Typisierung autonomer Softwareagenten

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Biologie und Informatik
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von
Michael Zapf
aus Friedberg (Hess.)

Frankfurt 2002
(D F 1)

Vom Fachbereich Biologie und Informatik der Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Bruno Streit
Gutachter:	Prof. Dr. Kurt Geihs Prof. Dr. Winfried Lamersdorf
Datum der Disputation:	08.10.2001

Inhaltsverzeichnis

Abbildungsverzeichnis

Danksagung

An dieser Stelle sei all jenen Menschen gedankt, welche mir persönlich und fachlich während der vergangenen Jahre zur Seite standen. Besonderer Dank geht an Prof. Dr. Kurt Geihs für die ausgezeichnete Betreuung meiner Forschungsbemühungen. Prof. Dr. Winfried Lamersdorf möchte ich für die Übernahme der Aufgabe des Zweitgutachters danken.

Mein Lob geht an Angelika Schifignano für diese faszinierend unkomplizierte Erledigung all jener komplizierten Verwaltungsangelegenheiten.

Das Agentensystem AMETAS wäre nicht das, was es heute ist, ohne die Mitwirkung meines Kollegen Klaus Herrmann, der mit mir gemeinsam dieses hervorragende System über die Jahre hinweg mit nicht ruhendem Enthusiasmus gepflegt und weiter entwickelt hat. Nicht unterschlagen werden sollen die Bemühungen all der Diplomanden, die mit zum Erfolg des Agentensystems beigetragen haben; insbesondere zu nennen Helge Müller und Ulf Jahr.

Ein besonderer Dank geht an meine Kollegen Christian Becker und Reza Farsi für die vielen gleichermaßen anregenden wie angenehmen Diskussionen.

Meinen Angehörigen sowie allen meinen Freunden schulde ich besonderen Dank für die persönliche Unterstützung während der vergangenen Jahre.

Verwendete Hilfsmittel

Die Arbeit wurde mittels des grafischen Textsatzsystems LyX auf Basis von LaTeX 2ε und dem KOMA-Script-Paket erstellt. Grafiken entstammen dem Programm *XFIG* sowie Clipart-Sammlungen. Die der Arbeit zu Grunde liegende Literatur ist dem Verzeichnis am Ende der Arbeit zu entnehmen.

Hinweis

In dieser Arbeit werden eingetragene Warenzeichen, Handelsnamen und Gebrauchsnamen sowie Firmenbezeichnungen verwendet. Auch wenn diese nicht als solche gekennzeichnet sind, gelten die entsprechenden Schutzbestimmungen.

1 Einleitung

In den letzten Jahren bildete sich mit den autonomen Softwareagenten ein viel versprechendes neues Betätigungsfeld der Informatik heraus. Die Entwicklung hin zu diesen Softwareagenten wurde maßgeblich von den Gebieten der Robotik und der Künstlichen Intelligenz vorangetrieben. Zunächst lag das Interesse vor allem daran, Maschinen zu konstruieren, welche den Menschen in gefährlichen oder unzugänglichen Umgebungen vertreten konnten. Populäre Beispiele für autonom agierende Roboter präsentiert die Raumfahrt, denn Raumsonden, welche in der Erforschung des Sonnensystems eine beträchtliche Distanz zurückzulegen haben, können aufgrund der Begrenztheit der Lichtgeschwindigkeit schon in der Nähe der nächsten Planeten Mars oder Venus nicht mehr in „Echtzeit“ gesteuert werden, da die Steuersignale minutenlang unterwegs sind und die entsprechende Rückmeldung ebenfalls diese Zeit benötigt. Sonden wie der *Sojourner* des Pathfinder-Programms [Sto96, NAS97] sind darauf angewiesen, ihren Weg auf möglicherweise unebenem Grund selbständig zu finden, ohne den Erfolg der Mission durch einen Unfall zu riskieren.

Das softwaretechnische Pendant zu diesen physischen, autonomen Maschinen stellen die so genannten *(Software-)Agenten* dar. Softwareagenten sollen in der Lage sein, einen Auftrag im Namen eines menschlichen Anwenders oder auch eines anderen Programms durchzuführen. Die Eigenschaften eines solchen Agenten wurden über die Jahre hinweg unterschiedlich beurteilt; je nach Betätigungsfeld legten Forscher unterschiedliches Gewicht auf verschiedene Aspekte.

Mit *Telescript* [Whi94] wurde Mitte der Neunzigerjahre von der Firma *General Magic* ein neuartiges Konzept eingeführt, bei dem Code zwischen vernetzten Rechnern ausgetauscht werden sollte. Die Vorstellung war, mit diesem Code einen bestimmten „Auftrag“ auf entfernten Rechnern zu verfolgen; der Anwender[1] sollte einen so genannten *mobilen Agenten* beauftragen, eine Aufgabe in seinem Namen zu erledigen. Der Agent wandert von einem Rechner zum nächsten und muss gegebenenfalls eigene Entscheidungen treffen, um seinen Auftrag zu erfüllen.

[1] Die in dieser Arbeit verwendeten Personenbezeichnungen lassen – sofern nicht explizit angegeben – keine Rückschlüsse auf das tatsächliche Geschlecht zu.

Problembeschreibung

Die Programmierung von Beispielanwendungen im Agentenumfeld folgt bislang bekannten Konzepten. Nach der Analyse eines Problems werden Strukturen identifiziert, welche zum Einsatz kommen sollen, was schließlich im Entwurf einer oder mehrerer Komponenten mündet. Dieses Vorgehen wird für jedes Projekt im Einzelnen durchgeführt, womit jede Anwendung für sich isoliert entworfen und in Betrieb genommen wird. Leider findet die so genannte *Wiederverwendung* von Code schon im Umfeld der objektorientierten Programmierung in nur begrenztem Ausmaße statt, was sich durch die Tatsache, dass die Agentenprogrammierung sich heutzutage der Objektorientierung bedient, auf diese überträgt. Die Fähigkeit von Agenten, zur Erfüllung ihres Auftrags eigenständig Entscheidungen zu treffen, die möglicherweise auch die Kommunikation mit zunächst unbekannten Partnern beinhaltet, wird durch die starre Festschreibung der Komponenten konterkariert.

Multiagentenanwendungen sind auf die Kooperation der Agenten untereinander angewiesen: Jeder angesprochene Agent muss die ihm übermittelten Daten eines anderen Agenten zu verarbeiten wissen. Bisher vertraute man darauf, dass der Entwurf einer agentenbasierten Anwendung diese Probleme umgeht, da der Entwurf *abgeschlossen* ist, es also stets eine genau definierte Menge von möglichen, interagierenden Komponenten gibt. Es stellt sich die Frage, ob durch eine solche Strategie nicht ein großes Potenzial des Einsatzes von Agenten verschenkt wird. Gerade im Hinblick auf künftige Implementierungen elektronischer Märkte erscheint es sinnvoll anzunehmen, dass Multiagentenanwendungen *offen* sind, dass Agenten fortwährend mit bislang unbekannten Partnern kooperieren müssen.

Um diese Beschränkungen zu lockern und den Weg zu offenen Anwendungen zu ebnen, kann das Konzept der *Typen* hilfreich sein. Die Vorteile der Verwendung typisierter Sprachen liegen auf der Hand: Zum einen kann der Compiler bereits vor der Ausführung eine Prüfung des Programms durchführen. Dazu wird sichergestellt, dass den Variablen nur typkonforme Instanzen zugewiesen sowie nur legale Operationen auf diesen Variablen ausgeführt werden. Zum anderen erlauben Klassen, welche die Rolle von Typen bei komplexen Datenstrukturen spielen, eine bequeme Wiederverwendung geschriebenen Codes durch das so genannte *Ableiten*, was dem Implementierer die Möglichkeit bietet, Anwendungen erweiterbar zu gestalten. Während des Ablaufs einer Anwendung wird die Verfügbarkeit eines Objekts einer bestimmten Klasse erwartet; an deren Stelle darf eine Erweiterung in Form einer abgeleiteten Klasse treten. Es ist somit möglich, durch Aggregation von Funktionalitäten eine Wiederverwendung des Codes zu erreichen; das Objekt präsentiert sich den unterschiedlichen Anfragern stets mit der von ihnen erwarteten Menge von Funktionalitäten.

Jedoch ist eine bloße Übertragung des Klassenkonzepts auf die Agentenwelt nicht uneingeschränkt sinnvoll: Einerseits können Agenteninteraktionen sehr viel komplexer aufgebaut sein als solche zwischen Objekten; die traditionellen Klient-/Serverrollen wechseln beständig, da in der Regel jeder der autonomen Agenten seinen eigenen Auf-

trag verfolgt. Andererseits muss die Unbeständigkeit von Agenten in Betracht gezogen werden, welche gerade eben neu in das System kommen und sogleich wieder verschwinden. Auf dieser Basis kann die übliche Zuordnung von Namen zu Typen nicht in allen Bereichen des Systems konsistent erhalten werden.

Ein weiteres Problem betrifft den Endanwender. Der bisherige Aktionsradius eines Endanwenders beschränkt sich auf die Bedienung einer Anwendung, die er zuvor erworben hat. Diese Anwendung legt fest, welche Aufgaben vom Anwender mit ihr gelöst werden können; dem Anwender obliegt es, seine Ziele entsprechend anzupassen.

Um für verschiedene Bereiche Agenten einsetzen zu können, würde es erforderlich sein, eine Reihe von verschiedenen Anwendungen zu erwerben. Die Agenten müssten äußerst flexibel gestaltet werden, um auf die Bedürfnisse des Anwenders anpassbar zu sein. Eine Gestaltung seitens des Anwenders ist kaum möglich, da die Gestaltung von Agenten erhebliche Programmierkenntnisse erfordern würde. Könnte der Anwender jedoch seine Vorstellungen vom *Einsatz* und *Verhalten* des Agenten hinreichend genau beschreiben, so ist es vorstellbar, dass ihm ein passender Agent etwa aus einem Katalog empfohlen werden kann. Es wäre damit erforderlich, Agenten mit einer Beschreibung auszustatten, welche geeignet automatisch ausgewertet werden könnte. Man kann in diesem Falle ebenfalls von einem *Typ* sprechen, welchem der gesuchte Agent angehören soll.

Diese gerade genannten Probleme können eine der Ursachen für die schleppende Weiterentwicklung der Agententechnologie darstellen, insbesondere für die bislang nicht erfolgreiche Suche nach der so genannten *Killer-Applikation*, also jene Anwendung, die überzeugend die Notwendigkeit autonomer Softwareagenten demonstriert. Es sollten Anstrengungen unternommen werden, die bisherigen Gepflogenheiten zu überprüfen und neue Paradigmen zu akzeptieren, möchte man die Vorzüge der *agentenorientierten Programmierung* erfahren.

Ziele der Arbeit

Diese Arbeit präsentiert einen Ansatz für die Typisierung autonomer Softwareagenten. Dabei sind folgende Ziele von besonderer Bedeutung:

1. Es muss zunächst eine genaue Begriffsdefinition stattfinden, um eine geeignete Definition des Begriffs „Agent“ zu erhalten. Welche Entitäten sind in dieser Arbeit als Agenten zu sehen und wie unterscheidet sich das Bild von den anderen Systemen zu Grunde liegenden Sichtweisen?

2. Zum anderen ist es wichtig, das Prinzip der Typisierung in verschiedenen Kontexten zu betrachten. Dabei soll klar werden, welche Bedeutung die Typisierung in hochgradig dynamischen Systemen hat. Was muss unter der Typisierung eines Agenten verstanden werden?

3. Die Grundlagen des entwickelten Typsystem müssen ein solides theoretisches Fundament aufweisen. Der Leser sollte in der Lage sein, die grundlegenden Ansätze nachzuvollziehen. Schnittpunkte mit ähnlichen Ansätzen sind vorzustellen, um eine Einordnung dieses Ansatzes zu erlauben.

4. Um zu demonstrieren, dass der hier vorgestellte Ansatz praktisch realisierbar ist, soll aufgezeigt werden, wie die Integration des Typsystem in ein bestehendes Agentensystem vorgenommen werden kann. Als Beispiel dient das dieser Arbeit zu Grunde liegende Agentensystem AMETAS, in welches dem Leser ein kurzer Einblick gewährt wird.

5. Neben der Integration in AMETAS ist die Fragestellung, welche Bedingungen ein Agentensystem erfüllen muss, um dieses Typsystem zu integrieren, von hoher Relevanz. Das Typsystem darf keine zu hohen Forderungen stellen, da die Integration des Typsystems in beliebige Agentensysteme das Ziel der Entwicklung sein muss.

6. Schließlich ist wichtig aufzuzeigen, wie das Typsystem zum Einsatz kommen kann. Dabei steht im Vordergrund, welche Auswirkungen die Verwendung des Typsystems auf die Erstellung Agenten-basierter Anwendungen hat. Muss der Entwurf eines Agenten auf die Verwendung des Typsystems abgestimmt werden?

Aufbau der Arbeit

Diese Arbeit ist wie folgt aufgebaut: In Kapitel zwei werden Grundlagen präsentiert, die für das Verständnis der darauf folgenden Kapitel erforderlich sind. Diese betreffen insbesondere den Agentenbegriff, welcher anhand zahlreicher Beispiele betrachtet wird, um einen Überblick über die zurzeit aktuelle Agentenforschung zu gewinnen.

Kapitel drei führt in die Grundlagen von Typen, Klassen und Schnittstellen ein. Die Bedeutung von Typen wird an einfachen Beispielen illustriert und in Verhältnis zu den aus objektorientierten Programmiersprachen bekannten Begriffen wie Klassen und Schnittstellen gesetzt.

Dem Agentensystem AMETAS, welches an der Professur für verteilte Systeme und Betriebssysteme der Universität Frankfurt entstand, ist das vierte Kapitel gewidmet. Dieses System entstand parallel zur Forschung an den Agententypen, was sich an zahlreichen Eigenschaften des Systems erkennen lässt. Es eignet sich daher in besonderer Weise für die Implementierung des vorzustellenden Agententypsystems.

Ein ausführliches Kapitel fünf legt die theoretischen Grundlagen des vorzustellenden Typsystems dar. Es werden zunächst Überlegungen angeführt, welche Gegebenheiten die Definition eines Typsystems beeinflussen. Dabei wird besonderes Augenmerk

auf die Art der Kommunikation zwischen Komponenten gelegt. Der so genannte Hybridtyp wird anschließend vorgestellt und auf theoretischer Basis untersucht.

Kapitel sechs bespricht die implementationsbezogenen Details. Dabei soll demonstriert werden, wie die im vorangegangenen Kapitel vorgestellten Konformitätsbedingungen praktisch umgesetzt wurden. Die Funktion des Mediators wird detailliert beschrieben, welcher das Kernstück der Typsystem-Unterstützung des Agentensystems bildet.

Kapitel sieben befasst sich mit der Anwendung des Typsystems und behandelt die möglichen Auswirkungen auf die Anwendungsprogrammierung. Die Anwendung des Typsystems wird dabei aus mehreren Perspektiven betrachtet: Die Sicht des Anwenders unterscheidet sich naturgemäß von jener des Entwicklers. Es werden anschließend verschiedene Entwurfsstrategien daraufhin betrachtet, welchen Einfluss sie auf die sichere Vermittlung von Agenten auf Basis der Hybridtypen haben.

In Kapitel acht wird ein Blick auf mögliche Erweiterungen des Typsystems geboten. Es werden Lösungsmöglichkeiten skizziert und eine Beurteilung ihrer Realisierbarkeit im Rahmen des bestehenden Typsystems beziehungsweise einer notwendigen Erweiterung durchgeführt. Schließlich werden die Ergebnisse dieser Arbeit in Kapitel neun übersichtlich zusammengefasst.

Eine Reihe von Anhängen bietet Details zu einigen Themen der vorangegangenen Kapitel; ein ausführliches Literaturverzeichnis ermöglicht dem Interessenten, sich über spezielle Themen weiter zu informieren.

2 Grundlagen

Die Beschäftigung mit autonomen Softwareagenten ist eine relativ neue Forschungsrichtung, auch wenn erste Schritte in diese Richtung bereits seit Jahrzehnten unternommen werden. In diesem Kapitel werden Grundlagen geliefert, auf denen die folgenden Kapitel aufbauen werden.

2.1 Objekte und Kommunikation

Die Agententechnologie profitierte in den letzten Jahren insbesondere durch die Verbreitung und die allgemeine Akzeptanz der objektorientierten Programmierung. Sprachen wie Java, welche dieses Paradigma integriert haben, sind beliebte Implementationssprachen für mobile Agenten.

2.1.1 Objekte und Aggregate

Es gibt zahlreiche Definitionen des Begriffs *Objekt*. Folgende Definition lehnt sich an der am häufigsten angeführten Charakterisierung von G. BOOCH [Boo91] an:

Definition 2.1 Objekt
Unter einem *Objekt* versteht man eine spezielle Datenstruktur, welche sich durch die folgenden Grundeigenschaften auszeichnet

- *Zustand*: Dieser setzt sich aus der Gesamtheit der diesem Objekt zugeordneten Speicherzellen zusammen.
- *Identität*: Ein Objekt besitzt eine eindeutig definierte, unverwechselbare Identität.
- *Verhalten*: Die Reaktion des Objekts auf Änderungen in der Umgebung sind innerhalb des Objekts definiert.

Neben diesen Merkmalen werden weitere Eigenschaften als *sekundäre Eigenschaften* angeführt, darunter die *Kapselung*, also das Verbergen der Implementierung und des Zustands des Objekts vor der Umgebung, und die *Vererbungshierarchie*, welche

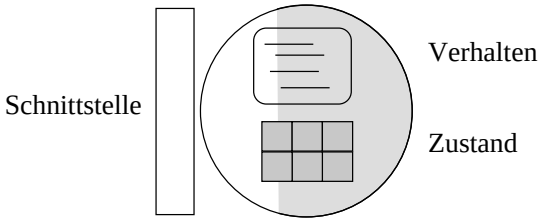

Abbildung 2.1: Schematische Darstellung eines Objekts

für die Möglichkeit steht, Eigenschaften von Objekten als von Elternobjekten *geerbt* darzustellen.

Abbildung 2.1 zeigt die bekannte, schematische Darstellung eines Objekts. Die Trennung zwischen Anfragern und der Implementierung des Objekts wird durch die *Schnittstelle* geleistet. Sie beschreibt, welche Funktionen von diesem Objekt angeboten werden. Der Klient besitzt im Idealfall nur über die Schnittstelle des Objekts Kenntnisse, während die dahinterstehende Implementierung unbekannt ist und gegebenenfalls auch ausgetauscht werden könnte [Grü97].

Nicht alle objektorientierten Sprachen sind in der Lage, für eine strikte Kapselung zu sorgen. *Simula* bietet beispielsweise beliebigen Zugriff auf Instanzvariablen. *C++* und *Java* hingegen erlauben durch die Markierung von Methoden und Feldern mit den Schlüsselwörtern *public, private* und *protected* in den Objektklassen eine Festlegung der Schnittstelle durch die *externe Sichtbarkeit* dieser Methoden und Felder. (Abbildung 2.1 symbolisiert die Verborgenheit privater Komponenten durch die dunkle, hintere Hälfte.) Java führt Schnittstellen als eigenständige, von Objekten und Klassen unabhängige Entitäten ein.

Diese Grundeigenschaften von Objekten bedingen besondere Vorkehrungen in der Programmiersprache und in den Programmen, weshalb man gerne auch von einem *Objektparadigma* spricht. Die Programmiersprache muss beispielsweise besondere Konstrukte anbieten, um solche Objekte zu generieren und ihre Eigenschaften durchzusetzen. Der Anwendungsprogrammierer muss das zu behandelnde Problem als eine Interaktion von Objekten modellieren.

Die erste objektorientierte Programmiersprache war *Simula 67* (1967), welche Objekte, Klassen, Vererbung und dynamische Typen einführte. Später folgte *Smalltalk,* aber die weite Verbreitung gelang erst mit der Sprache C++ (1985), welche ihre Beliebtheit der in der Systemprogrammierung weit verbreiteten Sprache C verdankt. Die Sprache Java vereinfachte einige der C++-typischen Konstrukte und findet seit Mitte der Neunzigerjahre des 20. Jahrhunderts Anwendung im *World Wide Web,* aufgrund der Plattformunabhängigkeit zunehmend auch in der Implementierung von systemunterstützenden oder infrastrukturellen Anwendungen *(Middleware).*

Prozeduren, wie sie aus dem traditionellen, prozeduralen Programmierparadigma bekannt sind, sind *zustandslos:* Die Abarbeitung der Unterroutine wird in der Regel nicht von vorangegangenen Aufrufen beeinflusst. Die objektorientierte Programmie-

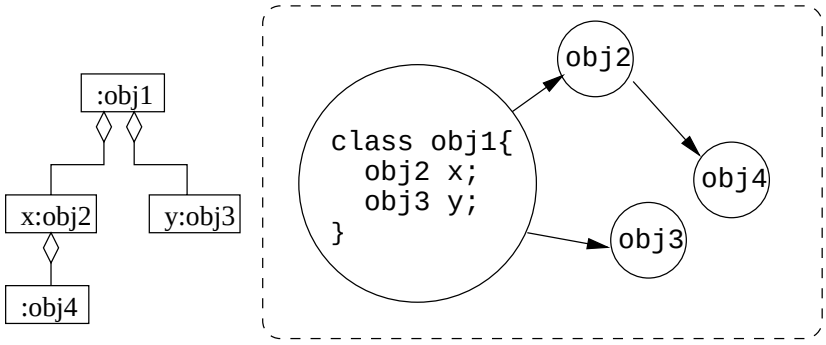

Abbildung 2.2: Aggregation von Objekten

rung führt mit den Objekten neue Konstrukte ein, deren wesentlicher Unterschied zu den bekannten Prozeduren ist, dass Objekte einen *Zustand* haben. Ein erneuter Aufruf einer zum Objekt gehörenden Prozedur kann nunmehr eine andere Wirkung haben als bei einem vorangegangenen Aufruf. Die einem Objekt zugehörigen Prozeduren werden *(Objekt-)Methoden* genannt; sie gleichen in ihrer Anwendung den Prozeduren, wobei der Aufrufer jedoch angeben muss, auf welches Objekt er sich bezieht.

Die Eigenschaft, eine *Identität* zu besitzen, ist eine Grundforderung an Objekte. Eine festgelegte Identität ist Grundvoraussetzung dafür, dass eine Komponente fortgesetzt mit demselben Objekt kommuniziert. Dies ist umso wichtiger, da ein Objekt einen Zustand besitzt und somit mit anderen Objekten, welche strukturell gleich sind, nicht ausgetauscht werden kann.

Mehrere Objekte können einen Verbund über *Aggregation* bilden. Abbildung 2.2 zeigt die Darstellung einer Aggregation in der Modellierungssprache *UML [OMG99];* rechts wird angedeutet, wie sich die Aggregation in der Praxis darstellt.[1] Anfrager referenzieren in diesem Beispiel gewöhnlich die *obj1*-Instanz, welche ihrerseits auf die übrigen Instanzen zugreift. Die Anfrager können, müssen aber nicht Referenzen auf die aggregierten Objekte besitzen. Die einzelnen Komponenten dieser Aggregation verfügen weiterhin über ihre eigene Identität.

Das *Verhalten* eines Objekts wird durch die Implementierung seiner Methoden festgelegt. Objekte kommunizieren mit der Umgebung durch die Annahme und Rückgabe von Werten durch Aufruf ihrer Methoden. Die Art und Weise der Verarbeitung der Daten repräsentiert demgemäß das Verhalten des Objekts.

2.1.2 Kommunikation

Im Paradigma der *prozeduralen Programmierung* stehen – wie der Name schon sagt – Prozeduren im Mittelpunkt. Eine Prozedur ist ein syntaktisch besonders gekennzeichnetes Fragment des Programmcodes, das von beliebigen Teilen des Programmcodes

[1] UML erlaubt auch weitere Varianten der Repräsentation von Objektfeldern. Siehe dazu [OMG99].

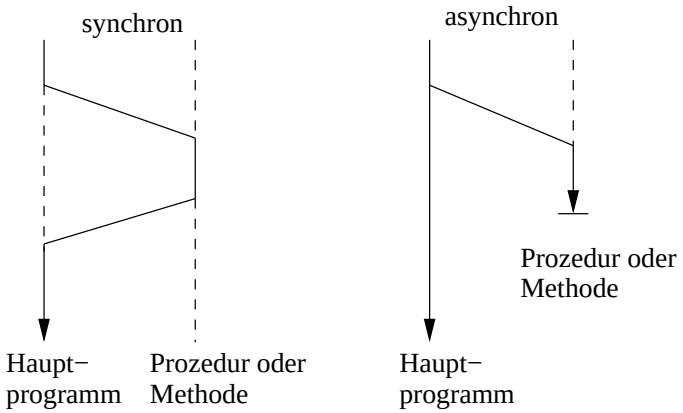

Abbildung 2.3: Synchroner und asynchroner Programmablauf

aufgerufen werden kann und die Programmkontrolle nach seiner Abarbeitung wieder zum aufrufenden Teil zurückgibt, welcher mit der nächsten Anweisung nach dem Aufruf fortfährt. Ein wichtiges Merkmal einer Prozedur ist die Möglichkeit, Parameter an sie zu übergeben.

Im Umfeld der verteilten Systeme wurde das Prozeduraufrufparadigma erweitert, sodass Prozeduren auf Rechnern angeboten werden können, die von einem Programm auf einem anderen Rechner aufgerufen werden. Dieses Verfahren wird *Remote Procedure Call* [Sri95] genannt. Die entfernten Prozedurserver kapseln den Zustand jedoch nicht wie Objekte; mittels so genannter *Context handles* kann eine Wiederherstellung eines früheren Zustands erreicht werden.

Prinzipiell kann ein Prozeduraufruf wie eine *Kommunikation* zwischen Hauptprogramm und der Unterroutine angesehen werden. Das objektorientierte Programmierparadigma suggeriert durch die Modellierung von Komponenten als abgeschlossene Einheiten die Vorstellung kommunizierender Entitäten. In C++ oder Java findet diese Kommunikation in Form von *Methodenaufrufen* statt, welche sich lediglich durch die Angabe des referenzierten Objekts von den traditionellen Prozeduraufrufen unterscheiden.

Prozedur- und Methodenaufrufe haben üblicherweise *Synchronität* gemein. Das heißt, dass der Programmablauf des aufrufenden Programms so lange unterbrochen ist, bis die Prozedur oder Methode beendet ist (siehe Abbildung 2.3). Durch den Unterprogrammaufruf gibt das Hauptprogramm die Kontrolle ab; es wird für den in der Abbildung angedeuteten Zeitraum nicht weiter ausgeführt. Im Falle eines asynchronen Aufrufs wird das aufrufende Programm weiterhin ausgeführt, während das zweite, aufgerufene Programm *parallel* zum ersten läuft und sich zu irgendeinem Zeitpunkt selbst beendet. Die Vorstellung der *Nebenläufigkeit* ist somit eng mit der Asynchronität verbunden [And91].

Der Begriff Synchronität stammt daher, dass der Aufrufer annimmt, dass keine Zeit vergeht, während die Anfrage vom Aufgerufenen verarbeitet wird; in dieser Zeit ist

der Aufrufer angehalten, kann also keine Aktionen durchführen. Im asynchronen Fall hingegen können weitere Ereignisse nach dem Aufruf stattfinden, ohne dass die Verarbeitung zum Abschluss gebracht worden ist [Mat89]. Die synchrone Verarbeitung ist bei den verbreiteten Programmiersprachen als Standard gegeben; um eine asynchrone Verarbeitung zu erreichen, muss durch eigene Kontrollflüsse (Threads) eine Nebenläufigkeit erzeugt werden.

Bei einer zweiseitigen Kommunikation können die beiden beteiligten Instanzen als *Klient* und *Server*[2] unterschieden werden:

- Ein Klient ist jene Komponente einer Kommunikation zwischen zwei Parteien, welche die Verbindung zum Gegenüber aktiv aufbaut, diesem Daten zusendet und gegebenenfalls Ergebnisse übernimmt.
- Ein Server ist jene Komponente, welche die Kommunikation zwischen zwei Komponenten passiv aufbaut, vom Gegenüber Daten übernimmt und diesem gegebenenfalls Ergebnisse zuschickt.

Man beachte, dass die Eigenschaft, Klient oder Server zu sein, von der jeweiligen Art der Kommunikationsaufnahme abhängt. Man spricht daher bevorzugt von der *Klientenrolle* und der *Serverrolle.* Ein Objekt, das bislang als Server auftrat, kann ebenso gut als Klient eines anderen Servers auftreten, um den Dienst zu verrichten, den es dem Klienten anbietet.

Der Betrieb von eigenen Serverprozessen zur Entgegennahme und Verarbeitung von Prozedur- oder Methodenaufrufen ermöglicht den Betrieb des Dienstes außerhalb des Klientenprogramms in einem eigenen Adressraum. So können Klienten und Server sogar auf verschiedenen Rechnern eines Netzwerks betrieben werden; die Anfragen des Klienten müssen dann über eine Infrastruktur wie DCE [OG97] oder CORBA [OMG00a] an den entsprechenden Server geleitet werden.

Auch im verteilten Falle wird zwischen einem *synchronen Aufruf* und einem *asynchronen Aufruf* unterschieden; in ersterem Falle wird der Klient so lange angehalten, bis der Aufruf durch den Server vollständig abgearbeitet ist, was das Entgegennehmen, das Verarbeiten sowie die Rücksendung eines Ergebnisses umfasst. In letzterem Falle fährt der Klient mit der Verarbeitung fort, ohne Informationen über die erfolgreiche Zustellung und Verarbeitung zu bekommen. Eventuell vorhandene Ergebnisse müssen zwischengespeichert werden, bis der Klient diese Ergebnisse abholt.

2.1.3 Nachrichten

Die Kommunikation zwischen verschiedenen Komponenten einer Anwendung kann durch gemeinsamen Speicher *(Shared memory)* organisiert werden [Tan92]. In Pro-

[2] Während das deutsche Wort *Klient* unmissverständlich auf das englische Wort *Client* verweist, existiert keine allgemein akzeptierte, unzweideutige Übersetzung von *Server*.

grammiersprachen wie C++ wird dies durch die Einführung globaler Variablen geleistet. Der Zugriff auf gemeinsamen Speicher durch unabhängige Komponenten muss jedoch geregelt werden, um Inkonsistenzen zu vermeiden; dabei finden Verfahren des gegenseitigen Ausschlusses Anwendung (etwa Monitore oder Semaphore). Die Zustandänderungen sind stets unmittelbar und werden nicht gepuffert. Ferner eignet sich ein solcher Austausch nur für Anwendungskomponenten im gleichen Adressraum.

Eine leistungsfähigere Alternative ist der Austausch von Nachrichten *(Message passing)*. Eine Nachricht ist eine als Einheit begriffene Zusammenstellung von Informationen, welche in einer vorgegebenen Weise strukturiert sind und als Ganzes zwischen den Komponenten einer Anwendung verschickt werden können. Der Sender nimmt dabei an, dass der Empfänger in der Lage ist, die der Nachricht zu Grunde liegende Struktur zu erkennen, um den Inhalt der Nachricht in einer definierten Weise zu verarbeiten. Prozeduraufrufe sowie Aufrufe von Objektmethoden können als ein Austausch zweier Nachrichten verstanden werden, nämlich

- einer Eingangsnachricht, welche die Parameter beinhaltet, die der Methode zur Verarbeitung übermittelt werden sowie
- einer Ausgangsnachricht, welche die Parameter beinhaltet, welche die Methode als Ergebnis an den Anfrager liefern soll.

Nachrichten werden über *Kanäle* transportiert; das empfangende Objekt kann anhand des Kanals, über welchen die Nachricht eintrifft, entscheiden, in welcher Art diese Nachricht zu verarbeiten ist. Dieses eher abstrakte Bild konkretisiert sich bei Objekten einer objektorientierten Sprache in der Art, dass den verschiedenen Kanälen die Methoden entsprechen, welche sich auf dem Objekt aufrufen lassen. Kanäle werden dabei mit Namen versehen, welche in diesem Bild den Methodennamen entsprechen. Die Kapazität des Kanals bestimmt wesentlich die Kommunikationsart [And91]:

- Kann der Kanal beliebig viele Nachrichten aufnehmen, so ist eine asynchrone Kommunikation möglich.
- Kann der Kanal nur höchstens eine Nachricht aufnehmen, welche durch den Empfänger quittiert wird, so kann man eine synchrone Kommunikation erreichen.

In der Praxis kann ein Kanal nicht unbegrenzt viele Nachrichten aufnehmen, sodass jede asynchrone Nachrichtenübermittlung synchronisiert wird, wenn kein Platz im Kanal mehr vorhanden ist.

Methoden und Prozeduren abstrahieren von den Vorgängen des Nachrichtenaustauschs; sie verbergen den eigentlichen Mechanismus. Im Falle lokaler Kommunikation sind Methodenaufrufe sehr ökonomisch in Bezug auf den Verwaltungsaufwand. Zwischen entfernt liegenden Objekten sind Methodenaufrufe nicht mehr möglich; es

können Hilfskonstrukte zur Verfügung gestellt werden, die die Kommunikation wie einen Methodenaufruf aussehen lassen, tatsächlich aber eine entfernte Kommunikation durchführen.

Die Verwendung von Nachrichten kann *explizit* realisiert werden. Zum Abschicken einer Nachricht sieht das Betriebssystem oder die Verteilungsplattform einen speziellen Befehl vor (beispielsweise *send*), welcher im synchronen Falle so lange blockiert, bis der Absender eine Bestätigung erhält, dass die Nachricht beim Empfänger angekommen ist. Der Befehl zur Entgegennahme von Nachrichten (etwa *receive*) blockiert so lange, bis eine Nachricht in der Eingangswarteschlange auftaucht. Wird *send* nicht-blockierend implementiert, so fährt der Absender mit der Verarbeitung fort, ohne Kenntnis über den Verbleib der abgesendeten Nachricht zu erlangen. In diesem Falle liegt eine asynchrone Kommunikation vor. Ein synchroner Methodenaufruf, wie er oben beschrieben wurde, lässt sich demnach durch die Hintereinanderausführung eines blockierenden *send*- und eines *receive*-Befehls erreichen.

Anstatt mehrere Kanäle anzunehmen, entlang derer die Nachrichten ihren Bestimmungsort erreichen, ist es auch möglich, nur einen Kanal vorzusehen, die Nachrichten aber entsprechend zu erweitern, sodass der Empfänger die Nachrichten selbständig den geeigneten Verarbeitungskomponenten zuleiten kann. Solche Mechanismen werden *Dispatcher* genannt und sind Bestandteil der meisten Architekturen auf Basis verteilter Komponenten.

2.2 Softwareagenten

Es gibt unzählige Definitionen von Softwareagenten; praktisch jede Forschergruppe hat im Laufe ihrer Forschung eine spezifische Sichtweise entwickelt. Eine der weithin akzeptierten Definitionen stammt von M. WOOLDRIDGE [Woo97]:

> *An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.*

Für die in dieser Arbeit vorgestellten Belange möge folgende Definition gelten, die sich weitestgehend an der wooldridgeschen Definition orientiert:

Definition 2.2 Softwareagent
Unter einem Softwareagenten versteht man eine Software-Entität, welche als abgeschlossene Einheit verstanden wird und während ihres Lebenszyklus eine eindeutige Identität besitzt. Sie ist in der Lage, mit äußeren Entitäten (ihrer *Umgebung*) zu interagieren, indem sie Informationen von diesen bezieht oder an diese weitergibt. Sie ist so gestaltet, dass sie in Folge der Interaktionen mit ihrer Umgebung an dieser Umgebung selbständig Modifikationen durchführt, welche als Ergebnis eines in ihrem Verhalten definierten Plans betrachtet werden können.

Der *Lebenszyklus* eines Agenten umfasst die Entstehung, die Interaktion mit der Umgebung sowie innere Zustandsänderungen, mögliche Verlagerungen auf andere Lokationen und die Terminierung. Die *Abgeschlossenheit* bezieht sich darauf, dass es stets eine gewisse Menge von Objekten geben muss, welche als *zum Agenten gehörig* identifiziert werden können. Agenten sind kein einzelner Bestandteil einer sie umfassenden Menge von Objekten, sondern können ohne diese bestehen und ausgeführt werden. Mit der *Selbständigkeit* verbindet sich die Vorstellung, dass der Agent keine einfache Weiterleitung von Aktionen des Anwenders vornimmt, sondern Aktionen von sich aus vornimmt, welche vom Anwender initiiert, aber nicht im Einzelnen bekannt sein müssen. Schließlich wird das gesamte Handeln des Agenten als *Plan* modelliert; der Agent soll eine bestimmte Aufgabe erfüllen.

Dieses Modell induziert eine Analogie zum menschlichen Verstand, der eigenständig Schlüsse aus den ihm im Gedächtnis zur Verfügung stehenden Informationen ziehen kann. Entsprechend benötigt der Agent – wie der Verstand – Verbindungspunkte zur Umgebung, mit deren Hilfe er Informationen gewinnen und Manipulationen durchführen kann; diese Vorrichtungen werden *Sensoren* und *Aktoren* genannt.

Diese Charakterisierung eines Agenten ist, wie viele andere, nicht so präzise und prägnant, wie man es von vielen Definitionen aus der Informatik gewohnt ist. So bildeten sich während der letzten Jahre zahlreiche Sichtweisen heraus, wann man einen Agenten, wann ein Objekt und wann ein Programm vorliegen hat. Der Grund für diese Unterschiede in der Sichtweise liegt darin, dass bereits der Begriff *Agent* an sich wenig aussagekräftig ist. Aus dem Lateinischen stammend – *agens* für *handelnd* – trifft dieser Begriff recht viele Strukturen, gerade in der Informatik. Jedes Programm, das ausgeführt wird, führt eine Reihe von festgelegten Operationen durch. Andererseits muss *jede Art von Programmcode* durch einen Prozessor ausgeführt werden; auch ein Agent kann sich nicht selbst ausführen. Viele Forscher glauben daher, dass eine exakte Festlegung des Begriffs des Agenten in ebenso weiter Ferne liegt wie eine Definition für *künstliche Intelligenz* [Nwa96]. Der Begriff *Agent* ist kein originärer Begriff der Informatik wie etwa *Fuzzy Logic,* sondern taucht in der Alltagssprache allgegenwärtig auf. Eine interessante Diskussion findet man in [FG97]; auch die OMG (Object Management Group) hat mit der MASIF-Spezifikation eine eigene Sichtweise [MBB+98]. Einen Agenten anhand seiner Kommunikationsformen festzulegen ist, was nicht verwundert, ein Bild, das im Umfeld der Erforschung von Agentenkommunikationssprachen verbreitet ist [FW91].

2.2.1 Typologie von Softwareagenten

Softwareagenten tauchten zuerst im Kontext von *Multi-Agenten-Systemen (MAS)* auf, welche eines der drei großen Hauptgebiete der Forschung an der *verteilten künstlichen Intelligenz* (VKI) darstellt, neben *verteiltem Problemlösen* und *paralleler KI*. Carl Hewitt prägte 1977 das Konzept des *Actors* in seinem *Actor-Modell* [Hew77]. Ein Actor ist ein in sich abgeschlossenes, interaktives und parallel abgearbeitetes Objekt. Es be-

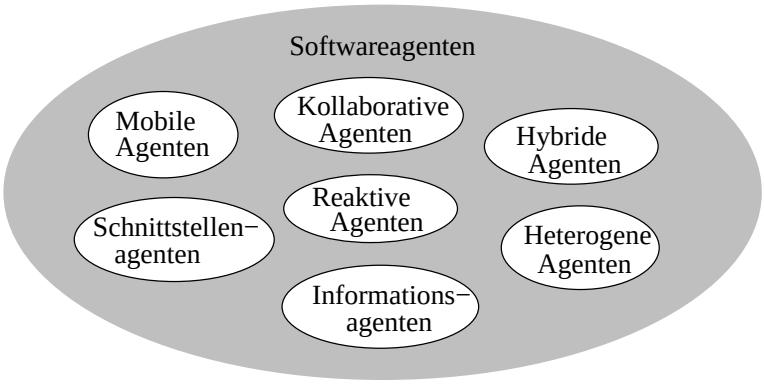

Abbildung 2.4: Typologie von Softwareagenten

sitzt einen internen Zustand und reagiert auf Nachrichten gleichartiger Objekte.

Die VKI-Forschung brachte Ende der Siebziger- und Anfang der Achtzigerjahre den Begriff des *deliberativen Agenten* hervor, welcher einen bestimmten Auftrag zu erfüllen versucht. Dazu bedient er sich eines *symbolischen Modells der Welt* und stützt seine Aktionen auf *symbolische Schlussweisen (symbolic reasoning)* [Woo95]. Der Agent „kennt“ das Ziel seines Handels und konstruiert einen geeigneten Handlungsplan, um das Ziel zu erreichen *(Intentionalität)*. Diese Sichtweise ist nach wie vor grundlegend für die Betrachtung eines Agenten im VKI-Umfeld. Wooldridge fasst in [WJ95] einige Grundeigenschaften von Agenten zusammen: Der Agent soll *autonom* handeln (ohne menschliche Intervention), *sozial* (in Interaktion mit anderen Agenten treten können), *reaktiv* (äußere Einflüsse beachten) und *proaktiv* sein (Ziele planmäßig verfolgen).

Es bildeten sich fortan zahlreiche Spezialisierungen des Begriffs *Agent* heraus, welche in Abbildung 2.4 zu sehen sind (nach [Nwa96]):

Kollaborative Agenten beruhen einerseits auf der Wahrung ihrer Autonomie, arbeiten aber mit anderen Agenten zusammen, unterstützen also Kooperation. Sehr viele Ansätze zählen sich zu dieser Klasse von Agenten.

Schnittstellenagenten sind durch Autonomie und Lernfähigkeit gekennzeichnet. Sie können insbesondere zur Unterstützung von Anwendern bei der Bedienung von Programmen oder bei deren Weiterbildung helfen. Viele Arbeiten wurden von *Pattie Maes* in diesem Gebiet durchgeführt; siehe auch [Mae94, Mae95].

Mobile Agenten sind Agenten, welche ihre Ausführung an einen anderen Ort verlagern können, also von Rechner zu Rechner wandern können. Der Agentenbegriff wird gerade in diesem Umfeld extrem ausgeweitet, da meist auf andere Agenteneigenschaften völlig verzichtet wird. Agenten werden häufig mit *mobilen Objekten* gleichgesetzt. Ein erster Vertreter mobiler Agentensysteme war General Magic mit *Telescript* [Whi94].

Informationsagenten dienen dem Zusammentragen von Informationen, die sie üblicherweise in sehr großen Netzen bis zum gesamten Internet vorfinden. Es gibt Überschneidungen mit dem Begriff der kollaborativen Agenten und den Schnittstellenagenten. Ein Beispiel für diese Art von Agenten wurde von ETZIONI und WELD vorgestellt, der *Internet Softbot* [EW94].

Reaktive Agenten verarbeiten Informationen, welche sie von der Umgebung erhalten *(Stimuli)* und antworten durch spezielle *Aktionen.* Sie müssen jedoch dazu kein symbolisiertes Modell der Welt zu Grunde legen. Der Fokus liegt in möglichst einfach gestalteten Agenten, welche im Verein mit anderen reaktiven Agenten zu einem möglicherweise komplexen Gesamtverhalten des Systems beitragen [Bro86, Bro91].

Hybride Agenten vereinigen mehrere der bislang genannten Charakteristiken, so zum Beispiel Reaktivität und Intentionalität [Mae91], aber auch Mobilität und Schnittstellenunterstützung.

Heterogene Agentensysteme beheimaten Agenten aus verschiedenen Klassen. Besonderes Augenmerk liegt auf der gegenseitigen Kommunikativität, weshalb eine einheitliche Kommunikationssprache sinnvoll ist. Vorschläge zu diesem Ansatz liefern GENESERETH und KETCHPEL in [GK94], indem sie *KQML* als Agentenkommunikationssprache (*Agent communication language,* ACL) propagieren. Eine andere verbreitete ACL ist FIPA ACL [TOH00, FIP00].

Eine weithin bekannte Entwurfsstrategie für Softwareagenten verbirgt sich hinter dem Kürzel BDI, welches für *Beliefs (Glaube), Desires (Wünsche), Intentions (Absichten)* steht. Die Modellierung orientiert sich an philosophischen Erkenntnissen über menschliche Schlussweisen und versucht, auf deren Basis die Implementierung von Agenten zu stützen [BIP88]. Das BDI-Modell wird allerdings in der aktuellen Forschung als nicht mehr zeitgemäß angesehen [GPP+98]; das *PECS*-Modell (für *Physis, Emotion, Cognition, Status*) ist einer der neueren Vorschläge, welcher sich aktuelle Erkenntnisse der Kognitionswissenschaften zu Nutze macht [Urb00]. Bemerkenswert ist die Hinzunahme des Aspekts der *Emotionen,* welche von einigen Wissenschaftlern als Schlüsselbegriff für das Verstehen der menschlichen wie der künstlichen Intelligenz verstanden werden. In [RTL00] wird beispielsweise eine Agentenarchitektur *EMA (emotionally motivated agent)* vorgestellt, welche demonstriert, wie „Gefühle“ wie Scham, Schuld, Stolz, Fröhlichkeit oder Langeweile zur Gestaltung eines Handlungsplans herangezogen werden.

2.2.2 Mobile Agenten

Mit dem wachsenden Grad der Vernetzung zwischen den Rechnern wuchs das Interesse an der Erforschung von Systemstrukturen, welche über mehrere Rechner verteilt vorliegen, aber zusammenarbeiten sollen. Gerade die Verbreitung der Programmiersprache

Java sorgte für einen Aufschwung in diesem Bereich. Mitte der Neunzigerjahre wurden so genannte *mobile Agenten* als neue Softwarestruktur eingeführt, welche von einem Rechner zum nächsten reisen können. Die Idee ist, die Verarbeitungslogik zu den Datenquellen zu bewegen anstatt die Daten der Datenquelle über das Netzwerk zu transportieren, was zu einem Verbrauch von Bandbreite führen würde. So können mobile Agenten besonders nutzbringend im Netzmanagement eingesetzt werden, um die Netzlast zu minimieren, welche durch Kontrollnachrichten der Standardnetzmanagement-Protokolle entstehen [ZHG99a, ZHG99b].

Der Einsatz mobilen Codes lässt sich nach [FPV98] in drei grundlegende Kategorien einteilen:

Code On Demand Der Nachfrager verfügt über die Ausführungsressourcen, aber nicht über den auszuführenden Code. Gewöhnlich wird dieser von einem Server heruntergeladen. Ein Beispiel für diese Technik sind die *Java-Applets.*

Remote Evaluation Der Nachfrager verfügt über den auszuführenden Code, aber nicht über die Ausführungsressourcen. Er transferiert diesen Code zu einem Server zum Zwecke der Ausführung. Dies findet sich etwa im Unix-Umfeld bei entfernter Shellskript-Ausführung oder bei der Auswertung von SQL-Datenbankabfragen.

Mobile Agents Im Prinzip liegt eine ähnliche Situation wie bei der entfernten Ausführung vor, jedoch verfügen Agenten über einen eigenen Zustand und die Fähigkeit zu autonomem Handeln.

Die Notwendigkeit autonomen Handelns ergibt sich für mobile Agenten daraus, dass sie auf einem anderen Rechner ausgeführt werden und der Benutzer sich vom Netzwerk abkoppeln möchte, während der Agent seine Aufgabe verrichtet. Mögliche Anwendungen lassen sich leicht im Bereich der Telekommunikation finden [KRR98].

Zentraler Bestandteil der meisten Agentendefinitionen im Bereich der Erforschung mobiler Agenten ist die *Autonomie*. Ein mobiler Agent sollte demnach in der Lage sein, eigenständig einen Auftrag auszuführen, der ihm von einem Auftraggeber – dies kann ein Anwender, aber auch ein Programm sein – gegeben wurde, wobei er gegebenenfalls den Ausführungsort wechseln kann. Die weiteren Eigenschaften wie geringer Bandbreitenverbrauch oder intelligentes Verhalten sind dabei weitere, mögliche Attribute. Die Forschung im Bereich der mobilen Agenten hat hierzu zahlreiche, unterschiedliche Ansätze hervorgebracht:

- **Systemnahe, mobile Komponenten**, welche als dem Betriebssystem zugehörig betrachtet werden (so genannte *Messengers*, [BFD96, FBDM98, WBD+98]). Diese einfachen Agenten können in einem System bei Bedarf verbreitet werden, verursachen nur geringe Netz- und Systemlast und erweitern die Funktionalität der Lokationen, an denen sie installiert werden.

- Abgeschlossene, eigenständige **Komponenten einer Anwendung,** welche sich in das bekannte Objektmodell einordnen lassen, meist unter Hinzunahme des Mobilitätsbegriffs. Hier finden sich zahlreiche Systeme, etwa Mole [BHR+97], Aglets [LO98, AO98], Voyager [SSD98, Obj01] und auch AMETAS, welches die Grundlage des in dieser Arbeit vorgestellten Typsystems liefert (siehe Kapitel 4.1).

- Agenten als Pakete eines **WWW-basierten Systems** von entsprechend erweiterten HTTP-Servern wie *ffMain* [LD98] oder *Wasp* [Fün97, Fün98]. Dabei wird ausgenutzt, dass einerseits die Infrastruktur für http-basierte Kommunikation bekannt und verbreitet ist sowie Anwender geübt im Umgang mit Webbrowsern sind, welche ihnen die Kommunikation mit den Agenten ermöglichen.

- **Theoretische Ansätze** wie *Mobile Ambients* [CG98, CG99], welche sich als Erweiterung des traditionellen Datentypbegriffs verstehen.

Alle Systeme unterstützen in unterschiedlichem Ausmaße das Konzept der Autonomie. Agenten sollen Aufträge im Namen eines Auftraggebers durchführen. Dabei soll die Erfüllung der Aufgabe von der Interaktion mit dem Auftraggeber entkoppelt stattfinden. Die Begriffe *Sensoren* und *Aktoren*, welche aus der VKI-Forschung stammen, lassen sich im Bereich der verteilten Systeme als Interaktion zwischen Agenten, Benutzern und Systemkomponenten wiederfinden.

2.2.3 Konsistenz und Migration

Die für eine Zusammenarbeit erforderliche Kommunikation unter Agenten wird recht unterschiedlich organisiert. Verbreitete Ansätze sind die aus den traditionellen Programmierparadigmen bekannten *Prozedur-* oder *Methodenaufrufe*, der Datenaustausch über gemeinsamen Speicher oder die Übertragung von Nachrichten *(Message passing)*, welche asynchron geschehen kann. Neuartige Konsistenzprobleme treten jedoch auf, wenn die untereinander kommunizierenden Partner mobil sind. Dies tritt insbesondere ein, wenn ein Objekt von zwei mobilen Agenten gemeinsam referenziert wird und einer der Agenten migriert, also seinen Ausführungsort wechselt. Je nach Implementierung verliert einer der Agenten die Referenz auf dieses Objekt oder jeder Agent erhält eine eigene Kopie des Objekts. In ersterem Falle kann die Verarbeitung durch den die Referenz verlierenden Agenten nicht fortgesetzt werden, es kommt zu Fehlern. In letzterem Falle werden Daten repliziert, geraten also bei fortwährender Modifikation in Inkonsistenz. Die Vermeidung solcher Anomalien kann durch Verzicht auf die Propagation von Objektreferenzen oder aufwändige Mechanismen zum Abgleich von Inkonsistenzen gelingen.

Grundsätzlich unterscheidet man zwei Arten der Migration, die *schwache* und die *starke Migration.* Die schwache Migration bietet dem Agenten die Möglichkeit, seinen

Ausführungsort zu wechseln, erzwingt aber einen Neustart des Agenten an einer festen Position in seinem Code. Die starke Migration hingegen erlaubt es dem Agenten, nach der Migration an jener Stelle im Code fortzufahren, an der seine Ausführung unterbrochen wurde. Während die schwache Migration insbesondere in Java problemlos zu implementieren ist, da Java die Rettung und Wiederherstellung des Objektzustandes erlaubt, erfordert die starke Migration spezielle Vorkehrungen im ausführenden System (bei Java in der virtuellen Maschine), um den Ausführungszustand wieder herzustellen. In [Fün98] und [IWKK00] werden mögliche Ansätze aufgezeigt, um eine starke Migration zu erreichen oder zumindest zu approximieren.

Eine weitere Charakterisierung der Mobilität kann durch die Begriffe *aktive* und *passive Mobilität* getroffen werden [Zap97]. Bei aktiver Mobilität wird die Migration durch den Agenten selbst bewirkt; der die Migration auslösende Befehl ist Teil des Agentencode. Diese Art der Mobilität passt zum Bild des Agenten als einer autonomen Instanz, welche einen gegebenen Plan verfolgt und zu diesem Zwecke Migrationen durchzuführen hat. Passive Migrationen hingegen geschehen ohne Absicht des Agenten. Der Agent wird während der Ausführung seines Auftrags an eine andere Lokation transportiert.

Offenbar erfordert die passive Mobilität die Bereitstellung einer starken Migration, da der Agent im ungünstigen Falle inmitten einer Aktion unterbrochen und von neuem gestartet wird. Die Vorkehrungen im Code, diesen Fall bei schwacher Migration zu behandeln, erfordern hohen Aufwand. Passive Mobilität kann erwünscht sein, um eine Lastbalancierung zwischen verschiedenen Lokationen zu erreichen. Jedoch ist dieses Mobilitätskonzept nur schwer in der Modellierung eines autonomen Agenten zu repräsentieren; der Agent müsste jederzeit mit einem Wechsel seiner Umgebung rechnen. So ist zu erwarten, dass passive Mobilität eher im Kontext systemnaher Agentenarchitekturen von Interesse ist.

Die Adressierung von Agenten ist bei mobilen Agenten ein weiterer kritischer Punkt. Während stationäre Agenten leicht über Mechanismen der Programmiersprache oder der Infrastruktur Zugriff aufeinander erlangen und sich so Daten zuschicken können, ist die Lokalisation von mobilen Agenten erheblich schwieriger [AO98]. Im Allgemeinen muss davon ausgegangen werden, dass feste Referenzen (wie über Speicherzellen) nicht vorliegen und dass der Agent nicht einmal an derselben Netzlokation wie ein anderer Agent verweilt, der mit diesem zu kooperieren wünscht. Es sind Hilfsmechanismen erforderlich, welche entweder Nachrichten an den entsprechenden Agenten weiterleiten oder eine Vermittlung zwischen Agenten ermöglichen. Diese Möglichkeit wird in dieser Arbeit genauer untersucht.

2.2.4 Standardisierungen

Die oben genannten Aspekte zeigen, dass es schwierig ist, ein konsistentes Bild von Agenten zu definieren. Standardisierungsbemühungen sind bislang wenig erfolgreich gewesen, da das Modell eines Agenten viel zu stark zwischen verschiedenen For-

schungsrichtungen divergiert. Eine Standardisierung bedeutet meist die Adoption eines fremden Agentenmodells unter Aufgabe der eigenen Vorstellungen.

Zudem sind einige der Aspekte einander widersprüchlich. Die Implementierung aufwändiger Kommunikationsmechanismen läuft einer effizienten Betriebssystemerweiterung zuwider. Die Systemnähe mancher mobiler Agenten lässt sich nicht in das in der VKI-Forschung definierte BDI-Agentenmodell einpassen. Passive Mobilität sowie synchrone Methodenaufrufe lassen sich nicht mit der Autonomie des Agenten vereinbaren.

Die *Object Management Group (OMG)* legte 1998 einen Entwurf für die Standardisierung von Agentensystemen vor, der als MASIF-Spezifikation bekannt wurde *(Mobile Agent System Interoperability Facility)* [OMG97, MBB+98]. In dieser Spezifikation werden Anforderungen an Agentensysteme gestellt, welche bis zu einem gewissen Grade interoperabel sein sollen; Grundlage für diese Interoperabilität soll CORBA sein. Der MASIF-Standard wurde bereits von Agentensystemen wie *Grasshopper* [IKV01] übernommen, ist jedoch nicht unumstritten, da einige der Forderungen zu starke Änderungen am Agentenbild erfordern.

Eine weitere Standardisierungsbemühung stammt von der *Foundation for Intelligent Physical Agents (FIPA)* [FIP01]. Die Standardisierung der FIPA bezieht sich auf folgende Punkte:

- Interoperabilität des Nachrichtentransports. Dies betrifft die Mechanismen wie Socketkommunikation, RMI, IIOP und andere.
- Unterstützung verschiedener Repräsentationen von Agentenkommunikationssprachen. Ein konkretes Beispiel ist FIPA ACL [FIP00], ein Derivat von KQML.
- Unterstützung verschiedender Formen der Kommunikationsinhaltssprache. Eine solche muss in der Lage sein, Vorschläge *(propositions)*, Aktionen *(actions)* und Namen von bestimmten Entitäten *(terms)* zu repräsentieren.
- Unterstützung mehrerer Verzeichnisdienstrepräsentationen. Verzeichnisdienste ermöglichen die Lokalisation von Agenten.

Es werden jedoch keine Festlegungen in Bezug auf den Lebenszyklus des Agenten, seine Mobilität, Domänen, Richtlinien der Kommunikation sowie Agentenidentität getroffen. Die FIPA-Spezifikation versteht sich als abstrakte Grundlage zur Realisierung konkreter Agentenplattformen.

2.2.5 Auswirkungen des Modells

Die in Definition 2.2 und den folgenden Bemerkungen induzierte Sichtweise auf Agenten wirkt sich in der praktischen Umsetzung aus. Die Eigenständigkeit von Agenten,

das vom Benutzer ungesteuerte Verfolgen des Plans erfordert *Nebenläufigkeit*. Ebenso scheint die Festlegung eines spezifischen Plans einer Sichtweise von Agenten als mobile Objekte, welche ihre Funktionalität anderen Objekten zur Verfügung stellen, zuwiderzulaufen. Bei Beachtung der Objektorientierung als Modellierungsgrundlage erhält man aus den oben genannten Charakteristiken eine Basis für den Entwurf eines autonomen Softwareagenten (siehe auch [Zap97]):

Entwurfsmuster eines autonomen Agenten

1. Ein Agent verfügt über genau eine Identität, welche die gesamte Aggregation der ihm zugeordneten Objekte referenziert. Diese Identität ist während des gesamten Lebenszyklus konstant.
2. Jeder Agent erfordert einen von anderen Agenten unabhängigen Kontrollfluss. Dies kann über eigenständige Prozesse oder Threads sichergestellt werden.
3. Agenten sollten keine Referenzen auf ihre Bestandteile an andere Agenten weitergeben.
4. Agenten müssen in der Lage sein, ihren Auftrag jederzeit eigenständig zu verfolgen und dürfen von anderen Agenten nicht an der Ausführung gehindert werden.
5. Andere Agenten, Benutzer und sonstige Einrichtungen sind Bestandteile der Umgebung eines Agenten. Die Interaktion zwischen Agent und Agent oder Agent und Benutzer ist ein Spezialfall der Interaktion eines Agenten mit seiner Umgebung.

In den oben genannten zahlreichen Agentensystemen werden durchaus sehr unterschiedliche Entwurfsmuster zu Grunde gelegt, die mit diesem hier vorgestellten nicht übereinstimmen. Einige der genannten Punkte führen bei strikter Anwendung zu scheinbaren Einschränkungen im Entwurf des Agentensystems und der Agenten. Wird beispielsweise die Kommunikation zwischen Agenten auf Basis von Methodenaufrufen organisiert, so ist eine strenge Eigenständigkeit (Punkt 4) kaum zu erreichen. Ein möglicher Ausweg ist der Ersatz der direkten (synchronen) Kommunikation durch eine vermittelte Kommunikation via asynchroner Nachrichten. Dies erlaubt zudem die Zusicherung der Abgeschlossenheit (Punkt 3).

Auch wenn ein direkter Methodenaufruf auf einem Objekt eines anderen Agenten verhindert wird und man somit die Klient/Server-Rollenverteilung zumindest auf der Ebene der Objektinteraktion aufhebt, heißt das nicht, dass die Agenten *autistisch* sind. Die Rollenverteilung Klient/Server findet sich auf einem höherem Abstraktionsniveau wieder, ganz so, wie ein Mensch einen Dienst verrichten kann ohne seine Eigenständigkeit zu verlieren. Die Kooperation findet dann auf dem Niveau der Pläne der einzel-

nen Agenten statt, nicht auf der Kommunikationsebene. Soll also Kooperation erreicht werden, müssen dies die einzelnen Agenten tatsächlich vorsehen.

Die Adressierung des empfangenden Agenten erfordert die in der Folgerung genannte, eindeutige Identität des Empfängers. Das Ablegen der Nachrichten in der Umgebung und das selbständige Abholen derselben impliziert eine asynchrone Kommunikation. Eine mögliche Implementierung einer solchen Kommunikationsinfrastruktur bietet der so genannte *TupleSpace* des *LINDA*-Systems [CG89].

Natürlich können Agenten so programmiert werden, dass sie die klassische Klient-Server-Sichtweise realisieren; ein Agent wartet mit der Fortsetzung seines Auftrages so lange, bis er die Daten von einem anderen Agenten erhalten hat. Dieser wiederum besitzt den Wunsch (*Desire* in der BDI-Sichtweise), Anfragen anderer Agenten zu beantworten. Eine solche Gestaltung missachtet viele Fähigkeiten eines Agenten und gibt andererseits Anlass zur häufig geäußerten Kritik, dass Softwareagenten keine prinzipiell neuen Lösungen böten. Interessant ist, dass die *Simulation* eines Server-artigen Verhaltens zeigt, dass es – in der Sichtweise der Verarbeitung – möglich ist, Agenten nicht als spezielle Objekte, sondern Objekte als spezielle Agenten aufzufassen [Zhu00], welche ihre Interaktionen an einem einfachen Anfrage-Antwort-Schema ausrichten.

2.3 Andere agentenartige Strukturen

Schon die Vielzahl der genannten Agententypen zeigt, wie mannigfaltig der Begriff des Agenten belegt ist. Es sollte nicht verwundern, dass der Agentenbegriff auch *werbewirksam* eingesetzt wird – so wie es mit den meisten aktuellen Schlagworten geschieht, ohne dass eine sorgfältige Analyse stattfand, ob das so betitelte Objekt tatsächlich Agenteneigenschaften aufweist. Hinzu kommt, dass gerade mit dem Begriff des mobilen Agenten viele weitere Strukturen unter der Vorstellung umherstreifender Programme unberechtigterweise als Agenten gesehen werden. Diese Strukturen werden in dieser Arbeit nicht als Agenten angesehen.

2.3.1 Viren und Würmer

Weit bekannt ist der Begriff des *Computervirus*; auch ist vielen der Begriff *Wurm* bekannt. Hierbei handelt es sich um Programme, welche zum Angriff auf einen fremden Rechner oder ein fremdes Netzwerk eingesetzt werden. Das Virus wird in der Regel durch die Ausführung von Programmen in ein System eingebracht, ohne dass dies der Anwender bemerkt. Sie verändern die Funktion einer oder mehrerer Systemkomponenten so, dass sie unerkannt repliziert werden und durch Datenaustausch mit anderen Rechnern auf dessen System übergreifen können. Viren tauchen meist auf Personalcomputern auf und haben destruktive Wirkung, was von der Löschung von Daten bis zur Beschädigung des Rechners durch Löschung des BIOS-ROMs gehen kann.

Würmer sind Programme, die sich in Netzwerken unbegrenzt auszubreiten trachten. Sie modifizieren dabei gewisse Systemkomponenten ähnlich, wie es Viren tun, sind aber in der Regel nicht dazu gedacht, die Rechner oder die auf ihnen gespeicherten Daten zu beschädigen. Ein mögliches Motiv des Einsatzes von Würmern ist das Verhindern eines ordnungsgemäßen Betriebs des angegriffenen Netzes durch Überlastung *(Denial of service)* oder die Ausspähung vertraulicher Daten. Bekannte Beispiele von Würmern sind der Internet-Wurm von Robert Morris im Jahre 1988 [HM95] oder der *ILOVEYOU*-Wurm aus dem Jahre 2000, welcher durch E-Mail-Nachrichten verbreitet wurde.

Würmer könnten nach obiger Definition nicht als Agenten gesehen werden: Sie nutzen bestimmte beabsichtigt oder unbeabsichtigt angebotene Replikationsvorkehrungen, um sich zu klonen; ein Wurm beendet seine Existenz in der Regel nach seiner Ausbreitung zum nächsten Rechner, wo er durch seine Klone ersetzt wird. Würmer tragen keine Informationen in sich, die einem bestimmten Auftraggeber zugute kommen sollen. Ähnliches gilt für Viren, die sich ebenfalls nur ausbreiten und keine feste Identität aufweisen.

2.3.2 Netzmanagement- und E-Mail-Benutzeragenten

Im Bereich des Netz- und Systemmanagements ist ebenfalls ein Agentenbegriff geprägt worden, welcher nur teilweise mit dem hier behandelten Begriff in Deckung zu bringen ist. Das *Simple Network Management Protocol (SNMP)* [CFSD90, Ros91, Sta93] sieht den Einsatz eines so genannten *Managers* sowie zahlreicher *Agenten* vor. Die Agenten dienen dazu, dem Manager auf Anfragen Daten der von ihnen verwalteten Geräte zur Verfügung zu stellen; umgekehrt führen sie Anweisungen des Managers aus. Die hinter diesem Konzept stehende Idee eines Agenten ist die eines *ausführenden Organs* des Managers. Der Agent vertritt in dieser Sichtweise den Manager an der Stelle, wo er installiert ist. Ein solcher Agent führt jedoch keine komplexen eigenen Aktionen aus,[3] sondern tut das, was der Manager im Augenblick von ihm verlangt. Er muss dazu nicht über einen eigenen Zustand verfügen.

Der so genannte E-Mail-Benutzeragent *(Mail User Agent)* spielt eine ähnliche Rolle; er vertritt den Benutzer vor dem E-Mail-Verteiler (wie *sendmail*). Eine solche Definition als *User agent* taucht in zahlreichen Architekturen auf, wann immer es um Programme geht, welche eine Schnittstelle zwischen System und Benutzer darstellen sollen. Auch hier erwartet man von einem solchen Agenten keine eigenen Aktionen, sondern die geeignete Weiterleitung der Benutzeraktionen sowie die Präsentation von Daten gegenüber dem Benutzer.

[3] Version 2 des SNMP-Protokolls versucht hier Abhilfe zu leisten. Aufgrund der geringen Verbreitung ist es von untergeordneter Bedeutung.

2.3.3 Webroboter/Webcrawler

Unter einem Webroboter versteht man ein Programm, welches Seiten im *World Wide Web (WWW)* absucht und dabei in der Lage ist, nach bestimmten Daten zu suchen. Webroboter (auch *Webcrawler*) machen sich Verweise auf den Seiten zu Nutze, um auf weitere Seiten zu stoßen und damit sukzessive große Bereiche des WWW zu durchsuchen.

Solche Programme, welche häufig in Suchmaschinen Anwendung finden, führen einen bestimmten Auftrag, nämlich die Recherche im Namen eines Benutzers, durch. Sie sind jedoch keinesfalls mobil, sondern automatisieren lediglich den Vorgang des fortgesetzten Springens zu weiteren Seiten (populär als *Surfen* bezeichnet); sie selbst verbleiben aber auf demselben Rechner.

Hier stößt man erneut auf eine Grauzone der Charakterisierung von Agenten; so werden Suchmaschinen immer wieder mit Agenten in Verbindung gebracht. Allerdings treffen nur wenige Eigenschaften auf sie zu. So kann man kaum von einem autonomen Verhalten reden, möchte man nicht jedes Programm als Agenten bezeichnen; außerdem treffen sie ihre Entscheidungen nach festgelegten Schemata und können nur begrenzt auf Umgebungseinflüsse reagieren. Auch eine Modellierung nach BDI-Muster lässt die relativ einfache Struktur der Verarbeitung nicht zu. Es existieren zwar Ansätze der „intelligenten“ Recherche (Stichworte *Knowbots* oder *Softbots* wie in Abschnitt 2.2.1 beschrieben), jedoch nutzen die gebräuchlichen Suchprogramme keine derartigen aufwändigen Strategien bei der Zusammentragung geeigneter Referenzen.

2.3.4 Applets

Die Programmiersprache und -umgebung Java führt das Konzept der *Applets* ein. Hier handelt es sich um Programme, welche in der Regel in eine Webseite eingebettet sind und zur Ausgestaltung der Seite mit leistungsfähigen Bedienelementen, mit speziellen Oberflächen oder Animation dienen.

Applets werden der Code-on-demand-Strategie gemäß bei Bedarf von jenem Rechner heruntergeladen, von dem die anzuzeigende Seite stammt. Nach dem Transfer auf den lokalen Rechner führt dieser ihren Programmcode aus. Als Java-Objekte sind Applets als eigenständige Einheiten zu begreifen; allerdings existiert ihre Instanz nur genau auf dem Klientenrechner, welchen sie nie verlässt. Die Technik der Übertragung von Applets ähnelt damit eher einem verteilten Dateisystem, von dem man ein Programm lädt und startet, als einem System mobiler Agenten.

2.4 Agentensysteme

Ähnlich wie die Ideen zum Wesen mobiler Agenten auseinandergehen, sind Implementierungen solcher Systeme teilweise recht unterschiedlich. Grundlegend gilt:

> *Ein* System mobiler, autonomer Softwareagenten *(kurz: Agentensystem) ist die Gesamtheit aller zum Betrieb mobiler, autonomer Softwareagenten notwendigen Softwarekomponenten.*

Allen Systemen gemein ist das Problem, dass keines der üblichen Betriebssysteme eine direkte Unterstützung von Agenten aufweisen kann. Diese Unterstützung ist vor allem für mobile Agenten notwendig, damit diese zwischen den Rechnern umherreisen können. Das Agentensystem beinhaltet des Weiteren Mechanismen für die Agenten, um sich gegenseitig Nachrichten zuzusenden oder in Kontakt mit der Umgebung (beispielsweise mit Benutzern) zu treten.

2.4.1 Java-basierte Mobile-Agenten-Systeme

Die Sprache Java, welche ursprünglich für den Betrieb von so genannten Set-Top-Boxen vorgesehen war, verdankt ihren Aufstieg dem explosionsartigen wachsenden Internet und dem darin betriebenen *World Wide Web (WWW)*. Die wichtigste Eigenschaft von Java ist die *Plattformunabhängigkeit:* Ein Java-Programm kann auf jedem Rechner ausgeführt werden, der über eine so genannte *Java-virtuelle Maschine* verfügt. Es handelt sich dabei um einen Interpreter, der den Java-Programmcode ausführen kann; dieser wird somit nicht direkt von Prozessor ausgeführt. Die Plattformunabhängigkeit ist für mobile Agenten eine sehr wichtige Eigenschaft, erlaubt es ihnen doch, von einem Rechner zum nächsten zu wandern, ohne dass Anpassungen ihres Codes oder eine Neuübersetzung notwendig wären.

Offenbar ist Java nicht primär für die Erstellung mobiler Agenten entworfen worden. Dies wird daran deutlich, dass Elemente wie die Zustandsrekonstruktion erst mit der Einführung der RMI-Technik[4] nutzbar wurden, welche für den Betrieb von Agenten wesentlich sind. Ferner fehlt die Unterstützung der *starken Migration* völlig; hier gibt es zahlreiche Ansätze, dies auf Umwegen zu realisieren [Fün98]. Auch existieren keine Vorrichtungen zur Verwaltung asynchroner Nachrichten. Diese Mechanismen müssen die Agentensysteme selbst zur Verfügung stellen.

Das Agentensystem AMETAS, welches die Infrastruktur zur Erforschung des Typsystems für autonome Agenten liefert, ist ebenfalls Java-basiert und wird in Kapitel 4 genauer betrachtet. Unter der Vielzahl von Java-basierten Agentensystemen seien die folgenden Systeme herausgegriffen.

Mole

An der Universität Stuttgart entstand 1995 mit *Mole* [BHR+97] das erste Mobile-Agenten-System in Deutschland auf Basis des damals noch im Betastadium befindlichen *Java Development Kit*.

[4] *Remote Method Invocation:* das Aufrufen von Methoden von Objekten außerhalb des eigenen Ausführungskontextes.

Mole ist ein in Java 1.1 implementiertes System, das als Betriebssystem-Ergänzung betrieben wird. Spezielle Prozesse *(Engines)* nehmen Mole-Agenten entgegen und senden sie zu anderen Prozessen weiter. Diese Prozesse betreiben so genannte *Lokationen*, an welchen sich Agenten zusammenfinden können. Mole definiert zwei Kategorien von Agenten:

- *Benutzeragenten* sind vom Benutzer eingesetzte Agenten, welche einen bestimmten Auftrag erledigen sollen. Sie können untereinander kommunizieren und von einer Ausführungsumgebung zur nächsten wechseln.
- *Systemagenten* bieten Dienste an, welche die Benutzeragenten zur Erledigung ihres Auftrags nutzen können. Sie bekommen spezielle, erweiterte Privilegien, um auf Systemkomponenten zugreifen zu können.

Zur Erstellung eines Mole-Agenten ist es erforderlich, die Klassen *UserAgent* oder *SystemAgent* abzuleiten, um entsprechend einen Benutzeragenten oder Systemagenten zu erhalten. Die Benutzeragenten bewegen sich von einer Lokation zur nächsten mittels eines speziellen Befehls *(migrateTo);* die Adressierung der Lokationen ähnelt dem Namensschema des *Domain Name System* (DNS).

Aglets

Das IBM-Forschungslabor in Tokio bietet eine eigene Java-Implementierung eines Agentensystems an, welches *Aglets Software Development Kit* genannt wird [LO98, AO98, Pap01]. Wie der Name schon andeutet, sollen *Aglets* als Pendants zu den bereits in Java vorhandenen Applets verstanden werden. Das Aglet-System definiert folgende Konzepte:

- *Aglets:* Mobiles Java-Objekt, das von einem Rechner zum nächsten wandern kann und autonom und reaktiv agiert.
- *Proxy:* Stellvertreter eines Aglets; dient der Abschottung des eigentlichen Aglets vor fremden Zugriffen und der Lokationstransparenz.
- *Kontexte:* Arbeitsumgebung für Aglets; entspricht den aus vielen Agentensystemen bekannten *Stellen.*
- *Nachrichten:* Spezielle Objekte, welche zum synchronen und asynchronen Datenaustausch zwischen Aglets vorgesehen sind.
- *Zukünftige Antwort:* Nachricht, welche bei asynchroner Verarbeitung als Platzhalter für die später eintreffende Antwort dient.
- *Identität:* Unveränderlicher Bezeichner eines Aglets, systemweit eindeutig.

Aglets entstehen durch Ableiten der Klasse *Aglet,* ihr Verhalten wird in Form der *run*-Methode implementiert. Die Migration zu anderen Kontexten wird über URLs spezifiziert, wobei als Protokoll *atp* angegeben wird. Die Verwendung von URLs erlaubt die Nutzung von vorhandenen Java-Mechanismen zur Verarbeitung von TCP-Nachrichten. Aglets können nur auf schwache Migration zurückgreifen. Bemerkenswert ist die zum *dispatch*-Befehl symmetrische *retract*-Anweisung, welche einen mit ersterem Befehl verschickten Agenten wieder zurückholt.

Grasshopper

Anfangs bei GMD Fokus in Berlin entwickelt, wird *Grasshopper* [IKV01] heute von einer eigenständigen Firma als Produkt vertrieben. Grasshopper ist eines der wenigen Agentensysteme, das die MASIF-Spezifikation der OMG umsetzt; laut Vertreiber wird es in zahlreichen Projekten, insbesondere in der Europäischen Union, eingesetzt.

Es handelt sich hier ebenfalls um ein System, welches in der Architektur zwischen Stellen und Agenten unterscheidet, welche zwischen diesen Stellen migrieren. Grasshopper unterstützt nur die schwache Migration, da die starke Migration eine Modifikation der zu Grunde liegenden Java-Plattform erfordert, was der Plattformunabhängigkeit zuwiderläuft.

Die Erzeugung eines mobilen Agenten erfolgt durch Ableiten der *MobileAgent*-Klasse. Die Abarbeitung eines Agenten findet durch Aufruf der *live*-Methode statt. Die Kommunikation zwischen Agenten findet über Methodenaufrufe statt, welche entweder lokal oder über Proxys und einen CORBA-ORB vermittelt realisiert sind.

Voyager

Die Firma *ObjectSpace* stellt mit *Voyager* ein System vor, das sich eher als System mobiler Objekte denn als Agentensystem begreift. Voyager implementiert einen *Object Request Broker* auf Java-Basis, welcher von CORBA-Objekten verwendet werden kann, denen Eigenschaften wie eine *Lebensdauer* und *Mobilität* verliehen werden kann. Jedoch können Anwendungen das Voyager-System als Basis nutzen, um weitere notwendige Vorkehrungen für den Betrieb eines Agentensystems zu liefern. Ein Beispiel ist *AgentSpace* [SSD98].

In Voyager können Objekte zu Agenten werden, indem sie die so genannte *Agenten-Fassette (Agent Facet) IAgent* zur Laufzeit dynamisch binden. Dieses Objekt liefert eine Schnittstelle, welche das Primärobjekt mit Agentenfähigkeiten ausstattet:

- Migration über *moveTo;*
- *setAutonomous* befreit den Agenten von der automatischen Instanzentsorgung, ermöglicht also eine eigenständige Verarbeitung, auch wenn keine Referenzen auf ihn mehr existieren;

- *getHome* liefert die Heimatposition.

Nach der Migration wird die Ausführung in einer vom Programmierer für den jeweiligen Migrationsvorgang anzugebenden Methode fortgesetzt; es handelt sich somit um eine – wenn auch flexible – schwache Migration.

2.4.2 Intelligente-Agenten-Systeme

Auch im Bereich der intelligenten Agenten gibt es eine Reihe von Entwicklungen, von denen drei kurz angeführt sein mögen.

Java Agent Template Lite

An der Universität von Stanford entstand das *Java Agent Template*, welches zurzeit als JATlite 0.4 frei verfügbar ist [JAT98]. JATlite ist ein Java-Klassenpaket, das die Erstellung von autonomen Agenten erlaubt, wobei die Inter-Agent-Kommunikation relativ frei gewählt werden kann. Das System sieht KQML als Standard vor und bietet geeignete Klassen zur Unterstützung der Implementierung an.

Agenten können auf einer dedizierten Infrastruktur oder als *Applet-Agenten* in einem WWW-Browser betrieben werden; sie können auch ihren Ausführungsort wechseln. Die Architektur ist in fünf Schichten unterteilt:

- die *abstrakte Schicht* und die *Basisschicht*, welche abstrakte Klassen zur Implementierung der grundlegenden Kommunikation bieten sowie die möglichen Ein-/Ausgabekanäle definieren;
- die *KQML-Schicht* zur Unterstützung der KQML-basierten Kommunikation;
- die *Routerschicht*, welche den Nachrichtenaustausch zwischen Agenten regelt und
- die *Protokollschicht* zur Unterstützung höherer Protokolle wie HTTP, FTP, SMTP.

JATlite wurde unter anderem in den Bereichen *CAD*, *Rapid prototyping* und *Finanzanalyse* zum Einsatz gebracht.

AgentBuilder

Bei *AgentBuilder* [AB99] handelt es sich um eine Entwicklungsumgebung für intelligente Agenten, wobei besonderen Wert auf das BDI-Entwurfskonzept gelegt wird. Die Entwicklung von Agentenanwendungen wird in Stufen eingeteilt (Analyse, Entwurf, Entwicklung, Einsatz), welche jeweils durch das System unterstützt werden. Die Entwicklung auf Basis des BDI-Konzepts wird durch grafische Oberflächen unterstützt.

Die Ausführung von Agenten erfolgt durch ein spezielles, interpretierendes Laufzeitsystem. Agenten können einer in Betrieb befindlichen Anwendung dynamisch hinzugefügt werden. Multi-Agenten-Anwendungen bestehen aufgrund der Fixierung auf das vollständige BDI-Modell nur aus wenigen Agenten; die Erstellung erfordert gute Kenntnisse in der Logikprogrammierung [RD00].

Jack

Bei *Jack* [AOS01] handelt es sich um ein weiteres BDI-zentriertes Agentenentwicklungssystem. Diese Umgebung basiert auf einer *proprietär erweiterten Java-Sprache,* deren Programme durch einen Compiler in gewöhnlichen Java-Code übersetzt werden. Das Gewicht liegt hier auf der Entwicklungsphase; Analyse und Entwurf werden nicht explizit unterstützt.

Jack richtet sich vor allem an Entwickler, die mehrere Multi-Agenten-Systeme zur Kooperation bringen wollen. Voraussetzung ist dabei die durchgängige Verwendung des BDI-Modells und eine Analyse der Korrespondenz von Grundelementen (wie Ziele oder Interaktionen) [RD00].

2.4.3 Andere Agentensysteme

Es finden sich zahlreiche Versionen von Agentensystemen oder Infrastrukturen für den Betrieb von Agenten, die nicht primär auf Java basieren. Meist werden *Skripte* verwendet, also textbasierte Programme, welche von einem Interpreter ausgeführt werden. Ein wichtiges Beispiel ist *AgentTcl* [Gra96], das eine Erweiterung der *Tcl*-Sprache darstellt.

Telescript ist – könnte man sagen – der Urahn der Mobile-Agenten-Systeme. Es wurde von der amerikanischen Firma General Magic 1994 ins Leben gerufen und war explizit für die Erstellung mobiler Agenten vorgesehen [Whi94]. Bei Telescript handelte es sich um einen Aufsatz, der die Programmierung auf Basis einer interpretierten, objektorientierten Sprache realisierte. Viele Konzepte von Telescript wurden in andere Agenten-Plattformen übernommen, etwa das Konzept der mobilen Agenten, der Stellen, auf denen die Agenten ausgeführt werden sowie die von den Agenten initiierte Migration.

Telescript wurde von General Magic hauptsächlich wegen der starken Verbreitung von Java aufgegeben, welches den Vorteil hatte, dass keine proprietäre Sprache erlernt werden musste. An die Stelle von Telescript trat das Java-basierte System *Odyssee,* welches allerdings mittlerweile ebenfalls eingestellt wurde.[5]

Das Agentensystem *ffMain* [LD98] basiert seine Kommunikation auf dem *http*-Protokoll, sodass Webserver mit der Fähigkeit ausgestattet werden können, Agenten zu verwalten. Die verwendete Sprache, in der die Agenten geschrieben ist, hängt von dem jeweiligen Server ab; Beispiele gibt es für *Perl* oder *Tcl* .

[5] General Magic hat das Betätigungsfeld vollständig gewechselt und bietet keinerlei Informationen zu Telescript oder Odyssee an.

Das *WASP*-System *(Web agent-based service providing)* [FM99] integriert Agentenfunktionalität in WWW-Servern mittels spezieller Servererweiterungen (so genannter *Java-Servlets*). Dabei werden an Webservern ankommende Agenten in die jeweilige Ausführungsumgebung *(SAE, Server Agent Environment)* umgeleitet. Die Agentenprogrammierung basieren auf Java, allerdings werden automatisierte Eingriffe in den Code vorgenommen, um eine Migration weitestgehend transparent, also stark, zu gestalten. Details hierzu sind in [Fün98] zu finden.

2.5 Zusammenfassung

Dieses Kapitel gab einen kurzen Überblick über die Grundlagen von Agentensystemen, Agenten und den ihnen zu Grunde liegenden Konzepte der Objektorientierung. Dieses Umfeld wird in der einschlägigen Literatur ausführlich behandelt, sodass auf die entsprechenden Literaturstellen verwiesen sei.

Die Betrachtung verschiedener Auffassungen des Begriffs *Agent* sowie die Struktur von Agentensystemen ist wichtig, um Aussagen über die Möglichkeit der Definition von Typen für Agenten treffen zu können. In den vergangenen Jahren sind zahlreiche Systeme entstanden, sodass ein Überblick über aktuelle Agentensysteme sowohl im Bereich der mobilen als auch der intelligenten Agenten, wie er abschließend geboten wurde, nur einen groben Eindruck vermitteln kann. Für weiter gehende Informationen sei auch hier auf die Literatur sowie Informationen im WWW verwiesen.

3 Typen und Klassen

In diesem Kapitel werden Grundlagen zur Theorie der Typen angeführt. Häufig wird unter einem Typ lediglich das Konzept der Datentypen verstanden; in objektorientierten Sprachen kennt man als Erweiterung die Klassen, welche als Schablonen zur Erzeugung von Instanzen dienen. Es ist hingegen möglich, den Typbegriff wesentlich allgemeiner anzuwenden.

3.1 Typen

Der Begriff des *Typs* wird meist in Verbindung mit *Datentypen* gebracht. Die intuitive Vorstellung eines solchen Datentyps ist, dass Objekte eines bestimmten Datentyps

- Elemente einer bestimmten Menge von Entitäten sind, welche mit einer festen Bedeutung versehen sind;
- eine genau definierte Menge von Operationen erlauben, dass also *Verknüpfungen* definiert sind;
- mit einer bestimmten Menge von Grundaussagen *(Axiome)* versehen sind, aus denen weitere Aussagen konstruiert werden können.

3.1.1 Einfache Typen

Die *natürlichen Zahlen* $\mathbf{N}$ werden häufig verwendet, um die Kardinalität einer Menge zu beschreiben. Neben den Elementen $0, 1, 2, \ldots$ sind auch gewisse Verknüpfungen wie $+, -, \cdot, \div$ zwischen ihnen definiert. Die Darstellung

101105110

ist als natürliche Zahl zu identifizieren, sofern man eine Konversion zwischen der Zahl und der hier gezeigten Darstellung definiert hat (implizit wird hier vorausgesetzt, dass die Darstellung im Dezimalsystem vorgenommen wird). Eine andere Konversion könnte diese Darstellung jedoch als Bild der Zeichenkette

ein

definieren: Die so genannte *ASCII*-Kodierung bildet Zeichen auf zugehörige Bytewerte ab; so steht der Wert 101 für das kleine E, mithin entsteht die oben genannte Zahl durch die Konkatenation dieser Werte. Die Korrespondenz zwischen der Darstellung des Wertes und dem Wert selbst ist von seinem Typ abhängig. Dies gilt ebenso für die Repräsentation des Wertes innerhalb des Computerspeichers: Aus 0-/1-Werten zusammengesetzt ist die Bedeutung der im Speicher abgelegten Daten nicht eindeutig festgelegt.

Typisierte Programmiersprachen sorgen dafür, dass den Speicherinhalten eine bestimmte Bedeutung zugeordnet wird. Dies erfolgt durch *Typdeklarationen.* So wird durch

```
int i;
```

im Falle eines Java-, C- oder C++-Programms festgelegt, dass die im folgenden Programmtext auftauchende, mit *i* referenzierte Speicherzelle Werte des Typs *int* beinhaltet. Die Semantik von *int* wird dabei so definiert, dass es sich um eine Teilmenge der natürlichen Zahlen handelt, welche durch eine festgelegte Zahl von Binärstellen repräsentiert werden kann (meist 32). Ferner sorgt die Programmiersprache dafür, dass Werte dieses Typs durch bestimmte Operatoren miteinander verknüpft werden können und legt auch den Typ des Ergebnisses (welcher nicht gleich den Argumenttypen sein muss) fest. Die *Korrektheit* von Ausdrücken – also die Tatsache, dass die Operatoren nur mit Datentypen zusammentreffen, die für sie vorgesehen sind – kann bereits durch den Compiler nachgewiesen werden. Der Ausdruck $(i+3)/10 < 5$ ist korrekt, wenn *i* als *int*-Variable deklariert wurde, da die auftauchenden Operatoren für den Datentyp *int* erklärt sind.

Gleich lautende Operatoren können bei unterschiedlichen Typen eine unterschiedliche Semantik aufweisen. Ein Beispiel liefert Java mit der Zeichenkettenverknüpfung:

```
String s = s1 + s2;
```

Das Ergebnis ist die Verkettung der zwei Zeichenketten *s1* und *s2*. Hätte man eine ASCII-Darstellung der Zeichenketteninhalte gewählt, ohne den Typ *String* zu deklarieren, so bestünde eine Zweideutigkeit in Bezug auf die Bedeutung des Operators:

$$101105110 + 101114$$

Der Compiler kann nicht entscheiden, welche Semantik dieser Ausdruck aufweist. Der Operator $+$ könnte die Addition der Zahlen oder die Verkettung der Zeichenketten zur Folge haben. Aus diesem Grunde werden in Programmiersprachen die Typen der *Literale* (Darstellungen von Werten) kenntlich gemacht, beispielsweise durch Anfügen oder Voranstellen bestimmter Marken oder Einfassen in Anführungszeichen.

Die Verwendung von Typen hat daher die wichtige Aufgabe, die in Berechnungen auftauchenden Werte mit einer *Semantik* zu belegen und dadurch die Menge der zugeordneten Operationen genau zu definieren. Es gibt auch *nichttypisierte* oder *typlose*

Sprachen wie beispielsweise *LISP* oder Skriptsprachen. Diese führen jeden Speicherinhalt auf denselben Typ, meist die Zeichenkette zurück; umgekehrt müssen sie dafür sorgen, dass jede Information korrekt und eindeutig als Zeichenkette repräsentiert werden kann. Da es textuelle Repräsentationen selbst numerischer oder logischer Werte gibt, ist die alleinige Verwendung von Zeichenketten keine Einschränkung; jedoch dürfen dann Operatoren nicht zweideutig definiert sein. Die Korrektheit eines Programms lässt sich in der Regel erst während des Ablaufs nachweisen.

Smalltalk ist eine objektorientierte Programmiersprache, welche die Modellierung von Daten als Objekte anders als C++ auch für elementare Datentypen vorsieht. Dieser Sichtweise zufolge sind Operatoren keine Verknüpfung zweier Werte, sondern Nachrichten, welche von den Objekten akzeptiert und verarbeitet werden. Daher sieht man die Rechnung 1 + 2 in Smalltalk so:

> Das Objekt 1 empfängt über den Kanal namens „+“ das Objekt 2 und antwortet über den Ergebniskanal mit einer Nachricht mit dem Objekt 3.

Dies mag auf den ersten Blick über die Maßen umständlich klingen, bietet jedoch eine zweckmäßigere Sichtweise in Bezug auf die Typisierung: Neben einer Menge gültiger Werte muss die Menge gültiger Methoden angegeben werden oder – in der Sprache der Nachrichten-orientierten Kommunikation – die von einer Instanz dieses Typs akzeptierten Nachrichten.

3.1.2 Typen als Menge von Werten

Typen können als Menge von Werten aufgefasst werden, wie CARDELLI in [Car97] oder MITCHELL in [Mit96] darlegen. Diese Werte sind aber nicht alleine die Belegung einer Variablen, sondern umfassen auch Aussagen über die Menge der möglichen Operationen, ohne die sich die Typen nicht unterscheiden ließen.

Mengen lassen sich im Allgemeinen so charakterisieren, dass ihre Elemente ein gemeinsames *Prädikat* erfüllen. Dies nennt man das *Mengenbildungsprinzip:*

$$x \in M \Leftrightarrow E(x) \text{ oder } M = \{\, x \,|\, E(x) \,\}$$

wobei E ein logisches Prädikat ist. Man beachte, dass diese Definition von CANTOR nach RUSSEL [Rus03] nicht widerspruchsfrei ist (so genannte *russelsche Antinomie*); dieses und ähnliche Paradoxien sind jedoch für die Betrachtungen in dieser Arbeit unerheblich, da sie lediglich auftauchen, wenn Aussagen über Elemente unterschiedlicher *Mengenbildungsstufen* getroffen werden. Im Zusammenhang mit Typen interessiert man sich für Mengen von Instanzen, sodass man als A die Menge aller möglichen Typinstanzen annehmen kann, ohne sich in die Gefahr der Paradoxien zu begeben.

Eine Typinstanz ist ein Element einer Menge, die einen Typ beschreibt. Sie einfach als Wert in gewöhnlicher Weise zu erklären ist jedoch nicht ausreichend, wie das

$$\begin{array}{rcl} B & = & \{w, f\};\ x,y \in B \\ x.meth() & \supset & \{und(B) \to B, oder(B) \to B, nicht() \to B\} \\ x.und(y) = w & \Leftrightarrow & x = y = w \\ x.oder(y) = f & \Leftrightarrow & x = y = f \\ x.nicht() = w & \Leftrightarrow & x = f \end{array}$$

Abbildung 3.1: Beschreibung des Typs *boolean*

Beispiel $B = \{w, f\}$ nahe legt. B besitzt hier zwei Elemente w und f, über deren Semantik nichts bekannt ist. Ist $B' = \{t, f\}$, so lässt sich ohne weiteres keine Aussage über das Verhältnis zwischen B und B' angeben. Erst durch die Ausformulierung des Prädikats wird klar, dass es sich um *boolesche Werte* handelt; diese ist in Abbildung 3.1 dargestellt. Das *Prädikat* erhält man nun durch die konjunktive Verknüpfung aller Aussagen.[1] Jede andere Beschreibung von *boolean* sollte dieser Beschreibung im Wesentlichen gleichen oder sich durch logische Umformungen in diese überführen lassen. Führt man B' mit den oben gegebenen Werten ein und setzt die Menge der Methoden als $\{and(B') \to B', or(B') \to B', not() \to B'\}$ fest, so ergibt sich direkt eine Entsprechung der Methoden $nicht()$ und $not()$. Die anderen Methoden lassen sich einander nicht eindeutig zuordnen, ohne die Definition der Methoden zu Rate zu ziehen. Schließlich erkennt man, dass die Paare (w, und) und $(f, oder)$ miteinander vertauscht werden können, eine Eindeutigkeit also nur modulo dieser Paarbildung möglich ist. Legt man entweder die Wertezuordnung oder die Methodenzuordnung fest, so kann ein *Isomorphismus* beide Typen aufeinander abbilden.

Liegt also eine Instanz vor, welche einen von zwei Werten annehmen und die angegebenen drei Methoden kennt, so kann man sie (vermöge des Isomorphismus) als Instanz des Typs *boolean*, also als Element der von *boolean* definierten, zweielementigen Menge ansehen. Die Instanz kann durchaus weitere Methoden kennen, die von *boolean* laut Abbildung 3.1 nicht erwartet werden; daher ist die Menge der Methoden in der Definition eine Obermenge der drei genannten Methoden. Wird das Prädikat weiter verschärft,[2] gelangt man zu Untermengen der Menge, die den Typ beschreibt; diese Untermenge beschreibt selbst wiederum einen Typ, den man *Subtyp* des gegebenen Typs nennt.

3.1.3 Rekursive Typen

Viele Typsysteme erlauben die Einsetzung des zu definierenden Typs, beispielsweise im Falle einer Liste:

[1] Wobei hier gleich in Anlehnung an die Schreibweise bei Methodenaufrufen die Teilprädikate wie Methoden formuliert werden. Die Prädikate seien als totale Abbildungen erklärt, um die hier nicht genannten Beziehungen zu implizieren.

[2] Sinnvollerweise sollte das Prädikat eine widerspruchsfreie Verschärfung erlauben, sonst erhält man den trivialen Typ: die leere Menge.

```
type liste = {
     element:Objekt;
     naechstes:*liste;
}
```

In diesem Falle würde eine solche Liste durch Entlangschreiten an *naechstes* durchlaufen werden; das Ende würde durch einen speziellen Wert, meist *nil* genannt, markiert. Man spricht hier von *rekursiven Typen*. Diese Art von Typen ist recht nützlich, wenn aus einer bestimmten Menge von Typen komplexe Typen beliebiger Struktur erzeugt werden sollen, wie in diesem Beispiel Listen oder Bäume.

Formal werden rekursive Typen durch einen Term $\mu X.A$ spezifiziert, welcher die Lösung der *Typgleichung* $X = A(X)$ darstellt. Dieser Formalismus ist eine Weiterführung des Lambda-Kalküls auf die Ebene der Typvariablen und gibt Anlass zur Definition so genannter *Typsysteme zweiter Ordnung*, welche die Möglichkeit vorsehen, Typvariablen in Termen zu verwenden und diese damit durch Typen zu *parametrisieren*. Im Gegensatz hierzu erlauben Typsysteme *erster Ordnung* lediglich Terme aus Variablen, welche Instanzen von Typen beinhalten können.

Rekursive Typen werden bei der Definition des Agententypsystems keine Rolle spielen; für weiter führende Details in Bezug auf rekursive Typen und Ordnungen von Typsystemen sei auf [Car97] und [Mit96] verwiesen.

3.2 Klassen und Schnittstellen

Typsysteme können als Gegenstände der theoretischen Informatik rein formal betrachtet werden. Sollen sie Anwendung in Programmiersprachen finden, müssen entsprechende Implementierungen von Typsystemen vorgenommen werden. Eine besondere Form wird im Kontext der objektorientierten Programmierung eingeführt: das *Klassen-* und *Schnittstellenkonzept*.

3.2.1 Klassen

Um Objekte in objektorientierten Programmiersprachen zu erzeugen, wird das Konzept der *Klassen* benutzt. Eine Klasse lässt sich als Schablone sehen, die als Vorlage zur Bildung von Instanzen der genannten Klasse dient. Diese Klasse definiert folgende Komponenten einer Instanz:

- Menge der Methoden, welche die Instanz anbietet;
- Implementierung der Methoden;
- Typen der Datenfelder;
- Abstammung.

Da Instanzen erst während der Laufzeit eines Programms entstehen, kann ein Anwendungsprogrammierer lediglich Klassen erstellen, die während der Laufzeit instantiiert werden, nicht die Instanzen selbst.

Klassen dienen in ihrer Eigenschaft als Implementationsschablonen auch als Referenz für den Typ der Instanz. So lassen sich Objekte danach einteilen, von welcher Klasse sie stammen. Dies ist gerechtfertigt, da Klassen in den verbreiteten Programmierumgebungen aus statischem Programmcode bestehen, der während der Laufzeit nicht veränderbar ist. Damit erhält man die Aussage:

> Jede Klasse definiert einen Typ, der alle von ihr erzeugten Instanzen umfasst.

Diese Aussage ist nicht umkehrbar. Typen werden durch ein Prädikat beschrieben, welches Bedingungen beinhalten kann, die nicht durch die Implementierung widergespiegelt werden können.

> Es gibt Typen, welche nicht durch eine Klasse implementiert werden können.

Klassen definieren die Menge der Methoden sowie deren Implementierung; auch die den Zustand definierenden Felder werden in der Klasse festgeschrieben. Bedingungen, die beispielsweise an Werte des Zustands geknüpft sind, werden damit nicht wiedergegeben; es ist nicht möglich, eine Klasse zu schreiben, welche dem Typ der irrationalen Zahlen entspricht.

3.2.2 Schnittstellen

Möchte der Implementierer keine Aussagen über die Implementierung der Methoden treffen, so können *Schnittstellen* definiert werden. Diese beinhalten lediglich die Namen der Methoden sowie die Datentypen der Eingangs- und Ausgangsparameter. Eine Klasse kann eine Schnittstelle *implementieren*, wenn sie für alle in der Schnittstelle genannten Methoden Implementierungen anbietet. Eine Schnittstelle definiert – wie eine Klasse – einen Typ anhand der von ihr deklarierten Methoden:

> Jede Schnittstelle definiert einen Typ, der alle Instanzen umfasst, die die aufgeführten Methoden anbieten.

Wiederum gilt nicht die Umkehrung; nicht jeder Typ ist durch eine Schnittstelle eindeutig beschreibbar. Schnittstellen geben keinerlei Auskunft über Details, wie die Methoden implementiert sind.

Schnittstellen spielen eine wichtige Rolle insbesondere bei Infrastrukturen, welche eine Verteilung von Objekten oder Prozeduren erlauben. Sie können verwendet werden,

um Codestrukturen vorzubereiten, in welche die entsprechende Implementierung einzusetzen ist. Die Objekte oder Prozedurserver bieten die Methoden/Prozeduren anhand der von ihnen implementierten Schnittstellen an. Klienten, die einen passenden Server suchen, können sich in einem speziellen Verzeichnis (bei CORBA namentlich *Interface Repository* und *Implementation Repository*) nach einer geeigneten Implementierung erkundigen.

Das offenkundige Problem ist, dass zwar einerseits die Implementierung nicht das entscheidende Kriterium sein sollte, wenn es um die Frage der Kompatibilität geht, dass aber die Schnittstelle alleine zu wenige Informationen bietet, um eine Kompatibilität entscheiden zu können. Daher wird bei Schnittstellen eine *implizite Semantik* in Bezug auf die Methoden vorausgesetzt: Gleich benannten Methoden wird unterstellt, dass sie das Gleiche leisten. Umgekehrt wird bei unterschiedlich klingenden Methodennamen eine unterschiedliche Semantik angenommen. Dies erzwingt eine für das jeweilige System eindeutige Festlegung auf die Benennung von Methoden für bestimmte Funktionalitäten; so müsste per Konvention eine Methode zum Ausdruck auf einem Drucker immer *print* heißen, *output* würde als unpassend erachtet.

3.2.3 Schnittstellenbeschreibungssprachen

Das *Distributed Computing Environment (DCE)* [OG97] definiert eine spezielle Schnittstellenbeschreibungssprache *(Interface Definition Language, IDL)*, welche programmiersprachenunabhängig gehalten ist, sich jedoch in der Struktur an die Syntax der Programmiersprache C anlehnt. Ein Beispiel einer DCE-IDL-Beschreibung könnte so aussehen:

```
[
   uuid(56482a9c-1298-120b-87ce-ba712c0ef120),
   version(1.0)
]
interface UserRegistration {
   // Registrieren
   void RegisterUser(
        [in, string, ptr] char *Name,
        [out]             long id
   );
   // Deregistrieren
   void UnregisterUser(
        [in] long id
   );
}
```

Die Schnittstellenbeschreibung listet alle Methoden auf, welche von dem Objekt zu implementieren sind. Dabei dienen Attribute der genaueren Bestimmung der Parame-

ter: *in* und *out* stellen klar, ob es sich um Eingabe- oder Ausgabeparameter handelt, während *ptr* einen Zeiger und *string* eine Zeichenkette kennzeichnet. Die so genannte *UUID* lässt eine Referenzierung dieser Schnittstelle auf Basis dieser eindeutig vergebenen Kennzeichnung zu. Eine Schnittstellenspezifikation auf Basis von DCE-IDL kann nicht als Parametertyp eingesetzt werden.

Eine weitere Schnittstellenbeschreibung liefert die *OMG IDL* [OMG00a]. Die Notation von Schnittstellen in dieser Sprache orientiert sich an der Syntax der Programmiersprache C++. Ein Beispiel für eine in OMG IDL formulierte Schnittstelle ist:

```
interface Konto {
   readonly attribute float kontostand;
   void einzahlen(in float betrag);
   void abheben(in float betrag);
}

interface Bank {
   Konto erzeugen(in string name);
   void entfernen(in Konto a);
}
```

Zum einen erlaubt OMG IDL die Definition von Attributen, welche wie in diesem Falle als *nur lesbar* deklariert werden können. Ferner ist es möglich, Schnittstellen als Argumente einzusetzen.

Soll eine IDL-Schnittstelle implementiert werden, bedient man sich eines *IDL-Compilers*, welcher geeignete Aufrufweiterleitungs-Objekte *(Stubs* und *Skeletons)* und bei Bedarf auch Codeschablonen zur Erleichterung der Implementierung liefert (siehe auch Abschnitt 8.3). Die jeweilige Umsetzung in die Implementierungssprache (Sprachabbildung oder *Language mapping*) ist entsprechend normiert; die in der IDL definierten Felder und Methoden können in einer Weise realisiert werden, die in der jeweiligen Programmiersprache verfügbar sind. So werden Felder wie *kontostand* durch Zugriffsmethoden realisiert, welche sich aus dem Feldnamen mit vorangestelltem *get* und *set* ergeben.

Auch die Programmiersprache Java definiert ein Schnittstellenkonzept, das jedoch nicht sprachunabhängig ist, sondern fester Bestandteil der Java-Sprache ist. Schnittstellen können prinzipiell als Parameter übergeben werden und entsprechen der Struktur von Java-Klassen. Als Signaturen dienen die Methodendeklarationen, wie sie in einer Klasse auftauchen, die Methodenrümpfe bleiben leer. Ein Beispiel für eine Java-Schnittstelle ist in Abbildung 3.5 zu sehen.

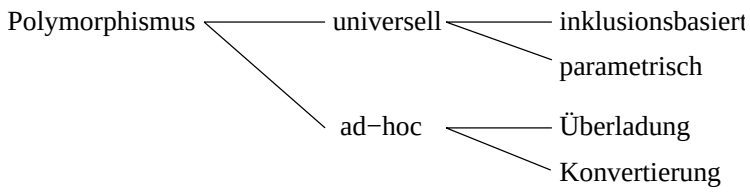

Abbildung 3.2: Polymorphismusarten

3.3 Polymorphismus und Vererbung

Ein wichtiges Konzept der objektorientierten Programmierung ist die *Wiederverwendung (Re-use)* von Komponenten. Dies wird in Form der *Vererbung von Klassen und Schnittstellen* realisiert.

3.3.1 Polymorphismus

Polymorphismus *(grch. Vielgestaltigkeit)* bezeichnet die Eigenschaft, dass Variablen und Werte unterschiedlichen Typen angehören können. Im Gegensatz hierzu lässt sich bei *monomorphen Programmiersprachen* stets der Typ einer Variable oder eines Werts genau angeben. Es gibt zahlreiche Versionen des Polymorphismus, wie Abbildung 3.2 zeigt.

Parametrischer Polymorphismus taucht bei Funktionen auf, welche in ihren Argumenten einen Bereich von Typen zulassen. Im Falle von Java könnte man eine Methode schreiben, welche *java.lang.Object* als Argumenttyp angibt. Die geeignete Verarbeitung muss innerhalb des Codes der Methode sichergestellt sein; so sollte zunächst der Typ des aktuellen Arguments bestimmt werden.

Inklusionsbasierter Polymorphismus liegt vor, wenn ein Objekt verschiedenen Klassen angehört, wobei diese nicht disjunkt sein müssen. Eine Klasse kann also Instanzen generieren, von denen alle oder ein Teil auch als Instanzen einer anderen Klasse aufgefasst werden können. Dies führt zum Begriff der Subtypen oder Subklassen, der später genauer erklärt wird.

Diese beiden Arten von Polymorphismus werden *universell* genannt; Methoden oder Funktionen, welche universell polymorph sind, verwenden denselben Code für Argumente zulässiger Typen. Im Gegensatz hierzu ist es beim *Ad-hoc*-Polymorphismus möglich, unterschiedlichen Code zu verwenden.

Beim *Überladen* von Funktionen oder Methoden wird derselbe Name verwendet, aber die Argumentlisten weisen unterschiedliche Typen auf. Je nachdem, welche Typen die aktuellen Argumente haben, wird die entsprechende Funktion oder Methode aufgerufen. Dies kann zur Übersetzungszeit oder zur Laufzeit (auch *spätes Binden* genannt) geschehen.

Eine weitere Form des Ad-hoc-Polymorphismus besteht in der automatischen Konvertierung von Argumenttypen *(Koerzion):* Es wird vor dem Aufruf versucht, die Argu-

menttypen in die erwarteten Typen umzuwandeln. So können etwa *int*-Werte in *long*-Werte verwandelt werden. Auch dies kann sowohl zur Übersetzungs- als auch zur Laufzeit geschehen.

3.3.2 Ableitung von Klassen

Die Modellierung in Form abgeschlossener Komponenten erlaubt es, Funktionen in abgeschlossenen Einheiten zu isolieren, sie mehreren Stellen einer Anwendung zugänglich zu machen oder auch in verschiedenen Anwendungen zum Einsatz zu bringen. Objekte können wie in einer Bibliothek zusammengestellt und weiteren Anwendungen zur Verwendung angeboten werden. Insbesondere der Implementierungsaufwand soll durch diese Wiederverwendung vermindert werden – es müssen weniger Klassen geschrieben werden.

Wie bereits erwähnt dienen Klassen als Schablonen zur Instantiierung von Objekten. Sie bestehen dabei

- aus den Methoden, welche das Objekt anbietet und
- aus den Feldern, welche den Zustand des Objekts umfassen.

Möchte man eine Klasse in verschiedenen Anwendungen verwenden, so müssen die Mengen der jeweils benötigten Methoden und Felder vereinigt werden. Neu hinzukommende Anwendungen müssen so entworfen werden, dass sie mit den bestehenden Methoden und Feldern arbeiten können.

Um dies zu vereinfachen, bieten objektorientierte Sprachen wie C++ oder Java das *Ableiten* von Klassen an. Dabei bezieht man sich bei der Definition einer Klasse auf eine bereits existierende Klasse und erweitert diese um weitere Methoden und Felder. Abbildung 3.3 zeigt ein Beispiel. Das Feld *i,* welches in der Klasse *Klasse1* definiert wird, wird in *Klasse2* verwendet, ebenso die Methode *meth1*. *Klasse2* verfügt somit über zwei Felder und zwei Methoden, von denen sie je eine von *Klasse1* übernimmt *(erbt).* In Java wird die Ableitung einer Klasse von einer anderen durch das Schlüsselwort *extends* erklärt. Dem Compiler und dem Laufzeitsystem müssen bekannt sein, wie sie die Definition der vererbenden Klasse finden. In der Regel wird dabei auf einen Zusammenhang zwischen Dateiname und Klassenname zurückgegriffen.

Wenn eine Klasse von mehreren anderen Klassen erbt, spricht man von Mehrfachvererbung[3]. Dieses Konzept wird von Sprachen wie C++ unterstützt, nicht jedoch von Java. Eine von zwei Klassen abgeleitete Klasse trägt die vererbbaren Merkmale beider Klassen. Dabei kann es jedoch zu Konflikten kommen; etwa, wenn beide Elternklassen gleichnamige Methoden weitergeben.

[3] Wörtlich aus dem Englischen übersetzt – und sinngemäß zutreffender – wäre *Mehrfachbeerbung;* üblich ist jedoch der Begriff *Mehrfachvererbung*.

```
class Klasse1 {
   int i;                  // Feld
   void meth1(int n) {     // Methode
       ...
   }
}
class Klasse2 extends Klasse1 {
   float k;
   void meth2() {
       i=1;          // Feld aus Klasse1
       meth1(i);     // Methode aus Klasse1
   }
}
```

Abbildung 3.3: Ableitung einer Klasse

Die in obigem Beispiel von *Klasse1* abgeleitete Klasse *Klasse2* wird auch *Unterklasse* oder *Subklasse* von *Klasse1* genannt. Eine Instanz von *Klasse2* ist immer auch eine Instanz von *Klasse1*. Sie besitzt lediglich weitere Methoden und Felder, welche die Definition der geerbten Merkmale nicht beeinflussen; sie können also ignoriert werden. Die Menge aller Instanzen von *Klasse1* wird also nicht alleine von *Klasse1* produziert, sondern von beliebig abgeleiteten Subklassen von *Klasse1*.

Eine weiter Möglichkeit der Modifikation einer Basisklasse ergibt sich beim Ableiten durch *Überschreiben (overriding)*. Dabei wird eine Methode der Basisklasse durch eine Methode der Subklasse ersetzt, welche eine gleiche Signatur aufweist. Ein Beispiel einer Überschreibung ist in Java in fast jeder Klasse zu finden: Die Methode *toString()* wird von einer Klasse so überschrieben, dass sie eine für diese Klasse geeignete Ausgabe liefert. Das Verhalten der vermeintlich geerbten Methode unterscheidet sich jedoch von jenem der Methode der Basisklasse; die abgeleitete Klasse reagiert in dieser Hinsicht anders als die vererbende Klasse.[4]

3.3.3 Subklassen und Subtypen

Jede Klasse gibt Anlass zur Definition eines Typs; dieser erklärt durch sein Prädikat alle Instanzen der Klasse als zu diesem Typ gehörig. Instanzen von Subklassen sind aber immer auch Instanzen einer Klasse, von der die Subklasse abgeleitet wurde. Der Typ, welcher der Subklasse zugeordnet ist, definiert somit eine Teilmenge des Typs, welcher der ursprünglichen Klasse angehört. Man spricht daher von einem *Subtyp*.

Da sich die Menge der Instanzen des Typs durch ein Prädikat definiert, ist die Men-

[4] Die Sprache Java ermöglicht den Aufruf der Methode der Superklasse mittels *super.methode()*.

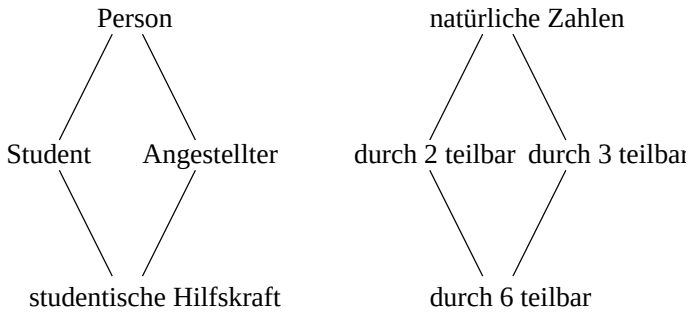

Abbildung 3.4: Subtyphierarchien

ge der Instanzen des Subtyps durch eine Verschärfung des Prädikats erklärt. Man unterscheidet dabei zwei Arten von Subtypbildung [LW93]:

- Einschränkung des Wertebereichs (Einschränkungssubtyp)
- Aggregation weiterer Methoden und Felder (Erweiterungssubtyp)

Abbildung 3.4 zeigt den Zusammenhang grafisch. Beide Graphen stellen eine Subtypbildungshierarchie dar. Links wird eine allgemeine Person durch die Eigenschaft *Student* erweitert; alternativ kann sie zu einem *Angestellten* erweitert werden. Beide Eigenschaften treffen sich im Begriff *studentische Hilfskraft.* Auf der rechten Seite ist erkennbar, wie eine Menge durch Subtypbildung in ihren Werten eingeschränkt wird.

Um den Unterschied beider Subtypbildungen zu erkennen, muss man sich die Implementierung in Form von Klassen vergegenwärtigen. Die Klasse einer *Person* kann Felder wie *Name, Alter, Geschlecht, Nationalität* und andere beinhalten. Ein *Student* besitzt darüber hinaus Informationen über *Studienfach, Semesterzahl* und *Hochschule.* Ein *Angestellter* schließlich definiert Angaben wie *Gehalt, Dienststelle, Vorgesetzter.* Klassen, welche diese Typen repräsentieren sollen, werden durch Ableitung gebildet. Je tiefer in die Ableitungshierarchie hinabgestiegen wird, umso mehr Felder werden hinzugefügt.

Der rechte Typ ist in seiner Subtypbildung nicht ohne weiteres durch Klassen repräsentierbar. Das Prädikat *ist eine natürliche Zahl* wird schrittweise durch Hinzufügen weiterer Prädikate verschärft: *ist durch 2 teilbar* sowie *ist durch 3 teilbar.* Beide Prädikate konjunktiv verknüpft ergeben die Teilbarkeit durch 6.

Beurteilt man jedoch die Subtypbildung nach dem zu Grunde liegenden Prädikat und dessen Verschärfung, so bietet sich ein einheitliches Bild: Jeder Student ist eine Person; jeder Angestellte ist eine Person. Die Menge der Personen umfasst diese verschiedenen Subklassen. Jede studentische Hilfskraft verfügt über alle Merkmale, welche in der Ursprungsklasse *Person* festgelegt wurden, also etwa über einen Namen.

> Das Hinzufügen weiterer Methoden und Felder in eine Klasse schränkt die Menge der möglichen Instanzen ein.

```
public interface AMETASPlaceUserDriverIf {
   ...
   AMETASMessage[] getMessages(boolean bDelete);
   ...
}

public interface AMETASAgentDriverIf extends
          AMETASPlaceUserDriverIf {
   ...
   void go(String sPlace, long nRelayTime) throws ...;
   ...
}
```

Abbildung 3.5: Schnittstellenvererbung

Die beiden vorgestellten Arten von Subtypbildung unterscheiden sich damit lediglich in der Weise, wie die grundlegende Menge eingeschränkt wird. Eine Klasse definiert somit nur die *Mindestmenge* an Methoden und Feldern, die eine Instanz haben muss, um der Klasse anzugehören. Analog verwendet ein Typ das *schwächste Prädikat* aller seiner Subtypen. Man erhält unmittelbar die Aussage:

Regel 3.1 Erwartungsprinzip des Subtyps
Eine Instanz eines Subtyps verhält sich jederzeit so, wie man es von einer Instanz des Typs erwartet.

Eine durch sechs teilbare natürliche Zahl kann immer verwendet werden, wenn irgendeine natürliche Zahl benötigt wird. Ebenso können studentische Hilfskräfte stets als Personen gewählt werden. Diese Charakterisierung – welche gerade die Verschärfung des Prädikats umschreibt – ist wesentlich für alle weiteren Betrachtungen von Subtypen in dieser Arbeit.

3.3.4 Schnittstellenvererbung

Anstatt die Vererbung auf die Wiederverwendung einer Implementierung festzulegen, ist es vor allem bei Systemen verteilter Objekte oder Prozeduren von Interesse, lediglich eine Schnittstelle wiederzuverwenden; man spricht auch von der *Schnittstellenableitung*. Die Implementierung, welche die abgeleitete Schnittstelle implementiert, gibt eine Zusage, neben der Implementierung der Methoden der Ableitung auch die Methoden der vererbenden Schnittstelle zu implementieren. Abbildung 3.5 zeigt ein Beispiel einer in Schnittstellenvererbung in Java.

Für eine Implementierung von *AMETASAgentDriverIf* ist es notwendig, die Methode *getMessages* zu implementieren, da diese Methode in der Elternschnittstelle auftaucht. Andererseits muss ein Objekt, das *AMETASPlaceUserDriverIf* implementiert, nicht die *go*-Methode bereitstellen.[5] Schnittstellenvererbung ermöglicht größere Spielräume in der Implementierung von Objekten, welche ihre Dienste anderen, möglicherweise entfernten Objekten anbieten, da die konkrete Implementierung nicht festgeschrieben ist. Im Gegenzug kann sich der Klient aber nicht sicher sein, dass die Implementierung sich wie erwartet verhält, auch wenn sie seine Nachrichten wie zugesagt akzeptiert. Schnittstellen abstrahieren von der Implementierung, liefern jedoch ein *unterspezifizierendes Prädikat*, während Klassen eine *Überspezifizierung* bewirken [WZ88, Grü97].

3.4 Vergleich von Typen

Die bislang vorgestellten Typen sind einfache Datentypen oder Typen, welche durch Klassen definiert sind. Subtypen repräsentieren Teilmengen der von den Typen definierten Mengen. Programmiersprachen wie C++ oder Java erklären Subklassenbeziehungen ausschließlich auf Basis eines Schlüsselwortes wie *extends*, was streng genommen den Umfang der möglichen Subklassen erheblich einschränkt: Würde die Subklasse die Methoden und Felder der Klasse selbst aufführen, ohne ihre Abstammung zu deklarieren, so würde sie nicht als Subklasse erkannt.

Sollen Typen miteinander auf Subtypbeziehungen verglichen werden, so interessiert weniger die Abstammung als vielmehr die *Eignung* des aktuellen Typs, an die Stelle eines anderen Typs zu treten. Dies kann anhand des Prädikats entschieden werden; ist das gegebene Prädikat eine Verschärfung des erwarteten Prädikats, so kann die zugehörige Instanz akzeptiert werden.

Schwierig ist es, diese Prädikate algorithmisch zu vergleichen, da ihre Struktur beliebig kompliziert sein kann und möglicherweise nicht einmal berechenbar ist. Daher ist es wichtig, Prädikate so zu formulieren, dass eine Auswertung einfach zu implementieren ist. Dies gelingt insbesondere für Datentypen: Wenn ein Prädikat Aussagen über Datentypen beinhaltet, lassen sich Regeln definieren, wie diese zu vergleichen sind. Voraussetzung aller Vergleiche ist eine Korrespondenz zwischen der Repräsentation der Datentypen und den Datentypen selbst. Beispielsweise muss eindeutig festgelegt sein, in welcher Weise der Datentyp *boolean* repräsentiert wird, etwa als Zeichenkette „boolean“ oder „bool“.

[5] Im konkreten Falle implementiert *AMETASAgentDriver* die entsprechende Schnittstelle; da diese Klasse von *AMETASPlaceUserDriver* abgeleitet ist, erfüllt sie die Anforderungen, da *AMETASPlaceUserDriver* die Methoden seiner Schnittstelle implementiert.

3.4.1 Vergleich von Datentypen

Die den Feldern eines Objekts zugehörigen Informationen sind der *Name* des Feldes und der *Datentyp*, also etwa *int i* oder *var i:int*, je nach Syntax der Programmiersprache. Die Programmiersprachen definieren Subtypbeziehungen zwischen diesen Basistypen. Dies betrifft insbesondere die numerischen Typen

- *long* $\succ$ *int* $\succ$ *short*
- *double* $\succ$ *float.*

Ein boolescher Wert kann keiner numerischen Variablen zugewiesen werden – es sei denn, die Sprache sieht eine explizite Konversion vor. Jedoch kann eine *long*-Variable stets mit einem *short*-Wert belegt werden.

3.4.2 Typen für Methoden und Prozeduren

Auch Methoden (oder Prozeduren in nicht-objektorientierten Kontexten) können einen Typ aufweisen. Dies ist für den Vergleich zweier Objekte wichtig, da ein Kriterium existieren muss, nach dem Methoden miteinander verglichen werden können. Da Methoden im Allgemeinen einen Rückgabewert liefern, liegt nahe, dass der Rückgabewert den Typ der Methode beeinflusst. So ist

int meth1() $\succ$ *short meth1(),*

was mit der Vorstellung konform ist, dass vermöge

int j=meth1();

der *int*-Variablen *j* nur ein Subtyp von *int* zugewiesen werden kann. Für die Eingangsparameter drehen sich jedoch die Verhältnisse um. Da ihnen beim Aufruf Werte zugewiesen werden und sie keine Werte entgegennehmen, stehen sie auf der linken Seite einer (impliziten) Wertzuweisung. Demnach ist

void meth2(int) $\succ$ *void meth2(long),*

was wiederum verständlich wird, wenn man sich überlegt, dass der Eingangsparameter des Typs *long* stets alle Werte entgegennehmen kann, die dem *int*-Bereich angehören. Diese beiden Bedingungen sind kombinierbar, da sie voneinander unabhängig sind. Sie ergeben das Prinzip der *Ko- und Kontravarianz:*

Definition 3.2 Kovarianz und Kontravarianz
Der einer Methode *meth1* zugeordnete Typ ist genau dann Subtyp des einer Methode *meth2* zugeordneten Typs (geschrieben: $meth1 \prec meth2$), wenn gilt:

- $eing_i^2 \prec eing_i^1 \; \forall i = 1, \ldots, n$ (Kontravarianz der Eingaben),
- $ausg^1 \prec ausg^2$ (Kovarianz der Ausgaben),

wobei $eing_i^j$ der Typ der *i*-ten Eingabevariable der Methode *meth*j und $ausg^j$ der Typ des Rückgabeparameters der Methode *meth*j darstellt.

3.4.3 Syntaktischer Typvergleich

Die in verteilten Infrastrukturen gebräuchlichen Schnittstellenbeschreibungssprachen (siehe Abschnitt 3.2.3) definieren jeweils eigene Regeln für den Vergleich von Schnittstellen. Von Interesse ist, ob ein Objekt konform zu einer Schnittstelle ist, das heißt der Objekttyp konform zum Schnittstellentyp ist. In diesem Falle wird davon gesprochen, dass das Objekt die Schnittstelle implementiert. Syntaktische Typvergleiche können gewöhnlich nur zwischen Beschreibungen derselben Sprache stattfinden; PUDER gibt in [Pud97] ein Verfahren an, einen Vergleich von Beschreibungen verschiedener Sprachen unter Zuhilfenahme eines Zwischenformats zu realisieren.

DCE

Im Falle von DCE IDL sind die Bedingungen an die Konformität recht strikt. Eine Typkonformität basiert in DCE auf dem Inklusions-Polymorphismus, wobei die Reihenfolge der Signaturen nicht geändert werden darf. Dies bedeutet, dass ein Typ T_1, welcher zu einem Typ T_2 konform sein soll, alle Signaturen von T_2 in unveränderter Reihenfolge aufführen muss, aber bei Bedarf weitere Signaturen anfügen kann. DCE ermöglicht somit eine Schnittstellenvererbung, erlegt aber der Implementierung keine Bedingungen auf. Die Austauschbarkeit zur Laufzeit hängt von der Gleichheit der verwendeten UUIDs ab; ferner muss die Versionsnummer der Erweiterung höher sein. Der Inklusions-Polymorphismus von DCE wird daher auch als *Versionierung* bezeichnet.

OMG IDL

Anders als DCE erlaubt CORBA, dass die Methoden in einer OMG IDL beliebig permutiert werden können, ohne die Konformität zu stören. Die Konformität hängt somit von folgenden Faktoren ab: Zu jeder Methode f_2 des Typs T_2 muss es in T_1 eine Methode f_1 geben, sodass

- die Methodennamen von f_1 und f_2 identisch sind;

- die Anzahl der Eingabeparameter sowie deren entsprechende Typen bei beiden Methoden gleich sind und
- die Anzahl der Ausgabeparameter sowie deren entsprechende Typen bei beiden Methoden gleich sind.

Die Typkonformität ist von der Spezifikation einer Schnittstellenableitung unabhängig. In CORBA kommen Schnittstellen ohne UUID oder Versionsnummer aus; sie werden zur Laufzeit in einem Schnittstellenverzeichnis angeboten [OMG00a].

Java

Die Schnittstellenkonformität von Java fällt gegenüber den anderen Versionen aus dem Rahmen, da Java-Schnittstellen Teil der Java-Sprachumgebung sind und zur Laufzeit als eigenständige Entitäten existieren. Ähnlich wie in OMG IDL spielen die Reihenfolge der Methodensignaturen bei Java keine Rolle, die Signaturen müssen aber paarweise gleich sein.

Java definiert die Schnittstellenableitung durch das Schlüsselwort *extends* in der Schnittstellendeklaration. Diese Ableitungsbeziehung ist von der Typkonformität unabhängig. Allerdings existiert zur Laufzeit eine strenge Bindung von Objekten zu den von ihnen implementierten Schnittstellen: Ein Objekt kann nur genau jene Schnittstellen implementieren, die in der *implements*-Klausel seiner Klassendeklaration aufgeführt sind, einschließlich aller in der Vererbungshierarchie vorangehenden Basisschnittstellen. Die Tatsache, dass zwei Schnittstellen einander syntaktisch gleichen, ist somit für die Laufzeit unerheblich, wenn das Objekt eine der beiden Schnittstellen nicht explizit implementiert.

3.4.4 Typübersetzung

Der in der Definition von Abschnitt 3.4.2 erklärte Vergleich von Methoden ist nur dann sinnvoll, wenn es eine eindeutige Zuordnung zwischen den Methoden des gegebenen Typs und des zu vergleichenden Typs gibt. Diese Zuordnung wird anhand der *Methodennamen* vorgenommen. Zwei Methoden gelten dann als einander zugehörig, wenn ihr Methodenname identisch ist.

Analoges gilt für die Felddeklarationen. Ein Feld eines Objekts kann ähnlich einer Methode gesehen werden, welche gerade den Typ des Feldes als Rückgabeparameter oder bei Zuweisungen als Eingangsparameter aufweist. Daher werden Felder ebenfalls anhand ihrer Namen zugeordnet. Daraus ergibt sich für den Vergleich zweier Objekttypen folgende Bedingung für die Subtypbeziehung:

1. Alle Feldnamen des Typs müssen im Subtyp verfügbar sein.
2. Alle Methodennamen des Typs müssen im Subtyp verfügbar sein.

3. Jedes Feld des Subtyps, das mit einem Feld des Typs korrespondiert, muss den gleichen Typ beinhalten.

4. Jede Methode des Subtyps, die mit einer Methode des Typs korrespondiert, muss Subtyp dieser Methode gemäß den Ko-/Kontravarianzregeln sein.

Die dritte Bedingung resultiert unmittelbar aus der Analogie eines Feldes zu einer Methode. Der Typ des Feldes muss mindestens so groß sein, dass er jeden Wert des Feldes aufnehmen kann, das der Typ deklariert. Umgekehrt darf das Feld keine Werte liefern, die als Inhalt des Feldes des Typs nicht erwartet sind. Diese Vergleichsvorschrift entspricht gerade der durch ODP [ITU95] definierten Vorschrift.

LISKOV und WING beschreiben in [LW93] eine Verallgemeinerung dieser Vergleichsvorschrift. Dabei wird die Bedingung der Namensübereinstimmung gelockert: Definiert wird eine *Korrespondenzabbildung* $\langle A, R, E \rangle$, welche eine *Abstraktionsabbildung A*, eine *Umbenennungsabbildung R* sowie eine *Erweiterungsabbildung E* beinhaltet.

- Die Abstraktionsabbildung bildet Werte (in diesem Sinne Objektzustände) des Subtyps auf jene des Typs ab. Sie muss total und surjektiv sein, kann jedoch mehrere Subtypwerte auf denselben Typwert abbilden.

- Die Umbenennungsabbildung stellt eine Korrespondenz zwischen den Namen von Feldern und Methoden des Subtyps zu jenen des Typs her. Sie ist in der Regel partiell und muss bijektiv auf dem definierten Bereich sein.

- Die Erweiterungsabbildung ordnet jeder Methode eines Subtyps, die neu hinzukommt, in Abhängigkeit zum Argument und der Instanz des Typs ein Programm zu, welches die Wirkung der neuen Methode als Komposition der im Supertyp bekannten Methoden erklärt.

Die Erweiterungsabbildung ist laut Liskov und Wing notwendig, um das Verhalten eines Subtyps zu erklären, wenn Methoden ausgeführt werden, welche der Typ nicht kennt. Wenn man beispielsweise ein Objekt erwartet, das einen Stapelspeicher implementiert und die Methoden *push* und *pop* kennt, dann wäre es sehr überraschend, wenn in Folge einer neuen Methode *empty* eines Subtyps der Stapel plötzlich geleert würde. Diese Funktionalität würde die Erweiterungsabbildung als wiederholte *pop*-Operation erklären.

Anders sieht der Fall einer Menge aus, welche stetig wächst, da sie keine Operation zum Löschen anbietet. Würde man nun einen Subtyp an deren Stelle setzen, der ein Löschen ermöglicht, so könnte diese Operation mit keiner Verkettung bekannter Operationen erklärt werden. Laut [LW93] kann man in diesem Falle nicht von einem Subtyp reden:

> *Wenn [der Subtyp] σ Methoden hinzufügt, welche zu keiner Methode von [dem Typ] τ korrespondieren, müssen wir einen Weg finden, diese neuen Methoden zu erklären … Wir müssen einen Weg aufzeigen, wie das Verhalten der neuen Methoden durch das Verhalten der bereits in τ definierten Methoden erklärt werden kann.*

Jedoch muss man sich vor Augen führen, dass die Annahme, dass diese Menge nicht schrumpfen kann, eine aus der Verfügbarkeit der Methoden des Typs hergeleitete Aussage ist. Das dem Typ zugehörige Prädikat ist sinnvollerweise *nicht* so zu wählen, dass es die Anfügung weiterer Eigenschaften verhindert; daher wurde in Abbildung 3.1 auch nur ein Teilmengensymbol und keine Gleichheit verwendet. Das Prädikat sollte nur aussagen, dass diese Art von Objekt eine Hinzufügung erlaubt; dass deshalb ein Anwachsen der Menge für jeden Vertreter des Typs angenommen werden kann, impliziert das Nichtvorhandensein einer Löschmethode in einem Subtyp. Die Erweiterungsabbildung wird aus diesem Grunde keine weitere Rolle in der Betrachtung von Subtypen in dieser Arbeit spielen.

Die Umbenennungsabbildung ist hingegen von großer Bedeutung: Ohne sie muss es eine feste Zuordnung zwischen Namen einer erwünschten Funktionalität geben. Um einen speziellen Dienst von einem Objekt erbracht zu bekommen, muss der Name des Dienstes bekannt sein, auch wenn die übergebenen Parameter oder der Aufbau der Schnittstelle eine eindeutige Zuordnung bereits implizieren. Dies ist vor allem wichtig, wenn Nachrichten an das Objekt über einen einzigen Kanal geleitet werden, wie es bei der asynchronen Übertragung von Nachrichten zwischen Agenten geschehen kann.

Laut Definition der Abstraktionsfunktion muss der Zustand des Subtyps auf einen entsprechenden Zustand des Typs abgebildet werden können. Dies ist einsichtig, wenn man den Zustand durch direkte Abfrage der Felder vergleichen kann: Die zusätzlichen Felder werden einfach ignoriert, wenn die Instanz des Subtyps wie eine Instanz des Typs behandelt wird. Die Abbildung muss aber keine reine Projektion sein; die Aufgabe der Übersetzung übernehmen die Methoden, welche auf den entsprechenden Teil des Zustands zugreifen. Damit wird jedoch auf die Implementierung dieser Methoden Bezug genommen; zumindest muss die Semantik übereinstimmen. Anschaulich beschrieben wird gefordert, dass nach einer Reihe von Operationen derselbe *Wert* bei einer Zustandsabfrage geliefert wird. Wenn auch diese Information in vielen Zusammenhängen wichtig ist, so ist sie für die Frage der Kommunikationskompatibilität unerheblich.

3.5 Wissensrepräsentation

Jegliche Beschreibung eines Objekts stellt eine Repräsentation von *Wissen* dar, sofern Wissen im Sinne von *Information* verstanden wird. So wird aus einer Schnittstellenbeschreibung ersichtlich, welche Methoden ein Objekt anbietet, ohne dass dies durch Ver-

suche herausgefunden werden müsste. Eine textuelle Beschreibung kann Aufschluss über den vorgesehenen Einsatz geben.

Eine Wissensrepräsentation im engeren Sinne, wie sie hier zu verstehen ist, soll *semantische* Informationen über das betreffende Objekt wiedergeben. Syntaktische Eigenschaften des Objekts betreffen die *Form* der Kommunikation mit dem Objekt sowie der Erzeugung und Entsorgung des Objekts. Sie treffen keine Aussagen über den *Inhalt* der Nachrichten sowie deren *Bedeutung.* Diese nicht-syntaktischen Merkmale können von entscheidender Bedeutung sein: Bewirkt der Aufruf einer bestimmten Methode die beabsichtigte Zustandsänderungen und Aktionen? Ist das Objekt für den geplanten Einsatz überhaupt vorgesehen?

3.5.1 Semantische Information

Die nicht der Syntax (also der Form) zugehörigen Informationen zu einem Objekt werden dessen *Semantik* zugeordnet. Die Repräsentation semantischer Informationen ist sehr viel schwieriger als im Falle syntaktischer Informationen, weil sie einer formalisierten Darstellungsweise von ursprünglich nicht-formalisierter Information bedarf.

- Diese Methode dient dem Ausdruck des Zustands des Objekts in Tabellenform.
- Dieses Objekt sammelt Werte von einem SNMP-Agenten und berechnet die mittlere Netzbelastung.
- Dieses Feld beinhaltet den Namen des Anwenders.

Diese Beispiele zeigen, dass die *Bedeutung* einer Entität, sei es ein Objekt oder eines seiner Teile, mannigfaltig beschrieben sein kann. Es fällt sehr viel leichter, mittels der menschlichen Sprache eine Beschreibung zu liefern, da die Objekte (zumindest derzeit) von Menschen erschaffen werden, welche die genannte Semantik von vornherein kennen und die Komponente entsprechend gestalten.

Soll diese Information weiteren Anwendern zugute kommen, muss sie so formuliert sein, dass diese die Information erfassen können. Dabei ist insbesondere auf die Zielgruppe zu achten: Anwender haben meist eine andere Sichtweise auf dasselbe Objekt als ein Implementierer. Dieselbe Beschreibung kann somit unterschiedlichen Nutzen haben; sie muss geeignet *interpretiert* werden können. Voraussetzung ist

1. die Formulierung in einer Sprache, die der Adressat versteht, insbesondere unter Verwendung eines ihm geläufigen Vokabulars;
2. der Bezug auf Sinnzusammenhänge, die der Ersteller und der Empfänger der Information teilen.

Es kann nämlich vorkommen, dass der Empfänger die der Beschreibung zu Grunde liegende Information mangels Fachwissen nicht erschließen kann, obwohl er jedes Wort der Beschreibung versteht. Die Begriffswelt, insbesondere die Bedeutung der in ihr vorhandenen Begriffe, legt eine so genannte *Ontologie* fest.

Der menschliche Verstand geht zur Erkennung der Bedeutung einer Beschreibung intuitiv, aber meist korrekt vor, da die Erzeugung einer solchen Beschreibung gerade auf den gleichen Mechanismen beruht, die später bei der Rezeption benötigt werden. Die erforderlichen Mechanismen zur Abstraktion von Information sind offenbar bei allen Menschen gleich, sind den meisten Menschen aber völlig unbewusst. (Eine genauere Darstellung der Konzeptionalisierung von Informationen auf unterschiedlichen Ebenen der Abstraktion – so genannte *Level-Bands* – ist beispielsweise in [Min85] zu finden.)

Eine Verwendung solchen Wissens in einer Anwendung erfordert eine explizite Formalisierung. Es muss möglich sein, die den Beschreibungen zu Grunde liegenden Informationen so zu formulieren, dass diese Beschreibungen in gleicher Weise interpretiert werden. Eine der viel versprechenden Möglichkeiten stammt aus dem Gebiet der Wahrnehmungspsychologie, die so genannten *Konzeptgraphen* [Sow84]. Diese werden unten genauer vorgestellt.

3.5.2 Zeichenketten

Eine Möglichkeit, Wissen zu repräsentieren, ist die Verwendung von *Bezeichnern.* Voraussetzung ist, dass es einen Konsens gibt, welche Semantik welchem Bezeichner zugeordnet ist. Die Zuordnung solcher Bezeichner, welche als Zeichenketten repräsentiert werden, basiert auf dem Prinzip der *Assoziation,* also dem Verknüpfen von weiteren (semantischen) Informationen an eine bestimmte Schlüsselinformation, dem Bezeichner:

- Eine Methode zur Ausgabe des Zustands eines Objekts wird in Java in der Regel mit *toString()* bezeichnet.
- Ein Agent, welcher die Netzbelastung feststellen soll, könnte *NetworkLoadMonitorAgent* genannt werden.
- Ein Feld mit dem Namen des Benutzers kann *Benutzername* heißen.

Offenbar sind zahlreiche Vorbedingungen notwendig, soll die Semantik auf dieser Basis repräsentiert werden. Beispielsweise wäre eine einheitliche Sprache (meist Englisch) festzulegen, um eine Diskrepanz zwischen *Benutzername* und *UserName* zu vermeiden.

Die Möglichkeiten der Formulierung innerhalb der menschlichen Sprache sind allerdings so zahlreich, dass ohne weitere Übereinkommen viele Übereinstimmungen übersehen werden. Die Methode zur Ausgabe des Zustands eines Objekts könnte auch

dumpState genannt werden; der Agent könnte auch *NetLoadAgent* heißen. Andererseits liefert die Angabe „den Zustand ausgeben“ keine eindeutige Beschreibung des folgenden Vorgangs. Es ist somit nicht zu erreichen, dass es eine 1-zu-1-Abbildung zwischen Bezeichnern und der Semantik eines Objekts gibt; man muss sogar verschiedene *Deutungen* miteinander identifizieren, um überhaupt eine Abbildung zu erhalten. Diese Identifizierung kann eine Beurteilung der Kompatibilität vereiteln, da sich bestimmte Eigenschaften wie die Kommunikation mit einer bestimmten Komponenten nur begrenzt abstrahieren lässt.

Zeichenketten können einerseits für eine *Klasse* von Dingen stehen (wie die eben genannten Beispiele) oder konkrete Werte tragen wie „WebSucher:Suchagent1“. Die Unterscheidung ist letztlich eine Frage der Interpretation, welche vom Anwender, aber nicht vom System durchgeführt wird. Sie ergibt sich daraus, dass eine „Bestellung“ im gewöhnlichen Sprachgebrauch *unterspezifiziert* ist, also weiterer Informationen bedarf, um identifiziert werden zu können. Hingegen ist eine „Abbruchnachricht“ für die meisten Kontexte genau genug spezifiziert.

Bereits diese ersten Beispiele offenbaren die größte Schwäche einer zeichenkettenbasierten Repräsentation: Die Repräsentation selbst unterliegt einer externen Interpretation. Eine Zeichenkette kann selbst nur eine bestimmte Gegebenheit *benennen*, die im jeweiligen Kontext sinnvoll ist, sie aber nicht *erklären*. Deutlich wird dies an den zahlreichen zweideutigen Begriffen *(Homonymen)* der menschlichen Sprache wie

- *Bahn*: im Personentransport – aber auch in der Astronomie, Physik;
- *Bank*: im Kreditwesen – aber auch in der Innenarchitektur;
- *Aufgabe*: die Niederlage – aber auch der Auftrag.

Umgekehrt können mehrere unterschiedliche Begriffe dieselbe Entität kennzeichnen. Man spricht dann von *Synonymen*. Beispiele sind *Programm* und *Code*, *Lampe* und *Leuchte* oder *Licht*, *Computer* oder *Rechner*, *Bahn* oder *Zug*.[6] Die unterschiedlichen *Sprachen* führen für praktisch jede Entität ein Reihe von Synonymen ein, welche durch Übersetzung ineinander übergehen.

Die semantischen Angaben sind also nur dann brauchbar, wenn sie interpretiert werden und dabei insbesondere Synonyme erkannt werden sowie der jeweilige Kontext spezifiziert ist. Je größer der Wortschatz zur Verwendung in den Angaben ist, umso aufwändiger wird die Auswertung, die im Allgemeinen nur vom Menschen selbst geleistet werden kann. Soll die Auswertung aber algorithmisch erfolgen, so bleibt nur die Einschränkung des verfügbaren Vokabulars.

[6] Wichtig ist auch hier der jeweilige Kontext; vor einem Fernseher sitzend betrachtet man eher selten einen *Code*.

> Möchte man eine semantische Information mittels einer einfachen (strukturlosen) Zeichenkette repräsentieren, so muss entweder der Gültigkeitsbereich der Information oder die Menge möglicher Informationen beschränkt werden.

In ersterem Falle würde man eine bestimmte Bezeichnung als zu nur einer bestimmten Anwendung zugehörig erachten, also beispielsweise „AS1Kaufagent“. Möglich ist dies, wenn es sich um eine *geschlossene* Anwendung handelt. Die Interpretation der Bezeichnung liegt dann implizit durch die Implementierung der Anwendung vor.

Die Verwendung von Bezeichnern ist nur in geschlossenen Systemen empfehlenswert, die eine Propagation der Bedeutung dieser Bezeichner erlauben. Offene Systeme – wie sie gerade bei mobilen Agenten diskutiert werden – erlauben schon aufgrund der Eigenschaft, dass ständig neue Komponenten auftauchen können, keine verlässliche Propagation solcher Informationen: Offene Anwendungen, an denen auch ursprünglich unbekannte Komponenten teilnehmen können, erfordern die Bekanntgabe aller verwendeten Begriffe im Voraus, damit überhaupt eine Formulierung von akzeptablen Repräsentationen möglich wird und Verwechslungen minimiert werden. Neue Agenten könnten sich schneller im Netz als ihre zugehörigen Beschreibungen verbreiten.

Eine andere Möglichkeit, die Aussagekraft von zeichenkettenbasierten Beschreibungen zu erhöhen, ist die Definition einer *Struktur* auf den Zeichenketten. Prinzipiell lassen sich auf diese Weise alle Repräsentationen realisieren, welche über ein lineares Format verfügen, etwa KIF [GF92, Gen95], eine LISP-ähnliche Notation oder auch XML [W3C00]. Klar ist, dass jede Komponente des Systems, welche Nutzen aus den Annotationen ziehen soll, eine Möglichkeit haben muss, diese strukturierten Zeichenketten auszuwerten.

3.5.3 KIF

Das *Knowledge Interchange Format (KIF)* ist eine Wissensrepräsentationstechnik, welche im Zusammenhang mit der Erforschung von Agentenkommunikationssprachen entstand. Insbesondere findet sie Beachtung als Inhaltssprache der *Knowledge Query and Manipulation Language (KQML)* [FW91, Lab96]. Abbildung 3.6 zeigt ein Beispiel einer KIF-basierten Beschreibung aus [Gru95]. Sie beschreibt eine physikalische Größe als eine Klasse, deren Instanzen sowohl einen Wert *(magnitude)* beinhalten, welcher vom Datentyp *double-float* ist, als auch eine Einheit aus den SI-Basiseinheiten. Die Notation *?q* bezeichnet in KIF freie Variablen.

Wie der Name schon sagt, ist KIF in erster Linie zum *Wissensaustausch* vorgesehen, nicht jedoch zum Speichern von Wissen [GF92, Gen95]. Objekte, die via KIF-Nachrichten kommunizieren, müssen dementsprechend eine Übersetzung des Wissens in ihre internen Repräsentationen zu KIF und zurück leisten.

Die Darstellung hat große Ähnlichkeiten mit Programmen, die in der Sprache LISP

```
(defrelation PHYSICAL_QUANTITY
   (<=> (PHYSICAL_QUANTITY ?q)
        (and (defined (quantity.magnitude ?q))
             (double-float (quantity.magnitude ?q))
             (defined (quantity.unit ?q))
             (member (quantity.unit ?q)
                     (set-of meter second kilogramm
                             ampere kelvin mole)))))
```

Abbildung 3.6: Beispiel in KIF

geschrieben sind. Die umständlich anmutende Wissensformulierung als Prädikatenkalkül trägt sicherlich dazu bei, dass die Sprache KIF relativ gering verbreitet ist.

3.5.4 Konzeptgraphen

Aus dem Forschungsgebiet der Wahrnehmungspsychologie stammt ein interessanter Ansatz, der in dieser Form zuerst von SOWA in [Sow84] aufgegriffen wurde. So deuten viele Phänomene darauf hin, dass der menschliche Verstand die Welt als ein System wahrnimmt, das aus *Dingen* besteht, die untereinander in einem *Verhältnis* stehen. Diese Dinge können konkrete Gegenstände, aber auch abstrakte Vorstellungen sein. Einen Hinweis liefern menschliche Sprachen: Die Grammatik ist selbst in recht unterschiedlichen Sprachfamilien so beschaffen, dass ein *Subjekt* durch ein *Prädikat* mit einem oder mehreren *Objekten* in Verbindung gebracht wird.

Sowa entwickelte, wie er in [Sow84] beschreibt, eine Repräsentation, welche er *Konzeptgraphen* nennt. Es handelt sich dabei um eine grafische Repräsentation von Informationen, die gleich Mehrfaches leistet:

- Sie repräsentiert Assoziationen.
- Sie repräsentiert Abstraktion und Spezialisierung.
- Sie verbindet so genannte *Konzepte* mit *Relationen* untereinander und spiegelt damit die im menschlichen Verstand angenommene Wissensrepräsentation wider.

Konzeptgraphen sind – formal beschrieben – bipartite[7] Graphen. Die Knotentypen heißen *Konzepte* und *Relationen*. Konzepte stellen die Objekte der Wahrnehmung dar, seien sie abstrakt oder konkret. Relationen bringen solche Konzepte in eine wechselseitige Beziehung. Konzeptgraphen eignen sich, Sachverhalte zu erklären. Sie können

[7] Es gibt eine Partition der Knoten in zwei Klassen, wobei keine Kante zwei Knoten derselben Klasse verbindet.

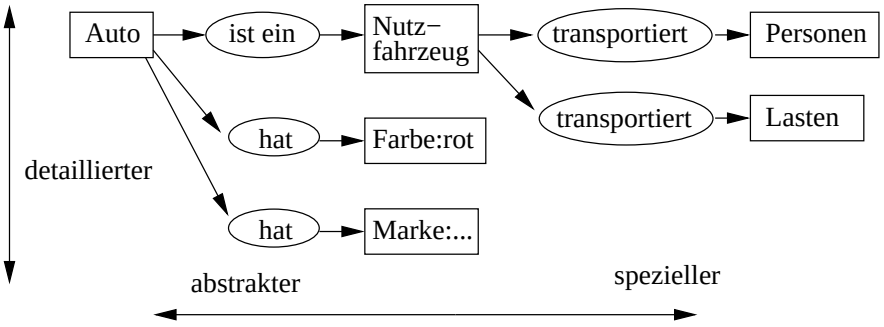

Abbildung 3.7: Konzeptbaum

Objekte des Vorgangs sowie Agenten (im Sinne von Ausführenden) oder auch Charakteristiken bezeichnen. Die Beschreibungen können zudem beliebig verfeinert werden, indem weitere Teilgraphen angehängt werden.

Konzeptgraphen können im Allgemeinen eine beliebige Struktur aufweisen, solange die Grundeigenschaften bewahrt sind. Allerdings wird sowohl die intuitive Deutung als auch die algorithmische Auswertung eines solchen Graphen sehr aufwändig. Eine mögliche Vereinfachung von Konzeptgraphen erhält man durch die Modellierung als Baum *(Konzeptbaum)*. Abbildung 3.7 zeigt hierzu ein Beispiel.

Wichtig ist die Idee der *Abstraktion*: Es ist ein besonderes Merkmal des menschlichen Verstands, Abstraktionen durchführen zu können [Min85]. Dadurch wird es ihm ermöglicht, eine Menge von Objekten als gleichartig zu erkennen. So weiß man, dass man Holz zum Feuermachen verwenden kann, ohne diese Erfahrung mit jeder Baumart einzeln machen zu müssen (auch wenn sich manche Arten besser eignen). Die umgekehrte Richtung ist die *Spezialisierung*. So werden aus einer Menge von *Menschen*, die sich auf einem Platz begegnen, ein paar *Deutsche*, *Engländer*, *Franzosen*; bei genauerer Betrachtung sind es *deutsche Kinder* und *englische Erwachsene* und so weiter.

Konzeptknoten beinhalten eine Zeichenkette, welche ein so genanntes *Konzept* bezeichnet. Dieses Konzept kann durch eine Auflistung von *Instanzen* genauer spezifiziert werden, ähnlich wie es bei den einfachen Zeichenketten definiert werden kann; in diesem Falle können sogar mehrere Instanzen genannt werden:

Person:{Peter,Maria}

Dieses Konzept bezeichnet genau zwei Entitäten, welche *Peter* und *Maria* genannt werden und *Personen* sind. Die Bedeutung des Begriffs *Person* muss systemweit eindeutig sein und es darf keine Doppeldeutigkeiten geben.

Formal lassen sich Konzeptgraphen so darstellen (eingeschränkt auf Baumstrukturen):

Definition 3.3 Konzeptgraphenrepräsentation
Ein Konzeptgraph *kg* ist eine Struktur mit folgender Definition:

- $kg = (k, I, \{rg_1, rg_2, \ldots, rg_n\})$ mit *Relationsteilgraphen* rg_i, *Konzeptnamen* $k \in N_K$, *Instanznamen* I und $n \geq 0$;
- $rg = (r, cg_1, cg_2, \ldots, cg_m)$ mit *Konzeptgraphen* cg_i und *Relationsnamen* $r \in N_R$ und $m \geq 1$.

Die Menge I der Instanznamen kann leer sein; für $n = 0$ liegt ein Blatt des Konzeptgraphen vor. Die Zahl m wird *Stelligkeit* der Relation genannt.

Man beachte, dass verschiedene Konzeptgraphen und Relationsteilgraphen gleiche Konzeptnamen beziehungsweise Relationsnamen aufweisen können ohne identisch zu sein. Konzept- und Relationsknoten wechseln sich auf jedem Weg von der Wurzel zu einem Blatt in der Abfolge ab; die Wurzel und jedes Blatt sind Konzeptknoten. Während die Relationsteilgraphen ungeordnet an einem Konzeptknoten hängen, gibt es eine Ordnung der Konzeptgraphen an den Relationsknoten.

Konzeptgraphen kennen eine *lineare Repräsentation*. Eine genaue Darstellung kann der Syntax der Typbeschreibung in Anhang A oder [Sow84] sowie [Pud97] entnommen werden. Anschaulich gilt für die textuelle Darstellung:

- Der Konzeptbaum wird mittels einer Tiefensuche (DFS) traversiert.
- Ein Konzeptknoten wird als [*Konzeptname*] notiert. Sind Instanzen des Konzepts angegeben, so schreibt man [*Konzeptname*:{*a*,*b*,*c*}] bei Zeichenketten, [*Konzeptname*: *Zahl*] bei Zahlen. Relationsknoten werden als (*Relationsname*) notiert.
- Geschwister-Teilbäume werden durch „-“ und „.“ eingeschlossen; die Teilbäume selbst sind durch Kommas getrennt.
- Relationen werden durch „->“ an Konzepte gebunden, ebenso Konzepte an Relationen.

Diese Vorschrift überführt den Konzeptgraphen von Abbildung 3.7 in folgende lineare Form:

```
[Auto]- ->(”ist ein”)->[Nutzfahrzeug]-
                          ->(transportiert)->[Personen],
                          ->(transportiert)->[Lasten].,
        ->(hat)->[Farbe:{rot}],
        ->(hat)->[Marke:{...}].
```

Eine Schwäche der Konzeptgraphen ist, dass Relationen ihrerseits Entitäten unserer Wahrnehmung sind. Als solche können sie von Konzepten repräsentiert werden. Die Struktur des Konzeptgraphen ist somit nicht eindeutig festlegbar. So könnte im Beispiel von Abbildung 3.7 das *Auto* durch die Relation *Marke* mit einem *Bezeichner* verbunden sein. Wenn auch keine Lösung dieses Problems, so doch zumindest ein *Versuch*, Ordnung zu schaffen, ist Sowas Vorschlag: Mit einem Satz von *Grundrelationen* kann jegliche Art von Relation zumindest annähernd beschrieben werden [Sow84]. Diese Grundrelationen sind in der Regel *dyadisch*, verbinden also genau zwei Konzepte miteinander.[8]

3.5.5 Ontologie

Der Begriff *Ontologie* ist ein philosophischer Terminus und beschreibt die *Lehre vom Sein, von den Ordnungs-, Begriffs- und Wesensbestimmungen des Seienden* [Dud90]. Die Ontologie macht Aussagen über das Wesen der Existenz von Dingen in der Welt. Nach Gruber [Gru95] ist eine Ontologie eine *explizite Spezifikation einer Konzeptionalisierung*. Von dieser Vorstellung abgeleitet ist der Begriff einer Ontologie als Sammlung von Informationen, auf denen Aussagen definiert werden können. Sie beschreibt eine *Wissensbasis*, die zur Deutung einer Aussage herangezogen werden muss. Sind Ontologien zu klein, dann sind manche Phänomene nicht beschreibbar; sind Ontologien zu groß, dann besteht die Gefahr von Missdeutungen.

Ontologien müssen derart beschaffen sein, dass sie nicht nur Begriffe *nennen*, sondern auch *erklären*. Eine mögliche Form einer Ontologie ist die *Begriffshierarchie*. Diese ähnelt den Konzeptgraphen in der Hinsicht, dass die Begriffe miteinander verknüpft sind und dabei in Form eines Baumes an dessen Wurzel den abstraktesten Begriff, an den Blättern hingegen die konkretesten Begriffe aufführen. Ein Beispiel zeigt Abbildung 3.8 anhand einiger konkreter Begriffe. Es wird klar, dass Ontologien nur schwer vollständig erfassbar sind, wenn sie ein sehr breites Themenfeld abdecken sollen.

Eine solche Begriffshierarchie *erklärt* Begriffe wie folgt:

- Ein *Mensch* ist ein *Lebewesen*.
- *Wasser* und *Hunde* haben gemein, dass sie etwas *Natürliches* darstellen.
- Wenn man sich mit *Literatur* beschäftigen möchte, kann man *Prosa*, aber nicht *Bauwerke* studieren.

Es ist offensichtlich, dass in vielen Fällen Querverbindungen fehlen. Indes ist die Aussagekraft einer solchen Ontologie für die meisten Anwendungen ausreichend – und sie lässt sich *algorithmisch* auswerten.

[8] Relationen, welche mehr als zwei Konzepte verknüfen, sind eher selten; so sind zum Beispiel in den europäischen Sprachen nur wenige Präpositionen vorhanden, welche eine Beziehung zwischen mehreren Objekten kennzeichnen (wie „*zwischen*“) [Min85].

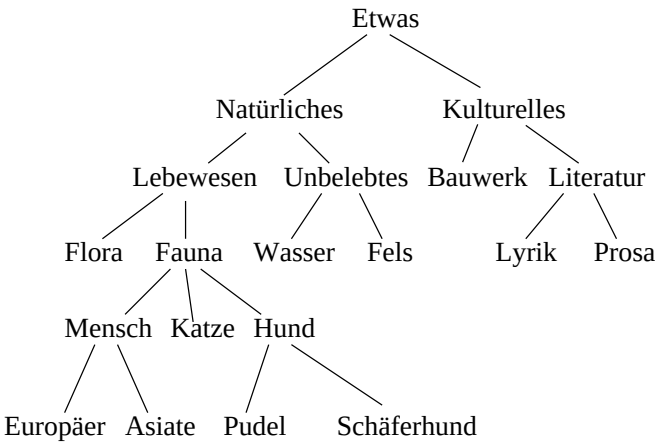

Abbildung 3.8: Hierarchie für einige konkrete Begriffe

Eine Ontologie, welche als Begriffshierarchie ausgelegt ist, ist insbesondere für Konzeptgraphen geeignet. Konzeptgraphen haben eine generelle Struktur, der zufolge generelle Begriffe von der Wurzel des Graphen ausgehend immer weiter spezialisiert werden. Möchte man zwei Konzeptgraphen vergleichen, so gibt es ein einfaches Verfahren, das im Wesentlichen den Inhalt der Knoten miteinander vergleicht. Jedoch kann es passieren, dass in einem Graphen Synonyme oder Verallgemeinerungen *(Hyponyme)* verwendet werden:

- Graph 1 erkläre, dass die gesuchte Entität ein Mensch ist und Michael heißt.
- Graph 2 erkläre, dass der gesuchte Entität ein Europäer ist und Michael heißt.

Ein direkter Vergleich würde scheitern, da die Begriffe unterschiedlich sind. Eine Ontologie kann nun helfen, beide Aussagen vergleichbar zu machen: Da laut Ontologie jeder Europäer ein Mensch ist, ist Graph 2 eine speziellere Version von Graph 1. Gibt man bei einer Suche den Graphen 1 an, so wäre eine Entität, zu welcher Graph 2 gehört, eine korrekte Auswahl.

Um präzise Aussagen unter Verwendung einer Ontologie treffen zu können, ist eine formale Darstellung hilfreich. Leider gibt es keine einheitliche Vorstellung des Wesens einer Ontologie, sondern nur Ad-hoc-Definitionen für den jeweiligen Kontext.

Definition 3.4 Ontologie
Eine *Ontologie O* ist eine Menge prädikatenlogischer Aussagen der Form

1. $P(x)\downarrow$ (definierte Konzepte)
2. $P(x_1,\ldots,x_n)\downarrow$ (definierte Relationen)
3. $\forall x=(x_1,\ldots,x_n), n\geq 1 : P(x)\Rightarrow Q(x)$ (Implikationen von Konzepten oder Relationen)

Eine Ontologie kann also Aussagen der Form „x ist eine Instanz von A“ (ohne zu sagen, ob sie wahr oder falsch sind) sowie auch Verhältnisse zwischen verschiedenen Entitäten als *definiert* bewerten und liefert logische Schlüsse: „*Odysseus* ist ein Agent, welcher in einem Netzwerk migriert“ übersetzt sich prädikatenlogisch zu

$$\exists x : Agent(Odysseus) \wedge migriertIn(Odysseus, x) \wedge Netzwerk(x).$$

Die Ontologie in Anhang C definiert Synonyme und Hyponyme, welche sich entsprechend prädikatenlogisch so darstellen:

$$\forall x : Agent(x) \Rightarrow Stellennutzer(x) \text{ und } \forall x, y : migriertIn(x, y) \Leftrightarrow wandertIn(x, y).$$

Dies impliziert die Aussage: „*Odysseus* ist ein Stellennutzer, welcher in einem Netzwerk wandert“.

Zur Verdeutlichung schreibt man auch $P(x) \downarrow O$ für die Tatsache, dass die Aussage $P(x)$ in O definiert ist. Ist ein Prädikat aus Teilprädikaten zusammengesetzt, so ist es genau dann in O definiert, wenn alle Teile in O definiert sind.

Eine andere Form der Ontologie ist die so genannte *virtuelle Wissensbasis*. Die Virtualität drückt sich dahingehend aus, dass keine tatsächliche Datenbank von Begriffen vorhanden ist, sondern dass die Wissensbasis aufgrund einer geeigneten Implementierung das Vorhandensein einer Wissensbasis simuliert. Im Forschungsfeld der *künstlichen Intelligenz* werden solche virtuellen Wissenbasen zunehmend den Datenbankorientierten Wissensbasen vorgezogen [FW91]. Für das in dieser Arbeit vorgestellte Typsystem wurde eine einfache Begriffshierarchie als Ontologie gewählt; diese ist in Anhang C zu finden.

3.6 Vermittlung

Im Laufe der Erstellung einer Anwendung ist es bislang erforderlich, den Code zusammenzustellen, ihn zu übersetzen und dann mit den Modulen, welche vom Code verwendet werden sollen, zu binden. Dabei werden Referenzen *aufgelöst*, das heißt durch die zugehörigen Adressen, auf welche sie Bezug nehmen, ersetzt.

Diese Vorgehensweise setzt voraus, dass die zu einer Anwendung gehörenden Komponenten sämtlich bekannt sind, bevor die Anwendung gestartet wird. Gerade im Kontext verteilter Systeme wird hingegen darauf abgezielt, Bindungen erst zur Laufzeit durchzuführen. So können bereits betriebsbereite Komponenten auf bestimmten Stellen des Netzwerks ihren Dienst verrichten, noch bevor die Anwendung fertiggestellt ist. Ist dem Nachfrager die Position des Dienstes im Netz unbekannt, so ist eine *Vermittlung* erforderlich, um den Adressaten festzustellen.

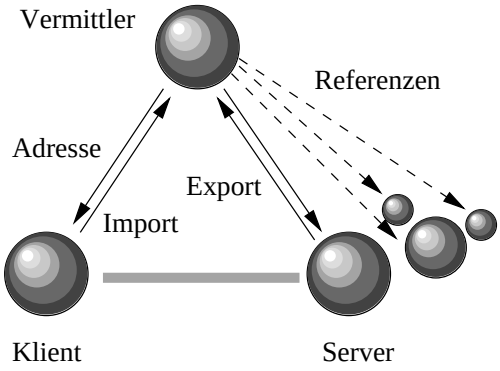

Abbildung 3.9: Konzept der Vermittlung

3.6.1 Prinzip der Vermittlung

Das Prinzip der Vermittlung ist eine nahe liegende Erweiterung des bekannten Klient-Server-Prinzips. Die zwischen Klient und Server aufzubauende Verbindung wird durch den Einsatz einer dritten Instanz, dem Vermittler (auch *Mediator* oder Trader genannt [ITU95, Pud97]), unterstützt, wie Abbildung 3.9 zeigt. Die Funktionsweise des Vermittlers ist kurz beschrieben:

- Nimm eine Beschreibung entgegen.
- Vergleiche diese Beschreibung mit den Beschreibungen, welche in einer Datenbank gelagert sind.
- Beurteile eine gespeicherte Beschreibung als *zutreffend* oder *nicht zutreffend*. Das Kriterium wird durch die Art der Beschreibungen festgelegt und kann durch den Anfrager parametrisiert werden.
- Liefere alle zutreffenden Beschreibungen an den Anfrager.

Mehrere Server-Objekte – von denen jedes einzelne seine Lokation kennt – geben einem Vermittler ihre Adresse bekannt (sie *exportieren* ihre Position), welcher nun Referenzen auf diese Objekte in einer Tabelle hält. Ein Klient bittet den Vermittler um die Adresse eines geeignetes Servers (er *importiert* die Position des Servers). Ist dieser Vorgang erfolgreich, kann der Klient mit dem Server direkten Kontakt aufnehmen und benötigt keinen Vermittler mehr.

Die Entscheidung über die erwünschte Adresse fällt zunächst der Vermittler. Der Klient präsentiert ihm Informationen über den erwünschten Server, welche von diesem im Rahmen seiner Export-Operation bekannt gemacht werden mussten. Möglichkeiten sind

- ein Bezeichner: Der Server ist dann unter diesem Namen registriert;

- die Aufrufschnittstelle: Der Server gibt an, welche Methoden von ihm angeboten werden;
- semantische Informationen: Der Server liefert Angaben zu seinem Einsatzzweck oder andere nicht-syntaktische Informationen.

Je nach den angebotenen Informationen muss der Klient eine passende Anfrage konstruieren und sie an den Vermittler richten. Die Art der Information bestimmt den Vergleichsalgorithmus; während einfache Bezeichner leicht zu überprüfen sind, sind Schnittstellen komplex, aber effizient miteinander zu vergleichen. Semantische Informationen wiederum erfordern eine geeignete Kodierung, etwa in Form von Konzeptgraphen.

Der Vorgang der Vermittlung kann in zwei Kategorien aufgeteilt werden.

Typvermittlung Liefere eine Beschreibung einer Gruppe von Entitäten ab (einen *Typ*), welche dem Vermittlungssystem bekannt sind und der dargebotenen Beschreibung möglichst genau entspricht.

Instanzvermittlung Liefere eine Menge von Adressen von Entitäten ab, welche dem Vermittlungssystem bekannt sind und der dargebotenen Beschreibung möglichst genau entsprechen.

Im Prinzip kann man sich vorstellen, dass bei der Instanzvermittlung eine Typvermittlung vorausgeht, wobei alle aktuell existierenden Instanzen des gefundenen Typs genannt werden. Die Typvermittlung ist von Interesse, wenn die gesuchten Instanzen erst erzeugt werden sollen; bei der Instanzvermittlung möchte man bereits existierende Instanzen finden.

3.6.2 Vermittlungsstandards

Die ISO hat im Rahmen der ODP-Standardisierung ein Konzept der Vermittlung definiert [ITU95]. Die oben eingeführten Begriffe wie *Export, Import, Vermittler* sind Bestandteil dieser Spezifikation. Um das Problem eines möglichen Zentralpunktausfalls *(single* oder *central point of failure)* zu vermeiden, wird das Konzept des Vermittlerverbunds eingeführt *(Federation of traders).* Diese Vermittler können einen ihnen übergeordneten Vermittler im Falle einer Anfrage konsultieren, sodass man eine Meta-Ebene der Vermittlung hinzugewinnt.

Die *Object Management Group* definiert für die *Object Management Architecture* einen so genannten *Trading Object Service* [OMG00b] (vormals *Common Object Services Specification Trading* oder kurz *COSS Trading).* In diesem Modell werden ebenfalls drei Parteien, der *Exporteur*, der *Importeur* sowie der *Vermittler (Trader)* aufgeführt, welcher die Vermittlung anhand von *Diensttypen (Service Types)* durchführt. Diese haben im Allgemeinen folgende Struktur:

- eine Schnittstellenbeschreibung, formuliert in OMG IDL;
- Eigenschaften des Dienstes *(Properties)*, welche verhaltensbezogene und nichtfunktionale Charakteristiken beschreiben.

Die Eigenschaften können des Weiteren als *obligatorisch (mandatory)* und *geschützt (readonly)* deklariert werden, das heißt, dass sie zum einen anzugeben sind und zum anderen nach dem Export nicht mehr modifiziert werden können. Die *Typkonformität* wird folgendermaßen festgelegt: Ein Typ β ist genau dann ein Subtyp des Diensttyps α, wenn

1. die Schnittstelle von β von der Schnittstelle von α abgeleitet oder gleich ist;
2. alle Eigenschaften von α ebenfalls in β definiert sind;
3. der Modus der Eigenschaften (obligatorisch, geschützt) in β mindestens so streng ist wie der Modus der entsprechenden Eigenschaft in α;
4. der Datentyp der sich entsprechenden Eigenschaften bei α und β gleich ist.

Die Eigenschaften werden als *Name-Wert-Paar* repräsentiert, wobei sich die Datentypen auf numerische/logische Werte, Zeichen, Zeichenketten oder Sequenzen dieser Typen beschränken sollten.[9] Ein Dienstangebot besteht aus der Angabe des *Diensttypnamens*, einem Verweis auf die Schnittstelle, welche vom Dienst implementiert wird sowie gegebenenfalls Werte für die Eigenschaften (mindestens für die obligatorischen). Dieses Angebot wird dem Vermittler durch den Exporteur bekannt gemacht. Ein Importeur (Klient) kann eine Anfrage formulieren, welche den Dienst zur gegebenen Schnittstelle liefert, wobei er Einschränkungen in Bezug auf die Eigenschaften liefern kann.

Unter der *wissensbasierten Typvermittlung* versteht man eine Vermittlung, welche sich semantischer Informationen bei der Auswahl treffender Angebote bedient. Ein Beispiel einer solchen Vermittlung präsentierte Arno Puder in seiner Dissertation [Pud97]. Der von ihm entworfene *AI-Trader* verwendet Konzeptgraphen, um Dienstangebote zu beschreiben [PMG95, PG97]. Vorteil dieser Beschreibungssprache ist eine *benutzerorientierte* Beschreibung; Konzeptgraphen können so entworfen sein, dass sie das intuitive Verständnis des Benutzers in Bezug auf den jeweiligen Dienst (oder allgemein auf das jeweilige Objekt) widerspiegeln.

3.7 Zusammenfassung

Dieses Kapitel gab eine kurze Einführung in die Theorie der Typen. Wichtig ist es, einen Typ nicht nur als Datentyp, sondern allgemein als eine durch ein Prädikat be-

[9] Bei anderen Datentypen steht kein Wertvergleich zur Verfügung, sondern lediglich ein Test auf Vorhandensein des Wertes.

stimmte Menge von Entitäten zu erkennen. Diese Sichtweise wird die Grundlage der Definition des in dieser Arbeit vorgestellten Typsystems für autonome Agenten sein.

Das Konzept der Typen in Programmiersprachen existiert schon seit dem Aufkommen höherer Programmiersprachen. Gegen die zunehmende Komplexität von Programmen werden Typen eingesetzt, um Programmfehler noch vor der Programmausführung durch einen Typprüfer entdecken zu lassen. Zur Laufzeit kann das System eine ungültige Operation noch vor deren Ausführung erkennen und verhindern.

Typen werden in vielfältiger Form realisiert. Das Prinzip der Klassen und Schnittstellen ist im Umfeld der objektorientierten Programmierung von besonderer Bedeutung. Mittels der Ableitung ist eine effektive Wiederverwendung von Code möglich. Instanzen eines Subtyps oder einer Subklasse können stets in Situationen verwendet werden, in denen eigentlich eine Instanz des Typs erwartet wurde.

Semantische Informationen zu Methoden oder Objekten können von Interesse sein, da Schnittstellenbeschreibungen stets unterspezifiziert sind. Dazu sind Formalisierungsmethoden des Wissens erforderlich. Konzeptgraphen stellen eine im Zusammenhang der Subtypbeziehung geeignete Repräsentation dar, da sie sowohl die Abstraktion als auch die Spezialisierung semantischer Angaben erlauben.

Abschließend wurde ein kurzer Blick auf das Prinzip der Vermittlung geworfen. Die Vermittlung ist eines der zentralen Einsatzgebiete des Typsystems für autonome Agenten. Mit ihrer Hilfe kann das Problem behandelt werden, das sich in offenen Anwendungen stellt, nämlich das Finden eines bestimmten Objekts, welches für eine Interaktion erforderlich ist.

4 Das Agentensystem AMETAS

Um die Randbedingungen für die Entwicklung eines Typsystems für autonome Softwareagenten einordnen zu können, sind gewisse Details zum zu Grunde liegenden Agentensystem AMETAS notwendig. Es wird offensichtlich, dass mit der Einführung des neuen Paradigmas der autonomen Softwareagenten völlig neue Situationen in der Kommunikation zwischen Komponenten in das Blickfeld geraten. In diesem Kapitel werden lediglich die Grundzüge von AMETAS, insbesondere die an der Kommunikation und Vermittlung beteiligten Komponenten, angeführt; weiterführende Details sind unter [ZH01] zu finden.

4.1 Struktur von AMETAS

Das Agentensystem AMETAS, dessen Name ein Akronym des Begriffs *Asynchronous Message Transfer Agent System* darstellt, entstand parallel zur der diesem Text zu Grunde liegenden Forschungsarbeit. Ziel war es, eine Plattform zu schaffen, welche einerseits die Erforschung bestimmter Prinzipien wie der Autonomie der Agenten erlaubt, andererseits ein Testbett für die praxisnahe Erprobung des zu entwickelnden Typsystems liefert.

Die Implementierung von AMETAS erfolgte unter Zuhilfenahme des *Java Development Kit 1.1*; zum Betrieb ist eine Java-Umgebung *(JRE)* ab Version 1.1.7 oder 1.3 für *Java 2* erforderlich. Alle Anwendungen, welche sich des AMETAS-Systems bedienen, sind entsprechend in Java geschrieben; sollen Java-sprachfremde Komponenten zum Einsatz kommen, bietet sich die in Java verfügbare Nativcodeschnittstelle *(Java Native Interface)* an, welche von Interesse ist, wenn auf spezielle Komponenten des Betriebssystems oder auf Geräte zugegriffen werden soll.

4.1.1 Komponenten

Zur Installation von AMETAS genügt es, das Java-Klassenarchiv *AMETAS.jar* auf einem lokal zugänglichen Verzeichnis im Dateisystem abzulegen und den Pfad zu dieser Datei im Klassensuchpfad von Java einzutragen. Danach sind einige einfache Konfigurationsaufgaben durchzuführen, die hier jedoch nicht dargestellt werden sollen und in [ZH01] nachzulesen sind.

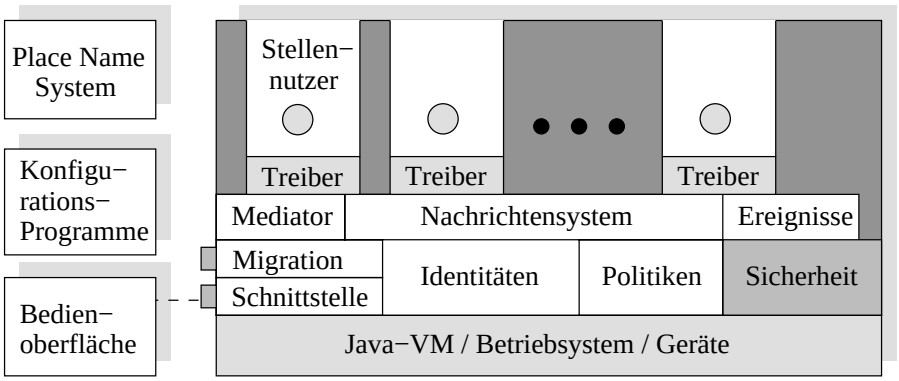

Abbildung 4.1: Komponenten von AMETAS

Das Java-Klassenarchiv beinhaltet neben den zum Betrieb der Infrastruktur notwendigen Klassen auch Klassen und Schnittstellen, die der Anwendungsentwicklung dienen. Abbildung 4.1 zeigt den schematischen Aufbau des gesamten AMETAS-Systems.

Zuunterst befindet sich das grundlegende System, zu dem die virtuelle Maschine von Java sowie das Betriebssystem und darüber erreichbare Geräte zählen. Die zwischen diesem unterliegenden System und den Treibern liegenden Komponenten bilden die so genannte *Agentenstelle* oder kurz *Stelle*. Der Begriff Stelle (engl. *Place*) wurde bereits in frühen Tagen der Agentenforschung geprägt, insbesondere im Kontext von Telescript [Whi94]. Stellen bieten den Anwendungskomponenten die notwendige Ausführungsumgebung, indem sie Kontrollflüsse *(Threads)* sowie Basisfunktionalitäten zur Verfügung stellen.

AMETAS implementiert ein flexibel konfigurierbares Sicherheitssystem, welches einerseits Anwendungen gegenseitig abschirmt und andererseits Zugriffe auf das unterliegende System kontrolliert. Die in Java 2 bereitgestellten Sicherheitsmechanismen wurden in AMETAS auf Basis von Java 1.1 reimplementiert und auf die speziellen Bedingungen in einem Agentensystem angepasst. Die Konfiguration erfolgt über *Politiken*, welche den bekannten *Identitäten* bestimmte Privilegien einräumen. Eine genauere Beschreibung des Sicherheitssystems findet man in [ZMG98] oder [ZH01].

Möchten Agenten migrieren, so erledigen die Stellen die notwendigen Aufgaben wie das Zusammentragen und Übermitteln von Zustand und Code der Agenten, handeln untereinander Migrationsparameter (wie etwa Verschlüsselung) aus und setzen die Ausführung der Agenten nach Ankunft fort. Die Adressierung einer Stelle wird durch ein eigenständiges System, dem *Place Name System*, ermöglicht, welches Stellennamen und Netzlokationen einander zuordnet.

Anwender können sich an eine Stelle über eine spezielle Bedienoberfläche anbinden; diese kann während des Betriebs angeschlossen und wieder abgetrennt werden. Des Weiteren ermöglichen diverse Konfigurationsprogramme die Modifikation von Betriebsparametern der Stelle, die Erzeugung von Klassenpaketen für Agenten, die Ge-

nerierung und Verwaltung von Identitäten sowie die Administration des *Place Name System*.

4.1.2 Adressierung von Stellen

Da die AMETAS-Infrastruktur in Form von eigenständig laufenden Prozessen, den Stellen, gebildet wird, ist es prinzipiell möglich, auf den zur Verfügung stehenden Plattformen (welche alle Multiprozessverarbeitung erlauben) mehrere solcher Stellen zu betreiben. Die Adressierung von Stellen ist von Interesse, wenn Bedienschnittstellen an die Stellen angeschlossen werden sollen oder Agenten zu einer Stelle migrieren. Da die Kommunikation – durch Java vorgegeben – auf der TCP/IP-Protokollfamilie beruht, welche die Kommunikationsendpunkte durch so genannte *Sockets* in der Anwendungsschicht repräsentiert, liegt es nahe, die Adressierung von Stellen durch Angabe der Rechneradresse und der Portnummer zu realisieren.

Dies erfordert vom Anwendungsentwickler und Anwender, dass sie die Positionen der anzusprechenden Stellen exakt kennen. Zum einen muss die genaue Bezeichnung des Rechners, zum anderen die Portnummer des Sockets bekannt sein. Möchte man die Möglichkeit des Betriebs mehrerer Stellen auf einem Rechner bewahren, verbieten sich Festlegungen von wohlbekannten Portadressen *(well-known ports)*.

Ein weiteres Problem stellen dynamisch vergebene IP-Adressen dar. Gewöhnlich werden IP-Adressen für Wählverbindungen vom Anbieter an seine Kunden für die Zeit ihrer Verbindung vergeben. Der Kundenrechner kann damit über einen längeren Zeitraum betrachtet unterschiedliche IP-Adressen aufweisen. Betreibt der Kunde eine Stelle, die von einem Agenten angelaufen werden soll, so ist die im Agenten vormals gespeicherte Information möglicherweise ungültig, wenn inzwischen die Verbindung getrennt und wieder aufgenommen wurde.

Dieses Problem lässt sich, wie in vielen anderen Gebieten lange üblich, durch die Verwendung *symbolischer Namen* zur Adressierung von Stellen behandeln. Es muss dazu eine eindeutige Abbildung zwischen symbolischen Namen und den Adressen und Portnummern der zugehörigen Stellen geleistet werden, ähnlich wie es das *Domain Name System* im Internet realisiert. Das Pendant in AMETAS wird entsprechend *Place Name System* oder kurz *PNS* genannt.

Eine Besonderheit unterscheidet das PNS vom DNS: Um der Tatsache der häufig wechselnden IP-Adressen der Stellen Rechnung zu tragen, ist es den Stellen möglich, ihre aktuelle IP-Adresse dem PNS mitzuteilen, welcher seine Datenbank selbständig aktualisiert. Agenten, welche zu einer Stelle reisen möchten, müssen lediglich deren symbolischen Namen kennen; die eigentliche Zuordnung zu den tatsächlichen Netzadressen bleibt ihnen verborgen, sodass sie auch im Falle wechselnder IP-Adressen keine besonderen Vorkehrungen treffen müssen.

Das PNS ist hierarchisch organisiert, ähnlich wie das DNS, ist aber von dessen Namensgebung unabhängig. Diese Hierarchisierung ist nicht nur notwendig, um eine Konzentration von Anfragen auf einen zentralen Punkt zu verhindern (welcher einen

Single point of failure darstellen würde, also bei Versagen das Gesamtsystem in Mitleidenschaft zöge), sondern auch, um die Propagation von PNS-Änderungen einzudämmen. Stellen ordnen sich daher *Domänen* unter:

Stelle1.uniffm.de.

Hauptstelle.michael.zuhause.de.

sind so genannte *vollqualifizierte Stellennamen*, welche aus dem Stellennamen (erster Teil des Bezeichners bis zum ersten Punkt von links) und dem Namen der Domäne (restlicher Teil) bestehen. Die Domänen sind hierarchisch angeordnet; im Beispiel ist *de* eine Unterdomäne der Wurzeldomäne, welche mit dem schließenden Punkt bezeichnet wird; *uniffm* ist eine Unterdomäne von *de*. Das PNS ist in der Lage, Anfragen bezüglich einer bestimmten Domäne an den zuständigen *Place Name Server* zu delegieren.

Diese Server sind eigenständige Prozesse, deren Programmcode durch Klassen implementiert ist, welche dem AMETAS-System beiliegen. Sie sind jedoch kein eigentlicher Teil des Agentensystems; außer den Stellen und den Benutzerschnittstellen benötigen keine anderen Komponenten Informationen zu den Lokationen der Stellen. Jede Stelle muss in ihrer Konfiguration einen der Server des PNS nennen; ihre Anfragen werden, wenn nötig, delegiert. Änderungen der IP-Adresse werden durch denselben Mechanismus an den zuständigen Server delegiert.

Um eine entfernte Stelle zu erreichen, muss ein Agent einen entsprechenden Methodenaufruf auf seinem Treiber unternehmen:

```
m_Driver.go(”Stelle1.uniffm.de.”, nRelayTime);
```

Das AMETAS-System bietet demzufolge keine absolute Lokationstransparenz; zwar wird durch die Verwendung symbolischer Bezeichner der tatsächliche Ort im Netzwerk verborgen, für den Agenten stellt sich das System dennoch als aus mehreren voneinander zu unterscheidenden Komponenten dar, welche er durch Aufruf der *go*-Methode erreichen kann. Eine vollständige Transparenz könnte man sich so vorstellen, dass im Zuge der Erfüllung des Auftrags das Stellensystem für eine automatische Migration des Agenten sorgt. Ein interessanter Anwendungsfall ist dabei die *Lastbalancierung* zwischen verschiedenen gleichberechtigten Stellen. Das System kann entscheiden, welches tatsächliche Ziel vom Agenten angelaufen werden soll, wenn dieser eine bestimmte Dienstleistung nutzen möchte.

Eine allgemeine, lokationstransparente Migration hätte weit reichende Konsequenzen für die Programmierung. Die Definition von Reiserouten ist praktisch nicht möglich, da es kein Modell für eine Zieladresse gibt; die Idee der Mobilität wird aus der Anwendungsschicht in eine tiefer liegende Infrastrukturschicht verschoben. Da es in der realen Welt sehr wohl die Vorstellung gibt, dass es unterschiedliche Lokationen mit

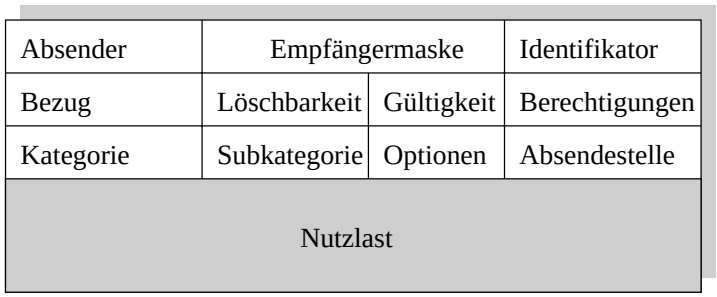

Abbildung 4.2: Aufbau einer AMETAS-Nachricht

unterschiedlichen lokalen Gegebenheiten gibt und es relevant ist, eine bestimmte Lokation aufzusuchen, wird die Modellierung solcher Agentenanwendungen ohne dedizierte Lokationen erschwert. Sollen dennoch verschiedene Orte unterscheidbar sein, müsste ein anderes, adressierbares Merkmal jeweils eindeutig vorhanden sein, was dadurch die Stellenadressen indirekt wieder einführt. Es wurde daher entschieden, in AMETAS das Konzept der explizit adressierbaren Stellen zu verwirklichen. Eine Vermittlung an unterschiedliche, gleichberechtigte Stellen kann in Form eines zusätzlichen Dienstes realisiert werden, ohne dem Agenten die Entscheidung der aktiven Migrationsauslösung zu entziehen.[1]

4.1.3 Nachrichtenaustausch

Komponenten einer Anwendung kommunizieren untereinander in der Regel durch Methodenaufrufe oder gemeinsam benutzen Speicher in Form von zugänglichen Instanz- oder Klassenvariablen. Als Bestandteile desselben eigenständigen Prozesses steht es generell allen Komponenten der Stelle frei, sich gleichermaßen Daten zu übermitteln. Die erforderliche Nebenläufigkeit der verschiedenen Komponenten in einem Multiagentensystem lässt sich in Java bequem durch die Verwendung von Threads bereitstellen.

Zahlreiche Agentensysteme verfolgen einen solchen Ansatz. So werden benutzerdefinierte Agenten von einer zur Verfügung gestellten Elternklasse abgeleitet, welche einen Satz bekannter Methoden bietet. Darüber hinaus können diese Agenten weitere Funktionalitäten durch spezifische Methoden anbieten, welche in Java leicht mit Hilfe der Reflexions-Programmierschnittstelle *(Reflection API)* in Erfahrung gebracht werden können.

AMETAS ermöglicht den Komponenten lediglich eine Kommunikation über Nachrichten, welche entkoppelt vom Sender und Empfänger von der Stelle entgegengenommen und weitergeleitet werden. Die Struktur einer Nachricht zeigt Abbildung 4.2.

[1] In Abschnitt 8.1.3 wird eine mögliche Erweiterung von AMETAS für lokationstransparente Migrationen vorgestellt.

Eine Nachricht ist aus einem *Nachrichtenkopf* sowie einem *Nutzlastteil* aufgebaut. Der Kopf besteht dabei aus folgenden Feldern:

Absender	Angabe des Absenders (vom System eingefüllt)
Empfängermaske	Angabe der Empfänger dieser Nachricht
Identifikator	ID der Nachricht (vom System eingefüllt)
Bezug	ID der Nachricht, auf welche sich diese Nachricht bezieht
Löschbarkeit	Angabe, wer diese Nachricht löschen darf
Gültigkeit	Zeitraum bis zur automatischen Entsorgung
Berechtigungen	Berechtigungen des Absenders (vom System eingefüllt)
Kategorie	Art der Nachricht (vom System eingefüllt)
Subkategorie	Genauere Spezifikation der Nachricht
Optionen	Optionen der Nachrichtenverarbeitung
Absendestelle	Stelle, an der sich der Absender befindet (bei entfernter Zustellung)

Der Aufbau eines Identifikators wird weiter unten genauer ausgeführt. Der Austausch von Nachrichten wird in anderen Systemen ebenfalls angeboten, ist in AMETAS jedoch der einzige Kommunikationsweg. Er läuft vollständig asynchron, was dem System seinen Namen gibt: *Asynchronous Message Transfer Agent System.*

Der Nutzlastteil besteht aus einem Feld von allgemeinen Objektreferenzen. Dies ermöglicht es, beliebige Nachrichteninhalte zu transportieren.[2] Die Erweiterung des AMETAS-Kommunikationssystems mit KQML-Nachrichten bedient sich gerade dieser Eigenschaft [Jah01].

Die *synchrone Kommunikation*, also jene, welche den Aufrufer während der Phase des Aufrufs blockiert, ist gewöhnlich effizienter realisierbar als die asynchrone Varianten. Dies liegt in der Natur der meisten Programmiersprachen, insbesondere Java, begründet: Methodenaufrufe sind dort immer synchron, lassen sich damit aber auch einfach in ausführbaren Code übersetzen, da sich die Struktur von Unterprogrammaufrufen bei den heutigen Architekturen bis auf Betriebssystem- und Hardwareebene verfolgen lässt.

Asynchrone Kommunikation hingegen ist für gewöhnlich auf der Anwendungsebene zu realisieren. Zur Realisation ist mindestens eine weitere Komponente erforderlich, welche Nachrichten durch einen synchronen Aufruf entgegennimmt, sie aufbewahrt

[2] Die Klassen der transportierten Objekte müssen allgemein verfügbar und serialisierbar sein.

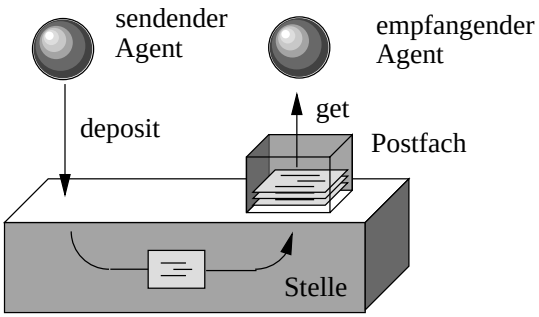

Abbildung 4.3: Nachrichtensystem

und dann durch einen synchronen Aufruf seitens des Empfängers an diesen weitergibt. Diese Aufgabe wird in AMETAS durch die Stelle geleistet, genauer durch das *Postfachsystem* (Abbildung 4.3). Komponenten, welche am Nachrichtenaustauschsystem teilnehmen möchten, verwenden spezielle Methoden in ihren Treibern, um Nachrichten abzusetzen oder sie aus ihrem Postfach abzuholen. Die Weiterleitung ist Aufgabe der Stelle. Jedes Postfach unterhält eine eigene *First-in-first-out*-Warteschlange (FIFO), um die Nachrichten zu speichern.

Zu bemerken ist, dass die Einrichtung eines Postfachs sich nicht danach richtet, ob es *tatsächlich* einen Agenten der genannten Adresse gibt. Postfächer sind Adressen zugeordnet; es ist somit möglich, dass es zu Postfächern keine früher, derzeit oder in Zukunft existierenden Interessenten gibt. Die Einrichtung eines solchen Postfachs zu einer Adresse *A* richtet sich alleine danach, ob ein Absender die Adresse *A* verwendet. Postfächer werden nach einer einstellbaren Zeit der Nichtverwendung entfernt. Dieses Verfahren birgt zwei Vorteile:

- Es können Nachrichten an Agenten versendet werden, welche die Stelle noch nicht besucht haben.
- Es können Nachrichten an Gruppen von Agenten versendet werden.

Es ist für eine Stelle prinzipiell nicht feststellbar, ob eine Empfängerangabe eine existierende Instanz referenziert oder nicht. Denn eine Stelle besitzt keine global eindeutige Sicht auf das Gesamtsystem und das Nachrichtensystem ist primär nur auf die lokale Kommunikation ausgerichtet. AMETAS sieht auch eine entfernte Zustellung von Nachrichten vor; dazu muss neben dem Empfänger auch ein Stellenname angegeben sein. Die Nachricht wird an dieser Stelle im bezeichneten Postfach abgelegt.

4.2 Stellennutzer

Ein grundlegendes Merkmal von AMETAS ist die Einteilung der kommunizierenden Komponenten in drei Kategorien:

- Agenten
- Benutzeradapter
- Dienste.

Alle diese Objekte – sie werden in der AMETAS-Terminologie *Stellennutzer* genannt – kommunizieren untereinander über das AMETAS-Nachrichtensystem, das heißt sie senden einander Nachrichten zu. Anders als in vielen anderen Agentensystemen ist es in AMETAS jedoch nicht möglich, dass zwei Stellennutzer in unmittelbaren Kontakt miteinander treten; vielmehr obliegt es der Infrastruktur (der *Stelle*), die Nachricht dem Adressaten zur Verfügung zu stellen.

4.2.1 Adressierung von Stellennutzern

Jedem Stellennutzer ist ein eindeutiger Identifikator zugewiesen, den er selbst nicht ändern kann, die so genannte Stellennutzer-ID. Diese begleitet ihn während seines gesamten Lebenszyklus und besteht aus

- einem Zeitstempel in Millisekunden;
- der IP-Adresse der Stelle, welche die ID generiert hat;
- einem Bezeichner *(Name)* und
- einem zweiten Bezeichner *(Gruppe)*

Vermöge der ersten beiden Komponenten erlangt man eine hinreichende Eindeutigkeit der ID; die anderen beiden Angaben werden vom Implementierer gesetzt. Damit ist es möglich, Gruppen von Stellennutzern zu bilden, was insbesondere für die Kommunikation zwischen verschiedenen Stellennutzern geeignet ist, welche keine Kenntnis der jeweiligen Identifikatoren haben. Um diese zu adressieren, werden *ID-Masken* eingesetzt. Diese sind wie Stellennutzer-IDs aufgebaut, können aber an beliebigen Stellen frei gelassen werden. Eine Stellennutzer-ID passt genau dann zu einer ID-Maske, wenn die vorhandenen Komponenten der Maske mit denen der Stellennutzer-ID übereinstimmen.

4.2.2 Agenten

So vage die Vorstellung eines „Agenten“ in der Fachwelt ist, so unterschiedlich fallen die Implementierungen aus. So findet man häufig eine Unterscheidung zwischen *mobilen* und *stationären* Agenten [BHR+97] oder auch zwischen *intelligenten* und *mobilen* Agenten, jedoch fehlen zumeist klar abgegrenzte Definitionen, anhand deren man feststellen kann: *Dieses Objekt ist ein Agent* oder *dieses Objekt ist kein Agent*.

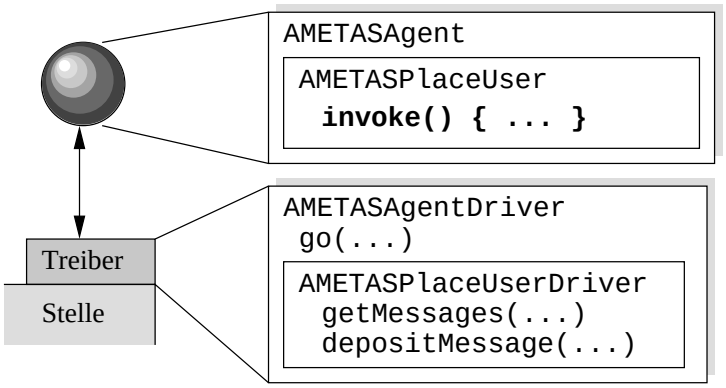

Abbildung 4.4: Agent in AMETAS

In AMETAS wird der Begriff der „Agentenschaft“ (*agenthood*) in zweifacher Weise unterstützt:

1. *implementationstechnisch*: Ein Agent ist eine Instanz einer Subklasse der Klasse *AMETASAgent*.

2. *konzeptionell*: Ein Agent ist eine autonome, in sich abgeschlossene Komponente, welche mit anderen Stellennutzern über das Nachrichtensystem kommuniziert. Er kann bei Bedarf die Ausführungsumgebung wechseln.

Wechselt ein Agent die Ausführungsumgebung, so bittet er die Infrastruktur um eine Verlagerung seines Programmcodes und seines Zustands an eine von ihm angegebene neue Position. Dieser Vorgang wird Migration genannt. Ein Agent, welcher im Laufe seiner Ausführung migriert, wird in AMETAS entsprechend allgemeiner Übereinkunft *mobiler Agent* genannt. In AMETAS können Agenten grundsätzlich migrieren, sodass man durchaus alle Agenten als mobil und mithin AMETAS als *Mobile-Agenten-System* bezeichnen könnte. Im Folgenden wird daher das Attribut „mobil“ im Zusammenhang mit Agenten nicht mehr explizit erwähnt.

Abbildung 4.4 zeigt den Aufbau eines Agenten in AMETAS.

Die Erstellung von AMETAS-Agenten erfolgt durch die Implementierung einer von der Klasse *AMETASAgent* abzuleitenden Klasse. Diese Implementierung erfolgt ausschließlich in Java; Agenten haben keinen Zugang zur Nativschnittstelle (JNI). Der Agentenautor ist dabei recht frei in der Strukturierung des Agenten; insbesondere ist es möglich, Agenten als Aggregation mehrerer Klassen zu entwerfen. Wichtig ist lediglich die Implementierung der *invoke*-Methode, welche die Abarbeitung des Agentenverhaltens initiiert. Weitere Methoden können von *invoke* aus aufgerufen werden. Abbildung 4.5 stellt einen einfachen, aber bereits mobilen AMETAS-Agenten dar.

Jedem Stellennutzer ist ein Treiber zugeordnet, welcher beispielsweise zur Abholung und Einreichung von Nachrichten oder zur Ausgabe von Informationen in die

```
import AMETAS.agentdev.*;
import AMETAS.data.*;

public class MeinAgent extends AMETASAgent {
   AMETASPlaceUserID m_idSender = null;
   public void invoke() {
      AMETASMessage[] ames = m_Driver.getMessages(true);
      if (ames!=null) {
          m_idSender = ames[0].getSenderID();
          Object[] aBody = { "Gehe zu Stelle1" };
          m_Driver.depositMessage
                  (new AMETASMessage(m_idSender, aBody));
      }
      try {
          m_Driver.go("Stelle1", DONT_RELAY);
      }
      catch(Exception e) {
          m_Driver.output("Migration fehlgeschlagen.");
      }
   }
}
```

Abbildung 4.5: Implementierung eines einfachen Agenten

Protokolldatei der Stelle dient. Abbildung 4.5 demonstriert unter anderem den Umgang mit Nachrichten. Gewöhnlich wird dem neu gestarteten Agenten vom Erzeuger eine Nachricht in sein Postfach gelegt, die dieser zu Initialisierungszwecken nutzen kann. Der Agent versucht in diesem Beispiel, die Nachrichten abzuholen; der ersten Nachricht entnimmt er den Identifikator des Absenders, legt ihn in einem Objektfeld ab und schickt dem Absender eine einfache Botschaft. Dieses Objektfeld ist Teil des Zustands, welcher bei Migrationen bewahrt wird. Jede spezielle Stellennutzerklasse verfügt über einen spezialisierten Treiber. So bietet der Treiber eines Agenten – wie hier zu sehen ist – diesem die Möglichkeit, eine Migration mittels der *go*-Methode zu veranlassen.

AMETAS hebt den Aspekt der *Autonomie* eines Agenten besonders hervor. Ein Agent muss in der Lage sein, den ihm übertragenen Auftrag so zu erledigen, dass er nur auf die Kooperation anderer Instanzen angewiesen ist, welche zur Erfüllung des Auftrags notwendig sind. Daraus ergeben sich weit reichende Konsequenzen für die Implementierung eines Agenten. Da er nur mit anderen Stellennutzern kommunizieren kann, muss er in Betracht ziehen, dass der von ihm angesprochene Stellennutzer seinerseits ein autonomer Agent sein kann, der einen eigenen Auftrag zu erfüllen ver-

sucht. Dieser könnte einfach die Kooperation verweigern und damit dem erstgenannten Agenten die Erfüllung seines Auftrags möglicherweise vereiteln. Multiagentenanwendungen müssen entsprechend so gestaltet sein, dass eine Kooperation stattfindet.

In AMETAS sind Agenten in ihrer Handlungsfreiheit im Vergleich zu anderen Agentensystemen deutlich eingeschränkt. So ist es AMETAS-Agenten nicht erlaubt,

- auf anderem Wege als über Nachrichtenaustausch mit anderen Stellennutzern zu kommunizieren;
- Systemressourcen wie Dateien, Netzverbindungen, grafische Oberflächen unmittelbar zu nutzen;
- anderen Komponenten eine Dienstschnittstelle anzubieten, welche Klienten die Nutzung von bestimmten Funktionalitäten per Methodenaufruf gestattet.

Bei dem Versuch, etwa ein Fenster mittels *new Frame()* zu öffnen, löst der *SecurityManager* unmittelbar eine Ausnahmebedingung aus und erwirkt den Abbruch der Ausführung des Agenten.

Diese Einschränkungen sind Folgerungen aus der Forderung der Autonomie. Eine direkte Kommunikation mit anderen Stellennutzern ist vorstellbar, wenn die Kommunikationspartner gegenseitig Objektreferenzen hielten. Sie könnten dann ihre Zustände direkt oder durch Aufruf bestimmter Methoden modifizieren. Dieses Vorgehen setzt jedoch voraus, dass beide Partner *kollokiert* sind, sich also zum Zeitpunkt der Interaktion an derselben Lokation im Netz befinden. Ferner müssen beide Partner darauf vertrauen, dass ihr Gegenüber nicht aufgrund seines eigenen Auftrags die Entscheidung trifft, eine andere Lokation aufzusuchen, also zu migrieren, bevor die Interaktion abgeschlossen ist.

Um die Mobilität von Agenten zu ermöglichen, ist es notwendig, dass die Migrationsziele eine geeignete Plattform zur Entgegennahme der Agenten zur Verfügung stellen. In AMETAS wird dies durch die Stellen geleistet; dadurch wird insbesondere eine systemweit durch die Verfügbarkeit der gleichen Grundfunktionalität vergleichbare Umgebung bereitgestellt.

4.2.3 Benutzeradapter

Um die Kommunikation zwischen Agenten und Anwendern zu ermöglichen, führt AMETAS das Konzept der *Benutzeradapter* ein. Diese Komponenten dienen zur Integration systemexterner Benutzer in das Agentensystem. In der Regel sind dies menschliche Anwender; es können aber auf einfache Weise Schnittstellen geschaffen werden, um andere Anwendungen oder Middleware-Strukturen (wie CORBA) anzubinden.

Als Stellennutzern stehen den Benutzeradaptern die gleichen Kommunikationsmöglichkeiten wie den Agenten zur Verfügung. Im Unterschied zu diesen genießen sie wesentlich größere Freiheiten beim Umgang mit Systemressourcen. Benutzeradapter

sind in der Lage, jede beliebige Art von Interaktion mit externen Informationsquellen zu unterhalten, welche die Java-Architektur erlaubt. In der Regel sind dies grafische Oberflächen, die dem Anwender erlauben, Informationen aus dem Agentensystem zu erhalten oder in das System einzubringen. So können Suchergebnisse eines Agenten in einer geeigneten Weise als Liste dargestellt werden; der Anwender könnte durch Mausklick ein Ergebnis aus der Liste aussuchen und dadurch einen Agenten mit neuen Instruktionen versehen, etwa zu dem gewählten Ort zu migrieren.

Die beiden Hauptaufgaben eines Benutzeradapters lassen sich somit wie folgt definieren:

1. Übersetzung von Informationen aus dem Agentensystem in eine für den Anwender verständliche Präsentation.

2. Übersetzung von Anwenderreaktionen in eine für die Komponenten des Agentensystems verständliche Form.

Dieser Eigenschaft von Benutzeradaptern erlaubt die vollständige, transparente Integration des Benutzers in das Agentensystem. Der Datenaustausch zwischen Anwender und Benutzeradapter ist für das übrige Agentensystem unsichtbar; umgekehrt können Details des Agentensystems und der in ihnen laufenden Anwendungen auf einfache Weise vor dem Benutzer verborgen werden. Der Benutzer tritt im System als ein weiterer Stellennutzer auf und als solcher kann er nur mittels Nachrichten mit den übrigen Stellennutzern kommunizieren. Es ist für einen AMETAS-Agenten unerheblich, ob der Stellennutzer, welchem er seine Daten übermittelt, ein Benutzer hinter einem Benutzeradapter oder ein anderer Agent ist.

Die Implementierung eines Benutzeradapter ähnelt jener eines Agenten; auch hier ist die *invoke*-Methode zu implementieren, welche die Ausführung des Adapters initiiert. Je nach den verwendeten Ressourcen kommen weitere Programmstrukturen hinzu (etwa Objekte mit Rückaufruf-Methoden für den Betrieb einer grafischen Oberfläche).

Benutzeradapter sind Agenten gegenüber in einer Hinsicht eingeschränkt: Sie können nicht migrieren. Eine Migration ist auch konzeptionell wenig sinnvoll, da ein Benutzer sich zwar an verschiedenen Rechnern an- und abmelden, aber nicht wie ein Agent durch das System reisen kann. Agenten sind während ihrer Lebensdauer stets auf irgendeiner Stelle des Systems auffindbar, wenn man die Reisezeit vernachlässigt; diese Annahme kann für Anwender nicht getroffen werden. Ein Anwender tritt typischerweise an einem Ort dem System bei und verlässt es nach einiger Zeit wieder; Informationen über sein Wiedererscheinen sind im Allgemeinen nicht verfügbar.

4.2.4 Dienste

Die dritte Kategorie von Stellennutzern sind die *Dienste*. Ein Dienst ist im Allgemeinen eine Komponente, welche eine bestimmte Funktionalität realisiert und anderen Kom-

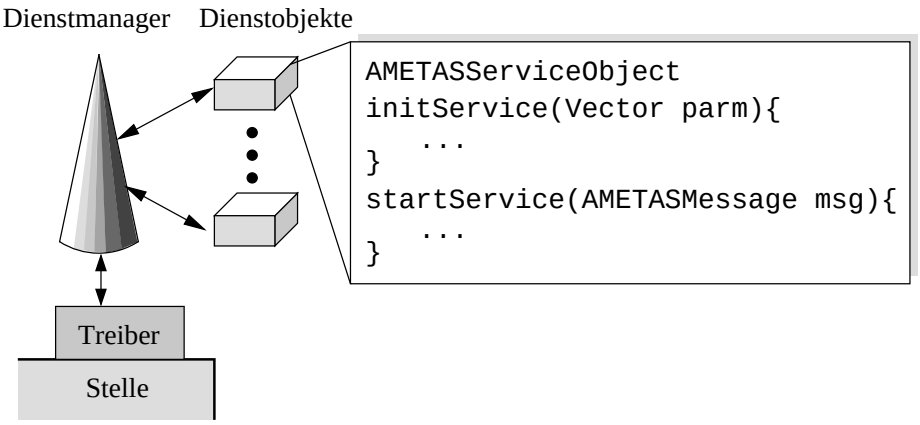

Abbildung 4.6: Dienst in AMETAS

ponenten zur Verfügung stellt, indem sie deren Anfragen entgegennimmt, verarbeitet und gegebenenfalls eine Antwort zurückschickt.

AMETAS-Dienste bestehen aus zwei Teilen:

- dem Dienstobjekt, das die Funktionalität des Dienstes implementiert;
- dem Dienstverwalter, welcher bei Bedarf Dienstobjekte instantiiert und Anfragen an diese delegiert.

Der Dienstverwalter ist ein vom Kernsystem zur Verfügung gestelltes Objekt. Das Dienstobjekt muss hingegen geeignet implementiert und installiert werden. Anders als Agenten können Dienste nicht migrieren; sie sind auf ihre Ausführungsumgebung festgelegt. Dies erlaubt es, Dienste für diese Umgebung spezifisch zu optimieren: Um Zugriff auf Systemkomponenenten zu erlangen, welche durch den architekturunabhängigen Entwurf von Java in der Programmiersprache gar nicht oder nicht einheitlich repräsentiert werden können, können native Methoden [Lia99, Ste01] eingesetzt werden. Abbildung 4.6 zeigt die schematische Darstellung eines Dienstes.

Soll ein Dienst erstellt werden, implementiert man in diesem Falle keinen Stellennutzer, denn dies ist stets eine Instanz der vom System vorgegebenen Klasse *AMETASServiceManager*. Stattdessen ist das Dienstobjekt bereitzustellen, das von einer von *AMETASServiceObject* abgeleiteten Klasse stammt. Bei der Implementierung der Funktionalität geht man grundlegend anders vor, als es bei den anderen Stellennutzerarten bereits erwähnt wurde: Anstelle der *invoke*-Methode bietet ein Dienstobjekt die *startService*-Methode, welche jedoch immer wieder von neuem aufgerufen wird, wenn eine Nachricht an diesen Dienst geschickt wird. Diese Nachricht wird durch den Dienstmanager direkt an die Methode des Dienstobjekts übergeben. Nach der Instantiierung gibt es die Gelegenheit, mittels der Methode *initService* eine Initialisierung vorzunehmen, bevor die erste Nachricht eintrifft; dazu kann über eine Konfigurationsdatei der Stelle ein Parametersatz über einen Vektor übergeben werden. Soll der

Modus	Bedeutung
SHARED	Gemeinsame Verwendung eines Dienstobjekts
NON_SHARED	Verwendung separater Dienstobjektinstanzen
SHARED_SESSION	Sitzung mit gemeinsamem Dienstobjekt
NON_SHARED_SESSION	Sitzung mit separaten Dienstobjektinstanzen

Tabelle 4.1: Aktivierungsmodi von Diensten

Dienst Aktionen außerhalb der Verarbeitung eintreffender Nachrichten vollziehen können, müssen eigene Kontrollflüsse (Threads) gestartet werden.

Der Dienstverwalter ähnelt in seiner Rolle dem aus CORBA bekannten *Objektadapter*. Als solcher sorgt er für die Weiterleitung von Anfragen an das Dienstobjekt. Dabei gibt es verschiedene *Aktivierungsmodi*, wie Tabelle 4.1 zeigt. Die Struktur von AMETAS legt Diensten eine besondere Rolle nahe: Sie sind *Informationsquellen* für Agenten. Als solche sind sie – mehr als andere Stellennutzerarten – Ziel von Anfragen möglicherweise zahlreicher, verschiedener Agenten. Als Quellen ihrer Informationen, welche sie Agenten zur Verfügung stellen, können sowohl externe Komponenten als auch andere Stellennutzer auftreten.

Dienste können somit genutzt werden, um Informationen zwischen Stellennutzern auszutauschen. In diesem Falle müssen die beteiligten Stellennutzer die Gewissheit haben, dass die von ihnen angesprochenen Dienstobjektinstanzen dieselben Informationen präsentieren, was sinnvollerweise durch die Verwendung einer gemeinsamen Instanz gewährleistet wird. Ein solcher Betriebsmodus wird in AMETAS durch die Verwendung des Parameters *SHARED* in der Dienstbeschreibung bewirkt. Wenn hingegen sichergestellt werden soll, dass die in einer Dienstobjektinstanz gehaltenen Daten nur einem Stellennutzer zugänglich sind, sind getrennte Instanzen zu empfehlen.

Soll die Verarbeitung folgender Anfragen von vorangegangenen Anfragen abhängen, so muss der Verarbeitungszustand bewahrt werden. Dies geschieht durch die Weiterverwendung derselben Instanz für weitere Anfragen. Man spricht hier von einer *Sitzung* zwischen den Kommunikationspartnern. Eine Sitzung ist ein Kommunikationskontext, der explizit eröffnet und geschlossen wird. Das System muss sicherstellen, dass der Kommunikationspartner während der gesamten Zeit der Sitzung nicht beendet oder ausgewechselt wird. Vom Dienst wird erwartet, dass er die eintreffenden Nachrichten korrekt anhand der Absender unterscheidet.

Vom Aspekt der Nachrichtenverarbeitung ist eine Aktivierung eines Dienstes unter dem Merkmal *SHARED_SESSION* nicht zu unterscheiden von einer Aktivierung als *SHARED*-Dienst. Die Eigenschaft des Dienstes, sich der Sitzung mit unterschiedlichen Klienten bewusst zu sein, ist ein Implementationsaspekt; in beiden Fällen wird der AMETAS-Kern über den Dienstverwalter nur eine Instanz starten und Anfragen an sie weiterleiten. Unterschiedlich werden die Dienste in Bezug auf ihre Terminierung behandelt. Während ein *(NON_)SHARED*-Dienst sich prinzipiell unmittelbar nach Ab-

arbeitung einer Anfrage beenden könnte, müssen *(NON_)SHARED_SESSION*-Dienste auf die explizite Abmeldung ihrer Klienten warten.

4.3 Aufbau von Multiagentenanwendungen

Die Entwurfsstrategie von AMETAS-Anwendungen ähnelt dem so genannten *Three-Tier-Prinzip* [Eck95] (Dreischichtenprinzip):

1. Benutzeradapter liefern die Präsentationslogik.
2. Agenten verarbeiten Daten, welche sie von den anderen Schichten erhalten.
3. Dienste bilden eine Schnittstelle zum unterliegenden System und versorgen Agenten mit Daten, welche sie aus externen Quellen gewinnen können.

Damit ist es leicht, Anwendungen mit unterschiedlichen Benutzerschnittstellen auszustatten, ohne die Verarbeitungslogik ändern zu müssen. Gleiches gilt auf der Dienstseite; lokale Spezifika der unterliegenden Plattformen werden vor der Verarbeitungslogik verborgen.

4.3.1 Eine einfache Anwendung

Um den Einsatz der verschiedenen Komponenten zu illustrieren, sei hier die Anwendung *StartAndKill* vorgestellt, welche zum Starten und Stoppen von Stellennutzern dient. Zweck der Anwendung ist, Agenten zu starten, welche sogleich zu anderen Stellen des Agentensystems migrieren können und eine Möglichkeit anzubieten, diese Agenten wieder aufzuspüren und zu terminieren. Die Aufgabe der Lokalisation von Agenten in einem Verbund von Stellen ist nicht trivial [BR98]. Nicht nur das eigentliche Auffinden kann erheblichen Aufwand nach sich ziehen; in der Regel liegt zudem keine Atomizität der Operation Lokalisation und Terminierung eines Agenten zu Grunde. Dies bedeutet, dass noch vor der eigentlichen Terminierung der Agent weitergereist sein kann, obwohl er schon lokalisiert wurde. AMETAS bietet als Grundfunktion einer Stelle eine Vorkehrung an, Agenten zu terminieren, indem eine spezielle Liste geführt wird, welche Agenten zu terminieren sind. Sind diese aktuell anwesend oder treffen diese an der Stelle ein, können sie sofort terminiert werden. Diverse Strategien zur Lokalisation eines Agenten sollen hier jedoch nicht ausgeführt werden; sie können etwa in [BR98] nachgeschlagen werden.

Die *StartAndKill*-Anwendung ist dreiteilig, wie die Abbildung 4.7 zeigt. Der Benutzeradapter bietet dem Anwender eine zweckmäßige Bedienoberfläche und entlastet ihn bei der Kommunikation mit dem den Auftrag ausführenden Agenten. Ferner führt der Benutzeradapter Buch über die von ihm aus gestarteten Agenten, sodass es nicht

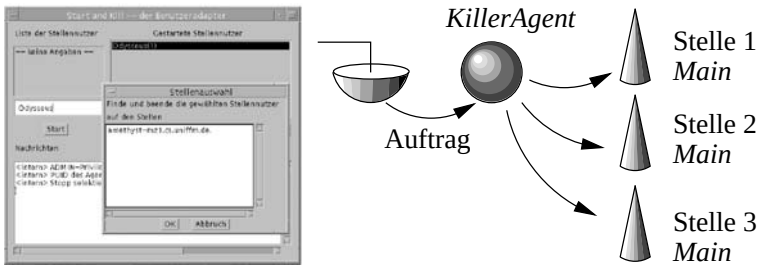

Abbildung 4.7: Einfache Agentenanwendung

notwendig ist, die Identifikatoren der Agenten durch den Benutzer eingeben zu lassen; er kann die zu stoppenden Agenten aus einer Liste auswählen.

Nachdem der Anwender die Agenten selektiert hat, kann er den Terminierungsauftrag aktivieren. Nach Beendigung der Eingabe finden folgende Aktionen statt:

1. Der Benutzeradapter trägt die in der Liste selektierten Agenten zusammen und sucht deren Identifikatoren in einer internen Tabelle zusammen.

2. Eine Nachricht wird formuliert, welche die Identifikatoren der zu terminierenden Agenten sowie die zu bereisenden Stellen enhält. Außerdem beinhaltet sie eine Angabe, wie lange die Terminierungsanforderung auf der jeweiligen Stelle aufzubewahren ist.

3. Ein spezieller Agent wird gestartet. Dieser bekommt durch den Benutzeradapter nach seiner Instantiierung die soeben formulierte Nachricht zugesendet.

4. Der Agent hat den Auftrag, die in der Nachricht genannten Stellen zu besuchen und jeweils eine Nachricht einem bestimmten Dienst namens *Main* zu übergeben. Während der Dienst den Terminierungsauftrag bearbeitet, setzt der Agent seine Reiseroute fort.

5. Der Dienst ruft eine spezielle Methode auf einer internen Schnittstelle der Stelle auf, um ihr die Daten zur Terminierung der einzelnen Agenten zu übergeben.

Die Terminierung der betreffenden Agenten obliegt einzig der jeweiligen Stelle, da diese direkten Zugriff auf die Komponenten besitzt, welche für die Steuerung von Agenten notwendig sind. Natürlich ist zu beachten, dass die Terminierung *berechtigt* ist, dass also der Auftraggeber des Agenten die betreffenden Agenten terminieren darf. Während gewöhnlich Dienste die Privilegien eines Klienten schon vor der Verarbeitung einer Anfrage prüfen können, so sind sie in diesem Falle auf Informationen der Stelle angewiesen, da die zu terminierenden Agenten möglicherweise erst noch eintreffen und bis zu diesem Zeitpunkt die Berechtigung nicht beurteilt werden kann.

Hat der Agent alle Stellen besucht und seine Terminierunganforderung an den jeweiligen Dienst abgesetzt, so wandert er nun zu denselben Stellen, um das Ergebnis der Terminierung in Erfahrung zu bringen. Sind Stellennutzer an den entsprechenden Stellen aufgrund dieser Terminierungsanforderung gestoppt worden, so hat der zuständige Dienst eine Nachricht für den Agenten hinterlegt, welche dieser bei seiner Ankunft abholen kann. Sind sämtliche Stellennutzer beendet worden – was der Agent mit Hilfe einer mitgeführten Liste leicht nachweisen kann – dann ist der Auftrag beendet und er kehrt zur Stelle zurück, an der er gestartet wurde. An der Ursprungsstelle angekommen sendet er eine Nachricht an den Benutzeradapter, welcher sie in geeigneter Form (etwa durch eine Auflistung) dem Benutzer präsentiert.

4.3.2 Anwendungen mit mehreren Agenten

Multiagentenanwendungen erfordern in der Regel eine Instanz, welche den Informationsaustausch zwischen verschiedenen Komponenten der Anwendung steuert. Zwar können Nachrichten durch das Postfachsystem der Stelle den einzelnen Komponenten zur Verfügung gestellt werden, jedoch sind in AMETAS die dort befindlichen Nachrichten nicht unbegrenzt lagerbar. Abgesehen von dieser systemspezifischen Einschränkung ist das Nachrichtensystem nicht in der Lage, situationsabhängig zu reagieren, also beispielsweise die Verbreitung einer bestimmten Nachricht zu unterbinden, wenn der Auftrag schon erledigt wurde.

Die Koordination solcher Anwendungen wird in AMETAS – je nach Anwendung – durch einen oder mehrere Stellennutzer bewerkstelligt. Die Auswahl der Art des Stellennutzers richtet sich wiederum an die jeweiligen Bedürfnisse, wobei auch Mischformen mit verschiedenartigen Koordinatoren auf unterschiedlichen Ebenen auftreten können:

- Agenten als Koordinatoren sind von der Anwesenheit eines menschlichen Benutzers unabhängig.
- Benutzeradapter können Ergebnisse direkt für eine Darstellung aufbereiten und zugleich die Anwendung steuern.
- Dienste eignen sich für Anwendungen, welche benutzerunabhängig gestaltet sind, wie etwa eine agentenbasierte Rechercheanwendung, deren Ergebnisse anfragenden Agenten zuteil werden soll.

Die in Abbildung 4.8 gezeigte Agentenanwendung illustriert den Einsatz der unterschiedlichen Stellennutzer und deren Koordination in einer Netzmanagementanwendung. Vorbild ist das Rahmenwerk *NetDoctor* [ZHG99a, ZHG99b], welches die Netz- und Systemverwaltung auf SNMP-Basis um die Verwendung autonomer Agenten erweitert. *NetDoctor* erlaubt die Verwaltung von Netzen und Rechnern durch die Bereitstellung einer Schnittstelle zur Kommunikation mit konventionellen SNMP-Agenten.

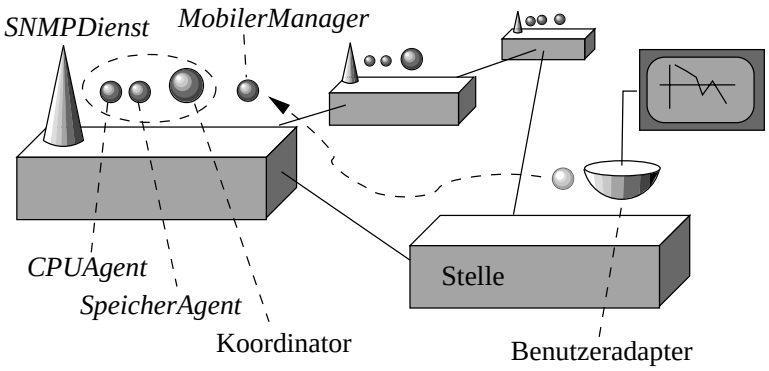

Abbildung 4.8: Mehrstufige Koordination am Beispiel von *NetDoctor*

Als Rahmenwerk ermöglicht es dem Anwender, seine individuellen Vorgaben an das Management zu implementieren.

Jede der Stellen verfügt über den sogenannten *SNMPDienst*, welcher die Kommunikation mit dem SNMP-Agenten (welcher sowohl lokal als auch entfernt liegen kann) einerseits und dem Agentensystem andererseits organisiert. Dazu setzt dieser die Agentennachrichten in geeignete SNMP-Nachrichten um und nimmt die Antworten des SNMP-Agenten entgegen. Des Weiteren werden an den Stellen spezielle Agenten installiert, welche sich bestimmter Teilaufgaben des Managements widmen können, etwa der Überwachung der CPU-Last, des verfügbaren Speichers oder (hier nicht dargestellt) der Verbindungsstatistik einer Netzschnittstelle. Ein Koordinator schließlich sammelt die Informationen der Spezialagenten und entscheidet, ob die aktuelle Situation ein Eingreifen erfordert.

Die Spezialagenten sowie der Koordinator werden nach dem Start des Benutzeradapters zu den Stellen entsandt; dies ermöglicht eine flexiblere Konfiguration des Managementsystems, als wenn die Agenten bereits vor Ort fest installiert wären. Der Benutzeradapter zeigt dem Administrator den aktuellen Zustand des Systems, aufbereitet durch die von ihm an den Stellen eingesetzten Agenten. Werden von den Koordinatoren keine ungewöhnlichen Symptome festgestellt, so obliegt die Koordination den lokalen Koordinatoren; der Administrator kann seinen Benutzeradapter schließen.

Sollte eine Situation auftreten, die nicht durch den lokalen Koordinator behandelt werden kann, so entsendet dieser einen Boten (Transportagent) an den Administrator, welcher ihn von der Situation in Kenntnis setzt. Der Administrator kann sich dann entschließen, einen Korrekturagenten zu entsenden; diese Entscheidung kann in häufig auftretenden, gleichartigen Situationen auch vom Benutzeradapter selbst getroffen werden. Der *MobilerManager* genannte Agent wandert zu den betroffenen Stellen und versucht, das Problem durch vorgegebene Maßnahmen in den Griff zu bekommen. Ist er nicht erfolgreich, kann der Administrator entscheiden, einen neuen Korrekturagenten zu entsenden oder gegebenenfalls direkte Korrekturen vor Ort durchzuführen.

Agentenanwendungen sind immer dann als Multiagentenanwendungen ausgelegt,

wenn die Funktionalität modularisiert vorliegen soll, sodass Teile der Anwendung während der Laufzeit ausgetauscht werden können. Migrationen sind für komplexe Agenten mit einer nicht zu unterschätzenden Netzlast verbunden; die Aufteilung von Agenten in spezialisierten Teilagenten ist daher ein Gebot der Ressourcenschonung. Als unabhängige Agenten bedürfen sie im Gegenzug einer Koordinierung; so wird beispielsweise in *NetDoctor* verhindert, dass der Agent *MobilerManager* zweifach angefordert wird, wenn zwei Ereignisse korreliert sind, also von derselben Ursache abhängen.

4.4 Vermittlung in AMETAS

Die obige, recht einfache Anwendung *StartAndKill* hat mit einem allgegenwärtigen Problem zu kämpfen: Wenn der Terminierungsagent an einer fremden Stelle ankommt, muss er zunächst herausfinden, wie er den Dienst addressieren kann, der für die Terminierung von Stellennutzern zuständig ist. Nachrichten können in AMETAS immer nur an Träger einer so genannten Stellennutzer-ID gerichtet werden; diese müssen also bereits aktiv sein, wenn der Agent eintrifft.

Eine Alternative ist, den Stellennutzer selbst zu starten, an den die Nachricht gerichtet werden soll. In der *NetDoctor*-Anwendung von Abbildung 4.8 ist dem Benutzeragent die ID jedes einzelnen Stellennutzers bekannt, da er sie alle selbst gestartet hat. In der Tat liefern die Methoden

```
AMETASPlaceUserID requestPUStartup(...)
AMETASPlaceUserID spawnAgent(...)
```

jeweils gerade jene Stellennutzer-ID, unter der man den gestarteten Stellennutzer mit einer Nachricht erreichen kann. Dieser Identifikator kann jedem Agenten mitgegeben werden, der an der Anwendung beteiligt ist.

Gerade bei Diensten liegt das Problem vor, dass diese meist schon beim Start der Stelle mitgestartet werden, dass somit die Anwendung keine Information erhält, unter welcher Adresse sie den Dienst erreichen kann. Sollen laufende Agentenanwendungen durch weitere Komponenten ergänzt werden, so müssten diese erst durch Nachfragen bei einem Koordinator herausfinden, welche IDs den anderen Stellennutzern gehören. Jedoch muss der Koordinator zuvor bestimmt sein. AMETAS unterstützt den Anwender in zweifacher Hinsicht: durch so genannte *Multicast-Nachrichten* und durch *Vermittlung*.

4.4.1 Verwendung von Multicast-Nachrichten

Die Angabe des Empfängers einer AMETAS-Nachricht wird stets durch eine Stellennutzer-ID-Maske bewerkstelligt. Soll eine Nachricht an einen bestimmten Agenten gerichtet werden, so genügt es, seinen Namen zu kennen, welcher in der Stellennutzer-ID

angegeben ist. Dieser Name wurde früher ausschließlich als Klassenname gedeutet, kann jedoch beliebig gewählt werden. Ein weiteres Feld, die *Gruppe*, kann zur gemeinsamen Adressierung von Agenten verschiedener Namen verwendet werden. Die Spezialagenten im Beispiel von Abbildung 4.8 könnten unterschiedliche Namen, aber gemeinsam den Gruppennamen *MeineNDAgenten* aufweisen. Soll eine Nachricht an all diese Agenten gesendet werden, so genügt es, diesen Gruppennamen als einziges Datum in die Maske zu schreiben. Das Nachrichtensystem richtet dann ein *Multicast-Postfach* ein, welches von den genannten Agenten abgefragt werden kann.

Diese einfachste Form der Adressierung unbekannter Stellennutzer hat Nachteile, welche eine allgemeine Verwendung zur Behandlung dieses Problems nicht empfehlenswert erscheinen lassen:

- Alle Stellennutzer einer Klasse haben den gleichen Namen und Gruppennamen. Sie sind ununterscheidbar bei Verwendung einer solchen Maske und erhalten somit immer dieselben Nachrichten.
- Jeder Implementierer kann den Namen seines Agenten frei wählen, ebenso die Gruppe. Es ist nicht möglich, ein unberechtigtes Lesen von Nachrichten zu verhindern.
- Name oder Gruppenname müssen bekannt sein. Es ist nicht festzustellen, ob eine Nachricht nicht ankam, weil die Angaben nicht korrekt waren, oder weil der Agent sie nicht lesen wollte. Name oder Gruppenname setzen nicht voraus, dass es tatsächlich Stellennutzer gibt, welche sie in ihrer ID verwenden.

4.4.2 Vermittlung

Der Vorgang einer Vermittlung wird durch den Anfrager eingeleitet, indem dieser dem Vermittler eine bestimmte Beschreibung von gesuchten Komponenten präsentiert. Den Abschluss der Vermittlung bildet die Antwort des Vermittlers, der eine Liste von Referenzen an den Klienten zurückgibt, welche dieser zur Kommunikation mit den zugehörigen Objekten benötigt. Je präziser die Anfrage ist, umso kleiner wird die Ergebnismenge.

In AMETAS werden sowohl Typvermittlung als auch Instanzvermittlung durch so genannte *Mediatoren* realisiert. Unter einer Instanzvermittlung versteht man im Falle von AMETAS die Lieferung einer Liste von Stellennutzer-IDs, welche Stellennutzern angehören, die einer bestimmten Beschreibung entsprechen und derzeit aktiv sind. Bei einer Typvermittlung möchte man erfahren, unter welchem Namen der gesuchte Stellennutzer gestartet werden kann. Dieser Name ist eindeutig einer Datei zugeordnet, welche von der Stelle geladen wird.[3] Abbildung 4.9 zeigt schematisch den Vorgang der Mediation. Die Stelle registriert die Beschreibungen der Stellennutzer in einem

[3] So genannter SPU-Container; siehe auch Abschnitt 6.1.

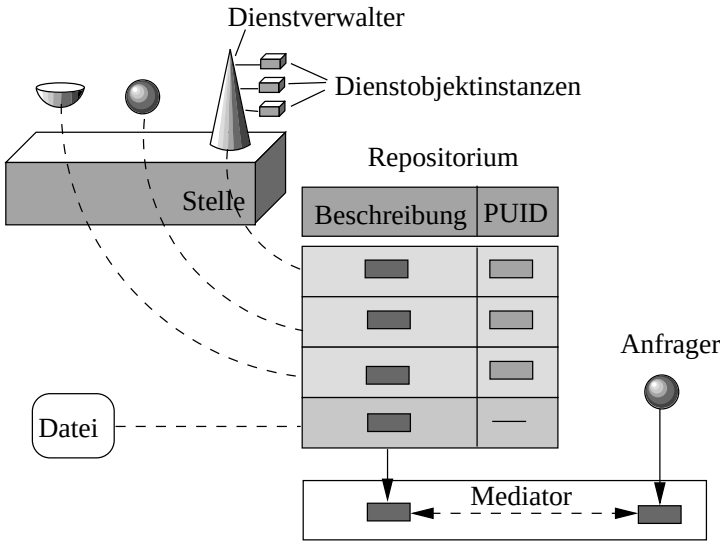

Abbildung 4.9: Mediation von Stellennutzern

Verzeichnis *(Repositorium)*, auf welches der Mediator zugreifen kann. Der Dienstverwalter wird dabei stellvertretend für die von ihm betriebenen Dienstobjektinstanzen registriert. Stellennutzer werden unmittelbar nach ihrem Start – bei Agenten auch bei deren Ankunft – erfasst und sind während ihrer Aktivität an der Stelle vermittelbar.

Die Mobilität der Agenten stellt ein Problem für die Vermittlung dar, da ein Nachfrager nicht darauf vertrauen kann, dass ein bereits vermittelter Agent noch anwesend ist. Daher wird die Eintragung eines Agenten nach dessen Abreise für eine voreingestellte Zeit aufbewahrt. Der Nachfrager kann sich in einem zweiten Schritt vergewissern, dass der Agent noch anwesend ist (siehe auch Anhang E). Im Falle der Terminierung eines Stellennutzers (einschließlich Agenten) verliert die Stellennutzer-ID ihre Gültigkeit. Die Eintragung in der Instanztabelle wird daraufhin entfernt. Nicht betroffen davon sind die Typregistrierungen.

Anfrager präsentieren eine Beschreibung, welche vom Mediator mit den im Verzeichnis aufbewahrten Beschreibungen verglichen wird. Die Art der Beschreibung ist somit Mediator-abhängig – dies muss von den Stellennutzern beachtet werden. In Abbildung 4.9 sind beide Vermittlungsarten angedeutet; während sich die oberen drei Einträge auf laufende Instanzen beziehen, referiert die letzte Eintragung eine Datei, welche einen mittels der `requestPUStartup`-Methode zu aktivierenden Stellennutzer beinhaltet. Der Anfrager erhält die Beschreibung des Stellennutzers, kann diese mit der von ihm verwendeten Anfrage vergleichen und gegebenenfalls den Stellennutzer starten. In den anderen Fällen würde der Anfrager die zugehörige Stellennutzer-ID bekommen, welche er als Adresse einer Nachricht verwenden kann.

In Abschnitt 6.7 ist nach der Einführung der Grundlagen des neuen Typsystems eine genauere Beschreibung der Zugänge zur Mediation zu finden.

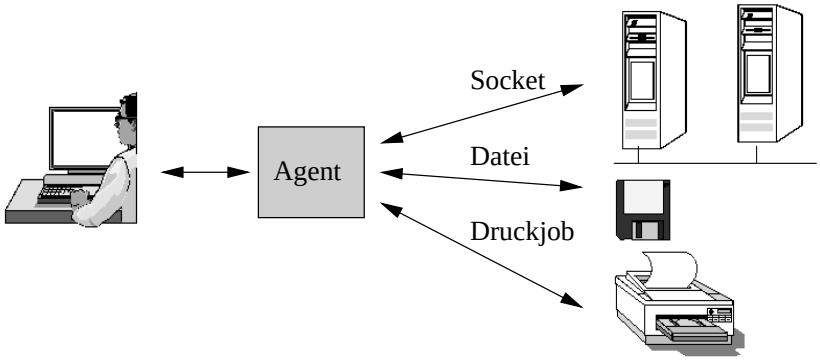

Abbildung 4.10: Allgemeine Agenteninteraktionen

4.5 Interaktion mit der Umgebung

Das Agentensystem AMETAS definiert die Art der Kommunikation eines Stellennutzers mit seiner Umgebung, solange es sich um einen weiteren Stellennutzer handelt. Im Allgemeinen ist dies nicht ausreichend, um nutzbringende Anwendungen zu betreiben. Unter *externen Einrichtungen* mögen im Folgenden also jene Entitäten verstanden sein, welche nicht Komponenten des AMETAS-Systems sind: das betreibende System einschließlich Hardware und Betriebssystem samt Java-Unterstützung, andere Prozesse sowie Geräte, welchen von Anwendungen des Agentensystems aufgrund fehlender Unterstützung in Java oder sicherheitspolitischer Einschränkungen durch das Agentensystem nicht direkt angesprochen werden können.

Ein Zugriff auf externe Einrichtungen ist alleine deshalb schon wünschenswert, weil Agentenanwendungen auf äußere Einflüsse reagieren und beobachtbare Wirkungen auf die Umgebung haben sollen. Allgemein lassen sich in Agentensystemen zahlreiche Interaktionsmöglichkeiten von Agenten mit ihrer Umgebung identifizieren (Abbildung 4.10).

Die meisten Agentensysteme erlegen Agenten keine Restriktionen in dieser Hinsicht auf, sodass gerne das von Java angebotene Fenstersystem zum Zwecke der Benutzerinteraktion mit dem Agenten verwendet wird. Interaktionen gestalten sich hierbei beispielsweise als

- Tastatureingaben in Textfelder, die der Agent präsentiert oder
- Mausaktionen, etwa das Ziehen von Objekten, das Anklicken von Markierungsfeldern oder Drücken von Knöpfen.

Die von AMETAS eingeführte Komponente *Benutzeradapter* bietet eine praktische Schnittstelle zur Interaktion mit Anwendern. Aber nicht nur mit Anwendern treten Agenten in Interaktion. Gerade im Bereich des Netz- und Systemmanagements ist es notwendig, auf Einrichtungen des Systems zuzugreifen, was das Lesen und Schreiben

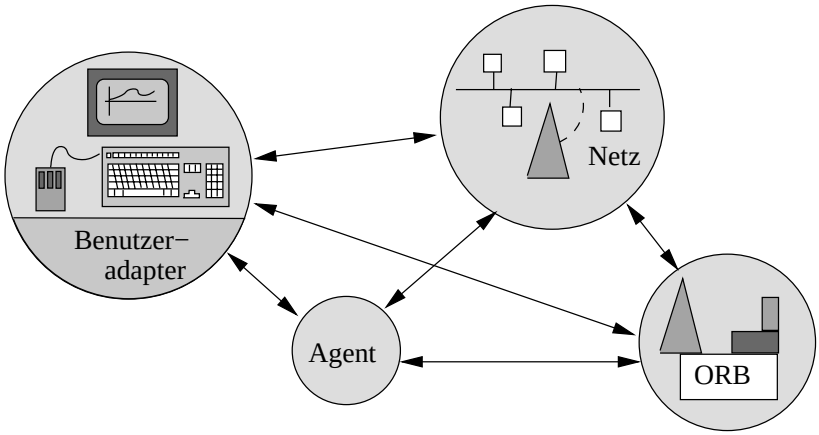

Abbildung 4.11: Kapselung externer Komponenten

von Dateien, das Öffnen von Socketverbindungen, die Kommunikation mit Geräten und die Einstellung von Systemparametern umfasst. Das Agentensystem AMETAS modelliert diese Kommunikation als *interne Kommunikation* zwischen Bestandteilen eines Stellennutzers. In Abbildung 4.11 sind verschiedene Szenarien zusammengefasst.

Es ist für die Kommunikation innerhalb AMETAS irrelevant, welcher Art die Quelle der Daten ist, welche in Nachrichten übermittelt werden. So können externe Anwendungen als interne Komponenten von Diensten betrachtet werden, welche einen Kommunikationskanal mit den AMETAS-spezifischen Teilen des Dienstes aufrechterhalten. Eine entsprechende Implementierung vorausgesetzt kann der Dienst dabei auch als CORBA-Klient auftreten und Antworten von CORBA-Diensten entgegennehmen.

Ein in *NetDoctor* eingesetztes Beispiel der Ressourcenintegration in einen Dienst sind Kommunikationskanäle zwischen den entfernten Stellen. Mittels dieser Kanäle können Daten übertragen werden, wenn die Entsendung eines Agenten als zu aufwändig empfunden wird. Der Administrator kann sich dadurch ein Bild des aktuellen Zustands des überwachten Netzes machen. Die Kommunikation über diese Kanäle ist nicht vom Agentensystem selbst vorgesehen. *NetDoctor* löst dieses Problem durch den Einsatz spezieller *Transmissionsdienste*, welche die Daten über eine ihnen bekannte Verbindung an die Zentralstelle liefern, indem sie sich Socketverbindungen bedienen. Diese Verbindung ist für das Agentensystem transparent; der Sendedienst stellt sich dabei als eine *Datensenke*, der Empfangsdienst als eine *Datenquelle* dar. Die Kommunikation mit den übrigen Stellennutzern geschieht über Nachrichten.

Der Vorteil dieser Integration von externen Einrichtungen ist die Bewahrung des Kommunikationsmodells zwischen den Stellennutzern (Abbildung 4.12) auf Basis des asynchronen Nachrichtenaustauschs. Die Umsetzung der Nachrichten in die entsprechenden gerätespezifischen Interaktionen obliegt den speziellen Komponenten; so muss ein Dienst, welcher Informationen aus dem Netz bereitstellt, selbst für die Einrichtung

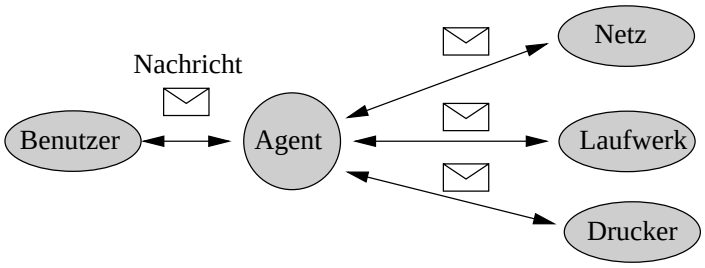

Abbildung 4.12: Homogene Kommunikation

von Socketverbindungen sorgen. Dies hat Auswirkungen auf den Entwurf eines Systems von Beschreibungen dieser Stellennutzer, welches in den kommenden Kapiteln vorgestellt wird.

4.6 Zusammenfassung

Dieses Kapitel stellte die Grundzüge des Agentensystems AMETAS dar, welches die Grundlage zur Entwicklung des im nächsten Kapitel vorgestellten Hybridtypsystems bot. Dieses Agentensystem bietet eine sehr große Menge weiterer Merkmale, die hier nicht alle aufgeführt werden können. Erwähnt seien das Ereignisbehandlungssystem, das Sicherheitssystem mit zahlreichen Unterfunktionen wie Authentifikation, Autorisation, Verschlüsselung und vieles mehr. Genaueres ist auf den AMETAS-Webseiten [ZH01] zu finden.

AMETAS ist ein generisches Agentensystem, das sich für den praktischen Einsatz in verschiedenen Gebieten eignet, obwohl es eine strikte Abstraktion externer Komponenten vorsieht. Agenten kommunizieren über Nachrichten, welche entsprechend umgesetzt werden und an die jeweiligen Gegebenheiten der Interaktion mit der externen Einrichtung angepasst werden, was für das Agentensystem transparent vorgenommen wird. Die Dreiteilung der kommunizierenden Komponenten (Stellennutzer) realisiert die konzeptionelle Trennung zwischen Präsentation, Verarbeitung und Gewinnung von Daten. AMETAS unterstützt explizit den Betrieb von Anwendungen mit mobilen Agenten; die Implementierung „intelligenter“ Merkmale ist dem Entwickler überlassen.

5 Hybridtypen

In diesem Kapitel werden theoretische Grundlagen des Hybridtypsystems vorgestellt. Im folgenden Kapitel werden dann Details zur Implementation präsentiert.

5.1 Anforderungen an ein Typsystem

Das Agentensystem AMETAS, welches im vergangenen Kapitel in seiner Struktur beschrieben wurde, erlaubt die Erstellung und den Betrieb autonomer Softwareagenten. Die Interaktion der Agenten ist eine Grundvoraussetzung für die so genannten *Multi-Agenten-Anwendungen:* Meist sind die Aufgaben nicht nur durch einen Agenten alleine zu lösen, sondern gerade durch die Kooperation verschiedener Agenten[1] werden Szenarien wie dezentrales Netzmanagement durch den Einsatz von Agenten in überzeugender Weise behandelbar. Daher stellt sich generell die Frage, wie eine sichere Interaktion (im Sinne der Art der ausgetauschten Nachrichten) zwischen Agenten in Multi-Agenten-Anwendungen gewährleistet werden kann.

5.1.1 Typsystem und Agenten

Ziel soll es sein, anhand einer Beschreibung (oder *Spezifikation*) in der Lage zu sein, vorherzusagen, ob die Interaktion eines Agenten mit einem anderen Agenten zu Fehlern führen kann oder muss oder erfolgreich verlaufen wird. Ist Letzteres für alle Interaktionen der Fall, spricht man allgemein davon, dass die beiden Agenten zueinander *kompatibel* sind. Andererseits soll es möglich sein, ungeachtet der Protokolldetails zu erkennen, ob der Agent für eine bestimmte Aufgabe geeignet ist. Mit Hilfe dieser Informationen wird die Erstellung von Multiagentenanwendungen erleichtert, da einerseits keine genaue Kenntnis der beteiligten Agenten notwendig ist, andererseits die Erweiterung bestehender Agenten möglich wird, solange sich das Verhalten des Agenten für solche Objekte, die die Grundversion erwarten, nicht verändert.

Ein aus dem Bereich der verteilten Systeme bekannter Schritt zur Spezifikation von Objekteigenschaften ist die Definition von Klassen und Schnittstellen (siehe Abschnitt 3.2). Sowohl eine Klasse wie auch eine Schnittstelle implizieren einen *Typ,* dem man

[1] In diesem Kapitel wird der Einfachheit halber von *Agenten* gesprochen, auch wenn im AMETAS-Kontext *Stellennutzer* gemeint ist.

wesentliche Eigenschaften seiner Instanzen anhand seiner Beschreibung entnehmen kann.

Ein nahe liegender Schritt ist nun, Agenten ähnlich wie Objekten einen Typ zuzuweisen, sie zu *typisieren*. Ist es möglich, einen Agenten mit einem Typ zu beschreiben, so kann man Aussagen über ihn treffen, noch bevor er mit anderen Objekten in Kontakt tritt. Jedoch stellt sich die Frage, ob Agenten *überhaupt typisierbar sind*. Hier muss man die grundlegende Philosophie der Agenten in Betracht ziehen. Leider unterscheidet sich diese zwischen verschiedenen Systemen erheblich. Wichtig ist die allgemeine Übereinkunft, dass Agenten abgeschlossene, autonome Einheiten sind. Als solche haben sie gewisse, unveränderliche Eigenschaften.

Wie in Abschnitt 3.1 dargelegt wurde, sind Typen lediglich Mengen von Entitäten, welche gemeinsame Eigenschaften aufweisen. Agenten, die in Java erstellt sind, sind in der Regel eine Ansammlung von Objekten, welche zusammen als Agent wahrgenommen werden. Die Implementierung des Agenten bleibt aufgrund der Eigenschaft, von den jeweiligen Klassen instantiiert zu sein, stets konstant. Betrachtet man die erbrachte Funktionalität, so ist die Festlegung auf *eine* bestimmte Implementierung nicht immer erwünscht, solange die ausgetauschten Nachrichten, insbesondere die Struktur dieser Nachrichten, gleich bleiben. All diese Eigenschaften sind fest in Form der Klassen kodiert. Daher ergibt sich die

Folgerung 5.1 Typisierbarkeit von Agenten
Ein Agent ist als abgeschlossene Einheit prinzipiell typisierbar, da die den Agenten bildende Aggregation von Objekten Eigenschaften aufweist, welche für den gesamten Lebenszyklus konstant sind. Dies sind insbesondere die Abstammung der Klassen, die Formate der ausgetauschten Nachrichten sowie das Verhalten des Agenten in Abhängigkeit von seinem inneren Zustand.

Da ein Agent seine eigenen Kommunikationsmuster und Einsatzziele kennt, kann er eine Spezifikation daraus ableiten, welcher der Kommunikationspartner genügen muss. Um die Kompatibilität zu beurteilen, ist für ihn also notwendig herauszufinden, ob die gegebene Typbeschreibung des Partners in die erwartete Spezifikation passt; man spricht dann von der *Konformität* des Typs mit der Spezifikation.

5.1.2 Klassische Typisierung

Um einem Agenten einen Typ zuzuweisen, könnte sich anbieten, die Herkunft der Instanzen zu Rate zu ziehen, also die Klassen zu betrachten. Allerdings stellt sich ein Agent als Aggregat mehrerer Instanzen dar. Prinzipiell könnte eine bestimmte Klasse dieser Instanzen ausgezeichnet werden (die *Hauptklasse*), etwa jene, deren Instanz vom Agentensystem unmittelbar angesprochen wird, um den Agenten zu betreiben. Der Agententyp wäre dann der Name der Hauptklasse.

```
public class SecAdminAgent extends AdminAgent
```

```
public class UserAdminAgent extends SecAdminAgent
...
if (agent instanceof SecAdminAgent) ...
```

Diese Zeilen stellen dar, wie eine Typisierung auf Basis des Namens der Hauptklasse genutzt werden könnte. Vorteile dieser Typisierung sind:

- Unterstützung der Typüberprüfung auf Basis von Klassen durch Compiler und Laufzeitsystem
- Optimale Performanz und leichte Nachvollziehbarkeit im Code.

Doch dieses Vorgehen hat schwer wiegende Nachteile. Zum einen wird die Typisierung an einer *konkreten Implementierung* ausgerichtet. Die zu erbringende Funktionalität muss von dieser genannten Klasse stammen; Alternativen sind nicht möglich, wenn sie sich nicht als von der genannten Klasse abgeleitet erklären. Die Typisierung auf Basis einer Klasse impliziert zugleich die Typisierung auf der von ihr implementierten Schnittstelle; es liegt also eine reine syntaktische Typisierung vor. Diese unterscheidet Typen anhand der Methodensignaturen der Schnittstellen. Soll also unterschiedliches Verhalten repräsentiert werden – eine Typungleichheit – so *muss* sich folglich die Schnittstelle unterscheiden, es müssen andere Methoden präsentiert werden. Eine einheitliche Kommunikation zwischen Agenten ist damit nicht zu erreichen.

Eine weitere Schwachstelle ist das Fehlen semantischer Angaben. Während die Klassentypisierung eine *Überspezifizierung* vornimmt, indem sie die Implementierung einer Funktionalität festschreibt, so liefert eine Schnittstellentypisierung eine *Unterspezifizierung*. Es ist nicht ersichtlich, ob die von der Signatur repräsentierte Funktionalität genau jene erwartete ist oder eine andere, welche sich die gleiche Signatur teilt [WZ88, Grü97]. Letztlich wird eine Semantik an den *Namen* der Methode (etwa *getOffer*) oder der Hauptklasse (wie *BuyerAgent*) gekoppelt. Dies erfordert eine Bekanntmachung solcher impliziter semantischer Konventionen. Angesichts des in Systemen mobiler Agenten häufigen Codeaustauschs kann eine systemweite Propagation von Informationen kein konsistentes Bild der Zuordnung von Bezeichnern und Semantik bieten.

5.1.3 Kommunikative Eigenschaften

Anstatt die Basis der Definition eines Agententyps auf Klassen zu errichten, können weitere, funktionale Gesichtspunkte des Agenten herangezogen werden. Von besonderem Interesse sind die Interaktionen zwischen einzelnen Agenten.

Generell ist es üblich, Objekte über *Methodenaufrufe* interagieren zu lassen. Zahlreiche Agentensysteme erlauben entsprechend der Maxime, Objekte mit Mobilität auszustatten, auch den Aufruf von Methoden auf Agenten, damit der aufrufende Agent

eine Dienstleistung des angesprochenen Agenten nutzen und mit diesem Daten austauschen kann. Daten können auch in Form von *Ereignissen* an andere Agenten übermittelt werden, was den Agenten, der das Ereignis empfängt, zu entsprechenden Aktionen veranlassen kann.

Es ist zu beachten, dass die Interaktionen eines Agenten mit einer bestimmten Entität nicht auf genau eine Weise eingeschränkt ist. Wird beispielsweise der Zugang zu einer Systemkomponente durch ein bestimmtes Objekt gewährt – wie etwa das Ansprechen eines SNMP-Agenten durch ein CORBA-Objekt – so unterscheidet sich die Interaktionsform deutlich von jener im Szenario, bei dem der Agent mittels einer Socketverbindung Kontakt aufnimmt. Soll der Agent eine Reiseroute von einem Anwender entgegennehmen, so kann dies grafisch gelöst werden oder durch Deponieren der Route in einer Datei, welche der Agent lesen muss. Es ist erforderlich, dass neben dem Objekt, mit dem der Agent zu interagieren hat, auch die Spezifika der Kommunikation exakt festgelegt werden. Diese Spezifika sind von der Implementierung des Objekts direkt abhängig. Entscheidet man sich im Laufe der Anwendungsentwicklung für eine Umstrukturierung von Objekten, mit denen der Agent in Kontakt treten soll, so hat dies unmittelbare Auswirkungen auf die Gestaltung des Agenten.

Abgesehen von dem Problem, dass in einem offenen System eine Änderung von an der Agentenkommunikation beteiligten Objekten nur in engem Rahmen möglich ist, sind für eine verlässliche Spezifikation eines Zugangs zu diesen Objekten neben den zu übermittelnden Daten auch Implementationsdetails unerlässlich, da es zahlreiche Formen der Interaktion mit der Umgebung gibt. Es ist daher von Vorteil, die Kommunikation zwischen Agenten in einer *homogenen* Weise zu gestalten.

Folgerung 5.2 Homogene Kommunikation
Die Verwendung einer homogenen Kommunikationsform zwischen den interagierenden Komponenten erlaubt die Abstraktion von Implementationsdetails. Die Kommunikation kann anhand der Folge der ausgetauschten Nachrichten vollständig charakterisiert werden.

Wie in Abschnitt 4.5 gezeigt wurde, erfüllen die in AMETAS definierten Stellennutzer die Bedingung der homogenen Kommunikation und eignen sich daher besonders für eine kommunikationsbasierte Typisierung.

5.1.4 Notwendige Eigenschaften des Typsystems

Agentensysteme sind besonders dadurch gekennzeichnet, dass sie Komponenten aufweisen, welche an verschiedenen Stellen aktiv sind, diese Lokationen nach Belieben wechseln oder von außen (im Falle eines *offenen Systems*) in das System kommen und es wieder verlassen. Die Definition eines Typsystems unter diesen Maßgaben ist daher besonders anspruchsvoll. Folgende Bedingungen sollten Grundanforderungen an das Typsystem sein:

Eindeutigkeit	Es soll keine zwei als identisch erkannte Typbeschreibungen geben, welche eine unterschiedliche Menge von Entitäten beschreiben.
Universalität	Es soll keine Entität geben, die nicht typisierbar ist.
Abstraktion	Eine Entität kann durch verschiedene Beschreibungen charakterisiert werden, wenn sie als *gemeinsame Verfeinerung* beider Beschreibungen angesehen werden kann.
Erweiterbarkeit	Die Menge der definierbaren Typbeschreibungen ist nicht begrenzt.
Auswertbarkeit	Der Vergleich von Typbeschreibungen soll algorithmisch lösbar sein.
Korrektheit	Die aus dem Vergleich der Typen gewonnenen Urteile in Bezug auf Kompatibilität sollen zutreffend sein.

Der erste Punkt fordert die Akzeptanz der Beschreibung eines Agenten. Es soll nicht vorkommen, dass eine zutreffende Typisierung an einer anderen Stelle des Systems als nicht zutreffend erachtet wird. Eine Typbeschreibung soll überall dieselbe Menge von Agenten repräsentieren.[2] Hintergrund dieser Forderung ist, dass es sich als schwierig gestalten kann, eine Typbeschreibung im gesamten Agentensystem zu propagieren, dass aber letztlich Einigkeit über die Bedeutung dieses Typs besteht (die *Intension*).

Punkt zwei stellt klar, dass die Typisierung jedes fragliche Objekt erfassen muss. Die Typbeschreibung sollte keine Eigenschaften aufweisen, die eine Typisierung bestimmter Objekte in der Sprache dieser Beschreibung ausschließt. In AMETAS betrifft dies somit alle *Stellennutzer*, da nur diese am Nachrichtenaustausch teilnehmen.[3]

Punkt drei führt das Konzept der Subtypen ein. Entitäten, auf welche zwei Typbeschreibungen zutreffen, sind spezielle Versionen beider Beschreibungen, liegen also in der Schnittmenge der von beiden Beschreibungen induzierten Typmengen. Subtypen stellen sich dabei als Teilmengen dar. Der Übergang zu einer allgemeineren Beschreibung wird Abstraktion genannt. Es ist also möglich, eine Beschreibung fortwährend zu spezialisieren und die Typmenge damit zu verkleinern.

Punkt vier ist notwendig, wenn man von einem offenen System ausgeht. Setzt man Offenheit voraus, so folgt die Aussage dieses Punktes bereits aus Punkt zwei, da das System immer wieder von neuen, bislang unbekannten Agenten besucht werden kann. Diesen Agenten müssen wiederum Typen zugewiesen werden können. Dies ist eine besondere Herausforderung an den Entwurf des Typsystems: Einerseits ist es nicht möglich, eine klar abgegrenzte Menge von Typen zu definieren, andererseits muss diese

[2] Es wird nicht gefordert, dass es nur *eine* Beschreibung eines Agenten geben kann.

[3] Interessant ist vor allem der Umstand, dass ein menschlicher Anwender über seinen Benutzeradapter auch typisierbar wird; dies betrifft insbesondere seine Interaktionen mit dem Agentensystem.

sich ständig verändernde Menge von Typen allen Stellen zugänglich gemacht werden. Dies ist nicht effizient lösbar und vernichtet die Bandbreitenersparnis, die man sich in vielen Szenarien vom Einsatz mobiler Agenten erhofft. Die Typbeschreibung muss also *ad-hoc*-interpretierbar sein; eine Typüberprüfung ist erst zur Laufzeit möglich. Die Interpretierbarkeit wird von Punkt fünf gefordert; diese Bedingung ist wesentlich, soll eine Typüberprüfung zur Laufzeit ohne menschliche Intervention überhaupt möglich sein. Eine *effiziente* Auswertbarkeit ist allerdings nicht gefordert, wenn sie auch wünschenswert wäre.

Schließlich stellt der letzte Punkt eine augenscheinlich triviale Forderung, die aber eminent wichtig ist. Wenn etwa Eigenschaften eines Agenten herangezogen werden, welche *zeitlich nicht konstant* sind, so kann es vorkommen, dass ein von einem Subtyp angeblich stammender Agent unerwartet nicht kompatibel ist.

Die klassische Typisierung anhand von Bezeichnern erfüllt nicht alle aufgeführten Forderungen. Da es sich dabei um eine Abbildung eines Typs auf einen Namen handelt – welcher dann stellvertretend für diesen Typ eingesetzt wird –, muss diese Abbildung an allen betroffenen Lokationen konsistent sein, sonst ist die Eindeutigkeitsanforderung nicht erfüllbar. Soll das System beliebig erweiterbar sein, wird diese Zuordnung immer umfangreicher und muss auf mögliche Konflikte geprüft werden, wenn nämlich gleiche Bezeichner gewählt werden. In diesem Falle ist die Typ-Bezeichner-Zuordnung nicht durchführbar und das betreffende Objekt nicht typisierbar. Die Konsistenz der Zuordnung ist wegen der Beweglichkeit der Agenten nicht zu realisieren, ohne diese erheblich zu behindern.

5.2 Übersicht über die Typdefinition

Da es möglich ist, einem Agenten einen Typ zuzuweisen, ist die Frage interessant, wie genau der Typ einen Agenten spezifiziert, das heißt wie groß die Typmengen sind. Je größer die Mengen werden, umso größer ist die Wahrscheinlichkeit, dass bei der Auswahl eines Vertreters dieses Typs nicht der erwünschte Agent getroffen wird. Ist die Typmenge zu klein, wird es unwahrscheinlicher, überhaupt einen Treffer zu erzielen. Die Typbeschreibung muss also nicht nur allgemein genug sein, um eine Auswahl an möglichen Vertretern zu bekommen, sondern auch präzise genug, um Irrtümern vorzubeugen.

Im Folgenden soll ein Typsystem vorgestellt werden, das die im vergangenen Abschnitt aufgestellten Bedingungen erfüllt. Eine Typbeschreibung setzt sich in diesem System im Allgemeinen aus drei Komponenten zusammen (Abbildung 5.1).

Der *syntaktische Typ* entspricht im Wesentlichen der Schnittstellenbeschreibung, wie sie von verteilten Objektsystemen wie CORBA oder auch von DCE RPC bekannt sind. Es werden die Typen der akzeptierten Eingangsnachrichten und der versendeten Ausgangsnachrichten deklariert. Dadurch wird es möglich, zu erfahren, ob die Instanz eines gegebenen Typs die Nachrichten überhaupt kennt, die von ihr als bekannt voraus-

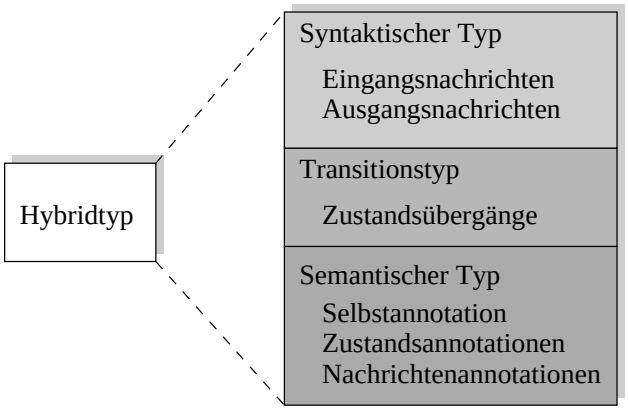

Abbildung 5.1: Aufbau des Hybridtyps

gesetzt werden und ob sie in gewisser Hinsicht voraussehbar darauf reagiert.

Objekte jedoch besitzen einen *Zustand*, welcher Einfluss auf die Menge der aktuell akzeptablen und gesendeten Nachrichten ausüben kann. Es kann daher auftreten, dass es von den bislang erhaltenen Nachrichten abhängt, welche Nachricht die nächste akzeptable ist. Diese Zuordnung trifft der *Transitionstyp*.

Besonders bequem erscheint es, Nachrichten nur in Form von Zeichenketten auszutauschen, da in diesem Falle eine Typumwandlung überflüssig wird. Dies verhindert jedoch eine sinnvolle Typisierung der Nachrichten. Auch bei anderen Datentypen kann es vorkommen, dass unklar ist, ob es sich um die erwartete Nachricht handelt. Einem Objekt, das beispielsweise ein Konto verwaltet, muss klar sein, ob die übergebene Zahl hinzugefügt oder abgebucht werden soll.

Es sind weitere Informationen erforderlich, die Auskunft über die *Bedeutung* oder *Semantik* der jeweiligen Nachricht, des einzelnen Datums oder des gesamten Objekts liefern. Das Hybridtypsystem sieht vor, das gesamte Objekt, aber auch einzelne Komponenten des Nachrichtenaustauschs mit *Annotationen* zu versehen. Da die ersten beiden Komponenten des Hybridtyps Querverweise zu Elementen der dritten Komponente, dem *semantischen Typ*, aufweisen, wird zunächst dieser Typbestandteil vorgestellt.

5.3 Semantischer Typ

Unter der *Semantik* einer Nachricht, eines Datums oder eines Objekts versteht man, anschaulich beschrieben, die *Bedeutung* der fraglichen Entität, also eine der Entität zugeordnete Information, welche eine Auskunft gibt, die in der Darstellung der möglichen Kommunikationsarten nicht repräsentiert wird.

5.3.1 Semantische Informationen

Semantische Informationen können beispielsweise sein:

- *Einsatzzweck*: Welchem Ziel dient der Einsatz der Entität?
- *Vorbedingungen*: Welche Situation muss gegeben sein, damit der Einsatz der Entität den erwarteten Zweck erfüllt?
- *Nachbedingungen*: Welche Situation herrscht nach dem Einsatz der Entität vor?

Dabei kann man gegebenenfalls den Einsatzzweck durch die Nachbedingungen formulieren: Das Ziel des Einsatzes ist, dass das von der Entität beeinflussbare System im Sinne der Erfüllung des Zwecks verändert wird. Allerdings kann der Zweck, welcher einen möglicherweise in der Zukunft liegenden Zustand bezeichnet, nicht allgemein anhand eines isolierten Einsatzes einer Entität unmittelbar erkannt werden. Soll ein Agent eine Handelsstatistik erstellen, so kann er dies durch Zusammentragen von Handelsdaten an einzelnen Stellen erreichen, wobei jedes einzelne Zusammentragen für sich genommen nichts über den Zweck aussagt.

Die Vorbedingungen spielen unter anderem dann eine Rolle, wenn eine Nachricht verschickt wird, sobald das System einen bestimmten Zustand einnimmt. So ist der Einsatz der Nachricht als Abschicken der Nachricht zu verstehen, mit deren Hilfe ein Benutzer zum Beispiel über die aktuelle Situation in Kenntnis gesetzt wird. LISKOW und WING beschreiben in [LW93] eine Repräsentation von *Vor-* und *Nachbedingungen* für den Aufruf von Methoden. Anhand dieser Informationen ist zu erkennen, welchen Einfluss der Aufruf dieser Methode auf den Zustand anderer Komponenten des Systems hat.

Die Formulierung semantischer Informationen stößt oftmals auf erhebliche Schwierigkeiten:

- Der Zweck entspringt meist einer *Idee* des Anwenders, welcher sich außerstande sieht, diese Idee zu formalisieren.
- Für die Formalisierung der Idee bedarf es einer formalen Repräsentation einer Begriffswelt (*Ontologie*), mit deren Hilfe die formale Darstellung mit einer realen Bedeutung verknüpft werden kann.
- Selbst einfache Gegebenheiten des alltäglichen Lebens erfordern eine bislang nicht überschaubare Menge von Informationen in der Ontologie.

Zu große Ontologien sind von heutigen Algorithmen nicht in angemessener Zeit zu verarbeiten; sind die Ontologien zu klein, ist die Formulierung der Idee eventuell unmöglich. Ein weiteres Problem ist der *Vergleich* semantischer Informationen:

- Die Darstellung semantischer Informationen ist nicht eindeutig.

Dies liegt vor allem im Menschen selbst begründet, da ein jeder die Welt „mit seinen Augen“ wahrnimmt. Was ein Mensch als zentralen Punkt der Information ansieht, kann von einem anderen als Nebensächlichkeit eingestuft werden. Entsprechend wird die jeweilige Beschreibung unterschiedlich ausfallen. Es erfordert eine grundlegende Kenntnis des Wesens beider Menschen, um Parallelen oder Widersprüche zu identifizieren. Damit stellen sich folgende Forderungen an eine Repräsentation semantischer Informationen:

1. Die Ontologie muss sorgfältig zusammengestellt sein; sie sollte das Einsatzgebiet bestmöglich abdecken. Gegebenenfalls muss das Einsatzgebiet so präzisiert werden, dass Missverständnisse minimiert werden.

2. Die Repräsentation sollte so beschaffen sein, dass sie die in ihr enthaltenen Informationen stets gleichartig reproduzierbar wiedergibt.

3. Diese dadurch implizierte Struktur sollte einen Vergleich solcher Informationen erleichtern.

Interessant ist zu beobachten, dass eine große Ontologie zwar die Formalisierung von Informationen erleichtert, den Vergleich aber erheblich kompliziert. Je größer die Ontologie ist, umso zahlreicher werden die Möglichkeiten, gleichartige Sachverhalte unterschiedlich zu beschreiben. Diese Freiheiten erfordern ihrerseits eine Erweiterung der Ontologie, deren Größe beliebig anwachsen kann.

5.3.2 Annotationen und Nachrichten

Möchte man Entitäten mit weiteren Angaben ausstatten, so kann man eine Zuordnung definieren, welche jede Entität auf genau eine Information abbildet. Diese Information nennt man *Annotation* (Anmerkung); sie wird in der Regel als ein Zusatz verstanden. Eine Nachricht, welche aus einer Zeichenkette besteht, kann durch eine Annotation zu einer Nachricht werden, welche einen gesuchten Gegenstand benennt; eine andere Annotation könnte denselben Inhalt zu einem Angebot des benannten Gegenstands erklären. Die Bedeutsamkeit der Annotation kann für unterschiedliche Empfänger variieren.

Betrachtet man Objekte, welche über Nachrichten miteinander kommunizieren, so gibt es zahlreiche Einsatzmöglichkeiten für Annotationen. Beschreibungen können beispielsweise definiert werden für

- eintreffende/abgehende Nachrichten;
- Bestandteile einer Nachricht;
- Absender/Empfänger einer Nachricht;

- Zustände des Objekts und
- das Objekt selbst.

Nachrichten können mittels Annotationen spezifische semantische Informationen zugewiesen werden. Eine solche Information könnte sein, dass die Nachricht ein gesuchter Gegenstand ist; oder man deklariert die Nachricht als *Fehlermeldung*. Ein Feld von Zeichenketten könnte einerseits eine Liste von Benutzern sein, aber vielleicht auch eine Reiseroute. Diese semantischen Informationen sind im Inhalt der Nachricht nicht notwendigerweise explizit vorhanden.

Anstatt komplette Nachrichten zu kennzeichnen kann man auch einzelne Komponenten der Nachricht erklären. Während die Einträge einer Adressangabe in Form mehrerer Zeichenketten vorliegen, kann jede dieser Zeichenketten durch eine geeignete Annotation spezifiziert werden (wie *Name* oder *Straße*).

Gewöhnlich werden bei Angaben zu den Nachrichten, welche im Zusammenhang mit einem Objekt auftreten, keine Angaben in Bezug auf Absender oder Empfänger gemacht. Dies liegt an der konventionellen Klient-Server-Sichtweise auf Objekte, deren Dienstangebot ausschließlich dem Klienten gegenüber als relevant erachtet wird. Das heißt, dass Aktionen, welche *nicht* die Interaktionen zwischen dem Klienten und dem Objekt als Server betreffen, bislang nicht von Belang waren.

Im allgemeinen Fall, insbesondere bei *aktiven Objekten* [Nie95], ist die Verteilung Klient/Server nur eingeschränkt sinnvoll. Andere Kommunikationspartner können den Gang des Nachrichtenaustauschs beeinflussen, indem sie zum Beispiel notwendige Daten liefern, um die Verarbeitung einer Nachricht eines Anfragers zu ermöglichen. Es kann sich somit als wichtig erweisen, die *Absender* und *Empfänger* einer Nachricht zu kennen.

5.3.3 Zustandsannotationen

Der *(beobachtbare) Zustand* eines Objekts definiert sich als die Gesamtheit der im Objekt vorhandenen Informationen, welche das Verhalten des Objekts in Bezug auf die Interaktion mit nicht im Objekt befindlichen Entitäten beeinflussen. Die Programmiersprache Java bietet dazu Instanzfelder an (*Fields* oder *Members* genannt), welche während der Ausführung des Objekts mit Werten belegt werden (siehe auch Abschnitt 5.5.1).

Im Allgemeinen manifestiert sich ein Zustand eines Objekts nur durch das spezifische Verhalten in diesem Zustand; er ist also von externen Entitäten bestenfalls durch Interaktionen mit dem Objekt feststellbar. Es können Zustände existieren, welche von außen in bestimmten Situationen nicht unterscheidbar sind. Zustände lassen sich somit *nicht* eindeutig durch die Folge von Interaktionen mit dem Objekt charakterisieren. Möchte man Zustände besonders kennzeichnen, so bieten sich Annotationen an. Einem Zustand kann mit Hilfe einer Annotation eine Bedeutung *(auf Eingabe wartend, Endzustand)* oder ein Name *(Zustand_4)* zugewiesen werden.

Zustandsannotationen haftet in ihrer Anwendung ein gewisses Risiko an. Im Allgemeinen lassen sich kaum Rückschlüsse auf die tatsächlichen Zustände und ihre Wechsel gewinnen, wenn man die eigentliche Implementierung nicht kennt. Neben dem beobachtbaren Verhalten, repräsentiert durch den Nachrichtenaustausch, führt man durch derart gekennzeichnete Zustände eine weitere Spezifizierung ein, welche für sich genommen nicht beobachtbar ist. Damit droht eine *Überspezifizierung*. Es ist durchaus möglich, dass zwei Objekte in ihrem beobachtbaren Verhalten völlig identisch sind und auch für dieselben Aufgaben eingesetzt werden können. Ihre Implementierung kann jedoch unterschiedlich sein und Zustände mit anderer Semantik und somit anderen Annotationen einführen.

Das Hybridtypsystem erlaubt die Definition von Zustandsannotationen. Der Anwender kann jedoch bestimmen, ob er diese Annotationen als Kriterium auf der Suche nach geeigneten Instanzen oder Typen zulässt oder nicht.

5.3.4 Selbstannotation

Während die Annotation von Zuständen oder Nachrichten einen detaillierten Vergleich zweier bestehender Beschreibungen ermöglicht, sieht sich der gewöhnliche *Anwender* vor dem Problem, *nichts* über die Kommunikationsweisen eines Agenten oder gar seiner inneren Zustände zu wissen. So könnte eine Motivation des Anwenders sein, einen Agenten zu suchen, der eine bestimmte Arbeit erledigt, ohne dass Details bekannt wären, *wie* diese Arbeit zu erledigen ist. Ein Netzadministrator könnte einen Überwachungsagenten benötigen, der alle Rechner seines Netzes auf kritische Zustände (wie knappen Plattenplatz) untersucht.

Im Hybridtypsystem ist für diesen Zweck die Möglichkeit integriert, den Agenten selbst zu annotieren. Die *Selbstannotation* soll über den Zweck des Agenten Auskunft geben. Sie ermöglicht eine Vermittlung nach der in [PG97, Pud97] beschriebenen Weise mit Hilfe eines *Traders*; Interessenten formulieren ihre Anfrage, leiten diese an einen solchen Vermittler weiter, welcher ihnen entweder eine existierende Instanz oder einen Typ angibt, welcher eine zur Anfrage passende Instanz liefert.

Da mit Ausnahme der Selbstannotation alle Annotationen den jeweiligen Entitäten (Nachrichten, Daten, Zustände) direkt zugeordnet sind und getrennt von diesen keinen Nutzen bringen, kann bei einem *rein semantischen Vergleich* nur die Selbstannotation herangezogen werden. Möchte man anschließend sichergehen, dass eine Kommunikation planmäßig verläuft, muss man diese Details aus dem vermittelten Typ gewinnen.

Definition 5.3 Semantischer Typ
Unter dem semantischen Typ (im weiteren Sinne) eines Agenten versteht man die Gesamtheit aller Annotationen, die in der Typbeschreibung vorhanden sind. Diese beziehen sich auf Nachrichtenelemente, Nachrichten, Zustände, Absender, Empfänger und die Entität selbst. Der semantische Typ (im engeren Sinne) ist lediglich die Selbstannotation der typisierten Entität.

Im Folgenden wird unter dem semantischen Typ stets die Selbstannotation verstanden, wenn es nicht anders angegeben wird.

5.3.5 Repräsentation und Vergleich der Annotationen

Beim Vergleich semantischer Informationen ist von Interesse, ob die durch eine Formalisierung repräsentierte Begebenheit dieselbe ist wie jene einer zweiten Formalisierung oder ob es Implikationen der einen Begebenheit zur anderen gibt, ob also letztere eine Folge der Gültigkeit der ersteren ist. Handelt es sich um beschreibende Texte, werden diese vom menschlichen Benutzer gelesen und interpretiert; dabei werden eigene Vorstellungen mit den Informationen verglichen, um eine Aussage über die Übereinstimmung zu treffen. Ziel der Formalisierung muss es sein, dass ein solcher Vergleich durch einen Algorithmus durchgeführt werden kann. Der im Folgenden verwendete Formalismus wurde bereits in Abschnitt 3.5 eingeführt.

Definition 5.4 Annotationen
Die Funktion *ann* ordnet Entitäten eine logische Aussage über diese Entität zu. Man nennt $\alpha = ann_O(E)$ die *Annotation* von E auf Basis der Ontologie O. Ist die Aussage wahr, dann spricht man von einer *korrekten* Annotation.
Eine Annotation α impliziert eine Annotation β in Bezug auf die Ontologie O (Notation $\alpha \prec^O \beta$), wenn $(\alpha \downarrow O) \wedge (\beta \downarrow O) \wedge (O, \alpha \vdash \beta)$. Ist die Ontologie festgelegt und bekannt, wird kurz $\alpha = ann(E)$ und $\alpha \prec \beta$ geschrieben.

Im Folgenden wird bisweilen davon gesprochen, dass α *konform* zur Annotation β ist, wenn $\alpha \prec \beta$ gilt. In der Regel findet man bei konformen Annotationen eine Verfeinerung einer Annotation vor; in diesem Sinne ist „*Odysseus* ist ein Agent“ eine Verfeinerung von „*Odysseus* ist ein Stellennutzer“, wenn man die entsprechende Ontologie verwendet.

Zeichenketten

Das Hybridtypsystem definiert eine einfache Struktur auf Zeichenketten: Eine Zeichenkette kann die Form *a* oder *a* : *b* haben, wobei *b* als ein möglicher Wert von *a* gelten soll. Auf diese Weise können in einfacher Weise *Konstanten* definiert werden. Beispiele:

„Befehl“

„Befehl:Berechnen“

„Antwort“

Der erste und dritte Fall beschreiben jeweils Begriffe als eine Menge von Entitäten; das können in diesem Falle Nachrichten sein. Der zweitgenannte Fall beschreibt eine

spezielle Instanz des Begriffs „Befehl“. Eine Instanz kann auch als Begriff für sich alleine stehen; dadurch verliert man aber die Möglichkeit der Hierarchisierung.

Annotationen können, wie oben bereits eingeführt, in Form einfacher Zeichenketten repräsentiert werden. Als Struktur auf den Zeichenketten sei lediglich die Notation $a : b$ gegeben, welche b als Instanz von a kennzeichnet. Eine Ontologie benötigt dadurch lediglich Konzepte und deren Implikationen, nicht aber Relationen. Das Hybridtypsystem verzichtet hier auf eine Ontologie; jede mögliche Zeichenkette ist damit gültig und es gibt keine Implikationen zwischen den Zeichenketten.

Regel 5.5 Vergleich zweier Zeichenketten
Seien $\alpha = a : c$ und $\beta = b : d$ zwei Annotationen auf Basis von Zeichenketten. Es ist $\alpha \prec \beta$ genau dann, wenn gilt:

- $a = b$ und
- $d \neq \varepsilon \Rightarrow c = d$

Die Regel erlaubt eine Abstraktion der Instanzen auf Basis der Begriffe. Wenn dabei die Annotation β eine konkrete Instanz bezeichnet, dann muss auch α diese Instanz bezeichnen. Damit ist „Agent:Odysseus“ eine konforme Annotation zu „Agent“, aber nicht zu „Stellennutzer“. Umgekehrt ist „Agent“ keine konforme Annotation zu „Agent:Odysseus“.

Konzeptgraphen

Konzeptgraphen können verschiedenartigste Informationen darstellen; die Ontologie definiert die verwendbaren Konzepte und Relationen. Die Ontologie sollte daher zumindest jene Begriffe erklären, welche in einer Annotation auftauchen können:

[Stellennutzer]	beschreibt als Selbstannotation den Stellennutzer;
[Nachricht]	ist die Kennzeichnung einer Nachricht;
[Nachrichtenelement]	kann zur Kennzeichnung eines Nachrichtenelements herangezogen werden;
[Zustand]	dient der Annotation von Zuständen.

Um zwei Annotationen auf Basis von Konzeptgraphen zu vergleichen, dient folgende

Regel 5.6 Vergleich zweier Konzeptgraphen
Seien $\alpha = (n_\alpha, I_\alpha, \{r_\alpha^1, \ldots, r_\alpha^m\})$ und $\beta = (n_\beta, I_\beta, \{r_\beta^1, \ldots, r_\beta^n\})$ zwei Annotationen auf Basis von Konzeptgraphen und O eine Ontologie. Es ist $\alpha \prec \beta$ genau dann, wenn gilt:

- $O, n_\alpha(x) \vdash n_\beta(x)$
- $I_\beta \subset I_\alpha$
- $\forall i \in \{1, \ldots, n\} \exists j \in \{1, \ldots, m\} : r_\alpha^j \prec r_\beta^i$

wobei für zwei Relationsteilgraphen $r = (n, c_1, \ldots, c_k)$ und $r' = (n', c'_1, \ldots, c'_l)$ genau dann $r \prec r'$ gilt, wenn $O, n(x) \vdash n'(x)$, $k = l$ und $\forall i = 1, \ldots, k : c_i \prec c'_i$ ist.

Man beachte, dass $n(x)$ das dem Konzept- oder Relationsnamen n entsprechende Prädikat ist („x ist ein n“). Konzeptgraphen müssen nicht unbedingt eine komplexe Ontologie zur Verfügung gestellt bekommen, um sinnvoll einsetzbar zu sein. PUDER beschreibt in [Pud97] das Konzept der *syntaktischen Typkonformität* bei Fehlen einer Ontologie gegenüber der *semantischen Typkonformität*, wenn eine Ontologie gegeben ist. Das Fehlen einer Ontologie verringert allerdings die Chancen, semantische Entsprechungen aufzudecken. Zu weiteren Details in Bezug auf den Konzeptgraphenvergleich sei auf [Sow84] und [PG97] verwiesen.

Damit ergibt sich als Regel zur Bestimmung der Konformität des semantischen Typs:

Regel 5.7 Semantische Konformität
Eine Typbeschreibung A ist semantisch konform zur Typbeschreibung B ($A \prec_{sem} B$), wenn der semantische Typ von A zu jenem von B konform ist, das heißt für die Selbstannotationen α (von A) und β (von B) gilt: $\alpha \prec \beta$. A wird dann *semantischer Subtyp* von B genannt.

5.4 Syntaktischer Typ

Der Begriff *syntaktischer Typ* geht zurück auf die *syntaktische Spezifikation* eines Objekttyps. Dahinter verbirgt sich die Strategie, die Eigenschaften eines Objektes anhand seiner *Schnittstelle* festzulegen, welche die *Aufrufsyntax* anhand der enthaltenen Methodensignaturen definiert. Die Schnittstelle kann herangezogen werden, um einerseits noch in der Übersetzungsphase, andererseits auch zur Laufzeit eine Nichtübereinstimmung der Datentypen aktueller und formaler Parameter festzustellen. Der syntaktische Typ repräsentiert die Typen eingehender und ausgehender Nachrichten und findet während der Laufzeit seine Anwendung.

5.4.1 Kovarianz und Kontravarianz

LISKOV und WING haben in [LW93] und [LW94] dargelegt, dass die Subtyprelation sich nicht nur auf die Menge vorhandener Methodensignaturen beziehen muss, sondern auch die Methodensignaturen *als solche* betreffen kann.

Prinzipiell ist es dem Empfänger eines Datums nicht möglich zu sagen, ob dieses einer Menge entspringt, die kleiner als erwartet ist, also ob die Menge einen Subtyp repräsentiert. Überlegt man sich, wie die Rollen von Sendern und Empfängern im Falle eines Methodenaufrufs verteilt sind, gelangt man unmittelbar zur Ko-/Kontravarianzregel, welche in Abschnitt 3.4.2 bereits dargestellt wurde.

Intuitiv neigt man zu urteilen, dass eine Methode, die als Eingabe einen allgemeineren Typ erwartet, sicherlich nicht das Erwartete leistet, da sie wahrscheinlich von der Spezialisierung des Arguments abstrahiert. Analog dazu könnte eine Einschränkung des Ergebnisraums so interpretiert werden, dass das Objekt nicht alle erwünschten Ergebnisse liefern wird. Gegen diese Vermutung ist anzuführen, dass die Implementierung sehr wohl mit Spezialisierungen von Datentypen umgehen könnte (interne Typumwandlung oder *Cast*) und es im anderen Falle nicht gesichert ist, dass die Implementierung tatsächlich auf Spezifika des Subtyps zurückgreift.

5.4.2 Agenten und Nachrichten

Die Bemerkungen aus Abschnitt 5.1.1 legen die Verwendung von *Nachrichten* zur Kommunikation zwischen Agenten nahe. Eine Nachricht ist ein Block von Daten, welcher meist eine bestimmte, einheitliche Struktur aufweist und in dieser Form von den Empfängern interpretiert werden kann.

Definition 5.8 Nachrichtentypen
Ein *Nachrichtenelementtyp* ist eine Datenstruktur der Form $e = (\alpha, t)$ mit einer Annotation $\alpha = ann(e)$, welche die Bedeutung des Nachrichtenelements festlegt und einem Datentyp $t = typ(e)$, welcher den Typ des Datums festlegt.
Ein *Nachrichtentyp* ist eine Datenstruktur der Form $a = (\alpha, E)$ mit einer Annotation $\alpha = ann(a)$, welche die Bedeutung der Nachricht festlegt sowie einem Vektor $E = el(a)$ von Nachrichtenelementtypen.

Bei der Festlegung, welchen Typs die Elemente sein können, ist Vorsicht geboten: Sind anwendungsspezifische Typen (beziehungsweise Klassen) erlaubt, muss sichergestellt sein, dass der Empfänger dieselben bezeichneten Klassen verwendet. Erhält dieser zum Beispiel zwei Nachrichten von verschiedenen Absendern, welche eine unterschiedliche Implementierung einer gleich lautenden Elementklasse verwenden, so ist unklar, welche Definition für den Empfänger verbindlich ist. In AMETAS sind daher für Nachrichtenelementtypen nur allgemein verfügbare, serialisierbare Klassen erlaubt. Da es keine definierten Verwandtschaftsbeziehungen in Java zwischen Basistypen und

Referenztypen (Klassen) gibt, müssen Basistypen wie *boolean* oder *int* in einem Objekt eingebettet sein. (Genaueres hierzu ist in Anhang B zu finden.)

Um zwei Nachrichtentypen miteinander vergleichen zu können, definiert man den Begriff der Nachrichtenkonformität[4]:

Regel 5.9 Nachrichtenkonformität
Seien a_1 und a_2 zwei Nachrichtentypen. Die Nachricht a_1 wird *konform* zur Nachricht a_2 genannt ($a_1 \prec a_2$), wenn gilt:

- Die Annotation von a_1 impliziert jene von a_2: $ann(a_1) \prec ann(a_2)$.
- Die Länge der Nachricht a_1 ist nicht kleiner als jene von a_2: $|el(a_1)| \geq |el(a_2)|$.
- Für alle $i = 1, \ldots, |el(a_2)|$ gilt: $ann(el(a_1)[i]) \prec ann(el(a_2)[i])$ und $typ(el(a_1)[i]) \prec typ(el(a_2)[i])$.

Die Postfix-Operation $[i] : e \mapsto e_i$ ist definiert als die Projektion des Vektors e auf seine i-te Komponente. Es müssen also zum einen die Annotationen der Nachrichten selbst konform sein als auch paarweise die Elemente. Dabei dürfen stets Elemente am Ende *hinzugefügt* werden – ein Zuviel an Daten kann durch Projektion immer behoben werden, wohingegen fehlende Daten nie ersetzt werden können. Dies erweitert die von Liskov und Wing genannte Ko- und Kontravarianzregel für Methoden. Im Folgenden wird auf diese Definition als *Ko- und Kontravarianzbedingung für Nachrichten* Bezug genommen. Der syntaktische Typ des Agenten wird nun wie folgt definiert:

Definition 5.10 Syntaktischer Typ
Der *syntaktische Typ* eines Agenten ist definiert als das Paar $T_{syn} = (N_{ein}, N_{aus})$, wobei N_{ein} und N_{aus} Mengen von Nachrichtentypen sind. Man nennt N_{ein} die *Eingangsnachrichten* und N_{aus} die *Ausgangsnachrichten*. Ein Agent hat den syntaktischen Typ T_{syn} genau dann, wenn er mindestens alle Nachrichten akzeptiert, für die es jeweils eine konforme Nachricht in den Eingangsnachrichten von T_{syn} gibt und höchstens solche Nachrichten sendet, zu denen es konforme Ausgangsnachrichten in T_{syn} gibt.

Der syntaktische Typ alleine besitzt eine *schwächere* Aussagekraft als die syntaktische Spezifikation von Objekten über Schnittstellen: Er trifft keine Aussage über einen Zusammenhang zwischen Eingangs- und Ausgangsnachrichten, wie es Schnittstellen durch die Signaturen leisten. Ebenso wenig vermag er anzugeben, ob das Objekt eine Nachricht *zu jeder Zeit* akzeptiert. Nur wenn die Implementierung eine Zustandslosigkeit bezüglich dieser Nachrichtenverarbeitung vorsieht, kann man die Akzeptanz aller genannten Nachrichten zu jeder Zeit fordern.

[4] Eigentlich muss es *Nachrichtentypkonformität* heißen, doch die Idee der Konformität basiert stets auf Typen, sodass hier und im Folgenden der Kürze wegen von Nachrichten statt von Nachrichtentypen gesprochen wird.

```
("Einzahlen", 10.0, 1234567)
("Abheben", 10.0, 1234567)
("Kontostand", 1234567)
("Fehler","Kontonummer fehlt")
("OK")
```

Abbildung 5.2: Üblicher Gebrauch von Diskriminatoren

Regel 5.11 Syntaktische Konformität
Ein syntaktischer Typ A_{syn} ist konform zu einem syntaktischen Typ B_{syn} ($A_{syn} \prec B_{syn}$) genau dann, wenn das durch A_{syn} charakterisierte Objekt ebenfalls den Typ B_{syn} aufweist. Ein Typ A ist *syntaktisch konform* zu einem Typ B ($A \prec_{syn} B$) genau dann, wenn der zu A gehörende syntaktische Typ konform zum syntaktischen Typ von B ist.

5.4.3 Diskriminatoren

Im Unterschied zu Methoden müssen Nachrichten nicht unbedingt eine Kennzeichnung wie den *Methodennamen* tragen, welche ihre Verarbeitung beim Empfänger beeinflusst. Ohne eine solche Kennzeichnung entscheidet der Empfänger anhand der Form der Nachricht, wie er diese Daten zu verarbeiten hat. Sind Nachrichten so allgemein wie oben zu sehen definiert, dann lassen sich Nachrichten im Prinzip *nur* an ihrer Struktur auseinanderhalten, also an den in ihnen enthaltenen Datentypen.

In den meisten Situationen ist es jedoch nicht praktikabel, alle Nachrichten nur anhand ihrer Struktur zu unterscheiden. Diese Vorgehensweise würde die Auswertung von Nachrichten erheblich erschweren, da zunächst die Datentypen zu prüfen sind, bevor der Inhalt gelesen werden kann. Ferner leidet die Lesbarkeit des Codes, denn das Verhalten des Empfängers würde maßgeblich von der Form, nicht vom Inhalt der Nachricht bestimmt.

Die von Implementierern bevorzugte Vorgehensweise ist vielmehr, die Art der Nachricht anhand des Inhalts oder einer Komponente festzulegen. Zweckmäßigerweise wird gleich das erste Element der Nachricht verwendet; es kann dabei eine Zeichenkette tragen oder irgendein anderes, zulässiges Objekt, etwa eine Zahl. Aufgrund der besseren Lesbarkeit werden aber erstere bevorzugt und ersetzen damit die von Methodenaufrufen bekannten Methodennamen (siehe Abbildung 5.2). Diese speziellen Datenelemente werden im Folgenden als *Diskriminatoren* bezeichnet, da sie über die Verarbeitung der eventuell folgenden Daten bestimmen. Wie zu erkennen ist, können Nachrichten formal identisch sein, sich aber in ihrer Bedeutung beträchtlich unterscheiden.

Die Überprüfung des Inhalts einer Nachricht fällt in der Implementierung unter Java mindestens genauso leicht wie die Feststellung des Typs der Nachricht, sodass in der Tat eine Diskriminatorauswertung zu empfehlen ist und auch häufig angetroffen

wird. Ein Typsystem, das auf solchen Nachrichten basiert, muss dies berücksichtigen. Das Hybridtypsystem führt die Idee der *Konstantenannotation* ein, um solche Diskriminatoren zu repräsentieren. Grundlage für diese Repräsentation ist die Tatsache, dass

- vergleichsweise wenige Diskriminatoren im Code verwendet werden können, da die Anzahl der von ihnen referenzierten Funktionalitäten begrenzt ist und
- Diskriminatoren in ihrer Semantik konstant sind, da eine wechselnde Zuordnung den Implementations-, aber auch den Fehlersuchaufwand beträchtlich erhöhen würde.

Bei Konstantenannotationen handelt es sich um Annotationen von Entitäten (in der Regel Nachrichtenelementen), wobei jeweils eine einzelne, spezifische Instanz bezeichnet wird. Die in Abbildung 5.2 zuoberst gezeigte Nachricht mit Diskriminator hätte dann folgenden Typ:

$$n_1 = ((String, \alpha_1), (float, -), (long, -))$$
$$\alpha_1 = Konstante{:}„Einzahlen“$$

Diskriminatoren können auch als besondere Subtypbildung des Datentyps *String* verstanden werden, wobei die Menge aller Zeichenketten auf die eine bezeichnete Zeichenkette eingeschränkt wird. Die Nachrichtenkonformität lässt sich somit unmittelbar auf Diskriminator-ergänzte Nachrichten erweitern:

Regel 5.12 Konformität von Nachrichten mit Diskriminatoren
Seien $a = (e_1, \ldots, e_n)$ und $b = (f_1, \ldots, f_m)$ zwei Nachrichten; ohne Einschränkung sei $m = n$ angenommen. Ist e_i ein durch die Konstante c annotiertes Nachrichtenelement, so ist für $a \prec b$ notwendig, dass $typ(e_i) \prec typ(f_i)$ und

- f_i ein durch dieselbe Konstante c annotiertes Nachrichtenelement ist *oder*
- f_i nicht annotiert ist.

Ist e_i nicht durch eine Konstante annotiert, so darf f_i ebenfalls nicht annotiert sein.

Wenn man also erwartet, dass das Objekt irgendeine Zeichenkette liefert, dann muss auch eine konstante Zeichenkette akzeptabel sein. Umgekehrt verlangt man, dass ein Objekt nicht eine beliebige, sondern die erwartete konstante Zeichenkette liefert. Für ungleich lange Nachrichten gelten wieder die Bemerkungen zur Ko- und Kontravarianz von Nachrichtentypen.

Prinzipiell könnte die Forderung in der Folgerung gelockert werden: Wenn e_i ohnehin eine Konstante ist, dann ist die Bedingung $typ(e_i) \prec typ(f_i)$ zu scharf; ein konstanter *Integer*-Wert von 1 kann in jedem Falle auch als eine *Short*-Instanz interpretiert

werden, da der gesamte *Integer*-Bereich zu einem einzigen Wert entartet ist, der zudem in den Bereich des Subtyps *Short* fällt. Dies stellt aber nicht nur höhere Ansprüche an die Implementierung des Typvergleichers, sondern erfordert auch in der realen Implementierung durch die Einführung von Bereichsprüfungen und Typumwandlungen erhöhten Aufwand. Viele Programmiersprachen (wie Java) bieten keine Bereichstypen und dementsprechend keine automatische Bereichsprüfung an.

Wie nun im konkreten Falle eine Konstante zu formulieren ist, hängt von der Repräsentation der jeweiligen Annotation ab. Sowohl bei Zeichenketten wie bei Konzeptgraphen existieren mannigfaltige Möglichkeiten, eine Formulierung zu finden, sodass eine sinnvolle *Konvention* gefunden werden muss. Zeichenketten-orientierte Annotationen verfügen zumeist nicht über das Hilfsmittel einer Ontologie und man sollte daher als Annotation

Konstante:1 oder *Constant:1*

als Beispiel einer Konstante mit Wert 1 wählen. Andere plausible Möglichkeiten wären *Wert:1* oder *Befehl:Rückkehr*, was die Notwendigkeit einer Konvention unterstreicht.

Im Falle von Konzeptgraphen liegt meist eine Ontologie vor, sodass die präzise Formulierung einer Konstanten weniger kritisch ist. Der erste Graph ist eine Spezialisierung des zweiten:

[Nachrichtenelement]->(beinhaltet)->[Konstante:1]

[Nachrichtenelement]->(beinhaltet)->[Wert:1]

vorausgesetzt dass die Ontologie eine „Konstante“ als besonderen „Wert“ kennt.

5.4.4 Mobilität und Nachrichten

Die Typspezifikation erlaubt die Repräsentation von *Mobilität.* Zu diesem Zweck gibt es einen Pseudo-Nachrichtenelementtyp *migration,* welcher stets an die aktuelle Stelle geschickt wird, wenn der Agent eine Migration durchführen will. Es ist dabei unerheblich, wie der tatsächliche Migrationsmechanismus gestaltet ist; in AMETAS ruft der Agent eine Methode seines Treibers auf, anstatt eine Nachricht zu schicken. Spezifika wie Sicherheitsvorkehrungen bei der Migration werden in der Typbeschreibung ignoriert.

Ein Agent, welcher im *out*-Block seiner Typbeschreibung einen solchen Nachrichtenelementtyp aufführt, erklärt sich als *mobiler Agent.* Ein Beispiel eines solchen Agenten ist in Anhang F zu sehen. Die Modellierung der Migration als Ausgangsnachricht ermöglicht eine einfache Prüfung zur Laufzeit und kann dazu dienen, einem Agenten die Migration nur zu gestatten, wenn er diesen Nachrichtentyp deklariert.

5.5 Transitionstyp

Die Mächtigkeit der Beschreibung durch den syntaktischen Typ alleine ist in einer Hinsicht geringer als von den üblichen Schnittstellenbeschreibungssprachen gewohnt. Methodensignaturen beschreiben Beziehungen zwischen Eingangs- und Ausgangsdaten – in Form von Eingabe- und Ausgabe- oder Rückgabeparametern einer Methode. Der syntaktische Typ hingegen liefert keine Zusammenhänge zwischen Eingangs- und Ausgangsnachrichten.

Man beachte jedoch, dass die so gebräuchlichen Methodensignaturen nur *eine* vorgegebene Art von Kommunikation repräsentieren, nämlich eine Nachricht aus einer festen Folge von Eingangsparametern von einer bestimmten Ausgangsnachricht (welche alle Rückgabeparameter umfasst) beantworten zu lassen. Die Kommunikationspartner sind implizit vorgegeben, nämlich einerseits der Inhaber der Schnittstelle und andererseits der die Kommunikation initiierende Klient.

Für ein System autonomer Agenten sind diese Vorgaben zu restriktiv. Sie setzen ein genau definiertes Klient-Server-Verhältnis voraus und sind auch nicht in der Lage, mögliche Kommunikationsvorgänge mit weiteren Objekten zu repräsentieren.

5.5.1 Zustände und Transitionen

Grundlage der folgenden Ausführungen sind die Begriffe *Zustand* und *Transition*. Der Begriff Zustand geht auf die Definition des *Objekts* nach Booch [Boo91] zurück, welcher als kennzeichnende Merkmale eines Objekts dessen Zustand, Identität und Verhalten herausstellt. Da in realen Implementierungen nur ein endlich großer Speicher zur Verfügung steht, werden im Folgenden nur solche Systeme mit endlichen Zustandsmengen betrachtet.

Auch wenn der Zustand eines Objekts von der Belegung jeder einzelnen Speicherzelle abhängen kann, ist eine solch feingranulare Sichtweise in den meisten Fällen ungeeignet. Auf der Menge der Zustände kann stets eine Klasseneinteilung definiert werden, welche Zustände als *äquivalent* erklärt. Zwei Zustände sind dann einander äquivalent, wenn sich das *Verhalten* des Objekts in den beiden Zuständen nicht unterscheidet. Das Verhalten wird anhand von *Tests* und *Beobachtungen* beurteilt; von Interesse ist dabei, welche Nachrichten vom Objekt akzeptiert und welche von ihm geliefert werden. Zustände werden daher nicht anhand der tatsächlichen Speicherbelegung definiert, sondern in abstrakter Form. Eine (Zustands-)Transition ist der Vorgang, bei dem sich der aktuelle Zustand eines Objekts ändert.

Definition 5.13 Zustände und Transitionen
Jedem Objekt O ist eine eindeutig festgelegte, endliche Menge Q zugeordnet, welche *Zustandsmenge* genannt wird. Ein *(abstrakter) Zustand* $q = st(O,t)$ ist ein Element von Q und steht stellvertretend für den Zustand des Objekts zum Zeitpunkt t. Man schreibt kurz $q = st(O)$ für den aktuellen Zustand.
Die Menge aller zu O gehörenden *Transitionen* bildet eine Relation $Tr \subset Q \times E \times Q$, wobei Q die Zustandsmenge von O und E eine Menge von *Ereignissen* ist. Für $tr = (q_1, e, q_2)$ bezeichnet man $q_1 = start(tr)$ als *Startzustand*, $q_2 = final(tr)$ als *Zielzustand* und $e = event(tr)$ als *(auslösendes) Ereignis*.
Ein *Transitionssystem* ist ein Tripel (Q, q_0, Tr) mit folgender Eigenschaft:

- $q_0 \in Q$ heißt *Startzustand* (des Transitionssystems)
- $\forall q \in Q\, \exists tr_1, tr_2, \ldots, tr_n \in Tr\, \forall i \in \{1, \ldots, n-1\}$:
  $q_0 = start(tr_1) \wedge final(tr_i) = start(tr_{i+1}) \wedge final(tr_n) = q$

Eine Transition kann durch eine eintreffende Nachricht ausgelöst werden oder Begleiterscheinung einer Nachrichtenaussendung sein, sie kann auch von äußeren Einflüssen unabhängig stattfinden. Ein Transitionssystem kann somit unmittelbar als einfach zusammenhängender, gerichteter Graph dargestellt werden, dessen Knoten die Zustände mit q_0 als Wurzel sind und dessen Kanten aus den Transitionen gewonnen werden. Die Kanten werden durch die Ereignisse der Transitionen markiert. Wird die Menge der auslösenden Ereignisse als *Entgegennahme von Nachrichten* interpretiert, wobei die einzelnen Ereignisse stellvertretend für die jeweilige Nachricht stehen, und legt man eine Menge $F \subset Q$ als Menge der *Endzustände* fest, so erhält man aus dem Transitionssystem einen *nichtdeterministischen endlichen Automaten (NEA)*.

Die in den folgenden Abschnitten beschriebene, interessantere Anwendung ergibt sich, wenn man neben der Entgegennahme von Nachrichten auch die *Aussendung von Nachrichten* oder in Kombination den *Austausch von Nachrichten* als transitionsauslösende Ereignisse auffasst. Die in diesem Falle resultierende Struktur ist eine *Mealy-Maschine* [HU88], wobei Nichtdeterminismus erlaubt sei. Es seien in diesem Zusammenhang folgende Notationen definiert:

- Seien a und A Nachrichtentypen. Mit $q \xrightarrow{a:A} q'$ wird eine Transition beschrieben, deren auslösendes Ereignis die Annahme einer Nachricht des Typs a und die Ausgabe einer Nachricht des Typs A ist. Das Objekt wechselt vom Zustand q in den Zustand q'.
- Eine Transitionsmarkierung a ist gleichbedeutend mit $a : \varepsilon$ oder $a : -$ und bezeichnet ein Ereignis, bei dem nur eine Nachricht des Typs a angenommen wird.
- Eine Transitionsmarkierung $\varepsilon : A$ oder $- : A$ bezeichnet ein Ereignis, bei dem nur eine Nachricht des Typs A ausgegeben wird.

- Eine Transitionsmarkierung ε oder $- : -$ oder eine fehlende Markierung bezeichnet ein extern nicht beobachtbares Ereignis.

Ist eine Eingangsnachricht angegeben, so kann diese Transition erst stattfinden, wenn diese Nachricht eintrifft. Ist jedoch keine Eingangsnachricht vorhanden, dann kann die Transition *jederzeit* stattfinden, sie wird dann auch *spontane Transition* genannt. Spontane Transitionen ohne Ausgaben sind extern nicht beobachtbar und werden auch *innere Transitionen* oder *Epsilon-Transitionen* genannt.

Ist sowohl eine Ein- als auch Ausgangsnachricht angegeben, so gilt die Transition nur als ausführbar, wenn die Eingangsnachricht eingetroffen ist. Diese Sichtweise deckt sich mit den Modellierungen von Milner in [MPW92], wo Eingangskanäle als *blockierend* definiert werden. Die Ausgangsnachricht kann vorher nicht versendet werden. Damit besteht zwischen Ein- und Ausgabe ein *zeitlicher Zusammenhang*. Ein Klient, der mit diesem Objekt kommuniziert, muss sich auf dem Empfang einer Antwort einstellen – jedoch nicht bevor er seine Anfrage abgesendet hat.

Definition 5.14 Stabilität von Zuständen
Sei q ein Zustand eines Transitionssystems. Man nennt q *stabil*, wenn es keine von q ausgehenden Zustandsübergänge gibt, welche eine leere Eingangsnachricht aufweisen. Man nennt q *instabil*, wenn jeder Zustandsübergang, der von q ausgeht, eine leere Eingangsnachricht aufweist. In den übrigen Fällen heißt q *semistabil*.

5.5.2 Methoden und Diensttypen

NIERSTRASZ behandelt in [Nie95] die Ersetzbarkeit von Transitionssystemen, also die Frage, ob an die Stelle eines erwarteten Transitionssystems ein anderes treten kann, sodass ein Klient keinen Unterschied merkt. Allerdings werden nur Eingabeereignisse betrachtet, da vom Objekt vorausgesetzt wird, dass es Anfragen gemäß spezieller *Diensttypspezifikationen* akzeptiert und demgemäß in einer vordefinierten Weise reagiert.

Nierstrasz bezeichnet als *Diensttypausdruck* einen Ausdruck der Form

$$S : \; x : m(A) \rightarrow R,$$

wobei x das Objekt repräsentiert, welches über den *bezeichneten Kanal* x_m ein Argument des Datentyps A annimmt, um ein Ergebnis des Datentyps R zu liefern. Diese bezeichneten Kanäle sind eine Analogie zu den Methoden eines Objekts, wobei die Bezeichnung des Kanals dem Methodennamen entspricht. Ein äquivalenter Ansatz, der seine Verwendung in der Praxis vor allem bei verteilten Objektsystemen findet, ist das Voranstellen des Methodennamens vor die Eingabeargumente.

In einer solchen Situation ist der Diensttyp durch die Bezeichnung des Kanals sowie der Typen der Eingabeparameter bestimmt. Während die Definition von mehreren Methoden gleichen Namens, aber unterschiedlicher Eingabeparameter erlaubt ist

(Überladen von Methoden), ist es bei den bekannten Programmiersprachen nicht möglich, lediglich den Ausgabetyp zu ändern. Nimmt man an, dass diese Diensttypen nicht vom Zustand des Objekts abhängen, so lässt sich in bekannter Weise eine *Schnittstelle* deklarieren, welche alle vom Objekt realisierten Diensttypen anhand der Parametertypen und des Namens des Eingabekanals bestimmt. Ein Verfahren, um zu bestimmen, ob ein Objekt (laut Spezifikation) in einem gegebenen Zustand eine Nachricht akzeptiert und in einer vorbestimmten Weise reagiert, ist demzufolge verhältnismäßig einfach:

1. Man vergleiche die Schnittstellen des aktuellen und erwarteten Objekts in Hinblick auf Gleichheit der Kanalbezeichnungen sowie entsprechender Kontra- und Kovarianzbedingungen der Ein- und Ausgabeparameter. (Das aktuelle Objekt darf mehr Kanäle anbieten als erwartet.)

2. Ist die Schnittstelle des aktuellen Objekts konform zur gesuchten Schnittstelle, dann genügt es zu prüfen, ob im gegenwärtigen Zustand das aktuelle Objekt sämtliche Anfragen akzeptiert, die von ihm in diesem Zustand laut gegebener Spezifikation zu akzeptieren sind. Aufgrund der zuvor gesicherten Konformität sind die Ausgaben des aktuellen Objekts dann auch konform zu den erwarteten Ausgaben.

In der Regel ist der aktuelle Zustand des Objekts nicht bekannt. Tritt während der Kommunikation Nichtdeterminismus auf, ist es meist nicht möglich, anhand der bisherigen Kommunikation auf den aktuellen Zustand zu schließen. An dessen Stelle tritt dann eine *Menge möglicher Zustände.*

5.5.3 Interaktionen

In [Nie95] wird vorausgesetzt, dass es eine eindeutige Beziehung zwischen eingehender Nachricht und dem resultierenden Diensttyp gibt. Dies liegt selbstverständlich nicht vor, wenn alleine der Nachrichtentyp einer den Stellennutzer erreichenden Nachricht betrachtet wird. Diese eindeutige Zuordnung kann dann gelingen, wenn die eingehenden Nachrichten *attribuiert* werden, beispielsweise mit einem Identifikator des Diensttyps (wie dem Methodennamen). In diesem Falle sind die Nachrichtentypen nur dann äquivalent, wenn sie sowohl im Datentyp als auch in ihrem Attribut übereinstimmen. Wird eine solche Attribuierung erzwungen (etwa bei Schnittstellen), so setzt dies ein *geschlossenes System* voraus, in dem es eine Konvention gibt, welche Attribuierungen äquivalente Dienste bezeichnen. Diese Voraussetzung ist, wie schon in 5.1.4 dargelegt, für Agentensysteme nicht sinnvoll. Der häufige Austausch von Code über möglicherweise weit entfernte Teile des Systems hinweg erzwingt einen ständigen Abgleich aller Konventionen, welcher der Idee des dezentralen Systems zuwiderläuft.

Ohne diese Schnittstellenspezifikationen gibt es keine Liste vorhandener Diensttypen. Möchte man nicht zwischen verschiedenen Eingangskanälen unterscheiden, dann

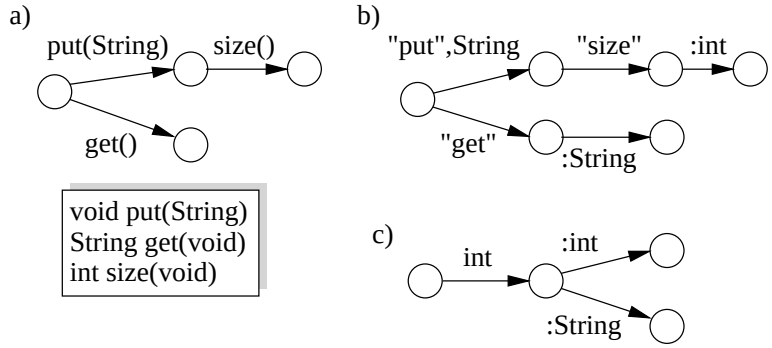

Abbildung 5.3: Explizite und implizite Diensttypen

werden die beteiligten Objekte auf einen Kanal beschränkt und es ist nicht möglich, anhand einer eintreffenden Nachricht unmittelbar die zugehörigen Reaktionen abzuleiten. Die zwischen Agenten ausgetauschten Nachrichten beinhalten per se keine Informationen, welchem Diensttyp sie zugehören.

Unter Verwendung eines Transitionssystems ist es möglich, die Menge der Diensttypen *implizit* festzulegen. Das System beschreibt alle möglichen Reaktionen auf Eingangsnachrichten in Form von ausgesendeten Antwortnachrichten. Vergleicht man zwei solche Beschreibungen, so liegt es nahe, erst die Diensttypen zu bestimmen, welche durch das Transitionssystem impliziert werden und dann diese miteinander zu vergleichen. Abbildung 5.3 zeigt den Unterschied zwischen explizit und implizit definierten Diensttypen.

In a) ist ein Transitionsgraph mit expliziten Diensttypen dargestellt. Hier sind Angaben von Ausgaben in einem solchen Transitionsgraphen nicht notwendig, weil die Schnittstellen miteinander verglichen werden können. Der Transitionsgraph in b) definiert dieselben Diensttypen wie a), aber implizit. Außerdem ist hier die Kanalbezeichnung als Teil der Nachricht modelliert. Der dritte Graph c) demonstriert, dass die implizite Definition größere Freiheiten bietet, indem sie die Bedingung, dass sich zwei Signaturen entweder in Namen oder Eingabeparameter unterscheiden müssen, aufhebt. Hier ist es möglich, dem Objekt eine Zahl zu schicken, worauf es entweder mit einer Zahl oder mit einer Zeichenkette antwortet. Die Auswahl der Ausgabe obliegt dem Objekt selbst.

Die Rollen von Anfrager und Angefragtem können sich im Laufe der Zeit auch umkehren; der zunächst Angefragte könnte seinerseits weitere Daten vom ursprünglichen Anfrager anfordern. Gewissermaßen wird hier eine *Symmetrie* zwischen den beteiligten Komponenten eingeführt: Nicht nur das zuerst angesprochene Objekt muss sich auf vielerlei Arten von Anfragen einstellen, offenbar muss auch der Anfrager mit Antworten verschiedenen Formats umgehen können. Es ist schwierig, in solch einer Situation auseinanderzuhalten, was Anfragen und was Antworten sind; fortan wird daher schlechthin nur noch von *Nachrichten* gesprochen. Aus demselben Grund ist die

Begriff *Dienst* – welcher die Vorstellung der Beantwortung einer Anfrage suggeriert – unpassend.

Definition 5.15 Interaktion
Seien A und B Objekte, welche untereinander mit der Nachrichtenfolge $n_1, n_2, \ldots$ kommunizieren. Eine *(A-)Interaktion* ist eine Teilfolge $n_j, n_{j+1}, \ldots$ der Nachrichtenfolge, wobei nur n_j von A nach B gesendet wird. Wenn es ein minimales $k > j$ gibt, sodass n_k von A nach B gesendet wird, dann ist n_{k-1}die letzte Nachricht der Interaktion, sonst heißt die Interaktion *nichtendend.*

Tritt A mit B in Interaktion, so schickt A eine Nachricht an B, welcher mit einer oder mehreren Nachrichten reagieren kann. Gehen die Nachrichten an A, so können diese die Antwort von B repräsentieren; allerdings können die Nachrichten auch an andere Objekte versendet werden, wenn B beispielsweise externe Informationen benötigt. Diese Nachrichten sind für A nicht sichtbar.

Eine Interaktion beginnt somit mit einer Anfrage, auf die eine möglicherweise zeitlich unbegrenzte Kette von Antworten folgt. A erlangt erst wieder die Kontrolle über die Kommunikation, wenn die Interaktion beendet ist, das heißt alle Nachrichten versendet wurden. Der klassische Fall eines Methodenaufrufs ist ein Sonderfall einer Interaktion: Es wird eine Nachricht angenommen und eine Nachricht als Antwort zurückgegeben. Eine Liste von Eingabeparametern kann im Hybridtypsystem durch die entsprechende Gestaltung der Nachricht aus mehreren Komponenten realisiert werden.

Die Definition eines Protokolls, wie sie in Form des Transitionsgraphen geschieht, ist von zwei wichtigen Anforderungen abhängig, welche im Folgenden für jeden Kontext, in dem das Hybridtypsystem eingesetzt wird, als gegeben vorausgesetzt werden:

1. Die Kommunikationskanäle sind *verlässlich*: Es gehen zwischen Absender und Empfänger keine Nachrichten verloren.

2. Die Kommunikationskanäle sind *ordnungserhaltend*: Eine Folge von Nachrichten, welche von einem Objekt A zu einem Objekt B gesendet werden, erreichen B in derselben Reihenfolge, wie sie von A abgesendet wurden.

Können Sender und Empfänger nicht unmittelbar auf ein verbindungsorientiertes Protokoll wie *TCP* zugreifen (da ihnen der direkte Zugriff auf Sockets verwehrt ist), so ist es Aufgabe der lokalen Infrastruktur, für die Erfüllung dieser zwei Bedingungen zu sorgen.

5.5.4 Konkurrierende Klienten

Allgemein betrachtet können an einem Kommunikationsablauf mehr als zwei Objekte beteiligt sein. Man stelle sich dazu vor, dass Objekt C einen Dienst anbietet, den

mehrere andere Objekte nutzen möchten, etwa *A* und *B*. Wird mit Hilfe eines Transitionssystems ein Kommunikationsprotokoll definiert, so kann es passieren, dass durch die gemeinsame Verwendung der Nachrichtenkanäle das Objekt eine Zustandsänderung durch die Kommunikation mit *B* erfährt, welche *A* nicht bemerkt.

Ein zweites Problem liegt in der Sicht der Klienten auf das gemeinsam verwendete Objekt. *A* kann mit einer derartigen Zustandsänderung rechnen, wenn *B* zeitgleich mit *C* kommuniziert. Jedoch kann es sein, dass *B* andere Funktionalitäten von *C* verwendet, als *A* erwartet. Nierstrasz gibt in [Nie95] ein Beispiel eines Pufferspeichers an, welcher eine Löschfunktion anbietet, die *A* nicht erwartet. Gemäß den Ko- und Kontravarianzbedingungen entspricht *C* zwar den Erwartungen von *A*, aber *B* nutzt diese für *A* verborgene Fähigkeit von *C* aus. Diese Zustandsänderung ist von *A* nicht erwartet, sodass man nicht von einer Subtypeigenschaft sprechen dürfte. Diesem Problem kann von seiten der Implementation entgegengewirkt werden. Das Hybridtypsystem setzt voraus, dass eine solche Störung des Protokolls nicht stattfinden darf:

> Ein Objekt, welches eine bestimmte Spezifikation als Typ anbietet, muss ungeachtet der Menge der Kommunikationspartner diesen Spezifikationen entsprechen.

Damit liegt die Verantwortung beim Implementierer des Objekts sowie demjenigen, welcher die Typspezifikation aufstellt. Kommt es zu derart unerklärbarem Verhalten des Objekts, dann gilt die Typspezifikation als *nicht zutreffend*. In diesem Sinne ist ein Stapelspeicher mit Löschfunktion, welcher jedem Klienten die Modifikation des einen Stapels zulässt, *durch das Hybridtypsystem in Bezug auf den Transitionstyp nicht typisierbar*. Man kann lediglich den syntaktischen und semantischen Typ verwenden, um die Eigenschaften dieses Objekts zu charakterisieren.

5.5.5 Interaktionen und Autonomie

In der Regel findet man in aktuellen Programmiersprachen kein Konzept selbstmodifizierenden Codes. Das Verhalten des Agenten sollte demnach durch die Implementierung vollständig determiniert sein. Es gibt allerdings andere Einflüsse, welche dem Objekt *A* unbekannt sind:

- Objekt *B* kommuniziert mit einem dritten Objekt.
- Der Zustand von *B* hängt von anderen Faktoren als dem Nachrichtenaustausch ab.

Im letzten Abschnitt wurde eine weitere Möglichkeit angedeutet, nämlich dass ein konkurrierender Klient das Protokoll stört. Diese Einflussnahme sei jedoch durch die entspechende Implementierung von Objekt *B* ausgeschlossen.

Alleine durch die Anwesenheit eines bestimmten anderen Agenten könnte *B*s Verhalten beeinflusst werden, ohne dass irgendwelche Nachrichten ausgetauscht würden. Einerseits können diese Vorgänge nicht ignoriert werden, da jeder äußere Einfluss eine Zustandsänderung bewirken kann. Andererseits sind solcherlei Einflüsse nicht nur schwierig zu formalisieren, sondern in den meisten Fällen nicht einmal von besonderem Interesse, da sie nicht direkt mit dem Nachrichtenaustausch in Verbindung stehen.

Das Hybridtypsystem repräsentiert solche verdeckten Einflüsse durch *Nichtdeterminismus*. Es ist für Objekt *A* in solchen Fällen nicht möglich zu erkennen, welchem Pfad *B* auf seine Anfrage folgen wird. Dieses Verhalten von *B* lässt sich als *autonome Entscheidung des Agenten* interpretieren und unterstützt somit in einer bedeutenden Weise das zu Grunde liegende Agentenbild.

Für den Vergleich solcher Transitionssysteme gilt, dass ein solcher Nichtdeterminismus *erwartet* werden muss; es sei denn, für den Kommunikationspartner ergibt sich durch die nichtdeterministische Auswahl eines Pfades kein Unterschied zu den erwarteten Pfaden. Diese Regel wird in den späteren Abschnitten präzisiert.

5.5.6 Spezielle Kommunikationsabläufe

Im allgemeinen Fall lässt sich *nicht* jede Kommunikation als Folge von Interaktionen modellieren, wie es in den vergangenen Abschnitten dargelegt wurde. Diesen Erläuterungen zufolge müsste eine Interaktion in einem stabilen Zustand beginnen und enden, wobei im Laufe der Interaktion nur instabile Zustände durchlaufen werden. Folgende Situationen müssen gesondert behandelt werden:

***B* beginnt zu senden:** Ein Beispiel hierfür ist ein Dienst, welcher nach seinem Start *spontan* Daten liefert. Interaktionen erfordern stets eine Eingangsnachricht zu Beginn. Um spontane Transitionen mit Ausgabe zu Beginn zu behandeln, nimmt das Typsystem einen *impliziten Vorzustand* an. Der Übergang in den eigentlichen Anfangszustand wird durch eine *Pseudonachricht* an *B* bewirkt, welche als Aktivierung von *B* verstanden werden kann.

***B* sendet fortwährend eine Folge von Nachrichten (Schleife):** Schließt sich die Kette von Ausgaben zu einer Schleife, so bedeutet dies, dass der Agent möglicherweise niemals einen stabilen Zustand erreichen wird. Hier liegt nach Definition eine *nichtendende Interaktion* vor. Nichtendende Interaktionen lassen sich wie jede andere Interaktion in Form eines Ausschnitts des gesamten Transitionssystems repräsentieren. Ein solcher Teilgraph lässt sich isomorph auf einen endlichen Automaten abbilden, welcher in äquivalenter Weise durch einen regulären Ausdruck beschrieben wird [HU88]. Mehrere verschiedene Ausgabeschleifen werden so behandelt, als ob jede von ihnen zur Menge der möglichen Ausgaben beiträgt; der Teilgraph umfasst also alle diese Schleifen.

***B* durchläuft semistabile Zustände:** Gerät das Objekt in einen semistabilen Zustand, kann es in diesem Zustand verharren, bis es eine Eingabe entgegennehmen kann. Es kann aber auch mit der Ausgabe fortfahren und wird dann meist in einen anderen (semi-)stabilen Zustand geraten. Jeder semistabile Zustand kann durch einen instabilen und einen stabilen Zustand ersetzt werden, wobei die Transitionen mit leerer Eingabe von dem neuen instabilen Zustand ausgehen und eine innere Transition zum neuen stabilen Zustand führt. Von diesem gehen die Transitionen mit nichtleerer Eingabe aus. Semistabile Zustände erlauben eine kompakte Darstellung der Typbeschreibung; sie führen allerdings durch den Nichtdeterminismus zu deutlich höherem Aufwand bei Vergleichen als stabile Zustände.

***A* oder *B* senden nie:** Betrachtet man den Fall, dass *A* nie sendet, so liegt der schon besprochene Fall einer spontanen Ausgabe mit eventueller Schleife vor. In allen übrigen Fällen wird es einen Zustand geben, in dem *B* eine Nachricht von *A* entgegennehmen kann. Erhält *A* auf eine Anfrage keine Nachrichten von *B*, dann könnte tatsächlich *B* auf Ausgaben verzichtet oder die Ausgaben an andere Objekte geleitet haben. Sollte *B* keine weiteren Anfragen von *A* mehr akzeptieren, dann befindet sich *B* aus der Sicht von *A* in einem *Endzustand*. Anderenfalls liegt ein neuer stabiler Zustand vor, der den Anfangspunkt der nächsten Interaktion markiert.

Diese Betrachtungen zeigen, dass die Gliederung einer beliebigen Kommunikation in eine Abfolge von Interaktionen – auf welcher der Transitionstyp basiert – ein sinnvolles Modell ist. Zusammenfassend gilt:

Folgerung 5.16 Interaktionseigenschaften

1. Jede Interaktion besitzt genau eine Eingangsnachricht. Diese kommt jeder Ausgabe zuvor. Die Übermittlung der Nachricht wird durch genau einen Zustandsübergang begleitet.
2. Eine Interaktion ist entweder nichtendend oder endet in einem stabilen Zustand.
3. Die von einer Interaktion bewirkte Ausgabe lässt sich durch einen regulären Ausdruck oder alternativ durch einen endlichen Automaten darstellen, welcher injektiv auf den Ursprungsgraphen abgebildet werden kann.

Das in [Nie95] demonstrierte Verfahren, welches die Ersetzbarkeit von Dienstanfragefolgen feststellt, wird nun so verallgemeinert, dass an die Stelle der Diensttypen die Interaktionstypen treten. Da sich Interaktionen bisweilen nur an ihren Ausgaben unterscheiden, ist die Prüfung der Eingangsnachrichten nicht hinreichend. Die beim Transitionstypvergleich anstehende Aufgabe ist zunächst, die von einem Zustand ausgehenden Interaktionen zu entdecken und diese mit den möglichen Entsprechungen im

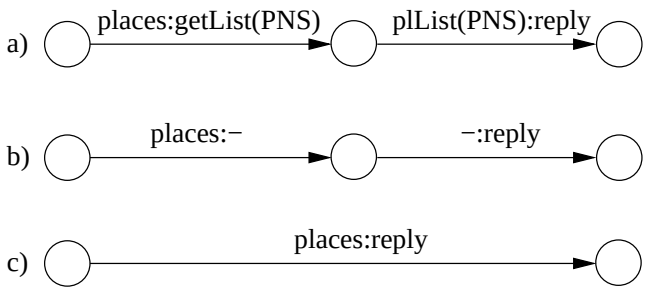

Abbildung 5.4: Sender- und Empfängerannotationen

anderen Typ zu vergleichen. Dies ist effektiv möglich, denn die Zahl der möglichen Interaktionen ist in jeder Typbeschreibung endlich.

5.5.7 Sender und Empfänger

In traditionellen Schnittstellenbeschreibungen wird implizit von einer Kommunikation ausgegangen, welche sich zwischen zwei Parteien ereignet; üblicherweise existiert ein Anfrager oder *Klient* sowie ein Angefragter oder *Server*. Eingehende Nachrichten verlaufen dabei stets vom Klienten zum Server, während ausgehende Nachrichten vom Server zum Klienten zurücklaufen. Das Hybridtypsystem ist jedoch im Hinblick auf die Symmetrie der Agentenkommunikation entworfen worden und bezieht auch die Kommunikation zwischen mehreren Objekten ein. Um dies zu beachten, sind so genannte *Senderannotationen* und *Empfängerannotationen* vorhanden. Diese Angaben befinden sich in den Transitionen der Typspezifikation.

Ein Objekt, das sich als einem Typ zugehörig betrachtet, geht davon aus, dass es Nachrichten von einem Kommunikationspartner erhält, welchem es als kompatibel vermittelt wurde. Diesem Partner gegenüber ist das Objekt verpflichtet, das in der Spezifikation definierte Verhalten zu realisieren. Eingangsnachrichten ohne Senderannotation werden als von diesem Partner stammend angenommen, während Ausgangsnachrichten als an diesen Partner gehend verstanden werden. Treten weitere Kommunikationspartner auf, so können diese auch mit diesem Objekt in Verbindung treten, dürfen die Kommunikation aber nicht stören.

Wird eine Senderannotation angegeben, so bedeutet dies, dass diese Nachricht *nicht* vom aktuellen Kommunikationspartner stammt. Abbildung 5.4 gibt ein Beispiel einer Sender- und einer Empfängerannotation. Die unter a) dargestellte Interaktion beschreibt, dass nach einer Anfrage des Kommunikationspartners ein anderes Objekt (hier als „PNS“ annotiert) angesprochen wird, das eine bestimmte Nachricht erhält. Dieses Objekt sendet seinerseits ein Ergebnis, welches als Antwort an den Anfrager geliefert wird. Für den Anfrager stellt sich jedoch der Vorgang wie unter b) und damit äquivalent wie unter c) dar. Die Tatsache, dass das angesprochene Objekt mit jemand anderem kommuniziert, bleibt ihm verborgen.

Subtyp\Supertyp	ohne	Annotation	ANY	OTHER	PLACE
ohne	ja	nein	ja	nein	nein
Annotation	nein	$\prec$	ja	ja	nein
ANY	nein	nein	ja	nein	nein
OTHER	nein	nein	ja	ja	nein
PLACE	nein	nein	nein	nein	ja

Tabelle 5.1: Vergleich von Sender- und Empfängerannotationen

Folgerung 5.17 Interaktion mit dritter Seite
Interaktionen mit Sender- und Empfängerannotationen, welche eine Menge von Entitäten beschreiben, welcher der aktuelle Kommunikationspartner nicht angehört, sind Spezialisierungen von Interaktionen mit leeren Nachrichten (Epsilon-Nachrichten).

Die Beschreibung a) ist somit konform zur Beschreibung c). Der umgekehrte Fall gilt nicht: Wenn man erwartet, dass das Objekt eine Kommunikation mit dritter Seite durchführt, dann ist es nicht ausreichend, wenn das Objekt nur eine interne Zustandsänderung angibt.

Bei einem Agentensystem wie AMETAS ist es den Teilnehmern möglich, anhand des Nachrichtenkopfes festzustellen, wer der Absender der Nachricht war. Es bietet sich an, Sender- und Empfängerannotationen den Nachrichten und nicht den Transitionen zuzuordnen, was prinzipiell möglich wäre. Die Angabe des Senders und Empfängers kann also als *Annotation* der Nachricht verstanden werden. Ist bereits eine *inhaltliche* Annotation vorhanden, so müssen beide Annotationen in den zu vergleichenden Typen in Deckung gebracht werden. Die formale Repräsentation einer Nachricht ändert sich demnach durch die Hinzunahme einer Sender- oder Empfängerannotation nicht.

Es gibt spezielle vordefinierte Annotationen für Sender und Empfänger. Tabelle 5.1 zeigt, welche Beziehungen zwischen erwarteter Annotation und tatsächlicher Annotation bestehen müssen, damit eine Übereinstimmung festgestellt wird.[5]

Ist der Absender im tatsächlichen Typ so annotiert wie erwartet, so sind die Annotationen verträglich. Falls keine der vordefinierten Annotationen verwendet wird, sind die Annotationen entsprechend auf Konformität zu prüfen. Die Annotation *PLACE* bezeichnet die aktuelle Stelle als Absender oder Empfänger; diese Annotation ist so speziell, dass es keine Verträglichkeit mit anderen Annotationen gibt. *OTHER* bezeichnet einen anderen Sender/Empfänger als den aktuellen; daher wird *OTHER* nicht durch die leere Annotation spezialisiert. *ANY* unterscheidet sich in dieser Hinsicht von *OTHER* und erlaubt auch, dass die Nachricht vom aktuellen Kommunikationspartner stammt.

Wird beispielsweise ein Objekt gesucht, das eine Nachricht von einem Objekt mit

[5] Die Bezeichnungen *Super-/Subtyp* beziehen sich auf Nachrichten. Im Falle von empfangenen Nachrichten ist zu beachten, dass aufgrund der Kontravarianzbedingung die aktuelle Eingangsnachricht ein *Supertyp* der erwarteten Nachricht ist.

der Annotation α entgegennimmt und das aktuelle Objekt gibt an, Nachrichten von irgendeinem Objekt zu akzeptieren (*ANY*), so ist das aktuelle Objekt als Subtyp verwendbar, denn Eingaben verhalten sich kontravariant. Das Objekt könnte intern die einzelnen Sender auswerten.

Interessant ist, dass nach einer Migration keine weiteren Interaktionen mit dem bisherigen Kommunikationspartner möglich sind (außer wenn der Agent unmittelbar zur Stelle zurückkehrt). Die Kommunikation kann also nur mit anderen Entitäten stattfinden, was eigentlich die Markierung durch Sender- und Empfängerannotationen erfordert. Dies kann jedoch durch die Syntax der Typbeschreibung nicht erzwungen werden. Es gibt keine Hinweise darauf, welche Stelle tatsächlich erreicht wird oder wann der Agent zurückkehrt. Die korrekte Markierung obliegt somit alleine dem Ersteller der Typbeschreibung; semantisch unplausible Formulierungen werden nicht entdeckt.

5.5.8 Initialisierung

Eine übliche Vorgehensweise bei der Verwendung von Agenten ist, ihnen nach dem Start eine Initialisierungsnachricht zuzuschicken, bevor sie vollständig einsatzfähig sind. Dies ist insbesondere notwendig, wenn das Agentensystem (wie im Falle von AMETAS) die Initialisierung von Agenten mittels spezieller Konstruktoren nicht zulässt. Im weiteren Ablauf akzeptieren sie diese Initialisierungsnachricht nicht mehr. Wenn der Typ den Agenten in Bezug auf die verfügbaren Interaktionen charakterisiert und ein Typ sich während der Existenz der Instanz nicht ändern darf (siehe Abschnitt 5.1.4), muss man streng genommen von einem *Typwechsel* reden.

Die Beibehaltung der ursprünglichen Typbeschreibung ist nicht möglich, da diese eine Nachricht zur Initialisierung fordert, welche nach dieser Initialisierung nicht mehr akzeptiert wird. Ein Versuch, die Korrektheit der Beschreibung zu retten, könnte darin bestehen, den Typ des Agenten zu *ersetzen*, sodass andere Interessenten den aktuellen Typ erhalten, welcher das Akzeptieren der Initialisierungsnachricht nicht beinhaltet. Alternativ könnte auf die Zuweisung eines Typs verzichtet werden, solange der Agent seinen *Betriebszustand* nicht erreicht hat.

Eine Lösung, welche die Ersetzung des Typs vermeidet, ist die Verwendung von Sender- und Empfängerannotationen. Die Initialisierungsnachricht könnte man als von einem *bestimmten Sender* ausgehend annotieren. Für jeden anderen Interessenten stellt sich dieser Vorgang dann als Epsilon-Übergang dar.

Wichtig ist zu bemerken, dass ein Initialisierungsvorgang weitaus komplexer sein kann, als es von einer Instantiierung mit einem parametrisierten Konstruktor bekannt ist. Der Transitionstyp kann dies darstellen, da er das starre Anfrage-Antwort-Schema von Methodenaufrufen ablöst.

5.6 Vergleich von Interaktionstypen

Wie es den Nachrichten entsprechende Nachrichtentypen gibt, so werden auch Interaktionstypen definiert:

Definition 5.18 Interaktionstyp
Ein Interaktionstyp r ist ein Paar der Form (n, A) mit

- $n = in(r)$ ist der *Eingangsnachrichtentyp.*
- $A = out(r)$ ist der *Ausgangsnachrichtenketten-Typ.*

Die Menge aller von einem Zustand q ausgehenden Interaktionstypen wird mit $int(q)$ bezeichnet. Der Zielzustand jeder Interaktion des Typs ist $fin(r)$; er wird für nichtendende Interaktionen als ∞ markiert.

Der Typ einer Ausgangsnachrichtenkette kann durch einen nichtdeterministischen endlichen Automaten repräsentiert werden, welcher als Kantenmarkierungen Nachrichtentypen trägt. Er beschreibt die Menge der auf eine Nachricht des Typs n möglichen folgenden Nachrichtenketten.

5.6.1 Interaktionstypkonformität

Die Prüfung der Konformität der Eingangs- und Ausgangsnachrichten schließt die Prüfung der Verträglichkeit der Annotationen der Nachrichten, der Absender und Empfänger sowie beteiligter Zustände ein. Während der erste Test sich auf die einfache Konformitätsprüfung der Nachrichten beschränkt, ist der Algorithmus zur Prüfung der Ausgaben komplizierter. Laut Definition beinhalten Interaktionen im Allgemeinen eine Kette von Ausgangsnachrichten. Eine Prüfung solcher Ketten läuft prinzipiell darauf hinaus, jede Kette gliedweise auf Kovarianz der Nachrichten zu vergleichen.

Um die Ausgaben zu vergleichen, muss man die den Ketten zugehörigen NEAs vergleichen. Die Kantenmarkierungen werden dabei als Ausgabe interpretiert. Da das Objekt, welches dieses Transitionssystem implementiert, selbst über die Alternativen bestimmt, ist jede Ausgabe als Reaktion auf die Eingabe möglich. Dies trifft *nicht* für Eingaben zu, wie später erläutert wird. Dementsprechend muss das „Enthaltensein“ der durch den Teilgraphen repräsentierten „Ausgabesprache“ im Graphen des Typs geprüft werden.

Definition 5.19 Ausgabeersetzbarkeit
Ein Interaktionstyp s ist *ausgabeersetzbar* durch einen Interaktionstyp r ($out(r) \prec_o out(s)$) genau dann, wenn gilt:

$$\forall n \in \mathbf{N}\ \forall w = a_1 \ldots a_n \in out(s)\ \exists w' = a'_1 \ldots a'_n \in out(r) :\ a'_i \prec a_i\ \forall i = 1, \ldots, n$$

mit Nachrichtentypen a_i und a'_i des Typs beziehungsweise Subtyps.

Die Definition der Ersetzbarkeit fordert gleiche Längen der Ausgabeketten. Dies ist eine wichtige Entscheidung im Entwurf des Typsystems, aber nicht notwendigerweise so zu treffen. Es könnte auch festgelegt werden, dass Präfixe der Ausgabe von dieser Interaktion beinhaltet werden; letztlich ist es unsicher, wann eine Ausgabe folgt oder ob sie jemals folgt. Präfixe zu erlauben führt jedoch in letzter Konsequenz dazu, auch die leere Nachricht zu akzeptieren, obwohl der Typ eine Ausgabe vorsah. Das Prinzip, demzufolge sich der Subtyp in jeder Situation wie der Typ verhalten soll, liefert hierzu keine Aussage.

Zum Vergleich der *regulären Sprachen* der beiden Transitionssysteme greift man auf den von Aho, Hopcroft und Ullman vorgeschlagenen Algorithmus [AHU74] zurück, welcher von Nierstrasz für NEAs in [Nie95] erweitert wurde. Im Unterschied zu den genannten Vorlagen ist bei diesem Vergleich zu beachten, dass man nicht von einem *Enthaltensein* der einen Sprache in der anderen sprechen kann, da eine Kovarianz der Nachrichten vorliegen kann.

Unter Beachtung der Ko- und Kontravarianzbedingungen lässt sich nun die Konformität zweier Interaktionstypen formulieren:

Definition 5.20 Interaktionstypkonformität
Seien r und s zwei Interaktionstypen. Man nennt r einen Subtyp von s, notiert $r \prec s$, genau dann, wenn gilt:

1. $in(s) \prec in(r)$: Die Eingangsnachrichten verhalten sich kontravariant.
2. $out(r) \prec_o out(s)$: Die Ausgangsnachrichtenketten verhalten sich kovariant.

Ein Interaktionstyp r ist konform zu einem Interaktionstyp s genau dann, wenn jede Interaktion des Interaktionstyps r an die Stelle einer Interaktion des Interaktionstyps s treten kann, ohne dass ein Unterschied durch den aktuellen Kommunikationspartner erkannt werden kann. Man beachte jedoch, dass Seiteneffekte in Form einer Kommunikation mit dritter Seite entstehen können; diese ist jedoch für den aktuellen Kommunikationspartner transparent. Im Folgenden wird der Kürze des Begriffes halber von *Interaktion* gesprochen, auch wenn präzise ausgedrückt der *Interaktionstyp* gemeint ist.

5.6.2 Zustandsmengen

Ein wichtiges Hilfsmittel zum Vergleich der beiden Transitionssysteme ist die Idee der *äquivalenten Zustände*. Zwei Zustände sind genau dann äquivalent, wenn das Verhalten des Transitionssystems sich nicht ändert, wenn man diese Zustände untereinander vertauscht. Diese Idee lässt sich unmittelbar auf Zustandsmengen ausdehnen. Dadurch wird es möglich, zwei NEAs miteinander zu vergleichen, indem man Zustände durch Zustandsmengen ersetzt und damit den NEA implizit in einen deterministischen endlichen Automaten überführt.

Definition 5.21 Ausgabeersetzbarkeit von Zustandsmengen
Seien X und Y Mengen von Zuständen, die während der Interaktionen r und s eingenommen werden. Unter $out(X)$ versteht man die Menge aller Nachrichten, welche durch eine Transition von einem beliebigen Zustand in X abgegeben werden. Es ist $X \prec_o Y$ genau dann, wenn gilt:

1. $\forall a \in out(X)\ \exists b \in out(Y) : a \prec b$
2. $\forall a \in out(X), X \xrightarrow{[a]^o_X} X', Y \xrightarrow{[a]^o_Y} Y' : X' \prec_o Y'.$

Dabei sind $[a]^o_X$ alle Nachrichten, die von Zuständen in X ausgehen und Supertypen der Nachricht a sind. Es ist bei der Ausgabe von a nicht zu erkennen, von welcher Transition die Nachricht a tatsächlich stammt. Analog sind $[a]^o_Y$ alle Ausgaben in Y, welche Supertypen von a sind. Der Übergang $X \xrightarrow{M} X'$ ist so zu verstehen, dass X' die Vereinigung aller Mengen ist, die durch einen Übergang durch ein Element $m \in M$ repräsentiert werden.

Die Zustandsmenge X kann also genau dann Y ersetzen (in Bezug auf die Ausgabe), wenn in X keine Ausgabe vorgenommen wird, die von irgendwelchen Zuständen in Y nicht zu erwarten ist und wenn die resultierenden Zustandmengen nach jeder möglichen Ausgabe wieder ausgabeersetzbar sind. Durch die Klassenbildung $[a]$ und die Sammlung aller Zielzustände in den Mengen X' und Y' hängen die Mengen X und Y direkt von der bisher ausgegebenen Nachrichtenkette ab. Man kann daher eine Zuordnung $st_r : w \mapsto X$ und $st_s : w \mapsto Y$ definieren.

Satz 5.22 Ausgabeersetzbarkeit von Interaktionen
Sei r eine Interaktion des Typs A und s eine Interaktion des Typs B. Die Startzustände der Interaktionen seien mit q^r_0 sowie q^s_0 bezeichnet. Die Interaktion s ist ausgabeersetzbar durch r genau dann, wenn $\{q^r_0\} \prec_o \{q^s_0\}$.

Beweis. „⇒“: Die Ausgaben beginnen in den jeweiligen Anfangszuständen q^r_0 und q^s_0 und enden beide in den jeweiligen (eindeutigen) Zielzuständen der Interaktionen. Nach m Rekursionsschritten und der Ausgabe $a_1 a_2 \ldots a_m$ möge die Bedingung 1 nicht mehr

wahr sein: r befinde sich in einem Zustand $x \in st_r(a_1 \dots a_m)$ und in $out(x)$ gebe es eine Nachricht c, zu welcher es keine Entsprechung in $out(st_s(a_1 \dots a_m))$ gebe. Dann kann $v_1 = a_1 a_2 \dots a_m c w$ eine Ausgabe von r sein, aber nicht von s, weil die Komponente an der Stelle $m+1$ keine Entsprechung hat. Es kann also keine Ausgabeersetzbarkeit nach Definition 5.19 vorliegen.

„$\Leftarrow$“: Sei $w = a_1 a_2 \dots a_n c$ eine Folge von Ausgaben, bei der die ersten n Komponenten sowohl von r als auch von s erwartet werden können; c jedoch sei eine Ausgabe von r, die in s nach diesen ersten n Nachrichten nicht auftauchen kann. Sei $c \in out(x_c)$, dann ist $x_c \in st_r(a_1 \dots a_n)$ und somit $c \in out(st_r(a_1 \dots a_n))$. Mit der Voraussetzung, dass die ersten n Ausgaben auch von s zu erwarten sind, befindet sich s nach diesen Ausgaben in einem Zustand $y \in st_s(a_1 \dots a_n)$, von denen *keiner* die Ausgabe c erlaubt (oder eine dazu konforme Ausgabe). Die Bedingung 1 ist dann für $X = st_r(a_1 \dots a_n)$ und $Y = st_s(a_1 \dots a_n)$ nicht mehr gegeben, sodass $X \not\prec Y$ gilt und folglich auch $\{q_0^r\} \not\prec \{q_0^s\}$. ■

Die Konformität der Nachrichten impliziert die Konformität eventuell vorhandener Annotationen der Nachrichten, ihrer Absender und Empfänger. Dabei müssen die Annotationen der Absender im Subtyp nach Kontravarianzregeln stets allgemeiner sein als vergleichbare im Typen; umgekehrt müssen Empfängerannotationen im Subtyp spezieller sein als im Typ (Kovarianz).

Dieser Satz ermöglicht zusammen mit Definition 5.21 eine einfache (hier nicht dargestellte) Implementierung eines Algorithmus, der zwei gegebene Interaktionen an ihren Ausgaben vergleicht. Ist die Auswertung von Zustandsannotationen erwünscht, dann werden diese wie zusätzliche Ausgaben behandelt, die untrennbar mit der vorangegangenen Ausgabe verbunden sind. Zwei Ausgaben, die von ihrem Typ also gleich sind, werden durch die Anfügung der Zustandsannotation unterscheidbar.[6]

5.6.3 Verträgliche Transitionssysteme

Die Interaktion r kann genau dann s ersetzen, wenn die Eingabe in r ein Supertyp der Eingabe in s ist und wenn die Ausgaben von r Subtypen möglicher Ausgaben von s sind. Aber auch wenn klar ist, wie Interaktionen zu vergleichen sind, ist noch nicht auf die Verträglichkeit der Transitionssysteme zu schließen. Insbesondere kann der Nichtdeterminismus zu Irrtümern beim Vergleich der Transitionssysteme führen: Wenn zwei Pfade von einem Zustand ausgehen, längs welcher eine gegebene Interaktion stattfinden kann, dann kann das Verfolgen des falschen Pfads zu einem Zustand führen, der mit dem entsprechenden Zustand am Ende der erwarteten Interaktion nicht vergleichbar ist.

[6] Hierbei wird gewöhnlich keine *reelle* Unterscheidung stattfinden: Die Instanz schickt die erwartete Nachricht zum erwarteten Zeitpunkt. Dennoch wird ihr die Subtypeigenschaft abgesprochen, wenn der Folgezustand eine andere Annotation aufweist. Es muss dem Anwender überlassen werden, ob er dies als tolerabel erachtet oder auf Zustandsannotationen verzichtet.

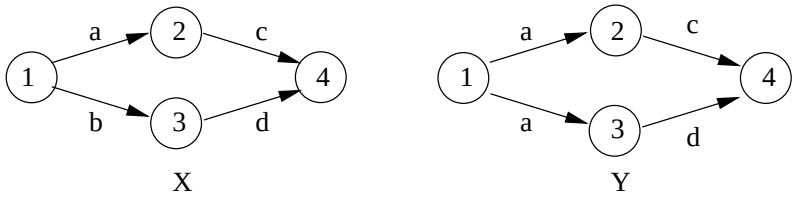

Abbildung 5.5: Nichtdeterministische Alternativen

Abbildung 5.5 zeigt zwei Transitionssysteme X und Y. Die von Y dargestellten Interaktionen im Zustand 1 besitzen beide als Eingangsnachricht eine Nachricht vom Typ *a*. Das Transitionssystem X weist ebenfalls zwei Transitionen am Startzustand auf, wobei aber nur eine Transition die Eingangsnachricht *a* beinhaltet. PUNTIGAM verfolgt in seinen Arbeiten [Pun96, Pun97] die auch von Nierstrasz verwendete *Trace-Semantik*: Der Subtyp muss auf jeden Fall alle Pfade realisieren, welche der Typ realisiert. Demzufolge ist der Typ X aus Abbildung 5.5 nicht konform zum Typ Y, denn eine der möglichen Spuren in Y ist *ad*, welche nicht in X auftritt.

Die Trace-Semantik ist eine von mehreren möglichen Interpretationsarten in Bezug auf Transitionssysteme. Gibt es eine Auswahl verschiedener Transitionen mit gleicher Nachricht an einem Zustand, so wird „automatisch“ jene Transition verfolgt, die zum Akzeptieren der präsentierten Kette führen wird. Man spricht hier von *externem Nichtdeterminismus*, da das Transitionssystem keine Kompetenz hat, eine Auswahl aus einer Menge möglicher Transitionen selbständig zu treffen. Im Gegensatz hierzu ist ein *interner Nichtdeterminismus* gegeben, wenn das Transitionssystem selbständig entscheidet, welchem Pfad es im Transitionssystem folgt. Gerade diese Deutung liegt nahe, wenn mit Hilfe von Transitionsgraphen ein Verhalten eines autonomen Agenten modelliert werden soll.

Das Transitionssystem Y ist so aufgebaut, dass es sowohl den Pfad *ac* als auch den Pfad *ad* akzeptieren könnte. Das System X hingegen akzeptiert *ac* und *bd*, nicht aber *ad*. Nach DE NICOLA und HENNESSY [dH84] liegt Prozessäquivalenz vor, wenn es keinen *Beobachter* gibt, welcher mittels Experimenten – welche Nachrichten sind, die der Beobachter den System zuleitet – einen Unterschied zwischen den beiden Systemen feststellen kann. Wenn anstelle von Y das System X zum Einsatz kommt, wäre nicht nachweisbar, dass die Spur *ad* tatsächlich fehlt, denn es gibt bei diesen Graphen keine Aussage über eine mögliche *Wahrscheinlichkeit* der Auswahl einer der Alternativen.

Folgerung 5.23 Verlust von Pfaden
Die Tatsache, dass ein Transitionssystem Y durch ein Transitionssystem X ersetzt werden kann, sodass alle Experimente zum gleichen Ergebnis führen, impliziert nicht, dass X sämtliche Pfade von Y realisiert.

Diese Folgerung hat weit reichende Konsequenzen: Eine Formalisierung der Ersetzbarkeit darf sich *nicht* an bestehenden Interaktionsketten ausrichten. Eine sich im-

plizit an der Trace-Semantik orientierende Formalisierung würde verlangen, dass jede Interaktionskette, welche von einer Zustandsmenge B des Typs ausgeht und zu einer nichtleeren Zielzustands-Menge führt, von einer Interaktionskette des Subtyps ersetzbar wäre, welche von der Zustandsmenge A des Subtyps zu einer nichtleeren Zustandsmenge (unter Beachtung von Ko- und Kontravarianz der Nachrichten) führt. Dies missachtet die Freiheit der Instanz, an bestimmten Punkten der Kommunikation zwischen Alternativen wählen zu können.

5.6.4 Beobachtbare Ersetzbarkeit von Zustandsmengen

Die beobachtbare Ersetzbarkeit muss *schrittweise* definiert werden, da sie sich nicht anhand akzeptierter Nachrichtenketten offenbart, sondern bei jedem einzelnen Kommunikationsschritt. Wenn man von der Ersetzbarkeit eines Transitionssystems spricht – und in der Folge dann von der Ersetzbarkeit des Typs durch den Subtyp –, dann steht folgende intuitive Vorstellung im Hintergrund:

> Man kommuniziert mit der Instanz eines Subtyps, indem man ihr eine Nachricht zusendet und eventuell eine Folge von Nachrichten empfängt. Eine Instanz eines Typs kann mehrere Alternativen haben, eine gegebene Interaktionskette abzuarbeiten. Eine Instanz eines Subtyps muss sich nach Abarbeiten dieser Kette in einem Zustand befinden, der dem Zielzustand einer dieser Alternativen des Typs entspricht. In einem solchen Zielzustand muss die aktuelle Instanz alle Eingangsnachrichten akzeptieren, die die Typinstanz akzeptieren würde und dabei mit den zugehörigen Ausgaben antworten, die die Typinstanz ausgeben würde, wenn sie die gegebene Eingangsnachricht empfangen hätte.

Daraus folgt, dass, wenn die aktuelle Instanz zu einem Zeitpunkt eine Nachricht *nicht* akzeptiert oder auf eine Nachricht *unerwartete* Ausgaben produziert, es keine Alternative im Typen geben darf, die nach Abarbeiten der gegebenen Interaktionskette diese Nachricht akzeptieren und die erwartete Antwort liefern würde. Die intuitive Vorstellung der beobachtbaren Ersetzbarkeit lässt sich in folgender Definition formalisieren:

Definition 5.24 Beobachtbare Ersetzbarkeit von Transitionssystemen
Seien die Transitionssysteme A und B gegeben. Sei w eine Kette von Interaktionen r_i mit A, sodass $w = r_1 r_2 \ldots r_n$ und q_w der aktuelle Zustand von A nach Abschluss der Interaktionskette w ist. Ferner gelte $q \xrightarrow{\varepsilon} q$. B ist *beobachtbar ersetzbar* durch A genau dann, wenn für jede Interaktionskette w folgende Implikation gilt:

$$\exists w' = r'_1 r'_2 \ldots r'_n, q_0^B \xrightarrow{w'} Q_{w'} \neq \emptyset, r_i \prec r'_i \forall i = 1, \ldots, n \Longrightarrow$$
$$\exists q' \in Q_{w'} \forall s' \in int(q') \exists s \in int(q_w) \exists s'' \in int(q') : in(s') \prec in(s) \wedge s \prec s''.$$

Das Epsilon in der Definition bezeichnet nicht die leere Nachricht, sondern die Interaktionskette ohne Elemente. Auf diese Weise wird der Startzustand der Transitionssysteme eingebunden. Wird eine Interaktion nicht akzeptiert, dann resultiert als Zustandsmenge die leere Menge. Dass es überhaupt Interaktionsketten gibt, sichert die Definition durch Ausnutzen der leeren Interaktionskette: In diesem Falle besteht w' auch nur aus dem leeren Wort, $Q_{w'}$ beinhaltet daher nur den Startzustand von B, nämlich q_0^B. Dann muss es für jede Interaktion im Startzustand q_0^B laut dieser Definition eine Interaktion im Startzustand q_0^A geben, deren Eingangsnachricht ein Supertyp ist *und* es muss im Gegenzug eine Interaktion im Typ geben, die ihrerseits ein Supertyp der Subtypinteraktion ist. Dadurch wird zugesichert, dass nur eine im Typ definierte nichtdeterministische Alternative ausgewählt wird. Von dieser ersten Interaktion ausgehend bewegt man sich sukzessiv durch die Zustände von Subtyp- und Typinstanz.

Die existentielle Quantifizierung für w' trägt der Tatsache Rechnung, dass es sehr wohl sein kann, dass eine Interaktion, die der Typ in einem Zustand vorsieht, den man nach Abarbeitung der Interaktionskette erreicht hat, vom Subtyp nicht akzeptiert wird. Dies alleine ist aber kein Grund, den Subtyp abzulehnen; vielmehr kann im Laufe der Abarbeitung ein anderer Pfad durch den Graphen beschritten worden sein. Wichtig ist, dass es einen solchen Pfad im Typ *überhaupt* gibt.

Weniger explizit als in Nierstrasz' Ansatz sind die so genannten *Fehlschläge* durch diese Definition beachtet. Eine Interaktionskette w kann die Subtypinstanz in einen bestimmten Zustand bringen, in dem eine neue, gegebene Eingangsnachricht a nicht akzeptiert wird. Wenn nun die beobachtbare Ersetzbarkeit gelten soll, dann muss es für den Typ eine Möglichkeit geben, diese Nachricht a abzulehnen, wenn er vorher w abgearbeitet hat. In der Tat wäre anderenfalls laut Definition in *jedem* Zustand nach Abarbeiten von w eine Interaktion s' möglich mit $a = in(s')$ als Eingangsnachricht. Dann muss aber auch der Subtyp eine Interaktion s anbieten, deren Eingangsnachricht ein Supertyp von a ist. Außerdem ist zugesichert, dass es beim Typ in diesem Zustand mindestens eine Interaktion s'' gibt, die zu der Ausgabe von s konform ist. (Ohne Einschränkung kann angenommen werden, dass $s' = s''$ ist.) Wird also eine Nachricht von einer Instanz abgelehnt, die als beobachtbar ersetzbar für eine Instanz eines erwarteten Typs angesehen wird, so lässt sich dies anhand des Transitionssystems des Typs erklären. Man definiere nun:

- $int_y(x) := \{r \in int(x) \mid \exists s \in int(y) : in(s) \prec in(r)\}$ sind die *relevanten Interaktionen* in x in Bezug auf y: Man blendet alle Interaktionen im Zustand x aus, die aufgrund ihrer Eingangsnachricht nicht auftreten können, da sie im Typ nicht erwartet werden.

- $Y_R := \{y \in Y \mid \exists x \in X \, \forall r \in int_y(x) \, \exists s \in int(y) : \; r \prec s\}$ sind die *relevanten Zustände* in Y, die aufgrund der von ihnen ausgehenden Interaktionen Zuständen des Subtyps entsprechen können. Man beachte: Da der Subtyp in der Lage ist, eine Auswahl nichtdeterministischer Alternativen zu treffen, kann ein Teil der möglichen Zustände des Typs verloren gehen.

- $R_Y := \bigcup_{y \in Y_R} int(y)$ sind die von relevanten Zuständen in Y ausgehenden Interaktionen;
- $R_X := \{r \in \bigcup_{x \in X} int_y(x) \,|\, y \in Y_R\}$ sind Interaktionen, die von Zuständen in X ausgehen, welche relevanten Zuständen in Y entsprechen;
- $[r]_Y := \{r' \in R_Y \,|\, r' \prec r\}$ sind alle Interaktionen in R_Y, die zu r konform sind;
- $[r]_X := \{r' \in R_X \,|\, r' \prec r\}$ sind alle Interaktionen in R_X, die zu r konform sind.

Definition 5.25 Beobachtbare Ersetzbarkeit von Zustandsmengen
Sei X eine Menge von Zuständen eines Transitionssystems A und Y eine Menge von Zuständen eines Transitionssystems B. Man nennt Y *beobachtbar ersetzbar durch* X ($X \prec Y$), wenn gilt:

1. $\forall x \in X\ \exists y_x \in Y :\ (\forall s \in int(y_x)\, \exists r \in int(x) :\ in(s) \prec in(r)) \wedge (\forall r \in int_{y_x}(x)\, \exists s \in int(y_x) :\ r \prec s)$
2. $\forall r \in R_Y,\ fin(r) \neq \infty : X \xrightarrow{[r]_X} X', Y \xrightarrow{[r]_Y} Y' \Longrightarrow X' \prec Y'$

Punkt 1 stellt klar, dass es für jeden Zustand der Zustandsmenge des Subtyps einen in der Art entsprechenden Zustand des Typs gibt, sodass im Subtypzustand zum einen sämtliche eingehende Nachrichten verstanden werden, die im Zustand des Typs verstanden werden (erster Teil des Prädikats), zum anderen es für jede dieser relevanten Interaktionen eine Entsprechung im Typ bei diesem Zustand gibt, dass also keine nichtdeterministischen Alternativen hinzukommen (zweiter Teil des Prädikats).

Punkt 2 führt die Rekursion durch: Jede Interaktion lässt die Transitionssysteme in einer vorhersehbaren Art (auch in Bezug auf Nichtdeterminismus) fortschreiten. Die Ersetzbarkeit der Zustandsmengen ist genau dann gegeben, wenn durch das Fortschreiten keine Nichtersetzbarkeit der neuen Mengen auftritt. Diese Definition ist so ausgelegt, dass sich aus ihr unmittelbar ein Algorithmus zur Bestimmung der beobachtbaren Ersetzbarkeit von Zustandsmengen ableiten lässt (siehe Abschnitt 6.5.3 im folgenden Kapitel). Dass aus der beobachtbaren Ersetzbarkeit dieser Zustandsmengen die beobachtbare Ersetzbarkeit der Transitionssysteme und mithin der zu Grunde liegenden Typen folgt, folgt aus dem

Satz 5.26 Äquivalenz der beobachtbaren Ersetzbarkeit
Das Transitionssystem B mit Startzustand q_0^B ist beobachtbar ersetzbar durch das Transitionssystem A mit Startzustand q_0^A genau dann, wenn $\{q_0^A\} \prec \{q_0^B\}$.

Beweis. „⇒“: Voraussetzung sei die Eigenschaft zweier gegebener Transitionssysteme A und B laut Definition 5.24. Man definiere folgende Funktion, die über die bislang

hintereinander abgearbeiteten Interaktionen Buch führt:

$$ch(X') := \bigcup_{(X,r)} ch(X)r \text{ mit } X \xrightarrow{r} X' \text{ und } ch(q_0) := \varepsilon.$$

Wie üblich ist $Ar := \{ar \mid a \in A\}$. $ch(X)$ sind damit alle Interaktionsketten, die vom Startzustand q_0 ausgehend zu einem beliebigen Zustand $x \in X$ führen. Setzt man voraus, dass die Rekursion eine bestimmte Anzahl von Schritten durchgeführt hat, dann liegen zum aktuellen Zeitpunkt die Zustandsmengen X und Y vor. Die vorangegangenen Zustandsmengen erfüllten entsprechend stets die Bedingung 1 der Definition 5.25.

Sei nun angenommen, dass $w \in ch(X)$ eine Interaktionskette ist, zu der es durch die Rekursionsbedingung auch eine Kette $w' \in ch(Y)$ geben muss, zu welcher w komponentenweise konform ist. Während w das Transitionssystem A in einen Zustand $q_w \in X$ überführt, bringt diese Interaktionskette B in einen Zustand der Zustandsmenge $Y = Q_{w'}$. Da die beobachtbare Ersetzbarkeit der Transitionssysteme vorausgesetzt ist und w' die Bedingung der Implikation erfüllt, liefert die Definition diese Aussage (α):

$$\exists q' \in Q_{w'} \forall s' \in int(q') \exists s \in int(q_w) \exists s'' \in int(q') : in(s') \prec in(s) \wedge s \prec s''$$

Man gehe nun davon aus, dass die Zustandsmenge Y nicht durch X ersetzt werden kann. Dies geschieht o.B.d.A. dadurch, dass Bedingung 1 der Definition 5.25 nicht gegeben ist; anderenfalls führe man die Rekursion fort. In $x = q_w \in X$ sei es möglich, dass für jeden beliebigen Zustand $y \in Y = Q_{w'}$ gelte, dass entweder eine Interaktion von y aus aufgrund der Eingangsnachricht nicht in q_w möglich ist oder dass q_w eine neue Alternative eingeführt hat. Aber dies steht im Widerspruch zu (α). Folglich muss die Bedingung 1 für beobachtbar ersetzbare Transitionssysteme stets erfüllt sein. Die Definition liefert, ausgehend von $X = \{q_0^A\}$ und $Y = \{q_0^B\}$, die Aussage des Satzes.

„$\Leftarrow$“: Man setze die beobachtbare Ersetzbarkeit der Zustandsmengen laut Definition 5.25 voraus. Um zu zeigen, dass die Aussage der Definition 5.24 für alle Interaktionsketten w gilt, führe man eine Induktion nach der Länge der Interaktionskette w durch.

$|w| = 0$: Ist die Interaktionskette leer, so befinden sich beide Transitionssysteme im Startzustand (beziehungsweise im ersten eigentlichen Zustand nach der Pseudointeraktion); also ist $Q_\varepsilon^B = \{q_o^B\}$ und für q_ε setzt man q_0^A. Die mit ε übereinstimmende Interaktion in B ist ebenfalls ε und $q' = q_0^B$. Die Durchführung der Rekursion hat $\{q_0^A\} \prec \{q_0^B\}$ bereits bestätigt, also muss es also für $x = q_\varepsilon$ ein $y \in Q_\varepsilon^B$ geben (nämlich $q' = q_0^B$), sodass gilt:

$$\forall s \in int(y) \exists r \in int(x) : in(s) \prec in(r) \wedge \forall r \in int_y(x) \exists s \in int(y) : r \prec s.$$

Genau dies fordert man laut Definition 5.24, wobei r durch die Übereinstimmung der Eingangsnachrichten offensichtlich auch in den relevanten Interaktionen $int_y(x)$ liegt.

$|w| = n+1$: Sei nun angenommen, dass tatsächlich alle Interaktionsketten der Länge $|w| = n$ die in der Definition gezeigte Implikation bewirken. Nun nehme man an, dass es

1. eine Interaktion r in A gebe, sodass in A vermöge $q_0^A \xrightarrow{wr} q_{wr}$ ein Zustandswechsel stattfinde und es

2. eine Interaktion r' in B gebe, sodass $r \prec r'$ gilt und ein Zustandswechsel $q_0^B \xrightarrow{w'r'} Q_{w'r'} \neq \emptyset$ gebe.[7]

Diese Annahme erfüllt die linke Seite der Implikation von Definition 5.24. Die Systeme werden durch $q_0^A \xrightarrow{w} Q_w =: X$ und $q_0^B \xrightarrow{w'} Q_{w'} =: Y$ in die jeweiligen Zustandsmengen X und Y überführt. Für alle $x \in X$ muss es laut Definition 5.25 ein $y \in Y$ geben, sodass sämtliche Interaktionen von diesem y aus in Bezug auf die Eingangsnachricht von x ausgehen könnten (unter Achtung der Kontravarianz), wobei keine weiteren Alternativen in x eingeführt werden. Gemäß Punkt 1 der Annahme besitzt mindestens ein x eine nichtleere Interaktionsmenge $int(x)$, da $r \in int(x)$; gemäß Punkt 2 muss es ein y geben mit $int(y) \neq \emptyset$ und $int_y(x) \neq \emptyset$. Damit ist ein weiterer Rekursionsschritt durchzuführen, da $r \in [r]_X$ und $r' \in [r]_Y$ durch die obigen Annahmen gilt; man erhält nun X' und Y' mit $X \xrightarrow{[r]_X} X' \neq \emptyset$ und $Y \xrightarrow{[r]_Y} Y' \neq \emptyset$. Die Rekursion mit (X', Y') in der Definition 5.25 liefert die vorausgesetzte Ersetzbarkeit der Zustandsmengen. Dafür ist jedoch notwendig, dass gilt:

$$\forall x' \in X' \exists q' \in Q_{w'r'} = Y' : (\forall s' \in int(q') \exists s \in int(x') : in(s') \prec in(s)) \\ \wedge (\forall s \in int_{q'}(x') \exists s'' \in int(q') : s \prec s'').$$

Insbesondere gilt dies für $x' = q_{wr} \in X'$. Die Wahl von s impliziert, dass $s \in int_{q'}(x')$ ist, also erhält man die rechte Seite der Implikation aus Definition 5.24. ■

5.7 Eigenschaften des Hybridtypsystems

Die Hybridtypen bilden mit den oben beschriebenen Vergleichsregeln ein Typsystem. Zunächst sollte ein Blick auf die Typregeln geworfen werden, die zeigen, dass dieses System von Typen zu Recht die Bezeichnung Typsystem trägt.

5.7.1 Typregeln

Nach [Car97] wird ein *Typsystem* durch eine Menge von *Typregeln* bestimmt. Die Typregeln basieren auf Beurteilungen *(Judgments)*, welche für das jeweilige System angegeben werden müssen. Die für das Hybridtypsystem gültigen Beurteilungen lauten:

[7] Dies ist wichtig, da es durchaus sein kann, dass ein solches r' nicht existiert; dieser Fall ist aber nicht von Interesse, da der Subtyp durchaus weitere Interaktionen mit unbekannten Eingangsnachrichten einführen darf.

$\Gamma \vdash \diamond$	Die Typumgebung Γ ist wohlgeformt.
$\Gamma \vdash A$	A ist ein wohlgeformter Hybridtyp in Γ.
$\Gamma \vdash a : A$	a ist eine Entität des Typs A in Γ.
$\Gamma \vdash A \prec B$	A ist ein hybrider Subtyp von B in Γ.
$\Gamma \vdash A \prec_{syn} B$	A ist ein syntaktischer Subtyp von B in Γ.
$\Gamma \vdash A \prec_{sem} B$	A ist ein semantischer Subtyp von B in Γ.

Üblicherweise spricht man im Zusammenhang von Typsystemen von einer *statischen Typumgebung* Γ, welche sich als Menge von Variablendeklarationen darstellt (etwa $x_1 : A_1,\ x_2 : A_2, \ldots$). Das Hybridtypsystem soll Einsatz in einem *offenen System* finden; demzufolge besteht die Typumgebung aus der Gesamtmenge aller aktuell verwendeten Hybridtypen innerhalb des Systems. Die Typumgebung ist genau dann wohlgeformt, wenn innerhalb des Systems den Agenten nur gültige (wohlgeformte) Hybridtypen zugewiesen werden. Die für dieses Typsystem geltenden Regeln lauten:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \prec A}$$ Reflexivität (gilt auch für $\prec_{syn}$ und $\prec_{sem}$)

$$\frac{\Gamma \vdash A \prec B \quad \Gamma \vdash B \prec C}{\Gamma \vdash A \prec C}$$ Transitivität (gilt auch für $\prec_{syn}$ und $\prec_{sem}$)

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \prec B}{\Gamma \vdash a : B}$$ Subsumtion (gilt auch für $\prec_{syn}$ und $\prec_{sem}$)

Es lassen sich weitere Regeln aufstellen, welche die Typkonformität aus der Konformität der einzelnen Komponenten des Typs herleitet; diese Regeln sind durch die in den vergangenen Abschnitten erklärten Bedingungen der Konformität von syntaktischem, semantischem und Transitionstyp bereits definiert und hier der Übersichtlichkeit halber nicht aufgeführt. Die Gültigkeit dieser drei Regeln lässt sich (informell) leicht plausibel machen:

Reflexivität Werden zwei identische syntaktische Typen verglichen, so sind die Ein- und Ausgangsnachrichtenmengen gleich; es lassen sich jeweils paarweise gleiche Nachrichten finden. Bei identischen semantischen Typen sind die Annotationen paarweise identisch; im gesamten Hybridtyp ist die Menge der Interaktionen identisch.

Transitivität Für die aus Nachrichtentypmengen beziehungsweise Annotationsmengen definierten syntaktischen und semantischen Typen ist die Aussage klar. Im Falle des vollständigen Hybridtyps gilt, dass die (implizit definierte) Menge von Interaktionen über die Konformität entscheidet, sodass auch hier die Transitivität aufgrund der Mengeninklusion gegeben ist.

Subsumtion Diese Regel sagt aus, dass ein Agent in jedem Falle eingesetzt werden kann, wenn eigentlich ein Supertyp erwartet wurde. Die Hybridtypen wurden gerade so definiert, dass sie Auskunft darüber geben, welche Arten von Interaktionen er *mindestens* versteht. Da sich Supertypen darüber definieren, dass sie weniger Interaktionen voraussetzen (aber andere nicht verbieten), ist diese Regel zutreffend. Dies gilt entsprechend für die Nachrichtentypmengen und die Annotationen.

5.7.2 Maximale und minimale Typen

Ein Nachrichtensubtyp darf stets Nachrichtenelemente hinzufügen. Ein Agentensubtyp darf also in Eingangsnachrichten auf Elemente verzichten oder diese generalisieren (Kontravarianz), während er bei Ausgangsnachrichten Elemente hinzufügen oder diese spezialisieren kann (Kovarianz). Es stellt sich die Frage, ob es maximale und minimale Typen gibt. Ein maximaler Typ ist Supertyp jedes anderen Typs, während ein minimaler Typ Subtyp jedes Typs ist.

Syntaktischer Typ

Ein Agentensubtyp darf stets Eingangsnachrichten hinzufügen; ein Supertyp kann daher Nachrichten weglassen. Umgekehrt darf ein Subtyp niemals mehr Nachrichten liefern, als vom Typ erwartet werden. Der Supertyp kann die Ausgangsnachrichten beliebig generalisieren und kürzen. Eine Ausnahme ist jedoch im Hybridtypsystem vorhanden, nämlich die Pseudonachricht *migration*. Diese ist keine spezielle Nachricht und kann somit von *java.lang.Object* nicht generalisiert werden. Ein maximaler syntaktischer Typ ist demnach so zu definieren, dass als einzige Elemente *java.lang.Object* und *migration* gesendet und keine Eingabe erwartet wird. Subtypen, die nichts senden, verstoßen nicht gegen die Subtypeigenschaft, da sie *keine unerwarteten* Nachrichten senden. Da keine Eingangsnachrichten erwartet werden, steht es jedem Subtyp frei, solche hinzuzufügen.

Der minimale Typ ist dual hierzu zu definieren, nämlich mit *java.lang.Object* als einzige Eingangsnachricht und ohne Ausgangsnachrichten. In diesem Falle benötigt man keine Beachtung von Pseudonachrichten. Wie man sich leicht überlegt, ist jede mögliche Eingangsnachricht stets eine Verlängerung und eine Spezialisierung dieser Eingangsnachricht.

Semantischer Typ

Für Zeichenketten sind keine Hierarchisierungen vorgesehen, sodass dort immer eine Übereinstimmung vorhanden sein muss. Konzeptgraphen hingegen hängen von der verwendeten Konzeptontologie ab. Ist diese als Konzepthierarchie vorhanden (wie in

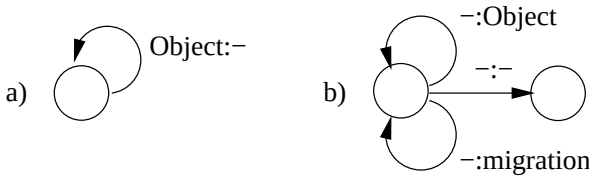

Abbildung 5.6: Minimale und maximale Transitionstypen

diesem Typsystem), so definiert diese ein maximales Element, das üblicherweise *Etwas* oder *Something* genannt wird. Eine Annotation, welche dieses Konzept als einziges aufführt, kann nur spezialisiert werden und nimmt daher die Position des maximalen Typs ein. Da die Typen in diesem System nur Stellennutzern zugeordnet werden, kann bei der Selbstannotation das Konzept *Stellennutzer* ohne Relationen zur Erzeugung eines maximalen semantischen Typs verwendet werden.

Einen minimalen Typ zu erstellen gelingt nicht, weil die Menge, welche Teilmenge aller von Konzeptgraphen beschriebenen Entitätenmengen ist, die leere Menge ist. Zwar könnte ein Konzept *Nichts* entworfen werden; dies ist aber nur von theoretischem Nutzen, da diese Annotation logischerweise keiner existierenden Instanz zugeschrieben werden kann.

Hybridtyp

Für die Bildung eines maximalen Hybridtyps ist wichtig zu bemerken, dass jeder Subtyp stets Interaktionen hinzufügen darf. Insofern ähnelt die Bildung des maximalen Typs jener des syntaktischen Typs. Ein maximaler Hybridtyp bestünde demnach aus einem einzigen Startzustand ohne Transitionen. Man beachte jedoch, dass Interaktionen nicht konform sind, wenn die Ausgangsnachrichtenfolgen in der Länge nicht übereinstimmen (Definition 5.19). Das Senden von Nachrichten ist für den Interaktionstyp relevant, sodass Interaktionen ohne Ausgaben nicht zu Interaktionen mit Ausgaben konform sind (siehe Abschnitt 5.6).

Der semantische Typ im weiteren Sinne reicht in den Vergleich des Transitionstyps hinein, indem dessen Annotationen von Sendern, Empfängern, Nachrichten und Zuständen zum Vergleich herangezogen werden. Vernachlässigt man diese Annotationen, so ist ein maximaler Transitionstyp in Abbildung 5.6 rechts dargestellt. Dieser Typ fängt die Situationen ab, dass ein Subtyp entweder spontan sendet oder nicht sendet. Wie beim syntaktischen Typ muss die Pseudonachricht *migration* separat behandelt werden, will man die mobilen Agenten nicht ausgrenzen. Um Zustandsannotationen zu beachten, muss die allgemeinste Annotation „Zustand“ verwendet und für die Transitionen als Empfänger *ANY* vorgesehen werden. Außerdem müssen die sendenden Transitionen verdoppelt werden, wobei *PLACE* als Empfänger angegeben ist, da *PLACE* nicht von *ANY* impliziert wird (siehe Abschnitt 5.5.7).

Der minimale Typ (ohne Annotationen) ist wieder dual zum maximalen Typ zu de-

finieren, wobei die Besonderheit des Vorhandenseins von Ausgaben zu beachten ist. Streng genommen müsste auch hier wieder die leere Menge als minimaler Ausgabentyp herangezogen werden, wenn nicht dadurch die Interaktionen gemäß ihrer Definition die Konformität verlieren würden. Erlaubt man einem Subtypen, nichts zu schicken, obwohl eine Antwort erwartet wird, so ist der in Abbildung 5.6 links dargestellte Typ ein minimaler Transitionstyp. Eine weitere Alternative wie im Falle des maximalen Typs hinzuzunehmen ist hier nicht möglich, da ein Supertyp keine Alternativen streichen darf. Damit ist auch eine generelle Beachtung von Annotationen wie beim maximalen Typ nicht möglich.

Um zu einem minimalen Hybridtyp zu kommen, müsste auch ein minimaler semantischer Typ vorhanden sein, welchen es jedoch nicht gibt, wie schon begründet wurde. Beim Vergleich muss somit auf den Selbstannotationsvergleich verzichtet werden.

5.7.3 Bewertung des Typsystems

Nachdem das Typsystem vorgestellt wurde, sollte ein kurzer Blick darauf geworfen werden, ob die in Abschnitt 5.1.4 genannten Bedingungen erfüllt werden und ob das Typsystem die *Korrektheitsbedingung* erfüllt.

Eindeutigkeit

Die Eindeutigkeit der Typbeschreibung ist genau dann gewährleistet, wenn die Typbeschreibung durch das gleiche Auswertungssystem verarbeitet wird. Wird das System wie in AMETAS unter Java angewendet, so ist es erforderlich, an allen Stellen eine kompatible Java-Version zu benutzen, welche zumindest sicherstellt, dass die in Nachrichten verwendeten Systemklassen überall gleich sind und dieselbe Hierarchie aufweisen.

Die Interpretation von Annotationen führt einen Freiheitsgrad in die Beschreibung ein. Hier muss klar sein, dass die verwendeten Begriffe überall dieselbe Semantik haben. Im Falle von Konzeptgraphen reduziert sich dies auf die Verwendung der gleichen Ontologien.

Universalität und Erweiterbarkeit

Eine Entität, welcher ein Typ zugewiesen wird, kann durch diesen Typ in Bezug auf ausgetauschte Nachrichten und auch semantisch charakterisiert werden. Diese zweite Möglichkeit erlaubt es, beliebige Entitäten zu beschreiben. In AMETAS sind die an der (externen) Kommunikation beteiligten Entitäten gerade die so genannten Stellennutzer. Ihr Kommunikationsmuster kann durch einen Hybridtyp beschrieben werden.

Das System ist beliebig mit weiteren Typbeschreibungen erweiterbar, da es keine obere Grenze für die Menge möglicher Hybridtypbeschreibungen gibt. Jede Beschreibung kann durch Hinzunahme weiterer Nachrichten, Interaktionen oder Verfeinerun-

gen der semantischen Beschreibung zu einer neuen, gültigen Beschreibung geändert werden.

Abstraktion

Die semantische Beschreibung mittels Konzeptgraphen erlaubt gemäß der Definition der Konzeptgraphen eine einfache Spezialisierung durch Hinzufügen weiterer Konzepte und Relationen. Dieser Vorgang steht für eine genauere Erklärung eines Sachverhalts. Insofern wird durch das Weglassen dieser Komponenten ein Sachverhalt oberflächlicher, allgemeiner dargestellt.

Die der syntaktischen und protokollbezogenen Beschreibung zu Grunde liegende Annahme ist, dass die Beschreibungen ein *Minimum* an Fähigkeiten des charakterisierten Objekts darstellen. Die Hinzunahme weiterer Interaktionen wirkt sich nicht auf die Funktionalität des erwarteten Basisobjekts aus, sofern man – wie es hier geschieht – die Möglichkeit der Modifikation des Zustands von dritter Seite ausschließt, also keine konkurrierenden Klienten zulässt. Das Objekt verhält sich stets so, wie es durch den Typ vorgegeben wird, auf den seine Beschreibung passt. Die zusätzlichen Funktionalitäten können ignoriert werden.

Dadurch ist es auch möglich, eine Mehrfachvererbung zu realisieren. Diese ist so definiert, dass der gemeinsame Abkömmling die Eigenschaften der Elterntypen erbt. Die Hinzunahme einer Beschreibung, die die Eigenschaften eines Elterntyps erfüllt, stellt sich als inkrementelle Modifikation, als gemeinsame Verfeinerung dar [WZ88].

Auswertbarkeit

Die Auswertbarkeit des Typs wurde in den obigen Abschnitten durch die theoretischen Betrachtungen gezeigt. In Kapitel 6 wird ein Algorithmus neben weiteren Details zur Integration des Typsystems im Agentensystem AMETAS präsentiert.

Korrektheit

Gewöhnlich müssen Typsysteme auf ihre *Korrektheit (engl. soundness)* überprüft werden. Darunter versteht man, dass die vom Typsystem gemachten Aussagen sinnvoll sind, dass also eine Instanz, welche einem Typ angehört, tatsächlich die angegebenen Interaktionen unterstützt oder die angegebene Semantik aufweist. Ein korrektes Typsystem bürgt für die korrekte Ausführung von Programmen, die als wohltypisiert erkannt sind [Car97].

Die korrekte Ausführung kann bei Anwendungen, die das Hybridtypsystem nutzen, nur im Bereich der Interaktionen (oder des einfachen Nachrichtenaustauschs) beurteilt werden. Eine unzutreffende semantische Beschreibung wird gewöhnlich nicht zu Fehlern führen, wenn es sich nicht um Konstantenannotationen handelt. Der Ersteller einer Hybridtypspezifikation muss korrekte Aussagen zu dem jeweiligen Agenten treffen, da

das Laufzeitsystem eine automatische Typisierung nicht erlaubt. Diese Voraussetzung muss getroffen werden, da Korrektheitsbetrachtungen sonst hinfällig sind.

Ist die Typisierung zutreffend vorgenommen worden, so zeigen die Betrachtungen zum Transitionstyp, dass man von einer korrekten Verarbeitung ausgehen kann. Das Typsystem ist in diesem Sinne korrekt. Aber man muss sich vor Augen führen, dass Hybridtypen nicht wie traditionelle Typen vor der Laufzeit getestet werden können; ferner findet auch keine dynamische Überprüfung statt, wenn die Interaktion im Gange ist. Das Hybridtypsystem kann den Klienten lediglich informieren, dass die Kommunikation wie geplant durchgeführt werden könnte. Im Falle von Nichtdeterminismus muss der Partner stets alle Möglichkeiten in Betracht ziehen; dies stellt nicht nur Bedingungen an die Implementierung des typisierten Agenten, sondern auch an jene des Partners.

5.8 Alternative und verwandte Ansätze

Abschließend sollen noch einige andere Ansätze vorgestellt werden, wie eine Erweiterung einer syntaktischen Typisierung vorgenommen werden kann, und zwar ebenfalls auf Basis von Protokollspezifikationen oder semantischen Spezifikationen.

5.8.1 Kommunizierende Prozesse

Um eine geeignete semantische Theorie paralleler Prozesse zu entwickeln, sind die Modellierungen dieser Prozesse als Funktionen, welche eine Menge von Eingaben auf eine Menge von Ausgaben abbilden, nicht ohne Informationsverlust durchführbar, da die parallel laufenden Prozesse eventuell untereinander interagieren und den Berechnungsablauf beeinflussen [dH84].

Die Äquivalenz zweier Prozesse kann mit Hilfe von *Beobachtern (Observers)* überprüft werden. Ein Beobachter kann eine Berechnung durch den Prozess veranlassen; er beliefert diesen Prozess mit Daten und nimmt von diesem die Resultate entgegen. Die Problematik der nichtdeterministischen Alternativen wird in [Hen85] betont. HENNESSY stellt klar, dass Prozesse, welche ihre Verarbeitungsfolgen in Form endlicher Automaten darstellen, nicht immer äquivalent sind, obwohl sie in der üblichen Interpretation endlicher Automaten die gleiche Sprache akzeptieren:

> *Es sollte nun offensichtlich sein, dass die traditionelle Automatentheorie … immer nur die Menge der Zeichenketten in Betracht zieht, welche akzeptiert werden können. Es wurde an anderer Stelle … dargelegt, dass dies für viele Anwendungen unpassend ist.*

Dies rührt vor allem von dem Problem her, dass Prozesse schrittweise abgearbeitet werden und sich bisweilen für einen von mehreren Übergängen entscheiden müssen. Übergänge können jedoch nicht mehr rückgängig gemacht werden.

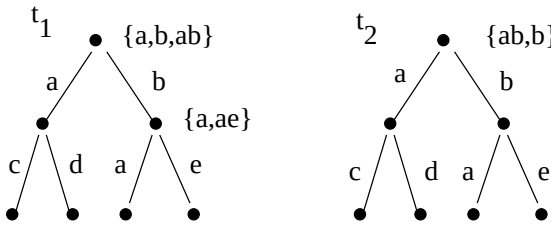

Abbildung 5.7: Akzeptanzbäume

Hennessy benutzt in [Hen85] so genannte *Akzeptanzmengen*, um eine Äquivalenz zwischen den von ihm entworfenen *Prozessbäumen* nachzuweisen. Unter einer Akzeptanzmenge des Zustands *x* versteht man die Menge aller im Zustand *x* akzeptierten Nachrichten. Damit ein endlicher Automat M_1 einen weiteren Automaten M_2 mit gleicher Sprache ersetzen kann, muss M_1 „weniger nichtdeterministisch“ sein als M_2. Dies drückt sich durch das Enthaltensein der Akzeptanzmengen der entsprechenden Knoten in den Prozessbäumen aus. Da beide dieselbe Sprache akzeptieren, gibt es bei M_1 weniger Möglichkeiten, während der Abarbeitung der Nachrichtenkette in eine Sackgasse zu geraten; passiert es dennoch, dann muss das Scheitern bei M_2 auch möglich sein. In *Akzeptanzbäumen* werden Zustände, welche in Folge einer nichtdeterministischen Alternative erreicht werden, in einem Knoten zusammengefasst, welcher durch die Mengen der möglichen Eingabemengen attribuiert wird (Abbildung 5.7).

Der t_1-Baum deutet an, dass im Ausgangszustand drei Zustände vereinigt sind, von denen einer die Nachricht *a*, der andere die Nachricht *b* versteht und ein dritter beide Nachrichten akzeptiert; bei Akzeptieren von *b* gelangt er in den Zustand, welcher einen Zustand beinhaltet, der *a* akzeptiert, während ein zweiter sowohl *a* als auch *e* akzeptiert. Ist keine Akzeptanzmenge angegeben, so ist damit genau ein Zustand deklariert, der die Nachrichten aller ausgehenden Kanten akzeptiert. Wie man leicht nachweist, ist der Baum t_2 in dem Sinne verträglich mit t_1, dass er dieselbe Sprache akzeptiert, aber die Menge der Möglichkeiten des Scheiterns ist kleiner: Die Akzeptanzmengen der Knoten des Baums t_2 sind in den Akzeptanzmengen der entsprechenden Knoten in t_1 enthalten. Dieser Ansatz wurde im Hybridtypsystem integriert, um Aussagen über die Einschränkung nichtdeterministischer Alternativen zu gewinnen.

5.8.2 Estelle und SDL

Estelle [Hog89, ISO91, ISO97] ist eine Spezifikationssprache, welche auf dem Modell der erweiterten endlichen Automaten basiert. Diese Automaten – *Moduln* genannt – definieren als Teil ihres Zustands auch lokale Variablen und können Übergänge durch diese Variablen steuern. Eine Systemspezifikation in Estelle stellt sich als Beschreibung miteinander interagierender, erweiterter endlicher Automaten dar.

Transitionen werden in Estelle durch *From-to*-Klauseln erklärt. Dabei sind die Konzepte von Nichtdeterminismus und spontanen Übergängen explizit enthalten. Beispiels-

weise definiert man mittels

```
from Zust1 to Zust2
   when Kanal1.Nachr1 begin
       <Aktion>
   end;
from Zust1 to Zust3
   when Kanal1.Nachr1 begin
       <Aktion>
   end;
```

eine nichtdeterministische Alternative, da die Eingangsnachricht *Kanal1.Nachr1* zu zwei unterschiedlichen Zuständen führen kann. Spontane Übergänge werden durch das Fehlen der *When*-Bedingung deklariert. Die Attribuierung von Zuständen und Nachrichten, wie es im Hybridtypsystem in Form der Annotationen auftritt, ist nicht vorgesehen.

Die *Specification and Description Language (SDL)* [Hog89, ITU99] ist in der Lage, ein System in Form konkurrierender Prozesse zu beschreiben. SDL definiert ebenfalls den *Prozess* als Basiselement der Modellierung. Verschiedene Prozesse stehen untereinander durch *Kanäle* in Verbindung, über welche sie sich *Signale* zuschicken können. Jeder Prozess verfügt über Eingangswarteschlangen, welche die Signale der anderen Prozesse, welche an diesen Prozess geschickt wurden, zwischenspeichert. Der Prozess kann die Signale selbständig aus seiner Warteschlange abholen.

Ein Prozess beinhaltet eine Beschreibung in Form eines endlichen Automaten, um das *Signalprotokoll* zu repräsentieren. Dabei werden sowohl interne Zustandsänderungen als auch Ein- und Ausgabevorgänge spezifiziert, welche sich auf die definierten Kanäle beziehen müssen. Die Kanten des endlichen Automaten sind mit Eingangs- und Ausgangssignalen attribuiert. Die Modellierung mit SDL ist hierarchisch organisiert:

- die Systemsicht, welche die Interaktion der Blöcke beschreibt;
- die Blöcke, welche die interagierenden Prozesse beschreiben;
- die Prozesse, welche durch Zustandsübergangsdiagramme erklärt sind und
- Makros/Prozeduren, welche von Prozessen aufgerufen werden.

Die zwischen den Prozessen ausgetauschten Daten werden über abstrake Datentypen (ADT) oder über in ASN.1-Format deklarierte Datentypen beschrieben.

Prinzipiell könnte also ein Agent als einzelner Prozess in der SDL-Sicht modelliert werden, wobei das Mailbox-System von AMETAS den benannten Kanälen entspricht und die Signale den ausgetauschten Nachrichten entsprechen. Jedoch sind Konzepte wie Mobilität nicht realisiert, auch wenn eine dynamische Prozesserzeugung und -beendigung möglich ist. Insbesondere die Integration semantischer Informationen wie der im Hybridtypsystem vorkommenden Annotationen ist nicht explizit vorgesehen.

5.8.3 π-Kalkül

Um die Abläufe in einem System paralleler Prozesse besser modellieren zu können, wurde von MILNER [MPW92] das so genannte π-Kalkül entwickelt. Es definiert *Prozesse* und *Namen*, wobei ein Prozess eine abgeschlossene Einheit ist, welche mittels *Kanälen* mit anderen Prozessen kommuniziert. Diese anderen Prozesse bilden die *Umgebung* des Prozesses. Kanäle werden durch Namen repräsentiert, wobei wiederum Namen über diese Kanäle transportiert werden, also auch zur Repräsentation von Daten dienen. Der Ausdruck

$$P_1 \equiv \overline{y}x.P_2$$

bedeutet, dass der Prozess P_1 über den (Ausgabe-)Kanal y den Namen x sendet und als Prozess P_2 weiterläuft. Umgekehrt zeigt

$$P_1 \equiv y(x).P_2$$

an, dass P_1 über den (Eingabe-)Kanal y einen Namen z entgegennimmt, ihn an den Namen x bindet (symbolisiert durch die Klammern) und als $P_2\{z/x\}$ fortgesetzt wird.

Das π-Kalkül eignet sich gut zur Modellierung paralleler Prozesse, die über bestimmte Kanäle kommunizieren. Parallel laufende Prozesse werden durch $P \mid Q$ dargestellt. Milner demonstriert in [MPW92], dass die Ausdrucksstärke des π-Kalküls ausreicht zu modellieren, dass ein Prozess einen zweiten beauftragt, eine Kommunikation mit einem dritten Prozess durchzuführen.

Die einzelnen Prozesse werden von Milner *Agenten* genannt, da sie eigenständig laufen und als abgeschlossene Einheiten definiert sind. Auch wenn das π-Kalkül eine Nähe zu autonomen Agenten suggeriert, welche sich Namen (Nachrichten) über Kanäle (Postfächer) zukommen lassen und dabei eigenständig laufen, so zeigt dieses einfache Modell einige Schwächen:

- Die Kommunikation ist synchron. Es gibt in dieser Hinsicht Erweiterungsvorschläge von BOUDOL [Bou92], das π-Kalkül als eine *chemische abstrakte Maschine* zu modellieren; Nachrichten können von den Ausgabekanälen in eine „Lösung“ übergehen und mit ihrem Gegenpart „reagieren“. Diese Erweiterung wird von CARDELLI und GORDON später (etwa in [CG99]) als *asynchrones π-Kalkül* aufgegriffen.

- Die Mobilität ist nicht explizit modelliert. Zwar können über Kanäle auch Namen von Agenten verschickt werden, aber das Kalkül macht keine Angaben darüber, ob der Agent tatsächlich migriert oder lediglich eine Referenz übermittelt wird [CG99].

Das π-Kalkül findet seine Anwendung vor allem dann, wenn eine Berechnung als parallele Abarbeitung miteinander kommunizierender Prozesse auf ihre Korrektheit überprüft werden soll.

5.8.4 Mobile Ambients

CARDELLI und GORDON schlagen in [CG98] eine Erweiterung des asynchronen π-Kalküls vor, welche auf die Modellierung der Mobilität eingeht.

Eine Umgebung *(Ambient)* ist ein abstrakter Behälter für weitere Umgebungen oder Prozesse. Nachrichten und Prozesse können eine Umgebung nie verlassen, aber ein Prozess kann die Ermächtigung *(Capability)* besitzen, seine unmittelbare Umgebung in eine andere Umgebung oder aus einer anderen Umgebung wandern zu lassen. Außerdem kann er das Recht haben, eine Umgebung zu öffnen, das heißt sie aufzulösen, wenn sie in seiner Umgebung liegt.

Ein System mobiler Agenten kann im Ambient-Kalkül wie folgt modelliert werden:

Nachricht Spezielle Umgebung, welche so definiert ist, dass sie an ihrem Zielpunkt die zu übermittelnden Daten deponiert.

Agent Umgebung, die das Agentenverhalten als Prozesse in sich birgt. Die interne Kommunikation bleibt außen unsichtbar. Die externen Kommunikationsvorgänge werden durch Nachrichten realisiert.

Stelle Umgebung, welche Umgebungen empfängt und versendet (Agenten). Um Nachrichten zwischen Agenten zu verwalten, kann eine Stelle eine dedizierte Umgebung definieren (Postfachsystem).

Reiseroute Verkettung von Ermächtigungen zum Betreten von Umgebungen (Stellen).

Übermittelbare Werte – also solche Einheiten, welche die Nutzlast einer Nachricht sein können – sind Namen oder Ermächtigungen [CG98]. Es ist nicht notwendig, spezielle Basistypen einzuführen, da es möglich ist, einzelne Bits als Umgebungen der speziellen Namen *null* und *eins* zu definieren und daraus dann sämtliche komplexen Datentypen aufzubauen.

Das Ambient-Kalkül ist in dieser einfachen Form untypisiert. Cardelli und Gordon beschreiben in [CG99], wie das Kalkül mit einer Typisierung ausgestattet werden kann. Im Wesentlichen werden Nachrichten und Umgebungen mit speziellen Namen dekoriert, um den Zutritt zu Umgebungen nur für geeignet dekorierte Nachrichten zu erlauben. Ein *Agententyp* wird erklärt als der Typ von Agentennamen, welche eine Kommunikation von Nachrichten eines bestimmten Typs erlauben. Dies wird ebenfalls in [CG99] anhand einer einfachen, *Telescript*-artigen Agentenprogrammierungssprache demonstriert. Wie die Autoren bemerken, sind

> *die diesem Ansatz entspringenden Typen … insofern ungewöhnlich, dass sie weder zu Kanal- noch Funktionstypen direkt korrespondieren. Sie garantieren die Korrektheit des Nachrichtenaustauschs und erlauben eine große Flexibilität in Bezug auf Mobilität.*

Die Typisierung könnte somit als *rein syntaktisch* angesehen werden und sieht keine weiteren semantischen Annotationen vor. Außerdem sieht der Ansatz keine Subtypbeziehung vor; die Konformität basiert also ausschließlich auf Gleichheit.

5.8.5 Semantische Vor- und Nachbedingungen

Liskov und Wing beschreiben in [LW93], wie eine Subtyprelation durch semantische Angaben spezifiziert werden kann. Dabei gilt es, das *Verhalten* eines Objekts möglichst genau zu spezifizieren. Die Frage, ob ein Objekt einem Subtyp eines gegebenen Typs entspringt, kann nach Vergleich der Datentypen erst entschieden werden, wenn das Verhalten des Subtyps jenem des Supertyps entspricht. Als Beispiel führen sie die Datentypen *Bag* und *Stack* an. Ein *Bag* verfügt über zwei Methoden *put* und *get*, während der Stack über *push, pop und swap_top* verfügt. Die *Vorbedingung (pre-condition)* für *get* ist, dass der Bag nicht leer sein darf; analog gilt dies für *pop* beim Stack. Die *Nachbedingung (post-condition)* lautet, dass der Bag modifiziert wurde und ein Wert geliefert wird, welcher aus dem Bag entfernt wurde. Beim Stack ist die Nachbedingung spezifischer: Der Stack wurde modifiziert; das zuoberst liegende Element wurde geliefert, das vom Stapel entfernt wurde. *swap_top* schließlich kann als eine *pop*-Operation, gefolgt von einer *push*-Operation des neuen Wertes, erklärt werden.

Wie in Abschnitt 3.4.4 schon erwähnt wurde, fordern Liskov und Wing die Erklärbarkeit des durch den Subtyp hinzukommenden Verhaltens durch die vom Typ gebotenen Funktionen. Hintergrund ist, dass vor allem bei gemeinsamer Benutzung eines Objekts durch zwei Klienten es nie dazu kommen darf, dass durch die Aktionen eines Klienten das Objekt sich für den anderen Klienten unerklärlich verhält. Ähnliche Bedingungen stellt Nierstrasz in [Nie95] im Kontext des Begriffs *Request satisfiability*, welcher die Anfrageersetzbarkeit im Falle parallel anfragender Klienten betrachtet: Der Subtyp muss in seiner Funktionalität auf die Funktionen des Typs eingeschränkt werden, damit keine Zustandsänderungen auftreten können, welche aus der Sicht eines Klienten, der die Instanz des Typs erwartet, nicht erklärbar wären.[8]

Semantische Annotationen in der von Liskov und Wing vorgeschlagenen Form geben eine präzise Auskunft über die Semantik der jeweiligen Methode. Problematisch ist jedoch das Aufstellen der Vorbedingungen und Prädikate für Methoden, welche deutlich komplexer als das häufig zitierte Bag/Stack-Beispiel sind. Die auf diesem Wege gewonnenen Beschreibungen sind in der Regel *überspezifiziert*, da sie den wesentlichen Ablauf der Methode offenlegen. Ferner erfordert die Implementierung des Vergleichs eine ineffiziente Auswertung beliebig komplizierter logischer Ausdrücke auf Implikation.

[8] Das hier vorgestellte Hybridtypsystem schließt den Fall konkurrierender Klienten aus, indem es die getrennte Behandlung von Nachrichten verschiedener Klienten vorschreibt.

5.8.6 Typisierte Komponenten

Einen ähnlichen Ansatz zur Typisierung wie das hier vorgestellte Hybridtypsystem verfolgt die Arbeitsgruppe Verteilte Systeme an der Universität Hamburg im Kontext von *Software-Komponenten* [GZMW01]. Einsatzgebiet dieses Typsystems ist die generative Systemerstellung auf Basis von Komponenten gemäß dem *CORBA Component Model*.

Komponenten werden in diesem System mit einer Typbeschreibung ausgestattet, welche diese auf drei Ebenen in Form einer Baumstruktur charakterisiert:

1. Der *Komponententyp* setzt sich durch eine Menge so genannter *semantischer Business-Event-Typen (BE-Typen)* zusammen. Die Modellierung lässt sich betont durch die Nähe zur Anwendung und Interaktion mit den Kunden leiten: Im Falle einer Bank könnte ein Business Event die Einrichtung oder Auflösung eines Kontos oder eine Überweisungsanforderung sein. Diese Geschäftsvorgänge sind üblicherweise eine Verkettung zahlreicher Einzelvorgänge wie etwa die Prüfung der Bonität und des Kontostands.

2. Jeder semantische BE-Typ wird durch einen *syntaktischen BE-Typ* genauer spezifiziert. Dieser legt die Menge der Nachrichten mittels *semantischer Nachrichtentypen* fest, welche im Rahmen dieses Vorgangs ausgetauscht werden können.

3. Jeder semantische Nachrichtentyp des syntaktischen BE-Typs wird durch den *syntaktischen Nachrichtentyp* strukturell festgelegt. Dieser legt Details wie die Aufrufsemantik (synchron, asynchron, entkoppelt [disconnected]), Kommunikationsmodelltyp (wie Events, Methodenaufrufe) sowie Operationstypen (Signatur im Falle von Methodenaufrufen) fest.

Bemerkenswert an diesem System ist vor allem, dass die Typbeschreibung eine Verhaltensspezifikation in Form endlicher Automaten in allen drei Ebenen unterstützt. Diese endlichen Automaten entsprechen in ihrer Aufgabe im Wesentlichen dem Transitionstyp des Hybridtypsystems, sind aber im Unterschied hierzu den semantischen Eigenschaften zugeordnet. Die im Hybridtypsystem verfügbare *Selbstannotation* kann mit der semantischen Beschreibung auf der Komponentenebene verglichen werden, da sie Aussagen über die allgemeine Anwendbarkeit oder die *Anwendungsdomäne* liefert.

Durch die Beschreibung auf den unterscheidlichen Ebenen ist die Feststellung der Kompatibilität auf verschiedenen Abstraktionsstufen möglich, was dem Anwendungsentwickler entgegenkommt, da er beim Entwurf, der idealerweise die Komposition bestehender Komponenten beinhalten sollte, in einer frühen Stufe nicht mit Implementations- und Kommunikationsdetails konfrontiert wird. Andererseits ist das Typsystem präzise genug, dass es mittels der Angaben zum Nachrichtenaustauschprotokoll die Interaktionskompatibilität prüfen kann. Ziel soll es sein (wie im Falle des Hybridtypsystems) zu erkennen, ob eine bestimmte Komponente eine erwartete Funktionalität aufweist, also einem Subtyp des erwarteten Typs entspringt.

Die Anwendung dieses Typsystems soll einer automatischen Anwendungsgenerierung zugute kommen, welche nach Präsentation einer Anwendungs-Anforderungsspezifikation in Form einer Dokumenttypdefinition (DTD) die zugehörigen Komponententypen in ihrer XML-Repräsentation auf Übereinstimmung prüft. Der Generator ist nach eventuellen weiteren Vorgaben in der Lage, eine Anwendung durch Zusammenschließen miteinander kompatibler Komponenten zu erstellen.

5.9 Zusammenfassung

Dieses Kapitel stellte das Kernstück dieser Arbeit, das Hybridtypsystem, vor. Zunächst wurde betrachtet, in welcher Weise Agenten oder ähnliche Objekte typisiert werden können. Dabei stellte sich heraus, dass gewisse Bedingungen an die Interaktionen zwischen den zu typisierenden Objekten gestellt werden sollten, damit Implementationsdetails wirksam verborgen werden können. So ist es sinnvoll, die Kommunikation homogen zu gestalten, also beispielsweise durch Nachrichtenaustausch zwischen allen Komponenten.

Ein Typsystem sollte Grundbedingungen erfüllen, gerade um im Umfeld autonomer, mobiler Softwareagenten sinnvoll eingesetzt werden zu können. Durch die Vereinigung semantischer, syntaktischer und protokollbezogener Informationen kann eine Charakterisierung vorgenommen werden, welche geeignet ist, Agenten zu typisieren. Eine Interpretation der Protokollbeschreibung nach der *Trace*-Semantik ist nicht zweckdienlich für mobile Agenten, da sie dem Autonomiegedanken zuwiderläuft. Das aus den Hybridtypen gebildete System unterliegt einer Reihe von Regeln und weist Eigenschaften auf, die es als Typsystem für die Typisierung mobiler, autonomer Softwareagenten qualifizieren.

Ein Blick auf verwandte Arbeiten zeigt, dass einige deren Konzepte Eingang in dieses Typsystem fanden. Insbesondere sind die Arbeiten von Hennessy zu erwähnen, welche die Problematik nichtdeterministischer Alternativen beleuchten sowie natürlich die Arbeiten von Nierstrasz, welche eine Basis für das beschriebene Typsystem liefern.

6 Integration des Typsystems in AMETAS

Das vorangegangene Kapitel lieferte die theoretischen Grundlagen zu dem in AMETAS zu integrierenden Typsystem. Zu diesem System gehören folgende Komponenten:

- Datenstruktur der Typbeschreibung
- Erzeugung der Typbeschreibung
- Algorithmus zum Vergleich zweier Beschreibungen
- Vermittlungsalgorithmus
- Infrastrukturelle Unterstützung
- Programmierschnittstelle

Das Basissystem soll nach Möglichkeit nicht in seiner Funktionalität geändert werden, sodass bisherige Anwendungen wie gewohnt weiterlaufen können. Die Programmierschnittstelle soll Details der Implementierung des Typsystems weitestgehend verbergen, dennoch alle Funktionalitäten zugänglich machen. Details sind in Anhang E zu finden.

6.1 Datenstrukturen

Eine Typbeschreibung, wie sie im Rahmen des Hybridtypsystems eingesetzt wird, soll einem Stellennutzer während seiner gesamten Laufzeit und auch vor seiner Laufzeit eindeutig zugeordnet sein. Es liegt also nahe, die Beschreibung an den Code zu „binden“. Sie innerhalb des Programmcodes unterzubringen, dürfte sich aber als nicht hilfreich erweisen, da der Typträger dann erst zur Ausführung gebracht werden muss, um die Typbeschreibung in Erfahrung zu bringen.

In AMETAS werden Stellennutzer in Form so genannter *Signed-PlaceUser-Container* (kurz SPU) aufbewahrt. Dies ermöglicht es, zum einen alle zum Stellennutzer gehörenden Klassen zur Verfügung zu haben, wenn dieser zur Ausführung kommt (und

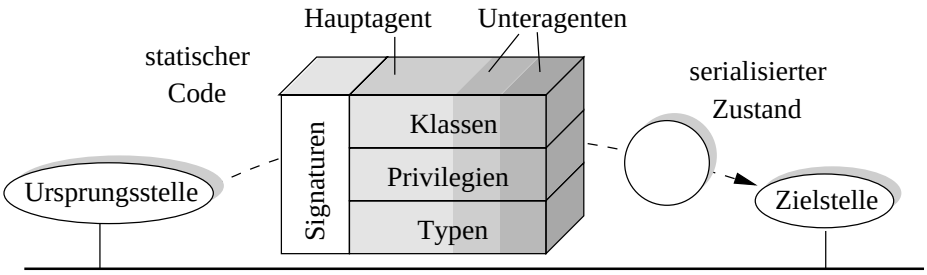

Abbildung 6.1: Signierter Stellennutzer-Container

damit ein mögliches Nachladen zu vermeiden), zum anderen die Integrität des Agenten zu verifizieren, indem über das gesamte Paket eine Signatur erstellt wird, welche bei der Ankunft überprüft werden kann. Eine solche Signatur kann wirksam gegen Angriffe auf Agenten eingesetzt werden [ZMG98].

Diese SPUs bieten sich dementsprechend sehr gut an, die neu hinzugekommene Typinformation als weiteren Bestandteil in sich zu tragen, wie Abbildung 6.1 illustriert. Da bei der Migration von Agenten stets der SPU zu übertragen ist, liegt bei Eintreffen des Agenten zeitgleich seine Beschreibung vor. Das Basissystem muss nun diese Information extrahieren und der weiteren Verarbeitung zuleiten; der Agent kann dann wie gewohnt gestartet werden.

Die in einer Typbeschreibung beinhalteten Informationen müssen einem Anwender in textueller Form präsentiert werden, damit sie von ihm entworfen, interpretiert und modifiziert werden können. Abbildung 6.2 zeigt ein Beispiel einer Typbeschreibung. Deutlich lassen sich die drei Blöcke des syntaktischen, transitionellen und semantischen Teils des Typs identifizieren. Der syntaktische Block (markiert durch das Schlüsselwort *messages*) ist in zwei Unterblöcke aufgeteilt, welche die Eingangs- beziehungsweise Ausgangsnachrichten deklariert. Der transitionelle Block *(states)* definiert das Transitionssystem durch Auflistung der einzelnen Transitionen. Schließlich listet der semantische Block *(annotations)* alle auftauchenden Annotationen auf. Im Falle der Abbildung 6.2 besteht er lediglich aus der Selbstannotation.

Um Verbindungen zwischen den Teilen der Typbeschreibung zu realisieren, werden *Marken* eingesetzt. Die Marke *Angebot* wird in der ersten Transition verwendet, um die erste Nachricht im *out*-Block zu referenzieren. Zustände werden ebenfalls durch eigene Marken repräsentiert.

Die der textuellen Repräsentation einer Typbeschreibung zu Grunde liegende Syntax wird in Anhang A angeführt. Nachrichten werden in der textuellen Beschreibung so dargestellt, dass einer optionalen Marke (für die Referenzierung der Nachricht) eine Liste von Nachrichtenelementen zwischen geschweiften Klammern folgt. Jedes Nachrichtenelement wird durch den Namen der Klasse des Elements aufgeführt, wobei optional eine Marke vorangestellt werden kann.

Zustände werden implizit eingeführt: Jede Marke, die innerhalb einer Transitions-

```
messages {
   in {
      Anfrage: { Name:String };
      BegrAnfrage: { Name:String, Limit:Float };
      Kaufe: { Name:String, Preis:Float, Kdaten:String[ ] };
   }
   out {
      Angebot: { Preis:Float };
      JaNein: { Boolean };
      Handel: { Bestaetigung:String };
   }
}
states {
   initial =Anfrage> initial:Angebot;
   initial =BegrAnfrage> initial:JaNein;
   initial =Kaufe> initial:JaNein
                 + initial:JaNein,Handel;
}
annotations {
   self=CG:{
      [Dienst]->(”führt durch”)->[Handel] ->(von)->[Buch]
   };
}
```

Abbildung 6.2: Typbeschreibung in textueller Form

beschreibung einen Zustand repräsentiert, bewirkt die Generierung eines zugehörigen Zustands. Eine Transition wird so beschrieben, dass der links vom Gleichheitszeichen stehende Zustand bei Empfang der benannten Nachricht zu dem folgenden Zustand übergeht, wobei hinter einem Doppelpunkt eine oder mehrere Ausgangsnachrichten genannt werden können. Dieser rechte Teil der Transitionsbeschreibung kann hinter einem Pluszeichen wiederholt werden und beschreibt damit alternative Übergänge bei Empfang desselben Nachrichtentyps. Soll eine spontane Transition stattfinden, wird die Epsilon-Nachricht durch die reservierte Nachrichtenmarke *none* bezeichnet.

Annotationen können als Zeichenketten sowie als Konzeptgraphen definiert werden. In letzterer Form folgen sie der textuellen Repräsentation, wie sie in [Sow84] und [Pud97] Verwendung findet. Der Inhalt der Annotation wird zwischen geschweiften Klammern genannt; die Art der Annotation wird durch das Kürzel „CG“ (Konzeptgraphen) oder „S“ (Zeichenketten) bestimmt.

Die in der textuellen Repräsentation eingeführten Marken entsprechen keiner im Kapitel 5 eingeführten Typkomponente, sondern stellen die Beziehungen her, wel-

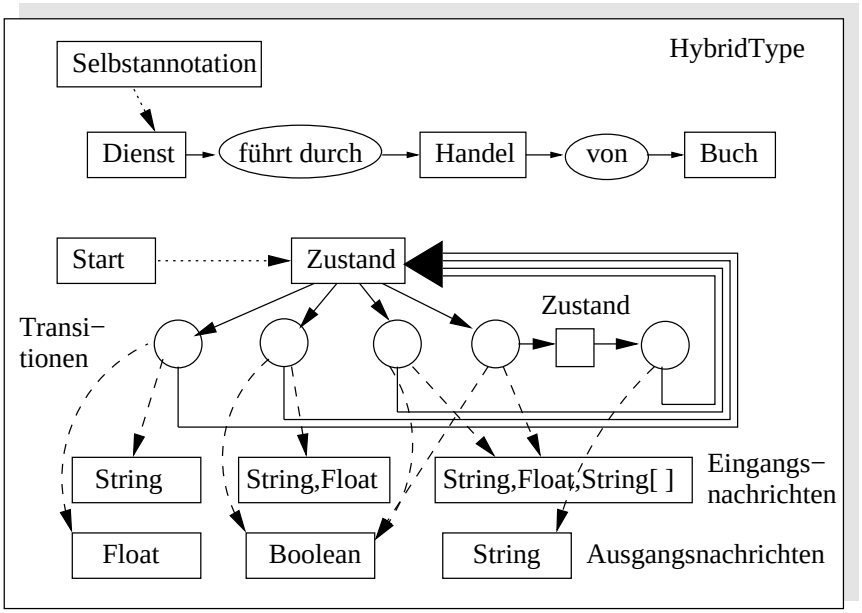

Abbildung 6.3: Typbeschreibung als Instanz

che in der theoretischen Betrachtung implizit vorhanden sind. Durch die Wahl intuitiver Namen (wie *Anfrage*) scheinen sie jedoch relevante Informationen hinzuzufügen. Dann müsste aber ein Vergleichsalgorithmus in der Lage sein, herauszufinden, ob in einem Vergleichstyp die Marke *Request* dieselbe Rolle spielt. Eine derartige Klassifizierung von Nachrichten oder anderer Typbeschreibungselemente ist alleinige Aufgabe der Annotationen, für welche es einen eigenen Vergleichsalgorithmus gibt, welcher in Abschnitt 6.6 genauer beschrieben wird.

Um einerseits solche impliziten Informationen durch Marken zu eliminieren, andererseits die Verarbeitung zu beschleunigen, wird die textuelle Repräsentation vor der Benutzung in eine spezielle Datenstruktur verwandelt. Abbildung 6.3 demonstriert die Struktur dieser internen Repräsentation für den in Abbildung 6.2 dargestellten Text. Die in der textuellen Repräsentation verwendeten Marken wie *JaNein* oder *Anfrage* sind in der internen Repräsentation nicht mehr vorhanden; lediglich die in den Konzeptgraphen verwendeten Zeichenketten sowie die Datentypbezeichner bleiben erhalten. Die interne Repräsentation wird in serialisierter Form als Typbeschreibung in den SPU eingefügt und kann so Agenten auf ihrer Reise begleiten. Anhang B stellt eine Liste der dem Typsystem angehörigen Klassen auf.

6.2 Typcompiler

Um die oben beschriebene Laufzeit-Datenstruktur zu generieren, wird ein Compiler verwendet. Dieser *Typcompiler* nimmt eine textuelle Repräsentation entgegen und pro-

duziert eine Instanz der Klasse *HybridType*, welche als Inhalt einer *AMETASType*-Instanz Verwendung finden kann.

Der Typcompiler liegt in Form einer Klasse *HybridTypeCompiler* vor und muss zur Benutzung instantiiert werden. Wird er als eigenständige Anwendung gestartet, kann er zum Test einer Typbeschreibung auf Korrektheit oder zur Erstellung der Beschreibung, die in den SPU importiert wird, verwendet werden. Der Typcompiler wurde auch im Hinblick auf die Verwendbarkeit zur Laufzeit des Agentensystems konzipiert; Einzelheiten hierzu findet man im Anhang E.

Zwei Merkmale der textuellen Beschreibung differieren strukturell von der Laufzeitrepräsentation: die Anfügung weiterer Alternativen bei Zustandsübergängen sowie die Möglichkeit, eine Kette von Ausgangsnachrichten anzugeben. Der Grund für diese nicht direkt repräsentierbaren textuellen Strukturen liegt darin, dass es äquivalente Formulierungen gibt, welche diese Strukturen ersetzen können. Der Compiler setzt die gegebenen Strukturen nötigenfalls in diese äquivalenten Strukturen um:

1. Es besteht kein Unterschied zwischen den Formulierungen *s1=n1>s2+s3* und *s1=n1>s2; s1=n1>s3*. Daher erzeugt der Compiler stets die rechte Formulierung, auch wenn ihm als Eingabe die linke Formulierung präsentiert wurde, was den Vorteil hat, dass die Transitionsdatenstruktur keine Liste der möglichen Endzustände führen muss.

2. Es besteht kein Unterschied zwischen den Formulierungen *s1=n1>s2:o1,o2* und *s1=n1>s3; s3=none>s2:o2*. Wird dem Compiler die erste Form vorgelegt, konstruiert er *implizite Zwischenzustände*, welche den Vorgang modellieren, dass eine Kette von Ausgaben in Folge einer bestimmten Eingabe erzeugt wird. Die Semantik der Transitionsbeschreibung gibt keine Auskunft darüber, wie der Sender intern die Abfolge der Ausgaben organisiert. Es ist damit prinzipiell korrekt anzunehmen, dass der Absender einen internen Zustandsübergang vollführt, bevor er die nächste Ausgabe vollzieht. Dies ist auf beliebig lange Ketten erweiterbar.

Die Vorteile einer solchen Handhabung liegen in der geringeren Komplexität der Datenstrukturen und damit auch des Vergleichsalgorithmus. Es besteht keine Notwendigkeit, äquivalente Formulierungen erkennen zu müssen. Der Typcompiler erlaubt es mithin, die leichter lesbare Formulierung zu verwenden, ohne sie gesondert behandeln zu müssen. Abbildung 6.3 illustriert die Anwendung der zweiten Regel an der vierten Transition.

Wird der Compiler auf der Kommandozeile aufgerufen, speichert er die Instanz der Typbeschreibung in einer Datei, welche dem SPU hinzugefügt werden kann. Wird er innerhalb des Agentensystems aufgerufen, so übergibt er die Typinstanz an den Aufrufer, ohne diese zu speichern. Genaueres hierzu ist in Abschnitt 6.7 zu finden.

6.3 Typkonformität

Der eigentliche Vergleich der Typbeschreibungen ist nicht innerhalb der Mediatoren, sondern innerhalb der den verwendeten Typen zugehörigen Klassen implementiert. Damit wird das Prinzip der Objektorientierung gewahrt, dass nämlich ein Objekt vermöge seiner Methoden die Verarbeitung seines Zustands definiert. Der Mediator ruft in allen Fällen lediglich ein

```
tc = typeGiven.compareTo(typeWanted, tcFail, ont);
```

auf; die zur Feststellung der Konformität notwendigen Operationen definieren die jeweiligen Typklassen. Die Konformität selbst ist vom Typ abhängig; es existiert jedoch eine Basisklasse *TypeConformance*, welche die Information tragen kann, ob der Vergleich eine vollständige Konformität ergab.

Der Vergleich von *stringbasierten Typen* (welche bislang in AMETAS Verwendung fanden) läuft im Wesentlichen auf einen Vergleich der Bezeichner hinaus, welche den Inhalt der Typinstanzen bilden. Es gibt hierbei zwei mögliche Ergebnisse:

Regel 6.1 Konformität stringbasierter Typen
Stringbasierte Typen lassen sich nur mit stringbasierten Typen vergleichen. Das Ergebnis des Vergleichs ist

- keine Übereinstimmung, wenn die zu Grunde liegenden Zeichenketten unterschiedlich sind;
- volle Übereinstimmung, wenn die zu Grunde liegenden Zeichenketten gleich sind.

Der Vergleich von *Hybridtypen* sollte hingegen genauere Informationen in Bezug auf die Konformität liefern; insbesondere ist man daran interessiert, *warum* ein Vergleich scheiterte. Die Konformitätsaussagen sind daher direkt vom verwendeten Typ abhängig und somit nicht in der Basisklasse *TypeConformance* festgeschrieben.

> Jedes in AMETAS einzusetzende Typsystem liefert eine spezifische Klasse zur Bestimmung der Typkonformität.

Der Hybridtypvergleich produziert neben der vollen Konformität über 20 weitere Aussagen, ob und inwiefern eine Nichtkonformität angenommen werden kann. Liefert beispielsweise

```
boolean bTrMis = tc.checkFor(HybridTypeConformance.TR_MISSING);
```

den Wert *true*, so ergab der Typvergleich, welcher das Konformitätsobjekt *tc* lieferte, dass der aktuelle Typ eine Transition des Vergleichstyps nicht beinhaltet. Es ist möglich, mehrere Konformitätsaussagen mittels Oder-Verknüpfung zu verbinden, da es sich um numerische Werte handelt; *checkFor* liefert dann *true*, wenn eine der verknüpften Aussagen zutrifft. Anhang D beschreibt die im Objekt *HybridTypeConformance* vorhandenen Felder vollständig.

Das Typkonformitätsobjekt hat im Allgemeinen eine doppelte Funktion:

1. Es gibt an, ob der Algorithmus Nichtkonformitäten festgestellt hat.
2. Es definiert die Vorgaben des Anfragers an den Vergleichsalgorithmus.

Diese Vorgaben sind essentiell für den folgenden Vergleich. Die Algorithmen, welche zur Durchführung des Vergleichs verwendet werden, können die Vorgaben als Entscheidungskriterien nutzen:

```
tc = new HybridTypeConformance(HybridTypeConformance.
                                ANN_MISMATCH);
mr = new AMETASMediationRequest(typ, tc, true);
```

So kann der Anfrager bestimmen, dass ihn eine Nichtkonformität in Bezug auf die Semantik nicht stört, solange der syntaktische und der transitionsbezogene Vergleich eine Konformität ergeben.

Zwar können die semantischen Angaben von besonderem Wert sein, um letztendlich Gewissheit über die Zuordnung der Nachrichten oder Zustände zu erhalten, aber gerade die Semantik unterliegt der Gefahr, trotz identischer Intension eine strukturell unterschiedliche Formulierung zu erhalten. Der Vergleich würde in diesem Falle wahrscheinlich nicht zum Erfolg führen. Tritt dieses Phänomen gehäuft auf, so kann sich der Anfrager genötigt sehen, die Semantik völlig auszublenden, bis ein Übereinkommen getroffen wird, wie bestimmte semantische Informationen zu formulieren sind.

Umgekehrt kann es sein, dass der Anfrager nichts Genaues von den Nachrichten und Protokollen des gesuchten Stellennutzers weiß, sehr wohl aber dessen Funktion kennt. Dann wird der Anfrager auf den syntaktischen und transitionsbezogenen Vergleich verzichten und *nur* auf die Semantik vertrauen. Informationen bezüglich der anderen beiden Typkomponenten kann der Anfrager aus dem Ergebnis extrahieren und daraus seine Kommunikation mit dem vermittelten Stellennutzer geeignet gestalten.

6.4 Test auf syntaktische Konformität

Üblicherweise wird bei einem Typvergleich ein *vollständiger* Vergleich, also einschließlich der Protokollbeschreibungen, durchgeführt. In bestimmten Fällen kann darauf verzichtet werden:

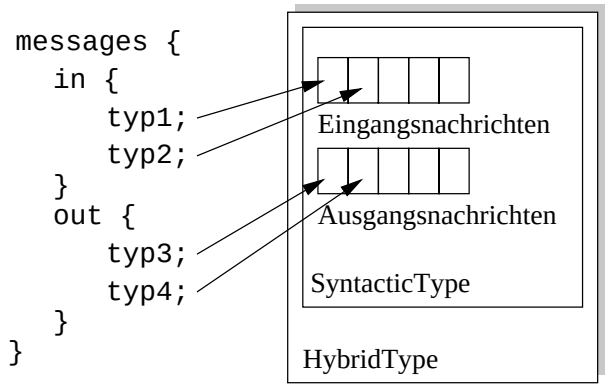

Abbildung 6.4: Syntaktischer Typ

- Das Protokoll ist unbekannt.
- Das Protokoll ist trivial, also lediglich Anfrage-Antwort-basiert.
- Das Protokoll ist nicht angegeben.

Es ist die Möglichkeit vorgesehen, in diesem Falle die *syntaktische Konformität* als maßgeblich für die Konformität der Typbeschreibungen zu bewerten. Falls vorhanden, können Annotationen die Auswertung unterstützen.

Der syntaktische Typ wird innerhalb des Hybridtypobjekts als aggregiertes Objekt repräsentiert, welches zwei Mengen von Nachrichtentypen definiert (Abbildung 6.4). Die Ordnung unter den Nachrichtentypen, welche durch die Speicherung in den beiden Feldern induziert wird, wird vom Vergleichsalgorithmus nicht beachtet. Die Nachrichtenlisten werden als ungeordnete Mengen betrachtet. Der Test auf Konformität eines syntaktischen Typs t_1 zu einem syntaktischen Typ t_2 verläuft folgendermaßen:

Eingangstest Alle Nachrichten in der Eingangsnachrichtenmenge von t_2 müssen in jener von t_1 typgleich oder als Superklasse auftauchen (Kontravarianz).

Ausgangstest Alle Nachrichten in der Ausgangsnachrichtenmenge von t_1 müssen in jener von t_2 typgleich oder als Superklasse auftauchen (Kovarianz).

Wird in einem der beiden Tests eine Nachricht im anderen Typ *zweimal* wiedergefunden, ohne dass eine klare Unterscheidung möglich ist, dann wird auf *Ambivalenz* geschlossen, also Mehrdeutigkeit. Dies kann zu Fehlurteilen in der Typüberprüfung führen, daher kann der Anwender vorgeben, keine Mehrdeutigkeiten zuzulassen. Eine Möglichkeit, solche Ambivalenzen zu vermeiden, ist der Einsatz von Konstantenannotationen.

Der Aufwand der Überprüfung der syntaktischen Konformität hängt von der Größe der Nachrichtenmengen ab. Da auf Mehrdeutigkeit geprüft und vorzeitig abgebrochen

wird, falls eine Nachricht nicht gefunden wurde, ist die Feststellung der vollen Konformität (abgesehen von Ko- und Kontravarianzen) am aufwändigsten.

Satz 6.2 Aufwand des syntaktischen Tests
Sind n_1^i, n_2^i, n_1^o, n_2^o die Größen der jeweiligen Eingangs- und Ausgangsnachrichtenmengen der beiden zu prüfenden Typen, so liegt die Komplexität des syntaktischen Tests im ungünstigsten Falle bei $O(n_1^i n_2^i + n_1^o n_2^o)$.

Dies liegt daran begründet, dass jede Eingangsnachricht des einen Typen paarweise mit jeder Eingangsnachricht des anderen verglichen wird; analog wird für die Ausgangsnachrichten verfahren. Der syntaktische Test geht somit von der Prämisse aus, dass es keine Abhängigkeiten der Nachrichten untereinander, weder in Ein- noch Ausgabe, gibt. Jede Eingangsnachricht muss jederzeit akzeptiert werden; jede Ausgangsnachricht kann jederzeit gesendet werden. Dies bedeutet insbesondere, dass *jedes Protokoll akzeptabel* ist, da jede Abfolge von Nachrichten auftreten kann.

Regel 6.3 Rein syntaktischer Test
Weist der Vergleichstyp (als potenzieller Subtyp) kein Protokoll auf, so wird nur auf syntaktische Konformität geprüft. Möchte der Anwender auf die Protokollprüfung verzichten, muss er *TR_MISSING* im Konformitätsobjekt setzen.

6.5 Test auf Transitionskonformität

Der Vergleich der Transitionstypen ist der bei weitem komplexeste Vorgang während des gesamten Typvergleichs. Ist der Transitionstyp vorhanden, so wird der Vergleich der Nachrichten von den durch das Protokoll definierten Interaktionen bestimmt. Es wäre unsinnig, Nachrichten miteinander zu vergleichen, die in einem erwarteten Kommunikationsvorgang nie auftauchen. Der Transitionstyp wird durch das Feld *TransitionType* des Hybridtyps repräsentiert; er beinhaltet eine Referenz auf den Startzustand. Abbildung 6.5 stellt dies schematisch dar. Annotationen, die mitsamt den Nachrichten sowie der Zustände auftauchen, werden im Laufe der Analyse der Zustandsübergänge verglichen.

Die Implementierung des Transitionstypvergleichs folgt dem Algorithmus 1 auf Seite 157. Die Klassen *StateSet*, *State* und *InteractionType* (siehe auch Anhang B) beinhalten den wesentlichen Teil dieser Implementierung. Der Algorithmus wird durch den Aufruf der Methode

```
HybridTypeConformance htc =
    sx.checkSubstitutability(StateSet sy, ...)
```

auf *sx*, einem Objekt der Klasse *StateSet*, gestartet. Der gesamte Vergleich wird gestartet, indem für *sx* die aus dem Anfangszustand des möglichen Subtyps bestehende Zu-

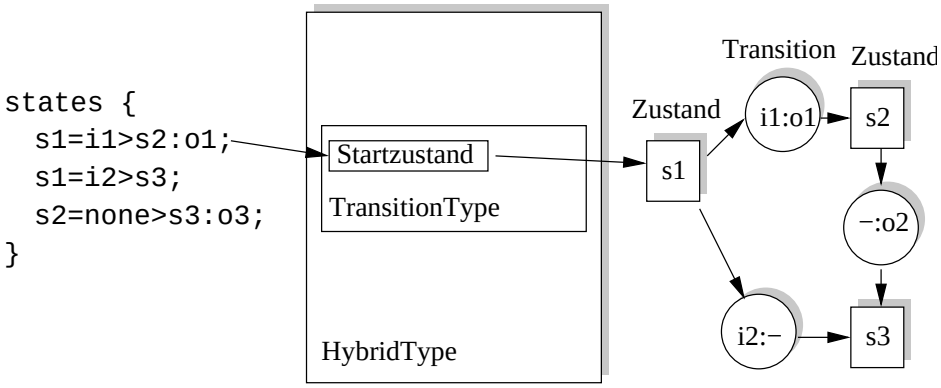

Abbildung 6.5: Transitionstyp

standsmenge gewählt wird; analog besteht die Zustandsmenge *sy* aus dem Startzustand des Typs. Als Ergebnis erhält man ein Objekt der Klasse *HybridTypeConformance*, welches Angaben über die Konformität trifft (siehe Anhang D).

6.5.1 Vorbereitung

Zunächst ist es notwendig, die während einer Kommunikation möglichen Interaktionen zu bestimmen. Eine Interaktion definiert sich aus einer Eingangsnachricht mit einer eventuell folgenden Kette von Ausgaben. Im konkreten Falle kann es vorkommen, dass ein Agent sofort mit dem Senden von Nachrichten beginnt, ohne zuvor eine auslösende Nachricht erhalten zu haben. Diesem Falle trägt die Einfügung eines *Pseudostartzustands* Rechnung.

Schritt 1
Ergänzung des Pseudostartzustands

Da jeder Typ diesen Zustand vorangestellt bekommt, ist diese Operation für das endgültige Urteil unerheblich. Der wesentliche Vorteil besteht darin, dass nunmehr vermöge einer *Pseudoeingangsnachricht* eine vollständige Interaktion vorliegt.[1] Der Pseudostartzustand und die Pseudoeingangsnachricht kann mit dem *Instantiierungsvorgang* (oder nach Migration dem *Deserialisierungsvorgang*) identifiziert werden (siehe auch Abschnitt 5.5.6).

Während des Vergleichs werden Zustände der Graphen in Mengen gleichwertiger Zustände zusammengefasst. Dieser Vorgang basiert auf der Strategie der Konvertierung eines nichtdeterministischen endlichen Automaten (NEA) in einen deterministischen endlichen Automaten (DEA). Um den Nichtdeterminismus zu eliminieren, werden all

[1] Diese Pseudonachricht kann – weil sie in gleicher Weise jedem Typ vorangestellt wird – einen beliebigen Typ aufweisen; in der vorliegenden Implementierung wurde die *migration*-Nachricht gewählt.

jene Zustände in einer Menge zusammengefasst, die durch einen der möglichen Übergänge erreicht werden können.

Sender- und Empfängerannotationen *(S/E-Annotationen)* bedürfen einer Vorverarbeitung. Abschnitt 5.5.7 erklärte bereits, wie Interaktionen mit S/E-Annotationen und mit Epsilon-Transitionen zu vergleichen sind. Jedoch ändern sich die Interaktionen beträchtlich, wenn S/E-Annotationen missachtet werden: Bei Missachtung der Senderannotation der ersten möglichen Interaktion kann der (eigentliche) Startzustand stabil sein, bei Beachtung aber instabil (wenn beispielsweise der Sender als *other* angenommen wird). Aus diesem Grunde führt diese Implementation *gesonderte* Durchläufe für die möglichen Fälle durch.

Schritt 2
Bestimmung der Transitionen mit Sender- oder Empfängerannotationen

Dies kann die Laufzeit im ungünstigsten Falle um $O(2^n)$ erhöhen (mit n als Anzahl der auftretenden S/E-Annotationen), da alle Kombinationen von Be- und Missachtung aller Annotationen durchlaufen werden. Eine effizientere Lösung dieses Problems ist ohne weit gehende Eingriffe in den Vergleichsalgorithmus nicht denkbar: Vor der Analyse ist nicht klar, welche Nachricht des Subtyps mit welcher Nachricht des Supertyps korrespondiert. Folglich ist es nicht möglich, vor der Analyse zu bestimmen, welche Annotationen zu aktivieren und welche zu deaktivieren sind, um eine Übereinstimmung mit dem Vergleichstyp zu erreichen. Die hierfür notwendigen Informationen ergeben sich erst während des Typvergleichs.

Sind alle Transitionen mit S/E-Annotationen gefunden, kann der Algorithmus für jede Kombination aktivierter und deaktivierter S/E-Annotation durchlaufen werden. Begonnen wird mit der Aktivierung aller S/E-Annotationen.

Schritt 3
Aktivierung der S/E-Annotationen und Durchführung der Analyse

Wird keine Übereinstimmung gefunden, so werden systematisch Annotationen deaktiviert und die Analyse wiederholt. Im positiven Falle wird der Vergleich beendet und das Urteil an die aufrufende Methode *Type.getConformanceTo* übergeben, welche das Ergebnis mit dem Urteil aus dem Vergleich des semantischen Typs kombiniert.

6.5.2 Bestimmung der Interaktionen

Grundlage des Vergleichs der Ersetzbarkeit von Zustandsmengen sind die von ihnen ausgehenden Interaktionen. Die Menge der von Zuständen der Menge *sx* ausgehenden Interaktionen wird mit $int(sx)$ bezeichnet (siehe Abschnitt 5.6.4). Um die Frage nach der Ersetzbarkeit der Zustandsmengen zu beantworten, sind nach dieser Definition die *int*-Mengen der jeweiligen Zustände im Einzelnen und der Zustandsmengen insgesamt

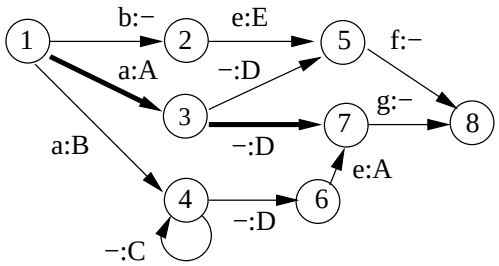

Abbildung 6.6: Transitionsgraph und Interaktion

zu vergleichen.

Die prinzipielle Idee hinter der Bestimmung der Interaktionstypen in Bezug auf einen Zustand ist, herauszufinden, welche Abfolgen von Transitionen während der Kommunikation auftreten können. Die *int*-Menge eines Zustands wird durch den Aufruf von *State.int()* bestimmt und in einem temporären Feld des Zustands aufbewahrt. Aufgrund der Abhängigkeit von den aktivierten S/E-Annotationen kann diese Berechnung nicht vorweggenommen werden. Der Algorithmus lässt sich aus den Zustandsmenge *sx* und *sy*, welche als Parameter der Methode übergeben wurden, die einzelnen Elemente aufzählen und führt dann jeweils die Bestimmung der *int*-Menge durch. Als Ergebnis wird ein Feld von Interaktionstypen geliefert.

Abbildung 6.6 zeigt einen möglichen Transitionsgraphen und eine darin ausgezeichnete Interaktion. Die Interaktion endet erst im Zustand 7, da der Zustand 3 nicht stabil ist. Der Diensttyp, der dadurch implizit definiert wird, sei mit *dienst(a):AD* bezeichnet, wodurch angedeutet wird, dass dieser Dienst als Eingabeparameter ein Datum des Typs *a* erwartet, als Ausgabe dann eine Konkatenation von Instanzen der Datentypen *A* und *D* bietet. Anders als die grafische Darstellung erlaubt diese Beschreibung keine Aussagen über Zustandswechsel. Aufgrund der Endlichkeit des Graphen ist die maximale Anzahl von Interaktionen endlich. Die Liste aller Interaktionen beschreibt dann genau die Menge aller von diesem Zustand aus angebotenen Interaktionstypen. Aufgabe der Methode *int* muss also sein, den Graphen geeignet zu traversieren, um alle Interaktionstypen zu erfassen. Für diese Traversierung wird eine *Tiefensuche* (*Depth-first search, DFS*) eingesetzt, welche in der durch *int* aufgerufenen Methode *getInteractionTypes* implementiert ist.

Voraussetzung für die erfolgreiche Traversierung ist, dass es mindestens eine Transition gibt, deren Eingabe von null verschieden ist und die nicht ignoriert werden kann. Der Algorithmus sorgt dafür, indem er den Ausgabetransitionen (welche keine Eingabe erwarten) bis zu einem stabilen Zustand folgt. Zunächst ist es dann möglich, diese erste Transition zu durchlaufen und dadurch in den Zustand *A* zu gelangen. Sind die Zustände *B, C, D...* Folgezustände des Zustands *A,* welche jeweils über die Transitionen *b, c, d...* ohne weitere Eingangsnachricht erreicht werden, so

- sind *b, c, d...* *Epsilon-* oder *Fremdnachrichten-Transitionen* (so genannte *E/O-*

Transitionen für *epsilon/other*) in Bezug auf die Eingangsnachricht;

- ist die Menge der Interaktionen die Vereinigung der um *b*, *c*, *d*... ergänzten Interaktionsmengen der Folgezustände und
- gehört jede Schleife, die von diesem Zustand ausgeht und einen bereits besuchten Zustand erreicht, in die Menge der Interaktionen sowie zu jeder der um *b*, *c*, *d*... ergänzten Interaktionen.

Zu beachten ist der Fall, wenn dieselbe Eingangsnachricht dazu führt, dass eine Kette von Zustandsübergängen stattfindet, die zwar nicht notwendigerweise gleich ist, aber im gleichen Zustand endet. Dann nämlich ist es für die weitere Verarbeitung unerheblich, welcher Pfad beschritten wurde; jedoch ist damit zu rechnen, dass Unterschiede in der Ausgabe zu finden sind. Es ist daher nicht sinnvoll, diese zwei Pfade als zwei *verschiedene* Interaktionen aufzufassen, wenn es keine Möglichkeit gibt, über einen dieser Pfade zu einem explizit anderen Zustand zu kommen, sondern festzulegen, dass die Menge der unterschiedlichen Pfade nunmehr gerade die Menge aller möglichen Ausgaben beschreibt, die nach Empfang der Nachricht auftreten können. Betrachtet man den Graphen aus Abbildung 6.6, so sollte hiermit klar sein, warum der Übergang zu Zustand 5 nicht ebenfalls zu diesem Diensttyp hinzugerechnet wird.

Regel 6.4 Komplexe Interaktionen
Zwei von einem Zustand ausgehende Pfade von Transitionen, deren initiale Transitionen konform sind und die nach Durchlaufen aller E/O-Transitionen denselben Endzustand erreichen, sind Teil einer einzigen Interaktion.

Ein weiteres Problem ergibt sich aus der Tatsache, dass die Ausgaben durch beliebige Strukturen im Graphen repräsentiert sein können. Insbesondere heißt dies, dass Ausgaben auch durch *Schleifen* entstehen können. Dieses Verhalten kann in Abbildung 6.6 aufgefunden werden. So könnte der Agent nach Empfang der Nachricht *a* statt wie vorhin ein *A* nun ein *B* liefern, was den Empfänger in Kenntnis setzt, dass nun ein anderer Pfad beschritten wird. Tatsächlich erreicht der Agent jetzt Zustand 4 und könnte in diesem bei fortgesetzter Aussendung von *C*-Nachrichten verweilen, bis er das Fortschreiten durch Aussenden einer *D*-Nachricht ankündigt. Der Zustand 6 ist wieder *stabil*, er erfordert eine erneute Eingangsnachricht. Wir erhalten zwei Diensttypen:

$dienst1(a):AC^*$

$dienst2(a):AC^*D$

Der Stern steht hierbei wie gewohnt für den Monoiden über dem Zeichen *C*, also für beliebig lange *C*-Ketten oder auch dem leeren Wort. Man beachte, dass hier explizit zwei verschiedene Diensttypen genannt sind. Der unter *dienst2* beschriebene Diensttyp umfasst Nachrichtenketten, welche mit Nachrichten des Typs *D* enden, der unter *dienst1* beschriebene jedoch nicht.

Die Liste von Ausgaben ist lediglich eine Liste *möglicher*, aber nicht zugesicherter Ausgaben. Ein Agent, welcher den *dienst2*-Typ in seiner Typbeschreibung beinhaltet, darf nur dann als kompatibel vermittelt werden, wenn der Nachfrager die *C*-Schleife ebenfalls erwartet. Umgekehrt darf der Nachfrager stets die Schleife erwarten, auch wenn der Agent sie nicht anbietet.

Regel 6.5 Ausgabealternativen
Tritt im Verlaufe einer Interaktion ein Ausgabealternative auf, welche nach eventuellem Durchlaufen weiterer Zustände einen früheren Zustand derselben Interaktion erreicht, so ist diese Ausgabealternative Teil dieser Interaktion.

Im Allgemeinen können weitere Zustände zwischen dem Abzweigen und der Rückkehr auf einen bereits besuchten Zustand liegen. Auch in diesem Falle muss der Nachfrager sich stets auf den Durchlauf dieser Schleife einstellen. Das erneute Verzweigen innerhalb einer solchen Schleife aus der Schleife heraus kann aber wiederum leicht entdeckt werden. Wie oben schon erwähnt wird in diesem Algorithmus eine Tiefensuche durchgeführt, mit deren Hilfe Zykel zuverlässig entdeckt werden können.

Aus dem obigen Satz folgt im Übrigen, dass *jede* Schleife, die von einer Interaktion abzweigt, Teil der Interaktion ist. Schleifen geben Anlass zur Definition einer nichtendenden Interaktion; diese kann aber ihrerseits eine kleinere Schleife beinhalten. Diese innere Schleife ist mit gleicher Begründung wie oben stets Teil dieser Interaktion.

Die Implementierung sammelt alle diese Schleifen, bildet eine Vereinigung und hängt diese Schleifen an alle beteiligten Interaktionen. Sobald alle Interaktionen von dem aktuellen Zustand aus gefunden wurden, werden diese als Feld an den Aufrufer gegeben. Schließlich werden die Interaktionen, welche im gleichen Endzustand landen, zu einer kombinierten Interaktion vereinigt. Das Ergebnis sind schließlich alle vom gegebenen Zustand ausgehenden Interaktionen, wobei je zwei Interaktionen in unterschiedlichen Zuständen enden. Nichtendende Interaktionen treten als Ergebnis in Form eines einzelnen Teilgraphen des von diesem Zustand ausgehenden Transitionsgraphen auf.

6.5.3 Prüfung der Ersetzbarkeit

Die Definition der beobachtbaren Ersetzbarkeit der Zustände in Abschnitt 5.6.4 ist rekursiv ausgelegt und liefert dadurch eine Vorlage für die Implementierung eines Vergleichsalgorithmus. Dieser wurde unter Zuhilfenahme der in Anhang B dargestellten Klassen implementiert. Im Wesentlichen basiert er auf Nierstrasz' Algorithmus in [Nie95], wobei dieser Algorithmus doppelt zum Einsatz kommt, da neben den Eingangsnachrichten auch die Ausgangsnachrichten zu vergleichen sind.

Der Aufruf von *checkSubstitutability* klärt vermöge des zurückgegebenen Konformitätsobjekts, ob die Zustandsmenge *sx* des Subtyps die Zustandsmenge *sy* des Supertyps ersetzen kann, das heißt, ob es möglich ist zu entscheiden, ob die weiteren Interak-

tionen ihren Ursprung in *sx* oder in *sy* haben oder ob im Gegenteil die Zustandsmenge des Subtyps an die Stelle der Zustände des Supertyps treten kann.

Kern des Vergleichs ist die Feststellung der Anfrage-Ersetzbarkeit, übertragen auf Interaktionstypen. Es ist zunächst erforderlich, diese möglichen *impliziten Diensttypen* festzustellen. Wie in [Nie95] wird hier ein rekursiver Ansatz verfolgt. Die im Verlaufe des Vergleichs festgestellten Diskrepanzen werden in einem Typkonformitätsobjekt gesammelt und an vorangegangene Aufrufe zurückgegeben, sodass diese Diskrepanzen bis zum initialen Aufruf propagiert werden. Anhand deren kann der Aufrufer entscheiden, ob über die von ihm vorgegebenen Ausschlusskriterien für die Konformität hinaus weitere Diskrepanzen auftreten, die er tolerieren kann oder nicht.

Algorithmus 1 Bestimmung der beobachtbaren Ersetzbarkeit

checkSubstitutability(X,Y):

01 Prüfe, ob (X,Y) bereits getestet wurde. Wenn ja, liefere *null*; sonst füge das Paar in die Tabelle ein.
02 Bestimme die Mengen $int(x)$ und $int(y)$ für jedes $x \in X$ und $y \in Y$.
03 Für jeden Zustand x in X
04 suche einen Zustand y in Y, sodass alle Eingangsnachrichten in y auch in x akzeptiert werden. Wird dieser nicht gefunden, brich ab mit *NEGATIV*.
05 Markiere alle jene Interaktionen in x als *relevant*, wenn es eine entsprechende Eingangsnachricht in einem y-Zustand gab.
06 Wenn alle Eingangsnachrichten von y auch in x auftauchen, dann prüfe nach, ob alle relevanten Interaktionen von x in y auftauchen. Wenn nicht, verwirf die Wahl von y und fahre bei 04 fort. Wenn kein y zu finden ist, brich ab mit *NEGATIV*.
07 Erstelle die Menge R_Y. Bilde dazu die Vereinigung aller Interaktionen von allen Zuständen in Y, die in Schritt 06 positiv getestet wurden.
08 Für jede Interaktion r in R_Y
09 Sammele in Y' alle Typzustände, die von einem Zustand y in Y_R durch Interaktionen erreicht werden, die ein Subtyp der Interaktion r sind.
10 Sammele in X' alle Subtypzustände, die von einem Zustand x in X_R durch Interaktionen erreicht werden, die ein Subtyp der Interaktion r sind.
11 Führe die Rekursion aus: *ret = checkSubstitutability(X′, Y′)*. Wenn *ret* *NEGATIV* ist, gib *NEGATIV* aus. Ignoriere *ret=null*.
12 Gib *POSITIV* aus.

Entsprechend notwendige Initialisierungen sind im abgebildeten Algorithmus nicht vollständig aufgezählt. Wichtig sind folgende Punkte:

- Erreicht der Algorithmus Schritt 07, dann gibt es für jedes x in X mindestens ein y in Y, sodass keine im Zustand y eingehende Nachricht in x zum Fehlschlag führt. Ferner bietet x keine nichtdeterministische Alternative an, die in y nicht vorgesehen ist.

- Schritt 09 bringt das Typ-Transitionssystem voran, indem alle Interaktionen, die zu einer gewählten Interaktion Subtypen sind, zugleich ausgelöst werden. Man beachte die Definition der Subtyprelation von Interaktionen: Die Kontravarianz bewirkt, dass die umgekehrte Subtypbeziehung bei den Eingangsnachrichten besteht. Wird ein *int*-Typ als Eingabe geschickt, so löst dies eventuell die Interaktionen mit Eingangstyp *long* aus, nicht jene mit Eingangsnachrichtentyp *short*.

- Schritt 10 bringt gleichermaßen das Subtyp-Transitionssystem voran, indem alle Interaktionen, die zur gewählten Interaktion Subtypen sind, ausgelöst werden.

- Die Rekursion spiegelt die rekursive Definition der beobachtbaren Ersetzbarkeit wider.

- Die Rekursion bricht ab, wenn ein Paar von Zustandsmengen bereits verglichen wurde. Da diese Paare über den gesamten Ablauf gespeichert werden, ist die maximale Anzahl von Durchführungen durch das Produkt der Größe der Potenzmengen der Zustandsmengen beider Systeme nach oben beschränkt.

Dieser Algorithmus bestimmt, ob zwei Zustandsmengen beobachtbar ersetzbar sind. Mehrelementige Zustandsmengen entstehen offenbar durch nichtdeterministische Alternativen. Ist der Transitionsgraph vollständig deterministisch, dann sind diese Mengen stets einelementig, da in diesem Falle in Schritt 09 und Schritt 10 nur genau eine Interaktion passend ist.

Essentiell für jeden Algorithmus ist die Angabe der Terminierungsbedingung, da sonst eine der Grundbedingungen der Definition eines Algorithmus (*Terminierung in endlicher Zeit*) verletzt sein könnte. Im Falle des Vergleich der Zustandsmengen ist allerdings klar, dass bei Analysen derselben Zustandsmengen-Paare stets dasselbe Ergebnis zu erwarten ist, nämlich dass die Subtyp-Zustandsmenge ihren Gegenpart des Supertyps ersetzen kann oder nicht. Da die Menge der Zustände bei beiden zu vergleichenden Graphen endlich ist (endliche Automaten!), ist die Menge aller Zustandsmengen ebenfalls endlich. Die maximale Anzahl von Durchgängen des Algorithmus errechnet sich somit aus allen Kombinationen aller Zustandsmengen. Um feststellen zu können, ob ein Paar von Zustandsmengen bereits analysiert wurde, werden die Zustandsmengenpaare in eine Tabelle eingetragen, welche eine Instanz der Klasse *StateSetTable* darstellt.

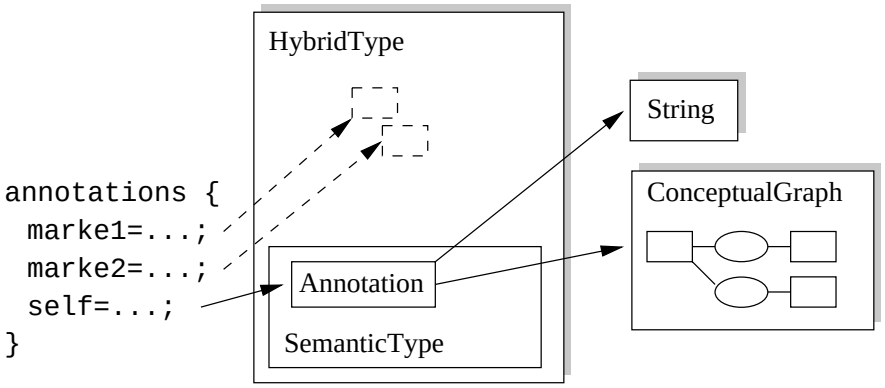

Abbildung 6.7: Semantischer Typ

Folgerung 6.6 Terminierung des Verfahrens
Der Vergleichsalgorithmus terminiert nach spätestens $O(2^{|X|\cdot|Y|})$ Schritten.

Hierbei seien X und Y die Mengen der Zustände der endlichen Automaten des Subtyps beziehungsweise des Supertyps. Leider offenbart sich bereits an dieser Abschätzung, dass der gesamte Algorithmus nicht *effizient laufen kann*, also eine exponentielle Laufzeit zu erwarten ist. Der in [Nie95] hergeleitete Algorithmus versucht, möglichst viele Zwischenergebnisse bereitzuhalten, die im Laufe der Analyse anfallen, sodass bereits frühzeitig entschieden werden kann, ob die weitere Analyse fortgesetzt werden muss oder nicht. Einige dieser Optimierungen können im vorliegenden erweiterten Algorithmus jedoch nicht übernommen werden, da sie den Zusammenhang zwischen Eingangsnachricht und Diensttyp implizit voraussetzen, der im Originalalgorithmus angenommen wurde.

6.6 Test auf semantische Konformität

Der semantische Test ist impliziter Bestandteil der beiden anderen Prüfungen auf Syntax- und Transitionskonformität; er tritt auf, wenn Annotationen miteinander verglichen werden sollen. Es gibt allerdings auch einen expliziten, eigenen *semantischen Test*, welcher die so genannte Selbstannotation der Typen miteinander vergleicht.

6.6.1 Konzeptgraphen und Stringdarstellungen

Semantische Informationen sind im Hybridtypsystem stets in Form von Annotationen anzutreffen. Wie bereits in Abschnitt 5.3 beschrieben, können semantische Informationen in Form von Zeichenketten oder Konzeptgraphen angeführt werden. Abbildung 6.7 ordnet den semantischen Typ in das Konzept ein. Die Prüfung des semantischen Typs erfolgt nach der Prüfung von Transitions- und syntaktischem Typ. Sie setzt natür-

lich voraus, dass die Art der Annotation – String oder Konzeptgraph – in beiden Fällen gleich ist. Der Test der Selbstannotation entspricht fast genau dem Test jeder anderen Annotation, außer dass die Werte *ANN_MISMATCH* und *ANN_VALUE_MISMATCH* im Ergebnis durch den Wert *ANN_SELF_MISMATCH* ersetzt werden, der demzufolge als Kriterium zur Konformität gesondert gewählt oder deaktiviert werden kann.

Stringannotationen

Diese Art von Annotationen bietet eine vergleichsweise einfache Art der Beschreibung an, verfügt jedoch über ein Strukturmerkmal: Jede Zeichenkette einer Stringannotation hat die Form

Zeichenkette[:Zeichenkette]

mit einem optionalen Teil hinter einem Doppelpunkt, welcher einen *Wert* für die durch den ersten Teil bezeichnete Entität darstellt, also wie *ClassName:MyAgent.* Stimmen die Bezeichner nicht überein, wird als Konformität ein *ANN_MISMATCH* geliefert. Anderenfalls bestimmt der Wert die Konformität. Ist er ungleich, erhält man einen *ANN_VALUE_MISMATCH*. Dieses Ergebnis führt in der Standardeinstellung zu einer Ablehnung der Konformität.

Konzeptgraphen

Im Zusammenhang mit diesem Typsystem möge unter dem Begriff Konzeptgraph immer ein Konzeptbaum verstanden werden. Die Prüfung der Konformität eines Konzeptgraphen g_1 zu einem Konzeptgraphen g_2 besteht in einem rekursiven Verfahren, während dessen die beiden Graphen vollständig traversiert werden. Der Algorithmus lässt sich folgendermaßen skizzieren:

1. Prüfe das (semantische) Enthaltensein des Graphen unter dem Wurzel-Konzeptknoten $k^0(g_2)$ zu jenem unter $k^0(g_1)$. In diesem Falle ist der Graph g_1 eine Spezialisierung von g_2.
2. Ein Graph $g_1 = sg(k_1)$ ist (semantisch) in einem Graphen $g_2 = sg(k_2)$ enthalten, wenn
   a) der Konzeptname $n_1 = n(k^0(g_1))$ ein Synonym oder Hyponym des Konzeptnamens $n_2 = n(k^0(g_2))$ darstellt;
   b) die Werte des Konzeptknotens $k^0(g_1)$ in der Menge der Werte des Konzeptknotens $k^0(g_2)$ enthalten sind, wenn diese nicht leer ist; und
   c) jeder Relationssubgraph $r_i(k^0(g_1))$ in der Menge der Relationssubgraphen von $k^0(g_2)$ (semantisch) enthalten ist (sofern vorhanden).

3. Ein Relationssubgraph r_1 ist (semantisch) in einem Relationsgraphen r_2 enthalten, wenn

   a) der Relationsname $n_1 = n(r_1)$ ein Synonym oder Hyponym des Relationsnamens $n_2 = n(r_2)$ darstellt;

   b) die Stelligkeit der Relationen gleich ist (gleiche Anzahl von angehängten Konzeptgraphen) und

   c) jeder Konzeptgraph k_i von $r_1$in einem Konzeptgraphen k_j von r_2 (semantisch) enthalten ist.

Damit ist klar, dass der Algorithmus rekursiv aufgebaut ist. Er terminiert, weil es sich bei den verwendeten Datenstrukturen um Bäume handelt, sodass jeder Pfad immer in einem Knoten endet, von welchem keine Relationen mehr ausgehen. Auch dieser Vergleich kann recht aufwändig werden, was sich aus der Notwendigkeit ergibt, die angehängten Relationssubgraphen paarweise miteinander zu vergleichen; Gleiches gilt für die angehängten Konzeptgraphen.

6.6.2 Ontologien

Konzeptgraphen sind von der Wahl der Konzept- und Relationsontologien abhängig. Im Hybridtypsystem werden Ontologien ähnlich wie Typen in zwei Repräsentationen verwendet: eine interne Repräsentation in Form miteinander verketteter Instanzen der Klasse *Ontology.Entry* sowie eine externe, textuelle Repräsentation, welche einfach lesbar und modifizierbar ist. Beispiele für Konzept- und Relationsontologien sind in Anhang C zu finden.

Es ist möglich, den Mediator des Hybridtypsystems bei seiner Initialisierung verschiedene Ontologiesätze laden zu lassen; diese werden durch die Wahl des Anwendungsprogrammierers innerhalb der *AMETASMediationRequest*-Instanz festgelegt. Im Hybridtypsystem sind stets zwei Ontologien – nämlich für Konzepte und Relationen – erforderlich; die Klasse *CGOntologySet* kapselt diese beiden Ontologien und erklärt sie zu einer *Wissensbasis*, indem *CGOntologySet* die Schnittstelle *KnowledgeBase* implementiert.

6.6.3 Gewichtung der Semantik

Die Formalisierung von semantischen Informationen lässt viele Freiräume. Es fällt bisweilen schwer, eine Information präzise genug zu beschreiben, sodass Anfragen erfolgreich beantwortet werden können. Dabei kann es durchaus passieren, dass ein Sachverhalt von zwei Personen völlig unterschiedlich gesehen wird. Demzufolge ist es unklug, der Semantik ähnlich starkes Gewicht beim Typvergleich zuzuordnen wie den recht strikten Vergleichen in Bezug auf Datentypen.

```
...
public boolean initialize() { return true; }
public abstract AMETASType
   typeForString(String sRep, String sMode);
public abstract AMETASMediationResult[]
   request(AMETASMediationRequest mreq);
public MediatorInfo getInfo() { ... }
...
```

Abbildung 6.8: Methoden des abstrakten Mediators

Der Anwendungsprogrammierer hat die Chance, die Gewichtung der Semantik festzulegen. Anhang D zeigt die möglichen Einstellungen des Konformitätsobjekts. Wie in Abschnitt 6.3 gezeigt wurde, wird das Typkonformitätsobjekt einerseits zur Angabe der Konformität nach dem Vergleich verwendet, es kann aber auch als Maske dienen, welche Kriterien für den Mediator beim Vergleich wesentlich oder unwesentlich sein sollen. Dazu muss lediglich ein mit einem entsprechenden Wert gesetztes Konformitätsobjekt der Typanfrage beigefügt werden.

6.7 Mediator

Die Aufgabe des Mediators besteht in der Vermittlung von Typen und Instanzen. Grundlage dieser Vermittlung ist der Vergleich von Typbeschreibungen, welche bislang in AMETAS Zeichenketten-basiert vorlagen und mit dem hier beschriebenen Typsystem wesentlich aussagekräftiger gestaltet werden können.

6.7.1 Allgemeiner Mediator

Ein Mediator muss von der Klasse *AMETASMediator* abgeleitet sein; er wird während der Initialisierung der Stelle, also erst während der Laufzeit, geladen. Das Agentensystem AMETAS nutzt hierbei die von Java vorgesehene Möglichkeit des späten Bindens und dynamischen Klassenladens. Vorteil ist, dass auf diesem Wege der Mediator sogar während des Betriebs ausgetauscht werden kann, ohne die Funktionstüchtigkeit des Gesamtsystems zu beeinträchtigen, was für den dauerhaften Betrieb von Agentenanwendungen notwendig ist.

Ein Mediator muss als Erweiterung der abstrakten Klasse eine Reihe von Methoden implementieren, je nachdem, welche Art von Typbeschreibungen von ihm verarbeitet werden sollen, wie in Abbildung 6.8 zu sehen ist. Die *initialize*-Methode kann überschrieben werden, wenn vor der Einsatztüchtigkeit des Mediators bestimmte Aktionen auszuführen sind, etwa das Laden von Dateien. Eine erfolgreiche Initialisierung wird durch Rückgabe von *true* mitgeteilt.

Die Methode *typeForString* ist notwendig, um eine textuelle Repräsentation der Typbeschreibung in die Laufzeitrepräsentation zu überführen. Die Semantik der Repräsentation wird durch den zweiten Parameter *sMode* bestimmt. Die Modi der Konvertierung sind Mediator-abhängig; so ist es denkbar, dass ein Mediator einen String bei Übergabe des Parameters *Plain* als Name, bei Übergabe von *PUID* als Stellennutzer-ID interpretiert. Die Festlegung solcher Modi liegt in der Verantwortung des Implementierers des Mediators; er muss die Anwendungsprogrammierer darüber in Kenntnis setzen. Ein Modus, den jedoch alle Mediatoren verstehen müssen, ist *DefaultServiceDescription*. Dieser dient dazu, Instanzen von *AMETASServiceDescription* zu konstruieren, welche neben dem Typ auch bestimmte Spezifika von Diensten kapseln (siehe auch Anhang E).

Ein anderer vordefinierter Modus ist *Default*, welcher je nach verwendetem Mediator eine entsprechende Semantik annimmt. Mit Hilfe der *getInfo*-Methode kann die Anwendung herausfinden, welcher Mediator an der aktuellen Stelle zum Einsatz kommt. Da die Mediation in AMETAS auf die lokale Stelle eingeschränkt ist, kann mit der Kenntnis des Mediators eine geeignete Stringrepräsentation einer Anfrage formuliert und dem Mediator zugeleitet werden.

Ein Mediator sollte dem Anfrager die Möglichkeit bieten, eine Liste aller aktuell laufenden Stellennutzer zu erhalten. Dies wird vom Anfrager durch Belegung des *AMETASMediationRequest*-Parameters durch *null* angedeutet. Da jeder Mediator ohnehin über eine Liste aller laufenden Stellennutzer verfügen muss, um den Vergleich der Beschreibungen durchführen zu können, ist es nahe liegend, diese Funktion dem Mediator zuzuordnen.

6.7.2 Der TrivialServiceMediator

Der so genannte *TrivialServiceMediator* ist Grundbestandteil des AMETAS-Systems. Sein Haupteinsatzgebiet wird durch seinen Namen angedeutet: Da den Agenten, welche an einer Stelle ankommen, die Stellennutzer-ID eines bestimmten Dienstes nicht bekannt ist, müssen sie – wenn sie keinen Multicast verwenden wollen (Abschnitt 4.1.3) – die Stellennutzer-ID mittels einer Beschreibung des gesuchten Dienstes in Erfahrung bringen. Auf eine Anfrage mittels *request* startet der Mediator seine Suche:

- Er holt sich die Liste aller zurzeit laufenden Stellennutzer aus dem Stellennutzerverzeichnis.
- Die übermittelte *AMETASType*-Instanz wird mit der zugehörigen Typinstanz verglichen.
- Wenn volle Konformität besteht, wird die gefundene Typinstanz einer Liste angefügt.
- Die vollständige Liste wird an den Aufrufer übermittelt.

Wichtig ist zu bemerken, dass dieser Mediator lediglich eine *Instanzvermittlung* auf *Stringbasis* unterstützt (Anfragetyp *INSTANCEBYNAME*). In der Regel wird daher die Typinstanz eine Instanz der Subklasse *AMETASServiceDescription* sein. Für andere Stellennutzer ist der Mediator in der aktuellen Version auch verwendbar, jedoch wird der Name des Stellennutzers, wie er vom SPU-Container vorgegeben wird, als Stringbeschreibung automatisch vorgesehen. Ein Anfrager muss diesen Namen exakt kennen. Die übrigen Anfragetypen *INSTANCEBYSPEC*, *TYPEBYNAME* und *TYPEBYSPEC* werden von diesem Mediator abgewiesen.

6.7.3 Der HybridTypeMediator

Kommt der einfache Vermittler noch ohne spezielle Initialisierung aus, so müssen beim Hybridtypmediator noch vor der Ausführung die für die Konzeptgraphenauswertung notwendigen Ontologien geladen werden. Die Klasse *Ontology* sorgt selbst für die Erzeugung der Laufzeitrepräsentation der Ontologie, wenn sie erfolgreich geladen werden konnte. Anschließend können Anfragen an den Mediator gerichtet werden.

Anfragen

Wie von der Basisklasse vorgegeben beantwortet der Hybridtypmediator Anfragen über den Aufruf der *request*-Methode. Um eine größtmögliche Kompatibilität mit bisherigen Anwendungen zu erreichen, simuliert dieser Mediator den trivialen Mediator, wenn er als Anfrage den Wert *null* übermittelt bekommt (Abfrage aller laufenden Stellennutzer) oder einen String-basierten Typ erhält, wobei die Anfrage vom Typ *INSTANCEBYNAME* ist.

Neben diesem Anfragetyp erkennt der Mediator auch die übrigen Typen, nämlich *INSTANCEBYSPEC*, *TYPEBYNAME* und *TYPEBYSPEC* an. Während *TYPEBYNAME* im Wesentlichen in der Verarbeitung der Spezifizierung *INSTANCEBYNAME* ähnelt (es werden anstelle der laufenden Stellennutzer die registrierten, gespeicherten Stellennutzer zum Vergleich herangezogen), sind die anderen beiden Modi die wesentlichen, in denen das Hybridtypsystem zum Einsatz kommt. Dabei werden wiederum alle laufenden oder alle registrierten Stellennutzer aus dem Stellennutzer-Verzeichnis geholt und der Reihe nach deren Typ mit dem gegebenen Typ verglichen. Als Ergebnis wird ein Feld von Ergebnissen *(AMETASMediationResult)* geliefert, welches im Falle von Instanzvermittlungen die Stellennutzer-ID und anderenfalls die SPU-Namen beinhaltet.

Typerzeugung

Die *typeForString*-Methode des Hybridtypmediators definiert eine Reihe von Modi zur Konstruktion eines Typs:

Default Es wird ein Konzeptgraph generiert, welcher den Stellennutzer anhand seines Klassennamens beschreibt, welcher als String übergeben wird. Dieser Konzept-

graph dient als Selbstannotation und macht damit den semantischen Typ aus. Die Typbeschreibung beinhaltet keine Angaben zum Transitions- oder syntaktischen Typ.

DefaultServiceDescription Dies konstruiert die traditionelle *AMETASServiceDescription*. Der übergebene String wird als Registrierungsname des Dienstes interpretiert; die übrigen Felder der Dienstbeschreibung werden mit Standardwerten belegt. Es wird ein Konzeptgraph konstruiert, welcher die Information über den Registrierungsnamen beinhaltet; der resultierende Typ ist *hybrid*.

TrivialServiceDescription Der Mediator konstruiert eine einfache Beschreibung als *AMETASServiceDescription* aus dem übergebenen String, indem er Standardwerte für die übrigen Felder der Beschreibung annimmt und den übergebenen String als Klassenname interpretiert. Der resultierende Typ ist bei diesem Modus *stringbasiert.*

PUID Der übergebene String wird als Stellennutzer-ID interpretiert. Es wird ein Konzeptgraph erzeugt, welcher die Information repräsentiert, dass der zugehörige Stellennutzer diese ID besitzt.

Hybrid Der Mediator erwartet bei diesem Modus die Übergabe einer textuellen Repräsentation eines Hybridtyps, so wie er in Abschnitt 6.1 gezeigt wird. Anschließend wird der Typcompiler aktiviert, um eine Laufzeitrepräsentation dieser Typbeschreibung zu erhalten.

Soll beispielsweise ein Agent einen Stellennutzer nach einer Beschreibung ausfindig machen, so muss diese Beschreibung statisch, das heißt innerhalb des Codes, auftauchen. Anderenfalls müsste der Anwender den Typ mit einem speziellen Programm konstruieren und dem Agenten zusenden. Es ist offensichtlich, dass der Einsatz des Compilers notwendig wird, wenn der Anfrager seine Anfrage als Text vorliegen hat. Die Konstruktion von Typen ist auf Basis der textuellen Beschreibung wesentlich einfacher und übersichtlicher und damit leichter auf Fehler zu überprüfen.

6.8 Unterstützung durch das Agentensystem

Das Agentensystem selbst muss eine Unterstützung anbieten, damit das Typsystem erfolgreich eingesetzt werden kann. Diese Unterstützung umfasst das Registrieren, Laden, Starten, Terminieren und das Migrieren von Agenten.

6.8.1 Grundsätzliche Anforderungen an ein Agentensystem

Das Hybridtypsystem wurde auf Basis der Agentenplattform AMETAS entwickelt. Es stellt sich aber die Frage, wie *generisch* der Ansatz gewählt wurde: Kann das Typsys-

tem prinzipiell auch in anderen Agentensystemen Anwendung finden?

Zuordnung

Einer der wichtigsten Punkte, der zu klären ist, ist die *Zuordnung* von Typen zu Agenten. Das Agentensystem muss es erlauben, Agenten mit Typbeschreibungen zu attribuieren. Dies stellt genau dann kein Problem dar, wenn ein Agent eine klar abgegrenzte Menge von Objekten beschreibt, die also als *eine* Entität jederzeit genau zu erkennen sind. Dies wird von den bekannten Agentenplattformen unterstützt.

Zugriff auf Beschreibungen

Die jeweilige Stelle muss eine Tabelle führen, in welcher Typbeschreibungen den Identifikatoren der Agenten eindeutig zugeordnet sind. Es muss jederzeit möglich sein, die Typbeschreibung eines Agenten zu erhalten; dies kann durch eine zentrale Stelle geregelt sein, wenn das dadurch gesteigerte Datenaufkommen auf dem Netz nicht als störend empfunden wird. In AMETAS werden die Typbeschreibungen mit den Agenten verteilt.

Typisierbare Kommunikation

Ein größeres Problem stellt die Modellierung der Kommunikation dar, weshalb in dieser Arbeit für eine Homogenisierung der Kommunikation plädiert wurde. Jedoch legt die Hybridtypbeschreibung die Nachrichtentypen lediglich als Folge von Datentypen fest; da jede Kommunikation ein Datenaustausch ist, kann jede Kommunikation als Nachrichtenübermittlung angesehen werden.

Ein synchroner Methodenaufruf ist demnach eine Abfolge einer Eingangs- und einer Ausgangsnachricht. Dies ist über den Transitionstyp leicht definierbar. Insbesondere gibt der Hybridtyp keine Auskunft darüber, dass eine Kommunikation asynchron sein *muss*, wie es in AMETAS der Fall ist. Die Annotationen können auf einfache Weise verwendet werden, den Aufrufmodus genauer festzulegen, etwa wie:

```
nachr1=CG:{[Nachricht]->(”ist ein”)->[Methodenaufruf]};
```

dies setzt die Definition einer geeigneten Ontologie voraus. Ebenso wäre eine Socketkommunikation möglich:

```
nachr1=CG:{[Nachricht]->(”verwendet”)->[Socket:{...}]};
```

Es ist notwendig, strikte Konventionen für die Erzeugung solcher Annotation zu finden, da sonst die Typüberprüfung durch die wachsende Zahl von Annotationen unsicherer wird.

Trennung der Kommunikationspartner

Dieser Punkt ist das wichtigste Kriterium, welches das Typsystem erfüllen muss. Der Agent muss stets in der Lage sein, den Initiator einer Kommunikation eindeutig zu bestimmen. Dies ist erforderlich, damit im Falle konkurrierender Klienten die Protokollspezifikation erfüllt werden kann. Anderenfalls kann der Zustand des angesprochenen Objekts durch zwischenzeitliche Kommunikation mit anderen Objekten in einer unvorhergesehenen Weise verändert werden. Leider schließt dieser Punkt eine Kommunikation auf Basis von Methodenaufrufen in allgemeiner Form aus: Es besteht für den Aufgerufenen in der Regel keine Möglichkeit herauszufinden, wer der Aufrufer ist. Dieses Manko kann behoben werden, indem ein spezielles Argument jedes Methodenaufrufs zur Übermittlung der Aufrufer-Identifikation verwendet wird.

Regel 6.7 Eigenschaften des Agentensystems
Ein Agentensystem, welches das Hybridtypsystem zur Vermittlung von Entitäten verwenden soll, muss folgende Kriterien erfüllen:

1. Die Entitäten müssen während der gesamten Zeit, während der sie Gegenstand einer Vermittlung sein können, als Einheit gesehen werden können, auf welche die Beschreibung stets zutrifft.
2. Es muss gewährleistet sein, dass der Typ jeder Entität in Erfahrung gebracht werden kann, welche Gegenstand einer Vermittlung sein kann. Es muss ferner dafür gesorgt sein, dass das System eine Zuordnung zwischen Typen und zugehörigen Entitäten verwaltet, sodass eine Vermittlung auf Basis der Durchsuchung dieser Zuordnung durchgeführt werden kann.
3. Jegliche Form der Interaktion mit der Umgebung muss als Nachrichtenaustausch modelliert werden können.
4. Eine zu vermittelnde Entität muss in der Lage sein, den Absender einer Nachricht eindeutig zu bestimmen.

6.8.2 Registrieren und Laden von Stellennutzern

Stellennutzer müssen vom Agentensystem geladen und gestartet werden, da der Betrieb von Stellennutzern einer Authentifizierung und Autorisation des Benutzers unterliegt. Aus diesem Grunde darf kein Stellennutzer einen anderen Stellennutzer selbst laden und instantiieren. Abbildung 6.9 zeigt den Vorgang des Starts eines Stellennutzers im Agentensystem AMETAS. Kennt der Klient den Registrierungsnamen (hier „A“ in der Abbildung) nicht, so kann er eine Typanfrage an den Mediator stellen. Existiert ein passender Eintrag im Stellennutzerverzeichnis *(PURepository),* so kann der Klient den Namen vom Mediator entgegennehmen und das Laden des Stellennutzers veranlassen.

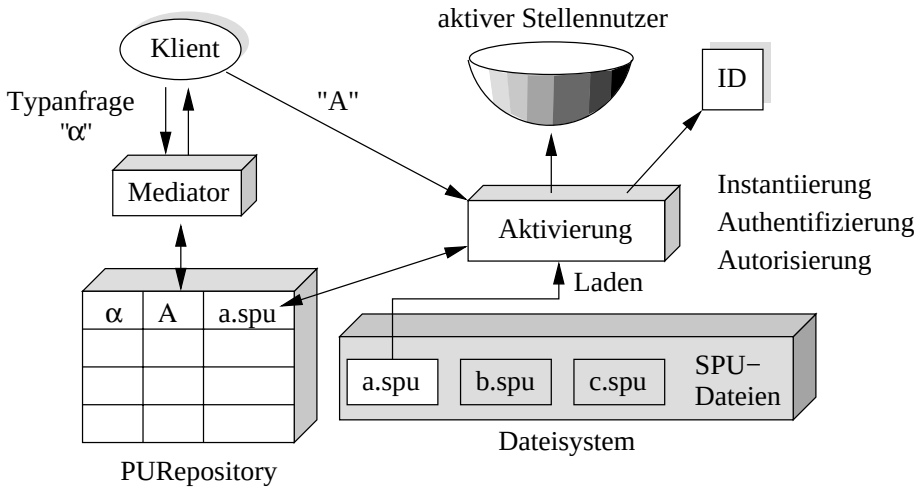

Abbildung 6.9: Starten eines Stellenutzers

Der tatsächliche Dateiname bleibt dem Klienten verborgen.

Die erfolgreiche Instantiierung bewirkt die Bildung einer neuen Stellennutzer-ID, welche wiederum im Repositorium zusammen mit dem Typ eingetragen wird – diesmal als *Instanzregistrierung*.

6.8.3 Terminierung und Migration

Agenten treffen bei ihrer Migration in Form eines Datenstroms bei der Zielstelle an. Das Objekt der Stelle, welches diesen Strom in Empfang nimmt, ist der so genannte *AgentPort*. Dieser verwendet ein spezielles Objekt, welches das Migrationsprotokoll kapselt und dabei Funktionen wie Entschlüsselung und Abgleich der zwischengespeicherten Klassen ausübt.

Abbildung 6.10 illustriert die Vorgänge beim Abschluss einer Migration in AMETAS. Nach dem Empfang liegt der Agent in Form des SPU-Containers, eines Identifikators sowie in seiner serialisierten Form vor. Die Klassen, welche im SPU-Container enthalten sind, werden in einem speziellen Klassenlader gespeichert; der SPU-Container selbst wird in einem Zwischenspeicher gehalten, sodass er für folgende Migrationen zur Verfügung steht.

Im SPU-Container befindet sich die Typbeschreibung des Agenten; diese wird zusammen mit dem Identifikator registriert. Damit ist die Instanzvermittlung möglich: Auf eine passende Anfrage kann dem Anfrager die ID des zugehörigen Agenten mitgeteilt werden. Der Agent wird anschließend deserialisiert, die Identität des Absenders, seine Signatur sowie die Signaturen der Autoren werden geprüft und entsprechende Privilegien zugewiesen [ZMG98]. Danach erhält der Agent seinen eigenen Kontrollfluss.

Eine Typvermittlung ist in diesem Szenario nicht möglich. Man beachte nämlich,

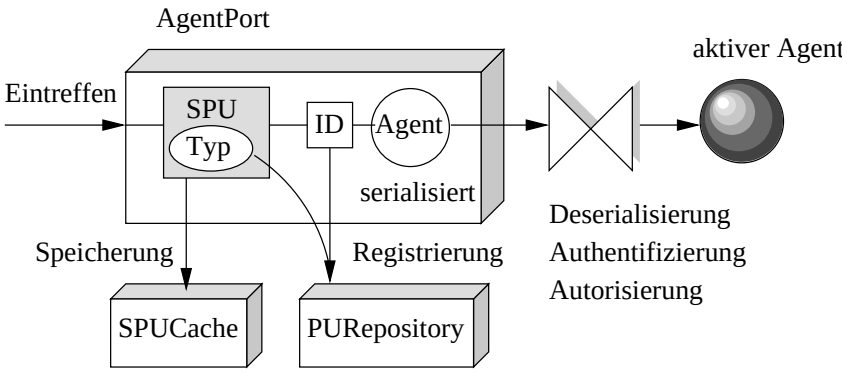

Abbildung 6.10: Eintreffen eines Agenten an einer Stelle

dass es sich um einen Agenten handelt, welcher von einer anderen Stelle stammt; der Absender des Agenten wird in der Regel kein Interesse daran haben, seinen Agenten geklont zu bekommen. Die Registrierung ist nicht dauerhaft, sondern wird nach Verlassen der Stelle nach einer voreingestellten Zeit entfernt.

Wird der Agent terminiert, so ist die Gültigkeit seines Identifikators unmittelbar beendet. Daher wird die Typbeschreibung terminierter Agenten nicht aufbewahrt. Hat der Agent jedoch die Stelle verlassen, so besteht die Chance, dass er wieder an diese Stelle zurückgekehrt. Aus diesem Grunde wird die Registrierung für eine bestimmte Zeit aufrechterhalten. Der Anwender hat die Möglichkeit, festzustellen, ob das Vermittlungsergebnis einen aktuell an dieser Stelle befindlichen Agenten betrifft oder ob dieser mittlerweile abgereist ist (siehe Anhang E).

Andere Stellennutzer (Benutzeradapter und Dienste) können nicht migrieren, also wird ihre Registrierung nach ihrer Beendigung sofort entfernt. Dies betrifft allerdings nur die Instanzregistrierung; eine vorgenommene *Typregistrierung* – welche die Speicherung des SPU-Containers in einem für die Stelle zugänglichen Dateisystem erfordert – bleibt bis zur expliziten Deregistrierung durch den Anwender oder Administrator bestehen.

6.9 Zusammenfassung

Dieses Kapitel demonstrierte die Integration des Hybridtypsystems in AMETAS. Besonderes Augenmerk wurde dabei auf die zentralen Komponenten wie Mediator oder Typcompiler gelegt. Zur Illustration der theoretischen Grundlagen des Typsystems wurden die Abläufe in den Vergleichsalgorithmen präsentiert. Die Laufzeit des Transitionsvergleichs-Algorithmus hängt offenbar von der Komplexität der Transitionsbeschreibung ab; im schlimmsten Falle ist ein exponentielles Anwachsen nicht vermeidbar. Für weiter gehende Informationen, insbesondere im Hinblick auf die Verwendung durch Anwendungsprogrammierer, sei auf Anhang E verwiesen, der einen Teil der Program-

mierschnittstelle präsentiert.

Um das Hybridtypsystem anwenden zu können, muss das Agentensystem gewisse Grundbedingungen erfüllen. Unter anderem ist eine Erkennung der Absender von Nachrichten unerlässlich, um die durch den Typ formulierten Kommunikationseigenschaften zu realisieren, ohne dass parallel ablaufende Kommunikationsvorgänge mit der aktuellen Kommunikation interferieren.

Das Agentensystem AMETAS unterstützt aktiv die Verwaltung von Typen zur Laufzeit. Der Anwendungsprogrammierer muss keine speziellen Vorkehrungen treffen, damit der Typ, welcher dem Agenten zugeordnet ist, dem lokalen Mediator zugänglich gemacht wird. Das System löst die Verteilung der Typbeschreibung durch das den Agenten begleitende Versenden der Typinstanz.

7 Anwendungen

Nachdem die Grundlagen und die Implementierung des Typsystems vorgestellt wurden, sollen in diesem Kapitel einige Beispiele gezeigt werden, wie das Typsystem eingesetzt werden kann. Es ist offensichtlich, dass der Einsatz des Typsystems Auswirkungen auf den Entwurf von Anwendungen hat, wobei es möglich sein soll, ohne das Typsystem zu arbeiten, wenn es nicht benötigt wird. Die Verwendung des Typsystems ist jedoch im Entwurf neuer Anwendungen anzuraten, um den Weg für neuartige Agentenanwendungen zu öffnen.

7.1 Offene Anwendungen

Unter einer *offenen Anwendung* versteht man eine Anwendung, deren Menge interagierender Komponenten nicht festgelegt ist. Solche Anwendungen folgen der so genannten *Open World Assumption*, also der Annahme, dass der „Welt“ für diese Anwendung keine fest definierten Grenzen gesteckt sind.

7.1.1 Typen und Welten

Bislang gehen Anwendungen in der Regel von der Annahme aus, dass die Welt geschlossen ist, das heißt, dass es einen Zeitpunkt während ihrer Laufzeit gibt, nach dem sich die Zahl der Strukturen, die in der Verarbeitung auftauchen, nicht weiter erhöhen kann. Dies betrifft insbesondere die Datentypen. Bei der *Closed World Assumption* kann man als Entwickler von einer bestimmten, vorgegebenen Menge von Typen ausgehen. Es liegt also die für die meisten Anwendungen angenehme Situation vor, dass einerseits Klarheit herrscht, welche Komponenten miteinander in Beziehung treten können und dass andererseits Nachrichten korrekt interpretiert werden, da ihre Semantik eindeutig ist und im Voraus festgelegt werden kann. So kann man einen Dienst schreiben, welcher einem Agenten wiederholt Nachrichten zusendet, welche lediglich aus einem Byte bestehen. Da der Agent den Absender kennt und durch die Annahme einer geschlossenen Welt auf die Semantik des Dienstes und der Nachricht schließen kann, ist er unmittelbar in der Lage, die Bedeutung der Nachricht zu erfassen. All diese Annahmen sind im Falle der offenen Anwendung nicht mehr ohne weiteres berechtigt:

- Da die Menge aller vorkommenden Typen nicht bekannt ist, ist es nicht möglich, zu einem bestimmten Zeitpunkt ein Urteil über die Menge aller gültigen Typen zu treffen. Die Typen neu auftauchender und mit der Anwendung interagierender Komponenten müssen wiederholt allen übrigen Komponenten an entfernten Netzknoten mitgeteilt werden. Dies verursacht eine beträchtliche Netzlast, welche gerade durch den Einsatz von Agenten eingespart werden sollte.

- Da die Menge der Typen nicht beschränkt ist, kann keine Festlegung getroffen werden, welche Semantik den Typen angehört. Wäre die Menge erlaubter Semantiken beschränkt, könnte man sich auf einen beschränkten Satz von Typen einigen. Ein *SecAdminAgent* kann in einem solchen System also ein Agent zur Sicherheitskonfiguration sein, aber dies ist lediglich Spekulation.

- Es ist somit nicht möglich, die Semantik der Typen a priori zu verbreiten, da immer wieder neue Typen auftauchen können. Aus diesem Grunde ist es nicht möglich, diese Typen durch Zeichenketten zu referenzieren, da es hierfür eine eindeutige Abbildung zwischen Namen und Typen geben müsste. Interessenten wären gezwungen, diese Informationen abzurufen, welche ihrerseits stetig aktualisiert werden müsste.

- Ist die Semantik eines Typs nicht a priori festgelegt, so kann der Empfänger keine Schlussfolgerungen über die Semantik der verwendeten Nachrichten ziehen. Die Nachrichten müssten selbst mit einer semantischen Beschreibung versehen werden, welche, würde diese durch eine Zeichenkette beschrieben, wieder Anlass gäbe, eine begrenzte Welt (für Nachrichten) anzunehmen. Im Allgemeinen ist dies nicht der Fall.

Wie man sieht, ist der Fall offener Anwendung ungleich komplizierter als der klassische Fall der geschlossenen Anwendungen. Alleine diese Komplexität, der Grad an Unbestimmtheit und Unsicherheit, ließ Entwickler bislang vor der Entwicklung solcher Anwendungen zurückschrecken.

7.1.2 Einfache Beispielanwendung

Die in Abschnitt 4.3.1 beschriebene Anwendung *StartAndKill* soll hier nochmal genauer betrachtet werden. Zur Illustration betrachte man die in Abbildung 7.1 gezeigte Situation in einem Agentensystem. Die *StartAndKill*-Anwendung erlaubt es, beliebige Stellennutzer zu starten und wieder zu stoppen. In diesem Szenario soll ein Agent namens *RunnerAgent* gestoppt werden.[1] Unklar ist jedoch, wo sich der gesuchte Agent befindet. Erschwerend kommt hinzu, dass er mobil ist und sich leicht der Verfolgung

[1] Die Bezeichnung *RunnerAgent* soll hier nur stellvertretend für die Stellennutzer-ID des Agenten stehen; da diese eindeutig ist, wird mit ihr auch nur genau ein Agent bezeichnet.

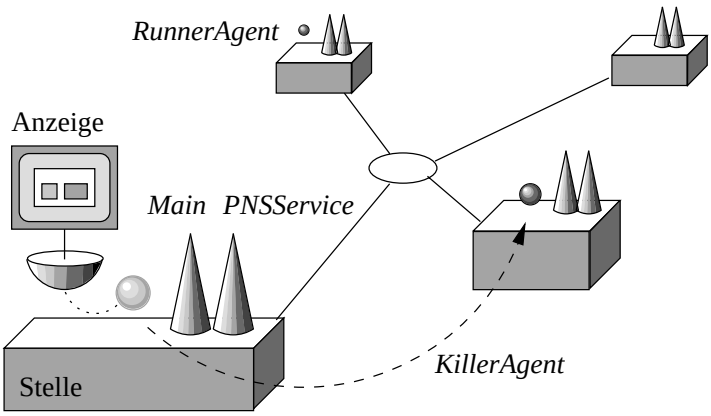

Abbildung 7.1: Beispielanwendung

entziehen kann. Deshalb ist der Migrationsmechanismus in AMETAS so konzipiert, dass ein Agent nur dann „ausreisen“ darf, wenn gegen ihn nichts „anliegt“. Sollte er sich auf einer Stelle einfinden, die seine ID auf ihrer Terminierungsliste stehen hat, wird er umgehend gestoppt, ebenso, wenn er sich gegenwärtig dort befindet, wenn die Terminierungsaufforderung eintrifft.

Den Zugriff auf den Terminierungsmechanismus kann nur ein Dienst anbieten, da es Referenzen auf interne Komponenten der Stelle bedarf. AMETAS definiert einen speziellen Dienst namens *Main*, welcher unter anderem für solche grundlegenden Funktionalitäten vorgesehen ist und bei jedem Stellenstart automatisch geladen wird. Gemäß den Richtlinien in Abschnitt 7.4 wird jedoch ein separater Dienst *PNSService* angeboten, um eine Häufung von Funktionen in einem Dienst zu vermeiden. Seine Aufgabe besteht in der Beschaffung von Daten aus dem Stellennamendienst, insbesondere bezüglich definierter und laufender Stellen. Anhand der Typbeschreibung kann ein Agent beide Dienste auseinanderhalten.

In einem *geschlossenen* System ist die Vermittlung der beiden Dienste an den Agenten völlig unproblematisch. Dort würde verfügt, dass die Terminierungsfunktionalität stets von einem Dienst namens *Main* sowie die PNS-Funktionen über einen Dienst namens *PNSService* angeboten werden. Liegt jedoch ein *offenes* System vor, so muss man davon ausgehen, dass eine Funktionalität nicht unbedingt mit einer einfachen Bezeichnung verknüpft ist, dass also der Dienst, welcher eine Funktionalität anbietet, einen a priori unbekannten Namen aufweist.

Abbildung 7.2 stellt den entsprechenden Codeblock dar, welcher den Dienst lokalisiert, und zwar für beide Fälle. Der Fall des geschlossenen Systems wird durch die Verwendung des *TrivialServiceMediators* gekennzeichnet; dieser ist, wie in Abschnitt 6.7 beschrieben, nur in der Lage, Dienste anhand ihres Namens zu finden. Demzufolge wird die Anfrage als stringbasierte Anfrage formuliert:

```
mrs = new AMETASMediationRequest("Main", true);
```

```
private AMETASPlaceUserID findMainService() {
   AMETASType typeMain=null;
   AMETASMediationRequest mrs = null;
   String sDescr = "annotations { "+
                   "  self=CG:{ "+
                   "    [Dienst]->(terminiert)->[Stellennutzer]" +
                   "  };" +
                   "}";
   m_mi = m_Driver.getMediatorInfo();
   if (m_mi.getName().equals("AMETASTrivialServiceMediator"))
      mrs = new AMETASMediationRequest("Main", true);
   else {
      if (m_mi.getName().equals("AMETASHybridTypeMediator")) {
         try {
             typeMain = m_Driver.createTypeForString(sDescr, "hybrid");
             HybridTypeConformance tc = new HybridTypeConformance();
             tc.setOnlySemantic();
             mrs = new AMETASMediationRequest(typeMain, tc, true);
         } catch (MalformedTypeException mtx) {...};
      }
   }
   AMETASMediationResult[] amrs = m_Driver.request(mrs);
   if (amrs==null || amrs.length==0) {
      m_Driver.output("Main nicht gefunden");
      return null;
   }
   ...
}
```

Abbildung 7.2: Ausschnitt aus dem *KillerAgent*

Mit Hilfe des Hybridtypsystems ist eine Öffnung des Systems möglich. Die Zeichenkette *sDescr* beinhaltet die textuelle Darstellung eines Typs. Um zu einer auswertbaren Darstellung zu gelangen, wird die Treibermethode *createTypeForString* verwendet, welche einen Zugang zum Typcompiler zur Laufzeit darstellt.

Der beschriebene Typ weist keine Aussagen in Bezug auf Nachrichten und Protokoll auf. Die Missachtung des Protokolls ist gewöhnlich nicht tolerabel, sodass das Konformitätsobjekt auf das Ignorieren der Nachrichten mittels

```
tc.setOnlySemantic();
```

eingestellt werden muss. Schließlich wird die Anfrage an den Mediator gerichtet, der eine Liste möglicher Treffer liefert. Jede Instanz von *AMETASMediationResult* be-

inhaltet eine Stellennutzer-ID, wenn es sich um eine vermittelte Instanz handelt, beziehungsweise einen SPU-Namen im Falle einer Typvermittlung.

Diese Instanzvermittlungsanfrage ist bereits in der Lage, zwischen diesem Dienst und dem PNS-Dienst zu unterscheiden: Der PNS-Dienst beinhaltet keine Charakterisierung, welche den in der Abbildung 7.2 dargestellten Konzeptgraphen impliziert:

```
self=CG:{
    [Dienst]- ->(hat)->[Name:{PNSService}],
              ->(liefert)->[Information]-
                           ->("gehört zu")->[Agentenstelle],
                           ->("gehört zu")->[PNS-Domäne].,
              ->("ist Instanz von")->[Klasse]
                           ->(hat)->[Name:{PNSService}].
};
```

7.1.3 Typen und elektronische Marktplätze

Als weiteres Beispiel einer offenen Anwendung sei in diesem Abschnitt der *elektronische Marktplatz* dargestellt. Ein solcher Marktplatz kann sich über viele Rechner in einem ausgebreiteten Netz erstrecken, möglicherweise über die ganze Welt verteilt. Ein Marktplatz auf Basis eines Agentensystems muss damit auf allen beteiligten Rechnern eine Stelle (oder vergleichbare Strukturen) anbieten und es den Käufern ermöglichen, diese Stellen zu finden und zu besuchen. Auf den Stellen sind gewöhnlich stationäre Anbieter zu finden, während Kunden ihre Agenten einsetzen, um Angebote zusammenzutragen und bei Bedarf in Kaufverhandlungen treten zu können.

Einen elektronischen Marktplatz als geschlossene Anwendung zu realisieren hätte zum einen den Vorteil, schnell zu einem Prototypen zu gelangen, aber andererseits den Nachteil, dass die beteiligten Komponenten festgelegt sind:

- Verschiedene Anbieter entstammen einer festen Menge bekannter Klassen. Sollen diese als Anbieter unterschiedlicher Waren und Dienstleistungen auftreten, so ist ein zusätzliches Vermittlungssystem erforderlich, in dem diese Angebote festgehalten werden. Man würde neben dem Dienstvermittlungssystem eine „Gelbe-Seiten-Vermittlung“ benötigen.

- Werden unterschiedliche und nicht festgelegte Klassen für die Implementierung der Anbieter eingesetzt, so muss es Möglichkeiten geben, diese Komponenten als Anbieter zu identifizieren. In AMETAS ist es möglich, entsprechende Hinweise in die Stellennutzer-ID einzutragen, also etwa *MovieService* als Name. Diese Erkennungsmarke muss allen Agentenentwicklern bekannt gemacht werden.

- Gleiches gilt für die als Agenten auftretenden Nachfrager. Der Agent muss die Kommunikation mit einem Anbieter eröffnen, um sich ihm gegenüber als Nachfrager zu erkennen zu geben. Eine Vorabinformation (etwa beim Betreten der

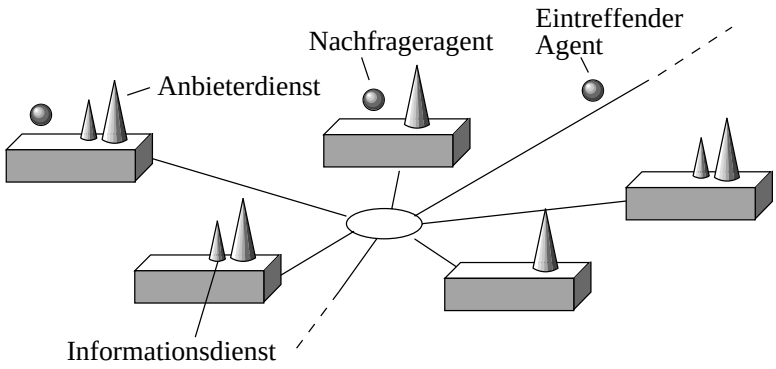

Abbildung 7.3: Elektronischer Marktplatz

Stelle) ist schwierig zu realisieren, wenn man davon ausgehen muss, dass auch Agenten die Stelle besuchen, die kein Interesse an einer Verhandlung haben.

Um den Einsatz des Typsystems im elektronischen Marktplatz zu illustrieren, seien zwei Szenarien vorgestellt. In ersterem soll die Kommunikation keine spezifischen Formate verwenden, sondern alleine auf der Kommunikation mit Standard-AMETAS-Nachrichten basieren. Das zweite Szenario demonstriert die Kooperation des Typsystems mit dem KQML-Subsystem von AMETAS. Abbildung 7.3 zeigt einen möglichen Aufbau eines elektronischen Marktplatzes für beide Szenarien. Unterschieden werden möge zwischen Agenten, welche als Nachfrager auftreten sowie Agenten, welche in diesem Szenario keine Funktion ausüben. Ebenso sind nicht alle Dienste diesem Szenario zuzurechnen. An einigen Stellen befindet sich neben einem *Anbieterdienst* auch ein *Informationsdienst*, welcher Handelsinformationen bietet, so etwa Statistiken über den Warenumsatz, die Anzahl der Nachfrager und die Preisbewegungen. Das Stellensystem ist offen, es können also von außerhalb jederzeit Agenten hinzukommen und das lokale Stellensystem wieder verlassen. Es ist leicht einzusehen, dass es eines Vermittlungssystems bedarf, um es insbesondere den „fremden“ Agenten zu erlauben, in Kontakt mit Diensten dieses Stellensystems zu treten.[2]

Elektronischer Marktplatz mit Standardkommunikation

Da in diesem Falle auf die Standardkommunikation zurückgegriffen wird, lassen sich geeignete Typen nicht nur anhand der Annotationen, sondern auch anhand der akzeptierten und gesendeten Nachrichten erkennen. Als Typbeschreibung für einen Verkaufsdienst sei die bereits in Abschnitt 6.1 gezeigte Beschreibung nochmals aufgeführt (nach [Zap01]):

[2] Dies ist im Übrigen geeignet, die *Fairness* sicherzustellen, da es nicht von der Geschwindigkeit der Propagation eines Anbieterdienst-Namens abhängt, wie viele Agenten diesen Dienst aufsuchen.

```
messages {
   in {
      Anfrage: { Name:String };
      BegrAnfrage: { Name:String, Limit:Float };
      Kaufe: { Name:String, Preis:Float, Kdaten:String[] };
   }
   out {
      Angebot: { Preis:Float };
      JaNein: { Boolean };
      Handel: { Bestaetigung:String };
   }
}
states {
   initial =Anfrage> initial:Angebot;
   initial =BegrAnfrage> initial:JaNein;
   initial =Kaufe> initial:JaNein
                + initial:JaNein,Handel;
}
annotations {
   self=CG:{
      [Dienst]->("führt aus")->[Handel]->("von")->[Buch]
   };
}
```

Der Dienst ist also in der Lage, drei verschiedene Nachrichten anzunehmen und mit bestimmten Ausgaben zu antworten. Dann kehrt er immer wieder in den Ausgangzustand zurück, implementiert also ein triviales Protokoll. Der Kunde kann eine Anfrage nach einem Buch stellen, das er mit einem eindeutigen Bezeichner referenziert. Der Dienst wird ihm dann ein Angebot schicken. Möchte der Kunde selbst einen Preisvorschlag unterbreiten, kann der Dienst mit einem booleschen Wert seine Zustimmung oder Ablehnung signalisieren. Schließlich wird der Handelsvorgang durchgeführt, wenn der Kunde die Ware mit der *Kaufe*-Nachricht zu einem bestimmten Preis bestellt. In diesem Falle kann der Dienst entweder ablehnen oder den Verkauf durchführen. Diese Alternativen stellen sich als nichtdeterministische Alternativen dar, welche von außen nicht steuerbar sind.

Die Beschreibung in der Selbstannotation kann helfen, den Dienst leichter zu lokalisieren. Im Übrigen ist es durch die Verwendung einer Ontologie (wie in Anhang C beschrieben) möglich, die Suche auf *Stellennutzer* anstelle *Dienste* auszuweiten. In diesem Falle würde noch immer diese Typbeschreibung zutreffen, da die Ontologie einen Dienst als speziellen Stellennutzer definiert.

Um die Trennung verschiedener Dienste zu demonstrieren, sei hier eine mögliche Typbeschreibung für den *Informationsdienst* aufgeführt (ebenfalls nach [Zap01]):

```
messages {
   in {
      GibMittelwert: { Name:String };
      WerIstHier: { Wer:String };
      WerBietetAn: { Name:String, Limit:Float};
   }
   out {
      Mittelwert: { Preis:Float };
      ListePU: { Anbieter:PlaceUserID[] };
   }
}
states {
   start =GibMittelwert> start:Mittelwert;
   start =WerIstHier> start:ListePU;
   start =WerBietetAn> start:ListePU;
}
annotations {
   self=CG:{
      [Dienst]->(”bietet an”)->[Information]
                  ->(beschreibt)->[Handel]
   };
   Wer=CG:{ [Datum]->(hat)->[Wert:{wer}] };
}
```

Die *Wer*-Nachricht zeigt die Verwendung einer Konstantenannotation, um eine bestimmte Nachricht zu markieren. Man beachte, dass die Nachricht *GibMittelwert* ihr syntaktisch gleicht, aber durch die Annotation unterschieden wird. Es bleibt jedoch vorauszusetzen, dass es keine Ware mit dem Bezeichner „wer“ gibt.

Anhand der Selbstannotation können die Dienste bereits unterschieden werden. Der Agent könnte einen Typ mit der Selbstannotation

```
self=CG:{ [Stellennutzer]->(”führt aus”)->[Handel] };
```

suchen. Diese Annotation passt nur auf den Verkaufsdienst, nicht auf den Informationsdienst. Bemerkenswert ist die Übereinstimmung der Nachrichten *BegrAnfrage* und *WerBietetAn:* Wird eine Nachricht des ersteren Typs gesucht, würde auch der letztere als passend erachtet werden. Der Typvergleicher kann jedoch im zweiten Typ keine passende Interaktion finden, denn der Anfrager erwartet eine Ja/Nein-Antwort, die im zweiten Typ nicht vorkommt.

Elektronischer Marktplatz mit KQML-Kommunikation

AMETAS bietet die Möglichkeit, KQML-Nachrichten für die Kommunikation zwischen Stellennutzern zu verwenden [Jah01]. Die KQML-Spezifikation sieht einen ei-

genen Vermittlungsmechanismus vor. Diese Vermittlung basiert im Wesentlichen darauf, dass ein KQML-Agent angibt, bestimmte Anfragen verarbeiten zu können. Dabei spielt nur das Format der Nachricht eine Rolle; der Inhalt wird nicht erfasst. Die Vermittlung wird durch Instanzen geregelt, welche *Facilitators* genannt werden [FW91]. Instanzen, welche eine bestimmte Funktionalität anbieten, melden dies beim lokalen Facilitator an; daraufhin kann dieser

- eintreffende Nachrichten an die passende Instanz weiterleiten, deren Antwort entgegennehmen und an den Anfrager schicken (Anfrage über *Broker*-Performative);
- auf eintreffende Nachrichten dem Anfrager eine Referenz auf den passenden Partner zuschicken (Anfrage über *Recommend*-Performative);
- eintreffende Nachrichten an die passende Instanz weiterleiten, welche ihrerseits direkt in Kontakt mit dem Anfrager tritt (Anfrage über *Recruit*-Performative).

Der Vorteil von KQML liegt in der Möglichkeit, komplexe Kommunikationsabläufe durch vordefinierte *Performative* zu modellieren und somit die Protokollkomplexität durch genormte Abläufe zu verringern [Jah01].

Die Vermittlung über KQML verhält sich orthogonal zum Hybridtypsystem; beide sind aufeinander nicht angewiesen. Wird KQML verwendet, empfiehlt sich die Verwendung des zugehörigen Vermittlungsmechanismus anstelle des Hybridtypsystems, da er innerhalb der Sprache KQML repräsentiert ist. Allerdings stellt sich ein Problem, das innerhalb von KQML nicht zu lösen ist: Agenten müssen die Adressen gewisser KQML-Instanzen, insbesondere der Facilitators, kennen, da sie sonst kein Vermittlungsgesuch abschicken können. Die Facilitators werden in AMETAS als Dienste realisiert, verfügen also nicht nur über eine Stellennutzer-ID, sondern sind über das Hybridtypsystem typisierbar. Das Vorgehen gestaltet sich somit für den Anfrager-Agenten wie folgt:

1. Feststellen der notwendigen KQML-Kommunikationspartner (insbesondere Facilitators) mit Hilfe des Hybridtypsystems.
2. Formulierung der Anfragen an den jeweiligen lokalen Facilitator.
3. Kommunikation mit den durch den Facilitator vermittelten Diensten.

7.2 Vermittlung aus Benutzersicht

Das Hybridtypsystem ist insbesondere im Hinblick auf die Erleichterung des Findens geeigneter Agenten entwickelt worden. Die oben beschriebenen Anwendungen bringen das Typkonzept in Verbindung mit einer Form von Transparenz, die man *Komponententransparenz* nennen könnte: Der Ablauf einer Anwendung soll nicht durch den

Einsatz bislang unbekannter Komponenten gestört werden, solange sich diese an die Typspezifikation halten. Für den Anwender stellt sich dieser Einsatz des Typsystems so dar, dass seine Anwendung wie erwartet funktioniert, ohne dass er sich um die Zusammenstellung der Multiagentenanwendung Gedanken machen muss.

Ein anderer Anwendungsfall stellt sich dar, wenn der Anwender selbst einen geeigneten Agenten für einen bestimmten Auftrag sucht. Dann interessiert er sich sehr wohl für die spezielle Wahl des Agenten. Dieser Ansatz – nämlich den Anwender in die Spezifika der Multiagentenanwendung einzubeziehen – verfolgt den konträren Ansatz zur Strategie, dem Anwender mit Hilfe der Benutzeradapter eine Agenten-basierte Anwendung wie eine traditionelle Anwendung darzustellen.

7.2.1 Auswahl durch den Anwender

Zwar ermöglicht die Hybridtypspezifikation den Vergleich von Protokollen in Bezug auf Eingangs- und Ausgangsnachrichten, aber gerade diese Protokolle sind für den Anwender, der eine Agentenanwendung nutzen möchte, von geringem Interesse. Dieser sollte vornehmlich daran interessiert sein, dass der Agent oder allgemein das von ihm gesuchte Objekt für einen bestimmten Zweck eingesetzt werden kann.

> Die für die Auswahl durch den Endanwender relevante Information des Hybridtyps ergibt sich fast ausschließlich aus dem semantischen Typ.

Die Anforderungen des Anwenders können des Weiteren in drei Kategorien eingeteilt werden, je nachdem wie unmittelbar der Anwender mit dem gesuchten Objekt zu arbeiten wünscht:

1. Der Anwender sucht eine Anwendung einschließlich der zugehörigen Bedienoberfläche.
2. Der Anwender sucht einen Agenten, der mit einer eventuell schon vorhandenen Oberfläche interagieren soll.
3. Der Anwender sucht ein Objekt (Stellennutzer) innerhalb des Agentensystems, mit dem sein Agent interagieren soll.

Diese Ausgangspunkte haben in AMETAS konkrete Auswirkungen, worauf sich die Typbeschreibung bezieht und welcher Instanz die Anfrage übergeben werden muss. Wird eine Anwendung gesucht, so benötigt der Anwender einen Benutzeradapter, der in der Regel mit bestimmten Agenten zusammenarbeitet. Die Anfrage muss er entweder einem anderen Benutzeradapter übergeben, welcher ihm durch eine *Typvermittlung* den Namen des SPU-Container zukommen lässt, welcher zu laden und zu starten ist, oder er muss durch die externe Schnittstelle (beispielsweise AMAI) die Beschreibung an die Stelle übermitteln, mit der er sich verbunden hat.

Im zweiten Falle könnte der Benutzeradapter des Anwenders die Typbeschreibung dem Mediator der lokalen Stelle übergeben. Auch in diesem Falle würde eine Typvermittlung stattfinden und der passende Agent könnte mittels *requestPUStartup* vom Benutzeradapter gestartet werden. Ist dem Anwender jedoch bekannt, dass die Funktionalität durch einen dritten Stellennutzer erbracht wird, so gibt er die Beschreibung des gesuchten Stellennutzers einem Agenten mit, der sich an diesen Stellennutzer wenden soll. Ein Beispiel dieses Vorgehens ist in Abschnitt 7.1.2 bereits illustriert worden, wenn man dem Benutzer unterstellt, er habe nach einem Dienst zur Terminierung von Stellennutzern gesucht.

Ein Beispiel zum zweiten Falle demonstriert indes die Schwierigkeit, eine geeignete Anfrage zu formulieren. Ein Administrator wünscht einen Agenten zu verwenden, der einen Schnappschuss derzeit aktiver Stellennutzer auf einer gegebenen Stellenmenge erstellt. (Zur Vereinfachung sei angenommen, dass die Stellennutzer während dieses Vorgangs ihren Aktivitätszustand nicht ändern oder migrieren.) AMETAS bietet die Möglichkeit, die Menge aktuell laufender Stellennutzer an einer Stelle zu bestimmen, indem der Mediator mit einer leeren Instanzanfrage aufgerufen wird; es ist also nicht erforderlich, weitere Stellennutzer in dieser Aufgabe zu involvieren. Die Formulierung muss somit innerhalb der Selbstannotation ausgeführt werden:

```
self=CG:{
     [Agent]->(″trägt zusammen″)->[Menge]
                ->(″besteht aus″)->[PUID]->(″gehört zu″)
                ->[Stellennutzer]->(″verweilt auf″)->[Stelle]
                ->(″gehört zu″)->[Reiseroute]
     };
```

Diese Annotation scheint relativ lange, beschreibt aber ziemlich genau, welche Aufgabe dieser Agent hat. Dabei ist dieses Beispiel bewusst einfach gewählt. Die Situation wird komplizierter, wenn relevante semantische Aussagen innerhalb der Annotationen der Nachrichten zu finden sind, da dann eine Konfrontation des Anwenders mit den Nachrichten und damit auch dem Protokoll unausweichlich wird.

Es sind Zweifel angebracht, ob ein Alltagsanwender (und auch ein Administrator) eine solche Beschreibung selbständig erstellen kann. Ein Anwender muss Unterstützung durch ein Programm erfahren, das möglicherweise auf Basis eines *Interviews* die notwendigen Informationen zusammenträgt. Damit könnte es möglich sein, die Vielzahl ähnlicher oder äquivalenter Beschreibung bedeutend zu minimieren und in einigen Fällen zu *kanonischen Beschreibungen* zu gelangen, welche die Wahrscheinlichkeit des Erfolgs bei der Mediation erhöhen.

7.2.2 Der generische Benutzeradapter

Leider ist es nicht ohne weiteres möglich, einen Agenten herauszusuchen, welchen man einen Auftrag anvertraut, ohne dass man grundlegende Informationen über diesen be-

sitzt – etwa, welche Nachrichten er zur Initialisierung benötigt. Ein Anwender ist nach dem Erwerb eines Textverarbeitungsprogramms zunächst gezwungen, die Bedienung des Programms zu erlernen, was letztlich bedeutet, dass er lernen muss, seine Form der Interaktionen mit dem Programm derart zu gestalten, dass die Aktionen vom Programm verstanden werden und die Intention des Anwenders korrekt umgesetzt wird.

Im Falle einer Agentenanwendung würde das Einsetzen eines bislang unbekannten Agenten die leicht nachvollziehbare Frage aufwerfen, wie mit diesem Agenten kommuniziert werden muss. Diese Kommunikation kann mit einer Typbeschreibung recht genau beschrieben werden, sodass die Anpassung eines Benutzeradapters auf den jeweiligen Agenten möglich ist. Ein Problem ergibt sich insbesondere dann, wenn keine Anpassung des Adapters durch den Anwender vorgenommen werden kann – also gerade dann, wenn ein nicht programmierkundiger Endanwender einen neuen Agenten einsetzen will.

In CORBA ist es möglich, mittels der Schnittstelle für dynamische Aufrufe *(Dynamic invocation interface, DII)* zur Laufzeit einen Aufruf auf einem Serverobjekt zu konstruieren, dessen Schnittstelle zur Kompilierungszeit unbekannt war. Dazu ist neben der Bindung zum betreffenden Serverobjekt lediglich die Kenntnis der Signatur der angebotenen Methoden erforderlich; mit diesen Informationen lässt sich ein generischer Klient konstruieren. Ähnliches ist für ein Agentensystem denkbar, aber aufgrund der inhärenten Protokollkomplexität wesentlich schwieriger zu realisieren. Man würde im Falle von AMETAS von einem *generischen Benutzeradapter* sprechen.

Die Interaktion mit dem Anwender erfordert die Eingabe von Daten durch den Anwender, um Nachrichten zu konstruieren. Hierfür müssen ihm geeignete Eingabefelder zur Verfügung stehen. Informationen zu den Nachrichtentypen sind dem syntaktischen Typ zu entnehmen. Auf der anderen Seite müssen dem Anwender die Informationen der eintreffenden Nachrichten zugänglich gemacht werden.

Eingabefelder für Nachrichtenelemente

Nachrichtenelementtypen sind im Hybridtypsystem auf Systemklassen sowie Basistypen (innerhalb entsprechender Einbettungsklassen) beschränkt. Die Erstellung einer geeigneten Nachricht hängt von bestimmten Faktoren ab:

- Werden skalare Basistypen oder spezifische skalare Klassentypen (etwa *String* oder *InetAddress*) verwendet, so sind einfache Eingabefelder ausreichend und die Konvertierung in die Laufzeitdarstellung ist vermöge der Informationen im syntaktischen Typ möglich.

- Der Datentyp *java.lang.Object* sowie daraus zusammengesetzte Felder erlauben ohne Kenntnis der tatsächlich geforderten Klasse keine angepassten Eingabevorkehrungen.

- Felder *(Arrays)* von Basistypen oder spezifischen Klassen ermöglichen eine entsprechende Eingabe, gegebenenfalls durch Koordinaten-Wert-Paare. Die Größe des Feldes ist unbekannt; eventuell können Hinweise in den Annotationen zu finden sein.[3]

Datenausgabe

Die eintreffenden Nachrichten müssen aufbereitet werden, sodass der Anwender ihren Inhalt verstehen kann.

- Skalare Basistypen sowie Zeichenketten sind unmittelbar durch ihre Ausgabe verständlich.
- Felder können durch Aufzählung der Elemente dargestellt werden. Die Feldgrenzen sind in der Feldinstanz selbst festgelegt.
- Generische Objekte wie Instanzen von *java.lang.Object* definieren meist über *toString()* eine kanonische Ausgabe, die vom Anwender verstanden werden kann.

Die Semantik der ein- oder ausgegebenen Daten wird anhand der Angaben im syntaktischen Typ nicht festgelegt, sondern erst durch Angaben in den Annotationen. Diese Annotationen können dem Anwender in geeigneter Form als Hilfe zu den jeweiligen Feldern präsentiert werden.

Der schwierigste Teil der Implementierung eines generischen Benutzeradapters betrifft die Realisierung des verwendeten Protokolls. Da der Benutzeradapter in dieser Form generisch sein soll, kann er keine automatische Protokollabwicklung leisten; es existieren keine Bindungen zwischen Eingangsnachrichten und Ausgangsnachrichten, die zur Erstellung von Antwortnachrichten ausgenutzt werden könnten.[4] Der Anwender muss selbst für die Fortführung der Interaktion mit dem Stellennutzer sorgen und jede einzelne Nachricht selbst erstellen. Allerdings können die deterministischen Alternativen in jedem Zustand leicht ausgewertet werden und den Anwender entlang des entsprechenden Protokollpfads führen. Im Falle nichtdeterministischer Alternativen obliegt es dann dem Anwender, zwischen diesen Alternativen zu wählen.[5]

Generische Benutzeradapter können aufgrund der Schwierigkeit, alle Fassetten der Anwendung adäquat in der Typbeschreibung zu repräsentieren, nicht so komfortabel wie spezifische Benutzeradapter sein; jedoch ist denkbar, dass für einen bestimmten, nicht zu speziellen Agententyp eine Oberfläche angeboten werden kann, welche sich je nach selektiertem Agenten anpassen kann.

[3] Für die Angabe von Feldlängen sind im Hybridtypsystem noch keine Annotationskonventionen festgelegt.

[4] Zu Bindungen zwischen Variablen siehe Abschnitt 8.2.2.

[5] Die Erzeugung entsprechender Codestrukturen für gegebene Protokolldefinitionen ist in Abschnitt 8.3 genauer erläutert.

7.3 Sicht des Implementierers

Das Hybridtypsystem bietet neben semantischen Informationen insbesondere auch Informationen über die Kommunikationsfähigkeit des Stellennutzers. Diese Informationen sind in den obigen Beispielen nicht zum Tragen gekommen. So interessiert sich ein Benutzer generell für den Einsatzzweck des Agenten und sucht den Agenten nicht anhand des Kommunikationsprotokolls aus. Kommt der Agent zum Einsatz, ist das Protokoll aber entscheidend für die Funktion der Anwendung.

Die Subtypbeziehung anhand der Typbeschreibung festzustellen könnte für Implementierer von Anwendungen von Interesse sein. Dabei spielt die Frage eine Rolle, ob ein gegebener Agent anstelle eines erwünschten Agenten eingesetzt werden kann, ohne dass die übrigen Komponenten modifiziert werden müssen. In diesem Falle kann der gegebene Agent als Subtyp des erwünschten Agenten gelten und an seiner statt eingesetzt werden.

7.3.1 Ableiten von Agenten

Eine weitere Situation, die sich dem Implementierer bieten könnte, ist die Erweiterung eines bestehenden Agenten. Der Agent soll im Rahmen einer neuen Anwendung eingesetzt werden, seine bisherige Funktionalität aber bewahren. In der üblichen Sprechweise der objektorientierten Programmierung würde man von der Notwendigkeit sprechen, ein Objekt *abzuleiten*, also eine Subklasse zu bilden.

Das Ableiten von Objekten wird anhand deren Klassen durchgeführt. Die neuen – abgeleiteten – Klassen fügen den Basisklassen Funktionalität hinzu, indem sie weitere Methoden implementieren und neue Felder einführen. Bestehende Methoden können auch überschrieben werden, das heißt sie werden neu innerhalb dieser Subklasse implementiert.[6] In Java kennzeichnet man das Ableiten von Klassen durch die *extends*-Klausel, beispielsweise wie in

```
public class java.lang.String extends java.lang.Object {
   ...
}
```

Vermöge dieser Festlegung kann man auf einer Instanz von *java.lang.String* jede Methode von *java.lang.Object* aufrufen, etwa *getClass.* Ähnlich wird in anderen Sprachen wie C++ verfahren:

```
class MyAccount : public virtual Account {
   ...
}
```

[6] Besondere Vorsicht sollte man beim Überschreiben von Methoden walten lassen, denn dadurch verändert man jene Komponente des Objekts, die dieses Objekt als Instanz einer Superklasse erscheinen lassen. Siehe dazu auch Abschnitt 3.3.

Damit stellt sich die Frage, wie das Ableiten von Agenten aussehen könnte, da nun ein Typsystem vorhanden ist, welches Typen in eine Typ-Subtyp-Beziehung setzen kann. Offenbar ist die Ableitung der Agentenklasse (wie in Abschnitt 5.1.2 angedeutet) keine Lösung, da sie die Typbeschreibung missachtet.

Man kann zwei Arten von Ableitungen unterscheiden (siehe Abschnitt 3.3): die *Implementationsableitung* und die *Schnittstellenableitung*. Die Implementationsableitung bezieht sich dabei direkt auf die Implementierung der abgeleiteten Klasse. Der Implementierer einer Subklasse kann – sofern der Code der Klasse vorliegt – seine Implementierung zusätzlicher Methoden auf die bestehende Implementierung in der Klasse stützen. Dies erfordert offenbar das Vorhandensein des Codes der Klasse, der seinerseits nach der Ableitung nicht mehr verändert werden sollte. Bei Sprachen, welche von Zeigern auf Speicherzellen Gebrauch machen (wie C++) kann die Änderung innerhalb einer Klasse dazu führen, dass sämtliche Subklassen neu übersetzt werden müssen, da die Verzeigerung korrigiert werden muss. Man nennt dieses Phänomen *zerbrechliche Basisklasse* oder *Fragile base class* [MS98].

Die Schnittstellenableitung ist eine Form der Ableitung, bei der ausschließlich die Schnittstellen in eine Subtypbeziehung zueinander gebracht werden. Dies ist vor allem dann von Interesse, wenn die aufrufenden Objekte nur über die Schnittstelle verfügen, nicht jedoch über Referenzen auf die Implementierung. Die Implementierung ist im Falle der Schnittstellenableitung nicht gezwungen, Code der Klasse mitzuverwenden, welcher die Schnittstelle implementiert, von der abgeleitet wurde. Der Implementierer sichert jedoch zu, dass der neu geschriebene Code zumindest die Funktionalität im erwarteten Umfang erbringt.

Da es sich beim Hybridtypsystem um ein System handelt, das die typisierten Instanzen von ihrer *äußeren Kommunikationsfähigkeit* her beurteilt, ist die Parallele zwischen Hybridtypen und Schnittstellen offensichtlich. Eine Ableitungsbeziehung zwischen zwei typisierten Agenten wird nicht zwischen deren Implementierung, sondern deren Typbeschreibung etabliert. Die nahe liegende Form der Schnittstellenableitung, welche auf die Elternschnittstellen über Bezeichner verweist, würde auf das Hybridtypsystem übertragen bedeuten, einen Hybridtyp als von einem zweiten mittels Namen benannten Typ abgeleitet zu sein. Wäre eine Hybridtypspezifikation der Form

```
supertypes typ1,typ2,...,typn;
messages {
   ...
}
states {
  ...
}
annotations {
  ...
}
```

erlaubt, so wäre die Semantik der Deklaration der ersten Zeile so festzulegen, dass sich hinter *typ1* und den übrigen Bezeichnern vollständige Hybridtypen verbergen würden. Damit würde ein Hybridtyp auf einen einfachen Bezeichner abgebildet, was dem in Abschnitt 5.1.2 genannten Charakter der klassischen Typisierung entspricht, nämlich eine Typbeschreibung samt ihrer zugehörigen Semantik auf einen Namen abzubilden. Denn hätte man einen *SecAdminAgent*, welcher als Superklasse einen *AdminAgent* nennt, so muss die Definition von *AdminAgent* immer dann vorliegen, wenn der Typ von *SecAdminAgent* ausgewertet wird. Bezeichner für die Referenzierung von Supertypen zu verwenden erfordert erneut die Propagation der Abbildung *Bezeichner-Typspezifikation* an alle Stellen, bei denen die Interpretation eines Subtyps ermöglicht werden soll.

Aus diesem Grunde ist es nicht möglich, im Hybridtypsystem eine Supertypbeziehung explizit zu deklarieren. Die Sub-/Supertypbeziehung wird durch den Vergleich der Typspezifikationen zur Laufzeit entdeckt; die Beschreibungen müssen nicht einmal textuelle Ähnlichkeiten haben, solange die transitionelle Ersetzbarkeit inklusive der Annotationskompatibilität nachgewiesen werden kann. Zusammenfassend gilt für die Konstruktion von Subtypen:

Regel 7.1 Ableiten von Agenten
Für die Erzeugung von Agenten, welche einem Subtyp eines gegebenen Agententyps entspringen, gilt:

1. Reine Schnittstellenvererbung: Die Ableitung von Agenten unter dem Hybridtypsystem erfordert entweder die Reimplementierung der durch den abgeleiteten Typ angebotenen Funktionalität oder die explizite Integration von Programmcode in Quellform oder übersetzter Form. Eine implizite Referenz auf eine existierende Implementierung eines Supertyps ist nicht möglich.

2. Prinzipielle Überschreibung: Bedingt durch die Reimplementierung wird jede Funktionalität eines Supertyps generell im Subtyp überschrieben. Jedoch muss die Überschreibung exakt die Funktionalität des Supertyps realisieren.

3. Abgeschlossenheit der Beschreibung: Die Ableitung eines Typs kann Supertypen nicht durch Namen referenzieren. Jede Typbeschreibung ist für sich abgeschlossen und offenbart Subtypbeziehungen ausschließlich durch den Vergleich mit dem potenziellen Supertyp.

Es ist natürlich möglich, über die Annotationen Aussagen über die Verwandtschaftsbeziehungen zwischen Typen zu treffen:

```
self=CG:{ [Agent]->(”abgeleitet von”)->[Stellennutzer]
                                      ->(hat)->[Name:{AdminAgent}]
}
```

Man beachte jedoch, dass man hier – über Umwegen – die Bedingung der Bindungs-

freiheit unterminiert, ein solches Vorgehen also wenig empfehlenswert erscheint.

Diese Folgerung mag gerade für den Implementierer unangenehm erscheinen, welcher die Unterstützung der Wiederverwendung von Klassen und Schnittstellen in Programmierumgebungen wie Java gewohnt ist. Für einen Agentenimplementierer, welcher einen Agenten des Typs t_1 implementieren möchte, welcher Subtyp von t_2 ist, ergibt sich demnach folgende Vorgehensweise:

1. Wenn der Quellcode des t_2-Agenten vorliegt, kann die Erweiterung der Funktionalität problemlos vorgenommen werden. Zu empfehlen ist ein strukturierter Aufbau, welcher die Transitionen des Typs widerspiegelt.

2. Wenn der Quellcode des t_2-Agenten nicht vorliegt, so hängt die Wiederverwendung von Code zur Erstellung des t_1-Agenten im Wesentlichen von bestimmten Vorkehrungen in der Implementierung des t_2-Agenten ab. So könnte in t_2 eine Klasse zur Verfügung stehen, welche jeweils einen Protokollpfad im Transitionstyp repräsentiert; dann können weitere Pfade unter Nutzung einer solchen Klasse hinzugefügt werden, wenn diese dem Entwickler zur Verfügung steht. Eine solche Möglichkeit muss entsprechend dokumentiert sein; die betreffenden Klassen müssen einzeln erhältlich und zur Weiterverwendung freigegeben sein.

3. Die neue Typspezifikation t_1 entsteht in beiden Fällen durch Kopieren der t_2-Spezifikation und Einfügen der zusätzlichen Funktionalität. Gegebenenfalls können in der t_2-Beschreibung bestehende

   a) Nachrichtentypen gemäß Ko- und Kontravarianzbedingungen verändert,
   b) nichtdeterministische Alternativen entfernt,
   c) Transitionen durch weitere (deterministischen Alternativen) ergänzt und
   d) Annotationen verfeinert werden.

In AMETAS wird der neue t_1-Agent dann den Nutzern oder den Anwendungsprogrammierern in Form eines SPU-Containers zur Verfügung gestellt.

7.3.2 Typisierte Anwender

Für den Implementierer von Agenten kann sich eine zur Benutzersicht duale Sicht der Vermittlung darstellen: Während der Benutzer einen geeigneten Agenten für eine bestimmte Aufgabe sucht, kann es wünschenswert sein, einen *geeigneten Benutzer* zu finden, mit dem eine Interaktion stattfinden soll. Bedingung ist, dass einem Benutzer eine Typbeschreibung zugewiesen werden kann, welche durch das Typsystem mit der Beschreibung eines erwarteten Typs verglichen werden kann.

Die Typisierung von Anwendern mag auf den ersten Blick befremdlich klingen – gibt es so etwas wie Kommunikationsprotokolle beim Menschen, hat ein bestimmter

Mensch eine Semantik, ist ein Typ außerhalb des Rechnersystems überhaupt vergleichbar? Das Agentensystem AMETAS liefert hierauf eine einfache Antwort:

> Benutzer werden in AMETAS durch so genannte Benutzeradapter repräsentiert. Diese Benutzeradapter teilen sich dasselbe Kommunikationssystem mit Agenten und Diensten.

Der menschliche Benutzter am Benutzeradapter ist aus Sicht des Agentensystems mit der im Agentenobjekt gekapselten Implementierung insofern vergleichbar, als dass er zwischen verschiedenen Alternativen in nichtdeterministischer Weise entscheiden kann. Auch wenn der Zugang des Benutzers zum Agentensystem in verschiedenen Agentensystemen unterschiedlich geregelt ist,[7] gibt es meist eine klar definierte Entität, welche die Kommunikation mit einem Benutzer bewerkstelligt, welche ihrerseits entsprechend typisierbar ist. Bei AMETAS sind dies die Benutzeradapter; diese bieten wie jeder Stellennutzer eine Kommunikationsschnittstelle an, welche ein Protokoll definieren kann, Nachrichtentypen deklariert sowie Annotationen aufweisen kann. Da ein Anwender darauf angewiesen ist, über ein solches Objekt wie dem Benutzeradapter in das System einzugreifen, ist er auf die Implementierung von Protokollen innerhalb dieses Objekts angewiesen. In der Regel ist es dem Benutzer nicht möglich, ohne entsprechende Umprogrammierung seines Adapters eine unvorhergesehene Aktion im System durchzuführen.

Folgerung 7.2 Typisierung von Anwendern
Aus der Sicht eines Agentensystems kann ein menschlicher Benutzer anhand seines aktuell verwendeten Adapters typisiert werden.

Abbildung 7.4 illustriert ein mögliches Szenario. Auf Stelle 1 ist ein Dienst installiert, welcher ein Netz überwachen soll. Entdeckt er eine Fehlfunktion, so sendet er einen Boten los, um den Administrator zu benachrichtigen. Bislang würde der Administrator einen bestimmten Benutzeradapter verwenden, der den Identifikator (Stellennutzer-ID) der gesamten Anwendung mitteilt und so dafür sorgt, dass die Boten ihn auffinden können. Unter Verwendung des vorgestellten Typsystems bietet sich eine größere Flexibilität:

- Der Botenagent muss weder die Lokation des Administrators noch den Identifikator des Benutzeradapters kennen.
- Der Botenagent kann seinen Ansprechpartner auswählen. So wird er in einem weitläufigen System mit zahlreichen Stellen vielen Benutzeradaptern begegnen, welche die Daten nicht geeignet darstellen können, eventuell nicht einmal Zugang zu diesen haben dürfen.

[7] Das Programmierhandbuch von Grasshopper (unter [IKV01]) schlägt vor, die Benutzereingabe durch eine grafische Schnittstelle des Agenten selbst zu bewerkstelligen.

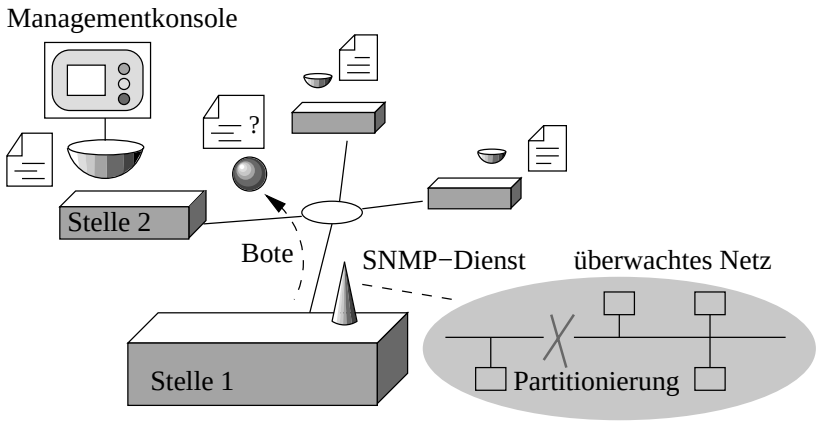

Abbildung 7.4: Netzmanagementszenario mit typisierten Anwendern

- Das Agentensystem muss dafür sorgen, dass der Agent vom Typ des Benutzeradapters auf die Privilegien des Anwenders schließen kann. Wenn also nur ein „Netzadministrator“ die Daten entgegennehmen darf, so sollte dies durch das Sicherheitssystem des Agentensystems sichergestellt sein.

Die Privilegien des Anwenders sind innerhalb der Annotationen im Typ des Benutzeradapters definierbar und damit innerhalb eines Typvergleichs verfügbar. Für die tatsächliche Privilegienüberprüfung sind die im SPU-Container enthaltenen Sicherheitsinformationen heranzuziehen.[8]

7.4 Allgemeine Richtlinien für die Typisierung

Wie die Beispiele dieses Abschnitts zeigen, ist die Erstellung und erfolgreiche Vermittlung von Hybridtypen von zahlreichen Faktoren, aber auch von der Geschicklichkeit des Implementierers oder Anwenders abhängig, geeignete Spezifikationen zu formulieren. Aufgrund der nicht vollständigen Repräsentierbarkeit der Einflüsse auf innere Zustandänderungen sowie der Semantik ist es möglich, Typbeschreibungen zu erzeugen, welche eine erfolgreiche Vermittlung vereiteln. In diesem Abschnitt soll die Frage erörtert werden, inwiefern die Typisierung den Entwurf von Anwendungen beeinflusst.

7.4.1 Trennung der Funktionalitäten

Gewöhnlich werden bei der Erstellung von Diensten mehrere Funktionalitäten in einem Dienst vereinigt. Zum einen erweist sich dies als ressourcenschonender, zum anderen ist eine Trennung in verschiedene Komponenten in gewissen Situationen ungeeignet.

[8] Zum Aufbau des SPU-Containers bei AMETAS siehe Abschnitt 6.1.

So können unterschiedliche Funktionalitäten voneinander abhängen, was im Falle getrennter Komponenten einen Austausch über Nachrichten erfordert. Ferner ist es leichter, einen geringeren Satz von Diensten zu verwalten, welche an zahlreichen Stellen zu installieren sind.

Mag die Bündelung von Funktionalitäten im Falle geschlossener Anwendungen geeignet sein, für offene Anwendungen ergibt sich ein schwer wiegendes Problem: Wie sollen Objekte typisiert werden, welche über eine große Menge verschiedener Funktionalitäten verfügen? Prinzipiell bedeutet dies nämlich

- die Vereinigung der Annotationen, wobei auf mögliche Widersprüche der Konzepte/Relationen bei Konzeptgraphen zu achten ist;
- die Vereinigung der Nachrichtenmengen für ein- und ausgehende Nachrichten und
- die Kombination der Transitionsgraphen zu einem gemeinsamen Graphen.

Da die Struktur der Nachrichten bei verschiedenen Diensten ähnlich sein kann, wird der syntaktische Typ möglicherweise mit einer Menge einander typgleicher Nachrichten angefüllt. Dies kann zu Verwechslungen führen, wenn die Kompatibilität beurteilt werden soll. Ferner ist es möglich, dass Nichtdeterminismen auftreten, die bei der Trennung in zwei verschiedene Stellennutzer vermieden werden könnten.

Eine Lösung dieses Problems besteht in der Mehrfachzuweisung von Typen zu Stellennutzern.[9] Dennoch muss der Implementierer dafür sorgen, dass die Klientenerwartungen, die sich in dem als zutreffend erachteten Typ manifestieren, erfüllt werden. Anderenfalls ist die Bündelung zahlreicher Funktionen innerhalb eines Stellennutzers zu vermeiden und eine stärkere Modularisierung in Form mehrerer Stellennutzer anzustreben.

7.4.2 Diskriminatoren

Häufig bieten Stellennutzer mehrere Funktionalitäten an, die mit Hilfe so genannter Diskriminatoren selektiert werden. Wie bereits in Abschnitt 5.4.3 dargelegt wurde, ersetzen die Diskriminatoren die von konventionellen Schnittstellenbeschreibungen bekannten Methodennamen. Diskriminatoren erleichtern damit einerseits die Implementierung sowohl auf Anfrager wie auf Adressatenseite, andererseits auch die Übersicht über die verfügbaren Funktionen anhand von Bezeichnern.

Es darf aber nicht übersehen werden, dass Diskriminatoren *nur Annotationen* sind. Sie charakterisieren die Nachricht, welche sie auszeichnen, semantisch mittels eines Bezeichners. So wird einer Nachricht mit Diskriminator *orderTicket* üblicherweise die

[9] Dies ist konzeptionell kein Problem, da das typisierte Objekt eine Spezialisierung jedes Typs ist, der in seinem Namen registriert wird. AMETAS unterstützt dies in der Version 2.5 jedoch nicht.

Bedeutung der Möglichkeit zur Bestellung von Tickets unterstellt, während eine Nachricht mit Diskriminator *killAgent* zum Terminieren von Stellennutzern gedacht sein kann. Konsequent bei allen Nachrichten angewendet bedeutet dies nichts anderes, als dass man eine syntaktische Spezifikation mit impliziten Semantiken erstellt, so wie es bei konventionellen Schnittstellenbeschreibungssprachen üblich ist. Es verbindet sich damit wiederum das Problem, die Diskriminatoren der entsprechenden Nachrichten kennen zu müssen. Dies läuft dem Gedanken der Ad-hoc-Interpretierbarkeit des Hybridtyps zuwider. Folglich sollte die Verwendung von Diskriminatoren eine Spezifikation der Nachricht mit weiteren semantischen Informationen nicht *ersetzen*, sondern *ergänzen*. Dann ist es dem Nachfrager möglich, die Nachricht anhand ihrer Bedeutung zu benennen; er kann sich *dann* informieren, wie der zugehörige Diskriminator heißt.

7.4.3 Vermeidung von Nichtdeterminismus

Der Transitionstyp erlaubt den Entwurf äußerst komplexer Protokollfolgen. Der Aufwand des Konformitätstests hängt dabei in erster Linie nicht von der Anzahl der Zustände und der Kanten ab, sondern vom Nichtdeterminismusgrad. Offensichtlich können nichtdeterministische Übergänge dazu führen, dass der Typkonsistenzprüfer einen „falschen Pfad“ verfolgt, bevor er merkt, dass der vermeintlich entsprechende Pfad zu diesem nicht passt. Rein deterministische Graphen sind hingegen effizient zu prüfen, da jede Kante nur einmal besucht werden muss.

Nichtdeterminismus entsteht vor allem dann, wenn mehrere Nachrichten gleichen Typs sind. Diese Nachrichten müssen durch Diskriminatoren auseinandergehalten werden, sonst ist der Empfänger nicht in der Lage, die erwünschte Funktionalität zu erbringen. Aber auch Sender- und Empfängerannotationen können zu Nichtdeterminismus führen, da sie wie Epsilon-Übergänge erscheinen (siehe Abschnitt 5.5.7), wenn sie einen anderen Kommunikationsteilnehmer repräsentieren.

Die Menge der Nichtdeterminismen sollte durch unterschiedliche Nachrichtentypen oder durch die Deklaration von Diskriminatoren minimiert werden. Ebenso ist darauf zu achten, dass durch Sender- und Empfängerannotationen keine zusätzlichen Nichtdeterminismen auftreten.

7.4.4 Ausführliche Annotationen und kleine Ontologien

Die Formulierung von Annotationen ist schwierig, da sie einerseits auf Zeichenkettenbasis genau festgelegt sein müssen, andererseits bei Konzeptgraphen der Kreativität des Typerstellers unterliegen. Dieser muss bei seinem Entwurf eine hierarchisch strukturierbare Vorstellung des Typs gewinnen. Zugleich muss er die Beschreibung *präziser als jede andere mögliche Beschreibung* formulieren, da nur dann eine Übereinstimmung erfolgt, wenn die Anfrage irgendeines Kunden *allgemeiner* als die gebotene Anfrage ist. Dies zwingt zu einer sehr ausführlichen Charakterisierung des jeweiligen Objekts.

Andererseits erscheinen die verfügbaren Ontologien – so ausführlich sie sein mögen – immer wieder zu klein, um eine Annotation mit Konzeptgraphen zu formulieren. Es besteht dann der Wunsch, weitere Konzepte und Relationen zu definieren. Dies ist jedoch nicht empfehlenswert: Durch die Vergrößerung der Ontologie werden Konzepte eingeführt, die an anderen Stellen aufgrund einer kleineren Ontologie nicht verstanden werden. Das Problem der Propagation von Bezeichnern, welche für bestimmte Sinninhalte stehen, setzt sich dann auf Ebene der Konzepte und Relationen fort. Es muss daher sehr umsichtig und zurückhaltend verfahren werden, wenn eine Vergrößerung der Ontologie erforderlich scheint.

7.5 Zusammenfassung

Es gibt zahlreiche Möglichkeiten, das in den vergangenen Kapiteln vorgestellte Typsystem in Agenten-basierten Anwendungen zu nutzen. Ein Feld, das besonders profitieren kann, sind die offenen Anwendungen. Standardszenarien sehen hier die Implementierung von elektronischen Märkten vor, aber selbst bei Anwendungen im Systemmanagement kann ein Vermittlungssystem die Lokalisation von Agenten anhand einer Beschreibung erleichtern.

Für den Anwender stellt sich das Typsystem anders dar als für den Entwickler. Der Anwender wird in die Lage versetzt, anhand einer Beschreibung einen Agenten aus einer Menge angebotener Agenten unterschiedlicher Funktionalitäten zu selektieren. Er erlangt dadurch größeren Einfluss in die Koordination und Gestaltung von Multiagentenanwendungen, ohne tief gehende Kenntnisse der Programmierung haben zu müssen. Andererseits wird der Benutzer zum vermittelbaren Zielobjekt für Agenten: Da der Benutzer alleine durch seinen Adapter im System repräsentiert wird und sich als Stellennutzer prinzipiell nicht von anderen Agenten unterscheidet, ist die Vermittlung von Benutzern an Agenten sinnvoll und eine konsequente Weiterführung der Gestaltung autonomer, interagierender Komponenten.

Abschließend wurden Hinweise in Bezug auf die Entwicklung von Stellennutzern und dem Entwurf von Typbeschreibungen aufgelistet, wenn man eine möglichst effektive Vermittlung erreichen möchte, welche den tatsächlich gesuchten Stellennutzer zu vermitteln vermag. Wichtige Punkte sind die Vermeidung der Überfrachtung von Stellennutzern mit unterschiedlichen Funktionalitäten, die Einschränkung von Nichtdeterminismen sowie die sorgfältige semantische Beschreibung.

8 Erweiterungen und Ausblick

Die Verwendung des Hybridtypsystems als Basis einer lokalen, typbasierten Vermittlung von Agenten ist die wohl am nächsten liegende Anwendung. In diesem Kapitel sollen drei Konzepte vorgestellt werden, wie das Typsystem sinnvoll erweitert werden kann:

- Erweiterte Konzepte der Vermittlung
- Erweiterungen des Typsystems
- Computergestützte Agentenentwicklung

8.1 Erweiterte Konzepte der Vermittlung

Abbildung 8.1 zeigt ein Szenario, in dem Agenten einen bestimmten Dienst suchen, aber die Stelle nicht kennen, wo sie ihn finden können. Der Stellennutzer könnte auf zwei verschiedene Weisen lokalisiert werden:

- systematisches Absuchen der Stellen, wo dieser Stellennutzer wahrscheinlich läuft;
- Verwendung eines (für einen begrenzten Bereich gültigen) zentralen Verzeichnisses.

Im ersten Fall (welcher durch a) in Abbildung 8.1 symbolisiert wird) wird auf das Zusammentragen von Informationen über Stellennutzer an einer zentralen Stelle verzichtet. Fall b) stellt jene Strategie vor; ein Agent wendet sich an einen speziellen Dienst, um die notwendige Information zu bekommen. Dieser Dienst muss über eine möglichst aktuelle Sicht des Systems verfügen.

8.1.1 Systematisches lokales Suchen

Das systematische Suchen ist eine ausschließlich auf Implementierungsebene des suchenden Agenten anzusiedelnde Strategie. Der Agent bekommt den Auftrag, einen bestimmten Stellennutzer zu kontaktieren; jedoch ist unklar, auf welcher Stelle sich die-

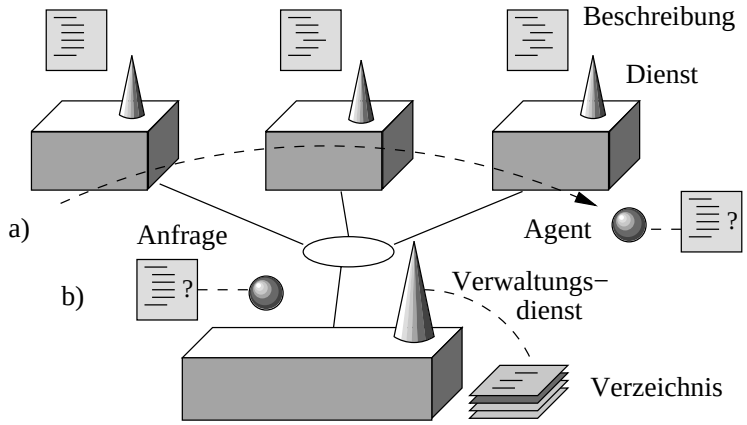

Abbildung 8.1: Globales Suchen

ser Stellennutzer befindet.[1] Unter Zuhilfenahme einer Reiseroute spricht der Agent der Reihe nach jeden Mediator lokal an und wertet die Ergebnisse aus. Hierbei handelt es sich um ein Konzept, das typisch für mobile Agenten ist und nicht-mobilen Objekten in der Regel nicht zur Verfügung steht: Anstatt eine bestimmte Information netzweit verfügbar zu machen, genügt es, den Agenten geschickt durch das Netz zu führen und ihn dabei eine lokal begrenzte Suche durchführen zu lassen. Als Analogie des täglichen Lebens bietet sich die Situation an, dass man in ein vormals unbekanntes Einkaufszentrum geht und es durchschreitet, bis man den Laden gefunden hat, den man sucht.

Ein Problem stellt sich indes, wenn zu viele Stellen vorhanden sind, die Suche also zu einem unvertretbar hohem Aufwand im Vergleich zur nutzbringenden Ausführung gerät. Dies ist vergleichbar damit, dass man in einem Supermarkt steht und eine bestimmte Ware sucht, aber mangels Übersicht genötigt ist, Regal um Regal abzusuchen. Wie nun Märkte ihre Kunden in den richtigen Bereich ihren Marktes mithilfe von hoch angebrachten Schildern und Hinweisen führen, so ist es vorstellbar, dass bekannt ist, dass eine gewisse Teilmenge der Stellen den gesuchten Stellennutzer beherbergen könnte und dass die Suche entsprechend eingeschränkt wird.

8.1.2 Globales Vermitteln

Das systematische Durchsuchen ist bei großen Verbunden von Stellen nicht zu empfehlen. An die Stelle dieser Strategie kann aber eine andere treten, welche Informationen nutzen kann, welche im System vorliegen und an einer bestimmten Lokation gespeichert werden. In Abbildung 8.1 wird diese Strategie unter b) beschrieben: Der suchende Agent wendet sich an einen speziellen Dienst, welcher ein *Verzeichnis* von Stellennutzern inklusive deren Lokationen verwaltet. Anstatt die Typbeschreibung an

[1] Ein ähnlicher Fall existiert bereits bei der Anwendung *StartAndKill*, wenn ein Agent gestoppt werden soll. Der *KillerAgent* weiß nicht von vornherein, auf welcher Stelle er den Agenten antreffen wird.

den Mediator zu richten, schickt der Agent die Typbeschreibung an diesen Dienst, der das Verzeichnis nach möglichen Übereinstimmungen durchsucht. Die Liste möglicher Treffer enthält neben der Stellennutzer-ID des gesuchten Stellennutzers auch die Stelle, zu welcher der Agent reisen muss, um den Stellennutzer anzutreffen, vorausgesetzt, dass dieser nicht weitergereist ist oder terminiert wurde.

Ein solches Verzeichnis muss zusammengestellt und laufend aktualisiert werden. Ein spezieller Dienst könnte in Kontakt mit ähnlichen Diensten an den anderen Stellen dieses lokal begrenzten Bereichs stehen und mit diesen Informationen über die gegenwärtig laufenden Stellennutzern austauschen. Dabei ist zu beachten, dass ausführliche Typbeschreibungen in der Größenordnung über 10 KB leicht die Größe ihres beschriebenen Agenten übersteigen können.

Anstelle einer direkten Kommunikation zwischen den Diensten kann der Einsatz von *Rechercheagenten* sinnvoll sein; diese tragen die Informationen in ähnlicher Weise zusammen, wie es der interessierte Agent im ersten Szenario tun würde. Die Informationen können dabei

- vorbereitet gespeichert sein oder
- auf Bedarf gesucht werden.

Der erste Fall erfordert das Sammeln der Typbeschreibungen aller in Frage kommenden Stellennutzer von allen Stellen. Man würde dann diese Informationen einer zentralen Stelle – genauer gesagt einem ihrer Dienste – zur Verfügung stellen, ohne diese Informationen an alle beteiligten Stellen zu verbreiten. Als zweite Möglichkeit können die jeweiligen Stellen einen bestimmten *Ankündigungsagenten* zu jener zentralen Stelle entsenden, sobald ein neuer Stellennutzer verfügbar wird. Immer betrachtet werden muss selbstverständlich die damit verbundene Netzlast, welche den Einsatz der Agenten als bandbreitensparende Technologie nicht konterkarieren darf. Diese beiden Strategien könnten durch die Termini *Pull-* beziehungsweise *Push-Strategie* umschrieben werden: In ersterem Falle müssen die Daten aktiv von den Stellen eingesammelt werden, in letzterem sorgen diese Stellen selbst für die Aktualisierung der Daten.

Die zweite Möglichkeit – Informationen auf Bedarf zu suchen – lässt sich so realisieren, dass der Verwaltungsdienst, welcher eine Anfrage von einem Agenten erhält, diese Anfrage an einen Rechercheagenten weitergibt, welcher dann ähnlich wie Strategie a) aus Abbildung 8.1 die in Frage kommenden Stellen absucht. Der Unterschied zum genannten Szenario ist, dass der Kundenagent nicht selbst die Recherche durchführen muss, sondern auf eine Dienstleistung zurückgreifen kann.

8.1.3 Lokationstransparenz

Unter *Lokationstransparenz* versteht man eine bestimmte Form der *Verteilungstransparenz*, bei welcher eine Anwendung mit ihren Komponenten kommuniziert, ohne über

deren Position im Netz informiert zu sein. Sie können auf demselben Knoten liegen wie die Anwendung, aber auch entfernt. In einem Agentensystem wie AMETAS spielt der so genannte *Stellenname* eine entscheidende Rolle: Um einen Auftrag zu erfüllen, muss ein Agent möglicherweise mehrere verschiedene Stellen besuchen und dazu deren Namen kennen.

Im Netz- und Systemmanagement haben Agenten besonderes Interesse, eine bestimmte Stelle zu besuchen, welche auf dem zu überwachenden Rechner betrieben wird [ZHG99a]. In vielen Fällen liegt indes keine Notwendigkeit vor, eine bestimmte Stelle aufzusuchen. Die Motivation, Stellen anzulaufen, ergibt sich in der Regel aus der Notwendigkeit, mit gewissen Stellennutzern zusammenzutreffen, um mit diesen Daten auszutauschen. Bei bisherigen, geschlossenen Anwendungen gibt es eine implizite Festlegung, dass die aufzusuchenden Stellennutzern stets an festen Stellen zu finden sind, sodass eine Vermittlung nur an diesen Stellen erfolgreich wäre. Der Anwendungsprogrammierer ist also genötigt, Kenntnis von den Stellen zu haben, welche Teil der Agentenanwendung sind.

Dieses Erfordernis ist offenbar eine Einschränkung in Bezug auf die Offenheit von Anwendungen. Zwar muss die Menge der verwendeten Agenten nicht beschränkt sein, jedoch ist die Menge der beteiligten Stellen in vielen Situationen eingeschränkt. Um dies zu beheben, muss derzeit mittels Mechanismen des Stellennamendienstes zunächst eine Menge von Stellennamen beschafft werden, welche dann der Reihe nach besucht werden. Jede Anwendung des Agentensystems, die von einer unvorherbestimmten Verteilung von Stellennutzern ausgeht, müsste dieses Vorgehen in entsprechender Form praktizieren.

Die Kombination des global vermittelnden Typsystems mit dem Migrationsmechanismus ermöglicht die Implementierung eines *lokationstransparenten Migrationsmechanismus*, welcher es erlaubt, einen Stellennutzer als „Ziel“ anzugeben. Ein solcher Aufruf könnte in einem AMETAS-Agenten so aussehen:

```
public class MyAgent extends AMETASAgent {
   MeetingList m_List; // Teil des Zustands
   public void invoke() {
      ...
      m_List = new MeetingList();
      try {
         AMETASType type = m_Driver.createTypeForString(...);
         m_Driver.meet(type, m_List);
      }
      catch (MigrationException mx) {
      ... // Migrationsproblem
      }
      catch (PlaceUserNotFoundException px) {
      ... // Keinen geeigneten PU zum Treffen gefunden
```

```
      }
   }
}
```

Das Objekt *m_List* dient dazu, mögliche weitere Treffer dem Agenten zu übergeben, bevor dieser migriert. Der Aufruf von *meet* führt zu einer Migration und lässt keine Chance, die Liste möglicher Ziele zu inspizieren; der Treiber des Agenten könnte durch das Ablegen in *m_List* verhindern, dass diese Ergebnisse verloren gehen. Die Stellennutzer-ID des durch *type* charakterisierten Agenten sollte in dieser Liste ebenfalls vorhanden sein, sodass eine lokale Vermittlung nach Ankunft unnötig ist. Eine konzeptionell ansprechendere Lösung wäre die Beschaffung der Liste der zu treffenden Agenten sowie die Entscheidung innerhalb des Agentencodes für eine der Möglichkeiten.

```
MeetingList list = m_Driver.getMeetingList(type);
MeetingPoint[] amp = list.getMeetingPoints();
for (int i=0; i < amp.length; i++) {
   // Auswerten von amp[i]; bFound=true, wenn entschieden
   // ...
   if (bFound) {
      try {
         m_Driver.meet(amp[i]);
      }
      ...
   }
}
...
```

Dieses Vorgehen verbirgt zwar oberflächlich die Lokation des zu treffenden Stellennutzers, ist jedoch konzeptionell nicht unterschiedlich zu der Verwendung des lokationszentrierten *go*-Befehls, wie er schon verfügbar ist, da innerhalb *amp[i]* auch die Zielstelle enthalten sein muss (denn der Treiber benötigt noch immer die exakte Angabe der Zielstelle). Prinzipiell könnte man in diesem Falle auch ohne eine *meet*-Methode auskommen.

8.1.4 Meta-Vermittlung

Anstatt einen Vermittler zu konsultieren, welcher über eine Sicht über alle vermittelbaren Stellennutzer verfügt, ist auch eine *Föderation von Vermittlern* denkbar (siehe auch [ITU95]), welche – anders als oben beschrieben – nicht für einen Abgleich ihrer lokalen Datenbanken sorgen, sondern eine *Hierarchie von Vermittlern* definieren. Jeder Vermittler lässt sich dabei durch eine Eigenschaft charakterisieren, welche von einem

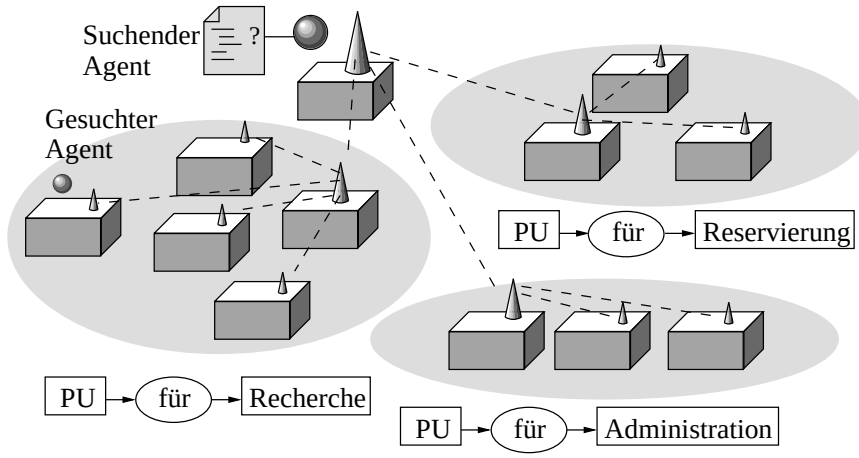

Abbildung 8.2: Hierarchische Vermittlung

Klienten direkt oder indirekt erfragt werden kann. Auf eine Anfrage kann ein Vermittler empfohlen werden, welcher zu diesem Teilgebiet der möglichen Anfragen genauere Informationen hat. Diese Vermittlung von Vermittlern nennt man auch *Meta-Trading* oder *Meta-Vermittlung*.

Die Meta-Vermittlung geht dabei nahtlos in die oben beschriebene globale Vermittlung über, wenn sich Vermittler auf die jeweilige Stelle beschränken und man einen übergeordneten Dienst als Ober-Vermittler einsetzt. Ein interessierter Agent wendet sich an diesen Ober-Vermittler und bekommt daraufhin eine passende Stelle empfohlen, auf welcher ein eigener Vermittler existiert. Dieser wiederum muss konsultiert werden, um den Ansprechpartner letztendlich zu finden.

Für gewisse Anwendungen wie dem elektronischen Marktplatz kann es Konventionen geben, dass dieser sich nur über eine gewisse Menge von Stellen ausbreitet, wobei man Zuständigkeiten definieren kann: In dieser Gruppe von Stellen findet man Internet-Suchdienste, in jener halten sich spezielle Fachbibliotheksdienste oder Reisereservierungsdienste auf. Ein Agent, welcher in diesen Marktplatz entsandt wird, kann sich dann an einen übergeordneten Vermittler wenden, welcher ihn automatisch in den richtigen Bereich verweist (Abbildung 8.2).[2] Anstelle von Diensten können durchaus auch Benutzeradapter vermittelt werden, etwa wenn die Agenten eingesetzt werden, um Benachrichtigungen durchzuführen.

Domänen kennzeichnen eine Menge von Stellen mit ähnlichem Einsatzgebiet. Eine Domäne von Stellen, an denen Administratoren arbeiten, könnte mit diesem Konzept einem Agenten auf Anfrage vermittelt werden. Um die entsprechende Domäne zu finden, muss der übergeordnete Vermittler in der Anfrage des suchenden Stellennutzers Hinweise finden, welcher Art der gesuchte Stellennutzer ist. Die jeweiligen Domänen

[2] Die in der Abbildung genannten Konzepte und Relationen sind verkürzt. PU steht für PlaceUser (Stellennutzer).

haben ihre eigenen Vermittler, welche sich beim Obervermittler bekannt machen, etwa durch den Eintrag eines in der Abbildung gezeigten Konzeptgraphen. Als Ergebnis der Anfrage kann der suchende Agent folgende Ergebnisse erhalten:

- den Identifikator des gesuchten Agenten inklusive der Stelle, wo er sich befindet; dies erfordert eine recht aufwändige Implementierung des Dienstes sowie weiterer, zugehöriger Komponenten;
- den Namen der Stelle, wo sich der nächste Vermittler befindet, welcher genauere Informationen über seine Domäne besitzt.

Die zweite Version erhöht den Implementierungsaufwand des suchenden Agenten. Je nach Situation ist abzuwägen, welche Strategie die günstigere ist.

8.2 Erweiterung des Typsystems

Die im letzten Abschnitt gezeigten Konzepte beziehen sich in erster Linie auf die Entwicklung geeigneter Dienste und erweiterter Mediatoren. Das Typsystem in der vorliegenden Version ist zwar bereits recht mächtig und erlaubt die Repräsentation einer Vielzahl von Typen, dennoch sind einige Möglichkeiten zur Erweiterung denkbar.

8.2.1 Probabilistische Angaben

Der Transitionstyp beinhaltet, wie in Abschnitt 5.5.5 beschrieben, eine recht genaue Beschreibung interner Zustandswechsel im Falle des Empfangs und des Sendens von Nachrichten. Der Agent kann dabei auf dieselbe Nachricht in einer von mehreren vorhersehbaren Weisen reagieren, wobei die Auswahl dieser Reaktion von außen nicht nachvollziehbar sein muss. Dies wird als *Nichtdeterminismus* der Aktionen des Agenten modelliert.

Nichtdeterministische Alternativen treten stets gleichberechtigt auf. Es ist anhand des Typs nicht entscheidbar, welche der Alternativen in der gegebenen Situation eher eintreten wird. Ferner kann nicht einmal gesagt werden, ob die angebotene Alternative *jemals* eintreten wird. So können sich Urteile ergeben, welche auf den ersten Blick nicht verständlich sind (Abbildung 8.3): Der unter ii) dargestellte Transitionstyp wird (bislang) vom Typsystem als Subtyp des Typs i) beurteilt. Dies ist aufgrund der Festlegungen, wie sie in Abschnitt 5.6.3 erläutert wurden, verständlich: Der Typ i) trifft keine Aussagen darüber, ob die Schleife am mittleren Zustand jemals verlassen wird. Deshalb reagiert Typ ii) zu jeder Zeit so wie erwartet, darf also als Subtyp bezeichnet werden.

Das Problem wird jedoch deutlich, wenn man die Schleife als eine *endliche* Schleife versteht. Dann verhält sich Typ ii) nicht wie ein Subtyp des Typs i), da er nie die

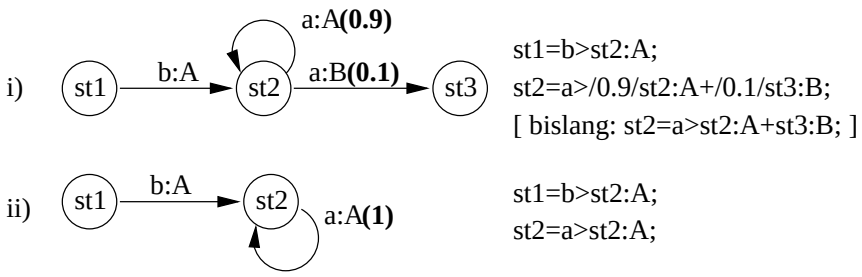

Abbildung 8.3: Probabilistische Alternativen

Schleife verlässt. Die Schleife kann als wiederholte Zusendung von Parametern interpretiert werden, welche der Agent irgendwann mit der Nachricht des Typs *B* quittiert, um auf weitere Nachrichten zu verzichten. Dies wird beim zweiten Agenten nie eintreten, was letztlich dazu führen kann, dass der Agent die vorgesehene Aufgabe nicht erfüllt.

Im Falle fester Schleifen (also etwa des Empfangs von fünf Nachrichten) kann man eine Modellierung ohne Schleifenkonstrukt erwägen, in der die Schleife *abgerollt* wird: Es werden die fünf Schleifendurchgänge als fünf Zustandsänderungen aufgefasst. Der letzte Zustand fordert dann explizit das Verlassen dieser endlichen Schleife durch die Akzeptanz einer neuen Nachricht. In diesen Fällen könnte man aber, um Zustände zu sparen, auf komplexere Nachrichten zurückgreifen, welche jene fünf Einzelnachrichten zu einer zusammenfassen.

Ist die Schleifengrenze nicht vom Typ her bestimmt – wie es in der Hybridtypbeschreibung angenommen wird –, kann eine Schleife nicht abgerollt werden. Das zugesicherte Verlassen der Schleife ist *nicht modellierbar*. Hier könnte eine Erweiterung des Typsystems Abhilfe schaffen, die als *probabilistische Übergänge* bezeichnet werden kann. Probabilistische Automaten sind endliche Automaten, deren Übergänge mit Wahrscheinlichkeiten ausgestattet sind [Paz66]. Diese Wahrscheinlichkeiten bieten nützliche Informationen an Stellen im Graphen an, an denen Nichtdeterminismen auftreten. Abbildung 8.3 zeigt, wie die Graphen mit Wahrscheinlichkeiten ausgestattet werden können. Die Schleife wird gemäß der Abbildung i) in 90 Prozent aller Fälle fortgesetzt; in 10 Prozent der Fälle wird sie abgebrochen. Teil ii) der Abbildung beschreibt, dass die Schleife in 100 Prozent der Fälle fortgesetzt wird. Die nichtverschwindende Wahrscheinlichkeit der Fortsetzung könnte als Anhaltspunkt zu behaupten, dass sich unter ii) kein Subtyp von i) findet, verwendet werden.

Die Wahrscheinlichkeiten können darüber hinaus dazu dienen, weitere semantische Informationen zu erfassen. Ein Vergleich der Wahrscheinlichkeiten lässt sich so entwerfen, dass eine Metrik auf den Wahrscheinlichkeiten definiert wird, welche die Abweichungen der sich entsprechenden Wahrscheinlichkeiten misst. Schwierig kann sich die Berechnung oder Abschätzung der beteiligten Wahrscheinlichkeiten erweisen. Das obige Beispiel suggeriert zwar, dass man die Anzahl der Schleifendurchläufe als

solche verwenden kann, dies deckt sich jedoch nicht mit der gebräuchlichen Semantik des Begriffs Wahrscheinlichkeit.

8.2.2 Gebundene Variablen

Die im syntaktischen Typ eingesetzten Nachrichten besitzen untereinander keine Beziehung; erst durch die Verknüpfung innerhalb des Transitionstyps sind Aussagen über die Abfolge der Nachrichten möglich – etwa, um eine Nachricht als *Antwort* auf eine *Anfrage* zu identifizieren. Jede Nachricht besteht ihrerseits aus einzelnen Elementen. Diese sind innerhalb des Typsystems völlig voneinander unabhängig; es gibt weder implizite noch explizite Verbindungen zwischen ihnen. Es ist möglich, verschiedenen Nachrichtenelementen dieselbe Marke zuzuweisen; diese erhalten dann beide eine Referenz auf dieselbe Annotation, wenn diese vorhanden ist. Dies wirkt sich jedoch nicht anders aus, als wenn die Annotation dupliziert würde.

Als Beispiel betrachte man die in Abbildung 8.4 dargestellte Situation. Die Typbeschreibung des Agenten sehe folgendermaßen aus:

```
messages {
   in {
      migReq: { plname:java.lang.String };
   }
   out {
      migrate: { migration, plname:java.lang.String };
   }
}
states {
   st0 =migReq> st1:migrate(place);
}
annotation {
   plname=CG:{
      [Bezeichnung]-> (”gehört zu”)->[Stelle]
   };
}
```

Der Agent würde somit bei Empfang der Nachricht *migReq*, die nur aus dem Namen einer Stelle besteht, eine Migration durchführen, indem er den Pseudodatentyp *migration* zusammen mit einer Zeichenkette als eine Nachricht an die aktuelle Stelle schickt. Gemeint ist offenbar, dass der Agent genau zu jener Stelle migriert, welche in der Eingangsnachricht bezeichnet wurde. Dies ist jedoch anhand der Typbeschreibung nicht ersichtlich. So könnte es sich bei dem übermittelten Stellennamen um den Namen einer Stelle handelt, zu welcher er am Schluss seiner Reise migrieren soll; die in seiner Migrationsnachricht verwendete Bezeichnung kann aus seiner eigenen Implementierung oder (wenn es sich hier nur um einen Ausschnitt der Typbeschreibung handelt)

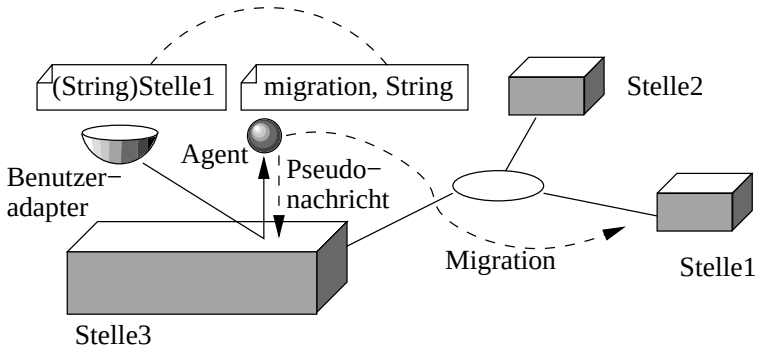

Abbildung 8.4: Bindung von Variablen

aus einer früheren Interaktion stammen. Die gegenseitige Abhängigkeit der Nachrichtenelemente wird als *Bindung* bezeichnet. Analog zu der Bedeutung im Funktionenkalkül würde hier gelten, dass der Wert des Datentyps in der Nachricht *migrate* vom Wert des Datentyps in der Eingangsnachricht *migReq* abhängt. Als Variable gesehen ist *migrate.plname* also nicht *frei.*

Auch wenn diese Eigenschaft zunächst wünschenswert und mit geringem Aufwand zu implementieren erscheint, erlaubt das Hybridtypsystem zumindest in der dieser Arbeit zu Grunde liegenden Version keine derartige Bindung:

1. Auch wenn bei einem solchen Vorgehen keine konkreten Werte vorgeschrieben würden, handelt sich um eine wertbezogene, syntaktische Konstruktion. Eine Typbeschreibung sollte nicht durch Werte parametrisiert sein.

2. Es ist nicht klar, welches Nachrichtenelement an welches gebunden ist. So könnte festgelegt werden, dass Elemente von Ausgaben nur an Elemente von Eingaben gebunden sein können. Damit wird jedoch wieder das klassische Klient-Server-Prinzip eingeführt: Der Agent nimmt Daten entgegen, welche er zu verarbeiten hat. Es sind Situationen vorstellbar, dass Elemente der Ausgabe eines Agenten in einer darauf folgenden Eingabe wieder erwartet werden – als Bestätigung. Diese Bindung von Eingaben zu Ausgaben kann im Allgemeinen zu Fehlinterpretationen führen.

3. Die Bindung führt zu viele implementationstechnische Bedingungen ein. Es ist, wie oben schon angedeutet, durchaus vorstellbar, dass die eintreffende Information für später aufbewahrt wird und erst im Laufe der Abarbeitung des Auftrags zum Einsatz kommt. Dies hängt von der aktuellen Situation in der Umgebung des Agenten und von seinem Zustand ab.

Aus diesen Gründen wurde auf das Konzept einer Bindung verzichtet. Sie könnte dem System so hinzugefügt werden, dass die Nachrichtenelemente eine bestimmte (Konstanten-)Annotation erhalten, welche die Bindung symbolisiert.

8.3 Computergestützte Agentenentwicklung

In den vergangenen Kapiteln wurde beschrieben, wie ein Typsystem für Agenten eingesetzt werden kann, um Aufgaben wie Vermittlung oder Konformitätsprüfung durchzuführen. Dabei ging man von der Vorstellung aus, dass die Typbeschreibung erst nach der Erstellung des Agenten entsteht und diesem sozusagen *angeheftet* wird. Zum Abschluss soll eine weitere Möglichkeit des Einsatzes von Typen ins Auge gefasst werden: die Verwendung des Typs als *Bauanleitung* für einen Agenten.

Die computerunterstützte Codegenerierung genießt bereits ein weites Einsatzfeld. Insbesondere im Umfeld der Verteilungsinfrastrukturen wie CORBA werden Compiler eingesetzt, um notwendige Komponenten der Anwendung zu generieren, welche Routineaufgaben übernehmen. Beispiele hierfür sind die *Stubs* und *Skeletons* bei CORBA.

8.3.1 Stub-Compiler

Ein *Stub* ist eine Softwarekomponente, welche die Lokation eines Objekts innerhalb eines verteilten Systems verbirgt, indem sie die an sie gerichteten Anfragen wie ein Stellvertreter entgegennimmt und in eine geeignete Form bringt, welche an die jeweilige Verteilungsinfrastruktur angepasst ist. Auf der Klientenseite entsteht für den Klienten der Eindruck, das von ihm angesprochene Objekt befinde sich in seinem Adressraum. Es wird somit durch Kapselung von Code innerhalb der Stubs eine *Lokationstransparenz* erreicht. *Skeletons* übernehmen die Aufgabe des Klienten auf der Seite des Dienstobjekts. Auch in diesem Falle ist die Implementierung vollständig durch die Schnittstelle des Objekts bestimmt.

Die Aufgabe von Stubs ist somit recht einfach umrissen und ihre Generierung in nahe liegender Weise automatisierbar. Diese Implementierung wird durch so genannte *Stub-Compiler* geleistet. Informationen, welche Funktionalität das anzusprechende Dienstobjekt und damit auch der Stub anzubieten hat, gewinnt der Compiler aus der *Schnittstellenbeschreibung* des Dienstes. Dazu genügt es in der Regel zu erfahren, welche Methoden von dem fraglichen Objekt angeboten werden; diese werden durch Signaturen spezifiziert.

Die Verwendung von Schnittstellen und Stub-Compilern ist typisch für Systeme, welche die Implementierung von Objekten auf einer verteilten Infrastruktur erlauben. Beispiele sind CORBA und auch Java RMI. Abbildung 8.5 zeigt ein Beispiel einer RMI-Anwendung. Während der Anwendungsprogrammierer sich lediglich um die Erzeugung der Klassen *Klient, RemHelloImpl* sowie der Schnittstelle *RemHello* kümmern muss, sorgt der RMI-Compiler *rmic* für die Erzeugung der grau markieren Klassen, welche den Stub und das Skeleton darstellen. Die minimale Funktionalität dieser Klassen ist bereits durch die Schnittstelle sowie der sie implementierenden Klasse eindeutig bestimmt; sie können also automatisch generiert werden.[3]

[3] Die Klassen *RemoteStub* und *Skeleton* sind Bestandteile des Java-Pakets *java.rmi.server.*

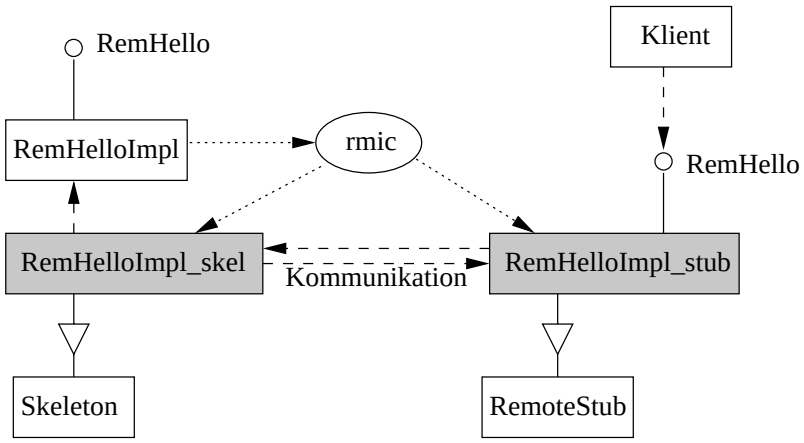

Abbildung 8.5: RMI-Anwendung

8.3.2 Codeschablonen

Codeschablonen dienen zur Erleichterung der Implementierung des Dienstobjekts. Objekte, welche eine gegebene Schnittstelle implementieren, müssen die gleichen Methodensignaturen wie jene Methoden in der Schnittstellenbeschreibung aufweisen. Ein Stub-Compiler stellt deshalb neben den Stubobjekten bei Bedarf Vorlagen zur Implementierung von Objekten her, welche die ihm übergebene Schnittstelle implementieren. Diese Vorlagen beinhalten leere Methodenrümpfe, sind aber zumeist schon syntaktisch korrekt, können also als eine triviale Implementierung dienen.

Im Falle vom *Java RMI* sind Codeschablonen nicht erforderlich, da die Erzeugung der unterstützenden Klassen anhand der Implementierung vorgenommen wird. CORBA-Implementierungen wie *Orbix* von *IONA* erlauben die Erzeugung von Vorlagen bei einer gegebenen Schnittstelle. Der Orbix-IDL-Compiler generiert bei Aufruf mit der Option *-s* Codeschablonen mit leeren Methodenrümpfen. Aus der IDL-Datei *bank.idl* wird die Schablone *bank.iC* generiert, wobei die Methoden die korrekte Signatur aufweisen. Der Implementierer muss lediglich den Inhalt der Methoden sowie Konstruktoren, Destruktoren und Objektfelder hinzufügen.

8.3.3 Generierung des Stellennutzer-Quellcodes

Die im Folgenden beschriebene automatische Codegenerierung ist lediglich als Skizze zu verstehen; eine Implementierung wurde noch nicht vorgenommen. Ziel soll es sein, eine Typbeschreibung in textueller (oder instantieller) Form an den Codegenerator zu geben, welcher ein Codegerüst erzeugen soll. Dieses ist vom Implementierer geeignet zu füllen, um die vorgesehene Funktionalität zu erlangen.

Die Typbeschreibung in der hier vorgestellten Form umfasst Angaben zur Syntax, zum Protokoll und zur Semantik in Form von Annotationen. Annotationen beinhalten

in der Regel semantische Informationen in Bezug auf eine Nachricht, ein Nachrichtenelement, einen Zustand oder Absendern/Empfängern. Diese eignen sich offenbar nicht für die Erzeugung einer Implementationsschablone. Die hierfür interessanten Bestandteile der Typbeschreibung sind der syntaktische Typ und der Transitionstyp. Als deren Elemente liegen Zustände, Nachrichten, deren Elemente sowie Angaben zu Absendern/Empfängern vor, wobei insbesondere von Interesse ist, ob die Daten vom initialen Absender oder von anderen Objekten stammen.

Da der Zustand eines Objekts dessen Verhalten bei Empfang einer Nachricht beeinflusst, liegt nahe, die Implementierung des Agenten auf dessen Zustände zu stützen. Für jeden aktuellen Zustand muss es einen Codeteil geben, welches das zugehörige Verhalten implementiert. Anhang F demonstriert die Generierung einer Implementationsschablone gemäß den folgenden Richtlinien anhand eines einfachen Beispiels.

Grobstruktur der Implementierung

Die grundlegende Struktur der Implementierung des Stellennutzers wird anhand der Zustände definiert. Dazu dienen folgende Schritte:

- Einführung einer globalen Variablen, welche den aktuellen Zustand speichert.
- Definition geeigneter Konstanten, welche die Zustände repräsentieren.
- Erzeugung eines Verzweigungskonstrukts zur Auswahl des auszuführenden Codes in Abhängigkeit vom Zustand.
- Erzeugung privater Methoden, welche den Code kapseln, der im jeweiligen Zustand zur Ausführung kommen soll.

Diese Zustandsvariable kann vom Datentyp *short* sein, da die Anzahl unterschiedlicher Zustände als gering angenommen werden kann. Die Konstanten erhöhen die Lesbarkeit des Codes, ebenso ist es vernünftig, den Code in verschiedene Methoden aufzuteilen. Das Verzweigungskonstrukt muss wiederholt durchlaufen werden, bis ein Endzustand erreicht ist.

Bereitgestellte Felder und Initialisierung

Für den initialen Kommunikationspartner sowie jeden annotierten Sender und Empfänger werden Felder zur Aufnahme der jeweiligen Stellennutzer-IDs vorgesehen. Zur Aufnahme der Eingangsnachrichtentypen müssen entsprechende Felder bereitgestellt werden.

Der Konstruktor des Stellennutzers sollte die Initialisierung des Zustands auf den Anfangszustand beinhalten. Ferner kann der Konstruktor genutzt werden, um Felder zu initialisieren, welche in verschiedenen Zuständen benötigt werden.

Mehrere *OTHER*-Annotationen sollten auf unterschiedliche Felder abgebildet werden. Die Felder für die Eingangsnachrichtentypen können zur Aufnahme von *MessageType*-Instanzen oder von Zeichenketten aus dem Typquellcode dienen, welche zur Laufzeit zur Generierung von *MessageType*-Instanzen verwendet werden.

Implementierung der Zustandsmethoden

Für jeden Zustand des Agenten wird eine Methode vorgesehen, welche in der Hauptschleife der *invoke*-Methode in Abhängigkeit vom Zustand aufgerufen wird.

Eine Tabelle kann dazu dienen, die jeweiligen Stellennutzer-IDs der Kommunikationspartner zu speichern. Neben den unterschiedlichen Kommunikationspartnern ist eine getrennte Speicherung von Zuständen notwendig, damit der Zustand sich für jeden Kommunikationspartner vorhersehbar ändert. Die hier vorgestellte, einfache Codegenerierung geht zunächst nicht von diesem Fall aus.

Entgegennahme von Nachrichten

Ist der Zustand stabil oder semistabil, muss die nächste Nachricht aus der Eingangswarteschlange abgeholt werden. Der Typ der Nachricht muss bestimmt werden, um die aktuelle Transition zu bestimmen. Ist der Absender in der Beschreibung angegeben, so ist dieser anhand der Stellennutzer-ID im Absenderfeld der Nachricht zu überprüfen.

Zustandsmethoden, welche einen Zustand mit nichtdeterministischer Auswahl von Folgetransitionen implementieren, führen jeweils eine Variable ein, welche innerhalb eines Verzweigungskonstrukts eingesetzt wird. Jede Verzweigung steht für eine dieser Alternativen.

Im Startzustand muss der Absender der initialen Nachricht bestimmt werden, welcher fortan als Kommunikationspartner angesehen wird. Dabei ist wichtig zu entscheiden, ob der Stellennutzer mit mehreren Partnern zugleich kommunizieren soll. Wie in Abschnitt 6.5 beschrieben wurde, kann das Instantiieren als initiale Nachricht verstanden werden; der Agent sollte in der Lage sein zu erfahren, welches Objekt ihn gestartet hat und würde dieses Objekt dann als initialen Kommunikationspartner ansehen.

Analyse der Transitionen in jedem Zustand

Wenn der aktuelle Zustand *stabil* ist, wird ein Verzweigungskonstrukt mit einer Verzweigung für jeden Eingangsnachrichtentyp erstellt. Jede Verzweigung kann ihrerseits durch ein zustandsabhängiges Verzweigungskonstrukt unterteilt sein, um verschiedene nichtdeterministische Alternativen auszuwählen. Ist der Zustand *instabil*, so wird ein zustandsabhängiges Verzweigungskonstrukt aufgebaut, um die Alternativen auszuwählen. Für *semistabile* Zustände gilt, dass die Nachrichtentyp-orientierte Verzweigung als Teil eines umfassenden Verzweigungskonstrukts implementiert wird.

Durch diese Implementierung stellen sich semistabile Zustände als Kombination eines instabilen mit einem stabilen Zustand dar, wie es in Abschnitt 5.5.6 bereits nahe gelegt wurde. Der *stabile Subzustand* wird als eine mögliche Alternative eingenommen.

Man beachte, dass keine Annotationen verwendet werden. Das heißt, dass die Überprüfung der Nachrichtendiskriminatoren im Einzelfall zu implementieren ist. Es ist jedoch vorstellbar, dass auch hier ein Automatismus ansetzen kann, diese Konstantenannotationen auf eine entsprechende Verzweigung abzubilden.

Bevor der neue Zustand eingenommen wird, müssen gegebenenfalls Ausgaben vorgenommen werden. Der Inhalt der Nachrichten ist von der jeweiligen Anwendung abhängig. Die Änderung der Zustandsvariablen ist erst von Bedeutung, wenn der Kontrollfluss in die *invoke*-Methode zurückkehrt und dort die Hauptschleife fortführt.

Zustandsänderungen und Ausgaben

Jede Methode vollzieht die in der Typbeschreibung angegebenen Zustandsänderungen durch Modifikation der globalen Zustandsvariablen. Eine Zustandsmethode, welche das Verhalten in einem Endzustand repräsentiert, sollte eine Testvariable setzen, welche die Beendigung der Hauptschleife signalisiert. Man muss sich im Klaren sein, ob der Stellennutzer am Ende des Protokolls terminieren soll oder nicht. Falls dies nicht der Fall sein sollte und der Stellennutzer zum Anfangszustand zurückkehrt, sollte dies auch in der Typbeschreibung deutlich werden.

Werden Ausgaben vorgenommen, sollten die erforderlichen Anweisungen zum Abschicken der Nachricht bereitgestellt werden. Der Nutzlastteil der Nachricht kann dabei bereits in seiner Größe vorbereitet werden. Gegebenenfalls können Felder des Nutzlastteils initialisiert werden. Der Empfänger der Nachricht bestimmt sich aus dem Absender der initialen Nachricht oder durch die Empfängerannotations-Referenz.

8.3.4 Eigenschaften des generierten Codes

In der bisher vorgestellten Weise der Anwendung des Typsystems existieren die zu typisierenden Objekte bereits, wenn die Beschreibung erstellt wird. Die Implementierung geschieht dann möglicherweise ohne Rücksicht auf Richtlinien der Typisierung (siehe Abschnitt 7.4). Die Erstellung der Typbeschreibung wird dadurch erschwert und die Wahrscheinlichkeit der erfolgreichen Vermittlung sinkt.

Ferner kann es vorkommen, dass die Typbeschreibung fehlerhaft ist. Abgesehen von der inhärenten Problematik der Formalisierung von semantischen Informationen kann sich die Feststellung von Zuständen äußerst schwierig gestalten, wenn die Implementierung die Zustandseinteilung des Typs nicht widerspiegelt. So könnte schon eine einfache *For*-Schleife Anlass zur Einführung mehrerer Zustände geben, die als solche nicht offen zu Tage treten. Zustandsübergänge sind meist nur *implizit* im Code repräsentiert.

Wird eine automatische Codegenerierung eingesetzt, wie in diesem Abschnitt vorgestellt wird, so weisen die hieraus entstehenden Stellennutzer von sich aus eine Struktur auf, welche ziemlich genau der durch den Typ vorgegebenen Struktur entspricht. Die Typbeschreibung trifft insbesondere auf solche Stellennutzer präzise zu, bei denen sich der Implementierer an die Vorgaben gehalten hat und keine eigenmächtigen, der Beschreibung nicht entsprechenden Änderungen vorgenommen hat. Insofern ist die Entwicklung von Agenten mittels einer solchen Codegenerierung keine Versicherung für eine zutreffende Typbeschreibung, aber eine mögliche, unbeabsichtigte Fehltypisierung kann verhindert oder zumindest unwahrscheinlicher werden.

8.4 Zusammenfassung

In Kapitel 7 wurden Anwendungen des Hybridtypsystems vorgestellt, welche die Funktionalitäten nutzen, die vom Typsystem und der Implementierung gebotenen werden. Bislang ist nur eine lokale Vermittlung vorgesehen; in diesem Kapitel wurde skizziert, wie eine globale Vermittlung von Stellennutzern gestaltet werden könnte. Diese Erweiterung kann im Wesentlichen auf Anwendungsebene erfolgen, so durch das Erstellen geeigneter Dienste.

In diesem Kapitel wurden weiter gehende Eigenschaften vorgestellt, welche die Implementierung noch nicht leistet, die dieser Arbeit zu Grunde liegt. Die Hinzunahme probabilistischer Angaben ist eine echte Erweiterung des bestehenden Typsystems, indem sie den bisherigen Beschreibungen weitere Attribuierungen in Form von Transitionswahrscheinlichkeiten zuweist, welche nichtdeterministische Alternativen zu unterscheiden erlauben. Man muss jedoch beachten, dass die Einführung von Wahrscheinlichkeiten bedeutende Folgen für die Sichtweise auf die Subtyprelation hat: Kann die Subtypeigenschaft abgesprochen werden, nur weil der betrachtete Typ eine Alternative seltener wählt? Diese Fragestellung erfordert eine eigene, tief greifende Erforschung.

Die Bindung von Variablen ist eine weitere Möglichkeit, den Nachrichtenaustausch zu charakterisieren. Allerdings bleibt zu bedenken, dass eine gegenseitige Bindung von Nachrichten(elementen) bedeutende Auswirkungen auf die Implementierung haben kann; eine Verarbeitung von empfangenen Daten würde a priori ausgeschlossen. Anwendung könnte eine solche Erweiterung finden, wenn die beteiligten Partner recht einfach gestaltet sind, beispielsweise Botenagenten, die nur eine Nachricht transportieren sollen.

Schließlich wurde ein Verfahren skizziert, wie anhand einer gegebenen Typbeschreibung eine Codeschablone generiert werden kann, welche insbesondere die Zustandsübergänge beinhaltet und vom Implementierer lediglich mit aufgabenspezifischem Code gefüllt werden muss. Dieser Ansatz folgt dem Beispiel der bekannten IDL-Compiler, welche Codeschablonen generieren können, um die Implementierung von CORBA-Dienstobjekten zu unterstützen.

9 Zusammenfassung

Mobile und autonome Softwareagenten sind ein recht junges Forschungsgebiet der Informatik, das besonderes Interesse auch in außerakademischen Bereichen findet. Das den Agenten zu Grunde liegende Konzept ist einfach, aber wirkungsvoll: Agenten sollen als persönliche Assistenten fungieren; sie sollen Aufgaben verrichten, welche vom Anwender großen Aufwand erfordern würden, so zum Beispiel das Management eines weitläufigen Netzwerks, welches zeitweise partitioniert sein kann. Andere Beispiele sind das Zusammentragen von Informationen nach einem bestimmten Muster, etwa die Recherche nach bestimmten Waren oder Schriftstücken im Internet. Sie können den Benutzer beobachten, ihm Lernhilfen geben und dabei möglicherweise einen menschlichen Ausbilder wenn nicht ersetzen, so doch wirkungsvoll unterstützen.

Ein Überblick über verschiedene Agentensysteme, wie er in Kapitel 2 angeboten wurde, offenbart unterschiedliche Sichtweisen auf Agenten. Konsens besteht in der Zusicherung der Autonomie der Agenten, wenn auch diese mit unterschiedlichem Nachdruck realisiert wird. Das Agentensystem AMETAS setzt das Gebot der Autonomie an vorderste Stelle und führt sämtliche Kommunikation auf den asynchronen Nachrichtenaustausch zurück.

Um die Entwicklung und den Betrieb von Multiagentenanwendungen zu unterstützen, ist eine Beschreibung eines Agenten sowie aller an der Kommunikation beteiligten Komponenten hilfreich. Eine solche Beschreibung kann einerseits Aufschluss darüber geben, ob dieser Agent in der Lage ist, gemäß einem bestimmten Protokoll zu kommunizieren; andererseits kann sie Informationen über den Einsatzzweck des Agenten anbieten. So können Agenten in einfacher Weise weiterentwickelt werden, ohne ihre bisherige Funktionalität zu verlieren, genauso wie in der objektorientierten Welt Klassen abgeleitet werden können. Endanwender sind in der Lage, ihre Anwendungen selbst zusammenzustellen, indem sie anhand einer Beschreibung einen geeigneten Agenten auswählen können oder von einem Programm geeignet in der Auswahl assistiert werden können.

Bisherige Beschreibungen bieten meist nur Schnittstellenbeschreibungen an, welche anhand von *Signaturen* angeben, welche Datentypen in der Eingangsnachricht erwartet werden und welche Typen die Antwort bestimmen. Für Agenten ist eine solche Beschreibung nur eingeschränkt von Nutzen: Interaktionen zwischen Agenten können wesentlich *symmetrischer* ablaufen, als man es bei gewöhnlichen Objekten kennt. Eine Charakterisierung eines Agenten anhand einer Aufrufschnittstelle würde somit die

Möglichkeiten der Interaktion deutlich einschränken. Von den bisherigen Sichtweisen der Klienten- und Serverrolle muss Abstand gewonnen werden.

Andererseits sind bisherige Verfahren zur Typisierung sehr auf die Bindung von Namen zu Typen fixiert. Typen werden anhand von Namen referenziert. Es ist erforderlich, sicherzustellen, dass diese Bindung *konsistent im gesamten System* vorliegt. Dies ist für konventionelle Programmiersprachen oder für die verbreiteten, verteilten Objektsysteme möglich; Agentensysteme hingegen sind von ständiger Fluktuation geprägt: Agenten betreten das System, wandern zwischen verschiedenen Stellen, verlassen das System. Ein diesem Phänomen ausgesetztes Typsystem für Agenten sieht sich dem Problem gegenüber, Informationen über Agenten so schnell zu verbreiten, dass diese stets vorliegen, sobald der Agent eine Stelle erreicht. Eine zentrale Registrierung ist keine sinnvolle Lösung, da einerseits diese Registrierung mit Definitionen angefüllt wird, welche nur kurzfristig und lokal gültig sind, andererseits eine potenzielle zentrale Fehlerquelle *(single point of failure)* geschaffen wird.

Diese Arbeit stellt ein neues Konzept vor, das sich in erster Linie für die Charakterisierung von Agenten, aber auch anderer Objekte eignet, wenn diese über eine eigene Ablaufkontrolle verfügen. Die Grundlagen dieser Beschreibung, welche geeignet ist, einen *Typ* auf Agenten zu definieren, ist die Komposition aus drei einzelnen, miteinander verwobenen Sichtweisen:

- Syntaktische Charakterisierung: Welche Nachrichten nimmt der Agent entgegen? Welche Nachrichten versendet er?
- Transitionelle Charakterisierung: Welche Zustände nimmt der Agent ein? In welcher Beziehung stehen die Nachrichten zueinander?
- Semantische Charakterisierung: Welche Bedeutung haben die Nachrichten, deren Elemente, die Zustände und der gesamte Agent?

Die Typbeschreibung eines Agenten stellt sich in zweierlei Form dar: *textuell* zur Bearbeitung, für den menschlichen Anwender lesbar, und als *Instanz* für die Verarbeitung im Agentensystem. Es muss gesichert sein, dass die Typbeschreibung des Agenten allerorts verfügbar ist; bei einem Agentensystem wie AMETAS bietet sich an, die Beschreibung als Teil des Datenpakets vorzusehen, das bei Migrationen den zu versendenden Agenten beinhaltet.

Der Agententyp ist nur *ad hoc* interpretierbar. Er nimmt keinen Bezug auf andere Typen durch Bezeichner oder andere Referenzen. Zwei Typen können miteinander verglichen werden, indem geprüft wird, wie sich der gebotene Agent während verschiedener Interaktionen verhält. Die wesentliche Bedingung an einen Subtyp ist, dass man ihn jederzeit anstelle des Typs verwenden kann. Es wurde demonstriert, dass diese Bedingung eingehalten werden kann und die Bezeichnung *Typ* dieser Charakterisierung zu Recht verliehen ist.

Nicht nur Anwender können vom Einsatz dieser Typbeschreibung profitieren, indem sie sich einen für ihre Bedürfnisse geeigneten Agenten vermitteln lassen; auch der Entwickler wird unterstützt, indem er Informationen erhält, welche Interaktionen möglich sind und ob ein weiter entwickelter Agent problemlos anstelle einer früheren Version eingesetzt werden kann. Schließlich wurde dargelegt, wie eine mögliche Fortentwicklung des Systems im Rahmen des computerunterstützten Entwurfs von Software (CASE) in Bezug auf Agenten aussehen könnte.

Das Typsystem zur Vermittlung autonomer Softwareagenten ist allgemein genug gehalten, um eine Vielzahl von Ansätzen für den Entwurf von Agentensystemen zu unterstützen. Es trifft verhältnismäßig wenige Annahmen in Bezug auf die zu typisierenden Objekte. Schon in AMETAS selbst wird demonstriert, dass auch Objekte, welche nicht zu den Agenten zu zählen sind, durch eine Typbeschreibung charakterisiert werden können. Die grundlegenden Bedingungen sind

- die Existenz eines ordnungserhaltenden und verlässlichen Datenaustausches zwischen den Objekten;
- die Fähigkeit der Objekte, Nachrichten verschiedener Quellen zu separieren;
- die Modellierbarkeit eines Objekts als Entität, welche durch Datenaustausch mit der Umwelt interagiert.

Nicht erforderlich sind Eigenschaften der Objekte wie *Mobilität* oder *Autonomie*. Diese sind für die Modellierung von Agenten relevant, nicht jedoch für den Austausch von Nachrichten zwischen allgemeinen Objekten. Die Bedingung, Nachrichten separieren zu können, ist die am schwersten wiegende Bedingung, welche zugleich einfache Ansätze wie Methodenaufrufe ausschließt, da diese ohne besondere Vorkehrungen keine Feststellung des Senders erlauben.

Das Typsystem ist offen gestaltet; die verwendeten Ontologien können jederzeit ersetzt oder ergänzt werden. Anwendungen des Agentensystems mit dem Typsystem werden zu möglichen Neudefinitionen dieser Ontologien führen, wenn auch die allgemeine Struktur der Typbeschreibung für die bislang absehbaren Anwendungen hinreichen sollte.

Möglicherweise ergeben sich durch den hier vorgestellten Ansatz weitere Einsatzgebiete für agentenbasierte Anwendungen. Das Typsystem bietet die Möglichkeit, Anwendungen so zu gestalten, dass Komponenten aus verschiedenen Quellen mitwirken können; bei Bedarf können sie zur Laufzeit ersetzt werden. Kooperationen können sich spontan ergeben, wenn sich Agenten an einer Stelle treffen. Eine Programmierung von Agentenanwendungen muss dann zunehmend den Autonomiecharakter der Agenten beachten, ihnen ein gewisses Maß an *Handlungsfreiraum* zugestehen, sodass sie die Fähigkeiten anderer Agenten zur Erfüllung ihres Auftrags nutzen können. Eine scharfe Trennung zwischen mobilen und intelligenten Agenten zu ziehen erscheint in solchen

Szenarien verfehlt. Die Arbeiten aus den verschiedenen Disziplinen der Informatik tragen gemeinsam zum Fortschritt einer Technologie bei, die das Grundgerüst zukünftiger Anwendungen bilden wird.

A Formale Notation der Typbeschreibung

Die der Typbeschreibung zu Grunde liegende Syntax lässt sich in der *Backus-Naur-Form* (BNF) formulieren. Zur Vereinfachung der Notation steht das Symbol ε für das leere Wort. Die in BNF verwendeten Metazeichen sind die beiden Winkelklammern ⟨ und ⟩, die Definition ::= und die Alternative |. Übrige Zeichen wie Kommas, Klammern und Semikola gehören der Typbeschreibung an.

A.1 Syntax

⟨Typ⟩ ::= ⟨Typkomponenten⟩
⟨Typkomponenten⟩ ::= ⟨Typkomponente⟩ | ⟨Typkomponente⟩ `;` ⟨Typkomponenten⟩
⟨Typkomponente⟩ ::= ⟨SynTyp⟩ | ⟨TranTyp⟩ | ⟨SemTyp⟩

⟨SynTyp⟩ ::= `messages {` ⟨InBlock⟩ ⟨AusBlock⟩ `}`
⟨InBlock⟩ ::= ε | `in {` ⟨Nachrichten⟩ `} ;`
⟨AusBlock⟩ ::= ε | `out {` ⟨Nachrichten⟩ `} ;`
⟨Nachrichten⟩ ::= ⟨Nachricht⟩ | ⟨Nachricht⟩ ⟨Nachrichten⟩
⟨Nachricht⟩ ::= ε | `{` ⟨Elemente⟩ `} ;` | ⟨NMarke⟩ `: {` ⟨Elemente⟩ `} ;`
⟨NMarke⟩ ::= ⟨Zeichenkette⟩
⟨Elemente⟩ ::= ⟨Element⟩ | ⟨Element⟩ `,` ⟨Elemente⟩
⟨Element⟩ ::= ⟨Datentyp⟩ | ⟨EMarke⟩ ⟨Datentyp⟩
⟨Datentyp⟩ ::= ⟨Zeichenkette⟩ ⟨Dimension⟩
⟨EMarke⟩ ::= ⟨Zeichenkette⟩
⟨Dimension⟩ ::= ε | `[]` ⟨Dimension⟩

⟨TranTyp⟩ ::= `states {` ⟨Transitionen⟩ `}`

⟨Transitionen⟩ ::= ⟨Transition⟩ ; | ⟨Transition⟩ ⟨Transitionen⟩

⟨Transition⟩ ::= ⟨Zustand⟩ = ⟨PNachRef⟩ > ⟨Ziele⟩ ;

⟨Zustand⟩ ::= ⟨Zeichenkette⟩

⟨PNachRef⟩ ::= none | ⟨NachRef⟩

⟨NachRef⟩ ::= ⟨SEMarke⟩ ⟨NMarke⟩

⟨SEMarke⟩ ::= ε | (⟨AMarke⟩)| (any) | (other) | (place)

⟨Ziele⟩ ::= ⟨Ziel⟩ | ⟨Ziel⟩ + ⟨Ziele⟩

⟨Ziel⟩ ::= ⟨Zustand⟩ ⟨Ausgaben⟩

⟨Ausgaben⟩ ::= ε | : ⟨RefListe⟩

⟨RefListe⟩ ::= ⟨NachRef⟩ | ⟨NachRef⟩ , ⟨RefListe⟩

⟨SemTyp⟩ ::= annotations { ⟨Annotationen⟩ }

⟨Annotationen⟩ ::= ⟨Annotation⟩ | ⟨Annotation⟩ ⟨Annotationen⟩

⟨Annotation⟩ ::= ε | ⟨AMarke⟩ = S: { ⟨Strdek⟩ } ; | ⟨AMarke⟩ = CG: { ⟨CGDek⟩ } ;

⟨AMarke⟩ ::= ⟨Zeichenkette⟩

⟨StrDek⟩ ::= ⟨Wörter⟩ | ⟨Wörter⟩ : ⟨Wörter⟩

⟨Wörter⟩ ::= ⟨ZKetteOhneDP⟩ | ⟨ZKetteOhneDP⟩ ⟨Wörter⟩

⟨CGDek⟩ ::= ⟨Konzept⟩ | ⟨Konzept⟩ ⟨RelSubgr⟩ | ⟨Konzept⟩ - ⟨RelListe⟩ .

⟨Konzept⟩ ::= [⟨KName⟩ ⟨Instanzen⟩]

⟨KName⟩ ::= ⟨Zeichenkette⟩

⟨Instanzen⟩ ::= ε | : ⟨Zahl⟩ | : { ⟨Liste⟩ }

⟨Liste⟩ ::= ⟨Zeichenkette⟩ | ⟨Zeichenkette⟩ , ⟨Liste⟩

⟨RelListe⟩ ::= ⟨RelSubgr⟩ | ⟨RelSubgr⟩ , ⟨RelListe⟩

⟨RelSubgr⟩ ::= -> (⟨RName⟩) -> ⟨CGDek⟩ | -> (⟨RName⟩) - ⟨CGListe⟩ .

⟨RName⟩ ::= ⟨Zeichenkette⟩

⟨CGListe⟩ ::= -> ⟨CGDek⟩ | -> ⟨CGDek⟩ , ⟨CGListe⟩

A.2 Zusätzliche Bemerkungen

Folgende Nebenbedingungen müssen über die syntaktischen Bedingungen hinaus gelten:

Typkomponenten Jede Komponente darf nur einmal auftauchen. Wenn *TranTyp* vorkommt, so muss auch *SynTyp* vertreten sein.

Zeichenkette Diese ist in ihrer Länge durch das Java-Laufzeitsystem beschränkt, kann insbesondere über 255 Zeichen lang sein. Sie darf keine Zeichen beinhalten, welche als Syntaxelemente *(Token)* verwendet werden, wie etwa Doppelpunkte, Semikola, Kommas oder geschweifte Klammern. Leerzeichen dürfen nur dann auftauchen, wenn die Zeichenkette mit einem doppelten Anführungszeichen beginnt und endet.

Zahl Diese stellt einen ganzzahligen Wert dar, welche im Bereich des Datentyps *int* liegen muss.

NMarke Die Marken können mit einer im Annotationsblock auftauchenden Annotation korrespondieren. Diese sollte semantische Informationen zu der zugehörigen Nachricht liefern.

EMarke Die Marken können mit einer im Annotationsblock auftauchenden Annotation korrespondieren; anderenfalls werden sie als nicht vorhanden angesehen. Die Annotation sollte semantische Informationen zu dem zugehörigen Nachrichtenelement liefern.

Datentyp Der Bezeichner des Datentyps muss einer im Java-Laufzeitsystem bekannten Klasse entsprechen und vollqualifiziert sein. Es muss sich um eine Systemklasse handeln, darf also keinen speziellen Klassenladers erfordern. Die gepaarten eckigen Klammern repräsentieren die Dimension des Feldes dieses Datentyps; je ein Paar ist für jede Dimension einzusetzen.

Zustand Die Bezeichner im Teil hinter dem Gleichheitszeichen können einen Zustand referenzieren, welcher in irgendeiner Zeile vor dem Gleichheitszeichen deklariert wird; anderenfalls ist der bezeichnete Zustand ein Endzustand. Der erste deklarierte Zustand ist immer der Startzustand.

NachRef Die Zeichenkette muss eine im Nachrichtenblock bezeichnete Nachricht referenzieren.

SEMarke	Der Bezeichner muss eine Annotation im Annotationsblock referenzieren. Diese sollte einen Sender (oder Empfänger) der zugehörigen Nachricht beschreiben.
KName	Der Bezeichner kann beliebig sein, sollte jedoch in der Ontologie der Konzepte auftauchen, welche zur Laufzeit eingesetzt wird. Gleiches gilt für *RName* in Bezug auf die Relationsontologie.
ZKetteOhneDP	Diese Zeichenkette enthält keinen Doppelpunkt.
AMarke	Der Bezeichner *self* besitzt eine Sonderbedeutung und darf höchstens einmal auftreten. Er markiert die Selbstannotation.

B Klassen des Typsystems

B.1 Strukturelle Klassen

Die erste Gruppen von Klassen sind die *strukturellen Klassen*. Ihre Instanzen definieren die Struktur der Typen und sind dadurch festgelegt. Der Zeitpunkt der Festlegung ist die Erstellung des Typs, sei es durch ein Werkzeug (*SecAdmin*) oder durch den Aufruf einer Methode des Mediators (*typeForString*). Die Liste der strukturellen Klassen umfasst

HybridType	Container für die Teile des hybriden Typs
SyntacticType	Container für die Ein- und Ausgabenachrichten
MessageType	Repräsentiert eine Nachricht
MessageItemType	Kleinste Komponente einer Nachricht
TransitionType	Container für den endlichen Automaten, welcher das Protokoll definiert
State	Zustand; Teil des endlichen Automaten
Transition	Zustandsübergang; Teil des endlichen Automaten
SemanticType	Container für die Selbst-Annotation
Annotation	Container für die Annotation; entweder String oder Konzeptgraph
ConceptualGraph	Repräsentation eines Konzeptgraphen
CGConceptNode	Repräsentation eines Konzeptknotens
CGRelationNode	Repräsentation eines Relationsknotens

Objekte dieser Klassen werden vom Hauptobjekt *HybridType* aggregiert und liegen in serialisierter Form im SPU-Container vor.

B.2 Intermediäre Klassen

Eine Reihe von Klassen ist lediglich von Interesse während des Vorgangs des Typvergleichs. Sie beinhalten jedoch einen Großteil der Codes, der für die Durchführung des Vergleichs notwendig ist.

StateSet	Menge von Zuständen
StateSetTable	Hashtabelle zum Speichern von Zwischenergebnissen
InteractionType	Repräsentiert eine Interaktion
Ontology	Definiert die Bedeutung von Konzeptnamen und Relationsnamen
CGOntologySet	Implementiert die Schnittstelle *KnowledgeBase* und beinhaltet die Konzept- und Relationsontologie
TypeConformance	Wird einerseits zur Parametrisierung des Vergleichs benötigt, dient andererseits als Rückgabewert, der den Anfrager über die Güte des Treffers in Kenntnis setzt

Objekte intermediärer Klassen werden bei einem Typvergleich instantiiert und nach dessen Vollendung entsorgt. Die Ontologie liegt vor dem Typvergleich bereits vor; sie wird durch eine spezielle Textdatei definiert, die bei Bedarf eingeladen wird.

B.3 Zusätzliche Elementtypen

Die von Java angebotenen Einbettungsklassen (*Wrapper*-Klassen) *Number, Long, Integer, Short, Byte, Double* und *Float* (alle im Paket *java.lang*) sind geeignet, Basistypen aufzunehmen und sie beispielsweise in einem Vektor als Element einzufügen. Nachrichten im AMETAS-System beinhalten ein Feld von Objekten als Nutzlast, sodass es erforderlich ist, Basistypen mittels solcher Klassen zu repräsentieren.

Allerdings implementieren diese Klassen nicht die Vererbungshierarchie der eingebetteten Datentypen. Alle Klassen außer *Number* sind von *Number* direkt abgeleitet. Dadurch gibt es keine Möglichkeit, eine *Integer*-Instanz in der Rolle einer *Long*-Instanz zu verwenden. AMETAS definiert daher Ersatzklassen, welche die folgende Hierarchie realisieren:

- $AByte \prec AShort \prec AInteger \prec ALong \prec ANumber$
- $AFloat \prec ADouble \prec ANumber$

Alle genannten Klassen gehören dem Paket *AMETAS.data* an. *ANumber* selbst ist direkt von *java.lang.Object* abgeleitet. Die jeweiligen Klassen verfügen über Zugriffsmethoden wie *longValue* oder *intValue*, wobei diese den Subklassen vererbt werden: Wird ein *int*-Wert über eine Ausgangsnachricht erwartet, wobei die Ausgangsnachricht gemäß der Kovarianzregel einen Subtyp schicken darf, dann wird auf der geschickten *AShort*-Instanz die Methode *intValue* aufgerufen, welche von der erwarteten Klasse *AInteger* geerbt wurde.

Auf Beziehungen zwischen Fließkomma- und Ganzzahltypen musste verzichtet werden, da Java keine Mehrfachvererbung zulässt, andererseits jedoch *ALong* nicht als Spezialisierung von *ADouble* gesehen werden kann: *ALong* kann einige Zahlen präziser als *ADouble* repräsentieren, welche Bitstellen für den Exponenten bereithalten muss.

C Beispiele für Ontologien

Ontologien dienen dazu, ein Vokabular zu definieren, welches bei der Formulierung von Konzepten und Relationen erlaubt ist. Es gibt unzählige Möglichkeiten, Konzepte und Relationen festzulegen. Im Folgenden seien Beispiele für Konzept- und Relationsontologien aufgeführt, welche zu Testzwecken Verwendung fanden.

C.1 Konzeptontologie

Die Konzeptontologie liegt in Form einer Textdatei vor. Diese wird zur Laufzeit vom *HybridTypeMediator* eingelesen und als Baumstruktur gespeichert. Der erste Begriff nach einer offenen runden Klammer bezeichnet dabei stets den Inhalt eines Knotens, dessen Kinderknoten durch die folgenden Begriffe bezeichnet werden. Die innerhalb der eckigen Klammern voneinander getrennten Begriffe sind Synonyme, also gleichbedeutend. Beispielsweise sind *Drucker* und *Scanner* spezielle *Peripheriegeräte*, welche ihrerseits wie *PCs* zu den *IT-Geräten* gehören.

```
#
# Konzeptontologie
#
#
( [ something | Etwas | Entität ]
  ( [ Gegenstand ]
    ( [ "elektrisches Gerät" | "elektronisches Gerät" ]
      ( [ "IT-Gerät" ]
        [ Computer | Rechner | PC | Workstation ]
        ( [ Komponente ]
          [ Netzwerkkarte | Netzschnittstelle ]
          [ CPU | Prozessor ]
          [ Hauptspeicher | Speicher ]
          [ Festplatte ]
        )
        [ Netzwerk | Kommunikationsweg ]
        ( [ Peripheriegerät ]
```

```
            [ Drucker ]
            [ Scanner ]
          )
        )
      )
      ( [ "nichtelektrischer Gegenstand"
          | "nichtelektronischer Gegenstand" ]
        [ Währung | Geld ]
      )
    )
    ( [ Mensch ]
      [ Anwender | Benutzer ]
      [ Administrator ]
    )
    ( [ Struktur ]
      ( [ Menge | Set ]
        [ Reiseroute | Itinerary ]
        [ Liste | List | Feld | Array ]
        [ Gruppe | Group ]
        ( [ Domäne | Domain ]
          [ "PNS-Domäne" | "PNS Domain" ]
        )
      )
    )
    ( [ Idee ]
      ( [ Plan ]
        [ Entwurf | Design ]
        [ Auftrag | Aufgabe ]
      )
      ( [ Paradigma ]
        [ Autonomie ]
        [ Mobilität ]
        [ Sicherheit ]
        [ Intelligenz ]
      )
      [ Theorie ]
    )
    ( [ Information | Daten | Datum ]
      ( [ "ausführbare Daten" ]
        [ Anwendung | Programm | Code ]
        ( [ Agentensystem ]
          [ AMETAS ]
```

```
 )
 ( [ Stellennutzer | PlaceUser ]
   [ Benutzeradapter | UserAdapter ]
   [ Dienst | Service ]
   ( [ Agent | "AMETAS-Agent" ]
     [ "mobiler Agent" ]
     [ "intelligenter Agent" ]
   )
 )
 ( [ "Agentenanwendung" ]
   [ "AMETAS-Anwendung" ]
 )
 [ "AMETAS-Stelle" | Agentenstelle ]
)
( [ Beschreibung ]
  [ Stellennutzer-ID | PUID ]
  [ Klasse | Class ]
  ( [ Typ | Typbeschreibung ]
    [ "hybrider Typ" | "Hybridtyp" ]
  )
  [ "syntaktischer Typ" ]
  [ "Transitionstyp" ]
  [ "semantischer Typ" ]
  ( [ Wissensrepräsentation ]
    [ Konzeptgraph ]
    [ Ontologie ]
  )
  [ Annotation ]
  ( [ Sprache ]
    [ "menschliche Sprache" ]
    [ Programmiersprache ]
  )
)
[ Instanz ]
( [ Nachricht | Message ]
  [ Anfrage | Request ]
  [ Antwort | Response | Reply ]
)
[ Ereignis ]
( [ Wert | Messwert | Value ]
  [ Belastung ]
  [ Menge ]
```

```
            [ Zunahme ]
            [ Abnahme ]
            [ Kontostand | Guthaben ]
          )
          [ Zustand ]
          [ Nachrichtenelement ]
          [ Bezeichnung | Name ]
          [ Befehl | Anweisung ]
          [ Parameter | Parametrisierung ]
      )
      ( [ Vorgang ]
        [ Transition | Übergang | Zustandsübergang ]
        [ Entstehen | Start ]
        [ Vergehen | Ende ]
        ( [ Änderung | Modifikation ]
          [ Verbesserung ]
          [ Beschleunigung ]
          [ Verschlechterung ]
          [ Verzögerung ]
          [ Abbremsung ]
        )
      )
      ( [ Tätigkeit | Aktion ]
        [ Management | Wartung | Verwaltung ]
        [ Messung ]
      )
    )
```

Wie dieses Beispiel demonstriert, ist die Festlegung einer in jeder Situation geeigneten Ontologie sehr schwierig; manche Konzepte können nicht eindeutig unter andere, übergeordnete Konzepte angesiedelt werden. Diese Ontologie ist nur als ein Entwurf zu verstehen, welcher im Laufe des Einsatzes des Typsystems kontinuierlich verfeinert werden kann.

C.2 Relationsontologie

Die Hierarchisierung von Relationen ist noch schwieriger als das Anordnen von Konzepten in einer Hierarchie. Offenbar fällt es leichter, Konzepte zu abstrahieren; Relationen werden bevorzugt einzeln betrachtet und nicht miteinander auf Ähnlichkeiten verglichen. Dennoch soll hier der Versuch unternommen werden, Relationen durch schrittweise Präzisierung zu verfeinern und Synonyme zu beachten.

```
# Relationshierarchie
#
#
(  [ "steht in Relation zu" ]
   /* Existenz */
   (  [ "existiert in" ]
      [ "existiert zeitlich bei" ]
      [ "existiert räumlich in" | "bleibt bei"
        | "verweilt bei" ]
      [ "ist ein" | "stammt ab von" | "ist Subtyp von" ]
      [ "ist Instanz von" ]
      [ "gehört zu" | "ist Teil von" | "nimmt teil an" ]
   )
   /* Passivität */
   (  [ "ist beeinflusst durch" ]
      [ "existiert durch" ]
      [ "benötigt" | "braucht" | "erfordert" ]
      [ "vermeidet" ]
      [ "sucht" | "lokalisiert" | "findet" ]
      [ "überwacht" | "beobachtet" ]
      [ "wartet auf" | "erwartet" ]
      [ "beschreibt" | "erklärt" | "deklariert" ]
   )
   /* Aktivität */
   (  [ "beeinflusst" ]
      (  [ "modifiziert" | "operiert auf" | "ändert"
           | "manipuliert" | "konfiguriert" ]
         [ "verwaltet" | "managet" ]
         [ "installiert" ]
         [ "deinstalliert" | "entfernt" ]
         [ "aktiviert" | "schaltet an" ]
         [ "deaktiviert" | "schaltet ab" | "terminiert" ]
      )
      (  [ "kommuniziert mit" ]
         [ "schickt Nachrichten an" | "benachrichtigt" ]
         [ "notifiziert" | "meldet an" ]
      )
      [ "erzeugt" | "generiert" ]
      [ "liefert" ]
   )
   /* Entwicklung */
   (  [ "ändert sich in" | "wechselt Zustand" ]
```

```
        [ "wandert zu" | "migriert zu" | "geht zu" ]
        [ "speichert" | "notiert" | "registriert"
          | "sammelt" | "trägt zusammen" ]
    )
    /* Attribute */
    (  [ "beinhaltet" ]
       [ "besitzt" | "verfügt über" | "hat" ]
       [ "besteht aus" | "ist zusammengesetzt aus" ]
    )
)
```

In der Regel werden die allgemein formulierten Relationen selten Anwendung finden, da sich Anwender wie Entwickler üblicherweise präziser ausdrücken, um Irrtümer zu vermeiden. Hier sind insbesondere die Synonyme von Bedeutung, da durchaus statt „hat“ das Wort „besitzt“ auftauchen kann.

SOWA stellt in [Sow84] eine Liste von Konzepten und Relationen auf, welche dort zur Bildung der Konzeptgraphen herangezogen werden. Zwar ist die Zahl der Relationen vergleichsweise gering, aber es kann eine Vielzahl an Sachverhalten beschrieben werden. Der wesentliche Nachteil von Sowas Liste ist jedoch, dass die entstehenden Konzeptgraphen schwieriger zu lesen und zu erzeugen sind, da – anders als hier – die Relationen nicht durch Verben in Sätzen dargestellt werden, sodass eine weitere Analyse der Bedeutung des Satzes notwendig wird. Die Wahl von Konzepten und Relationen, wie sie in dieser Arbeit vorgenommen wird, erlaubt eine leichtere Umsetzung menschlicher Sprache in Konzeptgraphen.

D Konformitätsobjekt

Das Konformitätsobjekt wird in der AMETAS-Version 2.5 als Instanz der Klasse *AMETASx.data.htype.HybridTypeConformance* repräsentiert. Die verschiedenen Bedeutungen des Inhalts werden in diesem Abschnitt anhand eines Ausschnitts des Quellcodes der Klasse beschrieben.

```
package AMETASx.data.htype;
import AMETAS.data.type.TypeConformance;

public class HybridTypeConformance extends TypeConformance {

   public HybridTypeConformance(int nFlags) {
     super(nFlags);
   }

   public final static int MSG_AMBIVALENCE = 1<<0;
   public final static int MSG_ANN_MISMATCH = 1<<1;
   public final static int MSG_CONST_MISMATCH = 1<<2;
   public final static int MSG_MISSING_IN = 1<<3;
   public final static int MSG_EXTRA_OUT = 1<<4;
   public final static int MSG_TYPE_MISMATCH = 1<<5;
   public final static int MSG_VARIANCE = 1<<6;
   public final static int MSG_ITEM_ANN_MISMATCH = 1<<7;
   public final static int MSG_SHRUNK_INPUT = 1<<8;
   public final static int MSG_INFLATED_INPUT = 1<<9;
   public final static int MSG_SHRUNK_OUTPUT = 1<<10;
   public final static int MSG_INFLATED_OUTPUT = 1<<11;

   public final static int TR_MISSING = 1<<15;
   public final static int TR_STATE_ANN_MISMATCH = 1<<16;
   public final static int TR_NONDETERMINISM = 1<<17;
   public final static int TR_DETERMINISM_BY_ANN = 1<<18;
   public final static int TR_OTHER_EPSILON = 1<<19;
   public final static int TR_SENDER_MISMATCH = 1<<20;
```

```
        public final static int TR_RECV_MISMATCH = 1<<21;
        public final static int TR_OUTPUT_MISMATCH = 1<<22;

        public final static int ANN_FORMAT_MISMATCH = 1<<25;
        public final static int ANN_MISMATCH = 1<<26;
        public final static int ANN_PARTIAL_MATCH = 1<<27;
        public final static int ANN_VALUE_MISMATCH = 1<<28;
        public final static int ANN_UNKNOWN = 1<<29;
        public final static int ANN_SELF_MISMATCH = 1<<30;
    ...
    }
```

Das Konformitätsobjekt besteht im Prinzip aus einem 32-Bit-Feld, wobei jedem Bit eine bestimmte Bedeutung zugewiesen ist; die Konformität besteht somit aus einer Oder-Verkettung von Beurteilungen. Eine vollständige Konformität wird in der Basisklasse *TypeConformance* durch

```
public final static int MATCHING = 0
```

definiert. Dieser Wert wird jeder Konformitätsklasse vererbt.

D.1 Nachrichtenblock

Nachfolgend ist die Bedeutung der einzelnen Beurteilungen aufgeführt. Man beachte, dass das Ergebnis nicht unbedingt alle vorliegenden Inkompatibilitäten beinhaltet, da die Auswertung schon beim ersten fatalen Fehler abbrechen kann.

`MSG_AMBIVALENCE` Zu einer Nachricht des Typs existieren mehrere treffende Nachrichten des Subtyps. So könnte eine Eingabenachricht *AInteger* im Typ sowohl durch *Object* als auch durch *ALong* im Subtyp realisiert sein. Beide sind Verallgemeinerungen, also gemäß Kontravarianzregel kompatibel.

`MSG_ANN_MISMATCH` Die Annotation der Nachricht stimmt nicht überein.

`MSG_CONST_MISMATCH` Eine Konstante der Nachricht stimmt nicht überein. Dies wird als schwer wiegender Verstoß interpretiert, da Konstanten häufig zur Differenzierung von Nachrichten mittels Diskriminatoren verwendet werden.

`MSG_MISSING_IN` Der Subtyp bietet eine Eingangsnachricht des Typs nicht an.

`MSG_EXTRA_OUT` Der Subtyp liefert eine Ausgangsnachricht, die der Typ nicht liefert.

`MSG_TYPE_MISMATCH` Einer der Datentypen der Nachrichtenelemente stimmt nicht überein (unter Beachtung von Ko- und Kontravarianz).

`MSG_VARIANCE` Es liegt Ko- oder Kontravarianz vor.

`MSG_ITEM_ANN_MISMATCH` Eines der Nachrichtenelemente weist eine nicht kompatible Annotation auf. Hier handelt es sich jedoch nicht unbedingt um eine Unstimmigkeit bei Konstanten.

`MSG_SHRUNK_INPUT` Die Eingangsnachricht des Subtyps, die zu einer Nachricht des Typs passt, beinhaltet einen Präfix der Elementfolge dieser Nachricht. Dies ist in der Regel tolerabel, da die übrigen Elemente einfach ignoriert werden.

`MSG_INFLATED_INPUT` Die Eingangsnachricht des Subtyps, die zu einer Nachricht des Typs passt, fügt dieser Nachricht weitere Elemente hinzu. Dies ist in der Regel nicht tolerabel, da der Subtyp in diesem Falle mehr Informationen benötigt als der Typ.

`MSG_SHRUNK_OUTPUT` Die Ausgangsnachricht des Subtyps, die zu einer Nachricht des Typs passt, beinhaltet einen Präfix der Elementfolge dieser Nachricht. Dies ist in der Regel nicht akzeptabel, da in diesem Falle der Subtyp zu wenige Informationen liefert.

`MSG_INFLATED_OUTPUT` Die Ausgangsnachricht des Subtyps, die zu einer Nachricht des Typs passt, fügt dieser Nachricht weitere Elemente hinzu. Dies ist in der Regel akzeptabel, da die zusätzlichen Daten ignoriert werden können.

Hier gilt zu beachten, dass diesen Entscheidungen das Datenrepräsentationsmodell von Java zu Grunde liegt. Eine Feldvariable kann demzufolge beliebig lange Felder referenzieren; somit dürfen Felder länger werden als erwartet. Sind Felder hingegen zu kurz, so entsteht eine Ausnahmebedingung. Wenn diese Annahme nicht getroffen werden kann,[1] sind auch Verlängerungen von Feldern unzulässig.

D.2 Transitionsblock

Die Urteile, welche den Transitionsblock betreffen, sind beim vollständigen Typvergleich von Interesse. Das Fehlen von Transitionen führt in der Regel direkt zum Abbruch des Vergleichs.

`TR_MISSING` Der Subtyp kann eine Transition des Typs nicht durchführen. Dies ist in der Regel ein schwerer Verstoß, es sei denn, die Transition ist Bestandteil einer nichtdeterministischen Alternative. In diesem Falle würde dieses Bit nicht gesetzt.

[1] Insbesondere bei Sprache wie C und C++, welche eine explizite Speicherallokation erwarten.

`TR_STATE_ANN_MISMATCH` Die Annotation eines Zustands stimmt nicht mit der Annotation des entsprechenden Zustands des Typs überein.

`TR_NONDETERMINISM` Es liegt ein Nichtdeterminismus vor; der Subtyp kann auf verschiedene Weise bei der gleichen Eingabe reagieren. Dies ist in der Regel tolerabel; wenn diese Tatsache bekannt ist, kann der Kommunikationspartner verschiedene Reaktionen erwarten.

`TR_DETERMINISM_BY_ANN` Es liegt Nichtdeterminismus vor, welcher lediglich durch Annotationen in einen Determinismus überführt wird.

`TR_OTHER_EPSILON` Der Subtyp gibt an, eine Nachricht an jemand anders zu senden oder von jemand anderem zu erwarten. Der Typ hingegen erwartet oder sendet nichts. Dies ist in der Regel akzeptabel.

`TR_SENDER_MISMATCH` Die Nachricht des Subtyps und die vergleichbare des Typs weisen unterschiedliche Absender auf. Dies ist in der Regel nicht akzeptabel.

`TR_RECV_MISMATCH` Die Nachricht des Subtyps und die vergleichbare des Typs weisen unterschiedliche Empfänger auf. Dies ist in der Regel nicht akzeptabel.

`TR_OUTPUT_MISMATCH` Die Transition des Subtyps liefert eine inkompatible Ausgabe. Dies ist in der Regel nicht akzeptabel.

D.3 Annotationsblock

Annotationen unterliegen dem Problem, dass gleiche Sachverhalte von unterschiedlichen Personen verschieden aufgefasst werden können. Obwohl beide Personen – etwa der Autor eines Agenten und der Nutzer eines Agenten – beabsichtigen, mit ihrer Typbeschreibung konform zu jener des anderen zu sein, kann dies durch zu unterschiedliche Sichtweisen vereitelt werden. Daher werden Annotations-Inkompatibilitäten gewöhnlich als nicht fatal eingestuft. Ausnahme bilden die Konstantenannotationen, welche verwendet werden, um zur Laufzeit zwischen Nachrichten zu unterscheiden.

`ANN_FORMAT_MISMATCH` Das Format der Annotationen stimmt nicht überein (String gegenüber Konzeptgraph).

`ANN_MISMATCH` Die Annotationen stimmen (sinngemäß) nicht überein. Da es sich hier möglicherweise um eine nicht beabsichtigte Fehlformulierung handelt, wird dies per Standard toleriert, kann aber bei Bedarf als fataler Fehler gelten.

`ANN_PARTIAL_MATCH` Die Konzeptgraphen überdecken sich nur partiell; beide weisen also Verfeinerungen auf, welche der jeweils andere nicht besitzt. Dies sollte in der Regel nicht akzeptiert werden.

`ANN_VALUE_MISMATCH` Die Annotationen stimmen sinngemäß, aber ihre Werte stimmen nicht. Dies wird normalerweise nicht akzeptiert.

`ANN_UNKNOWN` Die Annotation verwendet Konzept- oder Relationsbezeichungen, welche in der Ontologie nicht auftauchen. Dies zählt als nicht fatale Fehlformulierung.

`ANN_SELF_MISMATCH` Die Selbstannotation (also der semantische Typ im engeren Sinne) stimmt nicht überein. Dies wird in der Regel nicht akzeptiert, da es sich um den wesentlichen Kern der Vermittlung aus Benutzersicht handelt.

E Programmierschnittstelle

Um das Typsystem in AMETAS zu verwenden, gibt es eine Programmierschnittstelle, welche von Stellennutzer-Implementierern genutzt werden kann. Es gibt keine Konzentration aller Methoden in einer Klasse, vielmehr wird der Zugang über verschiedene Klassen der Pakete *AMETAS.data.type* (für allgemeine Typsystem-Funktionen) sowie *AMETASx.data.htype* (für Hybridtyp-spezifische Funktionen) ermöglicht. Um Aktionen in der Stelle zu bewirken, sind Aufrufe über die Treiberschnittstelle *AMETAS.place.AMETASPlaceUserDriverIf* vorgesehen.

Die hier genannten Klassen verfügen über weit mehr Methoden als angegeben; viele davon sind für den Anwendungsprogrammierer nur von geringem Interesse, da sie vom AMETAS-Kernsystem zur Unterstützung des Hybridtypsystems benötigt werden.

E.1 Allgemeine Typfunktionen

Die hier aufgeführten Klassen und Methoden sind von dem tatsächlich verwendeten Typsystem unabhängig.

E.1.1 TypeConformance

Das Typkonformitätsobjekt definiert in dieser Standardform lediglich zwei Zustände:

- vollständige Übereinstimmung und
- keine Übereinstimmung.

Es eignet sich daher bereits für String-basierte Typen; komplexere Typsysteme wie das Hybridtypsystem leiten diese Klasse ab (siehe Anhang D). Die Basisklasse legt für jede Instanz einer abgeleiteten Klasse die Kardinalität der Zustandsmenge auf 2^{31} fest (durch Identifikation jedes Zustands mit einem *int*-Wert). Abgeleitete Klassen definieren im Wesentlichen nur Konstanten für den Zustände; die Basisklasse bietet Methoden zur Abfrage der Zustände.

```
public boolean isFullConformance()
```

liefert *true*, wenn eine volle Konformität besteht, ansonsten *false*.

```
public boolean checkFor(int nFlags)
```

liefert *true,* wenn die in *nFlags* gesetzten Bits ebenfalls in diesem Objekt gesetzt sind. Für weitere Methoden sei auf die AMETAS-Programmierschnittstelle [ZH01] verwiesen.

E.1.2 AMETASType

Ein Objekt der Klasse *AMETASType* repräsentiert eine Typinstanz. Dabei dient diese Klasse nicht als Superklasse für spezielle Typimplementierungen, sondern als Container. Typimplementierungen werden über *TypeContent* vorgenommen.

```
public AMETASType(String sName)
public AMETASType(TypeContent tc)
```

Wird eine Zeichenkette zur Generierung des Typs übergeben, so wird eine *StringType*-Instanz generiert, welche aus der Zeichenkette generiert wird.

```
public void setKind(byte byKind)
public byte getKind()
```

Neben der Abfrage der vollständigen Typbeschreibung des Stellennutzers gibt es die Möglichkeit, die Art des Stellennutzers (also Agent, Benutzeradapter, Dienst) mit Hilfe dieser Methoden zu setzen oder zu erfragen. Dies ist vor allem dann effizient, wenn keine echte Vermittlung erwünscht wird, sondern die Elemente einer Menge von Stellennutzern auf ihre Art geprüft werden sollen.

Ein Typvergleich wird stets mit

```
public TypeConformance compareTo(AMETASType type,
                                 TypeConformance tc,
                                 KnowledgeBase kb)
```

eingeleitet. Diese Methode wird in der Regel vom Mediator aufgerufen, kann aber auch vom Stellennutzer direkt verwendet werden, um zwei Typen zu vergleichen. Werden die letzten beiden Argumente weggelassen, werden Standardvorgaben angenommen.

E.1.3 KnowledgeBase und MediatorInfo

KnowledgeBase ist lediglich eine Schnittstelle, welche die Methode *getName* bereithält.

```
public String getName()
```

Die Struktur von Wissensbasen ist nicht verbindlich festgelegt; daher können keine spezifischen Methoden zur Manipulation und Abfrage in einer Schnittstelle vorgesehen werden. Der Anfrager soll aber in der Lage sein, die Wissensbasen anhand ihres Namens zu unterscheiden.

Die Klasse *MediatorInfo* beinhaltet Informationen zum Mediator. Der Mediator definiert, welche Wissensbasen er unterstützt. Die dem aktuell aktiven Mediator zugeordnete Instanz kann der Klient über seinen Treiber erhalten (siehe Abschnitt E.3). Über

```
public KnowledgeBase[] getKnowledgeBases()
```

lassen sich aus diesem *MediatorInfo*-Objekt die verfügbaren Wissensbasen erfragen. Möchte der Klient seine Typanfrage anhand einer bestimmten Wissensbasis ausgewertet bekommen, kann er eine Basis auswählen und setzt deren Namen in die Typanfrage ein.

E.1.4 AMETASMediationRequest

Ein Objekt der Klasse *AMETASMediationRequest* kapselt eine Typanfrage. Diese wird der Methode *request* der Treiberschnittstelle übergeben.

```
public AMETASMediationRequest(AMETASType type,
                              TypeConformance tcFail,
                              boolean bRunning,
                              String sKBName)
```

Der Konstruktor einer Typanfrage übernimmt zum einen die Typinstanz (welche gegebenenfalls aus einer Textrepräsentation erzeugt werden muss), das Konformitätsobjekt, welches den Vergleichsalgorithmus steuert (siehe auch Anhang D), eine Angabe, ob eine *existierende Instanz* oder ein *Typ* gesucht wird und die Angabe der Wissensbasis, sofern der Mediator verschiedene Wissensbasen anbietet. Die Argumente können schrittweise von hinten weggelassen werden und werden durch Standardvorgaben ersetzt.

E.1.5 AMETASMediationResult

Ein Objekt der Klasse *AMETASMediationResult* kapselt ein Ergebnis eines Vermittlungsvorgangs. Dabei beinhaltet dieses Objekt genau ein Ergebnis; mehrere Ergebnisse werden in Form mehrerer einzelner Instanzen geliefert.

```
public TypeConformance getConformance()
```

Diese Methode liefert den Konformitätsgrad dieses Ergebnisses. Der Stellennutzer kann hiermit entscheiden, ob das Ergebnis seinen Erwartungen entsprach. Dies ist besonders dann nützlich, wenn die Typanfrage bewusst allgemein gehalten wurde, um das Risiko nicht gefundener Übereinstimmungen zu minimieren.

```
public AMETASType getResultType()
```

Mit dieser Methode kann der Stellennutzer den Typ erhalten, welcher von diesem Objekt gekapselt wird.

```
public AMETASPlaceUserID getPlaceUserID()
public String getRegistrationName()
```

Handelte es sich um eine *Instanzvermittlung*, so bezeichnet das Ergebnis einen laufenden Stellennutzer; dieser verfügt über eine Stellennutzer-ID. Diese ID wird vor allem benötigt, um dem Stellennutzer eine Nachricht zu schicken. Durch die Gewinnung der Stellennutzer-ID ist der Vermittlungsvorgang abgeschlossen. Im Falle einer *Typvermittlung* ist es erforderlich, den Registrierungsnamen zu erfahren, welcher für das Laden des SPU-Containers benötigt wird. Dies wird durch die zweite Methode geleistet.

E.1.6 AMETASServiceDescription

Die Klasse *AMETASServiceDescription* ist historisch gesehen die älteste Klasse des Typsystems; sie diente der Beschreibung von Diensten. In späteren AMETAS-Versionen wurde diese Beschreibung als Subklasse von *AMETASType* neu implementiert.

```
public AMETASServiceDescription(Object repres,
                                String sVerbalDescr,
                                short nMode,
                                long nTimeout,
                                long nCharge)
```

Die Repräsentation einer Dienstbeschreibung ist auf zwei Weisen möglich: Ist *repres* eine *String*-Instanz, so wird ein *String-basierter Typ* gebildet. Ansonsten muss *repres* eine Instanz der Klasse *TypeContent* sein. Die Dienstbeschreibung kann des Weiteren mit einer verbalen Beschreibung ausgestattet werden, welche vor allem administrativen Zwecken dient, aber für die Vermittlung irrelevant ist. Der Modus beschreibt einen der möglichen Dienstaktivierungsmodi (also *SHARED, SHARED_SESSION* sowie *NON_SHARED, NON_SHARED_SESSION*). Der Dienst beendet sich (in Abhängigkeit von seinem Modus) frühestens nach *nTimeout* Millisekunden Inaktivität. Ist ein Abrechnungssystem installiert, können die Kosten der Dienstbenutzung als *nCharge* angegeben werden.

E.2 Spezielle Hybridtypfunktionen

Das Hybridtypsystem bietet zahlreiche Methoden in den hier genannten Klassen, welche im Einzelnen in der AMETAS-Dokumentation nachgelesen werden können. An dieser Stelle sollen nur die wichtigsten Klassen, Konstruktoren und Methoden aufgeführt werden.

E.2.1 HybridType

Die Klasse *HybridType* repräsentiert einen vollständigen Hybridtyp und ist von *AMETAS.data.type.TypeContent* abgeleitet.

```
public HybridType(SyntacticType st, TransitionType tr,
                  SemanticType semt)
```

Der Konstruktor erlaubt die Erzeugung eines Hybridtyps ohne Verwendung einer textuellen Repräsentation. Dies ist dann sinnvoll, wenn für eine Typanfrage eine einfache Selbstannotation konstruiert wird; aus dieser kann ein semantischer Typ und anschließend der Hybridtyp generiert werden. Zwei Hybridtypen werden mittels der Methode

```
public TypeConformance getConformanceTo(TypeContent tcSuper,
                                        TypeConformance tc,
                                        KnowledgeBase kb)
```

miteinander verglichen. Der Vergleichsvorgang obliegt üblicherweise dem Mediator, kann aber hiermit von der Anwendung selbst angestoßen werden. *tcSuper* muss eine Instanz der Klasse *HybridType* sein, *tc* muss ein *HybridTypeConformance*-Objekt sein und *kb* muss von *CGOntologySet* stammen. Das gelieferte Objekt ist eine Instanz von *HybridTypeConformance.*

E.2.2 SemanticType, Annotationen und Konzeptgraphen

Das *SemanticType*-Objekt kapselt lediglich die so genannte Selbstannotation. Die übrigen Annotationen werden direkt von den entsprechenden Objekten (Nachrichten, Zustände und so weiter) referenziert.

```
public SemanticType(Annotation ann)
public Annotation getSelfAnnotation()
```

Annotationen können auf Zeichenketten- oder Konzeptgraphenbasis definiert sein. Die Klasse *Annotation* bietet den Konstruktor

```
public Annotation(Object content)
```

und erlaubt beide Repräsentationen als Inhalt. Zwei Annotationen lassen sich über

```
public HybridTypeConformance
       getConformanceTo(Annotation annOther,
                        HybridTypeConformance tc,
                        CGOntologySet cg)
```

miteinander vergleichen. Um zu den Ontologien zu gelangen, kann eine *getMediatorInfo*-Abfrage auf dem Treiber dienen.

Konzeptgraphen werden gemäß den Regeln zur Erstellung von Konzeptgraphen als Baum konstruiert, wobei sich Konzept- und Relationsknoten beständig abwechseln. Die notwendigen Klassen lauten *ConceptualGraph,* welche den Konzeptgraphen repräsentiert und das Wurzelkonzept markiert sowie *CGConceptNode* und *CGRelationNode.*

```
public ConceptualGraph(CGConceptNode root)
public CGConceptNode(String sConcept, Object value)
public CGRelationNode(String sRelation)
```

Letztere beiden Klassen sind von der Klasse *CGNode* abgeleitet, welche unter anderem die Methode

```
public void addSuccessors(CGNode[] succ)
```

beisteuert, um die Möglichkeit des Aufbaus eines Baums zu bieten. Man beachte, dass Konzeptknoten als Wert sowohl eine Liste von Zeichenketten *(String[])* als auch einen numerischen Wert *(Integer)* erlauben.

E.2.3 SyntacticType, MessageType und MessageItemType

Es ist möglich, Nachrichtentypen aus einzelnen Nachrichtenelementtypen zusammenzusetzen; für Details sei auf [ZH01] verwiesen. Eine interessante Möglichkeit, einen Nachrichtentyp auf einfache Weise zu generieren, liefert die Klasse *MessageType*:

```
public MessageType(String sSignature)
```

Dieser Konstruktor nimmt eine Zeichenkette der Form *Typ1, Typ2, . . . , Typ*n entgegen und konstruiert daraus einen vollständigen Nachrichtentyp. Besonders nützlich ist dies, wenn man eine Nachricht auf ihre Typkonformität prüfen möchte. Meist ist der Implementierer gezwungen, eine eingetroffene Nachricht im Einzelnen auf korrekte Datentypen zu prüfen; die Klasse *MessageType* bietet hierfür die Methode

```
public boolean canBeAssigned(Object[] aBody)
```

an. Diese Methode liefert genau dann *true*, wenn jedes Element des übergebenen Objektfeldes dem jeweiligen Nachrichtenelementtyp entspricht. Dieses Objektfeld ist gewöhnlich die Nachrichtennutzlast.

Die Bildung eines syntaktischen Typs erfolgt über den Konstruktor von *SyntacticType:*

```
public SyntacticType(MessageType[] amtInput,
                     MessageType[] amtOutput)
```

Die Klasse bietet die Methoden *findInputMessages* und *findOutputMessages* an, welche alle Nachrichtentypen liefern, deren Annotation mit der übergebenen Annotation übereinstimmen:

```
public MessageType[] findInputMessages(Annotation ann,
                                       CGOntologySet cg)
public MessageType[] findOutputMessages(Annotation ann,
                                        CGOntologySet cg)
```

Eine Anwendung hierfür ist das Auffinden einer bestimmten Nachricht nach der Feststellung, dass die Typen insgesamt kompatibel sind.

E.2.4 TransitionType, Zustände und Transitionen

Zustände können eine Annotation besitzen; sie verfügen in der Regel über mehrere von ihnen ausgehende Transitionen, sofern sie keine Endzustände sind. Die Generierung eines Zustands und das Setzen der Annotation erfolgt durch

```
public State(Transition[] atr)
public void setAnnotation(Annotation ann)
```

Eine Transition lässt sich mittels folgendem Konstruktor erzeugen:

```
public Transition(MessageType mtInput, Annotation sender,
                  MessageType mtOutput, Annotation receiver,
                  State stNew)
```

Man beachte, dass jede Transition nur genau eine Ausgabenachricht vorsieht. Soll eine Kette von Ausgangsnachrichten repräsentiert werden, so müssen entsprechende Zwischenzustände definiert werden.

Die Erzeugung eines Transitionstyps erfolgt dann mittels des Konstruktors:

```
public TransitionType(State stStart)
```

Der gesamte endliche Automat, welcher den Transitionstyp ausmacht, wird somit durch den Startzustand und alle darüber transitiv erreichbaren Zustände und Transitionen in der Klasse *TransitionType* verankert.

E.3 Treiberschnittstelle

Die Treiberschnittstelle *AMETAS.place.AMETASPlaceUserDriverIf* trennt den Stellennutzer von der eigentlichen Implementierung des Treibers und nimmt dadurch auch die Trennung zwischen der Anwendungsebene und der AMETAS-Kernebene vor. Neben den zahlreichen Methoden, etwa zum Holen und Verschicken von Nachrichten, gibt es Methoden, welche im Zusammenhang mit dem Typsystem stehen. (Die Ausnahmebedingungen sind hier nicht genannt; für Details wird auf die Beschreibung der Programmierschnittstelle von AMETAS verwiesen.)

```
AMETASPlaceUserID requestPUStartup(String sSPUName,
                                   AMETASMessage msgInit)
AMETASPlaceUserID requestPUStartup(AMETASMediationResult mrs,
                                   AMETASMessage msgInit)
```

Die *requestPUStartup*-Methoden dienen dem Start von eigenständigen Stellennutzern. Da es für einen Stellennutzer nicht möglich ist, Agenten in AMETAS direkt vermöge eines Konstruktors zu instantiieren, übernimmt die Nachricht *msgInit* die Rolle der Konstruktorparameter. Diese Nachricht wird von der Stelle automatisch mit der Subkategorie *INIT* versehen und kann vom neu entstandenen Stellennutzer, dessen ID dem Starter übergeben wird, verarbeitet werden.

Das Objekt *AMETASMediationResult* beinhaltet eine Zeichenkette, welche den Namen des SPUs bezeichnet, welcher in Folge einer Typvermittlung als passender Stellennutzer beurteilt wurde. Da die Referenzierung von SPUs zurzeit lediglich über eine Zeichenkette erfolgt, kann der Start auch durch diese Zeichenkette erfolgen.

```
AMETASType getTypeOf(AMETASPlaceUserID puid)
AMETASType getTypeOf(String sRegistered)
AMETASType getPUType()
```

Beide *getTypeOf*-Methoden liefern dem aufrufenden Stellennutzer den Typ des bezeichneten Stellennutzers. Dabei handelt es sich im ersten Falle um eine existierende Instanz, im zweiten Falle um den Namen des zugehörigen SPU-Containers. Dieser muss jedoch im Typverzeichnis der aktuellen Stelle registriert sein. Die dritte Form liefert den eigenen Typ. Dieser kann mittels

```
void setPUType(AMETASType type)
```

vom Stellennutzer selbst korrigiert werden, was besonders bei Diensten von Interesse ist, die unter verschiedenen Parametrisierungen gestartet werden können. (Genaueres ist unter [ZH01] zu finden.)

```
AMETASMediationResult[] request(AMETASMediationRequest mreq)
AMETASMediationResult[] request(AMETASMediationRequest mreq,
                                AMETASPlaceUserIDMask puidm,
                                AMETASIdentityID iid)
```

Diese Methoden übergeben die Typanfrage an den aktuellen Mediator. Der Stellennutzer kann den Mediator nicht auswählen, er kann jedoch beim Start der Stelle angegeben werden. Werden Übereinstimmungen gefunden, so werden diese als Feld übergeben, das der Stellennutzer selbst zu inspizieren hat. Die zweite Form erlaubt die automatische Filterung des Suchergebnisses mittels einer ID-Maske beziehungsweise mittels der Identitäts-ID, welche den Starter des jeweils gefundenen Stellennutzers angibt.

```
boolean stillPresent(AMETASMediationResult mrs)
long getLeavingTime(AMETASMediationResult mrs)
```

Diese Methoden dienen dazu, festzustellen, ob ein Stellennutzer, dessen ID als Ergebnis einer Vermittlung erhalten wurde, noch an der Stelle verweilt oder ob er (im Falle eines Agenten) mittlerweile abgereist ist.

```
MediatorInfo getMediatorInfo()
```

Es ist im Hinblick auf die Interaktion mit dem Mediator unerlässlich zu erfahren, welcher Mediator an der Stelle aktiv ist. Auf diesem Wege verschafft sich der Stellennutzer diese Information.

```
AMETASType createTypeForString(String sDescr, String sMode)
AMETASType createTypeForString(String sDescr)
```

Diese Methode bietet eine Schnittstelle zum Typcompiler im Hybridtypsystem. Typbeschreibungen liegen in der Regel in einer statischen Textform vor und müssen zur Verarbeitung zunächst in eine Typinstanz konvertiert werden. Der Stellennutzer muss den Mediator kennen; der Modus ist mediatorabhängig. Wird der Modus weggelassen, wird eine mediatorabhängige Standardvorgabe angenommen.

F Automatische Codegenerierung

Um die im Abschnitt 8.3 aufgestellten Regeln zur Erzeugung von einem Quellcode-Grundgerüst zu illustrieren, sei folgendes Beispiel gegeben:

```
messages {
   in {
      Route: { placeList: java.lang.String[] };
      Result: { freeSpace: AMETAS.data.ALong };
   }
   out {
      DriveQuery: {auth:java.lang.String };
      FinalRes: { critList: java.lang.String[] };
      Mig: { migration, pl:java.lang.String };
   }
}
states {
   Wait =Route> ChkNext:Mig(place);
   ChkNext =none> WaitResp:DriveQuery(other) +
                  ComeHome:Mig(place);
   WaitResp =Result(other)> ChkNext:Mig(place);
   ComeHome =none> Ending:FinalRes;
}
annotations {
   ...
}
```

F.1 Analyse und Codegenerierung

Die Annotationen spielen bei der Quellcodeerzeugung, wie sie in Abschnitt 8.3 vorgestellt wurde, keine Rolle.[1] Anhand der Selbstannotation kann gegebenenfalls entschie-

[1] Man kann sich überlegen, dass die Konstantenannotationen in der Rolle als Diskriminatoren zur Bildung von *If-else*-Blöcken herangezogen werden können; dies sind jedoch Implementationsdetails.

den werden, welcher Art der Stellennutzer ist. Zunächst werden die Zustände zusammengetragen:

Wait, ChkNext, WaitResp, ComeHome, Ending.

Weitere Marken referenzieren Nachrichten:

Route, Result, DriveQuery, FinalRes, Mig,

wobei die letzte Nachricht eine Pseudonachricht ist, erkennbar am Nachrichtenelement des Datentyps *migration.* Ferner sind die Marken zu beachten, welche Absender und Empfänger einer Nachricht kennzeichnen:

place, other,

beides vordefinierte Annotationen. Nach Anwendung der in Abschnitt 8.3 aufgestellten Regeln lässt sich folgender Code automatisiert generieren:

```
import AMETAS.agentdev.*;
import AMETAS.data.*;
import AMETASx.data.htype.*;

public class MyAgent extends AMETASAgent {
   private short m_nState;
   private static final short stWAIT=0;
   private static final short stCHKNEXT=1;
   private static final short stWAITRESP=2;
   private static final short stCOMEHOME=3;
   private static final short stENDING=4;
   private static final short stSTART=stWAIT;

   private MessageType m_mtRoute;
   private MessageType m_mtResult;

   private AMETASPlaceUserID m_puidOther1;
   private AMETASPlaceUserID m_puidOther2;

   private AMETASPlaceUserID m_puidInit;

   private transient boolean m_bTerminated=false;

   public MyAgent() {
      m_nState = stSTART;
      try {
```

```
      m_mtRoute = new MessageType(”java.lang.String[]”);
      m_mtResult = new MessageType(”AMETAS.data.ALong”);
   }
   catch (MalformedTypeException mx) {
   }
   catch (AccountExpiredException aex) {
   }
}

public void invoke() {
   m_bTerminated=false;
   while (!m_bTerminated) {
      switch (m_nState) {
          case stWAIT:
              stateWait(); break;
          case stCHKNEXT:
              stateChkNext(); break;
          case stWAITRESP:
              stateWaitResp(); break;
          case stCOMEHOME:
              stateComeHome(); break;
          case stENDING:
              stateEnding(); break;
          default:
              break;
      }
   }
}

private void stateWait() {
   m_Driver.checkMessages();
   AMETASMessage msgRoute = m_Driver.getNextMessage();
   if (m_mtRoute.canBeAssigned(msgRoute.getBody())) {
      m_puidInit = msgRoute.getSenderID();
      m_nState=stCHKNEXT;
      try {
         m_Driver.go(”SomePlace”,DONT_RELAY);
      }
      catch (MigrationException mx) {
      }
      catch (AccountExpiredException aex) {
      }
   }
}
```

```
private void stateChkNext() {
  short nCondition=0;
  switch (nCondition) {
     case 0:
        m_nState=stWAITRESP;
        String sElem1 = "<enter some text>";
        Object[] aBody = new Object[1];
        aBody[0] = sElem1;
        m_puidOther1 = null;
        AMETASMessage msgDriveQuery=
           new AMETASMessage(m_puidOther1, aBody);
        m_Driver.depositMessage(msgDriveQuery);
        break;
     case 1:
        m_nState = stCOMEHOME;
        try {
           m_Driver.go("SomePlace",DONT_RELAY);
        }
        catch (MigrationException mx) {
        }
        catch (AccountExpiredException aex) {
        }
        break;
     default:
        break;
  }
}

private void stateWaitResp() {
   m_Driver.checkMessages();
   AMETASMessage msgResult = m_Driver.getNextMessage();
   if (m_mtResult.canBeAssigned(msgResult.getBody())) {
     if (msgResult.getSenderID().equals(m_puidOther2)) {
         m_nState=stCHKNEXT;
         try {
            m_Driver.go("SomePlace",DONT_RELAY);
         }
         catch (MigrationException mx) {
         }
      }
   }
}
```

```
    private void stateComeHome() {
       m_nState=stENDING;
       String[] asElem1 = new String[0];
       Object[] aBody = new Object[1];
       aBody[0] = asElem1;
       AMETASMessage msgFinalRes=new AMETASMessage(m_puidInit, aBody);
       m_Driver.depositMessage(msgFinalRes);
    }

    private void stateEnding() {
       m_bTerminated=true;
    }
}
```

F.2 Bemerkungen zum generierten Code

Der oben dargestellte Quellcode trägt keinerlei Implementationsdetails, welche nicht durch die Typbeschreibung gedeckt sind. Einige Bemerkungen seien zu den einzelnen Methoden angebracht.

F.2.1 Felder

Da es keine gegenseitigen Beziehungen unter den Annotationen gibt, muss jedes Vorkommnis von *OTHER* separat behandelt werden. Während beim Empfang lediglich geprüft werden sollte, ob der Absender sich vom initialen Absender unterscheidet, benötigt man eine konkrete Adresse beim Verschicken.

Der initiale Absender, also jener, welcher den Agenten in seinem Startzustand anspricht, muss gespeichert werden, damit eine Antwort an diesen gerichtet werden kann. Im Falle konkurrierender Anfragen müssen die verschiedenen Anfrager separat gespeichert werden.

Da die Annotation *PLACE* nur in Verbindung mit der Nachricht *Mig* auftaucht, muss sie nicht gesondert repräsentiert werden. Prinzipiell könnte das Nichtvorhandensein dieser Annotation mit der Nachricht *Mig* bereits vom Typcompiler als Fehler gemeldet werden.

F.2.2 Konstruktor

Der Konstruktor eines Agenten wird nur bei seiner erstmaligen Erzeugung aufgerufen. Deshalb eignet er sich für einmalige Initialisierungen und Festlegung von Werten, welche nach einer Migration erhalten bleiben sollen. Dies betrifft etwa den Zustand im Falle dieses Agenten, aber auch die Nachrichtentypen, welche hier im Konstruktor gebildet werden, um Wiederholungen in verschiedenen Methoden zu sparen.

F.2.3 invoke

Die *invoke*-Methode besteht lediglich aus einer Schleife, welche die den Zuständen entsprechenden Methoden aufruft. Setzt eine der Methoden das Feld *m_bTerminated* auf *true*, so wird die Schleife verlassen. Dies zieht im AMETAS-Agentensystem die Terminierung des Agenten nach sich.

F.2.4 Zustandsmethoden

Für jeden Zustand des Agenten wird eine Methode konstruiert, die von *invoke* aufgerufen wird. Ist der Zustand stabil oder semistabil, so wird eine Abfrage der vorhandenen Nachrichten durchgeführt.[2] Anschließend kann mittels *canBeAssigned* überprüft werden, ob die Nachricht syntaktisch mit der erwarteten Nachricht übereinstimmt. Anderenfalls sind geeignete Maßnahmen zu treffen, etwa eine Fehlermeldung auszugeben oder die Nachricht zu verwerfen.

Das Pseudonachrichtenelement *migration* führt dazu, dass die Nachricht, welches es enthält, nicht als Nachricht umgesetzt wird, sondern entsprechend der AMETAS-API als *go*-Aufruf. Die Zielstelle muss geeignet ergänzt werden; bei Scheitern der Migration kann der *catch*-Block eine Fehlerbehandlung durchführen.

Treten Nichtdeterminismen auf, wird ein *switch-case*-Konstrukt aufgebaut, welches die verschiedenen Alternativen aufzählt. Der Implementierer muss dafür sorgen, dass die *switch*-Variable so gesetzt wird, dass die erwünschte Alternative gewählt wird.

Es bleibt zu beachten, dass AMETAS-spezifische Eigenschaften wie die Ereignisverwaltung von der Typdefinition nicht berührt werden, sofern die Entgegennahme von Ereignissen lediglich zur Entscheidung bei nichtdeterministischen Alternativen beiträgt oder die Ankunft von Nachrichten signalisiert. Der Stellennutzer muss in diesem Falle lediglich die Schnittstelle *AMETASNotifiable* implementieren, also eine Implementierung der Methode *notify* liefern. Außerdem kann der Agent weiterhin autonom Entscheidungen treffen, indem die Zustandsmethoden im Falle eines leeren Postfachs den Agenten beliebigen anderen Code ausführen lassen.

[2] Die Methoden *checkMessages* und *getNextMessage* sind nicht Bestandteil der AMETAS-API 2.5, sind aber für eine spätere Version vorgesehen. Zurzeit ist es möglich, das Postfach mittels *getMessages* auszulesen.

Literaturverzeichnis

[AB99] Reticular Systems Inc.: *AgentBuilder Users Guide*. 1999. – http://www.agentbuilder.com

[AHU74] AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D.: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. – ISBN 0–201–00029–6

[And91] ANDREWS, G. R.: *Concurrent programming, principles and practise*. Benjamin/Cummings Publishing Company, 1991. – ISBN 0–8053–0086–4

[AO98] ARIDOR, Y.; OSHIMA, M.: Infrastructure for Mobile Agents: Requirements and Design. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, September 1998

[AOS01] Agent Oriented Software Pty. Ltd.: *WWW-Seiten zu Jack*. 2001. – http://www.agent-software.com.au/

[BFD96] BIC, L. F.; FUKUDA, M.; DILLENCOURT, M.: Distributed computing using autonomous objects. In: *IEEE Computer* 29 (1996), August, Nr. 8

[BHR+97] BAUMANN, J.; HOHL, F.; RADOUNIKLIS, N. [u. a.]: Communication Concepts for Mobile Agent Systems. In: *Mobile Agents – First International Workshop, MA '97, Berlin*, Springer-Verlag, 1997. – LNCS 1219. – ISBN 3–540–62803–7, S. 123–135

[BIP88] BRATMAN, M. E.; ISRAEL, D. J.; POLLACK, M. E.: Plans and Resource-Bounded Practical Reasoning. In: *Computational Intelligence* 4 (1988)

[Boo91] BOOCH, G.: *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Co., 1991. – ISBN 0–8053–5340–2

[Bou92] BOUDOL, G.: Asynchrony and the pi-calculus / INRIA. 1992 (1702). – Forschungsbericht

[BR98] BAUMANN, J.; ROTHERMEL, K.: The Shadow Approach: An Orphan Detection Protocol for Mobile Agents. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, September 1998. – ISBN 3–540–64959–X, S. 2–13

[Bro86] BROOKS, R. A.: A Robust Layered Control System for a Mobile Robot. In: *IEEE Journal of Robotics and Automation* 2 (1986), Nr. 1, S. 14–23

[Bro91] BROOKS, R. A.: Intelligence without Reason. In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence*. Menlo Park, CA, USA: Morgan Kaufmann, 1991, S. 569–595

[Car97] Kap. 103: "Type Systems" in: CARDELLI, L.: *The Computer Science and Engineering Handbook*. CRC Press, 1997, S. 2208–2236. – ISBN 0–8493–2909–4

[CFSD90] CASE, J.; FEDOR, M.; SCHOFFSTALL, M.; DAVIN, J.: *A Simple Network Management Protocol (SNMP)*. Network working group, 1990. – Internet-Standard RFC 1157, http://www.ietf.org/rfc/rfc1157.txt

[CG89] CARRIERO, N.; GELERNTER, D.: Linda in context. In: *Communications of the ACM* 32 (1989), April, Nr. 4, S. 444–458. – ISSN 0001–0782

[CG98] CARDELLI, L.; GORDON, A. D.: Mobile Ambients. In: NIVAT, M. (Hrsg.): *Foundations of Software Science and Computation Structures*, Springer-Verlag, April 1998 (LNCS 1378). – ISBN 3–540–64300–1, S. 140–155

[CG99] CARDELLI, L.; GORDON, A. D.: Types for Mobile Ambients. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, USA*, 1999, S. 79–92

[dH84] DE NICOLA, R.; HENNESSY, M. C. B.: Testing equivalences for processes. In: *Theoretical Computer Science* 34 (1984), November, Nr. 1-2, S. 83–133. – ISSN 0304–3975

[Dud90] Dudenredaktion (wiss. Rat): *Duden: Das Fremdwörterbuch*. 1990. – S. 551

[Eck95] ECKERSON, W.: Three-Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client-Server Applications. In: *Open Information Systems 10* 3 (1995), Januar, Nr. 20

[EW94] ETZIONI, O.; WELD, D.: A softbot-based interface to the Internet. In: *Communications of the ACM* 37 (1994), Juli, Nr. 7, S. 72–76. – ISSN 0001–0782

[FBDM98] FUKUDA, M.; BIC, L. F.; DILLENCOURT, M.; MERCHAND, F.: Distributed coordination with messengers. In: *Science of computer programming* 31 (1998), Nr. 2. – Special issue on coordination models, languages, applications

[FG97] FRANKLIN, S.; GRAESSER, A.: Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents. In: *Intelligent Agents III. Agent Theories, Architectures, and Languages – ECAI'96 Workshop (ATAL), Budapest, Ungarn*, Springer-Verlag, 1997 (LNCS 1193)

[FIP00] Foundation for intelligent physical agents (FIPA): *FIPA ACL Message Structure Specification*. 2000. – FIPA Document Number XC00061D, http://www.fipa.org/specs/fipa00061/

[FIP01] *FIPA Abstract Architecture Specification*, Januar 2001. – FIPA Document Number XC00001I, http://www.fipa.org/specs/fipa00001/

[FM99] FÜNFROCKEN, S.; MATTERN, F.: Mobile Agents as an Architectural Concept for Internet-Based Distributed Applications – The WASP Project Approach. In: STEINMETZ, R. (Hrsg.): *Kommunikation in Verteilten Systemen (KiVS) – 11. ITG/GI-Fachtagung, Darmstadt*, Springer-Verlag, März 1999. – ISBN 3–540–65597–2, S. 32–43

[Fün97] FÜNFROCKEN, S.: How to Integrate Mobile Agents into Web Servers. In: *Proceedings of the Workshop on Collaborative Agents in Distributed Web Applications (WETICE'97), Boston, USA*, 1997, S. 94–99

[Fün98] FÜNFROCKEN, S.: Transparent Migration of Java-Based Mobile Agents. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, September 1998. – ISBN 3–540–64959–X, S. 26–37

[FPV98] FUGGETTA, A.; PICCO, G. P.; VIGNA, G.: Understanding Code Mobility. In: *IEEE Transactions on Software Engineering* 24 (1998), Nr. 5, S. 342–361

[FW91] FININ, T.; WIEDERHOLT, G.: An Overview on KQML / Stanford University Logic Group. 1991. – Forschungsbericht

[Gen95] GENESERETH, M. R. *Knowledge Interchange Format*. http://logic.stanford.edu/kif/. 1995

[GF92] GENESERETH, M. R.; FIKES, R. E.: Knowledge Interchange Format Version 3 Reference Manual / Stanford University Logic Group. 1992 (Logic-92-1). – Forschungsbericht

[GK94] GENESERETH, M. R.; KETCHPEL, S. P.: Software agents. In: *Communications of the ACM* 37 (1994), Juli, Nr. 7, S. 48–53. – ISSN 0001–0782

[GPP+98] GEORGEFF, M.; PELL, B.; POLLACK, M.; TAMBE, M.; WOOLDRIDGE, M.: The Belief-Desire-Intention Model of Agency. In: MÜLLER, J. (Hrsg.); SINGH, M. (Hrsg.); RAO, A. (Hrsg.): *Intelligent Agents V: Agents Theories, Architectures, and Languages – 5th International Workshop, ATAL'98, Paris, Frankreich*, Springer-Verlag, Juli 1998 (LNCS 1555). – ISBN 3–540–65713–4

[Grü97] GRÜNDER, H.: *Zur Anwendung des Objektmodells in verteilten Systemen*, Johann Wolfgang Goethe-Universität, Diss., Dezember 1997

[Gra96] GRAY, R.: Agent Tcl: A flexible and secure mobile-agent system. In: DIEKHANS, M. (Hrsg.); ROSEMAN, M. (Hrsg.): *4th Annual Tcl/Tk Workshop, Monterey*, 1996

[Gru95] GRUBER, T. R.: Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In: *International Journal of Human-Computer Studies* 43 (1995), Nr. 6, S. 907–928

[GZMW01] GRIFFEL, F.; ZIRPINS, C.; MÜLLER-WILKEN, S.: Generative Softwarekonstruktion auf Basis typisierter Komponenten. In: KILLAT, U. (Hrsg.); LAMERSDORF, W. (Hrsg.): *Kommunikation in Verteilten Systemen (KiVS) – GI/ITG-Fachtagung, Berlin*, Springer-Verlag, Februar 2001. – ISBN 3–540–41645–5, S. 325–338

[Hen85] HENNESSY, M.: Acceptance Trees. In: *Journal of the Association for Computing Machinery* 32 (1985), Oktober, Nr. 4, S. 896–928

[Hew77] HEWITT, C.: Viewing Control Structures as Patterns of Passing Messages. In: *Artificial Intelligence* 8 (1977), Nr. 3, S. 323–364

[HM95] HAFNER, K.; MARKOFF, J.: *Cyberpunk: Outlaws and hackers on the computer frontier*. Touchstone Books, 1995. – ISBN 0–684–81862–0

[Hog89] HOGREFE, D.: *Estelle, LOTOS und SDL*. Berlin: Springer-Verlag, 1989. – ISBN 3–540–50477–X

[HU88] HOPCROFT, J. E.; ULLMAN, J. D.: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1988. – ISBN 3–925118–82–9

[IKV01] IKV++ GMBH. *WWW-Seiten zum Agentensystem Grasshopper*. http://www.grasshopper.de/. 2001

[ISO91] International Organization for Standardization: *Information processing systems — Open systems interconnection — Guidelines for the Application of Estelle, LOTOS, and SDL*. 1991. – Dokumentnummer 10167

[ISO97] International Organization for Standardization: *Information processing systems — Open systems interconnection — Estelle — A formal description technique based on an extended state transition model*. 1997. – Dokumentnummer 9074

[ITU95] International Telecommunication Union: *ODP Trading Function*. Mai 1995. – ITU/ISO Committee Draft Standard ISO/IEC DIS13235 Rec. X.9tr

[ITU99] International Telecommunication Union/Telecom Standardization (ITU-T): *Recommendation Z.100 – Specification and Description Language*. November 1999. – http://www.itu.int/

[IWKK00] ILLMANN, T.; WEBER, M.; KARGL, F.; KRUEGER, T.: Migration of Mobile Agents in Java: Problems, Classification and Solutions. In: *International ICSC Congress: Intelligent Systems & Applications ISA '2000, Wollongong, Australien* Bd. 1, ICSC Academic Press, Dezember 2000. – ISBN 3–906454–24–X, S. 196–202

[Jah01] JAHR, U.: *Agentenkommunikationssprachen im AMETAS-Agentensystem*. 2001. – Diplomarbeit am Institut für Informatik der Johann Wolfgang Goethe-Universität

[JAT98] Stanford University: *WWW-Seiten zu Java Agent Template Lite*. 1998. – http://java.stanford.edu/

[KRR98] KOVACS, E.; RÖHRLE, K.; REICH, M.: Integrating Mobile Agents into Mobile Middleware. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, September 1998. – ISBN 3–540–64959–X, S. 124–135

[Lab96] LABROU, Y.: *Semantics for an Agent Communication Language*, Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Diss., 1996. – Zu beziehen über den Autor: jklabrou@cs.umbc.edu

[LD98] LINGNAU, A.; DROBNIK, O.: Agent-User Communications: Requests, Results, Interaction. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, 1998. – ISBN 3–540–64959–X, S. 209–221

[Lia99] LIANG, S.: *The Java Native Interface: Programmer's Guide and Specification (Java Series)*. Addison-Wesley, Juni 1999. – ISBN 0–201–32577–2

[LO98] LANGE, D.; OSHIMA, M.: *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998. – ISBN 0–201–32582–9

[LW93] LISKOV, B.; WING, J.: A New Definition of the Subtype Relation. In: NIERSTRASZ, O. (Hrsg.): *ECOOP'93 – Object-oriented Programming*, Springer-Verlag, Juli 1993. – ISBN 3–540–57120–5, S. 118–141

[LW94] LISKOV, B.; WING, J.: A Behavioral Notion of Subtyping. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), Nr. 6, S. 1811–1841

[Mae91] Kap. "Situated Agents Can Have Goals" in: MAES, P.: *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*. London: The MIT Press, 1991, S. 49–70

[Mae94] MAES, Pattie: Agents that reduce work and information overload. In: *Communications of the ACM* 37 (1994), Juli, Nr. 7, S. 30–40. – ISSN 0001–0782

[Mae95] MAES, Pattie: Artificial Life Meets Entertainment: Lifelike Autonomous Agents. In: *Communications of the ACM* 38 (1995), November, Nr. 11, S. 108–114. – ISSN 0001–0782

[Mat89] MATTERN, F.: *Verteilte Basisalgorithmen*. Springer-Verlag, 1989 (IFB 226). – ISBN 3–540–51835–5

[MBB+98] MILOJICIC, D.; BREUGST, M.; BUSSE, I. [u. a.]: MASIF – The OMG Mobile Agent System Interoperability Facility. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, September 1998. – ISBN 3–540–64959–X, S. 50–67

[Min85] MINSKI, M.: *The Society of Mind*. New York: Simon & Schuster, 1985. – ISBN 0–671–65713–5

[Mit96] MITCHELL, John C.: *Foundations for Programming Languages*. Cambridge: The MIT Press, 1996. – ISBN 0–262–13321–0

[MPW92] MILNER, R.; PARROW, J.; WALKER, D.: A calculus of Mobile Processes (Teil 1 und 2). In: *Information and Computation* 100 (1992), September, S. 1–77

[MS98] MIKHAJLOV, L.; SEKERINSKI, E.: A study of the fragile base class problem. In: *ECOOP'98 - Object-Oriented Programming; 12th European Conference, Brussels, Belgium*, Springer-Verlag, Juli 1998 (LNCS 1445). – ISBN 3–540–64737–6, S. 355–382

[NAS97] NASA. *Webseite zum Projekt Mars Pathfinder*. http://mars.jpl.nasa.gov/. 1997

[Nie95] NIERSTRASZ, O.: Regular Types for Active Objects. In: NIERSTRASZ, O. (Hrsg.); TSICHRITZIS, D. (Hrsg.): *Object-Oriented Software Composition*. Prentice Hall, 1995. – Frühere Version im Tagungsband zur Konferenz *OOPSLA '93*, veröffentlicht in den *ACM Sigplan Notices*, 28(10), Oktober 1993, S. 1–15. – ISBN 0–132–20674–9, S. 99–121

[Nwa96] NWANA, H. S.: Software Agents: An Overview. In: *Knowledge Engineering Review* 11 (1996), September, Nr. 3, S. 1–40. – http://agents.umbc.edu/introduction/ao/

[Obj01] OBJECTSPACE. *WWW-Seiten zu Voyager 4.0*. http://www.objectspace.com/products/voyager/. 2001

[OG97] The Open Group: *DCE 1.2.2: Introduction to OSF DCE*. November 1997. – http://www.opengroup.org/publications/catalog/f201.htm

[OMG97] Object Management Group: *OMG Mobile Agent System Interoperability Facility*. 1997. – OMG-TC-Dokument ORBOS/97-10-05

[OMG99] Object Management Group: *OMG Unified Modelling Language Specification Version 1.3*. 1999. – http://www.omg.org/

[OMG00a] Object Management Group: *The Common Object Request Broker Architecture and Specification*. Revision 2.4.1. November 2000. – OMG Formal Document 2000-11-03

[OMG00b] Object Management Group: *Trading Object Service Specification*. Version 1.0. Mai 2000. – OMG Formal Document 00-06-27

[Pap01] PAPAIOANNOU, T. *WWW-Seiten zum Agentensystem Aglets*. http://aglets.org/. 2001

[Paz66] PAZ, A.: Some aspects of probabilistic automata. In: *Information and Control* (1966), Februar, S. 26–60

[PG97] PUDER, A.; GEIHS, K.: Meta-level Service Type Specification. In: *Proceedings of the IFIP International Conference on Open Distributed Processing, Toronto, Kanada*, 1997

[PMG95] PUDER, A.; MARKWITZ, S.; GUDERMANN, F.: Service Trading Using Conceptual Structures. In: *International Conference on Conceptual Structures*, Springer-Verlag, 1995

[Pud97] PUDER, A.: *Typsysteme für die Dienstvermittlung in offenen verteilten Systemen*, Johann Wolfgang Goethe-Universität, Diss., Januar 1997

[Pun96] PUNTIGAM, F.: Types for Active Objects based on Trace Semantics. In: NAJM, E. (Hrsg.); STEFANI, J.-B. (Hrsg.): *Formal Methods for Open Object-based Distributed Systems – FMOODS '96, Paris, Frankreich*, Kluwer Academic Publishers, März 1996. – ISBN 0–412–79770–4, S. 4–19

[Pun97] PUNTIGAM, F.: Coordination Requirements Expressed in Types for Active Objects. In: *ECOOP'97 – Object-Oriented Programming*, Springer-Verlag, 1997 (LNCS 1241). – ISBN 3–540–63089–9, S. 367–388

[RD00] RICORDEL, P.-M.; DEMAZEAU, Y.: From Analysis to Deployment: A Multi-Agent Platform Survey. In: A. OMICINI, A. (Hrsg.); TOLKSDORF, R. (Hrsg.); ZAMBONELLI, F. (Hrsg.): *Engineering Societies in the Agents World*, Springer-Verlag, 2000 (LNCS 1972). – ISBN 3–540–41477–0, S. 93–105

[Ros91] ROSE, M. T.: *The Simple Book, An Introduction to Management of TCP/IP-based Internets*. Prentice-Hall, 1991

[RTL00] RAY, P.; TOLEMAN, M.; LUKOSE, D.: Could Emotions be the Key to Real Artificial Intelligence? In: *International ICSC Congress: Intelligent Systems & Applications ISA '2000, Wollongong, Australien* Bd. 2, ICSC Academic Press, Dezember 2000. – ISBN 3–906454–24–X, S. 45–50

[Rus03] RUSSEL, B.: *The principles of mathematics*. Cambridge University Press, 1903

[Sow84] SOWA, J. F.: *Conceptual Structures*. Addison–Wesley Publishing Company, 1984. – ISBN 0–201–14472–7

[Sri95] SRINIVASAN, R.: *RPC: Remote Procedure Call Protocol Specification Version 2*. Network working group, 1995. – Internet-Standard RFC 1831, http://www.ietf.org/rfc/rfc1831.txt

[SSD98] SILVA, A.; DA SILVA, M. M.; DELGADO, J.: An Overview of AgentSpace: A Next-Generation Mobile Agent System. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, 1998. – ISBN 3–540–64959–X, S. 148–159

[Sta93] STALLINGS, W.: *SNMP, SNMPv2, and CMIP*. Addison–Wesley Publishing Company, 1993

[Ste01] STEARNS, B.: *The Java Tutorial: Java Native Interface*. Sun Microsystems, 2001. – http://java.sun.com/docs/books/tutorial/

[Sto96] STONE, H. W.: Mars Pathfinder Microrover: A Low-Cost, Low-Power Spacecraft. In: *Proceedings of the 1996 AIAA Forum on Advanced Developments in Space Robotics, Madison, USA*, 1996

[Tan92] TANENBAUM, Andrew S.: *Modern Operating Systems*. Prentice-Hall, 1992. – ISBN 0–13–595752–4

[TOH00] TAHARA, Y.; OHSUGA, A.; HONIDEN, S.: Agent Communication Language Patterns and Their Tool Support. In: *International ICSC Congress: Intelligent Systems & Applications ISA '2000, Wollongong, Australien* Bd. 1, ICSC Academic Press, Dezember 2000. – ISBN 3–906454–24–X, S. 328–334

[Urb00] Kap. "PECS: A Reference Model for the Simulation of Multi-Agent Systems" in: URBAN, C.: *Tools and techniques for Social Science Simulation*. Heidelberg: Physica-Verlag, 2000. – ISBN 3–7908–1265–X

[W3C00] WWW Consortium (W3C): *Extensible Markup Language 1.0*. Oktober 2000. – http://www.w3.org/TR/2000/REC-xml-20001006

[WBD+98] WICKE, C.; BIC, L. F.; DILLENCOURT, M. [u. a.]: Automatic State Capture of Self-Migrating Computations in Messengers. In: *Mobile Agents – Second International Workshop, MA '98, Stuttgart*, Springer-Verlag, September 1998. – ISBN 3–540–64959–X, S. 68–79

[Whi94] WHITE, J. E.: *Telescript Technology: The foundation for the Electronic Marketplace*. General Magic, 1994. – Nicht mehr verfügbar

[WJ95] WOOLDRIDGE, M.; JENNINGS, N.: Intelligent Agents: Theory and Practice. In: *Knowledge Engineering Review* 10 (1995), Juni, Nr. 2

[Woo95] WOOLDRIGDE, M. J.: Conceptualising and Developing Agents. In: *Proceedings of the UNICOM Seminar on Agent Software*, 1995, S. 40–54

[Woo97] WOOLDRIDGE, M. J.: Agent-based software engineering. In: *IEE Proceedings on Software Engineering* 144 (1997), Nr. 1, S. 26–37

[WZ88] WEGNER, P.; ZDONIK, S. B.: Inheritance as an Incremental Modification mechanism or What Like Is and Isn't Like. In: GJESSING, S. (Hrsg.); NYGAARD, K. (Hrsg.): *ECOOP '88 – Object-Oriented Programming*, Springer-Verlag, 1988 (LNCS 322), S. 55–77

[Zap97] ZAPF, M.: Design paradigms in agend-based systems. In: KÖNIG, H. (Hrsg.); GEIHS, K. (Hrsg.); PREUSS, T. (Hrsg.): *Distributed Applications and Interoperable Systems – IFIP TC6 WG6.1 International Working Conference DAIS '97, Cottbus*, Chapman & Hall, 1997. – ISBN 0–412–82340–3, S. 101–107

[Zap01] ZAPF, M.: Type-based Mediation of Mobile Agents. In: *International ICSC Congress: Intelligent Systems & Applications ISA '2000, Wollongong, Australien* Bd. 1, ICSC Academic Press, Dezember 2001. – ISBN 3–906454–24–X, S. 236–241

[ZH01] ZAPF, M.; HERRMANN, K.: *WWW-Seiten zum AMETAS-Projekt*. April 2001. – http://www.ametas.de/

[ZHG99a] ZAPF, M.; HERRMANN, K.; GEIHS, K.: Decentralized SNMP Management with Mobile Agents. In: SLOMAN, M. (Hrsg.); MAZUMDAR, S. (Hrsg.); LUPU, E. (Hrsg.): *Integrated Network Management VI (IM'99)*, IEEE/IFIP, Mai 1999. – ISBN 0–7803–5748–5, S. 623–635

[ZHG99b] ZAPF, M.; HERRMANN, K.; GEIHS, K.: Netz- und Systemmanagement mit mobilen Agenten und SNMP. In: MAIER, E. (Hrsg.): *Management verteilter IV-Systeme – 13. GI-Fachtagung über den Betrieb von Informations- und Kommunikationssystemen, Seeheim*, Lufthansa Systems, Mai 1999. – ISBN 3–00–004192–3, S. 351–366

[Zhu00] ZHU, H.: Formal Specification of Multi-Agent Systems: Software Engineering Perspective. In: *International ICSC Congress: Intelligent Systems & Applications ISA '2000, Wollongong, Australien*, ICSC Academic Press, Dezember 2000. – Plenarrede. – ISBN 3–906454–24–X

[ZMG98] ZAPF, M.; MÜLLER, H.; GEIHS, K.: Security Requirements for Mobile Agents in Electronic Markets. In: *Trends in Distributed Systems for Electronic Commerce – International IFIP/GI Working Conference, TREC'98, Hamburg*, Springer-Verlag, Juni 1998 (LNCS 1402). – ISBN 3–540–64564–0, S. 205–217

Stichwortverzeichnis

Lebenslauf

Name: Michael Zapf
Wohnort: Karben
Geburtsdatum: 24.09.1969
Geburtsort: Friedberg (Hess.)

Schulausbildung

1975–1979	Grundschule Klein-Karben
1979–1981	Kurt-Schumacher-Schule Klein-Karben
1981–1988	Georg-Büchner-Gymnasium Bad Vilbel
05/1988	Abitur

Wehrdienst

10/1988–12/1989	Grundwehrdienst (beurlaubt ab Oktober 1989)

Studium

10/1989	Aufnahme der Diplomstudiengänge Mathematik und Informatik an der Johann Wolfgang Goethe-Universität Frankfurt/Main
04/1995	Erlangung des Diploms der Mathematik; Betreuung der Diplomarbeit durch Prof. Dr. Jürgen Wolfart
01/1996	Aufnahme des Promotionsstudiengangs Informatik unter Betreuung von Prof. Dr. Kurt Geihs

Berufliche Tätigkeit

01/1996–03/2001	Beschäftigung als wissenschaftlicher Mitarbeiter an der Professur für Verteilte Systeme und Betriebssysteme am Fachbereich Informatik der Johann Wolfgang Goethe-Universität Frankfurt/Main
seit 04/2001	Beschäftigung als wissenschaftlicher Mitarbeiter am Institut für Sichere Telekooperation der Fraunhofer-Gesellschaft Darmstadt