugeordneten eingegangenen Sprechakte. Ist eine Vorbedingung erfüllt, wird die Auswertungsmethode des Knoten ausgeführt und anschließend - entsprechend der Nachbedingung des Skripts - auf den nächsten gültigen Folgeknoten verzweigt.

6.9 Kommunikationsmodul

Das Kommunikationsmodul besitzt zwei Nachrichtenpuffer, einen für eintreffende Sprechakte und Sensings und einen für herausgehende Sprechakte und Aktionen. Für herausgehende Nachrichten realisiert das Kommunikationsmodul die unteren Ebenen der Informationsübermittlung vom Auflösen von Agentenadressen über das Management von Kommunikationskanälen bis hin zum Datentransfer.

Weil diese Komponente die einzige Schnittstelle des Agenten zur Außenwelt realisiert, können hier Sicherheitsmaßnahmen wie kryptographische Methoden zur Ver- und Entschlüsselung von Nachrichten oder Maßnahmen zur Authentifizierung und Autorisierung eingebaut werden. Eine Ausstattung dieses Moduls mit Security-Funktionalität ist jedoch nicht Gegenstand dieser Arbeit.

Beim Senden wird im Falle einer erwarteten Antwort Speicherplatz in Form einer Agenda reserviert und eine Referenz darauf in den reply-with-Slot geschrieben. Beim Eintreffen der Antwort schreibt das Kommunikationsmodul den Nachrichteninhalt in den vorgesehenen Speicher. Die sprechaktsendende Instanz - beispielsweise ein Skript - kann durch zyklisches Abfragen des Antwortspeichers feststellen, ob eine Antwort eingetroffen ist. Dieses Vorgehen entspricht dem asynchronen Kommunikationsmodell, bei dem auch ohne eine erwartete Bestätigung die Arbeit fortgeführt werden kann. In der nächsten Phase des Sendens werden die Empfänger aus dem entsprechenden Sprechakt-Slot extrahiert. Es kann sich um einen Agenten, eine Agentenmenge oder den besonderen Empfänger "BROADCAST" handeln. Dieser letztgenannte Empfänger bewirkt eine Anfrage an einen Informationsdienst (siehe Abschnitt 6.9.1) zur Ermittlung sämtlicher aktiver Agenten. Danach wird die zu sendende Nachricht einzeln an jeden Empfänger geschickt.

Das Empfangen verläuft ähnlich wie das Senden in mehreren Schritten: Ein Thread des Kommunikationsmoduls pollt eintreffende Nachrichten und legt diese nach einer Objektkonvertierung in einer FIFO-Queue ab[11]. Der Timeout wird geprüft und im positiven Fall die Nachricht an den Interpreter weitergeleitet. Handelt es sich um eine neue Agentenadresse, so wird zusätzlich diese in der lokalen Daten-

11. War die Nachricht nicht konvertierbar, so wird sie nicht weiter beachtet.

basis vermerkt. Der eigentliche Datentransport wird entweder mittels entfernter Methodenaufrufe über CORBA oder über Socket-Verbindungen abgewickelt.

6.9.1 Agentendämon

Damit Agenten Ansprechpartner im Netzwerk finden und adressieren können wird ein entsprechender Informationsdienst benötigt. Zu diesem Zweck dient der Agenten-Dämon. Er ist ein Hintergrundprozeß, der Informationen über zur Zeit laufende Agenten bzw. Resourcen[12] verwaltet. Dazu zählen die Kommunikationsplattformen, die ein Agent oder eine Ressource unterstützt und dessen Adresse. Jeder Agent und jede Ressource muß sich beim Start bei diesem Dämon registrieren lassen und vor dem Beenden wieder abmelden.

Der Agentendämon stellt als Informationsbroker einen Flaschenhals in der Kommunikation dar. Dieser Problematik wird durch einen Cachingmechanismus Rechnung getragen, der in jeden Agenten integriert ist. Nach erstmaliger namentlicher Adressierung eines anderen Agenten durch einen Sprechakt wird dessen Adresse, die durch den Dämon aufgelöst wird, lokal abgespeichert. Dasselbe geschieht auch bei empfangenen Sprechakten, sofern sie bearbeitet werden. Das Caching ist insbesondere bei der Abarbeitung von Kooperationsprotokollen, in deren Ablauf mehrfach Sprechakte mit den Kooperationspartnern ausgetauscht werden, von Vorteil. Hier spart die direkte Kommunikation Zeit. Ist ein direkt adressierter Kommunikationspartner nicht empfangsbereit (terminiert), wird das Kommunikationsproblem von der Kommunikationskomponente behandelt und der Caching-Eintrag gelöscht.

6.10 Zusammenfassung und Bewertung

Als Ausführungsumgebung für den von der Agentenprogrammiersprache generierten Code wurde eine *komponentenbasierte Agentenarchitektur* entworfen und implementiert. Die Basisaktivitäten von Agenten finden sich in den groben Verarbeitungsebenen Kommunikation, Management und Intentionalität wieder. Aufgabe des *Kommunikationsmoduls* ist es die unteren Schichten der Nachrichtenübermittlung zu realisieren und Basis-Filterungsmechanismen bereitzustellen. Die eigentliche Informationsverarbeitung wird durch die zentrale Komponente des Managementmoduls, den *Interpreter*, gesteuert. In ihm laufen die wichtigsten Informationen zusammen und werden an die betreffenden Agentenmodule weitergeleitet. Auf diese Weise ist dem Interpreter Kontrolle über die übrigen Komponenten gegeben. Im *Intentionalitätsmodul* findet das Ableiten von Maßnahmen aufgrund neuer In-

12. Ressourcen sind beliebige nicht-agentische Entitäten, die über Actions und Sensings an REkoS-Agenten angebunden sind.

formationen statt. Eine Inferenzmaschine führt, getriggert vom Interpreter, zyklisch Updates des mentalen Zustands durch und veranlaßt Auswahl, Start und Stopp von Handlungsskripten und Interaktionsprotokollen. Deren Ausführung wird von dedizierten Managementkomponenten kontrolliert. *Skript-* und *Kooperationsmanager* überwachen die Ausführungsthreads der ihnen übergebenen Skripte. Ein vorgeschalteter *Scheduler* managed die Freigabe ausgewählter Handlungen anhand ihrer Prioritäten und kann die Unterbrechung in Ausführung befindlicher Skripte veranlassen, wenn höchstpriorisierte Aktionen anstehen.

Der Anforderung, eine dienstunterstützende Agentenplattform zu sein wird die Architektur durch die umfangreichen Managementfunktionalitäten und die Konzentration auf effiziente Handlungsausführung weitgehend gerecht. Im einzelnen werden die eingangs aufgestellten Anforderungen wie folgt abgedeckt: Aktivitätenmanagement wird vorausschauend durch Scheduling und im nachhinein durch Unterbrechungs- und Abbruchmmechanismen aktiver Skripte aufgrund äußerer Einflüsse geregelt. Überwachung findet durch laufende Kontrolle der verbrauchten Zeit und Kosten - bei Taskaktivierung, Skriptauswahl und Empfang und Versenden von Nachrichten - statt; Accounting und Qualitätsmanagement werden durch Erhebung und Auswertung von Leistungsdaten im Interpreter und Intentionalitätsmodul realisiert.

Die Aspekte Präferenzen bei der Dienstnutzung und Transparenz der Diensterbringung werden nur peripher berücksichtigt. In bezug auf den ersten Punkt stellt die Agentenprogrammiersprache zwar ein Konzept zur Spezifikation von Dienstparametern bereit, jedoch sind weder Dienstbeschreibungssprache noch Architektur in der Lage Abhängigkeiten ausdrücken bzw. verarbeiten zu können. Für den zweiten Aspekt bietet die Agentenarchitektur mit den Reason-Pointern in Verbindung mit persistenter Datenhaltung die grundlegenden Konzepte zur Erstellung von Ablaufbeschreibungen. Allerdings wird keine direkte Unterstützung angeboten, so daß die Generierung von Erklärungen zur Diensterbringung entweder manuell in die Handlungsskripte oder automatisiert in das Intentionalitätsmodul zu integrieren ist.

Die Anforderungen an die Informationsverarbeitung werden durch die Architektur durch Unterstützung synchroner und asynchroner Kommunikation, parallele Verarbeitung und einfache Lern- und Persistenzfunktionalität abgedeckt. Trotz des performancebedingten Verzichts auf eine interpretierte Abarbeitung sind weitreichende Eingriffsmöglichkeiten in die Skriptabarbeitung gegeben. Die massive Nutzung von Threads zur quasi-parallelen Verarbeitung sowohl innerhalb der Architekturkomponenten als auch bei den Handlungsskripten stellt jedoch einen ressourcenhungrigen Ansatz dar - in der Praxis zeigte sich, daß eine kleinere SPARC-Workstation mit 4 Agenten bereits ausgelastet ist.

Das Konzept der Reason-Zeiger hat sich für die Nachvollziehbarkeit des Handelns bei Post-Mortem-Analysen und für das Debugging der Architektur als wert-

voll erwiesen. Auf der anderen Seite gestaltet sich durch die Notwendigkeit der Begründungslisten eine Formulierung von initialen mentalen Zuständen als schwierig. Einem Agenten können nicht einfach bei dessen Start einige Motivationen mitgegeben werden, ohne die entsprechenden Variablen zu berücksichtigen.

Positiv ist auch die Entscheidung einzuschätzen, Prolog als Repräsentations- und Verarbeitungssprache im Intentionalitätsmodul zu verwenden. Eine wissensbasierte Anwendung aus dem Bereich der Intelligenten Netze[13] wurde - neben der Implementierung einiger komplexer interaktionsprotokolle - nahezu vollständig in Prolog realisiert. Dies führte zu einer erheblichen Einsparung an Entwicklungsaufwand im Verhältnis zur Skript / C++-Programmierung.

Wissensmanagement spielt im Rahmen der Diensterbringung keine wichtige Rolle, daher wurden auch keine Belief-Revision-Techniken in die Architektur integriert. Jedoch gestattet es die Nutzung von Prolog im Intentionalitätsmodul, einfache anwendungsabhängige Belief-Revision-Politiken zu definieren: Neues Wissen gelangt über KQML-Sprechakte in den Agenten. Was mit dem neuen Wissen geschieht, wird in der Empfangsprozedur des Sprechakts festgelegt. Dort kann neben dem Nachrichteninhalt in Form eines Fakts auch auf den Absender und einen Zeitstempel zugegriffen werden. Diese Informationen können beispielsweise herangezogen werden, um zu entscheiden, ob und wann ein neues Fakt altes Wissen überschreiben darf.

Die REkoS-Agentenarchitektur stellt eine prozedurale Schnittstelle zur Programmierung von Fähigkeiten und eine deklarative Sprache für die Definition des mentalen Modells und der Sprechaktfähigkeiten bereit. Informationsspeicherung und -verarbeitung ähnelt den reaktiven BDI-Architekturen. Zwar ist die Realisierung der Handlungsskripte noch mühevolle C++-Programmierarbeit, dafür bietet die Verwendung der Architektur viele Vorteile, die zu einer erheblichen Aufwandsreduzierung bei der Realisierung flexibler und verteilter Systeme führen: Netztransparenz, einheitliche Kommunikationsschnittstelle, parallele Verarbeitung, Aktivitätenmanagement und Persistenzmechanismen.

13. De Agenten hatten große Tabellen mit Dienstinformationen unter Beachtung von Constraints zu verwalten. Derartige Anwendungen, wie auch Suchprobleme im allgemeinen lassen sich sehr viel einfacher in einer logikbasierten Programmiersprache darstellen und lösen als in prozeduralen Sprachen.

KAPITEL 7 Debugging

„Concurrency, problem-domain uncertainty, and non-determinism in execution together conspire to make it very difficult to comprehend the activity in a distributed intelligent system [...] we urgently need graphic displays of system activity linked to intelligent model-based tools which help a developer reason about expected and observed behaviour.“[1]

Debugging und Testen von Agentensystemen ist eine schwierige Aufgabe, es gilt kausale Zusammenhänge komplexer, nichtdeterministischer und verteilter Systems sichtbar zu machen. Fehlende Benchmarks und schwer formalisierbare Korrektheits- und Koordiniertheitsbegriffe erschweren eine systematische Herangehensweise, so daß man sich zumeist auf empirische Beobachtungen beschränkt.

Das REkoS-Testbett wurde entwickelt, um Beobachtungen und Eingriffe in laufende Agentensysteme auf mehreren Ebenen zu gestatten. Zum einen lassen sich Agenten einzeln testen, hierbei wird die Umgebung vollständig von den Testwerkzeugen kontrolliert. Zum andern erlauben Systemmonitore die Kontrolle von Mehragentenszenarien. Schließlich ist es möglich, eine umfangreiche post mortem-Analyse durchzuführen.

1. [Gasser, et al. 1987].

Im ersten Abschnitt werden das Debuggingkonzept und die Architektur des Testbetts vorgestellt. Der zweite Abschnitt beschreibt Eingriffsmöglichkeiten, die direkt über die graphische Oberfläche eines Agenten vorgenommen werden können. Anschließend wird der zentrale Prozeß des Debuggers, die Testbettapplikation, beschrieben. Sie stellt das Bindeglied zwischen den nachfolgend behandelten Werkzeugen und den kontrollierten Agenten dar. In den Abschnitten 4 und 5 werden agentenlokale Debugging-Tools vorgestellt: Es handelt sich um einen Monitor, der alle Aspekte des mentalen Zustands abdeckt und einen Stepper, mit dem die Ausführung von Skripten kontrollierbar ist. Abschnitt 6 stellt mit dem Kommunikationsmonitor ein Werkzeug zur Beobachtung und Einflußnahme der zwischen den Agenten stattfindenden Interaktionen vor. Die Laufzeitprotokollgenerierung und darauf aufbauende Analysefunktionen werden im 7. Abschnitt ausführlich erörtert, bevor in Abschnitt 8 die übrigen realisierten Debuggingfunktionalitäten kurz angesprochen werden. Zum Abschluß erfolgt eine Zusammenfassung und Diskussion der realisierten Konzepte.

7.1 Konzept für das Debugging verteilter Systeme

Zum Funktionieren eines verteilten Systems ist zunächst einmal die Korrektheit der einzelnen Glieder sicherzustellen, bevor das koordinierte Zusammenspiel der Systemelemente geprüft werden kann. Diese zwei Aspekte finden sich auch im Debugging-Konzept von REkoS wieder: Es existieren Werkzeuge sowohl zum Testen einzelner Agenten als auch des globalen Systemverhaltens.

7.1.1 Debugging einzelner Agenten

Agentenspezifische Debuggingwerkzeuge erlauben die Beobachtung und Kontrolle des Verhaltens *eines* Agenten. Debugging findet dabei auf der Ebene der Agentenprogrammiersprache statt, Aufsetzpunkte sind dementsprechend die Wahrnehmung, Skriptausführung und der mentale Zustand.

- Wahrnehmung: Eingehende Nachrichten müssen korrekt interpretiert werden. Hierunter fällt Wissensmanagement, Zielgenerierung sowie Taskaktivierung.
- Mentaler Zustand: Die Entwicklung des Zustands der mentalen Attitüden und deren Auswirkung auf Taskaktivierung und Skriptauswahl muß über die Zeit verfolgbar sein.
- Handlungsskripte: Die Abarbeitung der Skripte soll wie bei einem Quellcodedebugger kontrollierbar sein. Dazu zählt das Ansehen und Verändern von Variablenwerten, das Setzen von Breakpoints und der kontrollierte Abbruch.

- Verhalten unter Belastung: Lastsituationen sollten simulierbar sein, damit das Verhalten eines Agenten bei auftretenden Timeouts oder mangelnder Kommunikationsbandbreite getestet werden kann.

Der Agent kann isoliert und auch im Verbund mit anderen Agenten getestet werden. Isoliertes Testen ist sinnvoll, um Umgebungseinflüsse auszuschalten, damit der Agent als deterministisches System funktioniert. Beim Einsatz des Agenten in einem Mehragentenszenario kann hingegen dessen Verhalten innerhalb einer realen Anwendung beobachtet werden.

7.1.2 Debugging von Agentensystemen

Beim Testen des gesamten Systems wird dessen Verhalten anhand der stattfindenden Interaktionen beobachtet. Hier ist einerseits die Korrektheit und Effektivität der Kooperationsprotokolle zu prüfen und andererseits die Verteilung von Ressour-cen wie Zeit, Auslastung einzelner Agenten oder Kosten. Während das Testen der Kooperationsprotokolle auf Korrektheit systematisch erfolgen kann, fehlt für das Tuning eines Systems aufgrund der vielen Freiheitsgrade und Interdependenzen eine geeignete Methodik. Deshalb ist es notwendig, Mechanismen bereitzustellen, um die unterschiedlichen Aspekte des koordinierten Verhaltens einzeln und im Zusammenhang untersuchen und bewerten zu können.

Globales Debugging beschränkt sich im wesentlichen auf reines Beobachten, da ein Eingreifen in dynamische, unüberschaubare, nichtdeterministische Systeme nur schwerlich zielgerichtet durchgeführt werden kann. Aufgrund der Systemdynamik ist es ferner wichtig Mechanismen zur Informationsverdichtung und -filterung zur Verfügung zu haben, um eine Fokussierung auf die relevanten Aspekte zu ermöglichen.

7.1.3 Architektur des Debuggers

Die Debuggingwerkzeuge wurden als kommunikationsfähige Prozesse gestaltet, um sie räumlich von den assoziierten Agenten zu entkoppeln. Jedes Werkzeug besitzt eine graphische Benutzungsoberfläche, über die der Agent kontrolliert wird und in der der Agent seine Statusinformationen anzeigt. Für die Zwecke des Debuggings wurde die Agentenarchitektur um eine Testbettkomponente erweitert. Sie verwaltet eine Menge von Flags, mit denen einzelne Debuggingfunktionen ein- und ausschaltbar sind und interpretiert eingehende Testbettnachrichten.

Abbildung 7-1 zeigt den Kommunikationsfluß zwischen Debuggingwerkzeugen und Agenten. Aktionen in der Bedienoberfläche lösen Callback-Prozeduren aus. Diese generieren Steuerungsnachrichten, welche über ein Kommunikationsmodul an den / die Agenten gesendet werden[2]. Empfängt ein Agent eine Testbettnachricht,

wird diese über den Interpreter an die Testbettkomponente weitergeleitet, wo ihre Umsetzung stattfindet. In die Agentenkomponenten wurde Debugcode integriert, welcher ihm zugeordnete Flags abfragt und im positiven Falle die dazugehörigen Testfunktionen ausführt. Die Testbettfunktionen eines Agenten umfassen das Übermitteln von Zuständen, Zustandsveränderungen sowie verhaltensbeeinflussende Eingriffe. Üblicherweise beinhaltet die Realisierung einer Debuggingfunktionalität das Senden von Zustandsinformationen des Agenten an das Debuggingwerkzeug. Diese Informationen werden von der Testbettkomponente zu Statusmeldungen transformiert und dem Testbett zugestellt. Dort werden die Informationen in einem Konvertierungsschritt ausgepackt und über eine Callback-Funktion zur Anzeige in einem dafür vorgesehenen Fensterelement des entsprechenden Werkzeugs gebracht.

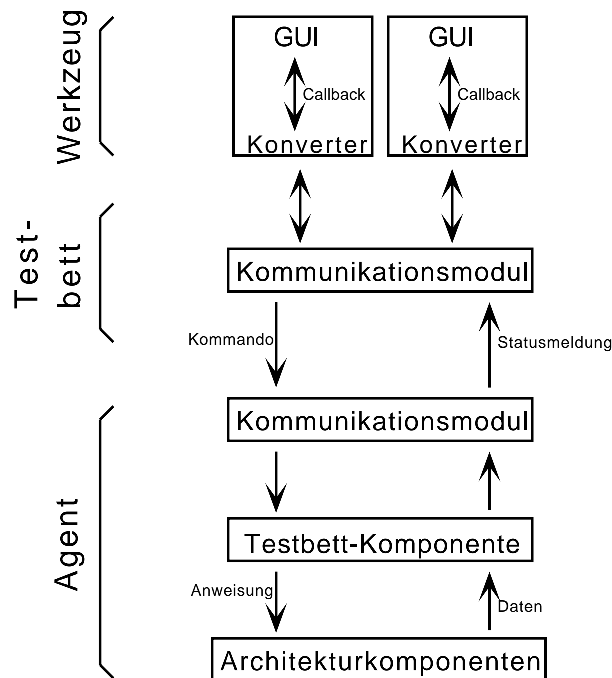

Abbildung 7-1: Kommunikation zwischen Testbett und Agenten

2. Damit Agenten kontrollierbar sind, werden Nachrichten des Testbetts immer vorn in die Nachrichtenpuffer der Kommunikationsmodule geschrieben. Da Sprechakte dies nicht gestatten, wurde ein eigenes Nachrichtenformat für den Austausch zwischen Debuggingwerkzeugen und Agenten konstruiert.

7.2 Agentendebugging

Jeder Agent besitzt eine Bedienoberfläche (GUI[3]), deren Menüleiste die Einträge Main und Debug enthält (Abbildung 7-2). Im Debug-Pulldownmenü kann die Ausgabe von Laufzeitmeldungen, insbesondere von Ausnahmebedingungen, die in die Architekturkomponenten integriert sind, gesteuert werden. Diese werden auf dem Terminal ausgegeben, von dem der Agent gestartet wurde.

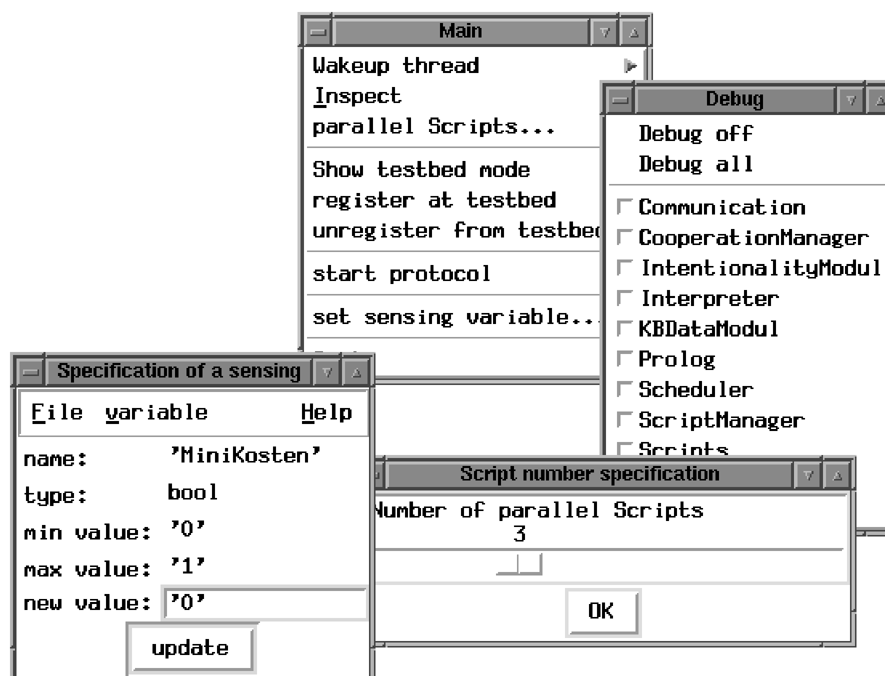

Abbildung 7-2: Bedienelemente zur Kontrolle eines Agenten

Über das Main-Pulldownmenü werden die übrigen Debugging-Funktionalitäten agentenseitig getriggert. Hier kann ein Schnappschuß initiiert werden, der sämtliche Zustandsvariablen, Objektinstanzen und Wissenszustände ausgibt. Auch kann die Zahl der maximal parallel ausgeführten Handlungsskripte als Maßnahme zur Verminderung des Thread-Verwaltungsaufwands auf ressourcenarmen Rechnern festgelegt werden. Desweiteren ist es möglich, den Agenten manuell beim Testbett zu registrieren und abzumelden. Mit der Registrierung wird die Testbettkomponente des Agenten initialisiert, woraufhin der Agent vom Testbett aus ansteuerbar ist. Bei

3. Graphical User Interface.

einer Abmeldung läuft ohne Kontrolle des Testbetts bis zu einer erneuten Anmeldung bzw. bis zur Terminierung (durch Auswahl von quit im Main-Menü) weiter.

Ebenfalls über das GUI einstellbar ist die Laufzeitprotokollerstellung, die in Abschnitt 7.7 beschrieben wird. Schließlich ist es möglich Einfluß auf die normativen Ziele mittels einer Variablenspezifikation zu nehmen (Abbildung 7-2, unten links). Diese direkte Manipulationsmöglichkeit vereinfacht das Hantieren mit den eigentlich nur aufgrund empfangener Sensings beeinflußbaren Variablen[4].

7.3 Die Testbettapplikation

Nach dem Starten des Testbetts erscheint das in Abb. 7-3 gezeigte Hauptfenster der Applikation. Es dient als Einstiegspunkt für die Aktivierung der gewünschten Testbett-Tools. Im linken Listenelement sind unter dem Titel available agents alle beim Testbett registrierten Agenten aufgelistet[5]. Agenten, die erst nach dem Start des Testbetts kreiert wurden, können durch Betätigung der update-Taste nachträglich in die Liste aufgenommen werden; hierbei wird eine BROADCAST-Anfrage gesendet mit der Bitte, sich beim Testbett zu registrieren. Terminiert ein Agent, so teilt er dies dem Testbett mit. Dies hat zur Folge, daß sein Name aus der Liste entfernt wird. Außerdem beendet das Testbett alle agentenspezifischen Tools dieses Agenten.

Bei den unter dem Menüpunkt Agent-specific tools anwählbaren Debuggingwerkzeugen handelt es sich um Tools zum Testen einzelner Agenten. Die Aktivierung eines solchen agentenspezifischen Tools bezieht sich immer auf die in der rechten Liste aufgeführten Agenten, welche durch Mausselektion aus der linken Liste übernommen und auf dem umgekehrten Weg wieder entfernt werden. Unter dem Menüpunkt Systemwide tools sind die Testbettwerkzeuge zur Kontrolle des gesamten Szenarios aufgelistet.

Mit dem Start eines Tools werden die angesprochenen Agenten, bzw. im Falle von systemweiten Tools alle registrierten Agenten über den Start des Werkzeugs informiert. Der Agent behandelt diese Initialisierungsnachricht, indem er die geforderten Zustandsinformationen an die Testbettoberfläche zurücksendet. Gleichzeitig startet das Testbett einen Unterprozeß, der die Funktionalität und Bedienoberfläche des Werkzeugs beherbergt. Im weiteren Verlauf signalisiert der kontrollierte Agent jede betroffene Zustandsänderung an das Testbett, welches die Nachrichten an das betroffene Tool weiterleitet.

4. Siehe Abschnitt 5.4.1. Tatsächlich erfolgt die Realisierung dadurch, daß dem Agenten ein Sensing mit dem Variablenwert gesendet wird.

5. Agenten mit deaktiviertem Debug-Modus (siehe Abschnitt 7.2) treten in der Liste nicht auf.

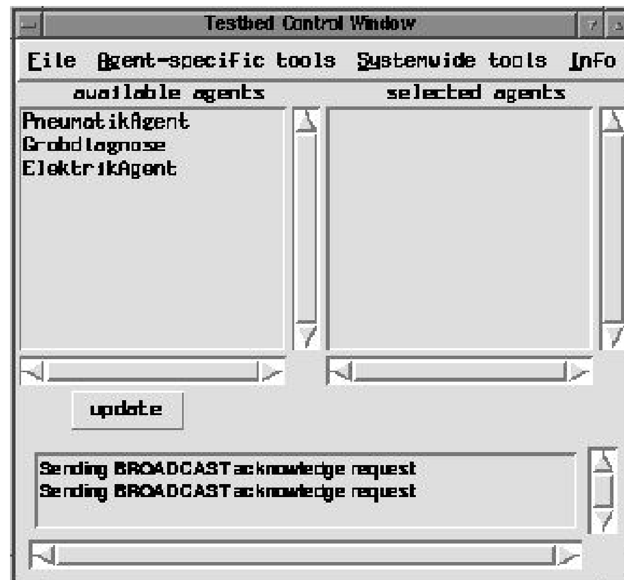

Abbildung 7-3: Einstiegsmenü des Testbetts

7.4 Überwachung des mentalen Zustands

Zwei Werkzeuge, der Goal-Monitor und der Task-Monitor, dienen zur Beobachtung und Kontrolle des mentalen Zustand eines Agenten. Ihre Oberflächen beinhalten Fenster zur Darstellung des aktuellen Zustands und der historischen Entwicklung der einzelnen mentalen Attitüden: Variablen, Motivationen, Ziele und Tasks können manuell gesetzt bzw. aktiviert und deaktiviert werden.

Das in Abschnitt 5.4 vorgestellte mentale Modell beruht auf Aktivierungshierarchien von Variablen, Motivationen, Zielen und Tasks. Jede aktive Task hat als Begründung mindestens ein aktives Ziel, welches durch Motivationen unterstützt wird, die wiederum auf Variablenzuständen beruhen. Ein willkürliches Starten einer Task ohne Berücksichtigung der Abhängigkeiten würde eine Inkonsistenz im mentalen Zustand zur Folge haben. Um derartige direkte Eingriffe seitens des Testbetts dennoch zu ermöglichen werden für die von Hand gesetzten Behaviours im Agenten Dummy-Attitüden als Begründungen generiert. So bleibt das mentale Modell konsistent.

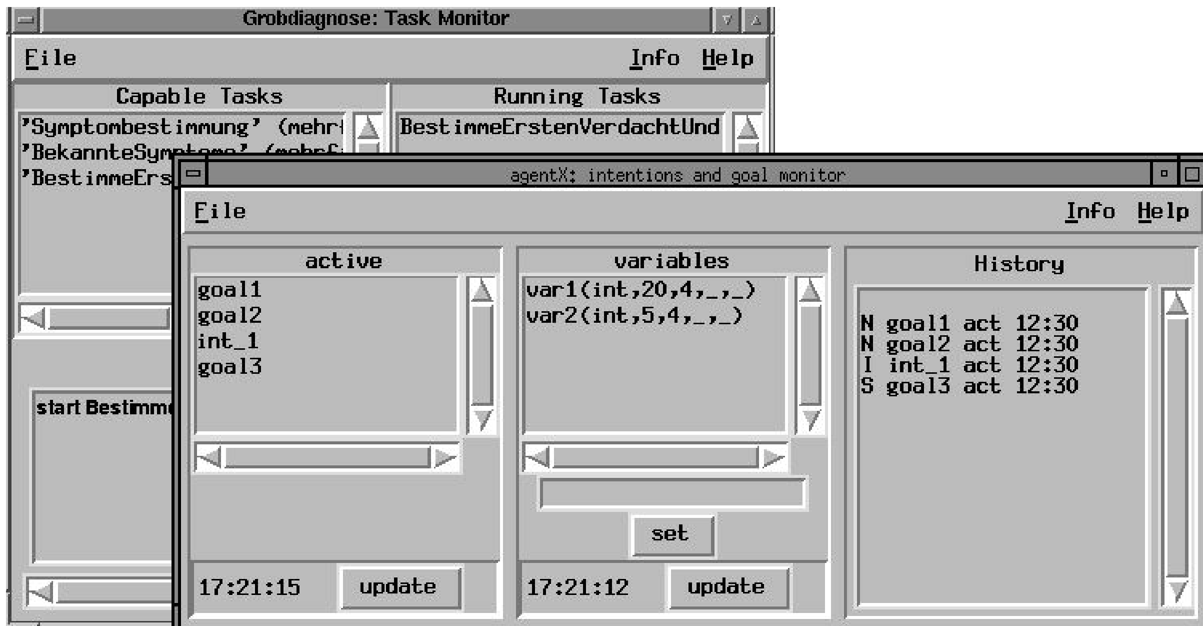

Abbildung 7-4: Testbettwerkzeuge zur Kontrolle des mentalen Zustands

7.5 Debugging von Skripten

Der Stepper ist das Debuggingwerkzeug für die Kontrolle der in Ausführung befindlichen Skripte eines Agenten. Er realisiert ein Quellcode-Debugging auf der Ebene der Skriptbeschreibungssprache. Nach dem Start des Tools protokolliert der Agent den Durchlauf der Zustände bei der Ausführung von Skripten im Trace Window (oben in Abbildung 7-5). Angezeigt werden Start und Ende von Skripten (durch vorangestellte doppelte Pfeile) und Steps (kurze Pfeile). Nach Abschluß eines Skripts werden zusätzlich die generischen Parameter Zeit und Kosten ausgegeben. Die Einflußnahme seitens des Bedieners erfolgt über die Tastenleiste am unteren Rand des Fensters.

Der Stepper kann in zwei Modi benutzt werden: Step und Run[6]. Ist der Step-Modus aktiviert, hat der Bediener Kontrolle über die Ausführung der Skriptelemente. Der Agent wartet vor der Ausführung eines jeden Steps und abschließend vor Beendigung des Skripts auf auf das manuelle Weiterschalten des Benutzers, welches mittels der Step-Taste geschieht. Im Step-Modus hat der Benutzer die Möglichkeit, Skriptvariablen anzuschauen und zu verändern. Weiterhin ist es möglich, sich tex-

6. Der aktive Modus wird oben rechts direkt unter dem Help-Menüeintrag angezeigt.

tuelle Informationen über das Skript anzeigen zu lassen. Hierbei wird der Kommentartext, wie er im Skriptdefinitionswerkzeug (Abschnitt 5.5.3) zu Dokumentationszwecken eingegeben wurde, angezeigt (unten rechts in Abbildung 7-5). Die Ausführung des aktuellen Skripts kann auch beendet werden, was einem Abbruch ohne Resultat gleichkommt.

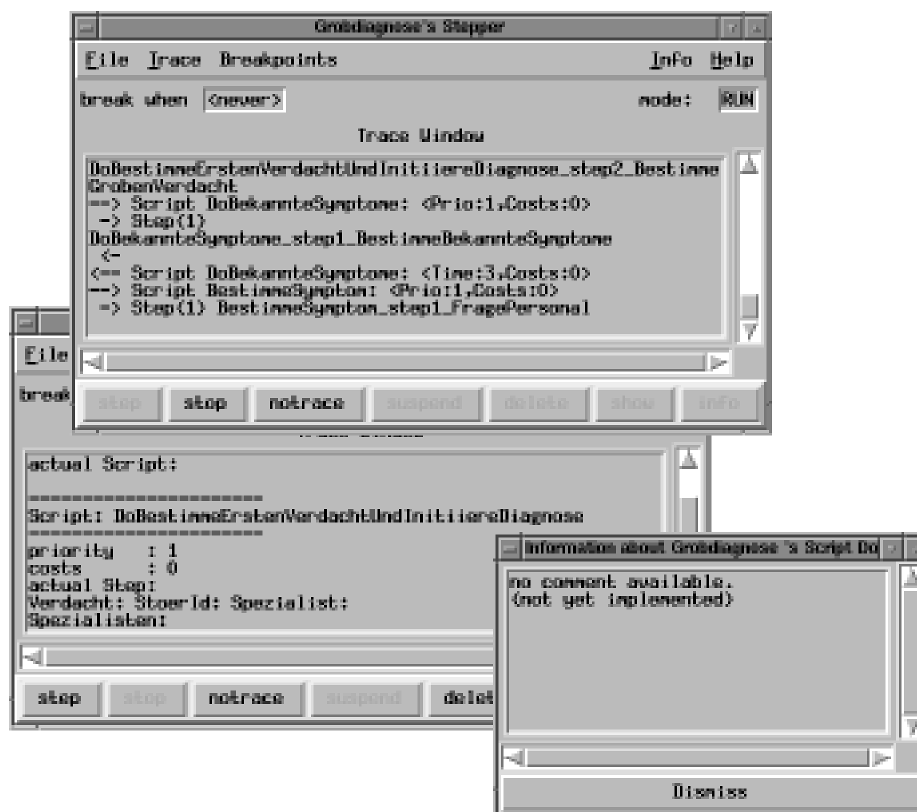

Abbildung 7-5: Skript-Debugger

Ein Wechsel in den notrace-Modus bewirkt, daß der Agent ohne zu warten mit der Ausführung fortfährt. Zurück in die schrittweise Abarbeitung gelangt man entweder durch Betätigen der Stop-Taste oder wenn ein Breakpoint erreicht wird. Als Breakpoints können Namen von Skripten und Skriptelementen oder beliebige Prolog-Terme angegeben werden. Letztere werden auf Erfüllbarkeit in der Wissensbasis getestet. Auf diese Weise sind Skripte auch aufgrund äußerer Einflüsse unterbrechbar.

Aus pragmatischen Gründen sind Eingriffsmöglichkeiten in den Ablauf von Skripten nur in den generischen, von der Agentenarchitektur bereitgestellten Klassen möglich. Weil Methoden für jeden Agenten spezifisch programmiert werden, ist hier eine automatische Verankerung von Testbett-Programmcode schwierig wenn

nicht gar unmöglich. Die Testbettfunktionalität wurde daher in die generische eval-Methode der Skriptklasse (siehe Abbildung 6-5 auf Seite 176) eingebaut. Außerdem wurde die Skriptklasse um ein Zustandsvariablenpaar erweitert, welches den Modus (Step oder Run) und einen Schlüssel für eine erfolgte Benutzerinteraktion abbildet.

```
1 void Script::eval() {
2   // bool mode_is_step = false;
3   Step *step;
4   while(go&&(step = getActualStep())) {
5                // testbed code start
6       if (! mode_is_step) {
7           mode_is_step = check_breakpoints();
8           cont = false;
9       }
10      if (mode_is_step) while (cond)
11          switch (userCmd) {
12          case INFO: sendInfo( step );break;
13          case EXIT: go = false; cont = false;
14              inform_scheduler( ABORT ); break;
15          case GET_VAR:
16              (void*)FunPtr =
17                  Getters->getHash(VarName);
18              sendInfo( (*FunPtr) );
19              break;
20          case SET_VAR:
21              (void*)FunPtr =
22                  Setters->getHash(VarName);
23              (*FunPtr)(Value);
24              break;
25          case STEP: cont = false; break;
26          }
27              // testbed code end
28      step->eval(this);
29  }
30  myThread.exit(0);
31 }
```

Abbildung 7-6: Script-Klasse mit integriertem Testbett-Code

Im step-Modus fragt der Testbett-Programmcode (Abbildung 7-6) in einer Schleife (Zeile 10 - 26) die Zustandsvariable (userCmd) nach einer getätigten Aktion im Skript-Stepper ab und führt diese im positiven Falle aus.

- Das Ansehen und Ändern von Skriptvariablen (Zeile 15 - 24) erfolgt über die in der konkreten (abgeleiteten) Anwendugsskriptklasse vorhandenen Getter- und Setter-Methoden (siehe Abbildung 5-8 auf Seite 149). Der konkrete Zugriff erfolgt über eine vom Definitionswerkzeug generierte und gefüllte Hash-Tabelle, in der auf die Zugriffsmethoden anhand der Variablennamen zugegriffen werden kann.
- Auch der Dokumentationstext von Skripten, Steps und Methoden wird in generischen Instanzvariablen gehalten, so daß ein direkter Zugriff nach Dereferenzierung der Anwendungsklasse möglich ist (Zeile 12).
- Der Aufforderung, den nächsten Schritt durchzuführen wird einfach durch Verlassen der Schleife des Testbett-Programmcodes nachgekommen (Zeile 25). Die eval-Methode der Skriptklasse läßt den nächsten Step ausführen, bevor im folgenden Durchlauf der Skriptschleife die Kontrolle wieder an die Testbett-Verarbeitungsprozedur gelangt. Dadurch wird die weitere Abarbeitung erneut unterbrochen.
- Der Abbruch eines Skripts wird einfach durch eine beende Skript-Nachricht (siehe Tabelle 8 auf Seite 173) an den Scheduler durchgesetzt, woraufhin es aus der Liste des Skriptmanagers entfernt wird. Außerdem wird das go-Flag der Skriptklasse auf false gesetzt, womit die Abarbeitungsschleife der eval-Methode abgebrochen wird (Zeile 13 - 14).

Im Run-Modus fragt das Debugging-Programmfragment die Zustandsvariable ab und verzweigt zum Step-Modus, falls ihr Wert entsprechend gesetzt ist. Andernfalls werden die Breakpoint-Bedingungen, sofern vorhanden, getestet (Zeile 6 - 9). Handelt es sich bei einem Breakpoint um den Namen eines Skriptelements (beispielsweise eine Methode), findet ein Vergleich mit dem aktuellen Stepnamen statt[7]. Im Falle eines Prolog-Terms wird eine entsprechende Anfrage über das Intentionalitätsmodul an die Prolog-Datenbasis gestellt. War die Anfrage erfüllbar, gilt der Breakpoint als erreicht.

7.6 Kommunikationsmonitor

Der Kommunikationsmonitor (siehe Abbildung 7-7) protokolliert die gesamte Kommunikation zwischen Agenten, soweit sie über Sprechakte, Sensings, Actions und Interaktionsprotokolle abläuft. Im oberen Fenster dieses Tools werden Nachrichten durch Angabe von Sender, Empfänger und Nachrichteninhalt dargestellt. Dabei wird das Senden einer Nachricht durch <Sender> to <Empfänger>, der Empfang durch <Sender> from <Empfänger> ausgedrückt. Ein Zeitstempel am Ende der

7. Der Name eines Steps wird als Membervariable in der Stepklasse verwaltet. Auf ihn kann durch Dereferenzierung der actualStep-Variable zugegriffen werden.

Zeile informiert über das Datum der versendeten (out at ...) bzw. empfangenen (in at ...) Botschaft. Handelt es sich um einen Sprechakt, der Bestandteil eines Interaktionsprotokolls ist, so wird zusätzlich die Protokoll-ID angegeben.

Abbildung 7-7: Kommunikationsmonitor

Durch eine Reihe von Filterfunktionen ist das Monitoring konfigurierbar. So kann die Anzeige bestimmter Informationstypen (Interaktionsprotokoll-Sprechakte, gewöhnliche Sprechakte, Actions oder Sensings) unterdrückt, die Darstellung auf empfangene oder verschickte Nachrichten reduziert und schließlich ein Fokussieren auf bestimmte Agenten erzielt werden. Auf diese Weise lassen sich die präsentierten Informationen auf bestimmte Aspekte hin reduzieren.

Neben der Monitorfunktionalität erlaubt dieses Tool auch ein aktives Eingreifen in das Systemgeschehen durch das Versenden von Sprechakten. Im unteren Bereich des Werkzeugfensters lassen sich Sprechakte spezifizieren und versenden. Die De-

finition eines Sprechakts erfolgt durch Auswahl von Performative, Operator und Content-Typ sowie Eingabe des konkreten Nachrichteninhalts. Grundlage ist das vom Agenten bei dessen Start geladene, mit dem Sprechaktdefinitionstool spezifizierte Modell. Weil das Testbett kein Agent im eigentlichen Sinne ist, können nur Sprechakte ohne erwartete Rückantwort (noReply) verschickt werden. Als Adressaten sind alle beim Testbett registrierten Agenten und das spezielle BROADCAST möglich.

7.7 Laufzeitprotokolle

Jeder REkoS-Agent ist in der Lage, ein Laufzeitprotokoll seiner Aktivitäten zu generieren. Die Steuerung der Protokollgenerierung erfolgt entweder agentenseitig über das Debug-Menü eines Agenten (siehe Abschnitt 7.2) oder global durch ein Testbettwerkzeug. Die erfaßten Laufzeitdaten werden mit einem Zeitstempel versehen für jeden Agenten in eine ausgezeichnete Datei geschrieben. Es möglich, die Datensammlung auf bestimmte Agentenkomponenten einzuschränken[8]:

- Intentionalitätsmodul: Die Entwicklung aller mentaler Attitüden wird protokolliert. Betroffen sind Variablen, wenn sie eine Wertveränderung erfahren sowie Motivationen, Ziele, Tasks und normative Ziele bei Zustandswechsel, Verstärkung und Abschwächung. Vermerkt wird dazu die jeweilige Ursache, die von der Architektur durch den Reason-Pointers repräsentiert wird. Außerdem werden alle Skriptaktivierungen gespeichert.
- Wissensbasis: Die Protokollierung des Wissens umfaßt alle - typischerweise über tell-Sprechakte - empfangenen Informationen, lokal gespeicherte Ergebnisse und das gesammelte Erfahrungswissen (siehe Abschnitt 6.5.1).
- Kommunikation: Empfangene und versendete Nachrichten werden ebenso vermerkt wie Kommunikationsfehler. Letztere umfassen Timeouts, unbekannte Nachrichtentypen und nicht interpretierbare Nachrichteninhalte.
- Skriptausführung: Für jedes Skript und jeden Step werden die Parameter, Start, Ende, Abbruch und eventuelle Unterbrechungen notiert. Auch über den vorgeschalteten Scheduler werden Daten erhoben. Hier ist das Management der Warteschlange durch zurückgehaltene, unterbrochene und austretende Skripte und die Entwicklung der Priorisierungen von Interesse.

8. Mit dem Umfang der erfaßten Protokolldaten nimmt die Laufzeitperformance eines Agenten ab. Daher ist das Fokussieren auf bestimmte Komponenten sinnvoll.

7.7.1 Auswertung von Protokollen

Nach einem protokollierten Testlauf liegen die Laufzeitdaten der einzelnen Agenten in den entsprechenden erstellten Dateien vor. Für eine problemgerechte Aufbereitung dieser Rohdaten wurde eine Applikation zur Auswertung von Laufzeitprotokollen entwickelt. Dieses Evaluierungswerkzeug funktioniert als deduktive Datenbank mit grafischer Benutzungsoberfläche. Es erlaubt die Integration der verschiedenen Protokolldateien und bietet vordefinierte statistische Auswertungen und Informationsaggregierungen bzgl. der Aspekte Kooperation, Kommunikation, Task- und Skriptabarbeitung sowie zeitliche Entwicklung der Motivationen und Ziele (siehe Abbildung 7-8, Statistics-Menü) an. Darüberhinaus können eigene Auswertungen in Form von Prolog-Anfragen vorgenommen werden.

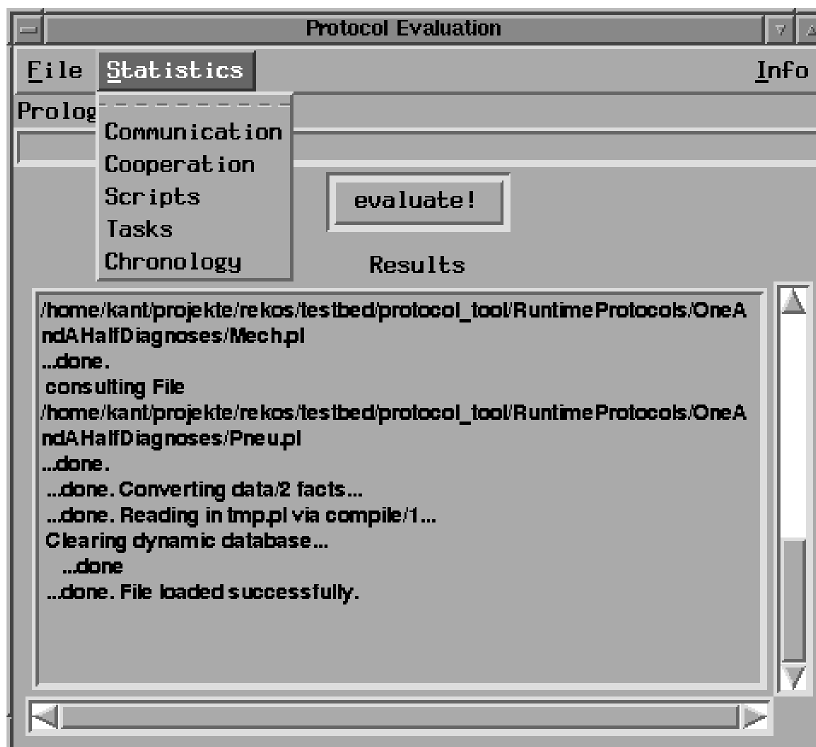

Abbildung 7-8: Werkzeug zur Auswertung von Laufzeitprotokollen

Das Einlesen der Protokolldaten wird mit einem Wissenskompilationsschritt abgeschlossen. Hierbei findet eine Transformation der unstrukturierten Rohdaten in geeignete Prologterme statt[9]. Die Ergebnisse der Datenauswertungen werden in Textfenstern, wie in Abbildung 7-9 dargestellt, angezeigt. Dabei werden agentenübergreifende und agentenspezifische Informationen in separaten Fenstern dargestellt.

- Kommunikation: Ausgewertet werden Informationen zur Zahl und Art der ausgetauschten Sprechakte, Kommunikationspartner, Übermittlungs- und andere Kommunikationsfehler, durchschnittliche Latenzzeiten.
- Kooperation: Interaktionsprotokolle werden nach Dauer, Zahl der Teilnehmer, Zuordnung zu Rollen, Verhältnis von Kommunikation und Handeln analysiert.
- Skripte und Tasks: Aufbereitete Daten zur Handlungsausführung umfassen Auswahlentscheidungen, Häufigkeit der Aktivierung, Dauer, Ursache, akkumulierte Kosten, erfolgreiche und ergebnislos abgebrochene Skripte und Tasks.
- Chronologie: Für jeden Agenten werden alle protokollierten Aktivitäten in der zeitlichen Reihenfolge des Auftretens aufgelistet.

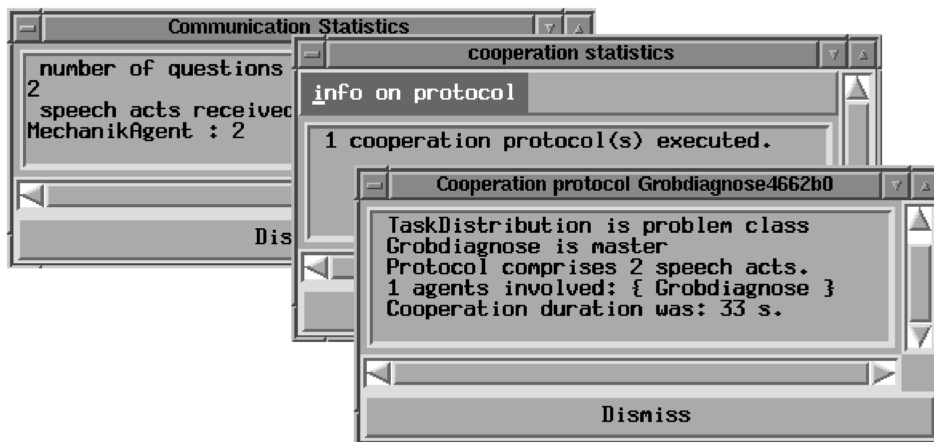

Abbildung 7-9: Präsentation von aggregierten Laufzeitdaten

7.8 Weitere Debugging-Werkzeuge

Mit dem *Speed Control Tool* kann die Ausführungsgeschwindigkeit des Agentensystems herabgesetzt werden. Die Verzögerung wirkt sich nur auf die Abarbeitung von Skripen aus; jeder Step innerhalb der regulären Skriptverarbeitung wird um den eingestellten Wert verzögert. Durch eine verlangsamte Ausführung sind die chronologischen Aktivitäten beim Online-Debugging besser zu verfolgen. Außerdem kann die gedrosselte Geschwindigkeit zu Belastungstests herangezogen werden: Timeouts durch nicht rechtzeitig beendete Dienste treten auf, so daß das Verhalten des Agenten in Fehlersituationen beobachtbar ist.

9. Dabei wird von den Indexierungsmöglichkeiten der Programmiersprache Gebrauch gemacht, um effiziente Zugriffsmöglichkeiten zu ermöglichen.

Ein weiteres systemweites Debugging-Werkzeug ist das *Agent Control Tool*. Mit ihm können Agenten befristet aus einem laufenden Szenario herausgenommen, terminiert oder initialisiert werden. Außerdem wird eine Ping-Funktionalität, angeboten, mit der die Erreichbarkeit bzw. Lebendigkeit beim Testbett registrierter Agenten anfragbar ist. Trifft auf ein Ping-Kommando an einen Agenten keine Antwort ein, kann ein Reset dieses Agenten durch den Initialisierungsbefehl versucht werden. Die zeitweise Nichterreichbarkeit eines Agenten wird durch eine Kontrollnachricht an dessen Kommunikationsmodul erreicht, das sich daraufhin für eine spezifizierte Zeit schlafen legt. Durch derartige Eingriffe sind Kommunikationsfehler simulierbar, wodurch beispielsweise die Robustheit von Interaktionsprotokollen untersucht werden kann.

7.9 Zusammenfassung und Bewertung

Das REkoS-Testbett wurde entwickelt, um ein verteiltes Agentensystem im laufenden Betrieb debuggen zu können. Dazu wurde es als eigenständiger, kommunikationsfähiger Prozeß realisiert, auf dem eine Reihe von Werkzeugen aufsetzt. Testen und Debugging von REkoS-Agenten findet auf allen Ebenen der Agentenprogrammiersprache statt: bei der Handlungsausführung, im mentalen Zustand und den verschiedenen Formen der Interaktionen. Weil sich die Werkzeuge sowohl für den Test einzelner Agenten als auch für Eingriffe in Mehragentenszenarien verwenden lassen, bietet sich konzeptuell eine dreistufige Vorgehensweise an: Zuerst wird die Funktionalität eines Agenten mittels agentenlokalen Debuggings empirisch sichergestellt, bevor er in ein System bereits laufender Agenten integriert werden kann. Hier setzen die Systemmonitore an, um anhand der stattfindenden Interaktionen und erbrachten Dienste das globale Verhalten zu beobachten. Zuletzt können Laufzeitprotokolle erstellt werden, deren Analyse Aufschluß über Fehler, potentielle Engpässe, den Nutzen der Interaktionen etc. bringt.

Der Einsatz der Testwerkzeuge bei der Entwicklung der zwei Anwendungsdemonstratoren ergab einige Erkenntnisse in bezug auf die Verwendbarkeit der Tools. Als sehr wertvoll hat sich der *Skriptstepper* beim Beobachten der Handlungsskripte, insbesondere in bezug mit dem Breakpoint-Konzept, erwiesen - Eingriffe in die Skriptverarbeitung durch Setzen von Variablenwerten erfolgt jedoch relativ unkontrolliert und kann zum Absturz des Agenten führen, da der ausgeführte C++-Code nicht sichtbar ist. Auch der *Kommunikationsmonitor* stellt ein notwendiges Werkzeug zum Beobachten von Agentenszenarien, aber auch zum Testen des „input-output"-Verhaltens einzelner Agenten dar. Hier gilt, daß der aktive Eingriff durch Versenden von Nachrichten zumeist nur beim Test eines einzelnen Agenten sinnvoll ist; gezielte, *kontrollierte* Eingriffe in ein Mehragentensystem sind schwerlich möglich. Belastungstests mit Hilfe der Werkzeuge *Speed Control Tool* und *Agent Control*

Tool sind nützlich, um die Stabilität des System durch Entfernen oder Behindern von Agenten zu testen. Die *Auswertung von Laufzeitprotokollen* stellt eine in ihrer Mächtigkeit neuartige Herangehensweise für post-mortem-Analysen dar. Durch Nutzung einer deduktiven Datenbank können beliebige Aspekte des Verhaltens einzelner Agenten wie auch des Systemverhaltens dargestellt und untersucht werden. Über den Nutzen des *Task-* und des *Zielmonitors* können keine konkreten Aussagen getroffen werden, weil die mentalen Modelle der implementierten Agenten zu einfach waren, als daß ein Debugging auf der intentionalen Ebene notwendig gewesen wäre.

KAPITEL 8 Zusammenfassung und Ausblick

Die hier vorgestellte Entwicklungsumgebung für agentenbasierte Dienste nimmt nicht für sich in Anspruch, eine endgültige Lösung zu bieten oder auch nur einen abschließenden Zustand erreicht zu haben. Nach dem Stand der Technik im Bereich der Agentenprogrammierung kann dies nicht der Fall sein, zu viele Designentscheidungen werden entweder ad hoc gefällt oder sind, mangels ausreichenden Erfahrungswissens, explorativer Natur. Geringe Einsatzerfahrungen, fehlende Benchmarks und der Mangel an vergleichbaren Ansätzen stehen einer qualifizierten Einschätzung des in dieser Arbeit verfolgten Ansatzes im Weg. Dennoch hat die Entwicklung zweier Anwendungen sowie der ständige Austausch mit den Mitarbeitern des Projekts, beschrittene und verworfene Design- und Implementierungsentscheidungen und nicht zuletzt die Nutzung der entwickelten Tools und Architektur zu einer Reihe von Erkenntnissen geführt, die im folgenden Abschnitt zusammengefaßt und bewertet werden. Im darauffolgenden Abschnitt folgt ein Vergleich der Arbeit mit verwandten Ansätzen. Abschließend werden Verbesserungs- und Erweiterungsvorschläge diskutiert.

8.1 Zusammenfassung

Mit der REkoS-Entwicklungsumgebung lassen sich offene, verteilte Systeme realisieren, deren Glieder diensterbringende Agenten sind. Auch ohne den Dienstkon-

text unterscheidet sich diese Arbeit damit von Architekturen, in denen Agenten in eine Umgebung eingebettet sind und von dieser interpretiert werden[1].

8.1.1 Der REkoS-Ansatz zum agentenorientierten Programmieren

Kooperative Diensterbringung beinhaltet sowohl Konzepte aus dem Distributed Problem Solving wie auch den Multi-Agent Systems. Der REkoS-Ansatz zum agentenorientierten Programmieren konzentriert sich zuallererst auf die werkzeugunterstützte Programmierung einzelner autonomer Agenten. Für die Intentionalität wurde in Abwandlung von BDI-Ansätzen[2] ein einfaches, aber effizientes hierarchisches Modell, bestehend aus Motivationen, Zielen und Tasks, entwickelt. Dienste werden mittels einer modularen Skriptsprache programmiert, deren Mächtigkeit an die Planbeschreibungssprache von dMARS[3] heranreicht. Mit Hilfe dieser zwei Konzepte lassen sich Dienste in Form von Agenten realisieren.

Den kooperativen Aspekt der gemeinsamen Diensterbringung deckt die Interaktionsfähigkeit von REkoS-Agenten ab. Mit einem an der planbasierten Kommunikation[4] orientierten Sprechaktmodell können Agenten Dienste von anderen Agenten erfragen oder delegieren. Dabei hilft die Modellierung auch des Empfängerverhaltens sowie eine gemeinsame Content-Beschreibungssprache Misinterpretationen vermeiden. Für längerfristige Interaktionen wie Verhandlungen oder Ausschreibungen wurde eine rollenbasierte Interaktionsprotokollsprache entwickelt. Eine ähnliche Herangehensweise beschreibt COOL[5], ist aber im Gegensatz zu REkoS auf zwei Teilnehmer bzw. Rollen beschränkt. Durch das Design von Kooperationsprotokollen werden die Systemaspekte der Modellierung von Agentenszenarien abgedeckt - zwar bietet REkoS keine eigene Methodologie zur Erstellung von Mehragenten-Anwendungen, hierfür kann aber auf existierende Ansätze zurückgegriffen werden[6].

Die Tatsache, daß zur Agentenprogrammierung auf vier Beschreibungssprachen zurückgegriffen werden muß, fällt nicht negativ ins Gewicht, da die Sprachen weit-

1. Interpretierte Systeme stellen COOL ([Barbuceanu, Fox 1996], Abschnitt 4.3) und die Agent-0-Familie ([Shoham 1993], Abschnitt 4.1) dar.

2. Siehe Abschnitt 2.4.2.

3. [d´Inverno, et al. 1997], siehe Abschnitt 4.2.

4. [Cohen, Perrault 1979], siehe Abschnitt 2.2.1.

5. [Barbuceanu, Fox 1996], siehe Abschnitt 4.3.

6. Z.B. auf JAFMAS, siehe Abschnitt 4.3.1.

gehend unabhängige Bereiche abdecken. Weiterhin steht mit der Toolbox ein Instrument zur Integration der einzelnen Entwicklungssprachen bzw. -werkzeuge zur Verfügung.

Einen weiteren Schwerpunkt der REkoS-Entwicklungsumgebung bildet das Testbett; hier sind Monitore und Debuggingwerkzeuge zur Beobachtung und Kontrolle einzelner Agenten wie auch ganzer Agentenszenarien unter einem Dach integriert. Debugging findet auf der Ebene der Agentenprogrammiersprache im laufenden, verteilten Betrieb statt. In Bezug auf die Funktionalitäten bietet nur COOL beim Protokolldebugging mächtigere Konzepte[7]. Mit dem Werkzeug zur Auswertung von Laufzeitprotokollen kann das Verhalten von Agenten oder Agentengruppen a posteriori nach unterschiedlichen Gesichtspunkten ausgewertet werden - einen ähnlichen Funktionsumfang lieferten bislang nur Simulationsumgebungen.

8.1.2 REkoS-Agentenarchitektur

Trotz der Verarbeitung mentaler Begriffe ist das Verhalten von Agenten ähnlich wie bei den BDI-Architekturen reaktiv, da bis auf einen einfachen Lernmechanismus auf kognitive Methoden verzichtet wurde. Die REkoS-Agentenarchitektur besteht aus einer Menge von Verarbeitungskomponenten, die auf die drei Verhaltensebenen Kommunikation, Management und Intentionalität verteilt sind. Die einzelnen Komponenten arbeiten asynchron, wobei die Verarbeitung von der im Interpreter realisierten Kontrollschleife anhand eines vorgegebenen Schemas gesteuert wird.

In der Funktionalität des Aktivitätenmanagements, das im Intentionalitätsmodul, Scheduler, Script- und Kooperationsmanager lokalisiert ist, kommen die hervorstechenden Merkmale der Agentenarchitektur zum Vorschein; sie unterstützt eine flexible Diensterbringung: Die Auswahl von Handlungsskripten findet in Abhängigkeit der Dienstparameter und normativer Ziele durch Auswahlregeln und anhand erlernten Erfahrungswissens statt. Sind die lokalen Ressourcen eines Agenten erschöpft, werden Teildienste automatisch per Delegation oder Verhandlung an andere Agenten ausgelagert.

Handlungsskripte laufen unabhängig voneinander in eigenen Threads. Skripte, zwischen denen Abhängigkeiten existieren, werden durch den Scheduler synchronisiert. Diese Komponente nimmt zudem eine dynamische Priorisierung laufender und eingeplanter Dienste vor und steuert so das Unterbrechen und Weiterverfolgen aktiver Skripte. Die Abarbeitung eines Skripts wird durch den Skriptinterpreter

7. [Barbuceanu, Fox 1996]. COOL unterstützt inkrementelles Entwickeln: Protokolle können noch zur Laufzeit verändert werden. Dies ist möglich, weil mit LISP eine Interpretersprache gewählt wurde, die gute Laufzeitkontrollmöglichkeiten bietet.

kontrolliert. Dabei findet eine ständige Überwachung der generischen Dienstparameter *Zeit*, *Qualität* und *Kosten* statt.

Beim Überschreiten vorgegebener Sollwerte sorgt das Fehlermanagement im Skript- und Kooperationsmanager für einen kontrollierten Abbruch. Eine weichere Fehlerbehandlung findet statt, falls die Ausführung von Subdiensten scheitert und sich die Parameter weiterhin im erlaubten Toleranzbereich befinden: Dann wird mittels eines Backtracking-Verfahrens im Intentionalitätsmodul nach alternativen Lösungen gesucht - die Modularität der Skripte und die baumartigen Zielstrukturen bieten hierfür Unterstützung.

8.1.3 Bewertung

Die REkoS-Agentenentwicklungsumgebung wurde zur Realisierung zweier prototypischer Anwendungen eingesetzt. Die hieraus resultierenden Erfahrungen lassen sich im Hinblick auf die gesetzten Ansprüche, eine Entwicklungs- und Ausführungsplattform für agentenbasierte Dienste bereitzustellen, wie folgt resümieren:

- **Integration von Agenten- und Systemsicht:** Einzelne Agenten werden durch die Programmierung der Intentionaliät und der Handlungsskripte realisiert. Vorhandene Sprechaktmodelle und Interaktionsprotokolle wie das Contract-Net-Protokoll zur Aufgabenverteilung sind ohne zusätzlichen Aufwand nutzbar, so daß die Einbindung eines Agenten in eine bestehende Agentengesellschaft ohne weitere Anpassungen möglich ist. Auf der anderen Seite lassen sich auch komplexe Problemstellungen durch Abbildung von Problembereichen auf Spezialagenten lösen; hierbei liegt das Schwergewicht der Implementierungsaktivitäten auf dem Design passender, domänenspezifischer Interaktionsprotokolle.
- **Skalierbare Agenten:** Basisdienste lassen sich durch Agenten ohne intentionale Verhaltensebene realisieren. Derart einfach gestrickte Agenten bieten jeweils genau einen Dienst an, den sie lokal, d.h. ohne Kooperation mit anderen Agenten erbringen. Solche rein reaktive Agenten verhalten sich wie klassische Server, wobei die Dienstnutzung mittels eines command-Sprechakts gestartet wird. Komplexere Dienste spiegeln sich in stärker stukturierten Handlungsskripten wider. Deren modulare Gliederung bietet Möglichkeiten für Unterbrechungen, die der Scheduler aufgrund von Neupriorisierungen veranlaßt. Beherrscht ein Agent zusätzlich Interaktionsprotokolle, kann er Teilaufgaben auf andere Agenten auslagern. Eine andere Art von Komplexität wird erreicht, wenn ein Agent mehrere Dienste anbietet. Hier kommt zur Auswahl der geeigneten Handlungen die intentionale Ebene ins Spiel: Zielstrukturen führen zur Aktivierung von Tasks mit nachfolgender Auswahl eines geeigneten Handlungsskripts. Verfügt der Agent über mehrere Skripte gleicher Funktionalität, findet die Zuordnung unter Zuhilfenahme normativer Ziele anhand von Optimalitätskriterien statt.
- **Fokussierte Bearbeitung:** Aus Effizienzgesichtspunkten ist die Zahl der gleichzeitig ausgeführten Handlungsskripte limitiert. Ist der Agent ausgelastet,

werden zusätzliche Dienstanfragen in eine Warteschlange eingereiht. Diese chronologisch vorgegebene Reihenfolge kann durch vorgegebene, statisch priorisierte Ziele aufgeweicht werden. Zusätzlich führt der Scheduler dynamisch eine Neuberechnung der Prioritäten im Falle von Mehrfachaktivierungen und Alarmsituationen durch und setzt diese entsprechend in ein intelligentes Aktivitätenmanagement um.

- **Testbett:** Erfahrungen im Umgang mit den Debuggingwerkzeugen zeigen, daß das Beobachten unterschiedlicher Verhaltensaspekte und globale Eingriffe in das Systemgeschehen nützlich in Bezug auf die Aufdeckung von Fehlerquellen sind. Eingriffe in die Verarbeitung eines Agenten haben sich jedoch als wenig zweckmäßig erwiesen, da die komplizierte Verarbeitungslogik gezielte Manipulationen ohne „Nebenwirkungen" kaum zuläßt. Den größten Nutzen bringt jedoch die Auswertung von Laufzeitprotokollen: Hiermit lassen sich sowohl Fehler als auch Engpässe entdecken. Weiterhin können die parallelen und zur Laufzeit nicht kontrollierbaren Aktivitäten einer größeren Agentengemeinschaft auf kausale Zusammenhänge hin untersucht werden.
- **Offenheit:** Einen Schwachpunkt bildet die Abbildung von Diensten auf Agenten. Da auf eine Dienstbeschreibungssprache verzichtet wurde, benötigen Agenten, um miteinander zu kooperieren, *dieselben* Dienstbeschreibungen. Somit sind zwei Dienste im REkoS-Kontext nur dann vergleichbar, wenn sie dieselben Namen und Argumente besitzen, eine Annahme, die im Widerspruch zur erstrebten Offenheit steht.

8.2 Vergleich mit anderen Ansätzen

REkoS ist zwar als Entwicklungsumgebung für kooperierende Dienste konzipiert worden, der hier benutzte, wenig spezifische Dienstbegriff und die Verwendung etablierter Agententechnologien erlaubt jedoch den Vergleich mit anderen allgemeinen Ansätzen.

Kein bekannter Ansatz nimmt für die Repräsentation und Ausführung von Aktivitäten den Dienstgedanken auf. Andere Systeme stellen neben der Programmierung Modellierungstechniken bereit; diese sind jedoch sämtlich Top-Down-Verfahren. Hierbei sind die Verarbeitungsmodelle der Agenten sehr einfach gehalten; von autonomer, flexibler oder intelligenter Ausführung kann hier keine Rede sein, eher von nebenläufigen Objekten. Nur die kognitiven Architekturen besitzen mächtigere Verarbeitungsmechanismen und Intelligenz, jedoch zeigen diese Modelle Schwächen in der Kommunikation und Reaktivität.

- **InteRRaP** ist ein Vertreter der Schichtenarchitekturen. Auf verschiedenen Verhaltensebenen wird reaktives, deliberatives und kooperatives Verhalten modelliert. Da die Ebenen einander kontrollieren, werden die Aktivitäten eines Agenten durch eine Art Verhaltensnetzwerk gesteuert. In bezug auf die Pro-

grammierung stellt diese Architektur nur ein relativ loses Rahmenwerk aus Konzepten und Programmierschnittstellen anstelle einer Programmiersprache bereit.

- **AgentBuilder** beruht auf einer reaktiven BDI-Architektur. Eine Entwicklungsmethodik, die auf allen Ebenen von Werkzeugen begleitet wird, unterstützt eine Top-Down-Systemmodellierung. Die Verarbeitungslogik von Agenten wird mittels Regeln programmiert, Fähigkeiten sind in Form von Plänen zu beschreiben. Einfache Sprechakte können hart verdrahtet in Pläne eingebaut werden. Interaktionsprotokolle werden nicht unterstützt, können aber durch Pläne realisiert werden.
- **dMars:** Auch dieses Entwicklungssystem verwendet das BDI-Verarbeitungsmodell und stellt Werkzeuge für die Programmierung und das Debugging bereit. Für die Modellierung wurde ein objektorientierter Ansatz adaptiert. Eine Planbeschreibungssprache mit prozeduralen und deklarativen Aspekten liefert einen ausdrucksstarken Formalismus zur Programmierung von Fähigkeiten, der die Mächtigkeit von REkoS-Skripten übersteigt. Entsprechend flexibel ist die Planausführung durch den in die Architektur integrierten Planinterpreter. Ansonsten sind Agenten jedoch mit nebenläufigen, nachrichtenverarbeitenden Objekten gleichzusetzen: Sprechaktkommunikation und Wissensverarbeitung findet nicht statt.
- **COOL** bietet eine komfortable Entwicklungsumgebung für kommunikationsbasierte Agentenanwendungen. Auch hier erfolgt ein Entwurf nach dem Top-Down-Prinzip, wobei auf das Design von Interaktionsprotokollen, die durch Regeln dargestellt werden, fokussiert wird. Eine Architektur existiert nicht, stattdessen sind Agenten Protokolle interpretierende Regelinterpreter.
- **TÆMS:** Hier werden nicht Agenten, sondern Aufgabenstrukturen modelliert. Agenten besitzen eine starke Architektur, in die eine Reihe von Koordinationsmechanismen eingebaut sind. Durch Kommunikation von Strukturen und dem Erkennen und Verarbeiten von Koordinationsbeziehungen werden die Aufgaben abgearbeitet. TÆMS stellt demnach einen Mechanismus bereit, um Probleme effizient verteilt zu lösen und hebt sich damit von den anderen Ansätzen ab, die sich auf Agentenkommunikation und Agentenprogrammiersprachen konzentrieren.
- **Kognitive Architekturen:** Wissensrepräsentation und Informationsverarbeitung sind die Grundpfeiler kognitiver Architekturen. Die Agentenarchitektur besteht aus Wissensspeichern und einem zumeist allgemeinen Inferenzmechanismus. Entsprechend sind eine Wissensbasis sowie Kontrollregeln und Heuristiken zur Steuerung der Verarbeitung zu programmieren. Agenten dieses Typs sind nicht für kooperative Anwendungen konzipiert, dafür fehlt es ihnen an Reaktivität und kommunikativen Fähigkeiten.

TABELLE 10. Agentenentwicklungssysteme im Vergleich

Ansatz	Einsatzgebiet	Sprache	Debugging	Kommunikation	Verhalten	Architektur
REkoS	MAS, DPS, Dienste	PS, Entwicklungsumgebung	+	+	reaktiv	+
InteRRaP	MAS	keine, (Spezifikation)	-	o	reaktiv, deliberativ	+
Agent-Builder	DPS	PS, Entwicklungsumgebung	+	o	reaktiv	o
dMars	DPS	PS, Entwicklungsumgebung	+	-	reaktiv	o
COOL	DPS	PS, Entwicklungsumgebung	++	+	reaktiv	-
TAEMS	DPS	Spezifikation	-	o	reaktiv	-
kognitive Architekturen	Einagentensysteme, MAS	Wissensrepräsentationssprache	?	-	deliberativ, (reaktiv)	++

8.3 Erweiterungen und Verbesserungen

Softwaresysteme erreichen selten einen finalen Zustand, und so verhält es auch mit den in dieser Dissertation entwickelten und realisierten Konzepten. Ausgeklammerte Themenbereiche und konzeptuelle Schwächen, die sich im Praxiseinsatz offenbaren geben Anlaß, über Verbesserungen und Erweiterungen nachzudenken. Diese reichen von relativ einfach zu realisierenden Maßnahmen zur besseren Integration der Werkzeuge über die Erweiterung um neuartige Funktionen bis hin zu strukturellen, das Design betreffende Änderungen.

8.3.1 Dienstvermittlung

Der Agentendämon dient ausschließlich als globales Repository für Agenten- und Ressourcenadressen. Durch Einbau weiterer Dienste könnte der Dämon zu einem Dienstvermittler ausgebaut werden. Dazu müßte jeder Agent bei seinem Start neben seinen Erreichbarkeitsinformationen auch noch die Liste der von ihm implementierten Dienste übermitteln. Der Dämon würde diese in einer Datenbank verwalten und dadurch Gelbe-Seiten-Funktionalität besitzen. Wenn ein Agent die Hilfe eines anderen Agenten benötigt, kann er die möglichen Kandidaten direkt beim Dämon erfragen, anstatt ein Kooperationsprotokoll oder eine Broadcast-Suchanfrage zu starten.

In großen, weit verteilten Szenarien, die eine gezieltere Kommunikation erfordern, können die von KQML bekannten Facilitators durch Agentendämonen realisiert werden. Jeder Dämon repräsentiert dann eine Menge von benachbarten (in bezug auf das Netzwerk) oder ähnlichen (in bezug auf die erbrachten Dienste) Agenten und kann darüberhinaus Sprechakte zu anderen Dämonen vermitteln.

Diese funktionalen Erweiterungen des Agentendämons ziehen wenige, relativ einfach zu realisierende Änderungen in der Programmiersprache und Architektur nach sich: Der Agentendämon muß selbst als Agent sichtbar sein, und dessen Funktion als Informationsvermittler muß entsprechend jedem Agenten bekannt sein. Die Kommunikation mit dem Dämonagenten erfolgt über die bekannten Konzepte der Sprechakte.

8.3.2 Dienstontologie

Weil auf eine Dienstbeschreibungssprache verzichtet wurde, stellt die Abbildung von Diensten auf Agenten einen Schwachpunkt dar. REkoS-Agenten repräsentieren Dienste durch Task-Beschreibungen, die aus einem Namen und einer Argumentliste bestehen. Derartige Beschreibungen ähneln Prozedursignaturen und besitzen eine vergleichbar schwache Ausdruckskraft.

Dieses Manko der erschwerten gegenseitigen Dienstnutzung kann durch die Verwendung von Ontologien, die Spezifikationen von Konzeptualisierungen darstellen, beseitigt werden. Ontologiebeschreibungssprachen wie Ontolingua[8] stellen einen Wissensrepräsentationsmechanismus bereit, mit dem sich Begriffe und deren Zusammenhänge erfassen lassen. Die Entwicklung von Ontologien für verschiedene Anwendungsbereiche sind Gegenstand der Forschung.

8. [Gruber 1992].

Damit ein Agent sinnvoll mit anderen Agenten kommunizieren kann ist es notwendig, daß er zumindest einen relevanten Teilbereich einer Ontologie beherrscht. Hierbei reicht allerdings eine rein syntaktische Ebene nicht aus, vielmehr benötigt der Agent ein „Verständnis“ der in der Ontologie definierten Zusammenhänge. Dieses Wissen wird üblicherweise in Form von Inferenzmechanismen in den Agenten integriert. Für die REkoS-Architektur, die mit einer geschlossenen Begriffswelt arbeitet, würde eine wie auch immer geartete Integration von Ontologien erhebliche Änderungen im Design nach sich ziehen: Das gesamte Intentionalitätsmodul müßte neu konzipiert werden. Dabei würde die stärkere wissensbasierte Ausrichtung zwangsläufig zu einer erheblichen Komplexitätssteigerung dieser Verhaltensebene führen. Es ist anzunehmen, daß hierdurch die Effizienz der Verarbeitung negativ beeinträchtigt würde.

8.3.3 Verbesserungen der Architektur

In großen Szenarien sind Mechanismen zur fokussierten Wahrnehmung wichtig, um die einzelnen Agenten vor einer Überflutung mit uninteressanten Nachrichten zu schützen. Die Wahrnehmung könnte effektiver ablaufen, wenn ein Agent die Umwelt beauftragen könnte, nur die für ihn relevanten Veränderungen an ihn zu übermitteln. Entsprechende Filter mit Signaturen verarbeitbarer Sprechakte und Nachrichteninhalte könnten in das Kommunikationsmodul integriert werden. Eine derartige Maßnahme würde den darunterliegenden Interpreter im Falle eines hohen Nachrichtenverkehrs durch Broadcasts und Sensings entlasten.

Die Aktivierungsbedingungen der Intentionen bilden ein Regelsystem, das bei wachsender Komplexität schwer managebar wird. Insbesondere das Prüfen aller Aktivierungsbedingungen bei einem Zustandswechsel der Wissensbasis kann zu Performance-Einbußen führen. Hier würde die Implementierung des Rete-Algorithmus zur inkrementellen Berechnung der Konfliktmenge Abhilfe schaffen.

Bei der Laufzeitdatenerfassung für Dienste werden Anfangs- und Endzeitpunkt der Verarbeitung gemessen; Verzögerungen, die durch vom Scheduler veranlaßte Unterbrechungen aufgrund niedriger Priorisierung auftreten, finden keine Berücksichtigung. Entsprechend führt dies bei ständiger hoher Auslastung eines Agenten zu stark verzerrten Datenerhebungen. Hier könnte eine detailliertere Wissenserhebung, die Unterbrechungen und Auslastung des Agenten berücksichtigt, zu aussagekräftigerem Wissen verhelfen. Jedoch sind die Zusammenhänge zwischen den relevanten Parametern nicht trivial, insbesondere wenn man die Einflußfaktoren von kooperativ erbrachten Diensten mit einbeziehen will: Hier kommen Verzögerungen durch Kommunikation und Auslastung anderer beteiligter Agenten hinzu.

Priorisierte Ziele bieten eine weitere Möglichkeit der fokussierten Verarbeitung. Gegenwärtig werden alle Ziele als gleich wichtig betrachtet. Über die Menge

der Reasons eines Ziels läßt sich jedoch eine mögliche Priorisierung mit nachfolgender fokussierter Behandlung der abhängigen Tasks herleiten: Ein Ziel mit mehr Aktivierungsbedingungen als eine anderes würde diesem gegenüber vorrangig behandelt werden. Eine andere Möglichkeit bestünde darin, Ziele mit statischen Prioritäten zu versehen. Diese wären bei der Modellierung zu erfassen Auch eine dynamische Priorisierung, abhängig vom Zustand der Wissensbasis, ist denkbar, wobei wiederum eine Erweiterung der Programmierschnittstelle notwendig wäre. Da jedoch ein ähnlicher Mechanismus auf der Ebene der Handlungsskripte bereits existiert, ist der Nutzen einer Zielpriorisierung als relativ gering einzuschätzen.

8.3.4 Security

Aspekte der Sicherheit sind in dieser Arbeit nicht behandelt worden. Basis aller Sicherheitsmechanismen sind Verschlüsselungstechniken, die Daten vor unerlaubten Zugriffen und Modifikationen schützen helfen. In erster Linie betrifft Sicherheit die Kommunikation, wofür entsprechende Kryptologietechnologie in die Kommunikationskomponenten der Agenten zu integrieren wäre. Die mit einer Dienstnutzung verbundenen Authentifizierungsvorgänge können durch Interaktionsprotokolle modelliert werden, die einerseits zwischen Nutzer und Dienstanbieteragent und im Falle von kooperativen Diensten zwischen den beteiligten Agenten ablaufen würden.

Auch außerhalb der Agentenarchitektur sind Sicherheitsmaßnahmen umzusetzen. Das für die Dienstnutzung wichtige Vertrauensverhältnis aller beteiligten Parteien kann von externen Sicherheitsanbietern, sogenannten Trusted Third Parties, in Form von Zertifikatsdiensten und sicheren Transaktionen hergestellt werden.

8.3.5 Testbett

Wünschenswert wäre die Verfügbarkeit einer Simulationsumgebung, die das Verhalten eines oder mehrerer Agenten vollständig kontrolliert. Hiermit könnten deterministische Testläufe stattfinden, und das Systemgeschehen wäre vollständig transparent und reproduzierbar. Angesichts des beschrittenen Wegs kompilierter, d.h. nicht interpretierter Agenten in einem verteilten System ist die Entwicklung einer adäquaten Simulationsumgebung ein schwer realisierbares Vorhaben. Zudem müßten Probleme der Zeitsimulation (simulierte vs. echte Parallelität) gelöst werden; hierfür existieren in den verfügbaren Simulationstestbetten keine befriedigenden Realisierungen.

Denkbar und mit weniger Aufwand realisierbar ist ein Verfahren zum systematischen Testen einzelner Agenten anhand einer Testsuite. Unter der Voraussetzung verfügbarer relevanter Testdaten könnten Tests automatisiert und auch reproduzierbar ablaufen. Der Testprozeß würde wie ein Agent fungieren und den zu testenden

Agenten mit den Testdaten „füttern“. Dabei würde er die bekannten Kanäle Sprechakte, Actions und Sensings benutzen. Der Programmierer kann dann mit Hilfe der bereits realisierten Monitor-Werkzeuge das Verhalten des Agenten kontrollieren.

Eine anspruchsvolle, aber lösbare Aufgabe stellt die maschinelle Unterstützung bei der Generierung von Testdaten dar. In der vom Programmierer spezifizierten Agentenbeschreibung, bzw. im nachfolgend generierten Code, finden sich viele Informationen für den Aufbau einer aussagekräftigen Testbibliothek: Aus dem Sprechaktmodell lassen sich die vom Agenten verstandenen Nachrichten ablesen; im mentalen Modell finden sich die Variablen und Schwellwerte, die für Aktivierungen verantwortlich zeichnen; Task-Definitionen geben Hinweise für das Testen von Parameterkombinationen.

Anhang

A.1 Grammatik für Skripte

```
Skript      ::= Skriptkopf Skriptrumpf
Skriptkopf  ::= <name>( Skriptargs ) : <typname>
Skriptargs  ::= Skriptarg | Skriptarg, Skriptargs
Skriptarg   ::= <argumentname> : <typname>
Skriptrumpf ::= Executable |
               return Skriptrumpf |
               IF BoolCond
                  THEN Skriptrumpf
                  ELSE Skriptrumpf
                  FI |
               DO Skriptrumpf UNTIL BoolCond |
               Skriptrumpf, Skriptrumpf
BoolCond    ::= true | false | <boolExecutable>
```

A.2 Ausschnitt einer generierten Step-Implementierungsdatei

```
#include "DoVerdacht_step9.h"
#include "Agent.h"
#include "VerdachtOderGescheitert.h"
```

```
#include "DoVerdacht.h"
#include "DoVerdacht_step10.h"
#include "DoVerdacht_step5.h"

void
DoVerdacht_step9::eval(Script* unspezSkript)
{
  DoVerdacht* skript = (DoVerdacht*) unspezSkript;
  VerdachtOderGescheitert* method =
    new VerdachtOderGescheitert(
      skript->getRegelmenge(),
      skript>getVerfeinerung());
  method->attach();
  method->addReason(skript);
  method->execute(skript->getAgent());
  if (method->getResult())
  {
    skript->setNextStep(new DoVerdacht_step10());
  }
  else
  {
    skript->setNextStep(new DoVerdacht_step5());
  }
  method->detach();
}
```

A.3 Ausschnitt aus einer generierten Skript-Headerdatei

```
#include "Script.h"

class DoVerdacht : public Script
{
  private:
    Verdacht* GroberVerdacht;
    HashTable* Symptommenge;
    // u.s.w.

  public:
    DoVerdacht(  Agent* agent,
                 Verdacht* GroberVerdacht,
                 String* StoerId);
    ~DoVerdacht();
    Verdacht* getGroberVerdacht();
    void setGroberVerdacht(
                 Verdacht* GroberVerdacht);
```

```
    HashTable* getSymptommenge();
    void setSymptommenge(HashTable* Symptommenge);
    // u.s.w.

    virtual void eval();

    List* getExcludedScripts()
    {
        if(excludedScripts) return(excludedScripts);
        List* el = new List();
        excludedScripts = el;
        excludedScripts->attach();
        return(excludedScripts);
    };
};
```

Literaturverzeichnis

[Abchiche, et al. 1992] N. Abchiche, A. Collinot, J.-M. David: Modelling cooperative reasoning. In: AAAI Workshop on Cooperation Among Heterogenous Intelligent Systems, San Jose, CA, 1992, S. 2 - 10.

[Agha 1990] G. Agha: Concurrent object-oriented programming. Communications of the ACM 33(9), 1990, S. 125 - 141.

[Albayrak, et al. 1996] S. Albayrak, U. Meyer, B. Bamberg, S. Fricke, H. Többen: Intelligent agents for the realization of electronic market services. In: The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM´96) (Hrsg. B. Crabtree, N. Jennings), Practical Application Company Ltd, London, 1996, S. 1 - 11.

[Allen, Perrault 1980] J. F. Allen, R. C. Perrault: Analyzing intention in utterances. Artificial Intelligence 15, 1980, S. 143 - 178.

[Allen 1983] J. F. Allen:Maintaining knowledge about temporal intervals. Communications of the ACM 26(11), 1983, S. 832 - 843.

[Allen 1984] J. F. Allen:Towards a general theory of action and time. Artificial Intelligence 23(2), 1984, S. 123 - 154.

[Ambite, Knoblock 1997] J. L. Ambite, C. A. Knoblock: Planning by rewriting: Efficiently generating high-quality plans. In: Fourteenth National Conference on Artificial Intelligence (AAAI-97), 1997, S. 706 - 713.

[Ambros-Ingerson, Steel 1990] J. A. Ambros-Ingerson, S. Steel: Integrating planning, execution and monitoring. In: Readings in Planning (Hrsg. J. Allen, J. Hendler, A. Tate), Morgan Kaufmann, 1990, San Mateo, CA, S. 735 - 740.

[Anderson 1991] J. Anderson: Cognitive Architectures in a rational analysis. In: Architectures for Intelligence (Hrsg. K. VanLehn), Lawrence Erlbaum Associates, 1991, Hillsdale, N.J., S. 1 - 24.

[Austin 1962] J. L. Austin: How to do things with words. 1962.

[Baker 1998] A. D. Baker: A survey of factory control algorithms which can be implemented in a multi-agent heterarchy. In: Journal of Manufacturing Systems 17(4), 1998, S. 297 - 320.

[Ballmer, Brennenstuhl 1981] T. Ballmer, W. Brennenstuhl: Speech act classification: A study in the lexical analysis of english speech activity verbs. 1981, Berlin, Heidelberg.

[Barbuceanu, Fox 1995] M. Barbuceanu, M. S. Fox: COOL: A language for describing coordination in multi agent systems. In: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95) (Hrsg. V. Lesser), AAAI Press, San Francisco, CA, 1995, S. 14 - 24.

[Barbuceanu, Fox 1996] M. Barbuceanu, M. S. Fox: The agent building shell: a practical architecture for multiagent systems. REkoS-Project Deliverable, Univ. of Toronto, 1996.

[Bates, et al. 1992] J. Bates, A. Bryan Loyall, W. Scott Reilly: Integrating reactivity, goals, and emotion in a broad agent. Technical Report CMU-CS-92-142, Carnegie-Mellon University, Pittsburgh, PA, 1992.

[Bates 1994] J. Bates:The role of emotion in believable agents. Communications of the ACM, 1994, 37(7), S. 122 - 125.

[Boddy, Dean 1994] M. Boddy, T. Dean: Deliberation scheduling for problem solving in time-constrained environments. Artificial Intelligence, 1994, 67, S. 245 - 285.

[Booch 1994] G. Booch: Object-oriented analysis and design with applications. 1994, Redwood City, CA.

[Boutilier, et al. 1995] C. Boutilier, T. Dean, S. Hanks: Planning under uncertainty: Structural assumptions and computational leverage. In: European Planning Workshop, 1995.

[Brando 1995] T. Brando: Comparing DCE and CORBA. Technical report MP 95B-93, MITRE Corporation, März 1995, http://www.mitre.org/research/domis/reports/DCEvCORBA.html.

[Bratman 1987] M. E. Bratman: Intentions, Plans, and Practical Reason. Cambridge, MA, Harvard University Press, 1987.

[Bratman, et al. 1988] M. E. Bratman, D. J. Israel, M. E. Pollack: Plans and resource-bounded practical reasoning. Computational Intelligence 4(4), 1988, S. 349 - 355.

[Bray 1998] T. Bray: RDF and Metadata. XML-Report, 9.6.1998, http://www.xml.com/xml/pub/Bray_Tim.

[Brazier, et al. 1997] F. M. T. Brazier, D. Dunin Keplicz, N. Jennings, J. Treur: DESIRE: Modelling multi-agent systems in a compositional formal framework. International Journal of Cooperative Information Systems 6, Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, (Hrsg. M. Huhns, M. Singh), 1997, S. 67 - 94.

[Briot 1989] J.-P. Briot: Actalk: A testbed for classifying and designing Actor languages in the Smalltalk-80 environment. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP´89), Cambridge University Press, 1989, S. 109 - 129.

[Briot 1992] J.-P. Briot: Object-oriented concurrent programming: Introducing a new programming methodology. In: 7th Int Meeting of Young Computer Scientists (IMYCS ’92), Gordon & Breach, 1992.

[Brooks 1986] R. A. Brooks: A robust layered control system for a mobile robot. IEEE Journal of Robotics and Automation 2(1), 1986, S. 14 - 23.

[Brooks 1991a] R. A. Brooks: Intelligence without reason. In: Twelfth International Joint Conference on AI (IJCAI-91), Sydney, 1991, S. 569 - 595.

[Brustoloni 1991]J.C. Brustoloni: Autonomous agents: characterization and requirements. Technical Report CMU-CS-91-204, Pittsburgh. Carnegie Mellon University, 1991.

[Burmeister, Sundermeyer 1991] B. Burmeister, K. Sundermeyer: Cooperative problem-solving guided by intentions and perception. In: Decentralized AI 3 - Proceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-91) (Hrsg. E. Werner, Y. Demazeau), Elsevier Science Publishers B.V., 1991, Amsterdam, S. 77 - 92.

[Burmeister, et al. 1993] B. Burmeister, A. Haddadi, K. Sundermeyer: Generic configurable cooperation protocols for multi-agent systems. In: Pre-Proceedings (MAAMAW-93, LNAI-957), 1993.

[Bylander 1994] T. Bylander: The computational complexity of propositional STRIPS planning. Artificial Intelligence 69, 1994, S. 165 - 204.

[Campbell, d’Inverno 1990] J. A. Campbell, M. P. d’Inverno: Knowledge interchange protocols. In: Second European Workshop on Modelling Autonomous Agents in a Multi-Agent World (Hrsg. Y. Demazeau, J.-P. Müller), North-Holland, Saint Quentin en Yvelines, France, 1990, S. 63 - 80.

[Carbonell, et al. 1991] J. Carbonell, O. Etzioni, Y. Gil, R. Joseph, C. Knoblock, S. Min-

ton, M. Veloso: PRODIGY: An integrated architecture for planning and learning. ACM SIGART Bulletin 2(4), 1991, S. 51 - 55.

[Carver, Lesser 1992] N. Carver, V. Lesser: The evolution of blackboard control architectures. Computer Science Technical Report 92-71, UMASS, 1992.

[Castelfranchi 1995] C. Castelfranchi: Guarantees for autonomy in cognitive agent architectures. In: Intelligent Agents - Theories, Architectures, and Languages (Hrsg. M. Wooldridge, N.R. Jennings), Springer Verlag, 1995, S. 56 - 70.

[Chang, Woo 1992] M. K. Chang, C. C. Woo: SANP: a communication level protocol for negotiations. In: Decentralized A. I. - Proceedings of the 3rd European Workshop on Modelling Autonomous Agents in Multi-Agent Worlds (MAAMAW-92) (Hrsg. Y. Demazeau, J.P. Müller), Elsevier Science Publishers B. V., 1992, Amsterdam, S. 31 - 54.

[Chapman 1987] D. Chapman: Planning for conjunctive goals. Artificial Intelligence 32, 1987, S. 333 - 378.

[Chapman, Montesi 1995] M. Chapman, S. Montesi: Overall concepts and principles of TINA. TINA Baseline, version 1.0, 17. Februar 1995, http://www.tinac.com.

[Chauhan 1997] D. Chauhan: JAFMAS: A Java-based agent framework for multiagent systems development and implementation. Diploma thesis, Universität ECECS Department, University of Cincinnati, 1997.

[Chavez, Maes 1996] A. Chavez, P. Maes: Kasbah: An agent marketplace for buying and selling goods. In: Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-96), London, 1996.

[Coen 1994] M. H. Coen: SodaBot: A software agent environment and construction system. A.I. Technical Report 1493, MIT, Artificial Intelligence Laboratory, 1994.

[Cohen, Perrault 1979] P. R. Cohen, C. R. Perrault: Elements of a plan-based theory of speech acts. Cognitive Science 3(3), 1979, S. 177 - 212.

[Cohen, Feigenbaum 1982] P. R. Cohen, E. A. Feigenbaum: The handbook of artificial intelligence. 1982, San Mateo, CA.

[Cohen, Levesque 1987] P. R. Cohen, H. J. Levesque: Persistence, intention, and commitment. In: 1986 workshop on Reasoning about Actions and Plans (Hrsg. M.P. Georgeff, A.L. Lansky), Morgan Kaufman Publishers, San Mateo, CA, 1987.

[Cohen, et al. 1989] P. R. Cohen, M. L. Greenberg, D. M. Hart, A. E. Howe: Trial by fire: Understanding the design requirements for agents in complex environments. AI Magazine 10(3), 1989, S. 32 - 48.

[Cohen, Levesque 1990] P. R. Cohen, H. J. Levesque: Intention is choice with commit-

ment. Artificial Intelligence 42, 1990, S. 213 - 261.

[Cohen, et al. 1990] P. R. Cohen, H. J. Levesque, J. H. T. Nunes: On acting together. In: Proc. AAAI-90, 1990, S. 94 - 99.

[Cohen, et al. 1994] P.R. Cohen, A. Cheyer, M. Wang, S.C. Baeg: An open agent architecture. In: Proceedings of the AAAI Spring Symposium, Series on Software Agents (Hrsg. O. Etzioni), Stanford, CA, 1994, S. 1 - 8.

[Cohen, Levesque 1995] P.R. Cohen, H. Levesque: Communicative actions for artificial agents. In: International Conference on Multi-Agent Systems (ICMAS-95) (Hrsg. V. Lesser), AAAI-Press, San Francisco, 1995, S. 65 - 72.

[Collins, et al. 1991] G. Collins, L. Birnbaum, B. Kurwich, M. Freed: Model-based integration of planning and learning. SIGART Bulletin 2, 1991, S. 56 - 60.

[Conry, et al. 1986] S. E. Conry, R. A. Meyer, V. R. Lesser: Multistage Negotiation in Distributed Planning. COINS Technical Report 86-87, Dept. of Computer and Information Sciences, Univ. of Massachusetts, 1986.

[Conry, et al. 1991] S. E. Conry, K. Kuwabara, V. R. Lesser, R. A. Meyer: Multistage negotiation for distributed constraint satisfaction. IEEE Trans. Syst., Man, Cybern. 21(6), 1991, S. 1462 - 1477.

[Crowston, Malone 1988] K. Crowston, T. W. Malone: Intelligent software agents. Byte 12, 1988, S. 267 - 271.

[Davies, Edwards 1994] W. H. E. Davies, P. Edwards: Agent-K: an integration of AOP and KQML. Technical Report, AUCS/TR9406, King's College, 1994.

[Davis, Smith 1983] R. Davis, R. G. Smith: Negotiation as a metaphor for distributed problem solving. AI Journal 20, 1983, S. 63 - 109.

[de Kleer 1986] J. de Kleer: An assumption-based TMS. Artificial Intelligence 28, 1986, S. 127 - 162.

[Dechter, et al. 1991] R. Dechter, I. Meiri, J. Pearl: Temporal constraint networks. Artificial Intelligence 49, 1991, S. 61 - 95.

[Decker, et al. 1990] K. Decker, V. Lesser, R. Whitehair: Extending a blackboard architecture for approximate processing. The Journal of Real-Time Systems 2(1/2), 1990.

[Decker, Lesser 1991] K. S. Decker, V. Lesser R.: Generalizing the partial global planning algorithm. Computer Science Technical Report 01003, University of Massachusetts, Amherst, MA, 1991.

[Decker 1995] K. S. Decker: Environment centered analysis and design of coordination mechanisms. Ph.D. Thesis, University of Massachusetts, 1995.

[Decker, Lesser 1995] K. S. Decker, V. R. Lesser: Designing a family of coordination algorithms. Computer Science Technical Report 94-14, UMASS, 1995. Erschienen

auch in gekürzter Fassung in: Proceedings of the First International Conference on Multi-Agent Systems.

[Dennet 1987] D. C. Dennet: The intentional stance. 1987, Cambridge, MA.

[Doran, et al. 1997] J. E. Doran, S. Franklin, N. R. Jennings, T. J. Norman: On cooperation in multi-agent systems. The Knowledge Engineering Review 12(3), 1997.

[Doyle 1979] J. Doyle: A truth maintenance system. Artificial Intelligence 12, 1979, S. 231 - 272.

[Dreyfus 1979] H. L. Dreyfus: What computers can´t do: The limits of artificial intelligence. 1979, New York.

[Durfee, Montgomery 1989] E. H. Durfee, T. A. Montgomery: MICE: A flexible testbed for intelligent coordination experiments. In: Proceedings of the 1989 International Workshop on Distributed Artificial Intelligence (IWDAI-89), 1989.

[Durfee, Lesser 1989] E. H. Durfee, V. R. Lesser: Negotiating task decomposition and allocation using partial global planning. In: Distributed Artificial Intelligence (Hrsg. L. Gasser, M.N. Huhns), Morgan Kaufmann, 1989, Pitman, London, S. 229 - 242.

[Durfee, Montgomery 1990] E. H. Durfee, A. Montgomery: A hierarchical protocol for coordinating multiagent behaviours: an update. In: Conference of the AAAI, 1990, S. 86 - 93.

[Durfee, Lesser 1991] E. H. Durfee, V. R. Lesser: Partial global planning: A coordination framework for distributed hypothesis formation. IEEE Transactions on Systems, Man, and Cybernetics 21(5), 1991, S. 1167 - 1183.

[d´Inverno, et al. 1997] M. d´Inverno, D. Kinny, M. Luck: A formal specification of dMARS. In: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages, ATAL´97. Auch: AAIS - Technical Note 73, Australian Artificial Intelligence Institute, Melbourne, 1997.

[Erman, Lesser 1975] L. D. Erman, V. R. Lesser: A multi-level organisation for problem solving using many diverse cooperating sources of knowledge. In: Fourth International Joint Conference on Artificial Intelligence (IJCAI-75), Stanford, CA, 1975, S. 483 - 490.

[Ferber, Jacopin 1991] J. Ferber, E. Jacopin: The framework of eco-problem solving. In: Decentralized A.I. 2 (Hrsg. Y. Demazeau, J. P. Müller), Elsevier Science Publishers B. V., 1991, Amsterdam, S. 181 - 193.

[Ferguson 1992a] I. A. Ferguson: TouringMachines: Autonomous agents with attitudes. Computer 25(5), 1992, S. 51 - 55.

[Ferguson 1992b] I. A. Ferguson: TouringMachines: An architecture for dynamic, rational, mobile agents. PhD, Universität University of Cambridge, 1992.

[Ferguson 1994] I. A. Ferguson: Integrated control and coordinated behavior: A case for agent models. In: Intelligent Agents. ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, Amsterdam, Netherlands, 1994, S. 203 - 218.

[Fikes, Nilsson 1971] R. E. Fikes, N. J. Nilsson: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 5(2), 197, S. 189 - 208.

[Finin, et al. 1994a] T. Finin, Y. Labrou, J. Mayfield: KQML as an agent communication language. In: 3rd Int. Conf. on Information and Knowledge Management, ACM Press, 1994.

[Finin, et al. 1994b] T. Finin, R. Fritzson, D. McKay, R. McEntire: KQML - a language and protocol for knowledge and information exchange. Technical Report, CS-94-02, University of Maryland, Valley Forge Engineering Center, Unisys Corporation, 1994.

[FIPA 1997] FIPA - Foundation for Intelligent Physical Agents: Agent communication language. FIPA 97 Specification Part 2, Genf, 1997. http://drogo.cselt.stet.it/fipa/spec/fipa97/fipa97.htm.

[Fischer 1993] K. Fischer: The rule-based multi-agent system MAGSY. In: CKBS’92 Workshop, DAKE Centre, Keele University, 1993.

[Fischer, et al. 1995] K. Fischer, J. P. Müller, M. Pischel: A model for cooperative transportation scheduling. In: First International Conference on Multi-Agent Systems (ICMAS ´95), San Francisco, CA, 1995.

[Fischer, et al. 1996a] K. Fischer, J. P. Müller, M. Pischel: AGENDA: A general testbed for DAI applications. In: Foundations of DAI (Hrsg. N.R. Jennings, G.M.P. O'Hare), 1996.

[Fischer, et al. 1996b] K. Fischer, J. P. Müller, M. Pischel: A pragmatic BDI architecture. In: Intelligent Agents II (LNAI 1037) (Hrsg. J. P. Müller, M. Wooldridge, M.Tambe), 1996.

[Fisher 1994a] M. Fisher: Representing and executing agent-based systems. In: Intelligent Agents - ECAI-94 Workshop on Agent Theories, Architectures, and Languages (Hrsg. M. J. Wooldridge, N. R. Jennings), Springer, 1994, S. 307 - 323.

[Fisher 1994b] M. Fisher: A survey of Concurrent METATEM - the language and its applications. In: Temporal Logic - Proceedings of the First International Conference (Hrsg. D. M. Gabbay, H. J. Ohlbach), Springer-Verlag, 1994, S. 480 - 505.

[Flusser 1985]V. Flusser:Ins Universum der technischen Bilder. Göttingen, 1985.

[Foner 1993] L. N. Foner: What´s an agent, anyway? A sociological case study. Agents Memo 93-01, MIT Media Lab, Cambridge, MA, 1993. http://

foner.www.media.mit.edu/people/foner/Julia/Julia.html.

[Foner 1997] L. N. Foner: Entertaining agents: a sociological case study. In: First International Conference on Autonomous Agents (Agents'97), Marina del Rey, CA, 1997.

[Forgy 1981] C. Forgy: Ops5 users' manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[Franklin, Graesser 1996] S. Franklin, A. Graesser: Is it an agent, or just a program? A taxonomy for autonomous agents. In: Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[Fricke, et al. 1998] S. Fricke, S. Albayrak, U. Meyer, B. Bamberg, H. Többen: A development and test environment for agent based telematic services. In: Intelligent Agents for Telecommunications Applications (Hrsg. S. Albayrak), IOS Press, 1998, S. 225 - 251.

[Garvey, Lesser 1993] A. Garvey, V. Lesser: Design-to-time real-time scheduling. IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Planning, Scheduling and Control 23(6), 1993, S. 1491 - 1502.

[Garvey, Lesser 1994] A. Garvey, V. Lesser: A survey of research in deliberative real-time artificial intelligence. Real-Time Systems 6(3), 1994, S. 317 - 347.

[Gasser, et al. 1987] L. Gasser, C. Braganza, N. Hermann: MACE: a flexible testbed for distributed AI research. In: Distributed Artificial Intelligence (Hrsg. M.N. Huhns), Morgan Kaufmann, 1987, Los Altos, S. 119 - 152.

[Gat 1992] E. Gat: Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In: AAAI-92, MIT Press, 1992, S. 809 - 815.

[Gat 1993] E. Gat:On the role of stored internal state in the control of autonomous mobile robots. AI Magazine 14(1), 1993, S. 64 - 73.

[Genesereth, et al. 1986] M. R. Genesereth, M. L. Ginsberg, J. S. Rosenschein: Cooperation without communication. In: Proc. AAAI-86, 1986, S. 51 - 57.

[Genesereth, et al. 1991] M. R. Genesereth, R. E. Fikes, e. al.: Knowledge interchange format version 3.0 reference manual. Logic-92-1 Report, Stanford University Logic Group, 1991.

[Genesereth 1992] M. R. Genesereth: An agent-based approach to software interoperability. In: DARPA Software Technology Conference, 1992.

[Genesereth, Ketchpel 1994] M. R. Genesereth, S. P. Ketchpel: Software agents. Communications of the ACM 37(7), 1994, S. 48 - 53.

[Gent, et al. 1997] I. P. Gent, E. MacIntyre, P. Prosser, T. Walsh: The scaling of search cost. In: Fourteenth National Conference on Artificial Intelligence (AAAI-97),

1997, S. 315 - 320.

[Georgeff, Lansky 1987] M. P. Georgeff, A. L. Lansky: Reactive reasoning and planning. In: Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), Seattle, WA, 1987, S. 677 - 682.

[Gilbert 1997] D. Gilbert: Intelligent agents - the right information at the right time. White paper, IBM, 1997. http://www.networking.ibm.com/iag/iaghome.html.

[Ginsberg 1989] M. Ginsberg: Universal planning: An (almost) universally bad idea. AI Magazine 10(4), 1989, S. 40 - 44.

[Glover 1989] F. Glover: Tabu search - part I. ORSA Journal on Computing 1(3), 1989, S. 190 - 206.

[Gruber 1992] T.R. Gruber: Ontolingua: a mechanism to support portable ontologies. Technical Report KSL 91-66, Stanford University, Knowledge Systems Laboratory, 1992.

[Goldberg 1989] D. E. Goldberg: Genetic algorithms in search, optimization, and Mmachine learning. 1989.

[Gulliver 1979] P. H. Gulliver: Disputes and negotiations - A cross-cultural perspective. 1979.

[Halpern, Moses 1992] J. Halpern, Y. Moses: A guide to completeness and complexity for modal logics of knowledge and belief. Artificial Intelligence 54, 1992, S. 319 - 379.

[Hanks, et al. 1993a] S. Hanks, D. Nguyen, C. Thomas: A beginner´s guide to the Truckworld simulator. Technical Report UW-CSE-TR 93-06-09, Dept. of CS & E, Univ. of Washington, 1993.

[Hanks, et al. 1993b] S. Hanks, M. E. Pollack, P. R. Cohen: Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. AI Magazine 14(4), 1993, S. 17 - 42.

[Hansen, et al. 1998] P. F. Hansen, C. A. Licciardi (Editoren): Service component specification: computational model and dynamics. TINA-C deliverable, version 1.0, 19. Januar 1998.

[Harnad 1990] The symbol grounding problem. Physica D 42, 1990, S. 335 - 346.

[Hart, Cohen 1990] D. Hart, P. Cohen: PHOENIX: A test bed for shared planning research. In: Proceedings of the NASA Ames Workshop on Benchmarks and Metrics, Moffet Field, California, 1990.

[Haugeneder, Steiner 1991] H. Haugeneder, D. Steiner: Cooperation structures in multiagent systems. In: GI-Kongreß 91 (Hrsg. Brauer, Hernandez), 1991, S. 160 - 171.

[Haugeneder, et al. 1994] H. Haugeneder, D. Steiner, F.G. McCabe: IMAGINE: A frame-

work for building multi-agent systems. In Proceedings of the 1994 International Working Conference on Cooperating Knowledge Based Systems (CKBS-94, Hrsg. S.M. Deen), DAKE Centre, University of Keele, UK, 1994.

[Hayes-Roth 1988] B. Hayes-Roth: Making intelligent systems adaptive. Technical Report STAN-CS-88-1226, Stanford University, 1988.

[Hayes-Roth 1990] B. Hayes-Roth: Architectural foundations for real-time performance intelligent systems. The Journal of Real-Time Systems 2, 1990, S. 99 - 125.

[Hayes-Roth 1995] B. Hayes-Roth: An architecture for adaptive intelligent systems. Artificial Intelligence 72, 1995, S. 329 - 365.

[Hertzberg 1994] J. Hertzberg: Planen von Aktionen und Reaktionen. LS-8 Report #7, Universität Dortmund, Lehrstuhl für KI, 1994.

[Hewitt 1991] C. E. Hewitt: Open information systems semantics for distributed artificial intelligence. Artificial Intelligence 47, 1991, S. 79 - 106.

[Hexmoor et al 1993a] H.H. Hexmoor, J.M. Lammens, S.C. Shapiro: An autonomous agents architecture for integrating "unconscious" and "concious", reasoned behaviours. In: Computer Architectures for Computer Vision - 93, 1993. Auch erhältlich als Technical Report 93-97, University at Buffalo, Dept. of Comp. Sci. and Engineering.

[Hexmoor et al 1993b] H.H. Hexmoor, J.M. Lammens, S.C. Shapiro: Embodiment in GLAIR: A grounded layered architecture with integrated reasoning for autonomous agents. Technical Report 93-10, University at Buffalo, Dept. of Comp. Sci. and Engineering, Februar 1993.

[Horvitz 1987] E. Horvitz: Reasoning about beliefs and actions under computational resource constraints. In: Third Workshop on Uncertainty in Artificial Intelligence, AAAI, Seattle, WA, 1987, S. 429 - 444.

[HP 1998] HP OpenView Network Node Manager 6.0 for the UNIX and Windows NT operating system. HP Product Brief, 1998, http://www.openview.hp.com/solutions/network.

[Huffman, et al. 1993] S. Huffman, C. Miller, J. Laird: Learning for instruction: A knowledge-level capability within a unified theory of cognition. In: 15th Annual Meeting of the Cognitive Science Society, Boulder, 1993. http://krusty.eecs.umich.edu/people/huffman/huffman.html.

[ISO 7498-4] OSI Basic Reference Model Part 4: Management Framework.

[Jennings 1992] N. R. Jennings: Towards a cooperation knowledge level for collaborative problem solving. In: Proceedings of the 10th European Conference on Artificial Intelligence, Wien, 1992, S. 224 - 228.

[Jennings, Wittig 1992] N. R. Jennings, T. Wittig: ARCHON: Theory and practice. In:

Distributed Artificial Intelligence: Theory and Practice (Hrsg. N. M. Avouris, L. Gasser), Kluwer Academic Press, 1992, S. 179 - 195.

[Jennings, Campos 1997] N. R. Jennings, J. R. Campos: Towards a social level characterisation of socially responsible agents. IEE Proceedings on Software Engineering 144(1), 1997, S. 11 - 25.

[Kaelbling, et al. 1996] L. P. Kaelbling, M. L. Littman, A. W. Moore: Reinforcement learning: A survey. Journal of Artificial Intelligence Research 4, 1996.

[Kalenka, Jennings 1995] S. Kalenka, N. R. Jennings: On social attitudes: A preliminary report. In: International Workshop on Decentralised Intelligent and Multi-Agent Systems (DIMAS'95), Krakow, Poland, 1995, S. 233 - 240.

[Kalenka, Jennings 1997] S. Kalenka, N. R. Jennings: Socially responsible decision making by autonomous agents. In: Proc. Fifth Int. Colloq. on Cognitive Science (ICCS'97), Donostia, Spain, 1997.

[Kass, Raferty 1994] R. E. Kass, A. E. Raferty: Bayes Factors. Technical Report 571, Dept. of Statistics, Carnegie-Mellon Univ., 1994.

[Kautz, Ladkin 1992] H. A. Kautz, P. B. Ladkin: Integrating metric and qualitative temporal reasoning. In: Ninth National Conference on Artificial Intelligence (AAAI-91), 1992, S. 241 - 246.

[Ketchpel 1994] S. Ketchpel: Forming coalitions in the face of uncertain rewards. In: National Conference on Artificial Intelligence, Seattle, WA, 1994, S. 414 - 419.

[Kinny, Georgeff 1991] D. N. Kinny, M. P. Georgeff: Commitment and effectiveness of situated agents. In: Twelfth International Joint Conference on AI (IJCAI-91), Sydney, 1991, S. 82 - 88.

[Kinny, Georgeff 1996] D. Kinny, M. Georgeff: Modelling and design of multi-agent systems. In: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96), auch: AAII Technical Note 96, Australian Artificial Intelligence Institute, 1996.

[Kinny, et al. 1996] D. Kinny, M. Georgeff, A. Rao: A methodology and modelling technique for systems of BDI agents. AAII Technical Note, 58, Australian Artificial Intelligence Institute, 1996.

[Kirkpatrick, et al. 1983] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi: Optimization by simulated annealing. Science 220(4), 1983, S. 671 - 680.

[Kripke 1963] S. Kripke: Semantical analysis of modal logic. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 9, 1963, S. 67 - 96.

[Kristiansen 1997] L. Kristiansen (Editor): Service architecture. TINA-C baseline, version 5.0, 16. Juni 1997.

[Laird, et al. 1987] J. Laird, A. Newell, P. Rosenbloom: Soar: an architecture for general intelligence. Artificial Intelligence 33, 1987, S. 1 - 64.

[Lavine, Schervish 1997] M. Lavine, M.J. Schervish: Bayes factors: what they are and what they are not. CMU-Technical Report 652, Januar 1997, http:// lib.stat.cmu.edu/cmu-stats/tr/tr652/tr652.html.

[Lebowitz 1986] M. Lebowitz: Integrated learning: Controlling experimentation. Cognitive Science 2, 1986.

[Lesser, Corkill 1983] V. R. Lesser, D. D. Corkill: The distributed vehicle monitoring testbed: a tool for investigating distributed problem solving networks. AI Magazine, 1983, S. 15 - 33.

[Levin, Moore 1977] J. A. Levin, J. A. Moore: Dialogue-games: Metacommunication structures for natural language interaction. Cognitive Science 1, 1977, S. 395 - 420.

[Levinson 1981] S. C. Levinson: The essential inadequacies of speech act models in dialogues. In: Possibilities and Limitations of Pragmatics (Hrsg. H. Parrett), J. Benjamin, Amsterdam, 1981.

[Lieberman 1995] H. Lieberman: Letizia: an agent that assists web browsing. Proceedings of the International Joint Conference on Artificial Intelligence, Montreal, 1995.

[Liu, et al. 1997] J. Liu, H. Quin, Y. Y. Tang, Y. T. Wu: Adaptation and learning in animated creatures. In: Autonomous Agents ´97 - Online Proceedings (Hrsg. J. Müller), 1997.

[Ljungberg, Lucas 1992] M. Ljungberg, A. Lucas: The OASIS air-traffic management system. In Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence, PRICAI ’92. Seoul, Korea, 1992.

[Lux, et al. 1992] A. Lux, F. Bomarius, D. Steiner: A model for supporting human computer cooperation. In: AAAI-92 Workshop on Cooperation among Heterogeneous Intelligent Systems, 1992.

[Maes 1989] P. Maes: The dynamics of action selection. In: Proceedings of IJCAI-89, Detroit, MI, 1989, S. 991 - 997.

[Maes 1990] P. Maes: Situated agents can have goals. Robotics and Autonomous Systems 6(1&2), 1990, S. 49 - 70.

[Maes, Brooks 1990] P. Maes, R. A. Brooks: Learning to coordinate behaviors. In: Eighth National Conference on Artificial Intelligence, Morgan Kaufmann, 1990, S. 796 - 802.

[Maes 1991] P. Maes: The agent network architecuture ANA. SIGART Bulletin 2(4), 1991, S. 136 - 139.

[Maes, Kozierok 1993] P. Maes, R. Kozierok: Learning interface agents. In: Proceedings of AAAI ’93 Conference, Washington, D.C., 1993, S. 459 - 465.

[Maes 1994a] P. Maes: Modeling adaptive autonomous agents. Artificial Life Journal 1(1 & 2), 1994, S. 135 - 162.

[Magedanz, Popescu-Zeletin 1996] T. Magedanz, R. Popescu-Zeletin: Intelligent Networks: basic technology, standards and evolution. Thomson Computer Press, London, 1996.

[Mahadevan, Connell 1991] M. Mahadevan, J. Connell: Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In: Eighth International Workshop on Machine Learning, 1991, S. 328 - 332.

[Malik, Binford 1983] J. Malik, T. O. Binford: Reasoning in time and space. In: Eighth International Joint Conference on Artificial Intelligence (IJCAI-83), 1983, S. 343 - 345.

[Malone, Crowston 1991] T. W. Malone, K. Crowston: Toward an interdisciplinary theory of coordination. Technical Report 120, Center for Coordination Science, MIT Sloan School, 1991.

[Mason, Johnson 1989] C. L. Mason, R. R. Johnson: DATMS: A framework for distributed assumption based reasoning. In: Distributed Artificial Intelligence (Hrsg. L. Gasser, M. Huhns), Pitman Publishing: London and Morgan Kaufman, 1989, San Mateo, CA, S. 293 - 318.

[Mataric 1991] M. J. Mataric: Behavioral synergy without explicit integration. SIGART Bulletin 2, 1991, S. 85 - 88.

[Mataric 1992] M. J. Mataric: Behavior-based systems: Main properties and implications. In: IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems, Nizza, 1992, S. 46 - 54.

[McCabe, Clark 1994] F.G. McCabe, K.L. Clark: April - agent process interaction language. In: Pre-proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages (Hrsg. M. Wooldridge, N.R. Jennings), Amsterdam, 1994, S. 280 - 296.

[McCarthy, Hayes 1969] J. McCarthy, P. J. Hayes: Some philosophical problems form the standpoint of artificial intelligence. In: Machine Intelligence (Hrsg. B. Meltzer, D. Michie), American elsevier, 1969, New York, S. 463 - 502.

[McCarthy 1977] J. McCarthy: Epistemological problems of artificial intelligence. In: Fifth International Joint Conference on Artificial Intelligence (IJCAI-77), Cambridge, MA, 1977, S. 1038 - 1044.

[McCarthy 1980] J. McCarthy: Circumscription - a form of non-monotonoc reasoning. Artificial Intelligence 13, 1980, S. 27 - 39.

[McDermott 1982] D. McDermott:A temporal logic for reasoning about processes and plans. Cognitive Science 6, 1982, S. 101 - 155.

[McDermott 1990] D. McDermott: Planning and acting. In: Readings in Planning (Hrsg. J. Allen, J. Hendler, A. Tate), Morgan Kaufmann, 1990, San Mateo, CA, S. 225 - 244.

[Michalski 1996] R. Michalski: Understanding the nature of learning. In: Machine Learning - An Artificial Intelligence Approach (Hrsg. Michalski, Carbonell, Mitchell), Morgan Kaufmann, Los Altos, CA, 1996.

[Miller, Drexler 1988] M. S. Miller, K. E. Drexler: Markets and computation: Agoric open systems. In: The Ecology of Computation (Hrsg. B.A. Hubermann), Elsevier Science Publishers, 1988, S. 133 - 176.

[Minsky 1985] M. Minsky: Society of mind. 1985, New York.

[Mitchell 1982] T. M. Mitchell: Generalization as search. Artificial Intelligence 18(2), 1982, S. 203 - 226.

[Mitchell, et al. 1991] T. M. Mitchell, J. Allen, P. Chalasani, J. Cheng, O. Etzioni, M. Ringuette, J. C. Schlimmer: Theo: A framework for self-improving systems. In: Architectures for Intelligence (Hrsg. K. VanLehn), Lawrence Erlbaum Associates, 1991, Hillsdale, NJ, S. 323 - 355.

[Mitchell 1991] T. M. Mitchell:Plan-then-compile architectures. SIGART Bulletin, 1991, S. 136 - 139.

[Montgomery, Durfee 1990] T. A. Montgomery, E. H. Durfee: Using MICE to study intelligent dynamic coordination. In: 2nd International IEEE Conference on Tools for Artificial Intelligence, IEEE Comput. Soc. Press, Herndon, VA, USA, 1990, S. 438 - 444.

[Moore 1985] R. C. Moore: Semantical considerations on nonmonotonic logic. Artificial Intelligence 25, 1985.

[Morik 1993] K. Morik: Maschinelles Lernen. LS-8 Report #1, Universität Dortmund, Fachbereich Informatik, Lehrstuhl für KI, 1993.

[Mulder, et al. 1997] M. Mulder, J. Treux, M. Fisher: Agent modelling in concurrent METATEM and DESIRE. In: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages, ATAL´97, 1997.

[Müller, Pischel 1993] J. P. Müller, M. Pischel: InteRRaP: Eine Architektur zur Modellierung Flexibler Agenten. In: Verteilte Künstliche Intelligenz (Hrsg. J. Müller), BI-Wissenschaftsverlag, 1993, S. 45 - 54.

[Müller, Pischel 1994] J. P. Müller, M. Pischel: Modelling interacting agents in dynamic environments. In: Eleventh European Conference on Artificial Intelligence (ECAI-94), Amsterdam, 1994, S. 709 - 713.

[Müller, et al. 1995] J. P. Müller, M. Pischel, M. Thiel: Modelling reactive behaviour in vertically layered agent architectures. In: Intelligent Agents: Theories, Architectures, and Languages (Hrsg. M. Wooldridge, N.R. Jennings), Springer-Verlag, 1995, Heidelberg, S. 261 - 276.

[Newell 1982] A. Newell: The knowledge level. Artificial Intelligence 18(1), 1982, S. 87 - 127.

[Newell, et al. 1989] A. Newell, P. S. Rosenbloom, J. E. Laird: Symbolic architectures for cognition. In: Foundations of cognitive science (Hrsg. M.I. Posner), MIT Press, Cambridge, MA, 1989.

[Newell 1990] A. Newell: Unified theories of cognition. 1990, Cambridge, MA.

[Nii 1986] H. P. Nii: Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. AI Magazine 7(2), 1986, S. 38 - 53.

[Nilsson 1984] N. J. Nilsson: Shakey the robot. Technical Report 323, Artificial Intelligence Cente, SRI International, Menlo Park, CA, 1984.

[Norman, Long 1995] T. J. Norman, D.P. Long: Goal creation in motivated agents. In: Intelligent Agents: Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages (Hrsg. N.R. Jennings, M.J. Wooldridge), LNAI-890, Springer-Verlag, 1995, S. 277 - 290.

[Norman, Long 1996] T. J. Norman, D. P. Long: Alarms: an implementation of motivated agency. In: Intelligent Agents II (LNAI 1037): Proceedings of the IJCAI-95 Workshop on Agent Theories, Architectures, and Languages (Hrsg. M. J. Wooldridge, J. P. Müller, M. Tambe), LNAI-1037, 1996, S. 219-234.

[Numaoka, Tokoro 1990] C. Numaoka, M. Tokoro: Conversation among situated agents. In: DAIW-90, 1990.

[Nwana 1996] H. S. Nwana: Software agents: An overview. Knowledge Engineering Review 11(3), 1996, S. 205 - 244.

[Pap 1957] A. Pap: Belief and proposition. Philosophy of Science 24, 1957, S. 123 - 136.

[Parunak 1997] H. V. D. Parunak: "Go to the ant": Engineering principles from natural multi-agent systems. Annals of Operations Research, special issue on Aartificial Intelligence and Management Science, 1997.

[Peng, Williams 1994] J. Peng, R. J. Williams: Incremental multi-step Q-learning. In: Eleventh International Conference on Machine Learning, Morgan Kaufmann, San Francisco, 1994, S. 226 - 232.

[Pitt, et al.] J. V. Pitt, M. T. Anderton, R. J. Cunningham: Normalized interactions between autonomous agents - a case study in inter-organizational project management. Proceedings of the International Workshop on the Design of Cooperative Systems, Juan-Les-Pins, Frankreich, 1995. Erhältlich als Esprit Project GOAL (Esprit 6283)

Report, Department of Computing, Imperial College of Science, Technology, and Mecicine, London.

[Pollack, Ringuette 1990] M. Pollack, M. Ringuette: Introducing the Tileworld: Experimentally evaluating agent architectures. In: Ninth National Conference on Artificial Intelligence (AAAI-90), Boston, MA, 1990, S. 183 - 189.

[Pollack 1992] M. E. Pollack: The uses of plans. Artificial Intelligence 57(1), 1992, S. 43 - 68.

[Pollack, et al. 1994] M. E. Pollack, D. Joslin, A. Nunes, S. Ur, E. Ephrati: Experimental investigation of an agent commitment strategy. Technical Report 94-31, Univ. of Pittsburgh, Dept. of Computer Science, 1994.

[Quinlan 1983] J. R. Quinlan: Learning efficient classification procedures and their applications to chess end games. In: Machine Learning - An Artificial Intelligence Approach (Hrsg. C. Michalski Mitchell), Morgan Kaufmann, 1983, Palo Alto, CA, S. 463 - 482.

[Rao, Georgeff 1990] A. S. Rao, M. P. Georgeff: Intelligent real-time network management. In: Specialized Conference on AI, Telecommunications, and Computer Systems , 1990.

[Rao, Georgeff 1991a] A. S. Rao, M. P. Georgeff: Asymmetry thesis and side-effect problems in linear time and branching time intention logics. In: Twelfth International Joint Conference on AI (IJCAI-91), Sydney, 1991, S. 498 - 504.

[Rao, Georgeff 1991b] A. S. Rao, M. P. Georgeff: Modeling rational agents within a BDI-architecture. In: Knowledge Representation and Resaoning (KR&R-91) (Hrsg. E. S. R. Fikes), Morgan Kaufman Publishers, San Mateo, CA, 1991, S. 473 - 484.

[Rao, Georgeff 1992] A. S. Rao, M. P. Georgeff: An abstract architecture for rational agents. In: Knowledge Representation (KR-92), 1992, S. 439 - 449.

[Rao, Georgeff 1995] A. S. Rao, M. P. Georgeff: BDI-agents: From theory to practice. In: First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, 1995.

[Rao 1996] A. S. Rao: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (Hrsg. W. Van de Velde, J.W. Perram), Springer-Verlag, 1996, S. 42 - 55.

[Reed 1998] C. Reed: Dialogue frames in agent communication. In: ICMAS ´98 - Third International Conference on Multi-Agent Systems (Hrsg. Y. Demazeau), IEEE Computer Society, Paris, 1998, S. 246 - 253.

[Reiter 1980] R. Reiter: A logic for default reasoning. Artificial Intelligence 13, 1980, S. 81 - 132.

[Reticular 1998] Reticular: AgentBuilder®: An integrated toolkit for constructing intelligent software agents. White Paper, Revision 1.2, Reticular Systems, Inc., San Diego CA, 1998. http://www.agentbuilder.com.

[Rhodes, Starner 1996] B. Rhodes, T. Starner: The remembrance agent: a continuously running automated information retrieval system. In: Proceedings of The First International Conference on The Practical Application of Intelligent Agents and Multi Agent Technology (PAAM '96), London, 1996, S. 487 - 495.

[Rosenschein, Genesereth 1985] J. S. Rosenschein, M. R. Genesereth: Deals among rational agents. IJCAI-85, 1985, S. 91 - 99.

[Rosenschein 1986] S. Rosenschein: Rational interaction: Cooperation among intelligent agents. PhD, Universität Stanford University, 1986.

[Sacerdoti 1974] E. Sacerdoti: Planning in a hierarchy of abstraction spaces. Artificial Intelligence 5, 1974, S. 115 - 135.

[Sandholm 1993] T. W. Sandholm: An implementation of the contract net protocol based on marginal cost calculations. In: Eleventh National Conference on Artificial Intelligence (AAAI-93), Washington, D.C., 1993, S. 256 - 262.

[Sandholm, Lesser 1997] T. Sandholm, V. Lesser: Coalitions among computationally bounded agents. Artificial Intelligence 94(1), 1997, S. 99 - 137.

[Schneiderman 1983] B. Schneiderman: Direct manipulation: a step beyond programming languages. IEEE Computer, Vol. 16(8), 1983, S. 57 - 59.

[Searle 1969] J. R. Searle: Speech Acts. 1969.

[Shoham, McDermott 1988] Y. Shoham, D. McDermott: Problems in formal temporal reasoning. Artificial Intelligence 36, 1988, S. 49 - 61.

[Shoham 1990] Y. Shoham: Agent-oriented programming. Technical Report, STAN-CS-1335-90, Stanford University, Computer Science Department, 1990.

[Shoham 1993] Y. Shoham: Agent oriented programming. Artifical Intelligence 60(1), 1993, S. 51 - 92.

[Simon 1957] H. Simon: Models of Man. New York 1957.

[Simon 1962] H. Simon:The architecture of complexity. Proceedings of the American Philosophical Society 26, 1962, S. 467 - 482.

[Simon 1983] H. Simon: Why should machines learn? In: Machine Learning - An Artificial Intelligence Approach (Hrsg. Michalski, Carbonell, Mitchell), Morgan Kaufmann, 1983, Palo Alto, CA, S. 25 - 38.

[Simon 1991] H. Simon: Cognitive architectures in a rational analysis: Comment. In: Architectures for Intelligence (Hrsg. K. VanLehn), Lawrence Erlbaum Associates, 1991, Hillsdale, N.J., S. 25 - 39.

[Singh 1991] M. P. Singh: Towards a formal theory of communication for multi-agent systems. In: Twelfth International Joint Conference on AI (IJCAI-91), Sydney, 1991, S. 69 - 74.

[Singh 1994] M. P. Singh: Multiagent systems: A theoretical framework for intentions, know-how, and communications. Heidelberg 1994.

[Singh, et al. 1998] M. P. Singh, A. S. Rao, M. P. Georgeff: Formal methods in DAI: Logic-based representation and reasoning. In: Introduction to Distributed Artificial Intelligence (Hrsg. G. Weiss), MIT Press, 1998 in Druck.

[Smith 1980] R. G. Smith: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Trans. Comput. 29(12), 1980, S. 1104 - 1113.

[Smith, et al. 1994] D. Smith, A. Cypher, J. Spohrer: Kidsim: Programming agents without a programming language. Commun. ACM, 1994, 37(7), S. 54 - 67.

[Struth 1994] G. Struth: Philosophical logics - a survey and a bibliography. Research Report RR-94-17, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), 1994.

[Sycara 1988] K. Sycara: Resolving goal-conflicts via negotiation. In: AAAI-88, 1988, S. 245 - 250.

[Sycara, et al. 1990] K. Sycara, S. Roth, N. Sadeh, M. Fox: An investigation into distributed constraint-directed factory scheduling. In: Sixth Conference on Artificial Intelligence Applications, IEEE Comput. Soc. Press, Santa Barbara, CA, USA, 1990, S. 94 - 100.

[Thomas 1994] S. R. Thomas: The PLACA agent programming language. In: Intelligent Agents: ECAI-94 Workshop on Agent Theories, Architectures, and Languages (Hrsg. M. J. Wooldridge, N. R. Jennings), 1994, S. 355 - 370.

[Valiant 1984] L. G. Valiant: A theory of the learnable. Communications of the ACM 27(11), 1984, S. 1134 - 1142.

[Veloso, et al. 1995] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, J. Blythe: Integrating planning an learning: The PRODIGY architecture. Journal of Experimental and Theoretical Artificial Intelligence, 7(1), 1995.

[Vere, Bickmore 1990] S. Vere, T. Bickmore: A basic agent. Computational Intelligence 6(1), 1990, S. 41 - 60.

[Wagner 1997] G. Wagner: Artificial agents and logic programming. In: ICLP'97 post-conference workshop Logic Programming and Multiagent Systems, 1997. http://www.informatik.uni-leipzig.de/~gwagner.

[Walker, Jordan 1995] M. A. Walker, P. W. Jordan: Design-World: A testbed of communicative action and resource limits. ACM SIGART 6(2), 1995.

[Watkins, Dayan 1992] C. H. Watkins, P. Dayan: Q-learning. Machine Learning 8(3), 1992, S. 279 - 292.

[Weerasooriya, et al. 1995] D. Weerasooriya, A. S. Rao, K. Ramamohanarao: Design of a concurrent agent-oriented language. In: Intelligent agents: theories, architectures, and languages. Lecture notes in artificial intelligence LNAI 890 (Hrsg. M. Wooldridge, N.R. Jennings), Springer Verlag, Amsterdam, 1995.

[Weld 1994] D. Weld: An introduction to least commitment planning. AI Magazine 15(4), 1994, S. 27 - 61.

[Wellman 1995] M. P. Wellman: The economic approach to Artificial Intelligence. ACM Computing Surveys, Symposium on Artificial Intelligence 27(3), 1995.

[Werner 1987] E. Werner: Toward a theory of communication and cooperation for multiagent planning. In: Theoretical Aspects of Reasoning about Knowledge (Hrsg. M. Y. Vardi), Morgan Kaufmann, 1987, S. 129 - 143.

[Wilkins 1985] D. E. Wilkins: Recovering from execution errors in SIPE. Computational Intelligence 1, 1985, S. 33 - 45.

[Wooldridge, Jennings 1995a] M. Wooldridge, N. R. Jennings: Intelligent agents. 1995.

[Wooldridge, Jennings 1995b] M. Wooldridge, N. R. Jennings: Intelligent agents: theory and practice. The Knowledge Engineering Review 10(2), 1995, S. 115 - 152.

[Wooldridge, Jennings 1998] M. J. Wooldridge, N. R. Jennings: Pitfalls of agent-oriented development. In: 2nd Int Conf on Autonomous Agents, Minneapolis, 1998.

[Wooldridge, et al 1999] M. J. Wooldridge, N. R. Jennings, D. Kinny: A methodology for agent-oriented analysis and design. Erscheint in: Third International Conference on Autonomous Agents (Agents ´99), Seattle, WA, 1999.

[Wright 1951] G. H. v. Wright: An essay in modal logic. 1951.

[Yang, Duddy 1996] :Z. Yang, K. Duddy: CORBA: a platform for distributed object computing. Operating Systems Review 30(2), 1996, S. 4 - 31.

[Zilberstein, Russell 1995] S. Zilberstein, S. Russell: Approximate reasoning using anytime algorithms. In: Imprecise and Approximate Computation (Hrsg. S. Natarajan), Kluwer Academic Publishers, 1995.

[Zilberstein, Russell 1996] S. Zilberstein, S. Russell: Optimal composition of real-time systems. Artificial Intelligence 82, 1996, S. 181 - 213.

Lebenslauf

Name	Stefan Fricke
Geburtsdatum	31. 5. 1964
Familienstand	ledig
Wohnort	Fidicinstr. 17 10965 Berlin
Beruf	Diplom - Informatiker
Eltern	Reinhard Fricke Elsbeth Fricke, geb. Winkler
Schulische Ausbildung	**8/1970 - 7/1974 :** Grundschule in der Hasenburg, Lüneburg **8/1974 - 6/1983 :** Wilhelm-Raabe-Gymnasium in Lüneburg mit Abschluß Abitur
Berufliche Ausbildung	**06/198 3 - 09/1983 :** Berufspraktikum im kommunalen Rechenzentrum Lüneburg **9/1983 - 1/1992 :** Studium der Informatik an der TU Berlin mit Abschluß Diplom
Berufliche Laufbahn	**5/1989 - 2/1992 :** studentische Hilfskraft an der TU Berlin am Institut für Quantitative Methoden **3/1992 - 12/1994 :** Tätigkeit als wissenschaftlicher Mitarbeiter an der TU Berlin im Institut für Quantitative Methoden im TeleCAS-Projekt **1/1995 - 6/1995 :** Privatarbeitsvertrag bei Prof. Dr. H. Krallmann am gleichen Institut **seit 7/1995 :** wissenschaftlicher Mitarbeiter am gleichen Institut