Universität-Gesamthochschule Paderborn

Fachbereich 17 Mathematik - Informatik

Diplomarbeit

Entwurf und prototypische Realisierung einer Spezifikationssprache für intelligente Software-Agenten

Stephan Flake

Schulstr. 13, 33102 Paderborn
Matrikel-Nr. 3994903

vorgelegt bei:
Prof. Dr. Franz-J. Rammig
Paderborn, 30. April 1999

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbstständig und nur unter Zuhilfenahme der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle wörtlich oder inhaltlich zitierten Stellen sind als solche kenntlich gemacht.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Paderborn, den 30. April 1999

(Stephan Flake)

Vorwort

Seit dem 1. August 1998 gelten die neuen Rechtschreibregeln. In meiner Diplomarbeit habe ich mich nach dieser Neuregelung gerichtet, weil ich viele der vorgenommenen Änderungen für angemessen halte, ohne jedoch ein entschiedener Befürworter der neuen Regelung zu sein. Mir werden trotz großer Sorgfalt einige Fehler bei der Anwendung der neuen Rechtschreibregeln unterlaufen sein, für die ich den Leser an dieser Stelle um Nachsicht bitte.

Bedanken möchte ich mich besonders bei meinen Betreuern Christian Geiger, Georg Lehrenfeld und Volker Paelke, die mir mit zahlreichen Vorschlägen bei der Erstellung dieser Diplomarbeit behilflich waren.

Marcus Huber von der Firma Intelligent Reasoning Systems in Kalifornien hat mir Zusatzinformationen zu seiner Agentensprache JAM gegeben, die mir sehr bei der Entwicklung der Spezifikationssprache CASA geholfen haben. Mein Dank geht auch an Dr. Bernhard Bauer und Dr. Donald Steiner von der Siemens AG, die mir das MECCA-System und nützliche Seminarunterlagen zur Verfügung gestellt und mich mit weiteren Informationen über FIPA ACL versorgt haben.

Ich bedanke mich bei Mark Meierjohann, Christian Geiger und meinem Bruder Frank dafür, dass sie mich auf zahlreiche Tippfehler sowie schwer verständliche Passagen hingewiesen haben, die ich ohne ihre Hilfe nicht alle gefunden und verbessert hätte.

April 1999 Stephan Flake

Inhaltsverzeichnis

KAPITEL 1 Einleitung

An rechnergestützte Systeme werden immer höhere Anforderungen gestellt, die es zunehmend schwieriger machen, Systeme als eine einzige große Einheit anzusehen und durch eine zentrale Kontrollinstanz zu verwalten. Immer öfter ist es notwendig, Produktionsprozesse oder andere logistische Aufgaben zwischen vielen beteiligten Stellen zu koordinieren. Darüber hinaus basieren die zu bewältigenden Aufgaben oft auf komplexen Problemen, wie z.B. das Scheduling beim Transport von Gütern mit begrenzten Transportmöglichkeiten, sei es innerhalb einer Lagerhalle oder auch weltweit über verschiedene Verkehrswege.

Die in solchen Systemen inhärent auf verschiedenen Komponenten vorliegenden verteilten Informationen sowie die Komplexität der Aufgaben machen es notwendig, neue Ansätze zur Modellierung und Realisierung dieser Systeme zu untersuchen. Dabei geht es weniger um die Berechnung optimaler Lösungen, sondern vielmehr um robuste und flexible Mechanismen zur Behandlung der auftretenden Probleme.

Agentenbasierte Technologie

Software-Agenten und Multi-Agentensysteme repräsentieren einen vielversprechenden Ansatz zur Analyse, Modellierung und Implementierung komplexer Systeme. Obwohl der Agentenbegriff schon vor über 30 Jahren im Bereich der Künstlichen Intelligenz (KI) und später auch im Teilbereich der Verteilten Künstlichen Intelligenz (VKI) verwendet wurde, hat sich erst zu Beginn der 90er-Jahre ein breites Interesse an der Agententechnologie entwickelt; leistungsfähige Computer und Rechnernetzwerke haben seitdem die Agententechnologie realisierbar gemacht. Mittlerweile gibt es eine ganze Reihe von Anwendungen, die mit Agentenkonzepten entwickelt wurden und erfolgreich eingesetzt werden. Wohl das bekannteste eingesetzte System auf Basis der Agententechnologie ist die Luftverkehrsüberwachung am Flughafen von Sydney.

Der Einsatz von Software-Agenten ist besonders zur Handhabung verteilter Informationen in vernetzten Systemen geeignet. Mobile Agenten können zum Beispiel im Internet nach Informationen suchen und nach ihrer Recherche zum Ausgangsrechner zurückkehren. Auch in anderen Bereichen können Agenten sinnvoll eingesetzt werden, beispielsweise als adaptive Benutzerschnittstellen zur besseren Interaktion zwischen Mensch und Maschine.

In Multi-Agentensystemen wird das Zusammenspiel interagierender Agenten betrachtet. Im Gegensatz zu den traditionellen Client/Server-Lösungen liegt in Multi-Agentensystemen keine globale Steuerung vor, sondern die einzelnen Agenten des Systems agieren mit „Armlängen-Reichweite“, d.h. sie interagieren nur mit den Komponenten bzw. Agenten, die für die Bearbeitung ihrer Aufgaben notwendig sind. Der Einsatz eines Multi-Agentensystems ist besonders in den Anwendungsfeldern sinnvoll, in denen mehrere Lösungsalternativen für Aufgaben existieren und mehrere Komponenten zur Problemlösung zur Auswahl stehen. Bei solchen Anwendungen haben Multi-Agentensysteme den Vorteil, dass sie verteiltes und paralleles Problemlösen mit Elementen zur Interaktion wie Kooperation, Koordination und Verhandlungsführung verbinden. Mit ihren Möglichkeiten zur Agenten-Interaktion sowie weiteren technischen Besonderheiten wie asynchroner Arbeitsweise oder einfacher Erweiterbarkeit gelten Multi-Agentensysteme als sehr flexibel und werden als zukunftsweisende Technologie z.B. im Workflow-Management oder im Computer Integrated Manufacturing (CIM) angesehen.

Doch wann genau ist ein Programm ein Agent? In der realen Welt wird jemand als Agent bezeichnet, der im Auftrag eines Anderen eigenständig handelt. In Anlehnung an das Verhalten eines menschlichen Agenten werden auch bestimmte Erwartungen an Software-Agenten gestellt, die konventionelle Programme nicht vorweisen. In diesem Zusammenhang werden Agenten oft mit Attributen wie „intelligent“ oder „autonom“ charakterisiert, doch diese Eigenschaften sind subjektiver Natur und über ihre Bedeutung für Agenten wird in der Fachwelt immer wieder kontrovers diskutiert. Dies liegt u.a. daran, dass der Begriff des Agenten heute viel umfassender benutzt wird als im ursprünglichen Gebiet der KI. Daher ist es nicht möglich, eine einzelne, allgemein akzeptierte Definition für Software-Agenten anzugeben, vielmehr gibt es inzwischen eine große Anzahl von Begriffsbestimmungen.

Agenten zwischen Theorie und Praxis

Viele Veröffentlichungen auf dem Agentengebiet befassen sich vornehmlich mit theoretischen Grundlagen, z.B. mit Modallogiken zur Beschreibung von Wissen und Meta-Wissen. Andere wiederum beschreiben konkrete Agenten und Multi-Agentensysteme, die zwar praktisch realisierbar sind, aber nur mit Einschränkungen der formalen Grundlagen oder sogar ganz losgelöst von agententheoretischen KI-Konzepten entwickelt worden sind.

Für die vielbeachtete BDI-Agentenarchitektur wurde diese Lücke zwischen Theorie und Praxis 1996 von A. Rao geschlossen. Interessanterweise ist er von einem implementierten System, dem Procedural Reasoning System (PRS), ausgegangen und hat hierfür nachträglich die eingeschränkte logische Programmiersprache AgentSpeak(L) und eine operationale Semantik entwickelt, die die wesentlichen Elemente von PRS formalisieren.

Diese formalen Grundlagen sind von C. Geiger aufgegriffen und zu der Agentenspezifikationssprache AgentGHC erweitert worden, in die u.a. auch bewachte Horn-Klauseln (Guarded Horn Clauses, GHC) aus *parallelen* logischen Programmiersprachen eingeflossen sind. Da es zu AgentSpeak(L) ein lauffähiges System gibt, besteht die berechtigte Hoffnung, dass es auch für das erweiterte Modell AgentGHC möglich ist, ein lauffähiges Programmpaket zu erstellen.

Ein neuer Ansatz für ein praktisches Agentenmodell

Aus den obigen Überlegungen heraus habe ich – nach einigen Verfeinerungen des AgentGHC-Modells – eine leicht verständliche Spezifikationssprache für Agenten entworfen. Für diese Sprache habe ich ein Programmpaket implementiert, das Spezifikationen aus Dateien einlesen kann und daraus Software-Agenten generiert, die sich an das formale Agentenmodell halten. Darüber hinaus habe ich eine Grundlage dafür geschaffen, solche Agenten zu Multi-Agentensystemen zu verbinden, indem ich eine erweiterbare Schnittstelle implementiert habe. Die Lauffähigkeit der Agenten wird anhand eines Beispiels aus dem Bereich rechnerunterstützter Produktion aufgezeigt.

Zur weiteren Beschreibung des zugrunde liegenden formalen Agentenmodells unterscheide ich zwischen einer Mikro- und einer Makrosicht. Die Mikroaspekte betreffen die interne Struktur eines einzelnen Agenten und bilden den Hauptteil meiner Arbeit, während sich die Makrosicht mit agentenübergreifenden Aufgaben befasst, insbesondere mit der Nachrichtenübermittlung zwischen Agenten und dem Zusammenschluss zu Multi-Agentensystemen.

Mikrosicht. Die Mikrosicht von AgentGHC basiert auf der **BDI**-Agentenarchitektur von Rao, die Agenten durch interne Zustände wie Überzeugungen (**B**eliefs), Verlangen (**D**esires) und Absichten (**I**ntentions) beschreibt. Neben einem abstrakten Interpreter für die operationale Semantik wird eine Notation zur Beschreibung des internen Agentenzustands auf Basis von AgentSpeak(L) angegeben und semantisch eindeutig festgelegt.

Zur einfacheren Beschreibung des internen Zustands eines AgentGHC-Agenten habe ich die Spezifikationssprache CASA entwickelt, die auf Elementen der Agentensprache JAM von M. Huber basiert. Mit den Sprachelementen von CASA wird u.a. der Ausgangszustand eines Agenten spezifiziert, z.B. initiale Ziele und Überzeugungen, aber auch Strategien zur Verfolgung von Zielen. Darüber hinaus habe ich im zugehörigen CASA-Interpretermodell den abstrakten Interpreter aus AgentGHC um die Behandlung verschiedener Ereignistypen sowie um Komponenten zur Überprüfung komplexer Strategien erweitert.

Ein Prototyp zum Generieren von CASA-Agenten ist von mir in der objektorientierten Programmiersprache Java entwickelt worden. Dies ist das Resultat des Versuchs, das formale und semantisch fundierte Agentenmodell AgentGHC in ein Programmpaket umzusetzen, ohne wesentliche Einschränkungen gegenüber dem theoretischen Grundmodell machen zu müssen. Bei der Implementierung von CASA-Sprachelementen habe ich Java-Klassen der Agentensprache JAM wiederverwenden können, musste aber an vielen Stellen auch Anpassungen vornehmen und habe weitere Sprachelemente sowie den CASA-Interpreter komplett neu implementiert.

Makrosicht. In Multi-Agentensystemen interagieren die Agenten üblicherweise durch den Austausch von Nachrichten. Die beteiligten Agenten müssen über eine gemeinsame Kommunikationssprache verfügen, damit sie den Inhalt von Nachrichten untereinander verstehen können. Im Rahmen meiner Diplomarbeit habe die beiden Agenten-Kommunikationssprachen KQML (Knowledge Query and Manipulation Language) und FIPA ACL (Foundation for Intelligent Physical Agents - Agent Communication Language) untersucht. Von diesen beiden Sprachen ist KQML die ältere und

weiter verbreitete, während FIPA ACL noch relativ neu ist und auf einer semantisch fundierten Grundlage aufbaut.

Mittlerweile gibt es eine ganze Reihe von Systemen, die eine hilfreiche Infrastruktur für Multi-Agentensysteme anbieten. Diese Frameworks bieten verschiedene Dienste für die Verwaltung von Agenten an, z.B. Verzeichnisdienste, bei denen Agenten ihre Dienste veröffentlichen können oder nach der Existenz und dem aktuellen Aufenthaltsort anderer Agenten gefragt werden kann. Ich habe die Fähigkeiten einiger dieser Frameworks untersucht und mithilfe eines dieser Systeme ein Anwendungsbeispiel implementiert.

Ein Anwendungsbeispiel

Als Beispiel für die Realisierung eines Multi-Agentensystems wird eine Aufgabe aus dem Bereich CIM in der Automobilproduktion aufgegriffen: Eine Lackieranlage wird um einen Farbsortierspeicher mit einer gewissen Anzahl an Gassen erweitert, in denen zu lackierende Karosserien so zwischengelagert werden, dass möglichst lange Blöcke von Karosserien mit derselben zu lackierende Farbe gebildet werden. Durch eine entsprechende Auslagerung langer Blöcke von Karosserien sollen in den Lackieranlagen die Farbwechsel reduziert und damit die Umrüstzeiten und -kosten gesenkt werden.

Für die Implementierung des Anwendungsbeispiels habe ich mich dazu entschlossen, als Kommunikationssprache FIPA ACL und als zugrunde liegende Infrastruktur MECCA (Multi-agent Environment for Constructing Cooperative Applications) der Siemens AG zu verwenden.

Zur Anbindung an MECCA habe ich eine individuell erweiterbare Schnittstelle für CASA-Agenten entwickelt, die ausgehende Nachrichten in FIPA ACL umwandelt und verschickt. Umgekehrt werden eingehende FIPA ACL-Nachrichten von der Schnittstelle als Ereignisse an den zugeordneten CASA-Agenten weitergeleitet.

Überblick

Diese Diplomarbeit ist in die folgenden Kapitel unterteilt:

Kapitel 2 beschreibt den Aufbau und eine Modellierung des Farbsortierspeichers. Darüber hinaus wird auch erklärt, warum dieses System für eine agentenbasierte Lösung interessant ist.

Kapitel 3 geht auf die agententheoretischen Grundlagen ein. Insbesondere werden verschiedene Definitionen und Architekturen für Agenten vorgestellt, bevor das Agentenmodell AgentGHC, das die Basis des weiteren Vorgehens bildet, genauer beschrieben wird.

Kapitel 4 behandelt Agenten-Kommunikationssprachen, u.a. werden die beiden populären Sprachen KQML und FIPA ACL gegenübergestellt.

Kapitel 5 schildert die Bedeutung von Kooperation für Multi-Agentensysteme. Einige der existierenden Frameworks für Multi-Agentensysteme werden vorgestellt und ihre Fähigkeiten miteinander verglichen.

Kapitel 6 beschreibt die von mir entwickelte Spezifikationssprache CASA und führt die zugehörige operationale Semantik auf, indem die einzelnen Komponenten der zugrunde liegenden Agentenarchitektur vorgestellt werden und ihr Zusammenspiel in einem abstrakten Interpreter formal festgelegt wird.

Kapitel 7 gibt ausgewählte Aspekte der Implementierung von CASA wieder. Es wird dargestellt, warum sich gerade die Programmiersprache Java für die Realisierung von CASA eignet. Zusätzlich zur Beschreibung der Implementierung einzelner agenteninterner Komponenten wird der Aufbau einer individuell erweiterbaren Schnittstelle für die Anbindung von CASA-Agenten an ein Multi-Agentensystem erläutert.

Kapitel 8 belegt die Anwendbarkeit von CASA zur Spezifikation von Agenten im Rahmen des Anwendungsbeispiels Farbsortierspeicher. Angegeben wird der Ablauf der Einlagerung von Karosserien in den Sortiergassen durch Einlagerungsroboter analog zu dem in Kapitel 2 entwickelten Modell.

Kapitel 9 fasst die Ergebnisse der Diplomarbeit zusammen und schildert bereits vorhandene und noch mögliche Erweiterungen für CASA.

KAPITEL 2

Der Farbsortierspeicher

Das Ausgangsproblem meiner Arbeit bildet die Modellierung eines komplexen Systems aus dem Bereich rechnerunterstützter Fertigung: eine erweiterte Lackieranlage für Automobilkarosserien. Diese Lackieranlage soll als zusätzliche Komponente einen Sortierspeicher besitzen, in dem Rohkarosserien vor der Lackierung zwischengelagert werden, um später möglichst viele Lackieraufträge gleicher Farbe direkt hintereinander abarbeiten zu können. Auf diese Weise sollen Zeit und Kosten für die Reinigung der Lackierspritzen verringert werden.
In diesem Kapitel wird zunächst der Aufbau der Lackieranlage beschrieben. Dann wird ein Lösungsansatz für die Modellierung vorgestellt, der das weitere Vorgehen in meiner Arbeit entscheidend beeinflusst hat; die einzelnen Komponenten der Lackieranlage werden als eigenständige Einheiten betrachtet und dazu mit dem nötigen Wissen über ihre Umwelt ausgestattet. Jede Komponente verfolgt eigene Ziele und benutzt bestimmte Strategien, um diese Ziele zu erreichen. Bei Bedarf kann sich jede Komponente auch mit anderen Komponenten durch Nachrichtenaustausch verständigen.

2.1. Ausgangsproblem

In einer Lackieranlage werden Fahrzeugkarosserien mit Transportrobotern von einem Depot zu Lackierlinien gebracht. Ein Lackierauftrag setzt sich aus einem Karosserietyp und einer Farbe zusammen, sodass im Folgenden mit einem Auftrag immer ein Tupel der Form (Karosserietyp, Farbe) gemeint ist.

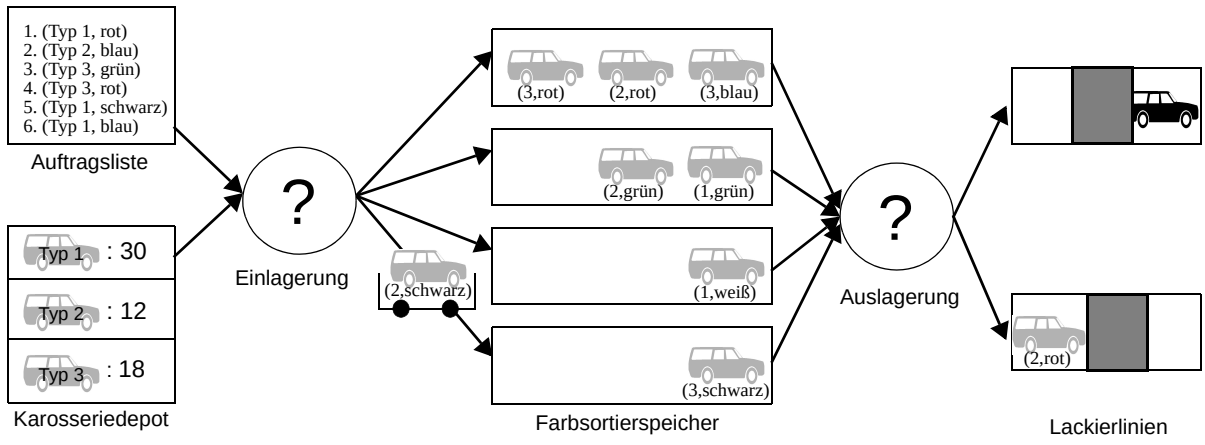
Jede Lackierlinie bearbeitet Aufträge in der Reihenfolge, in der sie von den Robotern angeliefert werden. Durch unterschiedliche Farben in aufeinanderfolgenden Aufträgen wird ein Farbwechsel in den Spritzen der Lackieranlage notwendig. Die Farbwechsel sind ein wichtiger Zeit- und Kostenfaktor der Lackieranlage; bei jedem Farbwechsel müssen z.B. die Lackierspritzen gesäubert werden, sodass sich die Produktion um die hierfür benötigte Zeit verzögert und zusätzliche Reinigungskosten entstehen. Diese Kosten liegen für eine Lackieranlage in einer Größenordnung von mehreren hunderttausend US-Dollar pro Jahr [Fritsche 95].

Ein vorgelagerter Farbsortierspeicher soll helfen diese sogenannten Umrüstkosten, die bei jedem Farbwechsel entstehen, zu senken. Er besteht aus mehreren Gassen, auch Sortierlinien genannt, mit deren Hilfe eine Vorsortierung von Lackieraufträgen gleicher Farbe erfolgen kann. Bei der Auslagerung aus dem Farbsortierspeicher können dann mehrere Aufträge gleicher Farbe hintereinander zu den Lackierlinien gebracht werden.

2.2. Beschreibung

In einer Lackieranlage durchlaufen Karosserien mehrere Bearbeitungsstufen, in denen sie u.a. Tauchbecken, Spritzkabinen, Trockneranlagen und Kontrollbereiche durchlaufen, bevor sie in einer Spritzkabine mit ihrem Decklack versehen werden. Der näher betrachtete Teil der Lackieranlage besteht aus einer Reihe von Komponenten mit den folgenden Aufgaben (vgl. Abbildung 1, aus: [Geiger 98]):

Abbildung 1 Lackieranlage mit Farbsortierspeicher



Die Auftragskomponente verwaltet eine Auftragsliste und vergibt Aufträge an Einlagerungsroboter als Tupel (Karosserietyp, Farbe). Das Karosseriedepot hält eine gewisse Zahl an Karosserien verschiedener Typen bereit und gibt auf Anfrage eine Karosserie des gewünschten Typs an einen Einlagerungsroboter ab. Die Einlagerungsroboter

transportieren Karosserien zu den Sortierlinien im Farbsortierspeicher. Die Sortierlinien nehmen Karosserien auf und geben sie später an die Auslagerungsroboter ab. Diese transportieren die Aufträge schließlich zu den Lackierlinien. Mit der Anlieferung an einer Lackierlinie soll ein Auftrag innerhalb der Lackieranlage als vollständig bearbeitet gelten.

Voraussetzungen an den Farbsortierspeicher:

1. Es wird davon ausgegangen, dass immer genügend Karosserien aller Typen im Depot bereit stehen. Wenn eine bestimmte Anzahl von Karosserien unterschritten wird, gibt der Depotagent einen Warnhinweis an den Auftragsagenten ab.
2. Ein- bzw. Auslagerungsroboter können nicht unbedingt alle Karosserietypen transportieren; je nach Bauart kann ein Roboter eine, mehrere oder alle Karosserietypen aufnehmen. Der Auftragsagent hat Kenntnis über die Roboterbauarten und welche Karosserietypen sie befördern können.
3. Die Sortierlinien haben eine bestimmte Kapazität, die nicht überschritten werden darf.
4. Die Anzahl der Farben ist größer als die Anzahl der Sortierlinien. Ansonsten kann man einfach jeder Farbe (mindestens) eine Sortierlinie fest zuordnen und Aufträge entsprechend ihrer Farbe in den fest zugeordneten Sortierlinien einlagern.
5. Die Anzahl der Sortierlinien ist größer als die Anzahl der Lackierlinien. Ansonsten kann man jeder Sortierlinie einfach (mindestens) eine Lackierlinie zuordnen und die Aufträge an die Lackierlinie direkt weiterleiten.

Bedingungen zum laufenden Betrieb:

1. Die Lackierlinien können eventuell nicht in allen Farben lackieren, z.B. weil eine Farbe gerade ausgegangen ist.
2. Jede Lackierlinie hat eine bestimmte Ausschussrate, die sich dynamisch ändern kann, z.B. durch Abnutzung oder Reparatur. Solche Änderungen können Auswirkungen auf die Verteilung der Karosserien an die Lackierlinien haben.
3. Lackierlinien können ausfallen. Das bedeutet, dass Roboter eventuell umgelenkt oder sogar Transporte ganz rückgängig gemacht werden müssen.
4. Die „Beliebtheit“ der einzelnen Farben ist bekannt. Sie kann sich z.B. durch den Prozentanteil an der bisherigen Gesamtproduktion ausdrücken. Dieser Anteil verändert sich natürlich durch den laufenden Betrieb ständig. Die Daten über die Beliebtheit können dazu dienen, für sehr beliebte Farben längere Blöcke in den Sortierlinien zu bilden als für weniger beliebte.
5. Jeder Auftrag soll innerhalb einer bestimmten Zeitspanne bearbeitet sein, z.B. muss auch ein Auftrag mit einer ausgefallenen Farbe irgendwann bearbeitet werden.
6. Dringende Aufträge sollen bevorzugt bearbeitet werden. Dafür muss gegebenenfalls sogar ein laufender Transport unterbrochen werden. Der Sortierspeicher wird für solche Aufträge umgangen, und der Auftrag gelangt auf direktem Wege zu einer Sortierlinie.

Die angesprochenen Bedingungen führen zu widersprüchlichen Anforderungen im Farbsortierspeicher, die es unmöglich machen, eine optimale Vorsortierung in ange-

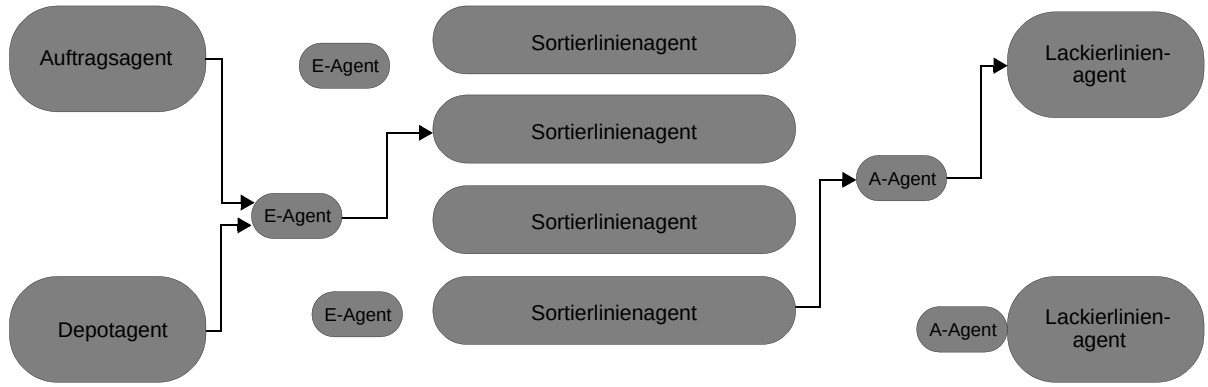
messener Zeit zu berechnen. Um das Hauptziel – die Reduzierung der Farbwechsel – zu verfolgen, müssen in den Sortierlinien möglichst lange Blöcke von Aufträgen gleicher Farbe gebildet werden. Dies steht im Widerspruch zu der Bedingung, dass *jeder* Auftrag möglichst schnell bzw. innerhalb einer bestimmten Frist bearbeitet sein muss. Es wird bei dem weiteren Vorgehen daher keine optimale Lösung für die Zwischenlagerung im Farbsortierspeicher gesucht, sondern vielmehr versucht, möglichst flexibel und robust auf die auftretenden Sonderfälle, wie z.B. wichtige Aufträge, zu reagieren.

2.3. Modellierung

Wie auch in der Systemmodellierung von C. Geiger wird ein agentenbasierter Ansatz zur Erstellung eines Modells für den Farbsortierspeicher gewählt [Geiger 98]. Auf den nicht eindeutig definierbaren Begriff des Agenten gehe ich noch gesondert im Kapitel 3 ein, an dieser Stelle mag vorübergehend die Assoziation eines Agenten als eigenständiges Objekt hilfreich sein. Im Gegensatz zur Modellierung von C. Geiger werden die Ein- und Auslagerungsroboter bei diesem Ansatz aber nicht zentral von einem Roboterlager aus verwaltet, sondern für jeden Roboter soll eine zugeordnete Steuerungseinheit als selbstständiger Teil der Produktionsanlage agieren, d.h. ein Software-Agent als Robotersteuerung nimmt Kontakt zu den anderen Komponenten der Anlage auf und verfolgt damit das Ziel, dass der Roboter möglichst immer mit dem Transport von Karosserien beschäftigt ist.

Abbildung 2 zeigt das von mir erstellte agentenbasierte Modell des Farbsortierspeichers. Es besteht aus $2 + r_e + s + r_a + l$ Agenten (oder: eigenständigen Objekten), wobei r_e die Anzahl der Einlagerungsroboter (E-Agenten), s die Anzahl der Sortierlinien, r_a die Anzahl der Auslagerungsroboter (A-Agenten) und l die Anzahl der Lackierlinien ist.

Abbildung 2 Agentenbasierte Modellierung des Farbsortierspeichers



Im nächsten Schritt versuche ich eine sinnvolle Strukturierung dieser Agenten vorzunehmen. Schon bei der informellen Beschreibung der Agenten fielen die Begriffe *Ziel* und *Wissen* und diese Begriffe werden nun auch zur strukturellen Beschreibung übernommen. Zusätzlich benötigen Agenten auch noch *Strategien*, um ihre Ziele zu erreichen.

Während Wissen, Ziele und Strategien die interne Struktur der Agenten wiedergeben, benötigt man für die Beschreibung des Informationsflusses zwischen den Agenten eine Schnittstellendefinition. Zum einen bieten Agenten *Dienste* für andere Agenten an, zum anderen führen sie selbstständig *Aktionen* aus, bei denen mit der Umgebung Verbindung aufgenommen wird. Es gibt auch Aktionen, die sich auf den Agenten selbst beschränken, jedoch soll auf diese hier nicht eingegangen werden.

Die Struktur für die Agenten im Farbsortierspeicher lässt sich anhand der o.g. Begriffe – zunächst noch informell – wie in Abbildung 3 darstellen. Die Angaben beschränken sich dabei zwar nur auf die wichtigsten Daten, doch man erkennt daran, dass es möglich ist, alle Agenten des Farbsortierspeichers auf diese Weise einheitlich zu strukturieren. Dieser vielversprechende Ansatz ist der Anlass dafür, eine neue Spezifikationssprache für solche Agenten zu entwickeln. Dabei soll untersucht werden, ob der Einsatz dieser Sprache die Entwicklung komplexer Systeme wie z.B. den Farbsortierspeicher vereinfacht.

Abbildung 3 Agentenstruktur im Farbsortierspeicher

Auftragsagent

Wissen:
eigener Zustand,
Auftragsliste mit Lieferfristen,
wichtige Aufträge
Ziele:
Abgabe von Aufträgen nach Dringlichkeit
Strategien:
Auswahl des nächsten Auftrags

Dienste:
Abgabe von Aufträgen
Aktionen:
Vergabe von wichtigen Aufträgen

Einlagerungsagent

Wissen:
eigener Zustand,
mögliche Karosserietypen,
aktueller Auftrag
Ziele:
ständiger Transport von Karosserien
Strategien:
Auftrag einholen und bei einer Sortierlinie abliefern

Dienste:
Annahme wichtiger Aufträge
Aktionen:
Auftrag einholen und Karosserie am Depot abholen, Sortierlinie ermitteln, Abliefern bei Sortierlinie, Lackierlinie ermitteln, Abliefern bei Lackierlinie

Auslagerungsagent

Wissen:
eigener Zustand,
mögliche Karosserietypen,
aktueller Auftrag
Ziele:
ständiger Transport von Karosserien
Strategien:
Auftrag einholen und bei einer Lackierlinie abliefern

Dienste:
Annahme von Aufträgen
Aktionen:
Auftrag einholen und Karosserie an Sortierlinie abholen, Lackierlinie ermitteln, Karosserie bei Lackierlinie abliefern

Depotagent

Wissen:
eigener Zustand,
Karosserieanzahl der verschiedenen Typen

Ziele:
ständig Karosserien jedes Typs bereit haben
Strategien:
Beobachten des Bestands und Warnung bei fehlenden Karosserien

Dienste:
Abgabe von Karosserien

Aktionen:
(keine selbst initiierten Aktionen)

Sortierlinienagent

Wissen:
eigener Zustand,
Kapazität,
aktuelle Belegung,
Verlangen nach Farben
Ziele:
möglichst lange Blöcke, nicht ganz leer, nicht ganz voll
Strategien:
berechne Verlangen nach Farben anhand aktueller Belegung

Dienste:
Bekanntgabe des Verlangens nach einer bestimmten Farbe
Aktionen:
Auslagerung eines Blocks

Lackierlinienagent

Wissen:
eigener Zustand,
aktuelle Farbe,
vorhandene Farben,
Verlangen nach Farben
Ziele:
Minimierung der Farbwechsel, ständiger Betrieb
Strategien:
berechne Verlangen nach Farben anhand aktueller Belegung

Dienste:
Bekanntgabe des Verlangens nach einer bestimmten Farbe
Aktionen:
Karosserie lackieren,
Weitertransport

Auf eine genauere Beschreibung der Agenten verzichte ich an dieser Stelle. Weitere Informationen zum Farbsortierspeicher findet man in [Geiger 98]. Dort wird auch eine interaktive 3D-Illustration des Farbsortierspeichers beschrieben.

In den Kapiteln 6 und 7, die sich mit dem Entwurf und der Realisierung der Agenten-Spezifikationssprache CASA beschäftigen, wird der Einlagerungsagent des Öfteren als Beispiel dienen. Die an der Einlagerung im Farbsortierspeicher beteiligten Agenten habe ich mithilfe von CASA spezifiziert und in einem Anwendungsbeispiel zu einem lauffähigen System verbunden. Diese Spezifikationen und die Implementierung der Einlagerung werden in Kapitel 8 behandelt.

KAPITEL 3 Der Agentenbegriff

”In essence, agents exist in a truly multi-dimensional space . . .”

- Hyacinth Nwana

Software-Agenten haben ihren Ursprung bei den Multi-Agentensystemen (MAS), die wiederum einen der drei Hauptzweige der Verteilten Künstlichen Intelligenz (VKI) bilden. Leider kann keine eindeutige Definition für den Begriff des Software-Agenten angegeben werden, aber es gibt zumindest eine weithin akzeptierte gemeinsame Basis von Mindestanforderungen an solche Agenten.
Die semantische Grundlage der Agentenforschung bildet die Agententheorie, in der Formalismen zur Beschreibung von Agenten entwickelt werden.
Auf einer höheren Ebene beschreiben Agentenarchitekturen den strukturellen internen Aufbau von Agenten und den Zusammenschluss von Agentengemeinschaften. Für diese Arbeit ist insbesondere das Agentenmodell AgentGHC von Bedeutung, das eine gute semantische Grundlage für die Entwicklung einer leicht verständlichen Spezifikationssprache für Agenten bildet. Vor allem lässt sich die im vorhergehenden Kapitel entwickelte interne Struktur von Objekten im Farbsortierspeicher mit diesem Modell gut umsetzen.
Inzwischen gibt es viele praktische Anwendungen von Agenten. Dabei haben sich einige besonders geeignete Einsatzbereiche herausgestellt, die in diesem Kapitel beispielhaft angesprochen werden. Es wird auch eine Typologie für Agenten angegeben, die sich an den verschiedenen Anwendungsgebieten orientiert.

3.1. Agentendefinitionen

Obwohl der Agentenbegriff gerade in den letzten Jahren sehr populär geworden ist und Agenten auf ein großes Interesse in der akademischen und industriellen Forschung gestoßen sind, gibt es leider keine einheitliche und allgemein zutreffende Definition dieses Begriffs. Ich werde im Folgenden einige oft zitierte Definitionen aus der Vielzahl von Veröffentlichungen zu diesem Thema vorstellen. Als Agent wird im Allgemeinen jemand bezeichnet, der im Auftrag eines Anderen handelt. Wesentlich dabei ist, dass er dabei eigenständig vorgeht, also Entscheidungen vom Agenten getroffen werden müssen, die der Erfüllung eines Auftrags dienlich sind. Analog zur Bedeutung des Agentenbegriffs in der menschlichen Sichtweise werden an Software-Agenten ähnliche Erwartungen gestellt. Eine Kurzdefinition über Eigenschaften, die ein Software-Agent haben soll, wird in [Huhns 97b] gegeben:

> *Agents are active, persistent (software) components that perceive, reason, act, and communicate.*

Die Autoren bezeichnen diese Begriffsbestimmung als Synthese der verschiedenen Ansichten über Software-Agenten. Eine ausführlichere und in der Fachwelt oft zitierte Definition stammt von Wooldridge und Jennings [Wooldridge 95]:

> ***A Weak Notion of Agency:*** *Perhaps the most general way in which the term agent is used is to denote a hardware or (more usually) software-based computer system that enjoys the following properties:*
> ***autonomy:*** *agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*
> ***social ability:*** *agents interact with other agents (and possibly humans) via some kind of agent-communication language;*
> ***reactivity:*** *agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;*
> ***pro-activeness:*** *agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.*

Andere Autoren ordnen Agenten noch weitere Eigenschaften zu, die sich an das menschliche Verhalten anlehnen. Besonders Forscher aus dem Gebiet der Künstlichen Intelligenz verwenden zur Beschreibung von „intelligenten“ Agenten mentalistische Begriffe wie Wissen, Glauben, Überzeugungen, Absichten und Ziele [Shoham 93]. Einige weitere Eigenschaften, die im Zusammenhang mit Agenten auftauchen, sind:

1. **Adaptives Verhalten:** Der Agent soll sich durch Signale der Umwelt und aufgrund eigener Erfahrungen dynamisch seinen Aufgaben besser anpassen.
2. **Mobilität:** Der Agent hat die Möglichkeit, sich in einem Rechnernetz von Rechnerknoten zu Rechnerknoten zu bewegen, dort z.B. Berechnungen auszuführen und Informationen zu sammeln.
3. **Rationalität:** Der Agent soll vernünftig handeln, d.h. er soll seine Ziele verfolgen.
4. **Glaubwürdigkeit:** Der Agent darf nicht lügen und absichtlich falsche Informationen bereitstellen.

Es gibt Veröffentlichungen, die sich ausschließlich bzw. in ganzen Kapiteln mit Gegenüberstellungen verschiedener Agentendefinitionen befassen (vgl. [Franklin 96], [Huhns 97b], [Knapik 98]), denn sicherlich sind die aufgeführten Begriffe noch genauer zu betrachten und gegeneinander abzugrenzen.

Die oben genannte Definition von Wooldridge und Jennings liefert aber schon einen nützlichen Abstraktionsmechanismus für den Systementwurf: Man kann Einheiten eines Systems als Agenten betrachten, d.h. man kann sie durch mentale Kategorien wie Überzeugungen, Ziele und zielgerichtetes Verhalten beschreiben. Es soll dabei nicht darum gehen, unbedingt alle Einheiten eines Systems als Agenten anzusehen, sondern nur insoweit, wie es das Verständnis für die Funktionalität bestimmter Einheiten erleichtert.

Betrachtet man beispielsweise ein Thermostat: Es arbeitet zwar autonom, doch diese Autonomie beschränkt sich ausschließlich auf Reaktionen zu Messungen seines Sensors. Änderungen der Umwelt werden direkt in Steuerungsaktionen umgesetzt. Es macht daher wenig Sinn, dieses simple System als Agenten zu beschreiben. Vom Autonomieaspekt her ist eine Klimaanlage genauso eigenständig wie ein Thermostat, doch die Entscheidungsvorgänge in der Klimaanlage sind viel komplexer, z.B. könnte sie sich den Gewohnheiten der Bewohner anpassen oder Wettervorhersagen zu Rate ziehen. Hier würde eine Beschreibung als einfacher Agent sinnvoll sein.

3.2. Agententheorie

Die Agententheorie beschäftigt sich mit der Entwicklung formaler Methoden zur Spezifikation von Agenten. Basis bildet oft der Begriff des *intentionalen Systems* zur Modellbeschreibung durch Attribute wie Glauben, Verlangen und rationale Vorgehensweise. Dabei wird zwischen verschiedenen Stufen von intentionalen Systemen differenziert (vgl. [Dennett 71], S. 243):

> *A first-order intentional system has beliefs and desires (etc.) but no beliefs and desires about beliefs and desires. ... A second-order intentional system is more sophisticated; it has beliefs and desires (and no doubt other intentional states) about beliefs and desires (and other intentional states) - both those of others and its own.*

Inwieweit es sinnvoll ist, Agenten in dieser Weise zu beschreiben, wird u.a. in [Wooldridge 95] ausführlich diskutiert. Die Autoren kommen zu dem Schluss, dass die folgenden Attribute zur Beschreibung eines Agenten nützlich sind:

Informationen (information attitudes)	**Zusatzattribute (pro-attitudes)**
Wissen (knowledge) Glauben (belief)	Verlangen (desire) Absicht (intention) Verpflichtung (obligation) Versprechen (commitment) Wahlmöglichkeit (choice)

Wissen und Glauben eines Agenten beziehen sich auf die Informationen, die ein Agent über sich und seine Umwelt hat, während die anderen Attribute eher das Verhalten des Agenten bestimmen. Es wird vorgeschlagen, dass in einem Agenten mindestens ein

Attribut jeder Spalte repräsentiert sein soll. Oft hängen Attribute der beiden Spalten aber auch sehr eng zusammen, z.B. wenn ein Agent eine Absicht aufgrund seiner Informationen über die Umwelt bildet, sodass sich auch diese Aufstellung nur zur groben Orientierung eignet.

Zur Formalisierung von Wissen und Glauben muss man sowohl eine Syntax angeben als auch ihre Semantik definieren. Es gibt in der Fachwelt nur wenige unterschiedliche Ansätze hierfür: Zur Beschreibung der Syntax gibt es die beiden fundamentalen Ansätze der *Modallogiken* und der *Meta-Sprachen*, zur Beschreibung der Semantik gibt es die beiden grundsätzlichen Methoden der *possible-worlds-Semantik* und der *interpreted symbolic structures* [Wooldridge 95].

Statt jedoch tiefer auf die formalen Methoden zur Definition von Agenten einzugehen, möchte ich im Folgenden konkreter auf die verschiedenen Architekturen und Agententypen eingehen, die sich in den letzten Jahren entwickelt haben.

3.3. Agentenarchitekturen

In diesem Abschnitt geht es darum, Softwarestrukturen zu beschreiben, die den Anforderungen der Agententheorie genügen. Auch bei der Bestimmung des Begriffs Agentenarchitektur gibt es wieder unterschiedliche Auffassungen, und im Rahmen dieser Arbeit soll damit eine Methodik für den Aufbau von Agenten gemeint sein, wie sie von P. Maes wie folgt definiert wird [Maes 91]:

> *[An agent architecture] ... 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 the agent determine the actions ... and future internal state of the agent. An architecture encompasses techniques and algorithms that support this methodology.*

Nicht explizit ausgedrückt, aber direkt aus dieser Sichtweise ableitbar ist die Aufteilung in Mikro- und Makroaspekte eines Agenten. Während Mikroaspekte die agenteninterne Sicht betreffen, z.B. die Beschreibung der internen Struktur und der Techniken zur Entscheidungsfindung, werden unter Makroaspekten agentenübergreifende Aufgaben verstanden, wie z.B. Inter-Agentenkommunikation. Ein wichtiger Gesichtspunkt ist in diesem Zusammenhang auch die Modellierung von kooperierenden, verhandelnden oder konfliktlösenden Agenten. Im Folgenden sollen die drei populärsten Agentenarchitekturen vorgestellt werden, wobei zunächst die Beschreibung der Mikroaspekte im Vordergrund steht. Allen Architekturen gemein ist eine Unterteilung in mindestens drei Komponenten:

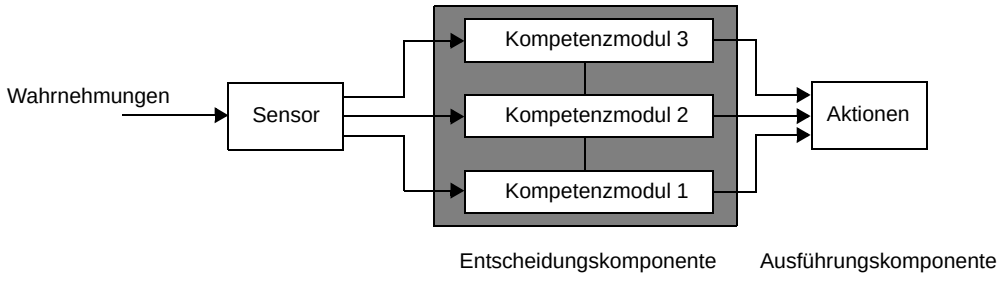
1. Über einen *Sensor* empfängt der Agent Nachrichten.
2. Eine *Entscheidungskomponente* verarbeitet die Nachrichten.
3. Eine *Ausführungskomponente* führt Aktionen des Agenten aus.

Diese drei Komponenten können nebenläufig arbeiten, doch sie verständigen sich untereinander, sodass z.B. empfangene Nachrichten eine Entscheidungsfindung und gegebenenfalls Aktionen nach sich ziehen.

3.3.1. Reaktive Agenten

Stellvertretend für das Konzept reaktiver Agenten wird im Folgenden die *subsumption architecture* von Brooks erläutert, bei der *aufgabenausführendes Verhalten* durch endliche Automaten modelliert wird, die wahrgenommene Informationen zu bestimmten Ausgaben umsetzen [Brooks 86]. Es ist kein internes symbolische Modell der Umwelt vorhanden und es werden keine komplexen Schlussfolgerungsmechanismen benutzt, sondern es gibt eine Reihe von Modulen, die jeweils mit den Fähigkeiten ausgestattet sind, einen fest umgrenzten Aufgabenbereich zu übernehmen. Entsprechend der Einteilung in Sensor, Entscheidungs- und Ausführungskomponente hat das Modell eines solchen reaktiven Agenten die in der folgenden Abbildung dargestellte Struktur.

Abbildung 4 Reaktive Agentenarchitektur



Die Kompetenzmodule arbeiten parallel, sind jedoch hierarchisch angeordnet. In den unteren Schichten sind die eher primitiven und konkreten Verhaltensweisen zu finden, in den darüberliegenden die komplexeren und abstrakteren. Für einen mobilen Roboter könnten z.B. die folgenden drei Module sinnvoll sein:

- Modul 1: Vermeiden von Zusammenstößen mit Hindernissen und Robotern
- Modul 2: Umherfahren
  (mit Benutzung der Fähigkeiten zur Vermeidung von Zusammenstößen)
- Modul 3: Umwelt erkundschaften
  (mit Benutzung der Fähigkeiten zum Umherfahren)

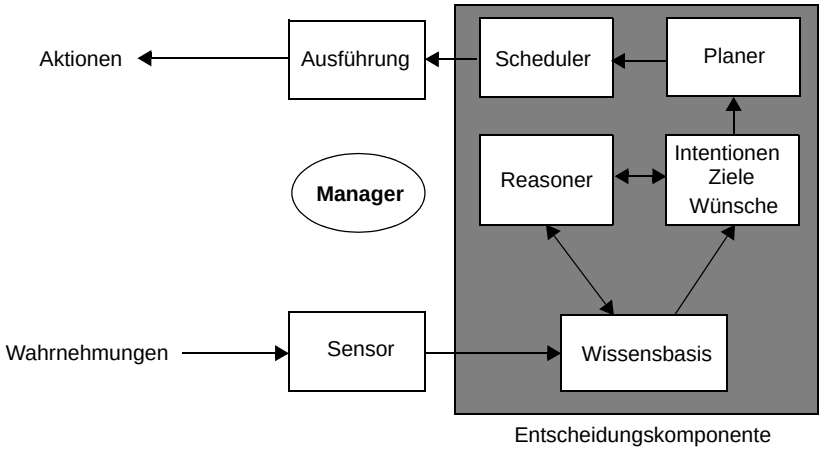
Übergeordnete Module setzen die Fähigkeiten ihrer untergeordneten Module ein, um ihre Aufgaben zu erfüllen. Innerhalb der Module wird eine regelbasierte Sprache eingesetzt, die auf endlichen Automaten basiert. Die durch die verschiedenen Module erzielte dezentrale Struktur gilt als robust und fehlertolerant und eignet sich für Aufgabengebiete, in denen schnelle Reaktionen auf Wahrnehmungen erforderlich sind. Obwohl mit dieser Architektur einige erfolgreiche Ergebnisse erzielt worden sind, hat sie doch einige Nachteile; u.a. muss das endgültige Verhalten durch Abstimmung der Module untereinander bestimmt werden. Da diese Abstimmung sehr schwer zu modellieren ist, hat sich keine allgemeine Methodik für den strukturellen Aufbau der Kompetenzmodule entwickelt. Man muss viel experimentieren und oft nach dem trial-and-error-Verfahren vorgehen [Jennings 96].

3.3.2. Deliberative Agenten

Deliberative Agenten haben ein ausdrücklich beschriebenes symbolisches Modell ihrer Umwelt, das auch als Wissensbasis bezeichnet wird. Der deliberative Aspekt kommt durch die Fähigkeit zu Schlussfolgerungen zum Ausdruck: Mithilfe von Regeln und der Wissensbasis und können Überzeugungen (beliefs) gewonnen und auch Entscheidungen getroffen werden, die sich z.B. in Aktionen des Agenten ausdrücken. Im Folgenden sollen BDI-Agenten als eine spezielle deliberative Architektur vorgestellt werden.

Struktureller Aufbau deliberativer Agenten. Einige Komponenten deliberativer Agenten sind bereits angesprochen worden. In der folgenden Graphik wird ihr Zusammenhang mit weiteren wichtigen Komponenten deutlich (aus: [Sattler 98]).

Abbildung 5 Deliberative Agentenarchitektur



Die Entscheidungskomponente teilt sich in mehrere Instanzen auf, die in der Graphik grau unterlegt dargestellt sind. Der Reasoner bezeichnet die Schlussfolgerungskomponente, mit der aufgrund von Wissen und Wünschen neue Überzeugungen gebildet werden. Der Planer übernimmt die Aufgabe, Intentionen zu verwalten und gegebenenfalls Pläne dynamisch an neue Situationen anzupassen. Im Scheduler erfolgt die Auswahl der auszuführenden Aktionen unter Berücksichtigung der vorhandenen Ressourcen. Ein Manager übernimmt die Koordination zwischen diesen Komponenten, dem Sensor und dem Ausführer, der für die eigentliche Abarbeitung der Aktionen zuständig ist.

Mentale Kategorien von BDI-Agenten. Der BDI-Ansatz stammt von Bratman und modelliert die drei Kategorien *belief*, *desire* und *intention* [Bratman 87]. Dieser Ansatz wurde von Rao und Georgeff weiterentwickelt und um die Begriffe *goal* und

plan ergänzt. In der folgenden Tabelle wird die Bedeutung dieser Begriffe näher erklärt. Eine ausführlichere Beschreibung liefert z.B. [Müller 96b].

mentale Kategorie (Originalbezeichnung)	entsprechender Begriff in deutscher Sprache	Bedeutung im Agentenumfeld
knowledge	Wissen, Fakt	Ein Fakt beschreibt eine (lokal) wahre Begebenheit über den Zustand des Agenten oder seiner Umgebung.
belief	Glauben, Überzeugung	Eine Überzeugung ist eine Annahme des Agenten über den Zustand seiner Umwelt.
desire	Verlangen, Wunsch	Ein Wunsch beschreibt einen vom Agenten angestrebten zukünftigen Zustand des Agenten oder der Umgebung.
goal	Ziel	Die Ziele eines Agenten sind eine konsistente Teilmenge der Wünsche; sie stehen nicht in Konflikt zueinander.
intention	Absicht, Vorhaben	Nicht alle Ziele können gleichzeitig verfolgt werden, sodass eine Auswahl getroffen werden muss, die in einer sogenannten Intentionsstruktur verwaltet werden.
plan	Strategie, Plan	Ein Plan beschreibt das Vorgehen eines Agenten zur Erreichung eines Ziels.

Anfang der 90er-Jahre haben Rao und Georgeff diese mentalen Kategorien mit einer Modallogik formalisiert und als semantische Grundlage den possible-worlds-Ansatz gewählt [Rao 91]. Sie geben den folgenden abstrakten Interpreter für BDI-Agenten an, der die Aufgabe des Managers aus Abbildung 5 übernimmt [Rao 92]. Kritisch angemerkt wurde bei dieser theoretischen Arbeit jedoch, dass die angegebene Modallogik nicht vollständig beschrieben und keine effiziente praktische Implementierung des vorgestellten Interpretermodells möglich ist.

Abbildung 6 Abstrakter BDI-Interpreter von Rao und Georgeff

```
initialize-state ();
repeat
    options := option-generator (event-queue);
    selected-options := deliberate (options);
    update-intentions (selected-options);

    execute ();

    get-new-external-events ();
    drop-successful-attitudes ();
    drop-impossible-attitudes ();
end-repeat
```

Bei der Initialisierung werden u.a. Daten in den Agenten aufgenommen, z.B. Fakten in die Wissensbasis. Anschließend gerät der Agent in einen fortwährenden Zyklus. Aus

einer Menge von Ereignissen (event-queue) wird die Menge der möglichen Ziele bestimmt (options). Hiervon werden die Ziele bestimmt, die mit den verfügbaren Ressourcen erreichbar sind (selected-options). Sie werden anschließend in die Menge der Intentionen aufgenommen (update-intentions). Dann erfolgt der Aufruf der Ausführungskomponente, in der Aktionen zur Erreichung der Ziele ausgeführt werden (execute). Nach der Abarbeitung von Intentionen werden neue Ereignisse durch den Sensor aufgenommen (get-new-external-events). Am Ende jedes Zyklus wird die Menge der Ziele aktualisiert, indem erreichte Ziele entfernt und nicht mehr erreichbare Ziele verworfen werden (drop-successful-attitudes und drop-impossible-attitudes).

Für eine praktische Realisierung dieses Interpreters müssen allerdings noch einschränkende Annahmen gemacht werden, da sich hinter einigen der benutzten Funktionen sehr komplexe Berechnungen verbergen, die eine effiziente Implementierung unmöglich machen. Mit den vorgenommenen Einschränkungen sind einige Realisierungen nach diesem Modell zustande gekommen, von denen die bekanntesten wohl das *Procedural Reasoning System* (PRS) und dessen Weiterentwicklung *distributed Multi-Agent Reasoning System* (dMARS) sind (vgl. [Georgeff 87], [Georgeff 94]). Diese Systeme fallen bei anderen Quellen allerdings schon unter hybride Architekturen, die erst im nächsten Abschnitt erklärt werden (vgl. [Wooldridge 95], [Nwana 96]). Zur Formulierung der semantischen Grundlagen von PRS und dMARS wurde die von Rao entwickelte Sprache *AgentSpeak(L)* benutzt [Rao 96]. Interessanterweise wurde AgentSpeak(L) erst nachträglich zu PRS und dMARS entwickelt und veröffentlicht. Die durch dieses „Reverse Engineering“ entstandene Sprache ist eine erfolgversprechende Ausgangsbasis für neue, erweiterte Agentensprachen, denn zu ihr gibt es ja bereits lauffähige Systeme.

Bei AgentSpeak(L) handelt sich um eine eingeschränkte logische Programmiersprache mit Ereignissen und Aktionen. Das Verhalten eines Agenten wird durch ein in AgentSpeak(L) geschriebenes Programm bestimmt. Der entscheidende Unterschied zu bisherigen Ansätzen liegt darin, dass die Überzeugungen, Wünsche und Ziele nicht als modale Formeln angegeben werden, sondern der Agentenentwickler diese Kategorien in der Programmiersprache formuliert. Der aktuelle Status des Agenten, der u.a. das Modell seiner Überzeugungen und seiner Umwelt enthält, wird durch den aktuellen Wissensstand angegeben. Zustände, die der Agent in der Zukunft erreichen will, sind seine Ziele, und die Programme (oder: Pläne) zur Erreichung von Zielen sind seine Intentionen. Rao beschreibt die Motivation für diesen Ansatz so (vgl. [Rao 96], S. 1):

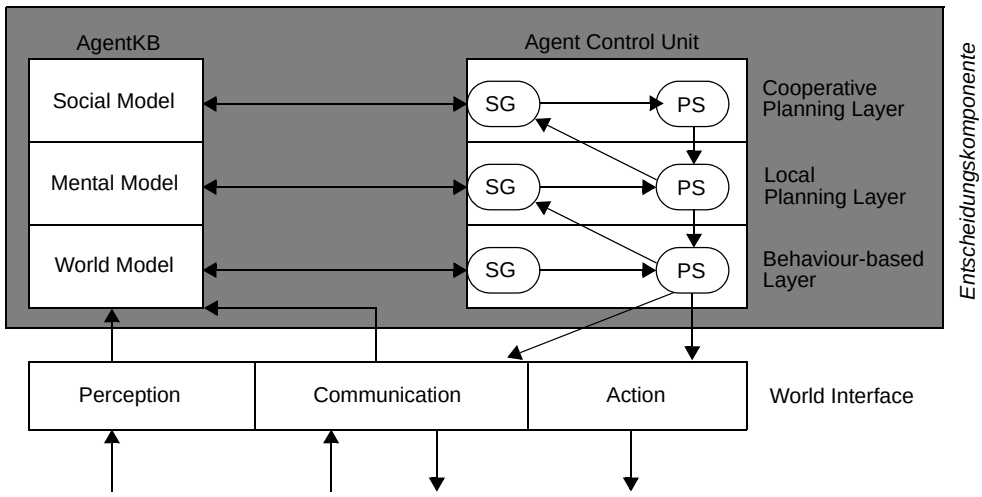
> *This shift in perspective of taking a simple specification language as the execution model of an agent and then ascribing the mental attitudes of beliefs, desires, and intentions, from an external viewpoint is likely to have a better chance of unifying theory and practice.*

Rao spricht hiermit ein großes Manko in der bisherigen Agentenforschung an: Während einige Forscher sich auf die theoretischen Basiskonzepte konzentrieren, entwickeln andere immer neue praktische Anwendungen ohne semantische Grundlagen. Um die Lücke zwischen diesen beiden Richtungen weiter zu schließen, hat C. Geiger in seiner Dissertation das Agentenmodell *AgentGHC* entwickelt, das auf den Konzepten von AgentSpeak(L) aufbaut (vgl. Abschnitt 3.5.).

3.3.3. Hybride Agenten

Viele Forscher schlagen vor, die Stärken von reaktiven und deliberativen Architekturen zu verbinden, indem ein Agenten aus zwei Teilen gebildet wird: Der reaktive Teil kann direkt auf Änderungen der Umwelt reagieren, während der deliberative Teil ein symbolisches Modell der Umwelt hat, Strategien entwickelt und Entscheidungen trifft. Dem reaktiven Teil wird oft Vorrang eingeräumt, sodass schnelle Reaktionen auf wichtige Ereignisse möglich sind. So gelangt man zu geschichteten Architekturen, wie z.B. TOURINGMACHINES [Ferguson 92], INTERRAP [Müller 96a] und auch das schon erwähnte PRS. Im Folgenden soll INTERRAP als Beispiel für eine hybride Agentenarchitektur kurz vorgestellt werden (vgl. Abbildung 7).

Abbildung 7 Hybride Agentenarchitektur INTERRAP



Jeder Agent hat eine Wissensbasis (AgentKB) und eine Kontrolleinheit (AgentControlUnit), die die grau unterlegte Entscheidungskomponente bilden. Das sogenannte World Interface umfasst die Sensorik (Perception), die Ausführungskomponente (Action) und eine Komponente zur Kommunikation mit der Umwelt. Die Kontrolleinheit ist in drei hierarchische Schichten unterteilt: Auf der untersten Ebene die verhaltensbasierte Schicht (Behaviour-based Layer, BBL), in der Mitte die lokale Planungsschicht (Local Planning Layer, LPL) und auf der obersten Ebene die Kooperationsschicht (Cooperative Planning Layer, CPL). Den Charakter einer reaktiven Architektur erhält INTERRAP durch die BBL, in der mit Reiz-Reaktions-Regeln (sogenannte patterns of behaviour) effiziente und robuste Implementierungen möglich sind. In der LPL wird lokales zielgerichtetes Verhalten behandelt, und schließlich ermöglicht die CPL, dass der Agent mit anderen Agenten kooperieren kann. Die Wissensbasis teilt sich ebenfalls in drei Teile auf und jede Kontrollschicht hat dort ein entsprechendes Modell für ihre benötigten Informationen. In jeder Kontrollschicht gibt es zwei Instanzen, die sich untereinander verständigen, aber auch Kontakt zu anderen Kontrollschichten aufnehmen können: SG ist für die Situationserkennung und Zielaktivierung zuständig, PS für die Planung und das Scheduling. Reichen die Fähigkeiten einer Schicht nicht aus, wird das Ziel an die

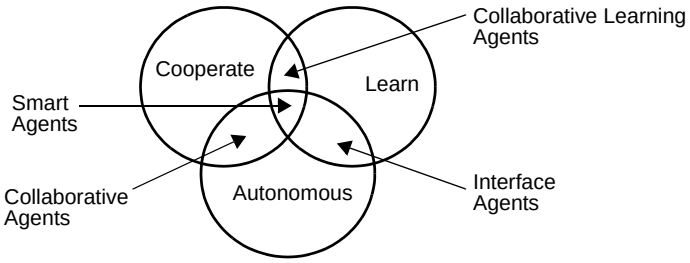
darüberliegende Schicht weitergeleitet. Der Ausführungsprozess läuft genau umgekehrt: Die oberen Schichten nutzen die Fähigkeiten der unteren. Mit Agenten dieser Architektur ist eine automatisierte Verladestation mit Gabelstaplern realisiert worden [Müller 96b].

3.4. Agententypen

Genauso wie es viele verschiedene Definitionen für den Agentenbegriff gibt, existieren auch viele Klassifizierungen für die unterschiedlichen Agententypen. Nwana gliedert Agenten zunächst nach unterschiedlichen Gesichtspunkten in mehreren Dimensionen [Nwana 96]:

- **Mobilität:** Unterscheidung zwischen statischen und mobilen Agenten, die die Fähigkeit haben, sich selbst in einem Rechnernetzwerk von einem Rechnerknoten zu anderen zu befördern und sich dort selbst auszuführen.
- **Architektur:** Unterscheidung zwischen reaktiven und deliberativen Agenten, wie sie schon im letzten Abschnitt vorgestellt worden sind.
- **Primärattribute:** Nwanas Minimalanforderungen an einen Agenten sind Autonomie, Lernfähigkeit und Kooperation. Mit diesen Attributen stellt er die in Abbildung 8 dargestellte Teilsicht für die Einteilung von Agenten auf.

Abbildung 8 Teilsicht einer Agententypologie (Nwana)



Aus dieser Einteilung gehen vier Agententypen hervor, die mindestens zwei der Attribute als maßgebliche Eigenschaften besitzen: kollaborative (zusammenarbeitende) Agenten, kollaborative lernende Agenten, Interface-Agenten und „schlaue“ Agenten. Auch hier sind die Grenzen fließend: Zum Beispiel können Interface-Agenten auch mit anderen Agenten kooperieren, nur gehört dies eben nicht zu ihren Haupteigenschaften.

- **Rolle:** Agenten können auch nach ihrer Hauptaufgabe eingeteilt werden. In diesem Zusammenhang sind insbesondere Informationsagenten zu nennen, die große Mengen von Daten sammeln, filtern und aufarbeiten können.
- **Hybride Agenten:** Unter hybriden Agenten versteht Nwana solche Agenten, die in sich mehrere „Agentenphilosophien“ vereinen.

Diese Einteilung ist nicht einheitlich, denn einige Kriterien beziehen sich auf die zugrunde liegende Technologie (z.B. mobile oder deliberative Agenten), andere wie-

derum auf die Aufgabengebiete (z.B. Interface- und Informationsagenten). Also müsste eigentlich eine mehrdimensionale Matrix aufgestellt werden, in der die Agenten entsprechend eingeordnet werden. Zur besseren Übersicht verzichtet Nwana aber darauf und gibt stattdessen die folgende Liste von relevanten Agententypen an, in die die meisten der existierenden Software-Agenten eingeordnet werden können:

Kollaborative Agenten. Bei diesem Agententypen steht neben Autonomie die Zusammenarbeit mit anderen Agenten im Vordergrund. Insbesondere Agenten aus dem Bereich der Multi-Agentensysteme (MAS) können diesem Typ zugeordnet werden. Es gibt viele Gründe, die für den Einsatz solcher Agenten sprechen: Es können Probleme gelöst werden, die für ein einzelnes zentrales Programm zu komplex sind, z.B. aufgrund von beschränkten Ressourcen. Weiterhin gibt es viele Gebiete, die sich schon durch ihre Struktur für eine verteilte Lösung anbieten, z.B. bei verteiltem Wissen.

Interface-Agenten. Diese Agenten arbeiten mit einem Anwender zusammen und weniger mit anderen Agenten, daher wird in diesem Zusammenhang auch oft von autonomen *personal assistants* gesprochen. Interface-Agenten haben die Fähigkeit, sich auf den Anwender adaptiv einzustellen und ihn dadurch besser bei der Bewältigung von Aufgaben zu unterstützen. Es gibt im Wesentlichen vier Möglichkeiten, wie ein Agent sich besser auf den Anwender einstellen kann: Durch Beobachtung und Imitation, durch positives und negatives Feedback des Anwenders, durch Anfragen bei anderen Agenten oder durch Mitteilungen des Anwenders. Forschung auf dem Gebiet der Interface-Agenten wird besonders am MIT Media Lab unter Prof. P. Maes betrieben. Zum Beispiel wurde ein Kalenderagent entwickelt, der seinem Anwender bei der Terminplanung behilflich ist. Im Laufe der Zeit übernimmt der Agent die Vorlieben seines Anwenders, z.B. kann der Agent lernen, dass sein Benutzer nicht gerne Termine an einem Freitag Nachmittag macht und Termine gerne auf den frühen Vormittag legt [Kozierok 93].

Mobile Agenten. Ein mobiler Agent kann in Netzwerken von Rechnerknoten zu Rechnerknoten migrieren und dort alle notwendigen Arbeitsschritte ausführen, bevor ein Ergebnis zurückgeschickt wird oder der Agent noch weiter zu anderen Rechnerknoten im Netzwerk wandert. Diese Vorgehensweise wird auch *remote programming* genannt und oft als Erweiterung des Prozedur-Fernaufrufs (remote procedure call, RPC) der Client/Server-Architekturen angesehen; u.a. soll durch die entfernte Ausführung des ganzen Programms die Netzwerklast verringert werden. Anwendungen findet man besonders bei Agenten zur Informationsbeschaffung im Internet, aber auch zunehmend im Bereich sogenannter elektronischer Marktplätze. Bei mobilen Agenten treten allerdings auch viele kritische Besonderheiten auf, u.a. Sicherheitsprobleme wie die Authentifikation beim Zielrechner, der Schutz vor Agentenattacken auf geschützte Bereiche des Zielrechners und der Schutz vor Attacken auf den mobilen Agenten. Einen kurzen Überblick über den Forschungsstand und einige Prototyp-Systeme gibt [Mattern 98].

Informations- bzw. Internet-Agenten. Diese Agenten versuchen, das explosionsartige Anwachsen von verfügbaren Informationen im Internet zu bewältigen. Sie übernehmen die Aufgabe, Informationen zu sammeln (z.B. die Ergebnisse von Anfragen bei verschiedenen Suchmaschinen), zu filtern, aufzuarbeiten und dem Benutzer zu prä-

sentieren. Informationsagenten sind eher statisch und von den bereits angesprochenen Interface- und mobilen Agenten ursprünglich leicht zu unterscheiden gewesen, doch durch die Ausweitung des WWW verschwindet diese Trennlinie immer weiter: Viele Interface- und mobile Agenten werden auch als Informationsagenten angesehen. Ein Anwendungsbeispiel ist der Internet Softbot von Etzioni und Weld [Etzioni 94].

Reaktive Agenten. Die diesem Agententypen zugrunde liegende Architektur ist bereits im vorherigen Abschnitt behandelt worden. Auch Nwana führt als Kritik an dieser Architektur die mangelnde Methodik bei der Erstellung von Applikationen auf. Trotzdem sind einige interessante reaktive Agentensysteme entwickelt worden, z.B. zur Simulation von Ameisenvölkern [Ferber 94].

Hybride Agenten. Auch dieser Agententyp ist bereits als Agentenarchitektur behandelt worden. Als Realisierungen hybrider Agenten sind neben INTERRAP, TOURING-MACHINES und PRS auch noch OASIS [Rao 95] und CIRCA zu nennen.

Nachdem nun ein kurzer Abriss über die verschiedenen existierenden Agententypen gegeben wurde, wird im Folgenden ein interessantes hybrides Agentenmodell vorgestellt, dessen Konzepte die Basis meiner Spezifikationssprache und des zugehörigen Ausführungsmodells bilden.

3.5. AgentGHC als praktisches Agentenmodell

Als Weiterentwicklung von AgentSpeak(L) und der objektorientierten Variante *AgentSpeak*, mit der Agentengruppen und die Kommunikation zwischen Agenten realisiert werden können, hat C. Geiger das Agentenmodell *AgentGHC* entwickelt [Geiger 98]. Ich beschreibe in diesem Abschnitt hauptsächlich die Konzepte der agenteninternen Sicht (Mikrosicht) und gehe nur kurz auf die agentenübergreifende Sicht (Makrosicht) ein. AgentGHC setzt im Wesentlichen auch das BDI-Modell um, in dem jeder Agent Überzeugungen (beliefs), Wünsche (desires) und Absichten (intentions) modelliert und ein symbolisches Modell seiner Umwelt hat. Dieses symbolische Modell wird im Folgenden als Wissensbasis bezeichnet.

Grundlegende Definitionen. Das Wissen eines Agenten wird als Menge **Bel** von *atomaren beliefs b(t)* bezeichnet. Dies sind Fakten, die z.B. Informationen über den momentanen Status des Agenten beschreiben. Die Menge **Goals** von Zielen beschreibt die Zustände, die der Agent in Zukunft erreichen möchte. Sie setzt sich zusammen aus zwei Teilmengen: Die Menge $\mathbf{G_T}$ der *Tests*, die das Vorhandensein bestimmter Fakten in der Menge **Bel** überprüfen, und die Menge $\mathbf{G_Z}$ der *tiefen Ziele*, zu denen erst noch eine Strategie gefunden werden muss, um sie zu verfolgen. Zur Formulierung von Strategien müssen erst noch ein paar weitere Begriffe vorgestellt werden: Ein Agent führt zum Erreichen eines Ziels u.a. *Aktionen* aus, die in der Menge **Act** zusammengefasst sind, z.B. kann der Agent durch Hinzufügen oder Löschen von Fakten seine Wissensbasis manipulieren. Auch *Nachrichten* als spezielle Form von Aktionen sind möglich, um mit anderen Agenten zu kommunizieren. Die Menge der möglichen Nachrichten wird mit **Msg** bezeichnet. Vorgesehen sind z.B. Nachrichten zum synchronen und asynchronen Senden (send, sendReceive) oder zum Verschicken von Anfragen (request).

Schließlich werden wahrnehmbare Ereignisse ε als Tupel $\langle e, Q \rangle$ angegeben, wobei e das Ereignis beschreibt und Q die Quelle (der Agent selbst oder ein anderer Agent) angibt. Die Menge zu bearbeitender Ereignisse heißt **Erg**. Ereignisse können z.B. neue zu erreichende Ziele sein, die von einem anderen Agenten in Auftrag gegeben wurden. Mit diesen Begriffen können nun Strategien beschrieben werden, die eine zentrale Rolle in AgentGHC spielen:

Strategien. Ein Plan P aus der Menge **Strat** aller Strategien wird als bewachte Horn-Klausel (*Guarded Horn Clause*, GHC) $P: H \rightarrow G_1 ... G_n \mid B_1 ... B_m \; [g]$ mit $g \in [0,1]$ definiert. Der Klauselkopf H ist ein wahrnehmbares Ereignis. $G_1 ... G_n$ werden als *Guards* oder auch *Kontextbedingungen* bezeichnet und können sowohl Tests, Ziele oder Nachrichten, jedoch keine anderen Aktionen sein. Die Bodyprädikate $B_1 ... B_m$ bilden den Plankörper und können Tests, Ziele, Nachrichten oder Aktionen sein. Die Abarbeitung innerhalb der Sektionen kann nebenläufig oder sequentiell erfolgen. Die Bewertung einer Strategie mit einer reellen Zahl $g \in [0,1]$ wird als Priorität bezeichnet. Bei Strategien wird zwischen drei hierarchischen Ebenen unterschieden: *Reaktive Strategien* haben als Kontextbedingungen nur Tests, d.h. diese Strategien sind ereignisgetrieben und greifen nur auf die Wissensbasis zurück, sodass sie sehr schnell auf ihre Anwendbarkeit überprüft werden können. *Deliberative Strategien* können zusätzlich auch tiefe Ziele als Kontextbedingungen haben, mit denen komplexere Überlegungen, die sich aber auf den Agenten selbst beschränken, modelliert werden können. Schließlich gibt es noch *kommunikative Strategien*, die in ihrer Kontextbedingung auch mit anderen Agenten kommunizieren können. Reaktive Strategien haben Vorrang vor allen anderen Strategien, und deliberative Strategien haben Vorrang vor kommunikativen Strategien.

Operationale Semantik. Ein Agent wird definiert als Tupel

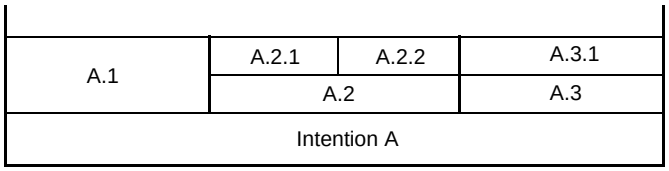
$$< \text{Bel, Strat, Msg, Act, RunTime(Erg, Int, M, A, } S_e, S_p, S_i) >$$

Die Struktur RunTime beschreibt den aktuellen Zustand des Agenten. Neben der Menge **Erg** wahrzunehmender, noch nicht bearbeiteter Ereignisse gibt es die Menge **Int** der aktuell verfolgten Intentionen. Die Mengen **M** und **A** beinhalten die noch zu versendenden Nachrichten und auszuführenden Aktionen. $\mathbf{S_e}$, $\mathbf{S_p}$ und $\mathbf{S_i}$ sind Selektionsfunktionen, die für die Auswahl von Ereignissen, Strategien und Intentionen benötigt werden. Mit diesen Angaben ist man in der Lage, einen zyklischen Ablauf im Agenten festzulegen, der im Folgenden beschrieben und zusätzlich in der Abbildung 10 beispielhaft skizziert wird (aus: [Geiger 98]).

Bei jedem Zyklus wählt der Agent aus der Menge **Erg** mit der Selektionsfunktion $\mathbf{S_e}$ ein zu bearbeitendes Ereignis aus. Zu diesem Ereignis werden die Strategien aus der Menge **Strat** betrachtet, die in der Lage sind, dieses Ereignis zu bearbeiten (Ermitteln relevanter Strategien). Diese Strategien werden auf ihre Anwendbarkeit geprüft, d.h. die Guards der Strategien werden einem Kontexttest unterzogen (Ermitteln anwendbarer Strategien). Dies kann bei reaktiven Strategien direkt durch Zugriffe auf die Wissensbasis erfolgen, jedoch zieht dieser Kontexttest bei deliberativen oder kommunikativen Strategien in der Regel eine komplexe Berechnung nach sich und kann zu einem rekursiven Aufruf des Interpreters führen. Aus den anwendbaren Strategien wird mit der Selektionsfunktion $\mathbf{S_p}$ eine Strategie ausgewählt und eine *Intention* gebildet bzw. erweitert.

Es wird zwischen Intentionen, die durch externe Ereignisse gebildet werden, und Intentionen, die durch interne Ereignisse erweitert werden, unterschieden. Ist das bearbeitete Ereignis von einer externen Quelle induziert worden, wird eine neue Intention zur Menge **Int** hinzugefügt. Ist das Ereignis vom Agenten selbst erzeugt, wird die erzeugende Intention um einen weiteren Plan ergänzt. Eine Intention ist demnach wie der abstrakte Datentyp Keller organisiert: es handelt sich dabei um einen Stapel teilweise ausgeführter Pläne. Es ist möglich, dass eine Intention mehrere parallele Unterziele verfolgt, sodass man in einer Stapelschicht auch mehrere Elemente vorliegen haben kann. Diese Struktur wird im Folgenden als Multikeller bezeichnet (vgl. Abbildung 9).

Abbildung 9 Multikeller: Beispiel für eine Intention mit parallelen Unterintentionen

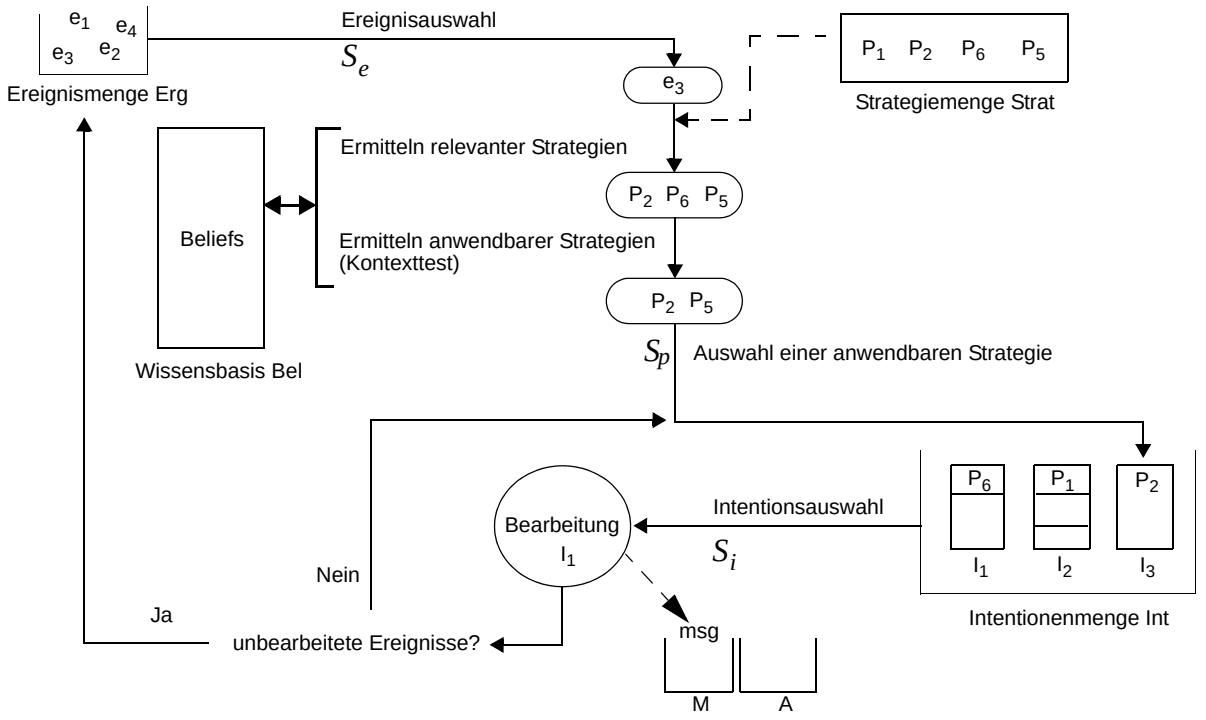


Die Selektionsfunktion $\mathbf{S_i}$ wählt außer einer Intention der Menge **Int** auch noch eines der obersten Kellerelemente aus. In Abbildung 9 kann dies eines der Elemente A.1, A.2.1, A.2.2 oder A.3.1 sein. Von diesem Element, das einen teilweise ausgeführten Plan repräsentiert, wird anschließend ein Abarbeitungsschritt ausgeführt, der eine neue Aktion, eine neue Nachricht oder ein neues internes Ereignis (z.B. ein Unterziel) bedeuten kann. Solange keine weiteren zu bearbeitenden Ereignisse warten, werden die Intentionen der Menge **Int** auf diese Weise weiter abgearbeitet. Wenn weitere Ereignisse in der Menge **Erg** vorliegen, beginnt der Zyklus von Neuem. C. Geiger gibt einen abstrakten Interpreter an, der dieses Vorgehen des Agenten formal beschreibt (vgl. Anhang A), und betrachtet die Verwaltung paralleler Unterziele sowie die explizite Unterbrechbarkeit von Intentionen.

Ein Beispielzyklus. In Abbildung 10 wird der Ablauf eines Zyklus an einem Beispiel aufgezeigt. Wie die einzelnen Selektionsfunktionen dabei definiert sind, soll an dieser Stelle nicht weiter von Bedeutung sein.

Zu Beginn des Zyklus wird das Ereignis e_3 aus der Menge **Erg** durch die Selektionsfunktion $\mathbf{S_e}$ ausgewählt. Von den vorhandenen Strategien sollen die Pläne P_2, P_5 und P_6 für e_3 relevant sein und nach Überprüfung ihrer Vorbedingungen P_2 und P_5 als anwendbar übrig bleiben. Mittels der Selektionsfunktion $\mathbf{S_p}$ wird in diesem Fall der Plan P_2 ausgewählt und mit ihm die neue Intention I_3 gebildet und in die Intentionenmenge **Int** aufgenommen. Schließlich erfolgt die Auswahl einer Intention und ein weiterer Abarbeitungsschritt des zugehörigen Plans wird ausgeführt. In diesem Beispiel wird bei der Ausführung von Plan P_6 aus der Intention I_1 eine zu versendende Nachricht msg in die Menge **M** eingefügt. Damit ist der Zyklus abgeschlossen. Da sich noch weitere Ereignisse in der Menge **Erg** befinden, wird im nächsten Zyklus wieder ein Ereignis ausgewählt und entsprechend weiterbearbeitet.

Abbildung 10 Operationale Sicht in AgentGHC



Zur Makrosicht in AgentGHC. Die Beschreibung von AgentGHC beschränkt sich bisher auf die agenteninternen Abläufe, aber es wurde auch eine Makrosicht für AgentGHC entwickelt: Eine *Agentenfamilie* entspricht dabei einer Klasse in einer objektorientierten Programmiersprache, ein Agent der Instanz einer Klasse. Abbildung 11 gibt die Struktur der Basisfamilie *UrAgentGHC* an, von der alle weiteren AgentGHC-Agenten abgeleitet werden.

Abbildung 11 Klassenstruktur eines AgentGHC-Agenten

```
agent_family UrAgentGHC
   public
      Dienste
      Informationen
   private
      Beliefs
      Strategien
      Aktionen
      Nachrichten

      Methoden      // für die Bearbeitung von Beliefs, Strategien, Aktionen und Nachrichten
end
```

In einer öffentlichen Sektion (public) legt der Agent Teile seiner Ziele und Test als Dienste und Informationen offen. Andere Agenten können nun Nachrichten an einen AgentGHC-Agenten schicken, auf die er mit einer Diensterbringung oder der Preisgabe einer Information reagiert. Die Diensterbringung erfolgt durch eine als Ziel interpretierte Nachricht und zieht eine entsprechende Bearbeitung im Agenten nach sich, d.h. Planauswahl, Intentionsbildung und -abarbeitung. Die private Sektion, die nicht für andere Agenten sichtbar ist, enthält alle übrigen Strukturen, die schon vorgestellt wurden, sowie Methoden zur Organisation der internen Strukturen. Im Anhang A findet man eine Übersicht über die vorgesehenen privaten Methoden der Agentenfamilie UrAgentGHC. Es wird auch die Möglichkeit geschaffen, sogenannte virtuelle Agentengruppen zu bilden. Es handelt sich dabei um familienübergreifende Gruppierungen, in die Agenten explizit eingetragen werden. Der Agent, der eine virtuelle Gruppe erzeugt, ist automatisch ihr Besitzer und verwaltet die Zugriffsrechte innerhalb der Gruppe.

AgentGHC als Basis für eine neue Spezifikationssprache

Mit AgentGHC kann man Objekte durch die für „intelligente“ Agenten gebräuchlichen Begriffe wie Wissen, Ziele und Strategien beschreiben. Eine solche Beschreibung gibt eine intuitive Sicht auf Komponenten größerer Systeme wieder und unterstützt daher die Modellierung von komplexen Systemen.

Gegenüber AgentSpeak(L) macht AgentGHC sinnvolle Erweiterungen, wie z.B. Parallelität innerhalb von Strategien, eine Planhierarchie beim Kontexttest oder die explizite Unterbrechbarkeit von Intentionen. Dadurch, dass AgentGHC auf einer formalen und bereits praktisch umgesetzten Grundlage aufbaut, besteht die berechtigte Hoffnung, dass eine Realisierung von AgentGHC mit den beschriebenen Erweiterungen möglich ist.

An einigen Stellen sollten allerdings erst noch Verfeinerungen und Ergänzungen im abstrakten Interpreter gemacht werden. So muss nicht unbedingt für jedes wahrgenommene Ereignis eine Strategie gesucht und eine Intention gebildet werden, zum Beispiel wenn der Inhalt der Nachricht lediglich die Information über ein neues ”Belief” ist. Auch der vorgeschlagene rekursive Aufruf des Interpreters beim Kontexttest deliberativer Pläne ist zu überdenken; rekursive Aufrufe innerhalb eines Interpreter*zyklus* sind nicht sehr hilfreich für das Verständnis des Agentenmodells.

Auf Basis von AgentGHC muss eine leicht zugängliche Sprache entwickelt werden, mit der insbesondere der Anfangszustand für Agenten spezifiziert werden kann. Eine solche Spezifikation dient bei der Generierung eines neuen Agenten zur Initialisierung der Komponenten des AgentGHC-Interpreters. In Kapitel 6 wird die Spezifikationssprache CASA vorgestellt, die alle Elemente aus der Mikrosicht von AgentGHC umfasst. In demselben Kapitel werden auch Lösungen für die oben angesprochenen Erweiterungen des abstrakten Interpreters aufgezeigt.

KAPITEL 4

Agenten-Kommunikationssprachen

An den Definitionen und Beschreibungen des vorangehenden Kapitels erkennt man eine fundamentale Eigenschaft von Software-Agenten: Sie müssen zu ihrer Umgebung in Kontakt treten können; nur so machen sie sich überhaupt nützlich. Unter die Umgebung eines Agenten fallen menschliche Benutzer, Hardware in Form von Maschinen jeglicher Art, andere Softwareprodukte und insbesondere Agenten, die nicht nur lokal auf demselben Rechner vorliegen, sondern auch auf anderen Rechnerknoten in einem Netzwerk verteilt sein können. Mit anderen Agenten in Kontakt zu treten bedeutet mit ihnen kommunizieren zu müssen. In diesem Kapitel werden daher zunächst die generellen Kommunikationsverfahren zwischen Softwareapplikationen kurz vorgestellt, insbesondere die für Agenten meist benutzte Form des Nachrichtenversands.

Damit Agenten miteinander kommunizieren können, müssen sie eine gemeinsame Sprache verstehen und auch den Inhalt der Sprache richtig deuten können. Bei der Entwicklung von geeigneten Kommunikationssprachen für Agenten hat die Sprechakttheorie geholfen, in der unterschiedliche Verständigungsformen wie Fragen, Antworten und Befehle untersucht werden. So können auch typische Abläufe von Dialogen aufgestellt werden, die die Basis für Interaktionsprotokolle zwischen Agenten bilden.

In den letzten Jahren haben sich Forscher darum bemüht, Standards für Agenten-Kommunikation zu entwickeln, sodass sich die Agenten verschiedener Entwickler auch gegenseitig verstehen können. Zwei Sprachen zur Inter-Agentenkommunikation sind besonders hervorzuheben: die Knowledge and Query Manipulation Language (KQML) im Rahmen des Knowledge Sharing Efforts an der University of Maryland und die Agent Communication Language der Foundation for Intelligent Physical Agents (FIPA ACL). Diesen beiden Sprachen ist jeweils ein Abschnitt dieses Kapitels gewidmet, in denen ein Überblick über die Konzepte ihrer interessanten und für die weitere Entwicklung von Agenten wichtigen Ansätze gegeben wird.

4.1. Kommunikationsverfahren

Es gibt für Programme verschiedene Möglichkeiten, mit anderen Programmen in Kontakt zu treten. Die anderen Programme können sich auf einem entfernten Rechner befinden, der über Kommunikationskanäle innerhalb eines Netzwerks erreichbar ist. Eine Übersicht über die verschiedenen Kommunikationsverfahren liefert [Sattler 98], an der sich dieser Abschnitt orientiert. Im Rahmen der klassischen Client/Server-Architekturen können Programme als Dienstnehmer rechnerübergreifend mittels des entfernten Prozeduraufrufs (remote procedure call, RPC) die Dienste anderer Programme in Anspruch nehmen. Als objektorientierte Erweiterung wird der *Object Request Broker* (ORB) angesehen. Ein Dienst wird nicht mehr als Prozedur, sondern als Objekt angeboten. Während beim RPC eine enge Kopplung zwischen Client und Server vorliegt, übernimmt beim ORB ein Vermittler (Broker) die Zuordnung einer Dienstanfrage, indem er einen geeigneten Dienstgeber ermittelt und die Anfrage an ihn weiterleitet. Bekannt geworden ist dieses Verfahren insbesondere durch die Common Object Request Broker Architecture der Object Management Group [CORBA]. Statt Kommunikation zwischen einem Absender und einem Empfänger gibt es auch noch Verfahren zur Gruppenkommunikation über einen zentralen Kommunikationskanal, bei denen auch mehrere Sender bzw. Empfänger beteiligt sein können. Die bekanntesten Verfahren sind einerseits der Multicast, bei dem ein Sender nur an eine bestimmte Gruppe von Empfängern Nachrichten verschickt, und andererseits der Broadcast, bei dem an alle Empfänger, die den Kanal abhören, gesendet wird.

Aus der VKI stammt der Ansatz, über einen gemeinsamen Arbeitsbereich, das sogenannte *Blackboard*, indirekt zu kommunizieren, indem jeder Agent Daten in einem bestimmten Bereich ablegen kann und anschließend alle anderen auf diese Daten zugreifen dürfen. Durch diesen zentralisierten Ansatz treten allerdings auch Probleme mit der Netzwerklast auf, wenn viele Agenten auf diesen Bereich zugreifen wollen. Für viele Anwendungen ist es auch gar nicht nötig, dass Daten mehreren Empfängern zugänglich gemacht werden müssen, sondern oft kann der Austausch der benötigten Informationen durch direkte Kommunikation erfolgen. Diese Form der Kommunikation wird im Folgenden näher betrachtet.

Nachrichten- und dialogbasierte Kommunikation. Der direkte Versand von Nachrichten stellt sich als intuitives und praktikables Verfahren zur Kommunikation zwischen Agenten dar. Zunächst muss man zwischen drei Ebenen unterscheiden, die für das Versenden von Nachrichten und Verstehen beim Empfänger nötig sind:

1. Auf der untersten Ebene dieser Betrachtungsweise von Kommunikation steht das *Transportprotokoll*, z.B. TCP, SMTP oder HTTP.
2. Die *Kommunikationssprache* legt im Wesentlichen den Typ der Nachricht fest. Es kann sich bei einer Nachricht z.B. um eine Bitte, eine Anfrage oder eine Antwort handeln. Wichtig ist zu beachten, dass diese Kommunikationssprache über den eigentlichen Inhalt von Nachrichten keine Angaben macht. Sie bietet nur eine wohldefinierte Hülle für den Transport der Nachricht zum Empfänger an, dessen Aufgabe es bleibt, den Inhalt der Nachricht richtig zu deuten. Kommunikationssprachen können lediglich helfen, dass Nachrichten beim Empfänger verstanden werden, indem sie in der Nachricht zusätzliche Angaben über die verwendete Sprache des Inhalts und die verwendeten Konzepte machen.

3. Auf der höchsten Ebene wird durch *Interaktionsprotokolle* der Ablauf von Dialogen zwischen Agenten festgelegt. Es kann sich dabei z.B. um komplexe Verhandlungsabläufe mit mehreren Anfrage- und Antwortstufen handeln.

Es ist ein großes Problem, dass Nachrichten nicht nur aus einfachen Aussagen bestehen, sondern oft implizit mit Aktionen und Absichten verbunden sind, z.B. die Frage: „Weißt du den Preis der Siemens-Aktie?“ Der gefragte Agent soll nicht einfach Ja oder Nein antworten, sondern die Absicht dieser Frage erkennen: Der Sender möchte den Preis einer Siemens-Aktie erfahren. Und selbst wenn der Agent dies erkennen kann, stellt sich immer noch die Frage, welchen Preis er nennen soll, denn die Aktie wird an verschiedenen Börsen gehandelt und hat dort in der Regel unterschiedliche Notierungen. Mit Fragestellungen dieser Art beschäftigt sich die Sprechakttheorie [Austin 62]. Ein *Sprechakt* bezeichnet generell eine Nachricht, aufgrund der die Umgebung verändert werden kann. Im Zusammenhang mit Agenten bedeutet dies, dass Nachrichten zunächst Auswirkungen auf den (mentalen) Zustand des Empfängers haben und sich in Aktionen des Agenten fortsetzen können. Ein Sprechakt besteht aus drei Komponenten:

1. Der *lokutionäre Akt* bezeichnet den reinen Übermittlungsvorgang der Nachricht.
2. Der *illokutionäre Akt* beschreibt die mit der Nachricht verfolgte Absicht (z.B. eine Frage, eine Bitte oder ein Befehl).
3. Der *perlokutionäre Akt* gibt die Auswirkungen auf die Umwelt an.

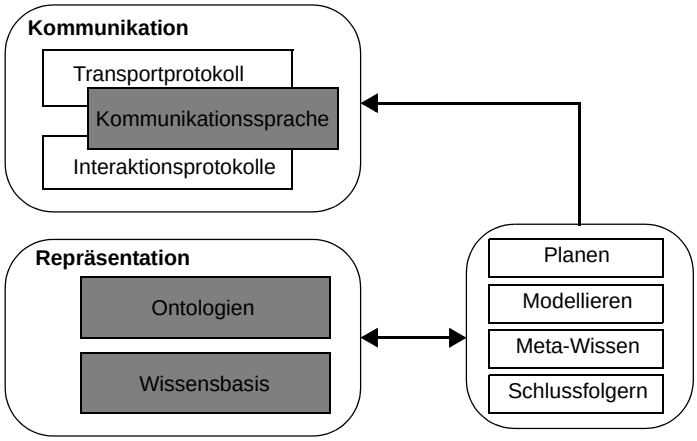
Insbesondere die illokutionären Akte sind Gegenstand der Untersuchung bei der Entwicklung von Kommunikationssprachen für Agenten. Da Nachrichten aber oft im Zusammenhang mit anderen Nachrichten stehen (z.B. Antworten auf Fragen), werden auch Dialogabläufe zwischen zwei oder mehreren Beteiligten untersucht und zum Teil vordefiniert. In den folgenden Abschnitten werden die beiden bekanntesten Kommunikationssprachen für Agenten vorgestellt. Im Anschluss daran werden diese beiden Sprachen gegenübergestellt und es wird auf ihre Verwendbarkeit für die Modellierung des Farbsortierspeichers eingegangen.

4.2. KQML

Die Knowledge Query and Manipulation Language (KQML) wurde im Rahmen des ARPA Knowledge Sharing Efforts (KSE) an der University of Maryland entwickelt. KQML legt eine Reihe von Nachrichtentypen fest, die auch *performatives* genannt werden und die Basis für eine Modellierung von komplexeren Agenten-Interaktionen bilden. Die Semantik der Nachrichtentypen wird allerdings nur informell beschrieben. Außerdem gibt es noch eine Klasse von Hilfsagenten (*communication facilitators*), die als koordinierende Vermittler zwischen Agenten eingesetzt werden können. Für die weitere Beschreibung von KQML wird eine Sichtweise für das Agentenmodell übernommen, die in Abbildung 12 dargestellt ist und zwischen den folgenden drei Komponenten unterscheidet [Finin 95]:

1. Die Kommunikationskomponente besteht aus den drei Teilen Transportprotokoll, Kommunikationssprache und Interaktionsprotokolle, die schon im vorherigen Abschnitt angesprochen wurden. Im weiteren Verlauf der Betrachtungen steht die Kommunikationssprache und hier insbesondere KQML im Vordergrund.

Abbildung 12 Modell für interagierende Agenten



2. Die Repräsentationskomponente besteht aus einer Wissensbasis und aus Ontologien. Für die Modellierung von Wissen wird vom KSE ein spezielles Format vorgeschlagen (knowledge interchange format, KIF), das eine Prädikatenlogik erster Ordnung in Präfixnotation mit einigen Erweiterungen (z.B. logische Operationen zum Ausdrücken von Regeln) darstellt [Genesereth 91]. Zum Beispiel wird durch den folgenden Ausdruck eine Schlussfolgerungsregel angegeben:

```
( =>    ( and (real-number ?x) (even-number ?n) )
        ( > (expt ?x ?n) 0 ) )
```

   Diese Regel gibt an, dass das Ergebnis der Potenzierung einer Zahl x mit einer geraden Potenz n immer positiv ist. Doch zur Realisierung von gemeinsamen Wissen bedarf es noch mehr. Jedes wissensbasierte System baut auf der Modellierung seiner Umwelt durch eine formale Repräsentation auf. Die Konzepte für diese Modellierung müssen für zusammenarbeitende Agenten zum gegenseitigen Verständnis aber dieselben sein, also müssen diese Konzepte agentenübergreifend spezifiziert werden und den beteiligten Agenten bekannt sein. Solche Spezifikationen werden auch Ontologien genannt, für die in [Gruber 93] eine genauere Beschreibung gegeben wird.

   Für die Beispielfrage „Weißt du den Preis der Siemens-Aktie?“ kann als Zusatzattribut zur Nachricht z.B. die Ontologie „Börse Frankfurt“ angegeben werden, die alle an der Frankfurter Börse gehandelten Aktien und Wertpapiere beschreibt. Damit weiß der Empfänger, in welchem Kontext er diese Anfrage behandeln muss.

3. In der dritten Komponente werden die Teile aufgeführt, die nicht direkt mit der Interaktion zwischen Agenten in Zusammenhang stehen. Dieser Teil ist nicht Gegenstand der Untersuchung im Rahmen von KQML.

Struktur von KQML-Nachrichten. KQML kann in drei Schichten aufgeteilt werden: die Inhaltsschicht, die Nachrichtenschicht und die Kommunikationsschicht.

In der Inhaltsschicht wird der eigentliche Inhalt der Nachricht abgelegt, der applikationsabhängig ist und nicht weiter durch KQML bestimmt wird. Vorgeschlagen wird

aber die Verwendung von KIF mit dem folgenden Hintergrund: Ein Agent A wandelt den gewünschten Inhalt seiner Nachrichten aus seiner Wissenssprache L_A in KIF um, verschickt den so umgewandelten Inhalt, und der Empfänger wandelt den Nachrichteninhalt von KIF in seine Wissenssprache L_B um. KIF wird also als Zwischensprache benutzt, um die direkte Übersetzung von L_A in L_B zu umgehen.

In der Kommunikationsschicht werden Angaben über den Nachrichtenversand gemacht, z.B. Angaben über den Sender und Empfänger in Form einer eindeutigen Kennung oder ein Betreff, mit der eine Referenz für Antworten übergeben werden kann.

Die Nachrichtenschicht bildet den Kern von KQML und beschreibt die Möglichkeiten zur Interaktion mit Agenten. Hier wird der Sprechakt für eine Nachricht festgelegt und es können hilfreiche Angaben über den Inhalt der Nachricht gemacht werden, z.B. kann eine Ontologie angegeben werden. Eine KQML-Nachricht ist syntaktisch sehr einfach aufgebaut; die einzelnen Bestandteilte der Nachricht werden beginnend mit dem Nachrichten- bzw. Sprechakttypen einfach aufgelistet. Ein Einlagerungsagent inRobot könnte eine Anfrage nach einem neuen Auftrag in KQML so formulieren:

```
(ask-one
    :sender      inRobot
    :content     (ORDER car ?bodyType ?color)
    :receiver    orderAgent
    :reply-with  getOrder
    :language    LPROLOG
    :ontology    FSS
)
```

Diese Nachricht benutzt den Nachrichtentyp ask-one, d.h. es wird eine Anfrage gestellt und *eine* Antwort aus der Menge möglicher Antworten zurückgegeben. Der Inhalt der Nachricht ist in LPROLOG formuliert und bedeutet, dass nach einem Auftrag gefragt wird und zwei Ergebnisparameter (bodyType und color) gebunden werden sollen. Als Ontologie kann FSS angegeben werden, wodurch festgelegt wird, dass diese Anfrage im Kontext des Farbsortierspeichers gestellt wird. Wenn der Auftragsagent auch noch Aufträge für andere Produktionsprozesse verwalten würde, ist diese Angabe sehr wichtig. Angenommen, der Auftragsagent würde auch noch die Aufträge für die Endmontage verwalten. Für diese Aufträge können Anfragen gestellt werden, die in LPROLOG aussehen wie in dem obigen Beispiel, aber ganz anders behandelt werden müssen.

Als Anwort auf die oben gestellte Anfrage könnte der Auftragsagent folgende Antwort in KQML formulieren, mit der er den Auftrag vergibt, eine Karosserie vom Typ sedan in der Farbe green zu lackieren:

```
(tell
    :sender      orderAgent
    :content     (ORDER car sedan green)
    :receiver    inRobot
    :in-reply-to getOrder
    :language    LPROLOG
    :ontology    FSS
)
```

Es gibt in KQML eine ganze Reihe vordefinierter Sprechakttypen, von denen einige in der folgenden Tabelle aufgeführt sind. Eine komplette Auflistung findet man in [DARPA 93]. Agenten, die KQML benutzen, müssen nicht alle Sprechakte implementieren, sondern nur die für ihre Aufgabe benötigten. Diese müssen sich dann aber an die vorgegebene Semantik halten und das ausführen, für was sie in KQML gedacht sind. Es ist auch möglich, weitere Nachrichtentypen für eigene Applikationen zu definieren.

Kategorie	performatives
Information	tell, achieve, cancel, untell
Anfrage	evaluate, ask-if, ask-one, ask-all
Antwort	reply, sorry
Netzwerk	register, unregister, forward, broadcast, route
Fähigkeitsdefinition	advertise, subscribe, monitor

Facilitatoren. Mit den vordefinierten Nachrichtentypen sind verschiedene Protokolle zum Austausch von Informationen erstellt worden, von denen hier nur einige vorgestellt werden sollen. Dazu wird ein einfacher zusätzlicher Agent benötigt, der Facilitator genannt wird und die Nachrichtenvermittlung übernimmt. Ein Facilitator verwaltet eine Liste von Diensten anderer Agenten und leitet Nachrichten an Dienstgeber weiter.

Das Ausgangsproblem der folgenden Szenarien soll das Interesse eines Agenten A an der Wahrheit des Fakts x sein. Agent B weiß über x die Wahrheit, und der Facilitator F ist für beide Agenten verfügbar. Wenn A den Agenten B kennt, kann er direkt bei B anfragen. Ist dies aber nicht der Fall, kann A auf verschiedene Weise Kenntnis darüber erlangen, ob x wahr ist. In Abbildung 13 benutzt A ein subscribe, sodass F im Folgenden darauf achtet, dass A die Wahrheit über das Fakt x erfährt, sobald F selbst über x Bescheid weiß. Wenn B irgendwann F das Fakt x mitteilt, dann kann F dies an A weiterleiten.

Abbildung 13 Informationsbeschaffung über einen Facilitator

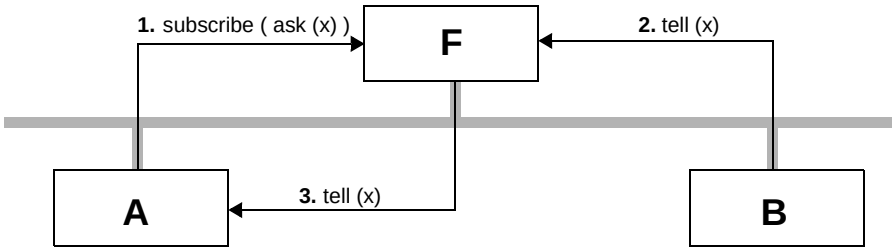
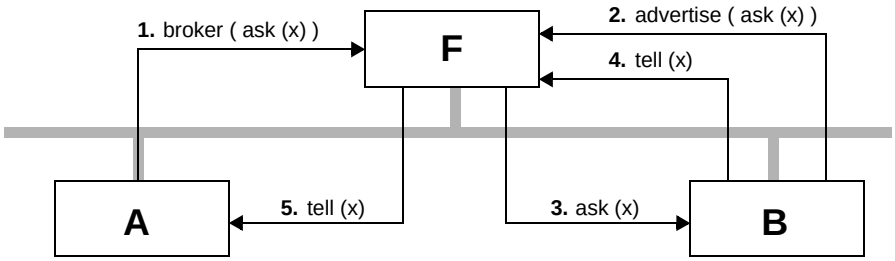


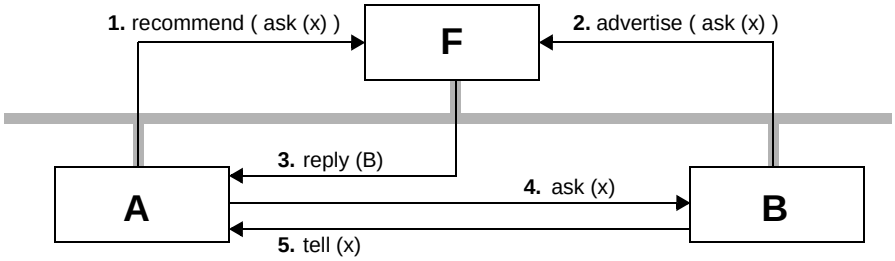
Abbildung 14 zeigt eine andere Situation: Hier fragt A bei F mittels einer broker-Nachricht nach einem Agenten an, der die Anfrage ask(x) behandeln kann. Unabhängig davon teilt B dem Facilitator durch ein advertise mit, dass B bereit ist, Anfragen über x zu beantworten. Wenn F diese Nachricht von B bekommt, kann er die Anfrage von A an B weiterleiten, bekommt daraufhin eine Antwort von B und gibt sie an A zurück. Statt broker(ask(x)) könnte Agent A auch recruit(ask(x)) benutzen. In diesem Fall schickt B seine Antwort nicht an den Facilitator, sondern direkt an A.

Abbildung 14 Facilitator als Broker



Schließlich kann Agent A auch noch über den Facilitator mittels recommend einen Agenten erfahren, der bereit ist, ihm x mitzuteilen. Dann kann A direkt Kontakt zu diesem Agenten aufnehmen und braucht im Folgenden nicht mehr den Umweg über F zu gehen, um Anfragen über x und evtl. auch weitere Fakten zu stellen (Abbildung 15).

Abbildung 15 Facilitator empfiehlt einen Agenten



Es bleibt noch zu klären, wie Agenten zu Beginn ihren ersten Facilitator finden, um von ihm aus andere Agenten und auch weitere Facilitatoren kennenlernen zu können: Einerseits kann man einen zentralen Facilitator einsetzen, der allen Agenten von Beginn an bekannt ist, andererseits kann man auch allen Agenten eine Funktion zur Verfügung stellen, mit der sie in der Lage sind, einen lokalen Facilitator zu ermitteln und sich dort zu registrieren.

Anwendungen mit KQML. Während Anfang der 90er-Jahre Experimente mit KQML zur Zusammenführung existierender Programme schon vielversprechende Ergebnisse brachten, steht inzwischen mit JATLite von der Stanford University ein KQML-basiertes System zur Verfügung, das auch eine Infrastruktur für Multi-Agentensysteme liefert [JATLite]. An dieser Stelle seien nur kurz einige Projekte erwähnt, die KQML benutzen.

SIPE ist ein von der ARPA Rome Lab Planning Initiative entwickeltes Planungs- und Schedulingsystem für Transportlogistik im militärischen Umfeld. In diesem Projekt wurden ein planender Agent, ein Scheduler, eine Wissensbasis und eine Schlussfolgerungskomponente zusammengeführt. Alle Komponenten gab es schon als Einzelprogramme, bevor sie in SIPE eingebaut wurden [Bienkowski 94].

Ein weiteres Projekt ist der Zusammenschluss mehrerer Programme zu einem Informationsagenten. Benutzt wurden CoBASE, ein kooperatives Front-end-System, SIMS, ein Programm zur Planung von Zugriffen auf Informationen, und LIM, das zur Umwandlung von relationalen Daten in Wissensstrukturen dient. Mit CoBASE wird eine Anfrage behandelt, und falls keine Antworten gefunden werden, wird die Anfrage mittels spezieller Approximationsmechanismen leicht verändert erneut gestellt. SIMS bekommt von CoBASE die Anfrage und benutzt Wissen über verschiedene Quellen, um die Anfrage optimal auf die verschiedenen Informationsquellen zu verteilen. Die Ergebnisse der so verteilten Anfragen werden zu LIM geschickt, wo mit diesen Ergebnissen Objekte aus den Tupeln einer Datenbank gebildet werden, die die ursprüngliche Frage beantworten [Finin 95].

4.3. FIPA ACL

FIPA ist eine Standardisierungsinitiative, die erst 1996 gegründet wurde und der eine Reihe namhafter Firmen und Forschungsinstitute aus verschiedenen Ländern angehören [FIPA]. Gegenstand der Betrachtung von FIPA ist die Schnittstelle zwischen Agenten und ihrer Umgebung, d.h. Menschen, physikalischer Umwelt (Hardware), anderen Agenten und weiterer Software.

In der ersten Spezifikation aus dem Jahre 1997 ist eine Einteilung in 3 normative und 4 informative Sektionen vorgenommen worden, aus der die folgenden Beschreibungen entnommen sind [FIPA 97]. Der normative Teil beschäftigt sich mit der Agentenverwaltung, der Agentenkommunikation untereinander und der Integration von existierender Software in Agentensysteme. Der informative Teil behandelt ausgewählte Anwendungsbeispiele, die erst formal beschrieben und dann im Rahmen von Feldstudien implementiert werden. Die Erfahrungen aus diesen Projekten sollen dann wieder der Spezifikation zugute kommen [Steiner 98]. Im Folgenden werden die Kommunikationssprache FIPA ACL und die damit zusammenhängenden Interaktionsprotokolle aus dem normativen Teil der FIPA-97-Spezifikation näher betrachtet.

Struktur von FIPA ACL-Nachrichten. FIPA ACL basiert auf einer formalen Grundlage, der *semantic language* (SL), und benutzt zur Spezifikation von Nachrichten wie KQML auch Sprechakte, die hier *communicative acts* (CAs) genannt werden. Es wird eine Reihe von Anforderungen an Agenten gestellt, um als FIPA ACL-konform zu gelten. Einige grundlegende Anforderungen sind:

1. Es soll eine not-understood-Nachricht verschickt werden, wenn der Agent nicht in der Lage ist, den Inhalt einer Nachricht zu verstehen. Agenten sollen in der Lage sein, not-understood-Nachrichten zu empfangen und handzuhaben.
2. Der Agent kann eine beliebige Teilmenge der vordefinierten CAs und Interaktionsprotokolle implementieren. Die Implementierung muss aber mit der semantischen Definition der FIPA-Spezifikation in Einklang stehen.
3. Der Agent muss in der Lage sein, Nachrichten in FIPA ACL-Syntax zu erstellen, und umgekehrt auch Zeichenketten, die in korrekter FIPA ACL-Syntax empfangen werden, als entsprechende Nachricht zu verstehen.

Es können natürlich auch noch weitere CAs und Interaktionsprotokolle definiert werden, aber dann muss der Agent sicherstellen, dass die Nachrichten bei den Empfän-

gern verstanden werden. Die FIPA ACL-Syntax ähnelt der Struktur von KQML-Nachrichten sehr, wie das folgende Beispiel zeigt:

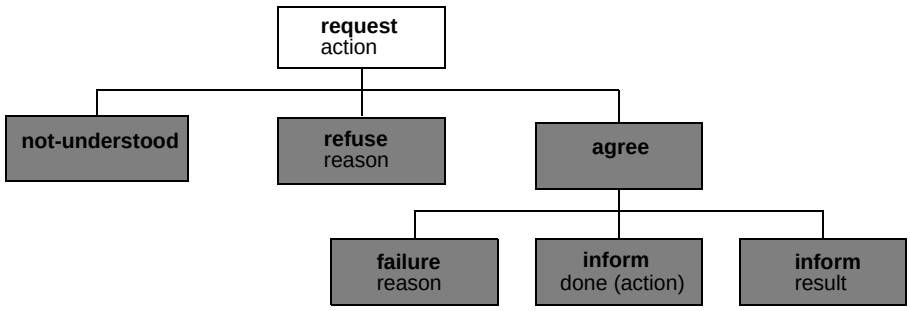
```
(inform
    :sender     inRobot
    :receiver   orderAgent
    :content    ( getOrder (bodyType color) )
    :reply-with getOrder17
    :language   SL
    :ontology   FSS
)
```

Es gibt noch weitere Parameter, z.B. kann man mit dem Schlüsselwort protocol zu einer Nachricht noch das Interaktionsprotokoll angeben, das gerade durchgeführt wird. Die einzelnen CAs werden von der FIPA formal in SL spezifiziert und zusätzlich informell beschrieben. Die folgende Tabelle listet einige CAs auf.

Kategorie	Communicative Acts
Informationsübermittlung	inform, confirm, disconfirm
Anfrage nach Informationen	query-if, query-ref, subscribe
Verhandlung	cfp (call-for-proposal), propose, accept-proposal, reject-proposal
Aktionsausführung	request, agree, cancel, refuse
Fehlerbehandlung	failure, not-understood

Interaktionsprotokolle. Von der FIPA werden auch typische Dialogabläufe zwischen Agenten als sogenannte Interaktionsprotokolle (IPs) spezifiziert, die durch Sequenzen von CAs angegeben werden. Auch hier ist eine applikationsabhängige beliebige Erweiterung erlaubt, aber bei den vordefinierten IPs sollen sich die Agenten an die vorgegebene Semantik der FIPA halten. Im Folgenden sollen exemplarisch zwei Interaktionsprotokolle vorgestellt werden. In den weißen Feldern stehen die CAs des initiierenden Agenten, in den grauen Feldern die möglichen Aktionen der anderen beteiligten Agenten.

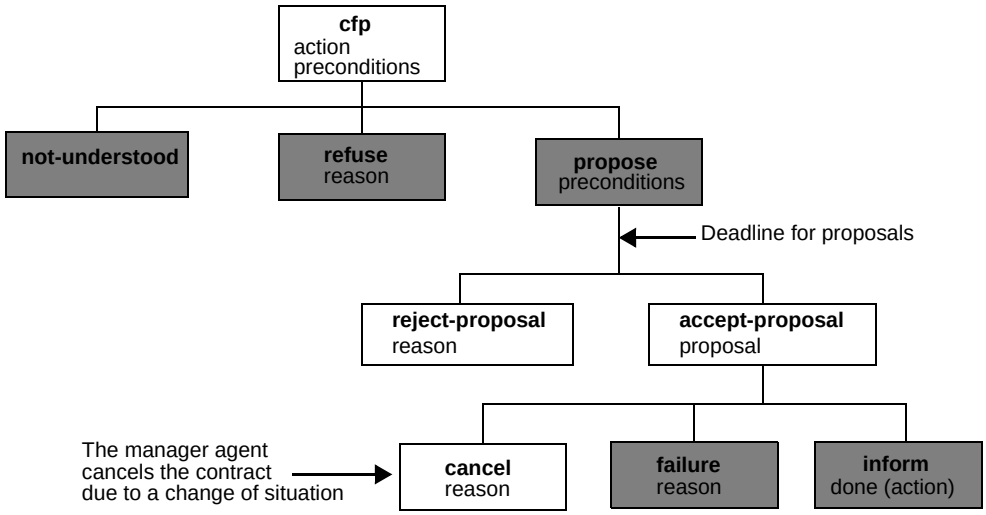
Abbildung 16 FIPA-request protocol



Der Ablauf einer einfachen Anfrage mit Antwort wird als Interaktionsprotokoll wie in Abbildung 16 dargestellt spezifiziert. Der initiierende Agent schickt die Bitte (request) an einen anderen Agenten, eine bestimmte Aktion (action) auszuführen. Der empfangende Agent versteht diese Nachricht entweder nicht und schickt eine not-understood-Nachricht zurück, oder er kann diese Bitte unter Angabe eines Grundes ablehnen (refuse), oder er ist mit der Ausführung der Aktion einverstanden. Im letzteren Fall sendet er erst eine Einverständnisnachricht (agree) und nach Ausführung der Aktion noch eine Nachricht über Erfolg (inform done), Misserfolg (failure) oder ein gewünschtes Ergebnis (inform result) an den Initiatoragenten zurück.

Ein weiteres interessantes Protokoll ist das *Kontraktnetz-Protokoll (contract-net protocol)*, das ursprünglich von Smith und Davis entwickelt worden ist [Smith 80] und von der FIPA leicht abgewandelt wurde, indem die CAs reject-proposal und accept-proposal eingefügt wurden (vgl. Abbildung 17).

Abbildung 17 FIPA contract-net protocol



Ein Agent möchte eine Aufgabe von einem oder mehreren anderen Agenten erledigen lassen und übernimmt für die folgende Verhandlung über die Vergabe dieser Aufgabe die Rolle des Managers. Die Aufgabe wird in der Regel durch eine Funktion charakterisiert, für die ein optimales Ergebnis gefunden werden soll. Ein anschauliches Beispiel ist die Suche nach dem günstigsten Anbieter eines Dienstes: In diesem Fall ist die Funktion der Preis für einen bestimmten Dienst, und das gesuchte Optimum ist der niedrigste der für diesen Dienst angebotenen Preise.

Der Manager schickt an andere Agenten eine call-for-proposal-Nachricht (cfp) ab, die eine Aktion und gewisse Bedingungen an die Ausführung der Aktion enthält, z.B. welchen Preis sie für die Ausführung der Aktion verlangen. Die empfangenden Agenten können daraufhin Vorschläge an den Manager zurückschicken (propose) oder auch unmittelbar erklären, dass sie nicht zur Ausführung der Aktion gewillt sind (refuse). Nachdem der Manager von den beteiligten Agenten eine Antwort erhalten hat, wählt er aus diesen Vorschlägen die besten aus. Es können dabei mehrere, einer oder keiner der Agenten ausgewählt werden. Die nicht ausgewählten Agenten erhalten vom Manager eine reject-proposal-Nachricht und scheiden für den weiteren Ablauf des Protokolls aus.

Die ausgewählten Agenten müssen nach Erhalt einer accept-proposal-Nachricht die gewünschte Aktion ausführen. Abschließend erfolgt eine Nachricht über Erfolg oder Misserfolg der Aktion an den Manager.

Aus unterschiedlichen Gründen kann es passieren, dass der Manager nicht von allen beteiligten Agenten Antworten auf seine cfp-Nachricht bekommt. Damit er in diesen Fällen nicht unendlich lange wartet, wird ein Zeitlimit für die Rücksendung von Antworten angegeben. Vorschläge, die nach Ablauf des Zeitlimits beim Manager eintreffen, werden nicht mehr berücksichtigt. Als Erweiterung gibt es noch das *iterierte contract-net protocol*, bei dem nach reject-proposal-Nachrichten erneut Anfragen für Vorschläge gestellt werden.

4.4. Vergleich von KQML und FIPA ACL

Die Nachrichtentypen beider Sprachen überdecken sich teilweise, auch wenn sie unterschiedlich benannt sind. Die folgende Tabelle kategorisiert die Nachrichtentypen in einer etwas anderen Form als in den vorherigen Abschnitten, um die bedeutendsten Unterschiede zwischen KQML und FIPA ACL herauszustellen. In der Tabelle werden aber aus Übersichtlichkeitsgründen nicht alle Nachrichtentypen aufgelistet.

Kategorie	KQML	FIPA ACL
Agentenmanagement	register, unregister, broker, advertise, recommend	Keine entsprechenden vordefinierten CAs. Stattdessen werden hierfür rein sprechaktbasierte Nachrichtentypen wie request oder inform und ein gesondertes Management-System (AMS) verwendet (vgl. Abschnitt 5.3.3.: MECCA).
Kommunikations-management	broadcast, forward, route	Keine entsprechenden vordefinierten CAs, sondern Übertragung solcher Aufgaben an das AMS
Behandlung mehrerer Antworten	ask-one (Sender fragt nach einer Anwort)	Keine entsprechenden vordefinierten CAs. Die FIPA ist der Ansicht, dass die Behandlung mehrerer Antworten auf der Ebene der Inhaltssprache liegt, z.B. wenn ein Agent einen Generator wie in KQML anfordert, sollte dies über die anwendungsspezifische Inhaltssprache geschehen. Vom Generator werden dann über requests Informationen angefordert.
	ask-all (Sender fragt nach allen Antworten)	
	stream-all (Alle Antworten, jede in einer eigenen Nachricht)	
	standby, ready, next, rest, destroy (Sender bittet den Empfänger, einen Antwortgenerator zu erstellen. Mit den hier angegebenen performatives kann der Generator manipuliert werden)	
Rein sprechaktbasierte Nachrichtentypen	ask-if	query-if
	tell, untell	inform
	insert, uninsert, delete-one, delete-all	In ACL ist keine direkte Manipulation des Wissens anderer Agenten erlaubt.
	subscribe	subscribe
	error	not-understood
	sorry	refuse, failure
	In KQML sind keine expliziten Nachrichtentypen zur Verhandlungsführung vorgesehen.	cfp, propose, accept-proposal, reject-proposal

KQML ist schon viel früher entworfen worden als FIPA ACL und es gibt eine Reihe von Anwendungen, in denen sich KQML bewährt hat. Leider haben sich durch die fehlende grundlegende formale Semantik viele Dialekte von KQML entwickelt, sodass keine einheitliche Behandlung von KQML-Nachrichten bei Agenten verschiedener Entwickler gewährleistet ist. Im Gegensatz dazu wird bei der FIPA-Spezifikation eine formale Semantik für die Nachrichtentypen zugrunde gelegt. Dies kann nur aufgrund eines bestimmten basierenden Agentenmodells gemacht werden und führt daher bei einigen Wissenschaftlern zu Kritik (vgl. [Labrou 97], [Wagner 97a]). Trotzdem können diese Bemühungen dazu führen, dass unterschiedliche Implementierungen von FIPA ACL eher kompatibel sind als es bei KQML der Fall ist. Leider ist FIPA ACL noch sehr jung und es sind noch keine Applikationen, die FIPA ACL benutzen, frei verfügbar. Im Rahmen meiner Diplomarbeit habe ich aber die Möglichkeit, das MECCA-System der Siemens AG, München (Abteilung ZT IK 6) zu benutzen. Dieses System ist FIPA-konform und wird im nächsten Kapitel genauer vorgestellt.

4.5. Anwendbarkeit auf den Farbsortierspeicher

Für den Farbsortierspeicher sind beide Sprachen generell anwendbar, denn die benötigten grundlegenden Sprechakte für die Durchführung der Einlagerung sind sowohl in KQML als auch in FIPA ACL enthalten: Anfragen, Antworten und Informieren. In der folgenden Tabelle sind die bei der Einlagerung wichtigsten Nachrichten aufgelistet und mit möglichen Sprechakten in KQML und FIPA ACL angegeben. Die angegebenen Sprechakte sind nur ein möglicher Ansatz zur Modellierung der Nachrichten, denn oft sind auch alternative Sprechakte möglich (z.B. tell statt reply).

Agent	Nachricht (in Klammern die gewünschte Aktion)	Parameter	KQML	FIPA ACL
Einlagerungsagent (inRobot)	Anfrage nach einem Auftrag beim Orderagenten (getOrder)	Robotertyp (robotType)	ask-one	request
	Anfrage nach einer Karosserie beim Depot (getBody)	Karosserietyp (bodyType)	ask-one	request
	Anfrage nach einer Sortierlinie beim Sortierlinienbroker (getSortline)	Farbe (color)	broker	request
Sortierlinienbroker (sortlineBroker)	Anfrage über das Verlangen nach einer Farbe (getDesire)	Farbe (color)	ask	cfp
	Antwort auf eine Anfrage „getSortline“ (takeSortline)	Sortierlinie (sortline)	tell	inform
Orderagent (orderAgent)	Antwort auf eine Anfrage „getOrder” (takeOrder)	Karosserietyp, Farbe (bodyType, color)	reply	inform
	Wichtiger Auftrag mit höchster Priorität (importantOrder)	Karosserietyp, Farbe (bodyType, color)	achieve	request
Depotagent (depotAgent)	Antwort auf eine Anfrage „getBody“ (takeBody)	Karosserietyp (bodyType)	reply	inform
	Warnmeldung an den Orderagenten (notEnoughBodies)	Karosserietyp (bodyType)	tell	inform
Sortierlinienagent (sortline)	Antwort auf eine Anfrage „getDesire“ (tellDesire)	Wert, Farbe (desire, color)	reply	propose, inform

In der Tabelle taucht ein Agent auf, der nicht in der Modellierung des Farbsortierspeichers in Kapitel 2 genannt wird: ein *Sortierlinienbroker.* Dieser Agent ist in KQML-Sichtweise ein Facilitator als Broker, und im Falle von FIPA ACL ein Manager im Kontraktnetz-Protokoll. Der Sortierlinienbroker wurde bei der Modellierung nicht mit angegeben, doch bei der Formulierung der konkreten Nachrichten stellt sich jetzt heraus, dass eine zusätzliche Komponente zur Bestimmung einer Zielsortierlinie sehr sinnvoll ist.

Beide Sprachen eignen sich für den Transport der Nachrichten zwischen den Agenten des Farbsortierspeichers und bieten sogar Unterstützung, wenn Dialoge zwischen mehr als zwei Agenten geführt werden müssen, daher fällt eine Wahl für eine der beiden Sprachen nicht leicht. Die vordefinierten Interaktionsprotokolle, und hier insbesondere das Kontraktnetz-Protokoll, geben letztendlich den Ausschlag, FIPA ACL gegenüber KQML vorzuziehen.

Mit dem Kontraktnetz-Protokoll kann ein Einlagerungsagent auf folgende Weise das Verlangen der Sortierlinien nach bestimmten Farben erfahren: Der Einlagerungsagent fragt beim Sortierlinienbroker an, zu welcher Sortierlinie er einen Auftrag mit der Farbe X bringen soll. Der Sortierlinienbroker agiert im Folgenden als Manager des Kontraktnetz-Protokolls und ermittelt von allen Antworten der Sortierlinienagenten die Linie, deren Verlangen nach der Farbe am höchsten ist und teilt sie dem Einlagerungsroboter mit. Der Sortierlinienbroker soll keine weiteren Aufgaben übernehmen, sondern über diese Abwicklung hinausgehende Verhandlungen sollen durch direkten Austausch von Nachrichten zwischen den Sortierlinienagenten oder ggf. über weitere noch zu spezifizierende Agenten stattfinden.

KAPITEL 5

Multi-Agentensysteme

Die Agententechnologie wird bisher vornehmlich in Forschung und Industrie untersucht und eingesetzt. Doch sie stößt auch auf ein immer größeres Interesse in der breiten Öffentlichkeit. Zu Beginn dieses Jahres hat zum Beispiel die Firma Sun mit der Java Intelligent Network Infrastructure (Jini) ein neues System vorgestellt, über das auch in der Tagespresse berichtet wurde [Spehr 99]. Mit Jini soll die Vision wahr werden, dass schon bald Haushaltsmaschinen und Geräte der Unterhaltungselektronik mit Agententechnologie ausgestattet sind und leicht vernetzt werden können.

In diesem Kapitel werden die Vorteile von Multi-Agentensystemen (MAS) näher erläutert sowie die Bedeutung von Kooperation in MAS dargestellt. Da die Makrosicht für Agenten nicht im Vordergrund dieser Diplomarbeit steht, wird allerdings keine Analyse der verschiedenen theoretischen Ansätze für MAS vorgenommen, sondern lediglich auf entsprechende Quellen verwiesen.

Es gibt mittlerweile eine solche Vielzahl sogenannter Frameworks für Agenten, dass eine vergleichende Übersicht dieser Programme im Rahmen dieser Arbeit nicht möglich ist. Daher werden nur die Konzepte einiger Agenten-Frameworks vorgestellt, mit denen ich mich während meiner Diplomarbeit beschäftigt habe. Diese Frameworks bieten eine Infrastruktur an, mit der man u.a. den Nachrichtenversand auf einer höheren Ebene abwickeln kann als mit den konventionellen Transportprotokollen. Darüber hinaus bieten Agenten-Frameworks weitere Dienste an, die gerade bei Gemeinschaften von Agenten sehr hilfreich sind, z.B. Verzeichnisdienste, die u.a. den aktuellen Aufenthaltsort mobiler Agenten verwalten, oder Informationsdienste, in denen die Fähigkeiten registrierter Agenten veröffentlicht werden. Es handelt sich also um Werkzeuge, die auf einer Ebene zwischen dem reinen Nachrichtentransport und Agentenapplikationen liegen; Frameworks werden daher auch als Middleware bezeichnet.

5.1. Vorteile

In diesem Abschnitt soll eine kurze Übersicht über die Vorteile von Multi-Agentensystemen gegeben werden.

Multi-Agentensysteme als offene Systeme. Multi-Agentensysteme werden oft auch als offene Systeme bezeichnet, die sich insbesondere durch die folgenden Merkmale auszeichnen [Burkhard 98]:

- kontinuierliche Bereitschaft
- Erweiterbarkeit
- dezentrale Steuerung
- asynchrone Arbeitsweise
- inkonsistente Informationen
- „Armlängen-Reichweite“ der zugehörigen Agenten

Diese Eigenschaften haben die folgende Bedeutung für Multi-Agentensysteme: Die kontinuierliche Bereitschaft eines Agentensystems beinhaltet, dass eigenständig agierende Agenten vorhanden sein müssen. Multi-Agentensysteme sind zur Laufzeit erweiterbar und können sich so den wachsenden Aufgaben dynamisch anpassen. Eine globale Steuerung erweist sich hierfür als eher hinderlich und statt Client/Server-Lösungen bietet sich die direkte Kommunikation und Kooperation gleichberechtigter Agenten an. Bei dieser Art der Zusammenarbeit ist eine globale Synchronisation nicht mehr erforderlich, sondern nur die an derselben Aufgabe beteiligten Agenten müssen sich aufeinander abstimmen. Die lokal in Agenten vorhandenen Informationen sind nicht immer aktuell, was zu inkonsistenten Informationen im Hinblick auf das Gesamtsystem führen kann. Dies ist jedoch als Eigenschaft des Systems und nicht als Nachteil anzusehen. Inkonsistente Informationen können durchaus berechtigte (z.B. datenschutzrechtliche) Gründe haben. Mit dem Begriff Armlängen-Reichweite wird ausgedrückt, dass Agenten nur mit den Komponenten des Systems zusammenarbeiten müssen, die zur Bearbeitung einer Aufgabe benötigt werden.

Vorteile von Multi-Agentensystemen. Den ständig wachsenden Anforderungen komplexer Systeme sind konventionelle Technologien immer weniger gewachsen. Daher wird seit einigen Jahren untersucht, wie das Modell des eigenständig handelnden Agenten hilfreich bei der Strukturierung komplexer Systeme eingesetzt werden kann. Agenten bieten eine intuitive Methode zur Abstraktion, die über das Modell objektorientierter Programmierung hinausgeht: Agenten werden als *aktive* Komponenten angesehen, die miteinander kommunizieren und kooperieren. (Genaueres zu Kooperation in MAS wird im nächsten Abschnitt behandelt.) In diesem Zusammenhang spricht man auch von der *Delegation von Aufgaben* an einzelne Agenten zur Abgrenzung gegenüber dem *Methodenaufruf* bei objektorientierter Programmierung.

Multi-Agentensysteme sind ein geeigneter Ansatz zur Modellierung von Systemen mit inhärent verteilten Informationen, wie z.B. bei Produktionsanlagen mit verschiedenen Bearbeitungsstationen. Durch ihre dezentrale Strukturierung eignen sich MAS für Probleme, die zu groß für ein einzelnes Programm sind; die Aufteilung einer Aufgabe auf verschiedene (nebenläufige) Komponenten kann die Aufgabenkomplexität reduzieren. Durch ihre Erweiterbarkeit können MAS im laufenden Betrieb an neue Anforde-

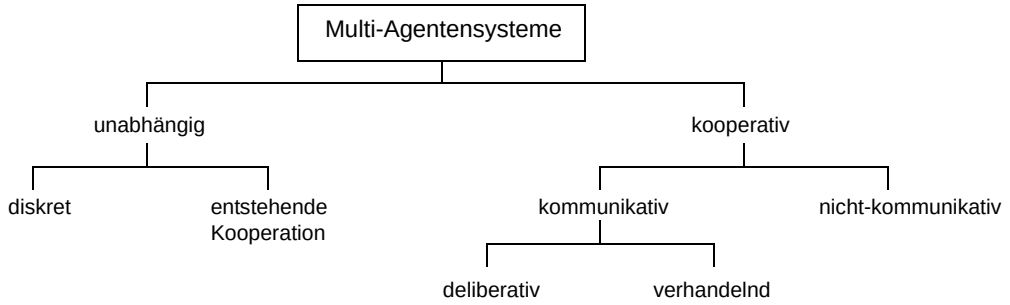
rungen angepasst werden und sind damit flexibler als konventionelle Systeme. Vor diesem Hintergrund kann insbesondere die Ablaufsteuerung in Produktionsanlagen vom Agentenansatz profitieren.

Diese und noch weitere Vorteile werden z.B. in [Jennings 96] ausführlich behandelt. Dort wird auch ein Abriss über die Entwicklung des agentenorientierten Ansatzes seit den 80er-Jahren gegeben sowie eine Reihe von Beispielapplikationen genauer beschrieben.

5.2. Kooperation in Multi-Agentensystemen

Um die Bedeutung von Kooperation in Multi-Agentensystemen einzuordnen, wird die in Abbildung 18 dargestellte Typologie betrachtet [Franklin 96]. Die angegebene Typologie erhebt nicht den Anspruch auf Vollständigkeit und auch die Grenzen der Unterteilung sind fließend. Dennoch dient sie zur Verdeutlichung des Zusammenhangs von Kooperation in MAS mit Konzepten wie Kommunikation, Koordination und Verhandlung.

Abbildung 18 Typologie von Kooperation in Multi-Agentensystemen



Ein Multi-Agentensystem wird als *unabhängig* bezeichnet, wenn jeder beteiligte Agent ausschließlich seine eigenen Ziele verfolgt. Das System heißt *diskret*, wenn es unabhängig ist und die Ziele jedes Agenten in keinem Zusammenhang mit den Zielen der anderen Agenten stehen. Dies ist zum Beispiel der Fall, wenn ein System aus den folgenden zwei Agenten besteht: Ein Agent wird zur Filterung von emails und der andere zum Beschaffen von Informationen aus dem Internet eingesetzt. Kooperation kann *entstehen*, ohne dass Agenten explizit die Absicht hierzu ausdrücken, z.B. wenn mehrere Roboter Bälle in einem Raum einsammeln. Für den Betrachter sieht es so aus, als ob die Roboter zusammenarbeiten, doch von der Agentensichtweise her sind die Roboter als unabhängige Agenten zu sehen, weil sie lediglich ihre eigenen Ziele verfolgen.

Den unabhängigen MAS stehen die *kooperativen MAS* gegenüber, in denen entweder *kommunikative* oder *nicht-kommunikative Kooperation* erfolgt. Bei letzterer koordinieren Agenten ihre Aktionen durch gegenseitige Beobachtung und entsprechende Reaktion auf das Verhalten anderer Agenten. Bei der kommunikativen Kooperation werden Nachrichten zwischen den Agenten ausgetauscht und es wird zwischen *deliberativen* und *verhandelnden Systemen* unterschieden. In deliberativen Systemen planen Agenten zusammen ihre Aktionen, um zu kooperieren. Bei dieser Art von Kooperation

kann auch eine Koordination der auszuführenden Aktionen erfolgen. In verhandelnden Systemen kommen gegenüber deliberativen Systemen zusätzlich noch Wettbewerbs- bzw. Konkurrenzaspekte hinzu.

Es gibt auch andere Sichtweisen über Kooperation in MAS, die u.a. in [Doran 97] und [Jennings 96] angesprochen werden. Doch anstatt genauer auf die unterschiedlichen Ansichten einzugehen möchte ich an dieser Stelle zur Beschreibung einiger Werkzeuge übergehen, die Unterstützung bei der Erstellung und Verwaltung von Agentengemeinschaften anbieten und damit die Grundlagen für Kooperation schaffen.

5.3. Agenten-Frameworks

In diesem Abschnitt werden einige Werkzeuge vorgestellt, die das Agentenmanagement in Multi-Agentensystemen übernehmen. Besonders stark entwickelt hat sich das Interesse an Frameworks, die *mobile* Agenten ermöglichen und verwalten können. In diesem Zusammenhang wird die Programmiersprache Java von vielen Frameworks bevorzugt: Agentenprogramme werden als Objekte angesehen und als plattformunabhängiger Bytecode zu anderen Rechnern in einem Rechnernetz, z.B. dem Internet, verschickt und dort ausgeführt.

Viele Seiten im WWW führen Listen der verfügbaren Werkzeuge zur Erstellung von Agenten und Multi-Agentensystemen auf, die teilweise von Unternehmen entwickelt wurden (z.B. [Concordia]) oder in Forschungseinrichtungen entstanden sind (z.B. an der Stanford University [JATLite]). Übersichten über weitere verfügbare Frameworks findet man u.a. auf den WWW-Seiten der Agent Society [AgentSociety] oder unter [MAS].

CORBA. Die Object Management Group, ein 1989 gegründetes internationales Konsortium mit heute über 600 Mitgliedern, hat es sich zur Aufgabe gemacht, einen Standard für verteilte interagierende Objekte zu entwickeln [OMG]. Mit Objekten sind zunächst beliebige Klasseninstanzen objektorientierter Programmiersprachen gemeint, jedoch stehen Objekte, die Agenten repräsentieren, im Vordergrund der folgenden Ausführungen. Die OMG hat u.a. eine Referenzarchitektur (Common Object Request Broker Architecture, CORBA) und eine Beschreibungssprache (Interface Definition Language, IDL) entwickelt, die inzwischen von vielen Frameworks unterstützt wird. Ein Object Request Broker (ORB) ist wie ein Facilitator anzusehen (vgl. Abschnitt 4.2.) und stellt die Verbindung zwischen Objekten her. Mit IDL kann man die Fähigkeiten von Objekten beschreiben und einem ORB mitteilen. Weiterhin werden in CORBA Basisdienste und Objektdienste angegebenen. Basisdienste werden in IDL beschrieben und definieren die Interaktion verteilter Objekte. Objektdienste betreffen Aufgaben wie z.B. Namengebung, Persistenz, Transaktionen, Lebenszyklusmanagement, Anfragen und Sicherheitsmechanismen. Genaueres über CORBA erfährt man z.B. unter [CORBA] oder in vielen Fachbüchern, wie z.B. [Knapik 98].

5.3.1. Aglets

Das IBM Tokyo Research Laboratory hat ein Werkzeug entwickelt, mit dem mobile Java-Agenten erzeugt werden können [Aglets]. Der Name Aglet ist aus den Begriffen Agent und Applet abgeleitet und bezeichnet ein Java-Objekt, das in der Lage ist, im Internet von einem Rechner zu anderen zu migrieren, dort Berechnungen auszuführen

und dann zu weiteren Rechnern oder zurück zum Ausgangsrechner zu „wandern“. Das Java Aglet Application Programming Interface (J-AAPI) definiert die notwendigen Methoden zum Generieren, Versenden, Zurückholen, Klonen und Verteilen von Aglets sowie die Nachrichtenbehandlung innerhalb von Aglets. Es besteht aus folgenden Interfaces und Klassen, die zur Verwaltung eines mobilen Objekts eingesetzt werden:

- Ein *Aglet* ist das mobile Objekt selbst.
- Ein *context* ist eine sichere Ausführungsumgebung auf einem entfernten Rechner.
- Ein *proxy* bezeichnet ein lokales Stellvertreter-Objekt. Es schützt die öffentlichen Methoden des Aglets vor Zugriffen „böswilliger“ Programme und verwaltet den aktuellen Aufenthaltsort des Aglets, der für andere Programme transparent ist.
- Ein *message manager* versendet und empfängt Nachrichten zwischen Aglets als Java-Objekte. Es ist sowohl synchrones als auch asynchrones Versenden von Nachrichten möglich.
- Ein *itinerary* gibt den Weg eines Aglets im Rechnernetz an.
- Jedem Aglet wird ein global eindeutiger *identifier* zugeordnet.

Auf der Seite eines entfernten Rechners muss entschieden werden, ob einem Aglet vertraut werden darf oder nicht, denn Zugriffe auf Dateisysteme oder andere Programme sollen nicht jedem beliebigen fremden Programm erlaubt sein. Hierfür wird ein *security manager* zur Verfügung gestellt, der überprüft, ob das Aglet auf sicherheitskritische Bereiche im Rechner zugreifen darf. Einige Beispiele für geeignete Anwendungsfelder von Aglets sind

- das *Einsammeln von Informationen* durch Datenbankzugriffe auf verschiedenen Rechnern im Internet,
- sogenanntes *web site monitoring*, bei dem ein Aglet bestimmte entfernte WWW-Seiten überwacht und bei Änderungen auf diesen Seiten eine email an seinen Benutzer schickt,
- der Einsatz beim *electronic commerce*, in dem Aglets als Käufer und Verkäufer auftreten und sich auf beliebigen Knoten im Rechnernetz treffen können.

5.3.2. Odyssey

Odyssey von der Firma General Magic ist das Nachfolgesystem von Telescript, einer interpretierten, objektorientierten Programmiersprache, die Anfang der 90er-Jahre entwickelt wurde. In Telescript gibt es drei Grundkonzepte (*Agenten*, *Orte* und *Reisen*), mit denen vor allem Szenarien elektronischer Marktplätze realisiert worden sind. Ein Agent ist in diesen Fällen ein Händler, der zu verschiedenen Rechnerknoten migriert (oder mit den Konzepten von Telescript ausgedrückt: zu verschiedenen Orten reist) und dort mit anderen Agenten durch Nachrichtenaustausch in Verhandlung tritt. Der Vertrieb von Telescript wurde 1996 eingestellt, doch dafür entwickelt General Magic nun das Nachfolgesystem Odyssey weiter, das komplett in Java implementiert ist [Odyssey]. Die drei Grundkonzepte von Odyssey sind *Agent*, *Agentensystem* und *Ort*:

- Ein Agent ist ein eigener Java-Thread und abgeleitet aus einer Odyssey-Oberklasse *Agent* oder *Worker*. Ein Worker hat eine Liste von Aufgaben und eine Liste von Zielrechnern. An jedem Zielrechner arbeitet der Worker die nächste Aufgabe seiner

Liste ab. Der Worker kann seine Aufgabenliste dynamisch ändern, z.B. kann er zunächst zu einem Informationsdienst migrieren und nach den Anbietern einer bestimmten Ware fragen. Die vom Informationsdienst erhaltene Liste von Anbietern und ihren Orten wird dann vom Worker zur Laufzeit in seine Zielliste aufgenommen.

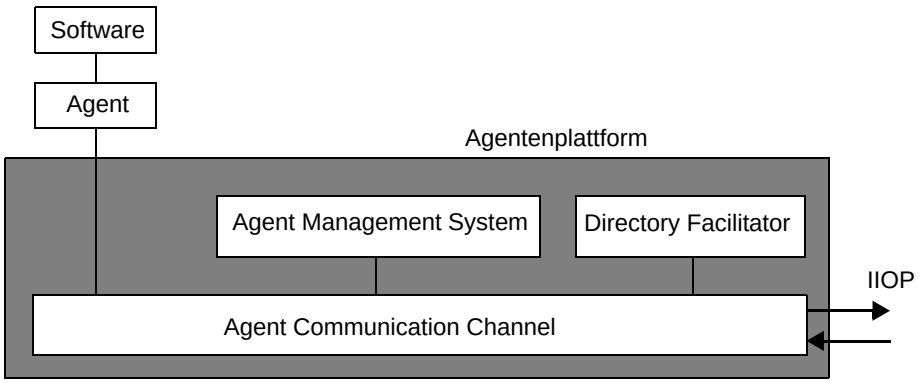
- Ein Ort (*Place*) ist die Ablaufumgebung von Agenten in einem Agentensystem, in der u.a. Zugriffsberechtigungen für Agenten verwaltet werden. Zur Ausführung eines mobilen Odyssey-Agenten müssen solche Orte auf dem Zielrechner des Agenten vorhanden sein, d.h. auf dem Zielrechner muss vor Ankunft des Agenten ein geeignetes Objekt der Odyssey-Klasse *Place* generiert worden sein.
- Das Agentensystem (*Agent System*) ist die Plattform für Odyssey-Agenten und kann Agenten generieren, verwalten, ausführen, übertragen und beenden. Es gibt einige Hilfsklassen zur Unterstützung von Agenten und Workern, z.B. die Klasse *Ticket*, mit der angegeben wird, wohin ein Agent reisen soll, oder die Klasse *Petition*, in der spezifiziert wird, mit welchen Agenten kommuniziert werden soll.

Odyssey unterstützt neben Javas Remote Method Invocation (RMI) auch OMGs Internet Inter-ORB Protocol (IIOP) und Microsofts Distributed Component Object Model (DCOM). Anwendungen mit Odyssey-Agenten sind hauptsächlich für das Internet bestimmt, insbesondere Agenten, die Angebote für bestimmte Waren einholen oder die Preise von Waren überwachen und sich bei Bedarf beim Benutzer melden.

5.3.3. MECCA

Das MECCA-System setzt die wesentlichen Bestandteile der FIPA 97-Spezifikation für das Agentenmanagement um [FIPA 97]. In der folgenden Abbildung wird der strukturelle Aufbau einer Agentenplattform dargestellt.

Abbildung 19 Agentenmanagement nach FIPA 97



Eine Agentenplattform besteht aus den drei Komponenten Agent Management System (AMS), Directory Facilitator (DF) und Agent Communication Channel (ACC), die jeweils auch als Agenten anzusehen sind. Software-Agenten können ihre Dienste beim DF registrieren lassen oder bei ihm anfragen, welche Dienste von anderen Agenten angeboten werden (*yellow pages*). Auf einer Agentenplattform muss mindestens ein

DF vorhanden sein (*default-DF*), es können aber auch mehrere unterstützt werden. Jeder DF kann sich bei anderen DFs registrieren, und auf ähnliche Weise registrieren sich auch AMS und ACC. Die bei einem DF registrierten Agenten formen eine Agentendomäne (*agent domain*), die in anderen Frameworks auch als virtuelle Gruppe bezeichnet wird. Auf jeder Plattform gibt es genau ein AMS, das sich bei mindestens einem DF registriert. Das AMS ist für die Verwaltung der Plattform zuständig, z.B. zur Erzeugung neuer Agenten und zum Beenden von Agenten. Auch die aktuelle Adresse mobiler Agenten wird vom AMS verwaltet (*white pages*). Jeder Agent hat Zugriff auf einen ACC, der die Verbindung zwischen Agenten untereinander sowie zu AMS und DFs herstellt. Der ACC stellt auch die Verbindung zu anderen Plattformen her und unterstützt dabei u.a. das standardisierte Protokoll IIOP der Object Management Group.

Das MECCA-System der Firma Siemens ist in Java implementiert und FIPA-konform. Beim Start des MECCA-Systems erscheint eine grafische Oberfläche für AMS und default-DF, von denen aus weitere Agenten hinzugefügt, gelöscht und ihre Nachrichten auf dem Bildschirm beobachtet werden können.

5.3.4. Voyager

Die Firma ObjectSpace bietet Java-Softwareprodukte für verteilte Anwendungen an, u.a. das Framework *Voyager ORB*, dessen frei verfügbare Version über das WWW heruntergeladen werden kann [Voyager]. Es gibt auch kostenpflichtige Erweiterungen zum Voyager ORB, z.B. einen Security Manager oder eine grafische Oberfläche.

Voyager ORB – im Folgenden kurz Voyager genannt – ist ebenfalls in Java implementiert, unterstützt CORBA und RMI und bietet insgesamt mehr Möglichkeiten als die anderen betrachteten Frameworks. Im Folgenden werden die wichtigsten Eigenschaften von Voyager herausgestellt:

- Man kann mit Voyager sehr leicht Objekte beliebiger Klassen entfernt generieren, wobei eine lokale Stellvertreter-Instanz (*Proxy*) am eigenen Rechner bereitgestellt wird. Die Stellvertreter-Klasse implementiert dieselben Interfaces wie die Originalklasse und wird sogar dynamisch generiert, falls sie noch nicht existiert. Wenn sich ein Objekt an einem entfernten Rechner befindet, werden Methodenaufrufe für dieses Objekt über den Proxy dorthin weitergeleitet.
- Ein Voyager-Programm, d.h. ein Objekt, das mit Voyager entfernt ausführbar geworden ist, kann sowohl als Dienstnehmer als auch als Dienstgeber fungieren. Es kann ohne weiteren Aufwand mit Programmen kommunizieren, die RMI oder CORBA-Standards benutzen. Voyager unterstützt IDL, IIOP sowie die Umformung von IDL in Java und umgekehrt.
- Alle versandfähigen Java-Objekte können zwischen Voyager-Programmen verschickt werden. Wenn z.B. eine Nachricht von einem Proxy zu einer veralteten Adresse des entfernten Objekts verschickt wird, kommt die Nachricht zurück, zusammen mit der neuen Adresse des Objekts. Der Proxy speichert die neue Adresse und verschickt die Nachricht daraufhin an die aktualisierte Adresse. Auf ganz ähnliche Weise kann man auch mobile Agenten verwalten.
- Nachrichten können auf verschiedene Arten verschickt werden: synchron, asynchron ohne Bestätigung (one-way), asynchron mit Bestätigung (future).

- Eine spezielle Architektur ersetzt den konventionellen Repeater bei Gruppenkommunikation und soll Zeit bei der Verteilung von Nachrichten an eine große Anzahl von Empfängern einsparen.

  Auf jedem Rechner werden die Objekte einer Gruppe in lokalen logischen Räumen – sogenannten *SubSpaces* – verwaltet. SubSpaces sind mit anderen SubSpaces derselben Gruppe auf entfernten Rechnern durch Verbindungen (SubSpace-Links) gekoppelt und formen zusammen einen *Space*™. Wenn eine Nachricht zu einem der SubSpaces geschickt wird, werden Kopien dieser Nachricht an die benachbarten SubSpaces sowie an die lokalen Objekte weitergeleitet. Ein spezieller Mechanismus in den SubSpaces garantiert, dass Nachrichten bei dieser Verteilung nur einmal zu den einzelnen Zielobjekten weitergeleitet werden.

Abbildung 20 Voyager Space

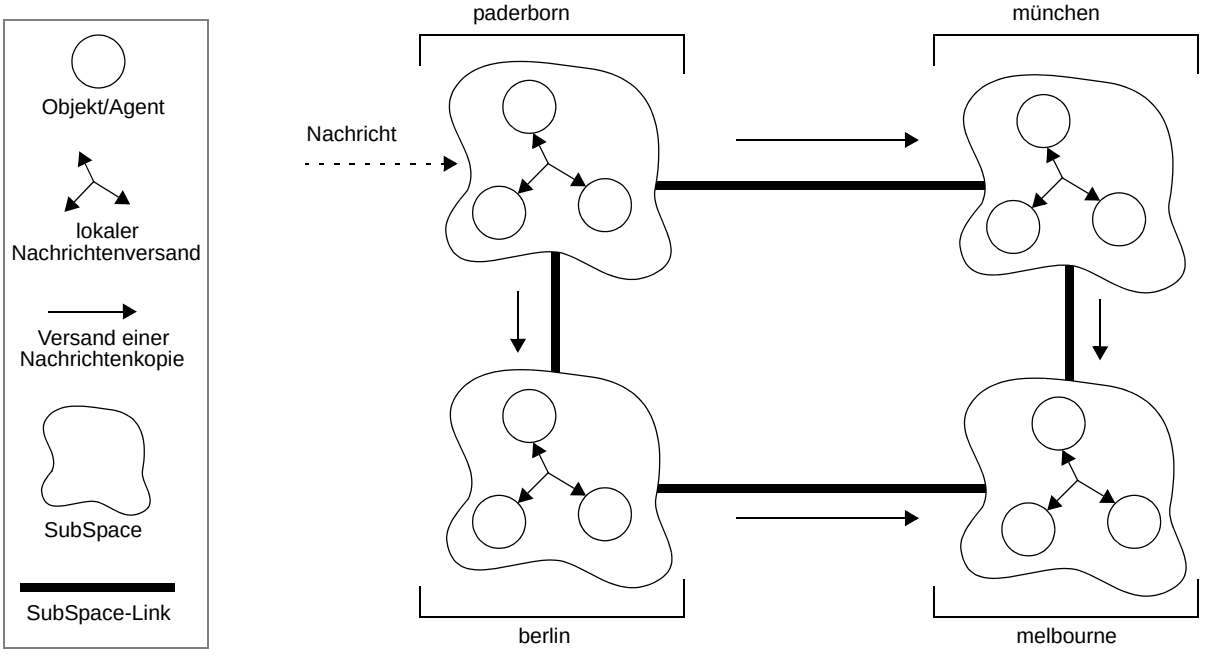


Abbildung 20 veranschaulicht diesen Verteilungsvorgang an einem Beispiel. Eine Nachricht wird an einen SubSpace im Rechner paderborn geschickt. Von dort aus existieren Links zu den Rechnern münchen und berlin, also werden Kopien dieser Nachricht dorthin verschickt. Danach wird die Nachricht im Rechner paderborn lokal an die zugehörigen Objekte verschickt. Von den Rechnern berlin und münchen existiert jeweils ein Link zum Rechner melbourne, also erhält dieser Rechner von beiden eine Kopie der Nachricht. Es wird automatisch gewährleistet, dass die Nachricht nur einmal an die lokalen Objekte im Rechner melbourne verteilt wird.

5.4. Vergleich und Bewertung

Alle betrachteten Frameworks sind in Java implementiert und generieren Agenten in jeweils eigenen Threads. In der folgenden Tabelle werden die vorgestellten Frameworks anhand einiger wichtiger Eigenschaften gegenübergestellt.

Eigenschaft	Aglets	Odyssey	MECCA	Voyager
Nachrichtenversand	synchron, asynchron mit Bestätigung	asynchron	synchron, asynchron mit und ohne Bestätigung	synchron, asynchron mit und ohne Bestätigung, Multicast
unterstützte Standards		IIOP, DCOM	IIOP, FIPA	IIOP, DCOM
sprechaktbasierte Kommunikation	nicht vorhanden	nicht vorhanden	FIPA ACL	nicht vorhanden
eingebaute Elemente zur Koordination	nicht vorhanden	nicht vorhanden	Interaktionsprotokolle	nicht vorhanden
Verzeichnisdienst	Liste mit Agentennamen und ihren aktuellen Orten	Liste mit Agentennamen und ihren aktuellen Orten	DF und AMS	integriert in einen lokalen Stellvertreter
Unterstützung für mobile Agenten	sehr ausgereift	ausgereift	nicht vorhanden	besonders ausgereift
Sicherheitsmechanismen	Security manager	nur Java-eigene Sicherheitsmechanismen	nur Java-eigene Sicherheitsmechanismen	Security manager
Grafische Oberfläche	sehr ausgereift	vorhanden	nur für AMS und DFs und zum Monitoring von Agentennachrichten	vorhanden, aber kostenpflichtig

Bei den Frameworks Aglets, Odyssey und Voyager steht die Generierung mobiler Agenten im Vordergrund. Sie nutzen die Möglichkeiten der Programmiersprache Java, um entfernte Objekte einzurichten, zu verwalten und zu manipulieren. Die mit mobilen Agenten verbundenen Sicherheitsprobleme werden teilweise durch Security manager behandelt.

Betrachtet man die vorgestellten Frameworks aber vor dem Hintergrund, sie zur Realisierung eines Multi-Agentensystems für eine *Produktionsanlage* wie den Farbsortierspeicher einzusetzen, fallen Aspekte für mobile Agenten und die dazu benötigten Sicherheitsmechanismen nicht sehr ins Gewicht. Die Agenten in einer Produktionsanlage liegen eher statisch auf einem Rechner vor, als dass sie von Knoten zu Knoten in einem Rechnernetz migrieren müssen. Allenfalls für komplexe Berechnungen kann es hier sinnvoll sein, auf einen anderen Rechner auszuweichen, z.B. weil ein anderer Rechner die nötigen Berechnungen viel schneller ausführen kann. Da dies aber eher die Ausnahme bildet, sind für die Realisierung des Farbsortierspeichers andere Kriterien bei der Auswahl eines geeigneten Frameworks wichtiger, wie z.B. ein Verzeichnisdienst, bei dem sich Agenten als Dienstgeber und Dienstnehmer zur Laufzeit registrieren und dort nach Partnern suchen können.

Bei der Implementierung eines agentenbasierten Modells sollten agententypische Konzepte wie sprechaktbasierte Nachrichten und Elemente zur Koordination, wie sie in Kapitel 4 vorgestellt worden sind, leicht umgesetzt werden können. Von den vorgestellten Frameworks unterstützt lediglich MECCA sprechaktbasierte Nachrichten sowie Koordination durch Interaktionsprotokolle. Bei den anderen Systemen muss man erst noch eine Implementierung von KQML oder FIPA ACL integrieren oder gar ganz ohne spezielle Agenten-Kommunikationssprache auskommen und eine eigene Notation entwickeln.

Jedes der vorgestellten Frameworks bietet eine grafische Oberfläche an, die hilfreich bei der Verwaltung des laufenden Systems ist. Mit diesen Oberflächen kann man z.B. neue Agenten zur Laufzeit generieren, Agenten explizit beenden und ihre Aktivitäten, insbesondere ihre Nachrichten, auf dem Bildschirm ausgeben lassen.

Da in dieser Arbeit als Anwendungsbeispiel der in Kapitel 2 vorgestellte Farbsortierspeicher realisiert werden soll, erscheint mir MECCA von den betrachteten Frameworks am besten geeignet. MECCA unterstützt als einziges dieser Frameworks eine Agenten-Kommunikationssprache, die auf Sprechakten basiert und auch Elemente zur Koordination beinhaltet. MECCA setzt außerdem ein durchdachtes Konzept zur Verwaltung von Agenten um, und auch MECCAs grafische Oberfläche reicht zur Verwaltung der Agenten im Farbsortierspeicher aus.

KAPITEL 6 Die Spezifikationssprache CASA

In diesem Kapitel wird die Agenten-Spezifikationssprache CASA vorgestellt. Die theoretische Grundlage für diese Sprache liefert die von C. Geiger entwickelte formale Sprachdefinition AgentGHC. Ich beschreibe zunächst die Umsetzung der Sprachelemente von AgentGHC in eine leicht verständliche Spezifikationssprache.
Anschließend gehe ich auf Ergänzungen und Verfeinerungen im abstrakten Interpreter von AgentGHC ein. Es werden neue Komponenten im Interpreter vorgestellt, die die Abarbeitung spekulativer Berechnungen überwachen. Ich habe Verfeinerungen bei der Typisierung von Ereignissen vorgenommen und mache konkrete Vorschläge für die Auswahlfunktionen im Interpreter.
Am Ende dieses Kapitels wird zusammenfassend der Aufbau eines CASA-Agenten anhand eines Beispiels aufgezeigt.

6.1. Sprachelemente von CASA

Bei der Mikrosicht nach dem AgentGHC-Modell hat jeder Agent u.a. lokales Wissen (Fakten), Strategien (Pläne) und aktuell verfolgte Ziele (Intentionen). Ein Agent hat in der Regel schon von Beginn an ein gewisses Grundwissen, eine Reihe von Plänen und auch schon ein oder mehrere Ziele. Zur Beschreibung des Anfangszustands eines Agenten benötigt man eine Spezifikationssprache, in der die o.g. Elemente leicht verständlich zu beschreiben sind. Dieser Abschnitt befasst sich mit Syntax und Semantik von Sprachelementen zur Beschreibung von Wissen, Plänen und Zielen. Die Sprachelemente lehnen sich an die von M. J. Huber entwickelte Sprache JAM an [Huber 98]. Die Schlüsselworte der Sprache werden immer in Großbuchstaben angegeben, andere Sprachelemente wie Ausdrücke, Datentypen und Argumente stehen in eckigen Klammern. Eine kontextfreie Grammatik von CASA ist im Anhang B zu finden.

6.1.1. Grundelemente

Grundelemente der Sprache sind *Ausdrücke*. Ein Ausdruck ist eine Variable, eine Konstante oder eine Zusammensetzung aus Grundfunktion und Argumenten.

Ohne *Variablen* kann keine allgemeine Beschreibung von Plänen erfolgen. Variablen werden z.B. bei der Deklaration des Planziels als Argumente benötigt, um Werte übernehmen zu können wie bei Prozeduren oder Methoden imperativer Programmiersprachen. Variablen werden aber auch gebraucht, wenn Wissen abgefragt und zwischengespeichert werden soll, um diese Information weiterzuverarbeiten. Außerdem sind Ablaufstrukturen ohne Variablen nicht komfortabel verwendbar (vgl. Abschnitt 6.2.2.). Eine Variable wird in CASA durch eine Zeichenkette mit vorangestelltem Dollar-Zeichen gekennzeichnet (z.B. $status). Variablen sind zunächst frei, d.h. ohne Typ und Wert, und können ohne vorherige Deklaration in Plänen benutzt werden. Variablen haben generell nur lokale Gültigkeit für die Ausführung eines einzelnen Plans. Wenn also derselbe Plan mehrfach zur Erfüllung verschiedener Ziele verwendet wird, haben Änderungen von Variablenwerten in dem einen Plan keinen Effekt auf die Variablen in anderen Plänen.

Werte der Datentypen Integer, Double und String können in CASA als *Konstanten* auftauchen und von Variablen angenommen werden. Variablen haben keinen festgelegten Typ, d.h. es können in beliebiger Reihenfolge Werte verschiedener Datentypen an eine Variable gebunden werden.

Eine Anzahl von *Grundfunktionen* soll es ermöglichen, mit Werten und Variablen Berechnungen durchzuführen und Ergebnisse zu erhalten. Als Ergebnistyp ist zusätzlich zu den drei schon angesprochenen Datentypen auch noch der Typ Boolean zugelassen, sodass Vergleichsfunktionen möglich sind. Die Syntax in Präfixnotation lehnt sich an Lisp an: erst der Funktionsname, dann die Argumente. Beispielsweise werden in dem Ausdruck (+ 6 12) die Zahlen 6 und 12 addiert. In der folgenden Tabelle sind die Grundfunktionen aufgeführt [Huber 98]. Die Datentypen Ganzzahl und Fließkommazahl sind als Argumente bei den Operationen nebeneinander erlaubt, jedoch ist es z.B. nicht zulässig, eine Zeichenkette mit einer Zahl zu vergleichen.

Funktion	Beschreibung	Funktion	Beschreibung
-	Subtraktion	+	Addition
*	Multiplikation	/	Division
Mod	Modulo	Abs	Absolutwert
!=	Ungleich-Operator	==	Gleichheits-Operator
>	Größer-als-Operator	>=	Größer-gleich-Operator
<	Kleiner-als-Operator	<=	Kleiner-gleich-Operator
&&	Konjunktion	||	Disjunktion
!	Negations-Operator		

Als Beispiel sollen die Werte der Variablen $status und $myXPosition mit konstanten Werten verglichen werden:

Ausdruck	Beschreibung
(== $status "waiting")	Die Auswertung des Ausdrucks liefert genau dann TRUE, wenn $status zur Zeit vom Typ String ist und den Wert "waiting" hat.
(>= $myXPosition 100)	Die Auswertung des Ausdrucks liefert genau dann TRUE, wenn $myXPosition zur Zeit vom Typ Integer oder Double und der Wert von $myXPosition größer oder gleich 100 ist.
!(== $myXPosition 50)	Die Auswertung des Ausdrucks liefert genau dann TRUE, wenn $myXPosition zur Zeit vom Typ Integer oder Double und der Wert von $myXPosition ungleich 50 ist. Äquivalent ist der Ausdruck (!= $myXPosition 50).

Des Weiteren gibt es in CASA noch *Bezeichner* (identifier) und *Relationen*. Ein Bezeichner gibt einen festgelegten Namen an und darf nur aus Buchstaben, Bindestrichen und Ziffern bestehen (vgl. Anhang B). Relationen bestehen aus einem Bezeichner und einer beliebigen Anzahl von Ausdrücken. Die Syntax einer Relation ist wie folgt festgelegt:

<relation> ::= <identifier> <expression>$_1$ ... <expression>$_n$

6.1.2. Wissen

In AgentGHC wird Wissen über den eigenen Zustand und das Umfeld eines Agenten durch *Beliefs* ausgedrückt. Es handelt sich dabei um Fakten, wie sie auch in logischen Programmiersprachen verwendet werden. Im Folgenden werden die Begriffe Belief, lokales Wissen und Fakt gleichbedeutend benutzt. Die durch Fakten gespeicherten Informationen müssen nicht die komplette Umgebung beschreiben, sie können sogar veraltet und gar nicht mehr zutreffend sein. Der formalen Beschreibung eines Beliefs in AgentGHC als $b(t_1,...,t_n)$ – mit b als Prädikatssymbol und $t_1,...,t_n$ als Terme erster Ordnung – entspricht in der Spezifikationssprache CASA die Angabe einer Relation, ein-

geleitet durch das Schlüsselwort FACT. Die Menge **Bel** der Beliefs wird in einer Datenstruktur verwaltet, die als Wissensbasis bezeichnet und in Abschnitt 6.3.3. näher beschrieben wird. Bei der Agentenspezifikation wird eine Menge von Initialfakten angegeben, die direkt in diese Wissensbasis aufgenommen werden. Gekennzeichnet durch das Schlüsselwort "FACTS:" folgt die Auflistung aller Initialfakten in der Agentenspezifikation.

```
FACTS:
    FACT <relation>;
    ...
    FACT <relation>;
```

In AgentGHC ist nicht näher spezifiziert, welche Struktur die einzelnen Terme haben. Für die Spezifikationssprache CASA werden daher folgende Grundtypen für die Argumente der Relationen festgelegt: Ganzzahl (Integer), Fließkommazahl (Double) und Zeichenkette (String). Einige Initialfakten für einen Einlagerungsagenten können so aussehen:

```
FACTS:
    FACT myId         "inRobot-1";
    FACT myPosition 20 30;
    FACT myState      "waiting";
```

Zugriff auf die Wissensbasis. Anweisungen zur Manipulation der Wissensbasis liefern die Möglichkeit, bei der Ausführung von Plänen das lokale Wissen über die Umgebung bei Bedarf anzupassen sowie Informationen über den eigenen Zustand (z.B. die aktuelle Position) zu ändern. In der folgenden Tabelle sind die Konstrukte zumHinzufügen, Löschen, Ändern und Abfragen von Fakten aufgeführt. Genaueres über die Auswirkungen der Aktionen in der Wissensbasis wird in Abschnitt 6.3.3. erläutert.

Konstrukt	Beschreibung
ADD <relation>	Ein Fakt wird zur Wissensbasis hinzugefügt. Diese Aktion schlägt nie fehl, d.h. als Ergebnis wird immer TRUE zurückgeliefert.
DELETE <relation>	Ein oder mehrere Fakten werden aus der Wissensbasis gelöscht. Auch wenn kein Fakt gefunden wird, schlägt diese Aktion nicht fehl.
UPDATE <relation>$_1$ <relation>$_2$	Diese Aktion verhält sich wie die Hintereinanderausführung von DELETE <relation>$_1$ und ADD <relation>$_2$ Diese Aktion schlägt nie fehl.
GETFACT <relation>	In der Wissensbasis wird nach einem Fakt mit demselben Relationennamen und derselben Argumentanzahl gesucht. Das erste passende Fakt wird übernommen. Alle Variablen werden neu gebunden. Wenn kein passendes Fakt gefunden wird, schlägt diese Aktion fehl.
GETVALUES <relation>	Diese Aktion verhält sich ähnlich wie GETFACT, aber hier werden gebundene Variablen zur Auffindung des Fakts mitbenutzt und nicht neu belegt. Wenn kein passendes Fakt gefunden wird, schlägt diese Aktion fehl.

6.1.3. Ziele

In AgentGHC wird zwischen flachen und tiefen Zielen unterschieden. Ein flaches Ziel wird auch Test genannt und überprüft die Gültigkeit eines Beliefs in der Menge **Bel**. Es entspricht in CASA der Frage nach der Existenz eines Fakts in der Wissensbasis und wird innerhalb von Plänen durch folgendes Sprachelement ausgedrückt:

```
EXISTS (FACT <relation>)
```

Die Abfrage über die Existenz eines Fakts erfordert keine komplexe Berechnung, da direkt auf die Wissensbasis zugegriffen und sofort ein Ergebnis (TRUE oder FALSE) geliefert wird. Für den Fall, dass von einem anderen Agenten nach der Existenz eines bestimmten Fakts gefragt wird, kann man mit dem obigen Sprachelement leicht einen Plan angeben, der das Ergebnis der Anfrage an den Absender zurückschickt.

Ein tiefes Ziel in AgentGHC wird durch $?g(t_1,...,t_n)$ ausgedrückt, wobei g ein Prädikat ist und $t_1,...,t_n$ Terme erster Ordnung sind. Ein tiefes Ziel gibt einen Zustand an, der vom Agenten in der Zukunft erreicht werden soll. In CASA werden tiefe Ziele durch das folgende Konstrukt angegeben:

```
ACHIEVE <relation> : <priority>
```

Die Ziele, die ein Agent von Beginn an verfolgt, werden Initialziele genannt und bei der Agentenspezifikation in der Sektion "GOALS:" nacheinander angegeben:

```
GOALS:
    ACHIEVE <relation name> <argument_1> ... <argument_n> : <priority>;
    . . .
    ACHIEVE <relation name> <argument_1> ... <argument_m> : <priority>;
```

Bei Initialzielen müssen die Argumente vom Typ Integer, Double oder String und die Gewichtung (priority) muss vom Typ Integer oder Double sein.

Ziele innerhalb von Plänen. Das ACHIEVE-Konstrukt ermöglicht es auch, aus Plänen heraus das Verlangen nach Erreichung eines Unterziels auszudrücken. Die Syntax hierfür gleicht dem Aufbau von Initialzielen, jedoch kann die Gewichtung nicht nur als Konstante, sondern auch als Ausdruck, z.B. durch eine Variable, angegeben werden. Welchen Einfluss die Gewichtung auf die später entstehende Intention hat, wird in Abschnitt 6.3.6. genauer erklärt.

6.1.4. Aktionen

Es gibt eine Reihe von nützlichen Aktionen zur Behandlung von Ausdrücken. Sicherlich ist es erforderlich, Variablen einen (errechneten) Wert zuweisen zu können. Außerdem werden Anwender in der Regel noch weitere Befehle benötigen, um individuelle Anforderungen an ihre Agenten umsetzen zu können. Es muss also die Möglichkeit geben, die Menge der Befehle beliebig zu erweitern. Dies kann durch Definition von weiteren Funktionen geschehen, die mit der EXECUTE-Aktion aufgerufen werden. Die folgende Tabelle zeigt die Aktionen auf, die fester Bestandteil von CASA sein sollen.

Konstrukt	Beschreibung
ASSIGN <variable> <expression>	Zuweisung eines Wertes (vom Typ Integer, Double oder String) an eine Variable. Diese Aktion schlägt nie fehl.
TEST <expression>	Diese Aktion meldet TRUE zurück, wenn der Ausdruck <expression> wahr ist, ansonsten schlägt sie fehl.
FAIL	Diese Aktion schlägt immer fehl.
EXECUTE <function> <arguments>	Mit diesem Konstrukt können Funktionen vom Agentenentwickler applikationsabhängig hinzugefügt werden. Parameteranzahl und -typen müssen zu den aufgerufenen Funktionen passen, ansonsten schlägt die Aktion fehl.

6.1.5. Nachrichten

Nachrichten nehmen eine Sonderstellung ein, weil sie sich nicht auf die Mikrosicht eines Agenten beschränken wie alle anderen Aktionen in Plänen. In AgentGHC ist die allgemeine Struktur einer Nachricht wie folgt festgelegt:

```
Nachrichtentyp (Sender, Empfänger, Inhalt, Antwort, Status)
```

Zu jeder Nachricht wird explizit der Sender und ein Status angegeben. Diese Angaben werden in CASA automatisch zu den Nachrichten ergänzt. Der CASA-Interpreter soll die Aufgabe übernehmen, die Angaben zu einer kompletten Nachricht zusammenzustellen und in ein für den Empfänger verständliches Format zu bringen. Hier bietet es sich an, das Format einer standardisierten Agenten-Kommunikationssprache wie FIPA ACL zu verwenden, sodass sich auch plattformübergreifend Agenten verschiedener Architekturen (und verschiedener Entwickler) auf einer gemeinsamen Basis verständigen können. Eine komplette Nachricht besteht aus den folgenden Elementen, die teilweise implizit von CASA ergänzt werden und für die Spezifikation von Nachrichten transparent sind:

Element der Nachricht	Angabe	CASA-Behandlung
Nachrichtentyp	implizit	Durch das Sprachelement ist auch der Nachrichtentyp festgelegt.
Sender	implizit	Die Angabe des Senders kann von CASA automatisch ergänzt werden, den der Agent kennt seine eigene Adresse.
Referenz	implizit	Als Referenz einer Nachricht wird die ID der ausführenden Intention von CASA automatisch zur Nachricht ergänzt.
Empfänger	als Parameter <receiver>	Der Empfänger muss bei der Spezifikation angegeben werden. Dies ist aber auch durch eine Variable möglich.
Nachrichtenname	als Parameter <msg name>	Der eigentliche Inhalt der Nachricht wird genauer spezifiziert durch einen Nachrichtennamen. Diese Angabe ist wie beim Empfängerparameter auch durch eine Variable möglich.
Inhalt	als Parameter <args>	Der Inhalt einer Nachricht ist eine Liste von Argumenten, angegeben durch CASA-Ausdrücke. Zur Erstellung einer Nachricht werden die Ausdrücke ausgewertet und als konstante Werte vom Typ String, Integer oder Double versendet.

Die Nachrichtenelemente "Antwort" und "Status" aus AgentGHC werden in CASA je nach Nachrichtentyp unterschiedlich behandelt. Bei einigen Nachrichtentypen sind sie gar nicht notwendig und werden einfach weggelassen, bei anderen werden sie explizit in der Spezifikation behandelt, z.B. durch GETREPLY oder WAIT. In der folgenden Tabelle wird die CASA-Syntax der vier Nachrichtentypen angegeben, die in AgentGHC vorgeschlagen werden.

Nachrichtentyp in AgentGHC	Sprachelement in CASA	Beschreibung
inform	INFORM <receiver> <msg name> <args>;	Versenden einer Nachricht ohne Warten auf eine Eingangsbestätigung des Empfängers. Antwort und Status sind nicht nötig.
informRec	INFORM_REC <receiver> <msg name> <args>; ... // some actions WAIT : <receiver>;	Versenden einer Nachricht und Warten auf die Eingangsbestätigung des Empfängers. In der Zwischenzeit können jedoch weitere Aktionen ausgeführt werden. Der Status wird implizit von CASA überwacht.
sendRec	REQUEST <receiver> <msg name> <args>; GETREPLY <list of variables>;	Anfrage und Warten auf eine Ergebnisnachricht (synchrones Senden). Zwischen REQUEST und GETREPLY sind keine weiteren Aktionen erlaubt. Der Status der Bearbeitung wird implizit überwacht. Die Antwort wird von CASA automatisch zur richtigen Anfrage geleitet und der Inhalt in die Variablen hinter GETREPLY geschrieben. Wenn die Anzahl der Variablen nicht zur Antwort passt, schlägt die Aktion fehl.
send	REPLY <receiver> <msg name> <args>;	Antwort auf das REQUEST eines anderen Agenten. Der Empfänger wartet auf diese Antwort, der Sender erwartet jedoch keine weitere Rückmeldung vom Empfänger.

Es soll auch die Möglichkeit geben, weitere Nachrichtentypen individuell zu erstellen. Dies ist in CASA durch die applikationsabhängige Ergänzung primitiver Funktionen und Ausführung mittels der EXECUTE-Aktion möglich (vgl. Abschnitt 6.1.4.).

Zur Behandlung von Anfragenachrichten. REQUESTs werden beim Empfänger wie neue Ziele behandelt: Es muss ein anwendbarer Plan gefunden werden, der die Anfrage behandelt und eine Antwort zurückschickt. Damit eine Anfrage auf diese Weise bearbeitet werden kann, muss der Anfrage genauso wie bei tiefen Zielen eine Gewichtung zugeteilt sein. Die komplette Syntax einer Anfrage sieht daher wie folgt aus:

```
REQUEST <receiver> <msg name> <args> : <priority>;
GETREPLY <list of variables>;
```

6.2. CASA-Pläne

Die Umsetzung von AgentGHC-Strategien und ihren Aktionen in eine leicht erlernbare und verständliche Sprache ist der umfangreichste Teil des Sprachentwurfs. Neben den schon vorgestellten Sprachelementen müssen noch verschiedene Ablauf- und Kontrollstrukturen angegeben werden, die eine strukturierte Beschreibung eines Plans erst ermöglichen.

6.2.1. Zusammenhang zwischen GHCs und CASA-Plänen

Die in Abschnitt 3.5. vorgestellten Strategien in AgentGHC sind priorisierte, bewachte Horn-Klausel der Form $P: H \rightarrow G_1 ... G_n \mid B_1 ... B_m [g]$ mit $g \in [0,1]$. In der Spezifikationssprache CASA werden Strategien als Pläne angegeben. Ein Plan enthält alle Komponenten der oben genannten bewachten Horn-Klauseln und wird noch um einige nützliche Elemente (oder: Felder) erweitert. CASA-Pläne haben folgende Struktur:

```
PLAN:
        {
            NAME:           <string>;
            DESCRIPTION:    <string>;
            GOAL:           ACHIEVE <relation>;
            TYPE:           <REACTIVE | DELIBERATIVE | COMMUNICATIVE>;
            PRECONDITION:   <list of conditions>;
            PRIORITY:       <numeric value>;
            BODY:           <list of actions>;
            FAILURE:        <list of actions>;
        }
```

Die folgende Tabelle liefert eine Übersicht über den Zusammenhang zwischen bewachten Horn-Klauseln und CASA-Plänen.

Element in einer bewachten Horn-Klausel	Element im CASA-Plan	Beschreibung
P	NAME	Name des Plans
	DESCRIPTION	informelle Beschreibung des Plans
H	GOAL	Ziel, das erreicht werden soll
	TYPE	Typ analog der Planhierarchie von AgentGHC
$G_1 ... G_n$	PRECONDITION	Vorbedingungen
$B_1 ... B_m$	BODY	Plankörper
$[g]$	PRIORITY	Priorität (oder: Wichtigkeit, Gewichtung)
	FAILURE	Ausführungsblock für Laufzeitfehler

Ein CASA-Plan soll immer Angaben zu NAME, GOAL, TYPE und BODY haben, während die übrigen Planfelder optional sind. Wenn keine Vorbedingungen angegeben sind, ist der Plan immer für das angegebene Ziel anwendbar. Wenn keine Gewichtung angegeben wird, wird für sie der Wert 0 angenommen. Während in AgentGHC für die

Gewichtung $g \in [0,1]$ gilt, wird für CASA festgelegt, dass lediglich $g >= 0$ gelten muss. Diese Änderung erlaubt es, die Gewichtung von Zielen, Unterzielen und Plänen additiv zu verknüpfen, ohne den Gültigkeitsbereich zu verlassen (vgl. Abschnitt 6.3.6.: Bestimmung der Gewichtung einer Intention).

6.2.2. Ablaufstrukturen

Zur wiederholten Durchführung von Anweisungen benötigt man ein Konstrukt, das einen Block von Anweisungen unter einer bestimmten Bedingung ausführt. Das Schlüsselwort WHILE leitet eine solche Schleife ein, gefolgt von einer Bedingung, die einen Wahrheitswert (TRUE oder FALSE) liefert. Solange die Bedingung zutrifft, wird der durch geschweifte Klammern gekennzeichnete folgende Anweisungsblock wiederholt ausgeführt.

Andere bekannte Schleifenkonstrukte (z.B. for-Schleife, do-while-Schleife) können leicht mit der WHILE-Schleife simuliert werden, sodass auf weitere Schleifenkonstrukte zunächst verzichtet wird. Bei Bedarf sollte es kein Problem sein, entsprechende Sprachelemente für solche Schleifen zu CASA zu ergänzen.

Eine andere Art von Ablaufstruktur ist die Möglichkeit, Anweisungsblöcke als nebenläufig zu deklarieren. In AgentGHC wird die parallele Abarbeitung von Aktionen durch Trennung mit Kommata gekennzeichnet. Dieser Notation entspricht in CASA das PARALLEL-Konstrukt. Das folgende Beispiel zeigt, dass das Parallelkonstrukt z.B. bei synchronem Senden den Vorteil hat, dass in einem anderen Parallelzweig während des Wartens auf eine Antwort schon weitergearbeitet werden kann.

```
PARALLEL
        {
            REQUEST $orderAgent "getOrder"; // send request and wait for answer
            GETREPLY $number $type $color;
        }
        {
            ASSIGN $myState "moving";
            EXECUTE gotoDepot;              // primitive function
        }
```

Wenn parallele Blöcke erlaubt sind, ist man allerdings mit Schreibkonflikten auf gemeinsamen Variablen konfrontiert. Wie geht man mit Änderungen der Variablenwerte um, wenn mehrere parallele Blöcke verschiedene Werte in dieselbe Variable schreiben wollen? Sicherlich gibt es mehrere Möglichkeiten, diesen Konflikt von Vornherein zu vermeiden oder im Nachhinein zu beheben.

Man kann für jeden Parallelblock Kopien aller Planvariablen anlegen und nach Ausführung des Blocks die Werte zurückschreiben. Ein Parallelblock entspricht dann einer Prozedur, bei der alle in ihr auftretenden Variablen call-by-value-and-result-Parameter sind. Kommt es nun nach Beendigung der Parallelblöcke zu einem Schreibkonflikt auf gemeinsam benutzten Variablen (d.h., in einer Variable sollen verschiedene Werte abgelegt werden), hat man die Wahl zwischen einer Reihe von Möglichkeiten, diesen Konflikt zu beheben: Man braucht keinen der neuen Werte zu akzeptieren, man kann einen Mittelwert berechnen oder man kann einen Wert durch eine vorgegebene Funk-

tion auswählen. Auf diese Weise bleiben Änderungen von Variablen in einem Block allerdings unsichtbar für alle anderen parallel bearbeiteten Blöcke.

Daher habe ich mich dazu entschlossen, diesem Problem auf andere Weise zu begegnen. Es werden keine Kopien für die Variablen angelegt, sondern von den Parallelblöcken aus wird direkt auf die Planvariablen zugegriffen, sowohl lesend als auch schreibend. Der Parallelblock, der als erstes auf eine (evtl. auch schon gebundene) Variable schreibend zugreift, reserviert diese Variable für weitere Schreibzugriffe für sich, sodass die anderen Parallelblöcke dann nur noch lesenden Zugriff auf diese Variable haben. Wenn nun doch noch ein anderer Parallelblock schreibend auf diese Variable zugreift, ist dies als Ausnahme zu behandeln, z.B. indem eine Warnmeldung erfolgt. Mit dieser Methode erkennen andere Parallelblöcke sofort Änderungen von Variablenwerten, d.h. es wird immer die aktuellste Information benutzt.

Als Folge dieser Art paralleler Variablenzugriffe ist es allerdings nicht möglich, geschachtelte Parallelkonstrukte in einem Plan zu erlauben, denn dann können zusätzlich auch noch innere Parallelblöcke zu ihrem äußeren Block in Konflikt stehen. Geschachtelte Parallelblöcke kann man aber durch Angabe von Unterzielen und weiteren Plänen auch indirekt ausdrücken, wie in dem folgenden Beispiel in halbformaler Syntax verdeutlicht wird.

```
Plan_A                                                    // no parameters
{       ...
        PRECONDITION: PARALLEL   { ACHIEVE Plan_B "First branch of A: " : 1.0; }
                                 { ACHIEVE Plan_B "Second branch of A: " :1.0; }
        ...
}

Plan_B $text                                              // $text is a parameter
{       ...
        PRECONDITION: PARALLEL   { ACHIEVE Another_Plan $text "branch 1 of B" : 2.0; }
                                 { ACHIEVE Another_Plan $text "branch 2 of B" : 2.0; }
        ...
}
```

Angenommen, der ausgewählte anwendbare Plan zum Unterziel "Another_Plan" gibt die übergebenen Parametertexte auf dem Bildschirm aus. Dann kann der entsprechende Teil eines Ausgabeprotokolls wie folgt aussehen, wobei die Ausgabe der vier Textzeilen wegen der gleichhohen Gewichtung der Unterziele aber auch in einer beliebigen anderen Reihenfolge erfolgen kann:

```
First branch of A: branch 1 of B
Second branch of A: branch 2 of B
First branch of A: branch 2 of B
Second branch of A: branch 1 of B
```

6.2.3. Kontrollstrukturen

Bedingte Anweisungen sind bei der Ausführung eines Plankörpers oft notwendig. Das IF-Konstrukt hat einen optionalen ELSE-Zweig und soll für die Spezifikationssprache folgende Syntax haben:

```
a)   IF <action>   { <list of actions> }        // simple if

b)   IF <action>   { <list of actions> }        // if with else
     ELSE          { <list of actions> }
```

Die Bedingung ist hier bemerkenswerterweise nicht ein einfacher Ausdruck, sondern eine Aktion (vgl. Abschnitt 6.1.4.). Nach Ausführung einer Aktion wird als Ergebnis immer ein Wahrheitswert zurückgemeldet. Dabei entspricht TRUE der erfolgreichen Ausführung einer Aktion und FALSE der fehlgeschlagenen Ausführung. Um dem dangling-else-Problem zu entgehen, sind die Blöcke zwingend durch geschweifte Klammern zu kennzeichnen. Eine weitere Kontrollstruktur ist das WAIT-Konstrukt mit der folgenden Syntax:

```
WAIT : <action>
```

Das WAIT-Konstrukt wird so lange immer wieder ausgeführt, bis die Aktion, die als Argument angegeben wird, TRUE liefert. Zum Beispiel kann so auf ein Fakt gewartet werden, das von einer anderen Intention erzeugt werden soll.

6.2.4. Sprachelemente in den einzelnen Planfeldern

Nachdem die einzelnen Sprachelemente für Pläne vorgestellt worden sind, muss nun noch festgelegt werden, welche dieser Elemente in den einzelnen Planfeldern erlaubt sind. CASA-spezifische Sprachelemente werden hauptsächlich in den Sektionen GOAL, PRECONDITION, BODY und FAILURE eingesetzt, daher werden diese Felder im Folgenden genauer betrachtet.

6.2.4.1. Planziel

Jeder Plan verspricht, ein Ziel zu erfüllen, daher ist das GOAL-Feld zwingend anzugeben. Das Ziel wird durch das Schlüsselwort ACHIEVE und eine Relation ausgedrückt, in der der Relationenname das Ziel beschreibt und die Argumente dazu dienen, Parameterwerte in lokale Variablen des Plans zu übernehmen. Wenn der Plan zur Durchführung eines Unterziels dient, werden Änderungen dieser (und nur dieser) Variablen nach erfolgreicher Beendigung des Unterziels an das Elternziel zurückgemeldet. Diese Variablen entsprechen also call-by-value-and-result-Parametern prozeduraler Programmiersprachen. Für einen Plan "getOrder" des Einlagerungsagenten im Farbsortierspeicher kann das Ziel zum Beispiel wie folgt lauten:

```
PLAN:
{       NAME:    "example plan 1";
        GOAL:    ACHIEVE getOrder   $orderAgent $number $body $color : 1.0;
        . . .
}
```

Die Parameter $orderAgent, $body und $color sind als lokale Variablen des Plans zu verstehen und werden an Werte gebunden, wenn der Plan für ein neues Ziel relevant ist. Wenn dieser Plan zur Anwendung kommt, d.h. wenn mit ihm eine Intention gebildet wird, werden die o.g. Variablen nach Abarbeitung des Plans auch als Ergebnislieferanten eingesetzt.

6.2.4.2. Vorbedingungen

Um die Konzepte für Vorbedingungen aus AgentGHC in CASA umsetzen zu können, muss je nach Plantyp eine bestimmte Teilmenge der Sprachelemente im PRECONDITION-Feld erlaubt sein. In der folgenden Tabelle sind die erlaubten Sprachelemente für die drei Plantypen aufgelistet. Das PARALLEL-Konstrukt nimmt als Ablaufstruktur eine besondere Stellung ein: Es soll nicht erlaubt sein, innerhalb eines PARALLEL-Blocks wieder ein PARALLEL-Konstrukt anzugeben, eine Schachtelung dieses Konstrukts ist also nicht zulässig.

Plantyp	PARALLEL	GETFACT GETVALUES TEST	ACHIEVE	REQUEST
reaktiv	X	X		
deliberativ	X	X	X	
kommunikativ	X	X	X	X

Es ist möglich, anhand der in der Vorbedingung benutzten Sprachelemente den Typ jedes Plans zu bestimmen, sodass eine explizite Angabe des Plantyps im TYPE-Feld nicht nötig wäre. Diese Angabe soll jedoch trotzdem immer vorgenommen werden und damit den Charakter jedes Plans kenntlich machen. Falls die Typangabe nicht zu den Vorbedingungen passt, z.B. wenn in einem reaktiven Plan als Vorbedingung ein ACHIEVE-Konstrukt auftritt, soll dies als Spezifikationsfehler gelten. Die Vorbedingung eines kommunikativen Plans "getOrder" kann für den Einlagerungsagenten so aussehen:

```
PLAN:
{       NAME:           "example plan 2: communicative";
        GOAL:           ACHIEVE getOrder $orderAgent $number $body $color  : 1.0;
        TYPE:           COMMUNICATIVE;

        PRECONDITION:   REQUEST "getOrder" $orderAgent : 1.0;
                        GETREPLY $number $body $color;
        . . .
}
```

Wenn der Einlagerungsagent einen Auftrag als Antwort bekommen hat, ist die Vorbedingung erfüllt, und der Plankörper kann abgearbeitet werden. Die Antwort muss drei Werte für die Variablen $number, $body und $olor beinhalten. Wenn die Anfrage aus irgendwelchen Gründen fehlschlägt, kommt der Plan nicht zur weiteren Ausführung infrage.

6.2.4.3. Plankörper

Im Plankörper sind alle vorhandenen Sprachelemente, Kontroll- und Ablaufstrukturen erlaubt. Aber auch hier darf es wie im PRECONDITION-Feld nicht zu einer Schachtelung von PARALLEL-Konstrukten kommen. Ansonsten können alle Sprachelemente in beliebiger Folge und Verschachtelung benutzt werden. Ein kommunikativer Plan "getOrder" für den Einlagerungsagenten hat z.B. als BODY-Feld Angaben über die notwendigen Aktionen nach Erhalt eines Auftrags.

```
PLAN:
{       NAME:               "getOrder - communicative";
        GOAL:               ACHIEVE getOrder $orderAgent $number $body $color  :
1.0;
        TYPE:               COMMUNICATIVE;

        PRECONDITION:       REQUEST "getOrder" $orderAgent : 1.0;
                            GETREPLY $number $body $color;

        BODY:               UPDATE(myOrder) (myOrder $number $body $color);
                            UPDATE(myState) (myState "orderReceived");
                            GETFACT myId $id;
                            EXECUTE print $id "got an order!";
```

Im obigen Beispiel werden einige Fakten in der Wissensbasis aktualisiert ("myOrder" und "myState"), und eine Meldung wird durch eine primitive Funktion abgegeben. Wichtig ist in diesem Zusammenhang, dass der Plan zur Erfüllung eines Unterziels dienen soll und daher der erhaltene Auftrag nach erfolgreicher Beendigung des Plans durch die Parameter $number, $body und $color an die aufrufende Intention zurückgegeben wird, sodass dort mit den erhaltenen Werten weitergearbeitet werden kann.

6.2.4.4. Fehlschlagen von Plänen

In welchen Fällen die einzelnen Aktionen fehlschlagen, wurde schon an verschiedenen Stellen erläutert. Die Aktionen im FAILURE-Feld werden ausgeführt, wenn bei der Abarbeitung des Plan*körpers* eine Aktion fehlschlägt. So kann das FAILURE-Feld dazu benutzt werden, den Plan in einem gültigen Zustand zu verlassen oder Fehlermeldungen abzugeben, bevor der Plan endgültig beendet wird. In der FAILURE-Sektion soll es aber nicht mehr erlaubt sein, komplexe Aktionen wie Unterziele oder parallele Aktionen auszuführen, sondern es sollen nur einfache Aktionen ausgeführt werden. In der folgenden Tabelle sind die im FAILURE-Feld erlaubten Aktionen aufgeführt.

zugelassene Aktionen im FAILURE-Feld	
GETFACT <relation>	UPDATE <relation>$_1$ <relation >$_2$
GETVALUES <relation>	ASSIGN <variable> <expression>
ADD <relation>	TEST <expression>
DELETE <relation>	EXECUTE <function> <arguments>
REPLY <expression> <string> <explist>	

Im Gegensatz zum Ausführungsmodell des Plankörpers soll das FAILURE-Feld als atomare Einheit gelten, d.h. direkt nach dem Fehlschlagen einer Aktion im Plankörper werden alle Aktionen innerhalb eines einzigen Interpreterzyklus abgearbeitet. Dieses Vorgehen soll dazu dienen, den Plan so schnell wie möglich abzubrechen und notwendige Berichtigungen sofort auszuführen. Die erlaubten Aktionen sind fast alle ohne komplexe Berechnungen sofort ausführbar, lediglich beim EXECUTE-Konstrukt könnte es passieren, dass sich hinter der aufzurufenden Funktion eine komplexere Berechnung verbirgt. Dennoch ist es sinnvoll, dieses Konstrukt im FAILURE-Feld zuzulassen, z.B. um Informationen mittels einer primitiven Funktion "print" anzuzeigen.

6.3. Interpreter

Das Ausführungsmodell jedes Agenten soll sich an dem Interpreterzyklus von AgentGHC orientieren. Allerdings muss der ursprüngliche abstrakte Interpreter an einigen Stellen noch genauer spezifiziert werden. In AgentGHC wird nur eine grobe Einteilung in interne und externe Ereignissen angegeben. Ereignisse werden im folgenden Abschnitt typisiert und können dadurch im abstrakten Interpreter unterschiedlich behandelt werden. Außerdem werden bei der ursprünglichen Beschreibung des Interpreters die Auswahlfunktionen nicht näher bestimmt. Ich gebe einige Vorschläge für diese Funktionen an. Weiterhin gehe ich auf spekulative Berechnungen ein, die die Überprüfung von Vorbedingungen bei deliberativen und kommunikativen Strategien bzw. Plänen betreffen.

6.3.1. Sensor

Im Sensor werden wahrzunehmende Ereignisse aufgenommen. Die Menge der Ereignisse im Sensor wird mit **Erg** bezeichnet. Ein zu bearbeitendes Ereignis $\varepsilon = \langle e, Q \rangle$ im Sensor ist ein Tupel mit $e \in$ **Erg** als wahrnehmbarem Ereignis und Q als erzeugender Quelle. Man muss zwischen internen und externen Quellen unterscheiden: Als *interne* Quelle kommen Intentionen der Intentionsstruktur infrage. Wenn es bei der Ausführung einer Intention ein Unterziel zu erreichen gilt, wird hierfür ein Ereignis erzeugt und an den Sensor geschickt. In einem späteren Interpreterzyklus wird dieses Ereignis dann ausgewählt und weiterverarbeitet. In AgentGHC werden zu *allen externen* Ereignissen Strategien gesucht, die beschreiben, wie auf diese Ereignisse zu reagieren ist. Jedoch ist es nicht für jede Art von Ereignis notwendig, erst Strategien auszuwählen. Es gibt auch Fälle, in denen sofort klar ist, was zu tun ist, z.B. wenn ein neues Fakt oder die Änderung eines Fakts von einer externen Quelle angezeigt werden. Die folgende Tabelle listet verschiedene Typen von Ereignissen auf, die im Interpreter behandelt werden können.

Ereignistyp	Quelle	Beschreibung
NEW_FACT bzw. UPDATE_FACT	extern	Mitteilung über ein neues oder verändertes Fakt von außen (Umgebung oder anderer Agent). Das Fakt wird ohne weitere Prüfung direkt in die Wissensbasis übernommen, d.h. der Agent vertraut den Nachrichten von außen.
NEW_PLAN	extern	Mitteilung eines neuen Plans zur Ergänzung der Planbibliothek von außen. So kann der Agent z.B. „lernen“, weitere Ziele zu erreichen. Auch hier erfolgt keine Überprüfung, d.h. der Agent muss der Korrektheit von Plänen vertrauen.
NEW_GOAL	intern	Ein Ereignis dieser Art ist ein Initialziel; die Ziele im GOALS-Feld der Spezifikation sind als interne NEW_GOAL-Ereignisse anzusehen.
NEW_GOAL	extern	Dies ist ein Ereignis, das wie in AgentGHC mit der Auswahl eines anwendbaren Plans und Abarbeitung in der Intentionsstruktur behandelt wird.
REPLY	extern	Eine Antwort auf eine vorher gestellte Anfrage. Eine Intention wartet auf diese Antwort. Durch Angabe einer Referenz kann die wartende Intention gefunden und die Antwort dort direkt weiterverarbeitet werden.
SUBGOAL	intern	Von einer Intention erzeugtes Ereignis. Dieses Ereignis wird wie im AgentGHC-Zyklus beschrieben behandelt und in der Intentionsstruktur auf den Stapel der erzeugenden Intention gelegt.
PRECONDITION_GOAL	intern	Dieses Ereignis ist Teil von spekulativen Berechnungen bei der Kontextüberprüfung relevanter Pläne (vgl. Abschnitt 6.4.).
SUSPEND	intern und extern	Suspendierung einer Intention. Die Abarbeitung einer Intention kann vorübergehend angehalten werden.Dem Ereignis muss eine Referenz auf eine Intention mitgegeben werden.
RESUME	intern und extern	Wiederaufnahme einer suspendierten Intention. Dem Ereignis muss eine Referenz auf eine Intention mitgegeben werden.

Für eine angemessene Behandlung der verschiedenen Ereignistypen werden zunächst die Ereignisse um eine Typangabe t erweitert zu einem Tripel $\varepsilon = \langle e, Q, t \rangle$. Nun kann anhand des Ereignistyps im Interpreter entweder sofort eine Änderung des internen Zustands oder die Initiierung einer neuen Intention durch Planauswahl erfolgen. Die Abbildung 21 auf der nächsten Seite zeigt die Änderungen im abstrakten Interpreter auf, die für die neuen Ereignistypen notwendig sind.

Für jeden Ereignistypen wird ein case-Fall eingerichtet, in dem das Ereignis entsprechend behandelt wird. Der in Abbildung 21 letzte angegebene Zweig

```
case t == (NEW_GOAL or PRECONDITION_GOAL or SUBGOAL)
```

ist der ursprünglich einzige vorhandene Fall im AgentGHC-Interpreter. Der nicht angegebene Rest des Interpreterzyklus bleibt zunächst unverändert. Am Ende dieses

Kapitels werden jedoch noch weitere Ergänzungen zum Interpreterzyklus gemacht (vgl. Abschnitt 6.4.).

Abbildung 21 Ereignisbehandlung im CASA-Interpreter

```
While Agent is alive do
  if Erg ≠ ∅ then
    ε = < e, Q, t > = Sε(Erg)      // Ereignisauswahl
    Erg = Erg \ {ε}
                                   // Neu im Interpreter: Unterscheidung nach Ereignistypen
    case t == NEW_FACT
      Bel = Bel ∪ {e}              // Wissensbasis ändern

    case t == NEW_PLAN
      Strat = Strat ∪ {e}          // Planbibliothek ergänzen

    case t == REPLY
      setReply (Int, e)            // Intention die Antwort zukommen lassen

    case t == SUSPEND
      suspend (Int, e)             // Intention suspendieren

    case t == RESUME
      resume (Int, e)              // Intention reaktivieren

    case t == (NEW_GOAL or PRECONDITION_GOAL or SUBGOAL)
      Rε = {pθ | p ∈ Strat ∧ θ ist relevanter Unifikator bzgl. ε}                  // relevante Strategien
      Aε = {pφ | φ= θδ ∧ pθ ∈Rε ∧ (θδ) ist anwendbarer Unifikator bzgl. ε }       // anwendbare Strategien
      Pε = Sp(Aε)

      ...
    // weiter wie im Originalinterpreter (vgl. Anhang A)
```

6.3.2. Auswahlfunktionen

An drei Stellen im Interpreterzyklus müssen Auswahlen getroffen werden: Zunächst muss ein Ereignis aus der Menge **Erg** im Sensor ausgewählt werden. Außerdem müssen für neue Ziele relevante und anwendbare Pläne gefunden werden. Relevante Pläne für ein Ziel sind die Pläne, deren Relation im GOAL-Feld zu der Relation des Ziels passen, d.h. denselben Relationennamen und dieselbe Argumentanzahl haben. Bei der Auswahl von anwendbaren Plänen muss jedoch eine Auswahlfunktion angegeben werden, da die Vorbedingungen mehrerer Pläne zur Erreichung des Ziels zutreffen können. Schließlich muss in der Intentionsstruktur eine der aktuell vorhandenen Intentionen ausgewählt werden. Die Intentionsstruktur ist als Multikeller anzusehen: Unterziele eines Intentionsstapels können auch nebeneinander eingefügt werden. Dadurch muss die Auswahlfunktion für Intentionen eventuell ein zweites Mal angewendet werden (vgl. Abschnitt 6.3.6.).

Die drei benötigten Auswahlfunktionen werden in der folgenden Tabelle zusammen mit verschiedenen Auswahltypen beschrieben. Bei der Agentenspezifikation gibt man

für jede Auswahlfunktion einen Typen an. Dabei sollen die in der Tabelle beschriebenen Typen nur eine Ausgangsbasis bilden und es soll die Möglichkeit geben, noch weitere Funktionstypen zu CASA hinzufügen zu können.

Funktion	**Auswahlart**		
	zufällig	reihenfolgetreu	nach Gewichtung
Ereignisauswahl	RANDOM	OLDEST_FIRST	- nicht vorgesehen -
Planauswahl	RANDOM	- nicht vorgesehen -	HIGHEST_PRIORITY
Intentionsauswahl	RANDOM	OLDEST_FIRST	HIGHEST_PRIORITY

Die Ereignisauswahl nach Gewichtung ist nicht vorgesehen, denn nicht jedes Ereignis hat einen solchen Wert, z.B. ein Ereignis vom Typ UPDATE_FACT. Die Ausstattung aller Ereignistypen mit einer Gewichtung könnte aber Gegenstand einer Erweiterung von CASA sein. Die Planauswahl nach Reihenfolge hätte denselben Effekt wie die Planauswahl nach Gewichtung, denn die Pläne werden in der Planbibliothek nach absteigender Gewichtung sortiert verwaltet (vgl. Abschnitt 6.3.4.). Daher ist diese Auswahlart nicht vorgesehen. Während das Ergebnis bei den Auswahlfunktionen RANDOM und OLDEST_FIRST immer eindeutig ist, kann es bei der Auswahlfunktion HIGHEST_PRIORITY passieren, dass mehrere Pläne bzw. Intentionen mit gleicher höchster Gewichtung existieren. In diesem Fall wird einer dieser Pläne bzw. eine dieser Intentionen zufällig ausgewählt. Die Auswahlfunktionen werden bei der Spezifikation eines Agenten angegeben und bleiben fest für den gesamten Laufzeit des Agenten. Die Syntax für die Angabe der Auswahlfunktionen sieht folgendermaßen aus:

```
FUNCTIONS:
    EVENTSELECT:        <select function>
    PLANSELECT:         <select function>
    INTENTIONSELECT:    <select function>
```

Für den Einlagerungsagenten könnte folgende Belegung der Auswahlfunktionen getroffen werden:

```
FUNCTIONS:
    EVENTSELECT:        OLDEST_FIRST;
    PLANSELECT:         HIGHEST_PRIORITY;
    INTENTIONSELECT:    HIGHEST_PRIORITY;
```

Die Auswahl von Ereignissen im Sensor in der Reihenfolge, in der sie eingetroffen sind, ist von den vorgeschlagenen Möglichkeiten die naheliegendste. Aber vor dem Hintergrund, dass für einen Einlagerungsagenten wichtige Aufträge eintreffen können, wäre auch eine Auswahlfunktion sinnvoll, die solche Ereignisse bevorzugt auswählt. Dies kann aber nur durch eine applikationsabhängige Auswahlfunktion geschehen oder dadurch, dass man *alle* Ereignistypen mit einer Gewichtung versieht und eine HIGHEST_PRIORITY-Funktion für Ereignisse ergänzt.

6.3.3. Wissensbasis

In der Wissensbasis werden zunächst alle Fakten gespeichert, die initial in der Spezifikation angegeben sind. Im Laufe der Zeit kann sich dieses Wissen durch Aktionen in Intentionen oder durch Ereignisse verändern. Die möglichen Aktionen und Ereignisse sind bereits beschrieben worden. Es muss jedoch eine semantische Grundlage für die Verwaltung der Fakten entwickelt werden, denn es stellen sich einige Fragen bei die Behandlung von Sonderfällen: Wann sind zwei Fakten gleich? Was passiert bei mehrfachem Einfügen desselben Fakts? Was passiert, wenn ein Fakt geändert werden soll, das gar nicht in der Wissensbasis existiert? Die verschiedenen möglichen Fälle werden in diesem Abschnitt systematisch erfasst und für jeden Fall eine Lösung angegeben. Zum besseren Verständnis wird dazu der Begriff der Gleichheit von Relationen in der folgenden Definition festgelegt.

Definition (Gleichheit von Relationen):

Gegeben seien die Relationen $r_1 = name_a(a_1 \ldots a_m)$ und $r_2 = name_b(b_1 \ldots b_n)$. Die Relationen r_1 und r_2 werden genau dann als **gleich** bezeichnet, wenn gilt:

- $name_a = name_b$
- $m = n$
- $\forall\, i \in \{1, \ldots, n\} : Typ(a_i) = Typ(b_i) \wedge Wert(a_i) = Wert(b_i)$

Hinzufügen von Fakten. Das Einfügen neuer Fakten kann mittels der Aktion ADD <relation> erfolgen. Grundsätzlich soll kein bereits existierendes Fakt durch diese Aktion verändert werden. Wenn der Relationenname noch nicht in der Wissensbasis existiert, wird das neue Fakt einfach zur Wissensbasis hinzugefügt. Falls der Relationenname aber schon existiert, werden alle betreffenden Fakten auf Gleichheit zu dem einzufügenden Fakt überprüft. So soll verhindert werden, dass Fakten doppelt in die Wissensbasis aufgenommen werden. Ein Fakt wird also nur hinzugefügt, wenn keine gleiche Relation in dem oben definierten Sinne in der Wissensbasis existiert. An dieser Stelle muss man sich fragen, inwieweit man das Hinzufügen von widersprüchlichen Fakten überprüfen soll, z.B. wenn das Fakt «available "TRUE"» in der Wissensbasis bereits existiert und die Aktion «ADD available "FALSE"» ausgeführt werden soll. Nach nach Ausführung der ADD-Aktion stehen beide Relationen in der Wissensbasis:

```
FACT available "TRUE"
FACT available "FALSE"
```

Je nachdem, wie der Zugriff auf ein Fakt mit dem Namen available erfolgt, ist das Ergebnis entweder "TRUE" oder "FALSE". Es soll aber nicht Aufgabe des Agenten sein, Wissen auf Konsistenz zu prüfen. Während es in diesem Beispiel noch leicht realisierbar wäre, kann es bei komplexeren Relationen sehr schwer werden Aussagen über konsistente Fakten in der Wissensbasis zu machen. Dazu benötigt man weitere Regeln (Constraints), wie sie auch in Datenbanken benutzt werden. Bei der Spezifikation von Agenten muss also darauf geachtet werden, an solchen Stellen besser eine UPDATE-Aktion zu benutzen.

Löschen von Fakten. Löschen von Fakten geschieht durch die Aktion DELETE <relation>. In der Wissensbasis werden alle Fakten mit dem gleichen Relationennamen gesucht. Es ist möglich, dass mehrere Fakten gelöscht werden, nämlich dann, wenn bei der DELETE-Aktion weniger (oder keine) Argumente angegeben werden als die betreffenden Fakten der Wissensbasis haben, z.B. seien in der Wissensbasis folgende Fakten vorhanden:

```
FACT sortline 1   10  20
FACT sortline 2   20  20
FACT sortline 3   30  20
```

Bei dem Versuch, ein Fakt zu löschen, können verschiedene Fälle auftreten, die in der folgenden Tabelle aufgeführt sind und sich auf das obige Beispiel beziehen.

Aktion	Beschreibung
DELETE sortline	Alle Fakten in der Wissensbasis mit dem Relationennamen "sortline" werden gelöscht. Im Beispiel bleibt kein Fakt in der Wissensbasis übrig.
DELETE sortline 1	Alle Fakten in der Wissensbasis mit dem Relationennamen "sortline" und dem ersten Argument mit Wert "1" (Typ: Integer) werden gelöscht. Im Beispiel wird also genau ein Fakt gelöscht.
DELETE sortline 10	Im Beispiel gibt es kein Fakt mit dem ersten Argumentwert "10", also bleiben alle Fakten bestehen.
DELETE sortline 1 80	Auch hier stimmen nicht alle der angegebenen Argumentwerte mit einem Fakt der Wissensbasis überein. Kein Fakt wird gelöscht.
DELETE sortline 1 10 20 "Sortline-1"	Es gibt zwar ein Fakt im Beispiel mit demselben Relationennamen und denselben Werten bei den ersten drei Argumenten, aber dieses Fakt hat kein viertes Argument, daher wird es nicht gelöscht.
DELETE cars	Es gibt keine Fakten im Beispiel mit dem Relationennamen "cars", also kann auch keins gelöscht werden.

Ändern von Fakten. Ändern von Fakten in der Wissensbasis geschieht entweder durch die Aktion UPDATE <relation>$_1$ <relation>$_2$ oder durch eine Nachricht vom Typ UPDATE_FACT, deren Inhalt entsprechend aufgebaut ist. Semantisch verhält sich die UPDATE-Aktion wie die Hintereinanderausführung der beiden oben beschriebenen Aktionen DELETE <relation>$_1$ und ADD <relation>$_2$.

Abfragen von Fakten. Auf die in der Wissensbasis gespeicherten Fakten kann man von Plänen aus mit den Aktionen GETFACT und GETVALUES zugreifen. Bei der Abfrage wird nach gleichen Relationen gesucht in dem Sinne, dass die Relationennamen und die Argumentanzahl übereinstimmen müssen. In den folgenden beiden Tabellen sind die verschiedenen Möglichkeiten angegeben, welche Bedeutung Konstanten und Variablen als Argumente beim Abfragen von Wissen haben können (wiederum basierend auf obigem Beispiel). Semantisch unterscheiden sich die beiden Aktionen dadurch, dass mit GETFACT alle angegebenen Variablen neu gebunden werden (ein

Fakt als Ganzes wird aus der Wissensbasis geholt), während bei GETVALUES die Variablenwerte zum Auffinden eines Fakts benutzt werden und unverändert bleiben (es werden nur neue Werte für noch ungebundene Variablen geholt).

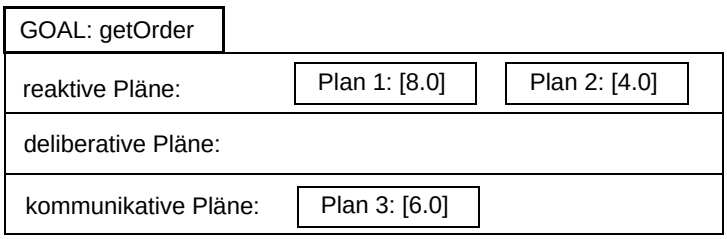
Aktion	Beschreibung
GETFACT sortline $Line $X $Y	Der erste Eintrag in der Wissensbasis, der den Relationennamen "sortline" und drei Argumente hat, wird gesucht. Die Werte dieses Fakts werden an die drei Variablen gebunden. Welche der Sortierlinien aus der Wissensbasis geliefert wird, ist in diesem Fall nicht von vornherein bestimmbar.
GETFACT sortline 1 $X $Y	Durch die Konstante 1 als erstes Argument werden die Variablen $X und $Y mit den Werten 10 und 20 belegt.
GETFACT sortline 4 $X $Y	Es gibt kein Fakt "sortline" mit erstem Argumentwert 4, also schlägt diese Aktion fehl.
GETFACT sortline $Line	Es gibt kein Fakt "sortline" mit nur einem Argument, die Aktion schlägt fehl.
GETFACT sortline 1 $X $Y $Name	Es gibt kein Fakt "sortline" mit 4 Argumenten, also schlägt auch diese Aktion fehl.
GETFACT sortline 1 10 20	Bei dieser Aktion gibt es keine Variablen, an die ein Wert gebunden werden soll. Die Aktion wird aber trotzdem erfolgreich beendet, weil es ein Fakt "sortline" mit den angegebenen Werten in der Wissensbasis gibt.
GETFACT sortline 1 30 30	Diese Aktion schlägt im Gegensatz zu der vorherigen fehl.

Aktion	$Line	$X	$Y	Beschreibung
GETVALUES sortline $Line $X $Y	nicht geb.	nicht geb.	nicht geb.	Dieser Fall entspricht genau einem GETFACT sortline $Line $X $Y
GETVALUES sortline $Line $X $Y	1	nicht geb.	nicht geb.	Diese Aktion entspricht genau einem GETFACT sortline 1 $X $Y
GETVALUES sortline $Line $X $Y	4	nicht geb.	nicht geb.	Diese Aktion schlägt fehl, weil es kein Fakt in der Wissensbasis gibt, das den angegebenen Werten entspricht. Der ausführende Plan wird abgebrochen.
GETVALUES sortline 1 $X $Y		10	nicht geb.	Durch diese Aktion wird der Variablen $Y der Wert 20 zugewiesen.
GETVALUES sortline 1 $X $Y		20	nicht geb.	Auch diese Aktion schlägt fehl, weil es kein Fakt in der Wissensbasis gibt, das den angegebenen Werten entspricht.
GETVALUES sortline 1 $X $Y		10	20	Diese Aktion wird erfolgreich beendet, weil es ein Fakt "sortline" mit den Argumentwerten gibt. Auf die Variablen hat diese Aktion keine Auswirkung.
GETVALUES sortline 1				Diese Aktion schlägt fehl, denn es gibt kein Fakt "sortline" mit nur einem Argument.
GETVALUES sortline 1 $X $Y $Z		egal	egal	Diese Aktion schlägt fehl, denn es gibt kein Fakt "sortline" mit vier Argumenten.

6.3.4. Planbibliothek

Sowohl die initial in der Spezifikation angegebenen als auch die durch Ereignisse vom Typ NEW_PLAN erhaltenen Pläne werden in einer Bibliothek verwaltet. Innerhalb der Bibliothek werden *Ordner* für die Pläne angelegt, die dasselbe Ziel zu erreichen versprechen. Jeder Ordner ist in drei Sektionen entsprechend den drei Hierarchiestufen (reaktiv, deliberativ, kommunikativ) von Plänen unterteilt. So kann für ein gegebenes Ziel ein geeigneter Ordner gesucht und innerhalb des Ordners auf die Pläne der verschiedenen Hierarchiestufen zugegriffen werden. Innerhalb jeder Hierarchiestufe sind die Pläne nach ihrer Gewichtung absteigend sortiert. Dies ist vorteilhaft, wenn die Planauswahlfunktion HIGHEST_PRIORITY gewählt wird und mehrere anwendbare Pläne zur Auswahl stehen: Die Pläne mit der höchsten Priorität stehen am Anfang der Liste, sodass der Zugriff auf diese Pläne sehr einfach geschehen kann. Abbildung 22 veranschaulicht den Aufbau eines Ordners der Planbibliothek an einem Beispiel.

Abbildung 22 Ordner in der Planbibliothek mit Plänen für das Ziel "getOrder"



Wir betrachten die verschiedenen Pläne für das Ziel "getOrder" eines Einlagerungsagenten. Der reaktive Plan 1 prüft in seiner Vorbedingung, ob in der Wissensbasis ein wichtiger Auftrag vorliegt. Wenn ja, kann dieses Fakt sofort weiterverarbeitet werden. Der reaktive Plan 2 prüft, ob in der Wissensbasis ein normaler Auftrag vorliegt. Plan 2 hat eine kleinere Gewichtung und steht daher in der Liste der reaktiven Pläne hinter Plan 1. Der kommunikative Plan 3 ist für den Fall vorgesehen, dass der Einlagerungsagent erst beim Auftragsagenten nach einem neuen Auftrag fragen muss. Weil Plan 3 ein kommunikativer Plan ist, spielt seine Gewichtung in diesem Fall keine Rolle; aufgrund der Hierarchieabstufung wird er als dritter und letzter Plan auf Anwendbarkeit geprüft.

6.3.5. Kontexttest

Wenn in einem Interpreterzyklus ein Ereignis ausgewählt wird, das die Verfolgung eines Ziels auslöst, muss für dieses Ziel ein geeigneter Plan aus der Planbibliothek gefunden werden. In AgentGHC wird zwischen relevanten und anwendbaren Plänen unterschieden. Relevante Pläne sind diejenigen, die versprechen, das gegebene Ziel zu erfüllen. Man erkennt relevante Pläne in CASA einfach daran, dass sie im GOAL-Feld denselben Relationennamen wie das gegebene Ziel führen und die Relationen dieselbe Argumentanzahl haben. Durch die in Abschnitt 6.3.4. beschriebene Struktur der Planbibliothek ist es leicht, alle relevanten Pläne zu finden: Sie stehen alle in einem Ordner der Planbbibliothek.

Aufwendiger wird es, die relevanten Pläne einem Kontexttest zu unterziehen. Entsprechend der Planhierarchie werden zunächst die Vorbedingungen der relevanten reaktiven Pläne überprüft. Dies geschieht durch Abfragen und Vergleichen von Fakten der Wissensbasis. Gibt es mindestens einen reaktiven Plan, dessen Vorbedingungen alle erfüllt sind, wird mittels der Planauswahlfunktion ein Plan ausgewählt, daraus eine neue Intention gebildet und in der Intentionsstruktur abgelegt (vgl. Abschnitt 6.3.6.). Gibt es keine anwendbaren reaktiven Pläne, so muss als nächstes überprüft werden, ob es deliberative Pläne gibt, deren Vorbedingungen erfüllt sind. Falls es auch hier keinen anwendbaren Plan gibt, folgt als letzte Stufe die Überprüfung von relevanten kommunikativen Plänen. Man spricht bei der Überprüfung von Vorbedingungen deliberativer und kommunikativer Pläne auch von spekulativen Berechnungen. Zur Beschreibung des Ablaufs dieser Berechnungen benötigt man aber erst einen Einblick in die Verwaltung von Intentionen. Auf spekulative Berechnungen wird daher in einem eigenen Abschnitt später noch näher eingegangen (vgl. Abschnitt 6.4.).

6.3.6. Intentionsverwaltung

Die sogenannte Intentionsstruktur übernimmt die Verwaltung der vom Agenten aktuell verfolgten Intentionen. In diesem Abschnitt wird zunächst die Struktur von Intentionen festgelegt und im Anschluss daran die operationale Semantik der Intentionsstruktur genauer erklärt.

Struktureller Aufbau von Intentionen. Eine Intention wird gebildet, wenn zu einem Ziel ein anwendbarer Plan zur Erreichung des Ziels gefunden wurde. Dieser Plan wird dem Ziel zugeordnet, aber da in dem Plan in der Regel Variablen benutzt werden, benötigt man auch noch eine weitere Komponente: eine eigene Liste der benutzten Variablen für ihre lokalen Werte. Es kann vorkommen, dass ein Plan für mehrere Ziele zur gleichen Zeit benötigt wird, und daher reicht es nicht, die Liste mit den Variablenwerten an den Plan zu koppeln, sondern es muss jedesmal eine solche Variablenliste neu angelegt werden, wenn ein Plan einem Ziel zugeordnet wird.

Für die Verwaltung von Intentionen müssen aber noch weitere Informationen gehalten werden. Die folgende Tabelle zeigt die einzelnen Komponenten einer Intention zusammenfassend auf.

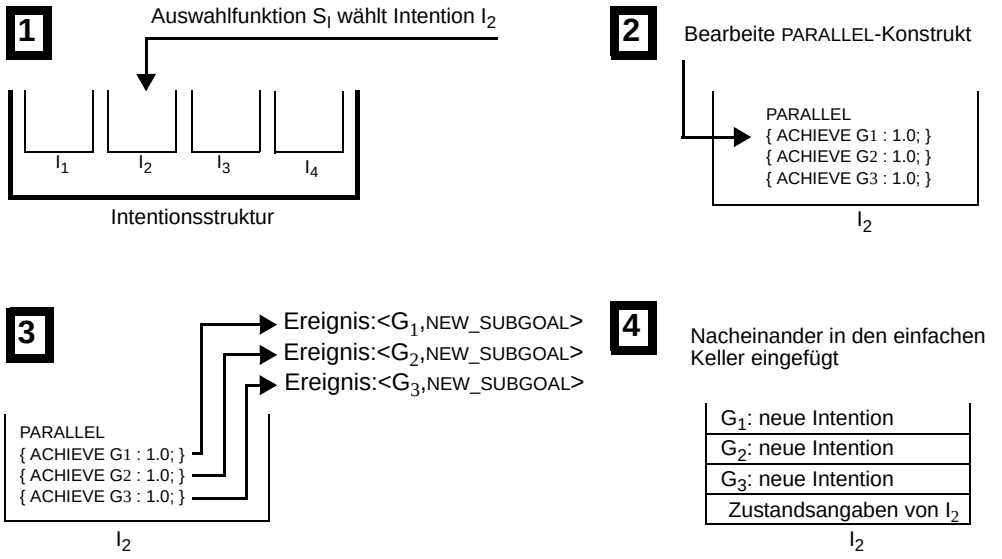
Intentionskomponente	Beschreibung
goal	Die Beschreibung des Ziels, also ein ACHIEVE-Konstrukt mit einer Relation
applicable plan	Zusammensetzung aus einer Referenz auf einen Plan aus der Planbibliothek und einer eigenen Liste der Planvariablen mit Platz für ihre lokalen Werte
unique ID	Eine eindeutige Identifikationsnummer
timestamp	Ein Zeitstempel, der den Zeitpunkt der Generierung der Intention speichert
status	Mögliche Werte: aktiv (active), wartend (waiting), unterbrochen (suspended)
parent goal	Referenz auf das Elternziel, falls es sich um ein Unterziel handelt
subgoals	Eine Intention muss ihre aktuell verfolgten Unterziele kennen, daher müssen diese in einer Referenzliste gespeichert sein.
runtime state	Status der Abarbeitung des anwendbaren Plans

Liste der Top-Level-Intentionen. Um beliebig viele Intentionen nebeneinander verwalten zu können, muss eine Liste der aktuellen Intentionen angelegt werden. Die Liste nimmt als Elemente alle Hauptintentionen (oder: Top-Level-Intentionen) auf; es handelt sich dabei um Intentionen, die durch die Bearbeitung von Ereignissen des Typs NEW_GOAL entstehen. Im Rahmen spekulativer Berechnungen werden allerdings auch noch Intentionen basierend auf Ereignissen vom Typ PRECONDITON_GOAL in die Liste aufgenommen (vgl. Abschnitt 6.4.). Die folgende Tabelle zeigt die wichtigsten Funktionen für die Verwaltung der Intentionen in einer Liste auf.

Funktion	Beschreibung, Ergebnis
isEmpty()	TRUE, falls keine Top-Level-Intention vorhanden sind, sonst FALSE
add(Intention i)	TRUE, falls eine neue Intention eingefügt werden kann, sonst FALSE
getUnblockedIntentions()	Liefert eine Liste der nicht suspendierten Top-Level-Intentionen
selectTopLevelIntention(Function f)	Liefert eine nicht suspendierte Intention, die durch die Auswahlfunktion f ermittelt wird
selectSubgoal(Intention i, Function f)	Liefert einUnterziel der (Top-Level-)Intention i, ausgewählt von der Funktion f
removeTopLevelIntention(Intention i)	Löscht explizit eine Top-Level-Intention
suspend(Intention i)	Unterbricht die Ausführung einer Intention
resume(Intention i)	Nimmt die Abarbeitung einer Intention wieder auf

Parallele Unterziele. Jede Intention kann während ihrer Abarbeitung auch Unterziele durch Abschicken eines NEW_SUBGOAL-Ereignisses an den Sensor erzeugen. Dort wartet es darauf von der Ereignisauswahlfunktion S_ε ausgewählt und weiterbearbeitet zu werden. Kommt es (nach ggf. komplexeren Berechnungen) dazu, dass ein Unterziel als Intention in die Intentionsstruktur eingefügt werden soll, wird es nicht als Top-Level-Intention, sondern als Unterintention in die subgoals-Liste der erzeugenden Intention aufgenommen. Man kann diese Art des Hinzufügens wie einen Keller betrachten: Eine Unterintention wird auf ihre erzeugende Intention „gepushed“, sodass nun zuerst das Unterziel erreicht sein muss, bevor die erzeugende Intention weiterbearbeitet werden kann. Allerdings kommt es bei parallelen Unterzielen mit einem einfachen Keller zu Problemen, wie die folgende Abbildung veranschaulicht.

Abbildung 23 Nebenläufige Berechnung



In Schritt 1 wird ein Abarbeitungsschritt der Intention I_2 durchgeführt. Dabei werden parallel drei Unterziele erzeugt (Schritt 2) und an den Sensor geschickt (Schritt 3). Da pro Zyklus nur ein Ereignis im Sensor ausgewählt wird, können die Unterziele nur nach und nach zu ihrer erzeugenden Intention hinzugefügt werden. Wenn man nur einen einfachen Keller betrachtet, würde der ursprünglich bezweckte parallele Charakter der Unterziele gänzlich verschwinden (Schritt 4). Es muss also die Möglichkeit geben, Unterziele auch nebeneinander im Keller abzulegen (Multikeller). Durch den Sensor entsteht zwar immer noch eine Verzögerung beim Anlegen der parallelen Unterziele, doch diese ist in Relation zur vollständigen Abarbeitung eines Plankörpers nur kurz, sodass man in der Regel von einer verzahnten Ausführung der Unterziele ausgehen kann.

Also muss die Intentionsauswahl in zwei Schritten erfolgen: Im ersten Schritt wird eine der Top-Level-Intentionen aus der Liste der Intentionsstruktur ausgewählt, im zweiten Schritt wird aus der Menge der eingekellerten „obersten“ Intentionen, im Folgenden auch Blätter genannt, ein Element ausgewählt. Die Reihenfolge des Einfügens ist also (bis auf sehr wenige Zyklen Verzögerung) nicht für die Abarbeitung der Intentionen maßgebend, sondern nur von der Auswahlfunktion S_I abhängig.

Man sollte sich darüber im Klaren sein, dass der Zugriff auf die Blätter einer Intention sehr aufwendig werden kann, da es sich durch die möglichen parallelen Unterziele im Grunde um eine Baumstruktur für jede Intention handelt. Analog zur Tiefensuche in Bäumen wird nach Auswahl einer Top-Level-Intention eine Liste der Blätter erstellt, und zwar immer wieder bei jedem Durchlauf des Interpreters. Für komplexe Intentionen - mit vielen Unterzielstufen und parallelen Unterzielen - kann es sich daher lohnen, eine Liste der Blätter zusätzlich zu verwalten. Der zusätzliche Verwaltungsaufwand für das Einfügen und Löschen von Unterzielen in dieser zusätzlichen Liste bedeutet einen im Verhältnis geringen Mehraufwand; in einem Plankörper sind in der Regel viele Aktionen auszuführen, d.h. als Element in dieser Liste wird ein Blatt über viele Zyklen

bestehen bleiben. So bleibt das wiederholte Durchsuchen von Intentionsstapeln nach Blättern während vieler Zyklen erspart.

Solange die Intentionsstapel jedoch nicht besonders groß werden, reicht das einfache Suchen nach Blättern analog zur Tiefensuche im Baum aus. Allerdings muss man dieses Vorgehen bei komplexen Intentionen beobachten und gegebenenfalls durch den Einsatz einer Liste wie vorgeschlagen ersetzen.

Die Bestimmung der Gewichtung einer Intention. An verschiedenen Stellen treten in AgentGHC und in CASA Gewichtungen auf. Der Zusammenhang zwischen Gewichtung von Zielen, Unterzielen und Plänen muss aber noch dargestellt werden:

Jedes Ziel und jeder Plan hat eine Gewichtung. Bei der Bildung einer Intention wird die Gewichtung von Ziel und anwendbarem Plan summiert. Soll während der Ausführung der Intention ein Unterziel erreicht werden, so addiert sich zu der Gewichtung des Unterziels (diese Angabe ist auch zwingend) noch die Gewichtung der erzeugenden Intention. Dies setzt sich immer weiter fort, wenn ein Unterziel weitere Unterziele erzeugt.

Je größer also die Unterzielhierarchie wird, desto größer wird die Gewichtung der neuen Unterziele, sodass diese bevorzugt gegenüber anderen Intentionen bearbeitet werden, wenn die Gewichtung in die Intentionsauswahl einfließt. Dies setzt allerdings voraus, dass sich die Gewichtungen von Plänen bzw. Zielen in ähnlichen Dimensionen bewegen.

Eindeutige Identifikation von Unterzielen. Neben der Liste für Top-Level-Intentionen wird in der Intentionsstruktur noch durch einen Zähler die Anzahl der bisher angelegten Intentionen festgehalten. Dieser Zähler wird auch zur Vergabe der eindeutigen Identifikationsnummer "uniqueId" für Top-Level-Intentionen verwendet.

Für eine Unterintention setzt sich die Identifikationskennung (ID) aus der ID ihrer erzeugenden Intention, einem Punkt als Trennzeichen und einer weiteren Nummer zusammen, die die erzeugende Intention vergibt. Beispielsweise bekommt die vierte erzeugte Unterintention der Top-Level-Intention 1 als ID die Zeichenkette "1.4" zugeteilt. Die erste Unterintention von "1.4" erhält dann die ID "1.4.1". So kann anhand der ID immer auf die richtige Intention zugegriffen werden, indem man sich ausgehend von einer Top-Level-Intention entlang der Unterintentionen bis zur gewünschten Ziel bewegt.

Die ID wird auch benötigt, wenn eine Intention durch ein Suspend-Ereignis unterbrochen werden soll. Die eindeutige ID wird dem Ereignis als Argument mitgegeben, sodass der Interpreter über die Intentionsstruktur die gewünschte Intention unterbrechen kann. Eine Intention wird unterbrochen, indem ihr Status auf "suspended" gesetzt wird. Zur Behandlung eines Resume-Ereignisses wird die Intention auf die gleiche Weise ermittelt und ihr Status wieder auf "active" gesetzt.

Die eindeutige ID wird auch bei REPLY-Ereignissen benötigt: Wenn eine Anfragenachricht abgeschickt wird, bekommt sie implizit die ID der auslösenden Intention als Referenz mitgegeben. Diese ID ist auch in der Antwortnachricht enthalten, sodass die wartende auslösende Intention über die ID wiedergefunden und ihr der Inhalt der Nachricht zugeordnet werden kann.

6.4. Spekulative Berechnungen

Im Interpreterzyklus von AgentGHC, wie er in Abschnitt 3.5. beschrieben ist, wird aus der Menge der relevanten Pläne durch einen Kontexttest eine Teilmenge von anwendbaren Plänen ermittelt. Allerdings wird der Ablauf des Kontexttests aus Gründen der Übersichtlichkeit in AgentGHC nicht näher beschrieben, sondern nur durch die folgenden Zeilen im abstrakten Interpreter angegeben:

$R_\varepsilon = \{p\theta \mid p \in \textbf{Strat} \wedge \theta \text{ ist relevanter Unifikator bzgl. } \varepsilon\}$ *// relevante Strategien*

$A_\varepsilon = \{p\varphi \mid \varphi = \theta\delta \wedge p\theta \in R_\varepsilon \wedge (\theta\delta) \text{ ist anwendbarer Unifikator bzgl. } \varepsilon\}$ *// anwendbare Strategien*

Die Überprüfung reaktiver Pläne stellt noch keine besonderen Probleme dar. Für einen relevanten reaktiven Plan können alle Vorbedingungen sofort überprüft werden, denn es handelt sich dabei nur um einfache TEST-Aktionen oder Abfragen der Wissensbasis. Doch wenn auch Vorbedingungen deliberativer oder kommunikativer Pläne zu überprüfen sind, wird in AgentGHC der Interpreterzyklus an dieser Stelle ggf. für eine längere Zeit angehalten, z.B. wenn ein kommunikativer Plan auf eine Antwort warten muss. Es kann auch zu einem rekursiven Aufruf des Interpreters kommen, wenn ein deliberativer Plan ein Unterziel in seiner Vorbedingung erreichen möchte.

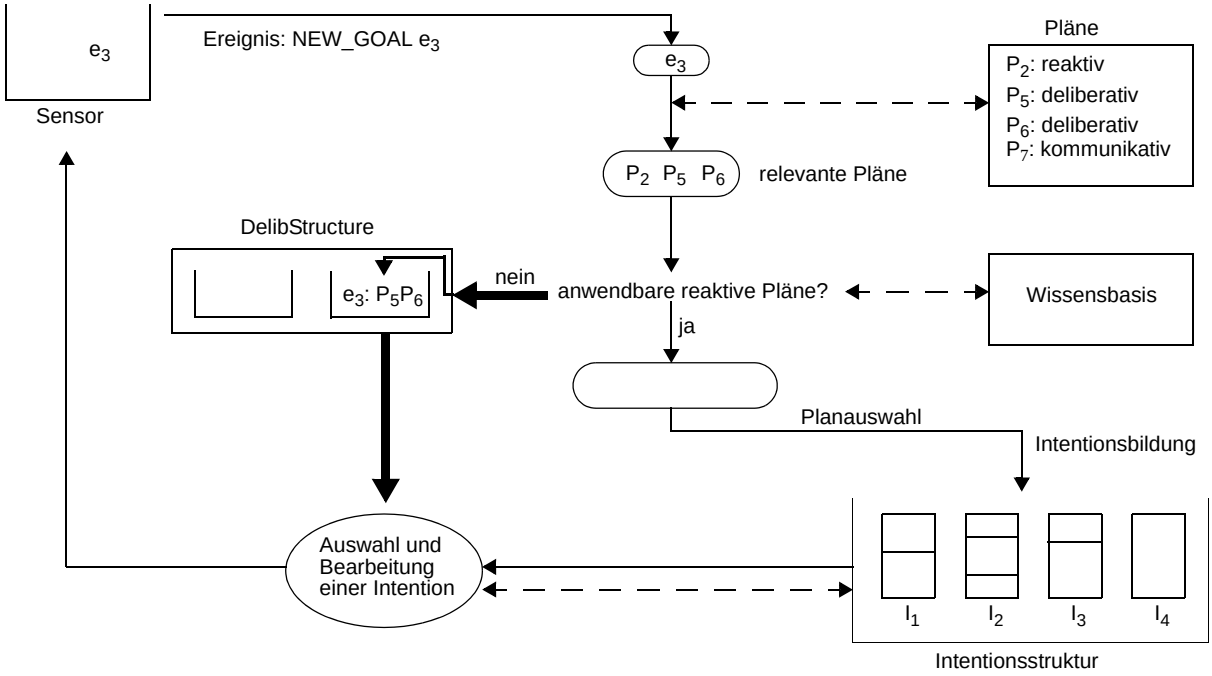
Im Folgenden wird ein Modell für die Überprüfung der Vorbedingungen deliberativer und kommunikativer Pläne, auch spekulative Berechnung genannt, vorgestellt.

6.4.1. Kontexttest für deliberative Pläne

Wie schon erwähnt kommt es in AgentGHC zu einem rekursiven Aufruf des Interpreters, wenn bei der Überprüfung von deliberativen Plänen ein Unterziel erreicht werden soll. Hierdurch kann auch eine Kaskade rekursiver Aufrufe des Interpreters auftreten, nämlich immer dann, wenn für ein neues Unterziel wieder deliberative Pläne überprüft werden müssen. Leider ist die Vorstellung von rekursiven Aufrufen eines Interpreters nicht sehr hilfreich für das Verständnis der operationalen Semantik eines BDI-Agenten.

Für den Kontexttest deliberativer Pläne wird daher eine neue Komponente in den abstrakten Interpreter eingefügt: die **DelibStructure**. In diese Komponente werden immer dann neue Elemente eingefügt, wenn Vorbedingungen *relevanter deliberativer* Pläne zu überprüfen sind. Ein Element der DelibStructure ist ein Ziel mit einer Reihe von deliberativen Plänen und Listen für ihre Variablenwerte. Die DelibStructure bearbeitet ihre Elemente unabhängig vom normalen Interpreterzyklus. Der Interpreter gibt ein neues Ziel unter Angabe der relevanten deliberativen Pläne an die DelibStructure ab, in der alle notwendigen Überprüfungen verwaltet werden. Die Abgabe eines Ziels an die DelibStructure ist in der nachfolgenden Abbildung 24 durch fett gedruckte Pfeile gekennzeichnet. Der Interpreterzyklus wird anschließend mit der Auswahl und Bearbeitung einer Intention fortgesetzt.

Abbildung 24 DelibStructure für deliberative Pläne



Die DelibStructure ist ähnlich wie die Intentionsstruktur aufgebaut, d.h. mehrere Elemente können nebeneinander verwaltet werden. Unter einem Element in der DelibStructure ist ein Ziel zusammen mit einer Reihe von relevanten deliberativen Plänen zu verstehen. Im Unterschied zur Intentionsstruktur werden aber hier aussschließlich die Vorbedingungen von Plänen ausgeführt, sodass von der DelibStructure aus keine Veränderungen der Wissensbasis und der Umgebung möglich sind.

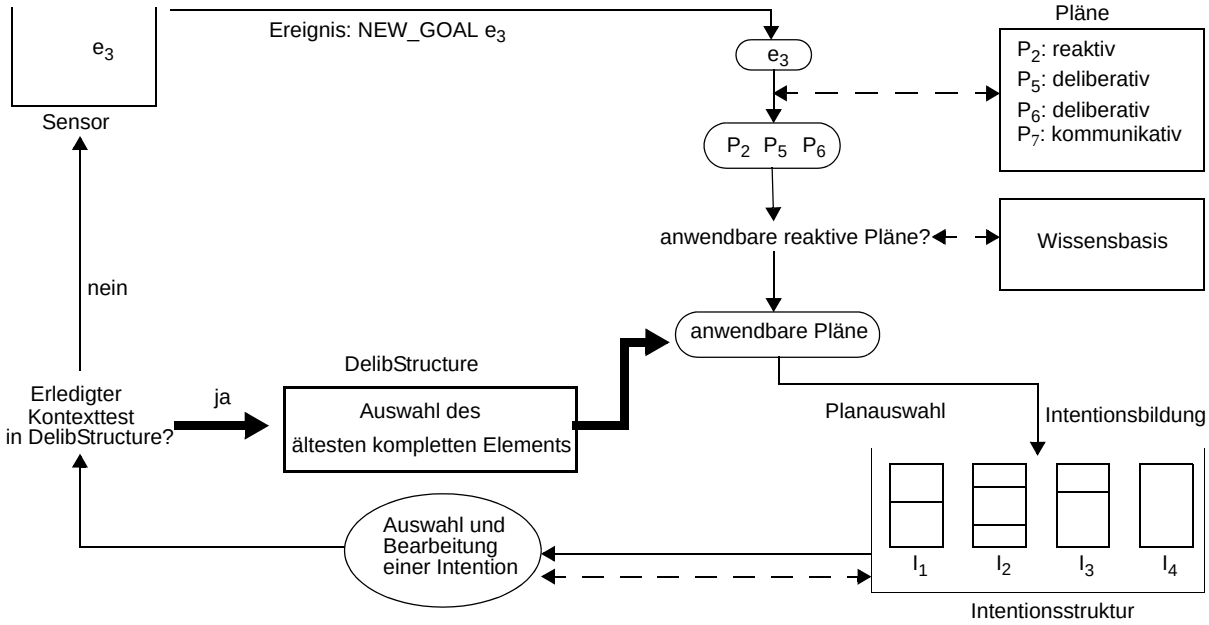
Unterziele in Vorbedingungen. In den Vorbedingungen deliberativer Pläne können Unterziele angegeben werden. Diese Unterziele werden als Ereignisse vom Typ PRECONDITION_SUBGOAL an den Sensor geschickt, dort später ausgewählt und weiterbearbeitet. Dabei können die folgenden Fälle auftreten:

1. Zu dem Unterziel gibt es anwendbare reaktive Pläne. In diesem Fall wird einer dieser Pläne ausgewählt, damit eine Intention in der Intentionsstruktur gebildet und das Ergebnis der Abarbeitung (Erfolg oder Misserfolg) an den erzeugenden Plan *in der DelibStructure* zurückgemeldet. Bei Misserfolg wird der Plan verworfen, bei Erfolg werden weitere Bedingungen abgearbeitet.
2. Es gibt keine anwendbaren reaktiven Pläne für das Unterziel, aber relevante deliberative Pläne. Dann wird das Unterziel zusammen mit den relevanten deliberativen Plänen *als neues Element* in die DelibStructure eingefügt und dort weiterbehandelt. Der erzeugende Plan in der DelibStructure muss solange warten, bis er die Meldung bekommt, ob das Unterziel erreicht werden konnte oder nicht.

3. Es gibt weder reaktive noch deliberative Pläne für das Unterziel. Dann wird versucht, das Unterziel in der CommStructure weiterzubearbeiten (vgl. Abschnitt 6.4.2.).
4. Es gibt keine anwendbaren Pläne für das Unterziel. In diesem Fall muss der erzeugende Plan in der DelibStructure verworfen werden.

Nachdem für ein Ziel alle relevanten deliberativen Pläne getestet worden sind, wird dem Interpreter signalisiert, dass nun (mindestens) ein Element in der DelibStructure komplett bearbeitet ist. Kurz bevor im Interpreterzyklus nach neuen Ereignissen im Sensor geschaut wird, erfolgt nun zusätzlich eine Abfrage nach kompletten Elementen in der DelibStructure (vgl. Abbildung 25).

Abbildung 25 Intentionsbildung mit deliberativen Plänen



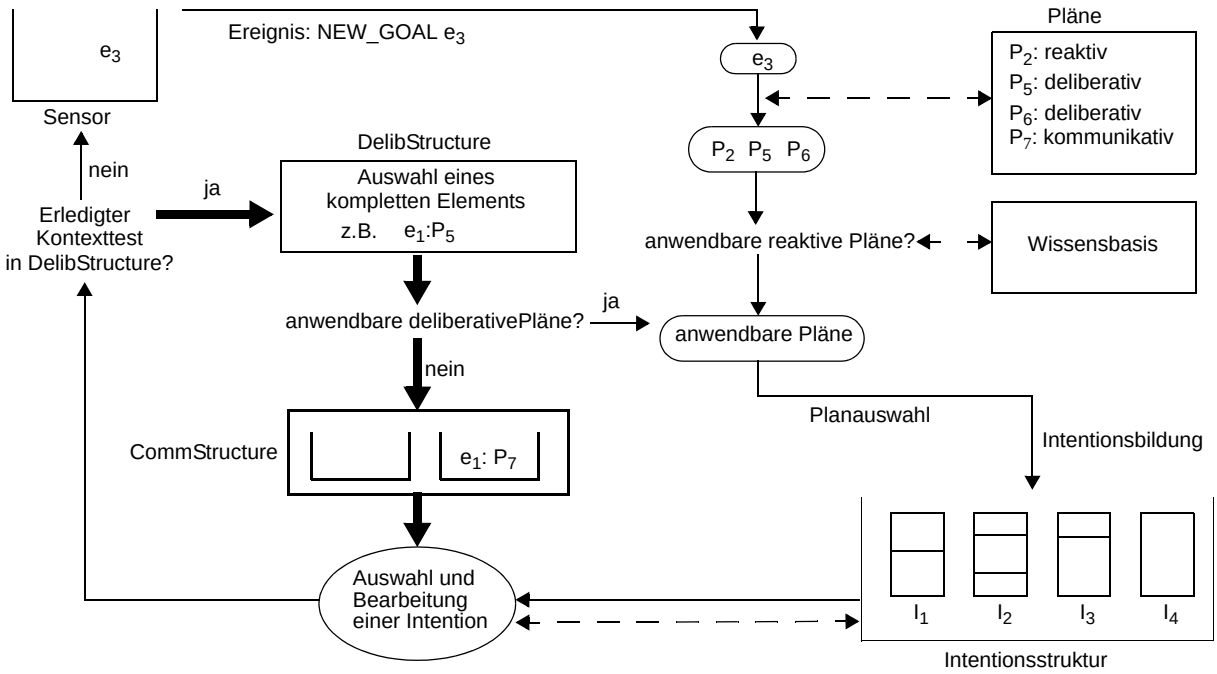
Es kann sein, dass inzwischen mehrere Elemente komplett sind. Dann soll das „älteste“ komplette Element aus der DelibStructure gewählt werden, d.h. das Element, das ursprünglich als erstes von diesen Elementen in die DelibStructure eingefügt wurde. Diese Auswahl eines Plans aus der DelibStructure hängt *nicht* mit der Auswahlfunktion OLDEST_FIRST für anwendbare Pläne zusammen. Anschließend wird wie gehabt einer der anwendbaren Pläne ausgewählt und damit eine Intention gebildet.

6.4.2. Kontexttest für kommunikative Pläne

Wenn für ein Ziel weder reaktive noch deliberative anwendbare Pläne gefunden werden, muss noch nach anwendbaren kommunikativen Plänen gesucht werden. Ausgelöst wird diese Suche dadurch, dass in der DelibStructure ein Element komplett bearbeitet ist, aber kein anwendbarer deliberativer Plan gefunden wurde. Dann wird das entspre-

chende Ziel einfach an eine weitere Instanz, die **CommStructure**, weitergegeben, in der es analog zur DelibStructure behandelt wird. Der Interpreterzyklus muss also nochmals erweitert werden, wie es in der Abbildung 26 dargestellt wird.

Abbildung 26 CommStructure

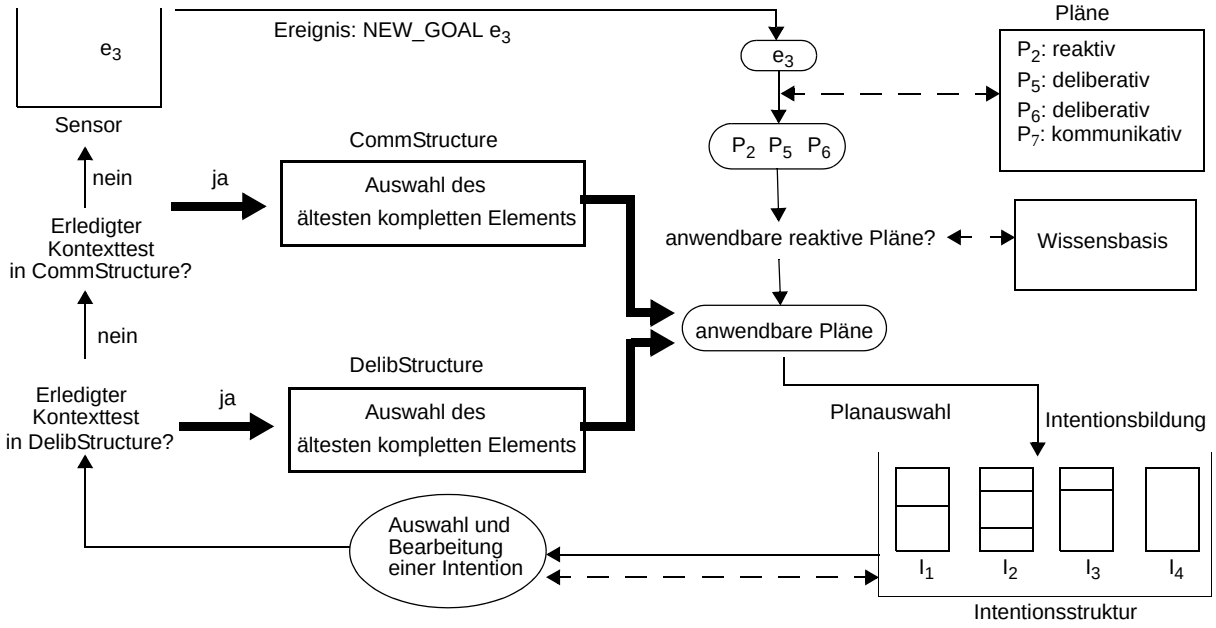


Als Beispiel soll das Ziel e_1 dienen, das nach erfolglosem Testen von Vorbedingungen deliberativer Pläne nun in die CommStructure mit dem relevanten kommunikativen Plan P_7 eingefügt wird. Danach wird wie gehabt eine Intention aus der Intentionsstruktur ausgewählt und ein Abarbeitungsschritt ausgeführt. Die Überprüfung des kommunikativen Plans P_7 erfolgt losgelöst vom Interpreterzyklus, sodass dieser nicht auf Antwortnachrichten bei kommunikativen Plänen warten muss.

Die Abfrage nach kompletten Elementen in der CommStructure erfolgt direkt nach der Abfrage zu kompletten Elementen in der DelibStructure. Zur besseren Übersicht wird dies in einer weiteren Grafik auf der nächsten Seite wiedergegeben (Abbildung 27).

Wenn für ein Ziel selbst nach dem Kontexttest für kommunikative Pläne kein anwendbarer Plan gefunden wurde, kann das Ziel nicht verfolgt werden und es muss eine Meldung nach außen abgegeben werden. Das Ziel wird verworfen, d.h. es wird keine Intention mit diesem Ziel gebildet und in der Intentionsstruktur abgelegt. Der Interpreterzyklus läuft an der Stelle weiter, wo die Auswahl und Bearbeitung einer vorhandenen Intention erfolgt.

Abbildung 27 Auswahl aus Delib- und CommStructure



6.4.3. Erweiterung des abstrakten Interpreters

Die in den beiden vorangehenden Abschnitten erarbeiteten Erweiterungen werden nun formal in den bestehenden abstrakten Interpreter eingefügt (vgl. Abbildung 28). Dabei bezeichnet **DelibStruct** die Menge der zu bearbeitenden Ziele mit ihren zugehörigen deliberativen Plänen und **DelibComplete** die Menge der vollständig bearbeiteten Elemente von DelibStruct. Entsprechendes gilt für die Mengen **CommStruct** und **CommComplete** bei Zielen mit kommunikativen Plänen. Die Funktion oldest() wählt das älteste vollständig bearbeitete Element aus der Menge DelibComplete bzw. CommComplete aus (z.B. anhand eines Zeitstempels) und liefert ein Tupel $\langle \varepsilon, A_\varepsilon \rangle$ mit einem Ereignis ε und einer Menge A_ε von anwendbaren Plänen.

Wenn keine vollständigen spekulativen Berechnungen vorhanden sind, wird wie bisher ein Ereignis aus dem Sensor ausgewählt und bearbeitet. Allerdings muss der abstrakte Interpreter bei der Ereignisbehandlung noch um eine Abfrage ergänzt werden: Wenn es keine anwendbaren reaktiven Pläne für ein Ziel gibt, muss es in die DelibStructure eingefügt werden. Damit ist der ursprüngliche – verborgene – rekursive Aufruf des AgentGHC-Interpreters für spekulative Berechnungen durch die Behandlung in einer weiteren Komponente ersetzt worden.

Abbildung 28 neuer Interpreter

```
While Agent is alive do
  if DelibComplete ≠ ∅ then
    (ε, A_ε) = oldest (DelibComplete)                    // Tupel: (Ereignis, Menge anwendbarer Pläne)
    DelibComplete = DelibComplete \ { (ε, A_ε) }
    if A_ε ≠ ∅ then
      P_ε = S_p(A_ε)
      if t == (NEW_GOAL or PRECONDITION_GOAL) then
        I = I ∪ [P_ε φ]                                  // neue Intention
      else
        push(P_εφ,Q)                                     // auf existierende Intention packen
      fi
    else                                                 // keine anwendbaren deliberativen Pläne
      R_ε = {pθ | p ∈ Strat ∧ p ist kommunikativ ∧ θ ist relevanter Unifikator bzgl. ε}
      CommStruct = CommStruct ∪ { (ε, R_ε) }             // Weiterbearbeitung in CommStructure
    fi
  elseif CommComplete ≠ ∅ then
    ε = (<e,Q, t>, A_ε) = oldest (CommComplete)          // Tupel: (Ereignis, Menge anwendbarer Pläne)
    CommComplete = CommComplete \ { (ε, A_ε) }
    if A_ε ≠ ∅ then do
      P_ε = S_p(A_ε)
      If t == (NEW_GOAL or PRECONDITION_GOAL) then
        I = I ∪ [P_ε φ]                                  // neue Intention
      else
        push(P_εφ,Q)                                     // auf existierende Intention packen
      fi
    else                                                 // keine anwendbaren Pläne, Ziel verwerfen
      // Error: no applicable plans at all
    fi
  elseif Erg  ≠ ∅ then                                   // schaue nach neuen Ereignissen
    ε = <e,Q, t> = S_ε(Erg)
    Erg = Erg \ {ε}
    case t == NEW_FACT:      Bel = Bel ∪ {e}             // Wissensbasis ändern
    case t == NEW_PLAN:      Strat = Strat ∪ {e}         // Planbibliothek ergänzen
    case t == REPLY:         setReply (Int, e)           // Intention die Antwort zukommen lassen
    case t == SUSPEND:       suspend (Int, e)            // Intention suspendieren
    case t == RESUME:        resume (Int, e)             // Intention reaktivieren
    case t == (NEW_GOAL or PRECONDITION_GOAL or SUBGOAL)
      R_ε = {pθ | p ∈ Strat ∧ p ist reaktiv ∧ θ ist relevanter Unifikator bzgl. ε}
      A_ε = {pφ | φ= θδ ∧ pθ ∈R_ε ∧ (θδ) ist anwendbarer Unifikator bzgl. ε }
      if A_ε == ∅ then
        R_ε = {pθ | p ∈ Strat ∧ p ist deliberativ ∧ θ ist relevanter Unifikator bzgl. ε}
        DelibStruct = DelibStruct ∪ { (ε, R_ε) }         // Weiterbearbeitung in DelibStructure
      else
        P_ε = S_p(A_ε)
        if t == (NEW_GOAL or PRECONDITION_GOAL) then
          I = I ∪ [P_ε φ]                                // neue Intention
        else
          push(P_εφ,Q)                                   // auf existierende Intention packen
    fi
  fi
  // es folgt die Intentionsauswahl und Ausführung wie im Originalinterpreter (vgl. Anhang A)
```

6.5. Agentenspezifikation

Nachdem in den vorangehenden Abschnitten die einzelnen Bestandteile von CASA behandelt worden sind, kann nun der Aufbau einer kompletten CASA-Agentenspezifikation angegeben werden. Jede Spezifikation besteht aus 4 Teilen:

- Angabe der Auswahlfunktionen
- Angabe der Initialziele
- Angabe der Initialfakten
- Angabe der Pläne

Am Beispiel des Einlagerungsagenten soll diese Strukturierung deutlich werden:

```
// Example: InRobot.spec
//=========================================
FUNCTIONS:
        EVENTSELECT:        Event_OldestFirst;
        PLANSELECT:         Plan_HighestPriority;
        INTENTIONSELECT:    Intention_HighestPriority;
//=========================================
GOALS:
        ACHIEVE busy : 1.0;
//=========================================
FACTS:
        FACT myId "InRobot-1" ;
        FACT myPos 0 0;
        ...
//=========================================
PLANS:

{
NAME:   "Top level goal: busy";
GOAL:   ACHIEVE busy;
TYPE:   REACTIVE;
PRIORITY: 1.0;
BODY:   ASSIGN $round 0;
        GETFACT myId $id;
        GETFACT orderAgent $orderAgent;
        INFORM $orderAgent "NEW_FACT" "robotType" $id "sedan";

        WHILE TEST (> 1 0) {                        // infinite loop
            EXECUTE print "\n" $id ":  round " $round "\n";
            ASSIGN $round (+ $round 1);
            ...
        }
}

// more plans are following here ...
```

KAPITEL 7 Die Implementierung von CASA

Neben der reinen Implementierung von Sprachelementen, die der CASA-Spezifikation entsprechen, sind weitere Programme notwendig, um ein vollständiges Programmpaket zur Generierung von CASA-Agenten zu erhalten. Daher wird in diesem Kapitel zunächst beschrieben, welche Anforderungen das zu implementierende Programmpaket erfüllen soll.

Es gibt sicherlich eine ganze Reihe von Programmiersprachen, mit denen man die CASA-Spezifikation implementieren kann. Die objektorientierte Programmiersprache Java (Version 1.1.5) ist jedoch aus verschiedenen Gründen besonders für die Implementierung von CASA geeignet, nicht zuletzt deswegen, weil auch schon existierende Klassen der Agentensprache JAM für CASA-Sprachelemente wiederverwendet werden können.

Ich habe mehrere Programmmodule erstellt, die dem logischen Aufbau von CASA-Agenten möglichst weit entsprechen sollen. Es ist nicht möglich, alle Aspekte der Implementierung hier aufzuführen, sodass ich mich in diesem Kapitel auf eine Auswahl der interessantesten Implementierungsarbeiten beschränke. Insbesondere beschreibe ich auch Komponenten, die in der CASA-Spezifikation nicht aufgetreten sind, aber für die Implementierung eine wichtige Rolle spielen, u.a. einen Parser für CASA-Spezifikationen.

Den Abschluss dieses Kapitels bildet ein Abschnitt über die Anbindung von CASA-Agenten an das MECCA-System, mit dem das Anwendungsbeispiel des Farbsortierspeichers realisiert worden ist.

7.1. Anforderungen an die Implementierung

Im Vordergrund der Implementierung steht die Umsetzung der Mikrosicht von CASA-Agenten, wie sie in Kapitel 6 vorgestellt worden ist. Es soll aber auch eine Schnittstelle für die Anbindung an ein existierendes Framework implementiert werden, die es ermöglicht, mit CASA-Agenten Multi-Agentensysteme wie z.B. den Farbsortierspeicher zu bilden.

Zur Implementierung der Mikrosicht ist eine Aufteilung in mehrere Programmmodule sinnvoll. Ein übergeordnetes Hauptmodul soll durch die folgenden drei Schritte die Generierung eines CASA-Agenten durchführen. Im ersten Schritt werden Optionen eingelesen, die angeben, welche Informationen zur Laufzeit des Agenten ausgeben werden sollen (Monitoring). Es soll möglich sein, Informationen zu allen Interpreterkomponenten ausgeben zu lassen. Im zweiten Schritt wird ein Parser zum Einlesen einer CASA-Spezifikation aufgerufen. Wenn kein Fehler beim Parsen auftritt, wird als dritter Schritt in die eigentliche Ausführung des Agenten übergegangen.

Der Parser soll CASA-Spezifikationen auf syntaktische Korrektheit überprüfen und Spezifikationsfehler melden. Bei syntaktisch korrekten Spezifikationen soll nach dem Parsen direkt ein neuer CASA-Agent generiert werden: Die drei Auswahlfunktionen werden festgelegt, die Initialfakten werden in der Wissensbasis gespeichert, die spezifizierten Pläne in der Planbibliothek abgelegt und die Initialziele als NEW_GOAL-Ereignisse an den Sensor geschickt.

Für jede Interpreterkomponente soll ein eigenes Programmmodul implementiert werden, d.h. jeweils ein Modul für den Sensor, die Wissensbasis, die Planbibliothek, den Kontexttest, die Intentionsstruktur und die Zyklussteuerung, im Folgenden auch Interpreter-Manager genannt. Diese Unterteilung soll dazu dienen, die Komponenten des Interpreters bei Bedarf gegen erweiterte oder verbesserte Versionen ohne großen Aufwand austauschen zu können.

Der Interpreter-Manager soll analog zum Zyklus des abstrakten CASA-Interpreters implementiert werden und übernimmt damit die Kontrolle über die übrigen Programmmodule. Der Sensor muss Ereignisse der vordefinierten Typen aufnehmen, verwalten und an die Zyklussteuerung entsprechend der Auswahlfunktion für Ereignisse weitergeben können. Das Programmmodul für die Wissensbasis soll die Fakten des Agenten verwalten und die Aktionen zum Einfügen, Verändern und Löschen von Fakten behandeln. Die Auswirkungen dieser Aktionen auf die Wissensbasis sind in Abschnitt 6.3.3. festgelegt worden. Die Planbibliothek soll Pläne entsprechend der Ordnerstruktur aus Abschnitt 6.3.4. verwalten und diese Ordner anderen Modulen zur Verfügung stellen. Das Programmodul für den Kontexttest erstellt Listen anwendbarer reaktiver Pläne und ist für die Durchführung spekulativer Berechnungen bei tiefen Zielen zuständig. Die Intentionsstruktur soll die parallelen Intentionsstapel des Agenten verwalten, die sich aus Zielen und instanziierten Plänen mit lokalen Variablenlisten zusammensetzen. Der Interpreter-Manager soll auf Top-Level-Intentionen bzw. ihre Unterintentionen zugreifen können und führt pro Zyklus einen Abarbeitungsschritt einer (Unter-)Intention aus. Bei der Ausführung von Aktionen ist insbesondere darauf zu achten, dass die Generierung paralleler Unterziele so verläuft, wie es in Abschnitt 6.3.6. beschrieben worden ist.

7.2. Java™ als Programmiersprache für CASA

Generell haben klassenbasierte, objektorientierte Programmiersprachen wie C++ oder Java den Vorteil, dass durch ihre Vererbungskonzepte Typspezialisierungen, Schnittstellendefinitionen und Wiederverwendungen von Klassen möglich sind. Ich möchte jedoch nicht genauer auf die einzelnen Konzepte objektorientierter Programmiersprachen eingehen und die verschiedenen infrage kommenden Sprachen gegenüberstellen, denn letztendlich sprechen ganz pragmatische Gründe für Java zur Implementierung von CASA.

In Java gibt es umfangreiche Klassenbibliotheken, die gerade für die Realisierung von CASA eine große Hilfe darstellen. Zum Beispiel gibt es die Klasse Thread (genauer: Java.lang.Thread) für die Erstellung und Verwaltung von Leichtgewichtsprozessen, die über einen gemeinsamen Adressraum verfügen. Implementierungen nebenläufig arbeitender Prozesse sind mit dieser Klasse einfacher zu realisieren als z.B. mit C oder C++. Im Hinblick auf spätere Erweiterungen von CASA hält Java viele weitere nützliche Klassenbibliotheken bereit, z.B. das Abstract Windowing Toolkit (Java.awt) für die Anbindung an eine grafische Oberfläche oder das Serializable-Interface für die Befähigung von Objekten, zu anderen Rechnerknoten zu migrieren. Auf diese Weise können CASA-Agenten auch zu *mobilen Agenten* werden.

Java-Programme werden mit einem Compiler in eine Zwischensprache – den sogenannten Java-Bytecode – übersetzt. Zur Ausführung eines Java-Programms wird dieser plattformunabhängige Bytecode durch eine abstrakte Maschine – die sogenannte Java Virtual Machine (JVM) – interpretiert. Da es JVM‘s für unterschiedliche Computer- und Betriebssysteme gibt, ist die Portierung von Java-Programmen in Form von Bytecode ohne Probleme möglich. Leider ist die interpretierte Programmausführung langsamer als die Ausführung von Maschinencode, aber für rechenzeitintensive Anwendungen gibt es bei vielen JVM‘s auch die Möglichkeit, zur Laufzeit mittels ”just-in-time”-Compilern Bytecode in maschinenspezifischen Code zu wandeln. Der so generierte Code macht die Ausführung eines Java-Programms fast so schnell wie entsprechende Programme in C oder C++ (vgl. [Flanagan 97], S. 8).

Für die Implementierung von CASA in Java sprechen aber auch praktische Gründe: Wie schon erwähnt baut die Syntax von CASA-Sprachelementen auf der Agentensprache JAM auf. Von JAM gibt es eine über das WWW verfügbare Implementierung in Java [Huber 98], von der ich die Basisklassen übernehmen durfte und für CASA angepasst habe. Der JAM-Parser ist mit einem Werkzeug erstellt worden, das eine Grammatikspezifikation einliest und daraus ein Java-Programm erzeugt, das als Parser für Ableitungen der Grammatik eingesetzt werden kann [JavaCC]. Auch für CASA kann mithilfe von JavaCC ein Java-Programm generiert werden, das CASA-Spezifikationen auf syntaktische Korrektheit überprüfen kann und dabei gleichzeitig die Interpreterkomponenten initialisiert.

Das Agenten-Framework MECCA mit seiner Kommunikationssprache FIPA ACL ist ebenfalls in Java implementiert, sodass auch die Anbindung von CASA-Agenten an dieses System ohne großen Aufwand möglich ist; es muss nur eine Klasse für die Schnittstelle zwischen CASA-Agenten und dem MECCA-System ergänzt werden.

Ich habe für die Implementierung die Java-Version 1.1.5 benutzt, mit der auch die übernommenen Klassen aus JAM und das MECCA-System übersetzt worden sind. Inzwischen ist jedoch auch schon Java-Version 1.2 mit einer Reihe von Erweiterungen verfügbar [Java 1.2].

7.3. Strukturierung in Pakete und Klassen

Für jede Komponente des abstrakten CASA-Interpreters habe ich ein eigenes Paket (package) angelegt. Für die meisten CASA-Sprachelemente konnte ich bereits existierende Klassen aus JAM wiederverwenden. JAM selbst besitzt keine gegliederte Paketstruktur, sondern fasst alle seine Klassen unter demselben Paketnamen zusammen, und daher habe ich zur besseren Übersicht die von JAM übernommenen Klassen ebenfalls in mehreren Paketen abgelegt. Ich unterscheide bei den Klassen der CASA-Implementierung zwischen den folgenden drei Kategorien:

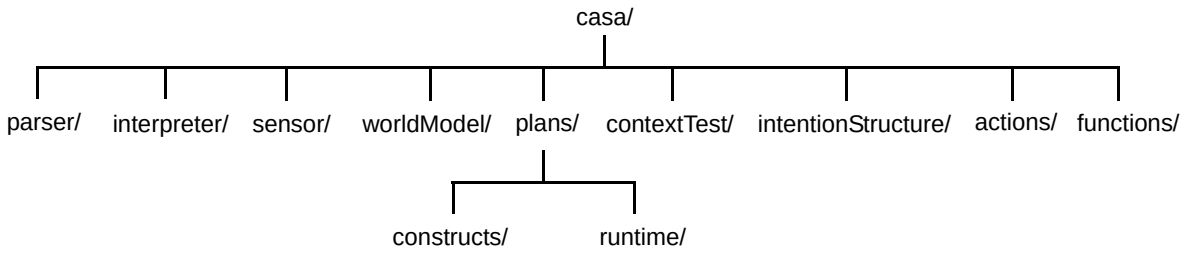
1. Nahezu unverändert wiederverwendete Klassen aus JAM.
2. Klassen aus JAM, die ich für CASA anpassen musste.
3. Von mir komplett neu implementierte Klassen.

In der folgenden Tabelle sind die Pakete mit ihren zugehörigen Klassen aufgelistet, wobei die Klassen der Kategorien 2 und 3 kursiv gedruckt sind. Auf diese wird in den nächsten Abschnitten noch genauer eingegangen, während die lediglich übernommenen Klassen nicht weiter behandelt werden; über ihre Implementierung erhält man in [Huber 98] und [Huber 99] weitere Informationen.

Paket	zugehörige Klassen
casa	*Haupt-Agentenklasse CasaAgent,* *Schnittstellenklasse AgentInterface*
casa.parser	*Parser für CASA-Spezifikationen,* Klassen für die Nichtterminale der kontextfreien Grammatik von CASA, weitere von JavaCC generierte Hilfsklassen
casa.interpreter	*Interpreter-Manager,* *Klasse für die drei Auswahlfunktionen im Interpreterzyklus*
casa.sensor	*Sensorkomponente,* *Klassen für CASA-Ereignisse*
casa.worldModel	Klassen für die Wissensbasis, Klassen für Variablenbindungen
casa.plans	*Klassen für den Aufbau der Planbibliothek und der Pläne*
casa.plans.constructs	Klassen für die Kontroll- und Ablaufstrukturen in Plänen
casa.plans.runtime	Klassen zur Verwaltung der Abarbeitung von instanziierten Plänen
casa.contextTest	*Klassen für die Überprüfung auf Anwendbarkeit von reaktiven Plänen,* *Klassen für spekulative Berechnungen (DelibStructure, CommStructure)*
casa.intentionStructure	*Intentionsstruktur zur Verwaltung paralleler Intentionsstapel*
casa.actions	Klassen für die in CASA vordefinierten Aktionen
casa.functions	Klassen für vordefinierte und applikationsabhängige Funktionen (Ausführung dieser Funktionen durch die EXECUTE-Aktion)

Verzeichnisstruktur und Installation von CASA. Jedes Paket der CASA-Implementierung ist in einem eigenen Verzeichnis abgelegt. So wird analog zur Paketstrukturierung ein Verzeichnisbaum gebildet, dessen Wurzel das Verzeichnis „casa/“ bildet (vgl. Abbildung 29). Vorausgesetzt, dass Java Version 1.1.5 oder höher bereits auf dem Zielcomputer vorhanden ist, muss zur Installation von CASA lediglich der Pfad zum Wurzelverzeichnis „casa/“ in der CLASSPATH-Umgebungsvariablen von Java ergänzt werden.

Abbildung 29 Verzeichnisstruktur von CASA



Die Generierung eines CASA-Agenten. Ein CASA-Agent wird durch den Aufruf des Java-Interpreters mit der Klasse CasaAgent, einer Spezifikationsdatei sowie einigen Optionen generiert. Dazu gibt man ein Kommando der folgenden Form an:

```
java CasaAgent (option)* <casa specification file>
```

Die Optionen sind in erster Linie zur Ausgabe von Laufzeitinformationen beim Testen von Agenten gedacht und werden in der folgenden Tabelle genauer beschrieben. Man sollte aber daran denken, dass der Einsatz dieser Optionen durch den zyklischen Ablauf im Interpreter zu einer großen Anzahl von Ausgaben auf dem Bildschirm führen kann.

Option	Beschreibung
-i	Informationen über neu eingefügte, komplett bearbeitete und fehlgeschlagene Intentionen in der Intentionsstruktur werden ausgegeben.
-p	Die Initialpläne werden direkt nach dem erfolgreichen Einparsen der Agentenspezifikation ausgegeben. Später werden nur noch Änderungen in der Planbibliothek gemeldet.
-w	Die Ausgangsfakten werden nach dem erfolgreichen Einparsen des Agenten ausgegeben. Während der Laufzeit werden alle Änderungen in der Wissensbasis dokumentiert.
-g	Alle vom Agenten neu generierten Ziele werden ausgegeben.
-a	Informationen über die Berechnung anwendbarer Pläne werden ausgegeben.
-m	Die Entscheidungen im Interpreter-Manager werden dokumentiert.
-s	Informationen über die Initialziele sowie die später erhaltenen und weitergeleiteten Ereignisse im Sensor werden ausgegeben.

Die Agentenspezifikation wird aus der Datei <casa specification file> eingelesen. Es muss darauf geachtet werden, dass diese Datei von dem aktuellen Verzeichnis aus gelesen werden kann; unter Umständen muss man den Verzeichnispfad vor der Datei angeben. Das Kommando zur Erzeugung eines einzelnen Einlagerungsagenten kann zum Beispiel so aussehen:

```
java CasaAgent -a -i -w  colorSorting/inRobot.spec
```

Nach der Instanziierung eines Objektes der Klasse CasaAgent werden zunächst die Optionen eingelesen, wobei unbekannte Optionen unterdrückt werden. Dann wird aus der Klasse CasaAgent heraus der Parser aufgerufen, der die angegebene Spezifikationsdatei einliest und dabei schon versucht, die Interpreterkomponenten zu initialisieren. Wenn beim Parsen kein Fehler gefunden worden ist, wird der Interpreterzyklus vom CasaAgent-Objekt aus gestartet. Die Generierung eines Einlagerungsagenten mit dem oben angegebenen Kommando wird auf dem Bildschirm zum Beispiel wie folgt dokumentiert (APL steht dabei für ”applicable plan list”):

Abbildung 30 Protokoll der Generierung eines Einlagerungsagenten

```
-----------------------------------
Constructor for CasaAgents
-----------------------------------

Monitoring the APL generation.
Monitoring the IntentionStructure.
Monitoring the World Model.

Parsing file and building agent components.
...
Program parsed successfully.

Starting to run agent colorSorting/inRobot.spec.

Interpreter: Initial World Model:
Agent's World Model is now:
0 : myId InRobot-1
1 : myPos 0 0
2 : myState waiting
3 : kType sedan
4 : kType van
5 : brokerAgent SortlineBroker
6 : sortLine 1 20 10
7 : sortLine 2 20 20
8 : lineAgent 1 Sortline-1
9 : lineAgent 2 Sortline-2
10 : orderAgent OrderAgent
11 : depotAgent DepotAgent
12 : depotPos 10 10

APL: Checking reactive plan: Top level goal: "busy"
APL: Adding reactive plan to result list: Top level goal: "busy"

IntentionStructure: Adding new top-level-goal "busy" (ID 1, utility 2.0)
...
```

Mit einem auf diese Weise generierten Agent kann allerdings noch nicht kommuniziert werden; es fehlt noch eine Schnittstelle für den Agenten, die in Abschnitt 7.5.2. genauer beschrieben wird.

7.4. Ausgewählte Teile der Implementierung

In diesem Abschnitt werden nur einige Aspekte der Implementierung aufgezeigt, die eine Vorstellung von dem Aufbau und den Zusammenhängen der verschiedenen Programmmodule vermitteln sollen.

7.4.1. Ein Parser für CASA-Spezifikationen

Da bisher noch nicht genauer auf das Einlesen von Spezifikationen eingegangen worden ist, gehe ich im Folgenden erst auf die wichtigsten Anforderungen an einen Parser für CASA-Spezifikationen ein, bevor ich die wichtigsten Aspekte der Implementierung beschreibe.

Anforderungen an den Parser±. Um einen gewissen Grad an Freiheit zur Gestaltung von Spezifikationen zu erhalten, sollen Kommentare erlaubt sein, wie sie in anderen Programmiersprachen auch möglich sind. Der Parser soll größere Abstände zwischen Sprachelementen, sogenannte ”white spaces”, ignorieren. Darunter fallen außer Leerzeichen zum Beispiel auch Tabulatorzeichen (”\t”) und Zeilenumbrüche (”\n”). Allerdings kommt es oft vor, dass diese Sonderzeichen in einer Zeichenkette angegeben werden. In diesem Fall darf der Parser die Zeichen nicht ignorieren, sondern muss sie als Bestandteil einer Zeichenkette behandeln.

Bei Syntaxfehlern in einer Spezifikation soll der Parser nicht einfach seine Arbeit abbrechen, sondern einen nützlichen Hinweis auf die Fehlerstelle ausgeben, sodass der Entwickler den Fehler beheben kann. Gerade diese Anforderung verursacht einen sehr hohen Implementierungsaufwand, denn es müssen alle potentiellen Syntaxfehler durch Abfragen abgefangen und entsprechende Fehlermeldungen ausgegeben werden.

Ein Werkzeug zur Generierung von Parsern. Es gibt ein Werkzeug, das die Entwicklung eines Parsers unterstützt; der sogenannte Java Compiler Compiler [JavaCC] liest eine Grammatikspezifikation und generiert daraus ein Java-Programm, das korrekte Ableitungen der Grammatik erkennt und bei Fehlern immer einen Hinweis auf die Fehlerstelle gibt. Dadurch, dass zusammen mit JavaCC bereits Beispiele für JavaCC-Grammatikspezifikationen geliefert werden, muss man sich nicht lange mit der Behandlung der oben genannten Sonderfälle aufhalten, sondern kann sich auf die Umsetzung der Grammatik im Parser konzentrieren. Eine Grammatikspezifikation für JavaCC besteht im Wesentlichen aus vier Teilen:

1. Im ersten Teil wird ein normales Java-Programm angegeben, das als Eingabeparameter eine Datei erhält und das Parsen in Gang setzt, indem es eine Methode aufruft, die das Startsymbol der Grammatik repräsentiert. In diesem Teil können auch weitere Java-Methoden, die beim Parsen generell nützlich sind, implementiert werden.
2. Im zweiten Teil wird die Behandlung von Sonderzeichen und Kommentaren geregelt. Für Kommentare habe ich die beiden üblichen Kennzeichnungen übernommen: Zwei Schrägstriche kündigen einen Kommentar im Rest der Zeile an und ein mehrzeiliger Kommentar steht zwischen den beiden Zeichenfolgen ”/*” und ”*/”.
3. Im dritten Abschnitt folgt die Definition der Terminale als sogenannte ”Tokens”.

4. Im vierten Teil werden Methoden implementiert, die jeweils eine Ableitungsregel repräsentieren. Jede dieser Methoden kann aus einer Reihe von Blöcken bestehen, wobei sich Blöcke für die eigentliche Behandlung der Ableitungsregeln und Blöcke mit reinem Java-Code abwechseln. Die Ableitungsregeln werden in einer speziellen Syntax angegeben, wie sie auch bei regulären Ausdrücken gebräuchlich ist. Bei der Ausführung einer Methode wird versucht, die Ableitungsregel anzuwenden, wobei auftauchende Terminale bzw. Tokens in lokalen Variablen gespeichert werden. In einem anschließenden Java-Block können die gerade eingelesenen Tokens dann benutzt werden, um zum Beispiel ein neues Java-Objekt für das Token zu erzeugen.

Dynamisches Laden der Auswahlfunktion. Es gibt eine Besonderheit beim Parsen der Auswahlfunktionen. Dort wird wie schon erwähnt jeweils der Name einer Java-Klasse angegeben, die das Interface der entsprechenden Auswahlfunktion implementiert. Beim Einlesen einer Spezifikation erhält der Parser also eine Zeichenkette, die lediglich einen Klassennamen angibt. Um von dieser Klasse jedoch ein Objekt erzeugen zu können, muss u.a. gewährleistet sein, dass diese Klasse überhaupt existiert und Objekte dieser Klasse instanziiert werden dürfen. Es kann also erst zur Laufzeit versucht werden die angegebene Klasse aufzuspüren, d.h. sie dynamisch zu laden. Hierfür habe ich eine Methode implementiert, die nach erfolgreicher Suche nach der angegebenen Klasse anschließend auch die Instanziierung eines Objekts übernimmt.

Initialisierung der Interpreterkomponenten. Schon während des Parsens werden die Interpreterkomponenten sukzessive initialisiert, z.B. werden die in der Spezifikation angegebenen Fakten direkt in eine Instanz der Klasse WorldModelTable eingefügt, die die Wissensbasis im CASA-Agenten repräsentiert. Wenn der Parser während des Einlesens der Spezifikationsdatei einen Fehler findet, bricht er die weitere Initialisierung ab und gibt als Fehlermeldung die betreffende Zeilennummer und den Fehlergrund an. Die Behandlung von reinen Syntaxfehlern wird von JavaCC bei der Generierung des Parsers automatisch ergänzt, sodass der Entwickler sich nicht um die Implementierung zur Abfrage solcher Fehler zu kümmern braucht. Allerdings ist es manchmal auch nötig, in den Java-Blöcken der Parser-Methoden explizit Ausnahmen (ParseExceptions) anzugeben, z.B. wenn eine kontextsensitive Überprüfung stattfinden muss und eine der geprüften Bedingungen nicht erfüllt ist.

Ein Beispiel hierfür ist die Überprüfung von Plantypen und ihren Vorbedingungen. Wenn z.B. ein ACHIEVE-Konstrukt als Teil einer Planvorbedingung eingelesen wird, kontrolliert der Parser den vorher schon festgelegten Typ des Plans. Wenn er vom Typ REACTIVE ist, wird eine Fehlermeldung ausgegeben, die explizit in der Implementierung angegeben werden muss.

Das entsprechende Codefragment wird in der folgenden Abbildung 31 angegeben. Es handelt sich um eine Parser-Methode, in der die möglichen Sprachelemente für Vorbedingungen getrennt durch senkrechte Striche aufgelistet werden. Je nach Sprachelement werden verschiedene weitere Parser-Methoden aufgerufen, z.B. expression() oder relation(), und nach erfolgreichem Einlesen dieser Bestandteile wird ein neues Objekt generiert. In diesem Fall wird eine abgeleitete Instanz der Klasse PlanConstruct als Ergebnis zurückgegeben. Dieses Objekt wird in der aufrufenden Methode in die Liste der Vorbedingungen des aktuellen Plans (currentPlan) aufgenommen.

Abbildung 31 Teil der Überprüfung auf korrekte Planspezifikation

```
PlanConstruct plan_simple_condition_element (Plan currentPlan) :
{   ...                                                    // Java code for local variables
    Action a; Token t; Expression e; Relation rel;
}
{       ...                                                // parser block (JavaCC code)
    |   t = <GETFACT>      rel = relation(currentPlan) ";"
        ...
    |   t = <ACHIEVE>      rel = relation(currentPlan) ":" e = expression(currentPlan) ";"
        {
                                        // Java-Code to check precondition and handle exception
            if (currentPlan.getType() == Plan.REACTIVE)
                throw new ParseException ("illegal precondition in reactive plan ...");

            a = new PreconditionAchieveGoalAction(rel.getName(), rel, e);
            return new PlanSimpleConstruct(a);
        }

    |   t = <REQUEST> ...
}
```

7.4.2. Das Paket casa.interpreter

Das Paket casa.interpreter enthält eine Klasse für den Interpreter-Manager, die den Interpreterzyklus wie in Abschnitt 6.4.3. beschrieben implementiert. Im Wesentlichen besteht diese Klasse aus Referenzen auf Objekte, die die Interpreterkomponenten repräsentieren, und einer run()-Methode mit einer Endlosschleife. Da sich der implementierte Interpreter genau an die Spezifikation hält, möchte ich nicht genauer auf die Realisierung dieser Klasse eingehen. In diesem Paket sind aber auch noch die in der folgenden Tabelle angegebenen drei Interface-Klassen für die Auswahlfunktionen enthalten. Die Funktionsarten RANDOM, HIGHEST_PRIORITY und OLDEST_FIRST sind insoweit implementiert, wie sie im Abschnitt 6.4.2. angegeben sind.

Interface-Klasse	Methoden	Implementierte Klassen
EventSelectFunction	selectEvent (Sensor s)	Event_Random Event_OldestFirst
PlanSelectFunction	selectPlan (DList list)	Plan_Random Plan_HighestPriority
IntentionSelectFunction	selectTopLevelIntention (IntentionStructure is) selectLeafIntention (DList leafIntentions)	Intention_Random Intention_HighestPriority Intention_OldestFirst

Wenn nun nachträglich noch eine weitere Funktionsart hinzugefügt werden soll, muss dafür eine neue Klasse im Paket casa.interpreter implementiert werden, die das entsprechende Interface erfüllt. Es nicht nötig, existierenden Code in den bereits implementierten Klassen zu ändern. Auch der Parser-Programmcode braucht nicht verändert zu werden, denn beim Parsen wird versucht, die angegebenen Klassen für die Auswahlfunktionen dynamisch zu laden.

7.4.3. Implementierung der Sensorkomponente

In diesem Paket sind die Sensorkomponente und die verschiedenen vordefinierten Ereignistypen in jeweils einer eigenen Klasse implementiert. Die Klasse Sensor verwaltet lediglich eine lineare Liste von wahrgenommenen Ereignissen und gibt sie auf Anforderung durch den Interpreter-Manager weiter. Die Ereignistypen, die im Abschnitt 6.3.1. spezifiziert worden sind, wurden in den folgenden Klassen implementiert:

Ereignistyp	Klasse	Beschreibung
NEW_FACT	NewFactEvent	neues Fakt
UPDATE_FACT	UpdateFactEvent	verändertes Fakt
NEW_PLAN	PlanEvent	neuer Plan
NEW_GOAL	GoalEvent	Initialziel der Spezifikation oder neues externes Ziel
REPLY	ReplyEvent	Antwortnachricht
SUBGOAL	SubgoalEvent	neues Unterziel einer Intention
PRECONDITION_GOAL	PreconditionEvent	spekulatives Ziel eines deliberativen Plans
SUSPEND	SuspendEvent	Suspendierung einer Intention
RESUME	ResumeEvent	Wiederaufnahme einer Intention

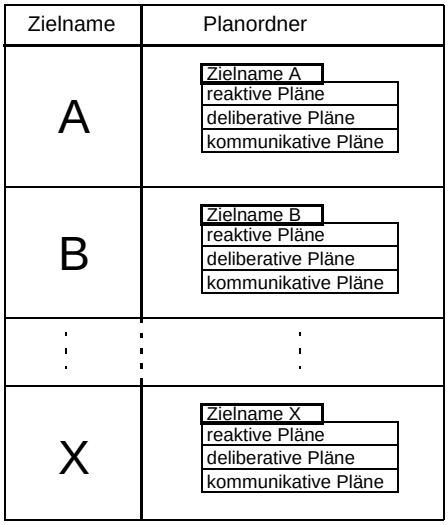
Diese Event-Klassen sind abgeleitet von der abstrakten Klasse CasaEvent. Ein Ereignis setzt sich zusammen aus einem Typen und dem eigentlichen Inhalt, der als Aktion beschrieben wird. Der Interpreter-Manager kann anhand des zugeordneten Ereignistyps feststellen, um welche Art von Aktion es sich bei dem betrachteten Ereignis handelt und führt daraufhin die nötigen Schritte für die Bearbeitung der Aktion aus. Die Objekte der Event-Klassen erhalten u.a. auch einen Zeitstempel, der ihre Ankunftszeit im Sensor belegt und bei den Auswahlfunktionen benutzt werden kann.

7.4.4. Implementierung der Planbibliothek

Die Planbibliothek wird in der Klasse PlanTable als Hashtabelle verwaltet, wobei jedes Tabellenelement aus einem Tupel (Zielname, Planordner) besteht. Der Zielname repräsentiert das Ziel, das alle Pläne im zugehörigen Planordner zu erreichen versprechen. Planordner sind in der Klasse PlanTableElement implementiert und haben je drei Listen für die verschiedenen Plantypen (vgl. Abschnitt 6.3.4.). Abbildung 32 zeigt den schematischen Aufbau der Planbibliothek.

Mithilfe der Hashtabelle und den Planordnern können die relevanten Pläne für ein neues (Unter-)Ziel gefunden werden, ohne dass jeder einzelne Plan in der Planbibliothek darauf geprüft werden muss, ob er zur Erreichung des gewünschten Ziels dient. Mit der Methode getRelevantPlans (String goalName) erhält man aus der Planbibliothek den gewünschten Planordner zu einem gegebenen Zielnamen.

Abbildung 32 Planbibliothek als Hashtabelle



Wenn zu einem Ziel kein relevanter Plan in der Planbibliothek gefunden werden kann, wird dies dem Interpreter-Manager gemeldet, der daraufhin eine Ausnahmebehandlung einleitet. Zur Zeit wird in diesem Fall eine Warnmeldung ausgegeben.

7.4.5. Umsetzung von parallelen Strukturen in CASA

Im Wesentlichen hält sich die Implementierung der Intentionsstruktur an die Spezifikation aus Abschnitt 6.4.6. Die verwalteten Intentionsstapel sind in der Weise parallel, als dass sie verzahnt abgearbeitet werden können. Die genaue Reihenfolge der Abarbeitung von Intentionen hängt allerdings von der individuell festzulegenden Auswahlfunktion ab, sodass ich an dieser Stelle nicht weiter auf diese Komponente eingehen werde. Stattdessen möchte ich im Folgenden auf den Teil der Implementierung eingehen, der sich auf die Abarbeitung von Planelementen und insbesondere von parallelen Planblöcken bezieht.

Abarbeitung von Planelementen. In jedem Interpreterzyklus wird eine auszuführende Intention aus der Intentionsstruktur ausgewählt. Diese Intention verwaltet u.a. eine Referenz auf das aktuell zu bearbeitende Element im Plankörper ihres zugeordneten Plans. Für dieses Element wird nun die execute()-Methode aufgerufen, in der die nötigen Schritte zur vollständigen Bearbeitung des Sprachelements implementiert sind. Die meisten Aktionen werden mit diesem Aufruf komplett abgearbeitet, z.B. die Aktionen zur Manipulation der Wissensbasis. Bei anderen Aktionen muss erst auf eine Rückmeldung gewartet werden, bevor zum nächsten Planelement übergegangen werden kann, z.B. bei Unterzielen oder bei Anfragenachrichten. Es kann auch vorkommen, dass eine Aktion fehlschlägt, z.B. wenn versucht wird, auf ein nicht existierendes Fakt in der Wissensbasis zuzugreifen. Nach ihrer Ausführung melden die execute()-Methoden daher zurück, ob das ausgeführte Sprachelement komplett bearbeitet, noch aktiv oder fehlgeschlagen ist. Die Intentionsstruktur reagiert dann entsprechend auf diese Rückmeldung: Bei kompletten Elementen wird der Bearbeitungszeiger auf das nächste

Planelement gesetzt, bei fehlgeschlagenen Aktionen wird die FAILURE-Sektion des Plans ausgeführt und danach die Intention abgebrochen.

Abarbeitung von parallelen Blöcken. Es soll auch möglich sein, die in einem PARALLEL-Konstrukt spezifizierten Blöcke nebenläufig abarbeiten zu lassen. Da auf einer Einprozessormaschine nebenläufige Aktionen lediglich verzahnt statt wirklich parallel abgearbeitet werden können, bietet sich hier der Einsatz von Leichtgewichtsprozessen (Threads) an, die unabhängig voneinander jeweils die Anweisungen eines ihnen zugeteilten Parallelblocks ausführen. Ohne weiteren Kontrollmechanismus würden die Threads aber sofort die kompletten Blöcke abarbeiten und sich nicht mehr an die Vorgabe halten, nur einen Abarbeitungsschritt pro Interpreterzyklus auszuführen. Daher werden die Threads immer nach der Durchführung eines Abarbeitungsschrittes angehalten, bis sie wieder von ihrer erzeugenden Intention zur Durchführung des nächsten Abarbeitungsschrittes gestartet werden. In welcher Reihenfolge diese Threads ausgeführt werden, ist dennoch nicht vorhersehbar.

In der Regel werden die Threads zu verschiedenen Zeiten ihren Block komplett bearbeitet haben. Die execute()-Methode für PARALLEL-Konstrukte überwacht daher den Status aller zugehörigen Threads und meldet erst dann die komplette Bearbeitung zurück, wenn alle Threads mit ihrer Arbeit fertig sind.

Für die Threads werden keine lokalen Variablen angelegt, sondern sie greifen auf die Variablen der erzeugenden Intention zu. Dies kann zu Schreibkonflikten führen, wenn zwei Parallelblöcke dieselbe Variable verändern wollen. Bisher ist für diesen Fall die folgende Lösung implementiert: Der Thread, der als erster auf eine Variable schreibend zugreifen möchte, markiert diese Variable mit seiner eindeutigen Kennung, sodass er bis zur Beendigung aller beteiligten Threads als einziger schreibend auf diese Variable zugreifen darf. Die anderen Threads können den aktuellen Variablenwert zwar immer auslesen, wenn sie aber auch auf die reservierte Variable schreibend zugreifen wollen, erfolgt eine Warnmeldung der folgenden Form:

```
Warning:  multiple threads are trying to write into variable $x
          Keeping old value for variable $x
```

Der Schreibzugriff auf die Variable wird also nicht ausgeführt und die Abarbeitung der Intention fortgesetzt.

Parallele Unterziele. Wenn innerhalb von Parallelblöcken Unterziele verfolgt werden sollen, kann dies zu der schon in Abschnitt 6.3.6. angesprochenen Situation führen, in der mehrere Unterziele im selben Interpreterzyklus als SUBGOAL-Ereignisse an den Sensor geschickt werden. Die Unterziele werden dann zwar nacheinander in unterschiedlichen Zyklen aus dem Sensor ausgewählt, aber später als Unterintentionen nebeneinander zu ihrer erzeugenden Intention gestapelt (Multikeller). Damit sie in der Intentionsstruktur korrekt eingefügt werden können, erhalten die Unterziele die eindeutige Identifikationsnummer ihrer erzeugenden Intention als Referenz.

Erst nach vollständiger Bearbeitung einer Unterintention werden die Variablenwerte der Parameter an die erzeugende Intention zurückgemeldet, denn alle Parameter eines Unterziels sind in CASA vom Typ call-by-value-and-result. Dann kann es wie oben beschrieben wieder zu Schreibkonflikten auf Variablen kommen, in diesem Fall bei den Variablen, die als Parameter des Unterziels angegeben werden. Auch hier erfolgt eine

Warnmeldung wie oben angegeben. Es gibt sicher Alternativen zu dieser Implementierung, von denen einige auch schon in Abschnitt 6.2.2. angesprochen worden sind. Sicher ist es für die Zukunft auch notwendig, weitere Parametertypen für Unterziele zu ermöglichen. Dazu sollten die Parametertypen aber erst im CASA-Modell genauer spezifiziert werden, damit sich die Implementierung nicht zu weit von der Spezifikation entfernt.

7.4.6. Spekulative Berechnungen beim Kontexttest

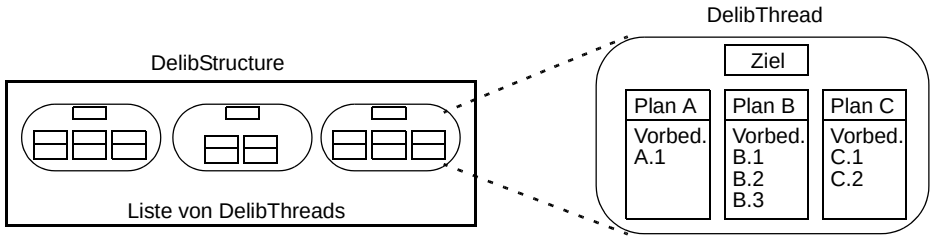
Für ein neues Ziel können alle relevanten Pläne leicht in dem entsprechenden Ordner der Planbibliothek gefunden werden (vgl. Abschnitt 7.4.4.). In dem Ordner können jedoch Pläne stehen, die zwar das Ziel zu erreichen versprechen, aber eine andere Parameteranzahl haben als das Ziel. Diese Pläne werden von vornherein als nicht anwendbar angesehen und bei der Implementierung nicht weiter für das Ziel betrachtet.

Es bleibt eine Menge von Plänen übrig, von denen zunächst die reaktiven Pläne auf ihre Anwendbarkeit geprüft werden. Es handelt sich dabei lediglich um Durchführungen von Tests und Abfragen der Wissensbasis. Da diese Operationen ohne komplexe Berechnungen ausführbar sind, können sie innerhalb des Interpreterzyklus auf die folgende Weise bearbeitet werden: Der Interpreter-Manager übergibt einer Instanz der Klasse APL (”applicable plan list”) die Liste reaktiver Pläne und wartet auf das Ergebnis der Überprüfung, d.h. auf eine Liste von anwendbaren reaktiven Plänen. Wenn diese Liste mindestens ein Element enthält, wird einer der anwendbaren Pläne mit der spezifizierten Planauswahlfunktion ausgewählt und zusammen mit dem Ziel eine Intention gebildet. Ansonsten wird die Überprüfung relevanter Pläne in der Komponente DelibStructure fortgesetzt.

Spekulative Berechnungen deliberativer Pläne. Jeder deliberative Plan kann in seiner Vorbedingung Unterziele angeben, die erst vom Agenten erreicht werden müssen, bevor der Plan als anwendbar gilt. Wird ein Unterziel in der Vorbedingung erreicht, kann die Überprüfung der Vorbedingungen fortgesetzt werden, ansonsten wird sie abgebrochen und der Plan als nicht anwendbar verworfen. Ist für einen deliberativen Plan P schon ein Unterziel U erfolgreich erreicht worden und stellt sich erst später bei weiteren Vorbedingungen heraus, dass P doch nicht anwendbar ist, können die durchgeführten Aktionen von U in der Regel nicht mehr rückgängig gemacht werden; zum Beispiel können in der Zwischenzeit aufgrund von U schon Entscheidungen in einer anderen Intention getroffen worden sein. Somit ist das Ziel U zwar erreicht worden, aber der Plankörper des auslösenden Plans wird gar nicht ausgeführt.

Die Implementierung der DelibStructure. Die DelibStructure ist als eigener Thread implementiert, sodass der Interpreter-Manager schon im normalen Zyklus weiterlaufen und z.B. Intentionen abarbeiten kann, während die DelibStructure die Vorbedingungen von deliberativen Plänen überprüft. In der folgenden Abbildung wird der interne Aufbau der DelibStructure näher erläutert.

Abbildung 33 Aufbau der DelibStructure



Jedesmal, wenn der Interpreter-Manager zu einem neuen Ziel keine anwendbaren reaktiven Pläne vorfindet, wird in der DelibStructure eine neue Instanz der Klasse DelibThread angelegt und in die Liste der DelibStructure eingefügt. Ein DelibThread speichert u.a. das zu erreichende Ziel und verwaltet die dazu relevanten deliberativen Pläne. Er entspricht einer Mini-Intentionsstruktur, in der anstatt Plankörpern die Planvorbedingungen abgearbeitet werden. Da erst *alle* Pläne des DelibThreads auf ihre Anwendbarkeit hin überprüft werden müssen, bevor die Auswahl eines der anwendbaren Pläne erfolgen kann, müssen sie im DelibThread gleichberechtigt untersucht werden. Die Planliste wird daher zyklisch durchlaufen und die Pläne werden nacheinander aufgefordert, einen weiteren Abarbeitungsschritt ihrer Vorbedingungen zu machen. Im obigen Beispiel werden im ersten Zyklus die Vorbedingungen A.1, B.1 und C.1 ausgeführt. Wenn als Vorbedingung ein Unterziel überprüft werden soll, müssen erst andere Threads (insbesondere der Interpreter-Manager) dieses Ziel behandeln, bevor der DelibThread das Resultat erhalten kann. Der DelibThread gibt daher nach jedem seiner Zyklen explizit mittels yield() die Kontrolle an einen anderen wartenden Thread ab.

Im Unterschied zur Intentionsstruktur werden in einem DelibThread aus verschiedenen Gründen keine Stapel gebildet. Zum Einen müssen Unterziele nicht automatisch auch in der DelibStructure auftauchen, zum Beispiel wenn es für das Unterziel einen anwendbaren reaktiven Plan gibt. Zum Anderen kann ein DelibThread auf diese Weise sehr komplex werden, zum Beispiel wenn in einer Vorbedingung parallele Unterziele angegeben werden, für die wieder eine Reihe von deliberativen Plänen untersucht werden müssen. Also wartet jeder Plan auf das Ergebnis der Durchführung seiner Unterziele, bevor die nächste Vorbedingung abgearbeitet wird.

Ein DelibThread ist komplett bearbeitet, wenn er die Vorbedingungen aller seiner Pläne überprüft und damit die anwendbaren deliberativen Pläne ermittelt hat. Er meldet dann der DelibStructure, dass er mit seiner Berechnung fertig ist. Der Interpreter-Manager fragt zyklisch bei der DelibStructure an, ob es komplett bearbeitete DelibThreads gibt. Wenn dies der Fall ist, wird der älteste komplette DelibThread vom Interpreter-Manager übernommen und weiterbearbeitet, wie es in der Spezifikation angegeben ist (vgl. Abschnitt 6.4.3.).

Die Implementierung der CommStructure. Die Implementierung der DelibStructure und der CommStructure ähneln sich sehr, daher gehe ich nur kurz auf die wesentlichen Unterschiede zur DelibStructure ein.

Neue Elemente können auf zwei Weisen in die CommStructure gelangen: Entweder direkt vom Interpreter-Manager aus, wenn weder reaktive noch relevante deliberative Pläne vorliegen, oder über die DelibStructure, wenn sich bei der Überprüfung der Pläne herausstellt, dass kein anwendbarer deliberativer Plan existiert. Die Elemente der CommStructure werden auf die gleiche Weise wie die Elemente DelibStructure verwaltet.

Beim Kontexttest für kommunikative Pläne ist es nötig, Anfragenachrichten anders kenntlich zu machen als Anfragen von Intentionen, damit die zugehörigen Antwortnachrichten später vom Interpreter-Manager in die richtige Komponente weitergeleitet werden können. Zur Zeit wird der eindeutigen Referenz, die jeder Nachricht mitgegeben wird, einfach ein „C“ vorangestellt, wenn es sich um eine Anfrage aus der CommStructure handelt.

7.5. Erweiterung zu einem Multi-Agentensystem

Obwohl die Implementierung der agenteninternen Sicht im Vordergrund der Arbeit steht, sollen auch die Grundlagen zur Kommunikation mit anderen CASA-Agenten realisiert werden; erst dadurch ist es möglich, mit CASA Multi-Agentensysteme wie den Farbsortierspeicher zu bilden. Der in AgentGHC vorgeschlagene Ansatz für Agentenfamilien mit der Oberklasse UrAgentGHC ist bisher nicht implementiert, aber die dort vorgeschlagenen Nachrichtentypen sind in die CASA-Spezifikation übernommen worden, sodass die Grundlagen für kooperierendes Verhalten in CASA-Agenten gegeben sind.

7.5.1. Nachrichtenaustausch zwischen CASA-Agenten

In diesem Abschnitt wird der Einsatz einer Schnittstellenkomponente erläutert, die CASA-Agenten bei der Generierung neuer Nachrichten sowie beim Übersetzen von empfangenen Nachrichten in Ereignisse unterstützt.

Wenn eine neue Nachricht generiert werden soll, übergibt der sendende CASA-Agent die dafür benötigten Daten der ihm zugeordneten Schnittstelle. Dort werden die Daten in FIPA ACL umwandelt und an den Empfängeragenten verschickt. Wie der Nachrichtentransport über das Netzwerk genau funktioniert, soll an dieser Stelle nicht näher betrachtet werden, denn dies ist nicht mehr Aufgabe von CASA oder der Schnittstelle.

Eingehende FIPA ACL-Nachrichten werden zunächst von der Schnittstelle des Empfängeragenten aufgenommen. Weil im Sensor von CASA-Agenten keine Nachrichten, sondern nur Ereignisse eingefügt werden können, muss die Schnittstelle empfangene Nachrichten erst in Ereignisse umwandeln und diese anschließend in den Sensor weiterleiten. Gelingt diese Umwandlung nicht, z.B. weil der Typ der Nachricht unbekannt ist, schickt die Schnittstelle eine not-understood-Nachricht an den Absender zurück.

Zusammenhang zwischen Nachrichten und Ereignissen. In der folgenden Tabelle werden die implementierten Nachrichtentypen aufgelistet, die jeder CASA-Agent ausführen kann. Nachrichten vom Typ INFORM_REC sind zwar noch nicht möglich, doch alle übrigen Nachrichtentypen aus der Spezifikation sind bereits realisiert (vgl. Abschnitt 6.1.5.).

In der Tabelle ist zusätzlich für jede Nachricht angegeben, in welchen Ereignistyp sie von der Empfänger-Schnittstelle gewandelt wird. Die aufgelisteten Ereignistypen entsprechen den im Abschnitt 6.3.1. spezifizierten *Ereignistypen mit externer Quelle.*

Nachricht des Senders	Ereignistyp der umgewandelten Nachricht im Empfängersensor
REQUEST <receiver> <msg name> <args>	NEW_GOAL
REPLY <receiver> <msg name> <args>	REPLY
INFORM <receiver> "NEW_FACT" <relation>	NEW_FACT
INFORM <receiver> "UPDATE_FACT" <relation>	UPDATE_FACT
INFORM <receiver> "NEW_PLAN" <plan>	NEW_PLAN
INFORM <receiver> "NEW_GOAL" <relation> <expression>	NEW_GOAL
INFORM <receiver> "SUSPEND" <STRING>	SUSPEND
INFORM <receiver> "RESUME" <STRING>	RESUME

Während bei Anfragen und Antworten alle Nachrichtennamen (<msg name>) vom Agentenentwickler selbst definiert werden, sind bei INFORM-Nachrichten einige Namen vordefiniert, z.B. UPDATE_FACT. Es ist möglich, bei INFORM auch andere Nachrichtennamen als die vordefinierten anzugeben, dafür muss dann allerdings in einer *individuellen* Agentenschnittstelle des Empfängeragenten eine entsprechende Behandlung implementiert sein.

7.5.2. Aufbau der Schnittstelle für CASA-Agenten

Im vorangehenden Abschnitt wurde deutlich, dass eine Aufteilung der Schnittstelle nötig ist, um die Voraussetzungen für die individuelle Erweiterung von Nachrichtenarten zu gewährleisten. Die Schnittstelle von CASA-Agenten setzt sich daher aus den folgenden drei Komponenten zusammen:

- Eine Komponente, in der individuelle Nachrichtentypen in CASA-Ereignisse umgewandelt werden. Diese Komponente sollte in Java programmiert werden, um problemlos in das Schnittstellenmodell eingefügt werden zu können.

  Es ist zweckmäßig, für bestimmte Typen von Agenten jeweils eine solche Schnittstellenklasse zu implementieren, also z.B. eine Klasse InRobot.java für Einlagerungsagenten oder eine Klasse Sortline.java für Sortierlinienagenten.

- Eine Komponente, in der die Umwandlung von empfangenen Nachrichten in Ereignisse analog zu Abschnitt 7.5.1. vorgenommen wird, d.h. diese Komponente setzt die Umwandlung der vordefinierten Nachrichtentypen in die entsprechenden Ereignisse um. Auf diese Weise erzeugte Ereignisse werden direkt an den Sensor des CASA-Agenten weitergeleitet.

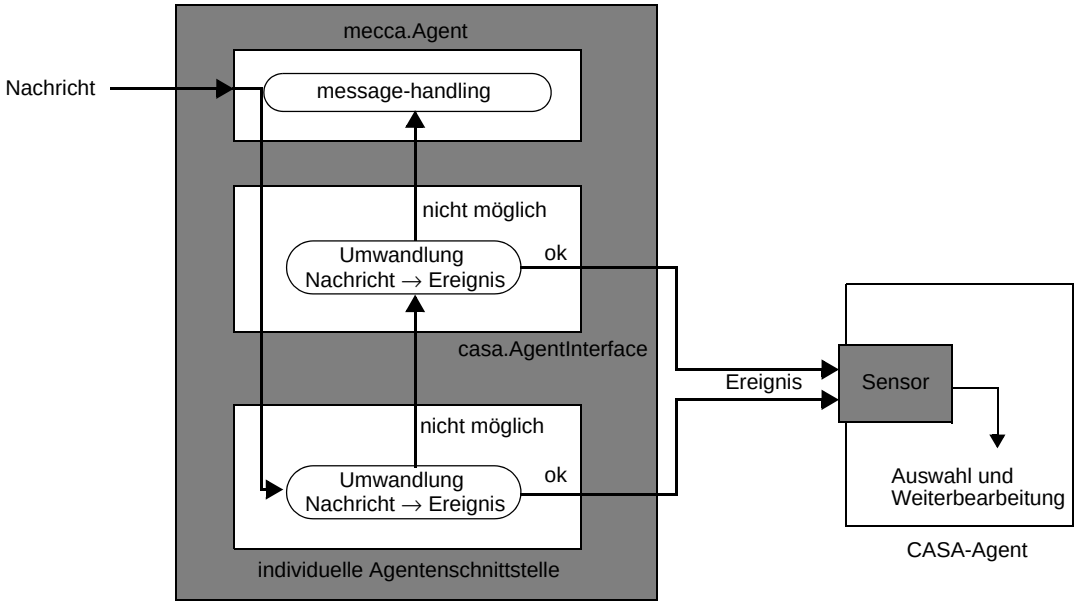
  Diese Komponente ist auch für die Aufbereitung von zu versendenen Nachrichten zuständig. Sie übernimmt vom CASA-Agenten die nötigen Daten und wandelt die Nachrichten in passende FIPA ACL-Nachrichten um.

  Implementiert ist diese Komponente in der Klasse casa.AgentInterface.

- Eine übergeordnete Komponente, die das Versenden und Empfangen von Nachrichten und sonstige administrative Aufgaben übernimmt, wie z.B. die Registrierung im Rahmen eines Multi-Agentensystems, das Auffinden geeigneter Kommunikationspartner oder das Versenden von not-understood-Nachrichten, wenn eine eingegangene Nachricht nicht zugeordnet werden konnte.

  Für diese Komponente wird die Klasse Agent aus dem MECCA-System verwendet.

Abbildung 34 Empfangen von Nachrichten in der Agentenschnittstelle



Zur Verdeutlichung des Schnittstellenaufbaus wird das Zusammenspiel der drei Komponenten anhand eines Szenarios in Abbildung 34 dargestellt. Die drei Komponenten sind so angeordnet, dass eintreffende Nachrichten zunächst an die individuelle Schnittstellenklasse weitergeleitet werden. Je nachdem, ob die Nachricht dort umgewandelt werden kann oder nicht, wird ein Ereignis an den Sensor geschickt oder die Nachricht an die übergeordnete Komponente (eine Instanz der Klasse AgentInterface) weitergegeben. Auch dort wird nach demselben Verfahren versucht, aus der Nachricht ein Ereignis zu erzeugen. Wenn dies nicht gelingt, wird die Nachricht schließlich an die dritte Komponente (eine Instanz der Klasse mecca.Agent) weitergegeben, die dann z.B. eine not-understood-Nachricht an den ursprünglichen Sender der Nachricht zurückschickt.

7.5.3. Anbindung von CASA-Agenten an MECCA

Mit den bisher beschriebenen Klassen ist es möglich, für CASA-Agenten Schnittstellen zu definieren. Agenten können aber erst dann Nachrichten verschicken, wenn eine Verbindung zu ihren Kommunikationspartnern besteht. Den Aufbau einer Verbindung kann man auf verschiedene Weisen erreichen. Einerseits gibt man dem Agenten von vornherein das Wissen über den Aufenthaltsort seiner Kommunikationspartner, z.B.

über die Angabe eines Ports oder einer IP-Adresse. Oder man lässt den Agenten bei einer Verwaltungsinstanz anfragen, wo sich der gewünschte Kommunikationspartner befindet. Der letztere Ansatz wird durch das MECCA-System unterstützt, das bei der Verwaltung von mehreren zusammenarbeitenden Agenten eine große Hilfe darstellt, wie auch schon in Abschnitt 5.3.3. erläutert worden ist.

In der Regel wird es in einem Multi-Agentensystem von Beginn an eine Menge von beteiligten Agenten geben. Damit diese Agenten nicht alle einzeln von einem Benutzer gestartet werden müssen, ist es ratsam, ein kleines Programm zu schreiben, das die Agenten der Reihe nach generiert. Im Falle von MECCA müssen zu Beginn die beiden Agenten DF (directory facilitator) und AMS (agent management system) erzeugt werden. Die übrigen am System beteiligten Agenten melden sich dann der Reihe nach bei den Agenten DF und AMS an. Das folgende – zum besseren Verständnis etwas vereinfachte – Codefragment zeigt am Beispiel des Farbsortierspeichers auf, wie ein Java-Programm zum Start eines Multi-Agentensystems mit MECCA und CASA-Agenten aussehen kann.

Abbildung 35 Java-Programm zur Generierung eines Multi-Agentensystems

```
public class ColorSortingSystem {
    String dfAddress        = "df@tcp://localhost:8000";
    String amsAddress       = "ams@tcp://localhost:8001";
    ...                                                    // initialize addresses of agents
    String inRobot2Address = "In-Robot-2@tcp://localhost:8010";
    ...                                                    // initialize agent specification files
    String inRobot2Program = "colorSorting/inRobot2.spec";


    public static void main (String[] argv) {
        CasaAgent agent;

        DF df = new DF ("df", dfAddress, new String, null);          // invoke MECCA system
        AMSAgent ams = new AMSAgent (amsAddress, dfAddress);

        ...                                                          // invoke CASA agents
        agent = new CasaAgent ("InRobot-2", inRobot2Program);
        agent.setInterface (new InRobot ("InRobot-2", inRobot2Address, dfAddress, amsAddress));
        agent.start ();
    }
}
```

Nachdem im Hauptprogramm die MECCA-Agenten DF und AMS generiert worden sind, werden die CASA-Agenten jeweils in den folgenden drei Schritten zum Multi-Agentensystem ergänzt:

1. Es wird eine neue Instanz der Klasse CasaAgent erzeugt. Als Parameter werden ein Agentenname und die zu parsende CASA-Spezifikationsdatei angegeben.
2. Wenn die CASA-Spezifikation ohne Fehler eingelesen werden konnte, wird die Schnittstelle generiert und dem Casa-Agenten mitgeteilt. Im obigen Beispiel ist eine individuelle Schnittstelle in der Klasse InRobot.java implementiert. Die Parameter zur Instanziierung der Schnittstellenklasse geben den Agentennamen und die

Adresse an, über die mit dem Agenten kommuniziert werden kann. Die weiteren Parameter teilen der Schnittstelle die Adressen von DF und AMS mit. Bei der Instanziierung der Schnittstelle wird automatisch versucht, sich bei DF und AMS anzumelden.

3. Nachdem die Anmeldung bei den Agenten DF und AMS erfolgreich verlaufen ist, beginnt der CASA-Agent seinen Interpreterzyklus durch den Aufruf seiner start()-Methode.

Bei dem angegebenen Programm werden die Agenten in einer einzigen virutellen Maschine gestartet, d.h die Agenten sind nicht auf verschiedene Rechner verteilt. Dennoch kommunizieren die Agenten ausschließlich über ihre Schnittstellen mittels des Transportprotokolls TCP. Echte Verteiltheit ist in diesem Fall einfach zu realisieren, indem CASA-Agenten aus verschiedenen JVM‘s und von anderen Rechnerknoten aus gestartet werden. Die Agenten müssen lediglich wissen, über welche Adressen DF und AMS zu erreichen sind. Auf diese Weise können beliebig viele weitere Agenten zum AMS und DF hinzugefügt werden.

Beim Farbsortierspeicher ist diese Möglichkeit z.B. dann sehr nützlich, wenn nach einiger Zeit ein weiterer Einlagerungsroboter zur Anlage ergänzt werden soll. In diesem Fall möchte man nicht das gesamte System anhalten und mit einem ergänzten Einlagerungsagenten neu starten, sondern diesen Agenten während des laufenden Betriebs in das System einfügen.

KAPITEL 8

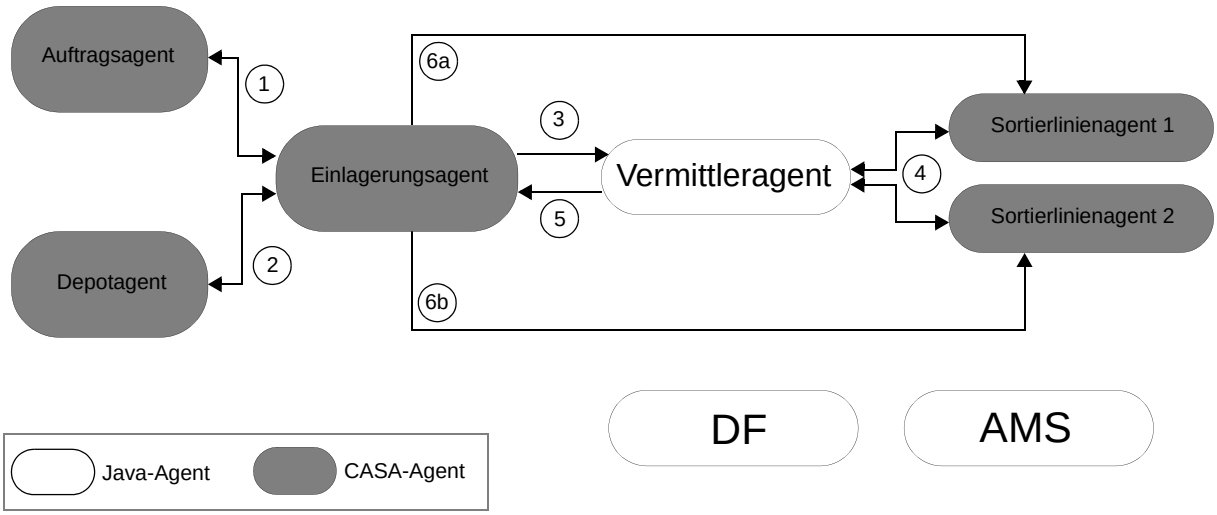
Beispiel: CASA-Agenten für den Farbsortierspeicher

Das in Kapitel 2 vorgestellte Modell des Farbsortierspeichers dient als Grundlage für das Beispiel eines Multi-Agentensystems mit CASA-Agenten. Betrachtet wird der Teil der Anlage, der sich mit der Einlagerung von Karosserien in den Sortierlinien beschäftigt.
Zunächst stelle ich die agentenübergreifende Sicht der Einlagerung vor, indem ich die Behandlung eines Auftrags anhand der auszutauschenden Nachrichten zwischen den beteiligten Agenten erläutere. Neben CASA-Agenten, die die Komponenten im Farbsortierspeicher repräsentieren, gibt es im implementierten Beispiel auch einen Vermittleragenten, der Einlagerungsagenten dabei unterstützt, eine geeignete Sortierlinie für ihren aktuellen Auftrag zu ermitteln.
In einem weiteren Abschnitt werden ausgewählte Strategien einiger CASA-Agenten vorgestellt. Dabei sind insbesondere die Strategien des Einlagerungsagenten interessant, weil bei seiner Spezifikation verschiedene Strategien zur Auswahl eines Auftrags angewendet werden.
Am Ende dieses Kapitels dokumentiert ein kommentierter Ausschnitt eines Laufzeitprotokolls die Lauffähigkeit der implementierten Beispielanwendung.

8.1. Ablauf der Einlagerung

Analog zu der Modellierung in Kapitel 2 werden die einzelnen Komponenten des Farbsortierspeichers durch CASA-Agenten repräsentiert. Das Anwendungsbeispiel, das in diesem Kapitel vorgestellt wird, umfasst nur den Teil der Einlagerung im Farbsortierspeicher. Zur Demonstration der Lauffähigkeit von CASA wird das kleinste System von Agenten zusammengestellt, das in diesem Zusammenhang sinnvoll ist. Es besteht aus insgesamt acht Agenten, davon sind die folgenden fünf in CASA spezifiziert: ein Auftragsagent, ein Depotagent, ein Einlagerungsagent und zwei Sortierlinienagenten. Zur Verwaltung des Systems werden noch die beiden MECCA-Agenten DF und AMS benötigt (vgl. Abschnitt 5.3.3.). Außerdem wird ein Vermittleragent eingesetzt, der für den Einlagerungsagenten eine geeignete Sortierlinie zu einer vorgegebenen Farbe ermittelt. In der folgenden Abbildung sind die beteiligten Agenten schematisch dargestellt.

Abbildung 36 Einlagerung im Farbsortierspeicher



Die eingezeichneten Linien verdeutlichen den Ablauf der Kommunikation bei der Bearbeitung eines Lackierauftrags. Zur besseren Übersichtlichkeit ist die Kommunikation der Agenten mit DF und AMS allerdings nicht dargestellt; Kommunikation mit DF und AMS ist z.B. beim Start jedes CASA-Agenten nötig, um sich zu registrieren und die Adressen der gewünschten Kommunikationspartner zu erfragen.

Im ersten Schritt des Einlagerungsvorgangs fragt der Einlagerungsagent beim Auftragsagenten nach einem neuen Auftrag an. Der Auftragsagent antwortet daraufhin mit einem Auftrag der Form (Auftragsnummer, Karosserietyp, Farbe). Anschließend fordert der Einlagerungsagent vom Depotagenten eine passende Karosserie an (Schritt 2). Nun soll der Einlagerungsagent die Karosserie zu der Sortierlinie bringen, die sich für die Pufferung des Auftrags am besten eignet. Dazu fragt der Einlagerungsagent bei dem Vermittleragenten an (Schritt 3), der die Aufgabe übernimmt, bei allen Sortierlinienagenten danach zu fragen, wie groß ihr Verlangen nach der vorgegebenen Farbe ist (Schritt 4). Die Sortierlinien antworten darauf mit einem bestimmten Wert, dessen

Ermittlung komplexe Berechnungen erfordern kann, z.B. kann es nötig sein, dass sich die Sortierlinien untereinander abstimmen müssen, bevor sie auf die Anfrage angemessen antworten können. Der Vermittler stellt nach Erhalt aller Antworten die Sortierlinie fest, die das größte Verlangen nach der angegebenen Farbe hat und gibt diese Information an den Einlagerungsagenten zurück (Schritt 5). Der Einlagerungsagent liefert den Auftrag dann bei dieser Sortierlinie ab (Schritt 6a oder 6b) und ist anschließend wieder für einen neuen Auftrag bereit.

8.2. Agenten-Spezifikationen

In diesem Abschnitt sollen nur die wichtigsten Pläne der Agenten erläutert werden, um einen Eindruck davon zu vermitteln, wie Spezifikationen eines lauffähigen Systems aussehen können. Darüber hinaus sind die aufgeführten CASA-Pläne zur besseren Lesbarkeit auf die wesentlichen Aktionen beschränkt, z.B. habe ich an vielen Stellen solche Aktionen weggelassen, die lediglich Informationen auf dem Bildschirm ausgeben.

Der Auftragsagent. In der CASA-Spezifikation des Auftragsagenten werden einige Aufträge als Fakten angegeben, die von Beginn an vorhanden sein sollen. Das Eintreffen neuer Aufträge wird zur Zeit dadurch simuliert, dass durch eine zyklische Top-Level-Intention von Zeit zu Zeit ein neuer Auftrag generiert und in der Wissensbasis abgelegt wird.

Die wesentliche Aufgabe des Auftragsagenten ist es, auf Anfragen eines Einlagerungsagenten einen Auftrag zu vergeben, den der Einlagerungsroboter transportieren kann; nicht jeder Einlagerungsroboter ist in der Lage, alle Karosserietypen aufzunehmen. Um das Beispiel nicht zu komplex werden zu lassen, wird angenommen, dass Einlagerungsroboter nur einen einzigen Karosserietypen transportieren können. Hierüber erhält der Auftragsagent Kenntnis, indem der Einlagerungsagent zu Beginn seiner Laufzeit eine entsprechende Nachricht verschickt. Der Auftragsagent speichert diese Information in seiner Wissensbasis und vergibt später aufgrund dieses Wissens nur geeignete Aufträge an den Einlagerungsagenten.

Durch die individuelle Schnittstelle des Auftragsagenten werden Auftragsanfragen in NEW_GOAL-Ereignisse mit der Relation "getOrder <sender>" umgewandelt. In der Spezifikation muss ein Plan angegeben werden, der das Ziel "getOrder" mit einem Argument (für die Übergabe des Sendernamens) zu erfüllen verspricht und eine Antwort erzeugt. Die folgende Abbildung führt einen Beispielplan hierfür auf.

Abbildung 37 Antwortplan zur Vergabe von Aufträgen

```
{   NAME:       "give an order to an InRobot";
    GOAL:       ACHIEVE       getOrder     $sender;
    TYPE:       REACTIVE;
    PRIORITY:   10.0;
    BODY:       GETVALUES     robotType    $sender $bodyType;
                GETVALUES     order        $nr $bodyType $color;
                REPLY         $sender      "getOrder" $nr $bodyType $color;
                DELETE        order        $nr $bodyType $color;

    FAILURE:    REPLY         $sender      "getOrder" "none" "n" "n";
}
```

Der Plan hat keine Vorbedingung, sodass er nach Auswahl des NEW_GOAL-Ereignisses "getOrder <sender>" direkt als Intentionsplan übernommen und ausgeführt wird. Mit GETVALUES werden die benötigten Fakten aus der Wissensbasis geholt: erst ein für den anfragenden Einlagerungsroboter geeigneter Karosserietyp, dann ein Auftrag mit diesem Karosserietyp. Wenn diese Fakten erfolgreich aus der Wissensbasis ausgelesen werden konnten, wird eine Antwort an den Einlagerungsagenten verschickt und der Auftrag in der Wissensbasis gelöscht. Wenn kein geeigneter Auftrag vorliegt, schlägt die GETVALUES-Aktion fehl, sodass die FAILURE-Sektion ausgeführt und dem Einlagerungsagenten mitgeteilt wird, dass zur Zeit kein geeigneter Auftrag für ihn vorliegt.

Der Depotagent. Die Vergabe von Karosserien wird durch den Austausch von Nachrichten zwischen Einlagerungsagent und Depotagent simuliert. Der Depotagent ist sehr ähnlich wie der Auftragsagent spezifiziert und wird daher nicht genauer erläutert.

Der Einlagerungsagent. In der Spezifikation des Einlagerungsagenten wird ein Plan mit dem Namen "busy" angegeben, der in einer Top-Level-Intention von Beginn an zyklisch abgearbeitet wird.

Abbildung 38 Vereinfachter Hauptplan des Einlagerungsagenten

```
{   NAME:       "plan for top level goal: busy";
    GOAL:       ACHIEVE busy;
    TYPE:       REACTIVE;
    PRIORITY:   1;

    BODY:  GETFACT    myId          $id;
           GETFACT    orderAgent    $receiver;          // inform orderAgent about the type of bodies I can carry
           INFORM     $receiver     "NEW_FACT" "robotType" $id "sedan";

           WHILE  TEST (> 1 0)      {                                                          // infinite loop
               GETFACT    myState       $state;
               IF   TEST     (== $state "waiting") {
                   PARALLEL                                    // parallel asking for order and moving to depot
                   {   ACHIEVE    getOrder     $nr $bodyType $color : 1.0;   }
                   {   ACHIEVE    gotoDepot    : 1.0;                        }

                   IF   TEST     (!= $nr "none") {                                  // getOrder was successful
                       ASSIGN       $bodyHelp          $bodyType;
                       PARALLEL                                  // parallel loading and asking for sortline
                       {   ACHIEVE    loadRobot         $bodyHelp: 1.0;      }
                       {   ACHIEVE    requestBroker     $color $line : 1.0;  }

                       IF   TEST     (&& (!= $bodyHelp "none") (!= $line "none") ) {          // delivering
                           ACHIEVE    gotoLine      $line : 5.0;
                           ACHIEVE    store         $nr $bodyType $color $line : 5.0;
                       }
                       ELSE {                                  // give back the order if something went wrong
                           INFORM     $receiver     "giveBackOrder" $nr $bodyType $color;
                       }
                   }
                   UPDATE       (myState) (myState "waiting");
               }
           }   // end of while
}
```

Mit dem Plan "busy" werden die notwendigen Aktionen von der Auftragsanfrage bis zur Ablieferung bei einer Sortierlinie eingeleitet, indem – teilweise parallele – Unterziele verfolgt werden. Die physische Übergabe von Karosserien sowie die Fahrten des Einlagerungsroboters werden durch Unterziele wie "gotoDepot" oder "loadRobot" simuliert, die entsprechende Meldungen auf dem Bildschirm ausgeben lassen.

Die interessantesten Stellen in diesem zyklischen Hauptplan des Einlagerungsagenten sind fett gedruckt. Während der Agent dem Einlagerungsroboter mitteilt, dass er sich zum Depot bewegen soll, kann gleichzeitig schon ein neuer Auftrag ermittelt werden. Daher werden die beiden Unterziele "gotoDepot" und "getOrder" parallel und mit gleicher Gewichtung spezifiziert. Für das Unterziel "getOrder" gibt es im Einlagerungsagenten eine Reihe von relevanten Plänen, auf die ich später noch eingehen werde. Nachdem der Roboter beim Depot angelangt ist und erfolgreich ein neuer Auftrag übernommen werden konnte, kann die Beladung parallel zur Ermittlung der Zielsortierlinie erfolgen, d.h. die Unterziele "loadRobot" und "requestBroker" können ebenfalls parallel verfolgt werden. Wenn das Beladen und die Ermittlung einer Sortierlinie erfolgreich abgeschlossen sind, wird der Einlagerungsroboter vom Agenten angewiesen, zu der Sortierlinie zu fahren und dort die Karosserie abzuliefern. Für den Fall, dass die gewünschte Karosserie nicht geladen oder keine Sortierlinie ermittelt werden konnte, wird der Auftrag mit einer INFORM-Nachricht an den Auftragsagenten zurückgeschickt.

In dem obigen Plan wird aus Übersichtlichkeitsgründen nicht angegeben, dass auch noch überwacht wird, wie oft schon erfolglos beim Auftragsagenten nach Aufträgen angefragt worden ist. Wenn eine bestimmte Anzahl erfolgloser Versuche erreicht ist, wartet der Einlagerungsagent so lange, bis er vom Auftragsagenten einen Auftrag *ohne weitere Nachfrage* erhält. Der Auftragsagent überwacht ständig, wie lange der Einlagerungsagent schon nicht mehr nach Aufträgen angefragt hat. Wenn eine bestimmte Zeit verstrichen ist, schickt der Auftragsagent dem Einlagerungsagenten von sich aus einen Auftrag zu. Dieser Auftrag wird zunächst in die Wissensbasis des Einlagerungsagenten als Fakt "order" oder "importantOrder" mit den entsprechenden Parameterwerten aufgenommen. Erst wenn ein Fakt mit einem dieser Relationennamen in der Wissensbasis vorliegt, verlässt der Einlagerungsagent seine Warteschleife und beginnt erneut mit dem Abarbeitungszyklus des Hauptplans.

Zur Erreichung des Unterziels "getOrder" hat der Einlagerungsagent mehrere Pläne mit unterschiedlichen Typen und Vorbedingungen zur Auswahl. Diese Pläne werden in Abbildung 39 angegeben (zur besseren Übersichtlichkeit habe ich bei jedem Plan einige Sektionen ausgelassen). Die ersten beiden angegebenen Pläne sind reaktiv, sodass bei der Planauswahl ihre Vorbedingungen überprüft werden. Konkret wird in den Vorbedingungen das Vorhandensein eines Fakts mit dem Relationennamen "order" bzw. "importantOrder" überprüft. Wenn die Planauswahlfunktion Plan_HighestPriority benutzt wird, ist sichergestellt, dass durch die höhere Priorität von Plan 1 wichtige Aufträge den normalen Aufträgen in der Wissensbasis vorgezogen werden. Wenn kein Auftrag in der Wissensbasis vorliegt, sind die Pläne 1 und 2 nicht anwendbar und der kommunikative Plan 3 muss in der CommStructure überprüft werden. In der Vorbedingung dieses Plans wird übeprüft, ob der Auftragsagent auf eine Anfrage hin einen Auftrag an den Einlagerungsagenten abgeben kann. Die Antwortparameter werden in der Zeile "GETREPLY" in den Plan lokal übernommen und gespeichert, sodass bei der Intentionsbildung die nötigen Werte erhalten bleiben.

Abbildung 39 Verschiedene Pläne für das Ziel "getOrder"

```
Plan 1   {  GOAL:          ACHIEVE getOrder $nr $bodyType $color;
            TYPE:          REACTIVE;
            PRECONDITION: GETFACT importantOrder $nr $bodyType $color;
            PRIORITY:      4.0;
            BODY:          ...
                           EXECUTE print $id ":  taking the important order (" $nr ","  $bodyType "," $color ")\n";
         }

Plan 2   {  GOAL:          ACHIEVE getOrder $nr $bodyType $color;
            TYPE:          REACTIVE;
            PRECONDITION: GETFACT order $nr $bodyType $color;
            PRIORITY:      2.0;
            BODY:          ...
                           EXECUTE print $id ":  taking the order (" $nr ","  $bodyType "," $color ")\n";
         }

Plan 3   {  GOAL:          ACHIEVE getOrder $nr $bodyType $color;
            TYPE:          COMMUNICATIVE;
            PRECONDITION: ....
                           REQUEST $orderAgent "getOrder" : 1.0;
                           GETREPLY $nr $bodyType $color;
            PRIORITY:      2.0;
            BODY:          ...
                     EXECUTE print $id ": got the order (" $nr ","  $bodyType "," $color ") from " $orderAgent "\n";
         }
```

Der Vermittleragent. Für diesen Agenten wurde keine CASA-Spezifikation angelegt, denn seine Aufgabe besteht hauptsächlich darin, Nachrichten zu empfangen und weiterzuleiten. Der Vermittleragent wird von dem Einlagerungsagenten unter Angabe einer bestimmten Farbe aufgefordert, ihm die Sortierlinie mit dem größten Verlangen nach Lackieraufträgen dieser Farbe mitzuteilen. Dazu initiiert der Vermittleragent ein contract-net-protocol, wie es in Abschnitt 4.3. vorgestellt worden ist. Die zur Durchführung des Protokolls notwendigen Nachrichtentypen sind im MECCA-System FIPA-konform implementiert, sodass der Vermittleragent einfach die entsprechenden Methoden aufrufen kann. Die einzige Entscheidung, die der Vermittleragent treffen muss, ist die Auswahl der Sortierlinie mit dem größten Verlangen nach der vorgegebenen Farbe. Dies lässt sich mit einigen Zeilen Java-Code aber schneller realisieren als hierfür extra einen CASA-Agenten einzurichten.

An diesem Beispiel erkennt man, dass es manchmal auch ausreicht, eine abgeleitete Klasse von mecca.Agent um etwas applikationsabhängigen Java-Code zu ergänzen. Erst ab einer gewissen Komplexität von Komponenten, z.B. wenn mehrere Entscheidungen zu treffen sind oder verschiedene Strategien zur Auswahl stehen, lohnt es sich, einen CASA-Agenten zu spezifizieren.

Die Sortierlinienagenten. Neben den aktuell eingelagerten Aufträgen verwalten die Sortierlinienagenten Werte, die das Verlangen nach den verschiedenen Farben repräsentieren, und teilen diese Werte dem Vermittleragenten auf Anfrage mit. Zur Zeit wird in den Sortierlinien für jede Farbe lediglich ein konstanter Wert angenommen, der auf Anfrage des Vermittleragenten zurückgeliefert wird. Für die gegebenenfalls sehr komplexe Neuberechnung des Verlangens nach Farben gebe ich keine Strategien an, denn

hierfür sinnvollen Strategien zu formulieren, die z.B. die Abstimmung von Sortierlinien untereinander betreffen, ist eine Aufgabe für sich.

8.3. Laufzeitprotokoll

Im Anhang C ist ein Laufzeitprotokoll angegeben, das einige Runden des Einlagerungsagenten und die Nachrichten der anderen Agenten im Farbsortierspeicher dokumentiert. Das Programm wurde auf einer Sun Ultra 1 Workstation ausgeführt, auf der die Initialisierung des MECCA-Systems und der Agenten 30 Sekunden und eine komplette Auftragsbearbeitung des Einlagerungsagenten zwischen 5 und 15 Sekunden dauert, abhängig davon, ob der Einlagerungsagent einen Auftrag in der Wissensbasis vorliegen hat oder ob er erst beim Auftragsagenten anfragen muss.

In dem Laufzeitprotokoll ist u.a. folgende Einzelheiten zu erkennen:

- Der Einlagerungsagent arbeitet Aufträge wie gewünscht ab und der Vermittleragent ermittelt die Sortierlinie mit dem größten Verlangen nach der angegebenen Farbe (Runde 1).
- Die Reihenfolge der Anfragen beim Depot- und beim Vermittleragenten kann sich von Runde zu Runde unterscheiden (z.B. in Runde 1 und Runde 16).
- Nach drei vergeblichen Versuchen einen Auftrag vom Auftragsagenten zu erhalten wartet der Einlagerungsagent, bis ihm ein Auftrag zugewiesen wird (Runden 17 bis 19).
- Der Auftragsagent verschickt nach einiger Zeit einen Auftrag an den Einlagerungsagenten, ohne dass der Einlagerungsagent ihn dazu auffordert (zwischen Runde 19 und 20).
- Sobald ein Auftrag in der Wissensbasis vorliegt, arbeitet der Einlagerungsagent weiter (Runde 20).

KAPITEL 9 Zusammenfassung und Ausblick

Bei meiner Diplomarbeit stand die Entwicklung einer Spezifikationssprache für ein formales Agentenmodell im Vordergrund. Im Einzelnen wurde Folgendes erreicht:

1. Ich habe das zugrunde liegende Agentenmodell AgentGHC um Ereignistypen erweitert. Statt der informell beschriebenen rekursiven Aufrufe im Interpreterzyklus habe ich zwei neue Komponenten zum Interpretermodell hinzugefügt, die für spekulative Berechnungen zuständig sind.
2. Ich habe die *Spezifikationssprache CASA* entworfen, mit der man „intelligente" Agenten im Sinne der KI-Sicht spezifizieren kann, d.h. Agenten werden durch mentalistische Kategorien wie Überzeugungen, Ziele und Strategien beschrieben. Die wesentlichen Merkmale dieser Sprache sind
   - priorisierte Pläne auf Basis von bewachten Horn-Klauseln mit der Möglichkeit, parallele Abarbeitung explizit kenntlich zu machen (auch in Vorbedingungen),
   - eine Hierarchie bezüglich der Abarbeitung verschiedener Plantypen,
   - „tiefe" Ziele mit der Durchführung spekulativer Berechnungen und
   - die explizite Unterbrechbarkeit und Wiederaufnahme von Intentionen.
3. Die *Implementierung eines Prototypsystems* für CASA wurde von mir in Java vorgenommen. Ein Parser liest Spezifikationen für CASA-Agenten aus Dateien ein und gibt bei Fehlern nützliche Hinweise zur Fehlerstelle aus. Das implementierte System generiert aus korrekten Spezifikationen CASA-Agenten entsprechend dem vorgestellten formalen Modell. Um Multi-Agentensysteme mit CASA-Agenten bilden zu können, habe ich eine Schnittstelle implementiert, mit der man CASA-Agenten an das MECCA-System anbinden kann.
4. Zur *Realisierung eines Anwendungsbeispiels* diente die agentenbasierte Modellierung des Farbsortierspeichers. Der Teil der Einlagerung von Karosserien in den Sortiergassen wurde durch mehrere CASA-Agenten simuliert und dabei von einem Vermittleragenten und dem MECCA-System unterstützt.

Der praktische Nutzen von CASA

CASA unterstützt mit den Metaphern mentalistischer Begriffe besonders die frühen Phasen des Systementwurfs. Wie bei der Modellierung des Farbsortierspeichers gezeigt wurde, kann mit der Agentensicht eine modulare und strukturtreue Abbildung von Pro-

duktionsprozessen erfolgen. Durch den Einsatz von Agenten sind auf diese Weise autonome und kooperative Produktionssysteme möglich, die flexibler als andere konventionelle Systeme arbeiten, u.a. weil Planung und Ausführung der Produktion nicht mehr zentral und streng getrennt nacheinander ablaufen müssen, sondern auf veränderte Bedingungen lokal zur Laufzeit angemessen gehandelt werden kann.

Näher zu untersuchen ist die Eignung von CASA-Agenten in anderen Einsatzgebieten, z.B. bei der Mensch-Maschine-Interaktion. Denkbar ist es, CASA-Agenten für verschiedene animierte Objekte zu erstellen, die zu gegebener Zeit die Aufmerksamkeit des Benutzers auf sich ziehen und ihre Hilfe anbieten.

Lauffähigkeit und Laufzeitverhalten

Das implementierte Beispiel ist noch nicht auf verschiedene Rechner verteilt, sondern läuft zur Zeit lokal auf einer Maschine ab. Ich habe dennoch bereits die Lauffähigkeit von verteilten CASA-Agenten durch ein weiteres Testsystem überprüft, bei dem sich zwei Agenten auf verschiedenen Rechnern befinden und die korrekte Bearbeitung der implementierten Ereignistypen aufgezeigt wird. Es werden mehrmals Nachrichten von einem Agenten A zu einem Agenten B geschickt, in denen die Änderungen von Fakten, neue Ziele sowie die Suspendierung und Wiederaufnahme der Abarbeitung von Intentionen angegeben werden. Agent B wandelt in seiner Schnittstelle die empfangenen Nachrichten in Ereignisse um und kann sie dann im Interpreterzyklus abarbeiten.

Die Laufzeiten von CASA-Agenten bewegen sich in einem akzeptablen Rahmen. Sie hängen insbesondere von der Anzahl und den Typen der zu überprüfenden Pläne sowie der Komplexität der Intentionen ab. Beispielsweise beträgt die Laufzeit eines CASA-Agenten für die Ausführung von über 12.000 Zyklen und Aktionen (darunter Überprüfungen deliberativer Pläne, Bildschirmausgaben, Operationen auf der Wissensbasis und parallele Unterziele) auf einer Sun Ultra 1 Workstation gut 30 Sekunden.

Erweiterungen von CASA

Auf der Ebene des CASA-Agentenmodells wurde bereits eine Fuzzy-Bibliothek zur Beschreibung von unscharfem Wissen eingefügt [Nwana 96]. Bisher nicht vorhanden sind Konzepte für adaptives Verhalten und Mobilität.

Auf der Sprachebene ist CASA von vornherein darauf ausgelegt, spätere Erweiterungen leicht einbinden zu können. Die EXECUTE-Aktion ermöglicht es, weitere primitive Aktionen in Java-Code zu ergänzen, ohne bestehenden Code ändern zu müssen. Auf diese Weise ist z.B. bereits die Anbindung an ein Java-Applet realisiert worden, in dem Bilder durch einen entfernten CASA-Agenten verschoben werden können. Jedoch fehlt es zur Zeit noch an einem durchdachten Konzept zur systematischen Unterstützung von grafischen Oberflächen durch CASA-Agenten. Dies stellt eine interessante Aufgabe für die Zukunft von CASA dar.

Für die Spezifikation und Überprüfung paralleler kommunizierender Objekte sind visuelle Notationen besonders hilfreich. Daher würde eine Werkzeugumgebung zur visuellen Entwicklung von CASA-Agenten eine große Hilfe darstellen, insbesondere wenn es um den Zusammenschluss mehrerer CASA-Agenten geht. Auch beim Debugging von CASA-Agenten sollte der Entwickler durch eine grafische Oberfläche unterstützt werden, denn durch die zyklische Abarbeitung kommt es schnell zu einer Vielzahl von Informationsausgaben auf dem Bildschirm.

Literaturverzeichnis

[AgentSociety] The Agent Society: http://www.agent.org

[Aglets] IBM Tokio Research Laboratory: *Aglets Software Development Kit*. http://www.trl.ibm.co.jp/aglets/index.html

[Austin 62] Austin, J. L.: *How to Do Things with Words.* Clarendon Press, 1962.

[Bauer 98] Bauer, B.; Steiner, D.: *MECCA: Konzepte und Praxis*. Seminarunterlagen der Siemens AG, München, 1998.

[Bienkowski 94] Bienkowski, M.; des Jardin, M.; Desimone, R.: *SOCAP: system for operations crisis action planning.* In: Proc. of the ARPA/Rome Lab 1994 Knowledge-Based Planning and Scheduling Initiative Workshop, 1994.

[Bigus 97] Bigus, J. P.: *Constructing intelligent agents with Java: a programmer‘s guide to smarter applications.* Wiley Computer Publishing, New York, USA, 1997.

[Bratman 87] Bratman, M. E.: *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, USA, 1987.

[Brooks 86] Brooks, R. A.: *A robust layered control system for a mobile robot.* IEEE Journal of Robotics and Automation, 2(1), 1986, S. 14-23.

[Burkhard 98] Burkhard, H.-D.: *Einführung in die Agenten-Technologie*. In: [it+ti 98], 1998, S. 6-11.

[Concordia] Mitsubishi Electric Information Technology Center America (ITA): *Concordia Version 1.1.2*. http://www.meitca.com/HSL/Projects/Concordia

[CORBA] Object Management Group: *CORBA, Common Object Request Broker Architecture*. http://www.corba.org

[DARPA 93] DARPA Knowledge Sharing Initiative, External Interfaces Work Group (Finin et al): *Draft Specification of the KQML Agent-Communication Language*. University of Maryland Baltimore County, Baltimore, MD, USA, 1993. http://www.cs.umbc.edu/kqml/

[Dennett 71] Dennett, D. C.: *Intentional Systems.* The Journal of Philosophy 68, 1971.

[Doran 97] Doran, J. E.; Franklin, S.; Jennings, N. R.; Norman, T. J.: *On Cooperation in Multi-Agent Systems.* The Knowledge Engineering Review, 12(3), 1997.

[Etzioni 94] Etzioni, O.; Weld, D.: *A Softbot-Based Interface to the Internet.* Communications of the ACM, 37(7), 1994, S. 72-76.

[Ferber 94] Ferber, J.: *Simulating with Reactive Agents.* In: Hillebrand, E.; Stender, J. (Hsg.): Many Agent Simulation and Artificial Life. IOS Press, Amsterdam, Niederlande, 1994, S. 8-28.

[Ferguson 92] Ferguson, I. A.: *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* Dissertation, Clare Hall, University of Cambridge, United Kingdom, 1992. Auch: Tech. Rep. No. 273, University of Cambridge Computer Laboratory

[Finin 95] Finin, T.; Labrou, Y.; Mayfield, J.: *KQML as an agent communication language.* 1995. In: Bradshaw, J.(Hsg.): Software Agents. AAAI/MIT Press, 1997. Auch: http://www.cs.umbc.edu/kqml/papers/

[FIPA] Foundation for Intelligent Physical Agents (FIPA): http://www.fipa.org

[FIPA 97] Foundation for Intelligent Physical Agents: *FIPA 97 Specification, Version 1.0*. Genf, Schweiz, 1997. Auch: http://www.fipa.org/spec/fipa97.html

[Flake 99] Flake, S.; Müller, W.; Geiger, C.; Lehrenfeld, G.; Paelke, V.: *Agent-Based Modeling for Holonic Manufacturing Systems with Fuzzy Control.* NAFIPS'99, 18th International Conference of the North American Fuzzy Information Processing Society, New York, USA, Juni 1999.

[Flanagan 97] Flanagan, D.: *Java in a Nutshell, A Desktop Quick Reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[Franklin 96] Franklin, S.; Graesser, A.: *Is it an Agent, or just a Program: A Taxonomy for Autonomous Agents.* In: Proc. of the Third International Workshop on Agent Theories, Architecture and Languages (ATAL), Budapest, Ungarn, 1996, S. 193-206. Auch: http://www.msci.memphis.edu/franklin/AgentProg.html

[Fritsche 95] Fritsche, R.: *Entwicklung und Beurteilung von Verfahren zur Reihenfolgeoptimierung in Farbsortierspeichern*. Diplomarbeit, Technische Hochschule Darmstadt, April 1995.

[Geiger 98] Geiger, C.: *Schneller Entwurf interaktiver dreidimensionaler Computeranimationen: Beiträge zur wissensbasierten Modellierung und Illustration komplexer technischer Systeme.* Dissertation, Universität-Gesamthochschule Paderborn, 1998.

[Genesereth 91] Genesereth, M. R.: *Knowledge interchange format.* In: Proc. of the 2nd International Conference on Principles of Knowledge Representation and Reasoning, 1991, S. 599-600.

[Georgeff 87] Georgeff, M. P.; Lansky, A. L.: *Reactive reasoning and planning.* In: Proc. of the Third National Conference on Artificial Intelligence (AAAI-83), Washington, D.C.,USA, 1987.

[Georgeff 94] Georgeff, M. P.: *Distributed Multi-Agent System (dMARS)*, 1994. http://www.aaii.oz.au/proj/dmars_tech_overview/dMARS-1.html

[Gruber 93] Gruber, T. R.: *A translation approach to portable ontologies.* Knowledge Acquisition, 5(2), S.199-220, 1993. Auch: http://www-ksl.staford.edu/kst/what-is-an-ontlogy.html

[Huber 98] Huber, M. J.: *JAM! Agents definitely not for dummies, Version 0.4+0.9i.* Intelligent Reasoning Systems, Oceanside, USA, 1998. http://members.home.net/marcush/IRS

[Huber 99] Huber, M. J.: *Jam Agents in a Nutshell, Version 0.60+0.80i.* Intelligent Reasoning Systems, Oceanside, USA, 1999. http://members.home.net/marcush/IRS

[Huhns 97a] Huhns, M. N.; Singh, M. P. (Hsg.): *Readings in Agents*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Huhns 97b] Huhns, M. N.; Singh, M. P.: *Agents and Multiagent Systems: Themes, Approaches, and Challenges.* In: [Huhns 97a], 1997, S. 1-23.

[it+ti 98] it+ti (Informationstechnik und Technische Informatik): *Agenten – eine neue Technologie auf dem Vormarsch*, Heft 4, Oldenbourg Verlag, München, 1998.

[JavaCC] Sun Microsystems: *Java™ Compiler Compiler™.* Version 0.8pre2, September 1998. http://www.sun.com/suntest/products/JavaCC/index.html

[Java 1.2] Sun Microsystems: *Java™ 2 SDK, Standard Edition Documentation, Version 1.2.1.* http://java.sun.com/products/jdk/1.2/

[JATLite] Stanford University: *Java Agent Template, Lite* http://java.stanford.edu/java_agent/html/

[Jennings 96] Jennings, N. R.; Sycara, K.; Wooldridge, M.: *A Roadmap of Agent Research and Development*. In: Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers, Boston, USA, 1998, S. 275-306.

[Jennings 98] Jennings, N. R., Wooldridge, M.: *Agent Technology: Foundations, Applications, and Markets*. Springer Verlag, Heidelberg, 1998.

[Knapik 98] Knapik, M.; Johnson, J.: *Developing intelligent agents for distributed systems: exploring architecture, technologies, and applications*. McGraw-Hill, New York, USA, 1998.

[Kozierok 93] Kozierok, R.; Maes, P.: *A learning Interface Agent for Scheduling Meetings.* In: Proc. of the ACM-SIGCHI International Workshop on Intelligent User Interfaces, Florida, 1993, S. 81-93.

[Labrou 97] Labrou, Y.; Finin, T.: *Comments on the specification for FIPA 97 Agent Communication Language.* Februar 1997. http://www.cs.umbc.edu/agentslist/archive/1996b/0393

[Maes 91] Maes, P.: *The agent network architecture (ANA)*. SIGART Bulletin, 2(4), 1991, S. 115-120.

[MAS] MultiAgent systems: *Research and Industrial information about multi agent systems*. http://www.multiagent.com

[Mattern 98] Mattern, F.: *Mobile Agenten.* In: [it+ti 98], 1998, S. 12-17.

[Müller 96a] Müller, J. P.: *The Design of Intelligent Agents: A Layered Approach*. Springer Verlag, Berlin, 1996.

[Müller 96b] Müller, J. P.: *An Architecture for Dynamically Interacting Agents*. Dissertation am Deutschen Forschungsinstitut für Künstliche Intelligenz (DFKI), Saarbrücken, 1996.

[Nwana 96] Nwana, H. S.: *Software Agents: An Overview.* Intelligent Systems Research AA&T, BT Laboratories, Ipswich, United Kingdom, 1996.

[Odyssey] General Magic, Inc.: *Odyssey Information*. http://www.genmagic.com/technology/odyssey.html

[OMG] Object Management Group: http://www.omg.org

[Orfali 97] Orfali, R.; Harkey, D.; Edwards, J.: *Instant CORBA.Prentice Hall, 1997.*

[Petrie 96] Petrie, C. J.: *Agent-Based Engineering, the Web, and Intelligence.* In: IEEE Expert/Intelligent Systems & Their Applications 11(6), 1996, S. 24-29. Auch: http://cdr.stanford.edu/NextLink/Expert.html

[Rao 91] Rao, A. S.; Georgeff, M. P.: *Modeling rational agents within a BDI-architecture.* Technical Report 14, Australian Artificial Intelligence Institute, Melbourne, Australia, 1991.

[Rao 92] Rao, A. S.; Georgeff, M. P.: *An abstract architecture for rational agents.* In: Rich, C.; Swartout, W.; Nebel, B. (Hsg.): Proc. of the Third International Conference on Principles of Knowledge Representation and Reasoning. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1992.

[Rao 95] Rao, A. S.; Georgeff, M. P.: *BDI Agents: From Theory to Practice.* In: Proc. of the 1st International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, USA, 1995, S. 312-319.

[Rao 96] Rao, A. S.: *AgentSpeak(L): BDI agents speak out in a logical computable language.* Tech. Rep. 64, Australian Artificial Intelligence Institute, Melbourne, Australia, 1996.

[Sattler 98] Sattler, K.-U.: *Vorlesungsskript Software-Agenten zur Informationssuche und -filterung, WS 98/99.* Institut für Technische und Betriebliche Informationssysyteme, Otto-von-Guericke-Universität, Magdeburg, 1998. Auch: http://wwwiti.cs.uni-magdeburg.de/~sattler/lectures/skript.html

[Shoham 93] Shoham, Y.: *Agent-oriented programming.* Artificial Intelligence 60(1), 1993, S. 51-92.

[Smith 80] Smith, R.: *The Contract Net Protocol: High-Level Communications and Control in a Distributed Problems Solver.* IEEE Transactions on Computers, Vol 29(12), 1980.

[Spehr 99] Spehr, M.: *Sun will alles vernetzen.* Frankfurter Allgemeine Zeitung, 19.01.1999, Nr.15, Seite T1.

[Steiner 98] Steiner, D.: *Die FIPA-Initiative für Agenten-Standardisierung.* In: [it+ti 98], S. 46-49.

[UMBC] University of Maryland, Baltimore County: *UMBC AgentWeb.* http://www.cs.umbc.edu/agents/

[Voyager] ObjectSpace, Inc.: *Voyager.* http://www.objectspace.com/products/index.html

[Wagner 97a] Wagner, G.: *Possible Worlds Semantics versus Operational Semantics.* April 1997. http://www.cs.umbc.edu/agentslist/archive/1996b/0609.html

[Wagner 97b] Wagner, G.: *Software mit Managerqualitäten.* c‘t (15), 1997, S. 234-243.

[Wooldridge 95] Wooldridge, M.; Jennings, N. R.: *Intelligent Agents: Theory and Practice.* In: Knowledge Engineering Review, 10(2), 1995, S. 115-152.

Abstrakter Interpreter von AgentGHC

Parallele Guards und Aktionen in Strategien werden durch Kommas getrennt, sequentielle Abarbeitung wird durch ein Semikolon kenntlich gemacht. Die Funktionen für Zugriffe auf die einzelnen Strategieteile sind links neben dem abstrakten Interpreter beispielhaft erklärt. Die Funktionen push, pop und top sind Operationen für das Einfügen, Löschen und Abfragen im Keller. Aus Übersichtlichkeitsgründen wird im abstrakten Interpreter nur ein einfacher Keller betrachtet und kein Multikeller. Ein Ausrufezeichen kennzeichnet einen Test, ein Fragezeichen ein tiefes Ziel.

Beispiel der Interpreterfunktionen

P : H → G_1,G_2 | B_2,B_3;B_4 [g]

head(P) = H
guards(P) = G_1,G_2
prior(P) = g
body(P) = B_2,B_3;B_4
first(body(P)) = B_2,B_3
rest(body(P)) = B_4

While Agent **is** alive **do**
  If **Erg** $\neq \emptyset$ **then**
    ε = <e,Q> = S_ε (**Erg**)
    Erg = **Erg** \ {ε}
    R_ε = {pθ | p $\in$ **Strat** $\wedge$ θ ist relevanter Unifikator bzgl. ε} *// relevante Pläne*
    A_ε = {pφ | φ= $\theta\delta$ $\wedge$ pθ $\in R_\varepsilon$ $\wedge$ ($\theta\delta$) ist anwendbarer Unifikator bzgl. ε} *// anwendbare Pläne*
    P_ε = $S_p(A_\varepsilon)$
    If <e,Q> ist externes Ereignis **then**
      Int = **Int** $\cup$ { P_ε φ } *// neue Intention*
    else
      push($P_\varepsilon\varphi$, Q) *// auf existierende Intention packen*
    fi
  fi
  i_a=S_I(**Int**) *// aktuelle Intention*
  If i_a $\neq \emptyset$ **then**
    top_{ia}= pop(i_a) *// oberste Strategie*
    $topBody_{ia}$ = first(body(top_{ia})) *// Menge der obersten Body-Elemente*
    $\forall$ elem $\in$ $topBody_{ia}$ **do parallel**
      $topBody_{ia}$ = $topBody_{ia}$ \ { elem }
      case elem = a(t) *// Aktion*
        Act = **Act** $\cup$ { a(t) }
      case elem = m(t) *// Nachricht*
        Msg = **Msg** $\cup$ { m(t) }
      case elem = !g(t) *// Ereignis*
        Erg = **Erg** $\cup$ <+!g(t), i_a>
      case elem = ?g(t) *// Test*
        If $\exists$ g(s) $\in$ **Bel** mit g(s)θ = g(t)θ **then**
          top_{ia} = head(top_{ia})θ $\rightarrow$ guards(top_{ia})θ | $topBody_{ia}\theta$; rest(body(top_{ia}))
        else
          fail
        fi
      case elem = true *// Strategie komplett abgearbeitet*
        x = pop (top (i_a)) *// nächste Klausel im Stack instanziieren*
        top_{ia} = head(x)θ $\rightarrow$ guards(x)θ | body(x)θ [prior(x) * prior(top_{ia})] und
        θ ist Substitutionsergebnis der Reduktion von top_{ia}
    od
    process(**A**) *// bearbeite Aktionen*
    process(**M**) *// bearbeite Nachrichten*
    push(top_{ia}, i_a) *// aktuelle Klausel wieder auf Intentionsstapel legen*
  fi

Private Methoden der Agentenfamilie UrAgentGHC

Sektion	Methoden	Beschreibung
Dienste	addService(Dienst, Strategie, Nachricht), delService(Dienst)	Veröffentlichung bzw. Abmeldung eines Dienstes
	getAllServices()	Ermitteln einer Liste der vom Agenten angebotenen Dienste
Informationen	addInformation(Information, Belief, Nachricht), delInformation(Information)	Veröffentlichung bzw. Abmeldung einer Information
	getAllInformation()	Liefert eine Liste der vom Agenten veröffentlichten Informationen
Beliefs	addBelief(Belief), getBelief(Belief), delBelief(Belief)	Einfügen, Abfragen und Löschen eines Beliefs der Wissensbasis
	queryBeliefs(Muster)	Liefert das Ergebnis der Unifikation des angegebenen Musters mit der Menge der Beliefs
Strategien	createStrategy(Strategie), modifyStrategy(Strategie), deleteStrategy(Strategie)	Methoden zum Erzeugen, Modifizieren und Löschen von Strategien
	hideStrategy(Strategie), releaseStrategy(Strategie)	Methoden zur Steuerung der Abarbeitung von Strategien
Aktionen	addAction(Aktion), delAction(Aktion)	Erweitern und Löschen in der Menge der möglichen Aktionen des Agenten
	executeAction(Aktion)	Ausführen einer Aktion
	hideAction(Aktion), releaseAction(Aktion)	Sperren bzw. Entsperren der Ausführung einer Aktion
Nachrichten	inform(Sender, Empfänger, Inhalt, Status)	Versenden einer Nachricht ohne Bestätigung der Ankunft (asynchrones Senden)
	informRec(Sender, Empfänger, Inhalt, Status)	Versenden einer Nachricht mit späterer Bestätigung der Ankunft (synchrones Senden)
	send(Sender, Empfänger, Inhalt, Status)	Versenden einer Antwortnachricht (asynchron)
	sendRec(Sender, Empfänger, Inhalt, Antwort, Status)	Versenden einer Antwortnachricht mit Bestätigung der Ankunft (synchron)
Gruppen	createVGroup(Gruppe, Mitglieder), deleteVGroup(Gruppe)	Erzeugen bzw. Auflösen einer virtuellen Gruppe von Agenten
	insertInVGroup(Gruppe, Agent), deleteFromVGroup(Gruppe, Agent)	Einfügen bzw. Löschen eines Agenten in der angegebenen Gruppe
	memberInWhichVGroup(Agent)	Ermitteln aller virtuellen Gruppen, in denen der angegebene Agent Mitglied ist
	getAllMembersInVGroup(Gruppe)	Auflistung aller Mitglieder einer virtuellen Gruppe
	chgPermInVGroup(Gruppe, Agents, Permission)	Änderung der Zugriffsrechte in der virtuellen Gruppe

Kontextfreie Grammatik für CASA

Zeichen	Erklärung
(<expr>)*	Der Ausdruck <expr> kann beliebig oft hintereinander auftreten, oder auch gar nicht.
(<expr>)+	Der Ausdruck <expr> kann beliebig oft hintereinander auftreten, aber mindestens einmal.
|	Logisches Oder: Möglichkeit, für ein Nichtterminal eine von mehreren Ableitungen zu wählen.
(<expr 1> | <expr 2>)	Genau eine der beiden Ausdrücke <expr 1> und <expr 2> muss in der Ableitung auftreten, außer es folgt direkt ein * hinter der Klammer.
[<expr>]	Der Ausdruck <expr> darf höchstens einmal in der Ableitung auftreten.
[<expr>]^4	Der Ausdruck <expr> muss genau vier Mal in der Ableitung auftreten.
<expr 1> - <expr 2>	Kurzschreibweise für Aufzählungen von Ziffern und Buchstaben: '0' - '9' meint z.B. die Ziffern 0, 1, 2, 3, ..., 8 und 9.
string	Eine Zeichenkette ist eine Reihe von Symbolen (character) zwischen zwei Anführungszeichen. Dies ist in der kontextfreien Grammatik nur halbformal dargestellt.

Terminale: Implizit in den Grammatikregeln zwischen einfachen Hochzeichen angegeben
Nichtterminale: Implizit in den Grammatikregeln durch eckige Klammern kenntlich gemacht
Startsymbol: agent_declaration
Produktionen:

```
<agent_declaration>      ::=  'FUNCTIONS:'  <eventFunction>
                                            <planFunction>
                                            <intentionFunction>
                              'GOALS:' ( <top_level_goal> )*
                              'FACTS:' ( <fact> )*
                              'PLANS:' ( <plan> )*
<eventFunction>          ::=  'EVENTSELECT:' <identifier> ';'
<planFunction>           ::=  'PLANSELECT:' <identifier> ';'
<intentionFunction>      ::=  'INTENTIONSELECT:' <identifier> ';'
<top_level_goal>         ::=  'ACHIEVE' <relation> ':' doubleValue ';'
<fact>                   ::=  'FACT' <relation> ';'
<plan>                   ::=  '{' <plan_components> '}'
<plan_components>        ::=  ( <plan_component> )+
<plan_component>         ::=  <plan_name> [ <plan_description> ] <plan_goal>
                              <plan_type> [ <plan_precondition> ] [ <plan_priority> ]
                              <plan_body> [ <plan_failure> ]
<plan_name>              ::=  'NAME:' <string> ';'
<plan_description>       ::=  'DESCRIPTION:' [ <string> ';' ]
<plan_type>              ::=  'TYPE:' <type_name> ';'
```

```
<type_name>                  ::=  'REACTIVE'
                                | 'DELIBERATIVE'
                                | 'COMMUNICATIVE'
<plan_goal>                  ::=  'GOAL:' 'ACHIEVE' <relation> ';'
<plan_precondition>          ::=  'PRECONDITION:' ( <plan_condition_element> )*
<plan_condition_element>     ::=  <simple_condition_element>
                                | 'PARALLEL' ( <plan_condition_block> )+
<plan_condition_block>       ::=  '{' <simple_condition_elements> '}'
<simple_condition_elements>  ::=  ( <simple_condition_element> ';' )+
<simple_condition_element>   ::=  'TEST' <expression>
                                | 'GETFACT' <relation>
                                | 'GETVALUES' <relation>
                                | 'ACHIEVE' <relation> ':' <expression>
                                | 'REQUEST' <expression> <string> <explist> ':' <expression> ';'
                                  'GETREPLY' <explist>
<plan_priority>              ::=  'PRIORITY:' <doubleValue> ';'
<plan_body>                  ::=  'BODY:' [ <plan_body_elements> ]
<plan_block>                 ::=  '{' <plan_body_elements> '}'
<plan_simple_block>          ::=  '{' <plan_simple_body_elements> '}'
<plan_body_elements>         ::=  ( <plan_body_element> )+
<plan_simple_body_elements>  ::=  ( <plan_simple_body_element> )+
<plan_body_element>          ::=  <action> ';'
                                | 'WAIT' ':' <action> ';'
                                | 'WHILE' <action> <plan_block>
                                | 'IF' <action> <plan_block> [ 'ELSE' <plan_block> ]
                                | 'PARALLEL' ( <plan_simple_block> )+
<plan_simple_body_element>   ::=  <action> ';'
                                | 'WAIT' ':' <action> ';'
                                | 'WHILE' <action> <plan_simple_block
                                | 'IF' <action> <plan_simple_block [ 'ELSE' <plan_simple_block> ]
<plan_failure>               ::=  'FAILURE:' ( <plan_failure_element> )*
<plan_failure_element>       ::=  <fact_action> ';'
                                | <other_action> ';'
                                | <communicate_action> ';'
<action>                     ::=  <fact_action>
                                | <goal_action>
                                | <communicate_action>
                                | <other_action>
```

```
<fact_action>          ::=  'GETVALUES' <relation>
                          | 'GETFACT' <relation>
                          | 'ADD' <relation>
                          | 'DELETE' <relation>
                          | 'UPDATE' '(' <relation> ')' '(' <relation> ')'
<goal_action>          ::=  'ACHIEVE' <relation> ':' <expression>
<communicate_action>   ::=  'INFORM' <expression> <string> <explist>
                          | 'REQUEST' <expression> <string> <explist> ':' <expression> ';'
                            'GETREPLY' <explist>
                          | 'REPLY' <expression> <string> <explist>
<other_action>         ::=  'EXECUTE' <identifier> <explist>
                          | 'ASSIGN' <variable> <expression>
                          | 'TEST' <expression>
                          | 'FAIL'
<relation>             ::=  <identifier> <explist>
<expression>           ::=  <value>
                          | <variable>
                          | <function_call>
                          | <predicate>
<explist>              ::=  ( <expression> )*
<value>                ::=  <integer>
                          | <double>
                          | <string>
<doubleValue>          ::=  <integer>
                          | <double>
<function_call>        ::=  '(' ( <function> | <identifier> ) <explist> ')'
<predicate>            ::=  'EXISTS' '(' 'FACT' <relation> ')'
```

Literale:

```
<variable>             ::=  '$' ( '_' | <letter> | <digit> )+
<integer>              ::=  [ <sign> ] ( <digit> )+
<double>               ::=  [ <sign> ] ( <digit> )+ '.' ( <digit> )* [ <exponent> ]
                          | [ <sign> ] '.' ( <digit> )+ [ <exponent> ]
<function>             ::=  '+' | '-' | '*' | '/' | '==' | '!=' | '<' | '<=' | '>' | '>='
                          | '&&' | '||' | '!'
<identifier>           ::=  ( '_' | 'A' - 'Z' | 'a' - 'z') ( <letter> | <digit> )*
<letter>               ::=  '-' | '_' | 'A' - 'Z' | 'a' - 'z'
<digit>                ::=  '0' - '9'
<string>               ::=  " ( character )* "
<sign>                 ::=  '-' | '+'
<exponent>             ::=  [ 'e' | 'E' ] [ '+' | '-' ] ( <digit> )+
```

Laufzeitprotokoll: Einlagerung im Farbsortierspeicher

Nach Aufruf des Java-Interpreters mit der in Abschnitt 7.5.3. vorgestellten und in Java-Bytecode übersetzten Datei ColorSortingSystem.java werden u.a. die folgenden Zeilen auf dem Bildschirm ausgegeben. Die in eckigen Klammern angegebenen Punkte weisen jeweils auf die Stellen hin, an denen ich Zeilen der Originalausgabe weggelassen habe:

```
CasaParser: Parsing file orderAgent.spec and building agent components.
CasaParser: Program orderAgent.spec parsed successfully.
Starting to run agent OrderAgent.
[ ... ]
Starting to run agent DepotAgent.
[ ... ]
Starting to run agent Sortline-1.
[ ... ]
Starting to run agent Sortline-2.

CasaParser: Parsing file inRobot.spec and building agent components.
CasaParser: Program inRobot.spec parsed successfully.
Starting to run agent InRobot-1.

InRobot-1:    round 1
InRobot-1:    testing plans to get order
InRobot-1:    going to depot at position (10,10)
InRobot-1:    still moving to depot
OrderAgent:   answer is order (4,sedan,red) to InRobot-1
InRobot-1:    got the order (4,sedan,red) from OrderAgent
InRobot-1:    asking agent SortlineBroker to get sortline for color red
InRobot-1:    requesting a body of type sedan from DepotAgent
DepotAgent:   giving a body of type sedan to InRobot-1
Sl-Broker:    asking sortlines about their desire for color red
InRobot-1:    body of type sedan loaded
Sortline-2:   send desire (3) for color red to SortlineBroker
Sortline-1:   send desire (6) for color red to SortlineBroker
Sl-Broker:    got desire (3) from sortline 2
Sl-Broker:    got desire (6) from sortline 1
Sl-Broker:    accepting proposal of sortline 1
Sl-Broker:    informing InRobot-1 that it should go to sortline 1
InRobot-1:    got the reply to go to sortline 1
InRobot-1:    going to sortline 1 at position (20,10)
InRobot-1:    still moving to sortline 1
InRobot-1:    giving order (4,sedan,red) to sortline 1
Sortline-1:   received order (4,sedan,red) from InRobot-1
InRobot-1:    available now

InRobot-1:    round 2
InRobot-1:    testing plans to get order
InRobot-1:    going to depot at position (10,10)
```

```
InRobot-1:    still moving to depot
OrderAgent:   can‘t send order to InRobot-1 (no bodies of type sedan)
InRobot-1:    got the order (none,n,n) from OrderAgent
[ ... ]
InRobot-1:    round 16
InRobot-1:    testing plans to get order
OrderAgent:   answer is order (11,sedan,green) to InRobot-1
InRobot-1:    got the order (11,sedan,green) from OrderAgent
InRobot-1:    requesting a body of type sedan from DepotAgent
InRobot-1:    asking agent SortlineBroker to get sortline for color green
[ ... ]
InRobot-1:    got the reply to go to sortline 2
InRobot-1:    going to sortline 2 at position (20,20)
InRobot-1:    still moving to sortline 2
InRobot-1:    giving order (11,sedan,green) to sortline 2
Sortline-2:   received order (11,sedan,green) from InRobot-1
InRobot-1:    available now

InRobot-1:    round 17
InRobot-1:    testing plans to get order
InRobot-1:    going to depot at position (10,10)
InRobot-1:    still moving to depot
OrderAgent:   can‘t send order to InRobot-1 (no bodies of type sedan)
InRobot-1:    got the order (none,n,n) from OrderAgent

InRobot-1:    round 18
InRobot-1:    testing plans to get order
OrderAgent:   can‘t send order to InRobot-1 (no bodies of type sedan)
InRobot-1:    got the order (none,n,n) from OrderAgent

InRobot-1:    round 19
InRobot-1:    testing plans to get order
OrderAgent:   can‘t send order to InRobot-1 (no bodies of type sedan)
InRobot-1:    got the order (none,n,n) from OrderAgent
InRobot-1:    too many failed attempts to get an order, waiting for order
[ ... ]
InRobot-1:    still waiting for order
OrderAgent:   send order (12,sedan,green) to InRobot-1

InRobot-1:    round 20
InRobot-1:    testing plans to get order
InRobot-1:    taking the order (12,sedan,green)
InRobot-1:    asking agent SortlineBroker to get sortline for color green
InRobot-1:    requesting a body of type sedan from DepotAgent
[ ... ]
InRobot-1:    got the reply to go to sortline 2
InRobot-1:    going to sortline 2 at position (20,20)
InRobot-1:    still moving to sortline 2
InRobot-1:    giving order (12,sedan,green) to sortline 2
Sortline-2:   received order (12,sedan,green) from InRobot-1
InRobot-1:    available now
[ ... ]
```