Eine Umgebung für mobile Agenten: Agentenbasierte verteilte Datenbanken am Beispiel der Kopplung autonomer „Internet Web Site Profiler“

Diplomarbeit

Nico Johannisson

betreut von

Prof. Dr. Joachim W. Schmidt
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

Prof. Dr. Friedrich H. Vogt
Technische Universität Hamburg-Harburg
Arbeitsbereich Technische Informatik 5

22. April 1997

Zusammenfassung

Mobile Software-Agenten sind eine Erweiterung des konventionellen Client/Server-Modells für verteilte Systeme. Sie bilden die Basis eines allgemeinen Verteilungsansatzes auf der Grundlage von autonomen, mobilen Einzelkomponenten und vereinigen in sich Klientenrollen mit Dienstleisterrollen unter einem einzigen, anwendungsnahen Modellierungskonzept.

Die autonomieerhaltende Mobilität der Komponenten und ihre lose Kopplung untereinander ermöglicht neuartige Dienste und flexible Informationssysteme, welche insbesondere föderative Strukturen der realen Welt direkt auf Programmierkonzepte abbilden.

Allerdings erfordern derartige mobile Agenten eine definierte Umgebung und flexible Interaktionsprotokolle, die kooperatives Handeln der autonomen Komponenten unterstützen. Dabei stehen Probleme der Koordination nebenläufiger und migrierender Aktivitäten im Blickpunkt, wobei die Anwendungsnähe der Modellierungskonzepte erhalten werden muß.

Gegenstand dieser Arbeit ist, die Grundlagen für Kooperation und Autonomie von Softwaresystemen aufzuzeigen und vor diesem Hintergrund eine geeignete Umgebung für mobile Agenten mit den zugehörigen, anwendungsnahen Interaktionsprotokollen zu definieren. Deren Einsatz wird anhand einer kommerziellen verteilten Anwendung demonstriert.

Das Agentensystem ist im Rahmen der vorliegenden Arbeit unter Verwendung der persistenten, polymorphen Programmierumgebung Tycoon prototypisch realisiert worden.

Für Britta Christa und Sophia Fiona.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Im Zuge der weltweiten Verbreitung leistungsfähiger Telekommuniaktions- und Netzwerktechnologien, die gleichermaßen Unternehmen wie auch Privatpersonen kostengünstig zur Verfügung stehen, wandeln sich Informationssysteme von abgeschlossenen, zentralistischen Lösungen hin zu offenen, integrativen und dezentralisierten Komponentensystemen.

Dabei ist der über diese Systeme erreichbare Bestand an Informationen um den Faktor 10 bis 100 größer als jemals zuvor [DDJ+97]. Einen maßgeblichen Anteil an dieser Entwicklung trägt das globale Netzwerk Internet bei, insbesondere mittels seiner endnutzerorientierten Dienste wie dem elektronischen Postdienst und dem multimedialen Hypertextdienst des WorldWideWeb.

Ein weiterer Aspekt dieser Globalisierungstendenz ist die vom weltweiten Austausch von Informationen getragene Dynamik der Märkte. Anders als noch vor zehn Jahren verändern sich organisatiorische Gegebenheiten, Geschäftsprozesse und Systemumgebungen der am Markt befindlichen Unternehmen kontinuierlich und rapide, anstatt über längere Zeiträume hinweg stabil und damit langfristig planbar zu bleiben.

Stabilität und Planbarkeit sind jedoch wichtige Voraussetzungen, unter denen große, monolitische Informationssysteme entworfen und betrieben werden können. In dem Maße, in dem diese Voraussetzungen schwinden, wächst der Bedarf an Konzepten, Modellen und Methoden zum Entwurf von Systemen, welche die Auswirkungen von Veränderungen in der realen Welt tolerieren und ihnen folgen können.

1.1 Das ODP-Referenzmodell

Das Referenzmodell für *Open Distributed Processing*[1] (ODP) der International Standardization Organization (ISO) definiert einen koordinierenden Rahmen für den Entwurf offener und verteilter Systeme, deren Architektur die Probleme der zentralen bzw. monolitischen Informationssysteme überwindet.

Der ODP-Standard betrachtet offene, verteilte Systeme, die Organisations- und Technologiegrenzen überschreiten. Systeme dieser Art sind gekennzeichnet durch das Fehlen einer zentralen Kontrolle und besitzen daher die folgenden Charakteristika:

- ▷ Heterogenität: Es kann keine systemweit einheitliche Systemtechnologie vorausgesetzt werden und die vorhandene Systemtechnologie wird sich über den Lebenszyklus des Informationssystems ändern.

[1] Eine weitergehende Einführung in das ODP-Modell bietet [PSW96] und [ISO95a, ISO95b, ISO95c]

- ▹ Autonomie: Ein verteiltes System kann einer Vielzahl von autonomen Management- und Kontrollautoritäten unterliegen, die keine zentrale Kontrolle des Systems ermöglichen.
- ▹ Evolution: Veränderungen der realen Welt bedingen innerhalb des Lebenszyklus eines Systems Anpassungen an neue Erfordernisse.
- ▹ Mobilität: Informationsquellen, Netzknoten und Systemnutzer sind physisch mobil. Programme und Daten können ebenfalls aus Gründen der Systemperformanz oder der Abbildung von Anforderungen der realen Welt zwischen Rechnerknoten verschoben werden.

Um mit diesen Charakteristika umgehen zu können, zielt der ODP-Standard auf den Entwurf von Systemen, die sich primär durch die Eigenschaften *Offenheit*, *Integrativität*, *Flexibilität*, *Föderalismus* und *Transparenz* auszeichnen.

Offenheit ermöglicht einerseits, ein System ohne Änderungen auf verschiedenen Ausführungsknoten bzw. in unterschiedlichen Systemumgebungen zu betreiben (Portabilität) und andererseits, verschiedene Systeme bzw. Systemkomponenten sinnvoll miteinander interagieren zu lassen (Interoperabilität). Die Integration hat zum Ziel, verschiedenste Ressourcen und Systeme zu einem Gesamtsystem zu vereinigen, ohne daß dies kostenaufwendige ad-hoc Lösungen erfordert. Die Flexibilität zielt auf eine Unterstützung der Evolution von Systemen, einschließlich der zu berücksichtigenden Altsysteme (engl. *legacy systems*). Änderungen bzw. Neukonfigurationen eines ODP-Systems sollten dabei insbesondere auch zu dessen Laufzeit erfolgen können. Die Basis für diese Flexibilität bildet die strukturelle Modularisierung des ODP-Systems in autonome Teilsysteme, die jedoch untereinander in Beziehung stehen. Diese autonomen Teilsysteme werden im Rahmen von Föderationen, die verschiedene technologische und administrative Domänen einbeziehen, zu einem ODP-Gesamtsystem vereinigt. Eine zentrale Anforderung aus dem ODP-Referenzmodell, welche die Erstellung von verteilten Anwendungen insgesamt erleichtert, ist schließlich das Verbergen von Details der verteilten Programmierung vor dem Anwendungsentwickler. Dies erfordert geeignete Abstraktionen im verwendeten Programmiermodell und der Programmierumgebung, die zu diesem Zweck z.B. Mechanismen für Replikationstransparenz, Migrationstransparenz und Ortstransparenz anbieten können.

Um der Modellierungskomplexität eines offenen verteilten Systems zu begegnen, definiert das ODP-Referenzmodell das Konzept der Sichten (engl. *viewpoints*). Ein ODP-System wird aus fünf Sichten betrachtet, wobei jede Sicht spezielle Aspekte der vollständigen Systemspezifikation in den Vordergrund stellt. Im einzelnen sind diese Sichten:

- ▹ Die Unternehmenssicht, der *Enterprise Viewpoint*, umfaßt Zweck und Strategie des Systems. In ihr werden die Systemanforderungen beschrieben.
- ▹ Die Informationssicht, der *Information Viewpoint*, legt Wert auf die Bedeutung der Informationen und die Identifizierung der Aktivitäten der Informationsverarbeitung innerhalb des Systems.
- ▹ Die Verarbeitungssicht, der *Computational Viewpoint*, betrachtet die funktionale Zerlegung des Systems in Objekte, welche Kandidaten für eine Verteilung sind.
- ▹ Die Engineeringsicht, der *Engineering Viewpoint*, konzentriert sich auf die für die Verteilung notwendige Infrastruktur, die sog. Middleware.
- ▹ Die Technologiesicht, der *Technologie Viewpoint*, befaßt sich mit der Auswahl der für das System notwendigen Technologie.

Ein ODP-konformes System sollte alle Anforderungen, die sich aus der Betrachtung des Systems unter den genannten Sichtweisen ergeben, gleichermaßen erfüllen.

Obwohl in der Unternehmenssicht eine anwendungsorientierte Modellierung des Informationssystems vorgenommen wird, definiert das ODP-Referenzmodell vergleichsweise wenige Begriffe auf dieser Ebene. Lediglich die Konzepte Gemeinschaft und Föderation sind Teil der standardisierten Unternehmenssprache, der *Enterprise Language*.

Eine Systemmodellierung aus der Unternehmenssicht führt zur Betrachtung von "Wirkenden" und zu der Frage, welche Rollen diese aus der Sicht des Unternehmens einnehmen. Das Unternehmen selbst bildet eine organisatorischen Domäne, die ihre eigenen Strategien, Verträge und Regeln weitgehend[2] autonom bestimmt. Für eine solche Domäne kann weiter zwischen Objekten externer Verteilung (z.B. Kunden, Lieferanten, ...) und Objekten interner Verteilung (Mitarbeiter, EDV-Systeme, Geschäftsstellen, ...) unterschieden werden. Innerhalb des Domäne bilden die Wirkenden eine Gemeinschaft; werden Objekte verschiedener Domänen betrachtet, so bilden diese Domänen eine Föderation.

Gegenstand dieser Arbeit ist ein relativ neuer Ansatz der verteilten Programmierung mit *mobilen Agenten*, welcher eine aktivitätenorientierte Modellierung von Informationssystemen aus der Unternehmenssicht ermöglicht und die dabei verwendeten Modellierungskonzepte direkt in ein Programmiermodell umsetzt.

Agenten sind inhärent aktive Softwarekomponenten (im Gegensatz zu verbreiteten Objektmodellen, die zwar häufig aktive Objekte erlauben, nicht aber verlangen) und bilden den Gegenstand der Verteilung im System. Das agentenorientierte Programmiermodell geht aus von einem koordinierten aber unvorhersehbaren Zusammenwirken von getrennt voneinander entwickelten Komponenten in Hinblick auf ein gemeinsames Ziel (siehe auch die *Business Objects* von CORBA [OHE96]).

Mobile Agenten besitzen zusätzlich die Fähigkeit, sich aufgrund interner Entscheidungsprozesse zwischen Rechnern über Netzwerke zu bewegen und mit anderen Agenten und ihrer Umwelt zu interagieren. Sie können somit als lokale Repräsentation entfernter Dienste fungieren oder stellvertretend Aufgaben erledigen, während ein mobiler Nutzer zeitweilig nicht mit dem Zielrechner verbunden ist. Durch ihr dynamisches Zusammenwirken können mobile Agenten Dienste, die einem Benutzer angeboten werden, individuell an seine Interessen und Wünsche anpassen und somit persönliche, kundenbezogene Sichten auf Standarddienste bereitstellen [GMI95b, Way94, Way95, Rei94].

1.2 Zielsetzung und Gliederung der Arbeit

Ziel dieser Arbeit ist es, die Grundlagen für die Erstellung von agentenbasierten, kooperativen Anwendungen zu definieren und eine Systemumgebung prototypisch zu realisieren, die eine Infrastruktur für verteilte agentenbasierte Anwendungen bildet. Der Entwurf der Umgebung und ihrer Komponenten erfolgt dabei mit der objektorientierten Modellierungsmethode *Mainstream Objects* [You95].

Der Einsatz der implementierten Umgebung soll weiter anhand eines praxisrelevanten Beispiels demonstriert werden. Modelliert wird ein einfaches, verteiltes Informationssystem (*MultiSiteAnalyzer*[3]), welches statistische Daten autonomer Einzelsysteme aggregiert und diese Daten ebenfalls als kooperative Komponente anderen Informationsystemen und Endnutzern anbietet.

Bei der Konzeption der Umgebung für mobile Agenten steht der Aspekt der Kooperation autonomer Komponenten und die anwendungsnahe Modellierung agentenbasierter Anwendungen aus der Unternehmenssicht des ODP-Referenzmodells im Vordergrund.

Die prototypische Implementation verwendet die im Rahmen des $FIDE_2$-Projekts am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg entwickelte, integrierte persistente und polymorphe Programmierumgebung *Tycoon*[4].

[2] Zwingende gesetzgeberische Einflüsse seien an dieser Stelle nicht betrachtet.
[3] Die Schreibweise entspricht der Produktbezeichnung.
[4] Tycoon steht für Typed Communicating Objects in Open Environments

Die Arbeit gliedert sich wie folgt. In Kapitel 2 werden die Grundlagen kooperativer verteilter Informationssysteme präsentiert und verschiedene Verfahren zu ihrer Realisierung gezeigt. Kapitel 3 stellt das im Rahmen dieser Arbeit entwickelte, anwendungsnahe Kooperationsmodell der *Business Conversations* vor und beschreibt dessen medienunabhängige Verwendung. Es folgt in Kapitel 4 eine Übersicht über existierende Agentensysteme und die wesentlichen Eigenschaften dieser Architekturen. Kapitel 5 zeigt dann die Konzeption der Umgebung für mobile Agenten anhand von Konzepten und notwendigen Systemkomponenten, unabhängig von der verwendeten Realisierungsplattform Tycoon. Die Kapitel 6 und 7 beschreiben das Beispielszenario und zeigen in Form eines abstrakten Programmiermodells, wie eine entsprechende Anwendung mit den Mitteln der entworfenen Agentenumgebung modelliert werden kann. Kapitel 8 bietet eine kurze Einführung in das Tycoon-System und gibt einen Überblick über die eingesetzte Systeminfrastruktur. Das Kapitel 9 zeigt dann die konkrete Umsetzung der einzelnen Komponenten der Agentenumgebung für das Tycoon-System. Kapitel 10 faßt rückblickend die Kernpunkte der vorliegenden Arbeit aus der Sicht des ODP-Referenzmodells zusammen und gibt einen Ausblick auf weiterführende Arbeiten auf dem Gebiet der agentenbasierten, kooperativen Systeme, für die diese Arbeit die konzeptuellen Grundlagen bereitstellt.

Teil I

Konzeption eines Agentensystems

Kapitel 2

Kooperierende Informationssysteme

Rechnergestützte Informationssysteme haben sich heute in vielen Anwendungsbereichen etabliert. Mittels dieser Systeme ist eine extrem große Menge an unterschiedlichsten Informationen zugänglich, die weltweit von unzähligen Firmen, Institutionen und Privatpersonen aus den verschiedensten Gründen gesammelt, verwaltet und angeboten wird.

Dieses Kapitel gibt eine Übersicht über die Aspekte der Verteilung und über existierende Ansätze zur Verwaltung autonomer, heterogener Informationsbestände.

Im Abschnitt 2.3 wird das Konzept der "offenen Dienstnutzungsumgebung" eingeführt. Die aus diesem Teil motivierten Begriffe *Kooperation* und *Autonomie* werden dann im Abschnitt 2.4 näher betrachtet.

2.1 Verteilte Anwendungen

Eine *verteilte Anwendung* setzt sich aus einer Menge von kommunizierenden Komponenten zusammen, die keinen gemeinsamen Adressraum voraussetzen und auf verschiedenen Rechnern verteilt sein können [Sch90]. Dabei stützt sich die verteilte Anwendung auf ein *verteiltes System*, das auf der Basis einer physikalischen Rechnerkopplung diese Kommunikation ermöglicht (Abb.2.1). Es ist dabei nicht relevant, ob diese Kopplung direkt über ein lokales Netzwerk existiert oder ob sie mittels zwischengeschalteter Transitsysteme verschiedene Netze einbezieht.

Bis Ende der 80er Jahre, mit dem Aufkommen der glasfasergestützten Breitbanddatenübertragung und der dafür notwendigen schnellen Vermittlungstechniken, wurden verteilte Anwendungen für lokale Netzwerke realisiert (Beispiel: Drucker-Server). Dies wurde durch die geringe Zuverlässigkeit und die ebenfalls geringe Übertragungsbandbreite der Orts- und Weitverkehrsnetze erzwungen. Dadurch war der mögliche Nutzen der Verteilung auf die optimierte Auslastung lokal vorhandener Ressourcen wie Prozessor, Haupt- und Sekundärspeicher oder Drucker beschränkt.

Heute sind insbesondere durch das Internet viele lokale Netzwerke in globale Netze integriert. Dabei erreicht die Übertragungsleistung über die Netzgrenzen hinaus oft schon die Leistung, die innerhalb des lokalen Netzes verfügbar ist[1].

Neben den permanenten Netzverbindungen, die in der Regel auch örtlich gebunden sind, gibt es in jüngster Zeit einen Trend, Endbenutzer im privaten Bereich den Zugang zu globalen Netzen zu ermöglichen. Dieser Zugang zu öffentlichen Netzen geschieht mittels der vorhandenen Telekommunikationsinfrastruktur über sogenannte *Provider*. Provider bieten als Dienst ihren permanenten

[1] Es ist zu erwarten, daß sich diese Angleichung von Leistungsmerkmalen auch in der Zukunft fortsetzt[Tsc94].

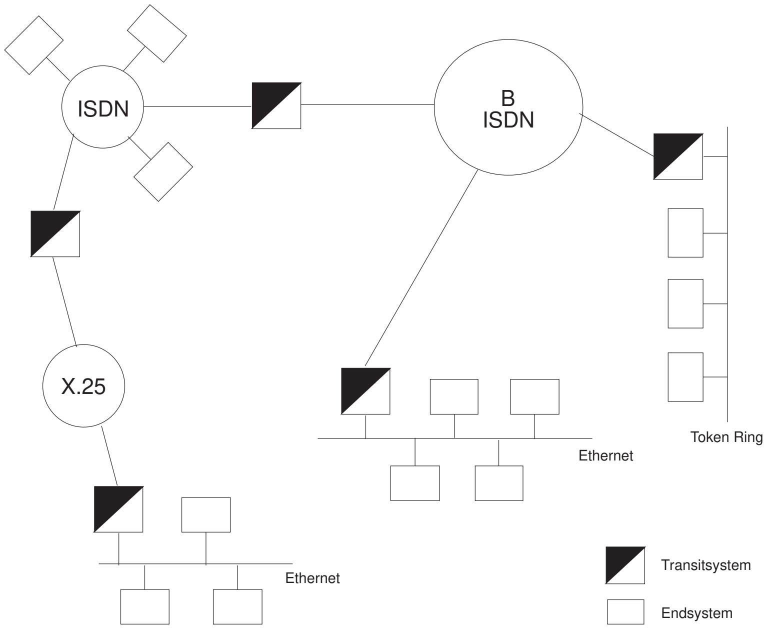

Abbildung 2.1: Kopplung von Rechnern zu einem verteilten System

Netzzugang Endbenutzern zur freien Verwendung an. Die Endbenutzer werden dann zeitweilig (für die Dauer der Verbindung zu ihrem Provider) Teil des globalen Netzes. Die den Endbenutzern zur Verfügung stehende Netzbandbreite ist nur ein Teil der dem Provider zur Verfügung stehenden Bandbreite, dafür jedoch hat eine Vielzahl von Endbenutzern gleichzeitig über einen Provider Zugang zum globalen Netz.

Durch den Ausbau der Telekommunikationsinfrastruktur auf Glasfaserbasis ist es in naher Zukunft für eine breite Anwenderschicht möglich, mittels ISDN eine Übertragungsbandbreite von 64 KBit/s zu nutzen.

Aus diesen Gründen können neben der Optimierung der Lastverteilung (und damit z.B. der Antwortzeiten eines Auftrags) auch völlig neue Arten von Anwendungen realisiert werden, die insbesondere auch für den privaten Endbenutzer interessant sind. In diesem Zusammenhang werden in [Tsc94] eine Reihe typischer Eigenschaften dieser neuen Anwendungsklassen genannt:

- Hoher Durchsatz und erhöhte Zuverlässigkeit. Im Weitverkehrsbereich ermöglichen sie eine unternehmens- und standortübergreifende Kopplung von Rechnernetzen, Großrechnern und Arbeitsplatzrechnern.

- Nutzung von Audio- und Videodaten in Realzeit für multimediale Anwendungen. Beispiele hierfür sind Videokonferenzen und multimediale Dokumente.

- Nutzung von offenen Umgebungen mit zahlreichen Teilnehmern und Ressourcen. Die Anwendungen nutzen die verbundenen Netze und die unterstützende Plattform für einen offenen Markt von Diensten. Informationen, Ressourcen und Know-How werden dynamisch angeboten und genutzt. Beispiele hier sind Rechenleistung, multimediale Informationsbanken, Unterrichtsmaterial, Expertenwissen und alle Formen des Unterhaltungs- und Freizeitmarktes.

- Kooperative Arbeit entfernter Teilnehmer und Gruppen. Die Anwendungen nutzen die Dienstleistungen des Netzes für die koordinierte Arbeit von entfernten Teilnehmern und Komponenten an einem gemeinsamen Gegenstand oder simulieren die gemeinsame Präsenz der Teilnehmer.

Diesen Anwendungsklassen gemeinsam ist eine verteilte Datenhaltung, der eine Vielzahl von heterogenen Plattformen zugrunde liegt und die eine starke räumliche Verteilung der Daten ermöglicht.

2.2 Realisierung verteilter Anwendungen

Bei der Realisierung einer verteilten Anwendung für ein verteiltes System sind eine Reihe von Besonderheiten zu beachten, die gerade aus der Verteilung entstehen.

Es existieren verschiedene Ansätze, Datenbestände zu verteilen. Diese lassen sich anhand des jeweils betrachteten Verteilungsgegenstandes klassifizieren. Die Klassifizierung, die im folgenden Abschnitt verwendet wird, identifiziert dabei: verteilte Daten, verteilte Objekte und verteilte Aktivitäten.

Ein weiterer Freiheitsgrad beim Entwurf und der Realisierung einer Anwendung ist die Art der verteilten Programmierung. Dieser Aspekt wird im Abschnitt 2.2.2 weiter behandelt.

2.2.1 Gegenstände der Verteilung

Im Folgenden werden die unterschiedlichen Ansätze zur Verteilung von Objekten[2] beschrieben und gegenübergestellt.

2.2.1.1 Verteilte Daten

Bei diesem Ansatz werden Nutzdaten auf mehrere unabhängige Datenbanksysteme über verschiedene Rechnerknoten verteilt. Dies entspricht dem klassischen Verteilungsansatz mit verteilten Datenbankmanagementsystemen (DBMS, z.B. das System R* [CP84]).

Der Zugriff auf die verwalteten Daten geschieht über Nachrichten an das DBMS der Art STORE bzw. QUERY. Die Nachrichtenformate für diesen Zugriff sind abhängig vom Datenmodell des angesprochenen DBMS und variieren auch innerhalb eines Datenmodells je nach Hersteller der verwendeten Management-Systeme.

Der konkurrierende Zugriff mehrerer Benutzer auf dieselben Daten erfordert ein Zugriffsprotokoll, welches die Konsistenz des Datenbestandes (unabhängig von der Verteilung) innerhalb der DBMS gewährleistet. Ein weit verbreiteter Transaktionsbegriff, der dieses Kriterium erfüllt, basiert auf dem ACID-Prinzip[3] [GR93].

Eine Fehlererholung für verteilte Transaktionen muß ebenfalls den ACID-Anforderungen genügen. Betrachtet werden dabei *kurze* Transaktionen mit Sperren auf Tupel- oder Dateiebene, die im verteilten Fall mit einem zweiphasigen Bestätigungsprotokoll zwischen den beteiligten DBMS koordiniert werden. Dadurch wird ein konsistentes, gemeinsames Ende einer verteilten Transaktion erreicht.

Für diesen Ansatz ist eine überwiegend homogene DBMS-Struktur notwendig, da der Datenzugriff Wissen über das vom DBMS verwendete Datenmodell voraussetzt und die Datenmanipulationssprachen (d.h. die Nachrichten) je nach DBMS variieren.

Obwohl auch in diesem Fall von Datenbankservern und Klienten (den Applikationen) gesprochen werden kann, handelt es sich bei diesem Ansatz um ein *geschlossenes* System, in das keine anderen Dienste integriert werden können.

---

[2] Objekte hier im sprachlichen Sinne

[3] ACID ::= *A*tomicy, *C*onsistency, *I*solation, *D*urability

2.2.1.2 Verteilte Objekte

Der in letzter Zeit immer stärker in der Praxis verwendete Client/Server-Ansatz geht von beliebigen Diensterbringern (engl. *server*) und Dienstnutzern (engl. *client*) in Rechnernetzen aus. In diesem Modell ist der Gegenstand der Verteilung eine Menge von Objekten, die mittels Daten- und Funktionsabstraktion die *Dienste* bilden. Dienste verbergen alle Details einer zugrundeliegende Implementation vor ihren Dienstnutzern. Somit bietet ein Dienst eine Schnittstelle für den Zugriff auf die Daten und die Funktionalität einer im System vorhandenen Komponente über die von ihr exportierten Operationen.

Diese Vorgehensweise ermöglicht es z.B., ein bestehendes, in Cobol implementiertes Informationssystem in einem Objekt zu kapseln und dieses als Dienst verfügbar zu machen. Danach kann die Implementation des Informationssystems durch eine andere Technologie (beispielsweise basierend auf SQL) ausgetauscht werden, ohne die bereits vorhandenen Dienstnutzer dadurch zu beeinträchtigen. Die Spezifikation der Dienstschnittstelle wird durch den Technologiewechsel nicht berührt.

Die Strukturierung und Verwaltung der Menge unterschiedlicher Dienste geschieht durch wenige, in der Dienstumgebung standardisierte Systemdienste[4]. Dabei handelt es sich z.B. um Vermittlungsdienste, die einem Klienten das Auffinden geeigneter Diensterbringer zu einer Dienstspezifikation ermöglichen (Trader-/Brokerdienste) oder einfache Namensverzeichnisse zur indirekten Adressierung von Diensten. Andere Systemdienste gewährleisten die Interoperabilität innerhalb einer heterogenen Systemlandschaft, indem sie z.B. Datenformate konvertieren, und bieten eine skalierende Kommunikationsinfrastruktur an.

Die Kommunikation mit den Dienstobjekten geschieht über den Methodenaufruf, d.h. über einen Nachrichtenaustausch, der direkt zur Exekution einer in das Netz exportierten Operation des adressierten Dienstobjektes führt. Der Dienst seinerseits verbirgt alle inneren Strukturdetails vor dessen Nutzer.

Fehlererholungsmechanismen und -strategien sind spezifisch zum jeweilig angesprochenen Dienst. Allerdings bietet die Middleware in der Regel allgemeine Mechanismen zur Fehlererholung an, die von dem Dienst verwendet werden können.

Ein Beispiel für ein verteiltes Objektsystem ist die Systemarchitektur CORBA der OMG[Gro91]. Eine detaillierte Beschreibung der in diesem System verwendeten Komponenten findet sich z.B. in [Ric97].

2.2.1.3 Verteilte Aktivitäten

Gegenstand der Verteilung bei diesem Ansatz sind aktive und autonome Objekte, die *Agenten* [5]. Sie bilden ein anwendungsnahes Verteilungskonzept ähnlich den *Business Objects* [OHE96]. In dieser Hinsicht stellen sie in sich abgeschlossene und separat entwickelte Anwendungskomponenten dar, die sich in eine bestehende Gesamtanwendung aus vielen solcher Einzelkomponenten integrieren. Dabei interagieren sie mit anderen Agenten ihrer Umgebung, indem sie sowohl Dienste für andere Agenten anbieten als auch Dienste als Klient nutzen.

Agenten kapseln Daten, Funktionen und eine Historie über ihrem Zustand. Sie sind insofern autonom, als daß sie einen eigenen Kontrollfluß besitzen und daraus ihr Verhalten gegenüber anderen Komponenten selbst bestimmen. Aus deren Sicht erscheint daher das Verhalten des Agenten willkürlich und unvorhersagbar. Agenten gelten in diesem Sinne als *pro-aktiv*, im Gegensatz zu den vorher genannten *re-aktiven* Objekten [AGP94].

Mobile Agenten sind darüber hinaus in der Lage, sich aufgrund interner Entscheidungen innerhalb des Netzes von einem Adressraum in einen andern zu bewegen und dort in einem neuen

[4] Engl. *middleware*.
[5] Der Begriff *Agent* wird im Abschnitt 4.1 in seinen verschiedenen Bedeutungen näher betrachtet.

Kontext zu wirken. Das führt zu einer dynamischen Topologie der Objekte des Netzes und der Kommunikationsbeziehungen, die zwischen diesen Objekten bestehen.

Insbesondere gibt es keine Möglichkeit, von außen den jeweiligen internen Zustand eines Agenten zu bestimmen oder aus einer zentralen Sicht heraus Aussagen über das Gesamtsystem aller Agenten zu machen, da die für diesen Zweck notwendige äußere Kontrolle fehlt.

Das Konzept der Verteilung basierend auf mobilen Agenten eignet sich aufgrund seiner Anwendungsnähe zur Modellierung langandauernder Aktivitäten, wie sie häufig in der realen Welt vorkommen.

Dabei ist eine auf dem ACID-Prinzip basierende Abarbeitung der Gesamtaktivität auf der Grundlage von Transaktionen, wie dies z.B. im Fall der verteilten Daten mit verteilten DBM-Systemen geschieht, nicht sinnvoll. Ein solches Vorgehen hätte ein Aufrechterhalten aller Sperren auf verwendete Objekte einer Transaktion über deren gesamten Ablaufzeitraum zur Folge. Konkurrierende Transaktionen müßten dann, aufgrund der im ACID-Prinzip geforderten Konsistenz- und Isolationsbedingung, auf die Beendigung der sperrenden Transaktion warten oder in Deadlock-Situationen komplett rückgängig gemacht werden. Bei Transaktionen, die viele Objekte verschiedener Dienste über lange Zeiträume benötigen, führt ein solches Transaktionsmodell zu einer starken Verminderung des Transaktionsdurchsatzes im System.

Dieser Effekt kann vermieden werden, wenn Sperren nicht bis zum Ende der Transaktion gehalten werden müssen und so die Wahrscheinlichkeit für Sperrkonflikte zwischen Transaktionen verkleinert wird. Die einzelnen Schritte einer solchen Transaktion können dabei nach dem ACID-Prinzip durchgeführt werden und damit die Konsistenz der verwendeten Objekte sicherstellen. Durch das frühe Freigeben von Sperren und der damit verbundenen Aufweichung der Atomarität der Gesamttransaktion werden jedoch zwischenzeitlich inkonsistente Zustände für andere Transaktionen sichtbar.

Zusätzlich muß für jeden Zustandsübergang der langandauernden Transaktion eine kompensierende Aktion angegeben werden, die die *Effekte* der ersten gegebenenfalls kompensiert[GMS87]. Dadurch können die Auswirkungen der einzelnen Schritte wieder rückgängig gemacht werden. Diese kompensierenden Aktionen werden auch *inverse* Aktionen genannt.

2.2.2 Ansätze zur verteilten Programmierung

Verteilte Programmierung ist die programmiersprachliche und systemtechnische Grundlage zur Erstellung von verteilten Anwendungen. Unabhängig davon, welches der im vorherigen Abschnitt dargestellten Verteilungsmuster verwendet wird, d.h. unabhängig davon, ob nun Daten, Objekte oder Aktivitäten den Gegenstand der Verteilung bilden, müssen die zum Design einer verteilten Anwendung verwendeten Hilfsmittel die verteilte Programmierung unterstützen.

Diese Unterstützung kann verschiedene Ausprägungen haben, die im folgenden vorgestellt werden.

- ▹ Betriebssystemansatz: Netzbetriebssysteme ergänzen Rechner mit lokalen Betriebssystemen um Komponenten zur verteilten Kommunikation. Sie ermöglichen den geteilten Zugriff auf Ressourcen eines u.U. heterogenen Rechnerverbundes, wie Plattenspeicher oder Rechenzeit. Die einzelnen Rechner des Verbundes bleiben als solches sichtbar. Bekannte Netzbetriebssysteme sind AMOEBA und MACH [Tan95].

- ▹ Direkte Programmierung: Bei diesem Ansatz werden dem Programmierer innerhalb einer gewöhnlichen Programmiersprache (z.B. C, C++, Cobol, ...) über Funktionsbibliotheken Betriebssystemoperationen für die verteilte Programmierung zur Verfügung gestellt. Im einfachsten Fall handelt es sich dabei um Funktionen zur Übermittlung von untypisierten Datenströmen, die in einer definierten Qualität an einen fest zu wählenden Kommunikationspartner übermittelt werden.[6] Die Schnittstellen und Operationssemantiken variieren je nach verwendetem Betriebssystem, und Interoperabilitätsaspekte müssen bei Bedarf vom Programmierer selbst berücksichtigt werden.

- Nachrichtenaustausch: Der Nachrichtenaustausch innerhalb eines programmiersprachlichen Rahmens folgt dem gleichen Prinzip wie der Nachrichtenaustausch auf der Ebene der Systemprozesse. Der wesentliche Unterschied ist, daß die Nachrichten innerhalb der Programmiersprache *typisiert* sind. Jede Anwendung kann ihre eigenen, nur innerhalb ihres Anwendungskontextes gültigen Nachrichtenarten definieren.

- Entfernter Prozeduraufruf: Der entfernte Prozeduraufruf ist eine Erweiterung des lokalen Prozeduraufrufes, wie er von herkömmlichen Programmiersprachen der 3. und 4. Generation angeboten wird. Eine Menge semantisch zusammengehöriger, entfernter Prozeduren bildet einen Dienst.

  Dabei wird die Aufrufsemantik der lokalen Prozeduren dahingehend erweitert, daß sie auch innerhalb eines anderen Adressraumes ausgeführt werden dürfen. Es wird in diesem Fall lokal (beim Klienten) lediglich die Signatur der Prozedur deklariert, ohne daß eine Implementation vorhanden ist.

  Die tatsächliche Implementation der Prozedur wird zur Laufzeit, vor ihrem ersten Aufruf, ermittelt. Dies geschieht in einer sog. Bindephase, in deren Verlauf ein geeigneter Diensterbringer für die Prozedur festgelegt wird.

  Die lokale Ausführung der Prozedur wird von einem generischen Kodefragment (einem sog. *Stub*) transparent durch eine definierte Folge von Nachrichten an das gebundene Diensterbringersystem ersetzt. Die aktuellen Argumente, die zum Zeitpunkt des Prozeduraufrufs bekannt sind, werden dabei in einer systemunabhängigen Darstellung an den entfernten Partner übermittelt.

  Der Diensterbringer seinerseits besitzt eine Implementation des Prozedurrumpfes, so daß dieser die Prozedur mit den übermittelten Aktualparametern des Klienten ausführen kann. Das Ergebnis der Ausführung wird dem Klienten zurückübermittelt. Der entfernte Prozeduraufruf ist in der Regel synchron, wenn die Prozedur ein Ergebnis liefern soll. Die Abwicklung des Nachrichtenprotokolls und die Wahl der geeigneten Darstellung der Argumente bzw. des Ergebnisses ist ein Systemdienst, der für die Anwendung im Idealfall nicht sichtbar ist [Joh95].

- Entfernte Objektaktivierung. Der objektorientierte Ansatz mit den Grundprinzipien der *Datenabstraktion*, der *Vererbung* und des *dynamischen Bindens* führt zur Strukturierung einer Anwendung in der Form von kommunizierenden Objekten. Die Interaktion zwischen Objekten und damit der Zugriff auf den vom Objekt gekapselten Zustand erfolgt ausschließlich über *Nachrichten*. Diese führen dann zum Aufruf von *Methoden* in dem Empfängerobjekt. Der wesentliche Unterschied zum o.g. Nachrichtenaustausch ist, daß jede Nachricht direkt einer Methode zugeordnet ist und damit immer eine Aufforderung zur Ausführung einer Operation darstellt. Im Vergleich zum entfernten Prozeduraufruf wird i.d.R. eine erweiterte Semantik in Bezug auf Objektreferenzen und Objektmobilität angeboten.

Im Gegensatz zu den "Add On"-Ansätzen, die versuchen kommerziell relevante Programmiersprachen um Aspekte der verteilten Programmierung zu erweitern, gibt es eine Reihe von Sprachen, welche Verteilung explizit berücksichtigen. Zu diesen Spachen gehören Emerald[Juh88], Obliq[Car94] und Phantom[Cou95]. Diese Sprachen nutzen die entfernte Objektaktivierung und bieten spezielle Sprachkonstrukte, um Teilaspekte der verteilten Programmierung gesondert zu unterstützen. Allerdings verwenden alle diese Sprachen das Konzept der *Netzwerkreferenzen* auf Objekte anderer Domänen. Ein solches Verfahren zur verteilten Programmierung ist aus Sicht

[6] Im weit verbreiteten Betriebssystem Unix werden sog. *Sockets* verwendet.

der Autonomie der Domänen nicht vertretbar und führt darüber hinaus zu einer unkontrollierbaren Ausdehnung von Objekten. Netzwerkreferenzen sind daher i.d.R. nur in lokalen Netzwerken sinnvoll einsetzbar [Mat96].

2.3 Offene Dienstnutzungsumgebungen

Die im vorigen Abschnitt behandelten Ansätze zur Programmierung verteilter Anwendungen bilden die Grundlage für die Realisierung von verteilten Systemen.

Über die genannten Mechanismen sind System oder Systemkomponenten in der Lage, mit anderen Systemen bzw. Systemkomponenten zu interagieren. Dies gilt insbesondere dann, wenn sich die Interaktionspartner in voneinander getrennten Adressräumen befinden.

Verteilte Anwendungen können mit den genannten Mitteln erstellt werden. Um allerdings die Investitionen, die ein Unternehmen in die Konzeption, Erstellung und Pflege eines verteilten Informationssystems getätigt hat, über einen langen Zeitraum zu sichern, ist es notwendig, neben den technischen Voraussetzungen zur Kommunikation auch eine geeignete Infrastruktur für diese Systeme zu berücksichtigen.

Informationssysteme unterliegen über die gesamte Dauer ihres Einsatzes sowohl einem technologischen Wandel, als auch organisatorisch bedingten Veränderungen. Um diesen stetigen Wandel zu begegnen, ist es sinnvoll, große Systeme in einzelne, voneinander getrennte Komponenten zu untergliedern, die über flexible Mechanismen interagieren können. Die auf diese Weise geschaffenen Informationssysteme sind sog. "Offene Systeme".

Der Term "Offene Systeme (engl.: *open systems*)" bezeichnet Systeme, welche durch bestimmte Maßnahmen zu Beziehungen mit der Außenwelt (d.h. mit anderen, ebenfalls offenen Systemen) befähigt sind und dabei die nachfolgend angeführten Qualitätsmerkmale besitzen. Betrachtet man in der Erweiterung des Konzeptes statt einzelner Systeme einen freien Verbund, so führ das zu der Vorstellung der "Offenen Dienstnutzungsumgebung". Hier steht der Begriff "offen" als Synonym für "frei zugänglich", "unbegrenzt", "heterogen", "selbständig und unabhängig" und "dezentralisiert" [Tsc94].

Die genannten Merkmale einer offenen Dienstnutzungsumgebung stellen sich im Einzelnen wie folgt dar:

- ▹ Unbegrenzt: Generell ist die offene Dienstnutzungsumgebung die unbegrenzt. Es bestehen prinzipiell keine Beschränkungen in semantischer, geographischer, organisatorischer oder technischer Hinsicht.

- ▹ Frei zugänglich: Eine offene Dienstnutzungsumgebung läßt prinzipiell jede Art von Benutzern, Anwendungen und Komponenten zu. Es wird grundsätzlich niemand von der Nutzung vorhandener Ressourcen oder Dienste ausgeschlossen.

- ▹ Heterogen: Jede Organisation innerhalb der Umgebung kann über die ihr zugehörigen Komponenten frei verfügen. Dadurch entsteht eine Ansammlung von Komponenten und Anwendungen, die sich den von der jeweiligen Organisation vorgegebenen Anforderungen kontinuierlich anpaßt.

- ▹ Selbständig und unabhängig: Jede Komponente ist an ihrem Betriebsort charakteristischen Bedingungen unterworfen. Die lokale Umgebung bestimmt ihr Verhalten und ihre Fortentwicklung. Den anderen Komponenten gegenüber äußert sich dies in einem unabhängigen, selbstbestimmten Verhalten. Jede Komponente ist aus der Sicht anderer Komponenten *autonom*.

- ▷ Dezentralisiert: In der offenen Umgebung entwickeln sich die Komponenten und ihre Eigenschaften und Verhaltensweisen nach lokalen Gesichtspunkten und ohne eine globale Kontrolle. Daher gibt es auch keinen global beobachtbaren Zustand und keine Institution, die Aussagen über einen momentanen Zustand aller Komponenten erlaubt.

Offene Dienstnutzungsumgebungen besitzen aufgrund ihrer potentiell unbegrenzten Ausdehnung einen inhärent föderativen Charakter. Sie sind eine Erweiterung des bereits eingangs geschilderten ODP-Modells, indem sie, für sich genommen, keine eigene ODP-Anwendung darstellen, sondern vielmehr das Konzept eines freien und globalen Marktes autonomer und miteinander kooperierender Dienste begründen. Die Basis solcher Dienste, sowohl im Hinblick auf die Modellierung als auch auf die verwendeten Realisierungsprinzipien, wird durch die ODP-Grundlagen gebildet.

Innerhalb von Dienstmärkten schließen sich Komponenten zum Zwecke des Erreichens eines gemeinsamen Ziels dynamische zusammen. Das ODP-Referenzmodell definiert hierfür sog. Tradingdienste [PSW96], die mit Informationen über Dienstleistungen anderer Anbieter handeln. Komponenten können so ihre Dienstleistungen an Traderkomponenten exportieren, wenn sie auf dem Dienstemarkt eine Leistung anbieten wollen. Andererseits können sie aber auch Dienstleister ermitteln (tatsächlich werden i.d.R. Dienstleistungen ermittelt; der eigentliche Dienstleister bleibt verborgen) und dessen angebotene Leistung nutzen.

Es ist daher nicht vorhersehbar, welche Komponenten zu einem Zeitpunkt mit welchen anderen Komponenten zusammenarbeiten werden. Aus diesem Grund sind Systemmechanismen notwendig, welche flexible und verbindliche Kooperationen zwischen beliebigen Komponenten ermöglichen.

2.4 Kooperation und Autonomie

In diesem Abschnitt werden die Begriffe Kooperation und Autonomie genauer betrachtet. Die Darstellung folgt in wesentlichen Teilen [Kir94],[Tsc94] und [DDJ+97].

Als autonom bezeichnete Einheiten haben das Recht auf Selbstbestimmung und Unabhängigkeit. Autonome Teile eines Systems handeln nach individuellen oder lokalen Gesetzen und können die allgemein vorherrschenden Regeln und Verhaltensweisen entweder zeitweise oder auf Dauer mißachten. Autonome Komponenten erlauben nicht den Einblick in ihre Interna. Ihr Zustand kann daher nur aus der Beobachtung ihrer Interaktion mit ihrer Umgebung geschlossen werden.

Die Betrachtung der Ursachen für Autonomie ergibt eine Unterscheidung zwischen organisatorischen und operationellen Ursachen.

Organisatorische Ursachen erwachsen aus dem Umstand, daß Komponenten *Eigentum* bestimmter Organisationen sind.

Daraus folgt, daß jede Organisation

- ▷ das generelle Umfeld und die Einsatzziele für ihre Informationssysteme nach eigenen Bedürfnissen bestimmt,
- ▷ ihre Ziele unabhängig von anderen Organisationen plant und realisiert, auch wenn diese dasselbe generelle Umfeld und Einsatzziel haben,
- ▷ ihre Anwendungen so plant und implementiert, daß lokale Anforderungen an Leistung, Sicherheit, usw. erfüllt werden,
- ▷ Hardware und Software-Komponenten entwickelt und installiert, die ihren Zielen und Anforderungen möglichst optimal entsprechen,
- ▷ ihre Komponenten ständig den sich ändernden Anforderungen und technischen Gegebenheiten anpaßt.

Die aus diesen Punkten erkennbaren unterschiedlichen Formen von Autonomie werden als *Design-Autonomie*, *Evolutions-Autonomie* und *Administrations-Autonomie* bezeichnet.

Der gemeinsame Aspekt aller organisatorisch begründeten Erscheinungsformen der Autonomie ist die Tatsache, daß die den Komponenten inhärenten Verhaltensregeln von den lokalen Organisationen bestimmt oder beeinflußt werden. Das bedeutet, daß diese verhandelbar sind, wenn es darum geht, zwischen Organisationen *Kooperationen* einzurichten und ihre Einheiten in gemeinsame Anwendungen einzubringen.

Neben den genannten organisatorischen Ursachen gibt es noch operationelle Ursachen für Autonomie. Das Verhalten lokaler Benutzer, die Störeinflüsse der Umgebung und der Betrieb insgesamt beeinflussen die Autonomie von Komponenten.

Dabei werden das Last- und Fehlerverhalten der Komponenten aus der Sicht externer Beobachter oder Auftraggeber betrachtet. Verschiedene Komponenten können auf Einflüsse ihrer Umgebung unterschiedlich reagieren. Die Reaktion auf auftretende Last (in Form von Bedien- und Wartezeiten bei Aufträgen für eine Komponente) oder das störungs- und fehlerbedingte Verhalten der Komponenten kann nicht von außen, d.h. von der Umgebung, beeinflußt werden, sondern liegt in der Verantwortung der Komponente selbst.

Die aus dieser Sicht erkennbaren Formen der Autonomie werden als *Zustands-Autonomie*, bzw. im Falle der Störeinflüsse, als *Fehler-Autonomie* bezeichnet. Im Gegensatz zu den organisatorischen Ursachen für Autonomie können die operationell begründeten Erscheinungsformen von Autonomie von den lokalen Organisationen nur indirekt bestimmt oder beeinflußt werden.

Autonome Komponenten eines Systems werden nachfolgend mit dem Begriff *Agenten* bezeichnet. Generell ist es dabei nicht relevant, ob es sich um eine Software-Aktivität oder um einen menschlichen Akteur handelt[7].

Agenten verfolgen in der Regel Ziele[8], d.h. sie haben einen in einem bestimmten Kontext definierten Zweck oder eine Aufgabe. Es ist daher naheliegend, spezialisierte Agenten zur Erfüllung ihrer Aufgaben oder einer übergeordneten Gesamtaufgabe zusammenarbeiten zu lassen. Ein solcher permanenter oder zeitweiliger Zusammenschluß von Einheiten zum Zweck koordinierter, zielgerichteter Aktivitäten heißt *Kooperation* [Tsc94].

Kernpunkt jeder Kooperation ist die gegenseitige Beeinflussung von Handlungen oder Handlungsplänen unter kooperierenden Agenten. Dabei gelten eine Reihe von Bedingungen für Kooperation:

- ▷ Zielidentität
- ▷ Veränderbarkeit der Pläne
- ▷ Ressourcenaustausch
- ▷ Regelbarkeit
- ▷ Kontrolle

Der wichtigste Punkt ist die *Zielidentität*. Damit wird gefordert, daß Agenten übereinstimmende Ziele oder Teilziele aufweisen. Die Handlungspläne der Agenten sollten insofern veränderbar sein, als daß sie sich durch die Interaktion mit anderen Agenten neuen Erfordernissen anpassen. Mit Ressourcenaustausch wird gefordert, daß neben dem Austausch von Kooperationsinhalten auch Randbedingungen wie Ort, Zeit, eingesetzte Mittel und Materialien zwischen den Agenten weitergegeben werden können. Die Fähigkeit zur flexiblen Planbeeinflussung und Behandlung von

[7] Auf die unterschiedlichen Agentenbegriffe wird in Abschnitt 4.1 näher eingegangen.

[8] Natürlich ist es möglich, Agenten zu erschaffen, die *kein* Ziel verfolgen. Aus der Sicht kooperativer Informationssysteme werden diese Agenten allerdings nicht betrachtet.

Ausnahmesituationen heißt Regelbarkeit. Die Kontrolle verlangt schließlich, daß die Handlungsabsichten eines Agenten, d.h. die Handlungspläne, im voraus erkennbar sein müssen, so daß eine maximale Transparenz des kooperativen Prozesses gewährleistet ist.

Kooperative Informationssysteme als Ganzes müssen durch die genannten Formen der Autonomie Änderungen über lange Zeiträume hinweg tolerieren und unterstützen.

Eine Architektur für derartige kooperative Informationssysteme untergliedert das System in vier Schichten[DDJ+97]:

Die Systemschicht bildet die Basis eines kooperativen Informationssystems. In dieser Ebene befinden sich die spezifischen, konventionellen Anwendungen bzw. Komponenten eines Unternehmens. Diese sog. Altsysteme (engl. *legacy systems*) stellen aus Unternehmenssicht die Kernfunktionalität des Informationssystems bereit und sind für sich genommen nicht kooperativ.

Darüber sorgt eine Systemintegrationsschicht für die Zusammenarbeit verschiedener, u.U. verteilter Systeme. Diese Schicht integriert die Komponenten der Systemschicht und bietet den höheren Schichten der Architektur eine einheitliche Sicht auf diese. Hier sind die einheitlichen Kooperationsmechanismen definiert, die die Interaktion mit anderen Systemen ermöglichen.

Die Arbeitsgruppenschicht und die Unternehmensschicht bilden organisatorische Aspekte des Unternehmens auf das Informationssystem ab. Dabei ist es die Aufgabe der Arbeitsgruppenschicht, die soziologische Struktur eines Unternehmens (Teams, Abteilungen, Profit Center, ...) abzubilden. In der Unternehmensschicht wird ein Unternehmen als Ganzes betrachtet. Dabei treten das Unternehmensmodell bzw. die Ziele und Strategien eines Unternehmens in den Vordergrund.

Im folgenden werden nur die Aufgaben der Systemintegrationsschicht erläutert, da diese die kooperativen Eigenschaften eines Informationssystems maßgeblich beeinflußt.

Die Systemintegrationsschicht soll den Anwendungsprogrammierer durch Abstraktion in den folgenden Punkten unterstützen:

1. Kooperation über die Zeit: Die Lebensdauer aller maschinellen Akteure richtet sich nach der Dauer der Geschäftsprozesse, in denen sie involviert sind. Die Lebensdauer ist allein durch den Geschäftsprozeß bestimmt und daher unabhängig von Parametern des verwendeten Systems, wie der Programmiersprache, der Lebensdauer von Betriebssystemprozessen oder von Änderungen der zugrundeliegenden Datenbankschemata.

2. Kooperation innerhalb des Raums: Maschinelle Akteure können innerhalb einer physikalisch oder logisch verteilten Umgebung migrieren. Dabei sind sie nicht von speziellen Systemplattformen oder Organisationsstrukturen abhängig.

3. Kooperation in mehreren Arbeitsformen: Softwaresysteme sollen für maschinelle Akteure einheitlich benutzbar sein. Dabei werden die folgenden Arbeitsformen gleichbehandelt: einfache Stapelverarbeitung, Transaktionsverarbeitung, direkte Manipulation durch menschliche Akteure über formularbasierte Eingabemasken, rechnerunterstützte Zusammenarbeit menschlicher Akteure, rechnergestütztes Workflow-Management und automatische Informationsverarbeitung mit mobilen Softwareagenten. Ebenfalls sollen verschiedene Medien auf gleiche Art unterstützt werden: E-Mail-Nachrichten, EDI-Nachrichten, HTTP-Requests, CORBA Object-Requests oder entfernte Prozeduraufrufe.

Durch die Abstraktion in den Modalitäten Zeit, Raum und Arbeitsform kann sich der Programmierer auf das *was* und *wie* des speziellen Problems konzentrieren und braucht sich nicht mehr um das *wann* (d.h. Stapelverarbeitung, interaktive Sitzungen oder langandauernde Konversationen mit einem Kunden), *wo* (d.h. zentralisierter Hostrechner, Server innerhalb eines Unternehmens oder ein Laptop eines Verkaufsmitarbeiters) und *wer* (d.h. Kunde, Verkäufer, Workflow-Management-System oder Internet-Agent) zu kümmern. Die Applikationen werden so von technischen Details wie dem Datentransfer, der Synchronisation und Fragen der Benutzeroberfläche oder der Dialogsteuerung, befreit.

Es finden sich in der Literatur eine Reihe unterschiedlicher Kooperationsmodelle (z.B. netzbasierte Modellierung[BBO+86], Nachrichtenmuster im Actor-System [AH87] oder selbstleitende Nachrichten [HG84]), von denen im nächsten Abschnitt ein anwendungsnahes Kooperationsmodell, der Language/Action-Ansatz, vorgestellt wird.

2.5 Ein Leitbild für Kooperation: Der *Language/Action* Ansatz

Dieses Kooperationsleitbild betrachtet die Sprache als wichtigste Dimension für kooperative Zusammenarbeit zwischen Menschen. Die Grundlage des Leitbildes ist die Sprechakttheorie, die von Austin [Aus62] eingeführt und von Searle [Sea69] erweitert wurde.

Der Kern der Spechakttheorie besteht darin, daß sich alle sprachlichen Äußerungen (die sog. Sprechakte) hinsichtlich ihrer Verwendung in fünf disjunkte Typklassen einteilen lassen: assertive Äußerungen (Feststellungen), direktive Äußerungen (Aufforderungen), kommissive Äußerungen (Versprechen), expressive Äußerungen (Ausdruck von Meinungen) und deklarative Äußerungen (Erklärungen).

Nach Searle ist eine Äußerung dabei unmittelbar an ihre Bedeutung gekoppelt. Sprechakte werden daher als Mitteilung einer legalen Zustandsänderungen ihres Sprechers innerhalb einer Konversation betrachtet.

Arbeiten aus dem Bereich der computerunterstützten Gruppenarbeit (CSCW) von Fernando Flores und Terry Winograd [Win87, FGHW88] identifizieren zwei hauptsächliche Arten von Sprechakten, die in der Domäne von Geschäftsprozessen eine Rolle spielen: Das sind zum einen die *Conversations for Possibilities* (CfP) und zum anderen die *Conversations for Action* (CfA). Eine CfA ist eine Folge von Aufforderungen und Versprechen, die direkt auf eine gemeinsame Aktivität der Gesprächspartner gerichtet ist. Dabei ist innerhalb einer CfA die Reihenfolge der einzelnen Sprechakte, d.h. die erlaubten Gesprächstransitionen, unter den Gesprächspartnern festgelegt.

Werden in einer Konversation keine Verpflichtungen eingegangen, sondern wird die "Funktion eines Managers" wahrgenommen, sprechen Winograd und Flores vom Typ einer CfP. In diesem Fall werden in einer Konversation nur Feststellungen, Meinungen oder Erklärungen ausgetauscht. Bei keinem Partner wird durch einen Sprechakt eine Aktion impliziert.

Eine CfA oder CfP strukturiert und koordiniert einzelne Sprechakte. Durch diese Sprechakte bilden die beteiligten Kommunikationspartner eine gemeinsame Historie, vor deren Hintergrund neue Sprechakte erzeugt und interpretiert werden. Bei einer CfA, deren Konzept dem Kooperationsmodell aus Kapitel 3 zugrunde liegt, vereinbaren die Partner wechselseitig zukünftige Aktionen, die direkt an die geäußerten Sprechakte der jeweiligen Partner gekoppelt sind.

Kapitel 3

Das Modell der *Business Conversations*

In Kapitel 2 sind die allgemeinen Grundlagen für kooperative Informationssysteme beschrieben worden. Dieses Kapitel stellt das Kooperationsmodell vor, welches in den folgenden Kapiteln die Grundlage der Kooperation mobiler Agenten bildet. Dabei wird anhand eines Beispiels verdeutlicht, wie insbesondere dynamische Aspekte der Kooperation durch das verwendete Modell erfaßt werden.

3.1 Grundlagen des Modells

Das hier beschriebene Kooperationsmodell folgt dem Kooperationsleitbild, das bereits in Abschnitt 2.5 dargestellt wurde.

Das Modell der *Business Conversations* bildet einen Rahmen für die anwendungsnahe Modellierung von Informationssystemen auf der Grundlage von Sprechakten. Es definiert eine Terminologie und eine Semantik, mit der Informationssysteme aus der Sicht von Sprechakten modelliert und spezifiziert werden können. Dabei orientiert es sich begrifflich an der Domäne der Geschäftsprozesse und erfaßt neben den statischen Aspekten eines Dienstes gleichermaßen die dynamische Interaktion von Kunde und Dienstleister über die Zeit.

Basis des Modells ist die Annahme, daß die Kommunikationsmuster der Sprechakte universell sind, d.h. sie sind von Art der Akteure (Software, Menschen) und von den verwendeten Kommunikationsmedien und -protokollen unabhängig.

Sowohl die Interaktion zwischen menschlichen Aktueren, als auch die Interaktion zwischen Menschen und maschinellen Akteuren oder gar die Interaktion zwischen zwei maschinellen Akteuren ist innerhalb des Modells transparent.

Das Modell der *Business Conversations* beschreibt die zielgerichtete Kommunikation zwischen zwei Akteuren, die in den Rollen Kunde und Dienstleister handeln.

Dabei basiert das Modell auf der Theorie der *Conversations for Action (CfA)*[1], daß eine Kommunikation mit dem Ziel einer Aktion zwischen einem Kunden und einem Dienstleister in einen Zyklus mit vier Phasen unterteilt werden kann. Diese Phasen sind in der Abbildung 3.1 zu sehen.

Die Beziehung zwischen einem Kunden und einem Dienstleister folgt danach immer dem dort skizzierten Ablauf: Im ersten Schritt wird der Kunde beim Dienstleister unverbindlich nach einer Dienstleistung anfragen. Am Ende dieses Schrittes steht dann, in der *Übereinkunfts*-Phase, die

[1] Siehe Abschnitt 2.5

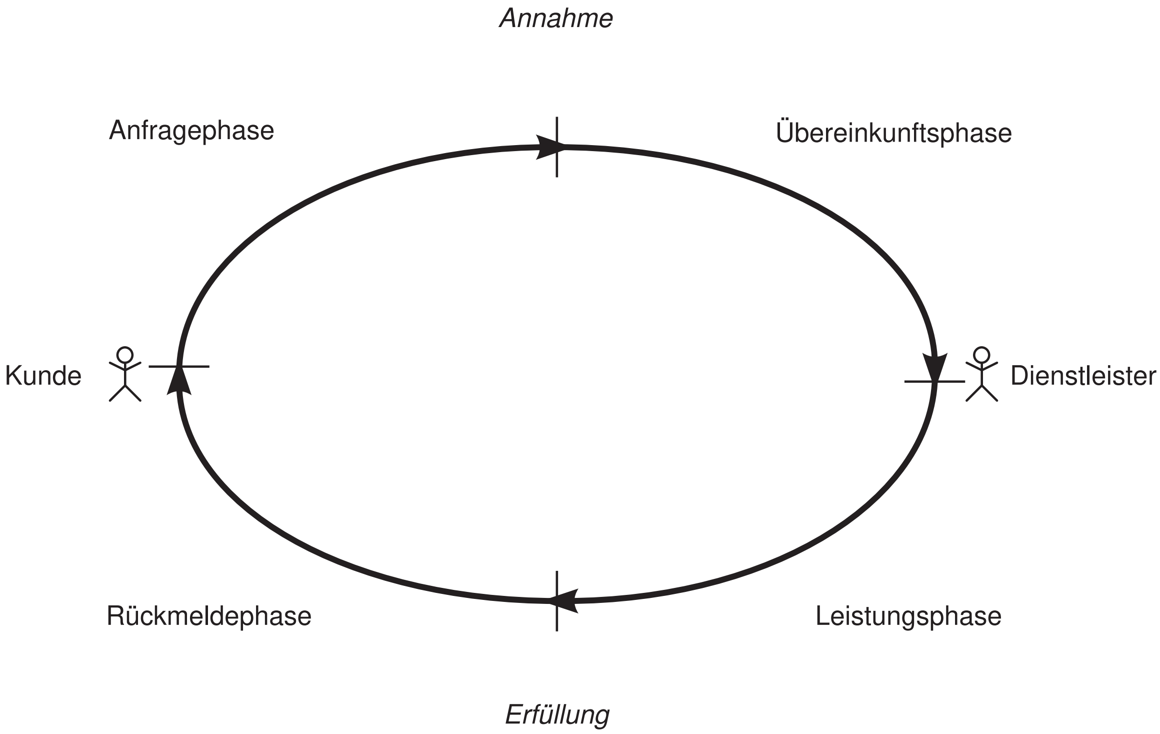

Abbildung 3.1: Kommunikation nach der Language/Action Perspektive

Verhandlung mit dem Dienstleister, einen gewählten Dienst tatsächlich verbindlich zu erbringen. Das Erbringen der vereinbarten Leistung geschieht in der dritten Phase, an deren Ende aus der Sicht des Dienstleisters die geforderte Leistung erbracht ist. In der vierten Phase, der *Rückmeldung*, wird der Kunde dem Dienstleister mitteilen, ob die erbrachte Leistung seinen Anforderungen entspricht.

Nur wenn auch diese Phase erfolgreich abgeschlossen ist, ist das gemeinsame Ziel von Kunde und Dienstleister, und damit das Ziel ihrer Kommunikation, erreicht. Der Austausch von Sprechakten innerhalb dieser vier Phasen wird unter dem Begriff der *Konversation* zusammengefaßt, der sich damit immer auf einen konkreten Gegenstand, die Dienstleistung oder Aktion, bezieht.

Der Zyklus kann durch den Verzicht auf einzelne Phasen vereinfacht werden, wenn die betrachtete Kunde-Dienstleister-Beziehung dies erlaubt. Zum Beispiel können die Phasen *Anfrage* und *Übereinkunft* zusammengefaßt werden. Dabei bleibt jedoch sowohl die Reihenfolge der Phasen als auch der Zyklus erhalten.

3.1.1 Einordnung

Das Modell der *Business Conversations* ergänzt bereits etablierte Modelle (wie z.B. Mainstream Objects, Entity-Relationship-Modell), welche die Modellierung von Informationssystemen nach *innen* gestatten (siehe Abb. 3.2). Diese Modelle können aus der Sicht einer Anwendung als systemnahe Modelle bezeichnet werden. Sie strukturieren Systemkomponenten *bottom-up* und daten- bzw. objektzentriert, d.h. auf Objekt oder Entitätsebene.

Die Business Conversations hingegen stellen den kooperativen Charakter eines Informationssystems in den Vordergrund. Durch die Betrachtung der von dem System nach *außen* erbrachten Dienstleistungsprozesse wird eine *anwendungs*nahe Modellierung des Systems unterstützt. Hierdurch wird der Abstand zwischen der realen Welt und ihrer Abbildung durch Informationssysteme verkleinert. Das Modell versucht nicht, Informationssysteme auf der Ebene von Objekten

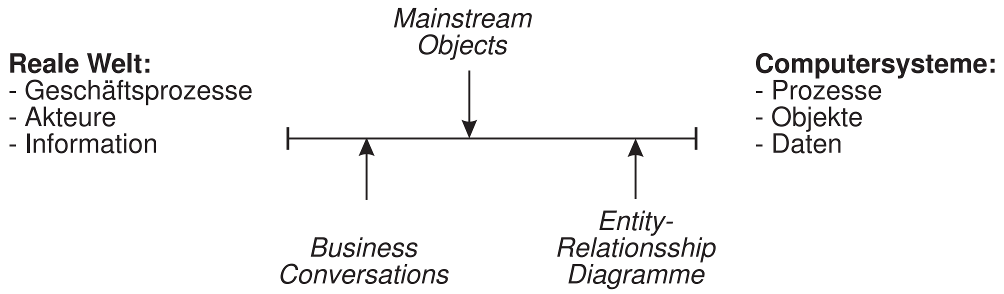

Abbildung 3.2: Modellbildung für Informationssysteme

oder Klassen zu beschreiben. Vererbungs-, Aggregations- oder Generalisierungsbeziehungen von Objekten innerhalb des betrachteten Informationssystems müssen durch andere, auf diesen Zweck spezialisierte Modelle beschrieben werden.

Das Informationssystem "WebSiteProfiler" zum Beispiel, das in Kapitel 6 detailliert vorgestellt wird, ist unter Verwendung mehrere Modelle entworfen worden. Mit Hilfe eines Entity-Relationship-Modells wurde die Darstellung der Daten in einem Datenbankschema beschrieben. Das Ergebnis dieses Schrittes ist in Abschnitt 6.1.1 zu sehen. Damit ist eine datenzentrierte Darstellung des WebSiteProfilers erreicht. Jedoch kann mit diesem Ergebnis noch nichts über Programmabläufe innerhalb des WebSiteProfilers ausgesagt werden. Antworten auf die Fragen, ob und wie mehrere Benutzer mit dem System gleichzeitig arbeiten können, welche Aufgaben nebenläufig ausgeführt werden können oder welche Reihenfolge in einzelnen Funktionen eingehalten werden muß, kann man mit dem Modell der *Mainstream Objects* aus Anhang C geben. Dieses Modell bietet eine objektzentrierte Darstellung des Systems. Für die Modellierung der kooperativen Eigenschaften der Anwendung, die die Integration des Systems in einen Gesamtzusammenhang erlauben, wird das Modell der Business Conversations benutzt.

3.2 Konzepte des Modells

Ein integraler Bestandteil des *Business Conversation*-Modells ist ein vereinbartes Verarbeitungsmodell, das die Interaktion zwischen zwei Partnern definiert und ihre jeweiligen Rollen innerhalb der Interaktion, d.h. innerhalb eines Gesprächs, festlegt. Dieses Verarbeitungsmodell muß von beiden Gesprächspartnern eingehalten werden, um den korrekten Fortgang des Gesprächs zu gewährleisten.

3.2.1 Das Verarbeitungsmodell von Konversationen

In einer Konversation wird jede zielgerichtete Interaktion (d.h. jeder mögliche Sprechakt der *Language/Action Perspective*) mit *Dialogen* realisiert. Mit Hilfe dieser Dialoge werden Informationen so strukturiert, daß der jeweilige Kommunikationspartner die im Sprechakt enthaltene Information seinem eigenen Ausführungskontext zuordnen kann.

Dialoge verhalten sich wie Formulare in der realen Welt: ihre Struktur ist festgelegt und durch den Verwendungszweck bedingt. In den durch die Struktur gesetzten Grenzen können sie allerdings von allen beteiligten Partnern genutzt werden, sei es durch Ausfüllen der vorgegebenen Freifelder oder durch Inspektion des vorhandenen Inhalts. Durch Formulare werden von der ausgebenden Seite mögliche Folgeaktionen impliziert, die mit dem Formular in direktem Zusammenhang stehen. Auf diese Weise repräsentiert ein Formular einen bestimmten Punkt in einer längeren Beziehung zwischen zwei Partnern, der Schlüsse auf gültige Fortsetzungen der Beziehung in die Zukunft ermöglicht.

$$
\begin{array}{rrll}
\text{Kunde (initial):} & d_0 & \rightarrow_{f_0} & (d_0, r_0)_{\in S_0} \\[1em]
\text{Dienstleister:} & (d_n, r_m)_{\in S_k \cup S_{Exc}} & \rightarrow_{f_{n,m}} & d_{p \in S_k \cup S_{Exc}} \\
\text{Kunde:} & d_{n \in S_k \cup S_{Exc}} & \rightarrow_{f_n} & (d_n, r_m)_{\in S_k \cup S_{Exc}}
\end{array}
$$

Abbildung 3.3: Das Verarbeitungsmodell als Ereignis-Aktions-Paare

Jeder Dialog beschreibt daher einen Zustand in einer *Konversation* zwischen einem Kunden und einem Dienstleister. Im Gegensatz zur zeitlich ausgedehnten Konversation besitzt ein Dialog keine nach außen sichtbaren Zwischenzustände.

Zustandsübergänge, d.h. der Beginn eines neuen Dialogs innerhalb einer Konversation, werden durch Auswertung und Verarbeitung (die *Aktion*) einer Anfrage des Kunden beim Dienstleister durchgeführt.

Dabei ist der Kunde derjenige Akteur, welcher den Zeitpunkt des Zustandsüberganges bestimmt: indem er eine Anfrage an den Dienstleister stellt, führt dieser seine mit der Anfrage assoziierte Bearbeitungsregel aus und generiert einen neuen (Antwort-)Dialog für den Kunden.

Die Menge der möglichen Anfragen, die ein Kunde ausgehend aus einem Zustand einer Konversation an einen Dienstleister richten kann, ist durch den Dialog bestimmt, welcher den aktuellen Zustand der Konversation repräsentiert. Dies entspricht in der Formularanalogie der Möglichkeit, in einem vorgegebenen Formular einen Kundenwunsch ankreuzen zu können.

Es sind darüber hinaus genau zwei Anfragen definiert, die nicht in der Menge der Anfragen enthalten sind, die ein Dialog explizit vorgibt:

- Die initiale Anfrage: Ein Kunde muß eine Konversation mit einem Dienstleister beginnen können, ohne sich vorher bereits in einer Konversationsbeziehung mit ihm zu befinden. In diesem Fall kann der Kunde genau eine Anfrage an den Dienstleister stellen, nämlich die nach der Eröffnung der Konversation. Die Spezifikation dieser Anfrage und der zug. initiale Dialog wird durch S_0 bezeichnet und ist Teil des *Business Conversation*-Systems.
- Eine spezielle Anfrage nach dem Abbruch der Konversation: Diese Anfrage löst ein Kunde implizit aus, wenn er eine von Dienstleister vorgegebene Zeitspanne lang keine Anfrage an den Dienstleister stellt. Tatsächlich ist dies keine wirklich vom Kunden stammende Anfrage, sondern wird vom System des Dienstleisters im Namen des Kunden generiert. Außerdem ist dem Kunden der explizite Abbruch der Konversation erlaubt. Die Spezifikation dieser Anfrage bzw. des Dialogs wird durch S_{Exc} bezeichnet und ist ebenfalls Teil des *Business Conversation*-Systems.

Eine Konversation zwischen einem Kunden und einem Dienstleister wird mit diesen Mechanismen auf eine festgelegte Abfolge von Dialogen und Anfragen übertragen. Der Dienstleister gibt dabei jeweils einen Dialog vor, den der Kunde erhält und so bearbeiten kann. Der Kunde ist jedoch immer der *Initiator* einer Konversation und der Dienstleister der *Akzeptor*.

Auf diese Weise werden Ketten aus Ereignissen und Aktionen [DHL90] gebildet. Die beiden Partner der Konversation definieren Bearbeitungsregeln, mit denen sie auf die Ereignisse des jeweils anderen Partners reagieren und dadurch Aktionen ausführen. Abbildung 3.3 zeigt das formalisierte Verarbeitungsmodell.

Die gültigen Transitionen einer Konversation, wie sie durch die Regeln implementiert wird, ist durch eine zugehörige Spezifikation bestimmt (siehe auch Abschnitt 3.2.4).

Der ausgezeichnete Dialog d_0 ist der Startdialog einer Konversation und die Menge D^* bezeichnet die Menge der finalen Dialoge einer Konversation.

Für das Verarbeitungsmodell ist eine Spezifikation S_l definiert durch die Dialogmengen D, D^*, die Menge von Anfragen R und die Transitionsrelation E.

$$S_l := (D, R, E, d_0, D^*)$$

mit

$$E \subseteq D \times R \times D, d_0 \in D, D^* \subseteq D$$

Menschliche Akteure simulieren Kundenregeln, indem sie den Inhalt der Dialoge d_i lesen und gemäß ihren Wünschen verändern. Maschinelle Kunden besitzen einen Programmkode für die Regeln, der die Dialoge entsprechend der kodierten Vorschrift interpretiert bzw. manipuliert. Die Regeln f_i des Kunden enden immer mit der Übermittlung des eventuell veränderten Dialogs zusammen mit einer Anfrage r_k.

Ebenso muß der Dienstleister auf die Dialoge und Anfragen des Kunden entsprechend einer Regel $f_{i,k}$ reagieren. Seine Aktion endet immer damit, daß er einen neuen Dialog, den er bei Bedarf vorher mit inhaltlichen Vorgaben gefüllt hat, an den Kunden übermittelt. Durch die wechselseitige Ausführung einer Kundenregel und einer Dienstleisterregel wird jede Konversation gemäß der zugrundeliegenden Spezifikation S_j systematisch entwickelt.

Innerhalb einer Konversation kann ein Dienstleister, dann jedoch in der Rolle eines Kunden, eine weitere Konversation mit einem anderen Dienstleister initiieren, sofern das zur Ausführung seiner Bearbeitungsregel notwendig ist. Dieses führt zum Konzept der *sekundären Konversation*.

3.2.2 Sekundäre Konversationen und Subkonversationen

Mit der Technik der sekundären Konversation ist eine Mischform der Beziehung zwischen Kunde und Dienstleister möglich (siehe Abbildung 3.4). Dabei wird der Dienstleister A_1 innerhalb einer Konversation mit einem Kunden A_2 selbst zum Kunden eines weiteren Dienstleisters A_3.

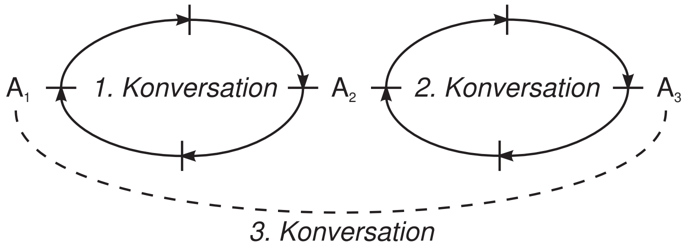

Abbildung 3.4: Verfeinerung mittels sekundären Konversationen

Synchrone sekundäre Konversationen unterbrechen die initiierende primäre Konversation für die Dauer ihrer Ausführung. Sie bilden damit eine eingeschobene Konversation, bei der ein impliziter Rollenwechsel des Akteurs stattfindet. Wird eine sekundäre Konversation asynchron ausgeführt, ist keine Unterbrechung der primären Konversation notwendig. Eine asynchrone sekundäre Konversation verhält sich wie eine primäre, kann jedoch die übergeordnete Konversation *verzögern*, wobei die Atomarität der Bearbeitungsregel aufgehoben wird.

Subkonversation erlauben einem Kunden A_1 die direkte Kommunikation mit einem Dienstleister A_3 , obwohl die Konversationsbeziehung zwischen dem Kunden A_1 nur zu einem Dienstleister A_2 besteht. Die Konversation zwischen A_1 und A_3 wird dabei im Namen von A_2 geführt, so daß dieser die Möglichkeit besitzt, in den Konversationsablauf einzugreifen.

3.2.3 Finale Dialoge

Jede Konversation besitzt eine definierte Menge von *finalen* Dialogen, die sie erreichen kann. Jeder Dialog ohne einen Folgedialog, d.h. ohne eine mögliche ausgehende *Anfrage*, ist ein finaler Dialog der Konversation. Die Konversationsbeziehung zwischen den beteiligten Partnern wird bei Erreichen eines solchen Dialogs beendet.

Das Modell definiert einen Finaldialog für das erfolgreiche Ende jeder Konversation (d.h. sowohl Kunde als auch Dienstleister bestätigen das erfolgreiche Erreichen des gemeinsamen Ziels) sowie einen finalen Fehlerdialog. Dieser Fehlerdialog ist implizit von jedem Dialog einer Konversation aus erreichbar, ohne daß dies in der Spezifikation vermerkt werden muß. Dieser Dialog bietet die Möglichkeit, Ausnahmebehandlungen in den Bearbeitungsregeln für Konversationen vorzusehen. Eine solche Ausnahme kann von beiden Partnern jederzeit in einer Konversation ausgelöst werden. Die entsprechende zugehörige Anfrage, mit der der Kunde den Übergang in einen Ausnahmezustand anzeigt, wird durch das Modell definiert.

Die Entscheidung, unter welchen Umständen welcher Dialog welcher Phase aus Abbildung 3.1 zugeordnet wird, ist eine Designentscheidung bei der Anwendungsentwicklung. Das Phasenmodell kann von einer Anwendung genutzt werden, um Punkte innerhalb einer Konversation mit dem Partner zu vereinbaren, mit deren Erreichen beide Partner verbindliche Aussagen über den Gesprächsverlauf treffen.

Das Modell der Business Conversations verwendet zwei Beschreibungsebenen für Konversationen, die im Folgenden getrennt vorgestellt werden:

- Die Ebene der *Konversationsspezifikationen*: Auf dieser Ebene wird die Struktur möglicher Konversation definiert. Diese Ebene entspricht der Typebene einer Programmiersprache, mit dem Unterschied, daß Konversationsspezifikationen eine Laufzeitrepräsentation besitzen. Konversationsspezifikationen bilden Fabriken (engl. *factories*), welche die Erzeugung von Konversationen, ihren Instanzen, steuern.
- Die Ebene der *Konversationensinstanzen*: Gegenstand dieser Ebene sind konkrete Konversationen, die zwischen einem Kunden und einem Dienstleister stattfinden. Konversationen sind immer konform zu einer zugehörigen Konversationsspezifikation. Sie werden anhand dieser Spezifikation erzeugt und sind an ein Kunde-Dienstleister-Paar gebunden.

3.2.4 Konversationsspezifikationen

Die Abbildung 3.5 zeigt ein Modell für Spezifikationen von Konversationen. Das Diagramm ist in der Notation der Mainstream Objects, die im Anhang C erläutert wird, dargestellt. Alle Begriffe aus dem Modell werden im Glossar im Anhang D.2 in ihrer deutschen und englischen Schreibweise erläutert.

Konversationsspezifikationen legen fest, welche Struktur Konversationsobjekte im Falle einer Instantiierung haben sollen. Sie umfaßt dabei sowohl den statischen Teil einer Konversation, d.h. die Datenmodellierung für den Informationsaustausch, als auch den dynamischen Teil der Beziehung zwischen Kunden und Dienstleistern, die Prozeßmodellierung.

Spezifikationen sind inspizierbar, d.h. beide Partner können untersuchen, ob die Konversation in ihren jeweiligen Kontext und zu ihrer Sicht des Kooperationsziels paßt.

Nach diesem Modell enthält jede Konversationsspezifikation mehrere abstrakte Dialogspezifikationen. Eine Dialogspezifikation ist als initialer Dialog ausgezeichnet, jede Konversation beginnt mit diesem Dialog[2]. Die abstrakten Dialogspezifikationen bestehen entweder aus Spezifikationen von Sub-Konversationen oder von (konkreten) Dialogen. Mit den Spezifikationen von Sub-Konversationen werden Konversationsspezifikationen als Dialogspezifikationen in anderen Konversationsspezifikationen wiederverwendet. Dazu können sie auf geeignete Art und Weise mit dem

[2] Aussagen zur initialen Anfrage siehe Abschnitt 3.2.1.

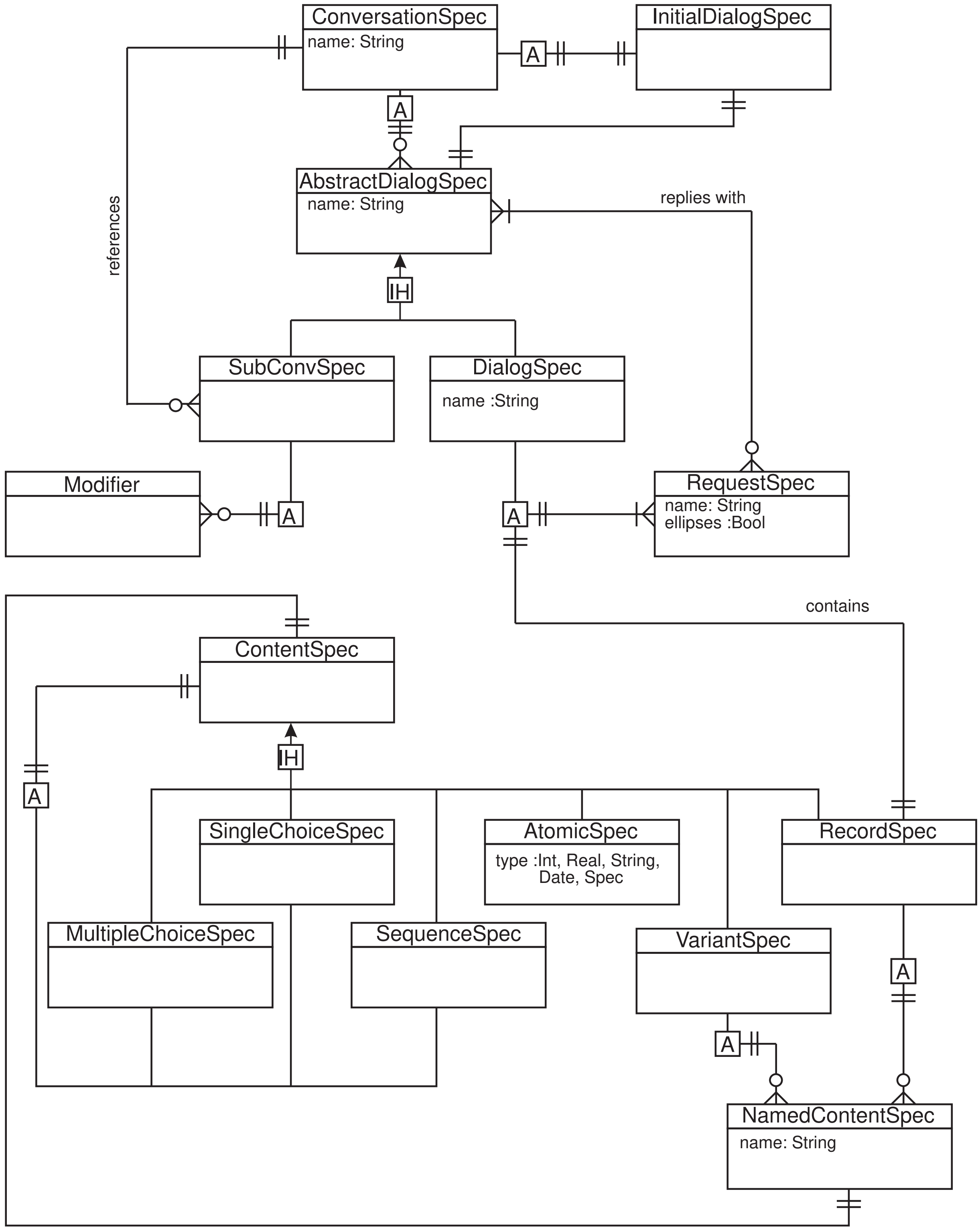

Abbildung 3.5: Spezifikationen von Konversationen, Dialogen und Inhalten

Modifier verändert werden. Dialogspezifikationen bestehen aus Inhaltsspezifikationen und Spezifikationen für Anforderungen. Jede Anforderungsspezifikation ist über die *replies-with Beziehung* mit der Spezifikation des nachfolgenden Dialogs verbunden. Die Markierung *ellpises* zeigt an, daß sich die Aktion, die beim Dienstleister auf die Auswertung des Dialoges folgt, über mehrere Dialoge, deren Spezifikation nicht sichtbar ist, hinziehen kann.

Die Inhalte können auf verschiedene Art und Weise spezifiziert werden. Zum einen gibt es einfache Basistypen wie ganze Zahlen, rationale Zahlen, Zeichenketten, einen Datumstyp und einen Spezifikationsobjekttyp. Komplexe Inhalte können mit Records, Varianten, Auswahllisten oder Sequenzen modelliert werden, wobei auch verschachtelte Inhalte zugelassen sind. Für alle Inhaltstypen existieren Operationen zum Erzeugen der entsprechenden Spezifikationsobjekte. Die Operationen auf den Spezifikationsobjekten sind in der Abbildung aus Gründen der Übersichtlichkeit nicht dargestellt.

Auf der Ebene der Konversationsspezifikationen können Meta-Dienste erstellt werden, die auf diesen Spezifikationsobjekten operieren. In erster Linie sind das grafische Editoren oder Anzeiger, Modellverifikatoren, Spezifikationsmakler und ähnliches mehr. Wichtiger Nutzen der Spezifikationen ist ihre Überprüfbarkeit, durch die eine gesicherte maschinelle Abwicklung von Konversationen unter Beibehaltung der Autonomie der Partner ermöglicht wird.

Alle Spezifikationen sind Objekte erster Klasse, sowohl in der Implementationssprache als auch im eigenen Typsystem. Sie können somit benannt, gespeichert oder verschickt werden und besitzen eine Laufzeitrepräsentation.

In Anhang B findet sich die Grammatik für die Konstruktion von Konversationsspezifikationen gemäß dem dargestellten Modell aus Abbildung 3.5.

3.2.5 Konversationsinstanzen

Die Abbildung 3.6 zeigt den Aufbau von Instanzen von Konversationen und den Zusammenhang mit Spezifikationen.

Von einer Konversationsspezifikation können beliebig viele Konversationen instantiiert werden. Jede Konversation befindet sich zu einem Zeitpunkt in einem Quadranten der Ellipse aus Abbildung 3.1. Sie erlaubt den Zugriff auf den aktuellen Dialog (*current-dialog Beziehung*) und enthält eine Liste von vergangen Dialogen, die durch *History-Element Objekte* dargestellt werden. Mit den abstrakten Dialogen werden wie bei den Spezifikationen Subkonversationen und konkrete Dialoge modelliert. Jeder Dialog wird durch eine Dialogspezifikation inhaltlich vorgegeben. Aus dem Dialog heraus ist der Inhalt, der aus der Spezifikation erzeugt wurde, erreichbar. Die Objektstruktur für die Instanzen der Inhaltsspezifikationen sind in der Abbildung 3.7 zu sehen. Für jeden Inhalttyp sind bestimmte Operationen definiert:

- ▷ Der Zustand der Objekte atomarer Typen kann gelesen und gesetzt werden.
- ▷ In Listenobjekten für eine einfache Auswahl (*single choice*) können die Alternativen gesetzt werden, man kann alle Alternativen auslesen und genau eine davon auswählen. Außerdem kann die gewählte Alternative bestimmt werden.
- ▷ In Listenobjekten für mehrere Wahlmöglichkeiten (*multiple choice*) enthält der Zustand beliebig viele Selektionen. Zusätzlich zu den Operationen bei Listenobjekten mit einfacher Auswahl können gewählte Alternativen wieder deselektiert werden.
- ▷ In einem Record-Objekt können die aggregierten Inhaltsobjekte über ihren Namen referenziert werden.
- ▷ In einem varianten Record-Objekt können zusätzlich gekapselte Varianten ausgewählt oder die Auswahl gelesen werden.

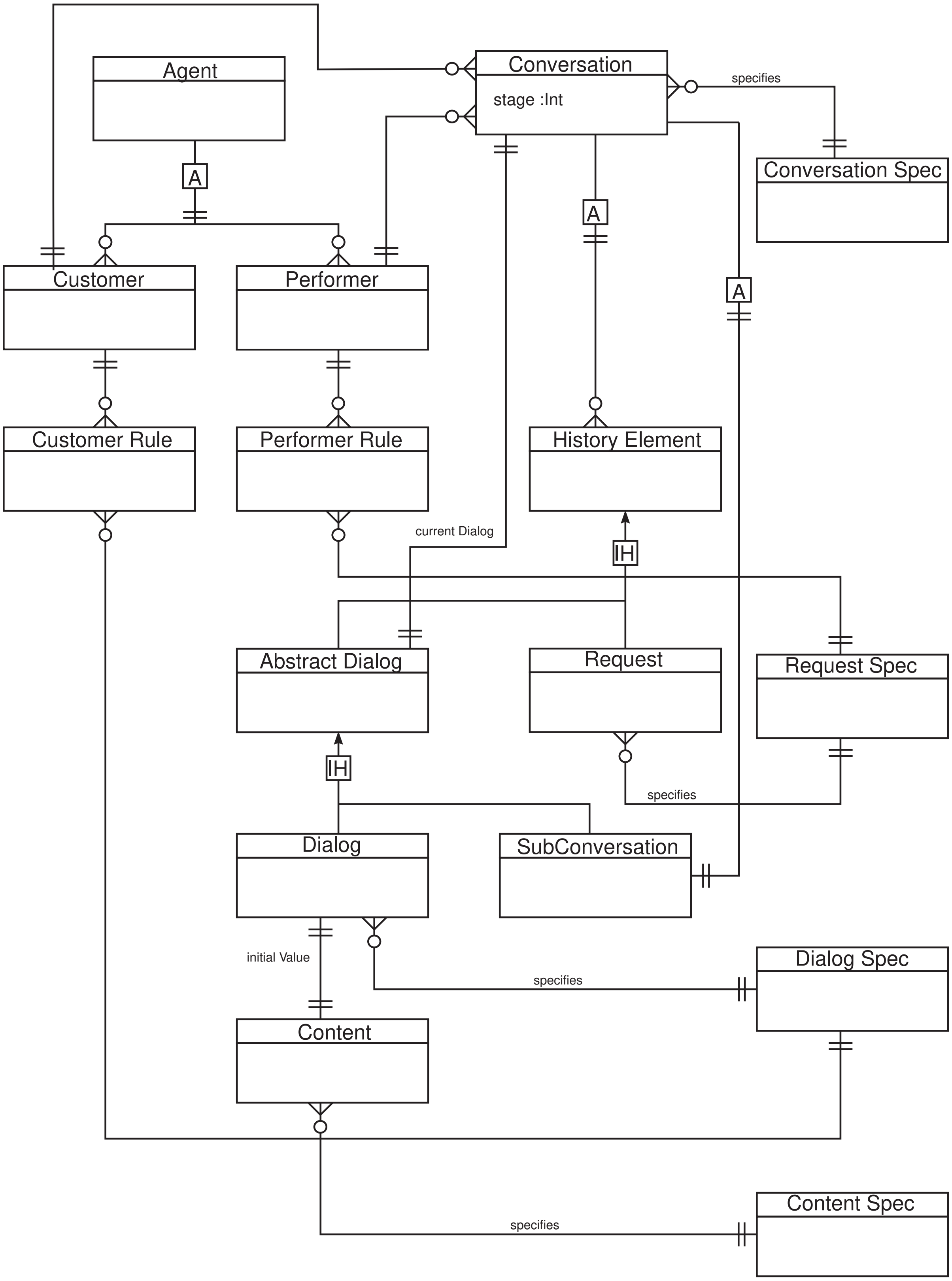

Abbildung 3.6: Instanzen von Konversationen, Dialogen und Inhalten

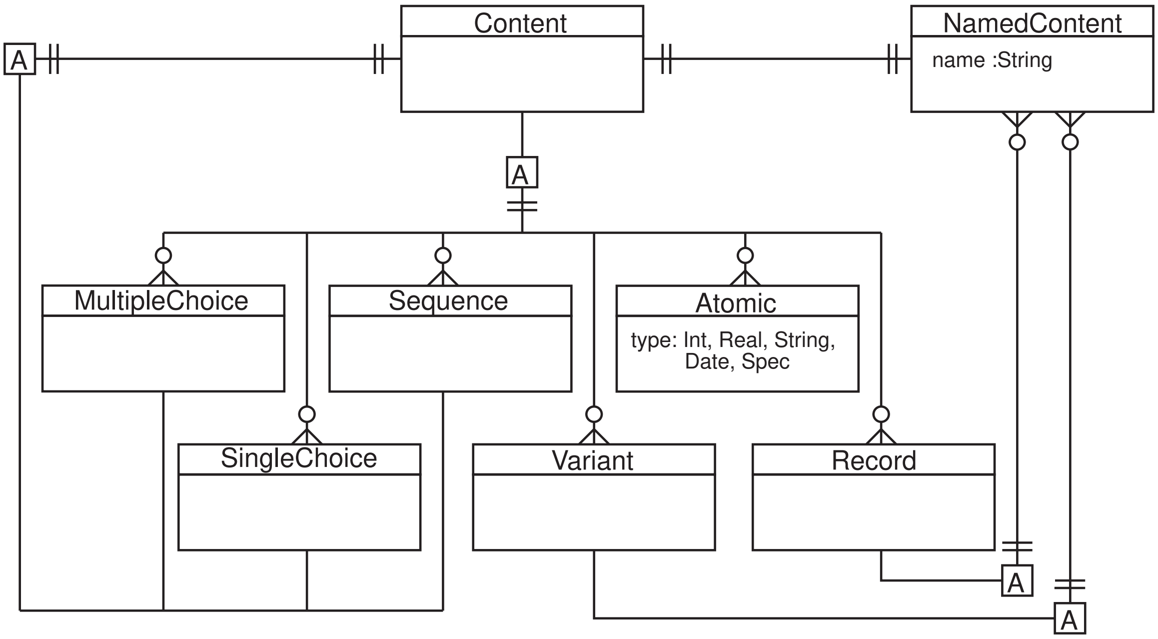

Abbildung 3.7: Instanzen der Inhaltsspezifikation

▷ Die durch Objekte des Sequenztyps aggregierten Objekte sind nicht direkt in das Dialogobjekt eingebunden. Stattdessen stellen Sequenz-Objekte Methoden für den Zugriff auf Massendatenbestände durch Klienten zur Verfügung. Die Erzeugung von Sequenzobjekten erfordert die Konkretisierung der von der Objektschnittstelle geforderten abstrakten Zugriffsfunktionen auf den Datenbestand, der von der Sequenz repräsentiert werden soll.

3.3 Invarianten des Modells

Die folgenden Invarianten sollen Interpretationsmöglichkeiten innerhalb des Modells einschränken. Ziel ist es, möglichst viele Punkte eindeutig festzulegen.

1. Alle Dialogspezifikationen einer Konversationsspezifikation haben verschiedene Namen. Alle benannten Inhaltsspezifikationen einer Dialogspezifikation haben verschiedene Namen. Alle Anfragespezifikationen einer Dialogspezifikation haben verschiedene Namen.

   Die Namen der Objekte dienen in ihren jeweiligen Sichtbarkeitsbereichen als Schlüssel zur Identifikation.

2. Jede Konversationsspezifikation hat eine finale Dialogspezifikation mit dem Namen *breakdown*.

   Damit bleibt jede Konversation auch in Ausnahmefällen in einem gesicherten Zustand. Es kann keine unvorhergesehenen Konfigurationen geben.

3. Jede Dialogspezifikation mit dem Namen *breakdown* hat benannte Inhaltsspezifikationen mit den Namen *lastDialog*, *reason*.

   Damit kann aus der Abbruchsituation später der Grund für den Abbruch, sowie der letzte erfolgreich bearbeitete Dialog, rekonstruiert werden.

4. Alle finalen Dialogspezifikationen haben keine Folgedialoge und deswegen keine Anfragespezifikationen.

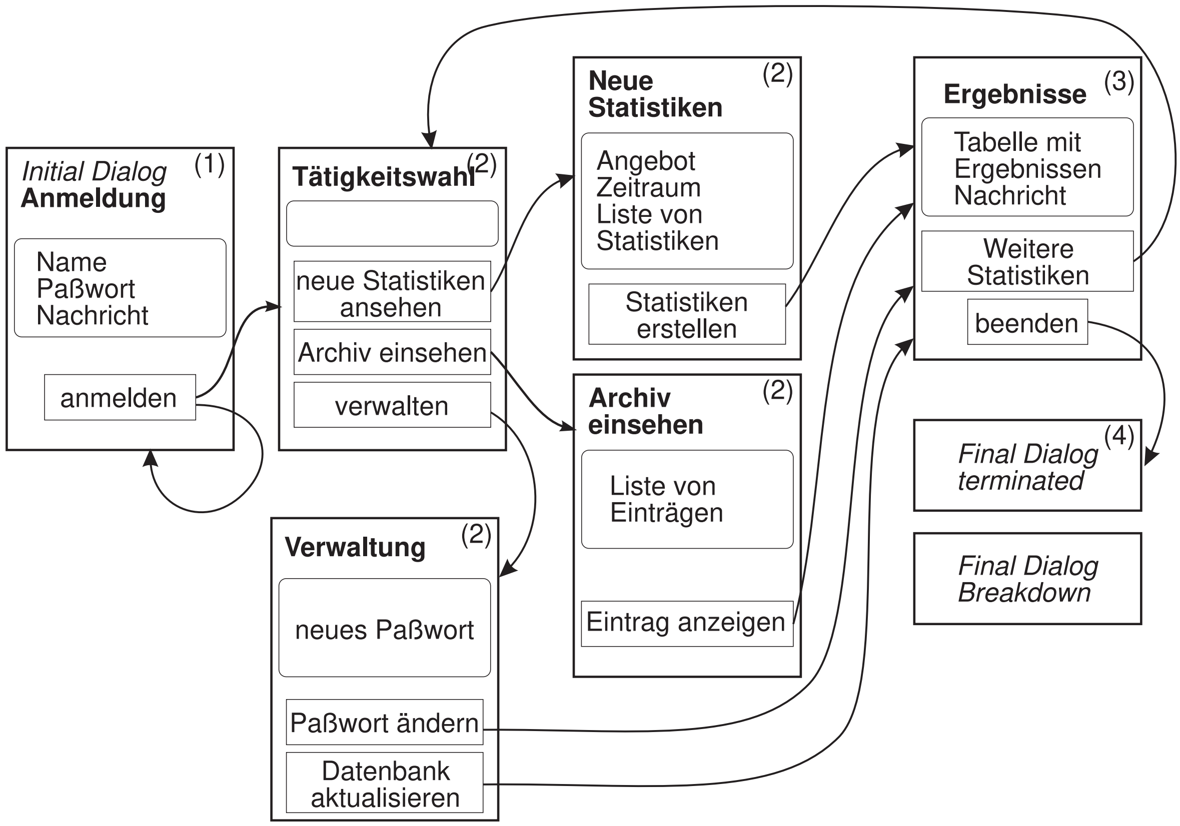

Abbildung 3.8: Konversationsspezifikation für den WSP

Wird ein solcher finaler Dialog in einer Konversation erreicht, so wird diese bei beiden Partnern beendet.

3.4 Ein Anwendungsbeispiel

Die Kooperationsschnittstelle des WebSiteProfilers, der im Abschnitt 6.1 näher beschrieben wird, soll an dieser Stelle die Verwendung des Konversationsmodells illustrieren. Seine Funktionalität, soweit für diesen Abschnitt erforderlich, besteht im Wesentlichen in der Erstellung verschiedener Statistiken über die Nutzung von WWW-Servern im Internet.

Um den WebSiteProfiler in das Modell einzufügen, muß der Programmierer eine Konversation eines Benutzers mit dem Programm entwerfen und spezifizieren. Dabei tritt der WebSiteProfiler in der Rolle des Dienstleisters auf.

Eine denkbare Dialogfolge und ihre Umsetzung in eine Konversationsspezifikation ist in der Abbildung 3.8 aufgezeigt.

Der initiale Dialog der spezifizierten Konversation dient der Autorisierung möglicher Benutzers des Informationssystems.

In diesem Dialog sind Objekte zur Aufnahme seines Namens und des zugehörigen Paßwortes vorgesehen. Zusätzlich zu diesen beiden Objekten ist ein Objekt für Nachrichten an den Benutzer als Inhaltsspezifikation in der Dialogsspezifikation eingetragen.

Die aus diesem Dialog mögliche Anfrage des Kunden ist lediglich: *“anmelden”*[3].

[3] Implizite Anfragen und Ausnahmen werden in diesem Beispiel nicht berücksichtigt.

Dabei wird der in der Spezifikation enthaltene Nichtdeterminismus durch die für diesen Dialog angegebene Bearbeitungsregel des WebSiteProfiles, abhängig vom Zustand der Inhaltsobjekte, aufgelöst.

Nach erfolgreicher Anmeldung kann der Benutzer auswählen, ob er neue Statistiken erstellen, das Archiv einsehen oder die aktuelle Zugriffsprotokolldatei in die Datenbank übernehmen möchte. In diesem Dialog sind keine Inhaltsobjekte vorgesehen.

In dem Dialog "Neue Statistiken", der aus dem Dialog "Tätigkeitswahl" mit der Anfrage "neue Statistiken ansehen" erreicht wird, kann der Kunde die Parameter seiner gewünschten Auswertung eintragen. Dabei ändert dieser destruktiv den Zustand der benannten Inhaltsobjekte. Diese Inhaltsobjekte werden vom Dienstleister, anhand der Dialogspezifikation "Neue Statistiken", erzeugt.

Die Ergebnisse einer Auswertung werden dem Kunden im Dialog "Ergebnisse" präsentiert, der wiederum durch den Kunden ausgewertet werden muß.

Wählt der Kunde in diesem Dialog die Anfrage "beenden", so endet die Konversation bei beiden Partnern. Andernfalls wird sie im Zustand "Tätigkeitswahl" fortgesetzt.

Durch die Schleife in der Konversationsspezifikation sind langandauernde Beziehungen zwischen einem Kunden und einem Dienstleister modellierbar.

In jedem Dialog hat der Kunde die Möglichkeit, die Konversation mit der systemübergreifend definierten Anfrage *"breakdown"* zu beenden. Auch der Dienstleister kann seinerseits die Konversation vorzeitig beenden. Zu diesem Zweck muß er dem Kunden den ebenfalls systemübergreifend definierten, finalen Dialog *"breakdown"* übermitteln.

Wenn die Konversation erfolgreich abgeschlossen wurde, wird der Dialog "terminated" an den Kunden zugestellt.

Auf der Basis dieser Spezifikation wird jetzt für jede durchzuführende Konversation eine Instanz erzeugt. Diese Konversationsinstanz bindet einen speziellen Kunden mit einem speziellen WebSiteProfiler und einer eigenen Vergangenheit. Jeweils eine solche Konversationsinstanz befindet sich auf der Seite des Kunden und auf der Seite des Dienstleisters. Die zur Instantiierung notwendige Spezifikation muß ebenfalls auf beiden Seiten vorhanden sein.

Ein Beispiel für eine Formulierung einer Dialogspezifikation aus der Abbildung 3.8 findet man in der Abbildung 3.9.

```
DIALOG Ergebnisse STAGE 3 WITH
  REQUEST
    Weitere Statistiken TO Taetigkeitswahl NOELLPISES
  END
  REQUEST Beenden TO Beendet NOELLIPSES END

  CONTENT Ergebnisliste OF
    SEQ OF
      SEQ OF
        REC
          unit OF INT
          count OF INT
        END
  END
END
```

Abbildung 3.9: Spezifikation für den Dialog mit Ergebnisstatistiken

Kapitel 4

Mobile Software-Agenten

Gegenstand dieses Kapitels sind mobile Software-Agenten. Anders als in den vorangegangenen Abschnitten bezeichnet der Begriff *Agent* im diesem Kapitel Software-Komponenten, die den bereits geschilderten Anforderungen bezüglich Kooperation und Autonomie gerecht werden. Es beginnt mit einer Übersicht der unter dem Begriff Agent zusammengefaßten unterschiedlichen Konzepte in der Informatik. Danach werden exemplarisch fünf Agentensysteme und Modelle vorgestellt, die eine Grundlage für die Modellbildung des in Kapitel 5 beschriebenen Agentensystems bilden.

4.1 Begriffsbestimmung

Der Begriff des "Agenten" ist nicht neu. Bereits seit ca. 10 Jahren wird diese Bezeichnung auf fast jedes Gebilde der Informatik angewendet, das ein Mindestmaß an Autonomie aufweisen kann. Die Folge dessen ist, daß sich keine konsistente Definition des Begriffs herausgebildet hat. Allerdings gibt es Ansätze, zumindest eine Klassifikation der unter diesem Begriff versammelten Konzepte zu erreichen [BW94]:

User Agents aus dem Bereich der verteilten künstlichen Intelligenz (VKI) sind adaptive, lernende und selbstkonfigurierende Softwarebausteine. Ihre Aufgabe besteht darin, Tätigkeiten auszuführen, die im Normalfall ohne Agenten von deren menschlichen *alter ego* wahrgenommen werden. Dabei versuchen die Agenten durch Beobachten von sich wiederholenden Tätigkeiten des menschlichen Benutzers auf Routineabläufe zu schließen und diese dann an dessen Stelle auszuführen.

Leitagenten (Agent Guides) werden Softwarekomponenten genannt, die in der Rolle eines Lehrers Hilfestellung oder spezifisches Wissen an einen menschlichen Benutzer vermitteln. Dabei unterstützen sie ihn kontextabhängig bei Problemen der Benutzung von komplexen Softwaresystemen. Sie bilden somit ein flexibles Hilfesystem mit einer natürlichen (da einem menschlichem Vorbild entsprechenden) Metapher.

Autonome Agenten werden charakterisiert durch ihren Grad an Autonomie in Bezug auf die Ausführung der ihnen zugewiesenen Aufgaben. Sie sind in der Lage, diese ohne weitere Interaktion mit dem Benutzer zu erledigen. Dabei arbeiten sie losgelöst von ihrem Benutzer und zwar sowohl zeitlich als auch örtlich. Dadurch sind sie in der Lage, große Datenbestände ohne aufrechterhalten einer permanenten Netzverbindung zu verarbeiten und nur die relevanten Informationen dem Benutzer zu präsentieren.

Symbiotische und kooperative Agenten sind in der Lage, einen menschlichen Benutzer bei der Erledigung seiner Aufgabe zu helfen. Sie sind damit eine Form der Leitagenten, können aber darüber hinaus dem Benutzer alternative Problemsichten darstellen oder ihn mit zusätzlichen, für die Lösung seines Problems relevanten, Informationen versorgen.

Anthropomorphe Agenten imitieren Menschen. Diese Agenten stammen aus der Domäne der "Multi-User" Spiele [Fon93] und haben innerhalb des Spiels die gleichen Fähigkeiten wie menschliche Spieler, führen dabei insbesondere natürlichsprachliche Dialoge.

Mobile Agenten sind eine Form der autonomen Agenten, die die Fähigkeit zur Mobilität besitzen, d.h. sich aufgrund eigener Entscheidungen aktiv an andere Orte bewegen zu können. Das bekannteste und kommerziell erhältliche System basierend auf Agenten dieser Art sind Telescript und Magic Cap [GMI95b] von General Magic, Inc.

Russel Beale und Andrew Wood geben in [BW94] eine Definition des Begriffs Agent, die in dieser Arbeit im weiteren Verlauf gelten soll: „Ein Agent kann als eine relativ einfache, heterogene, autonome und kommunizierende Software-Komponente definiert werden; viele dieser Agenten existieren gleichzeitig und arbeiten zusammen, um eine Aufgabe für einen Benutzer zu erfüllen.“

Über diese generelle Definition hinaus zählen die im weiteren betrachteten Agenten, aufgrund ihrer geforderten Mobilität, zur Klasse der *mobilen Agenten*.

4.2 Verwandte Arbeiten

In diesem Abschnitt werden verschiedene Referenzimplementationen für Agentensysteme vorgestellt. Dabei handelt es sich in erster Linie um Telescript, ein kommerzielles Agentensystem der Firma General Magic. Desweiteren werden mit Facile, Mole und COSY drei Ansätze aus der Forschung vorgestellt, wobei es sich bei COSY um ein System aus dem Gebiet der Verteilten Künstlichen Intelligenz (VKI) handelt. Abschließend wird kurz auf eine gänzlich andere Form mobiler Agenten, auf die der MIME-Agents, eingegangen.

4.2.1 Telescript

Telescript ist eine objektorientierte Programmiersprache für die verteilte Programmierung mit mobilen Agenten. Die Telescript-Technologie wurde für den Einsatz mit sog. "Persönlichen Digitalen Assistenten (PDA)" entwickelt. Für das amerikanische Unternehmen General Magic ist Telescript das Fundament für verteilte Anwendungen unter Einbeziehung von PDA, PC, Telefonen und Netzwerken.

Neben der Tatsache, daß Telescript das erste kommerziell verfügbare Programmiersystem für mobile Agenten war, bildet das Denkmodell hinter Telescript [Whi94] die Grundlage verschiedener anderer Agentsysteme, wie z.B. das im Abschnitt 4.2.3 beschriebene Mole.

Telescript kennt die folgenden Konzepte: Orte, Agenten, Reisen, Treffen und Kommunikationsverbindungen.

Orte strukturieren den Raum, in dem sich Agenten bewegen können. Sie bilden eine logische Struktur innerhalb einer verteilten Umgebung, die sich an den in ihr angebotenen Diensten und an den Autoritäten der beteiligten Rechnersysteme oder Dienstanbieter orientiert. Telescript geht von einer offenen Dienstnutzungsumgebung, insbesondere für den elektronischen Handel, aus. In dieser Form bildet die offene Dienstnutzungsumgebung einen *elektronischen Marktplatz*, auf dem sich Anbieter von Dienstleistungen mit ihren jeweiligen Kunden treffen und interagieren.

Ein elektronischer Marktplatz ist untergliedert in viele verschiedene und voneinander unabhängige Orte. Ein solcher Ort, evtl. physisch auf einem PDA gelagert, repräsentiert das Zuhause eines Benutzers. Andere Orte stellen z.B. virtuelle Kaufhäuser oder Unternehmen dar und sind auf andere Rechnersysteme, möglicherweise über die ganze Welt, verteilt. Orte bilden untereinander eine

Hierarchie, können daher wiederum andere Orte enthalten. Auf diese Weise wird die elektronische Welt, ähnlich der realen Welt, in Unternehmen, Abteilungen, Unterabteilungen, Räume o.ä. unterteilt.

Jeder Ort repräsentiert, innerhalb der elektronischen Welt, ein Individuum oder eine Organisation, die sog. Autorität, aus der realen Welt.

Orte können nun dynamisch von Agenten bevölkert werden, um mit den anderen Agenten desselben Ortes zu interagieren. Jeder Agent kann mittels einer Reise zu beliebigen anderen Orten gelangen, sofern der Zielort den Eintritt gewährt. Anbieter und Kunden, jeweils durch Agenten repräsentiert, können auf diese Weise zueinander gelangen, um miteinander zu handeln.

Zusammengefaßt besitzt das Telescript-System folgende Eigenschaften [GMI95a]:

- ▷ objektorientiert (ähnlich Smalltalk: alles ist ein Objekt)
- ▷ Klassen, Vererbung, Mixins (eine Art der Mehrfachvererbung)
- ▷ interpretierter Bytecode (virtuelle Maschine)
- ▷ orthogonale Persistenz als Grundlage für Mobilität
- ▷ Sicherheitsmechanismen
- ▷ spezielle, autonomieerhaltende Objektbindungen
- ▷ Ressourcenkonzept über Teleclicks
- ▷ Multithreading
- ▷ Interaktionsprotokolle
- ▷ Kommunikationsmechanismen
- ▷ Anbindung externer Dienste möglich

Neben einem Übersetzer, der Telescript-Kode in den Bytecode der virtuellen Maschine wandelt, besteht das System aus einer Laufzeitumgebung (engl. *engine*), die auf jedem Knoten des Telescript-Netzes installiert sein muß.

Diese Laufzeitumgebung bietet die Kerndienste für Mobilität und Kooperation von Agenten und den Betrieb der Orte. Sie beinhaltet im wesentlichen die virtuelle Maschine und die Anbindung an das Netzwerk. Desweiteren implementiert sie die Sicherheitsmechanismen, Authentisierung, Autorisierung und autorisiert die Mitnahme von Objekten anderer Besitzer im Falle einer Migration von Agenten an einen anderen Ort.

Die Telescript-Programmiersprache bietet eine Reihe von Mitteln, den Zugriff auf Objekte (d.h. deren Referenzen) zu kontrollieren. Zu diesem Zweck ist es möglich, Objektreferenzen zu schützen (sog. *protected references*), so daß die von ihnen bezeichneten Objekte nicht modifiziert werden können.

Darüber hinaus gehört zu jedem Objekt ein Besitzer (engl. *owner*), entweder ein Agent oder ein Ort. Nur Objekte, die ein Agent besitzt, kann dieser bei einer Reise an einen anderen Ort mitnehmen. Alle anderen Objekte werden am Ursprungsort zurückgelassen, so daß der Agent seine Referenzen auf diese verliert. Objekte, die in der transitiven referentiellen Hülle des Objektgraphs eines Agenten erreichbar sind, werden von der Engine daraufhin überprüft, ob sie dem Agenten gehören. Ist dies nicht der Fall, so wird vor der Migration der Objektgraph an der betreffenden Stelle gekappt (der Objektgraph wird *isoliert*). Auf diese Weise stellt die Engine sicher, daß die Autonomie von Agenten bewahrt wird.

Des weiteren kennt Telescript sog. Erlaubnisscheine (*permits*), die die Fähigkeiten eines Agenten oder Ortes kennzeichnen. Diese Erlaubnisscheine bestimmen, welche Systemeigenschaften einem Agenten oder Ort zur Verfügung stehen. Insbesondere kann ein Erlaubnisschein die Migrationsoperationen für einen Agenten sperren oder einem Agenten bzw. Ort erlauben, die Fähigkeiten *anderer* Agenten einzuschränken.

Jeder Agent und jeder Ort wird von der Telescript-Engine als eigener, autonomer Prozeß ausgeführt. Die Kooperation von Prozessen untereinander wird durch die Engine überwacht. Dabei erfolgt die Interaktion von Prozessen in Form eines Rendezvous [HH94], das durch die Engine vermittelt wird. Der Informationsaustausch zwischen den Partnern eines Rendezvous geschieht mittels des direkten Methodenaufrufs des Initiators im Objekt des Akzeptors. Dabei erhält der Initiator mittels des Sprachkonstruktes *meet* eine Bindung[1] an das Objekt des Akzeptors. Die Beziehung ist asymmetrisch, d.h. der Akzeptor erhält keine Bindung an den Initiator. Beide Rendezvouspartner dürfen ein solches Treffen jederzeit abbrechen. Die Engine sorgt für eine Isolation der Partner nach Beendigung des Treffens.

Neben der Möglichkeit der direkten Interaktion, bei der sich immer beide Partner am selben Ort befinden müssen und auch keine Migration erlaubt ist, stellt Telescript eine verbindungsorientierte Kommunikationsform zu Verfügung, die dem Socket-Mechanismus aus dem Betriebssystem Unix ähnelt [Sun90]. Diese Verbindungen sind migrationstransparent und erlauben den Austausch von Daten über Rechnergrenzen hinweg.

Innerhalb einer Kooperationsbeziehung zwischen Prozessen wird keine implizite Serialisierung durch die Telescript-Engine erzwungen. Statt dessen kann ein Objekt dies durch sog. Ressource-Objekte erreichen, die sowohl konditional als auch unkonditional genutzt werden können. Mit Hilfe dieser Ressource-Objekte werden (*konditionale*) kritische Abschnitte geschaffen, die einer einfachen Konfliktmatrix unterliegen. In dieser Matrix werden die exklusive Benutzung und die geteilte Benutzung einer Ressource unterschieden. Exklusive Benutzung ist immer inkompatibel zu jeder anderen Benutzungsart, geteilte Benutzung ist kompatibel zu einer geteilten Benutzung durch einen anderen Prozeß. Werden Konditionen verwendet, muß sich die Ressource zusätzlich in einem Zustand befinden, der vom anfragenden Prozeß erwartet wird.

Die Telescript-Engine bietet weiter zur Senkung des Übertragungsvolumens bei einer Migration von Agenten die Möglichkeit, die transitive referentielle Hülle eines Agenten zu begrenzen. Wird ein Objekt gefroren (engl. *freezing*), so werden alle abhängigen Objekte, die durch eine gegebene Klassenbibliothek erzeugt werden können, durch Identifikatoren der Bibliothek ersetzt.

Beim Auftauen (engl. *thawing*), werden diese Identifikatoren mit einer lokalen Klassenbibliothek zur Rekonstruktion der Originalobjekte verwendet.

4.2.2 Facile

Facile ist eine weitgehend[2] funktionale, streng typisierte Sprache höherer Ordnung. Sie ist eine Erweiterung der Sprache ML [HMM86, MTH89] zur Unterstützung von verteilter und nebenläufiger Programmierung. Facile wurde am ECRC[3] entwickelt und von Frederick Knabe im Rahmen einer Dissertation als Grundlage für ein Agenten-Programmiersystem verwendet [Kna95]. Die Sprache ist seit 1994 unter der Bezeichnung *Facile Antigua Release* am ECRC frei verfügbar.

An agentenorientierten Erweiterungen stehen in Facile Bindungstechniken für ubiquitäre Ressourcen, eine optimierte Kodeerzeugung für mobile Funktionen und Koderepräsentationen für heterogene Ausführungsumgebungen zur Verfügung.

Das Binden an ubiquitäre Ressourcen geschieht statisch typisiert über global verfügbare (z.B. mittels FTP verteilte) übersetzte Signaturen der jeweiligen Ressource. Facile identifiziert kompatible Ressourcen anhand eindeutiger Id's in den exportierten Signaturen, die zur Übersetzungszeit

[1] Unter Verwendung des dargestellten *Ownership*-Konzepts zur Wahrung der Autonomie der Agenten.
[2] Facile besitzt zusätzliche imperative Konstrukte.
[3] ECRC ::= European Computer-Industrie Research Centre.

erzeugt werden. Das Binden (bzw. Isolieren) selbst wird durch ein Manipulieren der transitiven, referentiellen Hülle eines Objektes (engl. *closure trimming*) bei dessen Übertragung vom Laufzeitsystem transparent vorgenommen (siehe auch Abschnitt 8.4).

Facile kann Funktionen in verschiedenen Koderepräsentationen übertragen, die entweder nach Platzbedarf oder Ausführungsgeschwindigkeit optimiert sind und sich für heterogene Systemumgebungen eignen.

Agenten werden mit Skripten realisiert, die über Nachrichtenkanäle (engl. *channels*) übertragen werden. Skripte sind in Facile spezielle Funktionsabschlüsse, die beliebigen Kode beinhalten dürfen und sich auf eine definierte Agentenumgebung beziehen können.

Die Agentenumgebung von Facile besteht aus einer Sammlung von Domänen mit je einem Verzeichnisdienst, dem *Domain Directory Service*, und einem Überwachungswerkzeug, dem *Agent Tracking Service*. Über den Verzeichnisdienst werden alle Dienste angeboten, die in einer Domäne vorhanden sind. Dieser Dienst wird selbst über einen Agenten implementiert. Über diesen Dienst können Benutzer vorhandene Agenten anfordern, deren Kopien dann lokal ausgeführt werden. Allerdings bietet Facile keine Persistenz, so daß ein einmal übertragener Agent nach einem Systemneustart erneut bei dem Verzeichnisdienst abgeholt werden muß.

4.2.3 Mole

Das Agentensystem Mole ist im Rahmen einer Diplomarbeit an der Universität Stuttgart entstanden [Hoh95] und basiert auf einer Klassenbibliothek für die Programmiersprache Java der Firma Sun Microsystems [Sun95a].

Java ist eine objektorientierte Programmiersprache, deren Syntax der Sprache C ähnelt. Java-Programme sind portabel, sie werden in einen Bytecode übersetzt und auf einer virtuellen Maschine ausgeführt. Die virtuelle Maschine ist für den Betrieb in einer offenen Umgebung ausgelegt und bietet daher eine Reihe von Sicherheitsmaßnahmen, die das Betriebssystem vor unerlaubtem Zugriff durch Java Programme schützen [Sun95b].

Java bietet in seiner momentanen Version keine orthogonale Persistenz. Daher geht Mole einen anderen Weg bei der Migration von Agenten als Telescript.

Mole-Agenten müssen alle Objekte, die nach einer Migration auf dem Zielrechner vorhanden sein sollen, explizit bei einem zur Umgebung gehörenden *Persistency Manager* anmelden. Dabei muß das Objekt liniarisierbar sein, d.h. es muß Methoden anbieten, um sich in eine Zeichenkettenrepräsentation zu konvertieren. Die Rückwandlung in ein Java-Objekt geschieht mittels des Persistency Managers, in dem dieser die erzeugte Zeichenkette auswertet. Tatsächlich beinhaltet die Zeichenkettenrepräsentation eines Objektes gültigen Java-Quellcode, der vom Java-Compiler übersetzt werden kann. Diese Form der reflektiven Programmierung erfordert, daß jedes Objekt einer Klasse entstammt, die eigens für die Mobilität vorbereitet worden ist.

Das Verarbeitungsmodell vom Mole ist an Telescript angelehnt. Mole kennt ebenfalls Orte, Agenten und Treffen, die semantisch zu den Konzepten von Telescript äquivalent sind. Auch das Kooperationsmodell stimmt mit dem von Telescript überein, es basiert auf dem Methodenaufruf in anderen Agenten, allerdings kennt Mole keine besonderen Bindungsarten an Objekte.

Bei Mole migrieren keine Threads in Ausführung. Stattdessen wird nach erfolgter Kode-Migration, die durch einen Methodenaufruf des Agenten im Objekt des Umgebungsortes initiiert wird, die Wiederherstellung der pseudo-persistenten Objekte durchgeführt. Danach wird der Agent erneut mit der *start*-Methode gestartet, so daß der Ausführungspunkt des Original-Thread verlorengeht. Der Programmierer muß dies bei der Erstellung von Mole-Agenten berücksichtigen. Weiter ist eine Migration nur dann erlaubt, wenn der Agent sich nicht in einer Meeting-Beziehung zu einem anderen Agenten befindet.

Neben der Interaktionsform über lokale Rendezvous, die ähnlich Telescript durch ein Treffen eingeleitet werden, bietet Mole die Kommunikation über den von Java bereitgestellten Socketmechanismus und globale Nachrichten. Diese werden beim Empfänger in einem vom Agentensystem verwalteten Briefkasten (engl. *mailbox*) des Agenten gesammelt und können dort von diesem abgeholt werden.

Die Ausführungsumgebung von Mole stellt eine Reihe von Basisdiensten zu Verfügung. Dies sind neben der Überwachung von Rendezvous und Migration der Agenten ein Namensdienst für die systematische Benennung und ein *Heartbeat*-Dienst, welcher ein periodisches Aufwecken eines Agenten durch den Aufruf der entsprechenden Methode im Agenten, ausgehend vom Agentensystem, ermöglicht.

Java selbst bietet einen Standardmechanismus zur automatischen Klasseninstallation an, bei dem die Java-Engine fehlende Klassen mittels dem HTTP[4] beim Ursprungsort des Agenten nachfordert. Diese Klassen werden dann automatisch auf dem Zielsystem installiert und stehen dem Agenten zur Verfügung.

4.2.4 COSY

Obwohl das AI-Agentensystem COSY [Had95], das in diesem Abschnitt beschrieben wird, keinen Mobilitätsbegriff kennt, ist es dennoch für die weitere Arbeit interessant. Das System besitzt, im Gegensatz zu den bisher vorgestellten Ansätzen, eine auf Kooperation und Autonomie der Agenten ausgelegte Architektur.

COSY Agenten basieren auf interpretierten Skripten und dem Austausch von Nachrichten mit ihren Kooperationspartnern gemäß spezifizierter Protokolle. Diese Protokolle sind anwendungsabhängig, d.h. sie können für jeden Agenten gesondert definiert werden.

Das Kooperationsmodell orientiert sich an den Sprechakten aus Abschnitt 2.5. Die Architektur verwendet eine strenge Klassifikation von Nachrichten, die zwischen den Agenten auf der Anwendungsebene ausgetauscht werden können. Dabei handelt es sich um elf verschiedene Nachrichtentypen der Art *inform*, *request*, *notify*, ..., die durch die Theorie der Sprechakte motiviert sind. Anhand der Nachrichtenart wird die vom Sender beabsichtigte Reaktion des Empfängers auf deren Inhalt ausgedrückt.

Die in COSY definierten Nachrichtenarten fallen in zwei Gruppen: eine Gruppe, deren Nachrichten Antworten des Kooperationspartners erwarten und eine zweite Gruppe, deren Nachrichten keine Antwort erfordern.

Neben den interpretierten Skripten besitzen COSY-Agenten eine Wissensbasis, die den Ablauf der Skripte beeinflußt. Mit der Wissensbasis werden Annahmen (*beliefs*), Wünsche (*desires*) und Absichten (*intentions*) eines Agenten repräsentiert[5].

Die wesentlichen Architekturkomponenten, die im folgenden betrachtet werden, sind: die Ausführungskomponente (engl. *Protocol Execution Component* (PEC)) und die Entscheidungskomponente (engl. *Reasoning and Deciding Component* (RDC)).

Jeder Agent im COSY-System definiert verschiedene Kooperationsprotokolle, die dynamisch durch das Skript von den Agenten gewählt werden können. Protokolle entsprechen den Konversationsspezifikationen aus Kapitel 3.

Die zur Kooperation verwendeten Nachrichtenarten sind allerdings vielfältiger, wobei die jeweilige Absicht der Nachricht bereits an der Nachrichtenart erkennbar ist. Es ist darüber hinaus einem Agenten möglich, primitive Nachrichten ohne Protokollspezifikation zu versenden, wobei der empfangende Agent aus der Nachrichtenart schließen kann, was der Sender beabsichtigt. Allerdings ist

[4] HTTP ::= HyperText Transfer Protocol

[5] Es handelt sich somit um ein BDI-System, wenngleich die Terminologie in [Had95] abweicht.

dann, falls die Nachrichtenart eine Antwort impliziert, nicht vorhersagbar, welche Antwortnachricht der Empfänger wählt.

Es können sich mehrere Protokolle gleichzeitig in Ausführung befinden. Jede Transition innerhalb eines Protokolls für sich ist hingegen atomar, d.h. alle anderen Protokolle sind während der Ausführung suspendiert. Die PEC wählt das zur Nachricht gehörende suspendierte Protokoll und aktiviert es. Wenn kein zugehöriges Protokoll existiert, wird ein neues instantiiert.

Die RDC hat die Aufgabe, aus dem Inhalt einer eingehenden Nachricht und dem momentanen Zustand des Agenten eine gültige Fortführung des Protokolls auszuwählen und seine Wissensbasis ggf. anzupassen. Dabei berücksichtigt die Komponente die bereits in der Wissensbasis repräsentierten strategischen und taktischen Absichten des Agenten. Die RDC liefert der PEC (d.h. dessen Scheduler) Informationen, welche Protokolle ausgeführt werden sollen.

4.2.5 Tcl/MIME Agents

Das Agentensystem, das hier vorgestellt wird, unterscheidet sich von den bisher genannten dadurch, daß es keine besondere Ausführungsumgebung voraussetzt[6]. Stattdessen werden diese Agenten als Programmskripte im Anhang von elektronischer Post versendet und tragen deshalb auch die englische Bezeichnung *active mail*.

Das in [Ros93] vorgeschlagene Agentensystem basiert auf einem Dialekt der frei verfügbaren Skriptsprache Tcl, Safe-Tcl. Programme dieser Sprache werden ohne Vorübersetzung interpretiert. Weiter bietet Tcl GUI-Elemente, eine einfache Syntax, und ist bereits als "Vermittlersprache" zwischen Systemen konzipiert. Der Interpreter ist kompakt und laut [Ros93] leicht in andere Systeme integrierbar.

Tcl-Agenten werden mittels einer Erweiterung des Multi-Purpose Internet Mail Extensions Standards (MIME) für aktive Mail verschickt. Über diesen Standard sind bereits eine Reihe von Multimedia-Erweiterungen für elektronische Post realisiert.

Agenten sind Tcl-Skripts, die beim Eintreffen mit dem Postsystem des Empfängers interagieren und abhängig von der Interaktion selbsttätig Aktionen ausführen können.

Dabei wird die notwendige Sicherheit durch ein Zwillings-Interpreter-Modell erreicht. Es besteht aus einem, auf einen sicheren Sprachsubset reduzierten, *untrusted*-Interpreter und einem vollständigen *trusted*-Interpreter. Der *untrusted*-Interpreter besitzt keinen Zugriff auf das zugrundeliegende Betriebssystem auf dem das e-mail System arbeitet. Alle Skripte, die in der aktiven Post enthalten sind, werden von diesem Interpreter ausgeführt.

Der Besitzer des Agentensystems kann nun die Funktionalität des *untrusted*-Interpreters schrittweise erhöhen, indem er Funktionen definiert, die vom *trusted*- in den untrusted-Interpreter exportiert werden, aber vom trusted-Interpreter ausgeführt werden. Auf diese Weise wird erreicht, daß die Sicherheit des Gesamtsystems (aus der Sicht des Benutzers) nicht gefährdet wird. Allerdings setzt dies voraus, daß die exportierten Prozeduren mit sorgfältig programmiert sind und keine Sicherheitslücken bieten, die einen unkontrollierten Zugriff auf den *trusted*-Interpreter ermöglichen.

[6] Abgesehen von dem Interpreter für die Agentenskripte.

Kapitel 5

Die Ausführungsumgebung

Dieses Kapitel beschreibt die Architektur der Ausführungsumgebung für mobile Agenten und deren Komponenten. Dabei wird auf bereits vorhandene Konzepte und Realisierungen verwiesen. Das Kapitel ist nicht an ein spezifisches Programmiersystem gebunden und zeigt Realisierungsmöglichkeiten für verschiedene Systemplattformen auf. Die Beschreibung erfolgt auf der Ebene von Komponenten, Schnittstellen und Strukturmodellen.

5.1 Übersicht

Die Abbildung 5.1 zeigt das Schichtenmodell einer Ausführungsumgebung für mobile Agenten nach [CHK94]. In der Ebene der Anwendungssysteme befinden sich die Applikationen, die einem Endbenutzer zur Verfügung stehen. Von der dort vorgenommenen Unterscheidung in Klienten und Dienstleister (engl. *client/server*) ist in dieser Darstellung abstrahiert worden. In der Regel ist diese auch willkürlich, da sich auf Rechnersystemen der realen Welt häufig eine Vielzahl unterschiedlicher Dienste und Dienstnutzer befinden, oder eine Applikation sowohl einen Dienst erbringt als auch die Dienste anderer Applikationen nutzt. Im Hinblick auf die im Abschnitt 2.3 geschilderten "offenen Dienstnutzungsumgebungen", die allgemeiner von kooperierenden Komponenten in wechselnden Rollen ausgeht, ist eine strikte Trennung von Klient und Dienstleister zu strikt.

Die Komponenten einer offenen Dienstnutzungsumgebung befinden sich in der Schicht der Anwendungssysteme. Eine Anwendung, wie sie sich dem Endbenutzer darstellt, besteht aus einem Zusammenschluß dieser unabhängigen Komponenten, die durch Kooperation untereinander einen Dienst für den Endbenutzer erbringen. Zur Interaktion mit einem menschlichen Endbenutzer[1] sind in dieser Schicht auch generische Visualisierungsdienste angesiedelt, die eine für Menschen geeignete Präsentation der Dienste bereitstellen.

Die Menge der tatsächlich vorhandenen Anwendungen ist nicht vorhersagbar. Die Population variiert je nach Endbenutzer und Einsatzgebiet des Rechnersystems. Eine homogene Anwendungsstruktur kann daher auf dieser Ebene nicht vorausgesetzt werden. Vielmehr muß es Ziel sein, die verschiedenen autonomen Anwendungen im Sinne einer Kooperation zu integrieren[2].

Die Schicht des Agentensystems leistet dies. Im Gegensatz zur Anwendungssystemschicht ist das Agentensystem auf jedem Rechnerknoten einer auf Agenten basierenden, offenen Dienstnutzungsumgebung vorhanden. Die Anwendungsprogrammierschnittstellen (engl. *application programming interface* (API)) des Agentensystems wird von den in der Anwendungssystemschicht befindlichen Applikationen verwendet, um die erforderliche Kooperation mit anderen Komponenten ihrer Schicht zu erreichen.

[1] Wie bereits im Abschnitt 2.4 dargelegt, können auch nicht-menschliche Akteure als Endbenutzer angesehen werden.

[2] Eine Motivation hierfür findet sich ebenfalls in Abschnitt 2.4

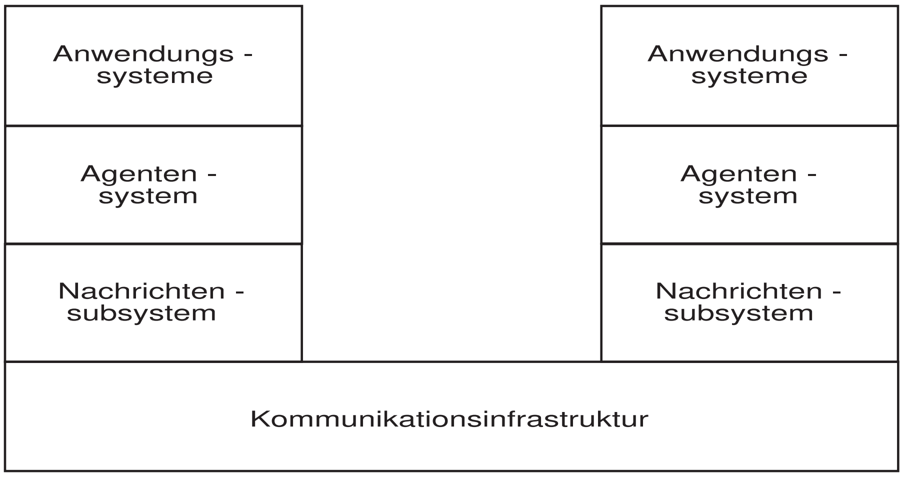

Abbildung 5.1: Modell einer Ausführungsumgebung für mobile Agenten

Auf die Dienste bzw. Komponenten des Agentensystems und des Nachrichtensubsystems wird in den folgenden Abschnitten genauer eingegangen. Die Kommunikationsinfrastruktur wird nicht näher betrachtet, da mit dem Nachrichtensubsystem eine Abstraktionsschicht über dieser liegt. Hier können ggf. erforderliche Anpassungen transparent vorgenommen[3] werden.

5.2 Agentensystem

Die in diesem Abschnitt genannten Architekturmerkmale basieren wesentlich auf Erfahrungen mit dem Telescript-System, Arbeiten von David Chess, et al. [CGH+95] und Arbeiten im Rahmen der Dissertation von Bernd Mathiske [Mat96].

Diese Architekturmerkmale sind im einzelnen:

- Anwendungsnahe, migrationsorientierte Strukturierung des Netzes für agentenbasierte Dienste.
- Anwendungsnahe Kooperationsformen und Protokolle.
- Basisklassen zur Erzeugung von Agenten.
- Migrationsunterstützung

5.2.1 Strukturierungsmittel

Eine wesentliche Leistung des Agentensystems ist die Strukturierung der "Welt", in der sich die mobilen Agenten bewegen. Da Migration von Agenten ein Kernelement des Agentensystem ist, d.h. ein Agent seinen Aufenthaltsort aktiv bestimmen kann, wurde eine räumliche Metapher zur Strukturierung gewählt. Diese soll helfen, den Umgang mit der Migration als sprachliches Mittel auf Anwendungsebene zu erleichtern.

Das Agentensystem realisiert die Konzepte *Domäne* und *Ort*. Eine Domäne ist einem Knoten des Rechnernetzes zugeordnet und verfügt über einen eigenen Kommunikationsendpunkt innerhalb des Netzes. Eine Domäne kann ein beliebiges Rechnersystem sein, das einen Komponentenstapel

[3] Die in dieser Arbeit vorgenommene Implementation des Agentensystem benutzt als Anbindung an die Kommunikationsinfrastruktur *Sockets*.

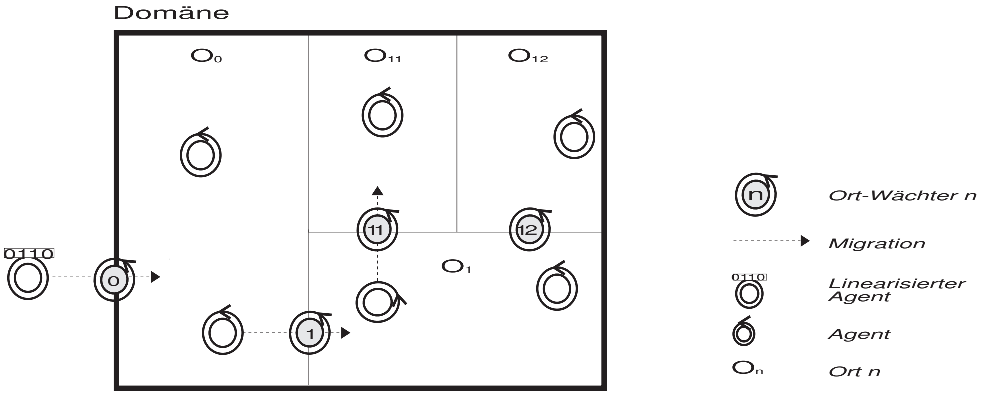

Abbildung 5.2: Logische Struktur der Agentenwelt

aus Abbildung 5.1 beherbergt. Jede Domäne ist innerhalb des Netzwerkes eindeutig adressierbar und besteht immer aus mindestens einem Ort, welcher die Wurzel der lokalen Ortehierarchie eine Domäne bildet.

In Abbildung 5.2 ist die vom Agentensystem geschaffene logische Struktur einer Domäne dargestellt. Orte sind untereinander hierarisch angeordnet. Ein Ort kann eine beliebige Anzahl anderer Orte und Agenten beherbergen. Dabei ist jeder Ort innerhalb seiner Domäne[4] eindeutig benannt.

Orte bilden dynamische Sichtbarkeitsbereiche für Agenten und enthaltene Orte. Jeder Ort wird durch einen spezialisierten Agenten repräsentiert, einen sog. Orte-Wächter. Dieser kann für jeden Agenten, der den Ort betreten möchte, entscheiden, ob dieser in den Sichtbarkeitsbereich des Ortes eindringen darf oder nicht. Lehnt ein Orte-Wächter den Zutritt ab, wird die Migration des betroffenen Agenten nicht durchgeführt.

Ein Agent kann nur mit Agenten in Kontakt treten, die sich mit ihm am selben Ort befinden. Ein Ort (bzw. dessen Orte-Wächter) kann aber sowohl von innerhalb des Ortes, als auch von außerhalb erreicht werden. Sie befinden sich daher in Abbildung 5.2 auf der Grenze zweier Orte.

Jeder Ortswechsel, ob domänenübergreifend oder lokal, geschieht mittels einer *Migration*. Der Wechsel zu einem anderen Ort ist nur dann möglich, wenn der migrierende Agent eine gültige Reiseroute zum gewünschten Zielort angibt. Eine solche Route besteht aus der Netzadresse der Zieldomäne und der Namen aller Orte, die sich auf dem Pfad zum Zielort befinden. Im Falle einer lokalen Migration ist eine ortsrelative Adressierung ausreichend.

Die Kontrolle der Migration übernimmt eine Komponente des Agentensystems, die *Migrationssteuerung*. Sie stellt sicher, daß jeder betroffene Orte-Wächter über den Migrationswunsch informiert wird und den Eintritt gewährt.

Durch diesen Mechanismus wird erreicht, daß jeder Ort über die an ihm befindlichen Orte und Agenten informiert ist. Das Erzeugen eines neuen Agenten oder Ortes ist ebenfalls zustimmungspflichtig durch den betroffenen Ort.

Orte können, genau wie Agenten, beliebig zur Laufzeit erzeugt und wieder vernichtet werden[5].

Mit den genannten Eigenschaften bilden Orte die Grundlage der anwendungsorientierten Strukturierung einer Domäne. Die Repräsentation eines Ortes durch einen aktiven Orte-Wächter erlaubt die Nutzung der selben Systemmechanismen für Orte wie für Agenten. Allerdings sind Orte im Gegensatz zu den Agenten immobil. Diese Einschränkung ist durch die räumliche Metapher motiviert, denn in der realen Welt sind Orte ebenfalls fixiert. Eine Anwendung kann wählen, ob

[4] Dies ist eine Implementationseinschränkung, die bei Bedarf aufgehoben werden kann.
[5] Die konkrete Implementation des Agentensystems verhindert die Vernichtung eines Ortes, sofern dieser nicht leer ist.

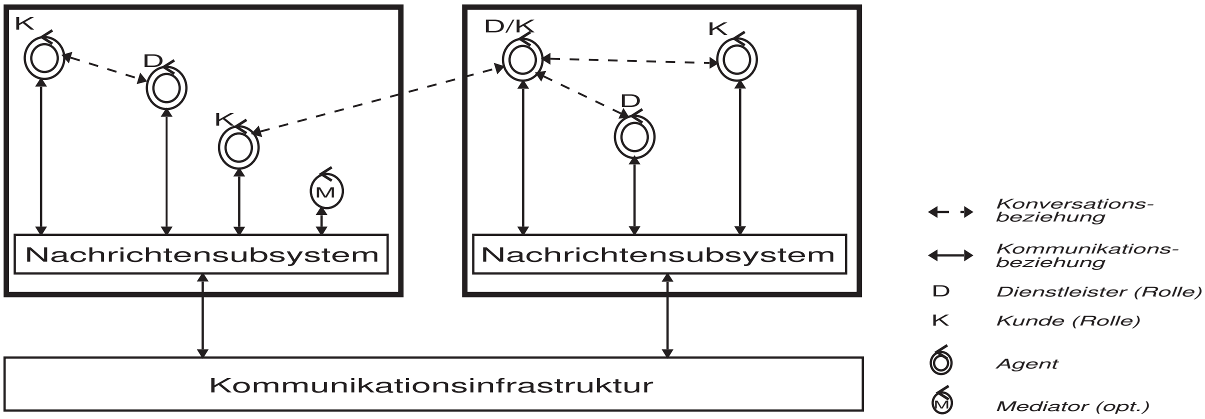

Abbildung 5.3: Interaktionsbeziehungen zwischen Agenten

eine Komponente durch einen Agenten oder Ort implementiert wird. Das Kapitel 7, welches das Programmiermodell für mobile Agenten vorstellt, zeigt dies anhand eines Beispiels.

5.2.2 Kooperation

Auch die Kooperationsformen, die eine Interaktion autonomer Agenten ermöglichen, sollen anwendungsnah sein. Um dies zu erreichen, wird das im Kapitel 3 vorgestellte Kooperationsmodell der *Business Conversations* in der Umgebung für mobile Agenten eingesetzt. Da diese medienunabhängig definiert sind und von einer maximalen Autonomie der Kommunikationspartner ausgehen, eignen sie sich für das Agentensystem.

Agenten[6] kooperieren mit anderen Agenten, indem sie mit ihnen *Konversationen* führen. Die dadurch entstehenden Beziehungen zwischen Agenten sind in Abbildung 5.3 dargestellt.

In einer Konversationsbeziehung ist es notwendig, daß ein Partner die Rolle des Kunden einnimmt und der andere Partner die des Dienstleisters (in den Abbildungen mit K und D gekennzeichnet). Agenten können diese Rollen je nach zu erfüllender Aufgabe einnehmen. Die Rollen gelten nur für die jeweilige Konversation mit einem Agenten.

Jeder Agent kann gleichzeitig mehrere Konversationsbeziehungen zu anderen Agenten aufrechterhalten. Die zeitliche Ausdehnung einer solchen Beziehung ist nur durch die Lebensdauer der beteiligten Agenten beschränkt. Diese Eigenschaft ist im Hinblick auf Anwendungen aus dem Bereich der computerunterstützten Gruppenarbeit (CSCW) hilfreich, da hier häufig lang andauernde Beziehungen zwischen den Akteuren eines Arbeitsprozesses vorkommen [Jab95]. Diese Beziehungen können direkt durch das Agentensystem modelliert werden.

Eine Konversation zwischen Agenten erfordert keine verbindungsorientierte Kommunikation der Partner. Stattdessen repräsentiert eine Konversation eine virtuelle Verbindung zu einem Agenten, die über das Nachrichtensubsystem (siehe Abschnitt 5.3) abgebildet wird.

Die Kooperation nach dem *Business Conversation*-Modell besitzt eine Reihe von Vorteilen gegenüber der direkten Objektinteraktion, die bei den Agentensystemen Mole und Telescript verwendet wird:

- ▷ Die Konversationsbeziehungen sind autonomiebewahrend. Durch den Austausch von definierten Formularobjekten werden keine privaten Objektbindungen eines Agenten dem Konversationspartner zugänglich gemacht. Daher kann auf besondere Bindungsarten für Objekte, wie z.B. die Ownership-Referenzen [Whi94, GMI95a], verzichtet werden.

[6] Die Aussagen in Bezug auf Agenten gelten in diesem Abschnitt auch für Orte (bzw. Orte-Wächter).

▷ Die Konversationsbeziehungen sind migrationssicher. Da in einer Kooperation nur Bindungen an Objekte des BC-Modells entstehen, bleibt auch die *faktische* Mobilität erhalten [Mat96]. Sind beliebige Objektinteraktionen erlaubt, kann ein Agent Bindungen an Objekte erhalten, die aus dem Anwendungskontext betrachtet nicht mobil sind. Dies trifft insbesondere auf Massendaten zu, die aufgrund ihres Volumens nicht über ein Netzwerk transportiert werden sollten[7].

▷ Das Modell besitzt ein definiertes Verarbeitungsmodell. Anders als bei der direkten Objektinteraktion, wird für jede Kooperation beliebiger Agenten die erforderliche Koordination vom Agentensystem vorgenommen. Daher ist keine weitere Synchronisation beim geteilten Zugriff auf Ressourcen notwendig, die durch Agenten realisiert sind. Der Anwendungsprogrammierer wird auf diese Weise von den Problemen der ausgeprägten Nebenläufigkeit im Agentensystem abgeschirmt.

▷ Das Modell ist medienunabhängig. Bei einer Beschränkung der Kooperation auf Konversationen ist ein Agent in der Lage auch zukünftige, nicht agentenbasierte Systeme über seine Kooperationsmechanismen zu bedienen. Insbesondere können klassische objektbasierte Verteilungsmechanismen, wie sie z.B. in CORBA [Gro91] vorkommen, auf das Konversationsmodell abgebildet werden.

▷ Die Kooperationsbeziehungen sind transparent. Die einzig mögliche Einflußnahme eines Agenten auf einen anderen Agenten, ist durch die Benutzung der Kooperationsschnittstelle des Agentensystems. Da aber die dafür erforderliche Kommunikation über ein gesondertes Kommunikationssubsystem abgewickelt wird, ist der vollständige Nachrichtenfluß an einer zentralen Stelle abgreifbar. Damit ist er anderen Diensten wie Logging, Tracing, Authentisierung, ...zugänglich[8].

▷ Konversationen unterliegen einer Prozeßspezifikation. Jede Konversation ist über ein zugehöriges Spezifikationsobjekt beschrieben. Diese Spezifikation liegt zur Laufzeit den beteiligten Agenten vor und wird am Beginn einer Konversation dem Partner mitgeteilt. Dies ermöglicht den Abgleich des potentiellen Gesprächsverlaufes durch die Agenten, bevor eine Konversation aufgenommen wird.

Diesen Vorteilen steht der Nachteil gegenüber, daß ein Agent seine zu kommunizierenden Anwendungsdatenstrukturen in die vom *Business Conversation*-Modell verlangten Formularobjekte konvertieren muß. Da aber häufig nur ein kleiner Teil des Zustandes eines Agenten einem anderen Agenten mitgeteilt werden muß, ist dieser Aufwand, gemessen an den Vorteilen des Modells, vertretbar.

In Abbildung 5.4 findet sich eine detailliertere Darstellung der Konversationsbeziehungen zwischen Agenten.

Sie zeigt den Zusammenhang von Konversationen und Konversationsspezifikationen, der durch das Agentenmodell erreicht wird.

Jede Rolle eines Agenten erfordert eine zugehörige Konversationsspezifikation. Diese wird bei einer Konversationsbeziehung zu einem anderen Agenten instantiiert und wirkt dabei als Generator für die Formularobjekte, die im Verlauf der Konversation zwischen den Partnern ausgetauscht werden.

Konversationen werden deklarativ über *ON*-Regeln programmiert. Daraus resultiert eine Aufteilung des Programmflusses in Ereignis/Aktions-Paare, die in einer Regelbasis des Agenten gespeichert wird. Diese strikte Strukturierung in Bearbeitungsregeln bietet weiter die Möglichkeit, eine Fehlererholung mit kompensierenden Aktionen [GMS87, DHL90] für jede Konversation anzugeben. Dabei kann jede solche *inverse*-Bearbeitungsregel die lokale Gesprächshistorie der aktuellen Konversation verwenden.

---

[7] Zum Beispiel ist der Versicherungsvertragsbestand einer Lebenversicherung *faktisch* immobil.

[8] Siehe auch der Mediator im Abschnitt 5.3

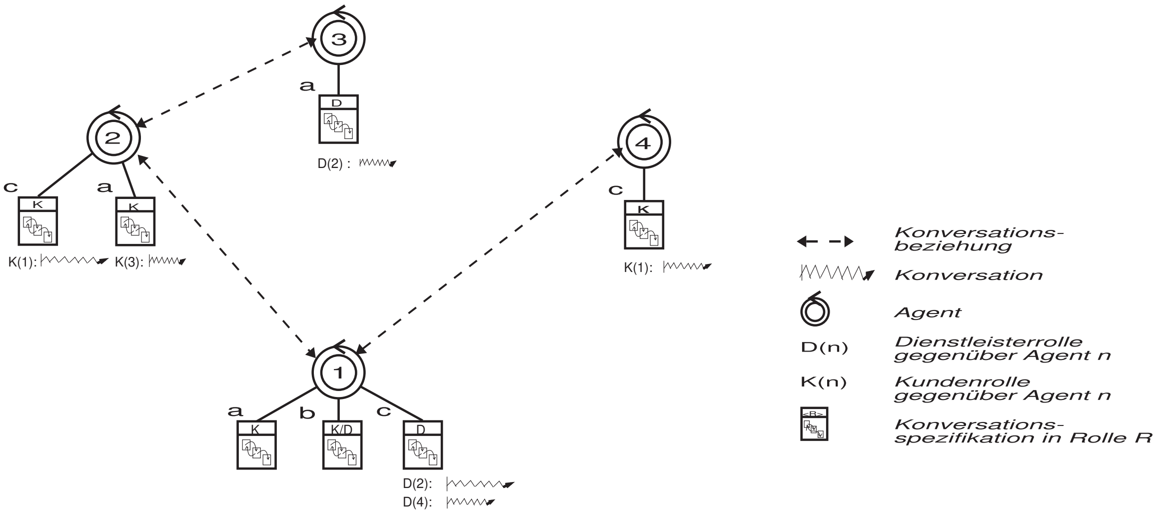

Abbildung 5.4: Konversationen zwischen Agenten

Ein Agent ist in der Lage, beliebig viele Konversationen gleichzeitig zu führen, wobei allerdings die Exekution einer Bearbeitungsregel eines Formulars, wie durch das *Business Conversation*-Modell verlangt, atomar geschieht. Jede Konversation bildet damit eine Art eigener *Thread*, einen *Gesprächs*faden, mit einem eigenen Kontext. Die Umschaltpunkte sind jeweils das Ende einer Bearbeitungsregel. Das Kontextmanagement, d.h. die Zuordnung des korrekten Kontextes zur jeweiligen Regel, leistet das Agentenskelett (siehe Abschnitt 5.2.3).

Bei der Migration eines Agenten an einen neuen Ort oder wenn dieser eine sekundäre Konversation beginnt, wird die atomare Ausführung der umgebenden Bearbeitungsregel aufgehoben. Im Falle der Migration ist dies erforderlich, um den Agenten durch den Migrationsdienst linearisieren und verschicken zu können. Hierbei wird sichergestellt, daß keine andere Bearbeitungsregel aktiviert wird, solange die erste nicht beendet ist. Dies bedeutet, das eine Migration aus der Sicht der initiierenden Bearbeitungsregel atomar ausgeführt wird, aus der Sicht des Agenten jedoch die Regel unterbricht.

Die sekundäre Konversation mit einem anderen Agenten verlangt, daß diese Konversation zuerst beendet wird, bevor die auslösende Konversation weitergeführt wird. Dieses synchrone Verfahren entspricht in seiner Verarbeitung dem Prozeduraufruf in einer Programmiersprache. Hierbei wird die Ausführung der initiierenden Bearbeitungsregel ebenfalls unterbrochen, jedoch können andere Bearbeitungsregeln in dieser Unterbrechung ausgeführt werden.

Der durch das Agentensystem erzeugte Nachrichtenfluß einer Konversation ist in Abbildung 5.5 veranschaulicht.

In dieser Darstellung sind die beiden Ebenen Kommunikation und Kooperation zu erkennen. Auf der Kommunikationsebene werden zwischen den Partnern systemdefinierte Nachrichten ausgetauscht. Mit diesen Nachrichten wird die Kooperation auf der Basis anwendungsorientierter Formular- und Spezifikationsobjekte realisiert.

Durch das *Business Conversation*-Modell ist der erforderliche Interaktionsablauf der Kooperation zwischen einem Kunden und seinem Dienstleister vorgegeben. Das Model trifft jedoch keine Festlegung darüber, wie ein Kunde einen geeigneten Dienstleister *ermitteln* kann. Dieses Problem, das sich auch unter der engl. Bezeichnung *trading* oder *brokering* im Zusammenhang mit der Client/Server-Programmierung findet [Lam94, PSW96], wird vom Modell nicht eingeschränkt. Das Agentensystem bietet für diesen Bereich lediglich eine Kernunterstützung, die es Agenten erlaubt, Konversationen mit anderen Agenten zu beginnen. Zu diesem Zweck ist es notwendig, daß ein Agent den Namen des anderen kennt.

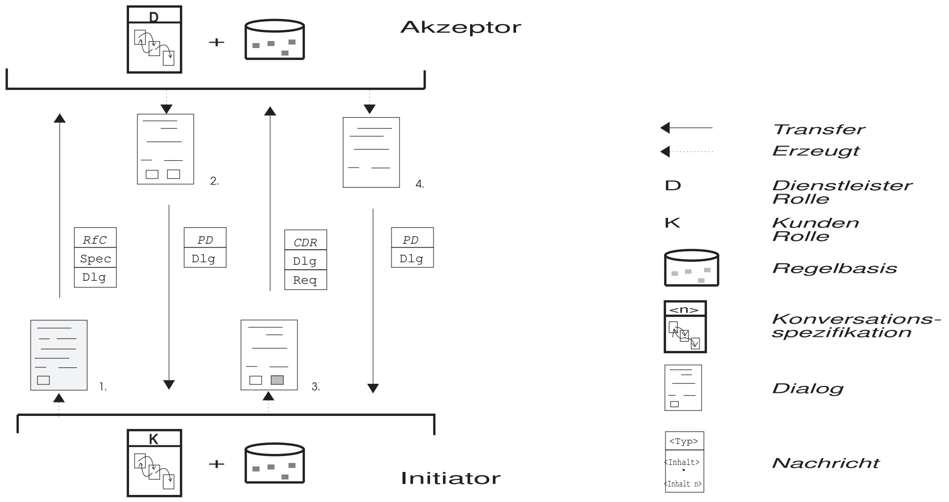

Abbildung 5.5: Nachrichtenfluß einer Konversation

Das Agentensystem verwendet die sich aus der hierarchischen Ortestruktur ergebenden Sichtbarkeitsbereiche zur Bindung der Agenten über deren Namen.

Dabei stehen folgende Varianten der Kontaktaufnahme zur Verfügung:

1. Rendezvous ohne Gesprächsbeginn.
2. Synchroner Gesprächsbeginn.
3. Asynchroner Gesprächsbeginn.

Im ersten Fall, dem Rendezvous zweier Agenten ohne einem Gesprächsbeginn, erhält der Initiator des Treffens ledigliche eine Referenz auf einen anderen Agenten. Dieses Treffen (engl. *meeting*) ist das Basisprotokoll der Kontaktaufnahme zweier Agenten und damit die Grundlage der beiden anderen Varianten.

Ein wichtiger Aspekt des Treffens ist, daß es sich um ein durch das Agentensystem moderierten Rendezvous handelt. Der Initiatoragent teilt seinen Kontaktwunsch dem Agentensystem mit, das die weitere Abwicklung des erforderlichen Protokolls[9] durchführt. Das Agentensystem ermittelt einen geeigneten Agenten innerhalb dessen aktuellen Sichtbarkeitsbereiches und leitet den Kontaktwunsch weiter. Dieser Agent, der Akzeptor, kann den Wunsch annehmen, ablehnen oder einen dritten Agenten für das Treffen vorschlagen.

Nimmt der Agent den Kontaktwunsch an, stellt das Agentensystem dem Initiator eine Referenz auf den Akzeptor zur Verfügung. Das Agentensystem bewahrt dabei die Autonomie der Partner, indem es nur künstliche Objektreferenzen[10] austauscht. Echte Objektreferenzen auf Agenten sind nur innerhalb der Schicht des Agentensystems selber vorhanden und werden unter keinen Umständen anderen Agenten sichtbar.

Lehnt ein Agent den Kontaktwunsch eines anderen Agenten ab, so wird dieser darüber vom Agentensystem informiert. Der Agent hat dann keine Möglichkeit, mit dem Akzeptor zu interagieren. Die Existenz des potentiellen Partners bleibt vor ihm verborgen.

Wenn der Akzeptor einen dritten Agenten vorschlägt, wird das Treffen *delegiert*. Dieser Mechanismus erlaubt, einfache Broker-Dienste in normale Agenten zu integrieren. Das Agentensystem

[9] Einzelheiten über die Abwicklung des Protokolls finden sich im Kapitel 9.
[10] Dies können zum Beispiel automatisch erzeugte, weltweit eindeutige Bezeichner sein.

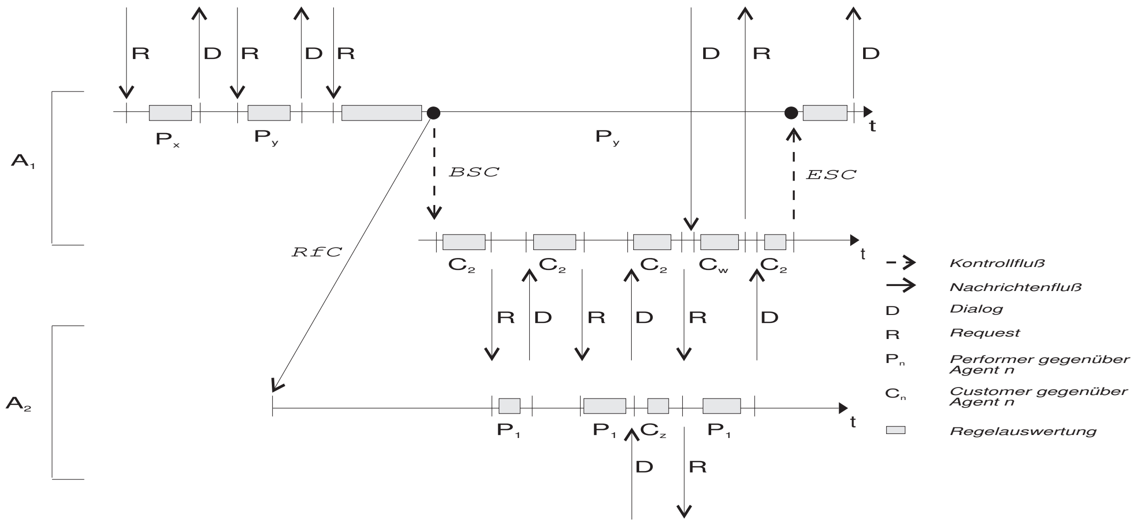

Abbildung 5.6: Regelauswertung innerhalb von Agenten

versucht in einem solchen Fall, und für den Initiator transparent, ein Treffen mit dem vorgeschlagenen Agenten zu arrangieren. Dabei wird das Protokoll iterativ auf den neuen Agenten angewendet.

Wird ein Treffen erfolgreich etabliert, so kann die erzeugte Referenz auf den Partneragent verwendet werden, eine Konversation mit diesem zu beginnen. Im Gegensatz zu dem in Telescript oder Mole verwendeten Rendezvousprotokoll ist eine Migration der Partner auch nach Herstellung eines Treffens erlaubt. Dies ist möglich, da die Agenten keine Objektreferenzen aufeinander erhalten sondern indirekte Identifikatoren verwenden, die über das Agentensystem verwaltet werden. Auf diese Weise bleiben die Objektgraphen der Agenten voneinander isoliert und ihre Autonomie erhalten.

Der synchrone und der asynchrone Gesprächbeginn unterscheiden sich nur im Grad der Nebenläufigkeit innerhalb eines Agenten. Ein synchron begonnenes Gespräch muß, im Unterschied zum asynchronen, erst vollständig beendet sein, bevor eine andere Konversation initiiert werden kann. Dies entspricht einer *sekundären* Konversation, die ebenfalls im Verarbeitungsmodell für *Business Conversations* definiert ist.

Im asynchronen Fall ist eine nebenläufige Gesprächsführung mit mehreren Partnern erlaubt. Dabei bleibt die Unteilbarkeit der Regelauswertungen erhalten, lediglich die Konversationen als Ganzes überlappen sich.

Ein Agent, der ein Gespräch mit einem anderen Agenten beginnt, nimmt automatisch für dieses Gespräch die Rolle eines Kunden ein. Komplementär dazu erhält der Akzeptor die Rolle des Dienstleisters. Die durch die unterschiedliche Gesprächsarten implizierten Interaktionsfolgen sind in Abbildung 5.6 wiedergegeben.

5.2.3 Agentengeneratoren

Ein Entwurfsziel des Agentensystems ist, die Anwendungsentwicklung nicht durch die Programmierung von Agenten zu behindern. Agenten sollen daher nicht vom Anwendungsprogrammierer an das Agentensystem angebunden werden. Vielmehr sind Generatoren[11] oder generische Bibliotheken sinnvoll, die es gestatten, spezialisierte Agenten durch Parametrierung eines allgemeinen Musters zu erzeugen.

[11] Z.B. implementiert durch Klassen einer objektorientieren Implementierungssprache

Diese Agentengeneratoren sind Bestandteil der Agentenumgebung, da sie auf die verwendeten Interaktionsprotokolle des Agentensystems ausgelegt sein müssen.

Die Parametrierung erfolgt gemäß der durch das Agentensystem implementierten Kooperations- und Interaktionsprotokolle. Immer dann, wenn der Agent von Außen, d.h. vom System beeinflußt wird, ist daß Agentenmuster zu spezialisieren. Für die bereits geschilderten Leistungen sind dies:

- ▷ Die Erzeugung bzw. Beendigung eines Agenten oder Ortes.
- ▷ Die Nachfrage eines fremden Agenten oder Ortes nach einem Treffen.
- ▷ Nur bei Agenten: Eintritt in einen neuen Ort bzw. Verlassen eines Ortes.
- ▷ Nur bei Orten: Eintritt eines neuen Agenten bzw. Verlassen eines Agenten.

An diesen Punkten der vom Agentensystem implementierten Protokolle kann der Anwendungsprogrammierer das Agentenmuster ändern. In Kapitel 7 wird gezeigt, wie dies in einem konkreten Fall aussehen kann.

Eine weitere Parametrierung im Sinne einer Konfiguration des Agenten erfolgt zur Laufzeit. Hier kann der Agent seine Regelbasis zur Bearbeitung von Konversationen mit anderen Agenten aufbauen und so seine kooperativen Fähigkeiten über die Zeit verändern. Nur wenn der Agent sowohl Konversationsspezifikationen als auch Bearbeitungsregeln für diese aufweist, kann er mit andern Agenten kooperieren. Ein Agent ohne Regelbasis für Konversationen kann lediglich über das Meeting-Protokoll wirken. Hier kann er Nachfragen bezüglich Treffen an andere Agenten weiterleiten. Orte haben zusätzlich die Möglichkeit, die Migration fremder Agenten in ihren Sichtbarkeitsbereich abzulehnen oder zu gestatten.

5.2.4 Migrationsunterstützung

Das Agentensystem hat weiter die Aufgabe, die Reise eines Agenten an einen gewünschten Zielort durchzuführen.

Bei einer Reise kann ein Agent innerhalb einer Domäne bleiben oder aber diese verlassen. Das Agentensystem sorgt in beiden Fällen für die Abwicklung des erforderlichen Migrationsprotokolls. Ist der Zielort in der gleichen Domäne angesiedelt, so werden nur die internen Datenstrukturen des lokalen Agentensystems angepaßt. Ist ein Ort einer anderen Domäne das Reiseziel, so muß das Agentensystem den Transport des Agenten über ein Rechnernetz durchführen. Zu diesem Zweck ist eine Konvertierung des Agenten in eine linearisierte Darstellung erforderlich. Dabei müssen die Objektreferenzen, die ein Agent an das lokale Agentensystem hat, getrennt werden. Diese Referenzen können dann nach der Migration an das in der Zieldomäne vorhandene Agentensystem gebunden werden, um dem Agenten eine Interaktion mit diesem zu ermöglichen[12].

Die Migration in eine fremde Domäne muß transaktional erfolgen, da sonst im Falle von Übertragungsfehlern Dubletten eines Agenten auftreten können. Auch besteht sonst die Gefahr das ein Agent bei der Übertragung verloren geht [Mat96, GMI95a]. Diese Gefahr erhöht sich, wenn eine Migration über ein Weitverkehrsnetz (WAN) erforderlich wird.

Eine weitere Aufgabe, die den Einsatz des Agentensystems bei einer transienten Verbindung zum Netz ermöglichen, ist die Verzögerung der Abreise eines Agenten. Dabei werden alle Agenten, die die Domäne verlassen wollen, gesammelt und bei der nächsten Verbindung zum Rechnernetz in einem Stapelverfahren verschickt. Auf der anderen Seite können zu diesem Zeitpunkt auch Agenten von einem vereinbarten "Parkplatz" abgeholt werden. Dies ist eine Domäne, die eine permanenten Verbindung zum Rechnernetz besitzt und von Agenten immer dann angelaufen werden kann, wenn die gewünschte Zieldomäne zum Migrationszeitpunkt nicht erreichbar ist.

[12] Dies ist immer möglich, da das Agentensystem für alle Zielsysteme ubiquitär ist.

5.3 Nachrichtensubsystem

Das Nachrichtensubsystem stellt die Grundlage der Kommunikation aller Systemkomponenten dar. Es hat die Aufgabe, das Agentensystem von der auf dem Rechnerknoten eingesetzten Kommunikationsinfrastruktur bzw. der konkreten Netztopologie abzukoppeln. Auf diese Weise können beliebige Middleware-Systeme zur Realisierung der erforderlichen Netzwerkkommunikation eingesetzt werden und das Agentensystem von Details der Netzwerkprogrammierung abgeschirmt werden. Im Hinblick darauf, daß Agenten sowohl lokal miteinander kommunizieren können als auch mit entfernten Agenten anderer Knoten Konversationsbeziehungen unterhalten, bietet das Nachrichtensubsystem ortstransparente Kommunikationsformen an.

Ein weiteres Erfordernis aus der Sicht des Agentensystems ist die Unterstützung der Mobilität der Klienten des Nachrichtensystems. Dieses muß gewährleisten, daß die Mobilität seiner Klienten während der Nutzung des Systems erhalten bleibt. Diese Mobilität bedingt weiter, daß das Nachrichtensystem nicht von einer permanenten Erreichbarkeit seiner Klienten ausgehen kann und schließt daher verbindungsorientierte Kommunikationsformen aus.

Da Agenten und Orte durch autonome Aktivitäten repräsentiert werden und diese daher voneinander unabhängig agieren, stellt ein zentrales Nachrichtensystem einen potentiellen Flaschenhals des Gesamtsystems dar. Es ist daher für den Durchsatz des Systems förderlich, ein derartiges zentrales System nur kurze Zeit durch seine Klienten zu blockieren. Aus diesem Grunde verwendet das System asynchrone, nachrichtenorientierte Verfahren als Basis der Kommunikation zwischen seinen Klienten. Dabei wird jede Nachricht eines Klienten an das Nachrichtensystem weitergegeben und gilt damit für diesen als versendet. Dem Nachrichtensystem obligt die Sicherung der tatsächlichen Zustellung einer Nachricht, wobei der Sender mit dem System vereinbaren kann, mit welcher Qualität diese versendet werden soll.

Auf der Seite des Nachrichtensystems erfordert dies eine die Reihenfolge erhaltende Zwischenspeicherung der Nachrichten in Nachrichtenwarteschlangen (engl. *message queue*) und ein store-and-forward Protokoll für deren Versand [KV89]. Diese Nachrichtenwarteschlangen werden vom Nachrichtensubsystem verwaltet und gewährleisten die korrekte Zustellung einer Nachricht, auch wenn die Zielapplikation zeitweilig nicht erreichbar ist [MSM95].

Ein weiterer Vorteil dieser Kommunikationsform ist, daß sie die einfachste (im Vergleich zu RPC oder ROI) und zugleich flexibelste Form darstellt. Von eine Applikation versendete Nachrichten können bei Bedarf beliebig umgeleitet, kopiert oder verändert werden. Ein Einsatzgebiet für diese Flexibilität ist das Überwachen des Nachrichtenflusses (engl. *monitoring*) oder externe Router, die Nachrichten an andere Zielsysteme weiterleiten. Das dem Agentensystem zugrundeliegende Nachrichtensubsystem, dessen Implementation in Kapitel 9 beschrieben wird, sieht hierfür das Konzept sog. Mediatoren vor, die eine aktive Schnittstelle für solche Systemdienste darstellen. Dabei handelt es sich aus der Sicht des Nachrichtensubsystems um externe Dienste, die von diesem beim Nachrichtenversand gesondert berücksichtigt werden. Ein solcher Mediator erhält transparent jede verarbeitete Nachricht und kann entscheiden, ob und in welcher Form diese Nachricht an das vom Sender gewünschte Ziel weitergeleitet wird.

5.4 Anforderungen an Implementationsplattformen

Für eine effiziente Implementation des beschriebenen Agentensystems sollte die verwendete Plattform[13] Unterstützung in den Punkten Mobilität, Aktivitätsorientierung, Kommunikation und Offenheit bieten. Unter dem Gesichtspunkt der Anwendungsnähe des Agentenkonzepts ist eine ausdrucksmächtige Sprache zur Beschreibung der Objekte einer Anwendung ebenfalls wünschenswert.

[13] Die Möglichkeit, einen eigenen "Agenteninterpreter" für eine sonst ungeeignete Implementationsplattform zu schaffen wird außeracht gelassen.

Die Mobilität der Agenten, die sich auf einem weltweiten elektronischen Marktplatz der Dienste bewegen, macht eine Berücksichtigung der Heterogenität in der Systemlandschaft unumgänglich. Eine Lösung für dieses Problem bietet die Ausführung des Agentensystem auf einer virtuellen Maschine, die eine Abstraktion von der realen Systemplattform bietet [Sun95a, CGH+95, Kna95, Mat96]. Diese virtuelle Maschine bildet die Ausführungsumgebung sowohl für das Agentensystem als auch für die Agenten selbst.

Nicht nur der Programmcode der Agenten muß in einer plattformunabhängigen Repräsentation vorliegen, sondern auch die über Systemgrenzen ausgetauschten Daten (z.B. die Nachrichten und der Ausführungszustand eines Agenten) müssen systemweit einheitlich dargestellt sein. Hierfür gibt es bereits Standards, wie die ASN.1 [NV92] der ISO oder die *Network Data Representation* (NDR) im DCE [Sch93], welche dieses leisten.

Das Agentenmodell impliziert ein ausgeprägt nebenläufiges Programmierleitbild. Jeder Agent im System stellt eine eigene, autonome Aktivität dar, die ihren eigenen Kontrollfluß besitzt.

Demzufolge ist eine Systemunterstützung für nebenläufiges Programmieren unabdingbar. Die Implementationsplattform sollte daher Prozesse[14] und Synchronisationsmechanismen, wie Semaphore, bereitstellen.

Mobilität erfordert weiterhin die Möglichkeit, einen Agenten und seinen aktuellen Zustand aus dem Kontext eines Agentensystems zu isolieren und in einem anderen Kontext (d.h. einer anderen Domäne) wieder zu etablieren. Die hierfür erforderlichen Bindungstechniken stehen ebenfalls in einigen Systemen bereits zur Verfügung [Mat96, GMI95a, Kna95].

Die Einbindung bereits bestehender Anwendungssysteme (engl. *legacy systems*), in den vom Agentensystem gebildeten elektronischen Dienstemarkt ist eine wichtige Voraussetzung für den Erfolg eines solchen Systems. Daher ist die Offenheit der Implementationsplattform notwending als Grundlage der Integration dieser Altsysteme in neue, agentenbasierte Dienste.

Eine Kommunikationsunterstützung ist durch die vom Agenten- und Nachrichtensystem geschaffenen Abstraktionen nur minimal notwendig. Weder das Agentensystem, noch das Nachrichtensubsystem erfordern hier spezielle, höhere Dienste (Trader, Schnittstellen-Repositories, Object Request Broker etc.), wie sie z.B. von DCE oder CORBA [Gro91] angeboten werden. Stattdessen ermöglicht das Agentensystem auf der Anwendungsebene eigene, wiederum durch Agenten implementierte Dienste, die nicht von der jeweils verwendeten Middleware abhängig sind.

[14] Prozeß wird hier als Oberbegriff verwendet und schließt leichtgewichtige Prozesse, sog. Threads, mit ein.

Teil II

Eine Beispielanwendung

Kapitel 6

Erfassung und Analyse von Internetnutzungsstatistiken

In diesem Kapitel wird die Beispielanwendung aus WebSiteProfilern und MultiSiteAnalyzern, die die weitere Arbeit begleiten wird, vorgestellt.

Das WorldWideWeb, im Folgenden auch kurz WWW, ist ein auf dem Internet basierender, weltweiter Multimedia-Hypertextdienst.

Das WWW eignet sich als Werbeträger für Unternehmen durch die Möglichkeit der Einbettung von Text, Bildern und Ton in Hypertext-Dokumente. Diese Dokumente, im WWW-Sprachgebrauch auch *Seiten* genannt, sind durch die große internationale Verbreitung des Internets einer Vielzahl potentieller Kunden zugänglich. Dazu kommt, daß die zur Darstellung dieser Seiten benötigte Software auf allen relevanten Rechnerplattformen frei verfügbar ist. Somit ist das WWW für werbende Unternehmen ähnlich attraktiv wie es die Printmedien oder die elektronischen Medien Funk und Fernsehen bereits seit langem sind.

Die Unternehmen, die diese "klassischen" Medien produzieren, haben die Chancen des WWW als Werbeträger sehr frühzeitig erkannt und sind mit ihren Produkten präsent.

Jeder Informationsanbieter (eng. *content provider*), der Dokumente im WWW auf einem Web-Server zum Abruf bereitstellt, erhält automatisch Protokolldateien über die Zugriffe auf sein Angebot. Diese dienten ursprünglich zur Auslastungsmessung und dokumentieren sekundengenau die Zugriffe auf den Server. Damit ist es technisch möglich, die Nutzung eines Angebots zu verfolgen. Genauere Aussagen, z.B. über das Zugriffsverhalten individueller Nutzer oder die Reichweite eines Angebotes, lassen sich erst nach Aufbereitung und statistischer Analyse dieser Daten gewinnen [Kab96].

Grundsätzlich wird das Thema „Analyse von Web-Server-Nutzung“ in einer anderen Diplomarbeit [Lau97] am Arbeitsbereich DBIS behandelt.

Die Abbildung 6.1 zeigt in einer Übersicht die Zusammenhänge von Anbietern und Benutzern des WWW.

Im Zentrum stehen die sogenannten *Websites*, die Internet-Zugänge der verschiedenen Informationsanbieter. Diese können wiederum beliebig viele Angebote bereitstellen. Die Websites sind in der Abbildung mit Sternen dargestellt, die Angebote jeweils durch Seitenpiktogramme. Die Angebote werden in dieser Übersicht außerdem nach Produkten und Unternehmen unterschieden. Auch wenn diese Unterscheidung in der Realität nicht haltbar ist, soll sie hier zur Verdeutlichung des Sachverhaltes dienen.

Die Produkte (z.B. die Hypertextausführung der Zeitschrift Chip, oder die des Focus) beinhalten u.a. Anzeigen anderer Unternehmen. Das Schalten dieser Anzeigen wird von den Produktherstellern als eigene Dienstleistung gegenüber den werbenden Unternehmen erbracht.

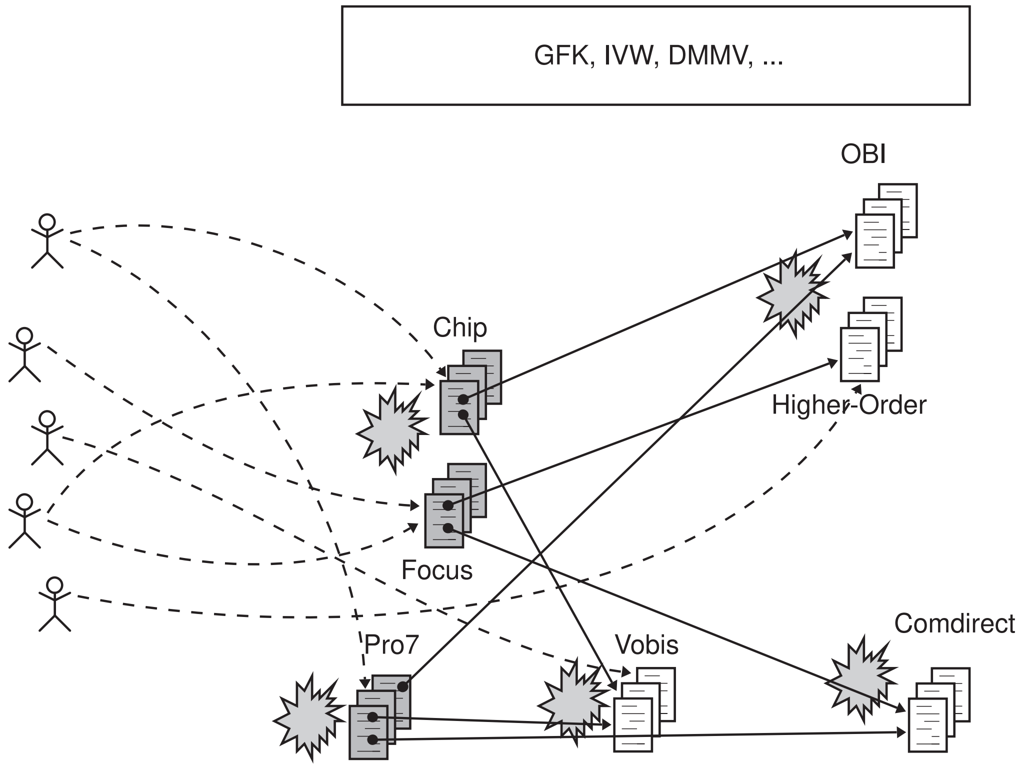

Abbildung 6.1: Der Zusammenhang von Internetnutzungsstatistiken

Im Gegensatz zu Anzeigen in Zeitschriften oder den alten elektronischen Medien, können die Anzeigen des WWW *interaktiv* gestaltet werden. Aus den Anzeigen werden dann Verbindungen (engl. *links*), mit denen der Betrachter auf Wunsch zu den zugehörigen Angeboten des werbenden Unternehmens geleitet wird.

Ein Unternehmen, welches Seiten im WWW anbietet, benötigt möglichst genaue und umfassende Statistiken über die Nutzung seines Angebotes. Diese Kennzahlen einer WebSite werden als Verkaufsargumente für die in den Angeboten vorgesehenen Werbeflächen verwendet.

Die Basis für diese Zahlen liefern Verbände, die das Mediengeschehen kontrollieren und messen, wie zum Beispiel die GfK (Gesellschaft für Konsumforschung) oder die IVW (Informationsgesellschaft über die Verbreitung von Werbeträgern). Sie nehmen ihre Aufgabe bei den „alten“ Medien wie Fernsehen und Zeitungen war, indem sie als neutrale Beobachter objektive Einschaltquoten und Verkaufsstatistiken veröffentlichen.

Diese Verbände haben ein starkes Interesse, vergleichbare Zahlen auch für das WWW zu erheben und zu verbreiten.

Der WebSiteProfiler ist das statistische Werkzeug zur Analyse der Zugriffshäufigkeiten auf die Angebote *einer* WebSite.

6.1 Der WebSiteProfiler

In diesem Abschnitt wird die Systemarchitektur des WebSiteProfilers anhand eines Übersichtsmodells vorgestellt.

Aus der Sicht eines Dienstnehmers bietet der WebSiteProfiler folgende Dienste:

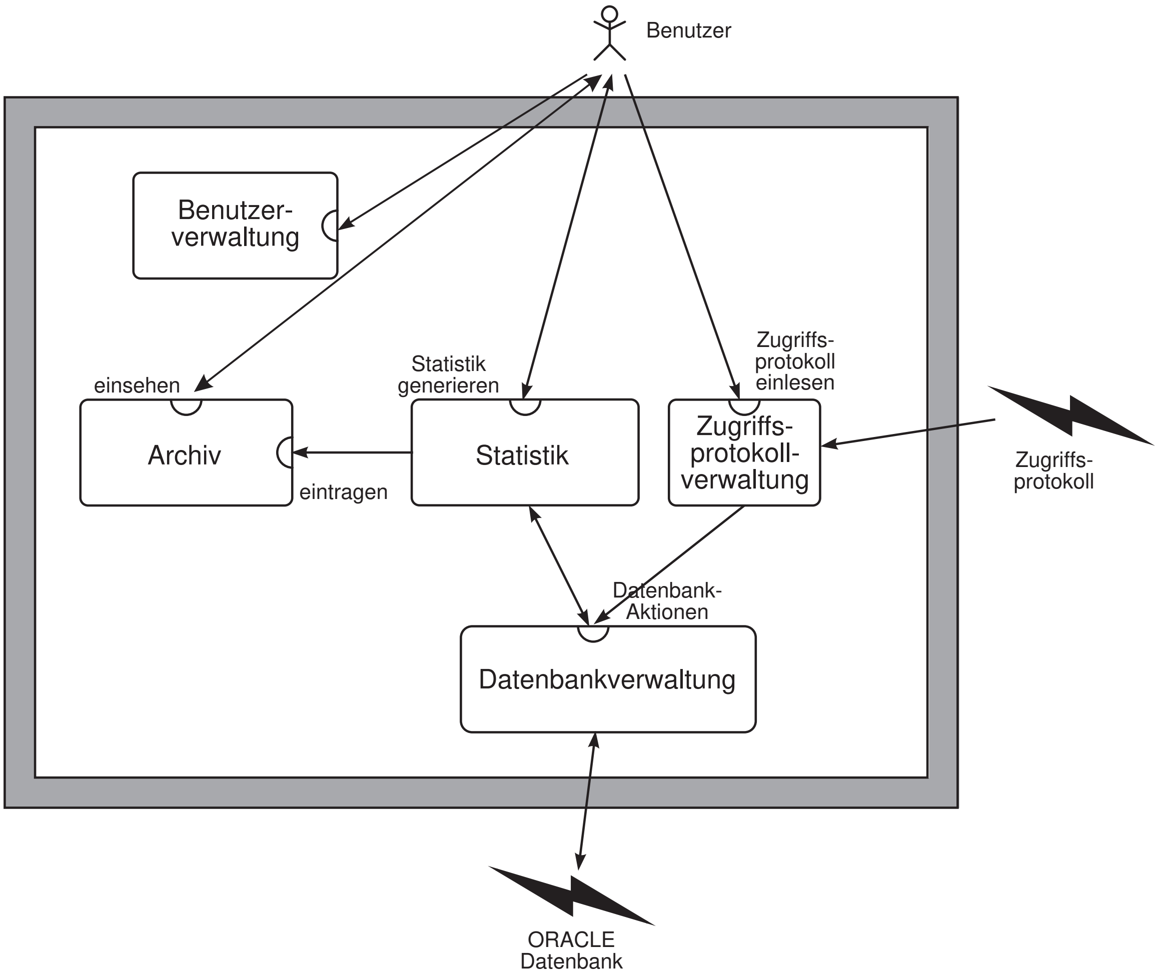

Abbildung 6.2: Übersichtsdiagramm für den WebSiteProfiler

- ▷ Erstellung von Nutzungsstatistiken über bestimmte Zeiträume hinweg.
- ▷ Aktualisierung der Datengrundlage für die statistischen Auswertungen.
- ▷ Archivierung und Bereitstellung im Archiv vorhandener Statistiken.
- ▷ Verwaltung zugelassener Benutzer.

Das Übersichtsmodell aus Abbildung 6.2 stellt die Komponenten des WebSiteProfilers und ihre Beziehungen untereinander dar. Es hält sich an die Notation für Übersichtsmodelle aus Anhang C.

Die Daten für die Nutzungsstatistiken werden aus der Zugriffsprotokolldatei gewonnen, die von jedem WWW-Server automatisch erstellt wird. Die Zugriffsprotokolldatei wird in bestimmten Abständen durch das zugehörige Verwaltungsmodul auf neue Einträge hin durchsucht, die dann in einer externen Datenbank aufgenommen werden. Das verwendete Datenbankschema wird in Abschnitt 6.1.1 vorgestellt.

Außerhalb des Systems stehen die Elemente *Datenbank, Zugriffsprotokoll* und der *Benutzer*. Die *Datenbank* wird in einem separaten konventionellen relationalen Datenbanksystem der Firma ORACLE, Inc., verwaltet.

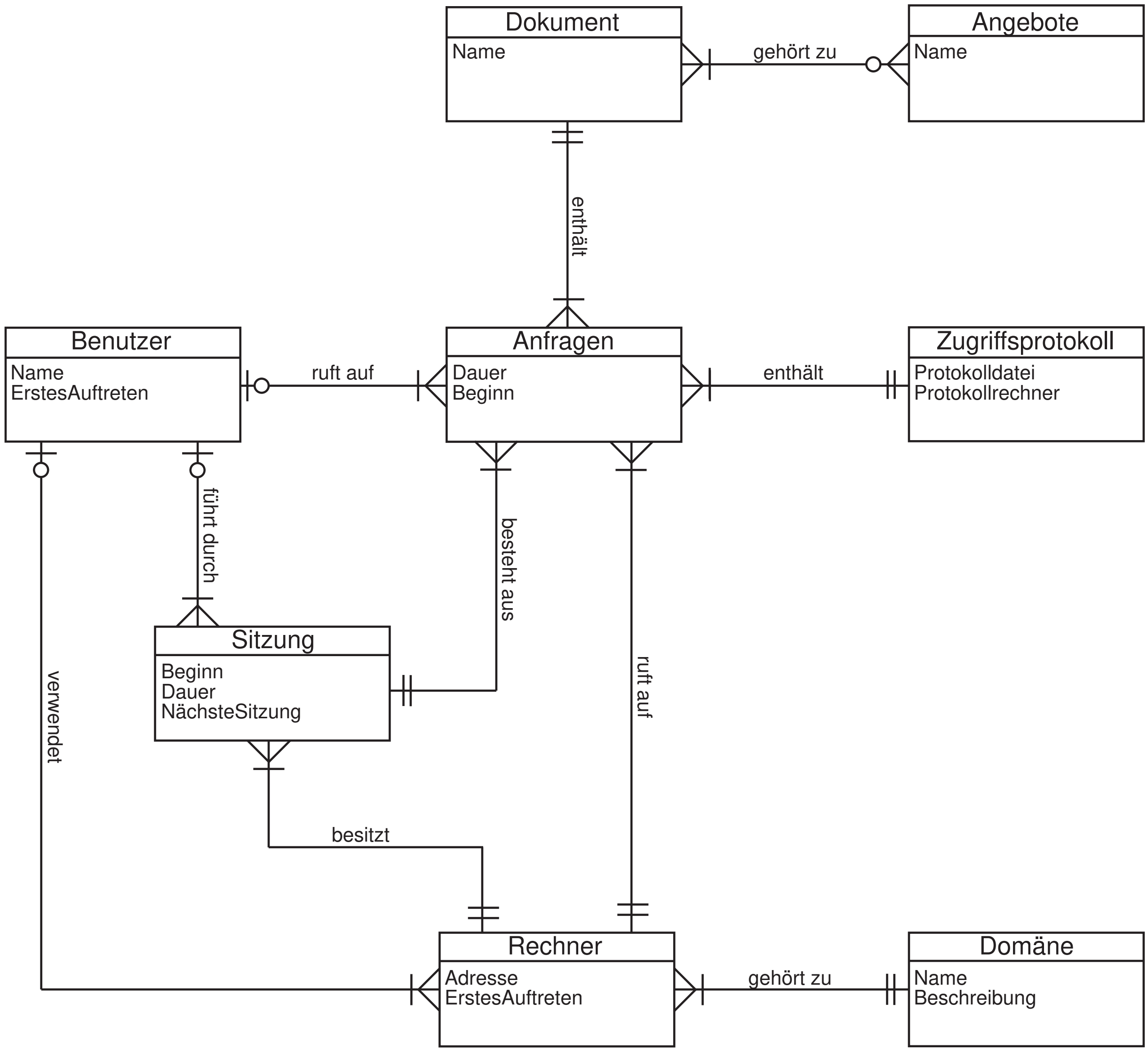

Abbildung 6.3: Objektstrukturdiagramm für die Datenbank des WebSiteProfilers

Das Statistikmodul bietet den Dienst *Statistik generieren* an. Dazu wird eine Anfrage über die Datenbankverwaltung an die Datenbank gestellt und die Antwort der Datenbank zu einer Grafik aufbereitet.

Das Modul Zugriffsprotokollverwaltung durchsucht die Zugriffsprotokolldatei nach neuen Einträgen. Diese werden entsprechend dem Datenbankschema aufbereitet und in die Datenbank eingefügt.

Im Archivmodul werden alle erstellten Statistiken in ihrer Ergebnisform gespeichert. Man kann aus dem Archiv entweder gezielt Statistiken oder eine Übersicht aller vorhandenen Einträge abrufen.

6.1.1 Datenbankschema

Die durch die Datenbank modellierte "Miniwelt" des WebSiteProfilers besteht im wesentlichen aus Gegenständen des World Wide Web. Das realisierte Datenbankschema ist in Abbildung 6.3 zu sehen.

Die in der Datenbank modellierten Objekte sind die Folgenden:

Benutzer Für geschützte Dokumente eines Angebots ist der Zugriff erst nach vorhergehender Authentisierung möglich. Dazu wird ein eindeutiger Benutzername vergeben. Bei der Anforderung von frei zugänglichen Seiten bleibt der Benutzer anonym. Daher ist der Benutzernamen in der Datenbank als optionales Feld gekennzeichnet. Wenn ein neuer Benutzer den Server besucht, wird der Zeitpunkt seines ersten Auftretens festgehalten.

Rechner Der Rechner, von dem aus die Dokumente angefordert werden, ist durch seine eindeutige IP-Adresse identifiziert. Auch hier wird der Zeitpunkt des ersten Auftretens gespeichert.

Domäne Das Internet ist in verschiedene Domänen unterteilt. Dabei ist jeder Rechner eindeutig einer Domäne zugeordnet.

Dokument Die Dateien, die auf dem Server zum Abruf bereitliegen.

Angebote Ein WorldWideWeb-Server kann mehrere Angebote enthalten. Zu jeden Angebot gehören verschiedene Dokumente eines Anbieters.

Sitzung Wenn ein Benutzer oder ein Rechner mehrere Anfragen in einem Zusammenhang stellen, werden die Anfragen in einer Sitzung zusammengefaßt. Zu jeder Sitzung wird ihr Beginn, ihre Dauer und eine Referenz auf die nächste Sitzung des gleichen Rechners oder Benutzers gespeichert.

Anfragen sind die zentralen Gegenstände der Miniwelt. Jede erfolgreich durchgeführte Anfrage wird in der Datenbank eingetragen. Der Eintrag enthält das angeforderte Dokument, evtl. den Benutzernamen, wenn er nicht anonym bleibt, die Sitzung, aus der die Anfrage stammt und den Rechner, von dem aus die Anfrage gestellt wird. Außerdem wird der Zeitpunkt der Anfrage und die Dauer bis zur nächsten Anfrage aus der gleichen Sitzung gespeichert.

Zugriffsprotokolldatei Die Datei, in der alle Anforderungen an den Server protokolliert werden.

In der Zugriffsprotokolldatei stehen für jede Anforderung von Dokumenten, die an den Server gerichtet werden, folgende Informationen: Die Internet-Adresse des Rechners, von dem die Anforderung kommt, eventuell ein Benutzername, wenn der Benutzer auf geschützte Seiten zugreift, die Uhrzeit der Anforderung, der Dokumentenname, ein Fehlercode und die Menge der übertragenen Daten. Diese Daten werden in der zeitlichen Reihenfolge, wie die Dokumente angefordert wurden, verzeichnet.

Auf diesem Datenbankschema sind 26 verschiedene Anfragen definiert, die vom Statistikmodul in Statistiken umgesetzt werden.

Die wichtigsten davon sind:

- ▹ Anzahl der Anfrage pro Stunde an den WebServer
- ▹ Anzahl der Anfragen pro Tag an den WebServer
- ▹ Anzahl der Benutzer des WebServers pro Tag
- ▹ Anzahl der Rechner, die den WebServer benutzen, nach Tagen aggregiert.

6.2 Der MultiSiteAnalyzer

Zuerst wird auf den inhaltlichen Zusammenhang von Servern im WorldWideWeb eingegangen. Daraus ergeben sich die Komponenten des Systems, welches anschließend vorgestellt wird.

6.2.1 Einsatzgebiete

Der MultiSiteAnalyzer bildet den integrativen Teil einer verteilten Datenbankanwendung für Internetnutzungsstatistiken. Während die Erstellung dieser Statistiken schon in Abschnitt 6.1 behandelt wurde, wird jetzt auf den Zusammenhang dieser Statistiken, die über das ganze Internet verteilt an jeder Website anfallen, eingegangen.

Dieser Zusammenhang ist ausschnittsweise und beispielhaft in der Abbildung 6.1 dargestellt. Am linken Bildrand befinden sich die Nutzer des Internet. Am oberen Bildrand stehen die bereits genannten unabhängigen Verbände, die das Mediengeschehen kontrollieren und messen.

Das Werkzeug "MultiSiteAnalyzer" hat aus der Sicht eines Werbekunden die Aufgabe, Nutzungsstatistiken über Seiten, auf denen er Anzeigen geschaltet hat, einzuholen, miteinander zu vergleichen und diese gegenüberzustellen. Dies wird immer dann erforderlich, wenn ein Werbekunde seine Anzeigen in *mehreren* verschiedenen Web-Angeboten geschaltet hat. Werden die in den Anzeigen enthaltenen Hyperlinks auf das Angebot des Werbekunden von einem Internet-Benutzer verfolgt, so findet er den dadurch entstehenden Zugriff auf seine Website in seiner eigenen WebSiteProfiler-Statistik[1]. Tatsächlich interessieren einen Werbekunden allerdings auch die reinen Zugriffe auf die Angebotsseiten der Unternehmen, in denen er seine Werbung plaziert hat. Diese Daten sind nur in den WebSiteProfilern der jeweiligen Unternehmen vorhanden und müssen dem interessierten Werbekunden zur Verfügung gestellt werden. Auf diesem Wege erhält der Kunde einen Eindruck über die Reichweite seiner Werbung, aufgeschlüsselt nach jeweiliger Werbeinsel [Sta93, Sta94b, Sta94a].

Für die Medienverbände spielen die in der Werbung enthaltenen Links auf andere Sites keine Rolle, sie untersuchen die verschiedenen Angebote i.d.R. unabhängig von ihrem konkreten Werbeinhalt. Dabei spielen dann Klassifikationen nach Zugriffszeit, Benutzergruppe, Angebotsart etc. die wesentliche Rolle.

In beiden Fällen sollen die Nutzungsstatistiken auf quantitativer Ebene verglichen werden. Dazu bieten sich Ranglisten an.

6.2.2 Systemübersicht

Bei dem MultiSiteAnalyzer handelt es sich nach [ÖV94] um ein verteiltes, homogenes System vieler Datenbanken. Die WebSiteProfiler, welche auf Rechnerknoten des Internet verteilt sind, bilden die einzelnen, autonomen Datenbanken. Der MultiSiteAnalyzer stellt nun Anfragen an diese Datenbanken und aggregiert die Einzelergebnisse in einem Anfrageergebnis.

Im Internet arbeiten mehrere MultiSiteAnalyzer, von denen jeder mit mehreren WebSiteProfilern kommuniziert. Da die Angebote einer Website für mehrere Vergleiche interessant sind, werden wiederum verschiedene MultiSiteAnalyzern diese Statistiken anfordern.

Dies führt zu der in Abbildung 6.4 gezeigten Kommunikationsstruktur.

Die WebSiteProfiler sind autonome Systeme, die ihre Datenbestände selbst verwalten. Dem MultiSiteAnalyzer, wird erlaubt, von außen Anfragen an die Datenbanken zu stellen. Er erhält jedoch keinen ändernden Zugriff auf die Daten des WebSiteProfilers.

Desweiteren müssen die Daten, die von den WebSiteProfilern gewonnen werden, im MultiSiteAnalyzer aufbereitet werden. Die oben genannten Medienverbände weisen ihre Statistiken in der Regel als Rangliste aus. Daran orientiert sich der MultiSiteAnalyzer. Diese Rangliste wird dem Benutzer als Ergebnis seiner Anfrage geliefert.

Insgesamt kommt man zu dem Systemaufbau aus Abbildung 6.5. Auch in dieser Anwendung wird eine Benutzerverwaltung und ein Archiv eingesetzt.

[1] Dabei auftretende Seitencache-Effekte werden in der Praxis häufig durch Einrechnung eines Cache-Faktors berücksichtigt.

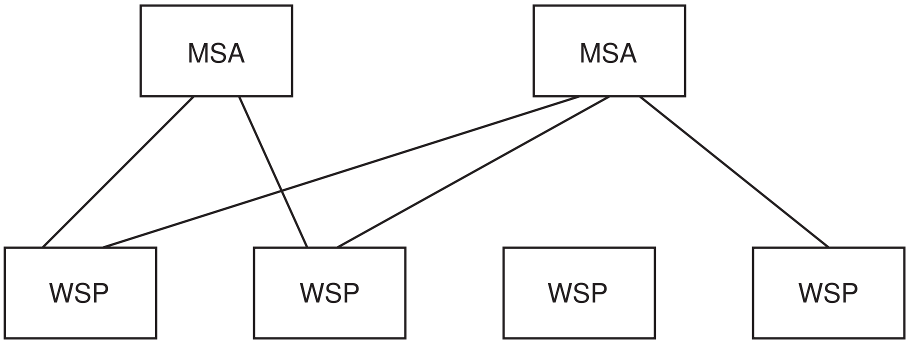

Abbildung 6.4: Die Kommunikation von WebSiteProfilern und MultiSiteAnalyzern

6.2.3 Integration von WebSiteProfilern

Die beiden Anwendungen "WebSiteProfiler" und "MulitSiteAnalyzer" sind voneinander logisch, und in der Regel auch physisch, isolierte Systeme. Die Integration dieser Systeme, im Sinne einer kooperativen Anwendung, steht im Vordergrund der weiteren Betrachtung. Aus diesem Grund wurde von dem klassischen Weg, einer Kommunikation mittels entferntem Prozeduraufruf oder dem direkten Austauch von dienstspezifischen Nachrichten, abgesehen. Statt dessen wird im folgenden Kapitel gezeigt, wie diese Integration mittels der im Kapitel 3 eingeführten *Business Conversations* geleistet werden kann. Dabei geschieht dies auf der Grundlage mobiler Agenten in einer Umgebung, wie sie im Kapitel 5 dargelegt wurde.

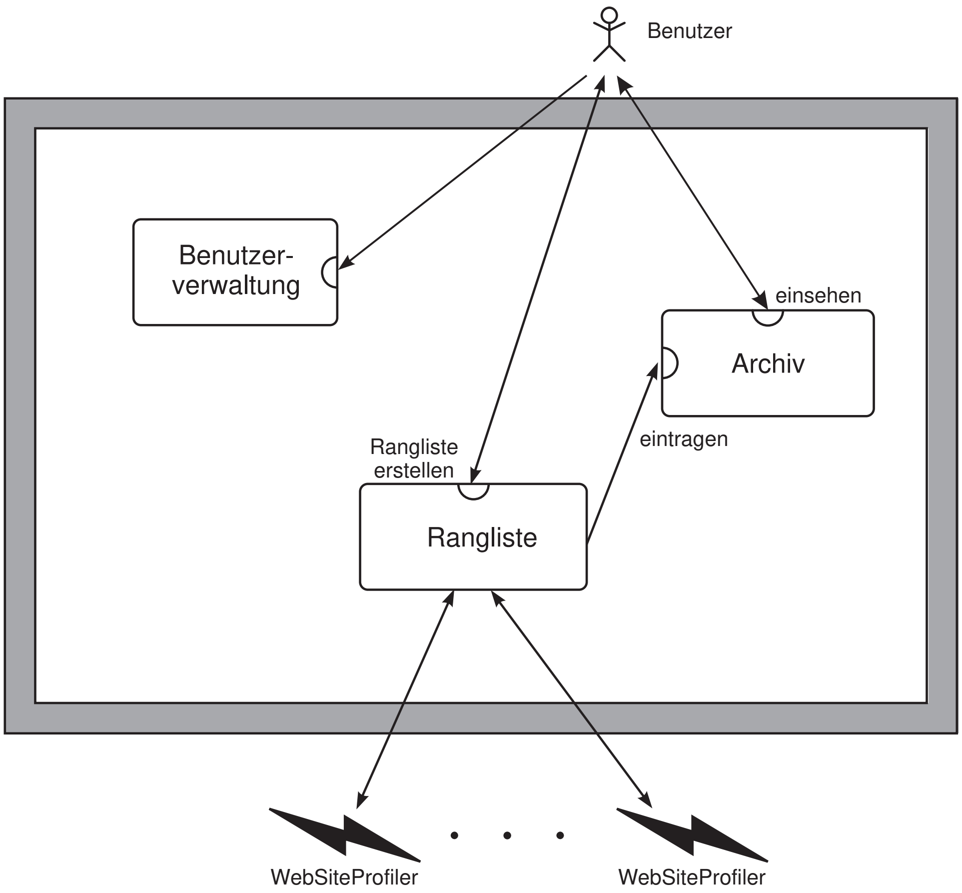

Abbildung 6.5: Die Systemübersicht für den MultiSiteAnalyzer

Kapitel 7

Das abstrakte Programmiermodell

In diesem Abschnitt wird die verteilte Datenbankanwendung des "MultiSiteAnalyzers" agentenbasiert modelliert. Ziel ist es, die Verwendung der im Kapitel 5 vorgestellten Konzepte anhand eines realistischen Beispiels zu demonstrieren. Es wird keine vollständige Realisierung des MultiSiteAnalyzers gezeigt. Lediglich die aus der Sicht des Agentensystems interessanten Teile sollen an dieser Stelle illustriert werden. Im Vordergrund stehen dabei die Agenten bzw. Orte und nicht der bereits geschilderte Aufbau des Agentensystems.

Zur Beschreibung der Agenten und Orte wird eine Pseudokode-Darstellung verwendet, die es erlaubt, von den Details der konkreten Implementierung zu abstrahieren. Diese wird dann in Kapitel 9 beschrieben.

7.1 Das Kooperationsmodell: Konversationen

Das primäre Ziel der mobilen Software-Agenten ist es, Koordinations-, Kommunikations- und Mobilitätsaspekte aus einer Anwendung herauszufaktorisieren. Auf diese Weise kann sich der Anwendungsprogrammierer auf den *Anwendungsteil* seiner Applikation konzentrieren, der durch die Agenten gebildete *Systemteil* ist generisch und steht daher zur Verfügung. Durch die Verankerung definierter Koordinations- und Kommunikationsmethoden im Systemteil können verschiedene Anwendungen miteinander interagieren, ohne daß dies explizit bei der jeweiligen Anwendungserstellung berücksichtigt werden muß.

Da das Modell der *Business Conversations* nicht nur eine Modellierungsmethode darstellt, sondern auch ein definiertes Verarbeitungsmodell, d.h. ein definiertes Interaktionsprotokoll zwischen den Aktoren, besitzt, bildet es die Grundlage der Kooperation unter Agenten.

Dabei erfordert das *Business Conversations*-Modell die Offenlegung der Kooperationsschnittstelle einer Anwendung in Form von *Konversationsspezifikationen*. Diese Prozeßspezifikationen sind vollständig medien- und aktorenunabhängig. Eine Konversationsspezifikation, die von einem Agenten genutzt wird, kann für eine Kooperation unter menschlichen Aktoren entwickelt worden sein. Das zugrundeliegende Spezifikationsobjekt ist uniform und kann daher ohne Anpassung für alle möglichen Arten von Aktoren wiederverwendet werden.

Die von den Beispielagenten benutzten Konversationsspezifikationen werden gemäß der Grammatik aus Abbildung B gebildet. Auf eine vollständige Wiedergabe der für die Anwendung erforderlichen Spezifikationen wurde aus Gründen der Übersichtlichkeit verzichtet, jedoch ist der in Abbildung 3.9 gezeigte Ausschnitt exemplarisch. Die grafische Darstellung einer solchen Spezifikation, wie sie in Abbildung 3.8 gezeigt wird, bietet einen Eindruck über die vollständige, vom *WebSiteProfiler* unterstützte Kooperationsschnittstelle, wie sie die Beispielagenten verwenden.

7.2 Beispielagenten

"Internet WebSiteProfiler" werden durch Orte innerhalb von Domänen repräsentiert. Diese Orte bilden die Ziele für reisende Agenten der ebenfalls durch Orte vertretenen "MultiSiteAnalyzer"-Anwendungen im Netz. In Abbildung 7.1 findet sich die Pseudokodedarstellung eines solchen "MultiSiteAnalyzer"-Ortes.

Darin sind Funktionen, die vom lokalen Agentensystem einer Domäne exportiert werden, durch den Prefix *agent* gekennzeichnet. Obwohl auch durch mobile Agenten referenziert, unterliegen diese Bindungen nicht der Mobilität der Agenten. Stattdessen verweisen diese Referenzen nach einer Agentenmigration transparent auf das in der jeweils aktuellen Domäne vorhandene Agentensystem. Die hierfür notwendigen Bindungstechniken werden im Abschnitt 8.4 näher beschrieben.

Jeder Ort wird innerhalb einer Domäne als Teil einer Orte-Hierarchie etabliert:

=> *agent.newPlace(owner msa)*

In diesem Kodefragment wird die Operation *newPlace(...)* des aktuellen Agentensystems nachgefragt.

Das Agentensystem erzeugt, abhängig von der Art des Besitzers (hier das Argument *owner*), einen neuen Ort. Bezeichnet *owner* einen Orte-Wächter, so wird der Ort des "MultiSiteAnalyzers" eine Hierarchieebene unter der des angegebenen Ortes erzeugt. Der neue Ort kann dann nur über den Ort *owner* erreicht werden. Bezeichnet *owner* jedoch eine mobilen Agenten, so wird der neue Ort unterhalb der Hierarchieebene angelegt, die der angegebene Agent momentan besucht. In beiden Fällen muß der übergeordnete Orte-Wächter der Erzeugung zustimmen[1].

Ist ein Ort erfolgreich vom Agentensystem angelegt worden, so initiiert es im zum Ort gehörenden Orte-Wächter die Ausführung der *birth*-Methode, die jeder Agent implementiert. Diese Methode wird innerhalb einer *künstlichen* Dienstleisterrolle vom Orte-Wächter atomar ausgeführt[2] und initialisiert die Regelbasis des Agenten. Dabei werden die Rollen, die der Agent innerhalb von Konversationen einnehmen kann, festgelegt. Der Agent bleibt nach der Ausführung der *birth*-Methode erhalten, wartet nun jedoch auf externe Ereignisse. Dies können Gesprächsaufforderungen bzw. Treffen mit anderen Agenten sein. Bei Orten wird auch das Eintreffen, Erzeugen und Verlassen anderer Agenten durch den Orte-Wächter registriert.

Der Beispiel-Ort des MSA vereinbart in seiner *birth*-Methode zwei Kundenrollen und eine Dienstleisterrolle. Innerhalb einer solchen Rolle befinden sich die Bearbeitungsregeln, die die Reaktion des Agenten auf Dialoge bzw. Anfragen, festlegen. Der MSA-Ort tritt gegenüber einem nicht näher ausgeführten User-Interface Agenten in Form eines Dienstleisters auf (definiert durch die *msaGUIRole*-Rolle). Diese Rolle hat die Aufgabe, Aufträge für die durch den Ort zu versendenden Boten-Agenten vom Benutzer der MSA-Anwendung entgegenzunehmen.

Zu diesem Zweck werden in der Bearbeitungsregel für "Parameter"-Dialoge bei "start"-Anfragen dynamisch die sogenannten *Boten-Agenten* erzeugt:

agent.new(self msaCollector "MSA-Bote")

Der Kode eines solchen Boten-Agenten ist in Abbildung 7.3 dargestellt.

Diese Boten, (die *msaCollector*-Agenten), dienen der Interaktion mit den "WebSiteProfiler" Systemen und tragen jeweils einen Auftrag des MSA-Benutzers mit sich. Jedem Boten wird der Ort seines "WebSiteProfiler"-Systems mitgeteilt, zu dem er reist.

Diese Auftragsvergabe geschieht über eine asynchrone Konversation zwischen dem Boten und dem MSA-Orte-Wächter, die von dem Wächter mittels

[1] Jede Domäne hat immer mindestens einen Ort; dieser bildet die Wurzel der Orte-Hierarchie der Domäne.
[2] Siehe Kapitel 5.2.2.

```
PLACE msa WITH s :S DO
  BIRTH self WITH s :S DO
    ROLE CUSTOMER collectorSetupRole SPEC csSpec CONTEXT :A RULES
      ON DIALOG "Auftrag" WITH conv, context DO
        conv.dialog.Ziel := context.target
        "ausfuehren"
      END
    END_ROLE

    ROLE PERFORMER msaGUIRole SPEC msaGuiSpec CONTEXT :Ok RULES
      ON INIT WITH conv, context DO
        d = conv.newDialog("Anmeldung")
        d
      END

      ON DIALOG "Anmeldung" REQUEST "anmelden" WITH conv, context DO
        d = ...
        d
      END

      ON DIALOG "Parameter" REQUEST "start" WITH conv, context DO
        FOREACH site IN conv.dialog.WspAdressliste DO
          agent.new(self msaCollector "MSA-Bote")
          s.addAgent(collectorAgent)
          collectorContext.target := site
          collector = agent.meet(self "MSA-Bote"
                      collectorSetupRole.spec 0)
          collectorSetupRole.startConversation(collector collectorContext)
        END
        d = conv.newDialog("AgentsState")
        d.Nachricht := "Agenten beauftragt."
        d
      END
    END_ROLE

    ROLE CUSTOMER retrieveResultsRole SPEC retRsltsSpec CONTEXT :S RULES
      ON DIALOG "Results" WITH conv, context DO
        s.addResult(conv.dialog.TabelleMitErgebnissen)
        "danke"
      END
    END_ROLE

    s.retrieveResults := retrieveResultsRole
    s.results := nil
  END_BIRTH

    ...
```

Abbildung 7.1: Pseudokodedarstellung des MSA-Ortes

```
  ...
  INCOMING_AGENT self WITH other, s DO
    if s.expectReturnOfAgent(other) then
       s.removeAgent(other)
       s.retrieveResults.startConversation(other s)
       fComeIn := true
    else fComeIn := false end
    fComeIn
  END _ INCOMING_AGENT
END_PLACE
```

Abbildung 7.2: Pseudokodedarstellung des MSA-Ortes (Fortsetzung)

collector = agent.meet(self "MSA-Bote" collectorSetupRole.spec 0)

erfragt wird. Das Agentensystem vermittelt das Treffen mit dem Agenten "MSA-Bote"[3], dabei erhält dieser Kenntnis sowohl von dem Initiator als auch von dessen gewünschter Konversationsspezifikation. Der letzte Parameter gibt an, wieviele Sekunden der Agent auf das Zustandekommen eines Treffens warten will. Dabei bildet die Angabe eine untere Schranke für die Dauer in der ein Treffen vermittelt werden soll.

Die Konversation mit dem Boten wird danach vom MSA-Orte-Wächter an eine bereits vorbereitete Rolle *collectorSetupRole* gebunden:

collectorSetupRole.startConversation(agent collectorContext)

Im letzten Schritt dieser Konversation migriert der Boten-Agent zu seinem Zielort. Dort führt er dann autonom eine Konversation mit dem ansässigen WSP-Agenten, um die statistischen Daten gemäß seines Auftrags von diesem zu ermitteln.

Ist die Migration des Boten zu seinem Bestimmungsort abgeschlossen, führt er eine synchrone, sogenannte *sekundäre*, Konversation mit dem WSP-Agenten in der Rolle eines Kunden:

agent.meetAndTalk(self "WebSiteProfiler" wspCustomerRole 30)

Nach Beendigung dieser Konversation reist der Boten-Agent wieder an seinen Ursprung zurück, um die ermittelten Daten an den MSA-Agenten weiterzureichen.

agent.migrateTo(self self.home())

Das Agentensystem der Domäne, an der der MSA-Ort angesiedelt ist, informiert den entsprechenden MSA-Orte-Wächter über das Eintreffen eines Agenten. Diese Benachrichtigung führt zu einer Aktivierung der *incoming_agent*-Methode des Ortes. Auf diese Weise erhält der Ort Kenntnis von dem Agenten, der eintreten will. Das erlaubt dem Orte-Wächter, mit dem ankommenden Agenten eine Konversation zu beginnen. Ist der Ort nicht gewillt, den Agenten in seinen Sichtbarkeitsbereich eintreten zu lassen, kann er dies dem Agentensystem durch das Ergebnis der Methodenausführung mitteilen.

[3] Um Namensüberschneidungen zu verhindern, kann der Name des Agenten auch aus einem generierten Bezeichner bestehen.

```
AGENT msaCollector WITH s :Ok DO
    BIRTH self WITH s :Ok DO
        ROLE CUSTOMER wspCustomerRole SPEC wspSpec CONTEXT :A RULES
            ON DIALOG "Anmeldung" WITH conv, context DO
                conv.dialog.Name := context.myName
                conv.dialog.Passwort := context.myPassword
                "anmelden"
            END

            ON DIALOG "Taetigkeitswahl" WITH conv, context DO
                ...
                "Neue Statistiken ansehen"
            END

            ON DIALOG "Neue Statistiken" WITH conv, context DO
                ...
                "Statistiken erstellen"
            END

            ON DIALOG "Ergebnisse" WITH conv, context DO
                context.results := conv.dialog.TabelleMitErgebnissen
                "beenden"
            END
        END_ROLE

        ROLE PERFORMER collectorSetupRole SPEC csSpec CONTEXT :Ok RULES
            ON INIT WITH conv, context DO
                d = conv.newDialog("Auftrag")
                d
            END

            ON DIALOG "Auftrag" REQUEST "ausfuehren" WITH conv, context DO
                target = conv.dialog.Ziel
                agent.migrateTo(self target)
                agent.meetAndTalk(self "WebSiteProfiler" wspCustomerRole 30)
                agent.migrateTo(self self.home())
                d
            END
        END_ROLE
    END_BIRTH
END_AGENT
```

Abbildung 7.3: Pseudokodedarstellung des MSA-Botenagenten

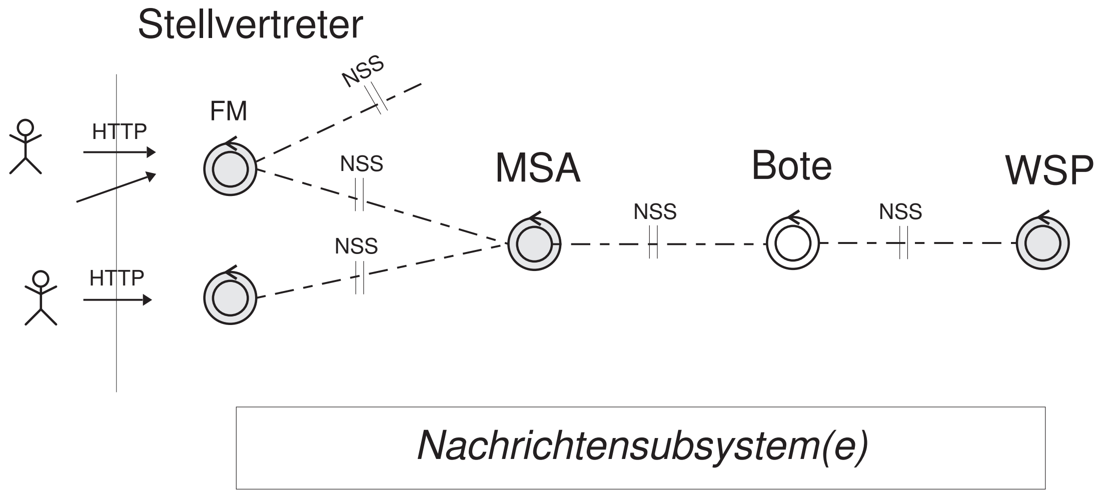

Abbildung 7.4: Synchronisation und Delegation mittels Agenten

Im dem gezeigten Beispiel beginnt der MSA-Orte-Wächter eine Konversation mit dem eintreffenden Botenagenten, die dazu verwendet wird, die mitgebrachten Statistikdaten des Boten-Agenten dem MSA-Orte-Wächter zugänglich zu machen. Dieser kann auf diese Weise die Daten aller eintreffenden Agenten aggregieren und diese Aggregation wiederum anderen Agenten zugänglich machen.

Durch diese Art der Programmierung bilden die Boten-Agenten eine Form von *intelligenten* Nachrichten, die z.B. auf auftretende Ausnahmesituationen bei ihrem Empfänger reagieren können.

7.3 Synchronisation und Delegation

Die verwendeten Koordinationsmechanismen Rendezvous und Konversation ermöglicht es, Kontakte zwischen zwei Agenten herzustellen, bei denen ein dritter Agent als Vermittler auftritt. Für das beschriebene Beispielszenario zeigt dies Abbildung 7.4.

Der mit FM bezeichnete Agent ist der Stellvertreteragent eines Benutzers aus der realen Welt. Stellverteter repräsentieren Benutzer und deren Autorität gegenüber anderen Agenten innerhalb der modellierten Welt. Solche Agenten sind nicht als vollständig autonom anzusehen, sie stellen die personalisierte Verbindung von der realen zur modellierten Welt dar. In der Abbildung ist diese Verbindung über eine HTTP-Anbindung realisiert. Der Stellvertreteragent fungiert hierbei auch als Protokollumsetzer zwischen den HTTP-Nachrichten und den *Business Conversations*.

Der Agent synchronisiert die unterschiedlichen Aktivitäten für Protokollumsetzung und Dialogaustausch über die Serialisierung der entsprechenden Nachrichten, die ihm über das Nachrichtensubsystem zugestellt werden.

Werden Rendezvousanfragen über den Treffen-Mechanismus des Agentensystems an den Agenten herangetragen, so können diese an andere Agenten *delegiert* werden. Dies ist insbesondere dann sinnvoll, wenn sich ein Agent permanent an einem Ort befindet, jedoch kein geeigneter Dienstleister vorhanden ist. In diesem Fall kann der Kontaktwunsch eines Agenten A_1 von dem durch das Agentensystem ermittelten Agenten A_2 an einen dritten Agenten A_3 weitergereicht werden.

Dieses Weiterleiten wird wiederholt, bis ein Agent entweder den Wunsch nach einem Treffen durch Agent A_1 annimmt, oder diesen Wunsch endgültig verweigert. Für den initiierenden Agent A_1 ist das Verfahren transparent.

Die Agenten, die im Laufe des Protokolls vorgeschlagen werden, müssen notwendigerweise am selben Ort wie der Initiator des Treffens existieren. Allerdings ist eine dynamische Erzeugung von Bearbeitern innerhalb des Weiterleitungsprotokolls erlaubt. Auf diese Weise können Bearbeiteragenten von einem zentralen Brokeragenten erzeugt werden, die individuelle Beziehungen zu Agenten aufrechterhalten.

Das Agentensystem behandelt dabei jede Weiterleitung wie ein gesondertes Treffen, so daß ein Bearbeiteragent auch innerhalb der zum Treffen angegebenen Warteperiode am Ort eintreffen darf.

Teil III

Realisierung des Modells

Kapitel 8

Die Programmierumgebung Tycoon

Die persistente, polymorphe Programmierumgebung Tycoon eignet sich Aufgrund ihrer bereits vorhandenen Systemdienste gut zur verteilten Programmierung und damit zur Implementation des Modells für verteilte Agenten. Hierfür sind besonders die migrationsfähigen Threads, das dynamische Linken und die Kommunikationsdienste für heterogene Systeme nützlich.

8.1 Allgemeine Konzepte

Die Implementation aller im weiteren präsentierten Module erfolgte in der streng typisierten, persistenten und polymorphen Programmierumgebung *Tycoon*, die am Arbeitsbereich DBIS der Universität Hamburg entwickelt wurde. Sie basiert in weiten Teilen auf der funktionalen Programmiersprache *Quest* [Car89]. Eine detaillierte Beschreibung der vom Tycoon-System verwendeten Sprache TL ist in [MMM93, MMS94] nachzulesen. An dieser Stelle soll nur ein Überblick über die Sprache TL und das Tycoon Language Environment gegeben werden.

8.1.1 Werte und Typen

Tycoon bietet sowohl statische als auch dynamische Bindungen von Werten an Bezeichner. Statische Wertbindungen werden folgendermaßen definiert:

let $a\ =\ 3$

Nach Evaluation dieses Terms ist der Bezeichner a an den Wert 3 statisch gebunden. Die rechte Seite der Bindung kann ein beliebiger Ausdruck sein. Wird diese Bindung innerhalb eines **begin ... end** Paars geschrieben, so ist sie nur innerhalb dieses Blocks sichtbar.

```
let a = 3
begin
   let a = 5
   let b = a
end
let c = a
```

Hier wird der Bezeichner c an den Wert 3 gebunden, da die lokale Bindung $a = 5$ außerhalb des Blocks nicht sichtbar ist. Dynamische Bindungen werden über Funktionen erzeugt. Tycoon verwendet Funktionen als Elemente erster Ordnung und kennt rekursive Funktionen. Funktionen werden durch das Schlüsselwort **fun** eingeleitet und können an Bezeichner gebunden werden.

let *succ =* ***fun****(x* ***:Int****) x+1*

Hier wird die Funktion, die den Nachfolger einer ganzen Zahl berechnet, an den Bezeichner *succ* gebunden. Der Bezeichner x ist eine dynamische Bindung, die innerhalb des Funktionsrumpfs $x+1$ sichtbar ist.

Im Funktionsrumpf sind sowohl die in der Signatur definierten Aktualparameter sichtbar, als auch die im statischen Sichtbarkeitsbereich der Funktion definierten globalen Bezeichner. Innerhalb des Funktionsrumpfs können zusätzlich beliebige weitere lokale Bindungen definiert werden.

Der Doppelpunkt leitet immer Typbezeichner ein. Folglich ist **:Int** ein (von TL vordefinierter Basis-) Typ des Formalparameters x. Es ist nicht immer notwendig diesen Typ explizit anzugeben, in vielen Fällen ist die Typprüfung des Tycoon-Systems in der Lage, diesen zu inferieren.

Da in TL alle Objekte typisiert sind, besitzt auch jede Funktion einen Typ. In dem oberen Beispiel der Funktion *succ* ist dies: **:Fun(:Int) :Int**, also eine Funktion die genau ein Argument vom Typ **:Int** erwartet und einen Wert vom Typ **:Int** als Ergebnis liefert.

Um Funktionen höherer Ordnung definieren zu können, ist solch ein Funktionstyp notwendig.

let *twice =* ***fun****(f* ***:Fun(*** ***:Int****)* ***:Int*** *x* ***:Int****)f(f(x))*

Diese Funktion erwartet als erstes Argument eine Funktion vom Typ **:Fun(:Int) :Int** und als zweites Argument einen Wert vom Typ **:Int**. Die Funktion selbst wendet die als Argument übergebene Funktion (dynamisch an den Bezeichner f gebunden) zweimal an.

Neben den vordefinierten Basistypen **:Int :Bool** etc. gibt es die Möglichkeit eigene Typen zu vereinbaren. Zu diesem Zweck bietet TL die Typkonstruktoren Tupel, Tupel mit Varianten, Records und Records mit Varianten.

- ▷ Tupel sind geordnete Mengen von Bindungen.

  ```
  Let Person = Tuple
    name :String age :Int
  end
  let p :Person = tuple
    "Peter" 15
  end
  ```

  Auf die Komponenten eines solchen Tupels (der Wertbindung p) kann über die Punktnotation zugegriffen werden.

  ```
  p.name
  => "Peter"
  ```

- ▷ Records sind ungeordnete Mengen benannter Bindungen.

```
Let PersonRecord = Record
  name :String age :Int
end
let p :PersonRecord = record
   let age = 12  let name = "Fred"
end
```

Auf die Komponenten eines Records wird ebenfalls über die Punktnotation zugegriffen. Im Gegensatz zu Tupeln können Records dynamisch um neue nicht-anonyme Bindungen erweitert werden.

```
let p2 = extend p with
   let semester = 1
end
```

Variante Tupel oder Recordtypen unterscheiden mehrere disjunkte und benannte Fälle, mit jeweils eigenen Komponenten. Komponenten, die allen Fällen gemeinsam sind, können vor der Fällunterscheidung definiert werden. Werte dieser varianten Type können mit den Konstruktoren **tuple_case** und **record_case** erzeugt werden, bei denen dann auch die gewünschte Variante angegeben werden muß. Auf die Komponenten einses varianten Tupels bzw. eines varianten Records kann mittels eines Projektionsoperators zugegriffen werden.

```
Let Person = Tuple
  name :String age :Int
   case employee with
     salary :Real
   case ceo
end
let peter = tuple_case ceo of :Person with
  "Peter" 40
end
let fred = tuple_case employee of :Person with
  "Fred" 31
  4600.0
end
```

8.1.2 Subtypbeziehungen und Subtyppolymorphismus

Signaturen der Form x :A stellen partielle Spezifikationen dar, die aussagen, daß x *mindestens* A erfüllt. Ein an den Bezeichner x gebundener Wert kann aber eine genauere Spezifikation B erfüllen. Die zugrunde liegende partielle Ordnung auf Typen "B ist präziser als A" wird durch die induktiv definierte B <:A (B ist Subtyp von A) in TL explizit beschrieben. [Mat93] Der Supertyp aller nicht parametrisierten Typen ist **Ok**.

```
Let Person = Tuple
  name :String age :Int
```

```
end
Let Student = Tuple
  name :String age :Int semester :Int
end
let fred :Student = tuple
  "Fred" 20 1
end
let getPersonAge(p <:Person) = p.age

getPersonAge(fred);
=> 20
```

Die Funktion *getPersonAge* ist eine polymorphe Funktion, da sie keinen expliziten Argumenttyp vorschreibt, sondern lediglich einen Mindesttyp spezifiziert. Daher arbeitet die Funktion *getPersonAge* auch mit Werten vom Typ :Student. Generell liefert die Funktion *getPersonAge* für alle denkbaren Typen ein Ergebnis, solange diese Subtyp des Types :Person sind. Für Tupeltypen gilt die Subtypbeziehung A <:B solange A strukturell mit B kompatibel ist. Für eine detaillierte Beschreibung der Subtypisierungsregeln siehe [Mat93]. Subtypbeziehungen gelten in TL für alle in der Sprache definierbaren Typen (einschliesslich der Basistypen).

8.1.3 Module, Schnittstellen und Bibliotheken

In TL werden Programmteile zur Unterstützung der Programmierung im Großen analog Modula-2 in Module und Schnittstellen getrennt. Ein Modul, welches Funktionen anderer Module nutzen will, muß diese importieren. Zugreifen kann das Modul allerdings nur auf die Funktionen die in der zugehörigen Schnittstelle exportiert wurden. Die Schnittstellen bilden somit die Typen der Module. Es besteht keine Notwendigkeit einer festen 1:1 Beziehung zwischen Modulen und Schnittstellen. Jedes Modul kann beliebig viele Schnittstellen haben, die dann unterschiedliche Sichten auf das Modul darstellen. Auch kann jedes Schnittstelle zu beliebig vielen Modulen gehören, wenn diese mindestens die in der Schnittstelle spezifizierten Elemente exportieren.

```
module a
import b
export
let f1(x :Int) = b.f(x)
end

interface A
export
f1( :Int) :Int
end
```

```
module b
export
let f(x :Int) = x + 1
let g(x :Int y :Int) = x + y
end

interface B
export
f( :Int) :Int
end
```

Die Verbindung von Modul zur Schnittstelle wird über sog. Bibliotheken hergestellt.

```
library test with
  interface B
  module b :B
```

```
    interface A
    module a :A
end
```

8.1.4 Persistenz

Alle in TL beschreibbaren Bindungen können persistent werden, d.h. die Ausführung des Tycoon-Prozesses überdauern. Tycoon nutzt hierfür einen Objektspeicher, in dem sämtliche Bindungen abgelegt werden. Die Bindungen müssen nicht explizit als persistent gekennzeichnet werden; die Tycoon Laufzeitumgebung sorgt mittels Garbage-Collection für das Beseitigen nicht mehr referenzierter Bindungen. Der Objektspeicher kann zu jeder beliebigen Zeit eingefroren werden, um so nach einem Neustart des Systems sofort alle bis dahin definierten Bindungen wieder bereitzustellen.

8.1.5 Erweiterbarkeit um externe Bibliotheken

Ein wichtiger Bestandteil des Tycoon-Systems und der Sprache TL ist die Möglichkeit, externe Funktionsbibliotheken der verwendeten Betriebssystemumgebung typsicher in TL-Programme einbinden zu können. Diese Bibliotheken liegen entweder als sog. *'shared-libraries'* vor oder werden statisch an die virtuelle Tycoon-Maschine gebunden. Eine Funktion einer externen Bibliothek wird über das **bind** –Konstrukt dem Tycoon-System bekannt gegeben.

```
let tycoonFunction =
    bind( :Fun( :Int) :Int "libraryName""functionName" "externalSignatur")

tycoonFunction(5)
=> 1017
```

Funktionen, die in der Sprache 'C' geschrieben wurden, können leicht in TL genutzt werden. Dafür muß für jede 'C'-Funktion eine 'TL'-Signatur gefunden werden und die Funktion entweder statisch an die Tycoon-Laufzeitumgebung gebunden oder wie beschrieben in einer dynamisch bindbaren Bibliothek organisiert sein.

Ein weiteres sehr nützliches Element des Tycoon-Systems ist die Möglichkeit, aus diesen 'C'-Funktionen wiederum 'TL'-Funktionen ausführen zu lassen. Ein solcher *'C'-Callback* wird auf der Tycoon-Seite eingerichtet mit einem Aufruf von:

```
let cTLFunktion =
   cCallback.new( "tlFunktion" "externeSignatur")
```

Der 'C'-Callback *cTLFunktion* kann nun an eine beliebige 'C'-Funktion als 'C'-Funktionszeigerargument übergeben werden. Wird innerhalb der 'C'-Funktion die übergebene Funktion aufgerufen, so wird vom Tycoon-System die Funktion "tlFunktion" evaluiert und das Ergebnis an die aufrufende 'C'-Funktion zurückgegeben. Ein 'C'-Callback unterliegt nicht der Garbage-Collection und muß daher explizit freigegeben werden, wenn er nicht mehr verwendet werden soll.

8.2 Nebenläufigkeit und Synchronisationsmechanismen

Tyoon unterstützt nebenläufige Programmierung über leichtgewichtige Prozesse, sog. Threads. Threads sind orthogonal zu allen anderen Konzepten der Progammiersprache TL verwendbar, insbesondere sind sie auch persistent [MS94]. Jede TL-Funktion kann bei Bedarf nebenläufig zu anderen TL-Funktionen evaluiert werden, wenn sie als eigener Ausführungfaden (Thread) vom aktuellen Thread abgespalten wird (*fork-Operation*). Da Threads TL-Funktionen verwenden, gelten für sie die normalen statischen Sichtbarkeitsbereiche der Sprache. Werden veränderliche Bindungen von mehr als einem Thread konkurrierend zugegriffen, so ist zur Wahrung der Konsistenz die Synchronisation dieser Zugriffe notwendig. TL bietet hierfür die bekannten Mechanismen Semaphore, Mutexe, Rendezvous und Ereignisvariablen in einer Standardbibliothek an. Alle diese Verfähren basieren auf der gezielten Unterbindung des Kontextwechsels der virtuellen Tycoon-Maschine [Pie96].

8.3 Kommunikationsdienste

Verteilte Programmierung mit dem Tycoon-System kann sich auf eine Reihe von TL-Bibliotheken stützen, die eine Reihe unterschiedlicher Kommunikationsabstraktionen für den Entwickler bieten. Die Basis bildet eine Kernbibliothek zur typsicheren Anbindung von Unix-Sockets (TCP bzw. UDP/IP) an das Tycoon-System. Aufbauend auf diesen Kommunikationskanälen (Streams), die nur explizit typisierte TL-Objekte übertragen können, existiert ein typisierter RPC-Mechanismus [Joh95]. Generell können beliebige TL-Objekte über die Kommunikationsmechanismen in heterogenen Systemumgebungen ausgetauscht werden. Die virtuelle Maschine übernimmt ggf. notwendige Konvertierungen automatisch. Der RPC des Tycoon-Systems benötigt keine gesonderte IDL (Interface Definition Language) zur Spezifikation von Dienstschnittstellen. Stattdessen werden TL-Typen zur Beschreibung entfernter Dienste verwendet. Auf diese Weise, und unter Verwendung von dynamischen Typen, können Funktionen eines entfernten Dienstes vollständig statisch typisiert verwendet werden[1].

8.4 Bindungstechniken und Mobilität

Aufgrund der orthogonalen Einbindung von Threads in das Sprachgefüge von TL können Threads auch als Parameter für entfernte Funktionen verwendet werden. Geschieht dies, so wird ein Thread und seine transitive, referentielle Hülle auf den Zielrechner kopiert (*deep copying*). Auf diese Weise sind Threads *mobil*. Da Threads eine reflexive Bindung an sich selbst besitzen, können sie ihre eigene Migration veranlassen. Diese Technik bildet die Basis für einen mobilitätsorientierten Programmierstil zur Beschreibung von langlebigen Aktivitäten [MMS95].

Wird ein Thread in einen anderen Adressraum kopiert, so enthält dieser alle Bindungen, die sich zum Zeitpunkt des Kopierens in seinem Sichtbarkeitsbereich befanden und transitiv alle darüber erreichbaren. Hierbei können auch Bindungen auf Objekte enthalten sein, deren Semantik eine Mobilität verbietet oder nicht sinnvoll ist. Dies sind externe Ressourcen, die nicht unter der direkten Kontrolle des Tycoon-Systems stehen, wie z.B. Datei-Deskriptoren oder Fenster. Andererseits gibt es Objekte, deren Mobilität nicht erwünscht ist, z.B. im Tycoon-System verwaltet Massendatenbestände oder ubiquitär vorhandene TL-Bibliotheken.

Tycoon bietet für den Umgang mit solchen Objektbindungen zwei Ansätze:

- ▹ Dynamisches Relinken.
- ▹ Neuerzeugung flüchtiger Ressourcen.

[1] Die vollständige Typinformation liegt ja bereits zur Übersetzungszeit des lokalen TL-Kodes vor.

Der Mechanismus des dynamischen Relinkens kann für ubiquitäre Ressourcen verwendet werden. Hierbei ist es erforderlich, jedes Objekt mit einem sog. Linksymbol zu versehen und es über einen Funktionsaufruf von *ubiquitous.register(...)* bei der virtuellen Maschine als ubiquitär zu registrieren. Kern des Verfahrens ist, jedes so registrierte Objekt immer dann gegen das entsprechende Linksymbol zu ersetzen, wenn das Objekt für den Versand linearisiert werden soll. Auf der Empfängerseite können dann, im Zuge der Delinearisierung, transparent alle symbolischen Referenzen gegen die lokal vorhandenen Objektreferenzen ersetzt werden.

Das Versenden flüchtige Ressourcen erfordert ein geordnetes zerstören und wiedererzeugen der zug. externen Datenstrukturen, um die scheinbare Mobilität zu erreichen. Zu diesem Zweck bieten die Funktionen des Moduls *volatile* eine Pseudo-Persistenz für Objekte außerhalb des Tycoon-Systems. Das dort vorgesehene Protokoll erfordert hierfür ein Registrieren von entsprechenden Funktionen für die jeweilige flüchtige Ressource, mit denen diese kontrolliert werden kann. Das Erzeugen und Zerstören, sowie die Einhaltung der korrekten Reihenfolge, geschieht automatisch durch das Tycoon-System.

Kapitel 9

Eine generische Tycoon-Bibliothek für mobile Agenten

In diesem Kapitel wird der konkrete Entwurf der Umgebung für mobile Software-Agenten und dessen Realisierung beschrieben. Die Implementation besteht aus einer Reihe generischer Modulbibliotheken für das Tycoon-System, welches im vorherigen Kapitel bereits vorgestellt wurde. Entsprechend der Konzeption des Agentensystems gliedern sich die im Rahmen dieser Arbeit realisierten Module in drei separate Bereiche:

1. Kooperationsunterstützung: Implementation des *Business Conversation*-Modells.
2. Kommunikationsunterstützung: Implementation des Nachrichtensubsystems.
3. Koordination: Realisierung des Agentensystems und der Agentengeneratoren.

Um den Grad der Wiederverwendbarkeit zu erhöhen, wurde bei der Umsetzung darauf geachtet, die Module der einzelnen Systembereiche unabhängig von ihrer Nutzung durch das Agentensystem zu realisieren. Die Module zur Kooperationsunterstützung sind daher auch für Anwendungen einsetzbar, die *nicht* agentenbasiert verteilt werden sollen. Gleiches gilt für das Nachrichtensubsystem, welches als allgemeine Basisinfrastruktur für kommunikationsintensive und stark nebenläufige Anwendungen dienen kann, auch wenn diese keinerlei Agententechnologie verwenden.

9.1 Realisierung des *Business Conversation*-Modells

Die im *TBCEnv*[1] angesiedelten Module bilden die Basis der Kooperation zwischen zwei Kommunikationspartnern, wie sie das *Business Conversation*-Modell definiert.

Die Module dieser Bibliothek erzeugen alle Objekte, über die ein Informationsaustausch zwischen autonomen Kooperationspartnern abgewickelt wird. Analog zum Modell wird auch hier zwischen Konstruktoren für Spezifikationsobjekte und den aus ihnen erzeugten Instanzobjekten unterschieden.

Eine wichtige Prämisse für die Umsetzung des *Business Conversation*-Modells ist, daß TBC-Objekte per Definition keinem der Gesprächspartner "gehören". Dies bedeutet, beide Partner stimmen darin überein, daß der jeweils andere über alle ihm zugänglichen TBC-Objekte frei verfügen kann. Insbesondere hat jeder Konversationspartner das Recht, jedes TBC-Objekt kopieren bzw.

[1] Tycoon Business Conversations Environment

manipulieren zu dürfen. Dies ist natürlich bei der Realisierung von Massendatenobjekten besonders zu berücksichtigen, um die Effizienz einer Applikation nicht zu gefährden.

Kopiervorgänge, in denen TBC-Objekte involviert sind, treten in der praktischen Anwendung häufig auf, da jede Konversation zwischen entfernten Partnern eine Übertragung von Dialogobjekten über ein Netzwerk erfordert. Auch die Tatsache, daß zu jeder Konversation automatisch eine Historie aller ausgetauschten Dialogobjekte angelegt wird, führt zu potentiellen TBC-Objektkopien im Falle einer Agentenmigration.

Die Schnittstellen der einzelnen Module sind im Anhang A.1 in Auszügen wiedergegeben.

9.1.1 Dialoginhalt und dessen Spezifikation

Grundlage jeder Konversation ist die Notwendigkeit, die Struktur eines Dialoges beschreiben zu können. Zu diesem Zweck bietet das Modul *tbcContent.tm* kombinierbare Konstruktoren für die Erzeugung von Inhaltsspezifikationen gemäß dem Modell aus Abbildung 3.5.

Ein einfaches Spezifikationsobjekt zur Beschreibung von Inhaltsobjekten des Typs "natürliche Zahl" wird durch

```
=> let anyInt = tbcContent.intSpec()
```

generiert. Alle Spezifikationskonstruktoren erzeugen, unabhängig von den durch sie beschriebenen Instanzobjekten, ein Objekt vom Typ *:tbcContent.Spec*, welches die Strukturinformation der Instanzobjekte kapselt.

Dabei eröffnet die uniforme Behandlung aller darstellbaren Spezifikationen durch Objekte erster Klasse die Möglichkeit der reflektiven Programmierung über diesen Spezifikationsobjekten. Diese Technik findet in der werkzeugunterstützten Modellierung [Gei95] und der dynamischen Modellüberprüfung Anwendung.

Ein komplexeres, zusammengesetztes Inhaltsspezifikationsobjekt wird z.B. wie folgt erzeugt:

```
=> let person = tbcContent.rcdSpec of
       tuple "name"  tbcContent.stringSpec() end
       tuple "first"  tbcContent.stringSpec() end
       tuple "age"  tbcContent.intSpec() end
       tuple "birthday"  tbcContent.dateSpec() end
   end
```

Mit Records bzw. varianten Records ist es möglich, benannte Komponenten zu aggregieren. Die Komponentenobjekte können bei einer Instantiierung des umfassenden Spezifikationsobjektes des Rekords über den entsprechenden Namen, z.B. *birthday*, erreicht werden.

Die Operationen, welche auf solchen Spezifikationsobjekten erlaubt sind, werden durch den offengelegten varianten Tupeltyp *:Spec* festgelegt. Bei Spezifikationsobjekten, die atomare Instanzobjekte beschreiben, kann nur der jeweilige Instanzobjekttyp ermittelt werden. Bei den Aggregationen kann zusätzlich noch die Menge der enthaltenen Spezifikationsobjekte und deren Bezeichner festgestellt werden. Die Auswahl- und Sequenzspezifikationsobjekte besitzen nur genau ein Elementspezifikationsobjekt, das ebenfalls über eine Methode des umgebenden Spezifikationsobjektes ermittelt werden kann.

Aus jeder durch kombinieren erzeugbaren Inhaltsspezifikation können mittels *tbcContent.create(...)* Instanzobjekte erzeugt werden. Deren Struktur wird ist das Spezifikationsobjekt beschrieben:

=> *let fred = tbcContent.create(person)*
=> *let heinz = tbcContent.create(person)*

Jedes Objekt nimmt im Laufe dieses Instantiierungprozesses einen definierten Ausgangszustand an, der von der Art des jeweiligen Objektes abhängt.

Die Operationen, die auf den so instantiierten Inhaltsobjekten erlaubt sind, werden durch den varianten Tupeltyp *tbcContent.T* beschrieben. Jedes erzeugte Inhaltsobjekt ist über die Operation *spec()* selbstbeschreibend und ermöglicht so den Zugriff auf das Spezifikationsobjekt, aus dem es erzeugt wurde.

Für atomare Objekte sind weiterhin Operationen zum Lesen und Schreiben des gekapselten Objektzustandes vorgesehen. Records ermöglichen, durch die auf ihnen definierte *ref(...)* Operation, den Zugriff auf die aggregierten Komponentenobjekte. Bei einem varianten Record besteht darüber hinaus die Möglichkeit, eine der Varianten zu wählen oder die aktuell gewählte Variante zu ermitteln. Zu jeder dieser Varianten gehören eigene Komponenten, so daß die Menge der mittels *ref(...)* erreichbaren Komponenten von der aktuell ausgewählten Variante bestimmt wird.

Die Auswahlobjekte für einfache und mehrfache Auswahlen erlauben die Selektion eines oder mehrerer Objekte aus einer vorzugebenden, endlichen Menge[2]. Die Schnittstelle dieser beiden Objektarten unterscheidet sich lediglich darin, daß die Mehrfachauswahl eine Operation *discard(...)* zum entfernen bereits getroffener Auswahlen aus der Ergebnismenge vorsieht. Im Gegensatz zur einfachen Auswahl darf die Ergebnismenge der Mehrfachauswahl auch leer sein.

Die Implementation des vom *Business Conversation*-Modell vorgesehenen Massendatencontainers *Sequence* kann für jede Sequenz einzeln bestimmt werden. Wird ein solches Sequenzobjekt erzeugt, so verwendet dieses eine einfache Standardimplementation für Sequenzen, bei der die in der Sequenz enthaltenen Daten *Teil* des Containerobjektes werden. Das Protokoll für Containerobjekte ermöglicht es, diese Standardimplementation für Sequenzen gegen andere auszutauschen, wenn dies z.B. aus Gründen der Effizienz oder im Hinblick auf Verteilungsanforderungen gewünscht wird.

Zu diesem Zweck definiert das Interface *TBCContent.ti* den Tupeltyp *:Sequence*, welcher das eigentliche Zugriffsprotokoll auf die im Sequenzcontainer gekapselten Massendaten festlegt. Eine konkrete Sequenzimplementation muß die hier definierten Operationen *elements(...)* und *fromIter(...)* anbieten, mit denen zwischen Elementen der Sequenz (TBC-Objekte vom Typ *:tbcContent.T*) und den modellierten Massendaten auf TL-Seite abgebildet wird. Eine derartige Sequenzimplementation kann dann an ein bereits instantiiertes Sequenzobjekt gebunden werden und ersetzt damit die Standardimplementation für dieses Objekt.

Neben der Erzeugung eines Inhaltsobjektes aus einem Spezifikationsobjekt bietet das Modul *tbcContent.tm* noch die Funktion *copy(...)*, mit der ein ggf. strukturiertes Inhaltsobjekt unter Beibehaltung seines aktuellen Zustands kopiert werden kann. Eine solche Kopie besitzt einen eigenen, vom Original getrennten Zustand und eine identische Struktur. Für den Sequenzcontainer ist zu beachten, daß der Objektzustand die jeweilige Sequenzimplementation ist. Daher nutzen Original und Kopie dieselbe Sequenzimplementation und referenzieren über diese auch dieselben Sequenzen.

9.1.2 Konversationsspezifikationen

Aufbauend auf den Inhaltsspezifikationsobjekten bietet das Modul *tbcSpec.tm* Konstruktoren für die Erzeugung von Anfrage-, Dialog- und Konversationsspezifikationen.

Ein Konversationsspezifikationsobjekt wird mit der Funktion *tbcSpec.new(...)* aus einer anzugebenden Menge von Dialogspezifikationsobjekten erzeugt. Jede dieser, mit *tbcSpec.newDialog(...)*

[2] Der Typ der in der Menge enthaltenen Objekte wird durch das zugehörige Elementtyp-Spezifikationsobjekt festgelegt.

erzeugten, Dialogspezifikationsobjekten enthält wiederum eine Menge benannter Inhaltsspezifikationsobjekte und eine Menge erlaubter Anfragen. Ein Dialogspezifikationsobjekt ist dabei als Spezifikation des initialen Dialogs der beschriebenen Konversation zu kennzeichnen.

9.1.3 Konversationen

Das Modul *tbConv.tm* stellt Funktionen zur Erzeugung von Konversationsobjekten zur Verfügung, welche aus den Konversationsspezifikationsobjekten des Moduls *tbcSpec.tm* generiert werden. Neben diesen Konversationsobjekten definiert das Modul die Schnittstelle zu Dialogobjekten, die in den Konversationsobjekten enthalten sind.

Jedes Konversationsobjekt besitzt einen eigenen, innerhalb von TL typisierten Zustand, über den ein externer Gesprächskontext modelliert werden kann. Darüber hinaus ist das Objekt selbstbeschreibend, da es den Zugriff auf das zugrundeliegende Spezifikationsobjekt der Konversation erlaubt.

Weiter bietet ein Konversationsobjekt über die Operation *newDialog(...)* die Möglichkeit, einen neuen Dialog gemäß der zugehörigen Dialogspezifikation zu erzeugen. Dieser neue Dialog wird gleichzeitig der aktuelle Dialog der Konversation. Eine Referenz auf das letzte erzeugte Dialogobjekt kann mit der Operation *currentDialog(...)* ermittelt werden.

Darüber hinaus führt ein Konversationsobjekt eine lokale Historie über Dialogobjekte. Auf sie kann mittels der Operationen *saveDialog(...)* und *copyHistoryDialog(...)* und zugegriffen werden. Die Identifikation eines Dialogs geschieht über dessen Bezeichner. Die Operation *saveDialog(...)* speichert ein angegebenes Dialogobjekt auf dem Stapel. Die Operation *copyHistoryDialog(...)* erzeugt eine Kopie des jüngsten Dialogobjekts im Stapel mit der angegebenen Bezeichnung.

Ein Konversationsobjekt wird über die Funktion *new(...)* aus dem angegebenen Konversationsspezifikationsobjekt erzeugt und an einen initialen Gesprächskontext gebunden. Ein Dialogobjekt, welches mit der Operation *newDialog(...)* eines Konversationsobjektes generiert wird, erlaubt den Zugriff auf dessen Inhaltsobjekte und das zugehörige Spezifikationsobjekt.

Desweiteren sind in dem Modul *tbConv.tm* eine Reihe von im *Business Conversation*-Modell vorgesehene, systemimmanente Dialoge, Anfragen und Spezifikationsobjekte definiert. Neben der initialen Anfrage, die von jedem Kunden zu seinem Dienstleister gesendet wird und die daher nicht Teil einer Gesprächsspezifikation zwischen diesen sein kann, ist das der initiale Dialog und ein spezieller Ausnahmedialog, der *breakdownDialog*. Diese Objekte sind ebenfalls nicht Teil einer anwendungsabhängigen Konversationsspezifikation. Ihre Struktur kann jedoch ermittelt werden, da alle Dialogobjekte – auch der initiale und der Ausnahmedialog – über ihre Spezifikationsobjekte selbstbeschreibend sind.

9.2 Nachrichtenorientierte Kommunikation

Grundlage des Agentensystems ist das Nachrichtensubsystem einer Domäne, welches den Nachrichtenverkehr zwischen allen kommunizierenden Komponenten abwickelt.

Gemäß dem Architekturmodell aus Abbildung 5.1 ist dieses Nachrichtensubsystem aus der Sicht der Agenten eine ubiquitäre Ressource, kann daher in jeder Domäne vorausgesetzt werden.

Das System ist bindungsarm und mobilitätsorientiert entworfen. Es ist daher isolierbar und relinkfähig, d.h alle vom System erzeugten Objekte enthalten keine direkten Referenzen auf Objekte anderer Systemnutzer. Dies gilt insbesondere für die wichtigsten vom Nachrichtensystem verwendeten TL-Objekte vom Typ : *Port*, welche die symbolisch adressierbaren Kommunikationsendpunkte der Systemnutzer repräsentieren. Das System verarbeitet statisch typisierte Nachrichten, deren Struktur in dem TL-Interface *ITCMessage.ti* offengelegt werden muß.

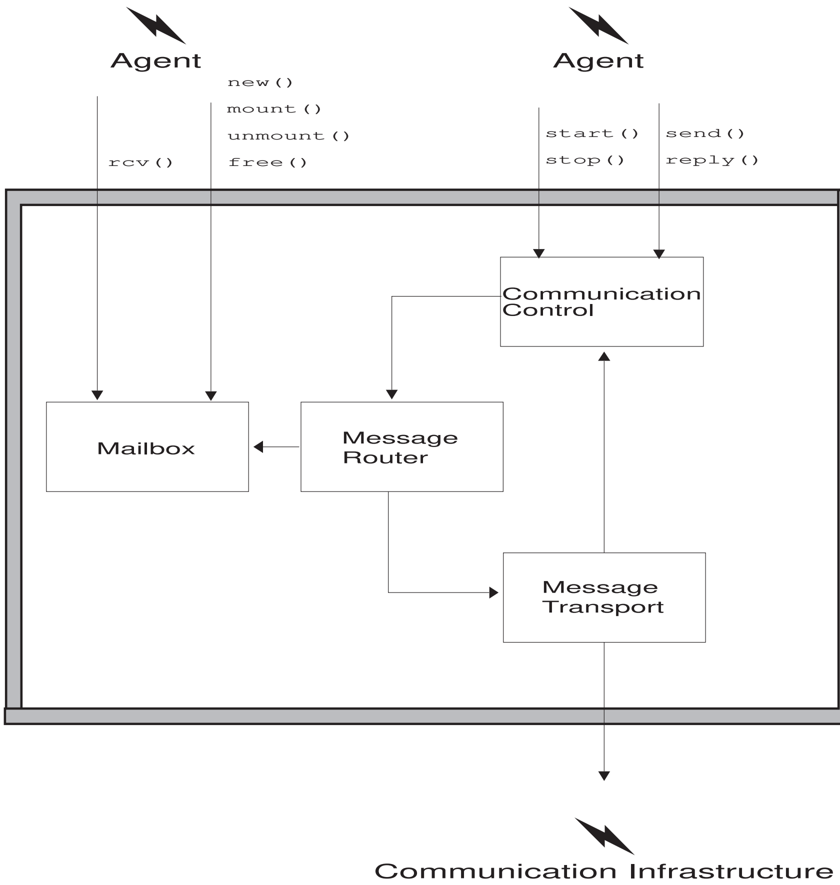

Abbildung 9.1: Modell des Nachrichtensubsystems

Die Struktur des als aktive Komponente ausgelegten Nachrichtensubsystems ist in Abbildung 9.1 dargestellt.

Die Architektur ist in Anlehnung an das *Message Handling System (MHS)* [KV89] der ISO-Empfehlung entworfen worden. Die wesentlichen Eigenschaften des realisierten Nachrichtensubsystems sind:

- Bereitstellen einer Abstraktion von lokaler und entfernter Kommunikation für dessen Nutzer.
- Abstraktion von der konkreten Netzwerkumgebung, den Netzwerkprotokollen und der Netzwerkprogrammierung.
- Unterstützung der Kommunikation mit Anwendungen, die temporär nicht erreichbar sind.

Die Abstraktion von lokaler und entfernter Kommunikation wird durch die zentrale Komponente des Nachrichtenrouters (engl. *Message Router*) erreicht, welcher den Fluß einer Nachricht anhand der angegebenen Zieladresse bestimmt. Im Falle eines lokalen Empfängers wird die Nachricht direkt an den angegebenen Kommunikationsendpunkt zugestellt und ggf. der Besitzer dieses Endpunktes benachrichtigt.

Ist ein entfernter Kommunikationsendpunkt Ziel der Nachricht, so wird diese über die Transportkomponente (engl. *Message Transport*) an das entsprechende Nachrichtensubsystem übermittelt und dort in Empfang genommen. Dabei übernimmt die Transportkomponente auch den Empfang der Nachrichten, die über das Rechnernetz zugestellt werden. In einem solchen Fall ist der Routingkomponente, über die Kontrollkomponente, zur (lokalen) Weiterleitung der Nachricht involviert.

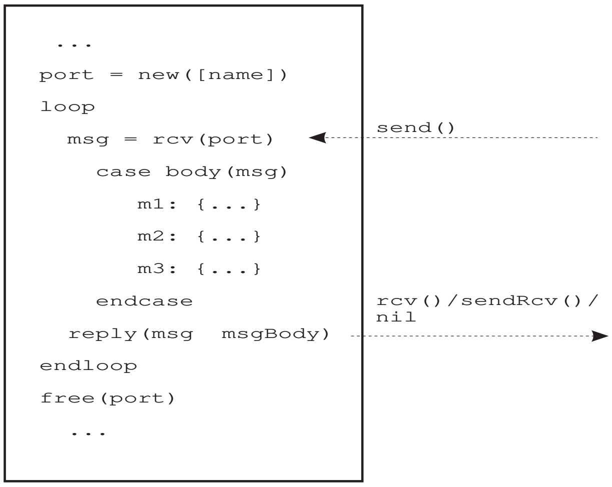

Abbildung 9.2: Benutzung des Nachrichtensubsystems

Alle empfangenen Nachrichten werden in Briefkästen (engl. *mailbox*) verwaltet, die ihren Besitzern nur indirekt und ohne eine Objektbindung zugeordnet sind. Ein Besitzer kann so den Ort eines Briefkastens verlassen und trotzdem weiter Nachrichten empfangen. Diese Nachrichten können später bei seiner Rückkehr entgegengenommen werden. Die Briefkästen werden über Nachrichtenwarteschlangen modelliert, die die Empfangsreihenfolge zweier Nachrichten erhalten.

Die Kommunikationskomponente ist nicht auf eine Nutzung durch Agenten oder das Agentensystem festgelegt. Normale Tycoon-Threads sind ohne besondere Maßnahmen in der Lage, die Funktionen des Nachrichtensubsystems zu verwenden und über dieses miteinander zu kommunizieren.

Das durch die Anwendungsprogrammierschnittstelle (API) definierte, generelle Anwendungsschema ist in Abbildung 9.2 dargestellt. Die vollständige Schnittstelle zum Nachrichtensubsystem ist im Anhang A.2 wiedergegeben.

Das Nachrichtensubsystem stellt Funktionen sowohl zum synchronen, als auch zum asynchronen Nachrichtenaustausch zur Verfügung. Dabei wird zwischen privaten und öffentlichen Kommunikationsendpunkten unterschieden. Nur die öffentlichen Kommunikationsendpunkte werden benannt und können daher symbolisch adressiert werden. Nachrichten, die an einen privaten Port versendet werden sollen erfordern, daß Sender und Empfänger ein Port-Objekt teilen[3]. Die Port-Objekte fungieren als weltweit eindeutige Bezeichner eines Kommunikationsendpunktes und können beliebig kopiert oder auch als Teil einer Nachricht versendet werden.

Das realisierte Nachrichtensubsystem, d.h. insbesondere die Transportkomponente, basiert auf TCP/IP Sockets zum Empfang und zum Versand von Nachrichten, jeweils von bzw. an andere Nachrichtensubsysteme im Rechnernetz. Dabei wird für den Zeitraum des Nachrichtversands eine Socketverbindung zwischen den betroffenen Nachrichtensubsystemen hergestellt. Die ggf. erforderliche IP-Adressauflösung geschieht in der aktuellen Implementation durch den Namensdienst des Internet, den *Domain Name Service* (DNS).

Eine Adressauflösung mittels der weltweit eindeutigen Identifikatoren der Kommunikationsendpunkte ist bislang nicht vorgesehen, jedoch lassen sich die in [Gö96] beschriebenen Mechanismen transparent einsetzen.

[3] Eine Kopie des Objektes ist ebenfalls verwendbar.

9.2.1 Mediatoren

Ein *Mediator* ist ein spezieller, optionaler Systemdienst des Nachrichtensubsystems. Auf jedem Knoten, auf dem ein Nachrichtensubsystem eingerichtet ist, kann ein solcher Mediator konfiguriert werden. Dieser stellt sich gegenüber dem Nachrichtensubsystem als normaler Klient dar, benutzt aber einen speziell vereinbarten Kommunikationsendpunkt für den Empfang von Nachrichten. Hat das Nachrichtensubsystem einen lokalen Mediator erkannt, so wird dieser beim Routing aller im folgenden über die Kommunikationskomponente versendeten Nachrichten berücksichtigt. Dabei erhält der Mediator jede Nachricht, die an einen öffentlichen Kommunikationsendpunkt adressiert ist, bevor diese an den Empfänger ausgeliefert wird. Er hat dann das Recht, die Nachricht zu manipulieren, zu duplizieren, zu unterdrücken, umzuleiten oder unverändert zustellen zu lassen.

Der für das Agentensystem beispielhaft implementierte Mediator verwaltet priorisierte Dienste, sog. Mediator-Services. Diese bilden eine nach absteigender Priorität sortierte Dienstkette, bei der jeder enthaltene Dienst eine Nachricht maximal einmal sieht. Nur, wenn alle Dienste die Nachricht *sm* mit *sm.delegateMessage()* an den jeweils nächsten Dienst in der Kette weiterleiten, wird diese dem Empfänger zugestellt.

Ein Mediator wird auf einem Knoten mit

```
=> let m = mediator.new()
```

eingerichtet. Danach können beliebige Mediatordienste mittels der Operation *m.addService(...)* am Mediator registriert werden.

Sowohl der lokale als auch der entfernte Nachrichtenverkehr unterliegt dem Mediator eines Knotens. Daher wird eine Nachricht, die nicht lokal zugestellt werden kann, zweimal betrachtet: einmal von dem lokalen Mediator und einmal in dem entfernten. Mediator-Dienste besitzen hierbei eine Sonderrolle die es ihnen erlaubt, Nachrichten zu versenden, welche nicht durch den lokalen Mediator bearbeitet werden.

Das Konzept der Mediator-Dienste entspricht dem der ODP-Interzeptorobjekte und ermöglicht es an zentraler Stelle, nachträglich Systemdienste einzurichten, die das Verhalten einer Domäne als Ganzes betreffen. Sie eignen sich daher zur Realisierung von Loggingdiensten, Protokollumsetzern, Routern oder zur zentralen Einrichtung von Sicherheitsmechanismen.

9.3 Agentensystem

Ähnlich dem Nachrichtensubsystem wird das Agentensystem einer Domäne durch ein ubiquitäres Modul, das Modul *agent.tm*, repräsentiert, welches die Administrations- und Applikationsschnittstelle zur Ausführungsumgebung gegenüber den Agenten bildet. Die vollständige Schnittstelle ist im Anhang A.3 wiedergegeben.

Das Agentensystem einer Domäne wird über die Operation *agent.startDomain()* auf dem aktuellen Knoten etabliert und ist danach als Migrationsziel für andere Agenten erreichbar. Die Komponenten des Agentensystems sind in Abbildung 9.3 dargestellt.

Alle Agenten und Orte einer Domäne werden in entsprechenden Verzeichnissen der Domäne verwaltet. Diese Verzeichnisse bilden die Grundlage der verschiedenen Dienste, wie sie vom Agentensystem angeboten werden. Neben der Verwaltung dieser Verzeichnisse lassen sich die Aufgaben des Agentensystems in drei getrennte Kategorien einteilen:

- ▷ Agentenerzeugung bzw. Umgebungssteuerung
- ▷ Kooperationssteuerung

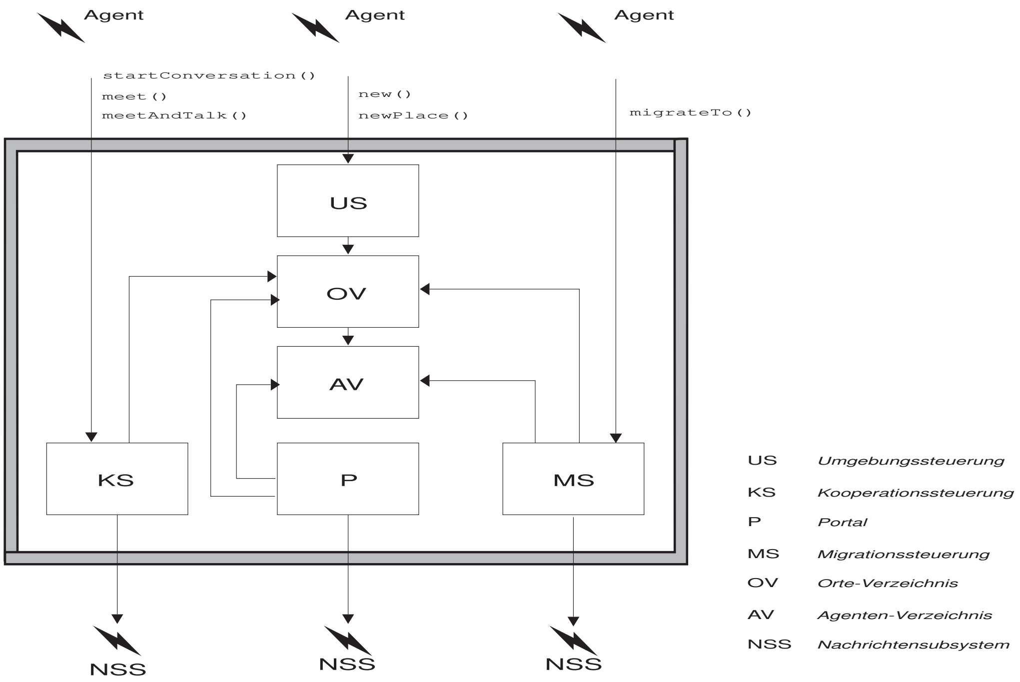

Abbildung 9.3: Komponenten der Ausführungsumgebung

▷ Migrationssteuerung und Portal

Neben den speziellen Aufgaben der Komponenten aus jeder dieser Kategorien, die in den folgenden Abschnitten näher dargestellt werden, sorgt das Agentensystem für die korrekte Abwicklung der jeweiligen Protokolle auf der Basis von Nachrichten. Diese Nachrichten werden zwischen den beteiligten Agenten und dem Agentensystem, über das Nachrichtensubsystem der Domäne, ausgetauscht.

Das generelle Strukturmodell der Objekte und ihrer Beziehungen, die durch das Agentensystem verwaltet bzw. verwendet werden, ist in Abbildung 9.4 dargestellt.

Die zentralen Objekte dieser Darstellung sind der *Agent* und das Bindungsobjekt (engl. *Binding*). Objektreferenzen an Agentenobjekte sind nur innerhalb eines Agentensystems definiert. Nach außen, d.h. gegenüber Agenten bzw. an der Schnittstelle zum Agentensystem, wird ein abstrakter Identifikator (*AgentHandle*) verwendet. Dieser ist weltweit eindeutig und wird über das Agentensystem, genauer über die dort verwalteten Bindungsobjekte, den konkreten Agentenobjekten zugeordnet. Auf diese Weise können diese indirekten Referenzen auf Agenten verwendet werden ohne deren Autonomie zu beeinträchtigen. Das Agentensystem verbirgt die konkreten Objektreferenzen auf Agenten vollständig durch seine Programmierschnittstelle. Teil des Identifikators eines Agenten ist zum einen sein Name, der der anwendungsnahen Bezeichnung des Agenten dient, und zum anderen der Identifikator seines Erzeugers und die Adresse des Ortes, an dem er erzeugt wurde.

Das Objekt Domäne (engl. *Domain*) repräsentiert in dieser Darstellung einen systemimmanenten Ort, welcher automatisch beim Start des Agentensystems eingerichtet wird. Dieser Ort bildet die Wurzel der in der Domäne vorhandenen Ortehierarchie. Er beherbergt die Administrationsschnittstelle der Domäne und hat als einziger Ort, durch seine Verankerung im Agentensystem, direkten Zugriff auf die Datenstrukturen und Verzeichnisse des Agentensystems. Auf diese Weise können sich Konversationen dieses Agenten auf den aktuellen Status des Agentensystems beziehen.

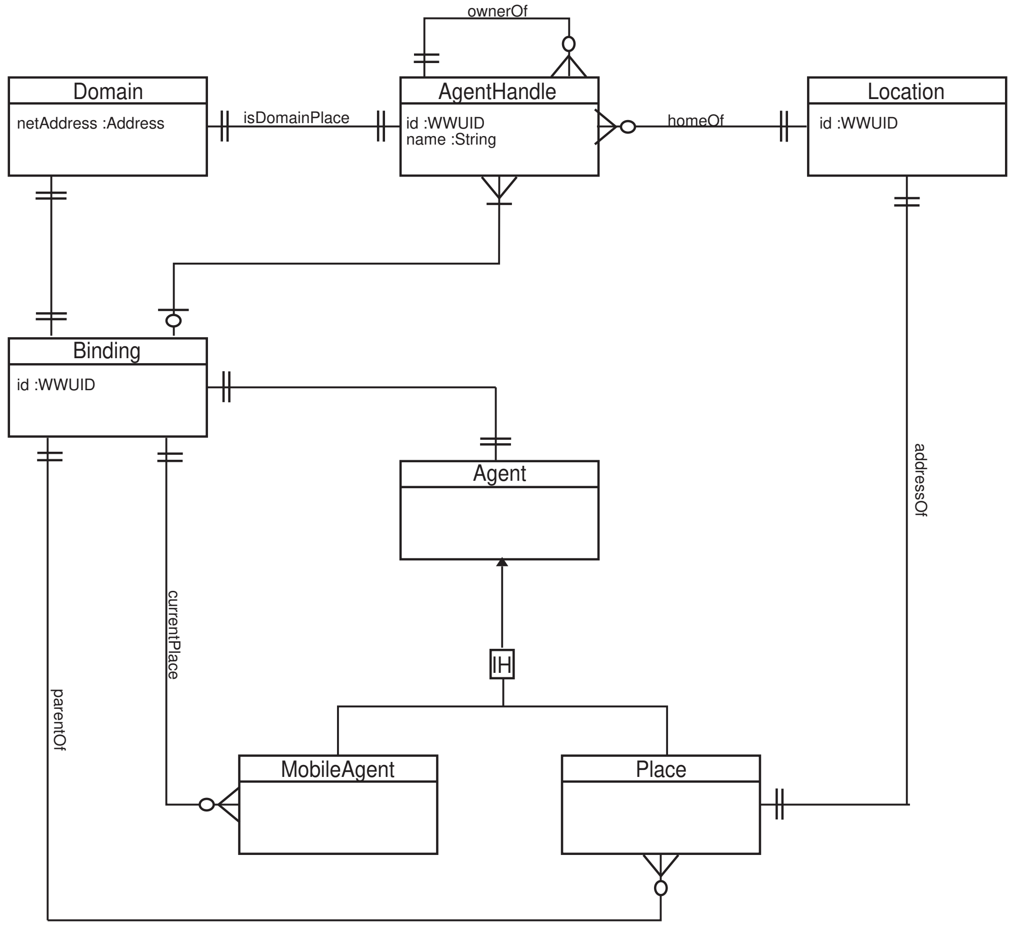

Abbildung 9.4: Strukturmodell der Objekte des Agentensystems

9.3.1 Agentenerzeugung

Die Erzeugung von Agenten, sowohl von mobilen Agenten als auch von immobilen Orten mit ihren Orte-Wächtern, wird durch das Agentensystem vorgenommen. Eine Beschreibung der Komponenten eines solchen *Agentenskeletts* wird im Abschnitt 9.4 gegeben. Wesentlich an dieser Stelle ist, das jeder Agent aus einem Anwendungsteil und einem Systemteil besteht. Der Systemteil ist für alle Agenten gleich und beinhaltet die Protokollabwicklungskomponente, die die Verbindung zwischen dem Anwendungsteil und dem Agentensystem darstellt.

Ein neuer Agent wird durch die Operation *agent.new(...)* erzeugt. Das zugehörige Protokoll, welches vom Agentensystem durchgeführt wird, besteht in den Schritten:

1. Ermittlung des Ortes, in dem der Agent erzeugt werden soll.
2. Erzeugung des Agentenskeletts. Mit dem Systemteil des Agenten können ab jetzt Nachrichten ausgetauscht werden. Der Anwendungsteil des Agenten ist jedoch noch inaktiv.
3. Erzeugung des Bindings zwischen Agent und Agentensystem. Das Binding enthält bereits alle Informationen zu dem neuen Agenten.
4. Zustimmung des Ortes einholen, welcher den Agenten aufnehmen soll. Dies führt zu einer Aktivierung der *incomingAgent(...)* Methode des zugehörigen Orte-Wächters, der auf diesem Wege den eindeutigen Bezeichner des neuen Agenten erhält.

5. Wenn der Orte-Wächter der Erzeugung zustimmt, wird der Agent als Besucher des Ortes geführt, ansonsten wird das Binding bzw. das Agentenskelett wieder zerstört und das Protokoll endet mit einer Ausnahme auf der Seite des Agenten[4], welcher die *agent.new(...)* Operation ausgelöst hat.

6. Aktivierung des Anwendungsteils des Agenten durch die Ausführung seiner *birth(...)* Methode.

7. Rückgabe des weltweit eindeutigen Bezeichners des neuen Agenten.

Die Erzeugung eines Ortes folgt demselben Protokoll. Die frühe Generierung des Agentenskeletts und des Bindings hat zur Folge, daß ein Agent gegebenenfalls wieder zerstört werden muß, bietet jedoch die Möglichkeit der aktiven Beteiligung des Agenten an der Protokollabwicklung.

Die Verzeichnisse des Agentensystems werden mit der Instantiierung des Bindungsobjektes (engl. *Binding*) aktualisiert. Daher kann der Orte-Wächter des Ortes, welcher den neuen Agenten aufnehmen soll, Informationen über diesen ermitteln, noch bevor er vom Agentensystem vollständig aktiviert wurde.

9.3.2 Kooperationssteuerung

Die Kooperationssteuerung des Agentensystems hat die Aufgabe, die Kooperation zwischen Agenten zu koordinieren. Zu diesem Zweck implementiert es die Protokolle für die Operationen *agent.meet(...)* und *agent.meetAndTalk(...)*.

Das grundlegende Protokoll für Agententreffen *meet*, in dessen Verlauf Agenten Referenzen auf fremde Agenten erhalten können, dient der Kontaktaufnahme eines Agenten mit einem anderen Agenten am selben Ort über dessen *Namen*:

```
=> let wspAgent =
       agent.meet(self "WebSiteProfiler" wspSpec timeOutSecs)
```

Der mit *self* bezeichnete Agent initiiert das Treffen-Protokoll mit dem Ziel, eine Referenz auf einen Agenten mit dem Namen "WebSiteProfiler" zu erhalten. Die Kooperationssteuerung durchläuft zu diesem Zweck folgendes Protokoll:

1. Ermittlung eines Agenten mit dem gesuchten Namen. Das Agentensystem versucht, innerhalb des Ortes an dem sich der suchende Agent a_1 befindet und innerhalb der mit *timeOutSecs* nach unten beschränkten Zeitspanne, einen Agenten mit dem angegebenen Namen zu ermitteln.

2. Kann ein solcher Agent a_2 ermittelt werden, so wird diesem der Wunsch nach einem Treffen mit Agent a_1 mitgeteilt. Andernfalls wird eine Ausnahme in dem Agenten a_1 ausgelöst.

3. Im Agent a_2 wird die *meet(...)* Methode ausgeführt, mit der er dem Wunsch nach dem Treffen mit a_1 zustimmen oder diesen ablehnen kann. Lehnt er ihn ab, so wird im Agenten a_1 eine Ausnahme ausgelöst. Im Falle einer Zustimmung kann der Agent auch einen weiteren Agenten a_3 bestimmen, welcher das Treffen mit a_1 behandeln soll. Geschieht dies, so wird dieser Schritt iterativ ausgeführt, bis ein Agent a_n entweder das Treffen mit a_1 endgültig ablehnt oder es selbst annimmt. Jede Weiterleitung wird wie ein gesondertes Treffen behandelt, so daß die im *timeOutSecs* vorgegebene Zeitspanne für jedes Treffen erneut zur Verfügung.

4. Rückgabe des weltweit eindeutigen Bezeichners des ermittelten Agenten.

[4] Agenten und Orte können auch von jedem Tycoon-Thread erzeugt werden.

Kernpunkt dieses Protokolls ist die Ausführung der *meet(...)* Methode im ermittelten Agenten. Auf diese Weise erhält der Agent Kenntnis von dem suchenden Agenten (d.h. dessen Identifikation) und von der Spezifikation einer Konversation, die dieser mit dem Agenten zu führen wünscht.

Die Implementation von *agent.meetAndTalk(...)* ist eine Spezialisierung der geschilderten *agent.meet(...)* Operation. Statt im letzten Schritt den Identifikator des ermittelten Agenten zurückzugeben, initiiert die Kooperationssteuerung gleich eine *synchrone* Konversation zwischen dem ermittelten Agenten und dem suchenden Agenten, zu der im *meetAndTalk(...)* angegebenen Konversationsspezifikation. Dies zwingt den suchenden Agenten in die Rolle des Kunden und erfordert vom ermittelten Agenten die Rolle des Dienstleisters bezüglich der Konversationsspezifikation.

Die Abwicklung des Verarbeitungsmodells einer begonnenen *Business Conversation* wird ohne die Mitwirkung der Kooperationssteuerung des Agentensystems zwischen den beteiligten Agenten durchgeführt. Die zuständigen Protokollkomponenten der Agenten sind jedoch vollständig im Agentenskelett verankert, welches wiederum durch das Agentensystem erzeugt worden ist.

9.3.3 Migrationssteuerung und Portal

Die Migration von Agenten zwischen Orten, sowohl innerhalb einer Domäne als auch in den Bereich einer anderen Domäne hinein, wird durch die Komponente der Migrationssteuerung durchgeführt. Ist bei einer Migration ein Wechsel der Domäne erforderlich, so wird in der entfernten Domäne zusätzlich deren Portaldienst involviert, welcher den reisenden Agenten entgegennimmt und in der neuen Domäne reaktiviert. Ist kein Domänenwechsel erforderlich, so ist für die Migration des Agenten lediglich eine Änderung in den lokalen Verzeichnissen des Agentensystems erforderlich. Im Einzelnen wird bei jeder Agentenmigration, ausgelöst durch die Operation *agent.migrateTo(...)*, von der Migrationssteuerung folgendes Protokoll verwendet:

1. Ermittlung, ob lokale Migration vorliegt. Diese wird durch das lokale Migrationsprotokoll implementiert, welches vom Portal (s.u.) verwendet wird. Die weiteren Schritte sind nur bei einem Wechsel der Domäne erforderlich.

2. Start eines temporärem Threads, dem Versanddienst, für die Kommunikation mit dem entfernten Portal. Das Agentensystem erhält keine Objektreferenz auf diesen Thread, so daß keine unerwünschten Bindungen an den zu versendenden Agenten im Agentensystem verbleiben können.

3. Vorbereitung des Agenten. Der Systemteil des Agenten wird aufgefordert, sich auf den bevorstehenden Versand vorzubereiten. Dabei werden insbesondere alle Bindungen an das Nachrichtensubsystem durch den Agenten isoliert und der Agent in einen definierten Ruhezustand überführt.

4. Versand. Der Agent wird als Teil einer Nachricht an das entfernte Agentensystem (genauer dessen Portal) versendet. Der lokale Versanddienst wartet auf die Bestätigung der erfolgreichen Reaktivierung des Agenten durch das entfernte Portal.

5. Abschluß der Migration. Wird die erfolgreiche Migration durch das entfernte Portal bestätigt, so reaktiviert der Versanddienst den Systemteil der nun überflüssigen, lokalen Kopie des Agenten. Der Agent terminiert daraufhin ordnungsgemäß. Dem letzten Aufenthaltsort des Agenten wird das Verlassen des Agenten über dessen *outgoingAgent(...)* Methode signalisiert. War die Migration nicht erfolgreich, so wird der lokalen Kopie des Agenten dies durch den Versanddienst mitgeteilt und der Agent bleibt erhalten.

Das entsprechende Protokoll auf der Seite des Portaldienstes, welcher beim Start jeder Domäne eingerichtet wird, ist wie folgt implementiert:

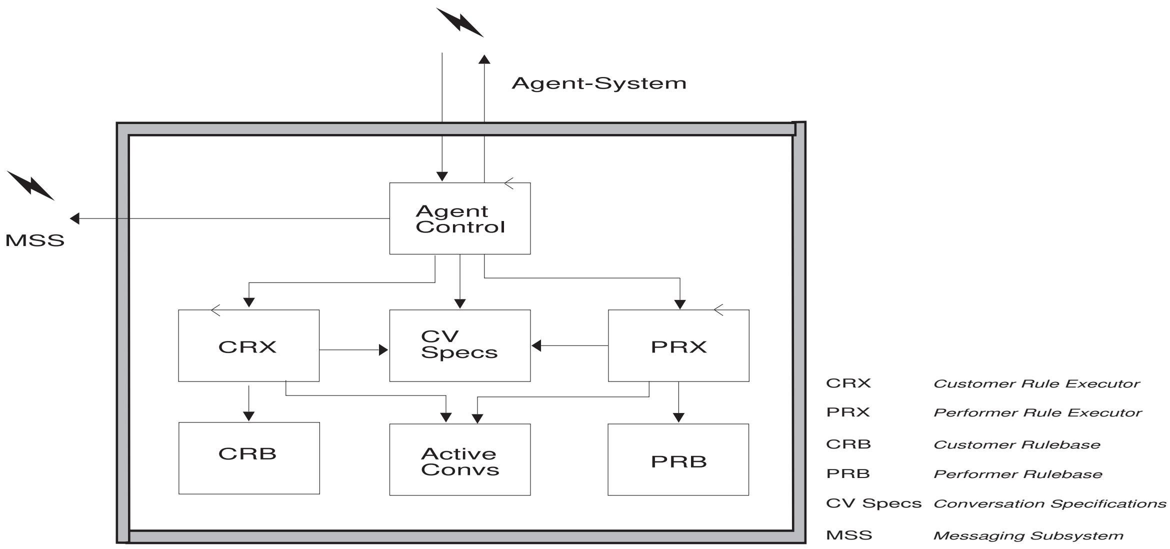

Abbildung 9.5: Übersichtsmodell eines Agenten

1. Eintreffen eines Agenten. Nachdem der Agent a_1, der sich im definierten Ruhezustand befindet, an dem Portal eintrifft, wird der Systemteil des Agenten vom Portal reaktiviert.

2. Lokale Migration. Das Portal holt die Zustimmung des Wurzelortes seiner lokalen Ortehierarchie zum Zutritt des fremden Agenten ein. Zu diesem Zweck löst es in dem entsprechenden Orte-Wächter die *incomingAgent(...)* Methode aus, über die der Wächter Kenntnis von dem Agenten a_1 erhält. Stimmt der Orte-Wächter dem Eintritt zu, so ändert das Portal die lokalen Verzeichnisse des Agentensystems. Ist das Reiseziel des Agenten ein Ort unterhalb der Wurzel, so führt das Portal diese lokale Migration iterativ aus. Immer, wenn dabei ein Ort verlassen wird, signalisiert das Portal dem entsprechenden Wächter dieses Ereignis über die *outgoingAgent(...)* Methode.

3. Abschluß der Migration. Hat jeder Ort auf dem Weg des Agenten Einlass gewährt, so bestätigt das Portal dem entfernten Versanddienst den erfolgreichen Domänenwechsel und reaktiviert den Anwendungsteil des Agenten. Andernfalls erhält der entfernte Versanddienst eine negative Bestätigung und die lokale Kopie des Agenten wird vernichtet.

Die Migration wird in dieser Implementation nicht transaktional gesichert, so daß im Falle eines Domänenausfalls oder einer gestörten Kommunikation während der Migration sowohl Dubletten eines Agenten auftreten können als auch ein Agent verloren gehen kann. In [Mat96] sind jedoch Mechanismen beschrieben, mit denen ein zweiphasiges Bestätigungsprotokoll [GR93] zur Sicherung der Migration gegen Systemausfälle, unter Verwendung des Tycoon-Systems, eingesetzt werden kann.

9.4 Agenten

Agenten bestehen aus einem Systemteil, der durch das Agentensystem generiert wird und einem Anwendungsteil, der über eine Parametrisierung zum Zeitpunkt der Generierung eines Agenten bestimmt wird.

Abbildung 9.5 zeigt das Übersichtsmodell der Komponenten, die den Systemteil jedes Agenten bilden.

Der Systemteil eines Agenten hat die Aufgabe, die Verbindung zwischen dem Anwendungsteil und dem Agentensystem bereitzustellen. Auf diese Weise ist es möglich, den Anwendungsteil eines Agenten vollständig von der Abwicklung der einzelnen Protokolle des Agentensystems abzukoppeln. Auf der anderen Seite kann sich jedes Agentensystem seinerseits auf die korrekte Abwicklung der einzelnen Protokollschritte verlassen, da diese nicht durch die konkrete Verwendung eines Agenten beeinflußt werden können.

Ein Agent besteht im wesentlichen aus drei aktiven Komponenten:

1. Die Kontrollkomponente.
2. Regelausführungskomponente für Dienstleisterrollen (*PRX*).
3. Regelausführungskomponente für Kundenrollen (*CRX*).

Die Kontrollkomponente stellt das Bindeglied zwischen dem Agentensystem und dem Agenten dar. Sie besitzt einen öffentlich zugänglichen Kommunikationsendpunkt, über den jedwede Interaktion der Außenwelt mit dem Agenten stattfindet.

Die zur Implementation des Verarbeitungsmodells der *Business Conversations* notwendige Regelauswertung geschieht getrennt von der Kontrollkomponente in der jeweiligen Regelauswertungskomponente. Die Auswertung der Bearbeitungsregeln erfolgt in einem eigenen Tycoon-Thread, da diese Regeln zum Anwendungsteil des Agenten gerechnet werden müssen. Daher kann das Verhalten dieser Regeln nicht vorherbestimmt werden, insbesondere kann durch sie eine Migration des Agenten veranlaßt werden. In deren Verlauf ist es gemäß dem bereits geschilderten Migrationsprotokoll notwendig, den Kontrollfluß eines Agenten in einen definierten Ruhezustand zu überführen, um ihn dann versenden zu können. Durch die Realisierung in voneinander getrennten Threads ist es direkt möglich, den Regelausführungspunkt innerhalb der Migrationsoperation zu fixieren, ohne dabei den Systemteil des Agenten zu inaktivieren. Dieser kann so für die weitere Protokollabwicklung, speziell bei der schrittweisen Reaktivierung des Agenten in der Zieldomäne, genutzt werden.

Der Bedarf für zwei Regelauswertungskomponenten begründet sich durch die im *Business Conversation* Modell definierte Konversationsform der sekundären Konversation zwischen zwei Agenten. Diese synchrone, eingeschobene Konversation erfordert einen temporären Rollenwechsel des Agenten für die Dauer der eingeschobenen Konversation. Der momentane Ausführungspunkt der initiierenden Bearbeitungsregel kann durch die Kontrollkomponente fixiert werden, indem der zugehörige Thread in den Zustand *suspended* überführt wird. Die Regelauswertung für die sekundäre Konversation übernimmt dann der noch verbleibende, aktive Thread. Ist die Konversation beendet, so wird die Bearbeitung der suspendierten Regelauswertung wieder aufgenommen.

Die Regelauswertungskomponenten verwalten die Menge der aktiven Konversationen, die Menge der möglichen Konversationen (über die dem Agenten bekannten Konversationsspezifikationen) und die jeweiligen Bearbeitungsregeln, die zu einer Konversationsspezifikation gehören.

Die Kontrollkomponente hat, parallel zur aktiven Regelauswertung durch die Regelauswertungskomponenten, Zugriff auf die Verzeichnisstrukturen des Agenten. Daher kann ein Agent, auf der Ebene von Systemnachrichten und unabhängig von seinem momentanen Zustand auf der Ebene der *Business Conversations*, Auskunft über seinen aktuellen, internen Zustand geben. Dies ist hilfreich, wenn ein Agent potentiell langandauernden Bearbeitungsregeln ausführt.

Die Abbildung 9.6 zeigt die Objektstrukturmodellierung der Objekte und ihrer Beziehungen, die vom Systemteil des Agenten verwaltet werden.

Ein neuer Agent wird durch das Agentensystem über die Operation *agent.new(...)* erzeugt:

```
=> let wspAgent = agent.new(initialContext
          birth meet die leave enter
```

agent.currentRootPlace() "WebSiteProfiler")

In diesem Beispiel wird ein Agent mit dem Namen "WebSiteProfiler" erzeugt und im Wurzelort der aktuellen Domäne angesiedelt. Der Anwendungsteil des Agenten wird durch die Funktionsparameter *birth(...), meet(...), die(...), leave(...) und enter(...)* bestimmt, die vom Systemteil des Agenten in die entsprechenden Protokolle integriert werden.

Der *initialeContext* eines Agenten ist nützlich, um Bindungen zu aggregieren, welche in allen Protokollmethoden sichtbar sein sollen. Die Aggregation dieser Bindungen in einem expliziten Kontextobjekt trägt dabei zur Strukturierung eines Agenten bei.

Die *birth(...)* Methode des Agenten erlaubt die Initialisierung der kooperativen Fähigkeiten eines Agenten und/oder die Erzeugung von Agenten, die ein Skript abarbeiten. Die übergebene Funktion wird ausgeführt, wenn der Agent durch das Agentensystem erfolgreich im angegebenen Ort erzeugt wurde. Die Anwendung kann dann innerhalb der Funktion die Rollen und Bearbeitungsregeln des Agenten registrieren:

```
let p = agent.addPerformerRole(:Ok  self  wspConvSpec  fun() ok)
    p.set(tbConv.initialDialogName()  tbConv.initialRequest()
       fun(cv :agent.Conversation(Ok)) :tbConv.Dialog  begin
          ...
    end)
```

Die auf diese Weise registrierten Regeln werden in den Kunden- und Dienstleisterregelbasen des jeweiligen Agenten gespeichert. Die Regeln, und damit die Fähigkeiten eines Agenten, werden in der aktuellen Implementation ebenfalls über Nachrichten übergeben. Es ist daher möglich, die Regelbasis eines Agenten nach dessen Erzeugung zu erweitern oder zu ändern und ihn so dynamisch an sich ändernde Anforderungen anzupassen.

Die *die(...)* Methode eines Agenten wird ausgeführt, bevor ein Agent terminiert. Zu diesem Zeitpunkt darf ein Agent weder neue Konversationen eingehen, noch seine Migration veranlassen oder andere Agenten erzeugen. Die Operation dient lediglich einer geordneten Beendigung der Aufgaben des Agenten. Die *die(...)* Methode wird nicht ausgeführt, wenn ein Agent (oder dessen Kopie) im Laufe des Migrationsprotokolls vernichtet werden soll.

9.4.1 Orte

Orte, genauer die Orte-Wächter, sind eine Spezialform der Agenten. Sie unterscheiden sich lediglich in der Parametrisierung bei der Erzeugung durch das Agentensystem. Der strukturelle Aufbau eines Orte-Wächters entspricht dem beschriebenen Agentenskelett.

Ein Ort wird vom Agentensystem z.B. durch

```
=> let redaktion = agent.newPlace(initialContext
       birth  meet  die  incomingAgent  outgoingAgent
       agent.currentRootPlace()  "SpiegelRedaktion")
```

erzeugt. Die Ortehierarchie einer Domäne wird inkrementell aufgebaut, indem ein neuer Ort immer unterhalb der Hierarchiestufe des angegebenen Ortes eingerichtet wird. Bezeichnet der angegebene Agent keinen Ort, so wird stattdessen der Ort verwendet, an dem sich dieser momentan aufhält[5].

[5] Ist dieser jedoch nicht zu ermitteln, so wird der neue Ort nicht erzeugt.

Da Orte immobil sind, besitzt ein Orte-Wächter nicht die Methoden *enter(...)* und *leave(...)*. Stattdessen werden Orte-Wächter vom Agentensystem über die Methoden *incomingAgent(...)* und *outgoingAgent(...)* über das Eintreffen und Verlassen von Agenten in ihren Sichtbarkeitsbereich informiert.

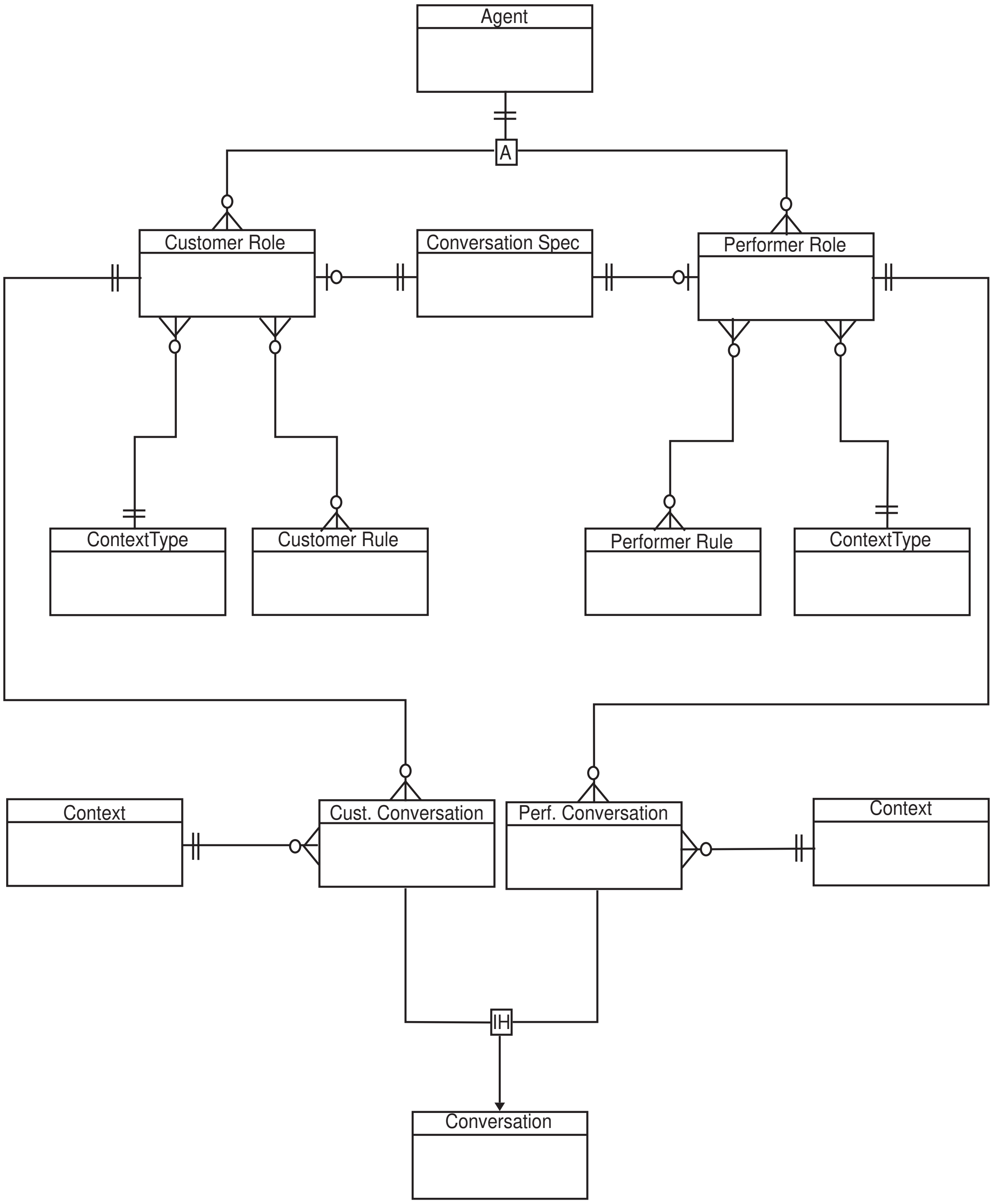

Abbildung 9.6: Objektstrukturmodell eines Agenten

Kapitel 10

Zusammenfassung

Die vorliegende Arbeit entwickelt eine Systemumgebung für agentenbasierte, verteilte Anwendungen und definiert einen konzeptuellen Rahmen für die anwendungsnahe Modellierung von kooperativen, verteilten Informationssystemen.

Vor dem Hintergrund des ODP-Referenzmodells betrachtet, liegen die Hauptbeiträge dieser Arbeit in der Modellierungsunterstützung auf der Ebene der Unternehmenssicht (*Enterprise Viewpoint*) und auf der Ebene der Verarbeitungssicht (*Computational Viewpoint*).

Die Konzepte der Unternehmenssicht *Gemeinschaft* und *Föderation* sind in dem beschriebenen Agentenmodell direkt repräsentiert. Gemeinschaften von kooperierenden Agenten werden durch das Strukturierungsmittel der *Orte* gebildet, die wiederum innerhalb autonomer Domänen angesiedelt sind. Föderationen zwischen diesen Domänen, bzw. den Agenten unterschiedlicher Domänen, sind immer dann anzutreffen, wenn Orte das Eintreten von Agenten in ihren Sichtbarkeitsbereich erlauben und so eine Kooperation zwischen Agenten verschiedener Domänen ermöglichen. Dabei nehmen Agenten innerhalb einer Kooperation feste Rollen ein, die ihr Verhalten für den Kooperationspartner determinierbar machen. Diese Rollen können für jeden Agenten individuell ausgestaltet werden, so daß ein Agent sowohl Dienste anbieten kann, als auch ggf. von anderen Agenten Dienstleistungen in Anspruch nimmt. Dabei wird jede Interaktion zwischen Agenten auf der Grundlage von anwendungsabhängigen Prozßspezifikationen geführt, die den Kooperationspartnern einen geordneten und autonomieerhaltenden Kommunikationsablauf zusichern.

Die Verarbeitungssicht, welche sich mit der Frage der zu verteilenden Objekte einer Anwendung befaßt, wird durch das Agentenmodell stark vereinfacht. Agenten sind der einzige Gegenstand der Verteilung, wobei aus der Sicht eines Agenten die *logische* Verteilung auf Orte einer semantischen Anwendungsstruktur im Vordergrund steht. Von der physischen Verteilung der Orte auf Domänen, und damit auf Netzknoten bzw. Rechner, wird sowohl in der Modellierungsphase als auch in der Realisierungsphase eines Anwendungsagenten vollständig abstrahiert. Durch entsprechende Rekonfigurationen einer Domäne kann die tatsächliche Orteverteilung zur Laufzeit unabhängig von der Agentenpopulation variieren und damit Auslastungskriterien oder veränderte Systemeigenschaften berücksichtigen.

Das im Rahmen der Arbeit entstandene medien- und aktorenunabhängige Kooperationsmodell der *Business Conversations* bestimmt die möglichen Interaktionsformen zwischen Agenten innerhalb des Agentenmodells. Das gewählte Interaktionsmodell ist dabei anwendungsorientiert und basiert auf einer Gesprächsmetapher, den Sprechakten. Es erlaubt die Definition von Konversationsspezifikationen, die allen Interaktionen zwischen Agenten zugrundeliegen. Zusammen mit dem Verarbeitungsmodell der *Business Conversations* bilden sie Interaktionsprotokolle, welche die kooperativen Fähigkeiten instantiierter Agenten beschreiben. Diese Konversationsspezifikationen sind Objekte erster Klasse und als solche zur Laufzeit eines Agenten verfügbar und orthogonal verwendbar.

Ein weiterer Aspekt des Kooperationsmodells ist die vollständige Bewahrung der Autonomie der Gesprächspartner. Erreicht wird dies einerseits durch eine asynchrone Verarbeitung der einzelnen Sprechakte bei dem jeweiligen Partner, auch wenn dieser lokal vorhanden ist. Andererseits werden durch eine strikte Kopiersemantik der verwendeten Dialogobjekte, autonomieverletzende Referenzen zwischen Agenten explizit ausgeschlossen.

Die für die Umsetzung der Orts-, Migrations-, und Lokalitätstransparenzen notwendige Systeminfrastruktur, bezogen auf das ODP-Referenzmodell die Ebene des *Engineering Viewpoints*, konnte ebenfalls im Rahmen der vorliegenden Arbeit geschaffen werden und erweitert bzw. vereinheitlicht die bisher im Tycoon-System vorhandenen Kommunikationsmechanismen. Die entsprechende Kernkomponente der Systemarchitektur ist in Form einer allgemeinen, agentenunabhängigen Kommunikationsunterstützung über ein separates Nachrichtensubsystem realisiert worden.

Die praktische Verwendbarkeit der realisierten Infrastruktur und der Modellierungskonzepte konnte anhand einer Beispielanwendung evaluiert werden. Dabei bot die Anwendungsdomäne der Erhebung von Nutzungsstatistiken für Werbeangebote im Internet WWW ein durchaus realistisches Szenario für agentenbasierte verteilte Datenbanken. Realistisch nicht zuletzt daher, weil die Informationsquellen ein hohes Maß an Autonomie voraussetzen, die nur durch föderative Strukturen korrekt abgebildet werden.

10.1 Stand der Implementation

Die prototypische Implementation der in der Arbeit dargestellten Systemkomponenten wurde abgeschlossen und die Beispielanwendung zu Demonstrationszwecken realisiert. Es hat sich im Verlauf der Programmiearbeiten gezeigt, daß die orthogonalen Verteilungseigenschaften der Implementationssprache TL und des Tycoon-Systems eine gute Basis für die Umsetzung der definierten Konzepte darstellen. Insbesondere die vorhandenen Bibliotheken zur Synchronisation nebenläufiger Aktivitäten, das dynamische Linken und die TCP/IP-Anbindung waren für die Arbeit unverzichtbare Grundlage.

Das Agentensystem selbst besteht aus ca. 2000 Zeilen TL-Kode. Hierin enthalten sind jedoch schon die Skelett-Implementationen für Agenten und Orte-Wächter und der beschriebene Portal- und Migrationsdienst. Das Nachrichtensubsystem benötigt zusätzlich ca. 1000 Zeilen TL-Kode. Für die Implementation des *Business Conversation*-Modells, d.h. der Spezifikations- und Objektgeneratoren, waren lediglich ca. 450 Zeilen erforderlich.

Die tatsächliche Größe der übertragenen Agenten während der Migration auf einen anderen Rechnerknoten variiert je nach Anwendung stark. Jedoch konnte durch die Beschränkung der zur Implementation verwendeten Datenstrukturen und durch den gezielten Einsatz von ubiquitären TL-Bibliotheken das Volumen der Agenten aus der Beispielanwendung auf ca. 70 KByte beschränkt werden.

10.2 Ausblick

Agentenbasierte verteilte Anwendungen stellen eine Alternative zu den klassischen RPC-basierten Client/Server Architekturen dar. Sie bieten insbesondere Vorteile im Hinblick auf die anwendungsnahe Modellierung von Systemen und der direkteren Umsetzung in Konzepte des Programmiermodells. Das im Rahmen dieser Arbeit entwickelte Agentenmodell und die zugehörige Systemumgebung kann dabei als Ausgangspunkt für weitere Arbeiten auf dem Gebiet der agentenbasierten, kooperativen Informationssysteme dienen.

Für den Aufbau eines agentenbasierten, offenen Dienstemarkts sind Komponenten, die den Handel mit Dienstleistungen wiederum als eigene Dienstleistung anbieten (sog. Trader oder Broker

[PSW96]), notwendig. Das Agentensystem muß hierfür nicht um spezielle Systemkomponenten erweitert werden, sondern kann vielmehr auf der Anwendungsebene entsprechende Agenten und/oder Orte anbieten. Diese können dann mit Konversationsspezifikationen anderer Agenten handeln und diese weitervermitteln. Weitere aus der Sicht kommerziell relevanter Anwendungen wichtige Standarddienste sollten die Visualisierung von Konversationen bzw. Konversationszuständen ermöglichen und die verbindliche Protokollierung von Konversationsverläufen (durch transparente Notariatsdienste) anbieten.

Unabhängig vom Agentenmodell dieser Arbeit kann das System der *Business Conversations* ebenfalls weiter ausgebaut werden. An dieser Stelle sind Grundlagenarbeiten möglich, die sich mit speziellen Aspekten der Konformanzüberprüfung von Konversationsspezifikationen oder der Wiederverwendung fremder Spezifikationen durch eigene Spezifikationen befassen. Desweiteren kann die Entwicklung von Konversationsspezifikationen mit Hilfe geeigneter grafischer Editoren bzw. Generatoren noch unterstützt werden.

Für das Agentensystem selbst sind auch eine Reihe von Erweiterungen sinnvoll. Wichtig für die Modellierung von langandauernden Aktivitäten mit mobilen Agenten ist die transaktionale Absicherung der Agenten auf zwei Ebenen:

- ▷ Die rechnerübergreifende Migration eines Agenten darf im Fehlerfall nicht zu Dubletten oder dem Verlust des Agenten führen. Dies kann mit entsprechenden Übertragungsprotokollen zwischen den beteiligten Komponenten Ursprungssystem, Zielsystem und Agent erreicht werden. Ein in [Mat96] geschildertes Verfahren hierfür ist das zweiphasige Bestätigungsprotokoll.
- ▷ Auch langandauernde Aktivitäten eines Agenten (z.B. durch Benutzerinteraktionen verzögert) sollten transaktional ausgeführt werden können, so daß die Konsistenz des Systemzustandes auch im Fehlerfalle gewahrt bleibt. Die Grundlagen hierfür sind bereits durch die Konzeption des Agentenmodells vorhanden, da das verwendete Verarbeitungsmodell eine Aufteilung jeder Aktivität in einzelne Regeln (d.h. Bearbeitungsschritte) erfordert. Dies ist eine Voraussetzung für den Einsatz von kompensationsorientierten Transaktionsmodellen für langandauernde Aktivitäten (z.B. Sagas [GMS87] oder ConTracts [WR91]).

Anhang A

Ausgewählte Schnittstellen der Bibliotheken zum Agentensystem

A.1 Spezifikationskonstruktoren

```
interface TBCContent
import iter newDate
export
    error :Exception reason :String end
    (* — content specification object: *)
    Let Rec Spec <:Ok = Tuple
        case int, real, date, string
        case rcd, variant with
            elements() :iter.T(NamedContentSpec)
        case singleChoice, multipleChoice, sequence with
            spec() :Spec
    end
    and NamedContentSpec <:Ok = Tuple
        name :String
        spec() :Spec
    end
    (* — content object: *)
    Let Rec T <:Ok = Tuple
        spec() :Spec
        case int with
            (* by default, the current state is 0 *)
            set(:Int) :Ok
            get() :Int
        case real with
            (* by default, the current state is 0.0 *)
            set(:Real) :Ok
            get() :Real
        case date with
            (* by default, the current state is today *)
            set(:newDate.T) :Ok
            get() :newDate.T
        case string with
            (* by default, the current state is  *)
```

```
        set(:String) :Ok
        get() :String
    case rcd with
        ref(name :String) :T
        (* get a content object of this record via its name.
            Raise error iff the refered object isn't found. *)
    case variant with
        (* by default, the first specified variant is selected *)
        select(:String) :Ok
        (* select a variant-element via its name *)
        selected() :String
        (* get the name of the currently selected variant. *)
        ref() :T
        (* get a content object of the current record variant. *)
    case singleChoice with
        (* by default, the first possible choice is selected *)
        get() :T
        (* get the current selected choice *)
        choose(c :T) :Ok
        (* select a new 'current selected' choice *)
        select(p(c :T) :Bool) :Ok
        (* select a new 'current selected' choice. Iterate over
            each valid choice and choose the last one for which p() returns
            true. *)
        setChoices(:iter.T(T)) :Ok
        (* set new valid choices and reset the current selection to default *)
        getChoices() :iter.T(T)
        (* get the set of possible valid choices *)
    case multipleChoice with
        (* by default, no choice is selected *)
        get() :iter.T(T)
        (* get the set of currently selected choices *)
        choose(c :T) :Ok
        (* add a new choice to the set of current selected choices *)
        discard(c :T) :Ok
        (* discard a choice from the set of current selected choices *)
        select(p(c :T) :Bool) :Ok
        (* select a new set of 'current selected' choices. Iterate over
            all valid choices and choose each of them for which p() returns
            true. *)
        setChoices(:iter.T(T)) :Ok
        (* set new valid choices and reset the current selection to default *)
        getChoices() :iter.T(T)
        (* get the set of possible valid choices *)
    case sequence with
        set(:Sequence) :Ok
        (* set reference to bulk data interface object *)
        get() :Sequence
        (* get current reference to bulk data interface object *)
end
and Sequence <:Ok = Tuple
    elements(A <:Ok  map(:T) :A) :iter.T(A)
    fromIter(A <:Ok  itr :iter.T(A)  map(:A  :T) :Ok) :Ok
end
```

```
    (* — general: *)
    setDebug(f :Bool) :Ok
    (* — specification constructors: *)
    intSpec() :Spec
    realSpec() :Spec
    stringSpec() :Spec
    dateSpec() :Spec
    (* create an atomic specification of its type. *)
    rcdSpec(elements :Array(NamedContentSpec)) :Spec
    (* create a record specification given an iteration of named content.
        Each named content has an unique name and its content specification. *)
    variantSpec(elements :Array(NamedContentSpec)) :Spec
    (* create a variant record specification given an iteration of variants.
        Each variant has an unique name and an iteration of its aggregated
        componenets, its 'elements' *)
    singleChoiceSpec(spec :Spec) :Spec
    (* create a single choice specification whose choices are
        specified by 'c' *)
    multipleChoiceSpec(spec :Spec) :Spec
    (* create a mulitple choice specification whose choices are
        specified by 'c' *)
    sequenceSpec(spec :Spec) :Spec
    (* create a sequence specification whose elements are specified by 'c' *)
    (* — content object constructors: *)
    create(spec :Spec) :T
    (* create a content object conforming to the specification object 's'. All
        content objects with a state are initialized to their documented
        default state. *)
    copy(:T) :T
    (* create a new content object with the same specification and state
        as the given content object. The objects have a seperate state
        afterwards. *)
end;
```

A.2 Nachrichtensubsystem

```
interface ITC
import :ITCMessage
export
    error :Exception reason :String end
    (* Raised if something goes wrong. 'reason' explains the problem.*)
    itcShutdown :Exception end
    (* See rcv() for explanation.*)
    itcMessageRejected :Exception end
    (* See rcv() for explanation.*)
    Let Address = ITCMessage.Address
    Let MsgBody = ITCMessage.T
    Message <:Ok
    (*An ITC message *)
    Port <:Ok
```

```
(* A communication port to receive messages on it. This handle
   has no direct binding to the itc system, nor to its owners
   thread. It is therefore 'migration-safe' and may be passed
   to other threads whithout any problems. *)
nilPort() :Port
(* The always wrong port *)
(* ====== communication ports: *)
newPort() :Port
(* Create a new anonymous communication port. *)
freePort(:Port) :Ok
(* Free a port. This will also free its name if it is a named
   port. *)
equalPorts(:Port :Port) :Bool
(* True iff ports have the same id. *)
unmountPort(:Port) :Ok
(* Unmount a port from an itc system. This is needed if threads attempt to
   migrate into the scope of another itc and refer to this port directly
   and not by an address (e.g. share a port value).
   After an unmount no send or rcv on this port is possible until it is
   re-mounted. If a port is named, the port name stays allocated at
   the current itc if the port has been created here by newPort(). *)
mountPort(:Port) :Ok
(* Mount a port at an itc system. This is necessary after migration into
   the scope of another itc system to make the port a valid one. If the
   given port is not known by the current itc,mountPort() will create it.
   Then, a subsequent unmountPort() call will free! the port. This does not
   happen if the port has been created by newPort(), of course. *)
portId(:Port) :String
(* get the world wide unique port id *)
getPort(id :String) :Port
(* get the port associated with <id> *)
namePort(c :Port name :String) :Ok
(* Name a port to select it via an address. The name has to be
   unique w.r.t the local itc. *)
existsPort(address :Address) :Bool
(* Check if port exists. Note: Threre is no guarantee that the
   checked port will conteinue to exist after this call. *)
waitForPort(address :Address  timeoutSecs :Int) :Bool
(* Wait until the named port exists. If no such port exists after
   timeoutSecs seconds the result is false. Otherwise it is true *)
(* ====== message passing: *)
Let MsgId = Int
(* Automatic generated per-port unique message id in case of
   a message is sent off for which a reply is expected.
   The replied message gets the MsgId which has been assigned
   to the original message by the corresponding send() call. *)
send(msg :MsgBody  to :Address  origin :Port) :MsgId
(* Asynchronous send a message to the port addressed by 'to'.
   If 'origin' is the nilPort, no reply for this msg is expected, otherwise
   any reply to this message will be delivered to the given port. *)
rcv(c :Port) :Message
(* Receive a message on port 'c'. This call blocks until a message
   arrives.
   Will raise itcShutdown iff a itc.stop(true) is issued while waiting for
```

```
        a message. Will raise itcMessageRejected iff
        a message delivery failed for which rcv awaits the reply
        or someone sends a message with state = rejectedState. *)
    sendRcv(:MsgBody  to :Address) :Message
    (* Synchronous send/recv pair. May raise the same exceptions like rcv(). *)
    reply(:Message  :MsgBody) :Bool
    (* Set a new message body and send the message to its origin.
        Will return false if the sender
        of the message expects no reply for this message or
        this message has been already replied. *)
    forward(message :Message  destinationPort :Port) :Ok
    (* Forward a message to the destinationPort. This only changes
          the reveicerAddress and, in case of message is a reply, the
          replyPortId of the message. The forwarded message is not mediated
          again. It is save to forward one message to mutliple destinations.
    *)
    (* ====== message access: *)
    State <:Ok
    okState,             (* delivery ok and appl. processing ok *)
    failedState,         (* delivery ok but appl. processing failed *)
    rejectedState :State   (* delivery failed. rcv() will raise exception. *)
    setState(:Message  :State) :Ok
    (* Set the current state of a message. *)
    getState(:Message) :State
    (* Get the current state of a message. *)
    getBody(:Message) :MsgBody
    (* Extract body part from a message *)
    getMessageId(:Message) :MsgId
    (* Get the message id from a message. This is usefull if a reply port is
        used for more than one message. Each of them carry an unique message
        id (w.r.t the receiving port). *)
    expectsReply(:Message) :Bool
    (* Return true iff a message expects to be replied. *)
    isReply(:Message) :Bool
    (* Return true iff the message given is a replied message. *)
    getOrigin(:Message) :Address
    (* Return the address of the origin of a message. It is possible
        that the returned address does not carry a port name with it
        (e.g. the name is equal to nilPort().name).
        The domain name is valid in any case. *)
    (* ====== mediator use only: *)
    deliver(message :Message  incomingPort :Port) :Ok
    (* Forward a message to its destination. This is only valid if
        incomingPort is the mediator port. Otherwise raises error. *)
    setBody(message :Message  body :MsgBody  incomingPort :Port) :Ok
    (* Change the body of a message. This is only valid if
       incomingPort is the mediator port. Otherwise raises error. *)
    (* ====== admin: *)
    init() :Ok
    (* (Re-)initalize itc state. Only valid iff itc is not running. *)
    start() :Ok
    (* Start itc service loop. *)
    stop(force :Bool) :Ok
    (* Stop itc service loop. If force is false then raise error
```

```
        iff stop is not possible safely. Otherwise all waiting itc
        clients are signaled about itc shutdown and itc is stopped. *)
    setSocketPort(no :Int) :Ok
    (* Set socket port number to use. Default 1204. Always valid but
        has no effect until an init() or start() happens. *)
    initMediator(name :String) :Ok
    (* Set name of local mediator port to use by itc. Default 'THE-MEDIATOR'.
        Always valid and forces itc to relink to the mediator port on the next
        processed message. If no port with 'name' exists, itc switches
        mediator routing off. *)
end;
```

A.3 Agentensystem

```
interface Agent
import location  tbConv  tbcSpec  wwuid
export
    error : Exception reason :String  end
    (* Raised if an operation failes *)
    meetingDenied : Exception  end
    (* Raised from an agent if it is not interested in a meeting an other
        agent asked for. *)
    deferedDialog() :tbConv.Dialog
    (* This special dialog may be returned by a performer rule instead of
        the dialog which is expected w.r.t the current conversation specification.
        It is intercepted by the agent and will not be delivered
        to the conversations customer agent. The reply is defered until a
        customer rule calls conversation.reply() for this conversation. *)
    (* === Abstract (migration-safe) agent handle: *)
     Let T =  Tuple
        name :String (* The high-level name of an agent *)
        id :wwuid.T  (* It's world-wide unique identificator *)
        owner :wwuid.T (* The owners id of an agent *)
        home :location.T (* The agents home location *)
     end
    equal(a1,a2 :T) :Bool
    (* Check if the agent handles a1 and a2 denote the same agents. *)
    (* === Agent types: *)
    new(A <: Ok  a :A
            birth(self :T  a :A) : Ok
                (* Executed after agent is created successfully in its place. *)
            meet(self,other :T  spec :tbcSpec.T  a :A) :T
                (* Executed if some agent wants to meet this agent. *)
            die(self :T  a :A) : Ok
                (* Finalizer. Executed if the agent is shut up. *)
            leave(self,fromPlace :T  a :A) : Ok
               (* Executed if the domain is ready to ship the agent to
                   another place. (niy) *)
            enter(self,atPlace :T  a :A) : Ok
                (* Executed if the domain has shipped this agent to
```

```
                another place. (niy) *)
           owner :T  name :String) :T
(* Create a new mobile agent. *)
newPlace(A <: Ok  a :A
           birth(self :T  a :A) : Ok
              (* Executed after agent is created successfully in its place. *)
           meet(self,other :T  spec :tbcSpec.T  a :A) :T
              (* Executed if some agent wants to meet this agent. *)
           die(self :T  a :A) : Ok
              (* Finalizer. Executed if the agent is shut up. *)
           incomingAgent(self,other :T  a :A) :Bool
              (* Executed if the domain is about to ship an agent into this
                 place. Will also be executed if an agent is to be created
                 in this place. *)
           outgoingAgent(self,other :T  a :A) : Ok
              (* Executed if the domain is about to ship an agent from this
                 place. *)
           owner :T  name :String) :T
(* Create a new place.  *)
(* === Agent roles: *)
 Let  Rec Conversation(S <: Ok) <: Ok =  Tuple
    c :tbConv.T(S)
    (* A conversation handle. *)
    peer :T
    (* The conversation partner. *)
    reply(S <: Ok  performerConversation :Conversation(S)
          dialog :tbConv.Dialog) : Ok
    (* Sends the dialog in the given performerConversation to the bound
          customer (see deferedDialog()).*)
 end
(* A Conversation representation with a local state of type S. *)
 Let RequestName = String
 Let CustomerRole(S <: Ok) <: Ok =  Tuple
    spec() :tbcSpec.T
    (* The bound specification for this role. *)
    isNil() :Bool
    (* Returns true iff role is a nil role. *)
    set(dialogName :String  action(:Conversation(S)) :RequestName) : Ok
    (* Action will be executed atomically whenever a performer returns a
          dialog with this dialogName. *)
    startConversation(performer :T  s :S) : Ok
    (* Immediately starts a conversation with the performer.
          Never blocks. s is the conversations local state. *)
 end
addCustomerRole(S <: Ok  self :T  spec :tbcSpec.T) :CustomerRole(S)
(* Add a customer role for conversations specified by spec to the agent.
     For each new started conversation a conversation private state S must
     be provided to startConversation() *)
nilCustomerRole(S <: Ok) :CustomerRole(S)
(* Generate a place-holder customer role. These always return
     isNil() with true *)
 Let PerformerRole(S <: Ok) <: Ok =  Tuple
    spec() :tbcSpec.T
    (* The bound specification for this role. *)
```

```
        isNil() :Bool
        (* Returns true iff role is a nil role. *)
        set(dialogName, requestName :String
            action(:Conversation(S)) :tbConv.Dialog) : Ok
        (* action will be executed atomically whenever a customer sends this
             request in this dialog. if action returns nilDialog then
             an asynchronous reply is allowed. *)
      end
    addPerformerRole(S <: Ok  self :T  spec :tbcSpec.T
        init() :S) :PerformerRole(S)
    (* Add a performer role for conversations specified by spec to the agent.
        For each new started conversation a conversation private state S is
        created by init() *)
    nilPerformerRole(S <: Ok) :PerformerRole(S)
    (* Generate a place-holder performer role. These always return
        isNil() with true *)
    (* === Agent Root-Place (ubiquitous) *)
    currentRootPlace() :T
    (* Get a handle to the current meetingPoints root place. *)
    (* === Agent services: *)
    migrateTo(self :T  to :location.T) : Ok
    (* Request an agent for a migration to another location. The agent
        will continue its operation on the other locations site. *)
    meet(self :T  name :String  spec :tbcSpec.T  nWaitSecs :Int) :T
    (* Resolve the agent name with respect to the current place of self
        and ask it for a meeting. Block max nWaitSecs unti the meet request
        succeeds. Returns the others agent handle. May raise meetingDenied
        if the meeting has been denied by the other agent or raise error
        if no agent could be located within nWaitSecs. *)
    meetAndTalk(S <: Ok  self :T  name :String  customer :CustomerRole(S)
       init :S  nWaitSecs :Int) :tbConv.T(S)
    (* Same as meet but implicitly involve the given customer role for the
        conversation. This call is synchronous, e.g. will block until
        the customer conversation ends. The result is the comp Leted
        conversation. May raise meetingDenied
        if the meeting has been denied by the other agent or raise error
        if no agent could be located within nWaitSecs. *)
    currentLocation(:T) :location.T
    (* Find out the current location for a given agent handle.
        (This may or may not be possible, due to the capabilities of
           the meetingPoint implementation, if the given agent is not
           known locally). *)
    currentPlace(:T) :T
    (* Finds out the current place for a given agent handle. This will
        only succeed if the given agent resides in the current host. *)
    (*=== administation *)
    startDomain() : Ok
    stopDomain() : Ok
end;
```

Anhang B

Grammatik der Konversationsspezifikationen

An dieser Stelle ist die Grammatik für die Konstruktion von Konversationsspezifikationen dargestellt. Die Notation ist aus der Darstellung der Programmiersprache TL in [Mat93] übernommen.

Conversation ::=
  CONV *ConvIdentifier* ***WITH*** *InitialDialog* { *Dialog* } ***END****;*
IntialDialog ::=
  Dialog;
Dialog ::=
  DIALOG *DialogIdentifier* ***WITH*** { *Request* } [*Content*] ***END****;*
Request ::=
  REQUEST *RequestIdentifier* ***TO*** *DialogIdentifier* { *DialogIentifier* }
  (***ELLIPSES*** | ***NOELLIPSES***) ***END****;*
Content ::=
  CONTENT *NamedContent* { *NamedContent* } ***END****;*
NamedContent ::=
  ContentIdentifier ***OF*** *Type;*
Type ::=
  BasicType | *CompoundType;*
BasicType ::=
  INT | ***REAL*** | ***STRING*** | ***DATE*** | ***SPEC****;*
CompoundType ::=
  REC *NamedContent* { *NamedContent* } ***END*** |
  VARIANT *NamedContent* { *NamedContent* } ***END*** |
  MCHOICE OF *Type* |
  SCHOICE OF *Type* |
  SEQ OF *Type;*
ConvIdentifier ::=
  identifier;
DialogIdentifier ::=
  identifier;
ContentIdentifier ::=
  identifier;

Anhang C

Überblick über die Methode der *Mainstream Objects*

Die Methode der *Mainstream Objects* [You95] dient der Modellierung von Geschäftsprozessen auf einer objektorientierten Basis. Ziel der Methode ist, einen Geschäftsvorgang derart zu strukturieren, daß die Umsetzung in ein Informationssystem unter Verwendung einer objektorientierten Programmiersprache leicht fällt. Um dieses Ziel zu erreichen, wird der Vorgang mit Hilfe mehrerer Modelltypen von verschiedenen Seiten betrachtet.

C.1 Das Objektstrukturmodell

Das Objektstrukturmodell ist das Kernelement der Entwurfsmethode. In diesem Modell werden die Klassen, die im Projekt realisiert werden sollen, ermittelt. Außerdem werden die Verbindungen zwischen Klassen untersucht und abgebildet, so daß das Modell die Grundlage des Systementwurfs bildet. Es spiegelt dabei die Welt des Benutzers wider, wobei alle Objekte mit Begriffen der Anwendungdomäne bezeichnet werden.

Neben dem Objektstrukturmodell präzisieren weitere Modelle andere Aspekte des Systementwurfes. Diese Modelle sind das Transaktionsfolgenmodell, das Objektzyklusmodell und das Übersichtsmodell.

Die Abbildung C.1 zeigt, wann welches Modell im Projektzyklus erstellt werden soll. Dabei nimmt das Objektstrukturmodell im Laufe des Projektes einen immer größeren Umfang ein.

Der wachsende Umfang entspricht dem steigenden Detaillierungsgrad des Modells. Bis zum logischen Entwurf werden lediglich die Objekte der abzubildenden Realität behandelt. Während des logischen Entwurfes wird die Struktur des zu erstellenden Anwendungsprogramms mit einbezogen. Im physischen Entwurf werden schließlich die fehlenden Implementierungsdetails ergänzt.

Ein Objektstrukturmodell beinhaltet die folgenden Komponenten:

- ▷ Gegenstandsobjekte
- ▷ Schnittstellenobjekte
- ▷ Kontrollobjekte
- ▷ abstrakte Objektklassen

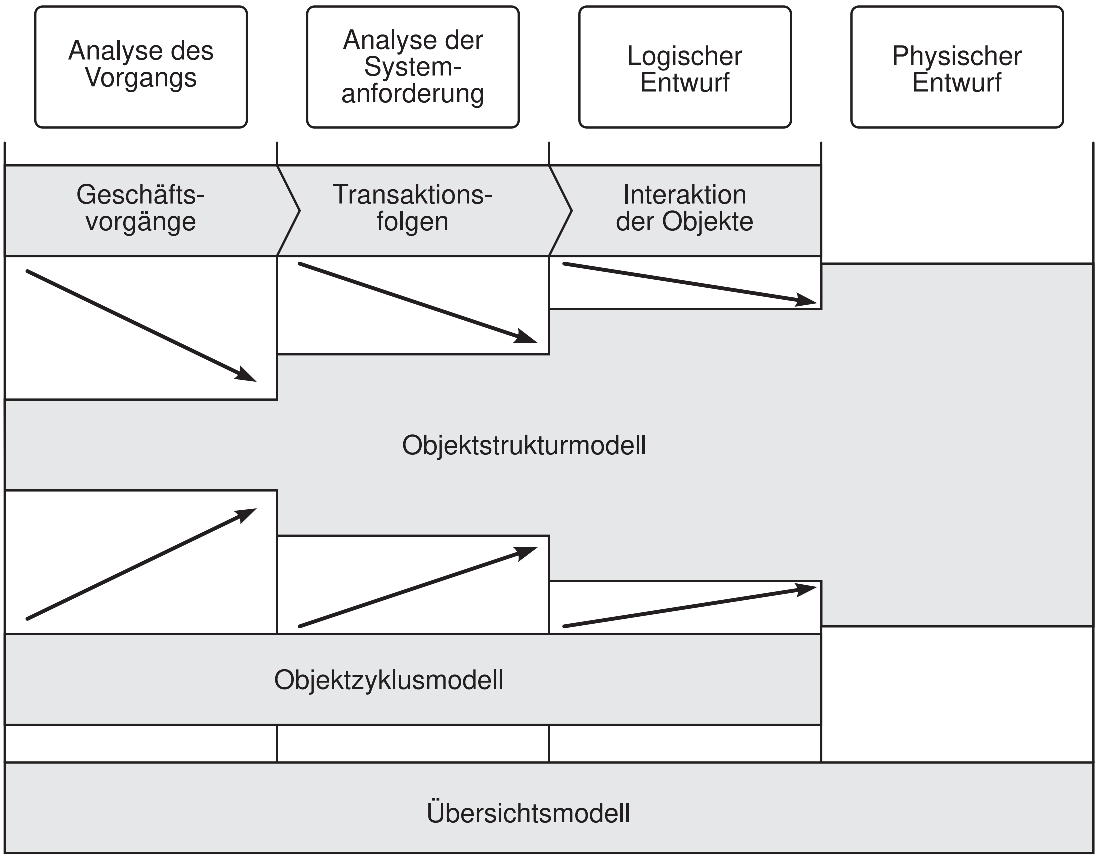

Abbildung C.1: Einsatz der Modelle im Projektzyklus

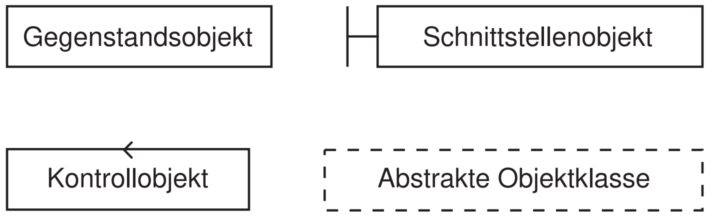

Abbildung C.2: Darstellung der Objekttypen

In der Abbildung C.2 sind die zugehörigen Symbole zu sehen.

Zur weiteren Beschreibung der Objekte (bzw. der Objektklassen) gehören die Operationen und Attribute eines Objektes und deren zugesicherte Eigenschaften (Typen, Bedingungen, ...).

Im Objektstrukturmodell werden zusätzlich zu den Objekten die folgenden Objektbeziehungen dargestellt (siehe auch Abbildung C.3):

Statische Beziehungen modellieren die Beziehung zwischen den Objekten. Sie sind an Gegenstands-Beziehungs-Modelle angelehnt. Die Beziehungen können mit Namen versehen werden. Kardinalitäten werden als Zahlenpaare an den verbundenen Objekten spezifiziert. Hierbei können für jedes der beiden verbunden Objekte Kardinalitäten der Art $(\min 1, \max 1)$, $(\min 0, \max 1)$, $(\min 1, \max n)$ oder $(\min 0, \max n)$ angegeben werden.

Vererbung wird durch Pfeile von den spezialisierten Klassen zur allgemeineren Klasse dargestellt.

Aggregation zeigt, welche Objekte zu neuen Objekten in einer Teil-Ganzes-Beziehung zusammengefaßt werden. Auch hier können die oben genannten Kardinalitäten verwendet werden.

Kommunikation über Nachrichten ist die einzige Verbindungsart, die Objekte verschiedener Typen erreichen kann.

Die Modelle für Geschäftsvorgänge, Transaktionsfolgen, Objektinteraktionen und Objektzyklen spielen in der vorliegenden Arbeit keine Rolle und werden daher nicht näher betrachtet.

C.2 Das Übersichtsmodell

Um die Entwicklung großer Systeme in kleinere, handhabbare Teilaufgaben zu untergliedern, zerlegt man diese Systeme in Subsysteme, die dann getrennt voneinander bearbeitet werden können. Diese Unterteilung von Systemen wird mit dem Überblicksmodell erleichtert. Außerdem bietet es den involvierten Entwicklergruppen die Möglichkeit, die Stellung ihres Subsystems im Gesamtsystem im Blick zu behalten.

Das Zusammenwirken der verschiedenen Subsysteme erfolgt in der Form von Diensten, die den anderen Subsystemen zur Verfügung gestellt werden.

Im Übersichtsmodell werden die folgenden Komponenten aus Abbildung C.4 aufgenommen:

Akteure: Personen werden als "Strichmännchen" und maschinelle Akteure als Blitz dargestellt.

Systemgrenze: Alles, was innerhalb einer Systemgrenze liegt, wird vom betrachteten Modell behandelt. Die Akteure liegen außerhalb.

Subsysteme: Jedes Subsystem wird als benanntes Rechteck mit abgerundeten Ecken dargestellt.

Dienste: Die Dienste werden durch Halbkreise in den Piktorgrammen der Subsysteme dargestellt. Dienste werden ebenfalls über ihren Namen identifiziert.

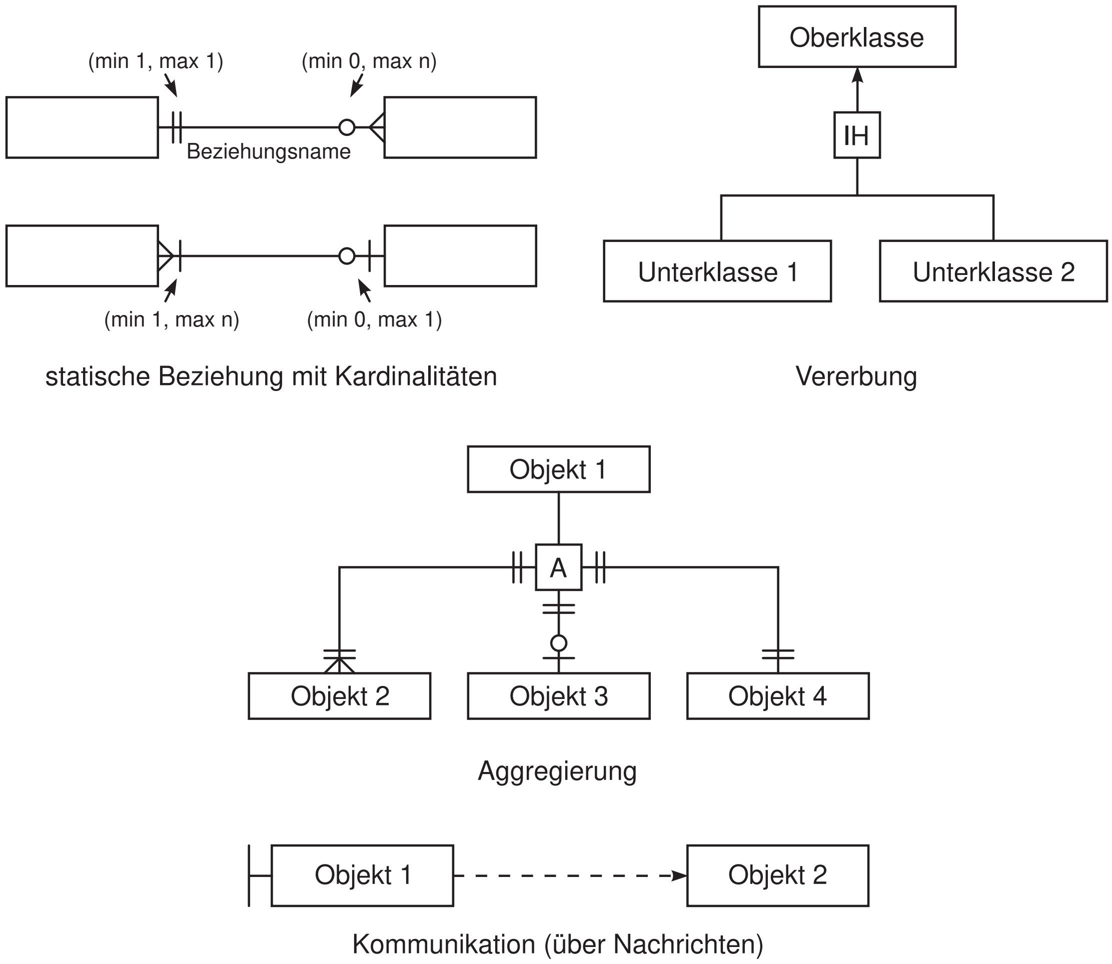

Abbildung C.3: Beziehungstypen im Objektstrukturdiagramm

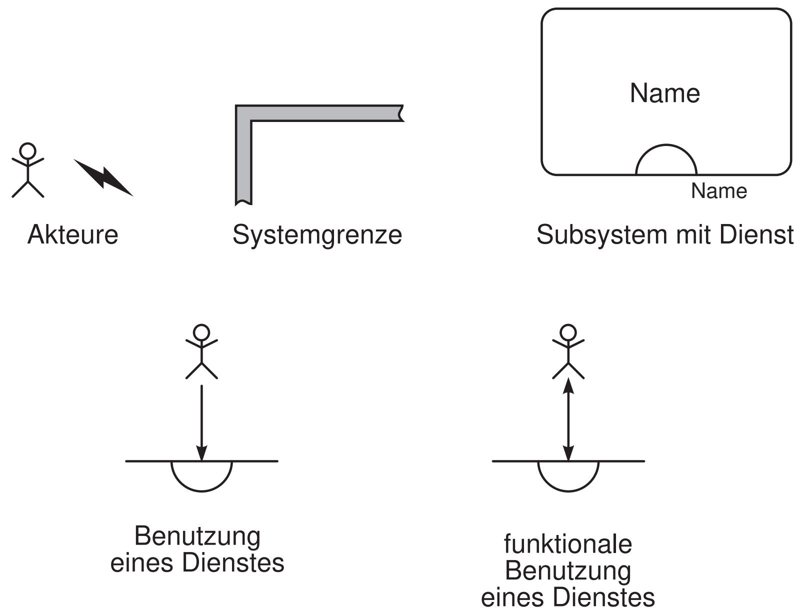

Abbildung C.4: Komponenten des Übersichtsmodell

Benutzung von Diensten: Wenn ein Akteur oder ein Subsystem einen Dienst nutzt, zeigt ein Pfeil vom Akteur auf den entsprechenden Dienst.

Anhang D

Glossar

D.1 Agententerminologie

Domäne (*domain*): Eine Komponente, die *Orte* und *Agenten* beherbergt und verwaltet. Domänen sind symbolisch adressierbar und besitzen immer mindestens einen Ort, welcher die Domäne selbst repräsentiert.

Aktivität (*activity*): Eine Aktivität ist eine beliebige Aufgabe, die durch einen oder mehrere *Agenten* modelliert wird. Dabei können Aktivitäten über Adressräume verteilt sein.

Mobiler Software-Agent (*mobile agent*): Ein mobiler Software-Agent ist eine Komponente, welche aus einer oder mehreren aktiven Einheiten besteht. Ein Agent ist Teil mindestens einer *Aktivität*. Mobile Agenten sind in der Lage, sich über Adressraumgrenzen zu bewegen. Agenten kommunizieren untereinander über typisierte *Nachrichten*.

Kommunikationsendpunkt (*port*): An einem Kommunikationsendpunkt empfängt eine *aktive Einheit* Nachrichten. Es gibt benannte und anonyme Kommunikationsendpunkte. Eine aktive Einheit kann einen benannten Kommunikationsendpunkt über dessen Adresse identifizieren und Nachrichten an diesen senden. Ein anonymer Port kann nur Nachrichten von aktiven Einheiten empfangen, die eine Objektbindung an den (privaten) Portidentifikator besitzt.

Portidentifikator (*port identifier*): Ein Portidentifikator ist ein weltweit eindeutiger Bezeichner, der automatisch einem neu erzeugten Port zugewiesen wird. Alle Ports werden über Nachrichtensubsysteme mittels dieses Identifikators referenziert.

Nachricht (*message*): Nachrichten sind typisierte Botschaften und werden über *Ports* ausgetauscht. Es gibt einen definierten Satz von Nachrichten, welcher die Grundlage der Kooperationsprotokolle auf Systemebene zwischen Agenten bilden.

Aktive Einheit (*thread*): Eine aktive Einheit ist in der Lage, eine Nachricht an eine andere (nicht notwendigerweise verschiedene) aktive Einheit zu versenden.

Ort (*place*): Ein Ort ist ein *Agent*, der eine *Domäne* aufgabenspezifisch logisch untergliedert. Ein Ort ist eine spezielle Form eines *Agenten*. Jeder Ort ist durch einen lokal zur Domäne eindeutigen Namen identifiziert. Orte können wiederum andere Orte und Agenten beinhalten und bilden eine Hierarchie. Sie sind Sichtbarkeitsbereiche für die Adressierung von Agenten.

Treffen (*meeting*): Ein Treffen findet zwischen zwei Agenten innerhalb eines Ortes statt. Treffen werden durch die Domäne vermittelt und koordinieren den Anfang einer Kooperation zwischen Agenten, indem sie diese untereinander sichtbar machen.

D.2 *Business Conversations*

Konversation (*conversation*): Der Oberbegriff für eine Folge von Dialogen zwischen einem *Kunden* und einem *Dienstleister*, mit dem Ziel einer Aktion. Eine Konversation kann einzelne (Teil-) Konversationen wiederverwenden.

Sprechakt (*speech act*): Ein Begriff aus der Language-Action Sicht auf CSCW nach T. Winograd [FGHW88]: "...human beeings are fundamentally linguistic beings: action happens in language in a world constituted by language.". Genauer werden Sprechakte in Zyklen von vier Phasen unterteilt: *Anfrage*,*Übereinkunft*, *Leistung* und *Rückmeldung*. In einer technischen Umsetzung des Modells wird ein Sprechakt als *Konversation* bezeichnet.

Phase (*phase*): Granularität des Sprechaktmodells. Die Einteilung in Phasen muß nicht in einer Implementation wiedergespiegelt werden. Sie dient allein der Veranschaulichung des Ablaufs von Sprechakten.

Anfrage (I) (*request*): Bezeichnung für Phase eins des Modells.

Anfrage (II) (*request*): Die Bearbeitung einer Anfrage führt im Modell in einen neuen Dialog. Anfragen sind die einzige Möglichkeit in einer Konversation einen neuen Zustand zu erreichen.

Übereinkunft (*agreement*): Bezeichnung für die zweite Phase des Modells.

Leistung (*performance*): Bezeichnung für die dritte Phase des Modells.

Rückmeldung (*feedback*): Bezeichnung für die vierte Phase des Modells.

Kunde (*customer*): Initiator einer Konversation. Innerhalb eines Sprechakts ist die Rolle des Kunden festgelegt. Aus der Sicht einer übergeordneten Konversation kann der Kunde jedoch gegenüber einem anderen Kunden wiederum in der Rolle des *Dienstleisters* auftreten.

Dienstleister (*performer*): Akzeptor einer Konversation. Innerhalb eines Sprechakts ist die Rolle des Dienstleisters festgelegt. Aus der Sicht einer übergeordneten Konversation kann der Dienstleisters jedoch gegenüber einem anderen Dienstleisters wiederum in der Rolle des *Kunden* auftreten.

Akteur (*actor*): Sowohl der *Kunde* als auch der *Dienstleister* nehmen aktiv (daher Akteur) an der Konversation Teil.

Regel (*rule*): Jeder Aktor führt Aktionen aus, die Regeln genannt werden. Der Dienstleister reagiert Aufgrund einer Anfrage gemäß einer zugehörigen Regel und erzeugt einen neuen Dialog. Der Kunde wiederum reagiert auf den Dialog gemäß seiner eigenen Regeln.

Annahme (*acceptance*): Die ersten beiden Phasen eines *Sprechakt*zyklus.

Erfüllung (*completion*): Die letzten beiden Phasen eines *Sprechakt*zyklus.

Dialog (*dialog*): Ein Schritt einer Phase eines Sprechakts. Das Mittel des Dialogs ist nicht bestimmt. Eine Phase kann beliebig viele Dialogschritte erfordern. Dialoge enthalten strukturierte Information, ihren *Inhalt*.

Inhalt (*content*): Ein Teil eines Dialogs in Form eines inspizierbaren Bedeutungsträgers.

Modifikator (*modifier*): Mit Modifikatoren kann eine bestehende Konversation in Form einer Sub-Konversation an die Erfordernisse der einbettenden Konversation angepasst werden. Dialoge der Sub-Konversation können konfiguriert, automatisiert oder ausgeblendet werden.

Historie (*history*): Die Historie bildet die gemeinsame Vergangenheit von Kunde und Dienstleister innerhalb einer Konversation. Sie wird transparent vom System fortgeschrieben und enthält alle relevanten Daten einer Konversation, insbesondere alle Anfragen und Dialoge. Sowohl Kunde als auch Dienstleister haben vollständigen Zugang zu den gesammelten Daten, die von beiden Seiten als verbindlich betrachtet werden.

Abbruch (*breakdown*): Ein Abbruch ist eine spezielle Anfrage an einen Dienstleister in einer Fehlersituation. Diese Anfragen werden in der Regel vom System im Namen des Kunden an den Dienstleister gestellt (z.B. bei einer Zeitüberschreitung des Kunden in einem Dialog).

Konversations-Stadium (*conversation stage*): Eine Konversation befindet sich immer in einem definierten Stadium. Damit werden die Phasen des Spechaktzyklus widergespiegelt.

Literaturverzeichnis

[AGP94] Jean-Marc Andreoli, H. Gallaire, and R. Pareschi. Objects Meet Rules: from Communication to Coordination through Declarativity. Technical report, Rank Xerox Resarch Center, 1994.

[AH87] G. Agha and C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[Aus62] J. Austin. How to do things with words. Technical report, Oxford University Press, Oxford, 1962.

[BBO+86] B. Baumgarten, H.-J. Burkhardt, V. Obermeit, P. Ochsenschläger, R. Prinoth, and R. Steinmetz. Datenbank und Kommunikationsorientierte Modellierung mehrseitiger Kooperation - Ein Vergleich auf der Basis von Netzen. Technical Report 115, Ges. für Mathematik und Datenverarbeitung, St. Augustin und Darmstadt, 1986.

[BW94] Russel Beale and Andrew Wood. Agent-Based Interaction. In *People and Computers IX*, Proceedings of HCI'94, pages 239–245, August 1994.

[Car89] L. Cardelli. Typeful Programming. Systems Resarch Center Report 45, Digital Equipment, Palo Alto, California, Mai 1989.

[Car94] L. Cardelli. Obliq: A Language with Distributed Scope. Technical report, Systems Resarch Center, Digital Equipment, Palo Alto, California, June 1994.

[CGH+95] David Chess, Benjamin Grosof, Colin Harrison, David Levine, and Colin Paris. Itinerant Agents for Mobile Computing. Technical report, IBM, Feb 1995.

[CHK94] D.M. Chess, C.G. Harrison, and A. Kershenbaum. Mobile Agents: Are they a good idea? IBM Research Report RC 19887, IBM, Oct 1994.

[Cou95] Antony Courtney. Phantom: An Interpreted Language for Distributed Programming. Technical report, Trinity College Dublin, Ireland, 1995.

[CP84] S. Ceri and G. Pelagatti. *Distibuted Databases, Principles and Systems*. R. Oldenburg Verlag, 1984.

[DDJ+97] Giorgio De Michelis, Eric Dubois, Matthias Jarke, Florian Matthes, John Mylopoulos, Mike Papazoglou, Klaus Pohl, Joachim Schmidt, Carson Woo, and Eric Yu. Cooperative information systems: A manifesto. In Mike P. Papazoglou and Gunther Schlageter, editors, *Cooperative Information System: Trends and Directions*. Academic Press, 1997.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. *ACM SIGMOD*, 1990.

[FGHW88] Fernando Flores, Michael Graves, Brad Hartfield, and Terry Winograd. Computer Systems and the Ontologie of Organisational Interaction. *ACM Transaction on Office Information Systems*, 6(2):153–172, April 1988.

[Fon93] Leonard N. Foner. Whats An Agent, Anyway? Technical Report 93-01, MIT Media Lab, 20 Ames St, Cambridge, 1993. Memo.

[Gei95] Andreas Geisler. Basisdienste zur Gestaltung einer reflektiven graphischen Entwicklungsumgebung für persistente Programmiersprachen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Juni 1995.

[GMI95a] General Magic, Inc. *Telescript Devloper Environment, Version 1.0 alpha*, October 1995. Internet WordWideWeb, see General Magic homepage.

[GMI95b] Telescript and Magic Cap. Technical report, General Magic, Inc., 1995. Online book, see General Magic homepage.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 249–259, May 1987.

[Gö96] Martin Göllnitz. Polymorphe persistente Client/Server Programmierung mit dynamischer hierarchischer Adressauflösung. Studienarbeit. Arbeitsbereich DBIS, Universität Hamburg, Fachbereich Informatik, 1996.

[GR93] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., 1993.

[Gro91] Object Management Group. The common object request broker: Architecture and specification. Document 91.12.1, Rev. 1.1, OMG, December 1991.

[Had95] Afsaneh Haddadi. *Communication and Cooperation in Agent Systems*. Springer Verlag, 1995.

[HG84] J. Hogg and S. Gamvroulas. An active mail system. In *Proceedings of the ACM SIGMOD*, number 14(2) in SIGMOD, pages 215–222, June 1984.

[HH94] Ralf G. Herrtwich and Günter Hommel. *Nebenläufige Programme*. Springer Verlag, 1994.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, LFCS, University of Edinburgh, Mar 1986.

[Hoh95] Fritz Hohl. Konzeption eines einfachen Agentensystems und Implementation eines Prototyps. Diplomarbeit, Universität Stuttgart, Abteilung Verteilte Systeme, August 1995.

[ISO95a] ISO/IEC. ITU-T X.901 ODP Reference Model. Part 1: Overview, 1995. Draft International Standard.

[ISO95b] ISO/IEC. ITU-T X.902 ODP Reference Model. Part 2: Foundations, 1995. Draft International Standard.

[ISO95c] ISO/IEC. ITU-T X.903 ODP Reference Model. Part 3: Architecture, 1995. International Standard.

[Jab95] S. Jablonski. Workflow Management Systeme: Motivation, Modellierung, Architektur. *Informatik Spektrum*, 18(1):13–24, 1995.

[Joh95] N. Johannisson. Generische Programmierung typsicherer Kommunikationsdienste. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1995.

[Juh88] E. Juhl. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, Department of Computer Science, University of Washington, December 1988.

[Kab96] Kabel New Media. Webtracking – Die 10 wichtigsten Fragen und Antworten. http://www.kabel.de/KNMantworten.html, September 1996.

[Kir94] Thomas Kirsche. Kooperation und Koordination in verteilten Systemen. In Hartmut Wedekind, editor, *Verteilte Systeme.* BI Wissenschaftsverlag, 1994.

[Kna95] Frederick Colville Knabe. *Language Support for Mobile Agents.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, October 1995.

[KV89] H. Kerner and F. Vogt. *ISO/OSI.* H. Kerner, Wolfsgraben, Technische Universität Wien, Austria, 1989.

[Lam94] Winfried Lamersdorf. *Datenbanken in verteilten Systemen.* Vieweg & Sohn Verlagsgesellschaft mbH, 1994.

[Lau97] Jan T. Laub. Analyse der Web-Server-Nutzung: Krititerien, Protokolldateien und Auswertungssoftware. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1997.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer Verlag, 1993. (In German.).

[Mat96] Bernd Mathiske. *Mobilität in persistenten Objektsystemem.* Dissertation, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich DBIS, Mai 1996.

[MMM93] B. Mathiske, F. Matthes, and S. Müßig. The Tycoon System and Library Manual. DBIS Tycoon Report 212-93, Universität Hamburg, Fachbereich Informatik, December 1993.

[MMS94] F. Matthes, S. Müßig, and J.W Schmidt. Persistent Polymorphic Programming in Tycoon: An Introduction. FIDE Technical Report FIDE/94/106, Universität Hamburg, Fachbereich Informatik, August 1994.

[MMS95] B. Mathiske, F. Matthes, and J.W. Schmidt. On Migrating Threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995. (Also appeared as TR FIDE/95/136).

[MS94] F. Matthes and J.W. Schmidt. Persistent Threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

[MSM95] Microsoft Message Queue Server (MSMQ). A White Paper from the Business Systems Technologie Series. Technical report, Microsoft Corporation, 1995. http://www.microsoft.com/msmq/overview.htm.

[MTH89] Robin Milner, Mats Tofte, and Robert Harper. The Definition of Standard ML. Technical report, MIT, Cambridge, MA, 1989.

[NV92] Gerald Neufeld and Son Vuong. Overview of ASN.1. *Computer Networks and ISDN Systems*, 23(5):393–415, Feb 1992.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, New York, 1996.

[ÖV94] M.T. Özsu and P. Valduriez. Distributed Data Management: Unsolved Problems and New Issues. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing*. IEEE Computer Society Press, 1994.

[Pie96] Andreas Piellusch. Synchronisation langlebiger Aktivitäten in persistenten Objektsystemen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Juni 1996.

[PSW96] Claudia Popien, Gerd Schürmann, and Karl-Heinz Weiß. *Verteilte Verarbeitung in Offenen Systemen*. Leitfäden der Informatik. Teubner, Stuttgart, 1996.

[Rei94] Andy Reinhardt. The Network with Smarts. *Byte*, pages 51–64, Oct 1994.

[Ric97] Ingo Richtsmeier. Vergleich objekt- und agentenbasierter verteilter Datenbankprogrammierung am Beispiel der Kopplung autonomer Internet WebSiteProfiler. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, January 1997.

[Ros93] Marshall T. Rose. MIME Extensions for Mail-Enabled Applications: application/Safe-Tcl and multipart/enabled-mail. Internet WWW, 1993. working draft.

[Sch90] Alexander Schill. *Migrationssteuerung und Konfigurationsverwaltung für verteilte objektorientierte Anwendungen*. Springer Verlag, 1990.

[Sch93] Alexander Schill. *DCE Das OSF Distributed Computing Environment*. Springer Verlag, 1993.

[Sea69] J. Searle. Speech Acts. Technical report, Cambridge University Press, Cambridge, 1969.

[Sta93] Michael K. Stanko. Media Einmaleins Teil 1 - *Reichweite und Kontakte*. *Teleimage*, (4), 1993.

[Sta94a] Michael K. Stanko. Media Einmaleins Teil 2 - *Kontaktverteilung*. *Teleimage*, (1), 1994.

[Sta94b] Michael K. Stanko. Media Einmaleins Teil 3 - *Zielgruppen und Affinitäten*. *Teleimage*, (2), 1994.

[Sun90] SUNOS 4.2. Network Programming Guide. Sun Microsystems, Inc., 1990.

[Sun95a] The Java Language: A White Paper. Technical report, Sun Microsystems, 1995. ftp://java.sun.com/1.0alpha3/doc/overview/java/java-whitepaper.ps.

[Sun95b] Hotjava (tm): The Security Story. Technical report, Sun Microsystems, 1995. http://java.sun.com/1.0alpha3/doc/security/security.html.

[Tan95] Andrew S. Tanenbaum. *Verteilte Betriebssysteme*. Prentice Hall, München, 1995.

[Tsc94] Volker Tschammer. *Integration Kooperierender Systeme*. R. Oldenburg Verlag, 1994.

[Way94] Peter Wayner. Agents Away. *Byte*, pages 113–118, May 1994.

[Way95] Peter Wayner. Free Agents. *Byte*, pages 105–114, Mar 1995.

[Whi94] J. E. White. Telescript Technologie - The Foundation for the Electronic Marketplace. Technical report, General Magic, Inc., 1994.

[Win87] Terry A. Winograd. A Language/Action Perspective on the Design of Cooperative Work. Technical Report STAN-CS-87-1158, Department of Computer Science, Stanford University, Stanford, CA 94305, April 1987.

[WR91] H. Wächter and A. Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann Publishers, Inc., 1991.

[You95] Yourdon, E. et al. *Mainstream Objects, an Analysis and Design Approach for Business.* Yourdon Press, Upper Saddle River, Nj, 1995.