Steuerung offener verteilter Anwendungssysteme

Konstruktion interaktionsorientierter Regelverarbeitungs- und Verhandlungsmechanismen — dargestellt am Beispiel elektronischer Dienstemärkte

Mark Tuan Viet Tu

Vom Fachbereich Informatik
der Universität Hamburg
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

Betreuer und 1. Gutachter:	Prof. Dr. Winfried Lamersdorf
2. Gutachter:	Prof. Dr. Christopher Habel
Tag der Disputation:	11. September 2000

Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?

T. S. Eliot

Zusammenfassung

Die Entwicklung und Verbreitung verteilter Anwendungssysteme wird immer stärker von offenen, globalen Netzwerkinfrastrukturen wie dem Internet vorangetrieben. Solche Systeme sind u.a. durch einen hohen Grad an *Dynamik*, *Heterogenität* und *Interaktion* gekennzeichnet und lassen sich deshalb sowohl theoretisch als auch (und vor allem) in der Praxis durch herkömmliche, rein algorithmische Berechnungsverfahren nicht *vollständig* spezifizieren und fehlerfrei entwickeln. Rein von außen und im laufenden Betrieb sind sie darüber hinaus, vor allem für die Anwender, schwer zu analysieren und zu überschauen. Daraus resultiert ein wachsender Bedarf an geeigneten Konzepten und Mechanismen, um derartige Systeme (besser) *steuern* und insbesondere in ihrem *Interaktionsverhalten* adäquat unterstützen zu können.

Grundsätzlich sind dabei zwei unterschiedliche Formen der Steuerung möglich: Bei der *Fremdsteuerung* wird von außerhalb des Systems Einfluss auf die Verhaltensweise der Komponenten geübt, während bei der *Selbststeuerung* die Systemkomponenten eigenständig versuchen, eventuelle Konflikte (bzgl. ihrer Interaktion) aufzulösen. Entsprechend schlägt diese Arbeit konkrete Lösungsansätze vor, die auf zwei Hauptkonzepten basieren: Einerseits werden spezifische, *interaktionsorientierte Regelverarbeitungsmechanismen* entwickelt und eingesetzt, um das Verhalten offener verteilter Anwendungen mittels externer benutzerdefinierter Regeln partiell zu steuern, ohne die Autonomie und die (ursprüngliche) Funktionalität der Anwendungen prinzipiell zu beeinträchtigen. Andererseits werden neuartige *Verhandlungsmechanismen* präsentiert, mit denen unterschiedliche Kooperationsanforderungen von zwei oder mehr Anwendungskomponenten *automatisch ausgehandelt* werden können. Dabei ermöglichen die Regelverarbeitungsmechanismen eine Form der o.g. Fremdsteuerung, während autonom verhandelnde Anwendungen als eine Form der *interaktionsbasierten* Selbststeuerung bzw. Selbstanpassung in einer offenen verteilten Umgebung betrachtet werden können. In dieser Arbeit wird zudem aufgezeigt, dass diese beiden Steuerungsformen nicht gegensätzlich sind, sondern als *funktional komplementär* zueinander verstanden werden können.

Als Beispiel für einen Anwendungsbereich, in dem die hier präsentierten Mechanismen konkret eingesetzt werden können, wurden *elektronische Dienstemärkte* gewählt, da diese ein Forschungs- und Anwendungsgebiet darstellen, in dem die oben genannten typischen Eigenschaften eines offenen verteilten Anwendungssystems besonders ausgeprägt sind und das zudem zurzeit besonders praxisrelevant ist. Dementsprechend werden die vorgeschlagenen Konzepte auch an einer Reihe von *Anwendungsszenarien* demonstriert, die sich auf dieses Umfeld beziehen.

Danksagungen

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Arbeitsgruppe Verteilte Systeme (VSYS) am Fachbereich Informatik der Universität Hamburg.

Dem Leiter dieser Arbeitsgruppe, Herrn Prof. Dr. Winfried Lamersdorf, danke ich herzlich für die Betreuung dieser Arbeit, für seine konstruktive Kritik und für seine stets aufmunternde Begleitung meiner fachlichen Interessen. Die stimulierende, einer fruchtbaren wissenschaftlichen Arbeit förderliche Atmosphäre innerhalb der Arbeitsgruppe VSYS ist zu einem großen Teil Prof. Lamersdorf zu verdanken.

Bei Prof. Dr. Christopher Habel, Leiter des Arbeitsbereichs Wissens- und Sprachverarbeitung (WSV), bedanke ich mich für seine Bereitschaft zur Übernahme des Koreferats, seine Anregungen hinsichtlich dieser Arbeit sowie das langjährige Interesse an meiner Arbeit.

Herrn Prof. Dr. Bernd E. Wolfinger, Leiter der Arbeitsgruppe Telekommunikation und Rechnernetze (TKRN), danke ich für die kritische Durchsicht dieser Arbeit und entsprechende Verbesserungsvorschläge.

Ganz herzlich möchte ich mich bei den Kollegen Frank Griffel, Dr. Michael Merz, Stefan Müller-Wilken, Marko Boger, Frank Wienberg, Olaf Kummer, Dr. Daniel Moldt, Harald Weinreich, Andreas Bartelt, Christian Zirpins und Volker Nötzold für viele inhaltliche Anregungen sowie tatkräftige Unterstützung und freundschaftliche Zusammenarbeit bedanken.

Mein besonderer Dank gilt den zahlreichen Studierenden, die vor allem mit ihrer aktiven Entwicklungstätigkeit im Rahmen von Studien- und Diplomarbeiten ganz erheblich zu dieser Arbeit beigetragen haben. Namentlich möchte ich Martin Göllnitz, Christian Kunze, Christian Langmann, Christian Seebode, Eberhard Wolff, Olaf Großler und Aureliusz Maresz erwähnen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Kaum ein anderes Gebiet der Informatik hat eine vergleichbare Verbreitung erfahren wie das der Verteilten Systeme in den zurückliegenden zwei Jahrzehnten. Rasante Fortschritte in Hardware-, Netzwerk- und Kommunikationstechnologie ermöglichten eine globale Vernetzung unterschiedlichster Rechnertypen, die auch immer weiter in die privaten Haushalte hineinreicht; es ist bereits eine *vernetzte Welt*, in der eine räumlich und zeitlich unbegrenzte Kommunikation zwischen beliebigen Menschen und Organisationen *praktisch* möglich ist, entstanden, die immer nachhaltiger das Kommunikations-, Arbeits- und Freizeitverhalten — und damit auch die gesamte Lebensweise — der Menschen beeinflusst und verändert[1]. Und obwohl Begriffe wie "Informationsgesellschaft" oder "Informationsrevolution" bereits abgedroschen erscheinen — wohl deswegen, weil es keines besonderen Vorstellungsvermögens bedarf, um den Stellenwert der Informationstechnologie und des damit verbundenen *Potenzials* zu erahnen — stehen wir sicherlich erst am Anfang dieser Revolution, sowohl zeitlich als auch technologisch gesehen. Denn es ist zwar gewiss, dass mit der existierenden Vernetzung bereits eine exzellente *Infrastruktur* für vielfältige neuartige Anwendungen vorhanden ist, jedoch ist es noch weitgehend unabsehbar — womit ein gewaltiges Potenzial für kreative Köpfe gegeben ist — *wie* solche Anwendungen in etwa zwanzig weiteren oder auch nur zehn Jahren aussehen werden. Es ist also zu erwarten, dass die Entwicklung bzw. Erfindung neuartiger Anwendungen bzw. von Technologien, die neuartige Anwendungen ermöglichen (engl. enabling technologies), schon in naher Zukunft größere Bedeutung erlangen wird als die Verbesserung der infrastrukturellen Bedingungen — trotz der weiterhin wachsenden Zahl von Anwendern — d.h. die Bedeutung verteilter Systeme wird sich immer stärker auf *verteilte Anwendungssysteme* verlagern.

Allerdings wird gerade durch diese immense Verbreitung in nahezu alle gesellschaftlichen Bereiche die Existenzberechtigung des Gebiets der Verteilten Systeme als selbständige Forschungsdisziplin und eigener Praxiszweig in zunehmendem Masse in Frage gestellt. Denn die bis heute immer rasanter werdende lokale und globale Vernetzung von Rechnern aller denkbaren Größen und Kapazitäten lässt Rechensysteme, die *nicht verteilt* sind, immer häufiger zu Ausnahmen werden. Auf der anderen Seite lassen das vielseitige Interesse und die da-

[1] Diese Veränderung drückt sich zunehmend auch in Form einer *Verwischung* der Grenzen traditioneller Konzepte aus. Siehe z.B. die zugleich provokative wie informative Diskussion in [DM98].

mit verbundenen zahlreichen Einflüsse aus sehr unterschiedlichen Bereichen — von den Wirtschaftswissenschaften bis zur Biologie — verteilte Systeme immer größere Bedeutung als Kommunikationsmedium, als Systeminfrastruktur und als *interdisziplinäres* Forschungsgebiet gewinnen. Diese Interdisziplinarität bzw. *Integrationscharakter*, welcher u.a. auch zur Prägung des so genannten *Community Computing* Paradigmas geführt hat (siehe [Rhe93, Ish98]), gilt insbesondere für die verteilten Anwendungssysteme, denn die auf der *Anwendungsebene* zur Verfügung stehende Funktionalität ist letztendlich dafür ausschlaggebend, welche Vielfalt von Problemen mit Hilfe eines Rechensystems gelöst werden kann.

Aus der genannten Durchdringung verteilter Anwendungssysteme und der Vielfalt bereits praktizierter als auch potentieller Nutzungsmöglichkeiten erwächst eine ebenso vielfältige und dynamisch wachsende Komplexität für die Entwicklung, Pflege, Verwaltung und Bedienung solcher Systeme, die einen entsprechend wachsenden Bedarf an *Steuerung* und insbesondere an dynamischer *Interaktionsunterstützung* für solche Systeme nach sich zieht.

Ziel der vorliegenden Arbeit ist es daher, Aspekte der Steuerung verteilter Anwendungssysteme zu untersuchen und entsprechende Lösungsansätze anzubieten. Dabei sind grundsätzlich zwei unterschiedliche Formen der Steuerung möglich: Bei der *Fremdsteuerung* wird von außerhalb des Systems Einfluss auf die Verhaltensweise der Komponenten geübt, während bei der *Selbststeuerung* die Systemkomponenten eigenständig versuchen, eventuelle Konflikte (bzgl. ihrer Interaktion) aufzulösen. Entsprechend werden im Rahmen dieser Arbeit zwei Hauptkonzepte ausgearbeitet: Einerseits werden spezifische, *interaktionsorientierte Regelverarbeitungsmechanismen* entwickelt und eingesetzt, um das Verhalten offener verteilter Anwendungen mittels externer Regeln, die von einem Benutzer *dynamisch* in das System eingegeben werden können, partiell zu steuern, ohne die Autonomie und die (ursprüngliche) Funktionalität der Anwendungen prinzipiell zu beeinträchtigen. Andererseits werden neuartige *Verhandlungsmechanismen* präsentiert, mit denen unterschiedliche Kooperationsanforderungen von zwei oder mehr Anwendungskomponenten *automatisch ausgehandelt* werden können. Dabei ermöglichen die Regelverarbeitungsmechanismen eine Form der o.g. Fremdsteuerung, während autonom verhandelnde Anwendungen als eine Form der *interaktionsbasierten* Selbststeuerung bzw. Selbstanpassung in einer offenen verteilten Umgebung betrachtet werden können.

Trotz des Anspruchs auf *Generizität*, der durchgehend mit den hier vorgestellten Mechanismen verknüpft wird, verfolgt diese Arbeit keine rein konzeptionellen bzw. theoretischen Lösungsansätze, die aus einer entsprechend *abstrakten* Modellierung verteilter Systeme abgeleitet werden können, sondern solche, die konkret in (möglichst vielen) *praktischen* Anwendungssystemen bzw. -szenarien eingesetzt werden können. Daher wurde frühzeitig ein konkreter Anwendungsbereich, nämlich der des elektronischen Handels sowie *elektronischer Dienstemärkte*, der innerhalb des Gebiets Verteilte Systeme Gegenstand zahlreicher aktueller Industrie- und Forschungsaktivitäten ist (siehe z.B. [LM98, GTL98]), als Anwendungs- bzw. Testumgebung für die hier behandelten Steuerungsmechanismen ausgewählt, um deren praktische Anwendbarkeit zu gewährleisten. Deshalb folgt in diesem Kapitel zunächst eine Einführung in elektronische Dienstemärkte in Abschnitt 1.1. Es folgen dann Beschreibungen der für diese Arbeit grundlegenden *dynamischen Aspekte* verteilter Anwendungssysteme (Abschnitt 1.2) und der *Basistechnologien* für elektronische Dienstemärkte (Abschnitt 1.3). Anschließend wird in Abschnitt 1.4 die Zielsetzung dieser Arbeit konkretisiert sowie

ein Überblick über die weiteren Kapitel gegeben.

1.1 Die Entstehung global vernetzter Dienstemärkte

Um den Anwendungskontext für die relevanten Konzepte und Techniken in dieser Arbeit zu beleuchten, wird in diesem Abschnitt das Gebiet der elektronischen Märkte vorgestellt. Insbesondere sollen dabei diejenigen Aspekte des Gebietes aufgezeigt werden, wo *Steuerungsmechanismen* einen konkreten Beitrag zur Verbesserung der systemtechnischen Unterstützung so genannter elektronischer Dienstemärkte leisten können.

Der schnell wachsende Deckungsgrad der globalen Netze, insbesondere des Internets, die zunächst als Kommunikationsmedium entstanden sind, weckt zunehmend auch das Interesse nach deren kommerzieller Nutzung[2], denn in Bezug auf die *Zugangsmöglichkeiten* stellt das Internet eine ideale — weil sowohl räumlich als auch zeitlich unbegrenzte — Infrastruktur für Handelstransaktionen bzw. -beziehungen dar. Aus kommerzieller Sicht kann es also in seiner Gesamtheit als ein *virtueller Marktplatz* betrachtet werden, der vierundzwanzig Stunden am Tag und sieben Tage in der Woche "geöffnet" hat und von praktisch jedem Ort der Welt erreichbar ist. Umgekehrt wird natürlich die Entwicklung und Nutzung globaler Netze auch zunehmend beeinflusst und vorangetrieben von kommerziellen Faktoren. Um die Wechselwirkungen zwischen der globalen Vernetzung und ihrer kommerziellen Nutzung zu untersuchen, hat sich in der Literatur der Begriff *elektronische Märkte* [Mer99b] herausgebildet, der im allgemeinen die Abbildung des konventionellen Marktbegriffes im Sinne eines (abstrakten) ökonomischen Orts des Gütertausches auf eine verteilte Systeminfrastruktur bezeichnet [Sch93b].

Um eine bessere technische Modellierung und damit auch verbesserte *systemtechnische Unterstützungsmechanismen* zur Entwicklung konkreter elektronischer Marktsysteme zu ermöglichen, wurde seit einigen Jahren im Bereich Verteilte Systeme der Begriff der "elektronischen Dienstemärkte" (EDMs) etabliert — maßgeblich durch die Veröffentlichungen von Merz et al. [Mer96, Mer99a]. Im Mittelpunkt dieser Bezeichnung und der entsprechenden Ansätze zur Modellierung des Online–Handels steht der *Dienstbegriff*, der eine technische Abstraktion *sowohl* aller Leistungen, die zwischen den Marktteilnehmern ausgetauscht werden *als auch* der systemtechnischen Funktionen, die von diesen genutzt werden, darstellt. Somit ermöglicht diese Abstraktion ein weitgehend einheitliches Modellieren und Entwerfen von Marktarchitekturen.

Dienstbegriff Der Dienstbegriff tritt in vielen verschiedenen Kontexten auf — von Datenkommunikation über Softwaretechnik bis hin zur Ebene der informalen Beschreibung von ökonomischen *Dienstleistungen* — und bezeichnet im allgemeinen die Abstraktion einer Menge von *Einzelfunktionalitäten*, die von einem *Diensterbringer* angeboten und einem (oder mehreren) *Dienstnutzer* genutzt werden. Im informationstechnischen Kontext hängt dieser Begriff sehr eng

[2] Der Schätzung von Forrester Research (www.forrester.com) zufolge wird der Online–Einzelhandel bis 2001 einen Umsatz von 17 Mrd. $US erreichen.

mit dem Paradigma der *Objektorientierung* zusammen, der von Bertrand Meyer wie folgt charakterisiert wird:

> **Definition 1.1.1** *Object–oriented software construction (definition 1): Object–oriented software construction is the software development method which bases the architecture of any software system on modules deduced from the types of objects it manipulates (rather than the function or functions that the system is intended to ensure). [Mey97, S. 116]*

Eine solche Objektsicht legt also besonderen Wert auf die *Dekomposition* eines Systems in Teilkomponenten (Module). Dies gilt ebenso für die Dienst–orientierte Sicht bzw. Dienstsicht, allerdings ist bei dieser ausschließlich die *Deklaration* der (Teil-)Komponenten von Bedeutung, während die Objektsicht auf die gesamte Realisierung des Systems zielt. Insofern bietet die Dienstsicht im Gegensatz zur Objektsicht eine wesentlich höhere Abstraktion, bei der lediglich die sog. *Schnittstellen*, die in erster Linie zur Deklaration der Funktionalität von Komponenten dienen, eine Rolle spielen. Insofern scheint allerdings der in dieser Definition herausgestellte Gegensatz zwischen "Objekt" und "Funktion" nicht in dem gleichen Masse auf die Dienstsicht zuzutreffen[3]; hier liegt das Augenmerk vielmehr auf dem Unterschied zwischen "Deklaration" und "Realisierung", wie die Darstellung der Dienstmodelle und entsprechender Schnittstellentypen in Abschnitt 2.1.2 deutlich machen wird.

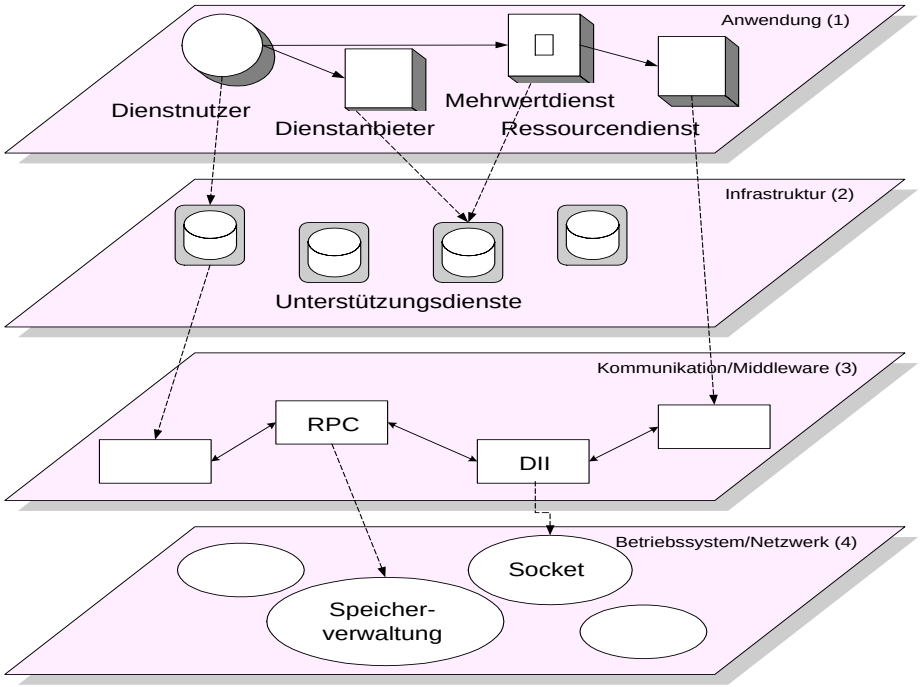


Abbildung 1.1: Dienst–basiertes Schichtenmodell elektronischer Märkte

[3]Meyer betont auch die Tatsache, dass die Funktionalität der Objekte früher oder später festgelegt werden muss; jedoch kommt es bei der Objektorientierung darauf an, *zuerst* die Dekomposition vorzunehmen.

Basierend auf dem Dienstbegriff ist es nun möglich, ein hierarchisches Modell elektronischer Märkte zu definieren, deren Komponenten — unabhängig von der systemtechnischen Ebene, der sie zugeordnet werden — ausschließlich als Dienste betrachtet werden. In Abbildung 1.1 ist eine typische Ausprägung eines solchen Modells, nämlich ein Schichtenmodell, illustriert. Die beiden unteren Schichten können gemeinsam der *Systemebene* zugeordnet werden und bieten weitgehend standardisierte Dienste zur Entwicklung offener verteilter Anwendungen im allgemeinen wie Netzwerk-, Kommunikations- und globale Namensdienste. Insbesondere die Dienste der Schicht 3 — die sog. Middleware–Schicht — haben aufgrund weltweiter Kooperationsbemühungen in den letzten Jahren große Fortschritte erzielt und ermöglichen bereits heute eine praktisch unbegrenzte Interoperabilität der verteilten Anwendungen auf der programmiertechnischen und -sprachlichen Ebene (siehe hierzu Abschnitt 1.3). Die beiden oberen Schichten können dagegen gemeinsam der *Marktinfrastrukturebene* zugewiesen werden und sind gegenwärtig gerade im Entstehen, d.h. von den zugehörigen Diensten gibt es zwar bereits lauffähige Forschungsprototypen und in einigen Fällen auch Produkte, aber diese unterziehen sich kontinuierlich wesentlichen Veränderungen. Vor allem existieren für die sog. Unterstützungsdienste der Schicht 2 — die beispielsweise Dienste zur Vermittlung von Produkten und Geschäftspartnern, zur Aushandlung von Verträgen, zur Ausführung, Koordinierung und Absicherung von Handelstransaktionen umfassen — noch weitgehend keine geeigneten Standards, um eine Interoperabilität der Produkte auf dieser Infrastrukturebene zu sicherzustellen. Viele der in dieser Arbeit behandelten Aspekte sind speziell in bezug auf elektronische Dienstemärkte auf dieser Ebene anzusiedeln.

1.2 Dynamische Aspekte offener verteilter Anwendungssysteme

Offene verteilte Anwendungssysteme im allgemeinen und elektronische Dienstemärkte im besonderen sind von einem außergewöhnlich hohen Masse an *Dynamik* gekennzeichnet. Obwohl diese Tatsache offensichtlich erscheint, ist es von großer Bedeutung für den Entwurf und Entwicklung von Marktinfrastrukturen, diese Dynamik in ihren einzelnen Aspekten und deren Wechselwirkungen zu untersuchen, um adäquate systemtechnische Lösungen realisieren zu können. In diesem Abschnitt wird eine Analyse der dynamischen Aspekte elektronischer Marktsysteme in offenen verteilten Umgebungen vorgenommen.

1.2.1 Offenheit

Die Dynamik von Anwendungssystemen, mit deren Entwicklung und Steuerung die vorliegende Arbeit sich beschäftigt, entstammt überwiegend ihrer Offenheit. Der Begriff der Offenheit spielt eine entscheidende und immer weiter wachsende Rolle in verteilten Systemen. Ihrer Bedeutung, welche frühzeitig erkannt worden war, wurde durch weltweit operierende Gremien Rechnung getragen. Insbesondere sind die Standardisierungsbemühungen der *International Standardization Organization* (ISO) zu nennen, die zum Referenzmodell des *Open Distributed Processing* [ODP94a] führten. Obwohl derartige Aktivitäten sehr hilfreich für

die Steigerung der Interoperabilität verteilter Systeme waren[4], können sie immer nur einen Teil der zahlreichen Facetten der Offenheit heutiger Systeme, die sich in einem vielseitigen, fortschreitenden Evolutionsprozesse befinden, erfassen.

Je nachdem, inwieweit diese einzelnen Facetten erfasst werden sollen, lassen sich unterschiedlich abstrakte Definitionen des Offenheitsbegriffes aufstellen. Beispielsweise definiert [Tsc94] ein offenes Kommunikationssystem wie folgt:

1. Ein offenes System ist *nicht begrenzt* auf Städte, Regionen, Länder, Kontinente, aber auch Verwaltungs- und Unternehmensstrukturen.

2. Es ist *frei zugänglich*, d.h. benutzeroffen und juristisch offen.

3. Es ist *heterogen*, d.h. bestimmte Hardware-, System- oder Anwendungsarchitekturen werden nicht vorausgesetzt oder forciert. D.h. für einen Netzteilnehmer besteht Planungsfreiheit bzgl. seiner individuellen Softwareentwicklung.

4. Es ist *selbstbestimmend* und *unabhängig*, d.h. Reaktionen auf beobachtbare Ereignisse sind nicht vorhersagbar. Anwendungen sind autonom und nicht durch eine zentrale Administration steuerbar.

5. Es ist *dezentralisiert* — insbesondere auf der Anwendungsebene — also ohne globale Kontrolle.

Im Gegensatz zu einer solchen *extensionalen* Definition, die auf die Charakterisierung einzelner Aspekte von Offenheit abzielt, kann die folgende Beschreibung als eine *intensionale* Definition eines offenen Systems betrachtet werden:

> A computing system is said to be open if its computations depend on external information and is said to be closed otherwise. [Weg96, S. 2444]

Diese gleichermaßen elegante wie präzise Umschreibung von Offenheit reduziert diesen Begriff jedoch letztendlich auf die Unterscheidung zwischen externer und interner Information und setzt daher die Spezifikation der System*grenzen* voraus, die in der Praxis wiederum häufig nur extensional im Sinne der ersten Definition möglich ist. Dennoch bietet diese zweite Definition sicherlich auf einer hohen Abstraktionsstufe eine prägnante Regel zur Unterscheidung zwischen offenen und geschlossenen Systemen.

Allgemein lässt sich der Offenheitsbegriff in zwei Hauptkategorien unterteilen: *Benutzer-* bzw. *Markt*offenheit und *System-* bzw. *technische* Offenheit. Da letztere sehr eng mit den (technisch spezifischeren) Begriffen der Interaktion bzw. der Heterogenität/Interoperabilität zusammenhängen, wird sie hier gesondert unter den entsprechenden Bezeichnungen dargestellt. Im verbleibenden Teil dieses Abschnitts wird zunächst der Offenheitsbegriff in bezug auf die erste Kategorie charakterisiert.

Der Marktbegriff an sich beinhaltet immer ein gewisses Mass an Offenheit, sowohl in bezug auf die Nachfrage als auch das Angebot. Aufgrund dieser Offenheit ist erst eine Fluktuation von Teilnehmern, Waren und Preisen und damit

[4] und durchaus zu praktischen Diensten wie z.B. dem *Trading* zur Dienstvermittlung [ODP95] führten, die sich auch in konkreten Produkten wiederspiegeln.

eine *Wettbewerbssituation*, die inhärent mit dem Marktbegriff verbunden ist, gegeben. Um ein effektives Handeln zu ermöglichen, wird diese Offenheit jedoch in vielen Fällen — vor allem auf Angebotsseite — gezielt eingeschränkt, wodurch *spezialisierte* Markttypen wie z.B. ein Gebrauchtwarenmarkt oder eine Wertpapierbörse entstehen. Im Gegensatz zu konventionellen Märkten weisen EDMs aufgrund ihrer *Virtualität*, d.h. ihrer Ungebundenheit an einen geographischen Ort und eine bestimmte Zeit, eine neue, vor dem Zeitalter des WWW unbekannte, Qualität der Offenheit aus. Solche Märkte ermöglichen in der Tat ein Handeln beliebiger Waren von beliebigen Orten aus und zu beliebigen Zeiten. In der Praxis eröffnet diese Offenheit neue Potenziale für Geschäftstransaktionen, die weitgehend noch nicht ausgeschöpft worden sind und gewiss auch ganz neue Risiken und Probleme mit sich bringen. Diese neue Offenheit der Online–Kommerz derart ungewohnt, dass beispielsweise die gängigen Steuerregelungen außer Kraft gesetzt werden und neue Gesetze nachträglich geschaffen werden müssen.

1.2.2 Das Interaktionsparadigma

Die Entwicklung von Softwareanwendungen im allgemeinen wird immer stärker von der Anforderung nach Interaktionsfähigkeit beeinflusst und bestimmt, denn mit der fortschreitenden Verteilung und Dezentralisierung der Rechensysteme ergibt sich die erforderliche Funktionalität immer öfter erst aus dem Zusammenspiel der einzelnen Rechner bzw. der darauf operierenden Softwarekomponenten. Aus Gründen der Erweiterbarkeit (einschließlich der Skalierbarkeit) und Wiederverwendbarkeit ist dieses Zusammenspiel in der Regel jedoch nicht auf eine statische Weise zum Zeitpunkt der Erstellung der Einzelkomponenten spezifizierbar, sondern muss möglichst ad–hoc und dynamisch zu Stande gebracht werden können, d.h. die Einzelkomponenten müssen apriori für eine größtmögliche Fähigkeit zur Kooperation mit Fremdkomponenten entworfen bzw. technisch unterstützt werden. Dieses Ziel konsequent zu verfolgen erfordert in der Tat eine vollkommen neue Sichtweise und entsprechende Methodik im Vergleich zu den traditionellen zentralen bzw. geschlossenen Systemen. Einige Autoren sprechen deshalb bereits von dem neuen Programmierparadigma der *Interaktionsorientierung* und versuchen u.a. auch eine fundierte theoretische Grundlage dafür zu entwickeln.

Am prägnantesten wird das Interaktionsparadigma in den Arbeiten Wegners [Weg97, Weg98, WG99a] vertreten. Darin argumentiert er, dass das Verhalten eines offenen verteilten Systems zu einem bestimmten Zeitpunkt nicht algorithmisch im Sinne Turings/Churchs [Weg96] modellierbar sei, da es abhängig von seinem aktuellen *Zustand* ist, der wiederum von vorherigen, *dynamisch* generierten — d.h. zum Entwurfszeitpunkt unbekannten — Eingaben abhängt, also *historiebehaftet* ist. Um seine These zu unterstützen, liefert Wegner eine Reihe von Argumenten, die von alltäglichen Analogien — wie dem Vergleich zwischen Kaufvertrag (Algorithmus) und Ehevertrag (offenes System) [Weg97] — bis hin zu formal–mathematischen Modellen, die auf so genannten *Interaktionsmaschinen* basieren, reichen. Dabei stellt eine Interaktionsmaschine eine Erweiterung der Turingmaschine um mindestens eine dynamische Eingabesequenz, die *während* der Berechnung modifiziert werden kann, dar. Informal betrachtet lässt sich eine solche Maschine deshalb ausdrucksstärker als eine Turingmaschine, da sie zeitabhängige Veränderungen in ihrer Berechnung der Ausgabe mit

berücksichtigt (und deshalb nicht auf eine mathematische Funktion, die zu jeder Eingabe immer die gleiche Ausgabe generiert, reduziert werden kann), während eine Turingmaschine zeitunabhängige Transformationen durchführt:

> Interaction machines are intuitively more expressive than Turing machines because they model the passage of time during the process of problem solving, whereas algorithms and Turing machines model only time–independent transformations. [Weg96, S. 2442]

Formal lässt sich die These der Nichtreduzierbarkeit von Interaktionsmaschinen (IMs) auf Turingmaschinen (TMs) dadurch nachweisen, dass IMs in ihrer schwächsten Form[5] gleich ausdrucksstark sind wie so genannte persistente Turingmaschinen (PTMs), die wiederum stärker sind als TMs. Die formale Argumentation ist in [WG99b, S. 10ff] zu finden. Aufgrund dieses negativen Ergebnisses — der Nichterfassbarkeit von IMs und damit von offenen verteilten Systemen im allgemeinen durch Algorithmen und Funktionen — drängt sich zugleich die Frage nach mathematischen Modellen auf, die interaktive Systeme adäquater beschreiben können, denn die Church–Turing–These, dass der intuitive Begriff der Berechenbarkeit der Mächtigkeit des Lambda–Kalküls bzw. der Turingmaschinen entspreche, scheint damit ungültig geworden und auch nicht mehr zeitgemäß zu sein, insbesondere angesichts der Tatsache, dass heutige Computersysteme massiv verteilt und offen sind. Allerdings sind formale Modelle für interaktive Systeme gerade erst im Entstehen, und Wegner, dessen Arbeiten maßgeblich dazu beitragen, schreibt am Ende seiner (zusammen mit Dina Goldin verfassten) Abhandlung "Mathematical Models of Interactive Computing":

> We are only beginning to understand how to formalize the richer behaviors expressible through interaction. Non–well–founded sets appear to be a stable extension of set theory for expressing sequential interaction. Further extension to handle distribution has not yet been mathematically formalized, but the idea that such interactions are definable by modifying the anti–foundation axiom suggests that distributed modes of interaction could in principle be set–theoretically formalized and proved consistent. This is one of the many open problems to be addressed in developing a comprehensive mathematical theory of interaction. [WG99b, S. 41]

Die Kernaussage in dem zitierten Abschnitt lautet, dass alle wesentlichen Hürden zu einer formalen Theorie interaktiver Systeme mit der *Verteilung*[6] von Komponenten zusammenhängen. Damit deckt sich diese theoretische Einsicht weitgehend mit den Praxiserfahrungen über die Vielfältigkeit und Komplexität offener verteilter Systeme, die Entwickler und Benutzer täglich machen[7].

[5] IMs werden unterteilt in Sequentielle (SIMs) und Multi–Stream Interaktionsmaschinen (MIMs), wobei die MIMs zu einer noch höheren Ausdrucksklasse gehören.

[6] Wegner verwendet den (formalisierten) Begriff der *Beobachtung*, um das Verhalten interaktiver Systeme zu modellieren. So besteht in seinen Arbeiten der Zusammenhang zwischen dem Begriff der Verteilung und dem der Beobachtung darin, dass eine *entfernte* Komponente eine ist, die nicht direkt beobachtet werden kann.

[7] Modelle der Interaktion wie die von Wegner zielen nicht nur auf theoretische, sondern streben durchaus praxisrelevante Ergebnisse an: "The extension of models of computation from TMs to SIMs and MIMs is of more than theoretical interest, since both SIMs and MIMs

Angesichts einer noch fehlenden allgemein gültigen theoretischen Grundlage für offene verteilte Systeme und vor allem angesichts einer real existierenden Lücke zwischen Theorie und Praxis, die sich häufig schon aus der *Komplexität* realer Systeme ergibt, stellt sich aber für diejenigen, die komplexe Anwendungssysteme entwerfen und realisieren müssen, die Frage, welche *praktische* Konsequenzen bzw. konkrete *Leitlinien* aus solchen grundlagenorientierten Erkenntnissen für die Softwareentwicklungspraxis abgeleitet werden können. Auch diesbezüglich bieten Wegners Arbeiten eine Reihe von Inspirationen, deren Kernaussage die ist, dass das Ziel einer vollständigen Verhaltensspezifikation bzw. eines vollständigen Korrektheitsnachweises von offenen interaktiven Softwaresystemen nicht zu erreichen sei und deshalb zu Gunsten einer *partiellen Systemspezifikation* aufgegeben werden soll:

> Giving up the goal of complete behavior specification requires a psychological adjustment but makes partial system specification respectable. Though a complete elephant cannot be specified, its parts and its forms of behavior (its trunk or modes of eating peanuts) are specifiable. Complete specification must be replaced by the more modest goal of partial specification by interfaces, views and modes of use. [Weg96, S. 2444]

Obwohl das befürwortete Ziel einer partiellen Systemspezifikation hier als "bescheiden" bezeichnet wird, ist dieses Attribut durchaus nicht "einschränkend" oder gar "negativ" gemeint, sondern vielmehr im Sinne einer neuen Sicht (bzw. eines neuen Paradigmas) zu verstehen, die von den statischen, streng logik–basierten Vorstellungen des Berechenbarkeitsbegriffes "befreit" und die Erforschung ausdrucksmächtigerer — und damit die realen Systeme adäquater erfassender — Modelle einleitet.

Die in der vorliegenden Arbeit dargestellten praxisnahen Steuerungsmechanismen gehen exakt von der These der Unerreichbarkeit von vollständigen Verhaltensbeschreibungen offener verteilter Systeme mittels Algorithmen und klassischer Logiken aus und machen direkt Gebrauch von der hier zitierten partiellen Systemspezifikation. Allerdings wird sich herausstellen bzw. im Verlauf der Arbeit gezeigt werden, dass gerade logik–basierte Formalismen in vielen Fällen sowohl zur partiellen Systembeschreibung als auch zur effektiven Unterstützung des Interaktionsverhaltens verteilter Anwendungssysteme sehr gut geeignet sind und damit keineswegs "im Widerspruch" zum Interaktionsparadigma stehen, sondern als deren sinnvolle Ergänzung betrachtet werden können.

1.2.3 Heterogenität

Heterogenität bezeichnet die Vielfalt an Hard- und Softwarearchitekturen, auf denen verteilte Anwendungen entwickelt und operiert werden können. Insbesondere steht sie für die Vielfalt an Betriebssystemplattformen, Netzwerkprotokollen, Programmiersprachen und Schnittstellenspezifikationsformaten, mittels derer die Anwendungen realisiert werden. Es ist einleuchtend, dass diese technische Vielfalt an sich eine wesentliche Quelle für die Dynamik sowie für zahlreiche Probleme verteilter Systeme sein kann.

capture important natural classes of problems. SIMs express two–agent interaction, including traditional object–oriented and agent–oriented sequential models, while MIMs express multiagent interaction and collaborative behavior." (ebd.)

Trotz einzelner Monopolisierungstendenzen etwa auf dem Gebiet der Betriebssysteme kann in der Praxis nicht davon ausgegangen werden, dass die Heterogenität irgendwann gänzlich aufgehoben wird[8], sondern vielmehr muss bei der Entwicklung verteilter Systeme eine solche Vielfalt immer als vorgegebene Randbedingung miteinkalkuliert werden und dementsprechend müssen konkrete Verfahren für einen möglichst effizienten Umgang mit ihr gefunden bzw. eingesetzt werden. Die (gemeinsame) Fähigkeit von verteilten Systemen, die Heterogenität zu überwinden, wird als *Interoperabilität* bzw. im allgemeinen als *technische Offenheit* bezeichnet.

Um Interoperabilität zu erreichen, ohne in die spezifische Architektur und Technologie der beteiligten Komponenten/Systeme einzugreifen bzw. sie verändern zu müssen, werden Mechanismen der *Vermittlung* (engl. mediation) [Wie95a] und Adaption [YS95] eingesetzt, d.h. es werden Mittlerkomponenten zwischen den technisch inkompatiblen Systemen eingeschaltet, die die Inkompatibilitäten (bspw. unterschiedliche Methodensignaturen) mittels geeigneter Datentransformationsverfahren eliminieren (siehe Abb. 1.2). Deshalb kann Interoperabilität auch als eine *vermittelte Interaktion* (engl. mediated interaction) betrachtet werden [Weg96, S. 2449].

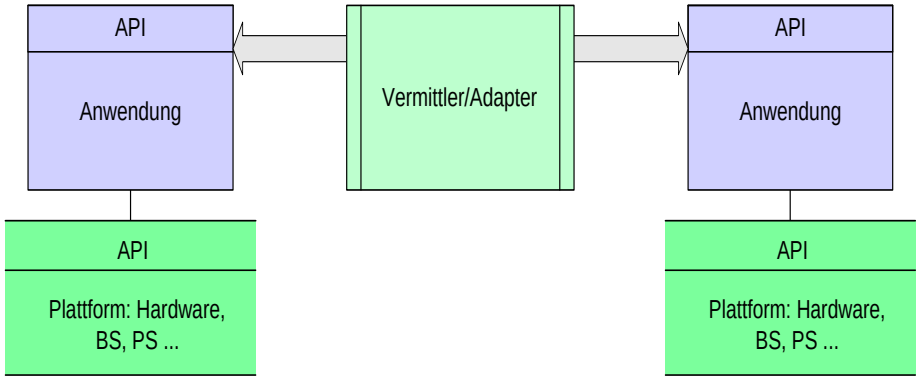


Abbildung 1.2: Herstellung von Interoperabilität mittels einer Vermittlungsinstanz

Da es aber offensichtlich sehr ineffizient wäre, für je zwei inkompatible Anwendungskomponenten einen spezifischen Vermittler bzw. Adapter einzusetzen, wurde Anfang der neunziger Jahre das Konzept der *Middleware* vorgeschlagen [Ber93] und seitdem von mehreren Industriekonsortien in unterschiedliche Produkte umgesetzt. Das Middleware–Konzept siedelt die Aufgabe der Gewährleistung von Interoperabilität in einer Vermittlungs*schicht* an und verlegt diese damit auch von der Anwendungs- in die Systemebene. Dabei unterstützt die Vermittlungsschicht die Kooperation zwischen beliebig vielen inkompatiblen Komponenten, die mittels unterschiedlicher Programmiersprachen (PS) und Betriebssystemplattformen (BS) etc. implementiert sein können (siehe Abb. 1.3).

Einer der größten Vorteile der Middlewarearchitektur besteht darin, dass sie nicht nur von den technischen Unterschieden in der Implementierung der einzelnen Anwendungen, sondern ebenso von ihrer *Verteilung* abstrahiert, so dass

[8] Ein solcher Zustand würde die Ausschaltung des Wettbewerbs auf dem entsprechenden Gebiet bedeuten und wäre äußerst nachteilig für die Kreativitätsentfaltung.

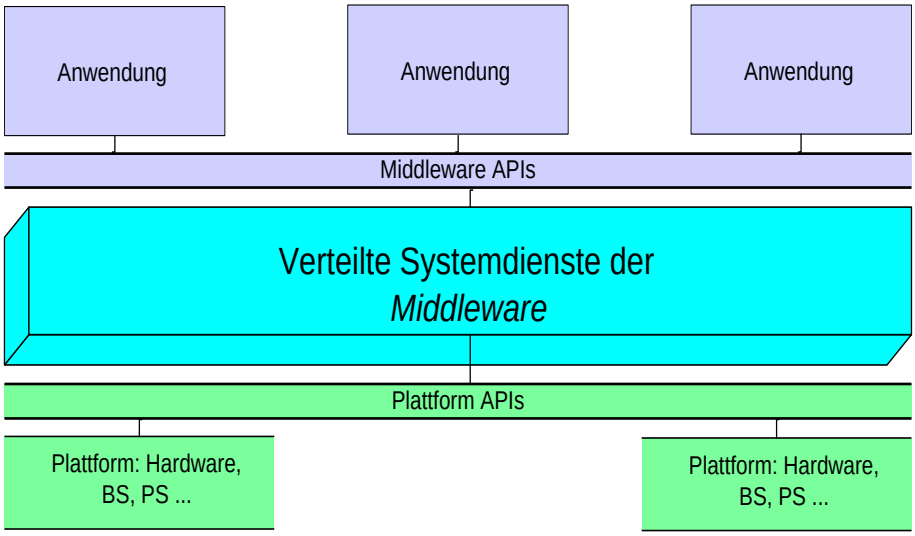


Abbildung 1.3: Middleware als Vermittlungsschicht zwischen Anwendungen und spezifischen Plattformen

eine Kooperation zwischen entfernten (heterogenen) Anwendungen sich in der gleichen Weise realisieren lässt wie eine Kooperation zwischen lokalen (homogenen) Anwendungen. Um diese mächtige Abstraktion zu gewährleisten, besteht die Middleware selbst aus einer Menge *verteilter* Systemdienste, die zusammen einer Anwendung mindestens die Funktionalität eines Betriebssystems anbietet, jedoch über Rechner-, Netzwerk- und Plattformgrenzen hinweg. Auf die Funktionalität solcher Middlewaredienste wird in Abschnitt 1.3.3 anhand einer konkreten Middlewarearchitektur näher eingegangen.

1.3 Basistechnologien für elektronische Dienstemärkte

Die in der vorliegenden Arbeit dargestellten Konzepte und Lösungen zielen sowohl auf generische Steuerungsverfahren für offene verteilte Anwendungssysteme im allgemeinen als auch auf eine *konkrete Realisierung* von Steuerungsmechanismen im Sinne einer Systemunterstützung für elektronische Dienstemärkte im besonderen. Daher ist es notwendig, das technologische Umfeld, in dem elektronische Dienstemärkte (gerade erst) entstehen, zu untersuchen, um erstens die technischen Anforderungen an die angestrebten Lösungen zu erfassen und zweitens um festzustellen, welche Werkzeuge, Produkte und spezifische (Programmier-)Techniken bereits vorhanden sind, die zur Realisierung der Lösungsansätze verwendet können. Dies gilt um so mehr, als z.Z. weder konkrete etablierte Marktinfrastrukturen noch entsprechende gültige, allgemein anerkannte (technische und rechtliche) Standards existieren, so dass für eine konkrete Umsetzung häufig erst entsprechende Dienstemarkt*szenarien* implementiert werden müssen, die direkt auf den in diesem Abschnitt vorgestellten Technologien aufsetzen. Jedoch soll hier zunächst nur ein allgemeiner Überblick einiger

ausgewählten Basistechnologien gegeben werden, um den technologischen Hintergrund, auf dem diese Arbeit fußt, zu beleuchten. Weitere Details, die für das Verständnis der einzelnen Lösungen notwendig sind, werden an den entsprechenden Stellen näher beschrieben.

1.3.1 Von *Software* über *Componentware* zu *Communityware*

Die globale Vernetzung durch das Internet als einen Verbund von immer mehr Netzwerken und Rechnern aller Art, die immer leistungsfähiger und gleichzeitig preiswerter werden, hat in den letzten Jahren nicht nur die Verbreitung von Informationstechnologie beschleunigt, sondern diese Technologie selbst — und insbesondere die Sicht auf die Softwareentwicklung — auch immer nachhaltiger verändert. Mit der prinzipiell uneingeschränkten Vernetzung scheint es bei der Entwicklung von Software im allgemeinen immer ratsam zu überlegen, wie das Potenzial der Verteilung einer Applikation auf mehrere Rechner bzw. der Kooperation mit bestehenden Applikationen ausgenutzt werden kann[9].

Viel mehr noch scheint sich der Gedanke durchzusetzen, sich des Internets als Medium für einen *globalen Baukasten* fertiger Softwarekomponenten [MJ97] zu bedienen und Anwendungen nicht mehr im herkömmlichen Sinne Zeile für Zeile zu programmieren, sondern aus den fertigen Teilen *zusammenzusetzen* und eventuell den jeweiligen Erfordernissen entsprechend zu *konfigurieren*. Hinter dieser visionär erscheinenden Sicht, die in der Praxis häufig mit Schlagworten und Paradigmen wie "Object Web" [OHE99], "Megaprogramming" [WWC92] und "Componentware" [Gri98] assoziiert wird, steckt die simple Überlegung, dass Softwareanwendungen einerseits immer größer und komplexer werden — und daher kaum im konventionellen Sinne programmiert werden *können* — und andererseits immer mehr (Teil-)Funktionalitäten bereits irgendwo realisiert worden sind — und neue Anwendungen daher nicht "von null an" programmiert werden *sollen*. Darüberhinaus ist es offensichtlich, dass ein Zusammenstecken von Anwendungen, beispielsweise per "Drag–and–Drop" in einer visuellen Umgebung, grundsätzlich einfacher sein soll als das Programmieren im klassischen Sinne, und daher auch von Anwendern, die keine Programmiersprachen beherrschen, bewerkstelligt werden könnte.

Obwohl das Componentware–Paradigma auch erst im Entstehen ist und bisher keine einheitlichen Definitionen dafür existieren, besteht ein besonderer und grundlegender Aspekt dieser Sicht darin, dass die Einzelkomponenten, aus denen eine Anwendung zusammengebaut werden kann, als durchkompilierte, lauffähige Objekte — d.h. als *Instanzen* im Gegensatz zu *Klassen* im objektorientierten Sinne[10] — betrachtet werden. Daraus ergeben sich unmittelbar zwei Konsequenzen: Erstens können die komponenten–basierten Anwendungen nicht auf eine statische Weise aus bestehenden Klassen erstellt (und durchkompiliert) sondern müssen auf eine *dynamische* Weise aggregiert werden. Die dadurch entstehende Anwendung besitzt also eine *inhärente Dynamik*[11]. Zweitens sind solche Anwendungen inhärent *interaktiv*, denn die Einzelteile, aus denen sie sich

[9] Umgekehrt muss natürlich auch überlegt werden, welche Gefahren für die Applikation (bspw. unerlaubte Zugriffe über das Netzwerk) von der Vernetzung der Rechner ausgehen.

[10] In der Tat scheint dies das einzige Merkmal zu sein, das die Komponenten–Orientierung von der Objekt–Orientierung *technisch* unterscheidet.

[11] und Verteilung, falls die Einzelkomponenten auf unterschiedlichen Rechnern laufen.

zusammensetzen, werden grundsätzlich als Blackbox betrachtet, d.h. es spielt nur das äußere beobachtbare Verhalten, nicht aber die innere Struktur (bspw. mit welchem konkreten Algorithmus und welcher Programmiersprache implementiert) eine Rolle. Aus diesem Grund passt das Componentware–Paradigma vorzüglich zu dem oben (in Abschnitt 1.2.2) beschriebenen Interaktionsparadigma, in dessen Literatur die komponenten–orientierte Sicht häufig auch als "programming in the large" bezeichnet wird. Allerdings ist diese etwas überplakative Bezeichnung keineswegs zu verwechseln mit Programmieren großer bzw. komplexer Systeme:

> Programming in the small (PIS) is algorithmic, whereas programming in the large (PIL) deals not with large programs but with interactive systems. An algorithmic program with a million arithmetic operations is not PIL, whereas medium–size embedded systems are. PIL is not simply scaled–up PIS; it has qualitatively different program structures and models of computation. The irreducibility of interaction to algorithms implies inexpressibility of PIL by PIS. [Weg96, S. 2444]

Noch visionärer und in der funktionalen Abstraktion höher angesiedelt als das Componentware–Paradigma drückt sich die interaktionsorientierte Sicht auf verteilte Anwendungssysteme in dem Begriff der *Communityware* bzw. des *Community Computing* [Ish98] aus. Während Componentware noch primär den Aspekt der technischen Realisierung von Softwaresystemen an sich zum Gegenstand hat, betont die Communityware–Sicht vielmehr die nahtlose Verschmelzung von netzwerkbasierter Softwaretechnologie und Anwendern zu so genannten virtuellen Gemeinschaften (engl. virtual community) [Rhe93]. Eine solche Gemeinschaft gründet sich ebenso wie "natürliche" Gemeinschaften auf einer Menge gemeinsamer Interessen, deren Durchsetzung durch die Zusammengehörigkeit der Mitglieder bzw. durch deren (sozialen) Interaktionen erleichtert werden soll. In [INH98] werden fünf konkrete Funktionen aufgelistet, die zur Unterstützung des Interaktionsverhaltens dienen:

- Einander kennen lernen
- Teilen von Wissen und Präferenzen
- Konsens erzielen
- Unterstützung des Alltagslebens
- Teilnahme an sozialen Ereignissen

Die Zielsetzung von Community Computing besteht nun darin, optimale Systemunterstützung für derartige Funktionen zu schaffen, wobei möglichst *vereinheitlichende* Modelle, in denen sowohl Systemkomponenten als auch Anwender als aktive *Rollen* abgebildet und somit anwendungssemantisch ununterscheidbar sind, realisiert werden sollen, um die o.g. Nahtlosigkeit der Systemintegration zu gewährleisten. In solchen Modellen verblasst also auch die Unterscheidung zwischen technischer und sozialer bzw. "Maschine–Maschine–" und "Mensch–Maschine–Interaktion". In der Praxis gibt es aber gegenwärtig nur einige prototypische Forschungssysteme, die einzelne Teilaspekte des Community Computing demonstrieren, daher ist der Begriff Communityware noch eher im

Sinne einer Metapher als eines technisch fundierten Softwareparadigmas aufzufassen. Jedoch bietet das Gebiet der Agententechnologie, das in Abschnitt 1.3.4 näher beschrieben wird, eine viel versprechende konzeptuelle als auch technische Grundlage zur konkreten Umsetzung dieser Metapher.

Gerade auf der Abstraktionsebene des Communityware–Begriffes spielt der Aspekt der dynamischen Steuerung und Konfiguration eine besonders wichtige Rolle, denn das Zusammenspiel aller Bestandteile einer virtuellen Gemeinschaft kann nur reibungslos funktionieren, wenn es möglich ist — bzw. wenn das System es erlaubt — auf dynamische entstehende Anforderungen flexibel zu reagieren. Insbesondere müssen auch gewisse Regeln spezifiziert und durchgesetzt werden können:

> Every community has rules that can be represented logically. The rules may specify how to elect leaders, make decisions, collect monthly fees and so on. [Ish98, S. 2]

Diese Beschreibung macht auch deutlich, dass solche Regeln auf eine möglichst *generische* Weise modelliert werden müssten, um der Vielfalt der Anwendungssemantik auf dieser Ebene gerecht zu werden. Im Verlauf dieser Arbeit wird u.a. gezeigt, wie eine regelbasierte Steuerung, die derart generischen Anforderungen genügt, realisiert werden kann.

1.3.2 Java

Als wichtigster Sprachbaustein für elektronische Dienstemärkte wird in diesem Abschnitt Java vorgestellt. Ziel dieser Kurzbeschreibung ist es, die Bedeutung dieser Sprache, die zur Implementierung der in der Arbeit beschriebenen prototypischen Systeme verwendet wurde, hervorzuheben.[12]

Außer HTML als der Sprache, mit der das WWW geboren wurde, hat keine andere Sprache das Internet und verteilte Softwaresysteme insgesamt so stark beeinflusst und verändert wie Java. Anders als HTML und ihre verwandten Sprachen wie SGML und XML, mit denen WWW–Seiten, also textuelle Objekte, kommuniziert werden, ist Java jedoch eine vollwertige Programmiersprache, die in erster Linie für die Entwicklung von Internet–fähigen Anwendungen entworfen wurde. Java ist aber noch viel mehr als eine konventionelle Programmiersprache: Zwecks maximaler Unterstützung für die schnelle und komfortable Erstellung jeglicher Anwendungen, die auf dem Internet nur denkbar sind, hat sich um den Sprachkern herum eine ganze Reihe von Subsprachen, Standards, Rahmenwerken und entsprechenden fertig implementierten Softwarekomponenten gebildet.

Java und mobile Objekte Java ist zunächst einmal *objektorientiert* [Mey97] — sie unterstützt alle wesentlichen Eigenschaften der Objektorientierung wie Kapselung, Vererbung, Polymorphismus — und im Vergleich mit allen praxisrelevanten Programmiersprachen ist sie dies mit der weitest gehenden Konsequenz. In Java ist alles ein Objekt, auch Anwendungen, Applets, Programm- und Benutzerschnittstellen sind Objekte. Außerdem erlaubt diese Sprache keine globalen Variablen und Funktionen wie etwa in C++ [Str91]. Insbesondere ist

[12] Relevante Eigenschaften bzw. Details dieser sehr umfangreichen Sprache und ihrer zugehörigen Komponenten werden an den entsprechenden Stellen näher erläutert.

Java eine der ersten Sprachen (und wahrscheinlich die erste praxisrelevante), die integrierte Mechanismen zur einfachen Implementierung von *verteilten* und *mobilen* Objekten bietet. In Verbindung mit der Objektorientierung sind Verteilung und Mobilität die wichtigsten Eigenschaften zur Umwandlung des Internet in ein globales Medium für vollwertige Softwareapplikationen, anstatt nur zum weltweiten Austausch von Hypertextdokumenten, wie es bisher weitgehend der Fall ist. Deshalb hat die (Fort-)Entwicklung der Sprache Java neue Begriffe wie *Object Web* ins Leben gerufen.

Verteilung kennzeichnet die Fähigkeit von Objekten einer Anwendung, Methoden bei Objekten einer anderen Anwendung aufzurufen, wobei die Anwendungen auf unterschiedlichen Rechnern laufen können. In Java wird dies ermöglicht durch den RMI– (*Remote Method Invocation*) Mechanismus, der die Java–Realisation des aus dem Bereich Netzwerkbetriebssysteme und Middleware bekannten RPC– (*Remote Procedure Call*) Konzeptes bezeichnet.

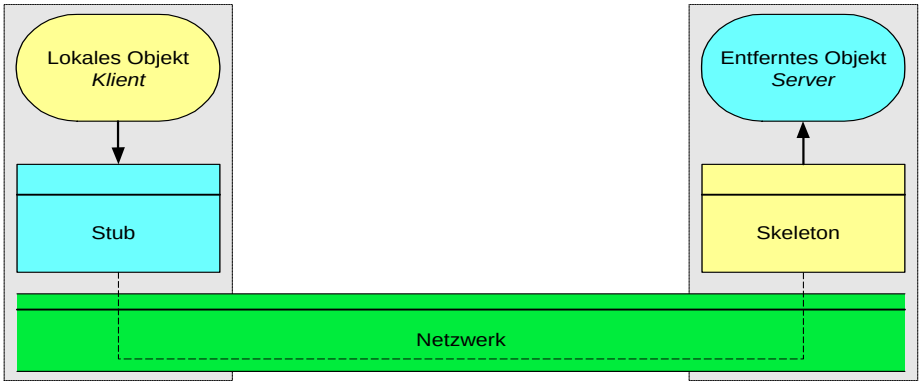


Abbildung 1.4: Struktur eines Remote Procedure Call

Ein solcher RPC–Mechanismus basiert auf der Verwendung von *Stellvertreterobjekten* sowohl auf Seite des aufrufenden wie auch des aufgerufenen Objektes, um maximale Verteilungstransparenz für die Anwendungen zu ermöglichen (siehe Abbildung 1.4). So befindet sich der Stellvertreter des aufgerufenen Objektes — als *Stub* bezeichnet — im Adressraum des aufrufenden Objektes und kann von diesem wie jedes andere lokale Objekt aufgerufen werden. Der Aufruf (samt zugehöriger Parameter) wird allerdings von dem Stub nicht direkt ausgeführt, sondern in das jeweilige Netzwerkformat umgewandelt — mittels des so genannten *Marschalling* — und zu dem Gegenstück des Stubs auf Seite des Aufgerufenen — *Skeleton* genannt — übertragen und von diesem per *Unmarschalling* zu einem für das aufgerufene Objekt lokalen Aufruf wiederhergestellt. Das Ergebnis des Aufrufs wird auf analoge Weise dem Aufrufer vermittelt, so dass die Illusion eines lokalen Aufrufes komplettiert wird. Die beiden einzigen Anforderungen an die Anwendungsobjekte bestehen darin, dass erstens alle (auch) entfernt aufzurufenden Methoden in einer dedizierten Schnittstelle enthalten sein müssen, aus der automatisch die Stellvertreterobjekte generiert werden. Bei RMI ist diese Schnittstelle jedoch inhaltlich und syntaktisch identisch mit einer lokalen Java–Schnittstelle, so dass kein zusätzlicher Aufwand entsteht, wenn eine solche Schnittstelle bereits für die lokale Verwendung des Objektes existiert. Und zweitens muss bei einem Aufruf an ein entferntes Objekt

immer mit zusätzlichen Fehlerfällen — z.B. aufgrund von Netzwerkproblemen — gerechnet werden, RMI verlangt deshalb, dass die Anwendung in der Lage ist, eine so genannte *RemoteException* zu behandeln.

Mobilität kennzeichnet die Fähigkeit von Objekten, komplett mit Implementation und aktuellem Zustand von einer Anwendung zu einer anderen zu migrieren, wobei die Anwendungen wiederum auf unterschiedlichen Rechnern laufen können. Dieses Konzept ist deutlich mächtiger als das der Verteilung: Mobile Objekte können verteilte dadurch ersetzen, dass sie einfach zueinander migrieren und sich gegenseitig Methoden *lokal* aufrufen. Verteilte Objekte allein können mobile allerdings nicht ersetzen, denn bei mobilen können zwischenzeitlich auch die die Objekte hausenden Anwendungen bzw. Rechner offline gehen bzw. heruntergefahren werden, wenn die entsprechenden Objekte gerade zu einem anderen Rechner migriert sind. Wie u.a. auch im Laufe dieser Arbeit deutlich wird, ermöglichen mobile Objekte völlig neue Programmier- und Anwendungsmodelle, deren Potenzial in der Praxis erst gerade ausgelotet wird — wie etwa das der *mobilen Agenten* — andererseits bergen solche Modelle auch neue Sicherheitsrisiken, deren Beherrschung weiterhin Gegenstand der Forschung ist. Java als Sprache enthält zwar keinen direkten, vollwertigen[13] Migrationsmechanismus und somit keine integrierte Objektmobilität. Jedoch bietet Java zwei mächtige Grundmechanismen, mit denen mobile Objekte relativ einfach und in unterschiedlichen Ausprägungen realisiert werden können: Mit der *Objektserialisierung* kann der *Zustand* eines Objektes über ein Netzwerk transportiert werden und mit dem *Class Loading* Mechanismus kann die Implementation eines Objektes je nach Bedarf von Anwendung zu Anwendung übertragen werden.

Über diese für verteilte Anwendungen grundlegende und innovative Eigenschaften hinaus verfügt Java über weitere wichtige, sehr praxisrelevante wie Robustheit, integrierte Sicherheit und leichte Anwendbarkeit. Vor allem ihre unter allen aktuellen Sprachen weitestgehende Plattformunabhängigkeit und ihre weite Verbreitung — die durchaus als Allgegenwärtigkeit bezeichnet werden kann — macht diese Sprache de–facto unverzichtbar für die Realisierung innovativer verteilter Anwendungen, die dennoch konkrete Relevanz für die Praxis besitzen sollen. Bekannte Schwächen von Java wie z.B. Performanz oder mangelnde Typverallgemeinerung spielen bei den für diese Arbeit relevanten Anwendungsklassen keine nennenswerte Rolle. Aus diesem Grund wurde sie als Implementationssprache für alle der Arbeit zu Grunde liegenden prototypischen Systeme und Anwendungsszenarien ausgewählt.

1.3.3 CORBA

Java ist, wie im letzten Abschnitt dargestellt, ein Grundbaustein für die Integration von Objekt- und Internettechnologie und somit zur Schaffung einer globalen Laufzeitumgebung für verteilte Anwendungen. Allerdings ist Java in erster Linie immer noch eine Programmiersprache, der zumindest eine Reihe von *infrastrukturellen Diensten*, die beispielsweise die Findung geeigneter Objekte in einem weltweiten Netzwerk unterstützen, hinzugefügt werden müssen, um eine eine solche globale Plattform in der Praxis zu verwirklichen. Genau hier setzt CORBA (*Common Object Request Broker Architecture*) an, indem sie

[13] *Applets* können zwar als mobile Objekte betrachtet werden, da sie von einem entfernten Webserver in einen Webbrowser geladen und dort lokal ablaufen können. Allerdings ist dies keine vollwertige Objektmobilität, da der Objektzustand nicht berücksichtigt wird.

eine komplette Infrastruktur für verteilte Objekte über Sprach-, Betriebssystem-, Netzwerk- und vor allem Produktgrenzen hinweg anstrebt. Insbesondere die Produkt- bzw. Firmenunabhängigkeit ist ein entscheidender Faktor dafür, dass CORBA inzwischen zum größten und wichtigsten Middleware–Projekt mit über 700 Partnern aus Industrie und Wissenschaft geworden ist.

Technisch gesehen ist der Erfolg von CORBA vor allem auf ihre sehr offene und modulare und damit flexible Konzeption zurückzuführen. Zunächst einmal spezifiziert CORBA für all ihre Bestandteile nur Schnittstellen, aber keine Implementationsaspekte[14], so dass maximaler Freiraum für konkurrierende Produkte — die trotzdem miteinander kompatibel sind — bestehen kann. Zweitens besteht CORBA aus einem sehr schlanken Kern, dem hauptsächlich für die Kommunikation zwischen beliebigen Objekten zuständigen ORB (*Object Request Broker*), der ganz allein existieren bzw. Schritt für Schritt um die höheren Dienste, die insgesamt die Globalinfrastruktur für die Anwendungen bilden, erweitert werden kann. Unterschiedlich komplexe Produkte für unterschiedliche Bedürfnisse können somit auf dem Markt nebeneinander bestehen und die Architektur selbst kann sich einer Evolution unterziehen, um sich den dynamischen Anforderungen der Praxis anzupassen. Abbildung 1.5 zeigt diese modulare Architektur.

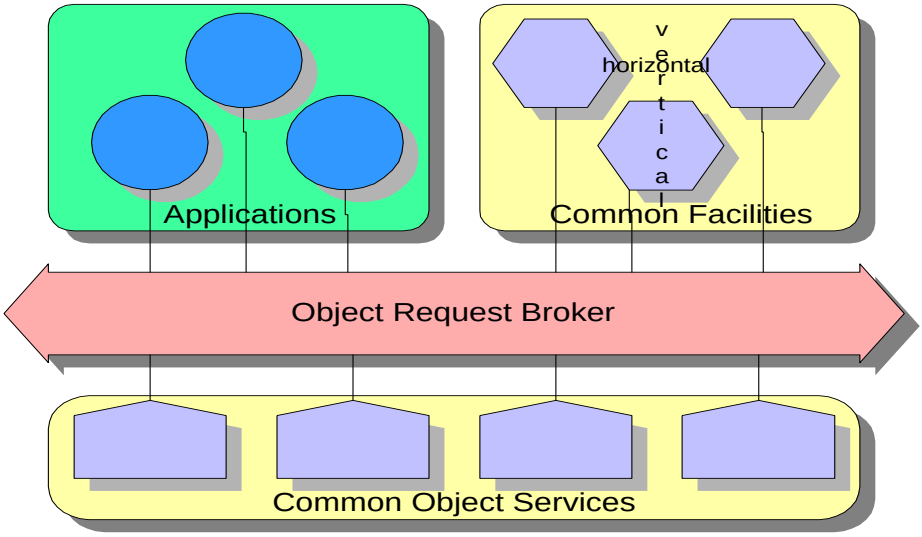


Abbildung 1.5: Modularer Aufbau der *Common Object Request Broker Architecture*

Der ORB als obligatorisches Rückgrat jedes CORBA–Produktes kann hierbei als ein Systemdienst betrachtet werden, der eine flexible, verteilungstransparente Kommunikation zwischen beliebigen Anwendungs- und Systemobjekten ermöglicht. In der Literatur wird der ORB deshalb häufig mit der Metapher eines Buses, der alle Objekte miteinander verbindet, veranschaulicht. Im Vergleich zu einem üblichen RPC, bspw. dem vorgestellten RMI, bietet ein ORB wesentlich mehr Komfort bzw. Kommunikationsvarianten:

[14]CORBA gibt auch keine Referenzimplementation vor wie etwa bei DCE [Sch93a], einer der ersten praxisrelevanten Middlewareplattformen.

- Sprachunabhängigkeit: Einer der größten Vorteile von CORBA — und einer Middlewarearchitektur — ist, dass sie Interoperabilität zwischen Objekten, die in unterschiedlichen Programmiersprachen implementiert sind, ermöglicht. Dies wird dadurch erreicht, dass die Funktionalität von CORBA–Objekten (zusätzlich) in einer neutralen, ausschließlich zur Beschreibung von Schnittstellen dienenden Sprache, der so genannten IDL (*Interface Definition Language*), spezifiziert wird, aus der die o.g. sprachspezifischen *Stub*– und *Skeleton*–Objekte, die in diesem Fall Bestandteil des ORB sind, mittels eines entsprechenden IDL–Compilers generiert werden. Bei einem Methodenaufruf übernimmt der ORB komplett die Transformation der Daten zwischen den unterschiedlichen Formaten. Die Neutralität der IDL hat allerdings zur Folge, dass sie nur Konstrukte enthält, die in allen unterstützten Zielsprachen ausgedrückt werden können.

- Statische und dynamische Methodenbindungen: Der ORB unterstützt nicht nur statische Methodenaufrufe mit Typüberprüfung zur Kompilationszeit, sondern auch dynamische Bindungen mit Typüberprüfung zur Laufzeit. Letztere sind von entscheidender Bedeutung für Anwendungen, die eine beliebige, erst zur Laufzeit entdeckte Funktionalität nutzen wollen.

- Selbstbeschreibende Objekte: Damit ein Anwendungsobjekt (oder ein Benutzer) die Funktionalität eines anderen Objektes zu jedem beliebigen Zeitpunkt entdecken kann, muss es einen allgemein zugänglichen Mechanismus geben, um Metadaten über Objekte zu erhalten. CORBA bietet hierzu das *Interface Repository* an, welches Objektmetadaten in Form von IDL zur Verfügung stellt. Bei der Initialisierung wird jedes CORBA–Objekt bei diesem Dienst mit ihrer Schnittstelle registriert, so dass es auf diese Weise automatisch anderen Objekten bekannt gemacht wird.

- Polymorphe Kommunikation: Der ORB ermöglicht im Gegensatz zum üblichen RPC–Mechanismus nicht nur die Kommunikation zwischen zwei, sondern zwischen beliebig vielen Objekten. Insbesondere kann im Falle der dynamischen Bindung ein Objekt über den ORB bei jedem beliebigen anderen eine bestimmte Methode aufrufen, die auch jeweils zu einem unterschiedlichen Ergebnis führen kann.

- System–Interoperabilität: Seit dem CORBA–Standard 2.0 können auch beliebig viele ORBs untereinander interoperieren, so dass Anwendungsobjekte über ORB–Grenzen hinweg einander aufrufen können, selbst wenn die ORBs von unterschiedlichen Herstellern kommen. Dies wird ermöglicht durch das so genannte *Internet Inter–Orb Protocol* (IIOP), welches die Kommunikation zwischen beliebigen ORBs standardisiert, selbst wenn diese intern proprietäre Protokolle verwenden.

Zu den optionalen Komponenten, die CORBA nach und nach immer mehr zu einer globalen, praxistauglichen Softwarearchitektur werden lassen, gehören *Object Services*, *Facilities* und *Frameworks*. Als Object Services werden all diejenigen Systemdienste bezeichnet, die die Kernfunktionalität des ORB um zusätzliche Eigenschaften erweitern. Über zwanzig Object Services, u.a. *Life Cycle, Event, Transaction, Persistence, Security* und *Trader*, sind bisher standardisiert worden. Im Gegensatz dazu sind die Facilities und Frameworks nicht der System-

und Kommunikations-, sondern direkt der Anwendungsebene zugeordnet. Sie stehen "neben" den Anwendungsobjekten und sollen ihnen gewisse Funktionen als Fertigbausteine bzw. -gerüste bereitstellen. Die Facilities werden weiterhin in *horizontale*, d.h. domänenunabhängige, und *vertikale* oder domänenspezifische unterteilt. Mit Frameworks greift man noch höher in die Anwendungsebene hinein und versucht, maximale Unterstützung für die Entwicklung einer bestimmten Klasse von Anwendungen dadurch zu bieten, dass gewisse Interaktionsmuster sowie fertig implementierte Anwendungsobjekte bereitgestellt werden, aus denen eine Anwendungsinstanz zusammengesetzt bzw. konfiguriert werden kann. Hierin spiegelt sich auch der bereits erwähnte Componentware–Ansatz wider, wonach die eigentliche Anwendungssemantik möglichst nicht mehr vom technischen Personal programmiert werden muss, sondern direkt von den Anwendern bzw. Domänenspezialisten zusammen komponiert wird. Einer der wichtigsten Frameworks, die gegenwärtig von der OMG ins Leben gerufen werden, ist der für *Business Objects* [OMG98a], der eine Reihe von (dynamischen) Mechanismen zur Spezifikation der Semantik von Geschäftsobjekten vorsieht (s. a. [Gri98]).

1.3.4 Das Agenten–Paradigma

Seit Anfang der neunziger Jahre erlebt die Agententechnologie eine durchgreifende und branchenübergreifende Verbreitung. Diese Renaissance einer Jahrzehnte zurückreichenden Idee findet sich nicht zufällig zur gleichen Zeit statt wie das explosionsartige Wachstum des Internets, sondern ist allem Anschein nach genau aufgrund dessen bewirkt worden. Die im Internet immer größer werdende Informationsflut für alle Anwender und Branchen schreit geradezu nach effektiveren und leichter zu bedienenden Verfahren, Werkzeugen und insbesondere Automatismen, um aus der Überschwemmung an Daten den größten Nutzen ziehen zu können. Der bisher so erfolgreiche *interaktive* Umgang mit dem Internet per Browser scheint also nicht mehr auszureichen, zusätzlich sind automatisierte Verfahren von Nöten, die dem Anwender die erforderliche Interaktion abnehmen. Insbesondere Fachleute und Entwickler aus zahlreichen unterschiedlichen Gebieten der gesamten Informationstechnologie wie Künstliche Intelligenz, Verteilte Systeme, Datenbanken, Softwaretechnik, Medizininformatik, Logistik, Netzwerkmanagement und E–Commerce etc. sehen in Softwareagenten ein Mittel, das ihren bisherigen Lösungsansätzen klar überlegen ist und somit deutliche Fortschritte auf dem jeweiligen Gebiet erwarten lässt. Und obwohl die teilweise euphorischen Erwartungen an diese Technologie — oder besser ausgedrückt: Paradigma — insbesondere von seitens der Industrie inzwischen wieder etwas gedämpft worden sind, wird ihre Entwicklung in zahlreichen Projekten fortgesetzt, aus denen nun auch immer mehr praxisrelevante Ergebnisse zustandekommen. In diesem Abschnitt wird die Agententechnologie, die sowohl als ein Hauptmittel zur Modellierung als auch eine Basistechnik zur Implementierung der in dieser Arbeit dargestellten Steuerungsmechanismen verwendet wird, sowie ihre Vor- und Nachteile erörtert.

Trotz, aber auch gerade wegen der großen Popularität der Agententechnologie gibt es keinen Konsens darüber, was ein Softwareagent ist. Es existieren eine Vielzahl von Definitionen eines Agenten, wie beispielsweise von Franklin und Graesser [FG96] zusammengestellt wurde. Den allermeisten, wenn nicht sogar allen, Definitionen sind jedoch zwei Grundeigenschaften gemeinsam, die als die charakteristischsten und wesentlichsten für die Agentenzugehörigkeit betrachtet

werden können:

Autonomie kennzeichnet die Fähigkeit, selbständig und vollständig ohne die Notwendigkeit von Benutzerintervention bestimmte Aufgaben zu lösen. Insbesondere impliziert Autonomie, dass eine Aufgabe komplett an die betreffende Komponente *delegiert* werden kann. In Bezug auf Softwareagenten kann man zwischen zwei Hauptformen der Autonomie unterscheiden: Entscheidungs- bzw. Lokationsautonomie (siehe weiter unten).

Interaktion kennzeichnet die Fähigkeit, Aufgaben zu durchzuführen, die einen Austausch von Information mit der Umgebung erfordern (siehe auch 1.2.2). In Bezug auf Softwareagenten wird Interaktion meist unterteilt in:

- Reaktivität: Der Agent nimmt Veränderungen seiner Umgebung wahr und kann darauf reagieren. Insbesondere ist er in der Lage, die für ihn bestimmten Nachrichten zu empfangen, verarbeiten und beantworten.
- Proaktivität: Der Agent ist *zielorientiert* und versucht, aktiv Einfluss auf die Umgebung auszuüben, um sein Ziel (besser) zu erreichen. Insbesondere ist er in der Lage, Nachrichten zu anderen Agenten zu initiieren, um Veränderungen zu bewirken.

Obwohl Autonomie und Interaktion zwei verschiedene Eigenschaften sind, von denen jede für sich für viele Aufgaben wünschenswert erscheint, macht gerade die *Kombination* dieser beiden Eigenschaften das Charakteristikum von Agenten aus. Nur autonome Anwendungen, wie z.B. viele der klassischen Steuerprogramme für Maschinen oder Kraftwerke, oder nur interaktive Programme, wie z.B. Webbrowser, werden deshalb generell nicht als Agenten bezeichnet.

Aufgrund der traditionell starken Trennung der einzelnen Fachgebiete der Informatik, insbesondere zwischen der KI und den Verteilten Systemen, hat sich die Forschung und Entwicklung der Softwareagenten in zwei Zweige aufgeteilt, nämlich *intelligente* (IAs) und *mobile* Agenten (MAs). Während bei den IAs sich die Autonomie hauptsächlich auf das Entscheidungsvermögen bezieht und als *Entscheidungsautonomie* bezeichnet werden kann, bezieht sich diese im Falle der MAs auf die Lokation des Agenten und wird entsprechend auch als *Lokationsautonomie* betrachtet. Allerdings ist diese, zwar nicht wirklich als beabsichtigt zu bezeichnende, jedoch bisher de–facto existierende Klassifikation außerordentlich ungünstig für die Gesamtentwicklung der Agententechnologie, denn die beiden Qualitäten der Intelligenz und Mobilität sind offensichtlich keineswegs konträr, sondern komplementär zueinander (siehe auch [TGML98]). Während Intelligenz zwar eine sehr wünschenswert erscheinende, jedoch vague und bisher nicht formalisierbare Qualität bleibt, ist Mobilität eine klar definierbare Eigenschaft, wie anhand der Charakterisierung von mobilen Objekten in Abschnitt 1.3.2 deutlich wurde. Wenn man zuerst einmal von dem technisch schwierig zu fassenden Intelligenzbegriffs absieht, was ist aber dann der Unterschied zwischen einem mobilen Objekt und einem mobilen Agenten? Die Klärung dieser Frage wird auch zur Beleuchtung des Agentenparadigmas an sich beitragen.

Mobile Objekte versus mobile Agenten Die kurze Antwort auf die Frage nach dem Verhältnis zwischen mobilen Objekten und Agenten lautet, dass

ersteren die Autonomie — in diesem Fall die Lokationsautonomie — fehlt. Ein mobiles Objekt löst nicht selbst seine Migration aus, es wird von einer externen Komponente migriert. Ein mobiler Agent dagegen ist in der Lage, seine eigene Lokation selbst zu bestimmen und sie gegebenenfalls durch Migration zu ändern. Wenn z.B. — wie in vielen Agentenprodukten — die Migration eines Agenten bzw. Objekten durch das Kommando bzw. die Methode "go(...)" ausgelöst wird, dann besteht der technische Unterschied zwischen einem mobilen Agenten und einem mobilen Objekten einzig darin, dass nur ersterer diese Methode bei sich selbst aufrufen darf. In der Tat wird diese Fähigkeit von vielen als Charakteristikum eines mobilen Agenten gesehen, so z.B. auch bei der Agentenplattform Voyager, einem der weit verbreiteten und technisch bedeutendsten Produkte auf diesem Gebiet ([Obj]).

Wichtiger jedoch als dieser technische Unterschied sind die Aspekte bezüglich der Modellierung. Der erste betrifft die *Granularität* der Modellierungseinheiten: Während ein mobiles Objekt normalerweise Teil einer größeren Anwendung ist, repräsentiert ein Agent meist eine selbständige Anwendung, die dem Anwender eine komplette Funktionalität bereitstellt. Ein Agent kann also aus einem oder mehreren mobilen Objekten bestehen, aber nicht umgekehrt. Zweitens sind Agenten grundsätzlich mit einem bestimmten *Framework* — im Sinne der o.g. CORBA–Frameworks — bzw. einer Agenteninfrastruktur assoziiert. Ein Agentenframework besteht im wesentlichen aus folgenden Bestandteilen:

Agentenprozess: Technische Spezifikation eines Agenten[15].

Platzprozess: Im Gegensatz zu einem Agentenprozess ist dies die Spezifikation einer Lokation, zu der ein Agent migrieren kann. In einem Platzprozess angekommen, kann die Ausführung des Agentenprozesses ab dem Punkt fortgesetzt werden, bevor er auf die Reise ging. Während der Ausführung kann der Agent entweder direkt mit dem Platzprozess — der beispielsweise die Funktionalität eines elektronischen Kaufhauses implementiert — oder mit anderen Agenten interagieren.

Kommunikationssprache: Obwohl mobile Agenten technisch grundsätzlich in der Lage sind, die nötige Interaktion zur Erledigung ihrer Aufgaben per Methodenaufrufe (entfernt oder lokal) abzuwickeln, wird häufig eine höhere Agentenkommunikationssprache (engl. *Agent Communication Language* – ACL) aus folgenden Gründen bevorzugt:

- Eine solche ACL garantiert eine *asynchrone* Kommunikation, die die interne Ausführung des Agentenprozesses nicht blockiert.
- Eine ACL besitzt eine höhere Abstraktion und reichere *Semantik*, die durch spezielle Sprachinterpreter verarbeitet werden kann.
- Schließlich kann eine ACL *standardisiert* werden, so dass technisch völlig verschiedene Agentensysteme auf einer hohen Abstraktionsstufe interoperieren können. Zu den wichtigsten standardisierten ACLs gehören KQML [FFMM94] und FIPA [FIP].

[15] Mit "Prozess" ist hier die Abbildung des jeweiligen Konzepts auf einen Rechnerprozess gemeint.

Werkzeuge und Protokolle: Zusätzlich stellen die meisten Agentenframeworks unterschiedliche Werkzeuge für eine benutzernahe Agentenerstellung und -anwendung zur Verfügung, wie etwa Skriptsprachen oder visuelle Entwicklungsumgebungen zur Spezifikation von Agentenrouten und -aufgaben oder Fertigagenten für bestimmte Rollen/Aufgaben, beispielsweise um eine Reise zu buchen. Darüberhinaus kann ein Agentenframework verschiedene Protokolle festlegen, um Steuerungsmöglichkeiten bzgl. Agentenabläufe zu realisieren. Beispielsweise bietet das Aglets–System von IBM ein mehrschichtiges Protokoll an, um unterschiedliche Sicherheitsanforderungen bzgl. des Schutzes von Platz- vor Agentenprozessen erfüllen zu können [LO98].

Vor- und Nachteile des Agentenparadigmas Die Vorteile von Agenten liegen in dem Sinne auf der Hand, dass sie direkt auf der dargestellten Kombination von Autonomie und Interaktionsfähigkeit basieren. Insbesondere sind Agenten prinzipiell dazu geeignet, eine Vielzahl von Geschäftsrollen, die konventionell von Menschen ausgeübt werden, abzubilden und die damit zusammenhängenden Aktivitäten zu automatisieren. Ein Reiseagent kann beispielsweise im Namen eines Konferenzteilnehmers Anmeldungen und Buchungen für Konferenz, Flug, Hotel, Auto und Rundfahrten durchführen, und das, wenn erforderlich, auf eine transaktional abgesicherte Weise [CGH+95]. Softwareagenten sind besonders für Aufgaben sinnvoll, die das Sammeln und Auswerten einer großen Menge von Information erfordern oder eine große kombinatorische Komplexität besitzen. Allerdings soll hervorgehoben werden, dass Eigenschaften wie Autonomie und Interaktionsvermögen *für sich* und auch deren Verknüpfung noch keine Vorteile implizieren, denn bei Abwesenheit eines gewissen Maßes an intelligentem Verhalten kann gerade diese charakteristische Eigenschaftskombination in ein unberechenbares Risiko für den Anwender umschlagen. Da Intelligenz aber eine (bisher) nicht formalisierbare und zumindest schwierig nachzuweisende Qualität ist, sind auch viele der den Agenten zugeschriebenen Vorteile, wie z.B. Flexibilität, Adaptivität und sogar soziales Verhalten, besser als mehr oder weniger begründete Erwartungen zu betrachten.

Konkretere Vorteile von Softwareagenten kann man im Zusammenhang mit der Mobilität vorfinden. Ein mobiler Agent ist bzgl. der Ausführung eindeutig flexibler als ein konventionelles Programm, dieser Vorteil kann z.B. von großem Nutzen für Benutzer mobiler Geräte sein, denn der Agent, einmal ins Festnetz abgeschickt, kann sich dort den jeweils besten Platz zur Erledigung seiner Aufgabe suchen, während der Auftraggeber sein Gerät wieder ausschalten kann. Auch in Fällen, in denen eine bestimmte Operation auf einem (oder mehreren) entfernten Server sehr häufig ausgeführt werden muss, ist es von Vorteil, sie nicht als entfernte Operation auszuführen, sondern einen mobilen Agenten zu dem Server zu schicken, der die Aufrufe dort lokal durchführt, die Einzelergebnisse vor Ort verarbeitet und schließlich das Gesamtergebnis zurückbringt. Dadurch kann die Netzwerkbelastung sowie die gesamte Verarbeitungszeit reduziert werden. Allerdings ist dieser potentielle Performanzgewinn immer abhängig von dem Verhältnis zwischen zu übertragenden Größe (engl. footprint) des mobilen Agenten und Gesamtmenge der durch die entfernten Aufrufe anfallenden Daten.

Zu den Nachteilen von Softwareagenten gehören in erster Linie Sicherheitsprobleme. Insbesondere müssen die Flexibilität und Dynamik von mobilen Agen-

ten durch neue Sicherheitslücken, die im allgemeinen auch für alle mobile Softwareobjekte gelten, erkauft werden. Es gibt deshalb zahlreiche Arbeiten, die sich sowohl mit dem Schutz der die Agenten beherbergenden Rechner/Applikationen als auch mit dem Schutz der Agenten vor anderen Agenten oder fremden Ausführungsumgebungen beschäftigen. Insbesondere ist der Schutz von Agenten vor böswilligen Rechnern, auf denen der Agentencode letztendlich als ein Programm ausgeführt werden muss, nur durch den teuren Einsatz unverfälschbarer Hardware wirklich zu garantieren. Ein anderer möglicher Nachteil von Agenten als Softwareparadigma ist, dass es trotz der erwähnten intuitiven Entsprechung von Agenten zu bestimmten natürlichen Geschäftsrollen bisher kein ausgereiftes technisches Programmiermodell gibt, das die Effizienz agentenbasierter Softwareentwicklung erhöht. Oft kostet es in der Praxis noch sehr großen Aufwand und Gewöhnungsvermögen, auf konventionellen Geschäftsmodellen (z.B. relationalen Datenbanken) basierte Software in agentenbasierte zu übertragen [KJ98, S. 193 f.].

Ein weiteres grundsätzliches Manko des Agentenparadigmas, das oben bereits erwähnt wurde, ist die Nachweisbarkeit der implizit unterstellten Annahme eines gewissen intelligenten Verhaltens in bezug auf die an die Agenten delegierten Aufgaben, ohne das die grundlegenden Eigenschaften der Autonomie und Interaktion keine Vor-, sondern vielmehr Nachteile bedeuten würden. In Bezug auf dieses Manko bietet die vorliegende Arbeit zwei Lösungsansätze: Erstens werden regelbasierte Steuerungsmechanismen realisiert, um das semantische Risiko von Agentenanwendungen zu minimieren, ohne dabei ihre Funktionalität einzuschränken. Zweitens wird ein dynamisches Framework zur flexiblen Integration vorhandener Algorithmen, die gewisse intelligente Fähigkeiten implementieren, in mobile Agenten konzipiert, welches zur Realisation verhandlungsfähiger Agenten eingesetzt wird.

1.4 Zielsetzung der Arbeit

In den vorangegangenen Abschnitten wurde der vielfältige Hintergrund der vorliegenden Arbeit beleuchtet und insbesondere die dynamischen, interaktiven Aspekte offener verteilter Anwendungssysteme, deren Steuerung und Erweiterung diese Arbeit insgesamt gewidmet ist, hervorgehoben. Die hierin beschriebenen Steuerungs- und Verhandlungsmechanismen sind in einem aktuellen Spannungsfeld zwischen neuen konzeptuellen Modellen, Architekturen und Standards auf der einen Seite und zahlreichen konkreten technischen Innovationen auf der anderen Seite, das die Entwicklung des Gebiets Verteilte Systeme im letzten Jahrzehnt bestimmt hat, entstanden.

Die Zielsetzung dieser Arbeit ist deshalb geprägt von dem zweifachen Anspruch, auf der einen Seite innovative *konzeptionelle Ansätze* zur Weiterentwicklung verteilter Anwendungsarchitekturen zu erarbeiten, auf der anderen Seite aber auch *konkrete Systemmechanismen*, die auf praxisrelevanten Softwareprodukten basieren und diese erweitern, prototypisch zu implementieren. Dabei spielt bei beiden Teilaspekten die *Integration* mit existierenden Technologien eine wichtige Rolle. Diese integrative Methodik erscheint angesichts der Komplexität heutiger Anwendungssysteme und ihrer inhärenten technischen Querbeziehungen untereinander als geradezu unumgänglich.

1.4.1 Anforderungen und Ziele

Ziel dieser Arbeit ist es, generische Mechanismen und Rahmenwerke zur Steuerung offener verteilter Anwendungssysteme zu erarbeiten und prototypisch zu implementieren. Daneben soll durch konkrete Beispiele und Anwendungsszenarien gezeigt werden, dass diese Mechanismen im Umfeld elektronischer Dienstemärkte eingesetzt werden können.

Bei der Umsetzung dieses Gesamtziels soll insbesondere der o.g. interaktive bzw. *interaktionsorientierte* Charakter offener verteilter Anwendungen berücksichtigt werden. Das heißt, die im Rahmen der Arbeit zu realisierenden Mechanismen sollen insbesondere dazu dienen, das Interaktionsverhalten zwischen verteilten Anwendungskomponenten zu steuern bzw. neue, steuerbare Interaktionsformen zu ermöglichen. Um dieses Ziel in konkrete Systemmechanismen umzusetzen, verfolgt diese Arbeit zwei Hauptansätze, die auch als ihre Teilziele verstanden werden können:

- Realisierung *regelbasierter Steuerungsmechanismen*: Zunächst einmal soll untersucht werden, inwieweit der Einsatz von benutzerdefinierten formalen Regeln als dynamisches Konfigurationsmittel und als Kooperationsunterstützung zur Steuerung offener verteilter Anwendungen beitragen kann. Insbesondere ist ein *generisches Regelformat* zu finden, dessen logische Mächtigkeit für eine möglichst große Anzahl von Anwendungsklassen ausreicht, dessen Komplexität jedoch algorithmisch und auch in der Praxis handhabbar bleibt. Anschließend soll gezeigt werden, mit welcher Architektur (bzw. mit welchen Architekturen) die *konkrete Integration* von Regelkonzepten in bestehende Umgebungen für verteilte Anwendungen sich realisieren lässt, wobei die spezifischen *Anforderungen* (siehe unten) dieser Umgebungen zu berücksichtigen sind.

- Realisierung *automatisierter Verhandlungsmechanismen*: Über die regelbasierten Steuerungsmechanismen — deren Mächtigkeit und Funktionalität im wesentlichen auf *logischen Funktionen* basiert — hinaus sollen automatisierte Verhandlungsmechanismen, die auf prinzipiell beliebigen *Protokollen* und *Strategien* basieren können, als eine erweiterte, zu den Regelkonzepten komplementäre Form der Kooperationsunterstützung untersucht werden. Im Kontext *autonomer interaktiver* verteilter Anwendungen, wie sie z.B. in der E–Commerce–Domäne zu finden sind, besitzen derartige Verhandlungsmechanismen, die eines der relevantesten Interaktionsmuster verkörpern, ein offensichtliches Anwendungspotenzial, das durch die Realisierung eines konkreten Verhandlungssystems ausgelotet werden soll. Insbesondere ist dabei ein entsprechendes *Framework* zu konzipieren, das eine möglichst große Offenheit im Hinblick auf die einsetzbaren Verhandlungsprotokolle und -strategien ermöglicht.

Aus der dargestellten Vielfältigkeit des dieser Arbeit zu Grunde liegenden Umfelds der offenen verteilten Systeme lassen sich zusammenfassend folgende Anforderungen an die angestrebten Steuerungsmechanismen ableiten, die die konkrete Umsetzung der beiden Teilziele wesentlich beeinflusst haben.

Generizität Die Vielfalt, Komplexität und Heterogenität heutiger Softwaresysteme impliziert zugleich auch ihre Schnelllebigkeit. Die alltägliche Erfahrung

lehrt uns, dass ein Softwareprodukt meistens bereits veraltet ist, wenn man sich gerade an es gewöhnt hat. Andererseits ist es auch sehr häufig festzustellen, dass gewisse Ideen und Konzepte immer wieder in Form neuer Produkte "wiedererfunden" werden, weil sie in einer Vielzahl von Anwendungssituationen entweder unverzichtbar oder wesentlich zur Anwendbarkeit der Software beitragen[16]. Deshalb ist es vor allem für Forschungsprojekte offensichtlich von höchster Bedeutung, generische Konzepte zu finden, die möglichst unabhängig sind von einer speziellen Anwendungssemantik. Dies bedeutet aber nicht, dass derartige Konzepte nicht praxisnah sein können (siehe unten).

Dynamik Steuerungsmechanismen sind offenbar umso notwendiger, je dynamischer die zu steuernden Systeme sind. In den meisten Fällen ist es aber auch vorteilhaft, wenn die Steuerungsmechanismen selbst dynamisch (einsetzbar) sind, um erstens direkt und bruchlos greifen zu können und zweitens jederzeit (gegen andere, die für die *aktuellen* Anforderungen besser geeignet sind) austauschbar sind. Im erweiterten Sinne bedeutet Dynamik auch, dass die Steuerungsmechanismen sich selbst der veränderten Umgebung anpassen können — also *selbst–adaptierend* sind — beispielsweise durch die Anwendung genetischer Algorithmen. Je nach den spezifischen Anforderungen der zu steuernden Systeme sollen all diese Aspekte der Dynamik von Steuerungsmechanismen auf ihre konkrete Wirkung untersucht werden.

Offenheit und Transparenz Zusätzlich zu der o.g. Generizität der Konzepte, die als *anwendungssemantische* Offenheit bezeichnet werden kann, sollen Steuerungsmechanismen i.a. auch *technisch* offen, d.h. unabhängig von bestimmten Hardware- und Softwarearchitekturen, Betriebssystemen, Programmiersprachen und Kommunikationsprotokollen und -sprachen sein. Allerdings ist die technische Offenheit einer *konkreten Realisierung* von Konzepten, wie sie durch die vorliegende Arbeit maßgeblich verfolgt wird, immer eine *relative* Anforderung. In der Tat lässt sich durch die Anwendung geeigneter Entwurfsprinzipien wie Modularität, Dezentralisierung und Verteilung und vor allem möglichst plattformunabhängiger Sprachen und Standards wie Java und CORBA ein hohes Maß an technischer Offenheit erzielen, jedoch ist aufgrund der Heterogenität beispielsweise zwischen verschiedenen Middlewarearchitekturen eine "hundertprozentige" Interoperabilität niemals gegeben. Eng verbunden mit der Offenheit und der Dynamik ist die Anforderung nach Transparenz, die im Kontext der Arbeit bedeutet, dass die zu steuernden Anwendungssysteme möglichst in ihrer ursprünglichen Form der Funktionalität, des Entwurfs und der Implementation erhalten bleiben können, d.h. die Steuerungsmechanismen bringen zwar *zusätzliche* Anwendungssemantik hinein, sollen aber grundsätzlich keine funktionale Veränderung der Zielsysteme nötig machen.

Praxisbezogenheit Ein wesentlicher Anspruch und Leitfaden, der die vorliegende Arbeit begleitet und ihre Ergebnisse maßgeblich beeinflusst hat, besteht darin, sowohl die allgemeine Problemstellung der Steuerung offener verteilter

[16] Ein Beispiel hierfür ist das Transaktionsmanagement: Viele der neueren Plattformen bzw. Standards für die Entwicklung verteilter Anwendungen, u.a. CORBA und JINI, die (bzw. deren Produkte) zunächst ohne dieses alte und altbewährte Konzept ins Leben gerufen wurden, haben es nachträglich eingebaut bekommen.

Anwendungssysteme als auch die entsprechenden Lösungsansätze an der Praxis zu orientieren. Dies bedeutet im einzelnen:

- Berücksichtigung der praktischen Rahmenbedingungen: Gewisse Technologien und Standards, die sich in der Praxis etabliert haben bzw. eine allgemein anerkannte praktische Bedeutung haben, sollen als konkrete Rahmenbedingungen bei der Erarbeitung der Lösungsansätze berücksichtigt werden. D.h. auch, dass dabei möglichst keine theoretischen Modelle bzw. Annahmen aufgestellt werden, um ideale Lösungen zu ermöglichen, die jedoch in der realen Praxis keinen Gebrauch finden würden.
- Beobachtbare Praxisreife: Die vorgeschlagenen Konzepte, Rahmenwerke und Algorithmen sollen bis zu einer konkreten beobachtbaren Praxisreife, die durch prototypisch implementierte Anwendungsszenarien zu demonstrieren ist, ausgearbeitet werden. Dies impliziert u.a., dass eventuell bei der Ausarbeitung der Konzepte an gewissen Stellen "Tiefe anstatt Breite" vorzuziehen ist, um die grundsätzliche Anwendbarkeit der Lösungsansätze nachzuweisen.

1.4.2 Überblick

Im weiteren Verlauf dieser Arbeit wird wie folgt vorgegangen:

In Kapitel 2 wird die theoretische und konzeptionelle Basis für eine regelbasierte Steuerung offener verteilter Anwendungen gelegt. Dazu wird zunächst präzisiert, was im Kontext dieser Arbeit unter "regelbasierte Steuerung" verstanden wird. Um eine generische, d.h. anwendungsunabhängige Regelverarbeitung zu ermöglichen, sind gewisse logische und mathematische Grundlagen notwendig, die anschließend eingeführt werden. Es folgt dann die theoretische Konzeption der in dieser Arbeit verwendeten Regelkonzepte und der dazu gehörigen Verarbeitungsfunktionen einschließlich der auf den formalen Grundlagen basierten Algorithmen.

Kapitel 3 behandelt die Frage, mit welcher Systemarchitektur die zunächst theoretisch konzipierten Regelkonzepte für das heterogene Gebiet der verteilten Anwendungssysteme nutzbar gemacht werden können. Dazu werden sowohl eine zentrale als auch eine dezentrale Architektur zur Integration der Regelkonzepte in verteilte Systemumgebungen präsentiert. Hierbei wird insbesondere der Frage nachgegangen, wie verteilte Anwendungen in ihrem Interaktionsverhalten durch Regelverarbeitungsmechanismen unterstützt werden können.

In Kapitel 4 geht es um verschiedene Aspekte der Umsetzung der architekturellen Konzepte in konkrete Softwarekomponenten zur Steuerung verteilter Anwendungssysteme. Neben der Implementierung der Architekturen in Form verschiedener Klassenbibliotheken wird auch die Realisierung einer Reihe von Anwendungsszenarien beschrieben, die demonstrieren, wie die Regelmechanismen zur Steuerung unterschiedlicher Anwendungsklassen eingesetzt werden können.

Mit Kapitel 5 beginnt dann die Behandlung von automatisierten Verhandlungen als einer funktionalen Erweiterung der regelbasierten Steuerungsmechanismen. Um eine generische, allgemein verwendbare Konzeption von Verhandlungsmechanismen zu ermöglichen, wird zunächst eine eingehende Analyse automatisierter Verhandlungsformen vorgenommen. Daraus wird dann eine semantisch offene, modulare Architektur selbständig verhandelnder Agenten konzipiert.

Kapitel 6 beschreibt dann den Entwurf und die Realisierung eines konkreten, funktional vollständigen Verhandlungssystems für mobile Softwareagenten. Dazu gehört neben der Bereitstellung geeigneter Agentenmechanismen vor allem die Realisierung konkreter Verhandlungsprotokolle und -strategien. Außerdem wird in diesem Kapitel aufgezeigt, wie automatische Verhandlungen ihrerseits von den im ersten Teil präsentierten Regelmechanismen gesteuert werden können.

Das letzte Kapitel 7 fasst schließlich die Ergebnisse der vorliegenden Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen, die in weiterführenden Arbeiten durchgeführt werden könnten.

Kapitel 2

Modellierung regelbasierter Steuerungsmechanismen

In diesem Kapitel werden wir uns mit Fragen der Modellierung und der Formalisierung von Steuerungsmechanismen für verteilte Anwendungen beschäftigen. Dazu ist es zunächst notwendig zu klären, was hier unter Steuerung verstanden wird. Um den formalen Hintergrund der in diesem Teil dargestellten Regelmechanismen zu beleuchten und um die formale Basis für die später folgenden Algorithmen zur Verarbeitung der Regeln zu legen, werden dann die logischen und mathematischen Grundlagen dieser Mechanismen behandelt. Abschließend wird der in dieser Arbeit vertretene Steuerungsansatz mittels einer generischen Regelverarbeitung, die auf einer speziellen Form der Prädikatenlogik erster Stufe basiert, erläutert.

2.1 Steuerung verteilter Anwendungssysteme

Mit der wachsenden Dynamik, die aus der Offenheit, Interaktion und Heterogenität verteilter Anwendungen in globalen Umgebungen resultiert (siehe Abschnitt 1.2), vergrößert sich offensichtlich auch der Bedarf an geeigneten Steuerungsmechanismen, um diese Dynamik zu beherrschen bzw. zunutze machen zu können. Ein einfaches, allgemein bekanntes Beispiel hierfür ist das Problem, eine bestimmte Information im WWW zu finden bzw. wiederzufinden, ohne gleich von der Datenflut überschwemmt zu werden. Allerdings ist es auf den ersten Blick nicht ersichtlich, was genau unter Steuerung zu verstehen ist und wie die entsprechenden Mechanismen aussehen sollen. Es ist z.B. sicherlich nicht sinnvoll und praktikabel, ein zentrales Management- oder Steuerungssystem einzuführen, das die Spielregel, beispielsweise in bezug auf Menge, Art und Lokation der veröffentlichten Informationen, für alle Komponenten in einer Umgebung wie dem WWW diktiert, obwohl dies offenbar das genannte Problem der Informationssuche auf eine einfache Weise erledigen würde.

Die wichtigste Annahme für die im folgenden diskutierten Steuerungsaspekte ist also, dass ein *zentralistischer* Ansatz im Sinne einer globalen Autorität, die verbindliche Richtlinien für alle beteiligten Komponenten des Systems festlegt, ausgeschlossen ist, weil ein solcher Ansatz dem Grundgedanken der *offenen* verteilten Anwendungssysteme widersprechen würde. Diese Grundannahme im-

pliziert auch, dass ein geeigneter Steuerungsansatz *punktuell* greifen muss, d.h. dass der Steuerungseffekt an einzelnen Komponenten des Gesamtsystems in der Regel unterschiedlich ausfällt. Anzustreben ist also eine individuell einstellbare Steuerung, die den *autonomen* Charakter der Anwendungskomponenten unterstreicht. Wenn wir nun den in Abschnitt 1.1 eingeführten Dienstbegriff aufgreifen, der eine einheitliche Abstraktion aller in einem Anwendungssystems beteiligten Einheiten bietet, kann Steuerung zunächst als *Anpassung der beteiligten Dienste* verstanden werden.

Um eine konkrete Realisation zu ermöglichen, bedarf diese Charakterisierung von Steuerung in zweierlei Hinsicht der Präzisierung, nämlich *was* an den Diensten angepasst werden kann und *wie* dies geschehen soll. Um die erste Frage zu beantworten, werden nun verschiedene Dienstaspekte betrachtet.

2.1.1 Dienstaspekte

Der Dienstbegriff als gemeinsame Abstraktion jeglicher Komponenten, die an einem verteilten Anwendungssystem beteiligt sein können, kann unter verschiedenen Aspekten betrachtet werden, die jeweils unterschiedlich formal beschreibbar und technisch beeinflussbar sind. Im Zusammenhang mit der Fragestellung der Steuerung bzw. Anpassung von Diensten sind folgende Aspekte zu unterscheiden:

Funktion Von jedem Dienst wird erwartet, dass er eine bestimmte Funktion erfüllt. Die Gesamtfunktion eines Dienstes umfasst i.a. eine Menge von *Teilfunktionen*, die in der Regel dadurch erfüllt werden, dass aus gewissen *Eingaben* entsprechende *Ausgaben* bzw. Resultate erzeugt werden. *Insofern* ähnelt die Funktion eines Dienstes einer (bzw. einer Menge von) *mathematischen* Funktion. Allerdings unterscheidet sich die Dienstfunktion in zwei wesentlichen Hinsichten von einer mathematischen: Erstens spielt der Zeitfaktor eine entscheidende Rolle für die (Anwendbarkeit einer) Dienstfunktion und zweitens ist die Funktion eines Dienstes i.a. von ihrem aktuellen Zustand (nächster Dienstaspekt) abhängig, d.h. das Resultat einer Dienstfunktion ist nicht wie bei einer mathematischen durch die Eingabe determiniert. Darüberhinaus kann dieses Resultat aufgrund der *Interaktion* zwischen den Diensten *während des zeitbehafteten Berechnungsprozesses* beeinflusst werden, z.B. kann die Interaktion eine Veränderung des Zustandes bewirken. Wie in Abschnitt 1.2.2 dargestellt wurde, führt eine solche Interaktion zu einer Komplexität, die nicht mehr durch mathematische Funktionen beschreibbar ist.

Zustand Wie bereits an der Funktion erwähnt, besitzt ein Dienst zu jedem Zeitpunkt einen Zustand, der das Resultat der Funktion beeinflussen kann. Worin besteht aber dieser Zustand? Um diese Frage zu beantworten, muss man zwei Varianten des Dienstzustandes unterscheiden. Zunächst kann der *implizite* Zustand eines Dienstes als die Menge der Werte aller zugehörigen (privat als auch öffentlich zugänglichen) Variablen zu einem Zeitpunkt definiert werden. Diese einfache und präzise Definition ist jedoch nicht sehr nützlich[1], da sie zu einer maximalen Anzahl von Zuständen führt, von denen vielleicht nur wenige einen unterscheidbaren Einfluss auf die Funktion

[1] Außer als Hilfsmittel für das Testen bzw. *Debugging* während der Entwicklung solcher Komponenten

(d.h. zu unterschiedlichen Resultaten führen können) bzw. eine anwendungssemantische Relevanz besitzen. Daher wird meistens eine *explizite* Modellierung des Zustandes bevorzugt, bei der eine Menge von Variablen ausgewählt bzw. neu eingeführt wird, deren Werte *relevante* Zustände des Dienstes repräsentieren. Da die Kenntnis des Zustandes in vielen Fällen wichtig für die Verwendung des Dienstes ist, werden die entsprechenden Variablen oder auch Attribute im programmiertechnischen Sinne in der Regel *extern zugänglich* gemacht, wobei sie dann meist als Eigenschaften (engl. *Properties*) bezeichnet werden. Der externe Zugriff auf diese Eigenschaften und damit auf den Zustand des Dienstes kann je nach Anforderung auf unterschiedliche Grade — z.B. die ausschließliche Lesbarkeit — begrenzt werden und entweder *direkt* als Attributswert oder *indirekt* über Zugriffsmethoden erfolgen. Der indirekte Zugriff auf die Diensteigenschaften ermöglicht eine einheitliche Umgangsform mit dem Dienst — nämlich ausschließlich über Methodenaufrufe — die als *operationale* Sicht [ODP94a, MJ97] bezeichnet wird.[2]

Struktur Die Struktur eines Dienstes ist die Art und Weise, wie er sich aus einzelnen Bestandteilen zusammensetzt. Dabei kann zwischen der *internen* und der *externen* Struktur unterschieden werden. Die interne Struktur betrifft die Abhängigkeiten des Dienstes von anderen Diensten, z.B. kann ein Dienst seine Funktion dadurch erfüllen, dass er alle Methodenaufrufe an einen oder mehrere weitere Dienste delegiert wie im Falle eines *Dispatcher*. Die interne Struktur ist in der Regel aufgrund des abstrahierenden Charakters der Dienstsicht gekapselt und somit verborgen vor den Entitäten, die den Dienst in Anspruch nimmt. Die externe Struktur betrifft dagegen die Art und Weise, wie der Dienst anderen Diensten oder Entitäten zur Verfügung steht, z.B. wie viele Methoden extern aufgerufen werden können, welche Namen und Parameter sie besitzen und wie diese Methoden in Schnittstellen gruppiert werden etc. Allerdings ist die Struktur als Abstraktion zu unterscheiden von ihrer Beschreibung, die unterschiedliche Formen und Detaillierungsgrade besitzen kann (s.a. Abschnitt 2.1.2).

Verhalten Von der Struktur und auch von der Funktion eines Dienstes ist ihr Verhalten zu unterscheiden. Beispielsweise können zwei Zahlungsdienste, die funktional gleichwertig zur Abwicklung einer Handelstransaktion genutzt werden, in ihrem Verhalten deutlich voneinander abweichen, da sie völlig unterschiedliche Protokolle implementieren [MTL96]. In diesem Sinne kann das Verhalten eines Dienstes auch als seine (beobachtbare) *Funktionsweise* bezeichnet werden (siehe z.B. [Gri98]), da letzterer Begriff die Beziehung zwischen Funktion und Verhalten deutlicher werden lässt. Das Verhalten eines Dienstes kann sich in vielfältigen Formen auf seine *Verwendbarkeit* und damit auch auf seine Qualität auswirken: z.B. durch die zulässigen Abfolgen von Methodenaufrufen — auch als *Dienstzugangs-* bzw. *Schnittstellenprotokoll* bezeichnet (siehe Abschnitt 2.1.2), durch die Zeitdauer für die Bearbeitung von Aufrufen (Performanz), durch die zu erfüllenden (Vor-)Bedingungen zur (einwandfreien) Gewährleistung der Funktion etc. Neben der eigentlichen Funktion ist das Verhalten eines

[2] Dies impliziert, dass die operationale Sicht keineswegs den Zugriff auf Diensteigenschaften untersagt bzw. verhindert, sondern lediglich eine bestimmte Form des Zugriffes vorschreibt.

Dienstes also der wichtigste, in der Praxis meistens ausschlaggebende Aspekt für ihre Benutzung.

Semantik Verantwortlich für die Erfüllung der Funktion und für das Verhalten eines Dienstes ist die Semantik seiner Implementation, also der tatsächliche Inhalt, der mit seiner Struktur kodiert ist. Leider ist die Formalisierung und Darstellung der Semantik von Software ein immer noch ungelöstes Problem (s.a. [Gri98, S. 71]). Um die vollständige Semantik einer Softwareentität herauszufinden, müsste man sich in der Regel den gesamten Quellcode ansehen, was — ganz abgesehen von der damit verbundenen Komplexität — schon deswegen unpraktikabel ist, weil Programmcode in den meisten Fällen nicht freigegeben wird. Da das Verständnis der Semantik, die letztendlich auch den (jeweiligen) Zustand und das Verhalten eines Dienstes bestimmt, für seine korrekte Verwendung — insbesondere eine automatisierte Verwendung durch andere Softwarekomponenten — und auch für seine Akzeptanz aber häufig unerlässlich ist, versucht man, wie unten in Abschnitt 2.1.2 erläutert, mit unterschiedlichen Ansätzen der (partiellen) formalen Repräsentation Abhilfe für das Problem der unterspezifizierten Semantik zu schaffen.

2.1.2 Dienstmodelle

Für die korrekte und effiziente Nutzung eines Dienstes bzw. um *irgendeine* Nutzung erst möglich zu machen, ist es erforderlich, gewisse Aspekte des Dienstes, die im letzten Abschnitt geschildert sind, nach außen bekannt zu machen, d.h. diese in einer geeigneten Form zu *beschreiben*. Für eine automatisierte Nutzung bzw. um die Interaktion zwischen möglich vielen verschiedenen Diensten zu ermöglichen, ist es darüber hinaus notwendig, die Beschreibung von Dienstaspekten in einer formalen und "standardisierten" Weise vorzunehmen. Dazu existieren unterschiedliche Ansätze, die als *Dienstmodelle* oder genauer ausgedrückt *Dienstbeschreibungsmodelle* bezeichnet werden können.[3] Solche Dienstmodelle können i.a. in so genannte strukturelle und semantische Modelle klassifiziert werden, die im folgenden näher erläutert werden. Im Kontext in dieser Arbeit werden die Begriffe "Dienst" und "Protokoll" jedoch ausschließlich in Bezug auf die Anwendungsebene verwendet, d.h. es steht das Verhältnis zwischen Dienstnutzer und Dienstanbieter im Vordergrund. Beispielsweise werden Protokolle hier lediglich im Sinne von *Dienstzugangs*protokollen betrachtet, und nicht als Regeln für die Interaktion zwischen verteilten Komponenten *eines* (Kommunikations-)Dienstes im OSI/ISO–Modell [Hal88]. Letzterer Protokollbegriff wird ausführlich im Bereich *Formal Description Techniques* (FDT) behandelt (siehe z.B. [Hog89]).

2.1.2.1 Strukturelle Modelle

Strukturelle Modelle zielen darauf ab, den o.g. Strukturaspekt eines Dienstes zu beschreiben. Jedoch beschränken sich solche Modelle ausschließlich auf die *externe* Struktur, um die mit der Dienstsicht verbundene *Blackbox*–Sicht zu betonen.

[3] Hinter einem Dienstbeschreibungsmodell steckt auch immer ein abstraktes Dienstmodell. Da beide jedoch sehr eng miteinander zusammenhängen, erscheint es sinnvoll, ihnen eine gemeinsame Darstellung zu widmen. Solange keine Mehrdeutigkeiten vorhanden sind, wird hier die Kurzform bevorzugt.

Dadurch werden beispielsweise Abhängigkeiten des beschriebenen Dienstes von anderen vor den Nutzern verborgen. Ein strukturelles Modell beschreibt also, über welche *Informationsstrukturen* der Dienst genutzt bzw. mit ihm interagiert werden kann. Derartige Modelle können jedoch große Unterschiede in bezug auf Form und Detaillierungsgrad aufweisen.

Im einfachsten Fall kann ein strukturelles Modell beschränkt sein auf die Benennung des Dienstes. Um irrtümliche Referenzierungen des Dienstes zu vermeiden, wird in der Regel die Anforderung nach der globalen *Eindeutigkeit* des Dienstnamens bzw. -bezeichners erhoben, die durch Techniken der Generierung so genannter UUIDs (*Universal Unique IDentifier*) gewährleistet wird [Cat94]. Der entscheidende Nachteil eines solchen Bezeichners — meistens eine durch mathematische Kodierungsverfahren generierte Zeichenkette — besteht darin, dass er überhaupt nichts über die intendierte Funktion des Dienstes aussagt. Dem gegenüber steht die Verwendung symbolischer Namen wie beispielsweise "ReiseAgent" als Alternative, die i.a. jedoch keine Eindeutigkeit besitzen. Eine Kombination beider Verfahren, d.h. eine Ergänzung des UUIDs um einen symbolischen Teil könnte zwar etwas Abhilfe schaffen, würde jedoch die grundsätzlich beschränkte Ausdrucksmächtigkeit von (unstrukturierten) Zeichenketten als Bezeichner auch nicht erhöhen können.

Wesentlich mächtiger als rein namensbasierte Dienstbeschreibungen sind Ansätze, die auf der Beschreibung von Schnittstellen basieren, die im folgenden als *schnittstellenbasierte Modelle* bezeichnet werden. Solchen Modellen ist gemeinsam, dass sie die Informationsstrukturen beschreiben, über die die Funktion bzw. Teilfunktionen des Dienstes von externen Entitäten konkret genutzt werden kann. Um strukturelle Dienstbeschreibungen klassifizieren und auf eine systematische Weise miteinander vergleichen zu können, ist eine *Typisierung* notwendig, die es erlaubt, formale Relationen zwischen den Beschreibungen zu definieren. Deshalb folgt nun eine nähere Betrachtung der schnittstellenbasierten Modelle in Verbindung mit ihrer Typisierung.

Schnittstellentyp Schnittstellenbeschreibungen für verteilte Systeme enthalten in der Regel Information über die Funktion eines Dienstes, die sich i.a. aus mehreren Teilfunktionen zusammensetzt. Abhängig von der Art der Anwendung können solche Teilfunktionen durch unterschiedliche Abstraktionen modelliert werden, z.B. als *Operationen, Ströme* oder *Signale* [ODP94a]. Aus softwaretechnischer Sicht bietet die operationale Modellierung die passendste Abstraktionsstufe, da sie eine natürliche Übertragung der aus unterschiedlichen Programmierparadigmen bekannten Konzepte von Prozeduren (aus iterativen Programmiersprachen), Funktionen (aus funktionalen Sprachen) und Methoden bzw. Nachrichten (aus objektorientierten Sprachen) in die verteilten Anwendungssysteme darstellt. Im folgenden werden Schnittstellen deshalb ausschließlich aus dieser so genannten *operationalen Sichtweise* betrachtet.

Nach dieser operationalen Sichtweise umfasst eine Schnittstelle die Operationen, über die externe Entitäten auf die durch diese Schnittstelle bereitgestellte Funktionalität zugreifen kann. Dementsprechend besteht ein Schnittstellentyp aus einem Namen und einer Menge von Operationstypen. Ein *Operationstyp* besteht aus dem Operationsnamen sowie den Parametern und Rückgabewert dieser Operation, die zusammen die *Signatur* einer Operation ausmachen. Zur Beschreibung der Parameter und Rückgabewerte benötigt man außerdem *Da-*

tentypen, die entweder *einfache* oder *strukturierte* Daten beschreiben[4]. Zusammenfassend sind Schnittstellentypdefinitionen also mindestens wie folgt aufgebaut:

Definition 2.1.1 (Schnittstellentyp)
Ein Schnittstellentyp *besteht aus*

- *einem Schnittstellentypnamen*
- *einer Menge von* Operationstypen

Ein Operationstyp *besteht aus*

- *einem Operationstypnamen*
- *einer Liste von* Datentypen *für die Parameter*
- *einer Liste von* Datentypen *für die Rückgabewerte*

Ein Datentyp *ist ausgeprägt durch*

- *atomare Datentypen (z.B.* integer, float, character *etc.)*
- *strukturierte Datentypen (z.B.* record, list, set *etc.)*

Basierend auf einer solchen Schnittstellendefinition können wie bereits erwähnt formale Relationen zwischen Schnittstellen, wie z.B. *Konformitäts*beziehungen, definiert werden. Solche Relationen dienen dem strukturellen Vergleich von Dienstfunktionen und insbesondere der Feststellung, ob ein Dienst durch einen anderen zumindest strukturell *substituierbar* ist. Vor allem im Kontext offener verteilter Umgebungen können derartige Ersetzbarkeitsaussagen offensichtlich einen großen praktischen Nutzen bringen, da sie ein Mittel zum effektiven Umgang mit der diesen Umgebungen inhärenten Dynamik darstellen. Die klassische Konformitätsrelation, die auf den Ko- und Kontravarianzregeln für Funktionssubstitution basiert [Car88, CW85], kann bezüglich operationaler Schnittstellentypen, hier in Anlehnung an [ODP94b, MJ97], wie folgt definiert werden:

Definition 2.1.2 (Konformität von Schnittstellentypen) *Für zwei Schnittstellentypen S_1 und S_2 gilt: S_2 ist ein Subtyp von S_1 bzw. S_2 ist konform zu S_1, wenn folgende Bedingungen erfüllt sind:*

1. *Für jede Operation in S_1 gibt es in S_2 eine Operation mit demselben Namen und derselben Anzahl von Parametern und Rückgabewerten.*
2. *Für die Parametertypen $p_1, p_2, ..., p_n$ einer Operation in S_1 und für die Parametertypen $q_1, q_2, ..., q_n$ der entsprechenden Operation in S_2 gilt: p_i ist ein Subtyp von q_i für alle $1 \leq i \geq n$. (Kontravarianz)*
3. *Für die Rückgabetypen $r_1, r_2, ..., r_n$ einer Operation in S_1 und für die Rückgabetypen $t_1, t_2, ..., t_n$ der entsprechenden Operation in S_2 gilt: t_i ist ein Subtyp von r_i für alle $1 \leq i \geq n$. (Kovarianz)*

[4]Parameter und Rückgabewerte können auch Objektreferenzen oder selbst echte Objekte sein. In diesem Fall werden zu ihrer Beschreibung *Objekttypen*, die wiederum Schnittstellentypen enthalten, benötigt [Mey97]. Schnittstellentypdefinitionen können also verschachtelt sein.

Attributtyp Wie in Abschnitt 2.1.1 vorgestellt, wird der Zustand eines Dienstes, der einen wesentlichen Einfluss auf die Dienstfunktion und das Dienstverhalten haben kann, häufig als eine Menge von externen Attributen modelliert. Auf diese Weise kann der Zustand von außen abgefragt oder auch beeinflusst werden, wodurch das Verständnis für die Funktionsweise des Dienstes und damit die Interaktion mit dem Dienst erheblich verbessert bzw. transparenter gemacht werden kann. Attribute sind darüber hinaus dazu gut geeignet, unterschiedliche Funktionsweisen des Dienstes, als für die Dienstnehmer wählbaren *Optionen* zu modellieren. Auf diese Weise kann das Dienstverhalten also direkt, d.h. unabhängig von der Modellierung eines Zustandes, beeinflusst werden. Auf die Verwendungsmöglichkeiten von Attributen zur externen Einflussnahme auf die Dienste wird in Abschnitt 2.1.3 genauer eingegangen.

Zunächst einmal stellt sich die Frage, wie Dienstattribute zu modellieren sind. Dazu existieren verschiedene Ansätze. Der erste besteht darin, explizit *Dienstattributtypen* einzuführen, die exakt wie übliche Programmattribute modelliert werden können. In der Tat erlauben viele objektorientierte Programmiersprachen wie z.B. Java, Programmattribute einfach als "öffentlich" zu kennzeichnen, damit sie von außen zugegriffen werden können. Ein dazu gegensätzlich erscheinender Ansatz wäre, auf Dienstattribute als eigene Typen zu verzichten und stattdessen entsprechende, zusätzliche Methoden (in die Schnittstelle) einzuführen, um den Zugriff auf die relevanten Attribute zu realisieren. Dieser Ansatz hat den Vorteil, dass die operative Sichtweise der Schnittstellen vollständig erhalten bleibt und deshalb formale Relationen wie die obige Konformitätsbeziehung einfacher formuliert werden können, d.h. nicht um Attributtypen erweitert werden müssen. Demgegenüber hat der erste Ansatz den Vorteil, dass Attribute, die gewisse Dienst*eigenschaften* — im Gegensatz zur eigentlichen Dienstfunktion — repräsentieren, explizit als solche gekennzeichnet werden und somit unterschiedliche Dienstaspekte besser zum Ausdruck kommen. Auch müssen bei diesem Ansatz aus Implementationssicht keine zusätzlichen Methoden implementiert werden, falls die entsprechenden Dienstattribute bereits als Programmattribute vorhanden sind. Darüber hinaus kann man durch einfach zu automatisierende Transformationen aus dem ersten Ansatz wieder zu der reinen operativen Sichtweise gelangen, wenn es darum geht, die formalen Relationen zu prüfen. Wegen der adäquateren Modellierung wird in dieser Arbeit dieser Ansatz favorisiert. Demnach kann ein Attributtyp wie folgt definiert werden:

Definition 2.1.3 (Attributtyp)
Ein Attributtyp *besteht aus*

- *einem Attributtypnamen*
- *einem Datentyp zur Darstellung des Attributwertes*
- *einem Zugriffsmodus zur Kennzeichnung, ob das Attribut* gelesen *oder* geschrieben *werden darf*

Diensttyp Eine weitere Frage im Zusammenhang mit Dienstattributtypen ist, wo diese in die Gesamthierarchie der Typen einzuordnen sind. In einigen Arbeiten wird die Ansicht vertreten, dass Attributtypen nicht als Bestandteil von Schnittstellentypen, sondern von Diensttypen sind. Nach [ODP95, MJ97] führt dies zur folgenden Definition des Diensttypbegriffs:

Definition 2.1.4 (Diensttyp, 1. Variante)
Ein Diensttyp *besteht aus*

- *einem Diensttypnamen*
- *einem Schnittstellentyp*
- *einer Menge von Attributtypen*

Diese Definition ist allerdings in einer wesentlichen Hinsicht eingeschränkt: Sie lässt keine Dienste zu, die mehr als eine Schnittstelle besitzen. In der Praxis werden Mehrfachschnittstellen jedoch sehr häufig benötigt, um Teilfunktionen eines Dienstes sinnvoll zu gruppieren. Beispielsweise hat ein Trader als Unterstützungsdienst zur Vermittlung von Diensten unterschiedliche Schnittstellen zur Registrierung, Abfrage und Verwaltung von Diensten [OMG96]. Technisch gesehen kann jede einzelne Schnittstelle durch z.B. ein Objekt implementiert werden, das unabhängig von der Implementation anderer Schnittstellen ausgeführt und verwaltet werden kann. Hier kann sich die Dienstsicht also deutlich von der Implementations- bzw. Objektsicht unterscheiden. Mehrfachschnittstellen sind auch nützlich zur Integration bestehender Software (*legacy code*) in ein neues System, womit in der Übergangsphase sowohl die alte als auch die neue Schnittstelle gleichzeitig angeboten werden können (siehe dazu auch [Gri98]). Ein weiterer Vorteil besteht bei der Mehrfachvererbung (sowohl von Schnittstellen als auch Implementation) darin, dass die geerbten Schnittstellen, die i.a. unterschiedliche Funktionalitäten beschreiben, von der erbenden Entität nicht "zusammengewürfelt", sondern weiterhin getrennt gehalten werden können. Daher lassen in der Tat viele Systeme und Plattformen für verteilte Anwendungen — u.a. auch CORBA [Sie96] — bereits Mehrfachschnittstellen zu. In diesem Fall werden in der Regel Attribute als direkter Bestandteil einer Schnittstelle betrachtet, um sie auch durch den Gruppierungsmechanismus erfassen zu lassen. Damit kommen wir zur folgenden Definition des Diensttypbegriffs einschließlich der um Attributtypen erweiterten Schnittstellentypdefinition:

Definition 2.1.5 (Diensttyp, 2. Variante)
Ein Diensttyp *besteht aus*

- *einem Diensttypnamen*
- *einer Menge von* Schnittstellentypen

Ein Schnittstellentyp *besteht aus*

- *einem Schnittstellentypnamen*
- *einer Menge von Attributtypen*
- *einer Menge von Operationstypen*

Eine formale Subtyp- bzw. Konformitätsbeziehung zwischen Diensttypen lässt sich auf eine simple Weise aus der in Definition 2.1.2 angegebenen Konformitätsbeziehung für Schnittstellentypen[5] ableiten: Ein Diensttyp D_2 ist ein Subtyp von bzw. ist konform zu Diensttyp D_1, wenn es zu jedem Schnittstellentyp in D_1 einen konformen Schnittstellentyp in D_2 gibt.

[5] Die Konformitätsbeziehung für die um Attributtypen erweiterten Schnittstellentypen muss nicht neu definiert werden, da diese erweiterte Form — wie bereits dargestellt — durch einfache Transformationen der Attribute in Operationen wieder zu der reinen operativen reduziert werden kann.

2.1.2.2 Semantische Modelle

Strukturelle Dienstmodelle wie die im letzten Abschnitt dargestellten haben prinzipiell den großen Vorteil, dass sie aufgrund ihrer rein syntaktischen Dienstbeschreibungen die Definition präziser formaler Relationen zwischen den Diensten, die einer vollständig automatischen Verarbeitung zugänglich sind, ermöglichen. Beispielsweise kann die erwähnte Konformitätsrelation dazu verwendet werden, Typfehler beim entfernten Aufruf der an einer Schnittstelle angebotenen Dienstfunktion festzustellen bzw. zu vermeiden. Jedoch stellt gerade dieser rein syntaktische Charakter struktureller Dienst(beschreibungs)modelle auch ihren größten Nachteil dar: den Mangel an Information über die Semantik bzw. die Verhaltensweise von Diensten. D.h. strukturelle Dienstmodelle können bestenfalls eine technische, jedoch keine semantische Interoperabilität gewährleisten. Ein simples klassisches Beispiel für diese Problematik ist der Vergleich zwischen den Typen eines Kellerspeichers (*stack*) und eines Puffers (*buffer*), die zwar identische operationale Schnittstellenbeschreibungen besitzen, jedoch unterschiedliches Verhalten aufweisen [AHU83].

Wie bereits in Abschnitt "Dienstaspekte" (2.1.1) dargestellt, erscheint es andererseits sehr schwierig, die Semantik von Diensten bzw. von beliebigen Softwareentitäten formal präzise zu spezifizieren. Darüber hinaus ist es zumindest eine noch offene Frage, ob das Verhalten eines interaktiven Dienstes in einer offenen verteilten Umgebung auf eine algorithmische Weise *vollständig* erfasst werden kann (siehe Abschnitt 1.2.2). Daher existieren in der Literatur unterschiedliche Ansätze zur *partiellen* Spezifikation der Semantik von Diensten, die u.a. in folgende Kategorien eingeteilt werden können (vgl. Darstellung in [MJ97, Gri98]):

Konzeptgraphen Zur Darstellung semantischer Entitäten, d.h. im allgemeinen Dinge, denen man eine *Bedeutung* zuschreibt, können so genannte Konzeptgraphen bzw. semantische Netze, die — vor allem durch die Arbeiten von [Sow84, Sow91] seit Anfang der achtziger Jahre beeinflusst — aus dem Bereich Wissensrepräsentation stammen, verwendet werden. Dabei wird ein Konzeptgraph definiert als ein endlicher gerichteter Graph, der aus zwei Typen von Knoten, nämlich Konzepte und Relationen, besteht [Sow84, S. 73]. Eine der grundlegendsten und wichtigsten Relationen in Konzeptgraphen ist die *IS–A* Relation, die Subtypbeziehungen zwischen den Konzepten ausdrückt bzw. festlegt.

Da beliebige Entitäten in diesem Sinne als Konzepte modelliert werden können, gibt es auch Ansätze, Konzeptgraphen zur Beschreibung von Diensten zu verwenden [PMG95, PMGG95, PB96]. Beispielsweise wird in [PB96] ein Ansatz zur Spezifikation von operationalen Schnittstellentypen und Diensttypen als Konzeptgraphen vorgeschlagen. Dieser Ansatz basiert allerdings im wesentlichen auf einer 1–1–Transformation der Bestandteile dieser Typen, wie etwa Operations- und Attributtypen, in entsprechende Konzepte.

Der Vorteil von Konzeptgraphen besteht in der inhärenten Vernetzung der Konzepte, aus der Abhängigkeitsbeziehungen zwischen den durch die Konzepte dargestellten Instanzen direkt bestimmt werden können. Durch Hinzufügen bzw. Entfernen von Kanten und Relationen lassen sich solche Strukturen auf eine simple und natürliche Weise modifizieren und erweitern. Das grundlegende Problem mit Konzeptgraphen als semantisches Beschreibungsmittel für Anwendungsdienste ist jedoch, dass sie eine entsprechende *standardisierte* Konzep-

tualisierung der Dienste, ohne die ein systematischer Vergleich und damit eine maschinelle Verarbeitung unmöglich erscheint, voraussetzen. Angesichts der Vielfalt und vor allem der dynamischen Variation der angebotenen Diensttypen in der Praxis erscheint eine solche Standardisierungsanforderung gegenwärtig als unerfüllbar.

Protokollspezifikationen Ein anderer Ansatz zur Beschreibung des Verhaltens von Diensten bzw. diensterbringenden Objekten, der wirklich über die strukturellen, signaturbasierten Modelle hinausgeht, besteht in der Verwendung von Automatenmodellen zur Spezifikation von Interaktionsprotokollen, die bei der Nutzung des Dienstes eingehalten werden müssen. Derartige Protokolle machen explizit, in welchen Zuständen sich ein Dienst befinden kann, welche Interaktionen (Operationsaufrufe) in einem bestimmten Zustand zulässig sind und wann Zustandstransitionen stattfinden. Dieser Ansatz ist am prägnantesten vertreten in den Arbeiten von Nierstrasz et al. [ND95, Nie95].

In [Nie95] beispielsweise wird eine um Protokollspezifikationen erweiterte Typtheorie für *aktive Objekte* im Sinne von Diensten, deren Verhalten — insbesondere die momentane *Verfügbarkeit* von Operationen — von dem aktuellen Objektzustand abhängt, vorgeschlagen. Die Protokolle werden hier durch reguläre Typen [HU79] repräsentiert, wodurch eine formale Subtyprelation zwischen den Dienst- bzw. Objekttypen ermöglicht wird, auf die bekannte Algorithmen aus der Automatentheorie, beispielsweise zur Feststellung, ob zwei reguläre Ausdrücke bzw. endliche Automaten äquivalent sind, angewendet werden können. Die Grundidee dieser Subtyprelation besteht darin, dass alle möglichen Transitionen von einem bestimmten Zustand im Supertyp auch in allen entsprechenden Subtypen stattfinden können. Insbesondere wird in dieser Arbeit die Idee der Protokollspezifikation auch auf die Klienten- bzw. Dienstnehmerseite — deren Protokoll das Verhalten bzgl. der *Inanspruchnahme* bestimmter Dienstfunktionen ausdrückt — übertragen, so dass formal ausgedrückt werden kann, welche Diensterbringer gerade zu einem Dienstnehmer "passen", indem die jeweiligen Protokolle auf ihre formale Kompatibilität geprüft werden.

Regelbasierte Spezifikationen Ein weiterer, inzwischen relativ weit verbreiteter Ansatz zur partiellen Beschreibung der Semantik von Diensten basiert auf der Verwendung von formalisierten Regeln, die die Bedingungen angeben, unter denen eine Dienstfunktionalität genutzt werden kann.

Vor allem werden solche Regeln zur Formulierung der *Vor-* und *Nachbedingungen* für den Aufruf einer Operation eingesetzt, und zwar mit der Semantik, dass unter der Annahme, dass die Vorbedingungen erfüllt sind, der Aufruf stattfinden kann und durch ihn die Nachbedingungen garantiert wahr werden. Daher werden Schnittstellen, die um solche Bedingungen erweitert werden, auch mit dem Konzept eines *Vertrags* assoziiert, das durch die Arbeiten Meyers [Mey97] begründet wurde. Basierend auf dem Konzept der Vor- und Nachbedingungen gibt es eine Reihe von Arbeiten, die sich mit der Spezifikation, Evaluation und der Definition geeigneter Subtyprelationen befassen. Beispielsweise wird in [LC93] eine entsprechende Erweiterung von CORBA–Schnittstellen vorgeschlagen. Einige Arbeiten schlagen vor, dass klassische Subtyprelationen (wie etwa in Definition 2.1.2 angegeben) zusätzlich um die Forderung, dass ein Subtyp schwächere Vor- und stärkere Nachbedingungen als seine Supertypen besitzen

muss, erweitert werden [Ame91, LC93].

Ein neuerer allgemeiner Ansatz zur Spezifikation der Dienstsemantik, der die Verwendung von Regeln mit beinhaltet, ist im BOCA- (*Business Object Component Architecture*) Modell der OMG zu finden [OMG98a]. Ziel der BOCA–Spezifikation[6] ist es, ein standardisiertes komponentenbasiertes Modell für Geschäfts- bzw. *Business–Objekte* zu verwirklichen. Ein solches Modell, falls es sich in der Praxis durchsetzen kann, würde die computerunterstützte Abwicklung von Handelstransaktionen bzw. die Kooperationen zwischen Unternehmen, deren systemtechnische Unterstützung heutzutage bereits auf der konzeptionellen Ebene mit der Vielzahl von unterschiedlichen Geschäfts- und entsprechenden Informationsmodellen zu kämpfen hat und meist von Fall zu Fall speziell gelöst werden muss, in erheblichem Masse verbessern. Konzeptuell kann BOCA als eine Erweiterung von CORBA um eine dynamische *Komponentensicht*, die sich hauptsächlich in der Möglichkeit der Spezifikation von semantischen Entitäten als Zusätze[7] für die diensterbringenden Objekte, betrachtet werden. Derartige semantische Zusätze können in BOCA Spezifikationen von Attributen, Zuständen, Relationen und insbesondere Regeln unterschiedlicher Typen sein (siehe auch Abschnitt 3.1). Im Gegensatz zu den o.g. Vor- und Nachbedingungen, die als passive Constraints in bezug auf das Verhalten der Dienstfunktion zu verstehen sind, sieht BOCA auch so genannte Aktionsregeln vor, die *aktiv* eine eigenständige Wirkung bzw. Funktionalität, die außerhalb der betreffenden Dienstfunktion liegen kann, erbringen können.

2.1.3 Steuerung des Verhaltens von Diensten

Nach dieser klassifizierenden Betrachtung der verschiedenen Dienstaspekte und der vorhandenen Dienstmodelle können wir uns nun der Aufgabe zuwenden, die Steuerung offener verteilter Anwendungen im Sinne der erwähnten Anpassung der beteiligten Dienste zu präzisieren.

Entsprechend der Unterteilung der Dienstmodelle in strukturelle und semantische Modelle lassen sich auch die in Abschnitt 2.1.1 genannten Dienstaspekte in funktionsbezogene (Funktion und Struktur) bzw. verhaltensbezogene (Zustand, Verhalten und Semantik) unterteilen. Prinzipiell können dann für beide Kategorien Steuerungs- bzw. Anpassungsmechanismen spezifiziert werden:

1. Funktionale Steuerungsmechanismen: sind solche, die direkt die Funktion bzw. die Struktur eines Dienstes modifizieren, z.B. durch Veränderung der Schnittstellen und/oder Austausch der den Dienst implementierenden bzw. erbringenden Objekte.

2. Verhaltensbezogene Steuerungsmechanismen: sind solche, die Einfluss auf das Verhalten oder die Funktionsweise eines Dienstes ausüben bzw. den dienstnutzenden Entitäten zusätzliche Anwendungssemantik zur Verfügung stellen.

In diesem ersten Teil der Arbeit werden zunächst Steuerungsmechanismen konzipiert, die der zweiten Kategorie angehören. Im zweiten Teil der Arbeit,

[6] Zur Zeit als gemeinsamer, revidierter Standardisierungsvorschlag mehrerer Softwarefirmen unter der Federführung von Data Access Technologies, Inc. bei der OMG eingereicht

[7] die in der BOCA–Terminologie als *dependents, appliances* oder auch als *plug-ins* bezeichnet werden.

insbesondere in Kapitel 5.3, werden dann auch funktionale Dienstanpassungsmechanismen vorgestellt.

Um das Verhalten oder die Funktionsweise eines Dienstes zu steuern bzw. den sich verändernden Bedingungen einer offenen Umgebung anzupassen, können im Prinzip alle verhaltensbezogenen Aspekte, die durch Dienstmodelle wie die o.g., insbesondere die semantischen, zum Ausdruck kommen, in Betracht gezogen werden. Da solche (sowohl strukturellen als auch semantischen) Modelle jedoch in erster Linie der *Beschreibung* von Diensten zum Zweck einer korrekten Dienstnutzung dienen, geben sie zunächst keine Anhaltspunkte für entsprechende Ansätze zur Steuerung im Sinne einer *aktiven Einflussnahme* auf die Dienste. Beispielsweise könnten Dienste, deren Beschreibung um Konzeptgraphen oder Protokollspezifikationen erweitert werden, nur entsprechend aktiv gesteuert werden, wenn deren Semantik bezüglich dieser erweiterten Spezifikation auch modifizierbar ist, z.B. wenn der betreffende Dienst mehrere Protokolle unterstützt, die je nach Anforderungen aktiviert werden können. Ein weiteres Problem mit Steuerungsansätzen, die auf reichhaltigen, aber speziellen Semantiken wie den Protokollspezifikationen basieren, besteht darin, dass sie nur auf bestimmte Typen von Anwendungen beschränkt sind und damit die Anforderung der Generizität (siehe Abschnitt 1.4.1) nicht erfüllen können.

Diese Arbeit verfolgt deshalb einen regelbasierten Ansatz zur Steuerung des Verhaltens von Diensten, der konzeptionell von dem bereits erwähnten Umstand, dass das Verhalten eines Dienstes i.a. von seinem aktuellen *Zustand* abhängt, Gebrauch macht. Aufgrund dieser Abhängigkeit kann eine direkte Einflussnahme auf den Zustand eines Dienstes eine Veränderung des (beobachtbaren) Verhaltens bewirken. Ein solcher Ansatz ist grundsätzlich anwendbar auf alle Dienste, für die das Konzept der *Zustandsänderung* eine anwendungssemantische Rolle spielt, u.a. auch für die erwähnten Protokollspezifikationen, da eine Zustandsänderung im Protokollautomaten inhärent verbunden ist mit der (Nicht-)Verfügbarkeit gewisser Teilfunktionen. Um eine noch größere Flexibilität zu erreichen, werden in dieser Arbeit beliebige Dienstattribute, mit denen — wie bereits in Abschnitt 2.1.1 erwähnt — der Dienstzustand implizit oder explizit repräsentiert werden kann, als direkte Steuerungspunkte gewählt. Das heißt, dass dieser Ansatz (systemtechnisch) auf alle Dienste anwendbar sind, für die das Konzept einer *Attributsänderung* explizit vorhanden ist, und zwar auch dann, wenn das Konzept des Dienstzustands nicht direkt modelliert ist.

Das Konzept der Modifikation von Attributen, um einen Steuerungs- bzw. Anpassungseffekt zu erzielen, ist jedoch an sich weder neu noch sehr mächtig. In vielen neueren, vor allem den komponentenbasierten Systemen und Produkten zur Entwicklung verteilter Anwendungen, z.B. Suns (Enterprise) Java Beans, OMGs CORBA (Property Service, BOCA–Modell), IBMs SanFrancisco (Plattform für Business–Objekte), Microsofts ActiveX/DCOM und ObjectSpaces Voyager, gibt es bereits integrierte Mechanismen der Anwendungskonfiguration (die auch häufig unter dem Begriff *Customizing* vermarktet wird) über die Einstellung von extern zugänglichen Attributen, die in der Regel als *Properties* bezeichnet werden, um deutlich zu machen, dass sie explizit zum Zweck der Beschreibung und Einstellung bzw. Anpassung von Dienst*eigenschaften* eingeführt werden (siehe hierzu die produktbezogene Darstellung von Properties in [Gri98]). Aus Sicht der Anwendungsentwicklung realisieren derartige Properties die Möglichkeit der externen *Parametrisierung*, die ein wohl bekanntes grundlegendes Konzept der Programmierung darstellt. Attributbasierte

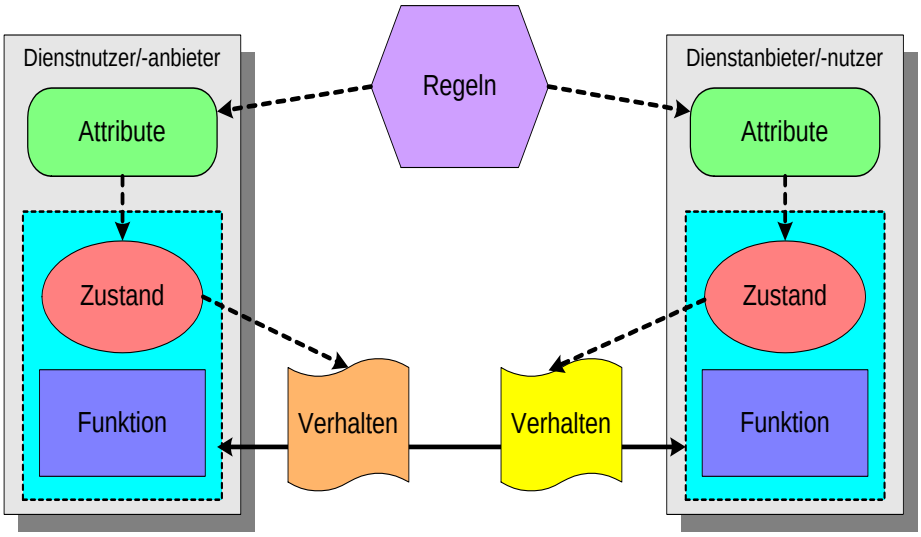


Abbildung 2.1: Regelbasierte Steuerung des Dienstverhaltens

Konfigurationsmechanismen haben sich also gerade wegen ihrer Simplizität, sowohl bezüglich der Konzeption, der Realisation als auch der Bedienung als eine der effektivsten Steuerungsmöglichkeiten in der Praxis manifestiert.

Um den Wirkungsgrad bewährter, attributbasierter Steuerungsmechanismen essentiell zu erhöhen und damit höhere Steuerungs- und Automatisierungsanforderungen zu erfüllen, werden sie in dieser Arbeit mit der Ausdrucksmächtigkeit logikbasierter, aktiver Regeln, deren Aktivierung eine voll *automatische* Konfiguration bzw. Anpassung des Dienstzustands zur Folge haben kann, verknüpft. Darüber hinaus liegt ein Schwerpunkt der vorgeschlagenen Steuerungsmechanismen darin, das Verhalten mehrerer Dienste je nach (individuellen) Anforderungen so aufeinander anzupassen, dass eine anwendungssemantisch reibungslose Interaktion zwischen ihnen stattfinden kann. Daher wird dieser Ansatz, der schematisch in Abbildung 2.1 dargestellt ist, auch als *interaktionsorientiert* bezeichnet [TGML99].

2.2 Logische und mathematische Grundlagen

Obwohl es nicht zu den primären Zielen der vorliegenden Arbeit gehört, neue formallogische Konzepte und Techniken zu erarbeiten, wurden während der Durchführung der entsprechenden praktischen Vorarbeiten immer wieder spezifische Anforderungen aus unterschiedlichen Anwendungsdomänen festgestellt, für die generische Lösungskonzepte zwar als äußerst sinnvoll erscheinen, jedoch konnten in der Literatur, zumindest durch die Recherchebemühungen des Autors, keine formalen Techniken bzw. Verfahren gefunden werden, die direkt auf diese Anforderungen anwendbar wären. Deshalb wurden solche Verfahren Schritt für Schritt entwickelt, um in erster Linie *exemplarisch* nachzuweisen, dass geeignete formale Techniken zu einer generischen Gesamtlösung konkreter Steuerungsprobleme erheblich beitragen können. Dies impliziert jedoch, dass diese Arbeit i.a. keinen Anspruch darauf erheben möchte, bereits die besten

bzw. optimalen Lösungen bezüglich der formallogischen Verfahren realisiert zu haben (siehe auch Kapitel 7).

Geeignete formale Grundlagen sind im allgemeinen entscheidend für die *Mächtigkeit* einer konkreten Realisation technischer Konzepte, die letztlich ihre Effektivität und insbesondere Generizität im praktischen Einsatz bestimmt. Andererseits können aber ausdrucksmächtige Formalismen häufig zu einer *Komplexität* führen, die entweder theoretisch (z.B. auf Grund von Entscheidbarkeitsproblemen) oder praktisch (z.B. durch übermäßige Anforderungen an Rechenressourcen) nicht beherrschbar ist. Deshalb ist es von großer Bedeutung, ein ausgewogenes Verhältnis zwischen diesen beiden Faktoren zu finden. Die in dieser Arbeit vorgestellten regelbasierten Steuerungsmechanismen machen Gebrauch von einigen Grundlagen der formalen Logik und der Mathematik, die im folgenden eingeführt werden, um das Verständnis der folgenden Konzepte und Algorithmen zu ermöglichen. Die Darstellung beschränkt sich in der Regel auf die für diese Arbeit relevanten Aspekte.

2.2.1 Prädikatenlogik

2.2.1.1 Aussagen- versus Prädikatenlogik

Klassische Logik — als Abgrenzung zu vielen neueren formalen Logiken wie etwa Temporal-, Modal-, Relevanz-, Nichtmonotone Logiken etc., die auf ihr basieren — ist eines der wichtigsten theoretischen Fundamente für viele Fachdisziplinen, vor allem die Mathematik und Informatik, und wird unterteilt in *Aussagen-* und *Prädikatenlogik*. Die Aussagenlogik beschäftigt sich ausschließlich mit logischen Verknüpfungen von (atomaren) *Aussagensymbolen*, denen einer der beiden Wahrheitswerte *wahr* und *falsch* zugewiesen werden kann. Der Hauptvorteil der Aussagenlogik ist ihre Simplizität und die *Entscheidbarkeit* der wichtigsten Eigenschaften von logischen Sätzen, nämlich *Allgemeingültigkeit* und *Erfüllbarkeit*. Ihr größter (potentieller) Nachteil besteht in der eingeschränkten Ausdrucksmächtigkeit aufgrund der Atomarität der Sätze bzw. "Aussagen". Im Gegensatz dazu bietet die Prädikatenlogik die Möglichkeit, innerhalb einzelner Formeln explizit Bezug auf *Individuen* zu nehmen — durch so genannte Prädikationen mittels Funktionen oder Relationen — und ist deshalb wesentlich ausdrucksmächtiger als die Aussagenlogik. Da die Ausdrucksmächtigkeit der Aussagenlogik für die Anforderungen der dieser Arbeit zu Grunde liegenden Anwendungsdomäne der elektronischen Dienstemärkte nicht ausreicht, wurde als Ausdrucksmittel für die Regelspezifikationen eine spezielle Form der Prädikatenlogik — genauer ausgedrückt, der Prädikatenlogik erster Stufe[8] — gewählt, deren relevanten Aspekte hier hauptsächlich in Anlehnung an [Fit96] und [EMC99] dargestellt werden.

2.2.1.2 Syntax der Prädikatenlogik

Jede prädikatenlogische Sprache bzw. *Signatur* besteht aus folgenden *gemeinsamen* Bestandteilen:

Junktoren

$\neg a$: Negation von a

[8] In der Prädikatenlogik erster Stufe wird ausschließlich über Individuen bzw. Objekte, und nicht etwa über Funktionen und Relationen, quantifiziert.

$a \vee b$: Disjunktion von a und b
$a \wedge b$: Konjunktion von a und b
$a \rightarrow b$: Implikation von a und b, gleich bedeutend mit $\neg(a \wedge \neg b)$
$a \leftrightarrow b$ (bzw. $a \equiv b$): Äquivalenz von a und b, gleich bedeutend mit $(a \rightarrow b) \wedge (b \rightarrow a)$

Konstanten
$\top$: verum (wahr)
$\bot$: falsum (falsch)

Quantoren
$\forall$: der All- bzw. Universalquantor (für alle)
$\exists$: der Existenzquantor (es gibt)

Trennsymbole ')', '(' und ','

Variablen v_1, v_2,... (auch informal als x, y, z,...)

Nun kommen wir zu den Bestandteilen, die jeweils von der Ausdrucksmächtigkeit der gewählten Sprache abhängen.

Definition 2.2.1 (Logische Signatur) *Eine* logische Signatur *erster Ordnung ist ein Tripel* $(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *mit*

1. $\mathbf{R}$ *ist eine endliche oder aufzählbare Menge von* Relations- *oder* Prädikatensymbolen, *die jeweils mit einer positiven ganzen Zahl assoziiert sind. Wenn* $P \in \mathbf{R}$ *mit* n *assoziiert ist, wird* P *als ein* n*–stelliges Relationssymbol bezeichnet.*

2. $\mathbf{F}$ *ist eine endliche oder aufzählbare Menge von* Funktionssymbolen, *die jeweils mit einer positiven ganzen Zahl assoziiert sind. Wenn* $f \in \mathbf{F}$ *mit* n *assoziiert ist, wird* f *als ein* n*–stelliges Funktionssymbol bezeichnet.*

3. $\mathbf{C}$ *ist eine endliche oder aufzählbare Menge von* Konstantensymbolen.

Die Sprache, die auf einer solchen Signatur basiert, wird mit $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *bezeichnet und wie folgt aus den* Termen *und* Formeln *aufgebaut.*

Definition 2.2.2 (Terme) *Die Familie der* Terme *von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist die kleinste Menge, die folgende Bedingungen erfüllt:*

1. *Jede Variable ist ein Term von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

2. *Jedes Konstantensymbol (Element von* $\mathbf{C}$*) ist ein Term von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

3. *Wenn* f *ein* n*–stelliges Funktionssymbol (Element von* $\mathbf{F}$*) ist und* $t_1, ..., t_n$ *Terme von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *sind, dann ist* $f(t_1, ..., t_n)$ *ein Term von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

Ein Term heißt geschlossen, *wenn er keine Variablen enthält. Für arithmetische Funktionen kann anstatt der Präfixschreibweise auch die Infixnotation verwendet werden, beispielsweise* $(x + y)$ *anstatt* $+(x, y)$.

Definition 2.2.3 (atomare Formeln) *Eine* atomare Formel *von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist jede Zeichenkette der Form* $R(t_1, ..., t_n)$*, wobei* R *ein* n*–stelliges Relationssymbol (Element von* $\mathbf{R}$*) und* $t_1, ..., t_n$ *Terme von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *sind. Auch* $\top$ *und* $\bot$ *sind atomare Formeln.*

Auch für mathematische Relationen kann anstatt der Präfixschreibweise die Infixnotation verwendet werden, beispielsweise $(x < y)$ *anstatt* $< (x, y)$*.*

Definition 2.2.4 (Formeln) *Die Familie der* Formeln *von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist die kleinste Menge, die folgende Bedingungen erfüllt:*

1. *Jede atomare Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist eine Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$*.*
2. *Wenn* A *eine Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist, ist auch* $\neg A$ *eine Formel dieser Sprache.*
3. *Für jeden binären Junktor* $\circ$ *gilt, wenn* A *und* B *Formeln von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *sind, dann ist auch* $(A \circ B)$ *eine Formel dieser Sprache.*
4. *Wenn* A *eine Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *und* x *eine Variable ist, dann sind* $(\forall x)A$ *und* $(\exists x)A$ *Formeln von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$*.*

Definition 2.2.5 (Rang) *Der* Rang $r(F)$ *einer Formel* F *erster Ordnung berechnet sich wie folgt:*

1. $r(A) = r(\neg A) = 0$*, wenn* A *atomar, aber nicht* $\top$ *oder* $\bot$ *ist.*
2. $r(\top) = r(\bot) = 0$
3. $r(\neg\top) = r(\neg\bot) = 1$
4. $r(\neg\neg Z) = r(Z) + 1$
5. $r(A \circ B) = r(A) + r(B) + 1$

Definition 2.2.6 (freie, gebundene Variablen) *Die* freien *Variablen in einer Formel werden wie folgt definiert:*

1. *Alle Variablen in einer atomaren Formel sind frei.*
2. *Freie Variablen in* $\neg A$ *sind alle freien Variablen in* A*.*
3. *Freie Variablen in* $(A \circ B)$ *sind die freien Variablen in* A *plus die freien Variablen in* B*.*
4. *Freie Variablen in* $(\forall x)A$ *bzw.* $(\exists x)A$ *sind die freien Variablen in* A *außer* x*.*

Eine Variable heißt gebunden*, wenn sie nicht frei ist.*

Definition 2.2.7 (Satz bzw. geschlossene Formel) *Ein* Satz *(oder* geschlossene Formel*) von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist eine Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$*, die keine freien Variablen enthält.*

2.2.1.3 Normalformen

Normalformen spielen eine große Rolle bei bei vielen Schluss- bzw. Inferenzmechanismen, automatischen Beweisverfahren und in der logischen Programmierung. Die wichtigsten Normalformen werden wie folgt definiert:

Definition 2.2.8 (Disjunktive Normalform) *Eine Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist in* disjunktiver Normalform (DNF)*, wenn sie von der Form* $P_1 \vee ... \vee P_n$ *ist, wobei jedes* $P_i, 1 \leq i \leq n$ *von der Form* $Q_1 \wedge ... \wedge Q_m$ *ist, wobei jedes* $Q_i, 1 \leq i \leq m$ *eine atomare oder eine negierte atomare Formel ist.*

Definition 2.2.9 (Konjunktive Normalform) *Eine Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *ist in* konjunktiver Normalform (KNF)*, wenn sie von der Form* $P_1 \wedge ... \wedge P_n$ *ist, wobei jedes* $P_i, 1 \leq i \leq n$ *von der Form* $Q_1 \vee ... \vee Q_m$ *ist, wobei jedes* $Q_i, 1 \leq i \leq m$ *eine atomare oder eine negierte atomare Formel ist.*

Definition 2.2.10 (Gentzen–Formel) *Eine Formel von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *heißt* Gentzen–Formel*, wenn sie von der Form* $(P_1 \wedge ... \wedge P_n) \rightarrow (Q_1 \vee ... \vee Q_m)$ *ist, wobei* $P_1 ... P_n, Q_1 ... Q_m$ *atomare Formeln sind.*

Definition 2.2.11 (Hornformel) *Eine Formel F von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *heißt* Horn–Formel*, wenn einer der folgenden beiden Fälle zutrifft.*

1. *F ist von der Form* $(P_1 \wedge ... \wedge P_n) \rightarrow \perp$*, wobei* $P_1 ... P_n$ *atomare Formeln sind.*

2. *F ist von der Form* $(P_1 \wedge ... \wedge P_n) \rightarrow Q$*, wobei* $P_1 ... P_n, Q$ *atomare Formeln sind.*

Im ersten Fall heißt F negative Hornformel*, im zweiten Fall* positive Hornformel.

Normalformen werden in erster Linie als Hilfsformate verwendet, um bestimmte Eigenschaften von beliebigen nicht–normalisierten Formeln nachzuweisen oder um neue Formeln aus bestehenden zu konstruieren. Es stellt sich also die Frage nach der Umformbarkeit von beliebigen Formeln in eine bestimmte Normalform, die somit die Ausdrucksmächtigkeit sowie den Wirkungsgrad dieser Normalform als Hilfsformat bestimmt. Hierzu gibt es Äquivalenzergebnisse aus der klassischen Logik, von denen die wichtigsten und für die weiteren Ausführungen relevanten hier wiedergegeben werden. Die entsprechenden Beweise können in vielen Standardwerken gefunden werden, u.a. auch in [EMC99], und werden deshalb hier nicht angeführt.

Zunächst sind bzgl. einzelner Formeln folgende Äquivalenzergebnisse relevant:

Satz 2.2.1 (Äquivalenzen von Normalformen — einzelne Formeln)

1. *Zu jeder Formel F von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *kann eine logisch äquivalente Formel F' in DNF konstruiert werden.*

2. *Zu jeder Formel F von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *kann eine logisch äquivalente Formel F' in KNF konstruiert werden.*

3. *Es gibt Formeln von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, *die zu keiner Gentzen–Formel logisch äquivalent sind.*

Weitere relevante Äquivalenzbeziehungen können nicht in bezug auf einzelne Formeln formuliert, sondern nur unter Betrachtung von Formel*mengen* dargestellt werden:

Satz 2.2.2 (Äquivalenzen von Normalformen — Formelmengen)

1. *Zu jeder Formelmenge* $\Phi \subseteq L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *existiert eine logisch äquivalente Formelmenge* Φ', *deren Elemente alle in DNF sind.*

2. *Zu jeder Formelmenge* $\Phi \subseteq L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *existiert eine logisch äquivalente Formelmenge* Φ', *deren Elemente alle in KNF sind.*

3. *Zu jeder Formel* F *von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *kann eine zu* F *logisch äquivalente endliche Menge von Gentzen–Formeln konstruiert werden.*

4. *Zu jeder Formelmenge* $\Phi \subseteq L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ *existiert eine logisch äquivalente Menge von Gentzen–Formeln.*

5. *Es gibt Formeln von* $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, *die zu keiner Menge von Horn–Formeln logisch äquivalent sind.*

2.2.1.4 Entscheidbarkeitsergebnisse

Die Prädikatenlogik, auch wenn man lediglich die erste Stufe betrachtet, ist in ihrer Allgemeinheit ein mächtiges Instrument zur Beschreibung logischer Bedingungen und Zusammenhänge, wie sie in vielen Anwendungsdomänen festgestellt werden können. Allerdings ist diese Ausdrucksmächtigkeit i.a. mit einer Verarbeitungskomplexität, die in der Praxis zu unlösbaren Problemen bzw. zu einem inakzeptablen Systemlaufzeitverhalten führen kann, verbunden. Am gravierendsten drückt sich dies in den Entscheidbarkeitsergebnissen aus, die in bezug auf die Grundfragen der semantischen Eigenschaften negativ ausfallen. Obwohl hier auf eine Beschreibung der in allen Standardwerken ähnlich dargestellten Semantik der Prädikatenlogik, die intuitiven Begriffen wie "erfüllbar" und "allgemeingültig" erst formal eine Bedeutung gibt, komplett verzichtet wird, seien die wesentlichsten Ergebnisse der Entscheidbarkeit in Kurzform festgehalten:

- Das *Gültigkeitsproblem* der Prädikatenlogik ist unentscheidbar: Zur Entscheidung steht die Frage, ob eine beliebige prädikatenlogische Formel *(allgemein-)gültig* ist, d.h. ob sie bei jeder *Interpretation* zu *wahr* evaluiert. Obwohl diese Frage unentscheidbar ist, ist sie zumindest *semientscheidbar*, d.h. es gibt ein Berechnungsverfahren, das nach endlicher Zeit abbricht, *wenn* die eingegebene Formel gültig ist.

- Das *Erfüllbarkeitsproblem* der Prädikatenlogik ist unentscheidbar und auch nicht semientscheidbar: Zur Entscheidung steht hier die Frage, ob es zu einer beliebigen Formel F mindestens ein *Modell* gibt, d.h. eine Interpretation, in der F gilt. Dieses Problem ist bekanntlich nicht einmal semientscheidbar. (Dies kann indirekt dadurch bewiesen werden, dass das Komplement der erfüllbaren Formeln, also die Menge aller *kontradiktorischen* Formeln genau wie die Menge der gültigen Formeln semientscheidbar ist.)

- Das Problem der *logischen Implikation* ist nicht entscheidbar, aber semientscheidbar: Zur Entscheidung steht hier die Frage, ob zwischen zwei beliebigen Formeln die Implikationsbeziehung besteht. Auch dieses Problem ist unentscheidbar, jedoch noch semientscheidbar, denn es kann auf das Gültigkeitsproblem zurückgeführt werden (siehe auch [GN87]).

Schon allein aufgrund der negativen Entscheidbarkeitsergebnisse muss also bei der Konzeption jedes konkreten Anwendungssystems, das sich die Ausdrucksmächtigkeit der Prädikatenlogik zunutze machen will, die Komplexitätsfrage in Betracht gezogen werden. In der Praxis bedeutet dies, dass geeignete *Einschränkungen* definiert werden müssen. Deshalb werden wir in Abschnitt 2.3 unterschiedliche Einschränkungen für die regelbasierten Steuerungsmechanismen vorschlagen, die zu unterschiedlich komplexen Verarbeitungsalgorithmen führen.

2.2.2 Das Simplexverfahren

Um die zunächst eingeschränkte Mächtigkeit der zur Steuerung verteilter Anwendungssysteme eingesetzten formalen Regeln (siehe Abschnitt 2.3) zu erweitern, wird in der zweiten Stufe das Simplexverfahren als mathematisches Hilfsmittel, insbesondere zur Feststellung der Implikationsbeziehung zwischen Formeln, benötigt werden. Deshalb sollen hier die für die nachfolgenden Verarbeitungsmechanismen relevanten Details dieses Verfahrens — hier in Anlehnung an [BM87, Sch87, Kun98] — dargestellt werden.

Das Simplexverfahren ist 1951 von George Dantzig an der Stanford Universität entwickelt worden. Es ist eines der effektivsten Verfahren zur linearen Optimierung. Obwohl es im schlimmsten Fall NP–vollständiges Laufverhalten [Sch87, S. 139ff] aufweist, ist das durchschnittliche Laufzeitverhalten polynomiell [Sch87, S. 142ff]. Der nachfolgende Abschnitt wird zunächst in die elementaren Begriffe dieses Verfahrens einführen. Im Anschluss werden die Algorithmen des Simplexverfahrens beschrieben, die zur Realisierung der regelbasierten Steuerungsmechanismen implementiert wurden.

2.2.2.1 Elementare Definitionen

Dieser Abschnitt soll ein paar elementare Definitionen einführen, die notwendig für das Verständnis der weiteren Abschnitte sind. Das Symbolverzeichnis am Ende der Arbeit (S. 355) enthält die hier verwendeten mathematischen Symbole, die in diesem Abschnitt deshalb nicht erklärt werden. Grundkenntnisse in linearer Algebra werden vorausgesetzt.

Definition 2.2.12 (Lineares Gleichungssystem) *Ein* lineares Gleichungssystem *sei von der Form:* $Hx = d$. *Hierbei ist* H *eine Matrix mit* m *Zeilen und* n *Spalten, die die Koeffizienten enthält,* x *ein Vektor mit* n *Zeilen, der aus den Variablen besteht und* d *der Lösungsvektor, der ebenfalls aus* n *Zeilen besteht.*

Definition 2.2.13 (Basis) *Ein Indexvektor* $B = (\beta(1), \ldots, \beta(r))$ *mit* $\beta(l) \in \{1, \ldots, n\}$ *heißt* Basis *von* (H, d), *wenn die zu den Variablen* $x_{\beta(l)}$ *gehörigen*

Spaltenvektoren $H_{.\beta(l)}$ *von* H *linear unabhängig sind* $(l = 1, \ldots, r)$. *Die Mengen*

$$BV = \{\beta(1), \ldots, \beta(r)\} \qquad \textit{und} \qquad NBV = \{1, \ldots, n\} \backslash BV$$

werden als Basis- *bzw.* Nichtbasismengen *bezeichnet. Entsprechend werden die Variablen* $x_{\beta(l)}, l = 1, \ldots, r,$ Basisvariable *und* $x_j (j \in NBV)$ Nichtbasisvariable *genannt.* [BM87, S. 3]

In den nachfolgenden Abschnitten wird sich auf diese Definitionen bezogen.

Verkürzte Tableaus, Basislösungen Das Simplexverfahren arbeitet auf einem zu Grunde liegendem linearen Gleichungssystem wie in Definition 2.2.12 eingeführt, also letztendlich auf einer Matrix. Die Darstellung der Einheitsvektoren einer Matrix ist letztlich redundant. *Verkürzte* Tableaus tragen dazu bei, die Datenstruktur des Simplextableaus, sprich die Matrix so redundanzfrei wie möglich zu gestalten, indem eine andere Darstellung der Einheitsvektoren gewählt wird. Ein *Normaltableau* hat die folgende Form:

(2.1)

x_B^t	x_N^t	
E	$\hat{A}$	b

Ein daraus gewonnenes verkürztes Tableau hat die Form:

(2.2)

	x_N^t	
x_B	$\hat{A}$	b

Beispiel 2.2.1 *Das Normaltableau:*

z	s_1	s_2	s_3	x_1	x_2	
1	0	0	0	−3	−2	0
0	1	0	0	2	1	100
0	0	1	0	1	1	80
0	0	0	1	1	0	40

ergibt das verkürzte Tableau:

	x_1	x_2	
z	−3	−2	0
s_1	2	1	100
s_2	1	1	80
s_3	1	0	40

mit:

$$x_B = \begin{pmatrix} z \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}, \quad x_N^t = (x_1, x_2)$$

Das Halten der Einheitsmatrix E entfällt, da diese aus dem Vektor x_B konstruiert werden kann.

Ein Austauschschritt — auch *Pivotschritt* genannt — ist eine Umformung der Matrix derart, dass die Spalte, in der das zur Umformung herangezogene Element steht, welches auch *Pivotelement* genannt wird, zu einem Einheitsvektor umgeformt wird.

Definition 2.2.14 (Austauschschritt) *Unter einem* Austauschschritt *zu einem Element a_{ij} zu einer Matrix $\hat{A} \in Mat(m,n)$, $i \in \{1,\ldots,m\}$ und $j \in \{1,\ldots,n\}$ wird das Multiplizieren der Zeile i mit dem Wert $\frac{1}{a_{ij}}$ mit dem anschließenden Subtrahieren der Zeile i von den Zeilen k ; $k \in \{1,\ldots,m$; $k \neq i$ derart, dass alle Elemente der Spalte j außer dem Element a_{ij} Null ergeben. Das Element a_{ij} wird auch* Pivotelement *genannt.*

Im Satz (2.2.3) sind die auf das verkürzte Tableau angepassten Umformungen des Austauschschrittes beschrieben.

Satz 2.2.3 *Sei a_{ik} ein Element von $\hat{A}$ mit $a_{ik} \neq 0$, $k \in \{1,\ldots,p\}$. Aus B und N erhält man eine neue Basis B' mit zugehöriger Nichtbasis N' durch*

$$\beta'(l) = \begin{cases} \beta(l), l \in \{1,\ldots,r\}, l \neq i \\ v(k), l = i \end{cases} \qquad v'(j) = \begin{cases} v(j), j \in \{1,\ldots,p\}, j \neq k \\ \beta(i), j = k \end{cases}$$

Setzt man $a_{i,p+1} = b_i, a'_{i,p+1} = b'_i$, so ergibt sich die Matrix $\hat{A}'$ vermöge folgender Umformungen:

$$\begin{array}{lll} a) & a'_{ik} = \frac{1}{a_{ik}} & \\ b) & a'_{jk} = -\frac{a_{jk}}{a_{ik}} & (j = 1,\ldots,r, j \neq i) \\ c) & a'_{il} = \frac{a_{il}}{a_{ik}} & (l = 1,\ldots,p+1, l \neq k) \\ d) & a'_{jl} = a_{jl} - \frac{(a_{jk}a_{il})}{a_{ik}} & \begin{array}{l}(l = 1,\ldots,p+1, l \neq k) \\ (j = 1,\ldots,r, j \neq i)\end{array} \end{array}$$

[BM87, S. 14]

Eine *Basislösung* ist der Ergebnisvektor, der in Definition 2.2.12 als Vektor d beschrieben ist. Es ist eine gültige Belegung der Variablen des Gleichungssystems.

Definition 2.2.15 (Basislösung) *Ein Gleichungssystem S nach Def. 2.2.12 sei lösbar und B eine Basis von S. Setzt man $x_v = 0$ (alle $v \in NBV$), so ist das Gleichungssystem eindeutig nach den Basisvariablen $x_\beta (\beta \in BV)$ auflösbar. Diese spezielle Lösung heißt* Basislösung *von S zur Basis B. Liegt S in der Form des Tableaus (2.2) vor, so ist die Basislösung gegeben durch:*

$$x_B = b \quad , \quad x_N = 0$$

[BM87, S. 15]

Zulässige Tableaus Ein *zulässiges* Tableau (Simplextableau) ist eine Matrix, die die Eigenschaft hat, keine negativen Werte im Ergebnisvektor zu haben. Dies kann durch geeignetes Umformen der Restriktionen erreicht werden. Der im nächsten Abschnitt vorgestellte Algorithmus arbeitet ausschließlich auf zulässigen Tableaus.

Definition 2.2.16 (Zulässige Tableaus)
Gegeben sei:

$$\begin{aligned} \textit{Zielfunktion}: &\quad x_0 + \hat{a^t} x_N = b_0 \\ \textit{Restriktionen}: &\quad x_B + \hat{A} x_N = b \end{aligned}$$

Damit schreibt sich in verkürzter Form*:*

(2.3)

	x_N^t	
x_0	$\hat{a}^t$	b_0
x_B	$\hat{A}$	b

Das Tableau (2.3) wird zulässig *genannt, wenn $b \geq 0$ gilt, dh. wenn für die zugehörige Basislösung $\bar{x}_B = b$, $\bar{x}_N = 0$ zulässig ist.* [BM87, S. 21]

2.2.2.2 Simplexalgorithmus für zulässige Tableaus

Die zur Implementation herangezogenen Algorithmen des Simplexverfahrens werden alle auf den Simplexalgorithmus für zulässige Tableaus zurückgeführt. Ein nicht zulässiges Tableau wird, wie an späterer Stelle gesehen werden kann, mit Hilfe des Zweiphasen–Simplexalgorithmuses in ein zulässiges umgeformt.

Algorithmus 2.2.1 (Simplexalgorithmus für zulässige Tableaus)

(0) Initialisierung*: Ausgangspunkt sei ein zulässiges Tableau (2.3) für das Problem (P).*

(1) Pivotwahl*:*

(a) *Pivotspaltenindex k: Wähle $k \in \{1, \ldots, p\}$ so, dass $a_{0k} > 0$ ist. Ist diese Wahl nicht möglich,* stop *(die zum Tableau gehörige Basislösung ist optimal).*

(b) *Pivotzeilenindex i: Wähle $i \in \{1, \ldots, r\}$ so, dass gilt:*

$$\frac{b_i}{a_{ik}} = Min\{\frac{b_j}{a_{jk}} \mid j \in \{1, \ldots, r\}, a_{jk} > 0\}$$

Ist diese Wahl nicht möglich, stop *(x_0 ist nach unten unbeschränkt).*

(2) Austauschschritt*:*

Rechne mit den Formeln des Austauschschrittes die Matrix

$$\begin{pmatrix} \hat{a}^t & b_0 \\ \hat{A} & b \end{pmatrix} \quad \textit{um in} \quad \begin{pmatrix} \hat{a}'^t & b'_0 \\ \hat{A}' & b' \end{pmatrix}$$

$$\textit{Setze} \quad \begin{pmatrix} \hat{a}^t & b_0 \\ \hat{A} & b \end{pmatrix} \quad := \quad \begin{pmatrix} \hat{a}'^t & b'_0 \\ \hat{A}' & b' \end{pmatrix}$$

Ersetze in B die Komponente $\beta(i)$ durch $v(k)$, in N die Komponente $v(k)$ durch $\beta(i)$, bezeichne die neuen Indexvektoren wieder mit B und N und gehe zu (1).

[BM87, S. 27]

Antizyklen–Technik Wird das Simplexverfahren formal auf ein Tableau angewendet, indem der Pivotspaltenindex i so gewählt werden muss, dass $b_i = 0$ gilt, so ändert sich nach Durchführung des Austauschschrittes zwar das Tableau, nicht aber die zugehörige Basislösung; ebenfalls unverändert bleibt der Zielfunktionswert. Bei eindeutig festgelegter Pivotwahl kann der Verlauf des Verfahrens zyklisch werden. [BM87, S. 30]

Beispiel 2.2.2 *Gegeben sei die folgende Aufgabenstellung:*

$$\left\{\begin{array}{r} z + 2x_3 + 2x_4 - 8x_5 - 2x_6 = 0 \\ x_2 - 7x_3 - 3x_4 + 7x_5 + 2x_6 = 0 \\ x_1 + 2x_3 + x_4 - 3x_5 - x_6 = 0 \\ x_j \geq 0 (j = 1, \ldots, 6), z \textit{ minimal} \end{array}\right\}$$

Nun wird der Algorithmus 2.2.1 unter Benutzung von Normaltableaus angewendet. Die Pivotwahl sei so festgelegt, dass bei mehreren Möglichkeiten für die Indexwahl immer der am weitesten links bzw. am weitesten oben stehende Index gewählt wird.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	b	*Basis* $\hat{B}$
1	0	0	2	2	−8	−2	0	$\hat{B} = (0,1,2)$
0	0	1	−7	−3	7	2	0	
0	1	0	(2)	1	−3	−1	0	
1	−1	0	0	1	−5	−1	0	$\hat{B} = (0,1,3)$
0	7/2	1	0	(1/2)	−7/2	−3/2	0	
0	1/2	0	1	1/2	−3/2	−1/2	0	
1	−8	−2	0	0	2	2	0	$\hat{B} = (0,4,3)$
0	7	2	0	1	−7	−3	0	
0	−3	−1	1	0	(2)	1	0	
1	−5	−1	−1	0	0	1	0	$\hat{B} = (0,4,5)$
0	−7/2	−3/2	7/2	1	0	(1/2)	0	
0	−3/2	−1/2	1/2	0	1	1/2	0	
1	2	2	−8	−2	0	0	0	$\hat{B} = (0,6,5)$
0	−7	−3	7	2	0	1	0	
0	(2)	1	−3	−1	1	0	0	
1	0	1	−5	−1	−1	0	0	$\hat{B} = (0,6,2)$
0	0	(1/2)	−7/2	−3/2	7/2	1	0	
0	1	1/2	−3/2	−1/2	1/2	0	0	
1	0	0	2	2	−8	−2	0	$\hat{B} = (0,1,2)$
0	0	1	−7	−3	7	2	0	
0	1	0	2	1	−3	−1	0	

Nach sechs Iterationen wird also wieder ein Tableau zur Basis $\hat{B} = (0,1,2)$ erreicht. Bei gleich bleibender Pivotwahl werden diese Tableaus in zyklischer Folge durchlaufen [BM87, S. 30f].

Die nachfolgende Definition ist notwendig, um eine geeignete Pivotwahl zu formulieren, die Abhilfe aus diesem Dilemma schafft.

Definition 2.2.17 (lexikographisch positiv) *Ein Vektor $u \in \mathcal{R}^S$ heißt* lexikographisch positiv *(in Zeichen: $u \succ 0$), falls gilt:*

1) $u \neq 0$;

2) die erste Komponente u_i von u $(1 \leq i \leq s)$, die von Null verschieden ist, ist positiv.

Ein Vektor $v \in \mathcal{R}^S$ heißt *lexikographisch größer* als u $(v \succ u)$, falls $v - u \succ 0$ ist. Sei $V \subseteq \mathcal{R}^S$ und $u \in V$, dann heißt u *lexikographisches Minimum* der Menge V $(u = \text{lex Min } V)$, falls $u \preceq v$ für alle $v \in V$ gilt. [BM87, S. 31]

Wenn im Algorithmus (2.2.1) das Verfahrenselement (1) durch (1') ersetzt wird, dann bricht der Algorithmus nach endlich vielen Schritten ab.

(1') *Verschärfte Pivotwahl*

a) Pivotspalte k: Wähle $k \in \{1, \ldots, n\}$ mit $a_{0k} > 0$. Ist eine solche Wahl nicht möglich, *stop*, Optimallösung erreicht.

b) Pivotzeile i: Wähle $i \in \{1, \ldots, r\}$ mit

$$\frac{1}{a_{ik}}(b_i, A_{i.}) = \quad \text{lex Min} \quad \{\frac{1}{a_{jk}}(b_j, A_{j.}) \mid j = 1, \ldots, r; a_{jk} > 0\}$$

Ist eine solche Wahl nicht möglich, *stop*, (P) nicht lösbar.

[BM87, S. 32]

Beispiel 2.2.3 *Nun wird die Aufgabe aus Beispiel 2.2.2 mit der modifizierten Pivotwahl bearbeitet:*

x_0	x_1	x_2	x_3	x_4	x_5	x_6	b	*Basis* $\hat{B}$
1	0	0	2	2	−8	−2	0	$\hat{B} = (0,1,2)$
0	0	1	−7	−3	7	2	0	
0	1	0	(2)	1	−3	−1	0	
1	−1	0	0	1	−5	−1	0	$\hat{B} = (0,1,3)$
0	7/2	1	0	1/2	−7/2	−3/2	0	
0	1/2	0	1	(1/2)	−3/2	−1/2	0	
1	−2	0	−2	0	−2	0	0	$\hat{B} = (0,2,4)$
0	3	1	−1	0	−2	−1	0	
0	1	0	2	1	−3	−1	0	

Nach zwei Austauschschritten ist ein Optimaltableau erreicht.

Die Regel zur verschärften Pivotwahl setzt ein Normaltableau der Form (2.1) aus Abschnitt 2.2.2.1 voraus. Das ist jedoch kein Problem, da aus einem verkürzten Tableau der Form (2.3) aus Abschnitt 2.2.2.1 der Einheitsvektor auf einfache Art zurückgewonnen werden kann.

In [BM87, S. 33–34] wird die Ansicht vertreten, dass die modifizierte Pivotauswahlregel die Gesamtrechenzeit des Simplexverfahrens nachteilig beeinflusst und dass sich das Simplexverfahren mit der einfachen Pivotwahl in praxisnahen Beispielen so gut wie nie in einer Entartung endgültig verfängt, sondern ihr immer zufällig entschlüpft. In [Sch87, S. 138] hingegen wird berichtet, dass solche Schleifen in praxisnahen Problemen weitaus öfter vorkommen als anzunehmen sei. Im Bereich der ganzzahligen Optimierung ist die Anwendung einer spezialisierten Pivotauswahlregel nicht nur aus theoretischer Sicht begründet, denn es entstehen viele hochgradige Entartungen, aus denen das Verfahren schneller austritt, wenn die spezialisiertere Pivotwahl verwendet wird [BM87]. Da die Implementation des ganzzahligen Simplexverfahrens angestrebt wird, ist es sinnvoll, dass die Antizyklentechnik zum Einsatz kommt.

2.2.2.3 Zweiphasen–Simplexalgorithmus

Die *Zweiphasen–Methode* beschreibt einen Algorithmus, um ein Tableau so umzuformen, dass es mit dem Simplexalgorithmus für zulässige Tableaus bearbeitet werden kann. Im Allgemeinen liegen nämlich Probleme der folgenden Form vor:

$$\left\{ \begin{array}{c} A^1u = b^1, A^2u \geq b^2, A^3u \leq b^3 \\ z - c^tu = q, z \quad \text{minimal} \quad , u \geq 0 \end{array} \right\} \tag{2.4}$$

mit $A^i \in Mat(m_i, s); c, u = (x_1, \ldots, x_c)^t \in \mathcal{R}^S; b^i \in \mathcal{R}^{m_i}; z, q \in \mathcal{R}$. [BM87, S. 55]

Um das Problem (2.4) in die Form (2.5) zu überführen, werden die folgenden Umformungen auf den Restriktionen vorgenommen:

- Jede $\leq$ Ungleichung wird durch das Hinzuaddieren einer Variablen (Schlupfvariable) zur linken Seite in Gleichungsform gebracht.
- Jede $\geq$ Ungleichung wird durch Subtraktion einer Variablen (Schlupfvariable) von der linken Seite in Gleichungsform gebracht.
- Jede Variable x, die nicht die Nichtnegativitätsbedingung erfüllt, wird als Differenz zweier nicht–negativer Variablen x' und x'' dargestellt. (In der Problemstellung (2.4) sind alle $u \geq 0$, dass muss aber nicht immer der Fall sein.)

$$\left\{ \begin{array}{c} A^1u = b^1, A^2u - v = b^2, A^3u + y = b^3 \\ z - c^tu = q, z \quad \text{minimal} \quad , u \geq 0, v \geq 0, y \geq 0 \end{array} \right\} \tag{2.5}$$

[BM87, S. 55]

Die Vektoren v und y heißen *Schlupfvektoren*, ihre Komponenten *Schlupfvariablen*. Die gegebenen Matrizen

$$H = \begin{pmatrix} A^1 & 0 \\ A^2 & -E \end{pmatrix} \quad x = \begin{pmatrix} u \\ v \end{pmatrix} \quad d = \begin{pmatrix} b^1 \\ b^2 \end{pmatrix}$$

$$R = \begin{pmatrix} A^3 & 0 \end{pmatrix} \qquad f = b^3 \qquad a = \begin{pmatrix} c \\ 0 \end{pmatrix}$$

[BM87, S. 55]

werden zusammengefasst und ergeben das Problem:

$$(P_0) \quad \left\{ \begin{array}{c} Hx = d, x \geq 0 \\ y + Rx = f, y \geq 0 \\ z - a^t x = q, z \quad \text{minimal} \end{array} \right\}$$

mit $d \geq 0, f \geq 0$. Die Bezeichnungen für die Dimensionen seien wie folgt festgelegt: $H \in Mat(m,n), R \in Mat(p,n)$, $d \in \mathcal{R}^p, x = (x_1, \ldots, x_n)^t, y = (x_{n+1}, \ldots, x_{n+p})^t; a \in \mathcal{R}^n$. [BM87, S. 55]

Das Problem (P_0) lässt sich nicht direkt in Gestalt eines zulässigen Tableaus darstellen, da in der Regel eine zulässige Basis unbekannt ist. Die erste Phase der Zweiphasen–Methode dient der Ermittlung eines zulässigen Tableaus (T) für das Problem (P_0), oder dem Nachweis, dass ein solches nicht existiert. Die zweite Phase führt ausgehend von (T) die Optimierung durch und stimmt daher mit dem Simplexverfahren für zulässige Tableaus überein.

Durch Einführung eines *künstlichen Vektors* $w = (x_{n+p+1}, \ldots, x_{n+p+m})^t$ werden die Restriktionen von (P_0) wie folgt modifiziert:

$$w + Hx = d, y + Rx = f, x \geq 0, y \geq 0, w \geq 0$$

[BM87, S. 56]

Somit ist eine zulässige Basis durch $(n+1, \ldots, n+p+m)$ gegeben. Die Komponenten von w werden als künstliche Variablen bezeichnet.

Algorithmus 2.2.2 (Zweiphasen–Simplexalgorithmus)
(0) Initialisierung*: Das laufende Tableau sei*

(T)

	x^t	
z	$\hat{a}^t$	b_0
z_1	$\hat{g}^t$	g_0
x_B	$\hat{A}$	b

Anfänglich setze:

$$\hat{a} = -a, b_0 = q, x_B = \begin{pmatrix} w \\ y \end{pmatrix}, \hat{A} = \begin{pmatrix} H \\ R \end{pmatrix}, b = \begin{pmatrix} d \\ f \end{pmatrix}$$

und wähle eine Hilfszielfunktionszeile $z_1 + \hat{g}^t x_N = g_0$ durch Aufsummieren eines Teils der in der Basis befindlichen künstlichen Variablen.

Phase 1*:*

(1) Algorithmus (2.2.1)*: Wende diesen Algorithmus auf (T) an unter ausschließlicher Benutzung der Hilfszielfunktionszeile z_1 für die Pivotwahl. Streiche dabei jede künstliche Variable, die in die Nichtbasis gelangt. Ist (T) z_1-optimal, gehe zu (2).*

(2) Test*:*

a*) Ist $g_0 > 0$,* stop*, P_0 besitzt keine zulässige Lösung.*

b) Sind alle die Hilfszielfunktion z_1 betreffenden künstlichen Variablen aus dem Tableau (T) entfernt, so gehe zu (3).

c) Sonst wähle in einer zu einer solchen Variablen gehörigen Zeile i ein beliebiges Pivot $a_{ik} \neq 0$, führe einen Austauschschritt durch und streiche anschließend die Pivotspalte. Ist der genannte Austausch wegen $\hat{A}_{i\cdot} = 0$ unmöglich, streiche Zeile i aus (T) und gehe zu b).

(3) Neue Hilfszielfunktion*: Sind noch nicht alle künstlichen Variablen aus (T) entfernt, so wähle eine neue Hilfszielfunktionszeile z_1, indem die Zeilen eines Teils der in der Basis befindlichen künstlichen Variablen aufsummiert werden. Gehe zu (1). Andernfalls zu Phase 2.*

Phase 2:

Wende auf (T) nun Algorithmus (2.2.1) unter Benutzung der Zielfunktion z (erste Zeile) an. [BM87, S. 66]

2.2.2.4 Dualer Simplexalgorithmus

Der Begriff und die Definition der Dualität wird benötigt, wie später gesehen werden kann, um (Un-)Gleichungssysteme mit ganzzahligen Variablen zu lösen.

Satz 2.2.4 *Jedem Tableau (P) entspricht genau ein Tableau von (D) gemäss folgender Zuordnung:*

(2.6)

a)	x_N^t	*max*		*b)*	u_B^t	*min*
x_0	$\hat{a}^t$	b_0	$\longrightarrow$	u_0	$-b^t$	b_0
x_B	$\hat{A}$	b		u_N	$-\hat{A}^t$	$\hat{a}$

(In den Tableaus ist vermerkt, ob es sich um eine Maximum- oder Minimumaufgabe handelt.)

[BM87, S. 81]

Analog lässt sich nach Satz (2.2.4) ein zu minimierendes Tableau von der Form (2.6 a) in die Form (2.6 b) überführen.

Definition 2.2.18 *Das Tableau (2.6 a) zur Aufgabe (P) heißt* dual zulässig*, wenn $\hat{a} \geq 0$ ist. Es wird als* primal zulässig *bezeichnet, wenn $b \geq 0$ gilt.*

[BM87, S. 82]

Der duale Simplexalgorithmus wird für die ganzzahlige Optimierung benötigt. Nach Erreichen eines Optimums mit Hilfe des Zweiphasen–Simplexalgorithmus sind alle Koeffizienten kleiner oder gleich Null, da der Zweiphasen–Simplexalgorithmus minimiert. Durch das Setzen von $z := (-z)$ in der Zielfunktionszeile ist immer gewährleistet, dass das Tableau dual zulässig ist. Das Setzen von $z := (-z)$ bewirkt außerdem, dass weiter mit dem maximierenden dualen Simplexalgorithmus gerechnet werden kann. Doch dazu mehr im Kapitel über ganzzahlige Optimierung (Abschnitt 2.2.2.5).

Algorithmus 2.2.3 (Dualer Simplexalgorithmus)

(0) Initialisierung: *Laufendes Tableau sei ein dual zulässiges Tableau der Gestalt (2.6 a). Anfänglich sei ein solches aus der Aufgabenstellung gewonnen.*

(1) Pivotwahl:

a) *Zeilenindex i: Wähle $i \in \{1,\ldots,r\}$ mit $b_i < 0$. Ist dies nicht möglich,* stop, *(2.6 a) ist primal zulässig.*

b) *Spaltenindex k: Wähle $k \in \{1,\ldots,p\}$ gemäß*

$$\frac{a_{0k}}{a_{ik}} = Max\{\frac{a_{0l}}{a_{il}} \mid l \in \{1,\ldots,p\},\ a_{il} < 0\}$$

Ist dies nicht möglich, stop, *es gibt keine zulässigen Punkte.*

(2) Austauschschritt *zum Pivot a_{ik}. Gehe zu (1).*
[BM87, S. 86]

2.2.2.5 Ganzzahliger Simplexalgorithmus

Bei der ganzzahligen Optimierung liegt das Problem in folgender Form vor:

$$(P) \quad \left\{ \begin{array}{l} z = c^{1t}\,x^1 + c^{2t}\,x^2 + q \rightarrow max \\ H^{11}\,x^1 + H^{12}\,x^2 \leq d^1 \\ H^{21}\,x^1 + H^{22}\,x^2 = d^2 \\ \lambda^i \leq x^i \leq \mu^i \ (i = 1,2) \\ x := (x^1, x^2) \in \mathcal{Z}^{n_1} \times \mathcal{R}^{n_2} \end{array} \right\}$$

[BM87, S. 189]

mit $H^{ij} \in Mat(m_i, n_j), i, j = 1,2$. Die Indexmenge $\{1,\ldots,n_1\}$ wird mit G abgekürzt; die Variablen x_k $(k \in G)$ heißen *ganzzahlige Variable.* In der Aufgabenstellung (P) wird zugelassen, dass die Vektoren λ^i auch Komponenten $-\infty$, die Vektoren μ^i Komponenten $+\infty$ besitzen $(i = 1,2)$. Die endlichen Komponenten von λ^1, μ^1 seien jedoch stets ganzzahlig. Außerdem wird stets vorausgesetzt, dass die Projektion von $M(P)$ in $\mathcal{R}^{n_1}$ beschränkt sein möge.

Die Aufgabe (P) heißt

a)	gemischt–ganzzahlig,	wenn $n_2 \neq 0$ ist,
b)	wesentlich–ganzzahlig,	wenn $n_2 = 0$ ist,
c)	rein–ganzzahlig,	wenn $n_2 = 0$, $m_2 = 0$, $c^1 \in \mathcal{Z}^{n_1}$, $d^1 \in \mathcal{Z}^{m_1}$, $H^{11} \in Mat(m_1, n_1, \mathcal{Z})$ gilt.

[BM87, S. 190]

Konstruktion der Schnitte Das ganzzahlige Optimierungsproblem wird Anfangs als ein stetiges behandelt und mit den aus den vorigen Abschnitten schon bekannten Verfahren gelöst. Die Optimallösung wird, sofern sie existiert und nicht ganzzahlig ist, durch einen Schnitt weggeschnitten ohne eine ganzzahlige Lösung mit wegzuschneiden. Danach wird die Zielfunktion erneut maximiert. So fortfahrend entsteht eine Folge linearer Programme bis eine Optimallösung auftritt, die die Ganzzahligkeitsanforderungen erfüllt [BM87, S. 192].

Das Optimaltableau einer der linearen Optimierungsaufgaben, die in diesem Sinne aus dem Problem (P) entstehen, werde mit

$$(T) \quad \begin{array}{c|c|c} & x_N^t & \\ \hline x_B & A & b \end{array} \qquad A = (a_{ij}), \quad \begin{array}{l} i = 0, \ldots, s \\ j = 1, \ldots, p \end{array}$$

bezeichnet. Zeile 0 ist Zielfunktionszeile, (x_B, x_N) umfasst alle Variablen von (P) einschließlich möglicher Schlupfvariablen, oBdA. sind die Variablen von $0, \ldots, p+s$ abgezählt. (T) ist *Arbeitstableau* (oder *laufendes Tableau* der beiden folgenden Algorithmen.

Im folgenden wird nun beschrieben, wie aus (T) ein Schnitt H zu konstruieren ist: Sind für die zu (T) gehörige Basislösungen $x^0 = (x_B^0, x_N^0) = (b, 0)$ alle Ganzzahligkeitsanforderungen erfüllt, so ist — da x^0 (P)–optimal ist — das Ziel des Verfahrens erreicht. Andernfalls gibt es eine Zeile $i \in \{1, \ldots, s\}$ mit

$$x_{\beta(i)} + \sum_{l=1}^{p} a_{il}\, x_{v(l)} = b_i,\ b_i \in \mathcal{Z}, \tag{2.7}$$

wo $x_{\beta(i)}$ eine ganzzahlige Variable, also $\beta(i) \in G$ ist.

Aus (2.7) soll nun ein Schnitt konstruiert werden. Die Restriktion (2.7) wird daher auch *schnitterzeugende Restriktion* genannt.

Dazu betrachte für jedes $l \in \{1, \ldots, p\}$ mit $v(l) \in G$

$$\begin{array}{rl} & a_{il} = [a_{il}] + r_{il},\ r_{il} \in [0, 1) \\ \text{sowie} & b_i = [b_i] + r_i,\ r_i \in (0, 1). \end{array}$$

Dann ist

$$\sum_{v(l) \in G} r_{il}\, x_{v(l)} + \sum_{v(l) \notin G} a_{il}\, x_{v(l)} - r_i = [b_i] - \sum_{v(l) \in G} [a_{il}]\, x_{v(l)} - x_{\beta(i)} \in \mathcal{Z}$$

für alle $x \in M(P)$. Setze

$$\begin{array}{l} N_+ = \{j \mid 1 \le j \le p,\ a_{ij} \ge 0,\ v(j) \notin G\}, \\ N_- = \{j \mid 1 \le j \le p,\ a_{ij} < 0,\ v(j) \notin G\} \end{array}$$

Lemma 2.2.5

$$\sum_{v(l) \in G} r_{il}\, x_{v(l)} + \sum_{l \in N_+} a_{il}\, x_{v(l)} + \sum_{l \in N_-} \frac{r_i}{r_i - 1} a_{il}\, x_{v(l)} \ge r_i$$

ist ein Schnitt, der x^0 wegschneidet. Dessen Schlupfvariable darf als ganzzahlig angenommen werden, wenn $N_- = \emptyset$. [BM87, S. 192–193]

Die im Lemma 2.2.5 beschriebene Restriktion wird nach dem Aufstellen durch das Multiplizieren beider Seiten mit (-1) umgeformt. Dadurch dreht sich die Relation von "$\ge$" in "$\le$" und die Restriktion kann unter Hinzunahme einer weiteren Schlupfvariablen in das Tableau aufgenommen werden:

$$-\sum_{v(l) \in G} r_{il}\, x_{v(l)} - \sum_{l \in N_+} a_{il}\, x_{v(l)} - \sum_{l \in N_-} \frac{r_i}{r_i - 1} a_{il}\, x_{v(l)} \le -r_i$$

Ein endliches Verfahren In diesem Abschnitt soll nun ein endliches Verfahren zur Lösung linearer ganzzahliger (Un-)Gleichungssysteme beschrieben werden. Zum Verständnis des Algorithmuses werden nachfolgend einige Notationen und der Begriff der lexikalischen dualen Zulässigkeit eingeführt.

Für die Matrix $C = (c_{ij})$, $i = 0, \ldots, s$; $j = 1, \ldots, m$; bezeichnet C^τ für $\tau \in \{0, \ldots, s\}$ die Teilmatrix von C, die aus den Zeilen 0 bis τ besteht.

Definition 2.2.19 *Das Tableau (T) heißt* bis einschließlich Zeile $\tau \in \{0, \ldots, s\}$ lexikographisch dual zulässig, *wenn alle Spalten von A^τ lexikographisch nichtnegativ sind. Ist $\tau = s$, so heißt (T) einfach* lexikographisch dual zulässig.

[BM87, S. 198]

Der nachfolgende Algorithmus ist unter der Voraussetzung, dass der optimale Zielfunktionswert — sofern er existiert — ganzzahlig ist, endlich:

Algorithmus 2.2.4 (Ganzzahliger Simplexalgorithmus)

(0) Initialisierung: *Das laufende Tableau sei*

$$(T) \quad \begin{array}{c|c|c} & x_N^t & \\ \hline x_B & A & b \end{array}$$

mit $A = (a_{ij})$, $i = 0, \ldots, s$; $j = 1, \ldots, p$. Anfänglich sei (T) gleich einem Optimaltableau von (P_{st}) mit der Zielfunktionszeile als Zeile 0. Setze $G = G \cup \{0\}$, $\beta(0) = 0$, $x_0 = z$, $s_0 = s; \tau = 0$. Existiert kein Optimaltableau von (P_{st}), stop 1.

(1) Konstruktion eines Schnittes: *Wähle ein minimales $l \in \{0, \ldots, s\}$ mit den Eigenschaften: $\beta(i) \in G$, $b_l \notin \mathcal{Z}$. Berechne aus Zeile l den Schnitt gemäß Lemma (2.2.5). Dessen Schlupfvariable bezeichne mit einer noch nicht verwendeten Variablen x_j, $j > p + s$. Ist $N_- = \emptyset$, fordere die Ganzzahligkeit der zusätzlich eingeführten Schlupfvariablen. Füge die Schnittrestriktion (als letzte Zeile) zum laufenden Tableau hinzu. Setze $s = s + 1$. Existiert keine solche Zeile l,* stop 2.

(2) Optimale Zwischenlösung: *Optimiere (T) mit dem dualen Simplexverfahren. Stelle dabei sicher, dass bei jedem Austauschschritt die lexikographisch duale Zulässigkeit von (T) bis einschließlich Zeile τ erhalten bleibt und b^τ im Sinne der lexikographischen Ordnung höchstens fällt. Existiert kein Optimaltableau,* stop 1.

(3) Aufdatieren von τ:

a) *Gibt es eine Zeile $j \in \{1, \ldots, \tau\}$ mit $\beta(j) > p + s_0$ oder $\beta_{(j)} \notin G$, so setze $\tau = min\{j \mid \beta(j) \notin G$ oder $\beta(j) > p + s_0\} - 1$. Streiche alle Zeilen j mit $\beta(j) > p + s_0$ aus dem Tableau.*

b) *Ist $\beta(j) \in G$ für alle $j \in \{1, \ldots, \tau\}$, $b_j \notin \mathcal{Z}$ für ein $j \in \{1, \ldots, \tau\}$, gehe zu (1).*

c) *Sonst setze $\tau = \tau + 1$.*

(4) Lexikographisch duale Zulässigkeit:

a) *Ist* $\tau \leq s$ *und* $\beta(\tau) \notin G$, *so tausche die Zeile* τ *mit einer Zeile* $\tau_1 > \tau$, $\tau_1 \in \{0, \ldots, s\}$, *mit* $\beta(\tau_1) \in G$. *Geht dies nicht, oder ist* $\tau > s$, stop 2, *sonst zu c).*

b) *Ist* $\beta(\tau) \in G$, *gehe zu c).*

c) *Die vorhandene lexikographisch duale Zulässigkeit von* (T) *in den Zeilen 0 bis* $\tau - 1$ *ist auf die Zeilen 0 bis* τ *zu erweitern. Dabei muss der Vektor* $b^{\tau-1}$ *unverändert und* $\beta(j) \in G$, $j = 1, \ldots, \tau$ *bleiben. Gehe zu (1).*

[BM87, S. 198–199]

Satz 2.2.6 *Es gelte die obige Voraussetzung. Ist die Zielfunktion auf* $M(P_{st})$ *beschränkt, so verläuft der Algorithmus (2.2.4) endlich. Endet er bei* stop 2, *so ist die Basislösung* (P)*–optimal.* [BM87, S. 200]

Die Verfahrensschritte (2) und (4) konkret:

Die Verfahrensschritte (2) und (4) sind in dem Algorithmus 2.2.4 nur allgemein beschrieben. Eine genaue Ausgestaltung dieser Verfahrensschritte wird hier gegeben.

(2) Optimale Zwischenlösung

2.1 Pivotwahl:

PZ: Wähle $i \in \{1, \ldots, s\}$ mit $b_i < 0$. Existiert kein solches i, gehe zu (3).

PS: Wähle $k \in \{1, \ldots, p\}$ gemäß

$$\frac{1}{a_{ik}} A^{\tau}_{.k} = \quad \text{lex max} \quad \{\frac{1}{a_{ij}} A^{\tau}_{.j} \mid 1 \leq j \leq p,\ a_{ij} < 0\}$$

Existiert kein solches k, *stop 1.*

2.2 Führe einen Austauschschritt zum Pivot a_{ik} durch. Gehe zu 2.1.

(4) Lexikographisch duale Zulässigkeit

4.1 Ist $\tau \leq s$ und $\beta(\tau) \notin G$, tausche Zeile τ gegen eine Zeile $\tau_1 > \tau$, für die $\beta(\tau_1) \in G$ gilt. Geht dies nicht, oder ist $\tau > s$, *stop 2.* Setze $N_\tau = \{l \mid 1 \leq l \leq p,\ a_{jl} = 0 \text{ für } 0 \leq j \leq \tau - 1$.

4.2 Pivotwahl:

PS: Wähle $k \in N_\tau$ mit $a_{\tau k} < 0$. Existiert dies nicht, gehe zu (1).

PZ: Wähle $i \in \{1, \ldots, s\}$, $\frac{b_i}{a_{ik}} = min\{\frac{b_j}{a_{jk}} \mid 1 \leq j \leq s,\ a_{jk} > 0,\ j \neq \tau\}$

4.3 Führe einen Austauschschritt zum Pivot a_{ik} durch. Gehe zu 4.2.

[BM87, S. 205–206]

Die Pivotwahl in (4) wird durch die Antizyklentechnik erweitert, so dass die Pivotzeile über das lexikalische Minimum ermittelt wird.

Beispiel 2.2.4 *Die Aufgabe sei gegeben durch:*

$$\begin{array}{lll} \textit{Zielfunktion:} & -4\,x_2 - 5\,x_3 + \frac{104}{7} & \Rightarrow \ max \\ & 2\,x_1 + 2\,x_2 + 5\,x_3 \leq 11 & \\ & 2\,x_1 + 2\,x_2 + 4\,x_3 \geq 7 & \\ & 2\,x_1 + 4\,x_2 + 7\,x_3 = 18 & \\ & x_1 \leq 5 & \\ & x_2 \leq 10 & \end{array}$$

Es seien x_1, x_2 ganzzahlig. Da nun der Zweiphasen–Algorithmus für das Minimierungsproblem konzipiert ist, ist $z := (-z)$ zu setzen (.s Abschnitt 2.2.2.6). Daraus ergibt sich für die Zielfunktion:

$$4\,x_2 + 5\,x_3 - \frac{104}{7} \ \Rightarrow \ min$$

Im weiteren seien die ganzzahligen Variablen mit einem $*$ gekennzeichnet. Das Anfangstableau ist dann das folgende:

	x_1^*	x_2^*	x_3	s_4	
z	0	-4	-5	0	$-\frac{104}{7}$
s_1	2	2	5	0	11
h_1	2	2	4	-1	7
h_2	2	4	7	0	18
s_2	1	0	0	0	5
s_3	0	1	0	0	10

Die s_i bezeichnen die Schlupfvariablen und die h_i die Hilfsvariablen. Mit dem Zweiphasen–Simplexverfahren kann dieses Tableau nun in ein Optimaltableau gebracht werden. Die Basislösung $(x_1, x_2, x_3) = (2, \frac{7}{2}, 0)$ erfüllt jedoch nicht alle Ganzzahligkeitsanforderungen. Es wird jetzt für z zusätzlich die Ganzzahligkeit gefordert. Das nachfolgende Tableau ist das Optimaltableau des Zweiphasen–Simplexverfahrens, wobei z wieder $-z$ gesetzt wurde, da der ganzzahlige Simplexalgorithmus für das Maximum konzipiert ist.

	s_1	x_3	
z^*	2	1	$\frac{6}{7}$
x_2^*	$-\frac{1}{2}$	1	$\frac{7}{2}$
s_2	1	1	4
x_1^*	1	$\frac{3}{2}$	2
s_3	-1	$-\frac{3}{2}$	3
s_4	$\frac{1}{2}$	-1	$\frac{13}{2}$

Schnitterzeugende Zeile wird Zeile 0. Der zulässige Schnitt ist:

$$x_4^* - 2s_1 - x_3 = -\frac{6}{7}$$

Dieser Schnitt wird als letzte Zeile in das Tableau aufgenommen:

	s_1	x_3	
z^*	2	1	$\frac{6}{7}$
x_2^*	$-\frac{1}{2}$	1	$\frac{7}{2}$
s_2	1	1	4
x_1^*	1	$\frac{3}{2}$	2
s_3	-1	$-\frac{3}{2}$	3
s_4	$\frac{1}{2}$	-1	$\frac{13}{2}$
x_4^*	(-2)	-1	$-\frac{6}{7}$

Mit Klammern ist das Pivotelement gekennzeichnet, welches für das duale Simplexverfahren herangezogen wird. Der Austauschschritt ergibt das Tableau:

	x_4^*	x_3	
z^*	1	0	0
x_2^*	$-\frac{1}{4}$	$\frac{5}{4}$	$\frac{26}{7}$
s_2	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{25}{7}$
x_1^*	$\frac{1}{2}$	1	$\frac{11}{7}$
s_3	$-\frac{1}{2}$	-1	$\frac{24}{7}$
s_4	$\frac{1}{4}$	$-\frac{5}{4}$	$\frac{44}{7}$
s_1	$-\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{7}$

Die nächste Schnitterzeugende Zeile ist Zeile 1. Der Schnitt ist:

$$x_5^* - \frac{3}{4}x_4^* - \frac{5}{4}x_3 = -\frac{5}{7}$$

Hinzufügen zum Tableau ergibt:

	x_4^*	x_3	
z^*	1	0	0
x_2^*	$-\frac{1}{4}$	$\frac{5}{4}$	$\frac{26}{7}$
s_2	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{25}{7}$
x_1^*	$\frac{1}{2}$	1	$\frac{11}{7}$
s_3	$-\frac{1}{2}$	-1	$\frac{24}{7}$
s_4	$\frac{1}{4}$	$-\frac{5}{4}$	$\frac{44}{7}$
s_1	$-\frac{1}{2}$	$\frac{1}{2}$	$\frac{3}{7}$
x_5^*	$-\frac{3}{4}$	$(-\frac{5}{4})$	$-\frac{5}{7}$

Der nächsten Pivotschritt liefert:

	x_4^*	x_5^*	
z^*	1	0	0
x_2^*			3
s_2			
x_1^*			1
s_3			
s_4			
s_1			
x_3			$\frac{4}{7}$

(Die Werte, die nicht von Interesse für die Basislösung sind, sind hier ausgeblendet.)

2.2.2.6 Minimieren/Maximieren

Die Algorithmen für das Simplexverfahren sind für unterschiedliche Optimierungsrichtungen konzipiert. In diesem Abschnitt soll gezeigt werden, wie diese Algorithmen für beide Optimierungsrichtungen funktionieren.

1. Algorithmus für zulässige Tableaus (Algorithmus 2.2.1) liegt für das Minimierungsproblem vor.
2. Zweiphasen–Algorithmus (Algorithmus 2.2.2) liegt für das Minimierungsproblem vor (enthält Algorithmus 2.2.1).
3. Ganzzahliger Algorithmus (Algorithmus 2.2.4) liegt für das Maximierungsproblem vor (enthält den dualen Algorithmus 2.2.3, der für das Maximierungsproblem vorliegt und Algorithmus 2.2.2).

Grundsätzlich gilt das Folgende für eine Zielfunktion der Form:

$$z + a_1x_1 + a_2x_2 + \cdots + a_nx_n = \alpha \tag{2.8}$$

Es gilt für den Maximimierungsalgorithmus:

setze Zielfunktion	Optimierungsrichtung	multipliziere Wert β im Optimum mit
z	Maximum	1
$(-z)$	Minimum	-1

Es gilt für den Minimierungsalgorithmus:

setze Zielfunktion	Optimierungsrichtung	multipliziere Wert im Optimum mit
z	Minimum	1
$(-z)$	Maximum	-1

Für die drei unterschiedlichen Algorithmen mit der Zielfunktion (2.8) sieht das dann folgendermaßen aus:

Algorithmus für zulässige Tableaus und Zweiphasen–Algorithmus:

Optimierungsrichtung	Zielfunktion	Wert im Optimum
Minimum	$z + a_1x_1 + a_2x_2 + \cdots + a_nx_n = \alpha$	β
Maximum	$z - a_1x_1 - a_2x_2 - \cdots - a_nx_n = -\alpha$	$(-1) \cdot \beta$

Ganzzahliger Algorithmus:

1. Optimierungsrichtung: Minimum
   (a) Zu Beginn des Zweiphasen–Algorithmus: Zielfunktion:
   $z + a_1x_1 + a_2x_2 + \cdots + a_nx_n = \alpha$
   (b) Am Ende des Zweiphasen–Algorithmus: Zielfunktion:
   $z - \gamma_1x_1 - \gamma_2x_2 - \cdots - \gamma_nx_n = \beta$

(c) Zu Beginn des ganzzahligen Optimierungsverfahrens: Zielfunktion: $z + \gamma_1 x_1 + \gamma_2 x_2 + \cdots + \gamma_n x_n = -\beta$ damit ist auch gleich die duale Zulässigkeit erbracht.

(d) Am Ende des ganzzahligen Optimierungsverfahrens: Zielfunktion: $z + \delta_1 x_1 + \delta_2 x_2 + \cdots + \delta_n x_n = \mu$ und der richtige Zielfunktionswert ist: $-\mu$

2. Optimierungsrichtung: Maximum

(a) Zu Beginn des Zweiphasen–Algorithmus: Zielfunktion: $z - a_1 x_1 - a_2 x_2 - \cdots - a_n x_n = -\alpha$

(b) Am Ende des Zweiphasen–Algorithmus: Zielfunktion: $z - \gamma_1 x_1 - \gamma_2 x_2 - \cdots - \gamma_n x_n = \beta$

(c) Zu Beginn des ganzzahligen Optimierungsverfahrens: Zielfunktion: $z + \gamma_1 x_1 + \gamma_2 x_2 + \cdots + \gamma_n x_n = -\beta$ damit ist auch gleich die duale Zulässigkeit erbracht.

(d) Am Ende des ganzzahligen Optimierungsverfahrens: Zielfunktion: $z + \delta_1 x_1 + \delta_2 x_2 + \cdots + \delta_n x_n = \mu$ der Zielfunktionswert ist wieder korrekt, da er zweimal negiert wurde.

2.3 Generische Verarbeitung von Steuerungsregeln

Dieses Unterkapitel wird nun die Modellierung konkreter Regelformate zur Konfiguration und Steuerung verteilter Anwendungssysteme sowie deren formale Repräsentation und logische Verarbeitungsfunktionalität behandeln. Angestrebt wird dabei einerseits eine möglichst große Generizität der Regelformate und der entsprechenden Verarbeitungsfunktionen, so dass die darauf basierenden Steuerungsmechanismen für möglichst viele Anwendungsdomänen verwendet werden können. Andererseits muss aber sichergestellt werden, dass unterschiedliche konkrete Anforderungen aus der Praxis mit den hier noch auf der formalen Ebene behandelten Mechanismen später (wie in den nächsten Kapiteln beschrieben) erfüllt werden können. Deshalb wird zunächst eine Taxonomie von Regeltypen, die für die Unterstützung verteilter Anwendungen in der Praxis sinnvoll erscheinen, vorgestellt. Daraus wird dann ein allgemeines Regelkonzept entworfen, für das generische Verarbeitungsfunktionen definiert werden können. Die Realisierung dieser Funktionen hängt von der Ausdrucksmächtigkeit des gewählten Regelformates ab, die wiederum — abhängig von den jeweiligen Anforderungen der Praxis — unterschiedlich ausfallen kann. Deshalb werden in der ersten Stufe Verarbeitungsfunktionen für eine Basismächtigkeit dargestellt, die dann in einer zweiten Stufe für eine höhere Ausdrucksmächtigkeit erweitert werden.

2.3.1 Regeltaxonomie

In vielen Experten- und wissensbasierten Systemen aus dem Gebiet der Künstlichen Intelligenz werden formale Regeln hauptsächlich dazu verwendet, um Fakten über eine bestimmte Domäne zu repräsentieren und zu verarbeiten. Insbesondere dienen Regeln dazu, aus bestehenden Fakten neue zu schließen und

damit neues Wissen zu gewinnen, was vor allem durch so genannte Deduktionssysteme [Bib92] geleistet wird. Probleme, die hierbei auftreten können, betreffen u.a. die Strukturierung und Verwaltung der Wissensbasis, die Kontrolle der Inferenzmechanismen (d.h. welche und wie viele Fakten in der Wissensbasis festgehalten und welche nur abgeleitet werden sollen), die Behandlung von Inkonsistenzen, insbesondere angesichts neu aufgenommener (z.B. empirisch erhobener) Fakten, die den bestehenden widersprechen, etc.

Im Gegensatz zu einer solchen *wissensorientierten* Sicht werden — wie in Abschnitt 2.1.3 skizziert — zur Steuerung verteilter Anwendungssysteme Regeln benötigt, die letztendlich eine Anpassung des Verhaltens der beteiligten Dienste ermöglichen und damit einer eher *aktionsorientierten* Sicht entsprechen. Grundsätzlich muss zur Erbringung eines Steuerungseffekts folgende Kriterien berücksichtigt werden:

1. Zu welchen *Zeitpunkten* kann eine Steuerung erforderlich sein bzw. wann ist zu prüfen, ob eine Steuerung notwendig ist?
2. Unter welchen (semantischen) *Bedingungen* liegt der Bedarf für eine Steuerung tatsächlich vor?
3. *Wodurch* ist der Steuerungseffekt konkret zu erzielen?

Typisch für diese Sicht sind so genannte Aktionsregeln, die die Durchführung einer vordefinierten Aktion auslösen, wenn sie getriggert werden. In dem in Abschnitt 2.1.2.2 erwähnten BOCA–Modell für Business–Objekte wird ein solcher Regeltyp als *Event–Condition–Action* oder *ECA Rule* bezeichnet, da die Regel durch Auftreten eines bestimmten *Ereignis*typs getriggert wird und die Durchführung der Aktion dann nur stattfindet, wenn eine vorgegebene *Bedingung* erfüllt ist. Eine solche Regel erscheint prinzipiell dazu geeignet, einen Steuerungseffekt zu bewirken, denn über den Ereignistyp kann spezifiziert werden, *wann* der eine Steuerung eventuell erforderlich ist, über die Bedingung kann festgestellt werden, *ob* der Bedarf tatsächlich vorhanden ist und über die Aktion kann spezifiziert werden, *wie* der Steuerungseffekt erzielt wird.

Im BOCA–Modell werden ECA–Rules unter die *Event–Condition* oder *EC Rules* subsummiert, die wie folgt definiert werden:

> An Event Condition Rule is an abstraction for any behavior specification that ist enforced based on one or more events. The triggering event is specified as well as other conditions that may have to be satisfied for the rule to fire. ECRule is abstract in that it does not describe any action. Actions are handled by subtypes of ECRule to allow for various types of action declarations. [OMG98a, S. 64]

Das Besondere an der EC Rule ist hier, dass sie als abstrakte Oberklasse dazu dient, den gemeinsamen Anteil von unterschiedlichen konkreten Regeltypen zusammenzufassen. außer den ECA Rules werden zwei weitere konkrete Regeltypen, die *Invariants* und die *State Transition Rules* angegeben. Eine Invariante, wie der Name suggeriert, repräsentiert eine Bedingung über dem Zustand eines Objekts, die nie verletzt werden darf (zumindest wenn die entsprechende Regel getriggert wird). Obwohl eine solche Regel nicht mit einer expliziten Aktionsspezifikation verknüpft ist, ist bei einer konkreten Realisation dennoch eine implizite Aktion durchzuführen:

> A requirement on the state of the object as an expression that must evaluate to true. The implicit action of [an] invariant is that a unit of work (as defined by Schedule) cannot complete in an invalid state (or an exception will be thrown). [OMG98a, S. 64]

Eine State Transition Rule ist mit zwei Zustandsspezifikationen verknüpft und gibt an, unter welchen Bedingungen die Transition von dem einen (Quell-) in den anderen (Ziel-) Zustand stattfinden kann. Sie kann also insbesondere dazu verwendet werden, Zustandsübergänge eines Systems, die durch endliche Automaten modelliert werden können, auf eine sehr *dynamische* Weise zu realisieren, denn erstens können solche Regeln grundsätzlich zur Laufzeit in das Anwendungssystem aufgenommen werden und zweitens kann die Menge der Regeln (die einem (Teil-)Automaten entspricht) schrittweise erweitert werden. Im BOCA–Modell wird die State Transition Rule wie folgt definiert:

> Object types are in one or more defined states at any one time. The StateTransitionRule specifies the conditions under which the type wil go from one state to another. It defines a trigger, a condition, when the rule should fire, the initial state (source) and the final state (target). [OMG98a, S. 64]

Alle drei im BOCA–Modell vorgeschlagenen konkrete Regeltypen können — wie in den später beschriebenen Anwendungsszenarien konkret gezeigt wird — zumindest indirekt dazu verwendet werden, um unterschiedliche Steuerungseffekte zu erzielen. Bei den ECA Rules hängt die Wirkung der Regel davon ab, wie die Aktion spezifiziert wird, die zunächst jede beliebige Semantik realisieren kann. Mit der Invariante wird ein indirekter Steuerungseffekt dadurch erreicht, dass gewisse Aktivitäten *verhindert* werden, nämlich solche, die eine solche Invariante verletzen würden. Mittels State Transition Rules kann ein Anwendungssystem aktiv von einem Zustand in einen anderen versetzt werden, worauf das Verhalten des Systems sich entsprechend ändern kann. Das BOCA–Modell belässt die Semantik dieser Regeltypen jedoch auf dem Abstraktionsniveau der oben zitierten Definitionen, das wesentliche Fragen bzgl. einer konkreten Realisation offen lässt. Insbesondere wird die Frage der Ausdrucksmächtigkeit und damit zusammenhängender Funktionen wie etwa das Vergleichen (der Wirkungen) von zwei Regeln nicht behandelt. Außerdem wird ein ganz wesentlicher (wahrscheinlich sogar der wesentlichste) Aspekt der Anwendung von Regeln auf verteilte Systeme, nämlich das Zusammenspielen von Regeln zur Realisierung von *Kooperationssemantiken*, gar nicht erwähnt.

Zusätzlich zu diesen drei Regeltypen wird im folgenden ein Regeltyp vorgeschlagen, der speziell für die Steuerung von verteilten Kooperationsanwendungen konzipiert ist und daher eine zentrale Rolle in der Realisation der in dieser Arbeit beschriebenen Mechanismen spielt.

2.3.1.1 Das Policy–Konzept

Mit der Entstehung und Ausbreitung *offener* verteilter Anwendungsysteme, die zunehmend globalen Charakter besitzen (s.a. Abschnitt 1.2.1), gewinnt der Begriff *Policy* bzw. *Policy Management* seit einigen Jahren immer mehr an Bedeutung. Der Policy–Begriff kann am besten mit "Richtlinie" übersetzt werden und bezeichnet dementsprechend meist eine Spezifikation, die entweder von den

Beteiligten beachtet oder vom System durchgesetzt werden muss, um eine "vorschriftsmäßige" Dienstnutzung zu gewährleisten. Diese intuitive Bedeutung von Policy macht deutlich, dass mit solchen Richtlinien versucht wird, einen *steuernden* Einfluss entweder direkt auf das Systemverhalten oder auf die mit dem System interagierenden Entitäten zu nehmen.

Allerdings hat dieser Begriff, der in vielen unterschiedlichen Anwendungskontexten und auf verschiedenen Abstraktionsstufen benutzt wird, keine präzise allgemeine Bedeutung und wird zudem häufig informal verwendet. Die am meisten *technisch–formale* Verwendung des Policy–Begriffes stammt ursprünglich aus dem Gebiet System- und Netzwerkmanagement, zu dem auch die meisten Ansätze in der Literatur zuzuordnen sind (siehe [Wie95b] für einen umfassenden Überblick). Charakteristisch für diese Sicht von Policies sind die Arbeiten von Moffett und Sloman, [MS93, Mof94, MMSS94, ST94, Slo94b], die Pionierarbeit auf dem Gebiet Policy–Management (für den Netzwerk- und Systemmanagementbereich) geleistet haben. In [Mof94] werden Policies als Objekte definiert, die folgende Attribute besitzen:

Modalität Die Modalität gibt an, ob es sich um eine *Autorisierung* oder eine *Motivation* handelt. Im ersten Fall dient die Policy dazu, dem Policy–Objekt ein bestimmtes Verhalten, z.B. die Ausführung einer Aktion, zu erlauben oder zu verbieten. Im letzten Fall veranlasst die Policy das Policy–Subjekt (bzw. *motiviert* es) dazu, eine Aktion an dem Policy–Objekt durchzuführen.

Policy–Subjekt Das Policy–Subjekt ist ein Objekt oder eine Menge von Objekten, die für die Durchsetzung der Policy verantwortlich bzw. bestimmt sind.

Policy–Objekt Das Policy–Objekt ist ein Objekt oder eine Menge von Objekten, an die sich die Policy richtet bzw. an den die Policy durchzusetzen ist.

Ziel Das Ziel einer Policy ist entweder ein höheres (abstraktes) Ziel (*high–level goal*) oder eine Prozedur, d.h. eine Sequenz von *Aktionen*.

Einschränkung Die Einschränkung repräsentiert Bedingungen über die Anwendbarkeit der Policy, z.B. die zeitliche Dauer oder Intervalle, in denen die Policy Gültigkeit besitzt.

Bei näherer Betrachtung der Arbeiten von Moffett und Sloman stellt sich jedoch heraus, dass ihre Modellierung von Policies — wie übrigens die meisten Ansätze im Systemmanagementbereich — keineswegs durchgehend formal, sondern höchstens als *semi–formal* zu bezeichnen ist. Z.B. sind als "höhere Ziele" durchaus auch textuelle Beschreibungen zugelassen, die nur von einem menschlichen Administrator als Policy–Subjekt interpretiert werden können. Darüber hinaus sind im Hinblick auf eine vollautomatische Verarbeitung der Policies mehrere Probleme mit einer solchen semi–formalen Modellierung verbunden. Ein erstes Problem rührt z.B. daher, dass Moffett keine Unterscheidung macht zwischen einem Ziel und der Aktion bzw. Aktionsfolge, die zu einem solchen Ziel führt. Als Beispiele für Policy–Ziele gibt er "recover from media failure" bzw. "run the BackUp job on department D's disk files" an [Mof94, S. 460], die offensichtlich Aktionen denotieren. Dies bringt einige Schwierigkeiten mit

sich: Erstens kann es der Fall sein, daß zum Zeitpunkt der Aktivierung der Policy die Ziele, welche durch die Aktionen erreicht werden sollen, bereits erfüllt sind, dann ist das System bereits mit der Policy konform und es müssen gar keine Aktionen durchgeführt werden. Zweitens gibt es Policies, bei den nur geprüft werden soll, ob sie von den entsprechenden Objekten eingehalten werden. Z.B. kann eine Policy verlangen, dass während einer Kommunikationsverbindung die Paketgröße genau 1024 Bytes sein muss. Die Aktivierung dieser Policy bedeutet dann nicht die Ausführung einer Aktion, um alle gesendeten Pakete auf die richtige Größe zu bringen, was in vielen Fällen auch unmöglich wäre, sondern die Überprüfung, ob die gesendeten Pakete die vorgeschriebene Größe besitzen. Solche Policies können als *passiv* bezeichnet werden, im Gegensatz zu den *aktiven* Policies, bei denen Aktionen ausgeführt werden, um aktiv das Ziel zu erreichen. (Passive Policies können im Sinne des o.g. BOCA–Modells einfach als Invarianten formalisiert werden.) Ein zweites Problem betrifft die Modalität von Policies. Es scheint, daß Autorisierung als eigene Modalität überflüssig ist, denn die Autorisierung eines Subjekts für bestimmte Operationen an einem Objekt kann durchaus als ein Ziel einer Policy formuliert werden, z.B. mit dem Prädikat *authorized(subject, object, operations)*. Autorisierungs– und Motivations–Policies können also formal gleich behandelt werden. Vielmehr scheint die genannte Unterscheidung zwischen passiven und aktiven Policies als eigene Modalitäten sinnvoll, da verschiedene Ausführungssemantiken vorliegen. Dies führt uns dazu, eine davon abweichende Policy–Struktur zu verwenden, die im wesentlichen auf der strikten Trennung von Ziel und Aktion beruht: Bei der Aktivierung einer Policy wird geprüft, ob das Ziel bereits erfüllt ist. Falls dies *nicht* der Fall ist, wird die spezifizierte Aktion (die zum Ziel führen soll) ausgeführt. Im Hinblick auf eine automatische Steuerung stellt sich dann die Frage nach geeigneten *generischen* Aktionstypen, die gewissermaßen aus dem Ziel abgeleitet werden können. Diese Frage sowie die genaue Formalisierung von Policies als Regeltyp werden im nächsten Abschnitt behandelt.

Der Interaktionsaspekt Da bei der Verwendung von Policies immer ein *explizites* Ziel im Vordergrund steht, eignet sich das Policy–Konzept besonders gut zur *aktiven, zielgerichteten* Steuerung eines Systems. Eine wesentliche Frage, die sich im Zusammenhang mit verteilten Anwendungssystemen — insbesondere solchen, die ein dynamisches interaktives (sowohl *kooperatives* als auch *kompetitives*) Verhalten aufweisen — stellt, ist, wie unterschiedliche Policies (die sich in den unterschiedlichen Zielen ausdrücken) einzelner Anwendungskomponenten, die miteinander interagieren wollen, aufeinander *abgestimmt* werden können. Beispielsweise kann es für eine Handelstransaktion zwischen zwei Parteien der Fall sein, dass die eine ein *Public Key* Verschlüsselungsverfahren mit einer Schlüssellänge von mindestens 32 Bits, während die andere Partei ein beliebiges Verschlüsselungsverfahren mit einem mindestens 64 Bits langen Schlüssel fordert. Solche Anforderungen können in Form von *Interaktionspolicies* festgelegt werden, jedoch muss zur Durchführung der entsprechenden Transaktion eine *gemeinsame Basis* gefunden werden, die die Policies beider Parteien erfüllt und (während der Transaktion) auf beiden Seiten entsprechend aktiviert wird.

Dieser Interaktionsaspekt, der in Abbildung 2.2 illustriert ist, wird bei der in den folgenden Abschnitten dargestellten generischen Regelverarbeitung eine zentrale Rolle spielen.

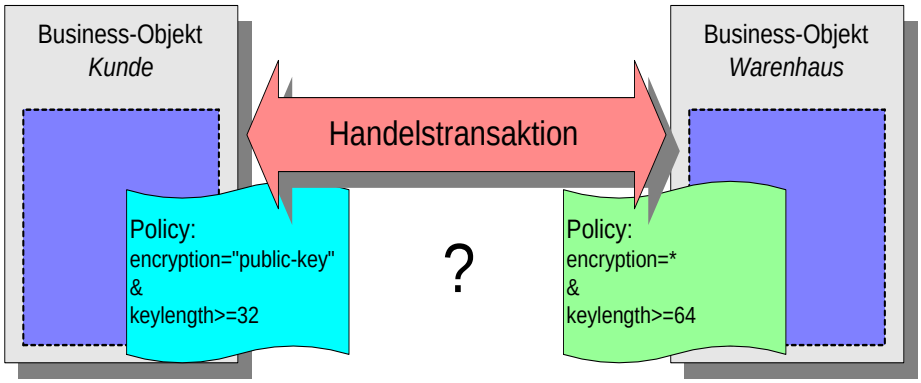


Abbildung 2.2: Problem der Abstimmung unterschiedlicher Interaktionspolicies

2.3.1.2 Allgemeines Regelkonzept

Die vorangegangene Beschreibung der BOCA–Regeltypen und des Policy–Konzepts lässt erkennen, dass zur Steuerung verteilter Anwendungssysteme viele verschiedene Regeltypen denkbar sind, die jeweils unterschiedliche Anforderungen in der Praxis erfüllen können. All den bisher vorgestellten Regelkonzepten ist jedoch gemeinsam, dass sie jeweils einen "deklarativen" Anteil, der eine Bedingung bzw. ein Ziel zum Ausdruck bringt, und einen "prozeduralen", der die zu erbringende Wirkung der Regel beschreibt, besitzen. Es scheint also sinnvoll zu sein, die Gemeinsamkeit dieser Regelkonzepte durch eine allgemeinere Abstraktion zu repräsentieren, um eine möglichst generische, formal korrekte Verarbeitung, die auf alle konkreten (von diesem allgemeinen Typ abgeleiteten) Regeltypen anwendbar sind, zu ermöglichen.

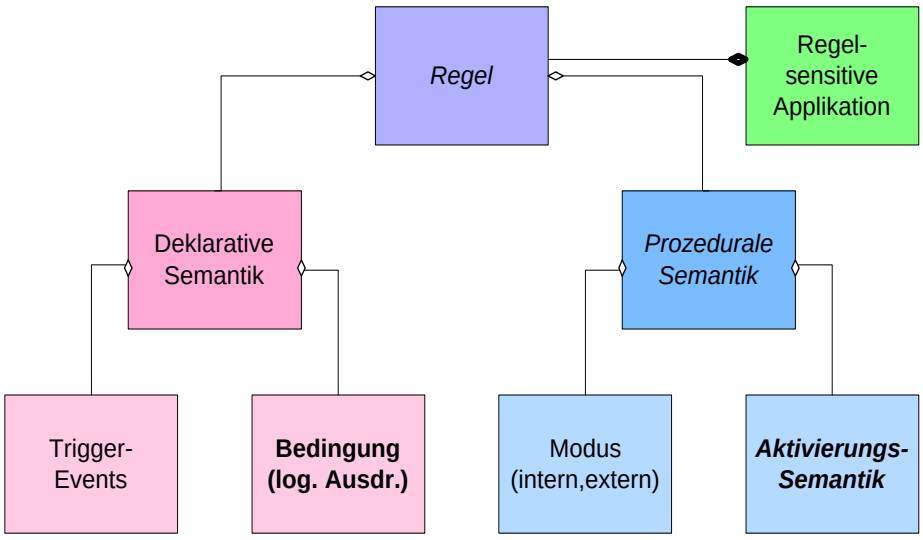


Abbildung 2.3: Grundbestandteile des allgemeinen Regelkonzepts

Die Grundbestandteile eines solchen Regelkonzepts sind in Abbildung 2.3

dargestellt. (Gemäß der UML–Notation sind hierbei abstrakte Konzeptnamen kursiv gesetzt.) Demnach ist eine Regel kein eigenständiges Objekt, dessen Funktionalität direkt genutzt werden kann, sondern immer mit einer *regel–sensitiven* Applikation assoziiert.[9] Die Spezifikation der Regel besteht aus einer deklarativen und einer prozeduralen Semantik. Der *deklarative* Anteil besteht mindestens aus einer Liste der Typen von *Trigger–Events*, welche die Aktivierung der Regel auslösen können, und einem logischen Ausdruck, der eine Bedingung (bzw. ein Ziel) ausdrückt. Abhängig vom konkreten Regeltyp kann der deklarative Anteil um weitere Spezifikationen, wie z.B. einer explitziten Aktionsbeschreibung oder der Quell- und Zielzustände (bei einer Zustandstransitionsregel), erweitert werden. Der *prozedurale* Anteil der Regelsemantik legt fest, wie die Wirkung der Regel zu erbringen ist. Hierbei kann im allgemeinen zwischen zwei Aktivierungs*modi* unterschieden werden. Bei der *internen* Aktivierung richtet sich die Wirkung der Regel direkt an die regel–sensitive Applikation, mit der sie assoziiert ist. Bei der *externen* Aktivierung dagegen richtet sich die Wirkung der Regel an eine externe Komponente, mit der die die Regel beinhaltende Applikation in einer Interaktionsbeziehung steht. Bei dem in Abbildung 2.2 illustrierten Beispiel handelt es sich also um zwei externe Policies, da beide jeweils an die andere Seite (im Sinne einer Forderung an die gemeinsame Transaktion) gerichtet sind. Die konkrete Ausarbeitung und Realisierung verschiedener Aktivierungsmodi wird in Kapitel 3.3 ausführlich behandelt. Der zweite Bestandteil der prozeduralen Semantik einer Regel ist die Aktivierungssemantik zur Festlegung der funktionalen Wirkung der Regel. Da dies jeweils vom konkreten Regeltyp abhängt, ist die Aktivierungssemantik bei diesem allgemeinen Regelkonzept als ein *abstrakter* Bestandteil gekennzeichnet, der von jedem abgeleiteten Typ durch eine konkrete Semantik zu ersetzen ist. Dadurch wird das gesamte allgemeine Konzept ein abstraktes.

2.3.1.3 Konkrete Regeltypen

Aus diesem allgemeinen Regelkonzept lassen sich nun die vier oben eingeführten konkreten Regeltypen ableiten, die sich hauptsächlich in der Aktivierungssemantik unterscheiden.

Requirement–Rule Die Requirement–Rule repräsentiert eine Bedingung, die erfüllt sein muss, wenn diese Regel durch ein entsprechendes Event getriggert wird. Dieser Regeltyp kann als eine Konkretisierung des erwähnten Konzepts der Invariante betrachtet werden, die allerdings auf einem *diskreten* Zeitkonzept basiert, da die Bedingung nur zu Zeitpunkten des Auftretens der triggernden Events geprüft wird. Daher scheint die Bezeichnung *Requirement* angemessener zu sein als *Invariant*. Wie in den Kapiteln 3 und 4 gezeigt werden wird, können Requirement Rules insbesondere zu einer Realisation von Vor- und Nachbedingungen (siehe Abschnitt 2.1.2.2), verwendet werden, die jedoch von der Methodenspezifikation *entkoppelt* ist und damit mehr Flexibilität bietet als herkömmliche Ansätze im Bereich Software Engineering. Die Aktivierungssemantik der Requirement–Rule ist in Abbildung 2.4 dargestellt.

Action–Rule Eine Action–Rule beschreibt eine (beliebige) Aktion, die auszuführen ist, wenn beim Auftreten eines triggernden Events die Bedingung

[9] Analog zur der BOCA–Sicht von Regeln als *appliances*.

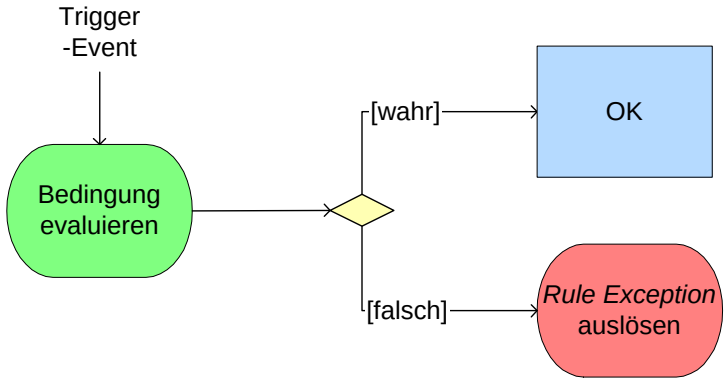


Abbildung 2.4: Aktivierungssemantik des Regeltyps Requirement–Rule

der Regel zu *wahr* evaluiert (siehe Abbildung 2.5. Diese Aktivierungssemantik setzt voraus, dass eine komplette Spezifikation der Aktion im deklarativen Teil der Regel vorhanden ist und dass diese Aktion von dem Regelsystem bzw. von dem Regelobjekt selbst auch ausgeführt werden kann. Unter der zusätzlichen Annahme, dass die Regeln zur Laufzeit mit einer regel–sensitiven Applikation assoziiert werden können, eignen sich Action–Rules nicht nur für Steuerungszwecke, sondern im allgemeinen zur *funktionalen* Erweiterung einer Anwendung. In der Tat stellen solche aktiven Regeln eine sehr dynamische Art der Funktionserweiterung dar, da sie von der eigentlichen Applikationssemantik entkoppelt ist und damit *außerhalb* des Hauptkontrollflusses der Anwendung liegt. In den folgenden Kapiteln und insbesondere in Kapitel 6.4 werden konkrete Möglichkeiten von Regeln zur funktionalen Erweiterung verteilter Softwaresysteme aufgezeigt.

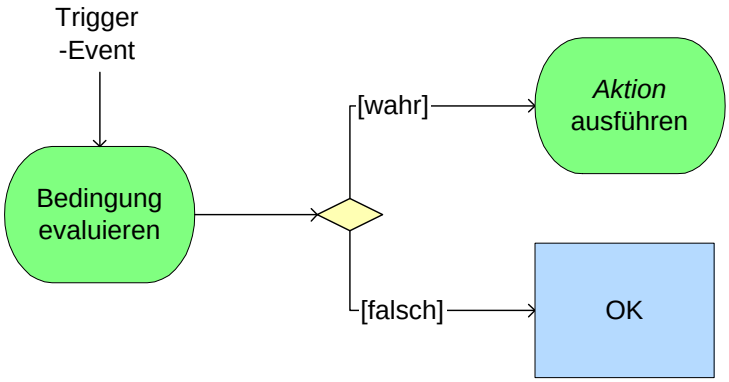


Abbildung 2.5: Aktivierungssemantik des Regeltyps Action–Rule

State–Transition–Rule Die State–Transition–Rule repräsentiert das Konzept des Wechsels von einem Zustand in einen anderen. Jede State–Transition–Rule beinhaltet zwei Zustandsbeschreibungen (die im deklarativen Teil enthalten sind) und die Transition von dem ersten in den zweiten findet

genau dann statt, wenn beim Auftreten eines triggernden Events die Bedingung zu *wahr* evaluiert und der erste Zustand als der *aktuelle* markiert ist (siehe Abbildung 2.6). Insbesondere eignet sich dieser Regeltyp zur Realisierung der in Abschnitt 2.1.2.2 erwähnten Protokollspezifikationen bzw. Automatenmodelle, wobei ein solcher endlicher Automat als "aufgespalten" in eine Menge einzelner State–Transition–Rules, die für sich selbst schalten können, zu betrachten ist.

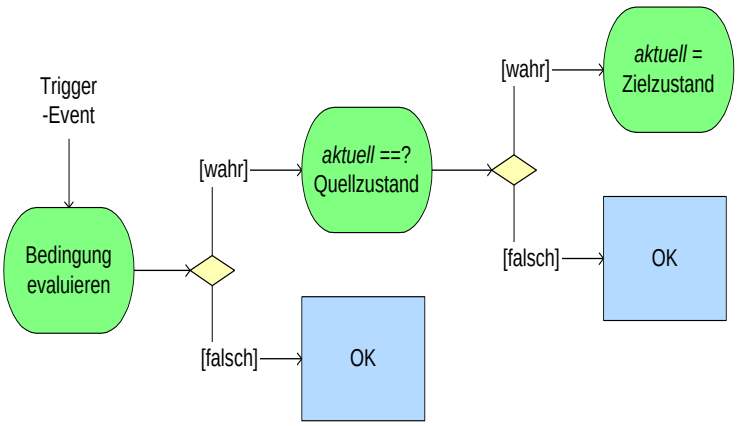


Abbildung 2.6: Aktivierungssemantik des Regeltyps State–Transition–Rule

Policy–Rule Eine Policy, wie in Abschnitt 2.3.1.1 dargestellt, repräsentiert ein Ziel, das durch die Ausführung einer vorgegebenen Aktion aktiv erreicht bzw. durchgesetzt werden kann. Wenn also beim Auftreten des triggernden Events die Bedingung, die hier als Ziel zu betrachten ist, zu *falsch* evaluiert, wird die vorgegebene Aktion ausgeführt. Danach wird die Bedingung erneut evaluiert, und wenn das Evaluationsergebnis weiterhin negativ ist, dann wird eine Rule–Exception ausgelöst (siehe Abbildung 2.7). In Abschnitt 2.3.2 wird ein generischer Aktionstyp vorgestellt, der genau für den Zweck der Durchsetzung von Policies konzipiert ist.

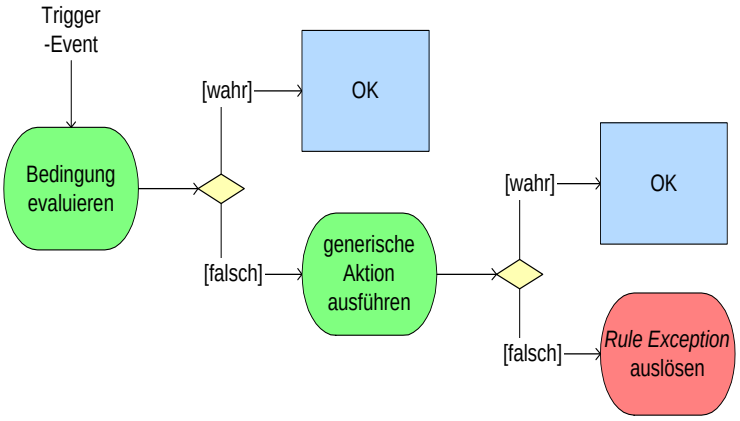


Abbildung 2.7: Aktivierungssemantik des Regeltyps Policy–Rule

2.3.1.4 Basisausdrucksmächtigkeit

Sowohl aus konzeptioneller als auch anwendungstechnischer Sicht stellt die logische Bedingung in einer Regel ihren wichtigsten Bestandteil dar, da sie letztendlich die Aktivierung der Regel bestimmt. Der Bedingungsteil bildet auch die Basis zum Vergleichen der Wirkung mehrerer Regeln des gleichen Typs. In der Praxis ist der Umfang der mit dem Bedingungsteil ausdrückbaren Sachverhalte entscheidend für die Anwendbarkeit der im vorangegangenen Abschnitt dargestellten Regeltypen. Aus anwendungsfunktionaler Sicht ist also eine möglichst große Ausdrucksmächtigkeit erwünscht. Aus verarbeitungstechnischer Sicht kann jedoch eine große Ausdrucksmächtigkeit zu einer Komplexität führen, die entweder überhaupt nicht — siehe Entscheidbarkeitsprobleme in Abschnitt 2.2.1.4 — oder nicht in vertretbarer Zeit bewältigt werden kann. Deshalb gilt es bei der konkreten Realisation regelbasierter Steuerungsmechanismen, eine Ausdrucksmächtigkeit zu finden, die einerseits genügend viele praktischen Anforderungen erfüllen kann, andererseits aber auch eine effektive Verarbeitung zulässt. Darüber hinaus erscheint eine Differenzierung unterschiedlich komplexer Ausdrucksmächtigkeitsklassen, die unterschiedlich effizient verarbeitet werden können, sinnvoll.

In dieser Arbeit wird deshalb zunächst eine Basisausdrucksmächtigkeit für den Bedingungsteil von Regeln vorgeschlagen, die relativ einfache und effiziente Verarbeitungsalgorithmen ermöglicht. In einer weiteren Stufe wird dann eine höhere Ausdrucksmächtigkeitsklasse behandelt. In der Basisstufe ist der Bedingungsteil einer Regel wie folgt spezifiziert:

Definition 2.3.1 (Basisausdrucksmächtigkeit des Bedingungsteils)

Bedingung *Eine* Bedingung *ist entweder eine atomare Bedingung oder eine logische Kombination von atomaren Bedingungen mit den Operatoren* $\wedge$ *(oder AND),* $\vee$ *(oder OR) und* $\neg$ *(oder NOT), wobei der Negationsoperator nur für atomare Bedingungen zugelassen ist.*

Atomare Bedingung *Eine atomare Bedingung ist eine geordnete, binäre Relation zwischen einem Eigenschaftswert und einem Literal oder die postulierte Existenz einer Eigenschaft. Auch die beiden Konstanten* $\top$ *(oder* TRUE*) bzw.* $\bot$ *(oder* FALSE*) sind atomare Bedingungen.*

Eigenschaft *Eine Eigenschaft ist ein Name–Wert–Paar, wobei der Name eine feste Zeichenkette ist und der Wert durch ein typisiertes Konstantensymbol dargestellt wird.*

Beispiel 2.3.1
Atomare Bedingungen sind z.B.:

1. *total_cost* < 90 oder
2. *customer_name* $=$ *"Hans Müller"*

Nicht–atomare Bedingungen sind z.B.:

3. *total_cost* < 90 *AND customer_name* $\neq$ *"Hans Müller"*

In der Basisausdrucksmächtigkeit der Regeln handelt es sich also um eine deutlich eingeschränkte Prädikatenlogik, mit der logische Bedingungen über *Eigenschaften* bzw. deren Werten ausgedrückt werden können. Gegenüber der Syntax der allgemeinen Prädikatenlogik, die in Abschnitt 2.2.1.2 vorgestellt wurde, sind folgende Einschränkungen vorhanden:

- Anwendung der Quantoren: Der Allquantor ist nicht zugelassen. Genau genommen ist auch der Existenzquantor nicht zugelassen, denn er kann nicht zur Quantifikation von Variablen verwendet werden. Logisch gesehen ist die postulierte Existenz einer Eigenschaft lediglich die entsprechende Prädikation eines Konstantensymbols, das diese Eigenschaft kennzeichnet, z.B. durch die Formel EXIST(total_cost).

- Anwendung der Junktoren: Der Negationsjunktor ist nur auf Ebene der atomaren Bedingungen zugelassen. Durch diese Einschränkung kann eine erhebliche Reduzierung der Komplexität der Verarbeitungsalgorithmen erreicht werden. (Siehe [Hor89] für eine ausführliche Behandlung des Negationsthemas.)

- Funktionen sind nicht zugelassen.

- Es sind nur eine vorgegebene Menge von binären Relationen zwischen jeweils einem Eigenschaftswert und einem Literal (d.h. Konstantensymbol) zugelassen. Damit ist es z.B. nicht möglich, Relationen zwischen den Werten zweier Eigenschaften *direkt* auszudrücken.

Die präzise Syntax der zugelassenen Ausdrücke ist in Anhang A.2.2 zu finden. Trotz dieser Einschränkungen bietet diese logische Sprache bereits ein für viele praktischen Anforderungen in elektronischen Dienstemärkten (siehe die Beispiele in Abschnitt 4.4) ausreichend mächtiges und vor allem sehr flexibles Mittel zur Formulierung von Bedingungen über Eigenschaften und damit auch über das Verhalten von beliebigen Diensten, die entweder eine unabhängige Funktionalität anbieten oder miteinander in Interaktion treten, insbesondere in Verbindung mit einem entsprechenden *Typsystem* zur Definition von unterschiedlichen numerischen und textuellen Eigenschaften sowie den auf ihnen zugelassenen Relationen, das in Abschnitt 3.2 beschrieben wird.

2.3.2 Generische Verarbeitungsfunktionen

Im folgenden werden logische Funktionen zur Verarbeitung beliebiger, zur Definition 2.3.1 konformer Ausdrücke eingeführt. Die allgemeine Anwendbarkeit dieser Funktionen auf alle definierten konkreten Regeltypen ermöglicht eine generische, d.h. von spezifischen Anforderungen unabhängige Aktivierung der Regeln, von der im Prinzip beliebige Anwendungssysteme Gebrauch machen können, um eine dynamische dezentrale Steuerung zu realisieren.

Die Mechanismen zur Implementation dieser Funktionen setzen zusätzlich zu den in Abschnitt 2.2.1.3 definierten spezielle Normalformen voraus, die nun eingeführt werden sollen. Zunächst einmal ist es häufig sinnvoll zu fordern, dass die Bedingung einer Regel in disjunktiver Normalform (DNF, Definition 2.2.8) vorliegt. Über verarbeitungstechnische Aspekte hinaus hat diese Darstellung

den konzeptuellen Vorteil, dass die Bedingung als eine Verknüpfung von *Alternativen* aufgefasst werden kann. Z.B. kann ein Kunde von einem Computerlieferanten fordern, dass der CPU–Takt mindestens 300 Mhz und die Festplatte mindestens 10 GB groß sein soll *oder* dass der CPU–Takt mindestens 300 Mhz und der Hauptspeicher 128 MB betragen soll. Solche (auf der obersten Ebene) als Alternativen formulierte Bedingungen sind in der umgangssprachlichen Praxis die Regel, dagegen wäre eine umgangssprachliche Entsprechung einer KNF–Bedingung nur schwer zu verstehen. Darüber hinaus erscheint es meist sehr sinnvoll zu fordern, dass es sich bei einer solchen Bedingung um *echte* Alternativen handelt, d.h. dass zwischen den Teilbedingungen keine Implikationsbeziehung besteht. In Bezug auf den genannten Kunden wäre es z.B. nicht sinnvoll, wenn die zweite Teilbedingung "CPU–Takt mindestens 300 Mhz und die Festplatte mindestens 20 GB groß" lautete, denn jeder (logisch denkende) Verkäufer würde die gesamte Bedingung zu seiner Gunsten auf die erste Hälfte (die von dieser impliziert wird, aber schwächer und damit leichter zu erfüllen ist) reduzieren. Die Forderung nach echten Alternativen in der Bedingung wird durch folgende Definition einer "schlanken" DNF ausgedrückt:

Definition 2.3.2 (Schwache LDNF)
Eine Formel $P = (c_1 \vee c_2 \vee c_3 \vee \ldots \vee c_n)$ *in disjunktiver Normalform heißt in schwacher LDNF (Lean Disjunctive Normal Form), wenn*
$\forall (1 \leq j \leq n) : \nexists (1 \leq i \leq n; i \neq j) : (c_i \rightarrow c_j)$

Obwohl Definition 2.3.2 echte Alternativen innerhalb des Bedingungsteils gewährleistet, ist sie in einer Hinsicht nicht wirklich als "schlank" zu bezeichnen: Sie lässt immer noch Ausdrücke zu, die bzgl. der Anzahl der Disjunkte zu kürzeren zusammengefasst werden können, wie z.B.

Beispiel 2.3.2 (Bedingung in schwacher LDNF)
$[(length \geq 100) \wedge (length \leq 120)] \vee [(length \geq 120) \wedge (length \leq 200)]$

die sich verkürzen lässt zu

Beispiel 2.3.3 (Bedingung in starker LDNF)
$(length \geq 100) \wedge (length \leq 200)$

Um schlanke Ausdrücke im Sinne von Beispiel 2.3.3 zu erhalten, muss Definition 2.3.2 durch folgende Variante ersetzt werden:

Definition 2.3.3 (Starke LDNF)
Eine Formel $P = (c_1 \vee c_2 \vee c_3 \vee \ldots \vee c_n)$ *in disjunktiver Normalform heißt in starker LDNF, wenn*
$\forall (1 \leq i, j \leq n; i \neq j) : \nexists c = (a_1 \wedge a_2 \wedge a_3 \wedge \ldots a_m) : (c \rightarrow c_i \vee c_j)$

Es ist leicht zu sehen, dass die starke Variante einer LDNF–Bedingung die schwache impliziert. Allerdings ist es ebenfalls leicht erkennbar, dass der Nachweis der starken Variante, der erfordert, dass Ausdrücke wie in Beispiel 2.3.2 auf Intervallabdeckung geprüft werden müssen, wesentlich schwieriger und teurer sein könnte. In praktischen Szenarien, wo derartige Beispiele anwendungssemantisch ausgeschlossen werden können, kann man deshalb mit Definition 2.3.2 ein effizienteres Systemverhalten erzielen.

Schließlich kann in Bezug auf semantisch möglichst kompakte Bedingungsausdrücke, die nicht nur eine effiziente (Weiter–)Verarbeitung erleichtern, sondern auch die Lesbarkeit der Ergebnisausdrücke erheblich verbessern, außer der Definition 2.3.3 noch gefordert werden, dass ebenso Querbezüge auf der Ebene der Konjunktionen ausgeschlossen werden. Die Definition wird zu einer *minimalen DNF* erweitert.

Definition 2.3.4 (MDNF)
Eine Formel $P = (c_1 \vee c_2 \vee c_3 \vee \ldots \vee c_n)$ *in schlanker disjunktiver Normalform heißt in MDNF (Minimal Disjunctive Normal Form), wenn*
$\forall c_i(1 \leq i \leq n) = (a_1 \wedge a_2 \wedge a_3 \wedge \ldots a_m) : [\forall(1 \leq j, k \leq m; j \neq k) : \nexists a : (a \rightarrow a_j \wedge a_k)]$
wobei a eine atomare Bedingung ist.

2.3.2.1 Evaluierung

Die am häufigsten benötigte Funktion zur Aktivierung von Regeln ist die Evaluierung bzw. die Auswertung des Bedingungsteils. In Bezug auf die o.g. Ausdrucksmächtigkeit ist die Auswertung eine relativ einfache Operation. Unter der Voraussetzung, dass ein Zugriff auf die Werte der Eigenschaften gegeben ist, kann einfach rekursiv der Ausdruck traversiert werden. Bei der OR–Verknüpfung kann dabei, wenn ein Teilausdruck bereits als wahr anerkannt wurde, auf die Auswertung des anderen Teiles verzichtet werden. Für die AND–Verknüpfung braucht der zweite Teil im Falle eines als nicht erfüllt erkannten Teilausdrucks nicht überprüft zu werden.

Für die Auswertung ist es damit auch nicht notwendig, die verwendeten Ausdrücke in bestimmte Normalformen zu überführen, es sei denn, man sucht nach weiteren Optimierungsmöglichkeiten für laufzeitintensive Anwendungen und greift daher wieder auf die eingeführten Normalformen zurück.

Zu einer sinnvollen und benutzerfreundlichen Evaluierungsfunktion gehört jedoch auch die Erkennung von Formeln, deren Wahrheitswerte nicht von einer konkreten Belegung der in ihnen vorkommenden Eigenschaften abhängen, sondern lediglich von ihrer logischen Form. Dazu gehören so genannte *Inkonsistenzen* und *Tautologien*. Und bei der Erkennung dieser Formeln wird sehr wohl Gebrauch von Normalformen als Zwischenformaten gemacht.

Zunächst sind Inkonsistenzen bzw. unerfüllbare Formeln solche, die auf die Konstante $\perp$ (*falsch*) reduziert werden können. Es gilt:

Lemma 2.3.1 (Inkonsistente DNF–Formel)
Eine Formel $F = (c_1 \vee c_2 \vee c_3 \vee \ldots \vee c_n)$ *in disjunktiver Normalform ist (nur dann) unerfüllbar, wenn*
$\forall(1 \leq i \leq n) : c_i \equiv \perp$

Eine solches c_i, das eine Konjunktion von atomaren Bedingungen ist, ist genau dann unerfüllbar, wenn es mindestens zwei Konjunkte enthält, die im Widerspruch zueinander stehen:

Lemma 2.3.2 (Inkonsistente Konjunktion)
Eine Konjunktion $c = (a_1 \wedge a_2 \wedge a_3 \wedge \ldots a_n)$ *ist unerfüllbar, wenn gilt*
$\exists(1 \leq i, j \leq n; i \neq j) : a_i \wedge a_j \equiv \perp$

Auf dieser Ebene muss also nur noch die Unvereinbarkeit von jeweils zwei atomaren Bedingungen festgestellt werden. Für gängige mathematische Relationen, die auf Zahlenmengen angewendet werden, kann dies durch Vergleichen der Literale in den beiden Bedingungen gelöst werden. Z.B. ist $(a > 7) \wedge (a < 5)$ offensichtlich ein Widerspruch, weil $d_v \leq c_v$ gilt, wobei $d_v = 5$ und $c_v = 7$ ist. Mit Hilfe einer Entscheidungstabelle, die in Anhang A.1 angegeben ist, kann dieses Testverfahren für die entsprechenden Relationen generalisiert werden.

Tautologien sind dagegen Formeln, die auf die Konstante $\top$ (*wahr*) reduziert werden können. Die Erkennung von Tautologien basiert auf der Äquivalenz

$$A \vee \neg A \equiv \top$$

D.h. es gilt:

Lemma 2.3.3 (Tautologische DNF–Formel)
Eine Formel $F = (c_1 \vee c_2 \vee c_3 \vee \ldots \vee c_n)$ *in disjunktiver Normalform ist eine Tautologie, wenn* $\exists(1 \leq i, j \leq n; i \neq j) : c_i \equiv \neg c_j$

Auf dieser Ebene ist also das Testen auf Äquivalenz erforderlich, das im nächsten Abschnitt dargestellt wird.

2.3.2.2 Vergleich

Einer der größten Vorteile einer logikbasierten Konzeption von Regeln — im Gegensatz etwa zu skript- oder auf anderen Pseudoprogrammiersprachen basierten Ansätzen — ist ihre formale Vergleichbarkeit. Besonders im Kontext verteilter Anwendungen, die in einer offenen Umgebung miteinander interagieren, ist es in vielen Fällen erforderlich, die fremden Anforderungen bzgl. einer Kooperation mit den eigenen zu vergleichen. Z.B. kann eine solche Anforderung, die explizit als Kooperationsregel an der Schnittstelle eines Dienstes sichtbar gemacht wird, häufig als ein "Angebot" aufgefasst werden, das nur von der Dienstnutzenden Anwendung akzeptiert wird, wenn es logisch stärker und damit "besser" als die eigene Anforderung ist.

Grundlage für logische Vergleichsfunktionen wie "stärker", "schwächer", "gleichwertig" (od. äquivalent) ist die logische *Implikation*. Um Implikationsbeziehungen zwischen Ausdrücken der Basismächtigkeit zu erkennen, wird folgender Satz verwendet:

Satz 2.3.4 (Implikation von Formeln in LDNF)
Für zwei Formeln $F = (f_1 \vee f_2 \vee f_3 \vee \ldots \vee f_n)$ *und* $G = (g_1 \vee g_2 \vee g_3 \vee \ldots \vee g_m)$ *jeweils in schlanker disjunktiver Normalform (LDNF) gilt*
$F \rightarrow G$, *wenn* $\forall(f_{1 \leq i \leq n}) : \exists(g_{1 \leq j \leq m}) : f_i \rightarrow g_j$

D.h., es ist lediglich nötig zu prüfen, ob jedes einzelne Disjunkt von F irgendein Disjunkt von G impliziert. Allerdings gilt diese Vereinfachung im allgemeinen nur unter der Voraussetzung der starken Variante der schlanken disjunktiven Normalform (siehe Def. 2.3.3), wie folgende Beispiele zeigen:

Beispiel 2.3.4 (Implikation von nicht schlanken Formeln)
$F := x > 0 \wedge x < 5$
$G := (x > 0 \wedge x < 3) \vee (x \geq 3 \wedge x < 5)$
$H := (x = 1) \vee (x = 2) \vee (x = 3) \vee (x = 4)$

Obwohl F und G offensichtlich äquivalent sind, gibt es kein Disjunkt g_j von G, so dass $F \rightarrow g_j$ gilt. Die dargestellte Intervallaufteilung kann natürlich in beliebig viele Teilintervalle erfolgen und für ganzzahlige x gilt sogar $F \equiv H$. Nach Definition 2.3.3 sind aber weder G noch H — unter Berücksichtigung der jeweiligen Domäne — schlanke Formeln, denn sie lassen sich zu Formeln mit weniger Disjunkten reduzieren.

Damit ist die Implikationsbeziehung zwischen LDNF–Formeln auf die Implikation zwischen Konjunktionen von atomaren Ausdrücken zurückgeführt. Auf dieser Ebene lässt sich folgendes Lemma anwenden:

Lemma 2.3.5 (Implikation von Konjunktionen atomarer Ausdrücke)
Für zwei Konjunktionen $K = (k_1 \wedge k_2 \wedge k_3 \wedge \ldots \wedge k_n)$ *und* $L = (l_1 \wedge l_2 \wedge l_3 \wedge \ldots \wedge l_m)$
jeweils aus atomaren Ausdrücken k_i *und* l_i *gilt*
$K \rightarrow L$, *wenn* $\forall(l_{1 \leq j \leq m}) : \exists(k_{1 \leq i \leq n}) : k_i \rightarrow l_j$

Das heißt die Implikation zwischen Konjunktionen wird weiter auf die zwischen atomaren Ausdrücken heruntergebrochen. Für Implikationsbeziehungen zwischen Atomen lassen sich nun Entscheidungstabellen aufstellen, die auf Vergleichen der in den den Atomen vorkommenden Literale, deren Bezeichner jeweils mit dem Index 'v', z.B. a_v, b_v, c_v etc., versehen werden, bezüglich einer Menge vordefinierter Relationen basieren. Beispielsweise gilt für:

Beispiel 2.3.5 (Implikation zwischen atomaren Ausdrücken)
$A := x = a_v$
$B := x > b_v$
$A \rightarrow B$, *wenn* $b_v < a_v$

Solche Entscheidungstabellen werden in Anhang A.1 gegeben. Das Beispiel 2.3.5 lässt sich dort in der Zeile '$\lhd$' = '=' und Spalte '$\rhd$' = '>' wiederfinden. In diesen Tabellen wird auch eine Fallunterscheidung bzgl. der Behandlung von zwischen reellwertigen (bzw. Fließkomma-) und ganzzahligen Eigenschaftstypen gemacht: Die Zusätze "+ 1" und "- 1" gelten nur für ganzzahlige Typen, wie etwa im folgenden Beispiel:

Beispiel 2.3.6 (Behandlung ganzzahliger Eigenschaftstypen)
$A := x \leq a_v$
$B := x < b_v$
Für ganzzahlige x: $A \rightarrow B$, *wenn* $b_v \geq a_v + \mathbf{1}$
Sonst: $A \rightarrow B$, *wenn* $b_v > a_v$

Alle weiteren Vergleichsfunktionen lassen sich nun wie folgt *per Definition* auf die Implikationsbeziehung zurückführen und damit berechnen:

Definition 2.3.5 (Vergleichsfunktionen *äquivalent, stärker, schwächer*)
Für zwei beliebige Formeln F und G gilt:
F äquivalent G ($F \equiv G$), *wenn* $F \rightarrow G$ *und* $G \rightarrow F$
F stärker G, *wenn* $F \rightarrow G$ *und nicht* $G \rightarrow F$
F schwächer G, *wenn* $G \rightarrow F$ *und nicht* $F \rightarrow G$

2.3.2.3 Unifikation

Unifikation ist die wichtigste Funktion, um eine *gemeinsame Basis* für eine Interaktion zwischen mehreren Parteien, die in der Regel unterschiedliche Anforderungen an eine Dienstnutzung stellen, effektiv zu konstruieren. Die Unifikationsfunktion nimmt als Argumente jeweils zwei Bedingungen und liefert als Resultat eine Bedingung, die beide Eingaben erfüllt. Das bedeutet, dass immer wenn die resultierende Bedingung zu wahr evaluiert, dann sind auch die Ausgangsbedingungen automatisch erfüllt. Zusätzlich gilt bei der Unifikation das *Minimalitätsprinzip*, d.h. es soll nicht mehr unternommen werden, als wirklich gefordert ist. Daher muss die schwächste gemeinsame Basis gefunden werden. Formal ist diese Funktion wie folgt definiert[10]:

Definition 2.3.6 (Unifikation)
Für alle wohl geformte Bedingungsausdrücke F_1, F_2, R *gilt:*
$R = unify(F_1, F_2)$, *wenn* $(R \rightarrow F_1) \wedge (R \rightarrow F_2) \wedge$
$\nexists R' : (R' \not\equiv R) \wedge (R' \rightarrow F_1) \wedge (R' \rightarrow F_2) \wedge (R' \rightarrow R)$

Da die Unifikation aus zwei Eingabebedingungen wieder einen Bedingungsausdruck generiert, lässt sie sich im allgemeinen rekursive auf beliebig viele Argumente anwenden. Die Unifikation von drei Bedingungen kann beispielsweise mit $unify(unify(F_1, F_2), F_3)$ berechnet werden. Für Bedingungsausdrücke der Basismächtigkeit in LDNF kann die Unifikationsfunktion mittels folgendes Algorithmus berechnet werden (siehe auch [TGML97a]):

Algorithmus 2.3.1 (Unifikation — Basisalgorithmus)
Seien F_1, F_2 *Ausdrücke in LDNF*
D_1 := *Menge aller Disjunkte in* F_1
D_2 := *Menge aller Disjunkte in* F_2
$TMP := \emptyset$

Für jedes d_i *in* D_1
    Für jedes d_j *in* D_2
        Wenn $(d_i \rightarrow d_j)$ *dann* $TMP := TMP \cup \{d_i\}$
        Wenn $(d_j \rightarrow d_i)$ *dann* $TMP := TMP \cup \{d_j\}$

$D_1 := D_1 \setminus TMP$
$D_2 := D_2 \setminus TMP$

Für jedes d_i *in* D_1
    Für jedes d_j *in* D_2
        Wenn $(d_i \wedge d_j) \not\equiv FALSE$ *dann* $TMP := TMP \cup \{(d_i \wedge d_j)\}$

unify (F_1, F_2) := *Disjunktion aller Elemente von TMP*

In diesem Algorithmus werden zunächst aus den zu vereinigenden Mengen von Disjunkten (die jeweils als eine Menge von *alternativen* Anforderungen betrachtet werden können) diejenigen herausgewählt, deren Erfüllung mindestens

[10] Wohlgemerkt ist die hier definierte Unifikation nicht mit der auf der Substitution von Variablen basierten Unifikation [GN87, S. 66] zu verwechseln. Die auf Substitution basierte liefert eine Belegung der Variablen zurück, während die hier verwendete eine Bedingung generiert.

auch eine Alternative in der anderen Menge wahr macht und somit beiden Ausgangsbedingungen genügt. Dann müssen noch aus beiden Alternativmengen diejenigen geprüft werden, zwischen denen jeweils keine direkte Abhängigkeit besteht. Falls solch ein Paar von unabhängigen Anforderungen *konsistent* zusammengeführt werden können, wird es ebenfalls in die Ergebnismenge aufgenommen. Analog zur Prüfung der Implikationsbeziehung im letzten Abschnitt, die auch die grundlegende Operation in diesem Algorithmus darstellt, liefert dieses unabhängige Prüfen einzelner Disjunkte bzw. Alternativen nur vollständige Ergebnisse unter der Voraussetzung der *Schlankheit* der Ausgangsbedingungen in dem Sinne, dass sie sich nicht zu Bedingungen mit weniger Disjunkten verkürzen lassen. Bei Nichterfüllung dieser Normalform besteht die Gefahr, dass keine gemeinsame Basis gefunden wird, obwohl dies möglich wäre. Jedoch ist es in jedem Fall gewährleistet, dass falls ein positives Ergebnis gefunden wird, dieses auch den Ausgangsbedingungen genügen kann.

2.3.2.4 Schlichtung

Per Definition liefert die Unifikationsfunktion den Wert $\perp$ oder *FALSE* zurück, wenn die Ausgangsbedingungen miteinander logisch unvereinbar sind, d.h. wenn keine nicht–leere Bedingung existiert, die beiden Seiten genügt. Dies kann schon dann eintreten, wenn in den Konjunktionen eine einzige Eigenschaft existiert, für die in den Bedingungen unterschiedliche Werte gefordert werden, beispielsweise in (... $\wedge$ *payment_protocol = "Ecash"*) und (... $\wedge$ *payment_protocol = "NetBill"*).

Das Problem der Findung einer gemeinsamen Kooperationsbasis im allgemeinen sowie der Auflösung konfligierender Anforderungen unterschiedlicher Kooperationspartner im besonderen tritt im Kontext verteilter Anwendungssysteme sehr häufig auf und wurde bisher in der Literatur meist als ein Problem erwähnt, zu dem zwar abstrakte Lösungskonzepte, jedoch keine konkreten technischen Verfahren vorgeschlagen wurden, die eine voll automatisierte Konfliktauflösung ermöglichen. Hier seien zwei solche Beispiele genannt. In der Spezifikation des ODP–Traders [ODP95, S. 38] wurden *arbitration policies* zur Steuerung des Prozesses der Unifikation von Klienten– und Trader–Anforderungen vorgeschlagen:

> An arbitration action template is a template for actions which combine a criteria argument (provided at an interface) with trader criteria and property values (available from the trader's own state). The action produces a resultant criteria which corresponds to the policy (in enterprise terms) for performing a given operation.
>
> The arbitration action represents some computational algorithm within the trader object. It corresponds to the enterprise specification's arbitration policy.

Allerdings wird in dieser Spezifikation weder die Semantik der Schlichtungsaktion beschrieben noch näher erläutert, wie eine solche *arbitration policy* inhaltlich auszusehen hat. Es ist auch nicht klar, wie eine gemeinsame Basis überhaupt (im Nichtkonfliktfall) gefunden werden kann, d.h. eine formale Spezifikation der Unifikationsfunktion wird nicht geliefert. Ein anderer, in dem allgemeineren Kontext der virtuellen Marktplätze (der IBM–Forschung im Bereich E–Commerce) entstandener Ansatz wird in [HFFS99] beschrieben und basiert auf dem Konzept einer *match–making engine*:

> A match–making engine (MME) is a mechanism capable of receiving the description and requirements of a service or a product and of a customer, and matching the description of each against the requirement of the other. (S. 413)

Diese Beschreibung spiegelt auf einer abstrakten Ebene exakt die im letzten Abschnitt definierte Semantik der Unifikationsfunktion wider. (Jedoch lässt auch dieser Ansatz offen, wie (generisch) das Format zur Beschreibung der konkreten Anforderungen auszusehen hat und wie ein solches Format technisch verarbeitet wird.) Basierend auf solchen MMEs wird dann ein Ansatz zur Konfliktauflösung umrissen, in dem ein *Dialog* zwischen dem Kunden und dem Marktplatz solange stattfindet, bis alle Stufen der MMEs erfolgreich durchlaufen sind. Obwohl nicht genau festzustellen ist, wie dieser Prozess genau abläuft, ist eine solche dialog–basierte Konfliktauflösung eher den *Verhandlungsmechanismen* zuzuordnen, die im zweiten Teil dieser Arbeit behandelt werden. Hier werden wir zunächst einen dem ersten Beispiel nahe liegenden, ausschließlich auf Regeln basierten Ansatz verfolgen.

Wenn die Unifikation fehlschlägt, dann existiert ein logischer Konflikt, der nicht allein mit formalen Mitteln behoben werden kann. Um den Konflikt aufzulösen, sind deshalb *zusätzliche* semantische Informationen erforderlich, die eine *Anpassung* (d.h. eine Modifikation) der Ausgangsbedingungen aneinander ermöglichen. Solche Informationen können in so genannten *Schlichtungsregeln* festgehalten werden, die als Metainformation für den Unifikationsprozess dienen. Ein einfacher Ansatz zur Schlichtung konfligierender Bedingungen besteht darin, nach einer bestimmten Ordnung *gewisse Eigenschaften* bei der Unifikation zu ignorieren, d.h. alle Atome einer Bedingung, in denen diese Eigenschaften vorkommen, werden durch entsprechende Atome der konfligierenden Bedingung überschrieben. Die Reihenfolge der Eigenschaften, auf die im Konfliktfall verzichtet werden kann, wird in der Schlichtungsregel entsprechend festgehalten, beispielsweise wie folgt:

Beispiel 2.3.7 (Schlichtungsregel)
$Policy1 := \{daysOfDelivery \leq 2 \wedge speed \geq 533 \wedge price \leq 1800\}$
$aPolicy1 := \{priority = [daysOfDelivery, speed]\}$

D.h. im Fall eines Konfliktes unter der Beteiligung von *Policy1* wird zunächst die Eigenschaft *daysOfDelivery* ignoriert, entsprechend wird das erste Atom aus der Bedingung gestrichen und der Unifikationsprozess wiederholt. Wenn er erneut fehlschlägt, dann wird auch die zweite Eigenschaft in der Prioritätsliste (*speed*) aus der Unifikation ausgeschlossen. Falls dadurch immer noch kein positives Ergebnis erzielt werden kann, wird nun die Konstante *FALSE* zurückgeliefert. Dieses einfache Verfahren, wie an dem Unifikationsalgorithmus 2.3.1 leicht zu erkennen, ist semantisch äquivalent mit dem Überschreiben der entsprechenden Atome mit denen in der jeweils konfligierenden Bedingung. Eine detaillierte Beschreibung dieses Verfahrens ist in [Göl97] enthalten.

Obwohl dieses Verfahren der Kompromissfindung durch Überschreiben von Atomen eine effiziente und einfach zu realisierende Lösung darstellt, ist es in vielen praktischen Szenarien nicht ohne weiteres anwendbar, da das simple Überschreiben ganzer Eigenschaften eine zu grobe und damit inakzeptable Verletzung der ursprünglich geforderten Bedingungen nach sich ziehen kann. Eine realistischere Variante dieses Verfahrens besteht darin, nicht eine Eigenschaft

komplett zu streichen (und damit implizit die Forderung der Gegenseite zulassen), sondern *schrittweise* in die Richtung der Gegenseite zu verändern, bis ein Schnittpunkt gefunden wird. (Natürlich sollte es auch möglich sein, gleichzeitig in beide Richtungen aufeinander zuzugehen.) Um ein unbegrenztes "Nachgeben" zu vermeiden, müsste in der Schlichtungsregel in diesem Fall außer der Reihenfolge der Eigenschaften auch die entsprechenden Grenzwerte angegeben werden, innerhalb derer eine Kompromisslösung zu suchen ist, z.B.:

Beispiel 2.3.8 (Erweiterte Schlichtungsregel)
$aPolicy1 := \{priority = [daysOfDelivery, speed] \wedge daysOfDelivery \leq 7 \wedge speed \geq 500\}$

Allerdings stellt sich hier aus rein logischer Sicht die Frage, warum solche Teilbedingungen über Grenzwerte nicht gleich in die eigentliche Kooperationsregel *Policy1* aufgenommen werden, so dass wesentlich effizienter ein positives Unifikationsergebnis berechnet werden kann, falls es existiert. außerdem müsste auch spezifiziert werden, mit welcher *Schrittweite* die Eigenschaftswerte während des Schlichtungsprozesses verändert werden. Da man sich mit solchen Überlegungen sehr schnell dem Thema Verhandlung nähert, das wie angekündigt im späteren Verlauf der Arbeit ausführlich behandelt wird, sollen diese Aspekte an dieser Stelle nicht weiter ausgeführt werden.

2.3.2.5 Durchsetzung

Wenn im allgemeinen von Regeln die Rede ist, stellt sich natürlicherweise die Frage nach ihrer *Einhaltung* immer passiven Sinne bzw. nach ihrer *Durchsetzung* im aktiven Sinne. Abhängig von der Semantik eines konkreten Regeltyps (siehe Abschnitt 2.3.1.3) ist diese Frage jedoch bereits implizit beantwortet. Im Falle der Requirement–Rule sorgt die Regel für die (passive) Einhaltung einer Bedingung, d.h. sie *hindert* die entsprechende Anwendung daran, die Regel zu verletzen. Im Falle der Action– und der State–Transition–Rule sorgt die Regel für die *Ausführung* einer vorspezifizierten Aktion, wenn bestimmte Bedingungen erfüllt sind. Lediglich die Semantik der Policy–Rule erfordert die explizite Durchsetzung eines Zieles durch die Regelaktivierung selbst, gerade deshalb kann aber dieser Regeltyp als der aktivste von den eingeführten betrachtet werden. Im folgenden wird dargestellt, wie eine beliebige wohl geformte Bedingung durch eine generische Aktion durchgesetzt werden kann.

Unter der Annahme, dass *alle* Eigenschaften (eines Anwendungssystems), die in einer Bedingung vorkommen können, über eine für das Regelsystem zugängliche Schnittstelle *beschreibbar* sind (siehe auch Abschnitt 3.2.3.2), besteht die einfachste generische Aktion zur Durchsetzung einer beliebigen Bedingung darin, diese zunächst auf Inkonsistenz zu prüfen (siehe Abschnitt 2.3.2.1), dann bei negativem Befund zu traversieren und dabei gemäß der Junktoren die Werte der Eigenschaften so zu setzen, dass die gesamte Bedingung zu wahr evaluiert: Bei mit '$\wedge$' verknüpften Teilen müssen alle zu wahr gemacht werden, bei mit '$\vee$' verknüpften dagegen nur ein Teil bearbeitet werden. Auf der Ebene von atomaren Bedingungen kann dann wieder eine Entscheidungstabelle verwendet werden, anhand derer in Abhängigkeit des Relationssymbols und Eigenschaftstyps der Wert der Eigenschaft so gesetzt wird, dass das Prädikat erfüllt ist. Der Negationsjunktor, der nur in atomaren Ausdrücken erlaubt ist, muss hierbei nicht

explizit behandelt werden, da mittels einer weiteren Entscheidungstabelle, die in Anhang A.1 angegeben ist, eine Normierung durchgeführt werden kann, um negierte Atome vorher zu beseitigen.

Beispiel 2.3.9 (Minimale Veränderung der Eigenschaftswerte)
Um die Bedingung ($speed \geq 10 \wedge speed \leq 20$) *durchzusetzen:*
Wenn speed = 21, dann speed := 20
Wenn speed = 5, dann speed := 10

Diese Vorgehensweise lässt sich je nach Bedarf erweitern bzw. optimieren. Wenn beispielsweise bestimmte zu setzende Eigenschaften (vorübergehend) nicht beschreibbar sind, dann kann die Durchsetzung der (in disjunktiver Normalform vorliegenden) Bedingung mit einem anderen Disjunkt versucht werden. Oder es werden zunächst alle Disjunkte überprüft und dasjenige mit den wenigsten Eigenschaften, die gesetzt werden müssten, ausgewählt. Durch diese Vorauswahl wird nicht nur eine effizientere Bearbeitung der Eigenschaften, sondern auch eine insgesamt möglichst geringe Veränderung des aktuellen Zustands des Anwendungssystems (der sich in den Eigenschaften ausdrückt) erreicht. Zu diesem so genannten *Minimalitätsprinzip* gehört auch, dass der Wert einer Eigenschaft jeweils so gesetzt werden soll, dass nicht nur die entsprechende Bedingung erfüllt wird, sondern dass der Abstand zum letzten Wert am geringsten ausfällt, wie durch Beispiel 2.3.9 veranschaulicht wird.

2.3.3 Erweiterung der Funktionen mittels des Simplexverfahrens

Obwohl die dargestellten Regeltypen, insbesondere die Policy–Rule, bereits in der Basismächtigkeit (Abschnitt 2.3.1.4) zur Realisierung eines flexiblen Steuerungsmechanismus für eine Vielzahl von Anwendungsszenarien (siehe die nachfolgenden Kapitel) eingesetzt werden können, ist ihre Ausdrucksmöglichkeit in einer wesentlichen Hinsicht noch deutlich eingeschränkt: Da in den atomaren Bedingungen jeweils nur *eine* Eigenschaft in Relation zu einem Literal gesetzt werden kann, kann eine Beziehung zwischen *mehreren* Eigenschaften höchstens *indirekt* über die Einschränkung durch bestimmte Eigenschaftswerte ausgedrückt werden. Wenn beispielsweise ausgedrückt werden soll, dass der Wert der Eigenschaft *speed* mindestens doppelt so hoch ist wie der Wert der Eigenschaft *cost*, dann könnte man mit der Basismächtigkeit nur eine Bedingung folgender Art formulieren: $[(cost \leq 2 \wedge speed \geq 4) \vee (cost \leq 4 \wedge speed \geq 8) \vee (cost \leq 6 \wedge speed \geq 12) \vee \ldots]$. Da diese prinzipielle Einschränkung für viele Anwendungsprobleme nicht annehmbar ist, muss nach Möglichkeiten gesucht werden, die Ausdrucksmächtigkeit der atomaren Bedingungen zu erhöhen. Aus Anwendungssicht wäre es sicherlich ideal, wenn eine beliebige (mathematische) Beziehung zwischen beliebig vielen Eigenschaften (die im mathematischen Sinne Variablen sind) formuliert werden könnte. Allerdings wäre eine solche Flexibilität mit einer Komplexität verbunden, die praktisch unlösbar erscheint, denn die dargestellten Verarbeitungsfunktionen müssen alle auch für die erweiterte Mächtigkeit realisiert werden. Insbesondere bezüglich derjenigen Funktionen, deren Implementation auf der Prüfung der Implikationsbeziehung zwischen Bedingungen basiert — nämlich die Vergleichs- und vor allem die Unifikationsfunktionen — stellt jede allgemeine Erweiterung der Ausdrucksmächtigkeit eine

erhebliche zusätzliche Komplexität dar, die durch komplett neu konzipierte Algorithmen bewältigt werden muss.

2.3.3.1 Erweiterte Ausdrucksmächtigkeit

Im folgenden wird deshalb gezeigt, wie eine allgemeine, d.h. auf beliebig viele Eigenschaften anwendbare, jedoch funktional begrenzte Erweiterung der bisher behandelten Basisausdrucksmächtigkeit durchgeführt werden kann. Anstatt eines einzigen Eigenschaftssymbols soll nun eine *lineare* Kombination von beliebig vielen Eigenschaften in einer atomaren Bedingung zugelassen werden. Die formale Definition eines atomaren Ausdrucks sieht wie folgt aus:

Definition 2.3.7 (Erweiterte atomare Bedingung)
Eine atomare Bedingung ist entweder ein Ausdruck der Form

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n <RELATION_{num}> <WERT>$$

wobei $a_i \in \mathbb{Q}$ *und die* x_i *Eigenschaftsnamen sind, oder*

$$<EIGENSCHAFT> <RELATION_{sonst}> <WERT>$$

Anstelle von $<WERT>$ kann ein Literal (d.h. ein Konstantensymbol), das i.a. für beliebige rationale Werte steht, eingesetzt werden und anstelle von $<RELATION_{...}>$ kann ein Element aus einer Menge vordefinierter Relationssymbole stehen. Allerdings ist für den allgemeinen Fall $n \geq 2$ die Menge der erlaubten Relationen beschränkt auf die numerischen Typen, nämlich $\{<, >, \leq, \geq, =\}$, da lineare Kombinationen von Eigenschaftswerten für nicht–numerische Typen wie z.B. *String* i.a. keinen Sinn machen bzw. nicht definiert sind. Um weiterhin die Möglichkeit zu erhalten, nicht–numerische Eigenschaftstypen zu verwenden, wird für diese die Syntax der Basismächtigkeit beibehalten.

Zur Formulierung von Bedingungen können nun jeweils beliebig viele Atome dieser neuen Form mit den Junktoren '$\wedge$', '$\vee$' und '$\neg$' gemäß der Definition 2.3.1.4 miteinander verknüpft werden.

Einschränkungen

Wie in den folgenden Abschnitten deutlich wird, besteht die Anpassung der logischen Verarbeitungsfunktionen hauptsächlich darin, das in Abschnitt 2.2.2 eingeführte Simplexverfahren zur Prüfung von Implikationsbeziehungen zwischen Bedingungen der erweiterten Mächtigkeit einzusetzen. Die Verwendung dieses bekanntlich sehr effizienten Verfahrens aus dem Gebiet der linearen Optimierung bringt jedoch folgende technische Einschränkungen mit sich:

- Numerische Typen werden nur vorzeichenbeschränkt akzeptiert. Damit wird gewährleistet, dass die zu implementierenden Funktionen für alle numerische Typen korrekte Ergebnisse liefern. Dies hängt mit den Besonderheiten der Simplex–Algorithmen für reellwertige und ganzzahlige Typen zusammen und wird in Abschnitt 4.2.1.5 näher erläutert.

- Das Simplexverfahren kann grundsätzlich nur mit den Relationen '$\geq$', '$\leq$' und '$=$' umgehen. Um dennoch strenge Ungleichheit mit den Operatoren

'>' und '<' ausdrücken zu können, müssen diese in die vom Simplexverfahren benutzbaren transformiert werden. Obwohl dies im strengen mathematischen Sinne für reellwertige Typen nicht möglich ist, besteht eine pragmatische Lösung darin, bei den entsprechenden Ungleichungen jeweils einen möglichst geringen Faktor ϵ auf die Seite des Wertes entweder hinzuzuaddieren oder davon zu subtrahieren. Wie klein der Faktor ϵ sein muss, um anwendungssemantisch korrekte Ergebnisse zu liefern, hängt u.a. von der Applikationsdomäne ab. Deshalb wird in der Implementation des Regelsystems hierfür ein Ungenauigkeitsfaktor benutzt, der frei einstellbar ist und einen Standardwert besitzt.

- Ein weiteres technisches Problem tritt bei der Evaluierungsfunktion auf und wird durch die Ungenauigkeit der Fließkomma–Arithmetik verursacht. Wenn periodische Zahlen auftreten, werden sie von der Fließkomma–Arithmetik ab einer gewissen stelle gerundet. Dieses ist bei den Typen *Float* und *Double* unterschiedlich. Damit ein aktivierter Ausdruck auch bei der Evaluierung wieder wahr ist, wird auch an dieser Stelle ein Toleranz–Wert einstellbar gemacht, mit dem die Genauigkeit bis zur x–ten Nachkommastelle festgelegt werden kann.

2.3.3.2 Erweiterung bisheriger Funktionen

Die Erweiterung der bisher vorgestellten Verarbeitungsfunktionen betrifft vor allem diejenigen, deren Realisation auf der Prüfung von Implikationen zwischen Bedingungen basiert, nämlich die Unifikation, die Schlichtung und alle Vergleichsfunktionen. Deshalb wird zunächst dargestellt, wie mit Hilfe des Simplexverfahrens die Implikationsbeziehung geprüft werden kann.

Prüfen von Implikationen zwischen erweiterten Bedingungen

Implikationen lassen sich durch folgende Transformation in eine Menge von Ungleichungssystemen auflösen, deren Lösbarkeit dann mit dem Simplexverfahren geprüft werden kann:

$$
\begin{aligned}
&d_i \rightarrow d_j \equiv \neg(d_i \wedge (\neg d_j)) \\
&\text{mit:} \\
&d_i := a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \\
&d_j := a_{j_1} \wedge a_{j_2} \wedge \ldots \wedge a_{j_m} \\
&\text{wobei die } a\text{'s atomare Bedingungen sind:} \\
&d_i \rightarrow d_j \equiv \\
&\neg[(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n}) \wedge \\
&(\neg a_{j_1} \vee \neg a_{j_2} \vee \ldots \vee \neg a_{j_m})] \\
&\text{oder:} \\
&d_i \rightarrow d_j \equiv \\
&\neg[(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_1})) \\
&\vee \\
&(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_2})) \\
&\vee \ldots \vee \\
&(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_m}))] \\
&\text{also:} \\
&d_i \rightarrow d_j \equiv \\
&\neg(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_1})) \\
&\wedge \\
&\neg(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_2})) \\
&\wedge \ldots \wedge \\
&\neg(a_{i_1} \wedge a_{i_2} \wedge \ldots \wedge a_{i_n} \wedge (\neg a_{j_m}))
\end{aligned}
\tag{2.9}
$$

Jede Zeile auf der rechten Seite (von dem Äquivalenzsymbol) des letzten Ableitungsschritts kann durch ein entsprechendes System von Ungleichungen repräsentiert werden und ist (wegen der ersten Negation) genau dann erfüllt, wenn dieses Ungleichungssystem keine Lösung besitzt. Wenn also nachgewiesen wird, dass *jedes* dieser Ungleichungssysteme keine Lösung hat, dann gilt: $(d_i \rightarrow d_j) \equiv TRUE$.

Ist $(d_i \rightarrow d_j) \equiv TRUE$, dann wird bei der Herleitung der schlanken disjunktiven Normalform (Definition 2.3.2) d_j entfernt, da es redundant ist. Beim Unifikationsalgorithmus (Algorithmus 2.3.1) wird dann d_i der Menge TMP hinzugefügt.

Das Simplextableau wird so aufgestellt, dass alle mit AND verknüpften atomaren Ausdrücke von d_i als Restriktionen angesehen werden. Anschließend wird die Lösbarkeit überprüft.

Existenz einer Lösung prüfen

Die Existenz einer Lösung wird geprüft, indem die mit AND verknüpften atomaren Ausdrücke in jeder Zeile als Restriktionen in das Simplextableau eingefügt werden. Der nachfolgende Algorithmus zeigt die Umsetzung mit Hilfe der in Abschnitt 2.2.2 vorgestellten Simplexalgorithmen.

Algorithmus 2.3.2 (Existenz einer Lösung prüfen)
2–phasig := Wenn $\geq$ oder = vorkommen, dann
Zweiphasen–Methode (Algorithmus 2.2.2)

```
Wenn (2–phasig) Dann {
        Wenn (alle Eigenschaftstypen reellwertig) Dann {
                Wenn(1. Phase ⇒ gültige Basislösung) Dann
                        return TRUE;
                Sonst return FALSE;
        }
        Sonst {
                Wenn(1. Phase ⇒ LM leer) Dann
                        return FALSE;
                Wenn(1. Phase ⇒ gültige Basislösung
                ∧ Ganzzahligkeitsanforderungen erfüllt) Dann
                        return TRUE;
                Sonst {
                        gehe über zur 2. Phase;
                        Wenn (2. Phase ⇒ Optimum
                        ∧ Ganzzahligkeitsanforderungen erfüllt) Dann
                                return TRUE;
                        Sonst {
                                Solange (ganzzahliger Optimierungsalgorithmus
                                2.2.4 nicht abbricht)
                                optimiere;
                                /*
                                Der ganzzahlige Optimierungsalgorithmus liefert
                                TRUE zurück, wenn es eine Belegung gibt, die die
                                Ganzzahligkeitsanforderungen erfüllt oder
                                FALSE zurück, wenn es keine solche Belegung gibt.
                                */
                        }
                }
        }
Sonst {
        return TRUE;
        /*
        In diesem Fall besteht ein Normaltableau ohne Hilfsvariablen.
        Es sind nur die Relationen ≤ enthalten.
        Die Schlupfvariablen sind alle positiv.
        Es existiert eine gültige Basislösung
        in der alle Variablen Null sind.
        */
}
```

Erweiterter Unifikationsalgorithmus

Der in Abschnitt 2.3.2.3 vorgestellter Unifikationsalgorithmus stellt innerhalb der generischen Funktionen des Regelverarbeitungssystems die komplexeste Operation dar, die bzgl. der Erweiterung der Ausdrucksmächtigkeit auch am meisten mit Hilfe des Simplexverfahrens überarbeitet werden muss. Deshalb soll anhand

dieses Algorithmuses gezeigt werden, an welchen stellen das Simplexverfahren zum Einsatz kommt. Innerhalb des Unifikationsalgorithmuses (2.3.1) sind diese Stellen nun jeweils mit den Kommentaren "// SV: ..." markiert.

Algorithmus 2.3.3 (Unifikation — Erweiterter Algorithmus)
Seien F_1, F_2 Ausdrücke in LDNF
// SV: Schlanke DNF (gemäss Def. 2.3.3) herbeiführen
D_1 := Menge aller Disjunkte in F_1
D_2 := Menge aller Disjunkte in F_2
$TMP := \emptyset$

Für jedes d_i in D_1
    Für jedes d_j in D_2
        Wenn $(d_i \rightarrow d_j)$ dann $TMP := TMP \cup \{d_i\}$
        Wenn $(d_j \rightarrow d_i)$ dann $TMP := TMP \cup \{d_j\}$
        // SV: Implikationen prüfen
$D_1 := D_1 \setminus TMP$
$D_2 := D_2 \setminus TMP$

Für jedes d_i in D_1
    Für jedes d_j in D_2
        Wenn $(d_i \wedge d_j) \not\equiv FALSE$ dann $TMP := TMP \cup \{(d_i \wedge d_j)\}$
        // SV: Konjunktion prüfen und Redundanz der Ausdrücke

unify (F_1, F_2) := Disjunktion aller Elemente von TMP

Das Simplexverfahren wird also überall dort eingesetzt, wo direkt oder indirekt die Prüfung von Implikationsbeziehungen (wie in Abschnitt 2.3.3.2 gezeigt) erforderlich ist. Bei der Überführung einer Bedingung in schlanke disjunktive Normalform werden zunächst alle Disjunkte entfernt, die von anderen Disjunkten impliziert werden. Bei dem Test, ob die Konjunktion zweier Disjunkte konsistent ist, muss ebenfalls eine Menge von Ungleichungssystemen aufgestellt und auf Lösbarkeit geprüft werden. Schließlich kann die Ergebnismenge TMP (durch Bildung neuer Konjunktionen) noch redundante Elemente enthalten, die ebenfalls durch Prüfung von Implikationen beseitigt werden, um schlanke Ergebnisse zu erhalten.

Evaluierung und Durchsetzung

Eine Bedingung wird evaluiert, indem die linke Seite jedes atomaren Ausdruckes ausgewertet wird und geprüft wird, ob der Wert in der angegebenen Relation zur rechten Seite steht. Das daraus resultierende Ergebnis wird dann mit den logischen Verknüpfungen zwischen den atomaren Ausdrücken ausgewertet. Soll eine Bedingung aktiviert werden, so wird zuerst über die Evaluation geprüft, ob die aktuelle Belegung der Eigenschaften nicht schon die Bedingung erfüllt. Ist das nicht der Fall, so findet die Aktivierung der Bedingung statt.

Die Durchsetzung von Bedingungen wird vorgenommen, indem die Eigenschaften über den Property–Service (siehe Abschnitt 3.2.3.2) auf die Werte gesetzt werden, die die Bedingung erfüllen. Dafür wird mit dem Simplexverfahren und der Standardzielfunktion minimiert, um die Bedingung mit den Belegungen

im Minimum zu aktivieren. Besitzt die Bedingung mehrere Disjunkte, so wird das Disjunkt durchgesetzt, welches die geringste Anzahl von Eigenschaften besitzt. Durch die Beschränkung auf vorzeichenunbehaftete Zahlen ist gewährleistet, dass im Minimum immer eine Lösung existiert. Im Gegensatz dazu kann das Maximum unbeschränkt, also die Lösungsmenge nach oben offen sein, was ein deterministisches Verhalten der Durchsetzungs–Funktion ausschließen würde. Der nachfolgende Algorithmus soll zeigen, wie bei der Durchsetzung durch Minimierung eines beliebigen Disjunkts vorgegangen wird.

Algorithmus 2.3.4 (Durchsetzung)

2–phasig := Wenn $\geq$ oder $=$ vorkommen, dann
Zweiphasen–Methode (Algorithmus 2.2.2)

```
Wenn(2–phasig) Dann {
    Wenn(1. Phase ⇒ gültige Basislösung) Dann {
        gehe über zur 2. Phase;
        minimiere mit Algorithmus 2.2.1;
        Wenn(alle Ganzzahligkeitsanforderungen erfüllt sind) Dann
            return Lösungsvektor;
        Sonst {
            Solange(ganzzahliger Optimierungsalgorithmus 2.2.4 nicht
            abbricht) {
                optimiere;
                /*
                Der ganzzahlige Optimierungsalgorithmus liefert
                den Lösungsvektor zurück, falls es eine Lösung gibt
                und eine Fehlermeldung, falls es keine Lösung gibt.
                Da es sich i.a. um eine unifizierte Bedingung handeln
                sollte,
                sollte dieser Fall in der Praxis nicht eintreten.
                */
            }
        }
    Sonst Fehlermeldung (darf bei unifizierter Bedingung nie vorkommen).
}
Sonst {
    minimiere mit Algorithmus 2.2.1;
    Wenn(alle Ganzzahligkeitsanforderungen erfüllt sind) Dann
        return Lösungsvektor;
    Sonst {
        Solange(ganzzahliger Optimierungsalgorithmus nicht abbricht) {
            optimiere;
            /*
            Der ganzzahlige Optimierungsalgorithmus liefert
            den Lösungsvektor zurück, falls es eine Lösung gibt
            und eine Fehlermeldung, falls es keine Lösung gibt.
            Da es sich i.a. um eine unifizierte Bedingung handeln sollte,
            sollte dieser Fall in der Praxis nicht eintreten.
```

```
                */
            }
        }
}
```

2.3.3.3 Bestimmung optimaler Bedingungen als zusätzliche Funktion

Die Verwendung des Simplexverfahrens ermöglicht es, zu den bisherigen generischen Verarbeitungsfunktionen ein weitere hinzuzufügen, nämlich die Bestimmung optimaler Bedingungen. Im letzten Abschnitt wurde das Simplexverfahren hauptsächlich als Hilfsmittel dafür eingesetzt, die Frage nach der Lösbarkeit von Ungleichungssystemen zu beantworten. Allerdings besteht der eigentliche Hauptzweck dieses Verfahrens, d.h. derjenige, für den es ursprünglich entwickelt wurde, darin, optimale Belegungen von Variablen einer Menge von linearen Ungleichungen effizient zu berechnen. Es liegt also nahe, diese Stärke des Simplexverfahrens auszunutzen, um optimale Bedingungen zu berechnen.

Dazu muss zunächst geklärt werden, was unter Optimierung in Bezug auf Bedingungen genau zu verstehen ist. Wenn beispielsweise im Vorfeld einer Handelstransaktion ein Kunde seine Bedingung (im Sinne einer *Policy Rule*) mit den Angeboten von mehreren potentiellen Anbietern abgleichen will, ist es offensichtlich nützlich nicht nur zu wissen, ob ein bestimmter Anbieter seine Bedingung erfüllen kann, sondern auch, *wie gut* er dies im Vergleich mit den anderen tut, d.h. es geht in solchen Fällen darum, das *beste Angebot* unter vielen herauszufinden. Dies hängt wiederum davon ab, was man unter "Angebot" und unter "beste" versteht. Um quantifizieren zu können, wie gut etwas ist, wird eine Nutzenfunktion benötigt, die für jede Eingabe einen numerischen Wert liefert. Beim Simplexverfahren wird eine solche Funktion genau durch die bereits eingeführte Zielfunktion bereitgestellt. In Bezug auf die Angebote sind zwei Fälle zu unterscheiden: Entweder liegen alle Angebote in Form fester *Belegungen* von Variablen (bzw. Eigenschaften) oder in Form von Bedingungen (die i.a. mehr als eine Belegung zulassen) vor. Im ersten Fall ist die Optimierungsaufgabe einfach und kann ohne das Simplexverfahren gelöst werden: Es müssen nacheinander alle Belegungen in die Zielfunktion eingesetzt werden. Die Ergebnisse werden dann sequentiell sortiert bzw. das Maximum oder Minimum von ihnen herausgewählt.

Im zweiten Fall wird das Simplexverfahren wie folgt angewendet. Es werden nacheinander die einzelnen Anbieter–Bedingungen mit der Kunden–Bedingung als Restriktionen in das Simplextableau aufgenommen (so dass jeweils nur *eine* Anbieter–Bedingung mit der Kunden–Bedingung im Tableau aufgenommen wird). Als Zielfunktion kann jede lineare Funktion über den Eigenschaftsvariablen des Kunden benutzt werden.[11] Im folgenden wird die Bestimmung der optimalen Bedingung in Bezug auf die Standardzielfunktion des Simplexverfahrens dargestellt:

$$x_1 + x_2 + \cdots + x_n \rightarrow min \ / \ max$$

[11] Die Wahl der Zielfunktion hängt letztendlich von dem Optimierungsziel ab bzw. repräsentiert das Optimierungskriterium. Sie kann z.B. aus einer einzelnen Variable bestehen, die gegen *max* bzw. *min* optimiert werden soll oder auch aus der gesamten linken Seite der Kunden–Bedingung.

x_i sind Eigenschaftsvariablen.
Das Tableau sieht folgendermaßen aus:

...	}Zielfunktion
...	}Kunden–Bedingung
...	}Anbieter–Bedingung

Dadurch werden *alle* in der Bedingung vorkommenden Properties entweder minimiert oder maximiert.
Die in diesem Abschnitt gemachten Annahmen — bezüglich der Zielfunktion und der Eigenschaften — gelten für die folgenden unterschiedlichen Fälle, falls nichts anderes geschrieben steht.

1. Fall:

a)

Die Bedingungen bestehen aus jeweils nur *einem* atomaren Ausdruck der Form:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \text{ <RELATION> <WERT>}$$

Die Kunden–Bedingung wird nach und nach mit den Anbieter–Bedingungen als Restriktionen im Simplextableau eingefügt. Lösungsmöglichkeiten:

Minimierung:

- Optimum → merken zum späteren Vergleich
- LM leer → Anbieter scheidet durch Nichterfüllung der Bedingung aus
- LM unbeschränkt → Kunde kann sich eine gültige Belegung aussuchen

Maximierung:

- Optimum → merken zum späteren Vergleich
- LM leer → Anbieter scheidet durch Nichterfüllung der Bedingung aus
- LM unbeschränkt → Kunde kann sich eine gültige Belegung aussuchen

b)

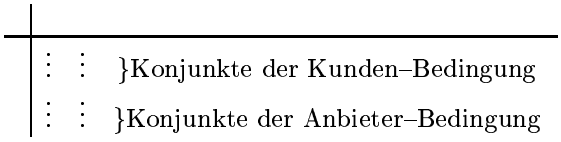
Es ist nicht möglich, die Eigenschaften innerhalb eines atomaren Ausdrucks in unterschiedliche Richtungen zu optimieren.

2. Fall:

a)

Die atomaren Ausdrücke sind *nur* mit AND verknüpft. Nun ist jedes Konjunkt eine Restriktion im Simplextableau. Der Weg der Lösungsfindung ist der gleiche

wie in Fall 1 a).



b)

Für den Fall, dass die in den verschiedenen Konjunkten vorkommenden Eigenschaften in unterschiedliche Richtungen optimiert werden sollen. Z.B. bei einem Autoverkauf soll — für den Anbieter — der Preis möglichst hoch, die Garantie möglichst kurz sein. Dies würde formuliert als

$$Preis \geq 20000 \wedge Garantie \leq 5\,.$$

Auch wenn das Beispiel vielleicht etwas übervereinfacht erscheint, verdeutlicht es ganz gut die beiden unterschiedlichen Optimierungswünsche innerhalb einer Bedingung.

Hierfür müssten zwei Simplextableaus aufgestellt werden, die in die jeweilige Richtung zu optimieren wären. Es ergeben sich folgende Lösungsmöglichkeiten:

- beide Tableaus ergeben Optimum → merken zum späteren Vergleich
- für mindestens ein Tableau gilt: LM leer → Anbieter scheidet durch Nichterfüllung der Bedingung aus
- für beide Tableaus gilt: LM unbeschränkt → Kunde kann sich eine gültige Belegung aussuchen
- mindestens ein Tableau hat Optimum, für die anderen gilt: LM unbeschränkt → merken, welche Konjunkte Optimum haben und welche nicht zum späteren Vergleich (und das Optimum selbst merken)

Durch das Optimieren der einzelnen Konjunkte in unterschiedliche Richtungen wird das Entscheiden von optimalen Bedingungen etwas komplizierter, bzw. die zu haltenden Informationen werden umfangreicher. Es müssen zusätzlich Informationen darüber angegeben und gehalten werden, welche Eigenschaften in welche Richtung zu optimieren sind.

Da es nur zwei Optimierungsrichtungen gibt, können die Konjunkte mit den Eigenschaften, die in die gleiche Richtung optimiert werden sollen, zusammengefasst werden. Es entstehen maximal zwei Tableaus. Dadurch gibt es auch maximal zwei Lösungen pro Konjunkt, nämlich eine für den zu maximierenden Teil und eine für den zu minimierenden Teil. Es muss sich also von Anfang an darauf festgelegt werden, welche der Beiden Lösungen für die Endlösung von Bedeutung sein soll, da nicht immer gewährleistet ist, dass es eine Lösung gibt, in der sowohl das Minimumergebnis das kleinste als auch das Maximumergebnis das größte ist.

Darüber hinaus ist es wichtig, dass die Optimierungsrichtung nicht an den Konjunkten bzw. an der Anordnung der Konjunkte festgemacht werden kann, sondern an den in den Konjunkten vorkommenden Eigenschaften. Es ist also nicht möglich, eine Bedingung zu behandeln, in der in einem Konjunkt mehrere Eigenschaften vorkommen, die in *unterschiedliche* Richtungen zu optimieren sind (s. Fall 1 b).

3. Fall:

a)

Als Erweiterung kann nun die in Fall 2 a) beschriebene Struktur mit OR verknüpft werden. Dies stellt ein kombinatorisches Problem dar, da jedes Disjunkt der Kunden–Bedingung mit jedem Disjunkt der Anbieter–Bedingung paarweise wie in Fall 2 a) beschrieben geprüft werden muss. Davon muss dann das beste Ergebnis vermerkt werden.

Es entstehen also $n \times m$ Simplextableaus, wobei n für die Anzahl der Disjunkte in der Kunden–Bedingung und m für die Anzahl der Disjunkte der Anbieter–Bedingung steht.

b)

Wenn noch zusätzlich unterschiedliche Konjunkte in den Disjunkten vorkommen, in denen Eigenschaften in unterschiedliche Richtungen zu optimieren sind, dann entstehen zusätzliche Teilprobleme wie in Fall 2 b). Es entstehen dann maximal $2 \times n \times m$ Tableaus. Auch hier gilt wieder, dass die zu maximierenden bzw. die zu minimierenden Eigenschaften nicht in einem Konjunkt vorkommen dürfen und die Festlegung der Optimierungsrichtung der einzelnen Eigenschaften auch über die Disjunktion hinaus eingehalten wird.

2.4 Zusammenfassung

In diesem Kapitel wurde die formale Grundlage für eine generische, regelbasierte Steuerung offener verteilter Anwendungssysteme gelegt. Die Generizität der dargestellten Steuerungskonzepte gilt auf dieser Ebene nicht nur bezüglich der Anwendungssemantik, sondern auch der möglichen Systeminfrastrukturen, auf den sie realisiert werden können, da diese Konzepte auf völlig anwendungsunabhängigen formalen Verfahren basieren und technologieneutral entworfen wurden.

Deshalb war es erforderlich, zunächst den (in Abschnitt 1.1) als eine *Abstraktion* der zu steuernden Anwendungen eingeführten Dienstbegriff mittels geeigneter *Dienstmodelle* zu präzisieren. Daraus wurde ein Konzept zur Steuerung des Verhaltens von Diensten abgeleitet, das auf der regelbasierten bzw. *regelgeleiteten* Modifikation des Dienst*zustands* bzw. der Dienst*attribute* im allgemeinen basiert. Ein solches Steuerungskonzept kann auch im Sinne einer *Anpassung als "moderate Veränderung"* (vgl. [Gri98, S. 19]) aufgefasst werden, da die Veränderung hier weder die Struktur noch die Funktionalität der gesteuerten Dienste betrifft.

Ein auf eingeschränkten Formen der Prädikatenlogik basiertes *allgemeines Regelformat* wurde dann eingeführt, auf dem eine Reihe von *generischen Verarbeitungsfunktionen* definiert wurden, die für alle *konkreten Regeltypen* anwendbar sind. Hierbei wurde insbesondere der *Interaktionsorientierung* verteilter Anwendungen durch eine speziell konzipierte Unifikationsfunktion Rechnung getragen, welche die unterschiedlichen, als Regeln gefasste Anforderungen verschiedener Anwendungen auf einen gemeinsamen Nenner — ebenfalls in Form einer Regel — abbildet. Die hierzu benötigten Algorithmen hängen jeweils von der Ausdrucksmächtigkeit des verwendeten Regelformates ab. Für die Verarbeitung von

Regeln der zweiten und höheren Ausdrucksmächtigkeitsstufe wurde das Simplexverfahren als ein mathematisches Hilfsmittel eingesetzt, das deshalb bezüglich der relevanten Algorithmen ausführlich beschrieben wurde.

Kapitel 3

Systemarchitektur einer interaktionsorientierten Regelverarbeitung

So wichtig eine formal–logische Konzeption für regel–basierte Steuerungs- und Konfigurationsmechanismen — wie die im vorangehenden Kapitel theoretisch konzipierten Verarbeitungsfunktionen — auch ist, bringt sie noch keinen unmittelbaren Nutzen für das sehr heterogene Gebiet der Verteilten Systeme und deren Anwender, wie es in Kapitel 1 dargestellt wurde. Damit derartige Formalismen, die in — zumindest äußerlich — ähnlicher Form in der Theoretischen Informatik und auch in der Künstlichen Intelligenz häufig anzutreffen sind, für verteilte Anwendungssysteme nutzbar gemacht werden können, bedarf es einer entsprechenden Systemarchitektur zur Umsetzung ihrer abstrakten Funktionalität in konkrete Systemunterstützungsdienste, von denen die Anwendungen bzw. Anwender dann direkt profitieren können. Eine solche Systemarchitektur muss die Anforderungen (vgl. Abschnitt 1.4.1) und Besonderheiten offener verteilter Umgebungen berücksichtigen, damit die entsprechend implementierten Systemkomponenten reibungslos in die existierende Landschaft verteilter Systeme und Architekturen integriert werden können.

In diesem Kapitel werden zunächst bestehende Ansätze aus verschiedenen Bereichen der Informatik, die von dem Regel– bzw. Policy–Konzept Gebrauch machen, untersucht. Aus der anschließenden Bewertung dieser Ansätze werden dann sowohl eine zentrale als auch eine dezentrale Architektur zur Anwendung des im letzten Kapitel formal konzipierten Regelkonzepts vorgestellt, die möglichst viele unterschiedliche Anforderungen in der Praxis erfüllen sollen.

3.1 Bestehende Ansätze

Aus der Literatur, insbesondere der KI–Literatur, gibt es eine Vielzahl von regel–basierten Ansätzen zur Verarbeitung von Information. Der überwiegende Teil dieser Arbeiten beschäftigt sich jedoch mit der Wissensverarbeitung in technisch und konzeptuell monolithischen, (teil-)intelligenten Systemen, insbesondere mit der Frage, wie neue *Fakten* mit Hilfe von Regeln aus bestehenden

Fakten einer geschlossenen Domäne geschlossen werden können. Obwohl solche Systeme in Bezug auf spezielle Ziele der KI, wie beispielsweise die Realisierung von Diagnosesystemen, relativ erfolgreich sind, sind ihre Mechanismen nicht direkt auf das Gebiet der verteilten Anwendungssysteme — insbesondere das Teilgebiet der elektronischen Dienstemärkte — übertragbar, da charakteristische Anforderungen dieses Gebiets wie Heterogenität, Offenheit, Autonomie und Interaktion nicht bzw. nicht primär berücksichtigt werden. Daher werden im folgenden exemplarisch nur Ansätze aus der Informatik–Literatur untersucht, deren Anforderungen und Techniken eine deutlich erkennbare Verwandtschaft mit den in dieser Arbeit aufweisen. Dabei wird sich zeigen, dass es durchaus schon Ansätze zur Integration bestimmter KI–Techniken in verteilte Anwendungssysteme gibt und dass in Zukunft solche integrierten Systeme vermehrt zu erwarten sind.

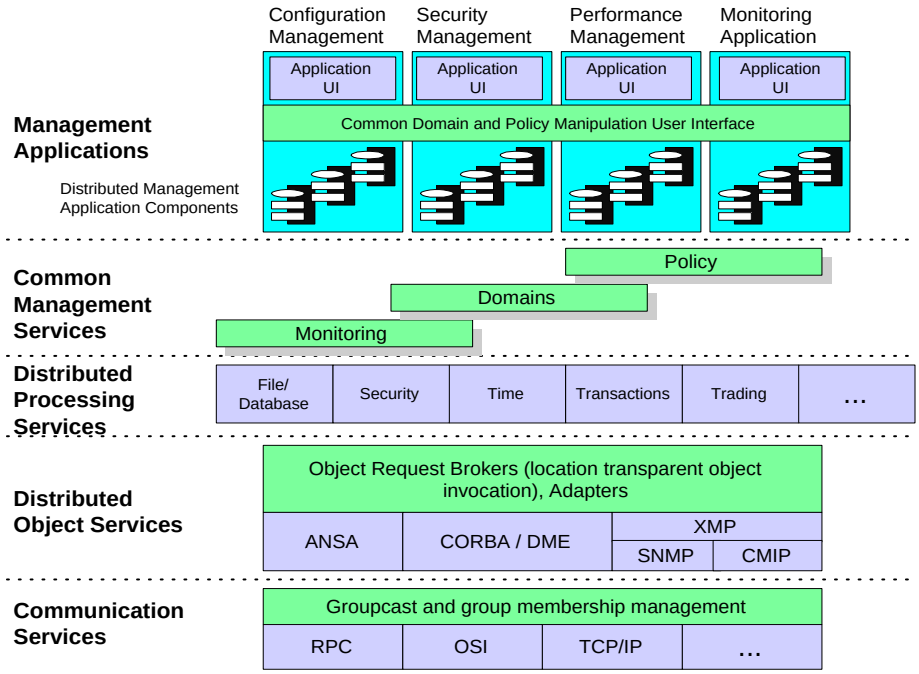


Abbildung 3.1: Schichtenarchitektur eines Management–Systems nach [Slo94b]

3.1.1 Policy Management

Wie in Abschnitt 2.3.1.1 bereits erwähnt, stammt das Konzept des Policy–Managements ursprünglich aus dem Bereich Netzwerk– und System–Management, einem den verteilten Anwendungssystemen relativ nahe verwandten Gebiet. Der Hauptunterschied dieser beiden Gebiete liegt in der jeweiligen Sichtweise auf die Objekte: Während es in dem ersten um die effiziente Verwaltung einer großen Menge von Objekten (in einem Netzwerk) geht, die aus Sicht des Administrators als *passiv* — d.h. die eigentliche Funktionalität der Objekte steht nicht im Interesse — betrachtet werden, werden in dem zweiten Gebiet die Anwendungsobjekte als *aktive*, autonome Objekte behandelt, deren Funktionalität,

insbesondere deren Fähigkeit zur gegenseitigen Interaktion, durch Regelmechanismen als eine Form der Systemunterstützung verbessert werden sollen. Trotzdem können viele Teilmechanismen aus dem einen Gebiet auch auf das andere angewendet werden.

Die allgemeine Gesamtarchitektur eines objektorientierten, Policy–gesteuerten Management–Systems ist (in Anlehnung an [Slo94b]) in Abbildung 3.1 illustriert. Demnach ist ein Management–System keine große monolithische Einzelapplikation, sondern besteht aus einer erweiterbaren Menge von Management–Applikationen, die nach außen hin eine konsistente Benutzerschnittstelle zur Verfügung stellen. Diese Applikationen werden von Administratoren über anwendungsspezifische Schnittstellen genutzt und nutzen ihrerseits primär Dienste der nächstliegenden Schicht (*Common Management Services*), um jeweils die administrierten Objekte zu überwachen (*Monitoring*), zu gruppieren (*Domains*) und um die Generierung, (persistente) Verwaltung und Aktivierung von Management–Policies (*Policy*) zu unterstützen. Alle Management–Applikationen und Dienste können in einer verteilten Umgebung interagieren und können deshalb die typischen Middleware- und Kommunikationsdienste der weiter darunter liegenden Schichten, auch direkt über Schichtgrenzen hinweg, in Anspruch nehmen. Der wesentlichste Aspekt dieser Schichtenarchitektur ist also die in den Abschnitten 1.1 und 2.1.1 dargestellte Dienstsicht.

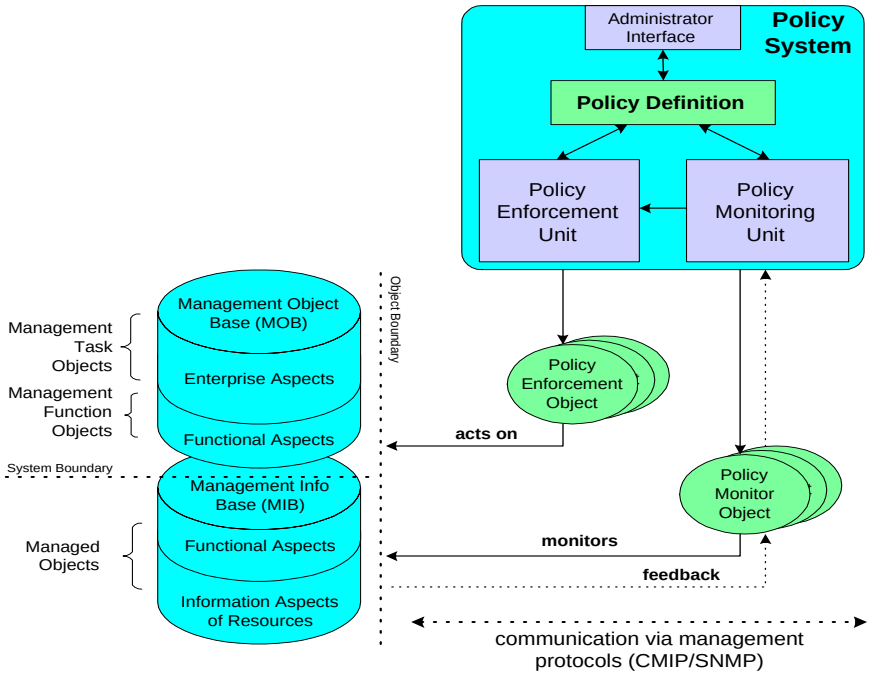


Abbildung 3.2: Architektur eines Policy–Systems nach [Wie95b]

Der *Policy*–Dienst selbst kann nach einer Architektur aufgebaut sein, die in Anlehnung an [Wie95b] in Abbildung 3.2 gezeigt ist. Im Kern besteht ein solches Policy–System aus einer Generierungs- (*Policy Definition*), einer Überwachungs- (*Monitoring Unit*) und einer Aktivierungs- (*Enforcement Unit*) Komponente.

Dabei können die Überwachungs- und Aktivierungseinheiten die Funktionalität der externen *Monitoring*– bzw. *Domains*–Dienste nutzen.

Die Aktivierungseinheit führt jeweils eine Policy–Definition, die in diesem Fall eine Menge von Management–Instruktionen beinhaltet, an einem Management–Objekt (MO) bzw. an einer Domäne von MO aus, die ihrerseits Management–Funktionen an den Anwendungsobjekten ausführen, z.B. um entsprechende Ressourcen und Management–Rechte zu vergeben. Die Überwachungseinheit kontrolliert Management–Objekte bzw. ihre Attribute. Sie registriert auch, ob eine Policy durchgesetzt werden kann. Falls dies nicht mehr der Fall ist, kann sie entsprechend die Funktion der Aktivierungskomponente beeinflussen (bzw. versuchen, darauf Einfluss zu nehmen).

3.1.2 Aktive Datenbanksysteme

Aktive Datenbankmanagementsysteme (ADBMS) gehören zu den Gebieten, die in den letzten Jahren die Entwicklung von Regelanwendungsmechanismen als Systemunterstützung für die auf Datenbanken aufsetzenden Applikationen am meisten voran gebracht haben ([DGG95, WC96] geben einen Überblick über die Ziele und Grundmechanismen dieses Gebiets). Im Gegensatz zu konventionellen, passiven Datenbanksystemen sind ADBMS in der Lage, einen Teil der Anwendungssemantik in Form von Regeln in das Datenbanksystem aufzunehmen und zur Laufzeit auszuführen, so dass die betreffende Applikation von diesem Teil befreit ist. Und zwar kann eine aktive Datenbank diejenigen Funktionen übernehmen, die als *reaktives Verhalten* der Anwendung beschreibbar ist, d.h. wenn gewisse Aktionen als *Folge* von entsprechenden Ereignissen und Umständen ausgeführt werden sollen.

Wenn beispielsweise eine Buchungsapplikation in der Lage sein soll, bei Überschreiten einer bestimmten Anzahl von Kunden zusätzliche Fahrzeuge anzufordern, dann kann dieses Verhalten als eine *Event Condition Action* bzw. ECA–Regel spezifiziert und in die Datenbank eingegeben werden, die dann bei Eintreten der relevanten Situation aktiv und unabhängig von der Applikation die Fahrzeugbestellung auslöst. Der Vorteil dieses Vorgehens besteht darin, dass die Realisierung des reaktiven Verhaltens an einer Stelle erfolgt, wo alle Informationen *zentral* gesammelt werden, so dass eine redundante und inkonsistente Überwachung bzw. Ausführung der geforderten Semantik vermieden werden kann. Die genannte Buchungsapplikation kann nämlich mehrfach in verschiedenen Geschäftsstellen installiert sein und alle Instanzen greifen auf die gemeinsame Datenbank zu, um Buchungen durchzuführen. Im allgemeinen werden in ADBMS ausschließlich ECA–Regeln, die exakt dem Regeltyp Action–Rule in Abschnitt 2.3.1.3 entsprechen, verwendet, um reaktives Verhalten zu integrieren[1].

In Anlehnung an [Pat99] ist die abstrakte Architektur eines aktiven Datenbanksystems in Abbildung 3.3 wieder gegeben. Demnach besteht die Regelverarbeitung einer ADBMS aus drei Hauptkomponenten: dem *Event Detector*, *Scheduler* und *DB Language Interpreter*. Die Aktivierung von Regeln läuft wie folgt ab:

1. Die Datenbank benachrichtigt den Event Detector, wenn Ereignisse in

[1] In einigen Systemen werden auch Regeln benutzt, deren Event– oder Condition–Teil fehlt bzw. nur implizit dargestellt ist.

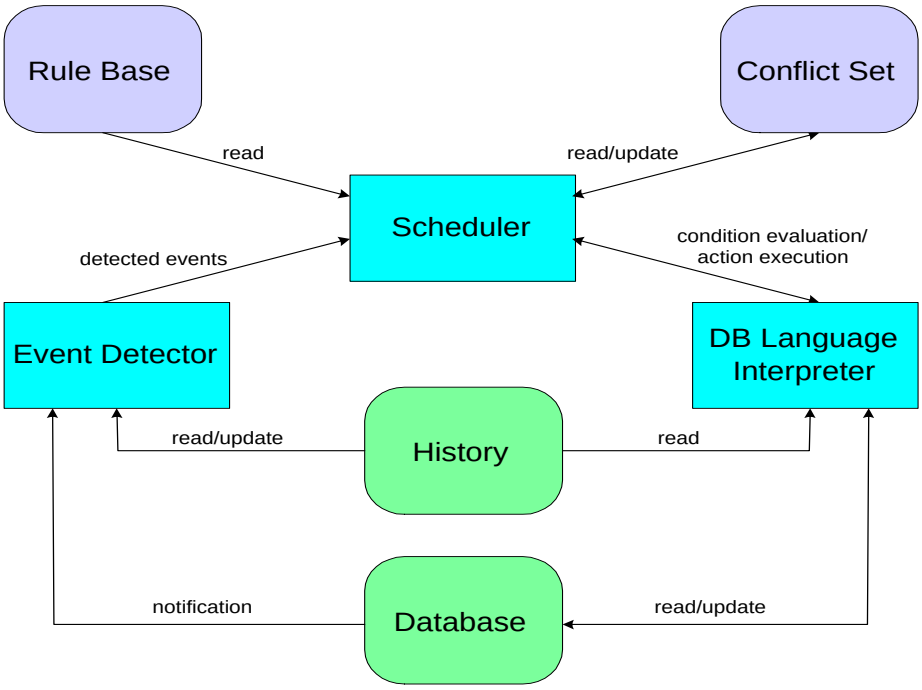


Abbildung 3.3: Abstract active rule system architecture nach [Pat99]

der Datenbank, für die der Event Detector sich registriert hat, eingetreten sind. Der Event Detector kann daraufhin die *History* befragen, die ausführliche Informationen über die stattgefundenen Ereignisse aufbewahrt.

2. Der Scheduler fragt Informationen über relevante Ereignisse bei dem Event Detector ab, vergleicht diese mit den Regelbeschreibungen in der *Rule Base* um festzustellen, welche Regeln durch die Ereignisse getriggert werden. Getriggerte Regeln, die nicht sofort aktiviert werden sollen, werden in die *Conflict Set* aufgenommen.

3. Der Interpreter wird immer dann von dem Scheduler aufgerufen, wenn ein Condition–Teil evaluiert oder eine Regelaktion ausgeführt werden soll. Die Aktivierung von Regeln kann Interaktionen mit der Datenbank beinhalten, wobei der Interpreter weitere Information aus der History lesen kann, um z.B. Aktionen im Sinne von Anfragen über Ereignisse auszuführen. Entsprechende Lese- und Schreibzugriffe auf die Datenbank können selbst als relevante Ereignisse zum Event Detector weiter geleitet werden und somit zur Wiederholung des Prozesses führen.

3.1.3 Wissensbasierte Systeme

Trotz der anfangs dieses Kapitels angemerkten Einschränkung der Anwendbarkeit von wissensbasierten Systemen im allgemeinen auf offene verteilte Umgebungen soll in diesem Abschnitt ein System vorgestellt werden, das sich bekannter KI–Techniken wie vorwärts und rückwärts verkettetes Schließen (engl.

Forward bzw. *Backward Chaining*) zur Verarbeitung von Regelwissen bedient, nämlich das in [BB98] beschriebene Agenten–Framework. Das Besondere an diesem System ist, dass es zu den ersten wissensbasierten Systemen gehört, die für typische Geschäftsanwendungen konzipiert sind und sich weit verbreiteter, de–facto Standardtechnologien für verteilte Anwendungen wie Java, Applets und JavaBeans bedienen. Daher verdient dieses System eine nähere Betrachtung in Bezug auf die Integration der Regelmechanismen der KI in verteilte Anwendungen.

In Bezug auf wissensbasierte Anwendungen besteht das Framework von [BB98] im wesentlichen aus zwei Bestandteilen: einem regelbasierten Wissensverarbeitungssystem und einem agentenbasierten Anwendungssystem, das von dem wissensbasierten System Gebrauch macht, um intelligente Agentenapplikationen zu entwickeln. Das wissensbasierte System orientiert sich an der konzeptionellen Architektur üblicher Expertensysteme, die drei Elemente enthält:

- Eine Wissensbasis (*knowledge base*), die alle *if–then*–Regeln und bekannte (von außen eingegebene) Fakten beherbergt.
- Ein Arbeitsspeicher bzw. Datenbank (*working memory*), in dem abgeleitete Fakten und Daten gespeichert werden.
- Eine Inferenz–Maschine (*inference engine*), die die Logik beinhaltet, um Regeln und Daten zu verarbeiten.

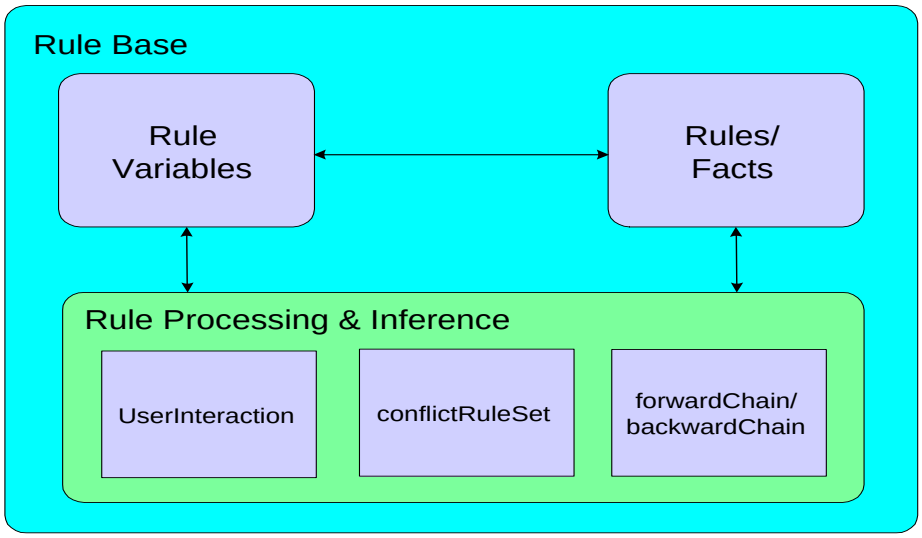


Abbildung 3.4: Schematische Darstellung der regelbasierten Wissensverarbeitungskomponente nach [BB98]

In der Implementation spiegelt das System von [BB98] jedoch nicht direkt diese Dreiteilung wider, sondern besitzt die in Abbildung 3.4 schematisch dargestellte Struktur. Dementsprechend ist die gesamte Wissensverarbeitungsfunktionalität in der Regelbasis (*Rule Base*) vereinigt. Sie enthält eine Sammlung von Regeln und Fakten (*Rules/Facts*), eine Menge von so genannten Regelvariablen

(*Rule Variables*) und die Implementation der gesamten Verarbeitungsfunktionalität, die hier schematisch zu *Rule Processing & Inference* zusammengefasst ist. Eine Regel in diesem Framework repräsentiert grundsätzlich immer eine Wissenseinheit, die aus einer Menge von *Antezedenzen* (*if*–Teil) und einer *Konsequenz* (*then*–Teil) besteht. Die Grundsemantik einer solchen Regel ist, dass die Konsequenz als neuer Fakt in die Wissensbasis aufgenommen werden kann, wenn die Antezedenzmenge zu wahr evaluiert. Einzelne Antezedenzen und Konsequenzen werden durch *Klauseln* in eingeschränkter prädikatenlogischer Form (äquivalent zu den atomaren Bedingungen der Basisausdrucksmächtigkeit in Abschnitt 2.3.1.4) repräsentiert. Antezedenzen können zwar im Prinzip mit den boolschen Operatoren *AND* und *OR* verknüpft werden, in der Implementation dieses Frameworks ist jedoch nur die (flache) Konjunktion zugelassen (vgl. [BB98, S. 85]). Fakten sind einzelne Klauseln, denen einer der beiden Wahrheitswerte *TRUE* oder *FALSE* zugeordnet ist. Alternativ zu den rein faktenorientierten Regeln gibt es auch die Möglichkeit, Funktionsaufrufe in Klauseln zu spezifizieren. Im Falle von Antezedenzklauseln müssen die Funktionen boolsche Werte zurückliefern (um die *if*–Semantik zu erfüllen) und werden als *Sensoren* bezeichnet. Bei Konsequenzklauseln können prinzipiell beliebige Funktionen, die als *Effektoren* bezeichnet werden, aufgerufen werden. Auf diese Weise wird eine Semantik von Aktionsregeln realisiert, die sehr ähnlich der des Regeltyps Action–Rule in Abschnitt 2.3.1.3 ist. Schließlich sind Regelvariablen Name–Wert–Paare, deren Name–Teil als Variable in Regelklauseln referenziert werden kann. Damit entsprechen sie den Eigenschaften in Definition 2.3.1.4.

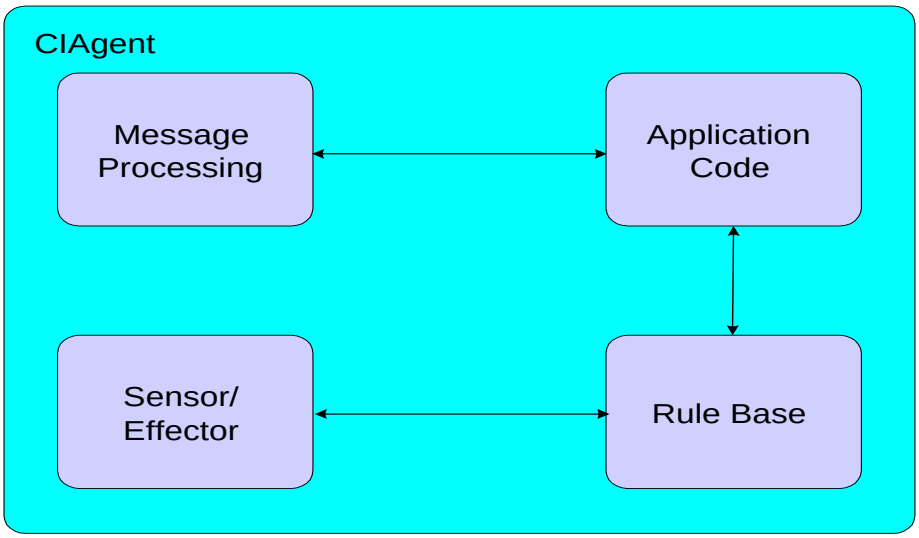


Abbildung 3.5: Funktionale Komponenten eines regel–basierten Agentensystems nach [BB98]

Die Funktionalität der Inferenzmaschine dieses Frameworks beinhaltet zwei Schließmethoden: Bei dem ersten, Forward Chaining genannten, Verfahren werden in Folge einer entsprechenden Inferenzaufforderung alle Regeln in der Regelbasis mit allen vorhandenen Fakten überprüft, um die Menge der getriggerten Regeln (*Conflict Set*) zu bilden. Im Gegensatz zur ereignisorientierten Regelver-

arbeitung (bei der eine Regel durch ein Ereignis entsprechenden Typs getriggert wird) heißt eine Regel in diesem Fall "getriggert", wenn ihre Antezedenzmenge insgesamt zu wahr evaluiert. Eine Kontrollstrategie, *conflict resolution procedure* genannt, wird benutzt, um die Reihenfolge der zu feuernden Regeln in der Conflict Set zu bestimmen. Die Regeln werden dann dadurch *gefeuert*, dass der Konsequenzteil in die Faktenmenge aufgenommen wird, wodurch neue Regeln getriggert und in die Conflict Set hinzugefügt werden können. Der Prozess wird so lange wiederholt, bis keine neuen Regeln mehr getriggert werden. Beim zweiten, als Backward Chaining bezeichneten, Verfahren wird zielorientierter vorgegangen, indem jeweils nur diejenigen Regeln betrachtet werden, die eine bestimmte, als *Ziel* aufgefasste Klausel im Konsequenzteil haben. Diese Regeln werden dann nacheinander überprüft, indem jeweils die Klauseln in der Antezedenzmenge evaluiert werden. Wenn die Evaluierung positiv ausfällt, wird die entsprechende Regel gefeuert. Falls der Wahrheitswert bestimmter Regelvariablen noch unbekannt ist, wird die entsprechende Klausel als neues Ziel hinzugenommen, woraufhin wieder passende Regeln herausgesucht werden. Dies wird so lange wiederholt, bis das ursprüngliche Ziel erreicht ist oder keine neuen Ziele mehr bestimmt werden können. Der Hauptvorteil des Backward Chaining besteht darin, dass während des Inferenzprozesses gezielt Benutzereingaben bezüglich der Wahrheitswerte von unbekannten Antezedenzen gemacht werden können, um eine als Ziel gestellte Frage (direkter) beantworten zu können. Deshalb eignet sich dieses Verfahren besonders gut für interaktive Diagnosesysteme.

Das Agentensystem, das auf diesem Regelsystem aufbaut, bietet dann unterschiedlich spezifische Klassen zur Erzeugung von Anwendungsagenten wie verschiedene Käufer- und Verkäuferagenten, die jeweils paarweise miteinander interagieren können, um eine Kauftransaktion zu bewirken. Dabei erben alle Agentenklassen von der Basisklasse *CIAgent*, deren Struktur in Abbildung 3.5 veranschaulicht ist. Diese Klasse realisiert einen regelbasierten Agenten dadurch, dass sie die dargestellte Regelbasis (gemäß Abbildung 3.4) insgesamt als Klassenbestandteil beinhaltet. Weitere Bestandteile dieser Klasse sind Methoden zur Kommunikation (*Message Processing*) in einer höheren Agentensprache, nämlich KQML, sowie Methoden, die als Sensoren und Effektoren aufgerufen werden, und schließlich die Anwendungsmethoden, z.B. um die Semantik eines Käuferagenten umzusetzen, selbst.

3.1.4 Bewertung

Obwohl regelbasierte Systeme in der Informatik, insbesondere auf dem Gebiet der Künstlichen Intelligenz, eine lange Tradition aufweisen, existieren nur wenige Mechanismen und Architekturen, die unmittelbar eine geeignete konzeptionelle und systemtechnische Basis für die Realisierung eines regelbasierten Steuerungsmechanismus für verteilte Anwendungssysteme liefern. Der Hauptgrund dafür liegt in dem bereits erwähnten geschlossenen, d.h. auf eine bestimmte Domäne beschränkten Charakter konventioneller, wissensbasierter KI–Systeme, in den Aspekte wie Verteilung, Heterogenität und auch Steuerung selbst meist keine Rolle spielen. Deshalb wurden als bestehende Ansätze, die eine Relevanz für die dieser Arbeit zugrunde liegenden Ziele besitzen könnten, aus unterschiedlichen Bereichen der Informatik nur solche ausgewählt, die einen deutlichen Bezug zu verteilten Systemen, entweder als Domäne oder aufgrund der verwendeten Konzepte und Techniken, haben.

In diesem Sinne erscheinen die zunächst untersuchten *Policy Management* Ansätze aus dem Netzwerk– und Systemmanagementbereich deshalb besonders relevant, weil sie erstens *verteilte* Objektsysteme als Zieldomäne haben und zweitens weil ihnen das Policy–Konzept (vgl. Abschnitt 2.3.1.1) als ein *Steuerungs*mittel zugrunde liegt. Ein genauer Blick auf die entsprechenden Systemarchitekturen hat allerdings schnell gezeigt, dass sie für die Anforderungen der Anwendungsebene nicht besonders geeignet sind, da Anwendungsobjekte auf Netzwerkebene als absolut passive Entitäten behandelt werden und die angestrebte Steuerung entweder gar nicht diese Objekte selbst (sondern die so genannten Management–Objekte) betrifft oder damit eine ganz andere Zielsetzung verfolgt wird, nämlich Optimierung der Managementziele und des Anwendungsverhaltens[2]. Darüber hinaus haben die meisten konkreten Policy–Formate, die bisher für den Management–Bereich vorgeschlagen wurden, nur relativ wenig mit formalen Regeln, wie sie aus der Logik oder den wissensbasierten Systemen her bekannt sind, zu tun, sondern sind höchstens als semi–formal zu bezeichnen (vgl. auch Abschnitt 2.3.1.1). Dementsprechend sind die konkreten Management–Architekturen für die Verarbeitung dieser spezifischen Formate konzipiert, z.B. als semi–formale Hilfsmittel für die Arbeit eines Systemadministrators, und sind deswegen nicht generisch genug, um auf die Anwendungsebene übertragen zu werden.

Die abstrakte Architektur der als nächstes vorgestellten Aktiven Datenbanksysteme liefert ein Beispiel dafür, wie der Grobaufbau eines *zentralen* Regelverarbeitungssystems zur direkten Unterstützung der Anwendungssemantik aussehen kann. Allerdings ist diese Architektur auf ECA–Regeln als einzigen Regeltyp beschränkt und vor allem fehlt hier eine explizite Schnittstelle zwischen dem Regelsystem und den dadurch unterstützten Anwendungen, da die gesamte Interaktion zwischen den beiden Ebenen *indirekt* über die Datenbank abgewickelt wird. D.h. Anwendungen, die keine Aktionen im Datenbanksystem auslösen, können nicht (direkt) von einer solchen Architektur Gebrauch machen.

Das als drittes untersuchte, regelbasierte Wissensverarbeitungssystem für Agenten zeichnet sich durch seine praxisorientierten Zielsetzungen, nämlich die Unterstützung von Geschäftsanwendungen, und durch seinen engen implementationstechnischen Bezug zu aktuellen verteilten Anwendungen aus und kann insgesamt als ein Versuch betrachtet werden, einen ersten konkreten Schritt zur Integration bekannter (jedoch simpler) KI–Techniken in das Gebiet Verteilte Systeme zu unternehmen. Allerdings erscheint die gesamte Vorgehensweise all zu "ad–hoc" bzw. nicht gerade systematisch. Insbesondere fehlt hier eine klare gesamte Architektur, die wichtigen softwaretechnischen Aspekten verteilter Anwendungen wie etwa Objektorientierung und Modularisierung Rechnung trägt. (So gibt es in der angegebenen Quelle keine einzige graphische Darstellung einer Systemarchitektur bzw. eines Klassenentwurfs.) Deshalb vermittelt dieses System insgesamt zwar einen praxisnahen Ansatz zur Programmierung von regelbasierten Agenten in der Sprache Java, aber keine Architektur, die bestimmte konzeptionelle Anforderungen erfüllen würde[3].

[2] Jedoch lässt sich durchaus eine Parallele zwischen den beiden Domänen bzw. entsprechenden Zielsetzungen ziehen, wenn die Management–Objekte der einen Domäne den Anwendungsobjekten der anderen gleichgestellt werden. D.h., unter der Voraussetzung, dass genügend *generische* Steuerungsmechanismen vorliegen, ist es möglich, ein Policy–Management–System für beide Domänen zu realisieren.

[3] Dieses unsystematische Vorgehen wird von den Autoren mit dem (Schein-)Argument be-

Ein wichtiger, für die vorliegende Arbeit grundlegender Aspekt, der in keinem der vorgestellten Ansätze explizit berücksichtigt wird, ist der schon häufig erwähnter Aspekt der Interaktion zwischen den zu unterstützenden Anwendungen. So gibt es zwar in einigen Systemen Mechanismen oder Konzepte — wie etwa *Scheduling* oder *Conflict Set* — zur Auflösung von Konflikten bzgl. der Aktivierungsreihenfolge mehrerer Regeln, die gleichzeitig getriggert wurden, jedoch fehlen all diesen Systemen Mechanismen zur automatischen Findung einer gemeinsamen *semantischen* Interaktionsbasis bzw. zur Auflösung potentieller semantischer Konflikte in Form entsprechender Interaktionsregeln, wie sie in 2.3 erarbeitet wurden.

Auf einer abstrakteren Ebene betrachtet lässt sich aus der Analyse der bestehenden Ansätze bzgl. der Systemarchitektur für ein regelbasiertes System zur Steuerung verteilter Anwendungen die Kernaussage ableiten, dass eine solche Architektur entweder *zentral* oder *dezentral* aufgebaut sein kann. Bei einer zentralen Architektur gibt es immer eine Instanz, die die Regeln aller oder zumindest *mehrerer* Anwendungen verwaltet, eventuell Konflikte auflöst und aktiviert. Entscheidend ist hier also die Anzahl der Anwendungen, die *gemeinsam* unterstützt werden, die Instanz bzw. das Regelsystem kann trotzdem *verteilt* implementiert sein. Bei einer dezentralen Architektur gibt es keine solche Instanz und die Regeln werden jeweils bei oder in der betreffenden Anwendung selbst deponiert. Demnach ist die Architektur der Aktiven Datenbanksysteme eindeutig als zentral zu bezeichnen. Die meisten agentenbasierten Systeme wie das vorgestellte CIAgent–Framework sind dagegen dezentral. Bei den Policy–Management–Architekturen gibt es sowohl zentrale als auch dezentrale, meist agentenbasierte, Ansätze.

Zentrale und dezentrale Architekturen können beide sowohl Vor- als auch Nachteile für die Anwendungen haben, die leicht zu erkennen sind. Der Hauptvorteil einer zentralen Architektur besteht darin, dass jede Anwendung das Regelsystem über eine klar definierte Schnittstelle als einen *Unterstützungsdienst*, der die gesamte regelbasierte Funktionalität bietet, nutzen kann. Wenn zur Laufzeit kein Bedarf für eine solche Funktionalität besteht, ist die Anwendung komplett von der Regelverarbeitungsfunktionalität entlastet. Außerdem können mit der zentralen Architektur effektive Scheduling- und Konsistenzerhaltungsmechanismen implementiert werden, die bei einer dezentralen Architektur wesentlich schwieriger umzusetzen sind. Dagegen kann eine zentrale Architektur prinzipiell immer einen Flaschenhals für die Anwendungen erzeugen, der ihre Ausführung erheblich beeinträchtigen oder sogar komplett lahm legen kann. Darüber hinaus haben die Anwendungen i.a. eine wesentlich größere (semantische) *Autonomie* bei einer dezentralen als bei einer zentralen Architektur, was häufig eine wichtige Anforderung für verteilte Systeme bedeutet. Angesichts der relativen Ausgeglichenheit und des komplementären Charakters der genannten Vor- und Nachteile wird im folgenden sowohl eine zentrale als auch eine dezentrale Architektur für die in Kapitel 2 beschriebenen Regelkonzepte präsentiert.

---

gründet, dass der Leser von einem ernsthaften Systementwurf nichts lernen könne! (siehe [BB98, S. 194])

3.2 Zentrale Regelverarbeitung

In diesem Abschnitt wird eine zentrale Architektur zur Realisierung eines Regelverarbeitungssystems für verteilte Anwendungen vorgestellt. Teilabschnitt 3.2.1 erläutert zunächst den Gesamtaufbau des Systems und motiviert die gewählte Schichtenarchitektur. Teilabschnitt 3.2.2 beschreibt die Hauptaspekte der Funktionsschicht. Teilabschnitt 3.2.3 geht dann näher auf die Komponenten der Dienstschicht, insbesondere auf den Dienst zur zentralen Verwaltung von Eigenschaften, ein. In Teilabschnitt 3.2.4 wird dargestellt, wie eine automatische Aktivierung von Regeln unter Verwendung eines Ereigniskanals erreicht werden kann.

3.2.1 Gesamtarchitektur des *Policy–Managers*

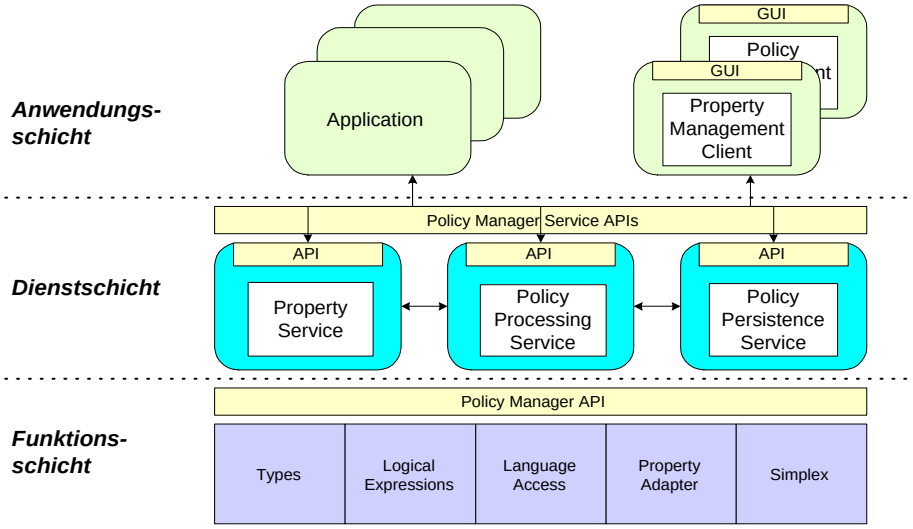


Abbildung 3.6: Schichtenarchitektur des Policy–Managers

Das im Rahmen dieser Arbeit entwickelte zentrale Regelverarbeitungssystem wurde zunächst nur mit dem Regeltyp *Policy–Rule* (vgl. Abschnitt 2.3.1.3) umgesetzt, da dieser — wie bereits argumentiert — am besten als ein generischer Mechanismus zur Steuerung verteilter Anwendungssysteme eingesetzt werden kann. Dementsprechend wurde das erste Prototypsystem als "Policy–Manager" getauft und im Laufe der Zeit weiterentwickelt, wodurch der Name erhaltenblieb[4].

Der Policy–Manager besitzt insgesamt eine Schichtenarchitektur, die in Abbildung 3.6 abstrakt dargestellt ist und (von unten nach oben) aus folgenden Schichten zusammengesetzt ist:

Funktionsschicht Diese Schicht stellt alle Hauptfunktionen zur Verarbeitung und Verwaltung von Regeln in Form von Systembibliotheken zur Verfügung,

[4]Die Grundfunktionalität des Policy–Managers war jedoch von Anfang an generisch für die Semantik des allgemeinen Regeltyps (siehe Abschnitt 2.3.1.2) konzipiert, so dass die Integration der anderen konkreten Regeltypen keine wirkliche Erweiterung darstellte.

die in dieser Architektur ausschließlich von der direkt darüber liegenden Dienstschicht genutzt werden.

Dienstschicht Die Dienstschicht stellt die Funktionalität zur Verarbeitung von Regeln und Eigenschaften in Form von Diensten bzw. Servern zur Verfügung, die zur Laufzeit von den Komponenten der Anwendungsebene aufgerufen werden können.

Anwendungsschicht Auf der Anwendungsschicht sind Applikationen, die gegenüber der den Dienstschichtkomponenten als Dienstnutzer bzw. Klienten auftreten, angesiedelt. Die Anwendungsschicht stellt außerdem (graphische) Standardwerkzeuge zur Verwaltung von Regeln und Eigenschaften dem Benutzer zur Verfügung.

Die Funktions- und Dienstschichten implementieren jeweils einen Satz von Anwendungsprogrammierschnittstellen (API), deren Kenntnis genügt, um die angebotene Funktionalität zu nutzen, ohne über Implementationsdetails informiert zu sein. Der wesentliche Vorteil dieser Schichtenarchitektur besteht darin, dass durch die *doppelte* Verlagerung von Standardfunktionalität von Anwendungen zu Diensten einerseits und von Diensten zu Bibliotheken andererseits ein hohes Maß an Flexibilität und Implementationstransparenz erreicht wird. Neben der bereits erwähnten Entlastung der Anwendungen von Regelmechanismen — wenn sie zur Laufzeit gerade keinen Gebrauch von der Dienstschicht machen — drückt sich der Vorteil der letzteren Funktionsverlagerung darin aus, dass hinter der Schnittstelle der Funktionsschicht gleichzeitig mehrere Bibliotheken, die die gleichen Signaturen auf unterschiedliche Weise implementieren, verwendet werden können. In der Tat wurde für die Basisausdrucksmächtigkeit und die erweiterte Ausdrucksmächtigkeit (vgl. Abschnitte 2.3.1.4 und 2.3.3.1) jeweils eine eigene Bibliothek (*PM Library* bzw. *EPM library*) realisiert. Mit Hilfe des *Parsers*, der ebenfalls zur Bibliothek gehört, wird erkannt, welcher Ausdrucksmächtigkeitsklasse ein Regelausdruck angehört. Dementsprechend kann diese Regel, wie in Abbildung 3.7 veranschaulicht, an die jeweils geeignete Bibliothek weiter geleitet werden. (Die Aufspaltung in unterschiedliche Bibliotheken geschieht sowohl für die Anwendungs- als auch für die Dienstebene völlig transparent, da die Parserklasse selbst von der Bibliothek angeboten wird.) Eine solche Vorgehensweise bringt in der Praxis einen erheblichen Effizienzgewinn, da die Algorithmen zur Verarbeitung der erweiterten Ausdrucksmächtigkeit deutlich komplexer und rechenaufwendiger sind als die bzgl. der Basismächtigkeit.

3.2.2 Komponenten der Funktionsschicht

Die Basis des Gesamtsystems bilden portable Systembibliotheken, die in der *Funktionsschicht* angesiedelt sind. Eine solche Bibliothek implementiert jeweils die gesamte operationale Programmierschnittstelle *Policy Manager API* und besteht aus einer Menge von Klassen*paketen*, die folgende Funktionsgruppen umfasst:

- Typsystem (*Types*): Obwohl bei der Modellierung der logischen Verarbeitungsfunktionen in Kapitel 2 nur allgemein zwischen numerischen und nicht–numerischen Eigenschaftswerten unterschieden wurde, ist es für eine konkrete Implementation der Funktionalität erforderlich, ein ausgeprägtes

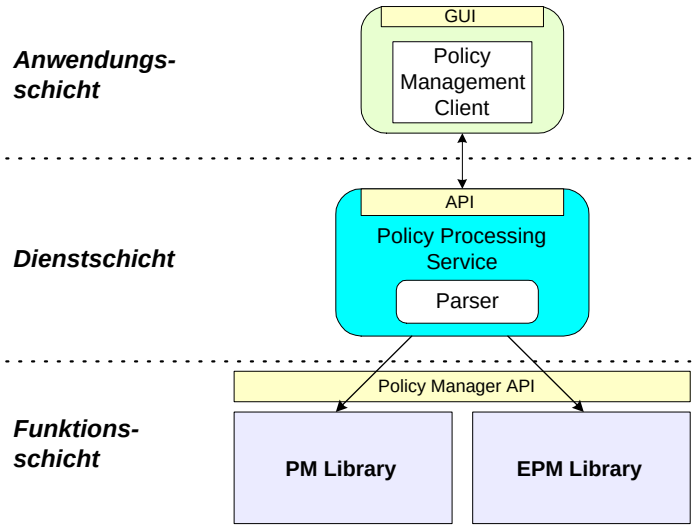


Abbildung 3.7: Transparente Benutzung unterschiedlicher Policy–Manager–Bibliotheken

Typsystem zu definieren, denn Missachtung von Typsemantik und Typkompatibilität kann sowohl logisch als auch programmiertechnisch zu schweren Fehlern führen. Deshalb wurde für die Bibliotheken ein Typsystem entwickelt, das alle Typen umfasst, die von den Verarbeitungsfunktionen behandelt werden können bzw. auf denen Regeln ausgedrückt werden können. Dazu gehören neben Basistypen auch strukturierte und benutzerdefinierte. (Einzelheiten sind in Kapitel 4 beschrieben.)

- Logische Ausdrücke und Funktionen (*Logical Expressions*): Diese Gruppe besteht aus Funktionspaketen, die insgesamt eine *objektorientierte* Umsetzung und Verarbeitung von logischen Ausdrücken erlauben. D.h. logische Ausdrücke werden zur Laufzeit mittels einer Klassenhierarchie in Programmobjekte transformiert, die nicht nur Typ und Wert des jeweiligen Teilausdrucks repräsentieren, sondern auch die passenden Verarbeitungsfunktionen dafür anbieten. Diese Verknüpfung von Daten mit der dazu passenden Funktionalität erleichtert den korrekten Umgang mit den logischen Funktionen, deren Semantik nicht jedem Anwendungsprogrammierer vertraut ist. Beispielsweise sind mit der *Rule*–Klasse, die im Paket *rules* enthalten ist und eine *beliebige* logische Verknüpfung von atomaren Ausdrücken (mit den Operatoren *AND* und *OR*) darstellt, Methoden verknüpft, um einen solchen Ausdruck in eine passende Normalform — wie etwa DNF — zu transformieren. Eine solche Methode würde allerdings auf der Ebene einzelner Atome keine Anwendung finden.

- Sprachliche Zugänge (*Language Access*): Da Regeln zur Steuerung verteilter Anwendungen nicht in die Applikationen einprogrammiert werden, sondern vorwiegend von den *Benutzern* mit Hilfe graphischer Werkzeuge (wie des *Policy Management Client* auf der Anwendungsschicht) formuliert und zur Laufzeit aktiviert werden sollen, ist ein sprachlicher Zugang zum

Regelverarbeitungssystem unerlässlich. Deshalb wurde für beide Bibliotheken des Policy–Managers je eine solche Sprache und ein entsprechender Parser entwickelt, mit Hilfe dessen Regeln in textueller Form erkannt und generiert werden können. Die Bestandteile der Sprache korrespondieren weitgehend mit den Elementen der Klassenhierarchie zur Verarbeitung von Regeln als Programmobjekte und der Parser ist in der Lage, die Transformation zwischen textueller Repräsentation und Objektimplementation in beiden Richtungen vorzunehmen.

- Eigenschaftsdienstadapter (*Property Adaptor*): Eigenschaften von Anwendungen, über denen Steuerungsregeln formuliert werden, können vielen unterschiedlichen Quellen entstammen. Obwohl der Policy–Manager einen standardisierten Eigenschaftsdienst zum Zweck der zentralen Verwaltung von Eigenschaften beliebiger Anwendungen zur Verfügung stellt (siehe Abschnitt 3.2.3.2), sind die Applikationen bzw. Dienstnutzer des Policy–Managers nicht gezwungen, diesen Eigenschaftsdienst zu verwenden. Dies wird durch eine *Adapterschicht* ermöglicht, die als eine Programmierschnittstelle zwischen dem Eigenschaftsdienst und der Funktionsschicht bereitgestellt ist. Damit können als Eigenschaftsdienst beliebige Komponenten benutzt werden, die diese Adapterschnittstelle erfüllen. Beispielsweise wurde für Testzwecke ein *JavaAdaptor* implementiert, der die in dem Java–Laufzeitsystem enthaltenen *System Properties* als Eigenschaftsspeicher für Testanwendungen nutzt.

- Simplex–Methoden (*Simplex*): Das Paket *Simplex* ist das einzige, das nur der erweiterten Bibliothek (*EPM Library*) angehört. Es stellt Klassen zur Verfügung, die die Simplexalgorithmen implementieren. Diese Algorithmen werden (nur) benötigt, um Regeln der erweiterten Ausdrucksmächtigkeit zu verarbeiten.

Die genaue Beschreibung der einzelnen Pakete (*Packages*), die zu diesen Funktionsgruppen gehören, erfolgt in Abschnitt 4.2.

3.2.3 Komponenten der Dienstschicht

In diesem Abschnitt werden architekturelle Aspekte der Dienstschicht näher erläutert. Teilabschnitt 3.2.3.1 stellt zunächst den als CORBA–Dienst konzipierten Policy–Manager vor. In Teilabschnitt 3.2.3.2 wird dann der Eigenschaftsdienst als eine unabhängige Komponente, deren Funktionalität bisher noch nicht beschrieben wurde, dargestellt.

3.2.3.1 Der Policy–Manager als CORBA–Dienst

Wie bereits erwähnt bietet die Dienstschicht den Anwendungskomponenten Regelverarbeitungsfunktionalität an, die nur *zur Laufzeit* genutzt wird und deshalb nicht mit den Applikationen zusammen kompiliert werden muss. Um eine solche dynamische Nutzbarkeit zu gewährleisten, muss ein geeigneter Kommunikationsmechanismus, der die Verbindung zwischen Dienstnutzern und Dienstanbietern herstellt, gewählt werden. Da es sich bei der Anwendungsschicht um verteilte Komponenten handelt, muss zumindest ein entfernter Aufrufmechanismus (engl. *Remote Procedure Call* bzw. RPC) verwendet werden. Da es sich außerdem

hier um eine technisch offene, d.h. *heterogene* Anwendungsumgebung handelt (vgl. Abschnitt 1.2), bietet sich die in Abschnitt 1.3.3 vorgestellte Middlewareplattform CORBA an, die sich durch ein hohes Mass an Sprachunabhängigkeit und Systeminteroperabilität auszeichnet. Mit CORBA als Verteilungsplattform wird es u.a. möglich sein, den Policy–Manager als einen Systemdienst für Applikationen, die in (de–facto) beliebigen Programmiersprachen implementiert sind und auf beliebigen Betriebssystemen laufen, zu realisieren.

Wie in Abschnitt 1.3.3 bereits erwähnt, wird eine solche flexible Kommunikationsfähigkeit durch den *Object Request Broker* (ORB) zur Verfügung gestellt. Mit Hilfe des ORB kann ein dienstnutzendes Objekt in einer verteilten, heterogenen Umgebung das gewünschte Zielobjekt auffinden, es eventuell aktivieren und mit ihm kommunizieren, als ob es ein lokales Objekt wäre. Die gesamte Aufgabe der Transformation der Daten zwischen Quell-, Netzwerk- und Zielformat wird dabei von so genannten *Stub*–Objekten übernommen, die automatisch aus den Schnittstellen der Anwendungsobjekte generiert werden. Unterschiedliche Systeme, in den jeweils ein ORB installiert ist, können mittels des *Internet Inter–Orb Protocol* (IIOP) dynamisch miteinander verbunden werden, so dass eine für die Anwendungen ebenso transparente *systemübergreifende* Kommunikation — und damit eine wirklich globale objektorientierte Verteilungsplattform — bereits realisiert ist.

Zusätzlich zum ORB als Kommunikationsmechanismus wurden im Rahmen des CORBA–Standards eine Reihe von Systemdiensten, die von beliebigen Anwendungen genutzt werden können, spezifiziert bzw. vorgeschlagen. Dazu gehören die *Common Object Services* und *Common Facilities* — insbesondere die *horizontalen* unter letzteren (vgl. Abschnitt 1.3.3). Beide Dienstgruppen sollen generische, standardisierbare Funktionen auf eine dynamisch zugreifbare Weise anbieten, wobei ihre Unterscheidung nur durch die Eigenschaft der "Anwenderbezogenheit" begründet ist[5]. Wenn also CORBA als Verteilungsplattform gewählt wird, erscheint es am sinnvollsten, den Policy–Manager architekturell als einen solchen Systemdienst umzusetzen. In der Tat wurde bereits in der Common Facilities Architecture von 1995 (siehe [OMG95]) eine *Rule Management Facility* als Bestandteil der Gruppe *Task Management Common Facilities* vorgeschlagen. Jedoch liegt bis heute keine entsprechende Spezifikation vor[6].

Die Architektur des Policy–Managers als eines CORBA–Systemdienstes ist in Abbildung 3.8 dargestellt. Die gesamte Funktionalität ist in drei Komponenten aufgeteilt: den *Property Service* zur Verwaltung von Anwendungseigenschaften und die beiden Policy–Dienste zur Verarbeitung und Verwaltung von Regeln. Der *Policy Persistence Service* realisiert einen persistenten Speicher für benamte Regelobjekte, die in unterschiedliche Namensräume, z.B. für unterschiedliche Applikationen, gruppiert werden können. Der *Policy Processing Service* bietet die gesamte Funktionalität der Funktionsschicht als CORBA–Dienst an. Der Property Service wird im nächsten Abschnitt näher erläutert. Alle drei Dienste sind über entsprechende Schnittstellen in CORBA–IDL aufzurufen und bilden insgesamt die Schnittstelle des CORBA–Policy–Managers. Die einzelnen

[5] So werden in [OMG97, S. 4–3] die Common Facilities als "interfaces for horizontal end–user–oriented facilities applicable to most application domains" bezeichnet.

[6] Im Gegensatz zu der produktiven und recht erfolgreichen Standardisierung von Common Object Services (bisher über zwanzig Dienste) scheint die Spezifikation von Common Facilities insgesamt kaum voran gekommen zu sein (bisher nur einer). Dies könnte auf die genannte vage bzw. nicht–technische Unterscheidung der beiden Gruppen zurückzuführen sein.

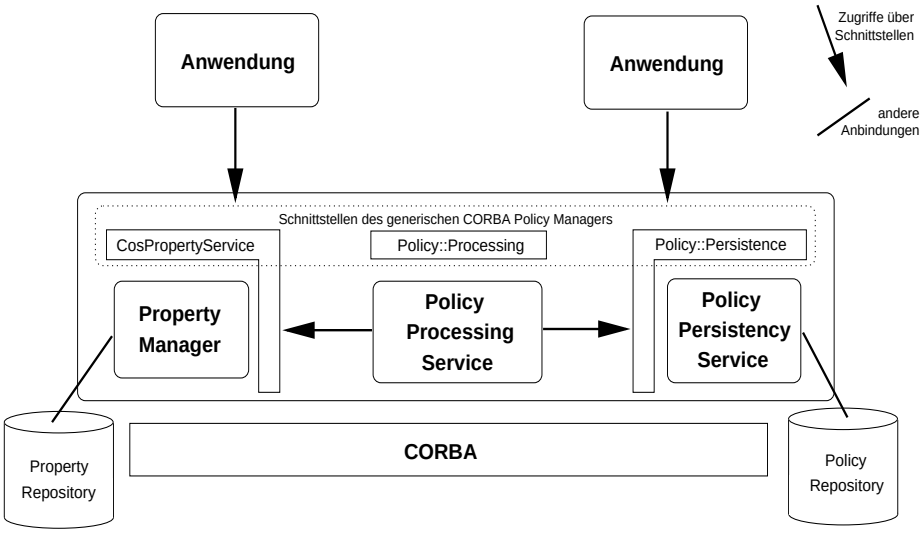


Abbildung 3.8: Der Policy–Manager als CORBA–Dienst

Schnittstellen sind in Abschnitt 4.2 beschrieben und im Anhang zu finden.

3.2.3.2 Eigenschaftsdienst

An dieser Stelle soll nun die Bedeutung von Eigenschaften für verteilte Anwendungen und ihre Verarbeitungsmöglichkeiten mittels eines generischen Dienstes behandelt werden.

Die Verarbeitung von Eigenschaften verteilter Anwendungen hat in der Praxis mittlerweile eine so zentrale Bedeutung erlangt, dass sie für die meisten Systeme unverzichtbar erscheint und deshalb häufig gar nicht als ein gesonderter Unterstützungsmechanismus, sondern als inhärenter Teil des Anwendungs- bzw. Entwicklungsprozesses wahrgenommen wird. Mit der — bereits erwähnten — zunehmenden Dynamik und Offenheit verteilter Systeme hat sich die Eigenschaftsverarbeitung bei dem derzeitigen Stand der Technik als *der* Konfigurationsmechanismus für jegliche Softwareanwendungen etabliert, da sie eine generische, leicht zu beherrschende und dennoch wirksame Möglichkeit zur Anpassung von funktionalem Verhalten darstellt. Eine "Eigenschaft" benennt dabei im allgemeinen ein beliebiges Merkmal eines Objektes und dient gleichzeitig als Platzhalter für dessen (aktuelle) Ausprägung. Z.B. hat die Eigenschaft "Farbe" vom Objekt "Rose" die Ausprägung "Gelb". Eigenschaften, von Softwaresystemen wie auch von materiellen und anderen Gegenständen, verkörpern also meistens intuitive Assoziationen mit dem jeweiligen Gegenstand, die jedem Benutzer vertraut sind und deshalb oft leichter zu verwenden sind als etwa Methoden einer Softwarekomponente.

Es gibt heute in fast jedem praxisrelevanten System, vor allem solchen im Java–Umfeld, Mechanismen zur Verwaltung und Modifikation von Eigenschaften, die, abhängig vom jeweiligen Produkt oder Hersteller, unter vielen Namen wie etwa "Properties", "System Properties", "Property Descriptor", "Attributes", "Customizing", "(Windows) Registry" etc. der Anwendungsebene bereitgestellt werden. Selbst der Begriff "Policies" wird häufig benutzt, um Eigen-

schaften zu bezeichnen, deren Ausprägungen verschiedene *Optionen* in Bezug auf die Funktionsweise eines Systems darstellen[7]. Jedoch ergibt sich aus dieser Vielfalt an Anwendungsmöglichkeiten dringend der Bedarf nach einer einheitlichen bzw. standardisierten Modellierung von Eigenschaften sowie einer Auslagerung der gesamten Eigenschaftsverwaltung in einen generischen Dienst, der beliebige Anwendungen unterstützen kann. Eine solche Auslagerung würde zumindest die technische Interoperabilität zwischen Eigenschaften verschiedenen Anwendungen erheblich steigern, beispielsweise können interoperable Typen für Eigenschaften wie "Farbe" oder "Währung" definiert werden, und einen einheitlichen Zugriff auf Eigenschaften einer Anwendung, auch durch andere Anwendungen (falls erlaubt), ermöglichen. Ein weiterer Vorteil, der sich aus der Entkoppelung von Eigenschaften von den betreffenden Anwendungen ergibt, ist die Möglichkeit, einem bestehenden, unmodifizierbaren Objekt beliebig viele Eigenschaften dynamisch zuzufügen[8]. Dies ist besonders sinnvoll bei Eigenschaften, die ohnehin "von außen" mit dem Objekt assoziiert werden — wie z.B. "Preis", "Qualität" oder "Aktualität".

Der CORBA *Property Service* Genau zum Zweck einer Auslagerung von Anwendungseigenschaften mit den erwähnten Vorteilen wurde im Rahmen der CORBA–Standardisierung ein generischer Eigenschaftsdienst spezifiziert, der als "Property Service" bezeichnet wird. Nach der Dienstspezifikation sind Eigenschaften wie folgt definiert:

> Properties are typed, named values dynamically associated with an object, outside of the type system. [OMG98b, S. 13–1]

Mit "outside of the type system" ist hier jedoch nicht gemeint, dass Eigenschaften keinem Typsystem unterliegen, sondern offensichtlich nur, dass der Typ einer Eigenschaft, die einem Objekt (dynamisch) zugeordnet ist, nicht aus dem Objekttyp abgeleitet werden kann.

In Abbildung 3.9 ist die Definition einer CORBA–Eigenschaft in IDL angegeben. Um einen wirklich generischen Dienst zu gewährleisten, müssen Eigenschaftswerte von beliebigen Typen erlaubt sein. Deshalb wird hier vom *any*–Typ, der einen beliebigen, in IDL spezifizierten Typ repräsentiert, Gebrauch gemacht. Desweiteren macht diese IDL–Definition deutlich, dass eine CORBA–Eigenschaft nicht nur aus einem Name–Wert–Paar besteht, sondern optional auch einen Zugriffsmodus enthalten kann.

Aufbauend auf dieser Definition werden dann, ebenfalls in IDL, Schnittstellen zur Generierung und Verarbeitung von *Eigenschaftsmengen* (PropertySetFactory und PropertySet) spezifiziert, die einen einheitlichen Zugriff auf Eigenschaften beliebiger Anwendungen ermöglichen. Diese Schnittstellen und ihre *persistente* Realisierung, die von der CORBA–Spezifikation berücksichtigt ist, werden in Kapitel 4 beschrieben. Die in Abbildung 3.8 als "Property Manager" bezeichnete Komponente stellt eine persistente Implementation des gesamten

[7] Ein Beispiel hierfür sind die gesamten Policies des OMG–Traders [OMG96]. Z.B. bezeichnet *max_hop_count* (deren Wert vom Typ *unsigned long* ist) die maximale Anzahl von Links, die ein Trader zu anderen entfernten Tradern aufbauen darf.

[8] In [Sie96, S. 221] wird dies sogar als charakteristisches Merkmal von Eigenschaften im Vergleich zu *Attributen* dargestellt.

```
typedef string PropertyName;

struct Property                // normale Eigenschaft
  {
    PropertyName propertyName;
    any          propertyValue;
  };

enum PropertyModeType
  {
    normal,                    // lösch- und änderbar
    read_only,                 // nicht änderbar
    fixed_normal,              // nicht löschbar
    fixed_readonly,            // nicht lösch- und änderbar
    undefined
  };

struct PropertyDef    // zugriffsbeschränkte Eigenschaft
  {
    PropertyName     propertyName;
    any              propertyValue;
    PropertyModeType propertyMode;
  };
```

Abbildung 3.9: IDL–Definition von Eigenschaften (Properties)

CORBA Property Service dar, d.h. sowohl Eigenschaften als auch Eigenschaftsmengen sind in einem dauerhaften *Repository* gespeichert und können somit über die Laufzeit einzelner Applikationen hinaus (wieder) verwendet werden.

3.2.4 Möglichkeiten des Zugriffs auf den Policy–Manager

In diesem Abschnitt soll nun die Frage erörtert werden, auf welche Weise die Funktionalität des zentralen Policy–Managers von der Anwendungsebene konkret genutzt werden kann. Dabei wird sich herausstellen, dass die bisher diskutierte Architektur um eine zusätzliche Aktivierungsschicht erweitert werden soll, um eine automatische, d.h. für die Komponenten der Anwendungsebene völlig transparente, Aktivierung von Regeln zu ermöglichen.

3.2.4.1 Zugriff über operationale Schnittstellen

Die in Abbildung 3.6 gezeigte Grundarchitektur des Policy–Managers ermöglicht zunächst einen direkten Zugriff auf die Regelverarbeitungsfunktionen über *operationale* Schnittstellen. D.h. die Komponenten in der Anwendungsebene können entsprechende Methoden der Komponenten der unteren Schichten, insbesondere der Dienstschicht, direkt aufrufen, um neue Regeln persistent abzulegen oder um Regeln unterschiedlicher Kooperationspartner zu unifizieren und dann ge-

meinsam zu aktivieren etc.

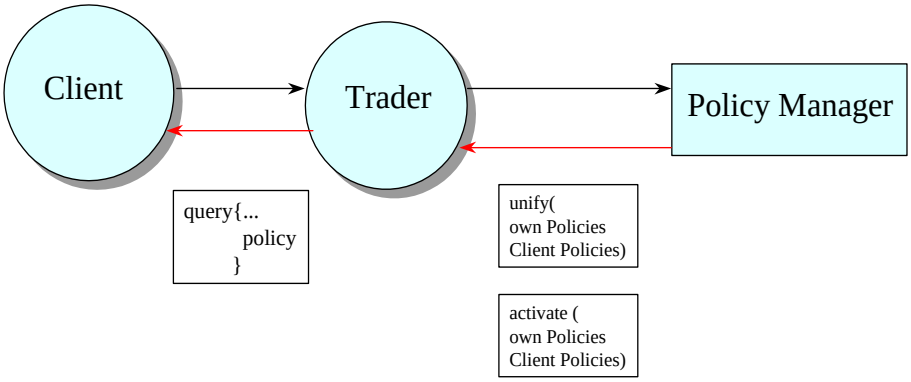


Abbildung 3.10: Verwendung des Policy–Managers über IDL–Schnittstellen

Abbildung 3.10 illustriert dies exemplarisch anhand der Interaktion zwischen einem *Trader* und einem Klienten, der über diesen einen geeigneten Dienstanbieter finden will. Hierbei übergibt der *Client* seine *Policy* als einen Parameter der *query*–Methode, die zur Dienstsuche beim *Trader* dient. Die Policy kann beispielsweise ausdrücken, dass die Anzahl der aufzubauenden *Links* zu entfernten Tradern auf fünf begrenzt sein soll: $hop\_count \leq 5$. Der Trader extrahiert diese Policy aus der übergebenen Parameterliste, sucht sich die eigene Policy bzgl. der *query*–Methode aus, die z.B. das Konjunkt $hop\_count \geq 3$ enthält, und ruft beim Policy–Manager die Methode *unify* auf, um die beiden Policies zu unifizieren. Falls das Ergebnis wie in diesem Fall positiv ist, ruft der Trader dann die Methode *activate* auf, um die resultierende Policy zu aktivieren. Als Ergebnis wird die Eigenschaft *hop_count* des Traders bzgl. dieser Suche auf 5 gesetzt.

Diese Zugriffsweise ist völlig konform zum allgemeinen Dienstbegriff, der die operationale Sicht auf Dienstanbieter betont (vgl. Abschnitt 2.1.2), und entspricht auch der von CORBA intendierten Verwendung von *Object Services*. Das Trader–Beispiel macht jedoch deutlich, dass hier ein wichtiger Aspekt des in Abschnitt 2.3.1.2 dargestellten Regelkonzepts nicht zum Tragen kommt, nämlich die explizite Verknüpfung von *Ereignissen* mit Regeln. Triggernde Ereignisse sind in diesem Beispiel nur implizit zu erkennen, und zwar in Verbindung mit den Aufrufen von Methoden wie *query*. Die fehlende Ereignisorientierung bei dieser Verwendungsweise stellt an sich noch keinen grossen Nachteil dar, allerdings ist sie eng verknüpft mit der eigentlichen Schwäche dieser Zugriffsart, die in der fehlenden *Transparenz* der Regelmechanismen gegenüber den Anwendungskomponenten besteht. Sowohl der *Client* als auch der *Trader* müssen hier direkt mit den Policies umgehen, d.h. die gesteuerten Objekte müssen mehr oder weniger die Steuerungsmittel selbst bedienen können.

3.2.4.2 Zugriff über ereignisorientierte Schnittstellen

Um den in der konkreten Anwendungspraxis gravierenden Nachteil des Zugriffs auf den Policy–Manager über operationale Schnittstellen aufzuheben, bedarf es einer Erweiterung der bisherigen Architektur um eine so genannte *Aktivierungsschicht*, die zu der in Abbildung 3.11 gezeigten Architektur führt.

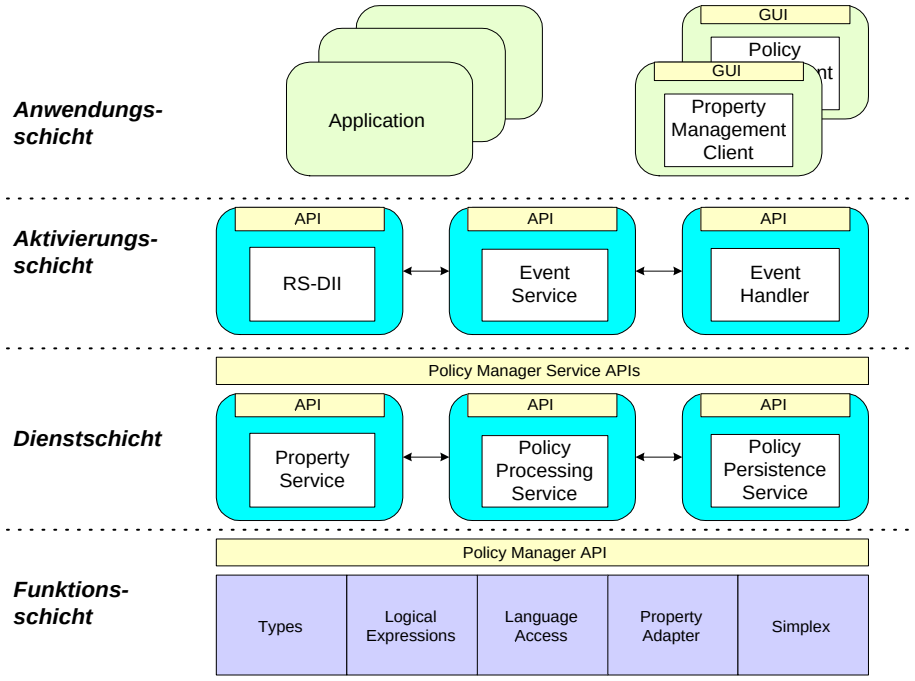


Abbildung 3.11: Erweiterung der PM–Architektur um eine Aktivierungsschicht

Die zusätzliche Aktivierungsschicht, die zwischen der Dienst- und Anwendungsschicht *optional* genutzt werden kann, besteht aus drei Hauptelementen:

- einem Ereignisdienst (*Event Service*): Um die explizite Modellierung von Ereignissen als Auslöser für die Regelaktivierung umzusetzen, werden Mechanismen zur Generierung, Übertragung und Verarbeitung von Ereignissen als einen eigenständigen Objekttyp zur *entkoppelten, asynchronen* Kommunikation benötigt. Die Behandlung von Ereignissen bzw. *Events* hat für aktuelle Anwendungssysteme eine ähnlich grundlegende Bedeutung wie die Verarbeitung von Eigenschaften (vgl. Abschnitt 3.2.3.2), da in Bezug auf die zunehmende Offenheit und Dynamik der Systeme die entkoppelte Kommunikation per Events häufig die effektivste darstellt. Wiederum analog zu den Eigenschaften ergibt sich immer mehr der Bedarf nach einer generischen und standardisierten Behandlung von Ereignissen in verteilten Systemen. Zu diesem Zweck wurde der *CORBA Event Service* entworfen und inzwischen standardisiert. In dieser Architektur wird deshalb dieser Dienst zur Behandlung von Ereignissen verwendet.

- einer Ereignisbehandlungskomponente (*Event Handler*): Neben dem Event Service, der einen generischen Kanal (engl. *event channel*) zum Verteilen jeglicher Eventtypen bereitstellt, werden zur *Interpretation* der Event–Inhalte und zur Abbildung dieser Inhalte auf entsprechende Regelmechanismen spezielle Behandlungskomponenten benötigt, die als Event Handler bezeichnet werden.

- einem Rule–sensitiven Aufrufmechanismus (*RS–DII*): Schließlich müssen

Eventmechanismen in den Aufrufmechanismus integriert werden, da Methodenaufrufe semantisch den größten Teil der relevanten Ereignisse in einem Anwendungssystem darstellen. Hierfür wird eine modifizierte Version des *Dynamic Invocation Interface* (DII), das einen flexiblen und dynamischen Mechanismus zur Umsetzung von entfernten Methodenaufrufen in vielen Verteilungsplattformen, insbesondere den CORBA–konformen, anbietet. Die Modifikation betrifft jedoch ausschließlich die interne Funktionsweise des DII–Mechanismus (nämlich Anreicherung um Rule– und Event–Funktionalität), d.h. die nach außen angebotene Schnittstelle des DII bleibt für die Anwendungskomponenten in ihrer ursprünglichen Form erhalten. Auf diese Weise wird eine für die Komponenten der Anwendungsschicht vollständig transparente Steuerung des Systemverhaltens mittels Regelmechanismen ermöglicht. Diese erweiterte Version des DII wird als *Rule–Sensitive Dynamic Invocation Interface* bzw. RS–DII bezeichnet.

Der genaue systemtechnische Entwurf sowie die Implementierung dieser Komponenten wird in [Gro98] erläutert. Abbildung 3.12 illustriert anhand des oben eingeführten Trader–Beispiels, wie mit Hilfe der Aktivierungsschicht nun eine *automatische* Aktivierung der Policies ohne Beteiligung der Anwendungskomponenten erzielt werden kann.

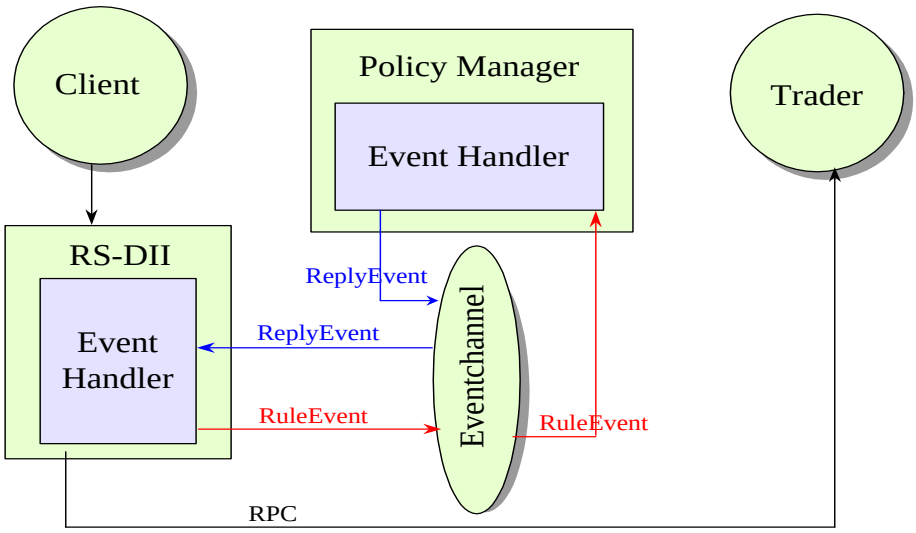


Abbildung 3.12: Automatische Aktivierung von Regeln

Die Policies des *Client*s als auch des *Trader*s werden vor der Interaktion dieser Komponenten beim Policy–Manager registriert (entweder durch diese Komponenten oder aber auch durch entsprechende Anwender bzw. Administratoren). Der *Client* ruft beim *Trader* die *query*–Methode mit einer Signatur, die keinen Policy–Parameter enthält, mittels des RS–DII auf. Bevor der Aufruf (per DII) weitergereicht wird, erzeugt diese Komponente mittels des Event–Handlers ein entsprechendes *RuleEvent*, das dem Policy–Manager signalisieren soll, dass ein relevantes Ereignis vorliegt. Ein solches Event wird über den Ereigniskanal nur dann beim Policy–Manager ankommen, wenn er sich für den entsprechenden

EventType registriert hatte. In diesem Fall wird der Event–Handler des Policy–Managers aktiv, der aufgrund des Ereignistyps und der in dem RuleEvent mit übergebenen Identitäten von Quelle und Ziel des Methodenaufrufs feststellen kann, welche Policies zu unifizieren und aktivieren sind. Nach der Aktivierung schickt dieser ein *ReplyEvent* über den Ereigniskanal zum RS–DII zurück, das erst jetzt den Aufruf zum *Trader* weiterleitet. An diesem Ablauf ist zu erkennen, dass die Komponenten der Anwendungsschicht, *Client* und *Trader*, von den Regelmechanismen komplett entbunden sind. Die Steuerungseffekte machen sich erst dann für sie bemerkbar, wenn eine Aktivität aufgrund einer Regelverletzung verhindert wird. Der *Client* bekommt in diesem Fall eine *RuleException* als Ergebnis, wenn die *query*–Methode wegen der fehlschlagenden Policy nicht ausgeführt wurde.

3.3 Dezentrale Regelverarbeitung

In diesem Abschnitt wird als Erweiterung der dargestellten zentralen Regelverarbeitung mittels des so genannten Policy–Managers eine dezentrale Architektur zur regelbasierten Steuerung verteilter Anwendungen vorgestellt. Abschnitt 3.3.1 beschreibt zunächst das Konzept Rule–sensitiver Anwendungen. Verschiedene Modi für die lokale und entfernte Aktivierung von Regeln als autonomen, aktiven Objekten werden in Abschnitt 3.3.2 beschrieben. Die Integration von Rule–sensitiven Anwendungsobjekten in eine entsprechende Middleware–Architektur ist Gegenstand des Abschnitts 3.3.3.

3.3.1 Das Konzept Rule–sensitiver verteilter Anwendungskomponenten

Aus Sicht eines Endbenutzers ist die Steuerung von Anwendungskomponenten mit Regelmechanismen sicherlich am einfachsten und durchschaubarsten, wenn er Regeln zu jeder Zeit *direkt* auf den Anwendungen definieren kann, ohne irgendeinen Systemdienst wie den im letzten Abschnitt dargestellten Policy–Manager zu verwenden, der möglicherweise eine große Zahl von Anwendungen steuert und somit für den Benutzer zu komplex erscheinen kann. Wenn die Regeln für eine Anwendung direkt bei dieser mit verwaltet werden, dann ist i.a. eine hohe Benutzerfreundlichkeit leicht zu gewährleisten, da die Anzahl der Regeln immer auf das notwendigste Maß begrenzt und die Wirkung der Regeln unmittelbar zu erkennen ist. Darüber hinaus können bei einem solchen dezentralen Management bestehende Regeln, deren Wirkung aus Benutzersicht sich im Laufe der Zeit bewährt haben, von einer Anwendung zur nächsten transferiert werden, ohne sie neu definieren zu müssen.

Die Weiterführung dieser Betrachtung von direkt steuerbaren Anwendungen führt zum Konzept *Rule–sensitiver verteilter Anwendungskomponenten* als technische Realisierung von Anwendungen, die einzeln mit Regeln angereichert werden können und deren Interaktion in einer verteilten Umgebung durch Regeln unterstützt werden kann [TGML99]. Eine Rule–sensitive Komponente ist also anschaulich ausgedrückt eine, die direkt auf Rules "reagiert", die ihr von außen "auferlegt" werden. Im folgenden betrachten wir zunächst die genauen Anforderungen an ein solches Konzept und die zur dessen Umsetzung benötigten Basiskomponenten.

3.3.1.1 Anforderungen

Um das zunächst abstrakt eingeführte Konzept der Rule–sensitiven verteilten Anwendungen zu präzisieren, müssen eine Reihe von Aspekten, die zugleich als Anforderungen an eine konkrete Umsetzung aufzufassen sind, berücksichtigt werden:

Autonomie Wie bereits beim Vergleich existierender Ansätze in Abschnitt 3.1.4 erwähnt wurde, besteht die Hauptmotivation für eine dezentrale Regelverarbeitung darin, die Autonomie der gesteuerten Anwendungen zu erhalten bzw. zu unterstützen. In diesem Sinne können Regeln für unterschiedliche Anwendungen auch unterschiedlich verwaltet und aktiviert werden. Mögliche Einschränkungen der Autonomie ergeben sich ausschließlich aus dem Bedürfnis, die Interaktion zwischen regelbasierten Anwendungen zu unterstützen (vgl. "Interaktionsorientierung"), da Autonomie und Interaktion naturgemäß miteinander konfligieren können.

Autonomie kann jedoch nicht nur auf die Rule–sensitiven Anwendungen, sondern gleichermaßen auch auf die Regeln selbst bezogen sein. Dies bedeutet, dass Regeln als *aktive* Objekte, die eine eigene Funktionalität besitzen, fungieren sollen (vgl. "Objektorientierung" und "Mobilität").

Generizität und Transparenz Generizität als eine der wichtigsten Anforderungen an wiederverwendbare Steuerungskonzepte ist in Abschnitt 1.4.1 im Sinne von "Anwendungsunabhängigkeit" eingeführt worden. In Bezug auf eine dezentrale Architektur, in der Regeln als semantische Ergänzungen für verteilte Anwendungen betrachtet werden können, bedeutet dies zusätzlich, dass ein möglichst breites Spektrum an Regelsemantik umgesetzt werden soll. Dies kann zum einen durch eine geeignete Auswahl an konkreten Regeltypen — in Verbindung mit geeigneten Aktivierungsmodi (siehe Abschnitt 3.3.2) — und zum anderen durch ein erweiterbares Regelsystem, wie es in Kapitel 2 konzipiert wurde, erreicht werden.

Die Generizität der entwickelten Mechanismen ist zugleich Voraussetzung für ihre *transparente* Anwendung. In Bezug auf die Dezentralisierung bedeutet Transparenz, dass Regelfunktionalität möglichst klar von Anwendungsfunktionalität getrennt und unabhängig von letzterer ausgeführt werden soll, so dass bestehender Code weiterhin benutzt werden kann. Allerdings kann diese Anforderung nicht absolut, sondern nur graduell erfüllt werden, da die Dezentralisierung impliziert, dass die Anwendungen zumindest in der Lage sind, Regeln bei sich aufzubewahren.

Interaktionsorientierung Die Unterstützung der semantischen Interaktion zwischen Anwendungen in einer offenen Umgebung ist ein Hauptziel dieser Arbeit. Im Kontext der dezentralen Regelverarbeitung bedeutet dies, dass Regeln unterschiedlicher Anwendungen diese jeweils so anpassen sollen, dass eine korrekte, konfliktfreie Interaktion der Anwendungen stattfindet. Um dies mit der Forderung nach Transparenz und Autonomie zu vereinbaren, muss die Anpassungsleistung von den Regeln erbracht werden. D.h. die Regeln als autonome Entitäten müssen die Eigenschaft der *Interaktionsorientierung* erfüllen.

Objekt– und Ereignisorientierung Um die Eigenschaften der Autonomie und Interaktionsorientierung zu erfüllen, erscheint es am sinnvollsten, Regeln nicht als passive Daten, die mit entsprechenden Funktionen verarbeitet werden, sondern als aktive Objekte, die sich selbst aktivieren können, zu realisieren. Objektorientierung folgt also als eine klare entwurfstechnische Konsequenz aus den abstrakteren Anforderungen. Eine andere Konsequenz ist die *Ereignisorientierung*, die nicht nur, wie bereits erwähnt, dem Regelkonzept inhärent erscheint, sondern auch sehr gut mit der Transparenz passt, denn eine ereignisbasierte Kommunikation zwischen Regelobjekten würde den Programmfluss der sie beherbergenden Anwendungen am wenigsten beeinträchtigen.

Dynamik Regeln sollen zu jedem Zeitpunkt zu einer laufenden Anwendung hinzugefügt bzw. wieder von ihr entfernt werden können. Abhängig von der jeweiligen Funktionalität des Regeltyps ermöglicht dies nicht nur einen flexiblen Kontroll- und Konfigurationsmechanismus, sondern auch eine dynamische Erweiterung der Anwendungssemantik. Beispielsweise kann so eine Online–Buchhandlung jederzeit um eine zusätzliche Benachrichtigungsfunktion erweitert werden, in dem ihr eine Aktionsregel hinzugefügt wird, die allen Interessenten eine E–mail verschickt, wenn das Ereignis "Buchtitel A trifft ein" stattgefunden hat.

Mobilität Da Regeln als aktive Objekte fungieren sollen, erscheint es konsequent, sie ebenfalls als *mobile* Objekte, die samt ihres aktuellen Zustands von einer Anwendung zur nächsten migrieren können, zu realisieren, denn somit kann die Forderung nach Dynamik auf eine sehr effiziente (und elegante) Weise erfüllt werden.

Transaktionalität Wenn die Aktivierung von Regeln, die durch gewisse Aktivitäten innerhalb der Anwendungen ausgelöst wurden, fehlschlägt, insbesondere wenn Regeln vom Typ Requirement– oder Policy–Rule verletzt werden, soll es möglich sein, die betreffenden Aktivitäten rückgängig zu machen, um einen möglichst konsistenten Zustand der Anwendung zu gewährleisten. D.h. Rule–sensitive Aktivitäten müssen transaktional durchgeführt werden.

3.3.1.2 Basiskomponenten

Um die o.g. Anforderungen an das Konzept Rule–sensitiver Anwendungen umzusetzen, lassen sich zunächst unabhängig von einer bestimmten Implementation als auch einer bestimmten Systemplattform eine Menge von (noch abstrakten) Komponenten identifizieren, die zur Realisierung einer Rule–sensitiven Anwendung benötigt werden. Die folgende Beschreibung der gewählten Menge an Basiskomponenten dient darüber hinaus dem Verständnis der später detailliert dargestellten Systemarchitektur und Implementation.

Rule–Container / Rule–Set Um die Anforderung der Dynamik zu erfüllen, müssen Rule–sensitive Anwendungen über bestimmte Funktionen verfügen, um jederzeit Regeln aufnehmen und beherbergen zu können. Diese Verwaltungsfunktionalität kann von einer getrennten Komponente, die als *Rule–Container* bezeichnet werden kann, bereitgestellt werden, auf die die

betreffende Anwendung Zugriff hat. Da eine Anwendung von vielfältigen Regeln, die unterschiedlichen semantischen Kategorien — z.B. Budgetverwaltung, Verhandlungsstrategien, Kooperations- und Sicherheitsprotokolle etc. — zugeordnet sind, beeinflusst werden kann, ist es sinnvoll, dem Rule–Container eine *Struktur* zu geben, die beliebig viele hierarchisch angeordnete Mengen umfassen kann. Um diese hierarchische Struktur deutlich zu machen, wird der Begriff *Rule–Set* verwendet. "Rule–Container" wird also synonym mit der "Root–Rule–Set", der Wurzel der Mengenhierachie, verwendet.

Rule–Objekte Aus den o.g. Anforderungen ergibt sich, dass Regeln als typisierte, aktive Objekte realisiert werden. Diese Objekte werden im folgenden als *Rule–Objekte* bezeichnet. Um sich selbst aktivieren und somit die Anforderung der Autonomie zu erfüllen, müssen Rule–Objekte als *Ereignisempfänger (engl. event listener)* konzipiert werden. Gemäß des Regelkonzeptes in Kapitel 2 sollen alle Rule–Objekte ihre gemeinsame logik–basierte Verarbeitungsfunktionalität von einer abstrakten Basisklasse erben, die zugleich die allen Regeln gemeinsamen Attribute wie z.B. Name oder Aktivierungsstatus bereitstellt. Darüber hinaus müssen Rule–Objekte gemäß der Mobilitätsanforderung mindestens eine Methode ("go") besitzen, mit der ihre (Selbst-)Migration veranlasst werden kann.

Rule–Events Gemäß der geforderten Ereignisorientierung des dezentralen Ansatzes sollen alle Interaktionen zwischen regelbehafteten Objekten ausschließlich über Ereignisse als ein lose gekoppelter, asynchroner Kommunikationsmechanismus abgewickelt werden. Hierzu werden spezialisierte Ereignistypen, die so genannten *Rule–Events*, benötigt, die nicht direkt zwischen Quell- und Zielobjekten, sondern über einen *Ereigniskanal (engl. event channel)* kommuniziert werden. Durch eine solche Indirektion ist gewährleistet, dass die kommunizierenden Objekte einander in ihrem jeweiligen Ausführungskontext nicht behindern, da die Events zwischengespeichert werden können. Auch die Adressen der Zielobjekte müssen der Quelle nicht bekannt sein, da ein auf Event–Typen basierter Registrierungsmechanismus (bekannt unter dem Begriff *Publish/Subcribe*) dafür sorgt, dass die Events allen und nur den richtigen Interessenten zugestellt werden. Rule–Events müssen so generisch sein, dass sie beliebige (Rule–)Objekte als Inhalt übertragen können, da Regeln in entfernten (den *externen*) Modi aktiviert werden können (siehe Abschnitt 3.3.2), und sind nach ihrer unterschiedlicher Funktionalität in mehrere *Typen* unterteilt: Auf der ersten Ebene der Rule–Event–Typhierarchie wird zwischen *triggernden* und *protokollrelevanten* Rule–Events unterschieden.

Property–Objekte Da der Bedingungsteil jeder Regel als ein logischer Ausdruck über Anwendungseigenschaften definiert ist, müssen diese den Regelmechanismen explizit zugänglich gemacht werden. Alle Eigenschaften einer Anwendung, über denen Regeln definiert oder die durch Regeln modifiziert werden können, werden als *Property–Objekte* bezeichnet und müssen von der Anwendung über eine *standardisierte* Schnittstelle für Regelinteraktionen bereitgestellt werden.

Property–Set Analog zur Rule–Set werden Property–Objekte mit unterschiedlicher Anwendungssemantik in unterschiedlichen Property–Sets verwaltet.

Rule–sensitiver Aufrufmechanismus Potentiell können zwar alle registrierbaren Ereignisse in einem Anwendungssystem, z.B. auch solche vom Betriebssystem oder der Systemumgebung i.a., als Trigger–Events für Rule–Objekte in Frage kommen. Allerdings sind in einem Anwendungskontext meist nur solche Ereignisse relevant, die eine Anwendungssemantik besitzen. Dazu gehören vor allem Aufrufe von Anwendungsmethoden — in einer verteilten Umgebung sind dies häufig *entfernte* Methodenaufrufe. Es muss deshalb zur Aktivierung von Regeln auf der Anwendungsebene ein generischer Aufrufmechanismus, der Rule–Objekte triggern und mit diesen interagieren kann, zur Verfügung gestellt werden. Eine wichtige Anforderung an diese grundlegende Komponente ist die *transparente* Abwicklung von Regelmechanismen, d.h. aus Entwickler- und Benutzersicht soll ein Rule–sensitiver Methodenaufruf exakt wie ein gewöhnlicher (entfernter) Aufruf durchgeführt werden können. Darüber hinaus muss der Rule–sensitive Aufrufmechanismus in der Regel *transaktional* sein, damit im Falle von Regelverletzungen ein konsistenter Anwendungszustand wieder hergestellt werden kann.

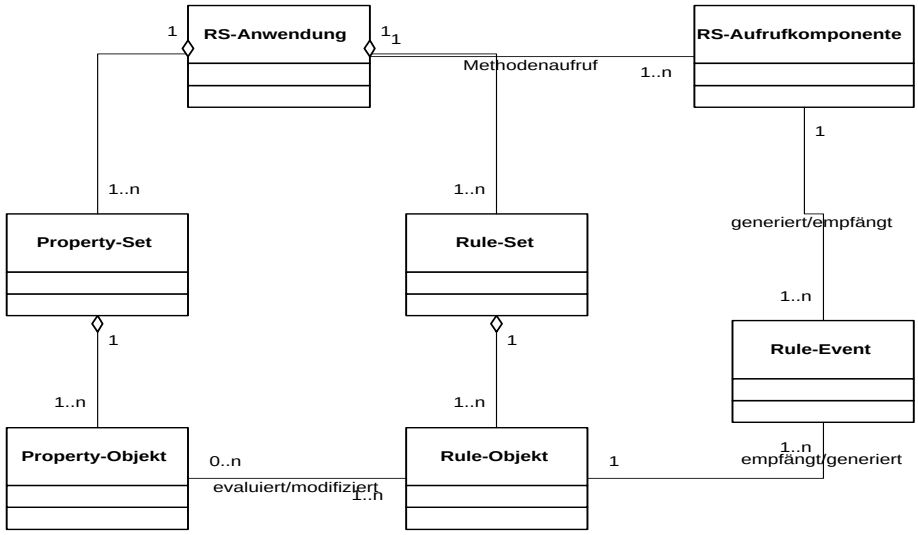


Abbildung 3.13: Beziehungen zwischen einer Rule–sensitiven Anwendung und deren Basiskomponenten

Aus diesem kleinen Satz an Basiskomponenten, von denen jede eine spezifische Rolle bei der Ausführung Rule–sensitiver Aktivitäten einnimmt, kann dann insgesamt eine *Rule–sensitive Anwendungskomponente* (bzw. RS–Anwendung) wie folgt zusammengesetzt werden (siehe Abbildung 3.13): Die RS–Anwendung aggregiert mindestens eine Property– und eine Rule–Set, die jeweils mehrere Property– bzw. Rule–Objekte enthält. Ein oder mehrere Methodenaufrufe können mittels der Rule–sensitiven Aufrufkomponente abgesetzt werden. Über Rule–Events triggert diese Komponente Rule–Objekte, die sich selbst aktivieren und dabei Properties der Anwendung modifizieren können. Mehrere Rule–Objekte können untereinander sowie mit der Aufrufkomponente per Rule–Events

kommunizieren, um ein synchronisiertes Aktivierungsmodell (siehe Abschnitt 3.3.3.2) auszuführen.

3.3.2 Aktivierungsmodi

Ein grundlegendes Konzept zur Umsetzung der Interaktionsorientierung von Regelmechanismen in einer dezentralen Architektur stellen die so genannten Aktivierungsmodi dar, die im folgenden vorgestellt werden. In Abbildung 3.14 ist die Gesamtübersicht aller Modi, bestehend aus zwei lokalen und zwei entfernten, dargestellt.

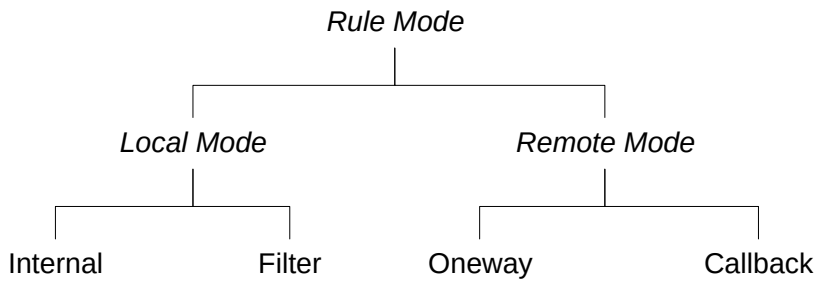


Abbildung 3.14: Übersicht der Rule–Modi

In Bezug auf eine Rule–sensitive Anwendung kann ein ihr zugeordnetes Rule–Objekt entweder in einem *internen* oder *externen* Modus aktiviert werden. Ersterer bedeutet, dass die Regelsemantik auf die sie gerade beherbergende Komponente — die in diesem Fall zugleich den *Anwendungskontext* für Regelinteraktionen liefert — angewendet wird. Dies impliziert z.B., dass Property–Objekte aus der Property–Set der beherbergenden Komponente zur Evaluation und Anpassung des Anwendungsverhaltens benutzt werden. Letzterer Modus bedeutet dagegen, dass die Regelsemantik auf eine externe, typischerweise entfernte Komponente angewendet werden soll.

Die externe Aktivierung einer Regel scheint besonders geeignet zu sein, um in einem Kooperationskontext die Erfüllung *fremder* Anforderungen zu automatisieren, also die geforderte Interaktionsorientierung zu unterstützen. Allerdings ist sie grundsätzlich mit dem Risiko behaftet, die Autonomie — ebenfalls eine wichtige Anforderung — zu verletzen. Um diesen prinzipiellen Konflikt aufzulösen, muss bei der externen Aktivierung ein *lokales Gegenpol* auf Seite der entfernten Komponente geschaffen werden, um ihre Autonomie zu bewahren. Hierfür dient der so genannte *Filter*–Modus, der ein Rule–Objekt zum Gegenpart des extern aktivierten werden lässt. Ein Rule–Objekt im externen Modus wird dadurch aktiviert, dass eine Kopie von ihm erzeugt und auf die entfernte Komponente (deren Referenz aus dem triggernden Event–Objekt entnommen wird) übertragen wird, dort wird sein Bedingungsteil mit dem der Filter–Rule unifiziert. Falls dies zu einem positiven Ergebnis führt, wird die Semantik der übertragenen Rule (mit dem Ergebnis der Unifikation als Bedingungsteil) bei der entfernten Komponente ausgeführt. Schließlich wird die Kopie des Rule–Objektes wieder gelöscht. [9]

[9]Dieses Verfahren impliziert, dass ein Anwendungsobjekt gegen externe Regeleinflüsse da-

Um unterschiedliche Kooperationssemantiken zu unterstützen, wird der externe Modus in zwei Submodi, *Oneway* und *Callback*, unterteilt. Beim Oneway–Modus wird die Regelsemantik, wie gerade beschrieben, ausschließlich auf die externe Komponente übertragen. Beim Callback–Modus wird nach der entfernten Aktivierung zusätzlich die Belegung der im unifizierten Bedingungsteil vorkommenden Properties auf die lokale (Ausgangs-)Komponente zurück übertragen und dort gesetzt bzw. aktiviert. Auf diese Weise kann gewährleistet werden, dass bzgl. der entsprechenden Properties eine exakt gleiche Konfiguration beider Seiten — und somit eine *gemeinsame* Interaktionsbasis — gegeben ist.

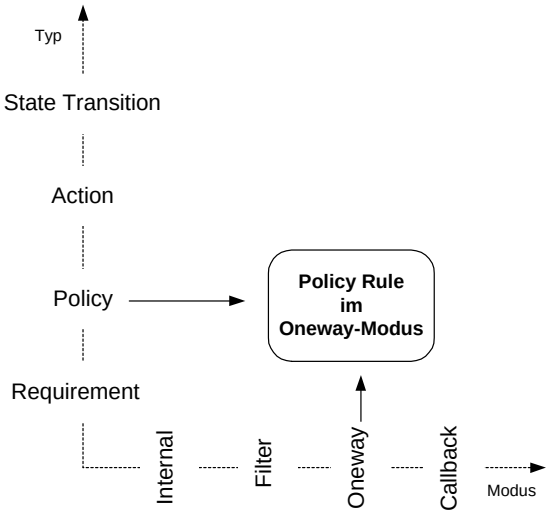


Abbildung 3.15: Zweidimensionalität der Regelsemantik

Durch die prinzipiell beliebige Verknüpfung der unterschiedlichen Regeltypen, die in Abschnitt 2.3.1.3 eingeführt wurden, mit den vorgestellten Aktivierungsmodi ergibt sich, wie durch Abbildung 3.15 veranschaulicht, eine Zweidimensionalität der Regelsemantik, die ein entsprechend vielfältiges und flexibles Spektrum an Möglichkeiten der interaktionsorientierten Steuerung von verteilten Anwendungen bietet.

3.3.3 Das Konzept einer Rule–sensitiven Middleware

Nachdem in den vorangegangenen Abschnitten die Basiskonzepte zur Realisierung einer dezentralen Regelverarbeitung eingeführt wurden, soll nun das Konzept einer *Rule–sensitiven Middleware* im Sinne einer Gesamtarchitektur zur Unterstützung Rule–sensitiver verteilter Anwendungen dargestellt werden.

durch komplett geschützt werden kann, wenn es entweder gar keine Filter–Rules enthält bzw. nur solche, die als Bedingungsteil die Konstante "FALSE" enthalten und somit immer eine negative Unifikation verursacht. Dagegen ist ein Anwendungsobjekt maximal kooperativ, wenn es zu jedem externen Rule–Objekt eine Filter–Rule mit "TRUE" als Bedingungsteil bereitstellt.

3.3.3.1 Architektur der Rule–sensitiven Middleware

Zwar sind die in Abschnitt 3.3.1.2 vorgestellten Basiskomponenten zur Konstruktion Rule–sensitiver Anwendungen *konzeptionell* unabhängig von einer bestimmten Verteilungsplattform bzw. Systemarchitektur und könnten in Form eines proprietären, d.h. technisch eigenständigen, Baukastens realisiert werden. Allerdings wäre eine solche Realisierung erstens sehr aufwendig, da alle geforderten Eigenschaften von der verteilten Kommunikation bis hin zur Transaktionalität neu implementiert werden müssten, und zweitens nicht kompatibel zu den weit verbreiteten Standards auf dem Gebiet verteilter Systeme. Um diese Nachteile zu vermeiden, bietet sich insbesondere die Verwendung der von einer Middlewareplattform — wie etwa CORBA (vgl. Abschnitt 1.3.3) — angebotenen Dienste an.

Allerdings erscheint es besonders sinnvoll, nicht erst auf der Implementations-, sondern bereits auf der Architekturebene eine Synthese der beiden Konzepte "Rule–sensitive Anwendung" und "Middleware" zum Konzept der Rule–sensitiven Middleware (oder RS–Middleware) vorzunehmen, um eine orthogonale Integration regelbasierter Steuerungsmechanismen in eine Middlewareplattform und damit eine hohe Generizität zu erreichen.

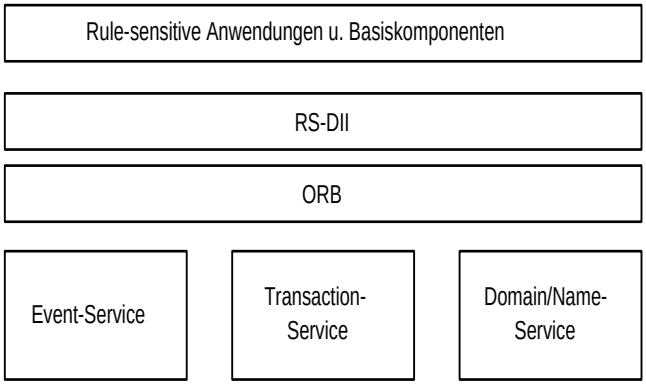


Abbildung 3.16: Architektur der Rule–sensitiven Middleware

Abbildung 3.16 zeigt die Grobarchitektur der RS–Middleware, die (von oben nach unten) aus folgenden Hauptkomponenten besteht:

- Auf der Anwendungsschicht sind die Rule–sensitiven Anwendungen angesiedelt, die die Schnittstelle *RuleSensitive* implementieren müssen, damit Regelmechanismen zur Laufzeit aktiviert werden können. Ebenfalls sind die meisten Basiskomponenten zur Realisierung dieser Anwendungen wie z.B. Rule–Objekte und Rule–Events dieser Ebene zuzuordnen.

- Kernstück der Rule–sensitiven Middleware bildet das so genannte *Rule–Sensitive Dynamic Invocation Interface* oder *RS–DII*, das eine konkrete Variante des Rule–sensitiven Aufrufmechanismus darstellt. In Abschnitt 3.2.4.2 wurde diese Komponente bereits als Mechanismus zur automatischen Aktivierung von Regeln vorgestellt. Im Gegensatz zu den dort *zentral* verwalteten Regeln erfordert eine dezentrale Architektur jedoch ein völlig neues Aktivierungsmodell (siehe nächsten Abschnitt 3.3.3.2) und

eine stärkere Integration des Rule–sensitiven Aufrufmechanismus mit den Middleware–Diensten.

Das RS–DII basiert auf dem in CORBA spezifizierten DII, das einen dynamischen und sehr flexiblen Mechanismus zur Abwicklung von Methodenaufrufen zwischen entfernten Anwendungskomponenten anbietet. Das RS–DII erweitert das DII in zwei Hinsichten. Erstens triggert es Rule–Objekte sowohl vor als auch nach dem eigentlichen Aufruf durch Generierung und Synchronisation entsprechender Events. Zweitens führt es den Aufruf auf eine transaktionale Weise durch, d.h. im Falle von Regelverletzungen oder fehlerhafter Regelaktivierung wird der Aufruf unterbrochen und seine bereits erzeugte Wirkung rückgängig gemacht.

- Das RS–DII, wie auch das normale DII, nutzt bei der Abwicklung entfernter Methodenaufrufe die Dienste des *Object Request Broker* oder *ORB*, der u.a. die Lokalisierung und die Aktivierung von entfernten Objekten unterstützt.

- Um viele wichtige Anforderungen an Rule–sensitive Anwendungen — u.a. Interaktionsorientierung, Ereignisorientierung und Transaktionalität — auf eine effiziente und interoperable Weise umzusetzen, können eine Reihe von "Standarddiensten", die in der CORBA–Architektur den *Common Object Services* (COSS) zuzuordnen sind, in Anspruch genommen werden. Dazu gehören zumindest:

  - ein *Domain/Name Service*: Trotz der angestrebten maximalen Autonomie von Rules als aktiven Objekten, die sich selbst aktivieren können, benötigt das RS–DII als Kopplung zwischen der Anwendungs- und der Rule–Semantik gewisse Informationen wie etwa die Gesamtzahl der aktuell in einer Anwendung beherbergten Rule–Objekte oder den Zu- und Abgang von Rule–Objekten *während* des Methodenaufrufes (z.B. durch Migration). Hierfür wird ein Dienst eingesetzt, der Domäneninstanzen verwaltet, an den Rule–sensitive Anwendungen sich registrieren müssen. Die Funktionalität dieses Dienstes wird praktischerweise mit der eines üblichen Namensdienstes zur Unterstützung der Anwendungen beim Auflösen von symbolischen Namen (Alias) in Objektreferenzen integriert.
  - ein *Event Service*: Dieser Dienst wird benötigt, um Rule–Events zu verschicken und zu empfangen (vgl. Abschnitt 3.2.4.2). Hierfür bietet sich der CORBA Event Service an, der ein hohes Maß an Interoperabilität der Event–Objekte garantiert.
  - und ein *Transaction Service*: Dieser Dienst wird benötigt, um die transaktionale Ausführung von Rule–sensitiven Aktivitäten zu implementieren. Auch hierfür bietet sich ein entsprechender (gleichnamiger) CORBA–Dienst an.

3.3.3.2 Aktivierungsmodell

Im folgenden wird ein Aktivierungsmodell, welches die dynamischen Aspekte der Rule–sensitiven Middleware festlegt, präsentiert. Dieses Modell durchleuchtet auch die wechselseitigen semantischen Beziehungen zwischen den Basiskomponenten einer Rule–sensitiven Anwendung. Zunächst wird die Gesamtsicht auf

das Zusammenspiel der Komponenten anhand eines Ablaufszenarios dargestellt. Danach wird die Ablaufsemantik einzelner Komponenten genauer beschrieben.

Gesamtablauf

Um die Rule–Sensitivität verteilter Anwendungen zu verwirklichen, müssen RS–DII, Rule–Objekte, Rule–Events und Rule–Modi auf eine genau spezifizierte Weise zusammenspielen. Um dieses Zusammenspiel zu veranschaulichen, wird in Abbildung 3.17 der Gesamtablauf bei der externen Aktivierung eines Rule–Objektes anhand eines (generischen) Anwendungsszenarios graphisch dargestellt.

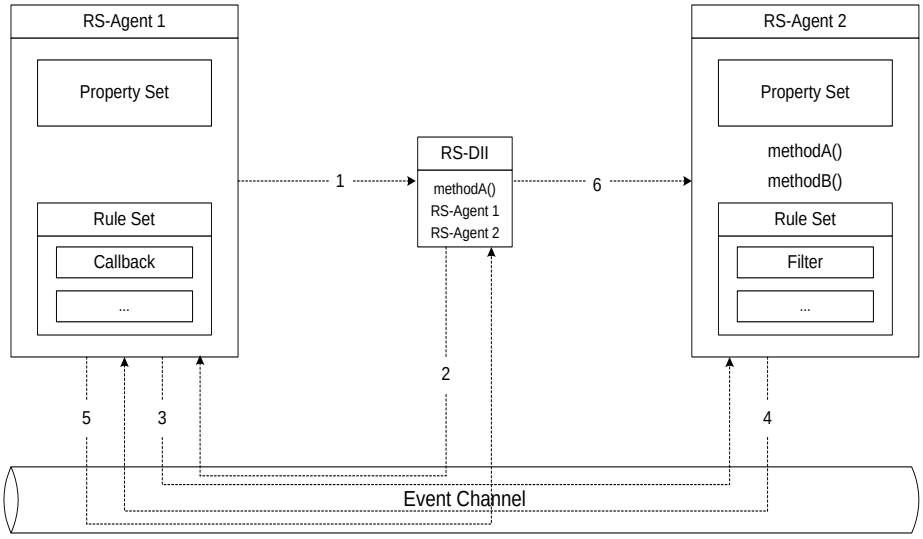


Abbildung 3.17: Aktivierung eines Rule–Objekts im Callback–Modus

In diesem Szenario interagieren zwei Rule–sensitive Anwendungen, genannt *RS–Agent–1* und *RS–Agent–2*, durch einen Methodenaufruf des ersten Agenten an den zweiten. RS–Agent–1 besitzt dabei eine Regel zur Anpassung gewisser Kooperationsparameter auf beiden Seiten, wie z.B. der Adresse eines gemeinsam genutzten Notariatsdienstes, bevor die Methode aufgerufen wird. Deshalb ist diese Regel bereits auf den *Callback*–Modus eingestellt.

Der Gesamtablauf findet wie folgt statt: RS–Agent–1 setzt den Aufruf mittels einer Instanz des RS–DII ab, indem es dieser die Signatur der Methode einschließlich aller Parameter und das Zielobjekt übergibt (1). Das RS–DII extrahiert zunächst die relevanten Daten, um die Kontext–ID zu erzeugen, und setzt ein Rule–Event (vom Typ *BeforeMethodRuleEvent*) mittels des Event Channel ab, das dieses an alle Rule–Objekte, die sich für den entsprechenden Event–Typ registriert haben, liefert (2). In diesem Fall wird das Rule–Objekt im Callback–Modus bei RS–Agent–1 getriggert, und gemäß der Semantik dieses Modus erzeugt es eine Kopie von sich selbst, verpackt sie in ein eine Instanz des *FilterRuleEvent* setzt das Event ab (3). Dadurch wird das Rule–Objekt im Filter–Modus bei RS–Agent–2 getriggert und die Unifikation der beiden Rules angestoßen. Falls dies zu einem positiven Ergebnis führt, wird die Aktion der

externen Regel auf Seite des RS–Agent–2 ausgeführt, wodurch gewisse Properties (die Kooperationsparameter) auf dieser Seite modifiziert werden können. Die Belegung dieser Properties wird dann mit einer Instanz des *ReplyRuleEvent* zum Rule–Objekt von RS–Agent–1 zurückgeschickt (4). Mit der übergebenen Belegung der Properties wird als Bedingungsteil wird die Aktivierung dieses Rule–Objektes so abgeschlossen, als ob es nun im Internal–Modus wäre, und ein ReplyRuleEvent wird danach an das RS–DII verschickt als Benachrichtigung über die erfolgreiche Aktivierung (5), womit das RS–DII den Aufruf beim Zielobjekt durchführen kann (6). Eine analoge Sequenz von Rule–Events findet statt, nachdem der Methodenaufruf durchgeführt wurde, und im Falle von Regelverletzungen, die durch *RuleExceptions* signalisiert werden, wird mit Hilfe des Transaction Service die Wirkung der Methode rückgängig gemacht.

Events

Um Rule–bezogene Interaktionen ausschließlich über Event–Objekte abzuwickeln, sind mehrere Event–Typen erforderlich. Abbildung 3.18 zeigt alle Event–Typen der Rule–sensitiven Middleware als Übersicht.

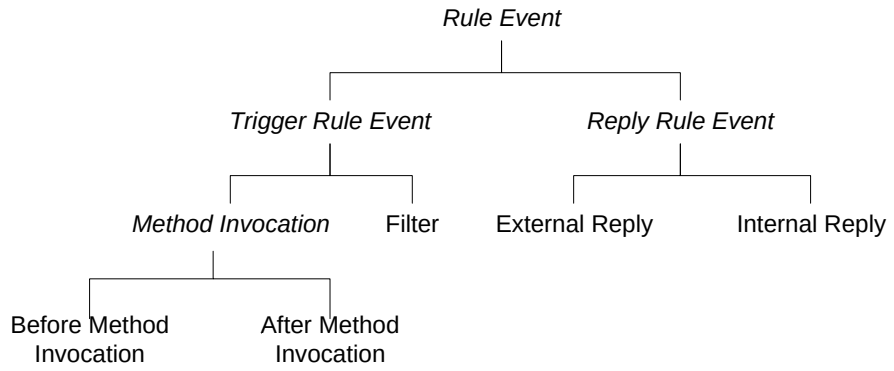


Abbildung 3.18: Übersicht der Rule–Events

Auf der abstrakten Ebene wird zunächst zwischen triggernden Events (*TriggerRuleEvent*), die einer beliebigen Quelle entstammen können, und Antwortevents (*ReplyRuleEvent*), die ein bestimmtes Protokollverhalten umsetzen, unterschieden. In der vorliegenden Architektur werden Triggerevents in solche, die mit einem Methodenaufruf verknüpft sind (*MethodInvocationRuleEvent*), und solche, die für ein Rule–Objekt im Filter–Modus bestimmt sind (*FilterRuleEvent*). Wie aus dem geschilderten Szenario ersichtlich, werden erstere automatisch durch das RS–DII erzeugt, während letztere durch Rule–Objekte in einem externen Modus generiert werden. Zur klaren Trennung zwischen Regeleinflüssen, die entweder nur vor oder nur nach einer Aktivität erforderlich sind, wird *MethodInvocationRuleEvent* noch in *BeforeMethodInvocationRuleEvent* und *AfterMethodInvocationRuleEvent* unterteilt. Die protokollrelevanten Antwortevents werden in solche, die von einem externen Filter–Objekt erzeugt werden (*ExternalReplyRuleEvent*), und solche, die als direkte Antwort für eine triggernde Anwendungs- oder Systemkomponente — wie typischerweise das RS–DII — bestimmt sind (*InternalReplyRuleEvent*), unterteilt. Durch diese ge-

naue Unterteilung ist es möglich, die meisten nicht relevanten Events bereits auf Typebene herauszufiltern.

RS–DII

In Abbildung 3.19 ist die genaue Funktionsweise des RS–DII in Form eines Activity–Diagramms spezifiziert.

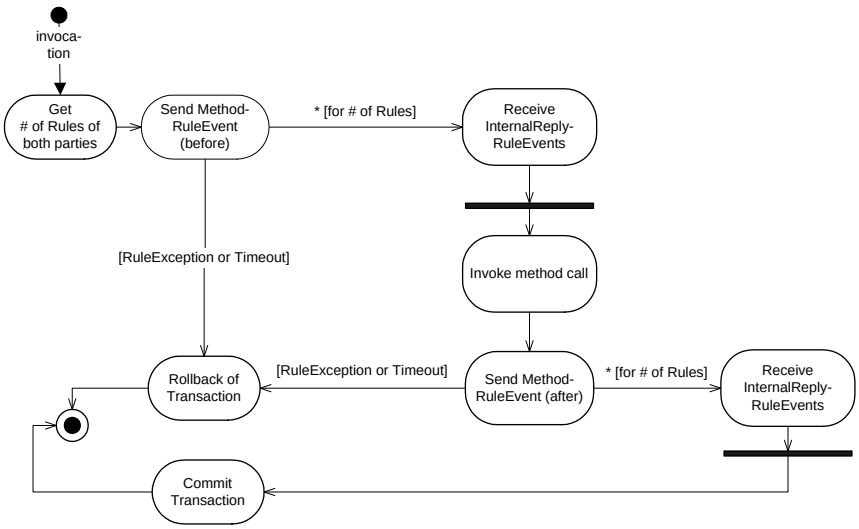


Abbildung 3.19: RS–DII Activity–Diagramm

Beim Auslösen eines Rule–sensitiven Methodenaufrufs ermittelt das RS–DII mit Hilfe des Domain/Name Service zunächst für jede an dem Aufruf beteiligte Anwendung die Anzahl n der aktuell vorhandenen Rule–Objekte. Die RS–Anwendungen werden ebenfalls dazu aufgefordert, ihre Rules, falls noch nicht initialisiert, an den Event–Channel anzuschließen. Dann wird eine Instanz des *BeforeMethodInvocationRuleEvent* an alle Rule–Objekte verschickt. Anschließend wartet das RS–DII auf n entsprechende Antwort–Events vom Typ *InternalReplyRuleEvent*. Wenn die erwarteten Antworten eingetroffen sind, wird die Methode beim Zielobjekt aufgerufen und das Ergebnis des Aufrufs zwischengespeichert. Danach wird ein Event vom Typ *AfterMethodInvocationRuleEvent* verschickt und auf die gleiche Anzahl von Antwort–Events gewartet. Wenn während dieses Ablaufs, der insgesamt als eine Transaktion abgewickelt wird, ein negatives Antwort–Event empfangen wird oder ein anderer Fehler auftritt, unterbricht das RS–DII sofort den Aufruf, veranlasst mittels des Transaction Service ein Rollback der bis dahin entstandenen Wirkung und generiert eine *RuleException*, die an den Aufrufer zurückgeliefert wird. Andernfalls wird Transaktion mit einem *Commit* beendet und das Ergebnis des Methodenaufrufs weitergeleitet. Mehrere gleichzeitige Rule–sensitive Aufrufe werden dadurch unterschieden, dass jedem von ihnen eine eindeutige Kontext–ID zugeordnet wird, die jedem entsprechenden Event mitgegeben wird.

Da alle Rule–Interaktionen hinter dem RS–DII verborgen bleiben, ist das Absetzen eines Rule–sensitiven Aufrufs programmtechnisch exakt gleich wie das

eines dynamischen (transaktionalen) Aufrufs über das normale DII. Der Aufrufer muss lediglich darauf vorbereitet sein, eine *RuleException* als einen weiteren Ausnahmefall zu empfangen (und eventuell zu behandeln).

Modi

Die von ihrem jeweiligen Aktivierungsmodus abhängige Funktionsweise der Rule–Objekte ist in den Abbildungen 3.20 bis 3.22 als Activity–Diagramme spezifiziert.

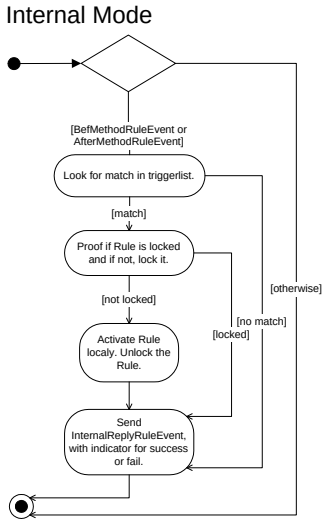


Abbildung 3.20: Activity–Diagramm des Internal–Modus

Im Internal–Modus (Abbildung 3.20) sind nur Trigger–Events vom Typ *BeforeMethodInvocationRuleEvent* oder *AfterMethodInvocationRuleEvent* relevant. Wird ein solches Event empfangen, dann wird es zunächst mit den Einträgen der Trigger–Liste verglichen. Falls kein Eintrag gefunden wird, der sowohl vom Typ als auch bzgl. der Methodensignatur passt, wird die (Selbst-)Aktivierung des Rule–Objektes mit einem (positiven) protokollrelevanten Antwort–Event abgeschlossen. Andernfalls wird der Status des Rule–Objektes auf *true* ("aktiviert") gesetzt, um eine Endlos–Triggerung und damit eine Deadlocksituation zu vermeiden[10]. Je nachdem, ob die anschließende Ausführung der typspezifischen Regelsemantik erfolgreich ist oder nicht, wird ein entsprechendes Antwort–Event gesendet und der Aktivierungsstatus wieder auf *false* ("bereit") gesetzt.

Ein Rule–Objekt im einem der beiden externen Modi Oneway oder Callback, deren Semantik gemeinsam in Abbildung 3.21 dargestellt ist, benötigt immer ein Gegenstück in Form einer Rule im Filter–Modus. Wenn das erste getriggert wird, findet zwischen den beiden Rule–Objekten eine 1–1 Kommunikation statt. Bei Eintreffen des Trigger–Events wird es mit den Einträgen in der Trigger–Liste verglichen. Wenn ein passender Eintrag gefunden wird, generiert

[10]Eine Deadlocksituation kann beispielsweise auftreten, wenn innerhalb eines verschachtelten Aufrufes mittels des RS–DII ein Rule–Objekt sich selbst triggert. Dies kann insbesondere bei Verwendung von Action–Rules auftreten, da die in der Rule spezifizierte Aktion grundsätzlich durch einen Methodenaufruf ausgeführt wird.

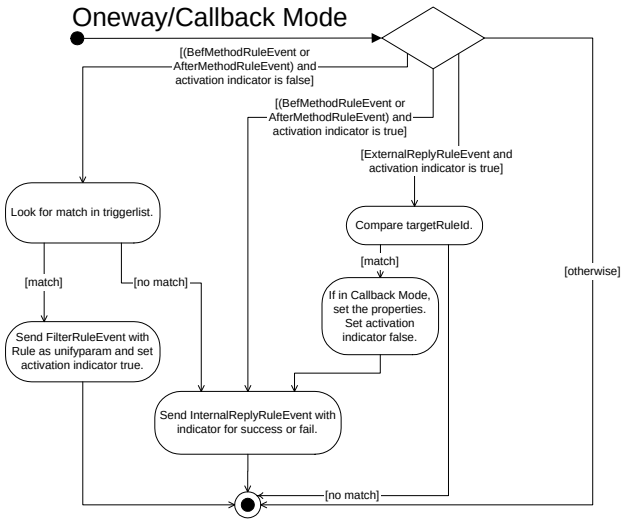


Abbildung 3.21: Activity–Diagramm für Oneway– und Callback–Modus

das Rule–Objekt eine semantische Kopie von sich selbst und sendet diese als Bestandteil eines Filter–Events an das Rule–Objekt im Filter–Modus. Anschließend wird der Aktivierungsstatus auf *true* gesetzt, so dass das Rule–Objekt als gesperrt gilt. In diesem Zustand wartet es, bis das externe Antwort–Event (vom Typ *ExternalReplyRuleEvent*) eintrifft. Anhand der in dem Antwort–Event enthaltenen Kontext–ID kann sichergestellt werden, dass die Antwort auch für den Empfänger bestimmt ist. In dem externen Antwort–Event sind ebenfalls die Belegung der unifizierten Properties enthalten. Diese Werte werden nur dann für die entsprechenden Properties lokal gesetzt, falls es sich um den Callback–Modus handelt. Schließlich wird ein Antwort–Event, welches protokollrelevant für das RS–DII ist, abgeschickt und der Statussperre wieder aufgehoben.

Eine Rule im Filtermodus, deren Semantik in Abbildung 3.22 dargestellt ist, reagiert ausschließlich auf Events vom Typ *FilterRuleEvent*. Um festzustellen, ob das Filter–Event wirklich für diese Rule bestimmt ist, werden zum einen das Event mit den Einträgen der Trigger–Liste abgeglichen und zum anderen auch die mit dem Event übergebenen Anwendungsklasse und zugehörige Rule–Set mit den eigenen verglichen. Dadurch wird gewährleistet, dass nur Rules für die gleiche (Sub-)Domäne miteinander unifiziert werden. Nach der erfolgreichen Unifikation wird die Semantik der externen Rule ausgeführt. Ähnlich wie beim Internal–Modus wird auch hier mit dem Aktivierungsstatus eine Sperre gesetzt, um Deadlocks im Falls verschachtelter Aufrufe über das RS–DII zu vermeiden.

3.4 Zusammenfassung

Um die anwendungsunabhängige Verarbeitungsfunktionalität der regelbasierten Steuerungskonzepte auszuschöpfen, sind unterschiedliche Systemarchitekturen erforderlich, da in der Praxis sehr unterschiedliche Anforderungen bzgl. der Bedienbarkeit, der Anwendungstransparenz und der Performanz existieren. Deshalb wurden in diesem Kapitel sowohl eine zentrale als auch eine dezentrale

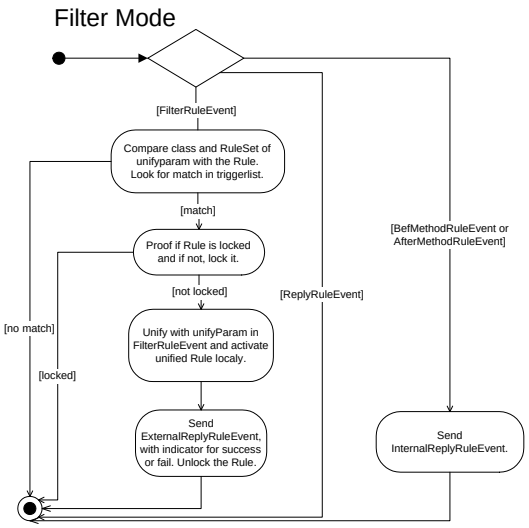


Abbildung 3.22: Activity–Diagramm des Filter–Modus

Architektur zur Integration von Regelmechanismen in verteilte Anwendungsumgebung behandelt.

Bei der *zentralen* Architektur werden die Regelmechanismen von einem Systemdienst grundsätzlich über operationale Schnittstellen, die direkt von den Anwendungen aufgerufen werden können, angeboten. Der Hauptvorteil dieser Architektur liegt in der geringen Komplexität der Realisierung und der geringeren Abhängigkeit der Anwendungen von den Regelmechanismen, insbesondere wenn diese nur sporadisch, z.B. bei der Initialisierung einer Anwendung, genutzt werden sollen. Um den Zugriff auf den zentralen *Policy–Manager* für die Anwendungen zu erleichtern, wurde neben den operationalen auch *ereignisorientierte* Schnittstellen konzipiert, worüber mittels so genannter *Event–Handler*–Komponenten eine automatische Aktivierung der Regeln (als zusätzliche *Aktivierungsschicht* in der Architektur) realisiert werden kann.

Dagegen bietet die *dezentrale* Architektur wesentlich mehr transparente Systemunterstützung für die Anwendungsebene, da Regelobjekte dynamisch und direkt einzelnen Anwendungen zugeordnet/zugeschickt bzw. bei diesen (transparent) verwaltet werden. Hierzu wurden die Konzepte der *Rule–sensitiven (verteilten) Anwendungen* und *Rule–sensitiven Middleware* vorgeschlagen und erläutert. Ein wesentlicher Bestandteil der dezentralen Architektur ist das Konzept unterschiedlicher *Rule–Modi*, womit ein dezentral verwaltetes Rule–Objekt entweder *lokal* oder *entfernt* aktiviert werden kann. Um die Semantik von Rule–Interaktionen zu präzisieren, wurde ein dezidiertes *Aktivierungsmodell* vorgestellt. Da Rule–Interaktionen in diesem Modell ausschließlich über asynchron verarbeitete *Events* abgewickelt werden, ist sichergestellt, dass die Regelverarbeitung zwar mit dem eigentlichen Anwendungsablauf synchronisiert wird, diesen aber grundsätzlich nicht beeinträchtigt bzw. behindert. Im allgemeinen gilt, dass die Komponenten der Anwendungsebene erst von ihrer Rule–Sensitivität erfahren, wenn eine Regelverletzung in Form einer ablauftechnischen *Exception* ihnen gemeldet wird.

Kapitel 4

Realisierung regelbasierter Steuerungsmechanismen

Dieses Kapitel behandelt Aspekte der Umsetzung der in Kapitel 3 beschriebenen Architekturen in konkrete Softwarekomponenten zur zentralen und dezentralen Steuerung von Anwendungen in einer offenen verteilten Umgebung. Es soll einen breiten Überblick über alle wesentlichen Implementationsaspekte und -entscheidungen vermitteln. Allerdings erfolgt die Darstellung größtenteils auf einer noch allgemeinen sprachlichen Stufe und reicht maximal bis zur Schnittstellen- (und nicht Quellcode-) Ebene hinunter.

Das Kapitel beginnt mit einer Beschreibung der Vorgehensweise, die der Implementierung zu Grunde liegt (Abschnitt 4.1). Anschließend wird die Implementierung des Policy–Managers als eines zentralen Dienstes zur Steuerung verteilter Anwendungen im Detail erläutert (Abschnitt 4.2). Es folgen dann Details über die Implementierung der dezentralen Rule–Management–Architektur (Abschnitt 4.3) und schließlich eine Gesamtbewertung der Ergebnisse (Abschnitt 4.5).

4.1 Methodische Vorgehensweise

4.1.1 Anforderungen an die Implementierung

Die Hauptanforderung an die Implementierung besteht darin, den Nachweis zu erbringen, dass die in Kapitel 2 und 3 vorgestellten Modellierungs- und Architekturkonzepte *technisch realisierbar* sind. In der Tat ist technische Realisierbarkeit häufig schon für sich eines der entscheidenden Kriterien praxisnaher Forschungsarbeiten, da viele Modelle, Konzepte und Architekturen in theoretischen Abhandlungen entweder sich nicht direkt in reale Systeme abbilden lassen — sei es durch unentdeckte Inkonsistenzen bzw. Fehler in der Konzeption / Theorie oder sehr häufig auch durch unzureichende Spezifikation der Komponenten — oder erst sich durch die konkrete Realisierung als "untauglich" bzw. praxisfern erweisen.

Um diese Anforderung zu erfüllen, erscheint es am sinnvollsten, möglichst früh während oder auch parallel zur Ausarbeitung der Konzepte kleinere *prototypische* Systeme, die jeweils nur eine gewisse Teilfunktionalität des Gesamtsy-

stems realisieren, zu erstellen und zu erproben. Dadurch ist sichergestellt, dass zumindest grundlegende Unzulänglichkeiten und Schwächen der betreffenden Konzepte / Architekturen *frühzeitig*, also bevor die Komplexität des Gesamtsystems unbeherrschbar geworden sowie unnötiger Aufwand investiert worden ist, entdeckt und entsprechende Konsequenzen gezogen werden können. Diese "pragmatische" Vorgehensweise hat sich in den letzten Jahren immer mehr durchgesetzt und erfährt auch zunehmend eine fundierte systematische Behandlung, die beispielsweise unter dem Begriff "eXtreme Programming" publiziert wird [Bec99].

Eine ebenso wichtige wie herausfordernde Anforderung ist die oft genannte *Wiederverwendbarkeit*. Diese Anforderung wurde bereits im Zusammenhang mit der Generizität (der Konzepte) erwähnt (siehe Abschnitte 1.4.1 und 3.3.1.1). Im technischen Sinne bedeutet Wiederverwendbarkeit jedoch nicht, dass möglichst *anwendungsunabhängige* Konzepte gefunden werden, sondern dass die Bestandteile einer Implementation so *granuliert* und *strukturiert* sind, dass möglichst viele (Architektur-)Konzepte sich damit realisieren lassen. Um diese Anforderung zu erfüllen, ist es erforderlich, einen entsprechenden *modularen* und *hierarchischen* Aufbau der Implementationskomponenten, insbesondere der Klassenbibliotheken, zu finden.

Die implementierten Mechanismen sollen außerdem in dem Sinne *praxisnah* sein, dass sie mit den innovativen Technologien, die in der Praxis verbreitet bzw. relevant sind, "konform" gehen. Dies bedeutet einerseits, dass passende Merkmale praxisrelevanter Technologien für die Implementierung zunutze gemacht werden sollen, und andererseits, dass typische Anwendungen in der Praxis, die auf den entsprechenden Technologien basieren, von den implementierten Steuerungsmechanismen direkt Gebrauch machen können. Gleichwohl soll die Implementierung jedoch möglichst unabhängig von bestimmten Produkten sein, um eine wegen der Schnelllebigkeit heutiger Produkte häufig notwendige Umstellung auf andere Produkte oder Produktversionen zu erleichtern. Diese Anforderung kann dadurch erfüllt werden, dass produktspezifische Funktionen mit Hilfe von Kapselungsklassen bzw. Adapterkomponenten vor den übrigen Implementationsbestandteilen verborgen werden.

Einschränkend ist jedoch anzumerken, dass die angestrebte technische Realisierbarkeit und Praxisnähe nicht mit "Produktreife" gleich zu setzen ist, da wichtige Aspekte einer (professionellen) Produktentwicklung wie etwa Robustheit, Skalierbarkeit oder auch Performanz zumindest nicht im Vordergrund der in dieser Arbeit verfolgten Ziele stehen. Vielmehr zielt die Implementierung der hier beschriebenen prototypischen Systeme darauf, die Realisierbarkeit der Konzepte durch entsprechende *Anwendungsszenarien* zu demonstrieren.

Die Anforderungen an die Implementierung sind also zusammenfassend:

Vollständiger Konzeptnachweis Um die technische Realisierbarkeit nachzuweisen, sollen prototypische Systeme implementiert werden, die zwar nicht produktreif sein müssen, jedoch möglichst alle architekturellen Anforderungen erfüllen.

Modularer, hierarchischer Aufbau Die Bestandteile der Implementation sollen so modularisiert und strukturiert sein, dass unterschiedliche Architekturkonzepte mit den gleichen Bausteinen realisiert werden können.

Praxisnähe Die implementierten Steuerungsmechanismen sollen kein isoliertes

proprietäres System bilden, sondern technisch kompatibel sein zu gängigen Technologien und Architekturen, die eine eindeutige Praxisrelevanz besitzen.

Produktunabhängigkeit Trotz der angestrebten Praxisnähe sollen die Prototypsysteme implementationstechnisch möglichst unabhängig sein von den konkreten Produkten, auf denen sie basieren.

4.1.2 Verwendete Technologien

Bezüglich der zu verwendenden Technologien ist zunächst die Entscheidung über eine geeignete Programmierumgebung zu treffen. Hierfür bietet sich die Programmiersprache Java, die in Abschnitt 1.3.2 bereits eingeführt wurde, und die mit ihr gelieferten Softwarebausteine als die z.Z. praxisrelevanteste Umgebung zur Implementierung offener verteilter Anwendungen an. Mit ihren Basiseigenschaften wie Portabilität (od. Plattformunabhängigkeit), integrierten Internetfähigkeiten (*applets, servlets, URL–Integration* etc.), verteilter automatischer Speicherverwaltung (*distributed garbage collection*), standardisierter Ausnahmebehandlung (*exception handling*) und vor allem der fast universellen Verfügbarkeit des Bytecode–Interpreters (*Java Virtual Machine*) bietet diese Sprache bezüglich der Integration in praxisrelevante Systeme gegenwärtig eine Fülle von Merkmalen dar, zu der es *de-facto* keine Alternative gibt.

In Bezug auf die Implementierung von *Regelverarbeitungsfunktionen* im allgemeinen und die Verarbeitung logischer Ausdrücke als Bedingungsteil von Regeln (siehe Kapitel 2) im besonderen erscheinen jedoch, zumindest auf den ersten Blick, *logikbasierte* Programmiersprachen als die natürlichere und bessere Alternative, da die gesamte Syntax und Semantik einer solchen Sprache direkt auf der Prädikatenlogik basiert. Hierzu zählt vor allem die Sprache Prolog [CM94]. Allerdings stellt sich bei etwas genauerer Betrachtung heraus, dass die "Logikverwurzelung" von Prolog doch mit einigen Einschränkungen behaftet ist. Die grundlegendste Einschränkung besteht darin, dass Prolog grundsätzlich nur *Horn–Formeln* (siehe Def. 2.2.11) verarbeiten kann, und zumindest ist es dem Autor der vorliegenden Arbeit nicht gelungen, die in Abschnitt 2.3.1.4 bzw. 2.3.3.1 dargestellte Mächtigkeit von Bedingungsausdrücken in Horn–Formeln zu transformieren. Diese, hier als These aufgestellte, Nichttransformierbarkeit wird durch Satz 2.2.2 unterstützt[1].

Desweiteren ist eine Entscheidung über die Auswahl der grundlegenden *Dienste* zur Unterstützung der Laufzeitumgebung zu treffen. Insbesondere müssen die Basisdienste der Rule–sensitiven Middleware, nämlich Event, Transaction und Domain/Name Service, nicht komplett neu implementiert werden, sondern die von Standardprodukten auf diesem Gebiet bereitgestellte Funktionalität zunutze machen. In Kapitel 3 ist bereits erläutert worden, dass hierfür eine CORBA–Plattform (die in Abschnitt 1.3.3 vorgestellt wurde) als die sinnvollste Wahl erscheint. Daher ist bezüglich der Implementierung lediglich ein konkretes Produkt auszusuchen.

[1] Hiermit ist jedoch nicht gemeint, dass die noch ziemlich eingeschränkte Mächtigkeit der in Kapitel 2 vorgestellten Bedingungsausdrücke bereits stärker ist als die Mächtigkeit von Prolog–Ausdrücken, da aufgrund der geschilderten Einschränkungen bzgl. ersterer eine direkte Vergleichbarkeit nicht gegeben ist.

Allerdings stellt sich die Wahl von innovativen Softwareprodukten häufig als ein *dynamischer Prozess* heraus, da solche Produkte einem ständigen Prozess der Erneuerung und damit des Wandels unterliegen. Deshalb wurden im Rahmen der Implementierung der bisher beschriebenen Konzepte eine Reihe von CORBA–Produkten erprobt und eingesetzt, von denen im Laufe der Zeit jeweils viele Versionen entstanden, die große funktionale Unterschiede — z.B. zunächst nur mit C++, später auch mit Java als Sprachanbindung — aufweisen. Hervorzuheben sind darunter die beiden Produkte *Orbix* von IONA [ION] und *VisiBroker* [INP] von INPRISE, die in den letzten Jahren (in dieser zeitlichen Reihenfolge) den CORBA–Markt durch ihre Fülle an technischen Merkmalen und Innovationen dominiert haben. Mit diesen beiden Produkten wurden beispielsweise mehrere unterschiedliche Versionen des Policy–Managers implementiert. Im folgenden wird jedoch für jede Komponente jeweils nur die letzte Version beschrieben, die in den meisten Fällen auf letzterem Produkt basiert[2].

Schließlich ist noch die Entscheidung über den *ORB* bzw. die Laufzeitumgebung selbst zu treffen. Auch hierfür bietet sich die CORBA–Plattform an, durch die der Begriff eines ORB (*Object Request Broker*) erst etabliert wurde. Wie bereits in Abschnitt 1.3.3 beschrieben, bietet CORBA einen ausgesprochen mächtigen Kommunikationsmechanismus für verteilte Anwendungen an, der vor allem auf Interoperabilität zwischen unterschiedlichen Betriebssystemen, Programmiersprachen und Organisationsdomänen (die jeweils mit einem eigenen ORB ausgestattet sind) abzielt. Jedoch gibt es auch Alternativen, die für weniger starke Interoperabilitätsanforderungen mehr Effizienz (da mit weniger Overhead verbunden) und Flexibilität als der CORBA–ORB bieten. Dazu zählen vor allem das *Remote Method Invocation*–Paket von Java (RMI) [Bog99] und der Voyager–ORB von ObjectSpace [Obj]. Vor allem letzterer stellt einen sehr effizienten und leicht anwendbaren Baukasten von Grundfunktionen wie verteilte Kommunikation, Persistenz, dynamischer Objekterweiterung und Objektmigration dar, die *orthogonal* miteinander verknüpft werden können. Insbesondere die Objektmigration, die eine Grundvoraussetzung zur Realisierung von *mobilen Agenten* (vgl. Abschnitt 1.3.4) ist, ist eine Eigenschaft, die von reinen CORBA–Produkten nicht unterstützt werden. Da die Agentenorientierung in den Anwendungsszenarien (Abschnitt 4.4.4) und auch im zweiten Teil dieser Arbeit eine wichtige Rolle spielt, wurde für die Implementierung *zusätzlich* zum CORBA–ORB auch der Voyager–ORB verwendet, der vor allem bei der Umsetzung der dezentralen Architektur zum Einsatz kam.

4.2 Implementierung des *Policy–Managers*

In diesem Abschnitt wird die Umsetzung der zentralen Regelverarbeitungsarchitektur in Form des Policy–Managers (vgl. Abschnitt 3.2) beschrieben. Die gesamte Funktionalität des Policy–Managers wird in Form einer Klassenbibliothek bereitgestellt (Abschnitt 4.2.1), während seine Ausprägung als CORBA–Dienst durch eine zusätzliche Implementationsschicht realisiert wird (Abschnitt 4.2.2). Weitere technische Details zur Implementation sind in [Göl97, Kun98] zu finden.

[2]Wichtiger erscheint aber in diesem Zusammenhang die oben erwähnte Anforderung der Produktunabhängigkeit!

4.2.1 PM–Bibliothek

Den Kern des Policy–Managers bildet eine objektorientierte, portable Klassenbibliothek, die in "Pur–Java", d.h. unabhängig von Zusatzprodukten, implementiert ist. Damit kann sie von jeder Java–Applikation aufgerufen werden.

4.2.1.1 Aufbau der Bibliothek

Der o.g. Anforderung nach einem *modularen, hierarchischen* Aufbau (s. Abschnitt 4.1.1) wird dadurch Rechnung getragen, dass die zu implementierende Funktionalität in Einzelmodule, die als *Package* bezeichnet werden, zerlegt wird. Zwischen den Packages existieren hierarchische Zugriffsbeziehungen, die in Abbildung 4.1 dargestellt werden.

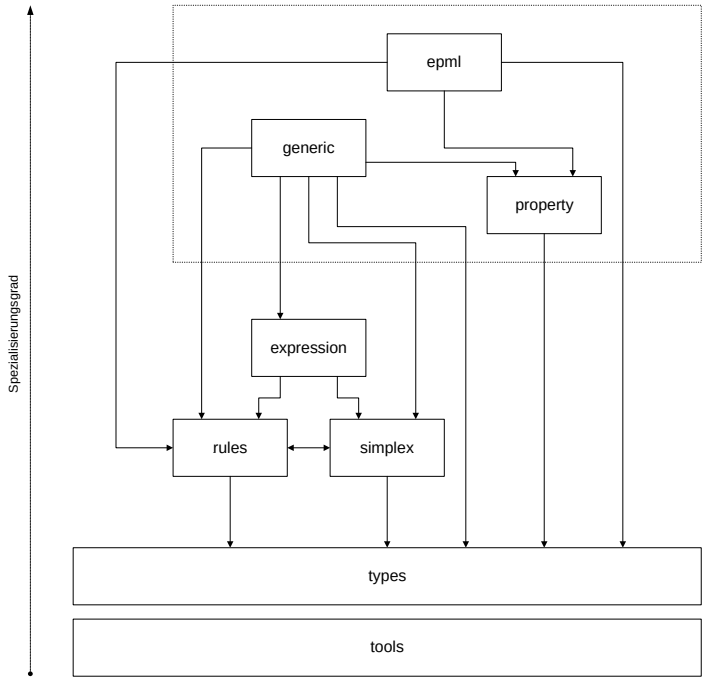


Abbildung 4.1: Aufbau der PM–Bibliothek

Je tiefer ein Package in dieser Hierarchie angesiedelt ist, um so allgemeiner sind die von ihm implementierten Funktionen, d.h. diese können von vielen anderen Packages, auch den in anderen Bibliotheken, genutzt werden. Umgekehrt bieten die höher angesiedelten Packages mehr speziellen Nutzwert für die Anwendungsebene. Beispielsweise wird die gesamte Rule–spezifische Funktionalität, um Anwendungen zu steuern, von den drei höchsten Packages bereitgestellt. Die tieferen Packages implementieren hauptsächlich grundlegende Hilfsfunktionen wie z.B. Simplex–Algorithmen und sollten von der Anwendungsebene nicht direkt genutzt werden.

4.2.1.2 Hilfsklassen

Das Package *tools* enthält grundlegende Hilfsklassen, die von anderen Packages gemeinsam benutzt werden können. Zum einen sind hierin speziell entwickelte Datenstrukturen (z.B. spezielle Varianten von Matrix, Vektor, List etc.), die vor allem zur Implementierung der logischen Verarbeitungsfunktionen benötigt werden, enthalten. Zum anderen enthält dieses Package Klassen, die zum Debuggen und Testen verwendet werden.

Ein zusätzlicher Zweck dieses Packages besteht darin, alle benötigten Hilfsklassen in der PM–Bibliothek zu versammeln, um eine maximale Unabhängigkeit von anderen Java–Bibliotheken zu erreichen.

4.2.1.3 Typsystem

Das Package *types* enthält die Implementation aller *Eigenschaftstypen*, die der Policy–Manager bearbeiten kann. D.h. alle in konkreten Rules referenzierten Eigenschaftsobjekte müssen diesem *PM–Typsystem* entsprechen. Um Werte miteinander vergleichen zu können, müssen sie entweder vom selben Typ sein oder zum selben Typ konvertiert werden können. Das PM–Typsystem unterscheidet im allgemeinen zwischen *Grundtypen*, die allen Rule–sensitiven Anwendungen bekannt sein sollen, und *benutzerdefinierten* Typen, die für bestimmte Anwendungsklassen spezifiziert und nachträglich in das Typsystem aufgenommen werden können.

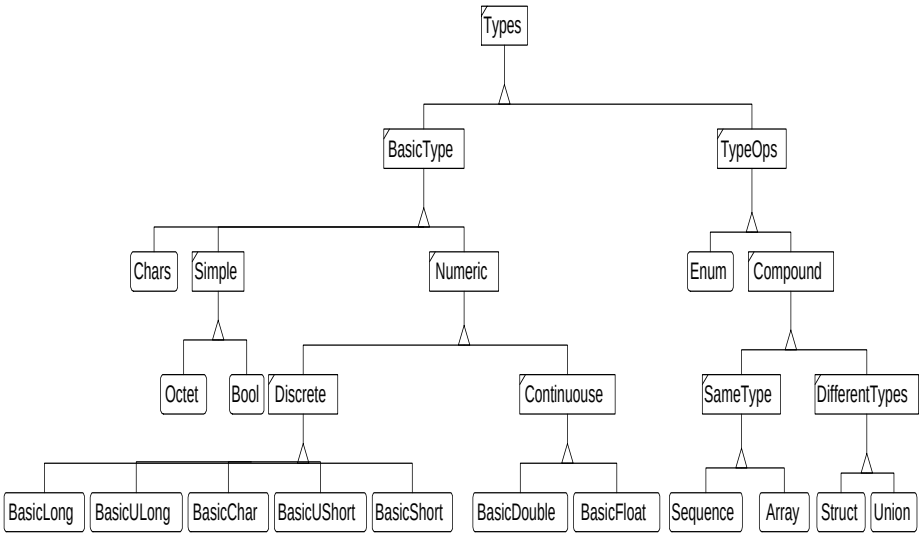


Abbildung 4.2: Klassenhierarchie der Grundtypen im Package *types*

Bei den Grundtypen, deren hierarchische Struktur in Abbildung 4.2 dargestellt ist, wird wie in üblichen Typhierarchien zwischen *atomaren* Typen (*BasicType*) und *strukturierten* Typen (*TypeOps*), die durch Typoperatoren gebildet werden, unterschieden.

Benutzerdefinierte Typen können zwar theoretisch beliebig sein, jedoch basieren viele in der Praxis verwendeten auf einem der numerischen Grundtypen,

die von spezifischen Wertebereichen eingeschränkt werden. Beispielsweise umfasst der Typ (deutsche) *Zensur* ganze Zahlen von 0 bis 15, während sein amerikanisches Gegenstück Buchstaben von A bis F abdeckt. Um die Definition solcher Typen zu unterstützen, wird die abstrakte Klasse *Numeric* deshalb mit einer Methode *getBounds()* ausgestattet, die den Wertebereich eines Subtypen in Form einer Instanz der in Tabelle 4.1 spezifizierten *BoundsSpecification*–Klasse zurückliefert.

Signatur	Beschreibung
QBasicNumeric getUpperBounds()	Gibt die obere Grenze des Wertebereichs wieder.
QBasicNumeric getLowerBounds()	Gibt die untere Grenze des Wertebereichs wieder.
boolean isUpperBoundInclusive()	Zeigt an, ob die obere Grenze inklusiv ist.
boolean isLowerBoundInclusive()	Zeigt an, ob die untere Grenze inklusive ist.
int getCategory()	Gibt die Kategorie wieder.

Tabelle 4.1: Klasse *BoundsSpecification* zur Unterstützung benutzerdefinierter Typen

4.2.1.4 Objektorientierte Repräsentation logischer Ausdrücke

Das Package *rules* stellt alle Basisklassen zur objektorientierten Verabeitung von logischen Ausdrücken zur Verfügung[3]. Abbildung 4.3 zeigt die Klassenhierarchie dieses Package, die mit der Klasse *Rule*, die einen Bedingungsausdruck darstellt, beginnt. Für jeden (vom Benutzer) eingegebenen Bedingungsausdruck in Textform erzeugt der Parser (siehe Abschnitt 4.2.1.9) ein Objekt vom Typ *Rule*. Daher ist die Klassenhierarchie so gewählt, dass das Erstellen eines syntaktischen Baums im Parser nicht notwendig ist.

Demnach besteht ein Rule–Ausdruck entweder aus einem Prädikatsausdruck oder einer logischen Verknüpfung von Prädikaten, die aus Effizienzgründen intern in Binärform repräsentiert wird. Ein Prädikatsausdruck, der einer atomaren Bedingung (vgl. Kapitel 2) entspricht, kann entweder eine Konstante, eine Existenzklausel oder ein Relationsausdruck sein. Soweit spiegelt die Hierarchie sowohl die Basis- als auch die erweiterte Ausdrucksmächtigkeit wider (vgl. Abschnitte 2.3.1.4 und 2.3.3.1).

Unter der Klasse *Relation* wird jedoch unterschieden zwischen Ausdrücken, die gemäß der Basismächtigkeit nur eine Property enthalten (*SingleTerm*), und solchen, die eine lineare Kombination beliebig vieler Properties beinhalten (*MultipleTerm*). Um die Implikationsbeziehung zwischen zwei *MultipleTerms* zu prüfen, wird das in Package 2.2.2 implementierte Simplex–Verfahren direkt angewendet.

[3]Die Klassen in diesem Package repräsentieren also ausschließlich Bestandteile logischer Ausdrücke und sind zu unterscheiden von den in Abschnitt 4.3.1.3 beschriebenen Klassen zur Erzeugung von *Rule–Objekten*.

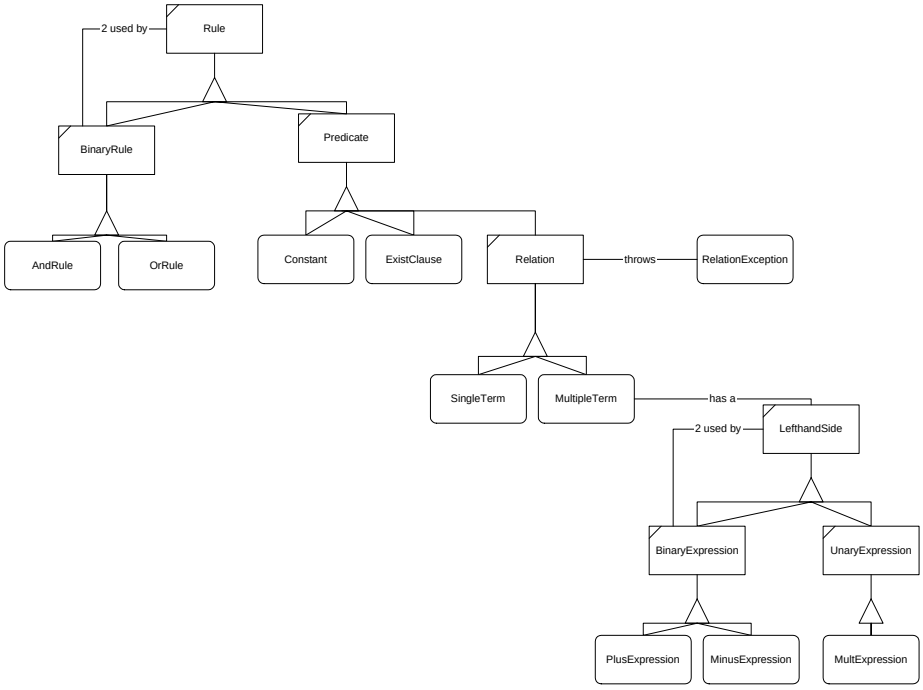


Abbildung 4.3: Package *rules*

4.2.1.5 Simplex–Algorithmen

Das Package *simplex* (siehe Abbildung 4.4) implementiert die in Abschnitt 2.2.2 beschriebenen grundlegenden Simplex–Methoden, die zur Entscheidung von Implikationen zwischen Bedingungsausdrücken der erweiterten Mächtigkeit (vgl. Abschnitt 2.3.3.1) benötigt werden. Kern dieses Packages ist die Klasse *SimplexAlgorithm*, die die Algorithmen des Simplexverfahrens enthält. Als Grunddatenstruktur verwendet *SimplexAlgorithm* ein Simplex–Tableau. Da es mehrere Möglichkeiten gibt, ein solches Tableau darzustellen, wird eine abstrakte Klasse *SimplexTableau* eingeführt, deren Methoden von allen konkreten Tableaus implementiert werden müssen. Auf diese Weise können unterschiedliche Tableaus verwendet werden, ohne die Algorithmen modifizieren zu müssen.

Die Klasse *ShortTableau* stellt eine Implementation des Simplex–Tableaus für verkürzte Tableaus dar. Beim Aufbau des Tableaus wird ein Ungleichungssystem, das einer Konjunktion von atomaren Ausdrücken entspricht (vgl. Abschnitt 2.3.3.2), Zeile für Zeile in ein zulässiges Tableau überführt. Jeder atomare Ausdruck wird dabei entsprechend des in ihm vorkommenden Relationssymbols in eine Restriktion transformiert. Tabelle 4.2 zeigt, wie diese Transformation gemäß der in Abschnitt 2.2.2.3 beschriebenen Zweiphasen–Methode durchgeführt wird.

Zur Bestimmung der Basislösung (eine gültige Belegung der Variablen der Restriktionen) wird die Hilfsklasse *BasisSolution* verwendet. Die Exceptions *EmptySolutionSetException* und *UnboundedException* drücken aus, dass es keine Lösung gibt bzw., dass die Lösungsmenge unbeschränkt ist. *NotValidTableauException* und *NotDualValidException* beziehen sich darauf, dass es sich um kein

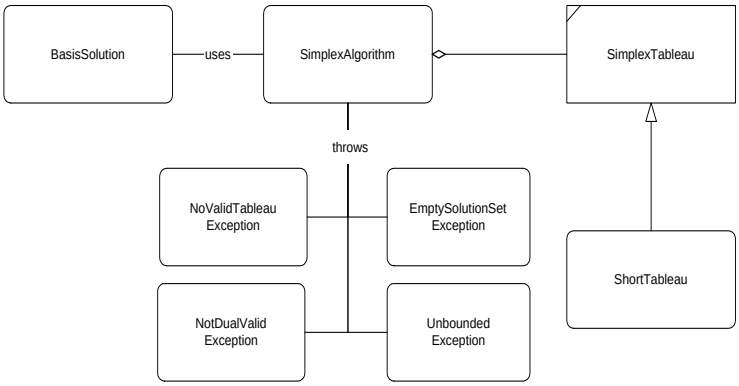


Abbildung 4.4: Package *simplex*

Relation	Aktion
$=$	$x_N = x_N \cup$ *Property–Namen* $x_B = x_B \cup \{h_i\}$
$\leq$	$x_N =$ *Property–Namen* $\cup\, x_N$ $x_B = x_B \cup \{s_i\}$
$\geq$	$x_N = x_N \cup$ *Property–Namen* $\cup \{-s_i\}$ $x_B = x_B \cup \{h_i\}$
$<$	Überführung in $\leq$ durch Subtraktion eines vordefinierten (einstellbaren) Wertes zu *immediate*.
$>$	Überführung in $\geq$ durch Addition eines vordefinierten (einstellbaren) Wertes zu *immediate*.
$\neq$	Darf an dieser Stelle nicht vorkommen, da vorher in ODER–Verknüpfung von $>$ und $<$ überführt.

Tabelle 4.2: Überführung atomarer Ausdrücke in zulässige Tableaus

primal zulässiges Tableau bzw. um kein dual zulässiges handelt.

Wie in Abschnitt 2.3.3.1 bereits erwähnt, müssen bzgl. Regeln der erweiterten Ausdrucksmächtigkeit gewisse Einschränkungen des Typsystems, das in Package *types* dargestellt ist, in Kauf genommen werden. Das liegt daran, dass *vorzeichenbehaftete* Typen zur Nichtexistenz einer Optimallösung führen können, da das Ergebnis des Simplexverfahrens dann unbeschränkt sein kann. In Bezug auf reellwertige Typen stellt dies noch kein Problem dar. Problematisch wird es jedoch bei *ganzzahligen* Property–Typen, da der ganzzahlige Simplex–Algorithmus (2.2.4) per Definition nur auf dem Optimum der stetigen Lösung arbeiten kann. Bei Nichtexistenz einer Optimallösung kann es zwar unendlich viele stetige Lösungen geben, es muss jedoch keine ganzzahlige Lösung dabei sein. Es gibt im wesentlichen zwei Alternativen, diesem Dilemma zu begegnen. Die erste besteht darin, auf die Benutzung von ganzzahligen Typen komplett zu verzichten. Damit würde das Typsystem in Bezug auf die erweiterte Ausdrucksmächtigkeit um die gesamten diskreten Typen reduziert werden. Die zweite Alternative ist, auf die Benutzung vorzeichenbehafteter Typen zu verzichten. In der erweiterten Version der PM–Bibliothek ist die zweite Lösung gewählt.

Dementsprechend wird bereits beim Parser (siehe Package *epml*) im Falle eines vorzeichenbehafteten Typen eine Exception ausgelöst.

4.2.1.6 Normalformen

Das Package *expressions* (siehe Abbildung 4.5) stellt spezielle Darstellungsformen logischer Ausdrücke — insbesondere die Normalformen DNF und KNF — zur Verfügung, die von den in Abschnitt 2.3.2 beschriebenen Verarbeitungsfunktionen vorausgesetzt werden. D.h. solche Funktionen werden als Methoden der Klassen dieser Ebene implementiert.

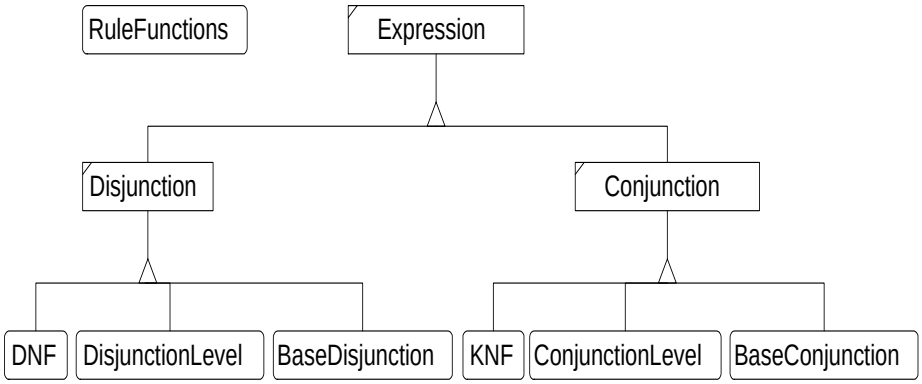


Abbildung 4.5: Package *expressions*

Wichtig für eine effiziente Implementierung der entsprechenden Algorithmen ist, dass eine *Mengensicht* auf den Bestandteilen einer logischen Formel realisiert wird, da in den beschriebenen Algorithmen meist mit Mengen von Disjunkten, Konjunkten und Atomen operiert wird. Um diese Mengensicht zu realisieren, wurde eine spezielle Listenstruktur mit einem typsicheren Iterator implementiert und im Package *tools* bereitgestellt (s.a. [Göl97, S. 28]). Die Klasse *DNF* enthält dann eine solche Liste der mit ODER verknüpften Konjunkten. Diese Konjunkte werden wiederum mit der Klasse *BaseConjunction* beschrieben, die eine Liste der atomaren Ausdrücke (vom Typ *Rule* im Package *rules*) enthält. Die Klasse *DisjunctionLevel* enthält eine Liste beliebiger Ausdrücke vom Typ *Expression*, die als Hilfskonstrukt zur Bildung von Instanzen der Klasse *DNF* verwendet wird. Analoges gilt für die Klasse *KNF*, welche die Konjunktive Normalform bereitstellt.

4.2.1.7 Zugang zu Properties

Sowohl die Generizität als auch Praxisbezogenheit der in den vorangegangenen Kapiteln konzipierten Regelformalismen besteht darin, dass sie ausschließlich durch die Modifikation von Eigenschaften — die im technischen Sinne als Properties bezeichnet werden — das Laufzeitverhalten von Anwendungen steuern bzw. beeinflussen. Properties finden sich in nahezu jedem Anwendungsbereich, vom System–Management über graphische Softwareentwicklungsumgebungen bis hin zu Konfigurationsschnittstellen für die allermeisten Standardanwendungen. Deshalb ist es sehr wünschenswert, auf *beliebige* Properties, die einem Mindestmaß an Konformitätsanforderungen genügen, zugreifen zu können. Deshalb

wird auf der Implementierungsebene ein so genannter Zugriffsadapter zwischen den Policy–Manager und den Property–Service (vgl. Abbildung 3.8) geschaltet, der die Anbindung von beliebigen Eigenschaftsdiensten, die das minimalistische Interface *AccessAdaptor* erfüllen, ermöglicht.

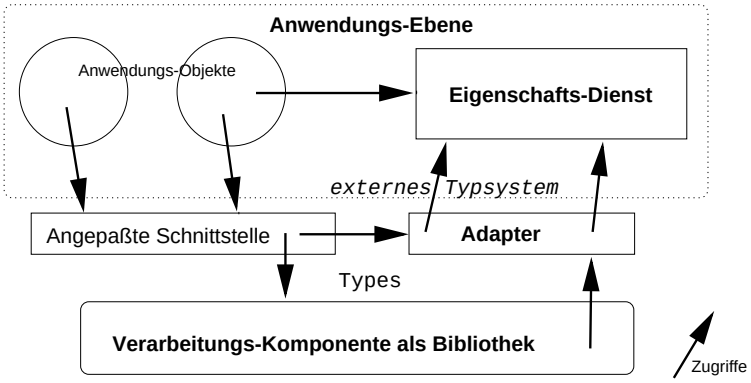


Abbildung 4.6: Benutzung der Adapter–Klassen im Package *property*

Dieses Interface besteht aus lediglich drei Methoden zum Lesen (*get()*), Schreiben (*set()*) und zur Prüfung der Existenz (*isDefined()*) einer Property. Durch diese zusätzliche Zugriffsschicht (siehe Abbildung 4.6) war es möglich, in den Anwendungsszenarien eine Vielzahl von nicht CORBA–konformen Eigenschaftsspeichern neben dem standardisierten Property–Service (s.a. Abschnitt 4.2.2), u.a. auch die *Registry* unter MS–Windows, zu verwenden. Das Wechseln von einem Eigenschaftsspeicher zu einem anderen kann sogar zur Laufzeit erfolgen. Damit kann der Policy–Manager auch in Fällen genutzt werden, in denen die relevanten Properties (z.B. eines Anbieters und eines Kunden) in *unterschiedlichen* Eigenschaftsspeichern bzw. dezentral abgelegt sind.

Zu Testzwecken und auch zur Unterstützung nicht CORBA–fähiger Anwendungen wurde ein leichtgewichtiger *JavaAdaptor*, der auf den in jeder Java–Laufzeitumgebung vorhandenen *System Properties* basiert, implementiert, der ebenfalls Bestandteil des Package *property* ist.

4.2.1.8 Generische Programmierschnittstelle

Das Package *generic* — dargestellt in Abbildung 4.7 — stellt aus programmiertechnischer Sicht die höchste Ebene der PM–Bibliothek dar. Es enthält als Haupt–Klasse ein *Policy*–Objekt, das als eine generische Programmierschnittstelle zur PM–Bibliothek betrachtet werden kann, da es alle Verarbeitungsfunktionen (im wesentlichen Evaluierung, Vergleiche, Unifikation, Optimierung und Durchsetzung), die auf einen Bedingungsausdruck angewendet werden können, in sich vereinigt. *Evaluation*, *Activation* und *Optimization* dienen als Hilfsklassen zur Implementierung der entsprechenden Funktionen. Durch diese Konzentration aller für den Anwender relevanten Funktionen auf eine einzige Klasse wird der Zugang zur PM–Bibliothek erleichtert, da in den meisten Fällen nur die Schnittstelle dieser Klasse direkt benutzt werden muss.

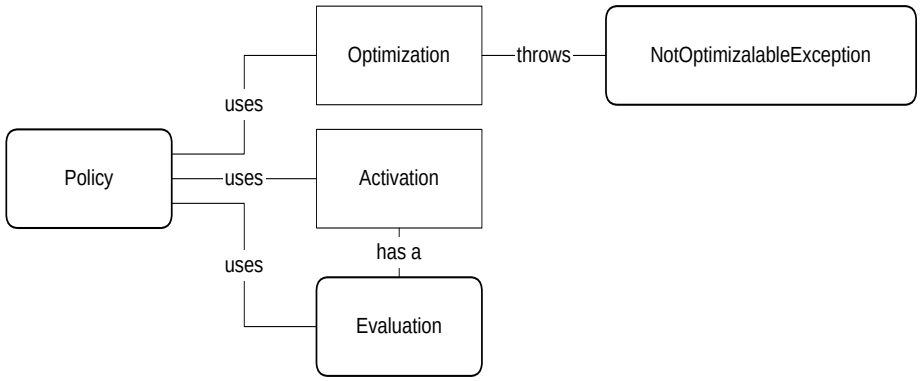


Abbildung 4.7: Package *generic*

4.2.1.9 Sprachzugang

Das Package *epml* bietet einen sprachlichen Zugang zu der kompletten Funktionalität der PM–Bibliothek. Die zu Grunde liegende Sprache, als *Extended Policy Manager Language* (EPML) bezeichnet, zielt vor allem auf die Unterstützung von Anwendern, die mittels jedes simplen Editors oder Eingabe–Tools Regeln zur Steuerung ihrer Anwendungen formulieren, und basiert technisch auf dem weit verbreiteten Unicode–Zeichensatz [Con96].

Die Bestandteile von EPML korrespondieren direkt mit den entsprechenden Klassen aus den Packages *rules* und *types*, die mit Absicht so entworfen wurden, um eine möglichst effiziente Transformation (bzw. Parsen) von der textuellen Repräsentation in Programmobjekte und umgekehrt zu erzielen. Dabei wurde jedoch auch auf eine benutzerfreundliche Gestaltung der Syntax geachtet. Beispielsweise wird eine Angabe wie `speed >= 5` zugelassen, obwohl daraus nicht deutlich erkennbar ist, um welchen Relations- und Werttyp es sich dabei handelt. In diesem Fall wird der Wert defaultmäßig als ganzzahlig und vorzeichenunbehaftet vom Typ *Long* interpretiert. Bei `speed S>= 5` wird der Wert mit einem Datenobjekt vom Typ *Short* eingeparst und bei `speed S>= 'low'` wird ein Typfehler erkannt.

Hauptklassen des Package *epml* sind ein lexikalischer *Analyzer*, der als ein endlicher Automat implementiert ist, und ein *Parser* zur Erkennung und Transformation von EPML als einer kontextfreien LL(1)–Grammatik. Die Zustände des entsprechenden Automaten sowie die komplette Syntax von EPML sind in Anhang A.2 angegeben.

4.2.2 CORBA–Anbindung

Die in Abschnitt 3.2.3.1 dargestellte Architektur des Policy–Managers als eine Sammlung von CORBA–Diensten lässt sich nun auf eine sehr geradlinige Weise verwirklichen, da alle benötigten regel–spezifischen Funktionen bereits von der PM–Bibliothek zur Verfügung gestellt werden. Im wesentlichen müssen noch die entsprechenden drei Dienste, nämlich *Policy Processing Service*, *Policy Persistence Service* und *Property Service* in der CORBA Interface Definition Language (IDL) spezifiziert, mittels des IDL–Compilers CORBA–Klassen generiert und dann implementiert werden.

Bezüglich des *Processing Service* lassen sich die CORBA–Klassen direkt abbilden in entsprechende in der PM–Bibliothek. Der *Persistence Service* hat die Aufgabe, CORBA–Objekte vom Typ *Policy* persistent zu verwalten, und da CORBA–Produkte wie der in Abschnitt 4.1.2 genannter *VisiBroker* die Objektserialisierung von Java unterstützt, lässt sich die Hauptfunktion dieses Dienstes — nämlich Objektpersistenz — auf eine simple Weise lösen.

Die Bereitstellung des *Property Service* stellte sich dagegen als eine ziemlich aufwendige Aufgabe heraus, da trotz der relativ frühzeitigen Standardisierung (1996) dieser Dienst in keinem der untersuchten CORBA–Produkte verfügbar war, so dass im Rahmen des Policy–Manager–Projektes eine eigene Implementierung durchgeführt werden musste. Dabei wurde festgestellt, dass der entsprechende CORBA–Standard (siehe [OMG98b]) mit einem deutlichen Manko, nämlich der fehlenden Unterstützung der Persistierung von Properties, behaftet ist. Der Grund dafür ist, dass in den Schnittstellen des CORBA Property Service die *PropertySets* und verwandte Klassen nicht mit Namen referenziert werden können. Um dennoch Property–Persistenz zu unterstützen, mussten diese Schnittstellen deshalb entsprechend erweitert werden. Weitere Details über die Implementation dieses Dienstes sind in [Sch98] zu finden.

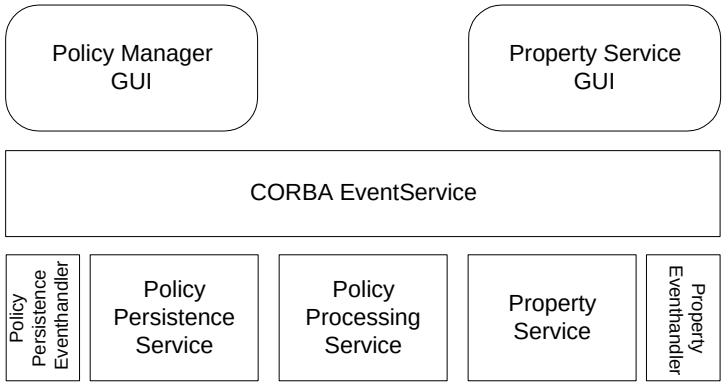


Abbildung 4.8: Implementationssicht der CORBA–Anbindung

Erfahrungen aus den ersten Prototypen des Policy–Managers haben gezeigt, dass es zur Unterstützung der Klienten (Dienstnutzer) sehr sinnvoll ist, eine ereignisorientierte Kommunikation (zusätzlich zu den operationalen Aufrufschnittstellen der Dienste) in den Policy–Manager zu integrieren. Deshalb wurde auf Implementationsebene (im letzten Prototypen) die in Abbildung 4.8 gezeigte Architektur umgesetzt. Darin wird — im Gegensatz zur funktionalen Sicht in Abbildung 3.8 — der CORBA Event Service (des VisiBroker–Produktes) als integrierter Ereigniskanal zur Kommunikation mit der Anwendungsebene verwendet. Durch die Erweiterung des *Property Service* und des *Policy Persistence Service* um jeweils einen spezifischen *Eventhandler* ist es nun möglich, dass registrierte Anwendungen über Änderungen von Properties bzw. Rules automatisch benachrichtigt werden, anstatt diese selber prüfen zu müssen. Diese zusätzliche Funktionalität erleichtert beispielsweise die Entwicklung von Werkzeugen, mit denen ein Benutzer die dynamischen Properties seiner Anwendung verfolgen kann, in erheblichem Masse. Die vollständige Spezifikation der drei Hauptdienste des Policy–Managers in CORBA–IDL ist in Anhang A.3 angegeben.

4.3 Implementierung Rule–sensitiver verteilter Anwendungen

Im folgenden wird die Implementierung der in Abschnitt 3.3 beschriebenen dezentralen Regelverarbeitungsmechanismen für verteilte Anwendungen erläutert (siehe auch [TGML99, Kun00, Mar99]). Auch diese Funktionalität wird in Form einer portablen Java–Bibliothek, die als RS–Bibliothek (Bibliothek für *Rule–sensitive* Anwendungen) bezeichnet wird, bereitgestellt. Im Gegensatz zur PM–Bibliothek sind die Klassen in dieser Bibliothek aber nicht völlig unabhängig von kommerziellen Erzeugnissen, sondern basieren auf den Produkten VisiBroker — als Lieferant der CORBA–konformen Event und Transaction Service — sowie ObjectSpace Voyager — als Basisklassen für das RS–DII und mobile Agenten, da die entsprechenden Teilfunktionalitäten als integrale Bestandteile des Konzepts Rule–sensitiven Anwendungen (s. Abschnitt 3.3.1), insbesondere *Rule–sensitiver mobiler Agenten* [TGML99], zu betrachten sind. Wie die Beschreibung der Packages zeigen wird, bietet die RS–Bibliothek jedoch eine hohe Abstraktion von den verwendeten Produkten, um eine mögliche Umstellung auf andere Produkte oder Produktversionen zu erleichtern.

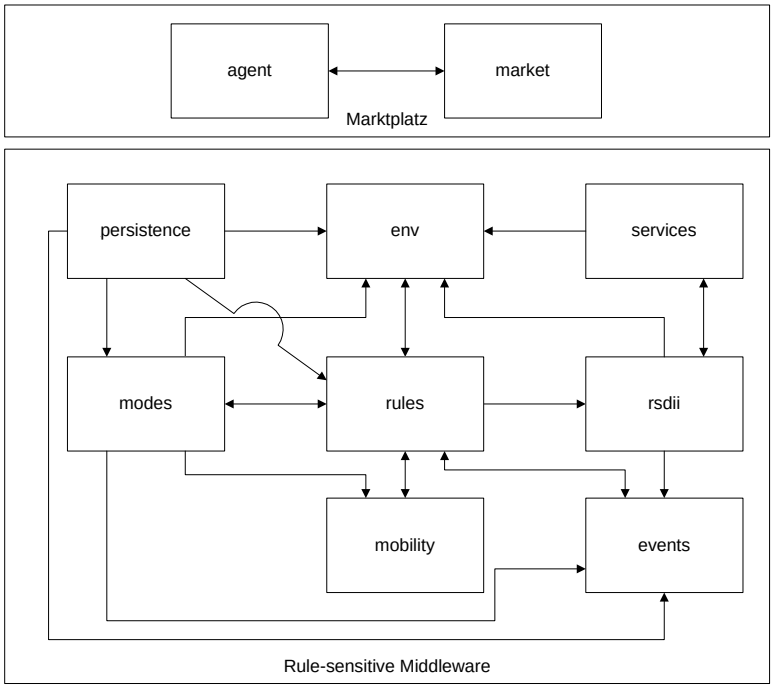


Abbildung 4.9: Aufbau der RS–Bibliothek

Die Gesamtstruktur der RS–Bibliothek ist in Abbildung 4.9 dargestellt. Die Pfeile stellen Aufrufabhängigkeiten zwischen den einzelnen Packages dar. Die Bibliothek ist gegliedert in zwei Hauptschichten: die der RS–Middleware (Abschnitt 4.3.1) und die der agenten–basierten Marktplätze (Abschnitt 4.3.2).

4.3.1 Rule–sensitive Middleware

Die Rule–sensitive Middleware ist implementiert als eine zusätzliche Schicht über einer Middleware. In diesem Fall besteht die zu Grunde liegende Middleware aus den Diensten von zwei Produkten, einerseits dem ORB von ObjectSpace Voyager [Obj] und den Event und Transaction Services von Inprise Visibroker [INP]. Der ORB von Voyager wird deshalb benutzt, da er eine direkte Unterstützung von mobilen Agenten, die auf der anwendungsnahen Ebene der Marktplätze (Abschnitt 4.3.2) eingesetzt werden (vgl. a. Abschnitt 4.1.2).

4.3.1.1 Mobilität

Da Rule–Objekte selbst mobil, d.h. in der Lage sein sollen, als vollwertige Objekte von einer Rule–sensitiven Anwendung zur nächsten zu migrieren, muss diese Eigenschaft von der RS–Bibliothek bereitgestellt werden. Um die Mobilität als eine eigenständige Funktionalität abzugrenzen, wird sie durch ein eigenes Package bereitgestellt (siehe Abbildung 4.10).

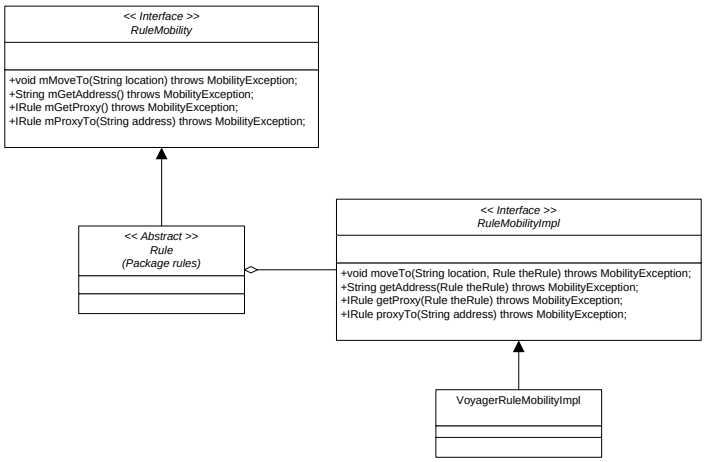


Abbildung 4.10: Package *mobility*

Dieses Package enthält die Schnittstelle *RuleMobility*, die von allen Objekten vom Typ *Rule* zu implementieren ist. Um dies zu erfüllen, kann eine Rule–Klasse eine beliebige fertige Implementation aggregrieren, die das Interface *RuleMobilityImpl* implementiert. Dies entspricht dem bekannten Design–Pattern *Bridge*, wie es in [GHJV95] beschrieben ist. Als fertige Implementation bietet das Package die Klasse *VoyagerRuleMobilityImpl* an, die auf der Mobilität von Voyager–Objekten basiert[4].

4.3.1.2 Rule–Events

Die zu verschiedenen Rule–Event–Typen (s. Abschnitt 3.3.3.2) korrespondierenden Klassen sowie alle Klassen zur Erzeugung, Verbreitung und Zustellung von Events sind im Package *events* implementiert.

[4]Hieran ist deutlich zu sehen, dass eine Umstellung auf ein anderes Basisprodukt einfach durch den Austausch dieser einzigen Klasse erfolgen kann, ohne andere Klassen modifizieren zu müssen.

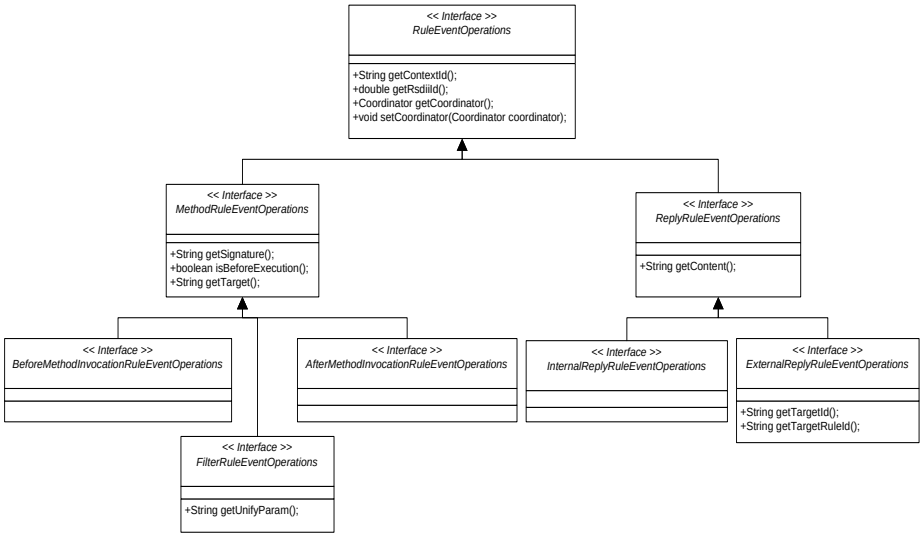


Abbildung 4.11: Funktionalität von Rule–Events

Die Events sind als CORBA–Objekte implementiert, da sie durch den Event Service von Visibroker transportiert werden sollen. Da diese CORBA–Objekte bereits in einer Vererbungshierarchie stehen, ist der Delegationsansatz zur Implementierung der Anwendungsfunktionalität von Rule–Events gewählt worden[5]. Demnach implementieren alle Event–Klassen ihre jeweiligen *Operations*–Schnittstellen, die in Abbildung 4.11 gezeigt werden. Die eigentliche Rule–Event–Typhierarchie (siehe Abbildung 3.18 und IDL–Spezifikation in Anhang B.2) korrespondiert exakt zu dieser Interface–Hierarchie, die die Funktionalität der Events genauer charakterisiert.

Da der Event Service von Visibroker nicht typisiert ist[6], können Events nicht gemäß ihrem Objekttyp zugestellt werden, sondern müssen durch einen zusätzlichen Filter auf Empfängerseite heraussortiert werden. Um jedoch die Vervielfältigung von unnötigen Events zu vermeiden und damit das Laufzeitverhalten des Gesamtsystems zu optimieren, werden in der vorliegenden Implementation nicht nur ein Event–Channel–Prozess, sondern drei Event–Channels benutzt, deren Aufgaben wie folgt verteilt sind:

- *Sender Channel*: Dieser Kanal ist zuständig für die *RS–DII–zu–Rule–Kommunikation*. Instanzen vom Typ *MethodRuleEvent* werden vom RS–DII in diesen Kanal verschickt und die Rule–Objekte bzw. Rule–Container bekommen diese Events über diesen Kanal geliefert.

- *Receiver Channel*: Dieser Kanal ist zuständig für die *Rule–zu–RS–DII–Kommunikation*. Das RS–DII bekommt Instanzen vom Typ *ReplyRuleEvent* über diesen Kanal geliefert, die von Rule–Objekten bzw. Rule–Containern dort hineingeschickt werden.

[5]In Java ist nämlich keine Mehrfachvererbung auf Implementations-, jedoch auf Interface–Ebene möglich.

[6]In dieser Hinsicht ist das Visibroker–Produkt (noch) nicht wirklich CORBA–konform.

- *Rule Channel*: Dieser Kanal ist zuständig für die *Rule–zu–Rule–Kommunikation*. Über diesen Kanal werden alle von Rule–Objekten (im einem externen Modus) ausgelösten Filter–Events (vom Typ *FilterRuleEvent*) und entsprechende Antworten vom Typ *ExternalReplyRuleEvent* verschickt und empfangen.

Klassen, die zur Erzeugung, Verbreitung und Zustellung von Rule–Events benötigt werden (*EventFactory, RuleEventSupplier, RuleEventConsumer, RuleEventSupplierFactory,..., LocalEventDispatcher* etc.), bilden eine zusätzliche Hierarchie neben der Rule–Event–Hierarchie. Auf die soll hier jedoch nicht näher eingegangen werden. Details darüber werden in [Kun00] beschrieben.

4.3.1.3 Rule–Objekte

Implementationen der Rule–Typen sind in einem eigenen Package *rules* zusammengefasst[7]. Die Schnittstellen der Rule–Klassenhierarchie sind in Abbildung 4.3 dargestellt.

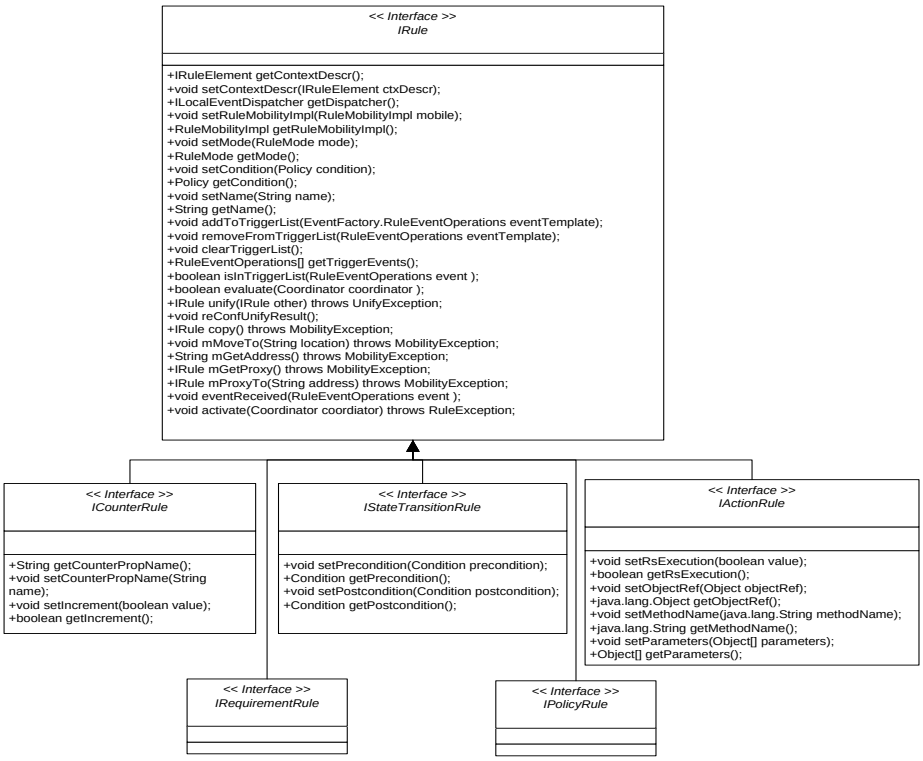


Abbildung 4.12: Schnittstellen der Rule–Klassen

Außer den vier konkreten Rule–Typen (in Abbildung 4.12 durch die Interfaces *IRequirementRule*, *IStateTransitionRule*, *IPolicyRule* und *IActionRule* repräsentiert), die in Abschnitt 2.3.1.3 konzeptionell eingeführt wurden, ist zur

[7]Dieses Package ist zu unterscheiden von dem gleichnamigen Package (Abschnitt 4.2.1.4) in der PM–Bibliothek, die als Systembibliothek von der RS–Bibliothek verwendet wird.

Unterstützung bestimmter Anwendungsklassen die *CounterRule* als ein weiterer Rule–Typ hinzugefügt worden. Dieser Typ, der beispielsweise zur Steuerung von Verhandlungsszenarien (s. Abschnitt 6.4) eingesetzt wird, ist deshalb nur auf der Implementierungsebene eingeführt worden, da er semantisch gesehen als ein Subtyp der *ActionRule* angesehen werden kann. Da dieser neue Typ jedoch ausschließlich zur Inkrementierung von Properties verwendet wird, ist es effizienter, eine spezifische Implementation zu realisieren, die auf eine explizite Spezifikation der Aktion verzichtet und stattdessen die Möglichkeit bietet, die zu modifizierende Property und die Inkrementierungsschrittweite dynamisch zu ändern. Auf diese Weise wird zudem die erwähnte Erweiterbarkeit des in Kapitel 2 beschriebenen Regelkonzepts konkret demonstriert.

Alle konkreten Rule–Klassen erben von der abstrakten Oberklasse *Rule*. Dort ist die gesamte Grundfunktionalität eines Rule–Objektes, die auch den überwiegenden Teil seiner operationalen Schnittstelle ausmacht, implementiert. Die Verbindung zwischen einem Rule–Objekt und der es beherbergenden Anwendung wird durch den Zugriff auf die Kontextbeschreibung (durch *getContextDescr()*) hergestellt. Darüber kann ein Rule–Objekt ermitteln, in welcher Rule–Set es sich befindet. Die Kontextbeschreibung lässt sich dynamisch ändern (durch *setContextDescr()*), z.B. wenn eine Rule durch Migration zu einer anderen Anwendung gewandert ist. Eine solche Migration wird dadurch ermöglicht, dass die Klasse *Rule* das Interface *RuleMobility* (in Abschnitt 4.3.1.1) implementiert, und zwar dadurch, dass mittels der Methode *setRuleMobilityImpl()* eine fertige Implementation der Mobilitätseigenschaft eingebunden wird. Über die Methode *mMoveTo()* kann dann eine Migration ausgelöst werden. Die weiteren Methoden der Klasse *Rule* dienen der Umsetzung der in Abschnitt 2.3.1.2 beschriebenen deklarativen und funktionalen Semantik einer Regel.

4.3.1.4 Rule–Modi

Die in Abschnitt 3.3.2 vorgestellten Aktivierungsmodi für Rule–Objekte und damit verknüpfte Zweidimensionalität der Regelsemantik bedeuten, dass ein Rule–Typ abhängig von dem ihm zugewiesenen Modus jeweils ein völlig unterschiedliches Verhalten aufweist. Der simpelste Ansatz zur Realisierung der Modi wäre eine feste Verknüpfung jedes Modus jeweils mit einem Rule–Typ. Da eine solche statische Zuweisung der Modi zu Rule–Typen jedoch einer n–fachen Vervielfachung der Typen gleichkäme, kommt nur eine dynamische Zuweisung der Modi als adäquate Lösung in Frage. Hierzu bietet sich wieder die Verwendung eines Design–Pattern an, und zwar des *State*–Pattern, welches eine Korrelation des Verhaltens einer Komponente mit ihrem aktuellen *Zustand* herstellt [GHJV95].

Die Implementation der Rule–Modi findet sich im Package *modes*, das in Abbildung 4.13 dargestellt ist. Dabei erben alle Rule–Modi von der abstrakten Klasse *RuleMode*, die durch ihre abstrakte Methode *activate()* erzwingt, dass jeder konkrete Modus eine eigene Implementation für die Aktivierung eines Rule–Objektes zur Verfügung stellt (wodurch die Abhängigkeit zwischen Verhalten und Modus hergestellt ist). Außerdem implementiert *RuleMode* die transaktionale Absicherung aller Modi, so dass der Modus eines Rule–Objektes nicht verändert werden kann, wenn dieses Rule–Objekt gleichzeitig an mehreren Transaktionen teilnimmt, was eintreten kann, wenn innerhalb des Rule–Systems mehrere Rule–sensitive Aufrufe nebenläufig durchgeführt werden. Diese Absi-

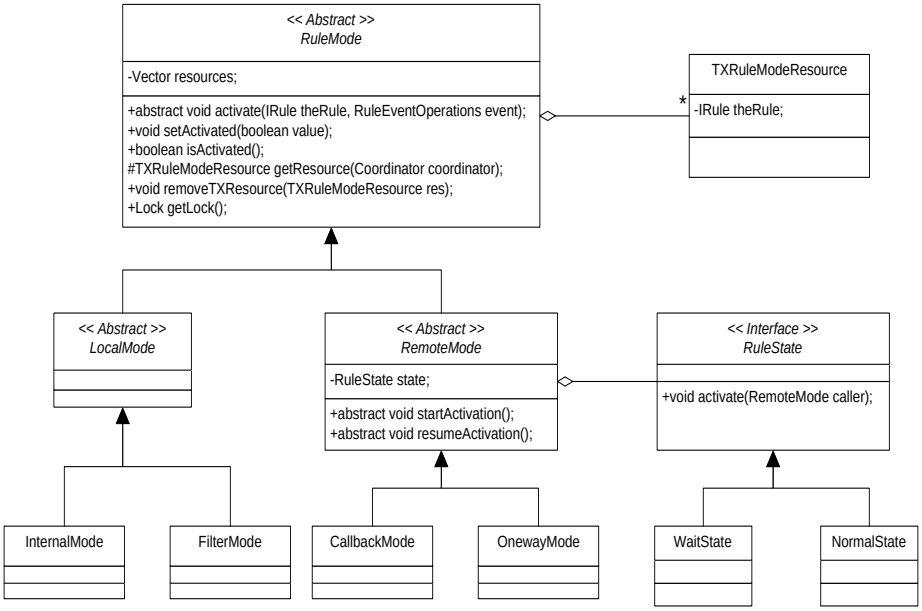


Abbildung 4.13: Package *modes*

cherung wird durch die Hilfsklasse *TXRuleModeResource* (deren Implementation auf dem Visibroker Transaction Service basiert) bereitgestellt, die von *RuleMode* aggregriert wird.

Wie auch in Abbildung 3.14 gezeigt, sind Rule–Modi hier weiter klassifiziert in die abstrakten Klassen *LocalMode* und *RemoteMode*. Die konkreten Unterklassen von *RemoteMode* — *OnewayMode* und *CallbackMode* — unterscheiden sich von den lokalen Modi — *InternalMode* und *FilterMode* — dadurch, dass sie ein Filter–Rule–Event verschicken und dann, auf das externe Antwort–Event wartend, im Wartezustand verharren können. Deshalb aggregriert *RemoteMode* zusätzlich einen *RuleState*, der entweder die Ausprägung *WaitState* oder *NormalState* annehmen kann.

4.3.1.5 RS–DII

Die Implementation des in den Abschnitten 3.2.4.2, 3.3.1.2 und 3.3.3.2 erwähnten Rule–sensitiven Aufrufmechanismus findet sich im Packgage *rsdii* (siehe Abbildung 4.14).

Das RS–DII wird hier durch eine generische Schnittstelle (Interface *Dynamic*) repräsentiert, die eine *send()*–Methode anbietet, mit der ein beliebiger Aufruf an einem beliebigen Zielobjekt innerhalb eines Timeouts, das mit *setTimeout()* eingestellt wird, durchgeführt werden kann. Diese Schnittstelle ermöglicht eine hohe Anwendungstransparenz, weil sich dahinter ein beliebiger dynamischer Aufrufmechanismus, beispielsweise CORBA–konform oder nicht bzw. Rule–sensitiv oder nicht, verbergen kann.

Als Ausprägung eines Rule–sensitiven DII wurde die Klasse *ViVoDynamic* entwickelt, deren Bezeichnung daher rührt, dass die Produkte Visibroker und Voyager zu ihrer Funktionalität beitragen. Ihre fünf statischen globalen Attri-

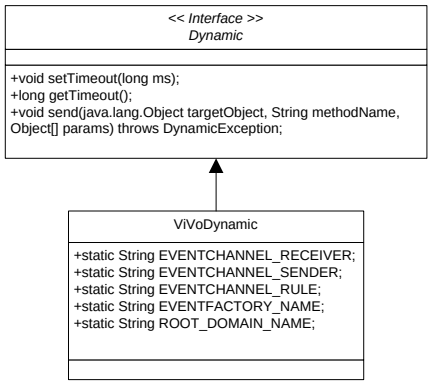


Abbildung 4.14: Package *rsdii*

bute legen die Namen der Event–Channels (s. Abschnitt 4.3.1.2), des Domänen–/Namensdienstes (s. Abschnitt 4.3.1.7) und der Event–Factory (zur Erzeugung von Trigger–Events) fest. Durch eine jeweilige Anpassung dieser Attribute ist es möglich, mehrere Rule–Systeme in disjunkten Domänen zu betreiben.

Allerdings konnte in dieser Implementation aufgrund technischer Inkompatibilitäten der eingesetzten Produkte der Transaktionskontext nicht als Parameter einer Methode übergeben werden[8]. D.h. die transaktionale Absicherung eines Methodenaufrufs kann nur auf Ebene der entsprechenden Properties (s. Abschnitt 4.3.1.8) bzw. *indirekt* mittels einer Action–Rule, deren Aktion eine Methodenspezifikation ist, der eine beliebige Parameterliste übergeben werden kann (vgl. Abbildung 4.3), realisiert werden.

4.3.1.6 Persistenz

Obwohl Persistenz nicht als explizite Anforderung aufgestellt wurde, ist sie dennoch in Bezug auf die praktische Anwendbarkeit von regelbasierten Steuerungsmechanismen offensichtlich unverzichtbar. Bereits bei der Erprobung der frühesten Prototypen stellte sich die Notwendigkeit heraus, die von einem Benutzer einmal formulierten und eingesetzten Regeln zu persistieren, um sie beim nächsten Durchlauf der Anwendung wieder zu verwenden. In praktischen Szenarien müssen Regeln zur Steuerung einer Anwendung nämlich meist nur einmal entworfen (und erprobt) werden, danach ändern sie sich von ihrem Typ und ihrer logischen Struktur kaum, sondern müssen nur bezüglich der konkreten Property–Werte, beispielsweise des *Timeouts* für eine Trader–Anfrage, angepasst werden.

Deshalb wurden frühzeitig Mechanismen zur einfachen und portablen Persistierung von Rule–Objekten entwickelt, um diese in beliebige Anwendungsprozesse wieder einbinden zu können. Hierzu bietet sich als nächstliegende und einfachste Lösung zunächst der in Voyager integrierter Persistenzmechanismus dar, da Rules aufgrund ihrer Mobilität ohnehin Voyager–Objekte sind, zu deren Persistierung im Prinzip nur die Standardmethode *save()* bei diesen Objekten

[8] Der Transaktionskontext des Visibroker Transaction Service ist nämlich nicht serialisierbar wird infolgedessen ungültig, wenn die Methode mittels des DII–Mechanismus von Voyager entfernt, also in einem anderen Prozess, aufgerufen wird.

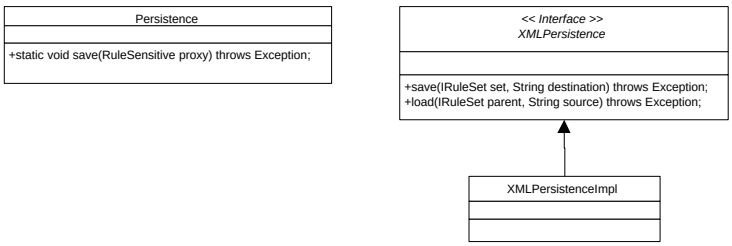


Abbildung 4.15: Package *persistence*

selbst aufgerufen werden muss. Dieser sehr einfache und elegante Persistenzmechanismus ist jedoch in der praktischen Benutzung mit einigen Problemen behaftet. Erstens werden beim Aufruf der *save()*–Methode die Netzwerkadresse (IP) des zu speichernden Objektes in einer mit ihm fest verknüpften Form mit persistiert, d.h. das Objekt kann später nur an derselben Adresse wiederbelebt werden. Damit ist die zu Grunde liegende Voyager–Datenbank nicht portabel. Zweitens basiert die Voyager–Objektpersistierung ihrerseits auf dem Java–Serialisierungsmechanismus, der grundsätzlich die gesamte transitive Hülle des betreffenden Objektes mit persistiert. Diese Eigenschaft ist zwar für viele verteilte Anwendungen vorteilhaft, da ein verteiltes Objekt meist nicht ohne die von ihm referenzierten Klassen arbeiten kann. In diesem Fall bedeutet sie aber, dass die Rule–sensitive Anwendung, die das Rule–Objekt enthält, (als Kopie) mit abgespeichert wird. Ändert sich bei der laufenden Anwendung dann beispielsweise ein Attribut, dann sind die Serialisierungsströme nicht mehr kompatibel.

Um diese Probleme zu vermeiden, wurde neben einem auf Voyager basierten Persistenzmechanismus noch ein zusätzlicher auf Basis der *Extensible Markup Language* (XML) [XML] entwickelt. Beide Persistierungsmethoden werden durch entsprechende Klassen des Package *persistence* implementiert, das in Abbildung 4.15 gezeigt ist. Dabei stellt das Interface *XMLPersistence* die Programmierschnittstelle der XML–basierten Persistierung dar. Die zur Interpretation der eine Rule–Set und ihre Elemente enthaltenden XML–Seite benötigte *Document Type Definition* (DTD) *RuleBase* ist in Anhang B.4 angegeben.

4.3.1.7 Domänen/Namensdienst

Die Implementation des in Abschnitt 3.3.3 eingeführten *Domain/Name Service* findet sich im Package *services*. Dieser Dienst ist — entsprechend den anderen Basisdiensten der RS–Middleware — als ein CORBA–Objekt implementiert. Seine in CORBA–IDL spezifizierte operationale Schnittstelle ist in Anhang B.1 zu finden. Abbildung 4.16 zeigt die Schnittstelle der entsprechenden Implementationsklasse mit den wichtigsten Methoden.

Als *Name*–Service bietet dieser Dienst einen Mechanismus zur Auflösung von *Aliasen* zu Objektadressen an, damit Rule–sensitive mobile Anwendungen die Adressen anderer Rule–sensitiver Anwendungen herausbekommen können, gleichgültig wo diese sich gerade aufhalten. Diese auf Voyager–Aliasen basierte Funktion prüft jeden Alias auf Eindeutigkeit, da er auch als *Kontext–ID* verwendet wird.

Als *Domain*–Service bietet dieser Dienst einen Mechanismus zur Generie-

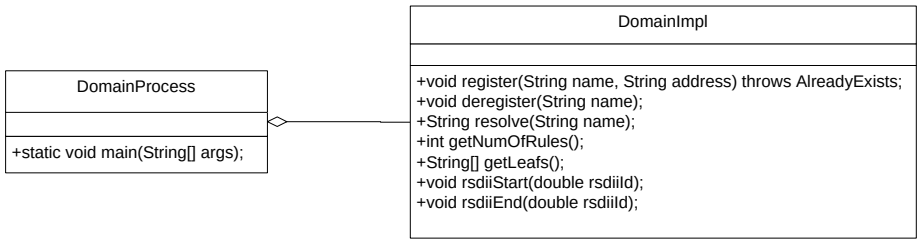


Abbildung 4.16: Package *services*

rung und Verwaltung disjunkter *Domänenprozesse* an, die den Interaktionsradius von Rule–Objekten in einer offenen verteilten Umgebung bestimmen. Jeder Domänenprozess repräsentiert also ein zu jedem Zeitpunkt konzeptuell *abgeschlossenes* Rule–System (aber mit dynamisch erweiterbarem Umfang), innerhalb dessen jedes Rule–Objekt das Verhalten jeder Anwendung beeinflussen kann. Dies wird dadurch umgesetzt, dass alle Rule–sensitiven Anwendungen sich an einem Domänenprozess anmelden müssen (mittels der Methode *register()*), und auch wieder abmelden (*deregister()*), wenn sie an der entsprechenden Rule–sensitiven Kommunikation nicht mehr teilnehmen wollen. Auf diese Weise kann das RS–DII auch ermitteln, auf wie viele Antwort–Events es von Rule–Objekten zu warten hat, da es über den Domänendienst die Anzahl aller aktuell registrierten Rule–sensitiven Anwendungen (und damit die Anzahl aller beteiligten Rule–Objekte) herausfinden kann. Deshalb kann sich eine Rule–sensitive Anwendung auch nur dann an- oder abmelden, wenn gerade keine Rule–Interaktionen in der Domäne stattfinden. Um dies festzustellen, meldet das RS–DII seinerseits jeden Rule–sensitiven Aufruf an der Domäne an (*rsdiiStart()*) bzw. wieder ab (*rsdiiEnd()*), wenn der Aufruf abgeschlossen ist.

4.3.1.8 Rule–sensitive Umgebung

Alle Schnittstellen und Klassen, die von einer Anwendung unmittelbar benötigt werden, um in einer Rule–sensitiven Umgebung agieren zu können, finden sich im Package *env* (*environment*).

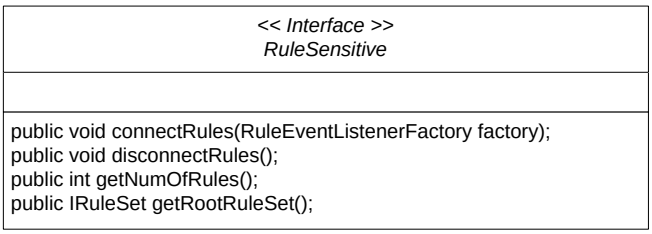


Abbildung 4.17: Interface *RuleSensitive*

Dazu gehören die Schnittstelle *RuleSensitive* (Abbildung 4.17) und ein hierarchischer Rule–Container, um Rule–Objekte zu beherbergen. Über das *RuleSensitive*–Interface können andere Rule–sensitive Komponenten — wie z.B. das RS–DII und Rule–Objekte — die Anwendung auffordern, ihre Rules an den

Event–Channel anzubinden oder abzutrennen (*connectRules()*, *disconnectRules()*), die Anzahl der aktuell beherbergten Rules abfragen oder auf die Funktionalität der Rule–Set–Hierarchie (Abbildung 4.18) zugreifen.

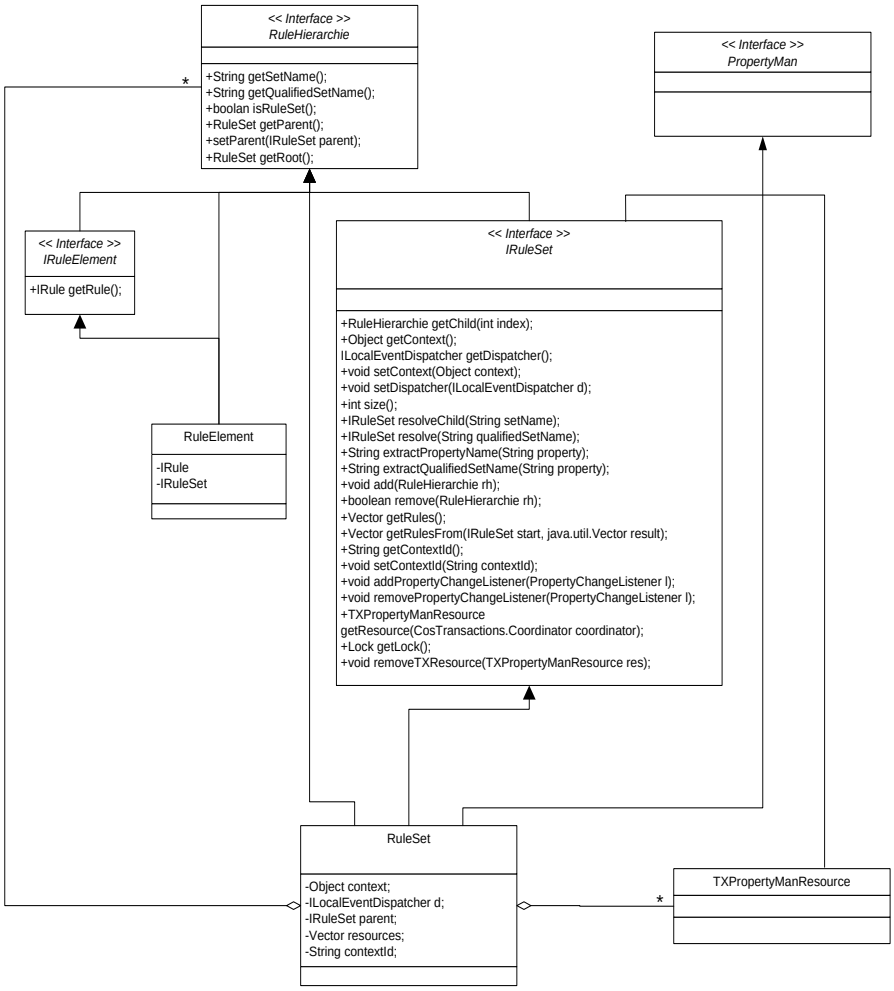


Abbildung 4.18: Hierarchie der *RuleSet*–Klassen

Die Rule–Sets sind nach dem Design–Pattern *Composite* [GHJV95, S. 163] implementiert, wobei die Klasse *RuleSet* das Composite und die Klasse *RuleElement*, die ein Rule–Objekt aggregriert, den Leaf darstellt. Beide implementieren das Interface *RuleHierachie*, das im wesentlichen aus Navigationsmethoden besteht. Außer diesem Interface implementiert jede konkrete (Nicht–Leaf-) Rule–Set noch das Interface *IRuleSet* und das Interface *PropertyMan* — welches von dem Interface *AccessAdaptor* der PM–Bibliothek (s. Abschnitt 4.2.1.7) erbt — wodurch sie ebenfalls als Property–Set für die Rule–sensitiven Properties der Anwendung fungiert (vgl. a. Abschnitt 3.3.3.2).

Jede Rule–Set in einer Hierarchie kann über einen eindeutigen Namen referenziert werden, wobei die Wurzel–Set "/root" heißen muss. Die Namen der in der Eltern/Kind–Relation stehenden Rule–Sets sind durch "/" von einan-

der getrennt. Dagegen sind die Namen der in der selben Relation stehenden Property–Sets durch ":" getrennt.

Die Root–Rule–Set besitzt eine Referenz auf die zugehörige Rule–sensitive Anwendung, die mit *setContext()* gesetzt wird. Darüber kann beispielsweise ein Rule–Objekt vom Typ *ActionRule* als Aktion eine Methode der es beherbergenden Anwendung aufrufen. Zusätzlich besitzt die Rule–Set auch eine (nicht notwendigerweise eindeutige) Kontext–ID in textueller Form, die mit *setContextId()* gesetzt wird. Diese ID stellt den aktuellen Anwendungskontext dar und dient beispielsweise zur Zustellung eines Rule–Events als Antwort auf ein triggerndes Event. Ebenfalls besitzt die Root–Rule–Set einen *LocalEvent-Dispatcher*, der mit *setDispatcher* gesetzt wird. Dieser Dispatcher übernimmt für alle Rule–Objekte einer Anwendung das Empfangen und Versenden von Interprozess–Events und verteilt sie lokal weiter. Dadurch kann der Overhead durch Interprozess–Events zwischen einzelnen Rule–Objekten vermieden werden und die kostspielige Kommunikation auf Interprozess–Ebene um ein n–faches reduziert werden.

4.3.2 Agenten–basierter Marktplatz

Im folgenden wird die Implementierung eines generischen agentenbasierten Marktplatzes als Basis zur Realisierung Rule–sensitiver, agentenbasierter Handelsszenarien (die in den Abschnitten 4.4.4 und 6.4 beschrieben sind) erläutert. Die zugehörigen Klassen sind in zwei Gruppen, Marktteilnehmer und Marktplatz, gegliedert.

4.3.2.1 Agenten

Marktteilnehmer sind in diesem Fall *mobile Agenten* (siehe Einführung in Abschnitt 1.3.4), die um die Eigenschaft *Rule–sensitive* erweitert worden sind. Der abstrakteste Typ eines solchen Agenten ist durch die Klasse *AbstractRSAgent* implementiert, die mit ihren Schnittstellen und zugehörigen Klassen in Abbildung 4.19 dargestellt ist.

Die Mobilität der Agenten basiert dabei auf dem Produkt Voyager [Obj], deshalb erbt *AbstractRSAgent* von der abstrakten Klasse *Agent* von Voyager und implementiert außerdem das Interface *RuleSensitive* (Abschnitt 4.3.1.8), um in einer Rule–sensitiven Umgebung agieren zu können. Dies erfolgt durch Aggregation der Klasse *RuleSensitiveImpl*, die ihrerseits von der Klasse *RuleSet* aus dem Package *env* erbt, womit sie Rule–Objekte aufnehmen kann. Damit enthält der Rule–sensitive Agent alle Basiseigenschaften eines Voyager–Agenten und einer Rule–sensitiven Anwendung, trägt jedoch noch keine anwendungsspezifische Funktionalität. Diese wird durch konkrete Subklassen bereitgestellt, die spezifische Marktteilnehmertypen wie z.B. *Käufer*, *Verkäufer*, *Verhandlungsagenten*, *Workflowagenten* etc. repräsentieren, auf die später in den Anwendungsszenarien näher eingegangen wird.

4.3.2.2 Marktplatz

Der Marktplatz als eine spezifische Rule–sensitive Umgebung wird durch das Package *market*, das in Abbildung 4.20 gezeigt ist, implementiert.

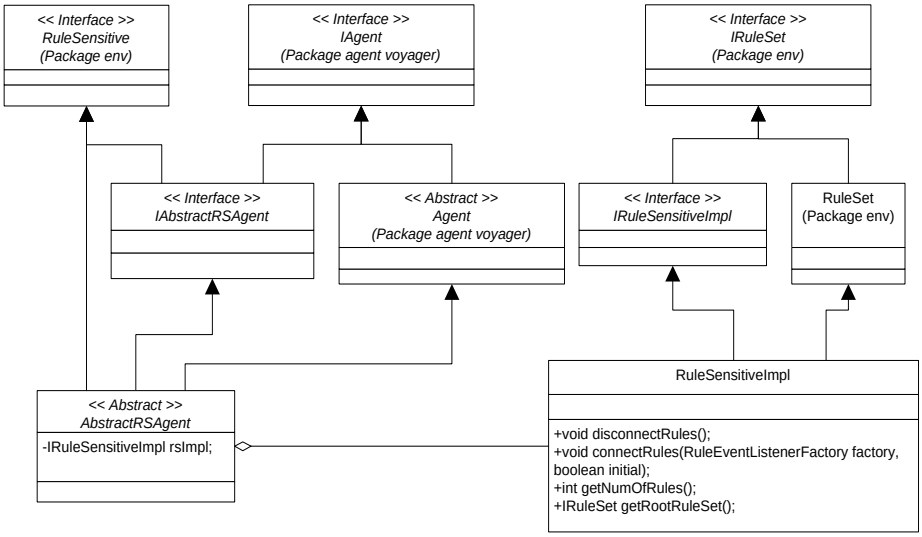


Abbildung 4.19: Package *agents*

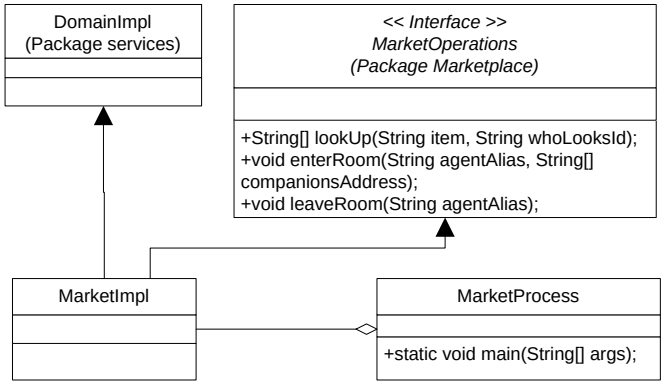


Abbildung 4.20: Package *market*

Dessen Hauptkomponente, die Klasse *MarketImpl* ist eine spezifische Ausprägung des Domänenprozesses (s. Abschnitt 4.3.1.7) und erbt deshalb von der Klasse *DomainImpl*. Der Marktplatz wird als CORBA–Objekt (s. IDL–Spezifikation in Anhang B.3) nach dem Delegationsansatz implementiert. Ein Marktteilnehmer, der bei ihm Handelstransaktionen Rule–sensitiv durchführen will, meldet sich mit der Methode *enterRoom()* an. Nach Beendigung des Geschäfts muss er sich entsprechend mit der Methode *leaveRoom()* wieder abmelden.

Zum Auffinden von aktuell auf dem Markt vorhandenen Geschäftspartnern nach den Waren, die sie anbieten, kann die Methode *lookUp()* verwendet werden. Dabei stellt der Marktplatz sicher, dass ein Marktteilnehmer zur Zeit nur an einer Transaktion innerhalb des Marktplatzprozesses teilnehmen kann und liefert als Ergebnis der *lookUp()*–Methode nur die jeweils freien Teilnehmer zurück. Ein solcher Marktplatzprozess wird durch die Klasse *MarketProcess* gestartet,

die anstelle der Klasse *DomainProcess* aus Package *services* aufzurufen ist.

4.4 Realisierung von Anwendungsszenarien

Im folgenden werden einige exemplarische Anwendungen vorgestellt, um zu zeigen, wie das Verhalten verteilter Anwendungen aus unterschiedlichen Domänen mittels regelbasierter Steuerungsmechanismen unterstützt werden kann. Diese Anwendungen zielen zwar nicht darauf, die gesamte Bandbreite der Einsatzmöglichkeiten abzudecken, sollten aber die Hauptmerkmale der präsentierten Regelarchitekturen in praktischen Szenarien demonstrieren. Dazu gehören graphische Administrationswerkzeuge (Abschnitt 4.4.1), Anwendungen, die den Policy–Manager entweder als CORBA–Dienst (Abschnitt 4.4.2) oder als Bibliothek (Abschnitt 4.4.3) benutzen, Beispiele für Rule–sensitive mobile Agenten (Abschnitt 4.4.4) und Beispiele für die erweiterte Ausdrucksmächtigkeit von Steuerungsregeln (Abschnitt 4.4.5).

4.4.1 Graphische Werkzeuge

Da in den meisten Szenarien die Regeln zur Steuerung von Anwendungen nicht in diese selbst einprogrammiert sind, sondern i.a. von einem Endbenutzer eingegeben werden sollen, wurden eine Reihe von graphischen Werkzeugen zur Spezifikation, Verwaltung und Erprobung von Regeln entwickelt.

Dazu gehören vor allem unterschiedliche *Rule–Editoren*, mit denen ein Benutzer jeweils auf *zentral* beim Policy–Manager oder *dezentral* bei mobilen Agenten abgelegte Rules und Properties zugreifen kann. Bezüglich des Policy–Managers gibt es darüber hinaus unterschiedliche Editoren, mit denen der Benutzer entweder direkt auf die PM–Bibliothek oder auf den entsprechenden CORBA–Server zugreifen kann.

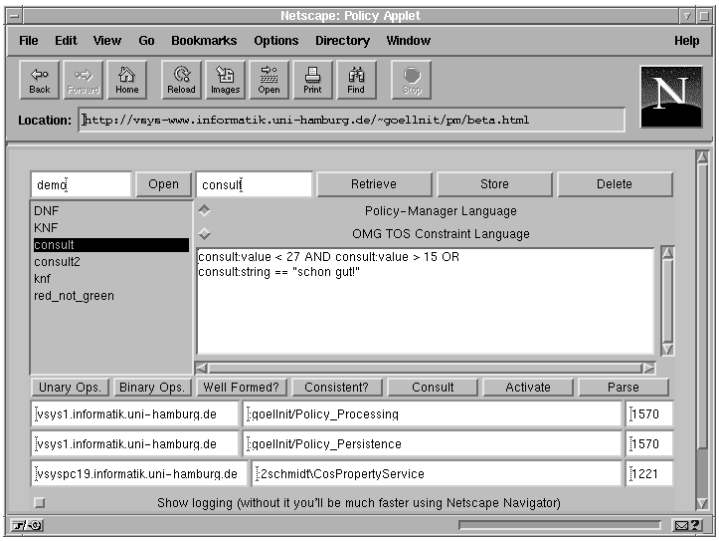


Abbildung 4.21: GUI des CORBA–Klienten für den Policy–Manager

Abbildung 4.21 zeigt z.B. die graphische Benutzeroberfläche (GUI) des Rule–Editors, der als ein Klient des CORBA–Policy–Managers implementiert ist. Dieses Werkzeug dient zunächst einmal zur Demonstration, wie ein solcher CORBA–basierter Policy–Manager–Klient, der auf dem mächtigen IIOP–Kommunikationsprotokoll basiert und deshalb prinzipiell sprach- und betriebssystemunabhängig ist, funktioniert. In diesem Fall handelt es sich um ein *Applet*, das in jedem Java–fähigen Webbrowser benutzt werden kann.

In dieser Sicht zeigt der Editor einen Bedingungsausdruck, der mittels einer *unären* Funktion des Policy–Manager verarbeitet werden soll. Wenn hier z.B. auf die Schaltfläche "Consult" geklickt wird, dann wird dieser Ausdruck anhand der betreffenden, aktuellen Werte im Property–Service zu *TRUE* oder *FALSE* evaluiert. Dabei kann der aufzurufende Property–Service, wie auch die beiden anderen Policy–Dienste (vgl. Abschnitt 4.2.2), mittels des untersten Textfeldes dynamisch spezifiziert werden.

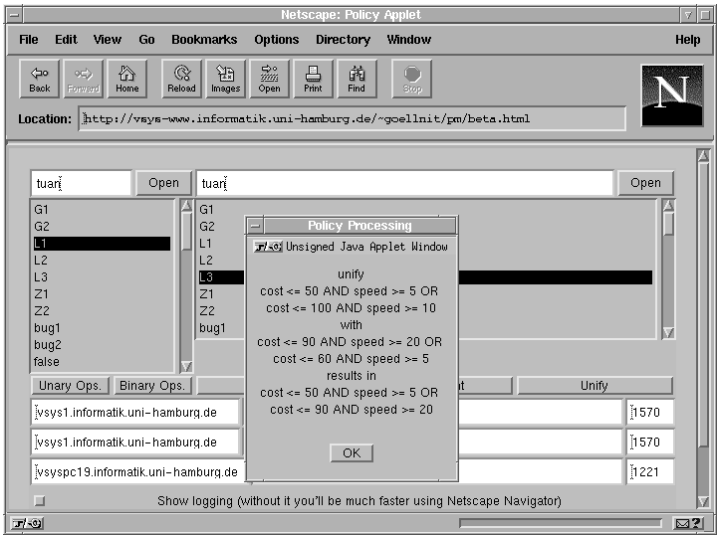


Abbildung 4.22: Binäre Funktionen im CORBA–Klienten des Policy–Managers

Abbildung 4.22 zeigt die zweite Sicht des CORBA–Klienten, der die *binären* Funktionen der PM–Bibliothek dem Benutzer anbietet. Hier handelt es sich um einen Aufruf der Unifikationsfunktion (*unify*), dessen Ergebnis im vorderen kleinen Fenster angezeigt wird. (Dies ist zugleich ein Beispiel für die Unifikation von zwei Bedingungen der Basisausdrucksmächtigkeit.)

4.4.2 Steuerung einer komplexen heterogenen Kooperationsanwendung

Im folgenden wird ein verteiltes Anwendungsszenario beschrieben, das mittels des Policy–Managers gesteuert wird. Dieses Szenario wurde entwickelt, um Kooperationsmechanismen von Anwendungskomponenten, die in mehreren *heterogenen* Systemumgebungen — in diesem Fall CORBA und DCE [Sch93a] — verteilt sind, zu demonstrieren (s. Details in [TGML97b]).

Es handelt sich dabei um ein CORBA–seitig realisiertes Videokonferenzsystem, mit dessen Hilfe verschiedene Personen gemeinsam oder auch zu Gruppen zusammengefasst per Bild und Ton kommunizieren können. Das System soll Funktionen bereitstellen, um neue Konferenzteilnehmer zu suchen und einzuladen, wobei im Beispiel ein thematisiertes Gespräch zur Lösung eines Problems vorausgesetzt wird, zu dem Personen mit bestimmtem Fachwissen (oder Kompetenzen etc.) Stellung nehmen sollen.

Da keine Annahmen über die Aufenthaltsorte der Konferenzteilnehmer festgelegt werden, sieht das System eine Repräsentation der potentiell Beteiligten in Form verteilt vorliegender "Profile" vor, die die jeweilige Person charakterisieren und relevante Personendaten — wie etwa Namen, Anschrift, Position, Lebenslauf, Spezialgebiet, Referenzen — sowie einen Hinweis auf ihren aktuellen "Kontaktpunkt" (Zugangspunkt zum Videokonferenzsystem) enthalten. Die einzelnen Profile sind dabei als DCE–Dienste realisiert, so dass in einer verteilten DCE–Domäne (beispielsweise in einem Unternehmensnetzwerk) jeder Interessent sich durch Bereitstellung seines persönlichen Profils registrieren kann — entweder selbständig durch eigene Erzeugung eines solchen DCE–Dienstes oder durch eine zentrale Administration.

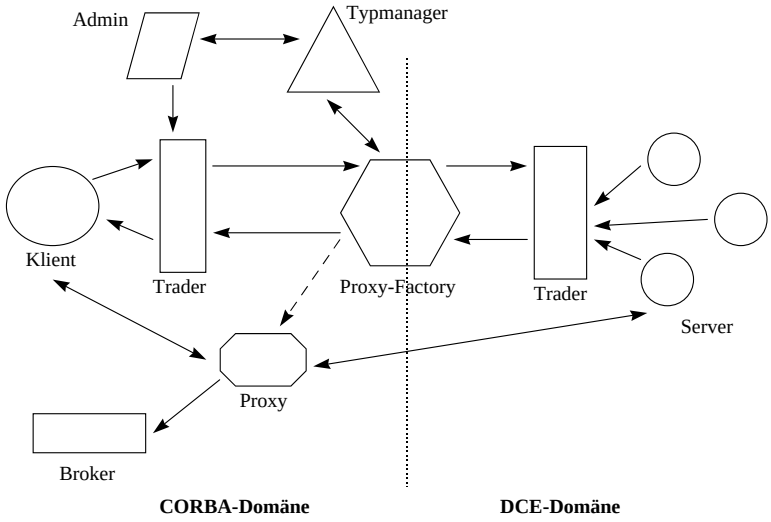


Abbildung 4.23: Übersicht der zu steuernden Komponenten des Anwendungsszenarios

Charakteristisch für dieses Szenario, deren Komponenten in Abbildung 4.23 illustriert sind, ist die Gesamtkomplexität der entsprechenden Unterstützungsmechanismen, die zur Überwindung der Heterogenität der Kooperationsumgebung benötigt werden. Diese Komplexität zeigt sich vor allem in den folgenden Eigenschaften des Anwendungsszenarios:

- Anzahl der Komponenten: Eine Vielzahl von Systemkomponenten (CORBA–Trader, DCE–Trader, Typmanager, Proxy–Factory, Proxies, etc.) und

Anwendungskomponenten (CORBA–Klienten und DCE–Server) müssen auf eine bestimmte Weise zusammenspielen, um die gewünschte Kooperation der Anwendungen zu ermöglichen.

- Funktionalität einzelner Komponenten: Jede einzelne Systemkomponente kann eine komplexe Funktionalität besitzen, die viele unterschiedliche Verwendungsoptionen besitzt. Der DCE–Trader [MJ97] beispielsweise kann über das sog. *Interworking* so eingestellt werden, dass er bei der Suche nach passenden Dienstangeboten weitere DCE–Trader in Anspruch nimmt.

- Dynamische Umgebung: Die Anzahl der Klienten und Server (und damit auch die Anzahl der zur Kommunikation benötigten Proxies) ist nicht fix, sondern unterliegt dem jeweiligen Anwendungsbedarf, der sehr unterschiedlich sein kann. Das Systemverhalten muss sich daher dem Bedarf anpassen können, um gewisse Dienstqualitätskriterien zu erfüllen.

Die Steuerung des Gesamtsystems erfolgt über Regeln vom Typ *Policy–Rule*, die im (CORBA–)Policy–Manager verwaltet werden. Die Administrationswerkzeuge und damit das gesamte System stellen so eine Anwendung dar, die explizit den Policy–Manager benutzt und über ihn gesteuert wird. Insbesondere werden die folgenden dynamischen Aspekte mittels Policies gesteuert:

- Proxy–Erzeugung: Das Verhalten der *Proxy–Factory*, wie zum Beispiel Zeitpunkte des Erzeugens von *Proxies* (zur Kommunikation zwischen den heterogenen Systemumgebungen) oder die Dauer, für die gerade nicht benötigte *Proxies* vorgehalten werden, wird über Policies gesteuert, die über das Administrationswerkzeug (den Rule–Editor) eingegeben und im persistenten Speicher des Policy–Manager abgelegt werden. Die jeweils gewünschte Policy wird dann referenziert und aktiviert, womit das gewünschte Verhalten zum Beispiel der *Proxy–Factory* erreicht wird.

- Trader–Verhalten: Individuelle Voraussetzungen und Eigenschaften einer Person werden als Diensteigenschaften beim Property–Service repräsentiert, so dass eine Trading–basierte Suche nach gewünschten Konferenzteilnehmern durchgeführt werden kann. Durch Vorgabe bestimmter Richtlinien zur Steuerung des Trader–Verhaltens (die als *Import–Policies* bezeichnet werden) beim Policy–Manager bzgl. der Suchverfahren erlauben die Kontrollobjekte des CORBA–Konferenzsystems die gezielte Auswahl von Konferenzteilnehmern. Beispielsweise kann eine solche Richtlinie die Anzahl der für eine Konferenz gewünschten Teilnehmer eines bestimmten Typs wie *Kunden* nach unten beschränken.

4.4.3 Steuerung der Selektion von Unterstützungsdiensten eines elektronischen Marktsystems

In [Mer99b] wird ein elektronisches Marktsystem beschrieben, das eine dynamische Anbindung von Unterstützungsdiensten erlaubt. Dabei wird ein elektronischer Dienstemarkt (hier im engeren Sinne) als eine Infrastruktur charakterisiert, die die freie Entstehung und Evolution von Angebot und Nachfrage bezüglich *elektronisch erbringbarer* Dienstleistungen ermöglicht. Die Inanspruchnahme einer solchen Dienstleistung erfolgt durch eine *Markttransaktion*,

die mindestens zwei Parteien, einen Anbieter und einen Kunden, involviert und beinhaltet die (elektronische) Lieferung des Dienstes gegen eine entsprechende (ebenfalls elektronische) Bezahlung. Dienstanbieter, die direkt in die Transaktion involviert sind, werden dabei von den Diensten unterschieden, die die Durchführung der Transaktion unterstützen — wie z.B. eine Online–Bank oder ein elektronischer Notar — und deshalb als *Unterstützungsdienste* bezeichnet werden.

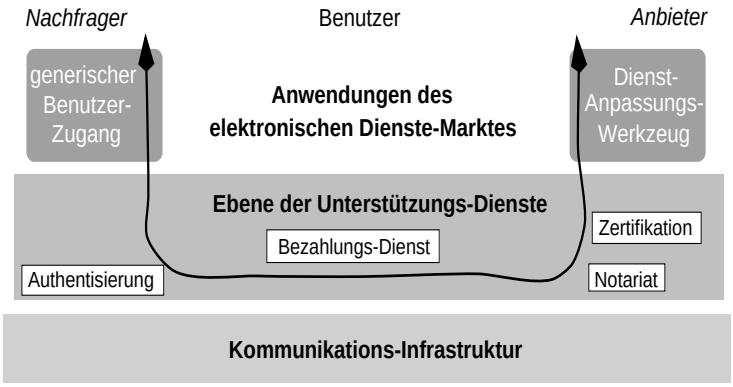


Abbildung 4.24: Dienst–orientiertes elektronisches Marktsystem

Abbildung 4.24 zeigt die allgemeine Struktur dieses Marktsystems, das eine explizite Schicht zur Anbindung von Unterstützungsdiensten enthält, die zwischen der Anwendungs- und der Kommunikationsebene angesiedelt ist. Da unterschiedliche Transaktionstypen unterschiedliche Unterstützungsdienste erfordern, und vor allem auch unterschiedliche Unterstützungsdienste vom selben Typ (z.B. verschiedene Banken) miteinander konkurrieren sollen, können diese Dienste nicht statisch in das Marktsystem integriert sein, sondern müssen für die jeweilige Transaktion dynamisch eingebunden werden. Darüber hinaus sollen die an der Transaktion beteiligten Parteien (entweder als Person oder durch die entsprechende Anwendung repräsentiert) die Möglichkeit haben, die jeweils gewünschten Unterstützungsdienste selbst zu spezifizieren bzw. zu selektieren.

Um eine solche Flexibilität zu gewährleisten, sind jedoch Steuerungsmechanismen erforderlich, da die Marktinfrastruktur die von den beteiligten Parteien spezifizierten Anforderungen erst *in Einklang* bringen müssen, um dann die entsprechenden Dienste hinzuzuziehen. Hierfür bietet sich offenbar die Funktionalität des Policy–Managers bzw. der diesem zu Grunde liegenden PM–Bibliothek an. Allerdings stellten sich bei der genauen Untersuchung der Spezifikation und Selektion von Unterstützungsdiensten spezifische Anforderungen heraus, die nur von einer zusätzlichen Funktionsschicht zu erfüllen sind. Deshalb wurde ein so genannter *Profile–Matcher* entwickelt, der die generische Funktionalität der PM–Bibliothek transparent für die Marktteilnehmer in das Marktsystem integriert [MTL96].

Abbildung 4.25 illustriert das Zusammenspiel des Profile–Matchers mit den zugehörigen Komponenten. Anforderungen der Marktteilnehmer, insbesondere in Bezug auf Unterstützungsdienste, werden in so genannten *User Profile* (Kundenprofil) und *Server Profile* (Dienstanbieterprofil) abgelegt, die über einen *Profile Editor* bearbeitet und extrahiert werden können. Der *Invocation*

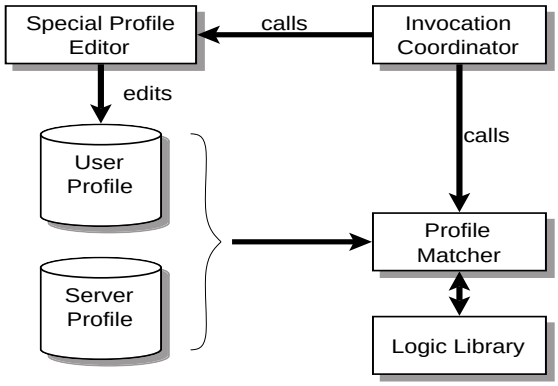


Abbildung 4.25: Interaktionen der Dienstaktivierungskomponenten

Coordinator, der für die Durchführung einer Markttransaktion zuständig ist, liest dabei zunächst die Profile der beteiligten Parteien ein und übergibt diese an den Profile Matcher, der daraus die einzubindenden Unterstützungsdienste — mit Hilfe der PM–Bibliothek — bestimmen und deren Referenzen zurückliefern muss. Unterstützungsdienste werden nach einem hierarchischen Schema auf der obersten Ebene in *Kategorien* eingeteilt. Innerhalb einer Kategorie (z.B. *Payment* oder *Notary*) werden Dienste nach dem jeweils zu verwendenden *Protokoll* (z.B. *Ecash*) klassifiziert. Innerhalb eines Protokolls werden dann *Instanzen* unterschieden. Anforderungen von Marktteilnehmern können nach diesem Klassifikationsschema also wie folgt aussehen:

Anbieter: $Payment = TRUE \wedge (Protocol = Ecash \wedge Instance = DB) \vee (Protocol = SET \wedge Instance = DB)$

Kunde: $Payment = TRUE \wedge Protocol = SET \wedge Instance = DB$

Es lassen sich so über beliebige Dienste Aussagen formulieren, es bleibt aber ungeklärt, was für eine Semantik gemeint ist, wenn einer der beiden Kommunikationspartner über eine Komponente *keine* Aussage macht.

Neben Aussagen über bestimmte Dienste, die eingebunden werden sollen, kann dies bedeuten, dass ein Partner über weitere Dienste keine Aussagen machen möchte, oder dass diese *nicht* genutzt werden *sollen*. Mit der letzteren, vorsichtigen Annahme, dass unbekannte Dienste nicht benutzt werden dürfen, ließen sich solche Dienstanforderungen mit der in Abschnitt 2.3.1.4 eingeführten Regelsyntax bereits formulieren und dann unifizieren, um die *gemeinsamen* Anforderungen zu bestimmen.

Um für die Einbindung der Unterstützungsdienste *explizit* auszudrücken zu können, dass ein Dienst eingebunden werden *kann* oder *muss* beziehungsweise auf *keinen Fall* benutzt werden darf, muss die formale Repräsentation von Regeln um zusätzliche *modale* Operatoren erweitert werden, deren spezielle Semantik in Bezug auf die Einbindung von Unterstützungsdiensten folgendermaßen lautet.

Notwendig oder $\Box$: Die hiermit spezifizierten Unterstützungsdienste müssen auf jeden Fall an der Transaktion teilnehmen.

Möglich oder $\Diamond$: Der Unterstützungsdienst kann benutzt werden, wenn es der Partner wünscht.

Darf nicht oder $\Box\neg$ ("Notwendig" mit Negation): Der betreffende Unterstützungsdienst darf auf keinen Fall für diese Handelstransaktion benutzt werden.

Für solche Formeln wird im wesentlichen eine Unifikation mit der gleichen Semantik wie in Abschnitt 2.3.2.3 benötigt, da die gemeinsamen Anforderungen der Beteiligten bestimmt werden müssen.

Um die Unifikationsfunktion nicht im allgemeinen auf die Modallogik [Che80] erweitern zu müssen, sollen Formeln mit Modaloperatoren in solche ohne Modi transformiert werden, um sie dann mit den bisherigen Funktionen verarbeiten zu können.

Der umgesetzte Transformationsalgorithmus erlaubt als Eingabe nur Sonderfälle der Modallogik. Erwartet werden Ausdrücke in folgender Form:

Anbieter: $\Box(P_1 \vee P_2 \vee P_3) \wedge \Diamond(K_1 \vee K_2) \wedge \Diamond(N_1 \vee N_2)$
Kunde: $\Diamond(P_1) \wedge \Diamond(K_3) \wedge \Box(N_2 \vee N_3)$

Wobei P_i, K_i und N_i jeweils Konjunktionen atomarer Ausdrücke sind, die als Tripel aus Kategorie–, Protokoll– und Instanz–Name genau eine Dienstinstanz spezifizieren, wie zum Beispiel:

$P_1 := (Payment = TRUE \wedge Protocol = ecash \wedge Instance = DB)$
$N_2 := (Notary = TRUE \wedge Protocol = INP \wedge Instance = N)$
$K_3 := (Security = TRUE \wedge Protocol = KMP \wedge Instance = RSA)$

Für den Fall, dass die Eingabeformeln nicht den Operator "darf nicht" ($\Box\neg$) enthalten, werden die Formeln mit folgendem Verfahren transformiert:

- Wird über eine Variable nur in einer Eingabeformel eine Aussage gemacht, schlägt das Verfahren fehl, da hier ein Dienst gefordert wird, der der anderen Partei unbekannt ist.
- Wenn bezüglich einer Variable mindestens ein Ausdruck mit dem Operator "notwendig" ($\Box$) auftritt, werden alle Ausdrücke bezüglich dieser Variable von den Zusatzoperatoren befreit. Auf diese Weise wird die Anforderung der einen Seite mit dem Zulassen des betreffenden Dienstes, so vorhanden, der anderen Seite verknüpft.
- Alle Ausdrücke mit Variablen, die in beiden Formeln nur mit dem Operator "möglich" versehen sind, werden gestrichen, da keine der Seiten die betreffenden Dienste *erfordert*, sondern nur zulässt. Ohne diesen Schritt würden auch alle Dienste als einzubinden gewertet, die von beiden Seiten nur zugelassen, also mit "möglich" gekennzeichnet, sind.

Für das obige Beispiel wären dann folgende Unifikationen zu durchzuführen:

1. unify$(P_1 \vee P_2 \vee P_3, P_1) = P_1$
2. unify$(N_1 \vee N_2, N_2 \vee N_3) = N_2$

Würden die Bezeichner *Protocol* und *Instance* nach Kategorien aufgeschlüsselt (zum Beispiel *Payment_Protocol*, *Notary_Instance*), kann die Unifikation jeweils auch in einem einzigen Schritt durchgeführt werden, ohne dass die Protokoll– und Instanz–Anforderungen verschiedener Kategorien zueinander in Widerspruch stünden:

$\text{unify}((P_1 \wedge N_1) \vee (P_1 \wedge N2) \vee (P_2 \wedge N_1) \vee (P_2 \wedge N_2) \vee (P_3 \wedge N_1) \vee (P_3 \wedge N_2),$
$(P_1 \wedge N_2) \vee (P_1 \wedge N_3)) = P_1 \wedge N_2$

Für den Fall, dass die Formeln den Operator "darf nicht" ($\Box\neg$) enthalten, muss eine abgeänderte Variante dieses Algorithmus eingesetzt werden. Das Beispiel könnte für diesen Fall wie folgt aussehen:

Anbieter: $\Box(P_1 \vee P_2 \vee P_3) \wedge \Diamond(K_1 \vee K_2) \wedge \Diamond(N_1 \vee N_2) \wedge \Box\neg(P_4 \vee P_5)$
Kunde: $\Diamond(P_1) \wedge \Diamond(K_3) \wedge \Box(N_2 \vee N_3)$

Nach DeMorgan ist $\Box\neg(P_4 \vee P_5) \equiv \Box(\neg P_4 \wedge \neg P_5)$. Somit werden der einfachen Unifikation folgende Formeln übergeben:

1. $\text{unify}((P_1 \wedge \neg P_4 \wedge \neg P_5) \vee (P_2 \wedge \neg P_4 \wedge \neg P_5) \vee (P_3 \wedge \neg P_4 \wedge \neg P_5),\ P_1) = P_1$
2. $\text{unify}(N_1 \vee N_2,\ N_2 \vee N_3) = N_2$

Dabei wurden alle umgeformten Ausdrücke mit Operator "darf nicht" in alle Disjunkte über dieselben Variable eingefügt. Da es sich aber bei P_i, K_i und N_i jetzt um negierte Konjunktionen und damit nach Umformung um Disjunktionen handelt, sind die Formeln so nicht mehr in disjunktiver Normalform und damit nicht mehr direkt unifizierbar[9]. Diesem Problem kann man damit begegnen, dass man statt der Tripel–Notation für die Spezifikation einer Instanz als *atomar* betrachtet und durch eine entsprechende *String*–Repräsentation ersetzt:

$$\begin{aligned} & P_1 := (Payment = TRUE \ \wedge \ Protocol = Ecash \ \wedge \ Instance = DB) \\ \rightsquigarrow \quad & P_1 := (Payment \ = "Ecash; DB") \end{aligned}$$

Durch diese Abstraktion ist die Implementation auch nicht mehr abhängig von der Tiefe der Hierarchie zur Beschreibung der Dienstinstanzen, solange es eine Ebene eindeutig benannter Kategorien gibt.

4.4.4 Steuerung elektronischer Handelsszenarien mit mobilen Agenten

Als nächstes werden Szenarien und Beispiele für Rule–sensitive mobile Agenten vorgestellt. Diese relativ einfachen Anwendungen wurden hauptsächlich zur Veranschaulichung der in Abschnitt 3.3 beschriebenen dezentralen, interaktionsorientierten Regelverarbeitungsarchitektur implementiert. Weitere, komplexere Anwendungsszenarien, die sich speziell mit der regelbasierten Steuerung von *automatisierten Verhandlungen* beschäftigen, werden später in Abschnitt 6.4 beschrieben.

[9] Aus dem gleichen Grund wurde die Anwendung der Negation auf atomare Bedingungen beschränkt (vgl. Abschnitt 2.3.1.4).

4.4.4.1 Mobile Agenten und ihre Steuerbarkeit

Obwohl die bisher unter dem Begriff "Rule–sensitive Anwendungen" bzw. "Rule–sensitive Middleware" behandelten Konzepte und Mechanismen zur dezentralen Regelverarbeitung so generisch konzipiert sind, dass sie sich im allgemeinen auf jede Art von verteilten Anwendungen anwenden lassen, können ihre wesentlichen konzeptionellen und implementationstechnischen Merkmale wie Interaktionsorientierung, Autonomie, Dynamik, Mobilität etc. wahrscheinlich am besten in Bezug auf *agentenbasierte* Anwendungen nachvollzogen werden.

Die Agentenorientierung hat sich ca. zur Mitte der neunziger Jahre als ein außerordentlich viel versprechendes, neues (bzw. wieder neu belebtes) Paradigma zur Entwicklung von Anwendungen, die Komplexität besser abstrahieren, leichter zu benutzen und i.a. intelligenter und flexibler sind als bis dahin existierende, angekündigt (s. a. Abschnitt 1.3.4). Damals entstanden eine ganze Reihe von Forschungsprojekten und auch Industrieunternehmen (allen voran Telescript bzw. MagicCap von General Magic) mit dem Ziel, die Agententechnologie für sehr praktische Anwendungsbereiche wie Internet–basierte Informationsverarbeitung, System–Management, und vor allem elektronischen Handel (E–Commerce) nutzbar zu machen (siehe [KJ98] für einen breiten Überblick). Die Vorstellung, Geschäftsanwendungen in Form von Softwareagenten zu realisieren, die *stellvertretend* für die beteiligten Personen und Unternehmen zumindest routinierte Handelstransaktionen *autonom* abwickeln und dabei sogar flexibel durch das globale Netz wandern können, schien derart sinnvoll und attraktiv zu sein, dass viele sie für die gängige Praxis der nahen Zukunft hielten. Rückblickend auf die Entwicklung der Agententechnologie der letzten Jahre lässt sich jedoch sagen, dass ein großer Teil der übermäßigen Erwartungen von einst noch weit von der Erfüllung entfernt sind und viele der Pioniere auf dem Gebiet der mobilen Agenten wie die Firma General Magic heute zwar immer noch agentenorientierte Anwendungen entwickeln, jedoch viel weniger allgemeine Ziele verfolgen.

Es gibt für diese gewissermaßen "schwieriger als erhoffte" Gesamtentwicklung des Agentenparadigmas viele Gründe, allerdings betreffen die meisten nicht oder weniger rein technische Eigenschaften von Softwareagenten (abgesehen von speziellen Sicherheitsproblemen) — denn Aspekte wie Zustandserhaltung, Mobilität, Persistenz und auch dynamische Funktionalität sind bereits gut gelöst — sondern vielmehr "höhere" Eigenschaften wie etwa Intelligenz, Autonomie, Interaktionsfähigkeit, die zwar (noch) nicht technisch–formal zu definieren sind, ohne die jedoch agentenorientierte Anwendungen keine wirklichen Vorteile gegenüber konventionellen aufweisen können. Ein Großteil der gegenwärtigen Forschungsbemühungen im Bereich Softwareagenten konzentriert sich deshalb auf die Realisierung bzw. Einbettung von intelligenten Fähigkeiten in Agentenanwendungen (s. a. [TGL99]). Beispielsweise sind Verhandlungsfähigkeiten eine der für Softwareagenten wünschenswerter Intelligenzformen, auf die in den folgenden Kapiteln 5 und 6 ausführlich eingegangen wird. An dieser Stelle soll zunächst eine kurze Erörterung der Eigenschaften Autonomie und Interaktionsfähigkeit folgen.

Obwohl in der Fachliteratur bis heute kein allgemeiner Konsens, geschweige formale Definition, darüber existiert, was unter "Softwareagent" genau zu verstehen ist, sind (fast) alle Autoren einhellig der Meinung, dass die beiden Eigenschaften Autonomie und Interaktionsfähigkeit für die Agentenorientierung cha-

rakteristisch sind. Autonomie bedeutet, dass der Agent seine Entscheidungen auf eine selbständige Art und Weise trifft und in allen (zumindest vorgesehenen) Situationen immer eine Entscheidung treffen kann. Insbesondere können an einen solchen Agenten gewisse Aufgaben komplett *delegiert* werden, die der Agent stellvertretend für seinen Auftraggeber erledigt. In Bezug auf mobile Agenten kann Autonomie entweder unter dem Aspekt der reinen Anwendungssemantik (*Entscheidungsautonomie*) oder unter dem der Migration (*Lokationsautonomie*) betrachtet werden. Interaktionsfähigkeit bedeutet für den Agenten zudem, dass die zur Lösung seiner Aufgabe benötigten Daten nicht von vorn herein bekannt sind — in welchem Fall Autonomie auch wenig interessant wäre, da die Aufgabe durch eine in sich abgeschlossene, algorithmische Berechnung gelöst werden kann (vgl. a. Abschnitt 1.2.2) — sondern dynamisch *während* seiner Ausführung durch Interaktion mir seiner Umgebung sammeln muss. Beispielsweise muss ein Einkaufsagent die ihm unbekannten Angebote, die während seiner "Einkaufstour" jederzeit neu eintreffen können, einsammeln und auswerten, um darauf basierend die endgültige Kaufentscheidung zu treffen.

Autonomie und Interaktionsfähigkeit sind also, in ihrer Kombination, häufig das wichtigste Argument zur Verwendung einer agentenbasierte Anwendung anstatt einer konventionellen ohne diese (ausgeprägten) Fähigkeiten. Allerdings können die selben Eigenschaften ebenso stark *gegen* die Verwendung von Agenten sprechen, denn es ist offensichtlich, dass solche Agenten das Risiko in sich bergen, "falsche" Entscheidungen zu treffen, d.h. solche, die der Auftraggeber nicht erwartet. Diesem semantischen Risiko könnte zwar dadurch begegnet werden, dass man die Entscheidungslogik der Agenten auf eine endliche Zahl von konkreten, d.h. vollständig spezifizierten, Situationen, auf die der Agent reagieren kann, beschränkt. Jedoch würde eine solche Vorgehensweise die Autonomie/Interaktionsfähigkeit sowie die Gesamtfunktionalität eines Agenten erheblich einschränken — insbesondere in offenen Umgebungen, in denen potenzielle Kooperations- oder Geschäftspartner im allgemeinen unbekannt sind — und damit letztendlich das genannte Für–Argument ad absurdum führen. In anderen Worten wird also das Risiko unerwünschter bzw. unvorhergesehener Entscheidungen um so größer, je größer die zu den Agenten delegierte Autonomie ist. Deshalb bietet sich das Konzept Rule–sensitiver Anwendungen als ein Mechanismus zur Steuerung des Anwendungsverhaltens und damit zur Reduzierung des semantischen Risikos auch und gerade für Endanwender von agentenbasierten Anwendungen an.

4.4.4.2 Beispielanwendungen

In folgenden Beispielen soll nun die Steuerung von agentenbasierten Anwendungen im Umfeld des E–Commerce mittels der dezentralen Regelmechanismen demonstriert werden. Alle beteiligten Agenten sind dabei als Rule–sensitive mobile Agenten, die auf dem Package *agents* in Abschnitt 4.3.2.1 basieren, implementiert. Zu ihrer Steuerung wurde ein generisches Werkzeug entwickelt, mit dem ein Benutzer jederzeit und von jedem Ort aus seinen (oder einen beliebigen) wandernden Agenten kontaktieren kann, um die bei dem Agenten aufbewahrten Regeln (entfernt) zu verwalten. Dazu gehören Funktionen, um bestehende Rules und Properties des Agenten anzuzeigen, sie zu editieren oder löschen bzw. um Rules zu vervielfältigen ("klonen") und zu anderen Agenten zu schicken etc. Die graphische Oberfläche dieses Werkzeugs ist in Abbildung 4.26 zu sehen.

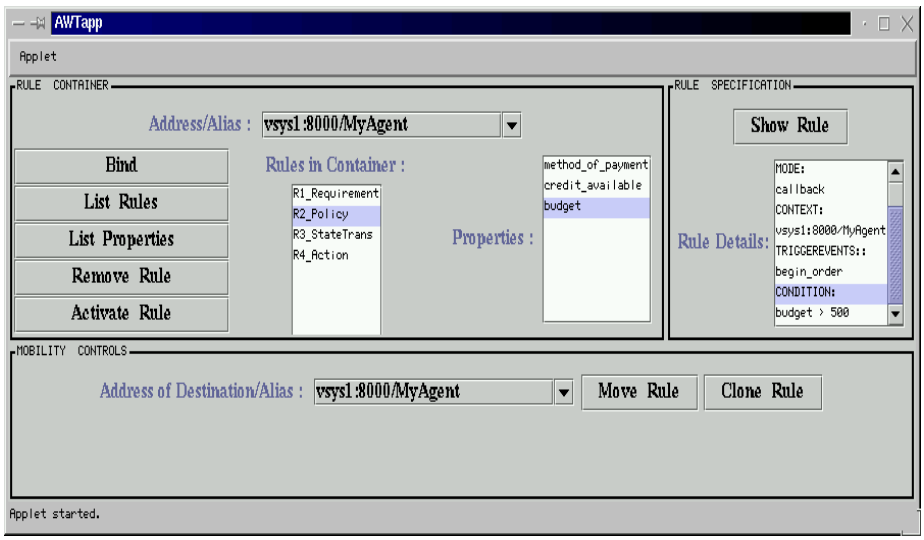


Abbildung 4.26: Verwaltungswerkzeug für Rule–sensitive mobile Agenten

Käufer/Verkäufer–Beispiel

Im ersten Beispiel handelt es sich um das "Standard–Szenario" im konventionellen E–Commerce–Umfeld, in dem ein Käufer als *Klient* gegenüber einem Verkäufer als *Server* auftritt, um eine bestimmte Ware zu erwerben. Die durchzuführende Handelstransaktion besteht hierbei aus einer Informations-, einer Bestellungs- und einer Bezahlungsphase, die jeweils durch eine entsprechende Methode auf Server–Seite modelliert und umgesetzt wird. In diesem Szenario müssen diese drei Methoden mittels des RS–DII Rule–sensitiv aufgerufen werden, wodurch entsprechende Rules entweder vor oder nach einem solchen Aufruf getriggert werden können.

Typ/Modus:	StateTransition/Internal
Trigger:	nach *order()*
Bedingung:	**budget** $\geq$ **0**
Vorbedingung:	state = "queried"
Nachbedingung:	state = "ordered"

Tabelle 4.3: Transitionsregel bei der Bestellung

Beispielsweise kann für den Käufer eine Regel vom Typ *Requirement* eingeführt werden, um sicherzustellen, dass sein aktuelles Budget durch (bzw. nach) eine(r) Warenbestellung niemals in den Minusbereich rutschen kann. Oder es kann (in bestimmten Fällen) sinnvoll sein, dass eine bestimmte Reihenfolge bezüglich der drei genannten Phasen eingehalten wird, z.B. "keine Bezahlung ohne zuvorige Bestellung". Eine solche Anforderung, die in konventionellen Anwendungen meisten durch einen statischen Flusskontrollmechanismus behandelt wird, kann hier nun durch eine simple Regel vom Typ *StateTransition* dynamisch erfüllt werden. Tabelle 4.3 zeigt die Spezifikation dieser Regel, wobei der Bedin-

gungsteil genau die Semantik der erwähnten Budgetregel ausdrückt, so dass diese nicht als zusätzliche Requirement–Rule spezifiziert werden muss.

Typ/Modus:	Requirement/Oneway
Trigger:	vor/nach *order()*
Bedingung:	**payment = "Ecash" OR payment = "NetBill"**
Filter:	**payment = "SET" OR payment = "Ecash"**
Unifikation:	**payment = "Ecash"**

Tabelle 4.4: Regel bzgl. des Zahlungsverfahrens

Die nächste Regel für dieses Szenario betrifft das Zahlungsverfahren. Im allgemeinen scheint es nicht sinnvoll zu sein, eine Ware zu bestellen, ohne die technische Möglichkeit zu besitzen, sie danach auch zu bezahlen. In elektronischen Marktsystemen wird in den meisten Fällen gefordert, dass dasselbe Zahlungsverfahren auf beiden Seiten eingesetzt wird (vgl. a. das in Abschnitt 4.4.3 beschriebene Marktsystem). Um die Konformität der Zahlungsverfahren bereits bei der Bestellung der Ware zu überprüfen, kann die in Tabelle 4.4 spezifizierte Regel verwendet werden. Es handelt sich also um eine eine Instanz des Regeltyps *Requirement* im *Oneway*–Modus, die bei der Aktivierung mit der entsprechenden Filter–Regel (deren Bedingung in der vorletzten Zeile der Tabelle angegeben ist) auf Server–Seite unifiziert wird. Der Aufruf der *order()*–Methode wird auf Grund dieser Regel vom RS–DII nur dann durchgeführt, wenn das Ergebnis der Unifikation (in der letzten Zeile) wie in diesem Fall nicht leer ist.

Typ/Modus:	Policy/Callback
Trigger:	vor *pay()*
Bedingung:	**amount $\leq$ 100 AND discount $>$ 0.04**
Filter:	**amount = 100 AND discount $\leq$ 0.08**
Unifikation:	**amount = 100 AND 0.04 $\leq$ discount $\leq$ 0.08**
Durchsetzung:	amount = 100; discount = 0.05

Tabelle 4.5: Regel zur Festlegung des Rabattes

Eine weitere Regel wird eingesetzt, um die Höhe des Rabatts zu regeln. Im Gegensatz zu dem in Deutschland noch geltenden strikten Rabattgesetz ist es im globalen Handel nicht ungewöhnlich, beim Kauf einer größeren Warenmenge einen beliebig hohen Rabatt zu gewähren. Deshalb müssen sich Käufer und Verkäufer in der Regel bei jeder Transaktion auf die Höhe des Rabattes, der prozentual vom Listenpreis des Artikels abzuziehen ist, verständigen. Zu diesem Zweck kann die in Tabelle 4.5 spezifizierte Regel vom Typ *Policy* im *Callback*–Modus verwendet werden. In diesem Fall verlangt der Käufer einen Rabatt von mindestens 4% und da der Verkäufer bereit ist, einen höheren Rabatt zu gewähren, gelingt die Unifikation der betreffenden Regeln. Allerdings besagt das Ergebnis der Unifikation lediglich, dass die Höhe des Rabatts zwischen 4 und 8% liegt, und deshalb muss der konkrete Wert, der durch die Durchsetzung der Policy auf Server–Seite eingestellt wird (in der letzten Zeile der Tabelle), auch dem Klienten mitgeteilt werden, so dass beide bzgl. des Rabattes exakt gleich

konfiguriert sind. Dies ist genau der Grund dafür, dass der *Callback*–Modus für diese Regel verwendet wird.

Einkaufsagenten–Beispiel

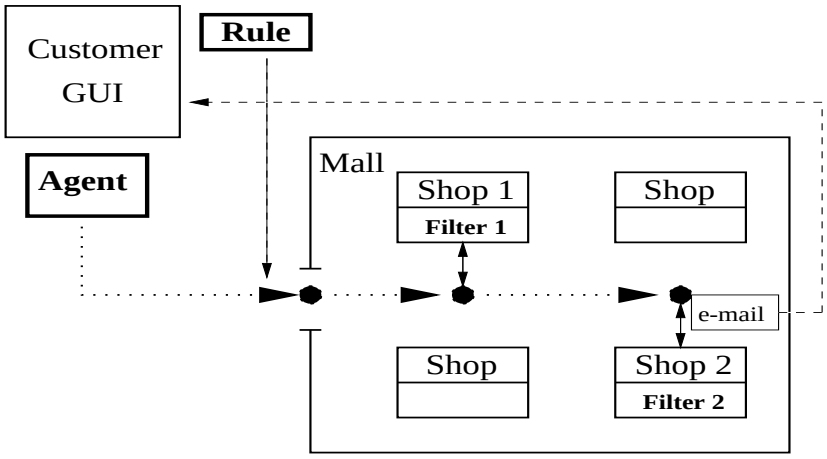


Abbildung 4.27: Einkaufsagenten–Beispiel

In diesem Beispiel handelt es sich um ein Anwendungsszenario, das häufig von Softwareunternehmen in ähnlicher Form zur Demonstration von mobilen Agenten angeboten wird. Dabei instanziiert der Anwender (bzw. Kunde) zunächst einen Agenten mit dem Auftrag, den günstigsten Preis für einen bestimmten Artikel zu ermitteln bzw. zu bestellen. Der Agent wandert zunächst mit dem Auftrag zu einem elektronischen Einkaufszentrum, erhält dort eine Liste der aktuell vorhandenen Geschäfte, die den gewünschten Artikel führen, wandert dann weiter zu den einzelnen Geschäften, führt jeweils eine Preisanfrage aus, sammelt die Ergebnisse in einer sortierten Reihenfolge ein und teilt dann dem Auftraggeber das günstigste Angebot mit oder bestellt die entsprechende Ware.

Abbildung 4.27 illustriert, wie das entsprechende, um Rules erweiterte Szenario abläuft. Zunächst werden alle Komponenten wie Geschäfte und Einkaufsagenten durch ihre Rule–sensitiven Varianten ausgetauscht, wobei die Geschäfte mit entsprechenden Requirement–Rules im Filter–Modus zur Regelung der Zahlungsmodalitäten vorkonfiguriert werden. Nachdem der Agent losgeschickt wurde und z.B. beim Einkaufszentrum (*Mall*) angekommen ist, kann der Auftraggeber mittels des graphischen Verwaltungswerkzeugs (s. Abbildung 4.26) nun nachträglich eine Regel bzgl. des Zahlungsverfahrens spezifizieren und dem Agenten als mobiles Rule–Objekt zuschicken. Diese Rule wird in den Rule–Container des Agenten aufgenommen und kann dann sofort auf entsprechende Trigger–Events reagieren. Diese werden durch das RS–DII immer dann ausgelöst, wenn die *query()*–Methode bei einem Geschäft aufgerufen wird.

Tabelle 4.6 zeigt eine vereinfachte Rule–Konfiguration für den Agenten und zwei Geschäfte. In diesem Fall scheitert die Unifikation der externen Regel des Agenten mit der Filter–Regel des *Shop 2*, was bedeutet, dass die Zahlungsverfahren der beiden miteinander unvereinbar sind. Deshalb scheidet dieses Geschäft

Typ/Modus:	Requirement/Oneway
Trigger:	vor *query()*
Bedingung:	**payment = "Ecash"**
Filter 1:	**payment = "SET" OR payment = "Ecash"**
Filter 2:	**payment = "SET" OR payment = "NetBill"**
Unifikation 1:	**payment = "Ecash"**
Unifikation 2:	**FALSE**

Tabelle 4.6: Zahlungsregeln für das Einkaufsagenten–Szenario

als möglicher Anbieter der Ware aus und *Shop 1* wird als einziger Anbieter ermittelt. Bei einem Durchlauf ohne die nachgeschickte Rule des Agenten wird allerdings *Shop 2* als günstigster Anbieter ermittelt, da es einen besseren Preis für den betreffenden Artikel bietet.

Zusätzlich wird der Agent in diesem Szenario noch mit einer Regel vom Typ *ActionRule* ausgestattet, die bei ihrer Aktivierung prüft, ob das dem Agenten zugeteilte Budget unter eine bestimmte Grenze unterschritten hat. Ist dies der Fall, sorgt die in der Regel spezifizierte Aktion dafür, dass eine E–Mail zu der Adresse des Auftragebers verschickt wird, um diesen über den aktuellen Stand zu benachrichtigen.

4.4.5 Beispiele der erweiterten Ausdrucksmächtigkeit

In diesem Abschnitt sollen einige anwendungsnahe Beispiele für den Einsatz der Policy–Manager–Bibliothek in der erweiterten Ausdrucksmächtigkeit, die Bedingungsausdrücke zulassen, deren atomaren Bestandteile jeweils eine lineare Kombination beliebig vieler Variablen bzw. Properties sind, gegeben werden.

Video–On–Demand

Im ersten Beispiel handelt es sich um einen Video–On–Demand Anbieter. Er hat ausgerechnet, dass sich das Geschäft für ihn nur dann lohnt, wenn der Preis mindestens doppelt so hoch ist wie die Übertragungsgeschwindigkeit[10]. Seine Policy gegenüber den Kunden lautet also:

$$\text{cost} \geq 2 * \text{speed}$$

Da diese Policy gemäß der Syntax des Policy–Managers nicht wohl geformt ist, muss sie noch in die folgende Form gebracht werden.

Anbieter: $\text{cost} - 2 * \text{speed} \geq 0$

Ein Kunde möchte maximal das doppelte zur Übertragungsgeschwindigkeit bezahlen und formuliert seine Policy.

Kunde: $\text{cost} - 2 * \text{speed} \leq 0$

Die Unifikation dieser beiden Policies ist demnach:

[10]Der Einfachheit halber werden die unterschiedlichen Attribute hier direkt in numerische Beziehung zueinander gesetzt. In der Praxis ist jedoch in den meisten Fällen zusätzlich ein entsprechender Umrechnungsfaktor zu berücksichtigen.

Gemeinsam: $\text{cost} - 2 * \text{speed} = 0$

Damit kommen die beiden ins Geschäft.

Computer–Artikel

Das nächste Szenario ist im Bereich der Online–Computer–Artikel Anbieter denkbar. Der Anbieter formuliert seine Angebote in Form einer Policy. In diesem Szenario hat der Anbieter zwei Angebote, einmal ein Gigabyte Mainboard mit einem AMD K6–II bestückt und einmal ein ASUS Mainboard mit einem IBM M2 bestückt.

Anbieter: Prozessorpreis = 239 AND Prozessortyp = "AMD K6 II 300"
AND
Mainboardpreis = 175 AND Mainboardtyp = "Giga SG100"
OR
Prozessorpreis = 198 AND Prozessortyp = "IBM M2 300"
AND
Mainboardpreis = 189 AND Mainboardtyp = "ASUS P5A"

Ein Kunde, der kein Wert auf besondere Details legt, sondern für den nur der Preis im Vordergrund steht, setzt sich für seinen Einkauf von Prozessor und Mainboard eine obere Preisgrenze. Der Kunde setzt sich als obere Grenze 400,- DM. Seine Policy lautet demnach:

Kunde: Mainboardpreis + Prozessorpreis ≤ 400

Die Unifikation der beiden Policies ergibt ein Angebot, daß aus der folgenden Policy abgelesen werden kann:

Gemeinsam: Prozessorpreis = 198 AND Prozessortyp = "IBM M2 300"
AND
Mainboardpreis = 189 AND Mainboardtyp = "ASUS P5A"

Schulnoten

Es folgt nun ein Beispiel, das die Anwendung der durch das Simplex–Verfahren ermöglichten Optimierungsfunktionen demonstriert (s. Abschnitt 2.3.3.3). Dabei wird auch von einem benutzerdefinierten Eigenschaftstyp Gebrauch gemacht (s. Abschnitt 4.2.1.3).

Monika und Rolf wollen zusammen für das Abitur lernen. Beide haben die Prüfungsfächer Mathe und Deutsch. Damit sie sich sicher sind, dass sie beide das gleiche Ziel bezüglich der Benotung anstreben, formulieren beide ihre Policy. Rolf ist ein wenig anspruchsloser, was die Benotung angeht und formuliert seine Policy folgendermaßen (hierfür wird der benutzerdefinierte Eigenschaftstyp *Zensur*, der einen Wertebereich von 0 bis 15 hat, verwendet):

Rolf: Mathe + Deutsch ≥ 16

Monika hingegen setzt ihre Ziele höher, ihre Policy lautet:

Monika: Mathe + Deutsch ≥ 20

Um ihre Policies auf Übereinstimmung zu überprüfen, benutzen sie den Policy–Manager und rufen den Unifikationsalgorithmus auf. Dieser liefert die gemeinsame Policy:

Gemeinsam: Mathe + Deutsch $\geq$ 20

Damit wird Monika Rolf wahrscheinlich dazu bringen, auch ein besseres Abitur–Zeugnis zu erlangen. Die beiden wollen jetzt noch sehen, wie die Benotung aussieht, die das (rechnerische) Optimum im Minimum ihrer gemeinsamen Policy darstellt. Dafür benutzen sie die Optimierungsfunktion des Policy Managers und lassen sich die (rechnerische) optimale Belegung im Minimum anzeigen. Als Ergebnis erhalten sie:

Optimale Belegung im Minimum: Mathe = 15; Deutsch = 5

Denkbar wäre auch das umgekehrte Ergebnis. Der Benutzte Simplexalgorithmus kann jedoch nicht die Semantik der Properties erkennen. Jetzt interessiert die beiden noch das maximale Ergebnis. Dafür benutzen sie wieder die Optimierungsfunktion des Policy Managers um die optimale Belegung im Maximum zu erhalten. Die optimale Belegung im Maximum hat dann die Belegungen beider Fächer mit 15 Punkten als Ergebnis:

Optimale Belegung im Maximum: Mathe = 15; Deutsch = 15

4.5 Zusammenfassung

Ein wesentliches Ziel der Implementierung war der Nachweis der technischen Realisierbarkeit der Steuerungskonzepte sowie ihrer Anwendbarkeit in praxisnahen Umgebungen. Deshalb wurde bei der Wahl von Programmierumgebungen, insbesondere einer Programmiersprache, und Basissoftwareprodukten darauf geachtet, dass die realisierten Mechanismen möglichst reibungslos in praxisrelevante Systeme und Plattformen im Bereich Verteilte Systeme integriert werden können.

Die Regelverarbeitungsmechanismen einschließlich der logikbasierten, generischen Verarbeitungsfunktionen wurden dann in Form portabler Klassenbibliotheken komplett in der Programmiersprache Java implementiert. Diese Bibliotheken besitzen eine modulare und *hierarchische* Struktur, so dass innerhalb der Implementation ein hoher Grad an Wiederverwendbarkeit erreicht wird. Dabei wurden anwendungsnahe Komponenten wie etwa die CORBA–Dienste oder Marktplatz–spezifische Funktionen als eigenständige Schichten realisiert, die in der Implementationshierarchie entsprechend höher angesiedelt sind.

Die unterschiedlichen *Anwendungsszenarien*, die dann entworfen und implementiert wurden, zeigen deutlich, dass die entwickelten Regelmechanismen tatsächlich zu einer sehr flexiblen und dynamischen Steuerung von vielen Anwendungssystemen eingesetzt können. Dies ist hauptsächlich darauf zurückzuführen, dass das gesamte Steuerungskonzept ausschließlich auf der Modifikation von Dienstattributen basiert. Da solche Attribute — in der Praxis meistens als *Properties* bezeichnet — in der einen oder anderen Form bei nahezu jedem Anwendungssystem zu finden sind, lassen sich die dargestellten Steuerungsmechanismen sehr leicht integrieren, ohne das jeweilige Zielsystem verändern zu müssen.

Allerdings könnten sich in Bezug auf einen breiten Praxiseinsatz die bereits genannten Einschränkungen der Implementation negativ auswirken. Deshalb wäre es sinnvoll, in diesem Fall weitere Bemühungen zur Behebung dieser

Schwächen zu unternehmen. Beispielsweise könnte die Einschränkung der PM–Bibliothek auf die Benutzung von *vorzeichenunbehafteten* Typen dadurch gelöst werden, dass — falls vorhanden — andere Simplex–Algorithmen, bei denen das Problem nicht auftritt, eingesetzt werden; oder durch eine entsprechende Abbildung von vorzeichenbehafteten auf vorzeichenunbehaftete, womit das Problem ebenfalls gelöst werden könnte. Darüber hinaus wären im Praxiseinsatz *Performanz*verbesserungen wünschenswert. Bspw. dauert ein Aufruf über die Rule–sensitive Middleware um ein Vielfaches länger als ein normaler (entfernter) Methodenaufruf. Dies ist hauptsächlich auf die Erzeugung und Koordination der Rule–Events und des Transaktionsmanagements zurückzuführen. Hierzu könnte insbesondere ein effizienteres Event–Management Abhilfe schaffen.

Kapitel 5

Konzeption von Verhandlungsmechanismen

In diesem Kapitel schliesst sich die Konzeption von Verhandlungsmechanismen als eine autonome, interaktionsorientierte Form der Steuerung offener verteilter Anwendungssysteme an. Da automatisierte Verhandlungen in der aktuellen Forschung bereits ein umfangreiches eigenständiges Themengebiet darstellen, wird zunächst ihre Bedeutung und Anwendungspotential in einem eigenen Abschnitt (5.1) motiviert. Anschliessend folgt in Abschnitt 5.2 eine Analyse automatisierter Verhandlungsformen, wobei der Schwerpunkt auf den technischen Aspekten liegt. Daraus wird dann in Abschnitt 5.3 eine Architektur selbständig verhandelnder Agenten, die auf dynamisch austauschbaren Verhandlungsmodulen basieren, konzipiert. Abgeschlossen wird das Kapitel mit einer Zusammenfassung in Abschnitt 5.4.

5.1 Motivation

5.1.1 Verhandlung als funktionale Erweiterung regelbasierter Steuerungsmechanismen

Regelbasierte Steuerungsmechanismen, wie sie in den vorangegangenen Kapiteln vorgestellt wurden, stellen ein Mittel zur Steuerung verteilter Anwendungen dar, das konzeptuell relativ einfach und vor allem systemtechnisch transparent ist, da die Funktionalität der gesteuerten Anwendungen zumindest semantisch komplett unabhängig ist von den Regelinteraktionen, die *ausserhalb* der Anwendungen selbst angestossen und durchgeführt werden. Dabei wird die Steuerungswirkung i.a. dadurch erzielt, dass bei gewissen Aktivitäten Regeln bezüglich des (gewünschten) Verhaltens der beteiligten Komponenten getriggert, durch einen Unifikationsmechanismus automatisch auf einen gemeinsamen Nenner gebracht und dann aktiviert werden, so dass diese Komponenten bei ihrer Interaktion ein *aufeinander abgestimmtes* Verhalten aufweisen.

Allerdings ist es sowohl intuitiv als auch durch die Darstellung der Regelverarbeitungsfunktionen klar geworden, dass regelbasierte Steuerungsmechanismen schnell an ihre Grenzen stossen können, so bald die Regeln der beteiligten Komponenten semantisch (bzw. logisch) miteinander unvereinbar sind und die Unifi-

kationsfunktion deshalb fehlschlägt. Wenn nämlich die beabsichtigte Interaktion zwischen den Komponenten in solchen Fällen dennoch zustande kommen soll, muss entweder eine "höhere" Instanz — wie z.B. die betreffenden Endanwender — oder ein expliziter zusätzlicher Automatismus eingeschaltet werden, um den Konflikt aufzulösen. Aus unterstützungstechnischer Sicht interessiert vor allem, wie letzteres aussehen kann. Zwar wurden in Abschnitt 2.3.2.4 *Schlichtungsregeln* als ein potenzieller Konfliktauflösungsmechanismus vorgeschlagen, doch wurde ebd. bereits die Ansicht geäussert, dass ein solcher Ansatz nicht sehr weit führen würde, da eine *ausschliesslich* auf logischen Regeln basierte Schlichtung grundsätzlich die Frage aufwirft, warum diese (Schlichtungs-)Regeln nicht von vorn herein mit den eigentlich zu unifizierenden Regeln verknüpft werden, um einen effizienteren einstufigen Unifikationsprozess zu ermöglichen. Darüber hinaus stellen sich auch grundsätzlich die Fragen, *was* sinnvolle Schlichtungsregeln sind und — falls es für eine spezifische Situation mehrere alternative gibt — *wie* sie einzusetzen sind (z.B. in welcher Reihenfolge, um den Unifikationsprozess zum Erfolg zu verhelfen).

Die konsequente Fortführung derartiger Überlegungen führt anscheinend unweigerlich zum Begriff der Verhandlung als eines *mehrstufigen* Prozesses, in dem *Angebote* generiert und (nach einer bestimmten Vorschrift) zwischen den Beteiligten ausgetauscht werden, bis eine *Einigung* (im allgemeinen) erzielt wird oder ein Beteiligter (Verhandlungspartner) den Prozess abbricht. In Verbindung mit der angestrebten *Generizität* (siehe Abschnitt 1.4.1) erscheint es deshalb sinnvoll, anstatt spezielle Techniken zur Auflösung konfligierender Regeln weiter zu verfolgen, ein technisches Rahmenwerk zur Integration beliebiger Verhandlungsmechanismen in verteilte Anwendungen zu entwickeln.

Im Gegensatz zu der eher passiven Rolle der um Regelmechanismen erweiterte Komponenten in Bezug auf ihre Steuerung spielen Anwendungen, die Verhandlungsfähigkeiten besitzen, eine *aktive* Rolle bei der Findung einer gemeinsamen Interaktions- bzw. Kooperationsbasis, da sie aus Sicht des jeweiligen Interaktionspartners die Angebote autonom in Form von *Verhandlungsnachrichten* generiert (wie im weiteren Verlauf der Arbeit noch konkret gezeigt wird). Deshalb können Verhandlungsmechanismen als eine Form der *Selbststeuerung* bzw. Selbstanpassung von verteilten, in einer offenen Umgebung interagierenden Anwendungen betrachtet werden, insbesondere wenn die beteiligten Anwendungen nicht einmalig auftreten, sondern häufiger aufeinander treffen.

5.1.2 Verhandlung als interaktionsbasierte Nutzenoptimierung

Verhandlungsmechanismen — vorausgesetzt, dass sie generisch konzipiert sind — können aber im allgemeinen als eine *verteilte*, interaktive Art der Nutzenoptimierung betrachtet werden, die auf eine Vielzahl von Domänen anwendbar sind. Bezüglich der elektronischen Informationsverarbeitung scheint ihre Bedeutung deshalb proportional zu der steigenden Verteilung und Interaktionsorientierung, die in Kapitel 1 beschrieben sind, zu wachsen. Zu den Gebieten, auf denen Verhandlungsmechanismen eingesetzt bzw. erforscht werden, zählen System- und Netzwerkmanagement (insb. *Active Networks* [Wie95b]), Verteilte KI (insb. *Contract Nets* [DS83]) und vor allem das relativ junge Gebiet E–Commerce bzw. E–Business, da dieses am engsten mit dem klassischen Anwendungsgebiet von Verhandlungstechniken, nämlich dem (konventionellen) Han-

del, zusammenhängt und deshalb zumindest auf konzeptioneller Ebene direkt von den zahlreichen theoretischen Abhandlungen der klassischen Wirtschaftswissenschaft profitieren kann.

Die Vorteile *automatisierter* Verhandlungsmechanismen bzw. einer systemtechnischen Verhandlungsunterstützung für E–Commerce–Anwendungen insgesamt und insbesondere für das Gebiet der elektronischen Dienstemärkte, das als Hauptanwendungsfeld für diese Arbeit betrachtet wird (vgl. Abschnitt 1.1), liegen auf der Hand. Erstens können viele elektronisch unterstützte Handelstransaktionen, die bisher immer noch Interaktionen mit Endanwendern erfordern, dadurch (voll-)automatisiert und entsprechend verbilligt werden. Bei einem Autokauf übers Internet beipielsweise muss der Interessent bisher immer den Anbieter oder Autohändler selbst kontaktieren, um einen günstigeren Preis als den ausgeschriebenen oder andere Konditionen auszuhandeln. Dies wird normalerweise über E–mails, Web–basierte Dialog–Tools oder auch Telefon ausgeführt und kann viel Zeit der Beteiligten in Anspruch nehmen. Wenn der Autohändler stattdessen ein vollautomatisches System zur Generierung von Angeboten und Gegenangeboten und auch zur Kommunikation mit den verhandlungswilligen Kunden verwenden könnte, würde er dadurch sehr viel kostbare Arbeitszeit sparen können.

Ein weiterer Vorteil ist, dass bestehende E–Commerce–Systeme, in denen automatische Verhandlungen möglich sind, z.B. in Form autonom verhandelnder Softwareagenten, einen zusätzlichen Nutzen für die Benutzer bringen bzw. neue Handelsformen entstehen lassen können. Während ohne Verhandlungen nur die Möglichkeit besteht, dass ein Kunde einen Agenten beauftragt, eine Ware zu suchen und zu kaufen, bieten Verhandlungen mehr Flexibilität: Wenn der Agent kein genau auf die Vorgaben des Auftragebers passendes Angebot findet, kann er mit Verhandlungen versuchen, ein solches aktiv *herbeizuführen*. Umgekehrt gilt das gleiche für den Anbieter: Durch Verhandlungen ist es für ihn möglich, auch aus einem zunächst nicht passend erscheinenden Interessenten vielleicht doch einen Kunden zu machen. Damit gibt es also für den Einkaufsagenten die Möglichkeit, nicht einfach nur den Kauf einer Ware abzulehnen, weil sie den Benutzervorgaben nicht entspricht, sondern mit dem jeweiligen Anbieter einen *Kompromiss* abzuschliessen, der nicht immer allein den Preis betreffen muss, denn häufig scheitert eine Handelstransaktion daran, dass keine *alle* Vorgaben aller Beteiligten erfüllende Lösung gefunden wird, obwohl diese existiert. Und da automatische Verhandlungen in Situationen verwendet werden können, in den menschliche Verhandlungen zu teuer wären, z.B. beim Kauf von Waren, die von einem typischen Kaufhaus angeboten werden, können sie zu neuen Handelsformen führen — wie etwa Kaufhäusern mit beliebig verhandelbaren Waren — die die Wirtschaft insgesamt noch stärker verändern werden. In der Tat hat der Trend zur Verhandlung von Waren, die in der konventionellen Kommerz immer noch zu festen Listenpreisen verkauft werden, im Internet bereits längst begonnen. Auf Online–Auktionen wie eBay [eBa] oder OnSale [OnS] werden immer mehr neuwertige Waren (wie Laptops) direkt von den Herstellern angeboten. Jedoch werden auch für solche Auktionen bislang noch keine vollautomatisierten Verhandlungsmechanismen eingesetzt.

Darüber hinaus wird dem Konzept des Verhandelns in einer Reihe weiterer Anwendungsgebiete, die nicht explizit Gegenstand dieser Arbeit sind, zunehmende Bedeutung beigemessen. Beispielsweise werden im Bereich *Quality of Service* (QoS) häufig Verhandlungsmechanismen als Mittel zur dezentralen Op-

timierung von QoS–Parametern vorgeschlagen (siehe z.B. [MPT98, FdM98]).

5.2 Analyse automatisierter Verhandlungsformen

5.2.1 Klassifikationskriterien

Die Fachliteratur aus den Gebieten Wirtschaftswissenschaften (insb. der *Spieltheorie*) und der Informatik (insb. der Verteilten KI und zunehmend auch E–Commerce) bietet zwar eine vielfältige Fülle an Abhandlungen über konventionelle als auch (semi-)automatische Verhandlungsmechanismen (siehe [BS97] und [Seb92], die sehr unterschiedliche Überblicke der bestehenden Ansätze geben), jedoch existiert keine klare und vor allem erschöpfende Klassifikation von Verhandlungen sowie entsprechenden Techniken, die eine systematische Untersuchung erheblich erleichtern würde. Bei genauerer Betrachtung stellt sich aber heraus, dass der Verhandlungsbegriff — der i.a. nur als eine zielgerichtete Interaktion, wobei das Ziel die *Einigung* über einen bestimmten Sachverhalt ist, charakterisiert werden kann — in der Theorie als auch in der realen Praxis so viele Formen annehmen kann, dass eine vollständige Klassifikation kaum möglich wäre. Es gibt daher nur unterschiedlich abstrakte *Kriterien*, die man anwenden kann, um Verhandlungsmechanismen zu untersuchen.

Die allgemeinste und wahrscheinlich auch sinnvollste Art der Klassifikation stammt aus dem Gebiet der ökonomisch motivierten Spieltheorie und orientiert sich an der Anzahl der Verhandlungspartner bzw. der Verhandlungsgegenstände. So werden in dem Standardwerk von Raiffa [Rai82] (informelle) Verhandlungsmechanismen nach den Kriterienpaaren *multilateral* vs. *bilateral* und *multi–issue* vs. *single–issue* gruppiert. Dies bietet bereits einen (kleinen) Satz an wichtigen Unterscheidungsmerkmalen, da beispielsweise eine multilaterale Verhandlung grundsätzlich anders abläuft als eine bilaterale, die letztendlich immer auf einen abwechselnden Austausch von Nachrichten hinausläuft. Allerdings finden sich jeweils innerhalb einer solchen Gruppe noch weitere wichtige Kriterien, die zu berücksichtigen sind, beispielsweise ob bei einer multilateralen Verhandlung *Koalitionen* unter den Teilnehmern gebildet werden (dürfen) oder welche Rolle die zur Verfügung stehende *Verhandlungszeit* spielt.

Eine weitere generelle Klassifikation von Verhandlungen geben Rosenschein und Zlotkin in [RZ94a, RZ94b]. Dabei werden die unterschiedlichen Verhandlungen anhand der jeweiligen Anwendungsgebiete unterschieden. Dabei werden drei verschiedene Domänen identifiziert, in denen Verhandlungen eine Rolle spielen:

- *Aufgabendomänen*: Hier geht es um Verhandlungen, bei denen Aufgaben unter den Agenten aufgeteilt werden sollen. Ein Beispiel ist die Zustellung von Briefen. Dabei kann die Zustellung jedes einzelnen Briefes als eine Aufgabe angesehen werden. Jeder Agent muss bestimmte Briefe austragen, hat also am Anfang eine Menge von Aufgaben, die er abarbeiten muss. Wenn die Agenten jedoch untereinander die Briefe als Resultat einer Verhandlung geschickt aufteilen, kann es sein, dass sie ihre jeweiligen Wege optimieren können. Entsprechend geht es bei der Aufgabendomüne meist um die Umverteilung von Aufgaben, d.h. jeder Teilnehmer an der Verhandlung muss eigentlich bestimmte Aufgaben erfüllen, aber diese können durch die Verhandlung umverteilt werden. Entsprechend muss jeder Agent

bei einem erfolglosen Abbruch der Verhandlung die ihm ursprünglich zugeteilten Aufgaben übernehmen.

- *Zustandsdomänen*: Dabei handelt es sich um solche Domänen, in denen jeder Agent die Welt in einen bestimmten Zustand bringen will. Wenn man sich beispielsweise eine Welt mit mehreren Klötzchenstapeln vorstellt, könnte es das Ziel des einen Agenten sein, den weissen Klotz auf dem schwarzen zu haben. In einer solchen Domäne kann es entgegengesetzte Ziele geben und damit auch Konflikte. Beispielsweise könnte ein anderer Agent anstreben, den schwarzen Klotz auf dem weissen zu haben. Dann hätte die beiden Agenten ein Konflikt, da sie genau entgegengesetzte Ziele haben.

- *Wertedomänen*: Bei solchen Domänen geht es genau wie bei den Zustandsdomänen auch um einen Zustand der Welt, jedoch kann jedem dieser Zustände ein bestimmter Wert zugeordnet werden. Zum Beispiel kann es im Interesse des einen Agenten sein, ein Treffen vormittags zu arrangieren (hoher Wert), aber auch nachmittags wäre möglich (niedriger Wert). In der Verhandlung versucht er dann das Treffen so zu arrangieren, dass er einen möglichst hohen Wert erzielt. Eine Zustandsdomäne kann somit auch als Wertedomäne angesehen werden, bei der nur jeweils ein Zustand einen Wert > 0 hat. Normalerweise ist dies aber nicht der Fall, und somit gibt es dann auch keine Konflikte.

Rosenschein und Zlotkin untersuchen diese Domänen und machen dann Vorschläge für Verhandlungsprotokolle für die jeweiligen Domänen (s. a. Abschnitt 5.2.2.1).

Kumar und Feldman [KF98] fokussieren sich beispielsweise nur auf *Auktionen* als eine Form der multilateralen Verhandlung. Diese haben auch deswegen eine besondere Bedeutung, weil zusammen mit der zunehmenden Aufhebung von Preisbindungen und der Einführung von Rabatten feste Preise immer mehr zugunsten von frei verhandelbaren Preisen abgelöst werden, wie bereits in Abschnitt 5.1.2 erwähnt wurde. Auktionen sind dabei eine Möglichkeit, diese Preise zu verhandeln und zwar eine sehr erfolgreiche, wie man an den vielen Auktionsanbietern im Internet (z.B. [eBa]) sehen kann.

Kumar und Feldman nehmen eine Klassifikation von Auktionen anhand von drei Merkmalen vor:

- *Art der Interaktion*: Bei öffentlichen Auktionen wird eine Auktion bei einem öffentlichen Treffen simuliert, wie man es beispielsweise von Gemäldeauktionen bei den verschiedenen Auktionshäusern kennt. Dabei werden die Gebote der einzelnen Parteien allen anderen zugänglich gemacht und diese müssen dann in einer relativ kurzen Zeit eigene Gebote bekannt geben. Daher müssen alle Teilnehmer zum selben Zeitpunkt an der Auktion teilnehmen und sie müssen auch dazu in der Lage sein, innerhalb kurzer Zeit auf Gebote entsprechend zu reagieren, während das Auktionssystem dazu in der Lage sein muss, die jeweiligen Gebote zuverlässig in einer kurzen Zeit allen Auktionsteilnehmern zur Verfügung zu stellen.

  Bei Auktionen mit versiegelten Geboten wird von jedem Teilnehmer zu einer bestimmten Zeit ein Gebot eingeholt, das dann ausgewertet wird. Man kann dann noch weitere Runden mit neuen Geboten anfügen oder es

bei dieser einen Runde belassen. Damit können die Teilnehmer zu unterschiedlichen Zeiten an der Auktion teilnehmen, denn sie müssen lediglich zu einem bestimmten Zeitpunkt ein Gebot abgegeben haben. Ausserdem können die Teilnehmer eine längere Zeit über das nächste Angebot nachdenken, bevor sie es abgeben.

- *Kontrolle der Gebote und der Angebote*: Die erste Alternative ist, dass entweder der Auktionator oder der Anbieter einen Vorschlag für den Preis machen. Dabei kann er entweder mit einem hohen Preis anfangen und diesen so lange reduzieren, bis er sein Vorrat verkaufen kann ("Holländische Auktion") oder er fängt mit einem niedrigen Preis an und steigert ihn so lange, bis nur noch so viele Käufer übrig sind, wie er mit seinem Vorrat bedienen kann. Alternativ können die Bieter mit beliebigen Geboten die Auktion beginnen lassen und diese dann immer höher schrauben.

- *Bestimmung des Preises*: Zwar bekommen immer diejenigen den Zuschlag, die das höchste Gebot abgegeben haben, aber es ist trotzdem die Frage offen, wieviel sie tatsächlich für die Ware bezahlen müssen. In einer "Amerikanischen Auktion" müssen die jeweiligen Bieter tatsächlich auch ihr Gebot bezahlen, was dann ein Problem ist, wenn es mehrere Auktionen mit identischen Angeboten gab und dann unterschiedlich viel für die Gegenstände bezahlt wurde. Also gibt es alternativ die Möglichkeit, dass alle Gewinner den Preis zahlen, der das niedrigste gewinnende Gebot war. Dies wird häufig ebenfalls als "Holländische Auktion" bezeichnet. Ein solches Verfahren wird beispielsweise häufig bei der Preisfestlegung für neue Aktienemissionen verwendet. Eine weitere Alternative ist, dass jeder Gewinner den Preis zahlt, der dem höchsten nicht erfolgreichen Gebot entspricht. Damit kann in bestimmten Situationen die Manipulation der Auktion durch überhöhte oder zu niedrige Gebote vermieden werden ("Vickrey–Auktion").

Neben dieser groben Klassifikation von Auktionen kann man noch weitere Variationen anhand anderer Merkmale unterscheiden. So kann man zum Beispiel unterschiedlich viele Informationen über die Gebote preisgeben: Man kann jedem Bieter die Höhe aller Gebote und die Identität des jeweiligen Bieters bekannt machen, oder man kann die Anonymität soweit treiben, dass lediglich das höchste Gebot aber keine Identitäten der Bieter veröffentlicht wird. Es gibt dann noch zahlreiche weitere Merkmale, die man variieren kann. Von diesen sollen hier nur noch die Regeln für das Schliessen der Auktion erwähnt werden: Man kann die Auktion zu einem bestimmten Zeitpunkt beenden, oder man kann das Ende als den Zeitpunkt bestimmen, an dem neue Gebote nur mit einer niedrigen Frequenz eintreffen.

Eine weitere interessante Art von Verhandlungen stellen Sandholm und Lasser in [SL95] vor. Dabei gibt es als Ergebnis einer Verhandlung nicht einfach eine bindende Übereinkunft, sondern es gibt unterschiedlich starke Bindungen. Man kann dann gegen Zahlung einer bestimmten Strafe von der Bindung wieder zurücktreten. Damit ist es möglich, Verhandlungen flexibler zu gestalten. Die Agenten können beispielsweise mit mehreren Agenten eine niedrige Bindung eingehen und dann einen auswählen. Man erleichtert ausserdem eine Abwägung des jeweiligen Risikos und kann Alternativen zu dem Risiko bieten, einen vorgeschlagenen Vertrag auf jeden Fall einzuhalten.

Insgesamt lässt sich bezüglich der Klassifikation von Verhandlungen festhalten, dass es ein sehr breites Spektrum an unterschiedlichen Verhandlungsarten gibt. Man kann zunächst grob zwischen bilateralen und multilateralen Verhandlungen unterscheiden. Auktionen als eine Form von multilateralen Verhandlungen spielen eine zunehmend wichtige Rolle im Kontext von E–Commerce und wurden einer eigenen Klassifikation unterzogen. Darüber hinaus wurde auch eine nach verschiedenen Domänen gerichtete Klassifikation vorgestellt.

5.2.2 Anforderungen an automatisierte Verhandlungen

Obwohl automatisierte Verhandlungsmechanismen sich stark an konventionelle Verhandlungen richten bzw. dazu intendiert sind, diese — zumindest teilweise — zu ersetzen, kann es erhebliche Unterschiede zwischen dem Ablauf einer automatisierten und und dem einer konventionellen Verhandlung, in der nur Menschen direkt mitaneinder verhandeln, geben, und zwar in Bezug auf folgende Aspekte:

- Psychologie: Der wahrscheinlich eindeutigste Unterschied zwischen einer automatisierten und einer konventionellen Verhandlung besteht darin, dass bezüglich ersterer psychologische Faktoren — zumindest nach dem gegenwärtigen Stand der Technologie — ausgeschlossen werden können[1], während bei letzterer die Psychologie nicht selten die *entscheidende* Rolle spielt bzw. den Ausschlag gibt. Interessant und zugleich herausfordernd ist deshalb die Frage, welche Auswirkungen die Psychologie auf eine Verhandlung, an der sowohl Softwareagenten als auch menschliche Verhandlungsführer teilnehmen, haben kann.

- Performanz: Im Vergleich zu menschlichen Verhandlungsführern bieten maschinelle Systeme sicherlich eine rechnerisch und auch *kombinatorisch* überlegene Performanz und können deshalb bei komplexen Verhandlungsgegenständen, z.B. bei einer Verhandlung um die Zuteilung einer grossen Menge von Aufgaben, auch mehr (rein rechnerische) Lösungsmöglichkeiten entdecken. Im Gegensatz dazu steht jedoch der schwer zu formalisierende Aspekt der *Kompromissfähigkeit*, bezüglich dessen die menschliche Flexibilität und Kreativität wohl kaum von maschinellen Systemen übertroffen werden kann.

- Berechenbarkeit: Die formale Berechenbarkeit der Algorithmen, die von einem maschinellen Verhandlungsführer benutzt werden, impliziert auch seine informale Berechenbarkeit in dem Sinne, dass seine Strategie leichter preisgegeben werden kann, was in den meisten Verhandlungsszenarien einen Nachteil zur Folge hat. Dies scheint in der Tat eine der grössten potenziellen Schwächen von automatisierten Verhandlungsmechanismen zu sein. Allerdings existieren bereits eine Reihe von nicht–trivialen formalen Strategien (s. Abschnitt 5.2.2.2), für die noch der konkrete Nachweis erbracht werden muss, dass (und wie leicht) sie von anderen Verhandlungsteilnehmern aufgedeckt werden können.

[1] Selbst wenn Softwareagenten eines Tages menschliche Psychologie verstehen oder simulieren können, stellt sich bzgl. einer vollautomatisierten Verhandlung die Frage, wozu psychologische Faktoren dienen sollen.

- Spielregeln: Ein weiterer grosser Unterschied zwischen automatisierten und konventionellen Verhandlungen betrifft das Verständnis bzw. die Notwendigkeit von klar definierten Spielregeln bzgl. der Durchführung einer Verhandlung. Dies wird in Abschnitt 5.2.2.1 näher erläutert.

Um automatisierte Verhandlungsmechanismen zu realisieren, die in praktischen Szenarien ihren Zweck erfüllen sollen, müssen diese Unterschiede berücksichtigt werden und entsprechend zunächst ein Satz an geeigneten Anforderungen an derartige Mechanismen aufgestellt werden. Ohne Einschränkung der Allgemeinheit werden maschinelle Verhandlungsführer im folgenden auch als (verhandelnde) Agenten[2] bezeichnet.

Im allgemeinen müssen zur Automatisierung von Verhandlungen folgende Anforderungen erfüllt werden:

Kommunikationssprache Verhandelnde Agenten müssen in der Lage sein, *Verhandlungsnachrichten* von anderen Teilnehmern zu empfangen, zu bearbeiten und als Reaktion darauf auch wieder neue zu erzeugen und zu versenden. Hierfür wird eine entsprechende Kommunikationssprache zur Generierung der Nachrichten benötigt. Wichtig hierbei ist nicht nur eine klar definierte Syntax und Semantik der Nachrichten, sondern auch eine geeignete Abstraktionsstufe und Standardisierung, so dass solche Nachrichten von möglichst vielen Agenten, und idealerweise auch von menschlichen Verhandlungsführern, direkt interpretiert werden können.

Verhandlungsprotokoll Bei den eben aufgelisteten Unterschieden zwischen automatisierten und konventionellen Verhandlungen wurde auch die Bedeutung von "Spielregeln" erwähnt. Es ist offensichtlich, dass bei einer vollautomatisierten Verhandlung die Spielregeln explizit gemacht und formalisiert werden müssen, um einen sinnvollen, d.h. gewünschten, Verhandlungsablauf zu gewährleisten. Die Formalisierung der Regeln für eine Verhandlungsklasse wie z.B. die genannte "Holländische Auktion" wird i.a. als ein Verhandlungsprotokoll bezeichnet. Was ein solches Protokoll im einzelnen regeln soll, wird gleich im nächsten Abschnitt beschrieben.

Verhandlungsstrategie Zur Verhandlung benötigt jeder Teilnehmer auch immer eine entsprechende Strategie, die normalerweise nicht preisgegeben wird. Was eine Verhandlungsstrategie leisten soll, wird in Abschnitt 5.2.2.2 erörtert.

Ausser diesen grundlegenden Anforderungen existieren noch weitere wichtige Aspekte, die beim Einsatz von automatisierten Verhandlungsmechanismen berücksichtigt werden müssen. Dazu gehören vor allem Aspekte wie Vertrauenswürdigkeit, einschliesslich Fragen der Rechtsgültigkeit und -bindung von automatisch ausgehandelten Verträgen, sowie Sicherheitsanforderungen (s. hierzu z.B. auch [Mer99a]). Allerdings betreffen diese Aspekte nicht direkt die Frage der Automatisierung von Verhandlungen und werden in dieser Arbeit nicht weiter behandelt.

[2] In der Tat werden in der später dargestellten Verhandlungsarchitektur mobile Softwareagenten eingesetzt (s. Abschnitt 5.3).

5.2.2.1 Anforderungen an Verhandlungsprotokolle

Ein Verhandlungsprotokoll — verstanden als ein Satz von formalen Regeln — zur Steuerung einer automatisierten Verhandlung muss möglichst alle semantischen und ablauftechnischen Aspekte der Verhandlung in Betracht ziehen. D.h. es darf im Gegensatz zu konventionellen bzw. informell geführten Verhandlungen durch menschliche Teilnehmer keine Annahmen über jegliches *implizites* Verständnis der Teilnehmer über die Verhandlungssemantik — z.B. wird bei einer informell geführten Autokaufverhandlung zumindest erwartet, dass nicht eine der beiden Parteien plötzlich ein Preisangebot für sein Haus anstatt für das Auto abgibt — machen, sondern muss klar definieren, was ein gültiges Verhalten bei der Verhandlung ausmacht, so dass möglichst jede Art von absichtlichem (strategisch motiviertem) oder unabsichtlichem inkorrektem Verhalten aufgedeckt werden kann. Die dabei zu berücksichtigenden Aspekte sind in der Tat sehr vielfältig und können in folgende Kategorien gruppiert werden (s. a. [TGML98]):

Verhandlungsgegenstand Eine Verhandlung kann einen oder mehrere Verhandlungsgegenstände zum Ziel haben, die genau festgelegt werden müssen. Eine Änderung der Verhandlungsgegenstände darf nur erfolgen, wenn das Protokoll dies explizit erlaubt. Ein Verhandlungsgegenstand wird durch eine Menge von *Attributen* spezifiziert, die sowohl *verhandelbare* als auch *nicht-verhandelbare* Elemente umfassen kann.

Verhandlungsteilnehmer An Verhandlungsteilnehmer können unterschiedliche Bedingungen gestellt werden, die in folgende Aspekte gruppiert werden können.

- Rollen: In vielen Verhandlungsklassen sind den Teilnehmern feste Rollen zugewiesen, die ihre Beziehung zum Verhandlungsgegenstand darstellen und festlegen, welche Aktionen sie während der Verhandlung durchführen können. Bei einer Ausschreibung beispielsweise darf ein potenzieller *Auftragnehmer* nur *Preis*angebote an den *Auftraggeber* unterbreiten.
- Kardinalität: Für jede Rolle muss spezifiziert werden, wieviele Instanzen es davon *mindestens* geben muss und wieviele davon *höchstens* an der Verhandlung teilnehmen können.
- Ein- und Austritt: Es muss darüber hinaus spezifiziert werden, ob und *unter welchen Bedingungen* Teilnehmer in eine laufende Verhandlung aufgenommen bzw. von dieser ausgeschlossen werden dürfen.

Verhandlungsablauf Das Verhandlungsprotokoll muss den Ablauf einer automatisierten Verhandlung möglichst genau regeln, um inkorrekte Verläufe der Verhandlung vermeiden bzw. rechtzeitig erkennen und behandlen zu können. Dazu gehört die Festlegung folgender Aspekte.

- Nachrichtenaustausch: Um völlig unkontrollierte Datenströme, beispielsweise durch ständiges Wiederholen einer Nachricht, zu vermeiden, darf der Austausch von Verhandlungsnachrichten zwischen den Teilnehmern im allgemeinen nicht beliebig sein, sondern muss gesteuert ablaufen. D.h. es muss festgelegt werden, wer an wen in welcher

Reihenfolge Nachrichten senden darf. Beispielsweise wird in vielen konventionellen Verhandlungen der Gesamtverlauf in *Runden* eingeteilt, in denen jeder Teilnehmer meist genau ein Angebot abzugeben hat oder alle gemeinsam über einen Vorschlag abstimmen müssen. In diesem Fall muss aus dem Verhandlungsprotokoll erkennbar sein, was eine solche Verhandlungsrunde ist und wieviele Runden (maximal) durchzuführen sind.

- Zeitüberschreitung (*Timeout*): Umgekehrt muss auch die Zeitspanne spezifiziert werden, in der eine Verhandlungsaktion oder die Verhandlung insgesamt abgeschlossen werden muss. Naheliegenderweise sind für jede Verhandlungsaktion ein *Timeout* und entsprechende Massnahmen bei dessen Überschreitung zu spezifizieren.
- Abstimmungsverfahren: Was es bedeutet, eine *Einigung* erzielt zu haben, muss durch das Verhandlungsprotokoll festgelegt sein. Eine Einigung muss nicht unbedingt durch allgemeinen Konsens aller Beteiligten zustandegebracht werden; es können auch beliebige andere Abstimmungsverfahren, wie z.B. absolute Mehrheit, relative Mehrheit oder auch eine einfache Teileinigung, angewendet werden.
- Abbruchbedingung: Bedingungen, unter denen die Verhandlung zu beenden (bei Erfolg) oder abzubrechen (bei Misserfolg) ist, sind zu spezifizieren.

Angebotsvalidierung Über die Steuerung des Nachrichtenaustausches im Sinne einer Flusskontrolle hinaus kann ein Verhandlungsprotokoll auch vorschreiben, welche Semantik bestimmte Nachrichten haben müssen. Beispielsweise wird bei einer Auktion meistens verlangt, dass jedes neue Angebot (vom Betrag) *höher* sein muss als das letzte gültige. Eine solche semantische Überprüfung kann entweder protokollinhärent oder auch durch eine vom Protokoll festgelegte externe Komponente durchgeführt werden (siehe Abschnitt 6.2).

Verbindlichkeit Im Falle eines erfolgreichen Abschlusses bzw. einer erzielten Einigung muss das Ergebnis der Verhandlung im allgemeinen nicht für alle Teilnehmer verbindlich sein. Umgekehrt kann sich die Verbindlichkeit des Verhandlungsergebnisses auch auf Teilnehmer, die diesem nicht zugestimmt haben, erstrecken, wie z.B. im Falle eines Parteitagsbeschlusses oder einer Aktionärsversammlung. Deshalb ist auch die Verbindlichkeit der Ergebnisse jeweils zu definieren.

Aus der obigen Darstellung lässt sich verallgemeinern, dass es neben statisch beschreibbare Anforderungen auch viele *dynamische* Aspekte einer Verhandlung gibt — wie etwa Ein- und Austritt von Teilnehmern, Timeout–Behandlung, Validierung etc. — die sich nicht (allein) durch ein statisches mathematisches Modell erfasst werden können. Deshalb ist es notwendig, ein adäquates Repräsentationsformat für Verhandlungsprotkolle zu wählen, das möglichst viele der genannten Aspekte auf eine allgemeine Weise ausdrücken kann.

Zusätzlich zu diesen formalen Anforderungen, die auf eine Automatisierung von Verhandlungen abzielen, sind bei einer konkreten Realisierung noch weitere anwendungsbezogene Anforderungen wie *Verständlichkeit* und *Überprüfbarkeit* an Verhandlungsprotokolle zu stellen, da nur so eine breite Akzeptanz für die

Protokolle erreicht werden kann. Das Thema der Spezifikation von Verhandlungsprotokollen wird in Abschnitt 6.2.1 weiter behandelt.

5.2.2.2 Anforderungen an Verhandlungsstrategien

Obwohl Verhandlungsprotokolle den Zweck erfüllen sollen, die Anzahl der möglichen Abläufe bei einer automatisierten Verhandlung einzuschränken und damit ein *zielgerichtetes* Verhandeln zu gewährleisten, müssen sie offensichtlich dennoch genügend Verhandlungsspielraum für die Teilnehmer belassen. D.h. abgesehen von Situationen, in denen gewisse Abbruchbedinungen erfüllt sind, muss es zu jedem Zeitpunkt während des Verhandlungsprozesses unterschiedliche Handlungsalternativen geben, von denen ein Teilnehmer auswählen kann, um die Verhandlung fortzuführen. Um solche Entscheidungen zu treffen, benötigt jeder Teilnehmer eine eigene *Verhandlungsstrategie*, die normalerweise nicht preisgegeben wird, da Verhandlungen in den meisten Fällen in Eigeninteresse durchgeführt werden. Für jede Art von Verhandlungen ist es von entscheidender Bedeutung, jeweils eine geeignete Strategie zu wählen oder zu erfinden. Dazu bietet die Literatur eine grosse Bandbreite von sehr informell gehaltenen, allgemeinen Ratschlägen bis hin zu mathematisch präzisen, jedoch speziellen Ansätzen. Eine ausführliche Diskussion vieler bekannten Verhandlungsstrategien (deren Formalitätsgrad ungefähr zwischen den genannten Extremen angesiedelt ist) findet sich beispielsweise in dem Werk von Raiffa aus dem Jahr 1982 [Rai82].

Wenn es in erster Linie um automatisierte Verhandlungsformen geht, stellt sich zunächst die Frage, wie eine Verhandlungsstrategie *formalisiert* werden kann. Allgemeine Kriterien zur Formalisierung von Verhandlungsstrategien umfassen u.a. folgende:

Nutzenfunktion Intuitiv zielt eine Verhandlungsstrategie immer darauf, ein "gutes" Ergebnis zu erzielen. Um dies zu erreichen, muss eine formalisierte Strategie die Güte einer Verhandlungsaktion, vor allem die Güte von Angeboten und Gegenangeboten, in Form einer messbaren Grösse bewerten können. Deshalb werden geeignete *Nutzenfunktionen* benötigt. Dabei müssen häufig — insbesondere können bei *multi–issue* Verhandlungen — mehrere gegensätzliche Faktoren wie z.B. Preis und Qualität oder auch Verhandlungszeit gleichzeitig berücksichtigt werden, was zu einem Optimierungsproblem führen kann.

Wissensbasis Um gute Ergebnisse erzielen können, muss ein verhandelnder Agent darüber hinaus in der Regel Kenntnisse über die Verhandlungsdomäne, z.B. den aktuellen Marktpreis für ein bestimmtes Produkt und dessen Streuung, oder auch spezifische Informationen über *einzelne* Verhandlungspartner, z.B. Ergebnisse der bisher mit ihm durchgeführten Verhandlungen, besitzen. Diese semantischen Informationen zu verwalten und auszuwerten kann eine komplexe Aufgabe sein, die häufig von einer externen Wissensbasis übernommen wird.

Protokollkonformität Es besteht eine sehr enge semantische Beziehung zwischen Verhandlungsprotokollen und -strategien. Erstens ist es im Sinne einer Automatisierung notwendig für die Strategie, Verhandlungsaktionen zu generieren, die zu dem verwendeten Protokoll konform sind. Zweitens

kann der Entwurf eines Protokolls grossen Einfluss auf das strategische Verhalten der Teilnehmer, wie die Beispiele in [RZ94b] prägnant zeigen.

Ressourcenmodellierung Eine Verhandlungsstrategie kann i.a. jedoch nicht beliebige Ressourcen verbrauchen, um gute Resultate zu erzielen, sondern muss in der Regel immer einen Kompromiss zwischen der Qualität eines Ergebnisses und den vorhandenen Rechenressourcen treffen. Dabei stellt die *Zeit* offenbar die wichtigste Ressource dar, weil der Zeitpunkt, zu dem ein Angebot vorgelegt wird, häufig einen direkten Einfluss auf das Verhandlungsergebnis hat.

Algorithmus Schliesslich, jedoch nicht zuletzt, wird auch ein konkreter Algorithmus zur Berechnung von Verhandlungsaktionen, einschliesslich Angebote und Gegenangebote, benötigt.

Gerade bezüglich der letztgenannten Anforderung existieren vielfältige Vorschläge in der aktuellen Literatur zum Thema automatische Verhandlungen, da im Prinzip jede beliebige Art von Algorithmen zur Berechnung von Angeboten denkbar ist. In Abschnitt 6.3 wird hierauf noch näher eingegangen.

5.2.3 Strukturelle Bestandteile von Verhandlungen

In diesem Abschnitt geht es um die Gemeinsamkeiten hinsichtlich der Bestandteile aller elektronischen Verhandlungen. Um eine effektive Unterstützung für einen automatisierten Verhandlungsprozess anbieten zu können, werden diese Grundbausteine herausgearbeitet. Diese sind der Grundbaustein für die weitere Arbeit, diese Aspekte ergeben eine Protokollbeschreibungssprache und bilden die Anforderungen für eine Steuerungseinheit.

5.2.3.1 Rollen

In Verträgen oder Verhandlungsbeschreibungen der realen Welt bezieht man sich in aller Regel nicht direkt auf die Identität der Teilnehmer. Die abstrakte Gestalt der Verhandlungsbeschreibung bzw. die Semantik des Verhandlungsprozesses würde verloren gehen, statt dessen wird von funktionalen Rollen gesprochen: z.B. Auktionator, Interessent, Anbieter.

In Verträgen werden die handelseinigen Teilnehmer aus Gründen der Einfachheit häufig mit einem Synonym versehen. Dieses spiegelt die Rolle des Vertragspartners: Mieter, Vermieter, Käufer oder Verkäufer. Die Beschreibung einer Rolle enthält deshalb nur Angaben über die Aufgaben und Rechte, die von dem Inhaber dieser Position zu erbringen sind. In der Informatik ist die Modellierung von Rollen üblich, dieser Umstand vereinfacht das Übertragen von Verträgen und Verhandlungsprotokollen in das Verhandlungssystem, wie in Abbildung 5.1 veranschaulicht wird (die Semantik der Symbole wird später in Abschnitt 6.2.1.2 präzisiert).

Damit Verhandlungsprotokolle wieder verwendbar sind, dürfen in ihnen keine Annahmen über konkrete Identitäten gemacht werden. In diesen Verhandlungsprotokollen, die den Ablauf einer möglichen Verhandlung beschreiben, werden statt dessen Platzhalter für die späteren Teilnehmer verwendet.

Zur Verdeutlichung sei das Beispiel eines Verhandlungsprotokolls genannt: Für den Kauf eines Autos wird es mit den Rollen des Verkäufers und des Käufers

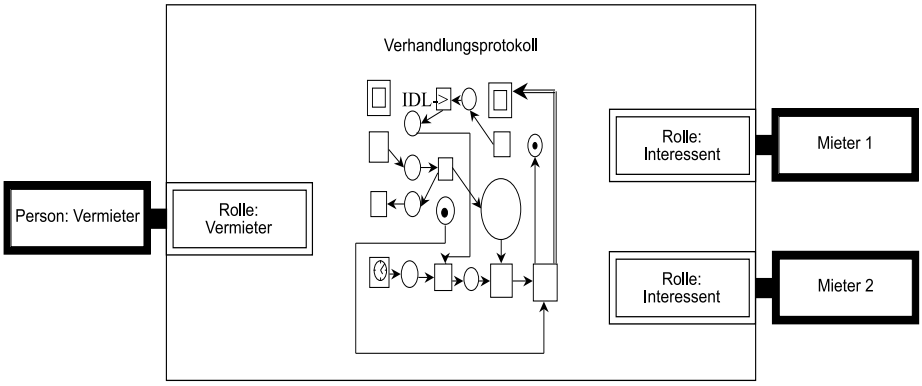


Abbildung 5.1: Beispiel der am Verhandlungssystem beteiligten Rollen

besetzt. Eine Rolle kann dabei mit mehreren Teilnehmern besetzt werden, bei einer Auktion würde die Rolle eines Bieters von allen potentiellen Bietern ausgefüllt werden. Ein Teilnehmer kann jedoch auch mehrere Rollen gleichzeitig besetzen. Ein Zwischenhändler könnte Waren einkaufen und so die Rolle eines Einkäufers übernehmen und diese dann wieder verkaufen.

Bei der Anmeldung eines neuen Teilnehmers für eine bestimmte Rolle in einer Verhandlung muss darauf geachtet werden, dass die Teilnehmer ihrer Rolle sowohl syntaktisch wie auch semantisch gerecht werden: Zunächst muss sichergestellt sein, dass der Teilnehmer mit dem System und den anderen Teilnehmern kommunizieren kann. Dazu muss er die rollenspezifischen Nachrichten syntaktisch verarbeiten und senden können. Ein weiterer wichtiger Aspekt ist das semantische Ausfüllen dieser Rolle. Es wäre unsinnig, einen Mieter in eine Verkäuferrolle zu stecken, selbst wenn der Teilnehmer zufällig in der Lage ist, die Kommunikationsleistung zu erbringen.

Der letzte Punkt behandelt die Sicherheit während der Verhandlung. Das falsche Besetzen der Rollen, z.B. ein Verkäufer besetzt während einer Auktion die Rolle eines Bieters um die Preise in die Höhe zu treiben, muss unter allen Umständen vermieden werden. Dies ist nicht immer Aufgabe des Verhandlungssystems, allerdings sollte die Sicherheit unterstützt werden, z.B. durch einen Notariatsdienst, der alle Verhandlungsnachrichten protokolliert, um eine Anzeige nachweisen zu können.

5.2.3.2 Kommunikationswege

Damit eine Verhandlung zustande kommen kann, werden zwischen den Teilnehmern einer Verhandlung Nachrichten ausgetauscht. Im realen Leben geschieht dies mündlich, telefonisch, über Briefe, Email oder andere Kommunikationswege. Es gibt aber auch Verhandlungen, z.B. Ausschreibungen, bei denen der ungehinderte Austausch von Informationen verboten ist. In den Beschreibungen dieser Verhandlungen ist festgelegt, wer an wen Nachrichten senden kann. Das Verhandlungssystem selbst muss solche Verbindungsstrukturen modellieren (siehe Abbildung 5.2).

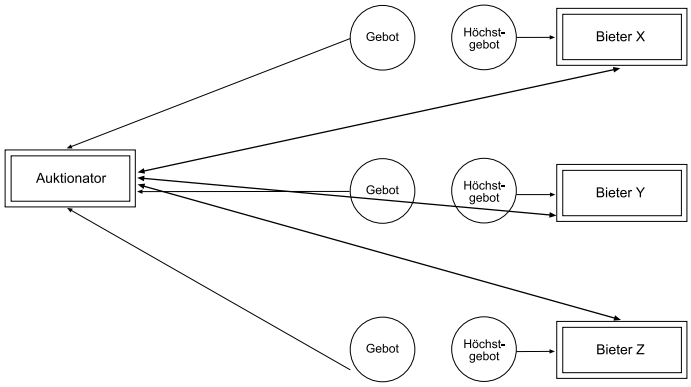


Abbildung 5.2: Beispiel Kommunikationswege

5.2.3.3 Nachrichtentypen

Während einer Verhandlung arbeiten die unterschiedlichen Parteien an einem Vertrag. Dieser Vertrag besteht aus verschiedenen Attributen, die z.B. den Verhandlungsgegenstand beschreiben. In einer Auktion werden nach der Beschreibung des Gegenstandes nur Beträge explizit ausgetauscht. Da die Währung im allgemeinen vorher festgelegt ist, beschränken sich die ausgetauschten Daten auf Zahlen, häufig sogar auf ganze Zahlen oder die Nachricht: "Das nächste Vielfache eines vorher festgelegten Betrages sei das neue Höchstgebot".

In komplexeren Verhandlungen, beispielsweise beim Aushandeln der Konditionen für ein grosses Projekt zwischen mehreren Universitäten, einigen Industrieunternehmern, staatlichen Stellen und Privatpersonen, werden ganze Vertragswerke per Post oder Email verschickt. Mündliche Absprachen werden in der Regel nur auf bestimmte Teile eines Vertrages angewendet.

Diese Beispiele implizieren, dass ein Angebot aus mehreren Informationen besteht, die eine Struktur bilden. Bei der Kommunikation über Ausschnitte eines Angebots muss klar sein um welche Bestandteile es genau geht. Aus formaler Sicht muss die betreffende Teilstruktur eindeutig identifiziert sein. In einer automatisierten Verhandlungsumgebung ist es notwendig, die oben genannten Attribute zu typisieren. Nur so kann formal sichergestellt werden, dass ein Einkäufer für Waschmaschinen den Vertrag nicht über ein Auto abschliesst.

In einem formalen System ist es wichtig, zusätzlich zu den Vertragsinhalten Informationen über den Verhandlungszustand an alle Parteien zu übermitteln. Im Verlauf der weiteren Arbeit werden diese Informationen im Zusammenhang mit dem Metaverhandlungsprotokoll erwähnt (siehe Abschnitt 6.2.3). Diese Informationen können von atomaren Daten bis hin zu komplexen Objektstrukturen reichen. Beispiele für solche Informationen sind:

- Kontrolldaten, wie die Anzahl der bereits von Teilnehmer X versandten Angebote.
- Zustandsdaten, wie die Angabe, dass ein Teilnehmer die folgenden Angebote nur akzeptieren oder ablehnen, aber kein Gegenangebot machen darf.
- Steuernachrichten, wie "Bitte geben Sie jetzt ein Angebot ab".

5.2.3.4 Zustand einer Verhandlung

Den Prozess einer Verhandlung kann man nicht mit statischen Mitteln beschreiben. Eine Verhandlung ist ein dynamischer Vorgang, das Ende ist nicht vorhersagbar. Eine Verhandlung befindet sich deswegen implizit immer in einem Zustand, der für den nächsten Schritt ausschlaggebend ist.

Auf der einen Seite haben wir Informationen über die Anzahl der Angebote, Teilnehmer, und wer das letzte höchste Gebot abgegeben hat. Ein Verhandlungssystem für Internetauktionen würde die Anzahl der Teilnehmer und das letzte Gebot für den Zustand speichern. Auf der anderen Seite entsteht ein Zustand durch die eventuell vorher festgelegte Reihenfolge der Verhandlungsnachrichten. Jeder Verhandlungsbestandteil kann sich in einem eigenen Zustand befinden. Dies betrifft die möglichen Angebote (siehe [KF98]), die Verhandlungsteilnehmer und alle weiteren Zustände, die den zeitlichen Rahmen der Verhandlung betreffen.

Alle diese Zustände ergeben den Gesamtzustand einer Verhandlung. Während eines Computerkaufs besteht der Gesamtzustand der Verhandlung beispielsweise aus:

- Den Zuständen der Teilnehmer: Käufer1: Angebot abgegeben; Verkäufer: überlegend.
- Den Zuständen der aktuellen Angebote: Das Angebot vom Verkäufer wurde vom Käufer abgelehnt. Ein Gegenangebot vom Käufer wurde zum Verkäufer überbracht.
- Dem zeitlichen Zustand (z.B. dritte Runde).

Je nach Zustand der Verhandlung sind bestimmte Aktionen der Teilnehmer zulässig. Der Zustand der Verhandlung bestimmt ebenfalls die Aktionen des Verhandlungssystems. Auch wenn der Zustand nicht unbedingt explizit modelliert werden muss, ist es sinnvoll, den Verhandlungsteilnehmern eine Schnittstelle anzubieten, die es ihnen erlaubt, den Verhandlungszustand in Erfahrung zu bringen. Dies erfordert eine Modellierung von zumindest reduzierten Zustandsinformationen.

Die Reduzierung ist sinnvoll, da die Mächtigkeit der Gesamtzustandsmenge exponentiell mit der Anzahl der Teilnehmer und Angebote und deren möglichen Zuständen wachsen kann. Vor allem muss die Zustandsmenge jedoch reduziert und aufgeteilt werden, da Teilnehmer vor allem in automatisierten Verhandlungen nur an bestimmten Zustandsinformationen interessiert sind, oder es vom Verhandlungsprotokoll, z.B. aus Gründen der Anonymität, nicht erwünscht ist, dass jeder Teilnehmer über vollständige Zustandsinformationen verfügt (siehe Abschnitt 6.2.3).

5.2.3.5 Kontrollstrukturen

Eine Verhandlung muss abhängig vom Zustand und den Informationen, die ausgetauscht werden, einen bestimmten Verlauf nehmen. Dieser Verlauf, der z.B. die Reihenfolge der einzelnen Angebote bestimmt, wird in dem Verhandlungsprotokoll festgeschrieben. Zusätzlich ist es nötig, dass Regeln aufgestellt werden können, die beschreiben wie die Nachrichten der einzelnen Teilnehmer kontrolliert werden.

Beispielsweise kann in einem Protokoll vorgesehen sein, dass ein Angebot um einen bestimmten Betrag höher als das letzte sein muss. Ansonsten müsste der letzte Vorgang wiederholt werden. Das Verhandlungssystem muss daher mindestens logische Regeln oder sogar komplexe Algorithmen bearbeiten können. Dazu werden die aus Programmiersprachen bekannten Konstrukte wie Bedingungen und Schleifen verwendet. Um mit vielen Parteien gleichzeitig eine Verhandlung zu führen, sollten aus Performanz–Gründen mehrere Teilnehmer gleichzeitig bearbeitet werden können.

Die Nachrichten müssen später wieder zusammengeführt werden, um eine sinnvolle Bearbeitung zu ermöglichen. Diese Problematik ist in der Informatik hinlänglich bekannt, es kommen sogenannte sequenzielle und parallele Strukturen vor. Die Parallelität muss an bestimmten Synchronisationspunkten wieder zusammengeführt werden können.

Ein Synchronisationsbeispiel ist das folgende: In einer Reisverhandlung geben die Verkäufer auf eine Anfrage des Einkäufers Angebote ab. Diese Angebote werden alle gleichzeitig abgegeben, die einzelnen Teilnehmer werden über den Zustand der Verhandlung informiert. Jeder Verkäufer darf beliebig viele Angebote abgeben, nach einer festgesetzten Zeit werden diese eingesammelt und dann dem Einkäufer vorgestellt. Während der Angebotsabgabe arbeitet das Verhandlungssystem demnach parallel, die Angebotsverarbeitung wird danach sequentiell abgearbeitet. Dieser Vorgang ist eine Art der Synchronisation, die in Abbildung 5.3 dargestellt ist.

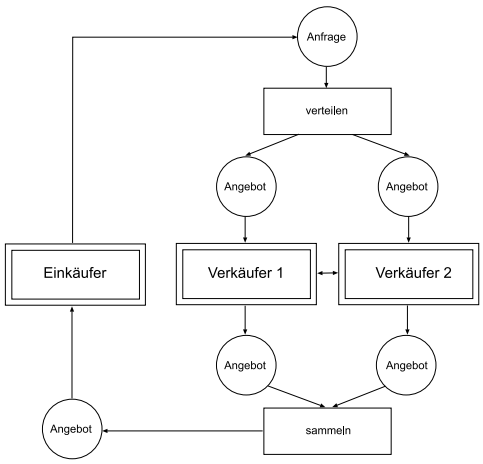


Abbildung 5.3: Synchronisation einer Verhandlung

5.3 Eine Architektur selbstständig verhandelnder mobiler Agenten

In diesem Abschnitt wird eine Architektur zur Automatisierung von Verhandlungen mittels Softwareagenten dargestellt. Hauptziel dieser Architektur ist es zunächst, ein möglichst flexibles Rahmenwerk zur *Integration* beliebiger Verhandlungsmechanismen zu liefern. Der Entwurf und die Realisierung spezifischer Komponenten für diese Architektur werden in Kapitel 6 behandelt.

Zunächst folgt in Abschnitt 5.3.1 die Basisarchitektur eines aus Verhandlungsmodulen zusammengesetzten Agenten. Anschliessend wird diese in Abschnitt 5.3.2 um eine Reihe von Unterstützungsdiensten zu einem vollständigen Verhandlungssystem erweitert.

5.3.1 Architektur der Agenten

5.3.1.1 Anforderungen

Die Realisierung von Verhandlungsmechanismen und das Agentenparadigma passen, zumindest konzeptuell gesehen, ideal zueinander. Einerseits können automatisierte Verhandlungsmechanismen — falls konkret umgesetzt — als eine sehr nützliche funktionale Erweiterung vieler existierender agentenbasierter Systeme im Bereich E–Commerce betrachtet bzw. für diese verwendet werden. Die meisten Agentenprodukte im Bereich verteilte Anwendungen bieten Beispiele an, in denen ein mobiler Agent vom Benutzer einen Auftrag erhält bzw. konfiguriert und dann ins Netz losgeschickt wird. Dort sucht er nach geeigneten Anbietern, sammelt relevante Informationen ein und kann eventuell auch eine Handelstransaktion autonom durchführen (vgl. Einkaufsagenten–Beispiel in Abschnitt 4.4.4). Wenn man sich solche Beispiele vor Augen hält, erscheint es offensichtlich, dass Verhandlungsfähigkeiten den Nutzwert der Agenten deutlich erhöhen würden. Andererseits erscheinen die Softwareagenten zugeschriebenen Basiseigenschaften — nämlich Autonomie und Interaktionsfähigkeit (s. Abschnitt 1.3.4) — ideale Voraussetzungen für bzw. Anforderungen an die Realisierung automatisierter Verhandlungsmechanismen.

Verhandlungen autonom durchzuführen kann auch als ein typisches und anschauliches Beispiel für eine *intelligenzerfordernde* Aufgabe betrachtet werden, wie die Analyse in Abschnitt 5.2, insbesondere bzgl. der Verhandlungsstrategien, gezeigt hat. Anders ausgedrückt ist das Ziel, generische Verhandlungsfähigkeiten in mobile Softwareagenten einzubetten, als eine Form der Integration von *intelligenten* und *mobilen* Agenten zu betrachten [TGL99]. Deshalb müssen beim Entwurf einer entsprechenden Architektur für verhandlungsfähige Agenten, die in typischen E–Commerce–Szenarien zum Einsatz kommen sollen, zunächst allgemeine Anforderungen an eine Integration von *Intelligenzmechanismen* in mobile Agenten in Betracht gezogen werden.

Für einen praxisorientierten Ansatz besteht die Hauptschwierigkeit bzgl. darin, dass die Einverleibung von beliebigen "intelligenten" Fähigkeiten in mobile Agenten sehr "teuer" kommen kann, da die meisten Inferenz- und Wissensverarbeitungsmechanismen viel komplexer sind als typische Methoden (wie "go", "select", "sort" etc.), die in simplen mobilen Agentenanwendungen zu finden sind. Grundsätzlich gilt, die Mobilität eines Agenten nimmt proportional zu seiner Grösse ab (vergleichbar mit der Übertragung von Webseiten). Dieses Problem wird noch verstärkt, wenn das konkrete Ziel bzw. die zu lösende Aufgabe zur Kompilationszeit nicht feststeht, und deshalb entweder viele Agenten für unterschieliche Aufgaben oder sehr universal konzipierte Agenten, die noch grösser (und "schwerfüssiger") sind, eingesetzt werden. Um diesem Problem zu begegnen, müssen folgende Anforderungen an einen entsprechenden Systementwurf gestellt werden:

Rollen–spezifische Fähigkeiten Ein Agent, insbesondere im mobilen Fall, soll jeweils nur mit den Fähigkeiten, die zur Erfüllung seiner aktuellen

Rolle(n) — z.B. *Auktionator* — erforderlich sind, ausgestattet sein. Spezifische Fähigkeiten sollen möglichst gut modularisiert sein, um die Grösse der Implementation zu minimieren.

Dynamischer Funktionserwerb Die jeweils benötigte Funktionalität soll zudem dynamisch in den Agenten hineingeladen können. Damit ist es nicht nur möglich, die Rollen zur Laufzeit zu wechseln, sondern auch die selbe Rolle mit unterschiedlichen Implementationen, die z.B. unterschiedliche Algorithmen verwenden, auszufüllen.

Flexible Konfigurierbarkeit Die geladene Funktionalität des Agenten soll auf eine flexible Weise konfigurierbar sein, damit sie in vielen ähnlichen, jedoch unterschiedlich *bedingten* Situationen verwendet werden kann, ohne die Implementation austauschen zu müssen.

5.3.1.2 Modulares Agenten–Framework

Um die o.g. Anforderungen zu erfüllen, wurde im Rahmen dieser Arbeit ein modulares Agenten–Framework entwickelt, das auf einer flexiblen und losen Verknüpfung von mobilen Agenten mit ihrer Funktionalität basiert. Abbildung 5.4 zeigt die abstrakte Struktur eines solchen Agenten.

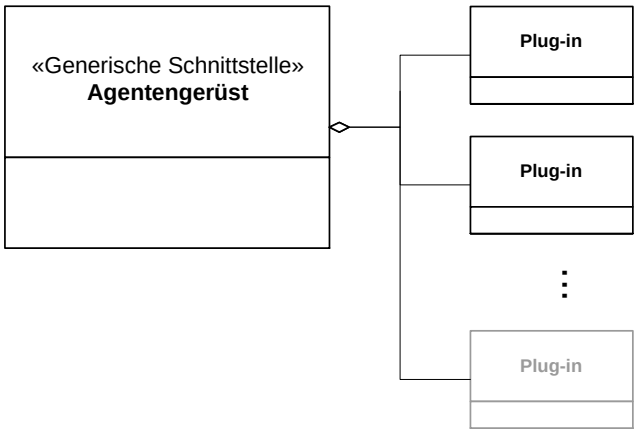


Abbildung 5.4: Abstrakte Modulstruktur eines Agenten

Dabei besteht der Agent aus einem mobilen, persistenten Gerüst mit einer stabilen Referenz, über die während der gesamten Lebensdauer des Agenten Methodenaufrufe abgesetzt werden können. Die eigentliche Anwendungsfunktionalität wird in Form einzelner Module, die als "Plug-ins" bezeichnet werden und zur Laufzeit in das Gerüst "hineingesteckt" werden können (siehe folgenden Abschnitt 5.3.1.3), bereitgestellt. Da solche Module eine beliebige Anwendungssemantik haben und völlig unabhängig voneinander entwickelt werden können, ist das Framework im Prinzip völlig generisch und kann von beliebigen Anwendungsdomänen genutzt werden.

Konzeptionell entspricht diese Modulstruktur zunächst der (extensiven) Anwendung des *Aggregation*skonzepts im objektorientierten Sinne und wird auch

als solche dargestellt. Das Besondere hieran ist, dass die Aggregation nicht statisch, sondern dynamisch stattfindet und damit sind auch die Module mit dem Agentengerüst nur lose gekoppelt. Daraus ergibt sich die geforderte Flexibilität: Module können, während der Agent seinen Aufgaben nachgeht, hinzugeladen, ausgestauscht oder wieder entfernt werden. Jedoch ergeben sich durch diese Dynamik auch neue Fragen und Probleme, die zusätzliche konzeptionelle als auch technische Lösungen erfordern. Zunächst einmal stellt sich die Frage, wie die äussere Schnittstelle des Agenten aussehen soll, denn durch die dynamische Aggregation von Modulen kann sich die Funktionalität des Agenten ändern, und wenn diese auch nach aussen hin nutzbar gemacht werden soll, müsste der Agent eine ebenso dynamische Schnittstelle anbieten[3]. Dynamische Schnittstellen lassen sich jedoch in vielen Programmiersprachen wie Java nicht oder nur mit grundlegender Veränderung der Sprache realisieren. Deshalb wurde in dem hier dargestellten Framework ein einfacher Lösungsansatz für dieses Problem gewählt: Der Agent bietet nach aussen hin eine generische Schnittstelle an, die Methoden zur Abfrage seiner aktuell vorhandenen Schnittstellen enthält[4]. Jedoch ist zu beachten, dass in der Regel nicht jedes neue Modul auch zu einer externen Schnittstelle führt, sondern dass viele Module — wie durch die gleich folgende Einführung der Verhandlungsmodule deutlich werden wird — nur *intern* genutzt werden. Neben dieser generischen Schnittstelle kann der Agent auch weiterhin eine statische Schnittstelle, die beispielsweise den festen Anteil seiner Implementation repräsentiert, besitzen.

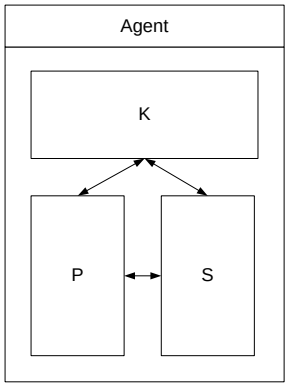


Abbildung 5.5: Verhandlungspezifische Module eines Agenten

Um einem solchen Agenten die Fähigkeit zur Verhandlung zu geben, sind in diesem Framework drei Hauptmodule vorgesehen, die in Abbildung 5.5 symbolisch dargestellt werden:

Kommunikationsmodul (K) Dieses Modul ist zuständig für das Versenden und Empfangen von Nachrichten in einer bestimmten Kommunikations-

[3]Eine damit zusammenhängende Frage ist, ob solche Agenten für jedes Modul eine getrennte Schnittstelle nach aussen anbieten, d.h. *Mehrfachschnitstellen* besitzen, oder ob sie nur eine einzige Schnittstelle, die jeweils um die Methoden eines neu erworbenen Moduls erweitert wird, haben sollen.

[4]Dies kann als eine vereinfachte Variante des *Reflection*–Mechanismus von Java 2, der nur auf Objekte mit festen Schnittstellen anwendbar ist, betrachtet werden. (Siehe a. [Gri98, S. 471 ff.])

sprache. Im Falle von *Verhandlungs*nachrichten muss der Inhalt von dem K–Modul mittels eines *Interpreters* als solcher erkannt werden, wird jedoch nicht von diesem weiter bearbeitet, sondern direkt zum Protokollmodul weiter geleitet.

Protokollmodul (P) Das P–Modul ist zuständig für die Konformität des Agenten zu einem Verhandlungsprotokoll. Deshalb wird der Inhalt jeder bei dem Agenten eingehenden Verhandlungsnachricht zunächst durch dieses Modul geschleust, um sie auf Protokollkonformität zu prüfen. Umgekehrt wird auch jede ausgehende Nachricht vom P–Modul überprüft, bevor sie durch das K–Modul in das richtige Format gebracht und verschickt wird. Dieses Modul kann entweder als eine eigenständige Einheit oder als ein *Front-end* zu einer zentralen Steuerungseinheit (siehe Abschnitt 6.2.3) realisiert werden.

Strategiemodul (S) Das S–Modul implementiert eine konkrete Verhandlungsstrategie, die den Spielraum des (vorgegebenen) Protokolls nutzt, um Verhandlungsaktionen zu generieren, die zu einem gemäss einer vorgegebenen Nutzenfunktion möglichst guten Verhandlungsergebnis führen sollen. Dieses Modul interagiert in erster Linie direkt mit dem P–Modul, um die Protokollkonformität zu gewährleisten, kann jedoch auch protokoll–externe Nachrichten unmittelbar über das K–Modul verschicken, z.B. im Falle einer *verteilten* Strategie (s. Abschnitt 6.3.1).

Diese modulare Aufteilung nach unterschiedlichen Verhandlungsaspekten (gemäss der in Abschnitt 5.2.2 dargestellten Anforderungen) hat neben technischen (auf der losen Verknüpfung der Module basierten) vor allem konzeptionelle Vorteile: Die Funktionalität jedes Moduls ist durch abgegrenzte Schnittstellen klar definierbar, so dass es unabhängig von den anderen entwickelt werden kann[5]. Während das S–Modul in der Regel einen beliebigen Algorithmus zur Generierung von Angeboten verwendet, der allerdings vertraulich bleiben soll, kann das P–Modul sinnvollerweise von einem unabhängigen Drittanbieter geliefert bzw. auf seine Korrektheit zertifiziert werden. Auf diese Weise ist eine klare Trennung zwischen "privaten" strategischen und "öffentlichen" Protokoll–bezogenen Angelegenheiten gegeben.

5.3.1.3 Dynamischer *Plug-in*–Mechanismus

Die Flexibilität der vorliegenden Agentenstruktur ist inhärent verbunden mit der geforderten losen Kupplung der Module. Beispielsweise soll es möglich sein, während einer Verhandlung die Strategie des Agenten durch Austausch des S–Moduls gegen ein anderes komplett zu ändern, ohne dass andere Teilnehmer davon etwas merken. Damit hat ein Benutzer im Prinzip die Möglichkeit, auf das Verhalten des von ihm beauftragten Agenten *beliebig* Einfluss zu nehmen. Diese Art von *funktionaler* Modifikation ist also im gewissen Sinne deutlich "radikaler" als die Steuerung durch Regelobjekte, die in den vorangegangen Kapiteln beschrieben wurde. Um eine solche lose Kupplung zu ermöglichen, bedarf es

[5]Im Falle des Strategiemoduls gilt dies jedoch nur unter der Voraussetzung, dass die Semantik des entsprecheden Protokolls durch eine formale Spezifikation zur Verfügung steht. Dies kann bspw. durch eine formale Sprache, wie sie in Abschnitt 6.2.1 vorgestellt wird, gewährleistet werden.

jedoch eines dynamischen Mechanismus zur Verknüpfung von Laufzeitobjekten. In diesem Abschnitt wird die Funktionalität und der konzeptionelle Entwurf eines *Plug-in*–Mechanismus vorgestellt, der diesen Zweck erfüllen soll. Die Beschreibung der Implementationsdetails dieses Plug-in–Mechanismus erfolgt in Abschnitt 6.1.1.

Plug-in–Typen

Anschaulich ausgedrückt erlaubt der Plug-in–Mechanismus, Funktionalität in Form von *Plug-ins* ("Steckmodule") einem Agenten dynamisch zuzufügen. Bevor darauf eingegangen wird, wie das Zusammenstecken von Modulen technisch funktioniert, soll zunächst geklärt werden, in welcher funktionaler Beziehung Plug-ins zu Agenten stehen können. Im vorliegenden Framework kann zwischen folgenden Plug-ins unterschieden werden.

- *Rollen*: Plug-ins dieses Typs verleihen einem Agenten, oder *Plug-in–Behälter* i.a., eine neue Funktionalität und repräsentieren eine Anwendungssemantik, die konzeptionell nicht mit der jeweiligen Ausführungseinheit fest verknüpft ist. Beispielsweise können in einer Handelstransaktion verschiedene Personen oder Institutionen die Rolle eines *Verkäufers*, *Käufers* oder *Notars* ausfüllen. Umgekehrt kann eine Entität unterschiedliche Rollen (gleichzeitig oder aneinander) einnehmen. Eine neue Rolle in einen Plug-in–Behälter hinzufügen bedeutet, dass dieser sowohl eine neue (oder zusätzliche) Schnittstelle und eine entsprechende Implementation bekommt.

- *Substitute*: Ein Substitut ist ein Plug-in, das eine neue Implementation für eine bestimmte Schnittstelle zur Verfügung stellt bzw. eine bestehende Implementation ersetzt, ohne die betreffende Schnittstelle zu ändern.

- *Konfigurationen*: Eine Konfiguration ist ein Plug-in zur dynamischen Rekonfiguration des Plug-in–Behälters. D.h. sowohl Schnittstellen als auch Implementationen der betreffenden Anwendung bleiben unverändert.

Offensichtlich ist die beabsichtigte Semantik des letzten Plug-in–Typs bereits in Form der dargestellten Regelmechanismen (insbesondere mit dem Regeltyp Policy–Rule) realisiert. Deshalb wird im folgenden nur der Realisierungsansatz für die beiden anderen Plug-in–Typen, die einen dynamischen Austausch der Implementation erfordern, vorgestellt.

Verknüpfung durch Delegation von Methodenaufrufen

Um den dynamischen Austausch von Implementationen zu ermöglichen, wurde für das modulare Agenten–Framework ein Delegationsansatz auf Methodenausführungsebene gewählt. D.h. Methodenaufrufe, die beim Agenten eingehen, werden i.a. nicht direkt ausgeführt, sondern können an beliebige, auch entfernt liegende, Zielkomponenten delegiert werden, falls solche vorhanden sind (d.h. vorher entsprehend registriert wurden). Wichtig hierbei ist, dass es sich um eine *transparente* Delegation handelt, d.h. sowohl Quelle als auch Ziel des Methodenaufrufs bekommen von der Weiterleitung der Methode nichts mit. Ein solcher Delegationsmechanismus bietet eine sehr flexible technische Basis für das "Zusammenstecken" der Module, da die Module in rein funktionaler Hinsicht nur

durch die Delegationsbeziehung miteinander verbunden sind und deshalb all ihre spezifischen Eigenschaften einschliesslich des physikalischen Ortes erhalten bleiben können.

Für den Delegationsmechanismus wurde darüber hinaus das Konzept einer *Kooperationsbildung* entwickelt, das eine *explizite* Definition der Delegationsbeziehung durch ein Kooperationsobjekt zulässt, so dass im Prinzip eine beliebige Methode bei der Zielkomponente anstelle der ursprünglich aufgerufenen Signatur bei der Quellkomponente aufgerufen werden kann. Diese semantische Flexibilität erlaubt eine grösstmögliche Autonomie zwischen den Modulen, z.B. können dadurch gleichwertige Methoden mit unterschiedlichen Namen wie etwa *deliver()* und *liefern()* aufeinander abgebildet oder unterschiedlich gestaltete Parameterlisten angepasst werden.

Insgesamt basiert der Plug-in–Mechanismus auf folgenden Hauptkonzepten.

- *Pluggable*: Das Konzept eines "Steckmoduls" soll orthogonal sein, d.h. Module sollen beliebig ineinander gesteckt werden können, unabhängig von ihrer Rolle in der asymmetrischen Delegationsbeziehung, nämlich *Quell-* oder *Zielobjekt*. Dies wird erreicht durch ein einheitliches, bidirektionales *Pluggable*, das die Delegation von Methodenaufrufen in beiden Richtungen erlaubt. Ein *Pluggable* bietet Methoden, um es jeweils mit einem Zielobjekt zu verbinden oder eine solche Verbindung wieder aufzuheben (*plug()* und *unPlug()*). Diese Methoden erfordern als Parameter ein Objekt, das ebenfalls vom Typ *Pluggable* ist.

  Ein besonderes Merkmal der Realisierung von *Pluggable* Objekten besteht darin, dass sie selbst dynamisch erzeugt werden können. Programmiertechnisch wird die Erzeugung solcher Objekte durch die statische Klasse *Pluggability* geleistet (siehe Abbildung 5.6), die funktional gesehen einem beliebigen Laufzeitobjekt die Eigenschaft der "Steckbarkeit" mittels der Methode *of()* verleiht. *Pluggability* macht ihrerseits Gebrauch von der Klasse *PluggableFactory*, die zusätlich Methoden zur Erzeugung neuer Laufzeitobjekte vom Typ *Pluggable* anbietet (siehe Abschnitt 6.1.1).

- *GenericForwarder*: Um die erwähnte Transparenz des Delegationsmechanismus zu gewährleisten, wird ein steckbar gemachtes Modul nicht verändert, sondern lediglich mit einem entsprechenden Hilfsobjekt verknüpft, das die Methodenaufrufe an dieses Modul abhört und an entsprechende Zielobjekte weiterleitet. Dazu müssen die Aufrufe als *MessageEvents* realisiert sein, so dass der *GenericForwarder* sie abfangen kann.

- *Cooperation*: Dies ist die Komponente, die die semantische Verbindung zwischen der aufgerufenen Signatur (Quelle) und der tatsächlich verwendeten (Ziel) herstellt. Solche *Cooperation*–Objekte werden bei der Registrierung von Zielobjekten in den sogenannten *CooperationSpace* eingetragen, der Methoden zur Selektierung von Zielobjekten für einen bestimmten Methodenaufruf und Abfrage der entsprechenden *Cooperation*–Objekte implementiert.

- *Operation*: Um den Aufruf einer beliebigen Methode delegieren zu können, wird ausserdem ein dynamischer Aufrufmechanismus — ähnlich dem in Abschnitt 4.3.1.5 beschriebenen RS–DII — benötigt. Dieser wird beim Plug-in–Mechanismus von der Klasse *Operation* gekapselt, die Methoden

zur Spezifikation einer beliebigen Signatur und zur Festlegung der jeweils geforderten Aufrufsemantik — z.B. synchron oder asynchron — anbietet.

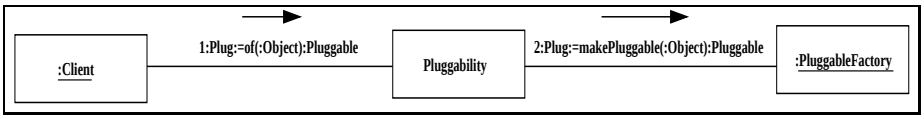


Abbildung 5.6: Konstruktion eines *Pluggable*–Objektes

Die Funktionsweise des *GenericForwarder*, der die Kernkomponente des Plug-in–Mechanismus darstellt, wird in Abbildung 5.7 illustriert. Wenn beim Quellobjekt ein Methodenaufruf eingeht, bekommt diese Komponente ein entsprechendes *MessageEvent* (1), aus dem sie die Signatur der Methode extrahieren kann (2). Der *CooperationSpace* wird dann benutzt, um die Zielobjekte zu bestimmen (3). Mit Hilfe der von diesem zurückgelieferten *Cooperation*–Objekte kann der *GenericForwarder* dann entsprechende Instanzen von *Operation* generieren, um diese dann bei den Zielobjekten abzusetzen (4).

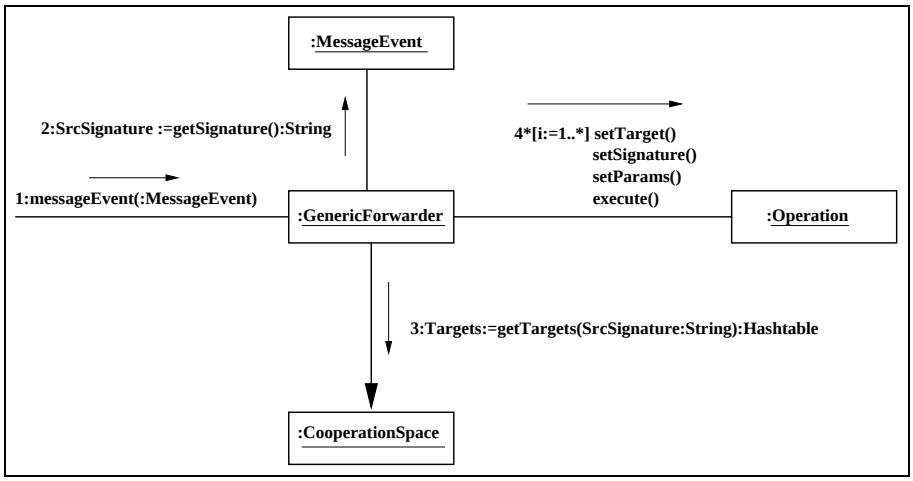


Abbildung 5.7: Funktionsweise des *GenericForwarder*

5.3.2 Architektur des Verhandlungssystems

5.3.2.1 Unterstützungsdienste

Die bisher dargestellte modulare Architektur verhandelnder Agenten ist zwar auf konzeptioneller Ebene bereits funktional vollständig, da jeder mit den entsprechenden Modulen ausgestattete Agent prinzipiell in der Lage ist, mit gleichartigen oder mit allen Verhandlungspartnern, die der selben Kommunikationssprache mächtig sind, zu interagieren. Allerdings müssen bezüglich einer konkreten Realisierung noch wichtige Fragen zu verschiedenen Aspekten der Unterstützung, Koordinierung und Steuerung derartig aufgebauter Agenten beantwortet werden — beipielsweise wie die Agenten die richtigen Verhandlungspartner finden können oder auf welche Weise der Verhandlungsprozess insgesamt

koordiniert werden soll, so dass Protokollverletzungen am besten vermieden werden können. Auf Grund der dynamischen Komponierbarkeit der Agenten stellt sich insbesondere die Frage, wie die Entwicklung und Verwaltung konkreter Verhandlungsmodule unterstützt wird. Deshalb wird im folgenden die Agentenarchitektur um zusätzliche Systemkomponenten für unterschiedliche Phasen des Verhandlungsprozesses, die hier insgesamt als Unterstützungsdienste bezeichnet werden, zu einer praktisch vollwertigen Verhandlungsarchitektur erweitert.

Zusätzlich zu den modularen Agenten besteht diese Verhandlungsarchitektur aus folgenden Komponenten:

Broker Die grundlegende Funktionalität dieser Komponente besteht darin, geeignete Teilnehmer für eine Verhandlung zusammenzuführen. Als Basis für das Finden der passenden Teilnehmer wird dabei ein formalisiertes Verhandlungsprotokoll (s. Abschnitt 6.2) verwendet, das aus Sicht des Brokers in erster Linie eine Sammlung von Rollenspezifikationen einschliesslich deren Kardinalität ist. Die Funktionsweise des Brokers ist prinzipiell mit der eines Traders [OMG96] vergleichbar, insbesondere wenn Rollenspezifikationen *Diensttypen* (vgl. Abschnitt 2.1.2.1) zugrunde gelegt werden.

Protokoll–Engine Um die Einhaltung eines vorgebenen Verhandlungsprotokolls auf eine effiziente Weise zu gewährleisten bzw. um die verhandelden Agenten dabei zu unterstützen, erscheint eine *zentrale* Steuerung im Sinne einer Workflow–Engine die praktisch sinnvollste Lösung. Hierzu ist die Protokoll–Engine vorgesehen, deren Aufgabe ist es, ein (vom Protokoll–Generator erzeugtes) ausgefülltes bzw. parametrisiertes Verhandlungsprotokoll zu interpretieren und die Teilnehmer während der Verhandlung so zu steuern, dass die Semantik des Protokolls gewährleistet (bzw. nicht verletzt) wird. Die Schnittstelle zwischen der Engine und den beteiligten Agenten ist das Protokollmodul (vgl. Abschnitt 5.3.1.2). Dies ist im Fall der zentralen Steuerung eine von der Engine erzeugte, dynamisch in das Agentengerüst einbindbare Komponente, die Methoden anbietet, um bei dem Agenten ein- und ausgehende Verhandlungsnachrichten auf Protokollkonformität zu prüfen. Die Protokoll–Engine muss im Sinne einer konventionellen Workflow–Engine möglichst viele Ausnahmesituationen behandeln können, die sowohl aus inkorrekt spezifizierten Protokollen resultieren (wie z.B. Deadlocks) als auch von den Teilnehmern (absichtlich) verursacht werden (wie z.B. Versuche zu täuschen, stören oder das Protokollmodul zu umgehen). Die Realisierung dieser Komponente wird in Abschnitt 6.2.3 beschrieben.

Protokoll–Generator Dies ist ein Werkzeug zur Spezifikation und Verwaltung formalisierter Verhandlungsprotokolle. Es bietet dem Benutzer die Möglichkeit, die Struktur von Protokollen auf eine graphische Weise zu entwerfen und als persistente Schablonen (*templates*) abzulegen. Wenn eine automatisierte Verhandlung starten soll, wird eine solche Schablone ausgewählt und mit den aktuellen Daten zu einem konkreten Protokoll parametrisiert. Die Realisierung dieser Komponente wird in Abschnitt 6.2.2 beschrieben.

Strategie–Baukasten Konkrete Strategien für automatisierte Verhandlungen müssen in der Regel entwickelt und bereitgestellt werden, bevor die Agenten den Verhandlungsprozess beginnen. Zweck eines Strategie–Baukastens

ist es, die Entwicklung neuer Verhandlungsstrategien zu unterstützen oder fertige anzubieten. In Abschnitt 6.3 wird ein Entwicklungsbaukasten dargestellt, der auf Genetischen Algorithmen basiert.

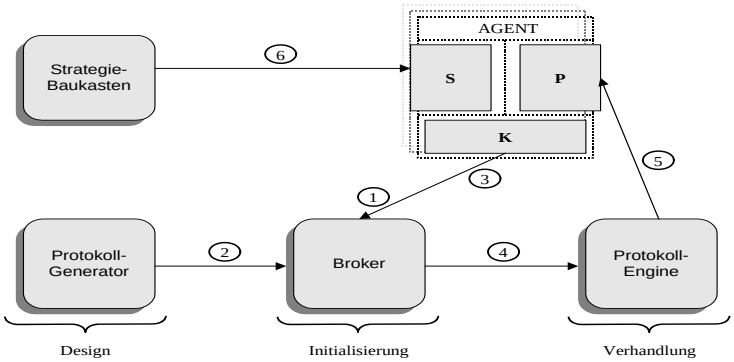


Abbildung 5.8: Schematische Gesamtarchitektur des Verhandlungssytems

Abbildung 5.8 zeigt eine Übersicht aller wesentlichen Komponenten der Verhandlungsarchitektur. Sie macht auch deutlich, wie die einzelnen Komponenten in die Phasen Design, Initialisierung und Verhandlung funktional eingeordnet werden können. Das Zusammenspiel der Komponenten *bis zur Initialisierung* der Verhandlung erfolgt in folgenden Schritten:

1. Ein Agent meldet sich beim Broker, um die Verhandlung zu einem gewissen Gegenstand zu initiieren.

2. Dazu muss er ein Protokoll-Template auswählen — das zuvor mit dem Protokoll–Generator erstellt wurde und bei diesem abgelegt ist — und sich für eine bestimmte Rolle im Template registrieren, z.B. die eines Auktionators.

3. Weitere Agenten registrieren sich als Interessenten für jeweils eine der (noch besetzbaren) Rollen des Templates, z.B. für die Rolle eines Auktionsteilnehmers.

4. Wenn die Startbedingungen für die Verhandlung, beispielsweise wenn die Mindestanzahl von Teilnehmern für eine Auktion erreicht ist, erfüllt sind, werden alle Beteiligten informiert und das ausgefüllte Template an die Protokoll–Engine übergeben.

5. Die Protokoll–Engine erzeugt für jeden Verhandlungsagenten ein entsprechendes Protokoll–Modul, das dynamisch in das Agentengerüst eingebunden wird.

6. Es wird dann ein zu dem Protokoll–Modul passendes Strategie–Modul, das aus einem Strategie–Baukasten ausgewählt werden kann, hinzugefügt und die Verhandlung kann nun beginnen.

5.3.2.2 Informationsfluss

Im folgenden wird die prinzipielle Funktionsweise der Verhandlungsarchitektur entlang der durch die Komponenten fliessenden Informationen erläutert. Dazu sollen zunächst die Informationsübergänge in einer vollständigen automatisierten Verhandlungsumgebung betrachtet werden (siehe Abbildung 5.9).

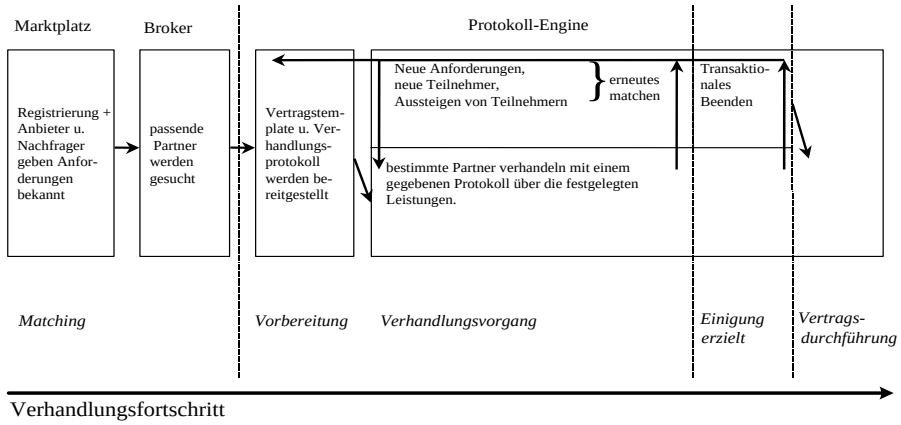


Abbildung 5.9: Informationsübergänge in der automatisierten Verhandlungsumgebung

Zuerst müssen sich potenzielle Verhandlungspartner bei einer Vermittlungsinstanz (dem Broker) registrieren. In einer *Marktumgebung* sind dies üblicherweise Anbieter und Nachfrager für bestimmte Waren, die bei der Registrierung ihre Anforderungen in Form eines strukturierten *Profils* dem Broker übergeben. Die Profile der Teilnehmer können auch Spezifikationen der gewünschten / angebotenen Ware in Form der in Kapitel 2 dargestellten Regeln (insb. *Requirement–Rules*) enthalten, so dass bei der Findung zueinander passender Teilnehmer eine möglichst präzise Auswahl (als *Matching* bezeichnet) vorgenommen werden kann.

Nachdem passende Teilnehmer für eine Verhandlung gefunden sind, übergibt der Broker das entsprechend parametrisierte Verhandlungsprotokoll einschliesslich des dazu gehörigen Vertragstemplates an die Protokoll–Engine, die mit dieser Information die Teilnehmer kontaktieren und sie auf die Verhandlung vorbereiten kann. Während der dann gestarteten Verhandlung kann es jedoch (aufgrund vordefinierter Ereignisse) zu *strukturellen* Veränderungen kommen, die ein erneutes Matching der Teilnehmer erforderlich machen um zu entscheiden, ob die Verhandlung fortgeführt werden kann. Dazu gehören

- Veränderung der Verhandlungsteilnehmerstruktur (neue Teilnehmer, Wegfall von alten Teilnehmern).
- Anträge der Teilnehmer auf Veränderung des Vertragtemplates. Dazu gehört sowohl der komplette Austausch als auch die ausschnittsweise Verände-

rung des bisherigen. Diese Ereignisse können nach der Zulassung eines Antrages auftreten.

- Ausnahmesituationen während der Verhandlung, wie z.B. Verbindungsabbruch oder Verklemmungen (engl. Deadlock) oder Nachrichten zugehörig zum Synchronisations- und Transaktionsverfahren. Dazu gehören Meldungen, die z.B. bei Verhandlungsbeginn und Verhandlungsende generiert werden.

Im Falle einer Einigung der Parteien bzw. ist das Verhandlungsziel erreicht, muss die Verhandlung für alle Teilnehmer beendet werden. Der dabei entstandene Vertrag wird ab jetzt als gültig angesehen. Das Verhandlungssystem muss für den Vertragsabschluss ein transaktionales Verhalten sicherstellen. Das ist für einige Verhandlungen notwendig, in denen Angebote unter bestimmten Bedingungen gemacht werden können, wie zum Beispiel mit begrenzter zeitlicher Gültigkeit. Das System muss feststellen, dass eine Einigung innerhalb der Bedingungen erzielt wurde und den Vertragsabschluss erzwingen. Dieses kann problematisch sein, da derjenige, der die (zeitliche) Bedingung gestellt hat, abstreiten kann, dass die Angebotsannahme rechtzeitig erfolgt ist. Wenn der Empfangszeitpunkt die massgebliche Grösse ist, muss das Verhandlungssystem dafür Sorge tragen, dass entweder alle Teilnehmer vom Vertragsabschluss Kenntnis nehmen können oder gar keiner (transaktionales Verhalten). Dafür kann das *Three–Phase–Commit*–Protokoll aus dem Datenbankbereich [OV91] eingesetzt werden. Schliesslich wird der gültig abgeschlossene Vertrag der Vertragsdurchführung übergeben, womit der Verhandlungsprozess beendet wird.

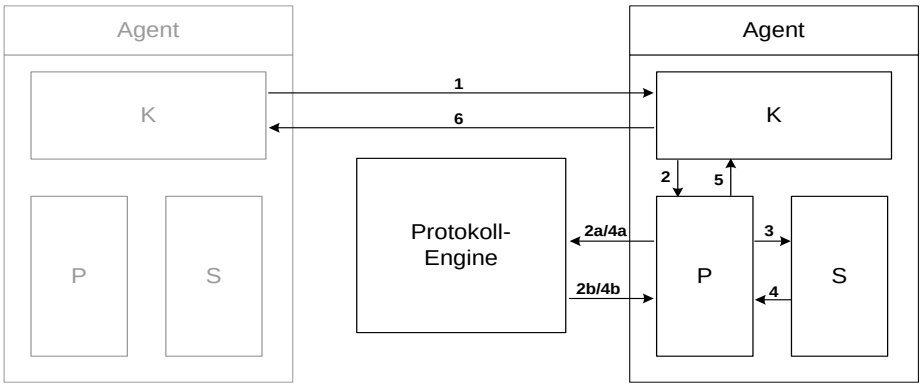


Abbildung 5.10: Informationsfluss zwischen Agentenmodulen und Protokoll–Engine

Während der Verhandlungsphase müssen die teilnehmenden Agenten Verhandlungsnachrichten verarbeiten und protokollkonform darauf reagieren. Dazu findet ein Informationsfluss zwischen den Modulen des Agenten und der Protokoll–Engine statt, der in Abbildung 5.10 veranschaulicht wird und folgende Schritte umfasst:

1. Eine Verhandlungsnachricht wird durch einen Verhandlungspartner an das Kommunikationsmodul des Agenten gesendet.

2. Das Kommunikationsmodul interpretiert durch Kenntnis der Kommunikationssprache und leitet sie an das Protokollmodul weiter.

3. Das Protokollmodul prüft die Gültigkeit der Verhandlungsnachricht mit Hilfe der Protokoll–Engine (Schritt 2a u. 2b) und leitet sie nur an das Strategiemodul weiter, wenn die Nachricht protokollkonform ist.

4. Das Strategiemodul generiert darauf hin eine neue Verhandlungsnachricht und leitet sie an das Protokollmodul weiter.

5. Das Protokollmodul prüft wieder mit Hilfe der Protokoll–Engine die Gültigkeit der Nachricht (Schritt 4a u. 4b) und leitet sie weiter an das Kommunikationsmodul.

6. Das Kommunikationsmodul kodiert die neue Verhandlungsnachricht in der Kommunikationssprache und sendet sie an den oder die entsprechenden Verhandlungspartner.

5.4 Zusammenfassung

In diesem Kapitel wurde die konzeptionelle Basis für ein agentenbasiertes, vollautomatisiertes Verhandlungssystem gelegt. Um eine möglichst domänenunabhängige und damit generische Architektur zu ermöglichen, wurde dabei der Begriff "Verhandlung" im allgemeinen als eine *zielgerichtete Interaktion* aufgefasst. Eine solche Interaktionsform kann einerseits als funktionale Erweiterung der in den vorigen Kapiteln behandelten regelbasierten Steuerungsmechanismen, andererseits aber auch als ein Mittel zur Nutzenoptimierung in einer offenen, *marktorientierten* Umgebung aufgefasst werden.

In diesem allgemeinen Sinne wurde zunächst eine *Analyse* automatisierter Verhandlungsformen mit dem Ziel vorgenommen, die *Anforderungen* an eine konkrete Realisierung so umfassend wie möglich zu erkennen. Es stellte sich dabei heraus, dass zu einer Automatisierung von Verhandlungsprozessen jeweils drei grundlegende Komponenten gehören: eine Kommunikationssprache, ein Verhandlungsprotokoll und eine Verhandlungsstrategie. Dabei können vor allem Verhandlungsprotokolle und -strategien sehr vielfältige Formen annehmen, weshalb eine möglichst *offene* Architektur angestrebt werden soll. Darüber hinaus lassen sich *strukturelle* Bestandteile eines automatisierten bzw. elektronisch unterstützten Verhandlungsprozesses identifizieren, die von einem entsprechenden System umgesetzt werden müssen.

Auf dieser Analyse aufbauend wurde dann eine generische und dynamische *Architektur* selbständig verhandelnder Agenten entworfen. Charakteristisch für solche Agenten ist ihre *modulare* Struktur, wobei die Module mittels eines *Plug-in*–Mechanismus nur lose miteinander verknüpft sind und auch zur Laufzeit gegen andere Module vom selben Typ ausgestauscht werden können. Darüber hinaus gehören zur Gesamtarchitektur des agentebasierten Verhandlungssystems verschiedene *Unterstützungsdienste*, die in ihrem koordinierten Zusammenspiel sowohl die Konzeption als auch die Durchführung von automatischen Verhandlungen wesentlich erleichtern sollen.

Kapitel 6

Realisierung eines Verhandlungssystems für mobile Softwareagenten

Es folgt nun in diesem Kapitel die Beschreibung eines Verhandlungssystems für mobile Softwareagenten, das als eine konkrete Ausprägung der in Kapitel 5 dargestellten agentenbasierten Verhandlungsarchitektur betrachtet werden kann. Ein konkretes System zur Durchführung automatisierter Verhandlungen muss neben den Agentenmechanismen vor allem die Fragen beantworten, wie konkrete Verhandlungsprotokolle und -strategien realisiert werden sollen. Die Semantik von Protokollen und Strategien ist im allgemeinen jedoch nicht an eine spezifische Agentenarchitektur oder gar -Implementation gebunden, deshalb soll ihr Entwurf und Realisierung auch möglichst unabhängig von den Agentenmechanismen betrachtet werden.

Dementsprechend folgt nach der Darstellung der Agentenimplementation in Abschnitt 6.1 die Realisierung von Verhandlungsprotokollen (6.2) und -strategien (6.3) in jeweils eigenen Abschnitten. Darüber hinaus stellt sich bei Verhandlungen durch autonom handelnde Softwareentitäten geradezu offensichtlich die Frage nach ihrer Steuerbarkeit. Hierzu bieten sich wiederum die in den Kapiteln 2 bis 4 entwickelten Regelmechanismen an. Deshalb wird im anschließenden Abschnitt 6.4 gezeigt, wie diese eingesetzt werden können, um insbesondere Protokoll- und Strategieaspekte zu steuern. Abgeschlossen wird das Kapitel mit einer Zusammenfassung in Abschnitt 6.5.

6.1 Agentenmechanismen

Für die Umsetzung der im letzten Kapitel vorgestellten Verhandlungsarchitektur gilt im allgemeinen die gleiche methodische Vorgehensweise wie die bzgl. der Implementierung regelbasierter Steuerungsmechanismen in Abschnitt 4.1, die deshalb hier nicht wiederholt wird. Zusätzliche, verhandlungsspezifische Anforderungen werden an den entsprechenden Stellen genannt.

Im folgenden wird die Realisierung der modularen Agentenarchitektur (siehe Abschnitt 5.3.1) beschrieben. Dazu gehören im wesentlichen die Implementie-

rung des Plug-in–Mechanismus (6.1.1) und die Realisierung konkreter Kommunikationsmodule (6.1.2).

6.1.1 Implementierung des *Plug-in*–Mechanismus

Die Grundkonzeption des auf dem Konzept eines bidirektionalen *Pluggable* basierten Plug-in–Mechanismus wurde in in Abschnitt 5.3.1.3 erläutert. In Abbildung 6.1 sind alle Hauptklassen der Implementation und deren Aufrufabhängigkeiten gezeigt.

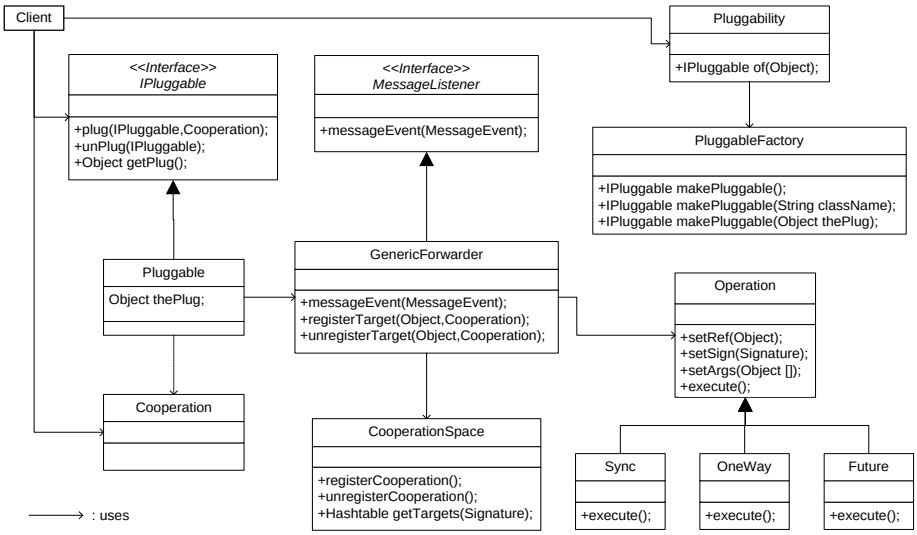


Abbildung 6.1: Klassendiagramm des Plug-in–Mechanismus

Der *Client* ist hierbei ein beliebiges Java–Objekt, von dem die Initiative bei der Verwendung des Plug-in–Mechanismus ausgeht. Er kann zunächst die Klasse *Pluggability* verwenden, um ein beliebiges Java-Objekt dynamisch für den Plug-in–Mechanismus zugänglich zu machen. Mit Hilfe der Klasse *Cooperation* kann er dann ein *Ziel*–Objekt in ein *Quell*–Objekt — beide vom Typ *Pluggable* — mittels der Methode *plug()* hineinstecken, was bedeutet, dass wenn die in dem *Cooperation*–Objekt spezifizierte Methode beim Quellobjekt aufgerufen wird, wird der Aufruf zum Zielobjekt weiter geleitet.

6.1.1.1 Erzeugung von *Pluggable*–Objekten

Die Erzeugung von *Pluggable*–Objekten, die — in Anlehnung an das Entwurfsmuster *Adapter* in [GHJV95] — Objekte anderer Klassen adaptieren, um ihnen die Eigenschaft der dynamischen Komposition zu verleihen, wird durch die Klasse *PluggableFactory* erledigt. Die beiden möglichen Formen des Adaptierens sind der Klassenadapter und der Objektadapter. Die *PluggableFactory* erzeugt die *Pluggable*–Objekte mit Hilfe der Methoden *makePluggable()*, die in mehreren Formen angeboten werden. Entsprechend den beiden Adaptertypen wird entweder eine Objektreferenz oder ein Klassenname übergeben. Die Objektreferenz wird an ein Objekt der Klasse *Pluggable* übergeben, die das Interface *IPluggable* implementiert. Die Klasse wird mit Hilfe der Voyager–Methode

construct() dynamisch erzeugt, die einen Proxy zu dem erzeugten *Pluggable*–Objekt liefert. Die Übergabe des Klassennamens an die zweite Variante der *makePluggable()*–Methode führt auch zu einer dynamischen Konstruktion des entsprechenden Objektes. Die Klasse muss ebenfalls *IPluggable* implementieren. Die dritte *makePluggable()* erzeugt ein leeres Objekt der Klasse *Pluggable*, dem in einem zweiten Konstruktionsschritt die Objektreferenz übergeben wird, die es repräsentiert. Diese Übergabe erfolgt an die *setPlug()* Methode der Administrationsschnittstelle von *Pluggable* — *IPlugAdm*. Die Trennung der Set- und Get-Semantik für die Objektreferenz des zu adaptierenden Objektes hat den Sinn, dass Klassen, die unter der Schnittstelle *IPluggable* auf das Objekt zugreifen, dies nur lesend können.

Die Klasse *Pluggability* bietet durch die statische Methode *of()* eine eingeschränktere Sicht auf *PluggableFactory*. Diese Schnittstelle ist den bei Voyager üblichen statischen *of()*–Methoden nachempfunden, die jeweils eine dynamische Zuweisung der gewünschten Eigenschaften bewirken. Der Aufruf *Pluggability.of(myObject)* wird letztlich an die entsprechende Konstruktionsmethode der *PluggableFactory* delegiert, die zur Konstruktion eines Objekts der Klasse *Pluggable* und der Übergabe der *myObject*–Referenz an dieses Objekt führt.

6.1.1.2 Die Komposition von *Pluggable*–Objekten

Die Komposition oder das Zusammenstecken der Objekte erfolgt durch Aufruf der *plug()*–Methode des *Pluggable*–Objektes, das als Quellobjekt dient. Das *Pluggable*–Objekt, das mit einem entsprechenden *Cooperation*–Objekt als Zielobjekt registriert wird, stellt einen Kooperationspartner in Bezug auf die Ausführung einer bestimmten Methode des Quellobjektes dar, wobei die genaue Ausprägung der Kooperation durch das *Cooperation*–Objekt spezifiziert wird.

Diese Registrierung wird durch den Aufruf der *registerTarget()* Methode beim *GenericForwarder*–Objekt, das vom *Pluggable*–Objekt benutzt wird, ebenfalls (durch die *plug()*–Methode) vorgenommen. Der *GenericForwarder* benutzt ein *CooperationSpace*–Objekt, um die Verbindung zwischen Zielobjekt und Kooperation zu speichern.

Der letzte Schritt einer Komposition ist die Registrierung des *GenericForwarder* als *MessageListener* bei dem Quellobjekt, das bei der Konstruktion des *Pluggable*–Objektes übergeben wurde (s. Abschnitt 6.1.1.1). Diese Registrierung erfolgt bei der *Proxy*–Schnittstelle des durch Voyager erzeugten Objektes. Das bedeutet, dass Objekte, die ohne Wissen über den Plug-in–Mechanismus programmiert worden sind, dann aber mit ihm verwendet werden sollen, nicht mit *new()* erzeugt werden dürfen, sondern mit Hilfe der *construct()* Methoden von Voyager. Ohne die Konstruktion durch Voyager funktioniert der *MessageListener*–Mechanismus nicht, der die Grundlage der Nachrichtenweiterleitung darstellt. Innerhalb des *CooperationSpace* wird die Information des *Cooperation*–Objektes in Hashtabellen gespeichert. Der Schlüsselwert beim Zugriff auf diese Information ist die Quellsignatur, die mit jedem *MessageEvent* geliefert wird.

6.1.1.3 Die Klasse *Cooperation*

Wie bereits in Abschnitt 5.3.1.3 erwähnt, stellt die Klasse *Cooperation* die explizite Modellierung der semantischen Verbindung zwischen Quell- und Zielobjekt dar, wodurch eine beliebige Delegation von Methodenaufrufen an kooperieren-

de Komponenten ermöglicht wird. Abbildung 6.2 zeigt die Schnittstelle dieser Klasse.

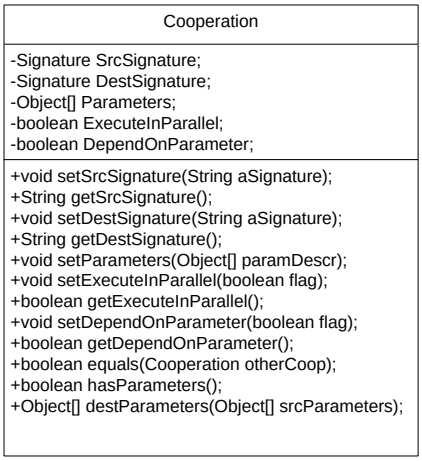


Abbildung 6.2: Schnittstelle der Klasse *Cooperation*

Sie speichert die Quell- und Zielsignatur der beteiligten Methoden, jedoch nicht die dazugehörigen Objektreferenzen. Das bedeutet, dass die Kooperation von allen Objekten der Klassen wiederverwendet werden kann, die die Spezifikation dieser Signaturen erfüllen. Dieses Vorgehen weicht das Konzept der Einkapselung strenger OO-Methodik etwas auf. Die Methodensignaturen der Objekte sind Bestandteil der öffentlichen Schnittstelle eines statisch oder dynamisch getypten Objektes. Diese sind jedoch nicht unabhängig von diesem referenzierbar. Durch die *Cooperation*–Objekte wird die Signatur der auszuführenden Methoden unabhängig von den Objekten gespeichert. Die dynamische Ausführung einer Methode ist jedoch davon abhängig, an welche Objektreferenz dieser Aufruf gebunden wird. Dies ist eine Problematik, die normalerweise von dem Laufzeitsystem der Programmiersprache erledigt wird. Der Plug-in–Mechanismus benutzt eine dynamische Aufrufschnittstelle und ruft explizit eine Objektreferenz. Implementiert das gerufene Objekt die gerufene Methode nicht, findet eine Ausnahmebehandlung statt.

Eine Delegation bedeutet i.a. nicht, dass die aufgerufene Methode ausschließlich bei dem bzw. den Zielobjekt(en) ausgeführt werden soll. Deshalb enthält die Klasse *Cooperation* die Information darüber, ob die Quell- und Zielmethode *parallel* oder *seriell* ausgeführt werden sollen, und ob die Ausführung der Zielmethode von Parametern der Quellmethode abhängig ist oder nicht. Mit der Methode *setParameters()* wird eine Beschreibung der Parameterliste der Zielmethode übergeben. Das Objekt–Feld *Object[]* enthält Information über Position und Typ der Parameter. Damit ist es möglich, die Parameterliste anzupassen, d.h. zu verkleinern. Die Methode *destParameters()* nimmt diese Konvertierung vor.

In Anhang C ist ein konkretes Beispiel für die Benutzung dieses Plug-in–Mechanismus aufgelistet und kommentiert.

6.1.2 Agentenkommunikation

Entscheidend für die konkrete Anwendung von Softwareagenten im Sinne autonomer, interaktiver Einheiten, die trotz eines potenziellen Interessenskonflikts (wie im Falle von Verhandlungen) miteinander *kooperieren* müssen, um überhaupt zu einem Erfolg (z.B. einem Vertragsabschluss) zu kommen, ist die Wahl einer geeigneten Kommunikationssprache. Kooperation auf autonomer Basis bedeutet zumindest, dass sich die Beteiligten auf eine gemeinsame Syntax und Semantik von (Interagenten–) Nachrichten verständigen, die sie *asynchron* — d.h. ohne in ihrem internen Ablauf behindert zu werden — verarbeiten können. Eine weitere Anforderung an die Kommunikationssprache besteht darin, dass ihr Abstraktionsniveau möglichst technologisch neutral sein soll, damit sie von beliebigen Agentenarchitekturen und -systemen eingesetzt werden kann.

Dies wurde frühzeitig von den Anhängern des Agentenparadigmas erkannt und dementsprechend gab es breit angelegte Versuche, eine *generische* Agentenkommunikationssprache zu spezifizieren, die von möglichst vielen Agentensystemen für möglichst viele Anwendungssdomänen genutzt werden kann. Die *Knowledge Query and Manipulation Language* (KQML) ist der bekannteste Vertreter solcher Sprachen, obwohl auch KQML nie bis zur semantischen Ebene vollständig spezifiziert wurde und inzwischen von weiteren Entwicklungen wie der FIPA–ACL [FIP] überholt bzw. abgelöst zu sein scheint (siehe Überblick in [Sin98] sowie weitere Diskussionsbeiträge in [DG00]). Da es bis heute keine wirklich standadisierte ACL (*Agent Communication Language*) gibt, wurde im Rahmen dieser Arbeit KQML gewählt, um das Kommunikationsmodul für die modulare Agentenarchitektur zu realisieren. Im folgenden wird zunächst eine kurze Einführung in die wichtigsten Eigenschaften von KQML gegeben und anschließend die Implementierung des KQML–basierten Kommunikationsmoduls erläutert.

6.1.2.1 KQML

KQML [DAR93, FFMM94, Sin98] modelliert den Austausch von Information und Wissen zwischen Agenten. Die grundlegende Annahme, welche auch den Kern des zu Grunde liegenden semantischen Modells ausmacht, ist dabei, dass jeder Agent eine (eigene) Wissensbasis benutzt und deren Inhalte mit anderen Agenten austauschen will. Die Kommunikation bezieht sich immer auf die Inhalte der Wissensbasis und beinhaltet Aussagen über das Hinzufügen, Löschen und Verändern von Wissensbasisinhalten. Dabei ist es jedoch nicht notwendig, dass die Agenten wirklich eine Wissensbasis besitzen, sie müssen nur den anderen Agenten mit Hilfe ihrer Implementation die Daten, die sie besitzen, als Wissensbasisinhalte, d.h. gemäß einer entsprechenden Syntax, präsentieren können. Man spricht deshalb von einer *virtuellen Wissensbasis.*

Insbesondere ist KQML eine *Sprechakt*–orientierte Sprache [Aus71]. Eine KQML–Nachricht wird dementsprechend als *Performative* bezeichnet. Die Nachrichten beinhalten Aussagen darüber, was der Agent für Annahmen über sich und seine Umgebung macht, und Aussagen über die Wissensbasen anderer Agenten. Darüberhinaus können Nachrichten über die Ziele der Agenten, z.B. Aufforderung zur Verwirklichung eines Zustands, formuliert werden. KQML definiert zunächst eine bestimmte Syntax, die auf ASCII–Zeichenketten basiert (siehe Abbildung 6.3).

```
<performative> ::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)

<expression> ::= <word> | <quotation> | <string> |
                 (<word> {<whitespace> <expression>}*)

<word> ::= <character><character>*

<character> ::= <alphabetic> | <numeric> | <special>

<special> ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
              @ | $ | % | : | . | ! | ?

<quotation> ::= '<expression> | `<comma-expression>

<comma-expression> ::= <word> | <quotation> | <string> | ,<comma-expression>
                       (<word> {<whitespace> <comma-expression>}*)

<string> ::= "<stringchar>*" | #<digit><digit>*"<ascii>*

<stringchar> ::= \<ascii> | <ascii>-\-<double-quote>
```

Abbildung 6.3: KQML–Syntax in BNF

Diese sehr rudimentäre Syntax ist jedoch nur obligatorisch in Bezug auf die äußere Struktur einer KQML–Nachricht, die Inhalte selbst können in KQML–Syntax, aber auch in jeder anderen Sprache, deren Name im Feld `:language` anzugeben ist, verfasst sein. Außerdem hängt die Interpretation des Inhalts einer Nachricht von der jeweiligen Anwendungsdomäne, die im Feld `:ontology` zu spezifizieren ist, ab. Dadurch ist eine hohe Generizität gewährleistet, allerdings wird auch das Problem der Semantik weitgehend in die jeweilige Sprache des Inhalts bzw. in die Ontologie verlagert.

KQML selbst spezifiziert nur die Semantik der Hauptschlüsselwörter sowie eines relativ kleinen Satzes an Performativen. Die Schlüsselwörter und deren Bedeutung sind in Tabelle 6.1 zusammengefasst.

Tabelle 6.2 zeigt einen Auszug der in KQML spezifizierten Performativen. Daran ist erkennbar, dass fast alle Performativen zum Austausch von Wissen zwischen den Agenten beabsichtigt sind. Die letzte in dieser Liste (`achieve`) dient jedoch dazu, den Empfänger der Nachricht zur Erfüllung eines Zieles aufzufordern. Der eigentliche Unterschied einer solchen Sprache und der operativen

Schlüsselwort	Bedeutung
`:content`	Inhalt der Nachricht
`:force`	Widerrufbarkeit der Nachricht
`:in-reply-to`	Vorherige Nachricht, auf die hiermit geantwortet wird
`:language`	Name der Sprache, in der der Inhalt verfasst ist
`:ontology`	Name der Ontologie (z.B. *Geschäftsverhandlung*) des Inhalts
`:receiver`	Empfänger der Nachricht
`:reply-with`	Sender erwartet eine Antwort unter Angabe dieses Zeichens
`:sender`	Sender der Nachricht

Tabelle 6.1: KQML–Schlüsselwörter

Performative	Bedeutung
`tell`	Mitteilung vom Sender
`deny`	Die eingebettete Performative ist für den Sender nicht (länger) gültig
`untell`	Äquivalent zu einem `deny` von `tell`
`evaluate`	Sender bittet um Evaluierung des Inhalts
`reply`	Antwort auf eine Frage
`ask-if`	Ob der Inhalt in der VKB des Empfängers ist
`ask-about`	Erbittet alle relevanten Sätze zum Inhalt (Thema)
`ask-one`	Erbittet eine Antwort zu der Frage
`ask-all`	Erbittet alle Antworten zu der Frage
`sorry`	Nicht in der Lage, eine Antwort zu liefern
`achieve`	Sender fordert Empfänger auf, etwas wahr zu machen

Tabelle 6.2: KQML–Performativen (Auszug)

Dienstsicht (s. Abschnitt 2.1.2) besteht darin, dass der oder die Empfänger einer Nachricht nicht dazu gezwungen werden, darauf zu reagieren. Allerdings wird bei all diesen Performativen generell angenommen, dass alle Beteiligten ein *kooperatives* Verhalten zeigen, denn sobald dies nicht gilt, macht es für einen Agenten kaum Sinn, sein Wissen preiszugeben. Für im allgemeinen nicht–kooperative Interaktionsformen wie die hier angestrebten automatischen Verhandlungen für den Bereich E–Commerce sind diese Performativen (mit ihrer beabsichtigten Semantik) wenig hilfreich, sondern andere, die die Semantik von Verhandlungsnachrichten (wie z.B. *(counter-)offer*, *accept*, *reject* etc.) repräsentieren, erforderlich. Allerdings gibt es bis heute keine offizielle Spezifikation solcher verhandlungsbezogener Performativen[1]. Das folgende Beispiel veranschaulicht, wie KQML–Nachrichten einer bestimmten Domäne aussehen können.

Beispiel In diesem Beispiel versucht ein Agent, Fluginformation zu bekommen, die unterspezifiziert im `:content` Teil übergeben wird. Die Parameter `:language` und `:ontology` beschreiben Sprache bzw. Ontologie des Inhalts.

```
(ask-all
    :language anACL
    :ontology Travel
    :reply-with travelAgent007-01
    :content (
          :carrier  JAL *
          :flight   405 *
          :departure NRT *
          :arrival  CDG *
          :deptTime "03/21/97 PM" *
          :expirationTime 300)
)
```

[1] Obwohl der Bedarf dafür bereits bei der ersten offziellen Version von KQML erkannt wurde [FMFM94].

Die entsprechende Antwort benutzt `tell` als Performative und beinhaltet nun die komplette Fluginformation.

```
(reply
    :language anACL
    :ontology Travel
    :in-reply-to travelAgent007-01
    :content (
          :carrier JAL
          :flight 405
          :departure "(NRT) Tokyo, Japan (Narita)"
          :arrival "(CDG) Paris, France (Charles De Gaulle)"
          :deptTime "12:15PM 03/21/97")
)
```

6.1.2.2 Implementierung des Kommunikationsmoduls

Das Kommunikationsmodul wurde mit Hilfe des beschriebenen Plug-in–Mechanismus und der JKQML–Bibliothek von IBM [IBM] implementiert. JKQML stellt eine objektorientierte Implementation von KQML–Nachrichten in Java zur Verfügung, die es erlaubt, beliebige Laufzeitobjekte als Nachrichteninhalt über ein Netzwerk zu übertragen[2]. Die Implementation des Kommunikationsmoduls beinhaltet zum einen die Integration der Funktionen von JKQML und zum anderen die Aufbereitung der Schnittstelle zu anderen Modulen, insbesondere zum Protokollmodul (siehe Abschnitt 5.3.2.2), zu den die Nachrichteninhalte weiter zu leiten sind.

Die Integration von JKQML Abbildung 6.4 illustriert die Hauptklassen zur Integration der JKQML–Funktionalität in das modulare Agenten–Framework.

Zum Versenden und Empfangen von KQML–Nachrichten mit Hilfe von JKQML werden für das K–Modul hauptsächlich folgende Klassen/Schnittstellen aus der JKQML–Bibliothek benötigt:

- Eine Instanz der Klasse *KQMLManager*. Diese ist dafür zuständig, den ganzen Nachrichtentransport zu organisieren und Nachrichten zu versenden.
- Eine Instanz einer Klasse, die das Interface *ContentInterpreter* implementiert. Sie stellt eine Methode zum Empfangen und Beantworten von KQML–Nachrichten bereit.
- Eine Instanz einer Klasse, die das Interface *NamingService* implementiert. Sie ist dafür zuständig, Namen von Agenten auf URL's abzubilden.

[2]Im Rahmen dieser Arbeit wurde auch eine eigene Implementation von KQML in Java realisiert — die wie die meisten KQML–Implementationen nur ASCII–basierte Nachrichteninhalte zulässt [Chm98] — um eine flexiblere Erweiterung bzgl. der Performativen um weitere, verhandlungsbezogene zu ermöglichen. Jedoch stellte sich heraus, dass die direkte Übertragung von Vertragsobjekten eine wesentlich einfachere Lösung darstellt und zudem keine zusätzlichen Performativen benötigt werden, wenn die spezifische Semantik von Verhandlungsnachrichten in das Vertragsobjekt selbst eingebettet wird.

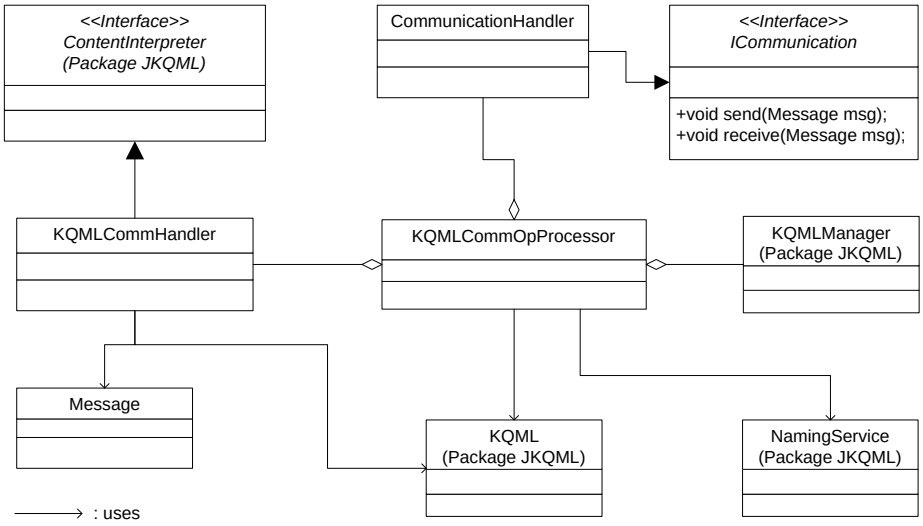


Abbildung 6.4: Klassendiagramm des Kommunikationsmoduls

- Instanzen der Klasse *KQML*. Sie stellen die eigentlichen KQML–Nachrichten dar.

Die Einbettung in das K–Modul erfolgt folgendermassen:

- Die Klasse *KQMLCommOpProcessor* implementiert die Fähigkeit, sowohl die internen *Message*–Objekte, als auch die ausgehenden *KQML*–Nachrichten zu versenden. Sie enthält einen *KQMLManager*.
- Die Klasse *KQMLCommHandler* stellt einen *ContentInterpreter* zur Verfügung, der *KQML*–Nachrichten empfängt und auf interne *Message*–Objekte abbildet.
- Durch die Klasse *CommunicationHandler* wird die Verbindung zu einem anderen Modul hergestellt. Sie implementiert das Interface *ICommunication*, das aus Sicht der anderen Module die Schnittstelle des K–Moduls darstellt.

Die Weiterleitung von Nachrichten zu anderen Modulen Die Weiterleitung der Nachrichten bedeutet, dass der Inhalt der Nachrichten entgegengenommen werden muss. Die Bedeutung des Inhaltes ist für das K-Modul transparent. Er wird einfach der KQML–Nachricht entnommen und weitergeleitet. Die Weiterleitung von Nachrichten ist nicht beschränkt auf die Verbindung zwischen Kommunikations- und Protokollmodul, sondern ein allgemeines Kennzeichen aller Module des Agenten. Es ist also zweckmäßig für dieses Designziel, einen allgemeinen Mechanismus zu entwerfen, der von allen Modulen wiederverwendet werden kann.

Die Weiterleitung wird deshalb intern an ein Objekt der Klasse *CommunicationHandler* delegiert. Es benutzt die Klasse *Message*, um beliebige Inhalte zwischen Modulen zu transportieren. Der Nachrichtenfluss ist gerichtet. Das

bedeutet, dass zwischen dem Senden und dem Empfangen von Nachrichten unterschieden werden kann. Dabei bedeutet:

- **Senden einer Nachricht** Die Nachricht wird an ein anderes Modul gesendet.
- **Empfangen einer Nachricht** Die Nachricht wird von einem anderen Modul empfangen.

Diese Entkopplung von Sende- und Empfangssemantik ermöglicht eine gerichtete dynamische Verbindung zwischen je zwei Modulen. Für die Implementation des Kommunikationsmoduls bedeutet dies, dass die Weiterleitung des Inhalts einer KQML–Nachricht nur durch Objekte desselben Moduls angestossen werden darf (da Instanzen von *Message* nur intern verwendet werden). Das Senden einer Nachricht über das K–Moduls geschieht durch die dynamische Kopplung von Methodenaufrufen zwischen zwei Modulen.

Die Methode *send()* des Interfaces *ICommunication* ist dafür zuständig, Nachrichten an ein anderes Modul zu senden. Sie stellt also den Modulausgang für *Message*–Objekte dar. Die Methode *receive()* nimmt eine Nachricht entgegen, repräsentiert also den Eingang des Moduls. Die Verbindung der Module erfolgt unter Zuhilfenahme des Plug-in–Mechanismus. Dies wiederum bedeutet, dass es niemals eine direkte Referenzierung der Objekte untereinander gibt, die das *ICommunication*–Interface implementieren. Eine Kommunikation wird dynamisch hergestellt, indem die *send()*–Methode des sendenden Objektes mit der *receive()*–Methode des empfangenden Objektes verbunden wird. Die *Cooperation*–Objekte des Plug-in–Mechanismus schränken die Benutzung der Objektreferenzen auf die an den Kooperationen beteiligten Methoden ein.

Um asynchrones Messaging zu ermöglichen, müssen die Nachrichten bestimmte Eigenschaften haben. Da das Senden einer Nachricht in diesem Fall von seiner Beantwortung entkoppelt ist, muss die Information, auf welche Nachricht sich die Antwort bezieht, zusätzlich gespeichert werden. Diese Information wird auch als *Asynchronous Completion Token* bezeichnet [HSP96]. Die einfachste Lösung besteht darin, diese Information mit der Nachricht zu versenden. Deshalb wird bei der Konstruktion der *Message*–Objekte ein solches Token übergeben. Auch KQML kennt solche Referenzen, z.B. den `:in-reply-to` Parameter (siehe Tabelle 6.1).

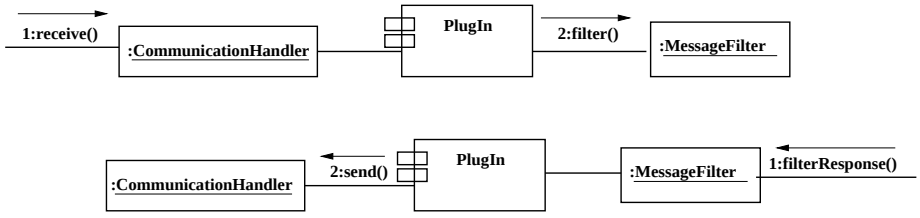


Abbildung 6.5: Nachrichtenweiterleitung durch den Plug-in–Mechanismus

Die Verbindung des K-Moduls mit einem anderen Modul wird in Abbildung 6.5 illustriert, wobei *MessageFilter* den Teil des anderen Moduls, der für die Bearbeitung von ein- und ausgehenden Nachrichten zuständig ist, repräsentiert.

Um den Inhalt einer von dem K-Modul empfangenen Nachricht weiter zu leiten, wird innerhalb dieses Moduls die Methode *receive()* beim *CommunicationHandler* aufgerufen, wodurch automatisch ein Aufruf der Methode *filter()* mit dem gleichen Argument beim *MessageFilter* ausgelöst wird. Um eine Nachricht über das K-Modul zu versenden, wird beim *MessageFilter* die Methode *filterResponse()* aufgerufen, wodurch die Methode *send()* beim *CommunicationHandler* ausgelöst wird.

6.2 Realisierung von Verhandlungsprotokollen

In diesem Abschnitt wird die Realisierung von Protokollen zur Steuerung automatisierter Verhandlungen beschrieben (siehe auch [TLGL99, Lan99, Gro00]). Wie die in Abschnitt 5.2 vorgenommene Analyse automatisierter Verhandlungsformen zeigte, spielen Verhandlungsprotokolle eine ganz entscheidende Rolle bei der Bestrebung, *vollständig* automatisierte Verhandlungen — d.h. solche, an den ausschließlich Softwarekomponenten beteiligt sind (bzw. sein können) — zu verwirklichen. Um dieses Ziel zu erreichen, müssen *formalisierte* Protokolle nicht nur in ihrer Semantik *entworfen*, sondern auch zu einer maschinell verarbeitbaren Form *generiert* und letztlich *ausgeführt* werden. Dementsprechend wird in den folgenden Unterabschnitten auf Systemmechanismen zur Unterstützung der Formalisierung, des Entwurfs, der Generierung und der Ausführung von Verhandlungsprotokollen eingegangen.

6.2.1 Eine Petrinetz–basierte Spezifikationssprache

Zur Beschreibung von Verhandlungsprotokollen wird zunächst ein Formalismus benötigt, der genügend mächtig ist, um die Anforderungen an solche Protokolle (siehe Abschnitt 5.2.2.1) zu erfüllen.

6.2.1.1 Beschreibungsmethoden für Verhandlungsprotokolle

In der Literatur finden sich unterschiedliche Konzepte, die zur Beschreibung von Verhandlungsprotokollen verwendet werden können. Diese lassen sich grob in zwei Kategorien einteilen: Bei der *nachrichtenorientierten* Sicht wird ein Verhandlungsprotokoll als ein Regelsatz für den geordneten Austausch von Nachrichten betrachtet, während es bei der *prozessorientierten* Sicht als eine Prozessbeschreibung aufgefasst wird, die insbesondere durch Petrinetze formalisiert werden kann.

Nachrichtenorientierte Sprachen

Nachrichtenorientierte Sprachen basieren auf dem Prinzip, die Interaktion zwischen Kommunikationspartnern ausschließlich durch die dabei ausgetauschten Nachrichten zu modellieren. In einigen Systemen wird anstelle von Nachrichten auch von Aktionen gesprochen [Mül96].

Zu nachrichtenorientierten Beschreibungssprachen gehört eine feste Menge an elementaren, typisierten Nachrichten, den Nachrichtenprimitiva, wie z.B. QUIT oder REJECT. Insofern ist auch KQML (vgl. Abschnitt 6.1.2.1) mit ihren Performativen eine nachrichtenorientierte Sprache. Einige dieser Nachrichten können parametrisiert sein. Mögliche Parameter sind dann zum Beispiel

Angebote. Zusätzlich gibt es eine Menge von Regeln, die festlegt, wer auf welche Aktionen wie reagieren kann [ZS96, S. 250ff]. Einige Verhandlungssysteme ordnen die Regeln explizit bestimmten Rollen zu, indem diese in der Definition des Verhandlungsprotokolls auftauchen. Beispielsweise wird in [Mül96] eine Verhandlung wie folgt definiert.

Sei D eine Verhandlungsdomäne, dann ist eine Verhandlung durch folgendes Tupel spezifiziert:

$NEG = (A, R, \rho, N, U, P, S)$ wobei

- $A = \{a_1, ..., a_k\}, k \geq 2$, eine Menge von Agenten ist.
- $R = \{r_1, ..., r_l\}, l \leq k$, eine Menge von Rollen ist.
- ρ eine Funktion ist, die Agenten Rollen zuweist: $\rho(a) = r$ for $a \in A, r \in R$.
- $\emptyset \neq N \subseteq D$ die Verhandlungsmenge ist.
- $U = \{u_1, ..., u_k\}$, mit $u_i : N \mapsto \mathbb{R}$ die Nutzenfunktion für Agent a_i ist.
- $P = (K, \pi : R \times K \mapsto 2^K)$ ein Verhandlungsprotokoll ist, wobei K eine endliche Menge von Kommunikationsprimitiva ist.
- $S = \{\sigma_i : P \times R \times K \times 2^D \times U \mapsto K \times N | 1 \leq i \leq k\}$ eine Menge von Verhandlungsstrategien ist. (Die Ausgabe von σ_i ist eine Reaktion auf die eingehende Nachricht und eine modifizierte Verhandlungsmenge $N' \subseteq N$.)

In diesem Sinne ist ein Verhandlungsprotokoll also eine Funktion, die für einen Teilnehmer einer bestimmten Rolle festlegt, mit welchen Kommunikationsprimitiven er auf eine vorgegebene Nachricht reagieren kann. In Abbildung 6.6 ist ein Beispiel für ein solches Verhandlungsprotokoll angegeben. Es beschreibt eine Verhandlung zwischen zwei Teilnehmern. Der eine handelt in der Rolle des *leader*, der andere als *follower*. Die Kommunikationsprimitiva sind "PROPOSE", "ACCEPT", "MODIFY", "CONFIRM", "start" und "done", wobei die beiden letzten jedoch nur der internen Flusskontrolle dienen und nicht kommuniziert werden.

```
R = {leader, follower}, K={PROPOSE, ACCEPT, MODIFY, CONFIRM}
p(leader, start) = {PROPOSE(p)}
p(leader, ACCEPT(p)) = p(follower, ACCEPT(p)) = {CONFIRM(p)}
p(leader, MODIFY(p,p')) = p(follower, MODIFY(p,p')) =
{ACCEPT(p'), MODIFY(p',p'')}
p(follower, PROPOSE(p)) = {ACCEPT(p), MODIFY(p,p')}
p(leader, CONFIRM(p)) = p(follower, CONFIRM(p))={done}
```

Abbildung 6.6: Beispiel eines nachrichtenorientierten Protokolls

Der *leader* beginnt die Verhandlung mit der Nachricht "PROPOSE" und einem von ihm gewählten Vorschlag als Parameter. Der andere Teilnehmer darf darauf mit einem "ACCEPT" reagieren oder den Vorschlag verändern und den Vorschlag sowie die Veränderung mit "MODIFY" zurücksenden. Die beiden Verhandlungsteilnehmer schicken sich gegenseitig solange "MODIFY"–Nachrichten, bis einer von beiden mit "ACCEPT" reagiert. Darauf muss der

jeweils andere Teilnehmer mit einem "CONFIRM" reagieren, worauf die Verhandlung beendet ist.

Aus der Beschreibung dieses Protokolls wird deutlich, dass ein solches *statisches* Modell nicht in der Lage ist, die dynamischen Aspekte eines Verhandlungsprozesses, insbesondere die zeitbehafteten, zu erfassen. Deshalb werden nun Petrinetze als ein Mittel zur Prozessbeschreibung vorgestellt.

Petrinetze als Basis zur Beschreibung von Verhandlungsprotokollen

Die in Abschnitt 5.2.3 aufgeführten strukturellen Bestandteile von Verhandlungen entsprechen denen von höheren Petrinetzen. Es bietet sich daher an, Petrinetze als Beschreibungsmittel für die strukturelle Komponente von Verhandlungsprotokollen zu verwenden. Unter einem Petrinetz versteht man ein abstraktes Modell eines diskreten Informationsflusses, das auf einem exakten mathematischen Formalismus beruht [Jen97]. Das Konzept wurde in den sechziger Jahren von C. A. Petri entwickelt, um eine natürliche und mächtige Repräsentationsform für den Informations- und Kontrollfluß in Systemen mit asynchronen und nebenläufigen Prozessen zu finden, zwischen deren Ereignissen Präzedenzrelationen bestehen.

Petrinetze besitzen eine graphische Repräsentation, die analog zu den statischen Modellen zunächst die statische Struktur des Petrinetzes wiedergibt. Darüber hinaus besitzt ein Petrinetz jedoch dynamische Eigenschaften, die sich in seiner Ausführung ergeben und die sich direkt innerhalb des Modells ausdrücken lassen. Ein Petrinetz ist daher ein integriertes Modell zur Beschreibung und Abwicklung nebenläufiger Prozesse. Die auf einem formalen Modell basierte Dynamik der Petrinetze ist, im Unterschied zu Datenflußdiagrammen oder Netzplänen, durch formale Methoden simulierbar und analysierbar, so dass Petrinetze eine anschauliche graphische Repräsentation mit einer formalen Analysierbarkeit verbinden.

Durch ihre Integration von Struktur und Verhalten sind Petrinetze als Basis für eine anwendungsnahe Modellierung verteilter Vorgänge geeignet, was sich nicht zuletzt auch durch ihren Einsatz in einer Vielzahl von Projekten im Bereich der verteilten Systeme erwiesen hat [Jen97, Band 3]. Im Rahmen dieser Arbeit werden daher Petrinetze als formales Modell für komplexe Anwendungsvorgänge verwendet.

Die Petrinetztheorie stellt ein sehr mächtiges Modell zur Verfügung und ist aufgrund ihrer Dynamik zur Beschreibung komplexer Verhandlungsprotokolle geeignet. Daher wurde im Rahmen dieser Arbeit ein auf gefärbten Petrinetzen basiertes Modell zur Beschreibung und Ausführung solcher Protokolle gewählt.

Zur Zeit gibt es einige Systeme, die zur Beschreibung von gefärbten Petrinetzen eingesetzt werden. Sehr bekannt ist beispielsweise Design/CPN, das an der Universität Aarhus unter der Leitung von Prof. Kurt Jensen entwickelt worden ist (siehe z.B. [Jen97, Band 3]). An der Arbeitsgruppe Theoretische Grundlagen der Informatik der Universität Hamburg wurde dieses System um objektorientierte Konstrukte erweitert. Dieses und die bekannten anderen Systeme haben jedoch in Bezug auf den Entwurf von Verhandlungsprotokollen auch gewisse Nachteile:

- Es sind im wesentlichen theoretisch ausgerichtete Systeme, die vor allem zur Simulation und zum Nachweis von bestimmten Schaltverhalten ent-

wickelt wurden.

- Sie enthalten kein konkretes Konstrukt zur Modellierung von Rollen.
- Die verwendete Sprache *designML*, die auf der funktionalen Sprache *ML* aufbaut und für Ungeübte vergleichsweise schwer zu erlernen ist.

Deshalb wurde im Rahmen dieser Arbeit eine Sprache zur Spezifikation von Verhandlungsprotokollen entwickelt, die zwar sowohl syntaktisch als auch semantisch auf gefärbten Petrinetzen basiert, jedoch für die speziellen Anforderungen von Verhandlungsprotokollen entworfen ist und zusätzliche Konstrukte zur Formulierung von Metainformationen bezüglich einer zu steuernden Verhandlung enthält. Diese Spezifikationssprache trägt den Namen OOPAMELA[3] und ist eine Weiterentwicklung[4] der am Arbeitsbereich Verteilte Systeme des Fachbereiches Informatik der Universität Hamburg entstandenen Sprache PAMELA [MJ97].

6.2.1.2 Eigenschaften und Aufbau von OOPAMELA–Protokollen

OOPAMELA als eine formale Sprache, die speziell für die Spezifikation von Verhandlungsprotokollen konzipiert wurde, zeichnet sich durch folgende Eigenschaften aus:

- Teilnehmer und Rollen: OOPAMELA erlaubt die explitze Modellierung von Teilnehmern und deren Rollen.
- Verbindungsstrukturen: Die möglichen Interaktionen bzw. Informationswege zwischen den Teilnehmern werden durch Verbindungsstrukturen (Stellen, Transitionen, Kanten) festgelegt.
- Komplexe Daten: Daten, die zwischen den Teilnehmern ausgestauscht werden, sind durch gefärbte (typisierte) Marken repräsentiert. Dadurch lassen sich strukturierte Verträge und deren Änderungen modellieren.
- Metabeschreibungen: Verhandlungsspezifische Informationen, wie z.B. wie Rollen zu instanziieren und der Ein- und Austritt von Teilnehmern während der Verhandlung abzuwickeln sind, können in Form strukturierter Metabeschreibungen angegeben werden.
- Zustände: Aus dem impliziten Zustand der Verhandlung, der jeweils durch die aktuelle Markierung des Netzes gegeben ist, und zusätzlichen Verwaltungsinformationen lassen sich explizite Verhandlungszustände ableiten, die direkt von den Teilnehmern genutzt werden können.
- Kommunikation mit externen Komponenten: OOPAMELA erlaubt eine aktive Interaktion zwischen dem Schaltverhalten des Petrinetzes und der Funktionalität externer Komponenten über wohl definierte Schnittstellen. Dadurch können u.a. Marken, die Vertragsobjekte repräsentieren, von den Teilnehmern modifiziert werden.

[3] Object–Oriented Petrinet–based Activity Management Execution Language

[4] Hauptsächlich wurden Erweiterungen vorgenommen, um verhandlungsspezifische Semantik (wie z.B. Teilnehmerrollen, Metabeschreibungen und Subnetze zur Validierung von Teilnehmerverhalten) beschreiben zu können.

- Subnetze als semantisches Steckmodul: Darüber hinaus kann beim Schalten einer Transition ein beliebig komplexes Subnetz, das bspw. die Vertragsvalidierung durchführt, aufgerufen werden.

Ein OOPAMELA–Protokoll ist systematisch aus mehreren Teilen zusammengesetzt. Eine graphische Darstellung ist in Abbildung 6.7 zu sehen.

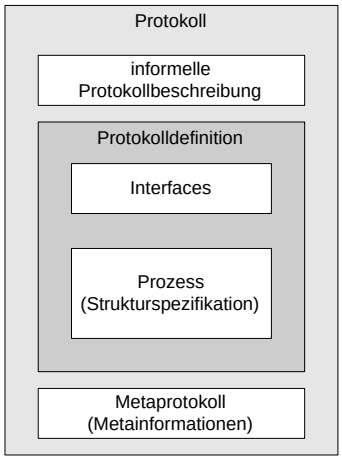


Abbildung 6.7: Aufbau von OOPAMELA–Protokollen

Der erste Teil ist eine informelle Beschreibung des Protokolls. Der zweite beschreibt die eigentliche Protokolldefinition. Den dritten Teil bildet das so genannte Metaprotokoll, das zusätzliche Verwaltungs- und Instanziierungsinformationen zu dem beschriebenen Prozess enthält. Die Protokolldefinition besteht selbst wieder aus zwei Bestandteilen: einer Interfacedefinition, in der alle Schnittstellen der beteiligten Rollen importiert oder deklariert werden, und einer Prozessdefinition, in der der strukturelle Aufbau eines Protokolls beschrieben wird. Die Prozessdefinition enthält eine Liste von Stellendeklarationen sowie die Beschreibung der Transitionen.

Ein komplettes Beispiel eines Auktionsprotokolls in OOPAMELA findet sich in Anhang D.2. Im folgenden werden die Bestandteile von OOPAMELA in ihrer Syntax beschrieben:

```
protocol        ::=    header body
header          ::=    description
description     ::=    // free description text
```

Ein Protokoll beginnt mit einer Beschreibung. Dieser werden die beiden Zeichen "//" vorangestellt. Diese Beschreibung ist informell und kann aus einem beliebigen Text bestehen. Ein Broker (siehe Abschnitt 5.3.2) kann diese Beschreibung verwenden, um dem Benutzer die Protokollauswahl zu erleichtern. Verhandlungssysteme können das Format standardisieren, um eine automatische Auswertung oder eine Kategorisierung vorzunehmen.

```
Body            ::= protocol ''protocolname''
                      {
                         mainpart
```

```
                    }
                       metaprotocol
                    {
                       metapart
                    }
Mainpart        ::= imports interfaces process
```

Auf die Beschreibung folgt das Schlüsselwort *protocol* und ein Protokollname. Dieser muss innerhalb des Verhandlungssystems eindeutig sein. Der folgende Protokollteil befindet sich innerhalb geschweifter Klammern.

6.2.1.3 Protokolldefinition

Eine Protokolldefinition besteht aus drei Teilen, die jeweils voneinander abhängig sind:

1. Einer Importdefinition, mit der zusätzliche Referenznetze (siehe Abschnitt 6.2.3.2) geladen werden können.
2. Einer Interfacedefinition, in der alle benutzen Interfaces deklariert werden.
3. Einer Prozessdefinition, in welcher der strukturelle Aufbau eines Protokolls beschrieben wird. In der Prozessdefinition werden die aus dem letzten Teil bekannten Schnittstellen verwendet.

Imports

Die Importdefinition besitzt folgende Syntax:

```
imports         ::= import < ''netname'' >
```

Dieses Statement wird für das Importieren zusätzlicher Netze benutzt. Diese Netze können dazu benutzt werden, die Kommunikation mit den Verhandlungsteilnehmern zu steuern. Eine ausführliche Beschreibung ist in Abschnitt 6.2.3.2 zu finden.

Interfaces

Die Interfacedefinition besitzt folgende Syntax:

```
interfaces      ::= interfaces { [interfacedecl]* }
```

Jedes in Zukunft vom Protokoll benutzte Interface muss hier deklariert werden:

```
interfacedecl   ::= interface ''interfacename''
{
   idl-definition
};
```

Diese Deklaration erlaubt es, an dieser Stelle ein CORBA–konformes Interface zu definieren. Dieses Interface ist dem Protokollsystem in Zukunft ebenfalls bekannt.

Prozess

Die Prozessdefinition ist vom Konzept her mit der in OOPAMELA identisch. Sie enthält Stellen und Transitionenbeschreibungen. Unterschiede gibt es jedoch in der Beschreibung der Stellen und der Transitionen sowie in den möglichen Typen.

```
process        ::= places transitions
```

Stellen (places)

Die Syntax für die Stellenbeschreibung lautet wie folgt:

```
places          ::= places { [placedecl]* }
placedecl       ::= type ''placename'' [relation];
```

Die Stellendeklaration erfolgt nach dem Schlüsselwort *places*. Jede Stelle wird durch eine Typangabe und einem prozeßweit eindeutigen Stellennamen beschrieben. Der Typ entspricht einem CORBA–Typ und kann daher ein atomarer Typ wie integer, boolean, String, etc. oder ein Interfacename sein. Zusätzlich zu den bekannten atomaren Typen gibt es die Typen anonymous, der den Typ *void* ersetzt.

```
type    ::= int | double | float | string |
             interfacename | anonymous
```

Jede Stelle kann wie in OOPAMELA nur Marken enthalten, die dem Typ der Stelle entsprechen. Dabei ist zu beachten, dass für die Schnittstellen die aus CORBA bekannte Subtyppolymorphie gilt: Jeder Typ kann anstelle seiner Obertypen benutzt werden.

Transitionen

Nach der Aufzählung der Stellen erfolgt die Beschreibung der Transitionen. In OOPAMELA besteht eine Transitionsdefinition aus der Kopfzeile und den darauf folgenden Verbindungen zu den Stellen sowie den Aktionen, die ausgeführt werden, wenn die Transition schaltet. Die Kopfzeile hat die Syntax:

```
transitiondecl  ::= trans ''transitionname''
                    [roledecl | relation]
                    {
                        transitionbody
                    }[;]
```

Der Transitionsname muss eindeutig sein. Die Position der Transition innerhalb des Netzes ergibt sich aus der durch die *get-* und *put-* Anweisungen gebildeten Struktur. Auf den Transitionsnamen folgt ein optionaler Bestandteil, der die Anbindung an Rollen beschreibt.

```
roledecl        ::= uses ''interfacename''
[{1|variable}] [alias rolename]}* with ''referencenetname''
```

Zuerst wird der erste Teil beschrieben:

```
uses interfacename[[variable]][alias rolename]
```

Der Interfacename besagt, welches Interface die Rolle haben soll. Über das Interface kommuniziert die Transition mit dem Exemplar der Rolle. Das Interface muss, wie oben beschrieben, bekannt sein. Wird die Indexangabe [variable] weggelassen, so kann von dieser Rolle und damit von der Transition nur ein Exemplar erzeugt werden. Wird die Indizierung angegeben, so wird die Variable zur Laufzeit mit der tatsächlichen Anzahl der besetzten Exemplare belegt.

Der optionale Rollenname wird als Alias benutzt, falls der Interfacename als Typ für mehr als eine Rolle dient. Der Interface- oder der Aliasname muß eindeutig sein. Abbildung 6.8 zeigt die entsprechende Struktur, in der jedes Transitionsexemplar an jeweils ein eigenes Rollenexemplar der Rolle B angebunden ist. Die OOPAMELA–Beschreibung dafür lautet:

```
trans transX uses RolleA[M], RolleB[M]
```

In diesem Fall wird, um die Eindeutigkeit der Rollennamen zu bewahren, intern automatisch ein Alias vergeben und den Rollenexemplaren zugeordnet.

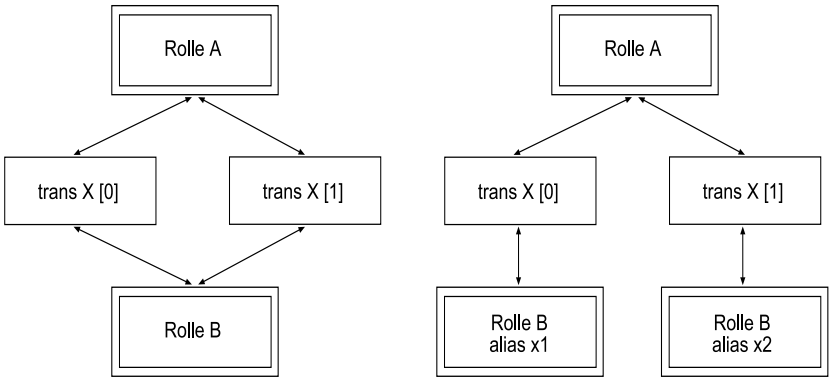


Abbildung 6.8: Anbindung einer Transition an mehrere Rollen

Der zweite Teil:

```
with ''referencenetname''
```

gibt das Referenznetz an, das für den Aufruf benutzt werden soll. In diesem Referenznetz (siehe Abschnitt 6.2.3.2) ist es möglich eine explizite Fehlerbehandlung für den Aufruf externer Verhandlungsteilnehmer zu spezifizieren.

Im folgenden wird der Transitionskörper beschrieben. Er besteht aus den *get*- und *put*- Anweisungen, die die Struktur des Netzes bilden, dem Aktionsteil, der festlegt, welche Funktion die Transition beim Schalten ausführt, und dem Timeoutfehlerverhalten. Die Syntax und Grammatik ist OOPAMELA möglichst ähnlich.

```
transitionbody  ::= {getdecl [getdecl]* | putdecl}
                    [putdecl]* [actions] [timeout]
```

Jede Transition muss mindestens eine *get*- oder eine *put*- Anweisung enthalten. Darauf folgen die optionalen Action– und Timeout–Anweisungen.

```
getdecl ::= get [variable from placename [[relationcond]]
            [if comparedecl];
putdecl ::= put [variable to placename [[relationcond]];
```

Beide Anweisungen entsprechen im wesentlichen ihren OOPAMELA–Vorbildern.

Metaprotokollaufbau

In dem Metaprotokoll werden die Informationen abgelegt, die das Verhandlungssystem benötigt, um ein konkretes Exemplar des mit dem Verhandlungsprotokoll beschriebenen Verhandlungstyps zu erzeugen und verwalten zu können. Das Protokoll dient dem Zweck, eine komplexe Kommunikationsbeziehung zwischen Kommunikationspartnern zu beschreiben und im Endeffekt auch darin, ein System bei der Verwaltung des Kommunikationsprozesses zu unterstützen. Daher ist es nicht verwunderlich, dass in einem Metaprotokoll hauptsächlich Informationen über die Zusammensetzung und das dynamische Verhalten der Kommunikationspartner festgelegt sind.

```
metaprotocol    ::= metaprotocol
                    {
                     [metaprotocolentries]*
                     [metarolespec]*
                     [metatransspec]*
                    }
```

Ein Metaprotokoll besteht aus Listen von Einträgen. Zunächst werden die globalen Einträge aufgezählt. Diese sind, solange keine speziellen Angaben in den später erläuterten Listen gemacht werden, für die gesamte Verhandlung und für alle Rollen gültig. Weitere Listen enthalten Informationen über Rollen und an Rollen angeschlossene Transitionen. Für alle Listeneinträge gelten fünf Aussagen:

- Alle Einträge sind optional.
- Für jeden mehrfach in einer Liste vorkommenden Eintrag gilt der letzte.
- Jeder Eintrag ist einschränkend.
- Parameterbehaftete Einträge werden groß geschrieben. Eintragsnamen und Parameter werden wie in der Sprache Smalltalk durch Doppelpunkte getrennt.
- Parameterlose Einträge werden klein geschrieben und können genauer spezifiziert werden, indem ein Block in geschweiften Klammern folgt. Im allgemeinen können sie ersetzt werden durch parameterbehaftete Einträge mit dem Parametertyp *bool*.

Diese Aussagen haben folgende Konsequenzen: Zunächst kann jeder Eintrag im Metaprotokoll gar nicht, einmal oder mehrfach vorkommen. Dadurch, dass jeder Eintrag eine einschränkende Eigenschaft besitzt, ist es möglich, in späteren Sprachversionen neue Eigenschaften hinzuzufügen, ohne dass alte Protokolle ein neues Verhalten bekommen.

Globale Einträge

Die Einträge der globalen Liste werden hintereinander aufgezählt.

```
metaprotocolentries ::= {MinimumParties: anyint;}|
                        {MaximumParties: anyint;}|
                        {GlobalTimeout: anyint;} |
                        {DefaultTimeout: anyint;}|
                        {constantNoOfParties;}
```

- *Minimum-* und *MaximumParties* begrenzen die Anzahl der zur Laufzeit möglichen Teilnehmer eines Protokollexemplares. Alle an Rollen gebundene Teilnehmer werden mitgezählt.

- *GlobalTimeout* besagt, wie lange ein Protokollexemplar schalten darf. Diese Angabe dient dem Erzeugen von Kommunikationsrunden, die nach einer bestimmten Zeit ein Ergebnis liefern oder abgebrochen werden.

- *DefaultTimeout* legt eine Schaltzeitbegrenzung für alle Transitionen fest, die an eine Rolle angeschlossen sind. Wird für eine solche Transition kein explizites Timeout festgelegt, so gilt das DefaultTimeout. Ist kein DefaultTimeout angegeben findet kein Timeout statt.

- Mittels der *ConstantNoOfParties* läßt sich festlegen, dass die Teilnehmerzahl nach Start des Prozesses nicht mehr schwanken darf. Verläßt ein Teilnehmer den Prozess, muss sofort ein Teilnehmer nachrücken. Bis dahin ruht der Prozess.

- Der *Anonymous*-Eintrag besagt, dass die Teilnehmer sich gegenseitig nicht kennen.

Rolleneinträge

Die Rolleneinträge gruppieren Verwaltungsinformationen für jeweils eine Rolle. Sie enthalten immer mindestens einen Eintrag. Jeder Eintrag überschreibt mögliche globale Einstellungen bezogen auf die angegebene Rolle.

```
metarolespec ::= ''rolename''
                 { metaroleentry
                     {, metaroleentry }* }
```

Die möglichen Einträge haben teilweise Entsprechungen unter den globalen Einstellungen. Für die für diese Einträge geltenden Aussagen ist oben Gesagtes ebenfalls gültig.

```
metaroleentry ::= {MinimumParties: anyint;}                |
                  {MaximumParties: anyint;}                |
                  {knows: {rolename {, rolename}* | all};  |
                  {newPartiesVoting [ { rolevoting* }| ; ] }|
                  {constantNoOfParties;}
```

6.2.2 Der Protokoll–Generator

Im folgenden wird die Konzeption und Funktionalität des im Rahmen dieser Arbeit entwickelten *Protokoll–Generators* vorgestellt. Diese Komponente unterstützt sowohl die graphische Erstellung von OOPAMELA–konformen Protokollen als auch die Generierung von konkreten Protokoll–Instanzen aus vorgefertigten Schablonen (*Templates*) auf einer abstrakten Meta–Ebene, d.h. ohne dass der Benutzer genaue Kenntnisse von OOPAMELA sowie Petrinetzen im allgemeinen haben muss.

6.2.2.1 Abstraktionsstufen von Verhandlungsprotokollen

Verhandlungsprotokolle in OOPAMELA können auf verschiedenen Abstraktionsstufen betrachtet werden. Auf niedrigster Ebene befindet sich das in OOPAMELA beschriebene Protokoll in reiner Textform. Diese Ebene ist z.B. für Korrektheitsanalysen vonnöten und dient daher der Vertrauensgewinnung der Benutzer des Verhandlungssystems, auch wenn diese selbst den OOPAMELA–Code nicht lesen oder verstehen können. Aus dem OOPAMELA–Code lässt sich eine graphische Abbildung erzeugen, die vor allem die Struktur der Verhandlung verdeutlicht. Aus Übersichtsgründen können bestimmte Bestandteile, wie z.B. die Beschränkung der Teilnehmerzahl oder der Validierungsmechanismus ausgeblendet werden. Das Wissen über die Semantik von Petrinetzen ist jedoch immer noch Voraussetzung, um solche Abbildungen zu verstehen.

Wird von dem zu Grunde liegenden Petrinetz abstrahiert, so erhält man eine Abstraktion, in der die Semantik des Verhandlungsprotokolls in den Vordergrund rückt. Das zu Grunde liegende Petrinetz darf nicht mehr Bestandteil des Verstehens auf dieser Ebene sein. Dies ist die Ebene, die für die Mehrheit der Benutzer eines Verhandlungssystems geeignet erscheint. Im Hinblick auf die Verwendung von Verhandlungsprotokollen in der Praxis (vgl. auch Analyse in Abschnitt 5.2) kann angenommen werden, dass der genaue Entwurf der allgemeinen Semantik eines Verhandlungsprotokolls oder einer ganzen Protokollklasse — bspw. des Typs "Auktion" — in Form eines Petrinetzes in der Regel nur einmal, und von einem professionellen Designer, durchgeführt werden muss. Nachdem ein solches Protokoll in seiner Struktur vorliegt, kann es dann durch einen geeigneten *Konfigurationsprozess*, der im wesentlichen die Parametrisierung der vorgesehenen Variablen beinhaltet, immer wieder neu generiert und zur Steuerung einer konkreten Verhandlung eingesetzt werden.

Viele Verhandlungsprotokolle sind sich in ihrer Grundstruktur ähnlich und unterscheiden sich in einzelnen Details häufig erst auf Metaprotokollebene. Diese Erkenntnis greift der Protokollgenerator auf, indem er die Gemeinsamkeiten verschiedener Verhandlungsprotokolle zu *Strukturprotokollen* zusammenfasst und ein konkretes Protokoll durch Parametrisierung und Anpassung aus dem Strukturprotokoll gewinnt. Die möglichen Parameter hängen vom Strukturprotokoll ab und werden durch dessen Designer festgelegt. Ohne zusätzliche Angaben von diesem sind die Parameter die Einträge des Metaprotokolls und die Beschränkung einzelner Transition bezüglich der Schaltvorgänge. Ansonsten wird jede Veränderung des Strukturprotokolls als Parametrisierung oder Anpassung bezeichnet.

6.2.2.2 Funktionalität des Protokoll–Generators

Die Aufteilung in Struktur- und konkretes Protokoll entspricht der Aufteilung des Generators in zwei Module. Mit dem ersten wird das Strukturprotokoll in Form eines grafischen Petrinetzes erstellt und unter einem semantisch sinnvollen Namen gespeichert. Die Erstellung eines Auktionsprotokolls beispielsweise ist in Abbildung 6.9 illustriert. Zusätzlich wird eine Beschreibung erfaßt und gespeichert. Das Metaprotokoll findet keine Beachtung. Für jedes Strukturprotokoll kann ein spezieller *Wizard*[5], das zweite Modul, angegeben werden. Zunächst wird ein Strukturprotokoll anhand des Namens und einer zusätzlichen Beschreibung ausgewählt. Der Wizard stellt eine Oberfläche zur Verfügung, mit der der Benutzer aus dem Strukturprotokoll durch Angabe von Parametern ein konkretes vollständiges Verhandlungsprotokoll erzeugen lassen kann. Das resultierende Protokoll kann einerseits mit einer grafischen Petrinetzrepräsentation gespeichert werden, so dass es sich beispielsweise mit dem Design–Tool ansehen lässt, auf jeden Fall wird es aber als OOPAMELA–Code im Textformat ausgegeben. Ein Standardwizard steht zur Verfügung und wird benutzt, wenn der Designer des Strukturprotokolls keinen anderen Wizard angibt. Da das Standardtool keine semantischen Informationen über das zu bearbeitende Protokoll besitzt, arbeitet es auf einer sehr niedrigen Abstraktionsebene und erfordert ein großes Mass an Systemwissen über OOPAMELA von dem Benutzer.

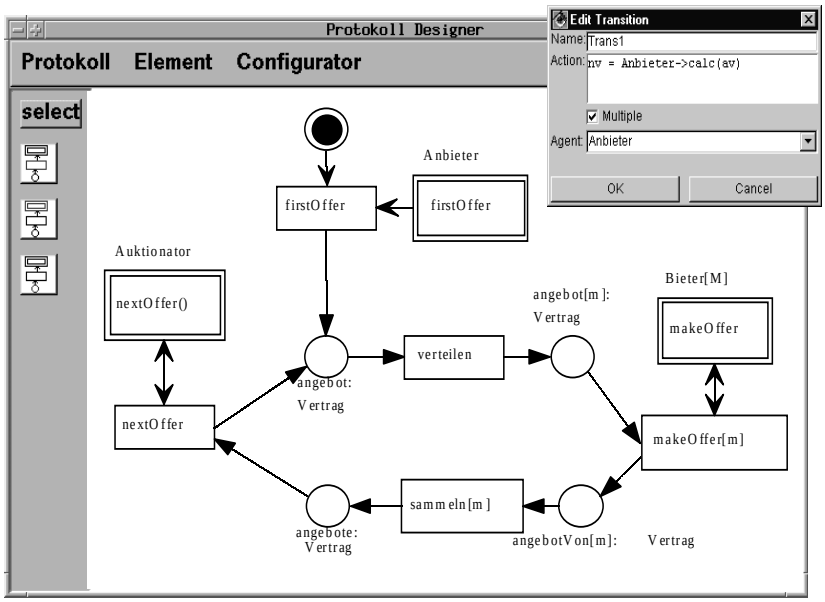


Abbildung 6.9: Graphische Erstellung eines Auktionsprotokolls

Der Designer kann zu dem von ihm erzeugten Strukturprotokoll einen Wizard angeben, der zum Konfigurieren und Erzeugen des konkreten Protokolls benutzt werden soll. Diesen Wizard muss er gegebenenfalls selbst erstellen. Dadurch ist es dem Designer möglich, das semantische Wissen über die von ihm

[5]In der Praxis wird mit "Wizard" in der Regel ein (zusätzliches) Programmmodul bezeichnet, mit dem spezielle Konfigurationen vorgenommen werden können.

erzeugte und beschriebene Klasse von Verhandlungsprotokollen in den Wizard aufzunehmen. Dies ermöglicht eine Oberflächengestaltung und Benutzerführung auf einem maximal hohen Abstraktionsniveau. Jeder Wizard wird von dem *Wizard–Loader* geladen. In dem Wizard–Loader werden alle verfügbaren Strukturprotokolle angezeigt, und nach Auswahl eines von ihnen wird der zugehörige Wizard gestartet. Die Wizard–Architektur ist prinzipiell nicht festgelegt. Der Entwickler eines Wizards muss sich lediglich an die Anforderung halten, dass der Wizard mindestens das Interface *configurator.WizardInterface* implementiert (siehe Abbildung 6.10).

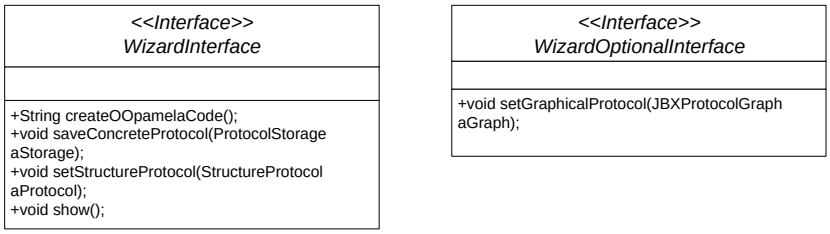


Abbildung 6.10: Wizard–Interface

Sofort nachdem der Wizard geladen wurde, wird dessen Methode *setStructureProtocol()* aufgerufen, um ihm sein Protokoll zu übergeben, welches er konfigurieren soll. Danach wird seine Methode *show()* gerufen, um den Wizard anzeigen zu lassen. Die Methode *createOOPamelaCode()* wird aufgerufen, um den Wizard aufzufordern, zu dem von ihm erstellten OOPAMELA–Netz den OOPAMELA–Code zu erzeugen. Dieser wird als eine lange Zeichenkette zurückgegeben. Die Methode *saveConcreteProtocol()* wird benutzt, um das fertige Protokoll im *ProtocolStorage* abzulegen.

Wenn der Wizard das sichtbare Netz bearbeiten oder darstellen soll, muss das Interface *configurator.WizardOptionalInterface* implementiert werden. Die einzige Methode *setGraphicalProtocol()* wird nach dem Start aufgerufen, nachdem *setStructureProtocol()* aufgerufen wurde, und sie dient dazu, dem Wizard die graphische Darstellung des OOPAMELA–Netzes mitzuteilen. Da der Benutzer vom Wissen über OOPAMELA befreit sein sollte, empfiehlt es sich, dieses Interface nur zu Testzwecken zu implementieren.

Ein Standardwizard steht zur Verfügung und wird benutzt, wenn der Designer des Strukturprotokolls keinen anderen Wizard angibt. Da das Standardtool keine semantischen Informationen über das zu bearbeitende Protokoll besitzt, arbeitet es auf einer noch niedrigen Abstraktionsebene und erfordert ein entsprechend großes Mass an Systemwissen über OOPAMELA von dem Benutzer.

Der Standard–Wizard besteht aus einem Fenster mit vier Seiten in "Notebook"–Format (siehe Abbildung 6.11): einer Seite zum Bearbeiten des Timeout–Verhaltens, einer für die Anzahl der Partner, einer um die Zahl der Aktionen eines Agenten zu beschränken und einer, auf der das resultierende Petrinetz dargestellt wird. Zusätzlich sind die beiden Buttons "Create" und "Cancel" vorhanden, mit denen das fertige Verhandlungsprotokoll als ASCII–Datei erzeugt bzw. die Bearbeitung abgebrochen wird.

Auf der Seite zum Timeout werden die Metaprotokolleinträge bezüglich des Timeoutverhaltens der Gesamtverhandlung und der einzelnen Transitionen, die an Rollen angebunden sind, bearbeitet. Die Einträge für die einzelnen Transi-

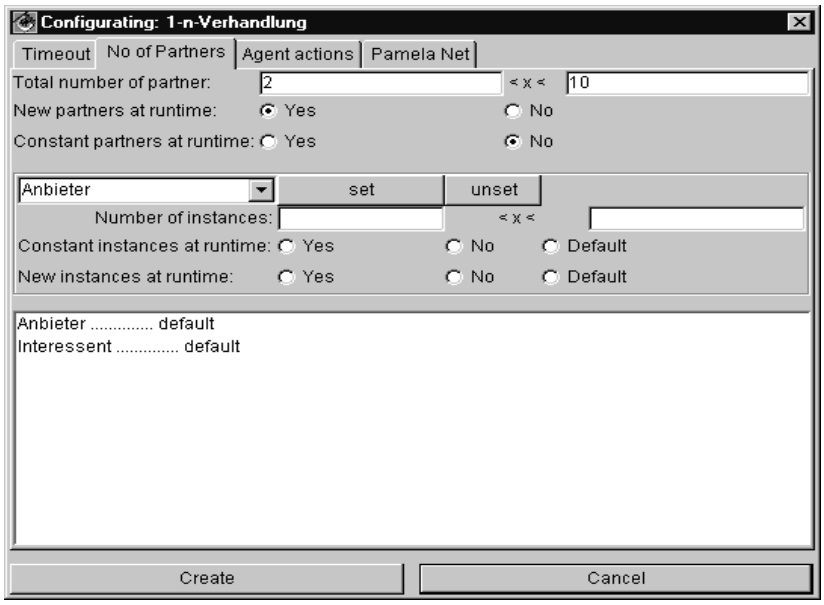


Abbildung 6.11: Der Standard–Wizard zur Parametrisierung des Strukturprotokolls

tionen werden in der unteren Liste angezeigt. Die möglichen Transitionen lassen sich aus einer Combobox auswählen. An dieser Stelle kann man sehen, dass es sinnvoll ist, Transitionen mit einem aussagekräftigen Namen zu versehen. Die Seite "No of Partners" zeigt die entsprechenden Einträge zu den Beschränkungen der Teilnehmerzahl der Verhandlungsrunde insgesamt sowie der einzelnen Rollen an. Dazu gehören sowohl minimale und maximale Anzahl als auch Einstellungen wie konstante Anzahl von Teilnehmern oder ob welche zur Laufzeit der Verhandlung beitreten können. Mit der Seite "Agent actions" wird die maximale Schaltzahl der an Rollen angebundenen Transitionen festgelegt. Wird dort eine für eine Transition eingetragen, wird eine Stelle mit der entsprechenden Anzahl von Marken an die Transition angebunden. Die Marken dienen als verbrauchbare Ressourcen. Diese Seite ist ein Beispiel dafür, wie ein Wizard auch das OOPAMELA–Netz verändern kann. Auf der Seite "Pamela Net" wird das aktuelle Netz angezeigt. Die auf der dritten Seite hinzugefügten Stellen werden dabei nicht nach grafisch sinnvollen Aspekten plaziert.

6.2.3 Die Protokoll–Engine

In diesem Abschnitt wird die technische Konzeption und Realisierung der in Abschnitt 5.3.2.1 eingeführten Protokoll–Engine als einer zentralen Komponente zur Steuerung verhandelder Agenten beschrieben.

6.2.3.1 Anforderungen und Architektur

Die Aufgabe der Protokollengine ist es sicherzustellen, dass die Verhandlungsparteien ein OOPAMELA–Protokoll einhalten und dass im Falle einer Protokollverletzung diese gemeldet wird. Die Schwierigkeit liegt u.a. darin zu entscheiden,

welcher Verhandlungsteilnehmer das Protokoll nicht eingehalten hat. Bei Verhandlungen mit mehreren Teilnehmern sollte der nicht protokollkonforme Teilnehmer aber zweifelsfrei identifiziert werden können, da die übrigen Agenten die Verhandlung eventuell fortsetzen wollen (z.B. mit einem neu zu findenden Teilnehmer, der den Ausscheidenden ersetzt).

Eine Abweichung von protokollkonformen Verhalten liegt auch dann vor, wenn eine Verletzung des Metaprotokolls vorliegt. Dies ist bspw. der Fall, wenn die Antwortzeit überschritten wird. Jede Kommunikation mit den Agenten ist zeitlich beschränkt; sollte ein Verhandlungsteilnehmer keine protokollkonforme Antwort innerhalb des geforderten Zeitlimits erbringen, erzeugt die Protokoll–Engine eine Standardantwort. Nicht-protokollkonformes Verhalten liegt vor, wenn entweder keine oder eine falsche Antwort gegeben wird. Um falsche Antworten zu identifizieren, müssen alle Nachrichten eines Teilnehmers überprüft werden. Da Regeln zur Überprüfung der Verhandlungsnachrichten beliebig komplex sein können und eine unabhängige Aufgabe darstellen, erscheint es sinnvoll, die Aufgabe der Verifizierung von Verhandlungsnachrichten im allgemeinen und Vertragsobjekten im besonderen auf eine externe Komponente zu verlagern. Diese Komponente sollte über die gleiche Schnittstelle wie die Verhandlungsteilnehmer angesprochen werden. Ungültige Nachrichten, die innerhalb des Zeitlimits eingehen und überprüft werden, werden dem Verhandlungsteilnehmer wieder vorgestellt, so dass dieser den Inhalt der Nachricht korrigieren kann. Der Eintritt in eine laufende Verhandlung wird durch eine Kontrollinstanz, die bereits bei der Registrierung der Teilnehmer ansetzt, geregelt. Auf diese Weise wird den Anforderungen nach Protokollkonformität und Gültigkeit Rechnung getragen.

Die Protokoll–Engine als Bestandteil des in Abschnitt 5.3.2 vorgestellten modularen Verhandlungssystems besitzt ihrerseits eine modulare Architektur, die in Abbildung 6.12 dargestellt ist.

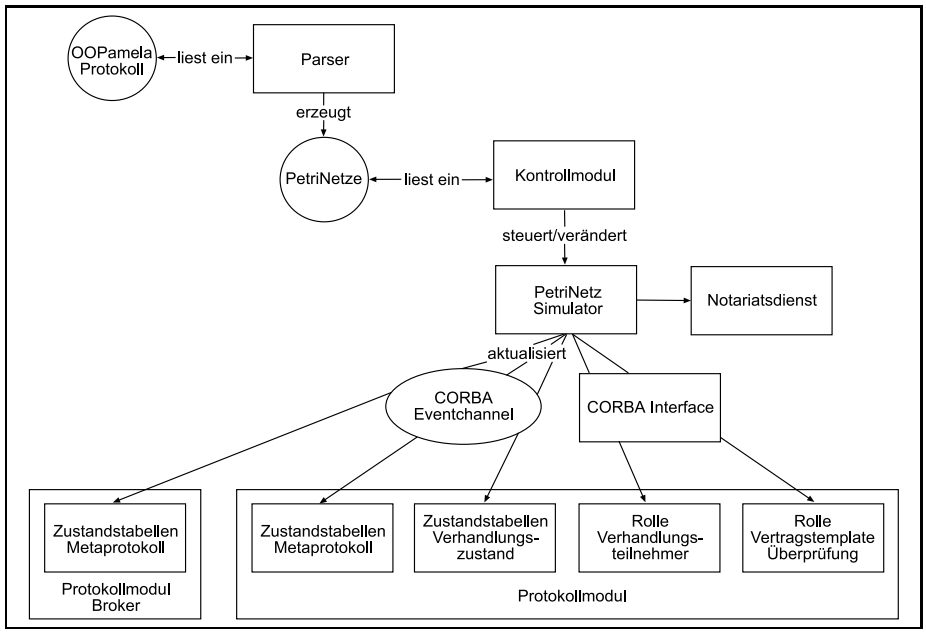


Abbildung 6.12: Gesamtarchitektur der Protokoll–Engine

Die Aufteilung der Module entspricht den einzelnen Aufgaben der Protokoll–Engine. Diese Komponenten können einzeln ausgetauscht werden, um so eine Anpassung für ein neues Verhandlungssystem zu erreichen. Für eine neue Verhandlungssprache würde die Komponente des Parsers ausgewechselt werden. Mit einem modifizierten Protokollmodul würden neue Agenten an einer Verhandlung teilnehmen können. Die primäre Aufgabe der Protokoll–Engine, die Steuerung und Kontrolle einer Verhandlung, findet sich in den folgenden fünf Modulen wieder.

- Ein *Parser* verarbeitet in OOPAMELA spezifizierte Verhandlungsprotokolle und erzeugt aus diesen ein entsprechendes Petrinetz.
- Während einer Verhandlung überprüft das *Kontrollmodul* den Zustand der Verhandlung und überwacht alle Ereignisse, die während einer Verhandlung auftreten.
- Ein *Petrinetzsimulator* arbeitet das Verhandlungsprotokoll ab.
- Die Kommunikation mit allen Verhandlungsteilnehmern erfolgt über ein *Protokollmodul.* In diesem werden Verhandlungszustände gespeichert und können Methoden eines Agenten aufgerufen werden.
- Für rechtliche Fragen wird ein Ablaufprotokoll der Verhandlung erstellt, an Hand dessen der durchgeführte Verhandlungsablauf rekonstruiert werden kann. Diese Aufgabe übernimmt ein *Notariatsdienst.*

Weitere Details über die Implementierung der Protokoll–Engine, insbesondere des Parsers, finden sich in [Gro00]. Als Petrinetzsimulator wurde die am Arbeitsbereich "Theoretische Grundlagen der Informatik" (TGI) entwickelte Software *Renew*, die insbesondere das Konzept der *Referenznetze* (s. Abschnitt 6.2.3.2) unterstützt, eingesetzt [KMW99]. Im folgenden wird auf die Spezifikation von Subnetzen zur Validierung des Teilnehmerverhaltens sowie auf die Realisierung des Protokollmoduls und des Kontrollmoduls eingegangen.

6.2.3.2 Subnetze zur Kontrolle des Teilnehmerverhaltens

Mit Hilfe von Referenznetzen [KMW99] ist es möglich, während der Ausführung des Gesamtnetzes eine bestimmte Aufgabe an eine Instanz eines Subnetzes, das von dem übergeordneten Netz erzeugt wird, jedoch unabhängig von ihm schalten kann. Die gestellte Aufgabe wird in dieser Instanz gelöst. Wichtig ist ein definiertes Ende der erzeugten Instanz. Alle Marken, die eventuell noch in dieser Instanz liegen, werden mit ihr gelöscht, sobald es keine weitere Referenz auf dieses Netz gibt. Diesen Umstand macht sich die Protokollengine zunutze.

Während der Spezifikation von Verhandlungen in OOPAMELA werden technische Details der Kommunikation einer Verhandlung in einem Metaprotokoll ausgedrückt. Die teilnehmenden Agenten müssen über ein beliebiges Kommunikationsmedium mit der Engine an der Verhandlung teilnehmen können. Die detailspezifischen Probleme dieser verteilten Kommunikation interessieren bei der Erstellung eines Verhandlungsprotokolls noch nicht. Der Protokoll–Generator (siehe Abschnitt 6.2.2), mit dem diese Netze erstellt werden, kennt die in Abbildung 6.13 dargestellte Notation für den Aufruf eines Agenten.

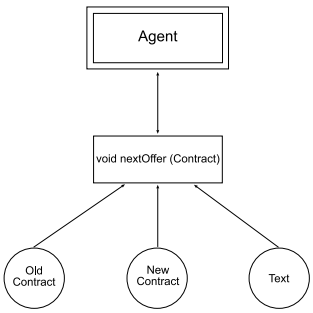


Abbildung 6.13: Aufruf einer Agentenmethode in OOPAMELA

Die Überprüfung der Agenten oder das Timeoutverhalten werden später im Metaprotokoll spezifiziert. Um eine möglichst große Transparenz für den Aufruf der Agenten zu erreichen, ist die Engine in der Lage, Teilnetze, die nur für die Kontrolle des Agentenverhaltens beim Aufruf einer Methode von ihm — z.B. zur Abgabe eines Vertragsangebotes — zuständig sind, zu verarbeiten. Diese Netze werden durch einen Ausdruck der Syntax:

```
#import<callNetName>
```

im OOPAMELA–Verhandlungsprotokoll definiert. Diese Netze werden immer dann durchlaufen, wenn ein Agent in einem Verhandlungsprotokoll aufgerufen wird. Dabei ist zu beachten, dass die Notation für den Aufruf des Agenten in OOPAMELA sich ändert. Der herkömmliche Aufruf:

```
trans ``callAgent'' uses ``myAgent''
{
   get ...
   put ...
   action ...
}
```

Dieser Aufruf wird um das Schlüsselwort "with" ergänzt und ergibt so:

```
trans ``callAgent'' uses ``myAgent'' with ``callNetName''
{
   get ...
   put ...
   action ...
}
```

Zum Zweck der Validierung von Vertragsangeboten und gleichzeitig der Kontrolle der im Protokoll spezifizierten Zeitbegrenzung für die Abgabe eines solchen Angebotes wurde mit Hilfe der *Renew*–Entwickler ein Standard–Referenznetz entworfen, das in vielen, wenn nicht den meisten, Verhandlungen verwendet werden kann, da die von diesem Netz aufgerufene spezifische Validierungsfunktion an eine externe Komponente ausgelagert werden kann.

Das Standardnetz in Abbildung 6.14 sieht daher außer der Kommunikation mit dem Agenten einen Timer vor, der sicherstellt, dass ein Netz weiterschaltet,

auch wenn der Agent nicht mehr vorhanden ist. Außerdem werden die Änderungen, die ein Agent innerhalb des Vertragtemplates vornimmt, überprüft. Das Standard–Referenznetz der Protokoll–Engine erfüllt diese Bedingungen:

Dieses Netz ist in herkömmlicher Petrinetz–Notation geschrieben. Die Syntax des Aufrufs in der Transition "Start":

```
:new (strFunc, Con1, Role, Tester)
```

bedeutet, dass mit dieser Transition das Netz gestartet wird. Das zu erreichende Ende des Ablaufs markiert die Transition "Ende".

Für die Ausführung von Petrinetzen benutzt die Engine den Petrinetzsimulator *Renew* [KMW99]. Dieser Simulator beherrscht den Aufruf von Referenznetzen aus anderen Netzen heraus. Diese Netze werden als eigenständige Objekte behandelt. Sie werden ähnlich wie in einer objektorientierten Programmiersprache durch das Schlüsselwort *new* erzeugt. Nach dem Ausführen der Methode *new* innerhalb der Transition wird das entsprechende Netz in den Speicher geladen und initialisiert. Es kann dann parallel zum aufrufendem Netz ausgeführt werden. Jedes Referenznetz benutzt die Engine nur einmal, um einen Agenten aufzurufen. Für den Fall, dass sich Zyklen innerhalb des Verhandlungsprotokolls befinden, und so ein Agent mehrmals aufgerufen wird, erzeugt die Engine für jeden Aufruf ein neues Referenznetz . Auf diese Weise ist sichergestellt, dass Marken, die zum Beispiel durch die Fehlerbehandlung entstehen, nicht innerhalb eines Netzes liegenbleiben und so den Ablauf stören können.

Die wesentlichsten Transitionen dieses Netzes erfüllen folgende Funktionen.

1. Die erste Transition "Start" wird aus einem anderen (übergeordneten) Netz mit folgenden Parametern aufgerufen.

   - aufzurufende Funktion
   - zu übergebender Vertrag
   - die Rolle des Agenten
   - die Rolle der externen Validierungskomponente

2. Die nächste wichtige Transition, "Clone", verdoppelt einen Vertrag, so dass Änderungen an diesem überprüft werden können.

3. Die Transition "callRole" ist für den Aufruf des Agenten zuständig. Sie übergibt den Vertrag einem Agenten. Der veränderte Vertrag wird dem Tester übergeben.

4. Die Transition "Tester", die mit dem Aufruf der Validerungsfunktion eines externen Testers verknüpft ist, ist für die Überprüfung der Änderungen zuständig. Für den Fall, dass die Änderung nicht gültig ist, wird er durch die Transition "Check1" dem Agenten wieder vorgestellt, andernfalls wird eine Veränderung des Verhandlungszustandes an alle Protokollmodule gemeldet. Nun könnte der Agent sich als uneinsichtig erweisen und das Netz in eine Endlosschleife führen.

5. Die Transition "Timer" ist dazu da, mit Hilfe der Transition "DefaultCon" nach einem festgesetzten Zeitintervall einen Vertrag zu erzeugen. So ist sichergestellt, dass dieses Netz innerhalb eines Zeitraums terminiert.

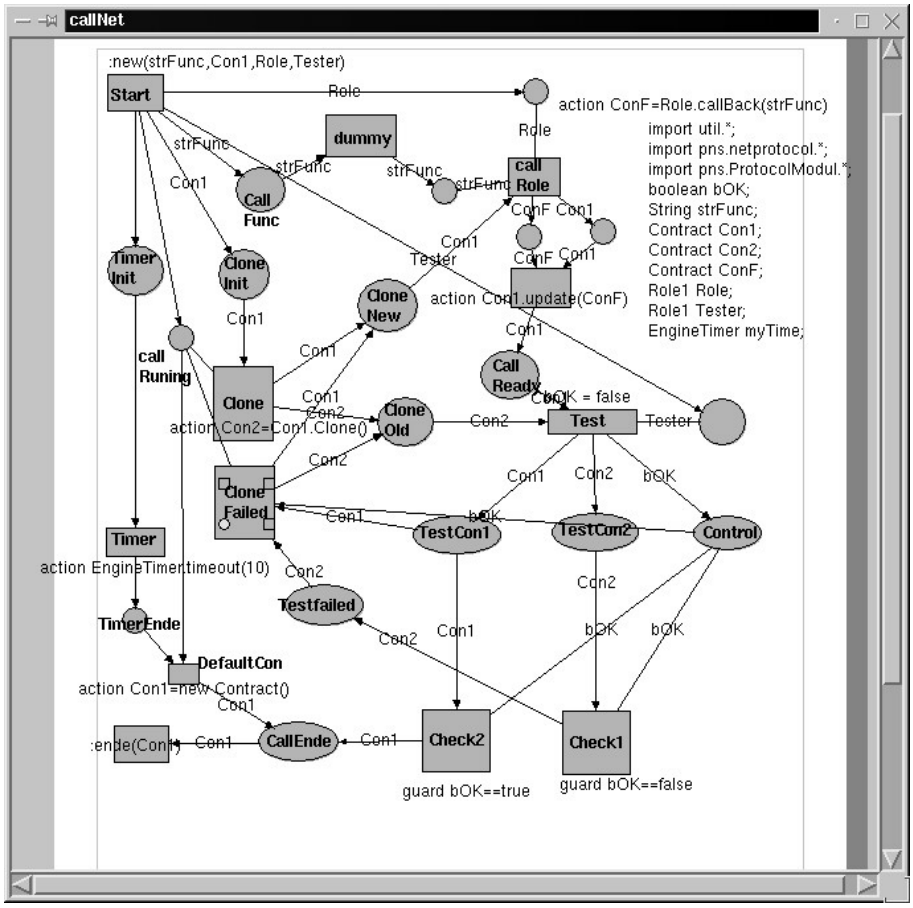


Abbildung 6.14: Das Standard–Referenznetz zur Validierung einer Teilnehmeraktion

6. Die Transitionen “Check1” und “Check2” überprüfen das Ergebnis des Testers und führen entweder das Vetragstemplate noch einmal dem Agenten vor (Fehlerfall) oder beenden den Aufruf des Agenten.

7. Die Transition “Ende” ist für die Resynchronisation mit dem aufrufendem Netz verantwortlich. Das Verhandlungsprotokoll wartet mit einer entsprechenden Transition auf die Beendigung des Netzes.

Dieses Netz stellt eine mögliche Variante zur Überprüfung der Aktionen von Verhandlungsteilnehmern dar. Dem Benutzer der Protokoll–Engine ist es freigestellt, dieses Netz zu verändern und so den Gegebenheiten einer Verhandlung anzupassen. Jedes andere Netz kann ebenfalls mit der erwähnten *import*–Anweisung in die Protokoll–Engine geladen werden. Die vollständige Spezifikation des Standardreferenznetzes in der Sprache OOPAMELA findet sich in Anhang D.3.

6.2.3.3 Das Protokollmodul

Das Protokollmodul implementiert die Schnittstelle zwischen den an einer Verhandlung teilnehmenden Agenten und der Protokoll–Engine (vgl. Abschnitt 5.3.1.2). Es speichert den Zustand einer Verhandlung, prüft, welche eigenen Verhaltensweisen bezüglich der Teilnehmerrolle für den Agenten erlaubt sind und teilt dem Benutzer das Auftreten bestimmter semantisch interessanter Ereignisse mit. Zusätzlich prüft das Protokollmodul, ob ein- oder ausgehende Nachrichten protokollkonform sind und teilt dies dem Verhandlungssystem bzw. dem Teilnehmer mit. Durch Austausch des Protokollmoduls kann ein Agent an unterschiedliche Verhandlungsprotokolle oder sogar Verhandlungssysteme angepasst werden.

Implementationstechnisch kann dieses Modul wird entweder von der Engine oder vom Agenten selbst erzeugt werden. Da die Funktionsweise des Protokollmoduls von der Protokollsemantik abhängig ist und die Aufgabe der Protokoll–Engine gerade darin besteht, die Agenten bei der Einhaltung der Protokollkonformität zu unterstützen, erscheint es konsequent, das Protokollmodul von der Engine generieren zu lassen. Um ein Protokollmodul dynamisch generieren zu können, bedarf es jedoch einer allgemeinen Konzeption für dieses Modul, die auf jedes Verhandlungsprotokoll anwendbar ist. Ein Problem für ein solches universelles Protokollmodul ist, dass unterschiedliche Methoden des Agenten aufgerufen werden müssen. Den Aufruf der Agenten erfährt die Engine erst zur Laufzeit durch das Verhandlungsprotokoll. Aus diesem Grund benutzt die Protokollengine sog. Parameterarrays und Referenznetze für die Kommunikation mit den Agenten. Daraus ergibt sich der in Abbildung 6.15 gezeigte Aufbau für das Protokollmodul.

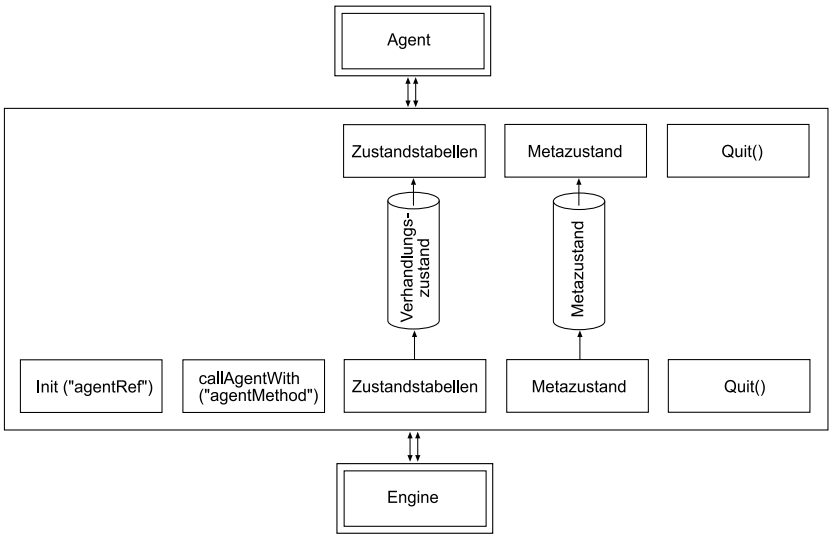


Abbildung 6.15: Interne Struktur des Protokollmoduls

Die Verbindung zwischen dem Protokollmodul und dem Agenten wird mittels des Plug-in–Mechanismus (siehe Abschnitt 6.1.1) hergestellt. Die Kommunikation zwischen beiden ist bidirektional, wobei der Agent folgende Methoden des Protokollmoduls aufrufen kann:

1. Abruf des Verhandlungszustandes: Der Verhandlungszustand spiegelt die Anzahl der Teilnehmer, Angebote und den Verhandlungsfortschritt wider.
2. Abruf des Metaverhandlungszustandes: Der Metaverhandlungszustand reflektiert Ereignisse, die das Hinzukommen oder Wegfallen bestimmter Teilnehmer betreffen.
3. Beenden der Verhandlung: Jeder Verhandlungsteilnehmer darf eine Verhandlung auf diese Weise (vorzeitig) verlassen[6].

Die Engine benutzt folgende Methoden, um einen Agenten durch eine Verhandlung zu führen:

1. Initialisieren des Protokollmoduls: Am Anfang einer Verhandlung wird die Verbindung zwischen Protokollengine und Protokollmodul hergestellt.
2. Beenden einer Verhandlung: Diese Methode ist als Konsequenz für ein Fehlverhalten des Verhandlungsteilnehmers gedacht.
3. Aktualisieren des Verhandlungszustands
4. Aktualisieren des Metaverhandlungszustandes
5. Eine generische Methode, um eine beliebige Methode eines Agenten aufzurufen: Da die Schnittstelle des Agenten erst zur Laufzeit durch das Verhandlungsprotokoll definiert wird, reicht diese Methode die Aufrufe der Engine an den Agenten weiter. Das Aufrufverhalten dieser Methode (synchron/asynchron) wird durch den Rückgabewert definiert. Alle Methodenaufrufe werden immer dann synchron ausgeführt, wenn diese Methode einen Rückgabeparameter besitzt.

Verhandlungszustände

Die Kenntnis über den Zustand einer Verhandlung (vgl. Abschnitt 5.2.3.4) ist für die erfolgreiche Durchführung einer Verhandlung von großer Bedeutung. Um einen Agent/Benutzer zu unterstützen, eine eigene Strategie während einer Verhandlung zu verfolgen, muss der aktuelle Verhandlungszustand registriert und auf eine vom Protokoll erlaubte Weise den Teilnehmern mitgeteilt werden. Es existieren zwei verschiedene Definitionen von "Verhandlungszustand":

- *Implizit* ist ein Verhandlungszustand repräsentiert durch die aktuelle Markierung des Netzes.
- *Explizit* ist ein Verhandlungszustand, wenn die für einen Teilnehmer "relevanten" Informationen, um die nächste Aktion zu berechnen, in ihm enthalten sind. Als Bestandteil eines expliziten Zustands wurde identifiziert:
    1. Die Menge der aktuell vorliegenden (noch nicht entschiedenen) Angebote.

[6]Jedoch kann der Verhandlungsprozess nach einer gewissen Verzögerung trotzdem fortgesetzt werden, wenn ein Agent die Verhandlung verlässt, ohne diese Methode aufzurufen (vgl. Abschnitt 6.2.3.2).

2. Die Menge der aktuell vorhandenen Teilnehmer.
3. Die Relation zwischen Elementen der ersten und Elementen der zweiten Menge, z.B. Teilnehmer A hat Angebot 2 "vorgelegt", Teilnehmer B und D haben Angebot 2 "zugestimmt", C hat Angebot 2 "abgelehnt" und Angebot 3 (= Gegenangebot) "vorgelegt" etc. In der Implementation kann diese Information auch als Bestandteil der Angebote gesehen werden.

Immer wenn eine neue, zulässige Aktion eines Benutzers/Agenten den aktuellen Zustand ändert, dann liegt ein neuer Zustand vor, der allen (berechtigten) Interessenten gemeldet werden soll. Außerdem gibt es noch zwei besondere Zustände, um den "START" und das "ENDE" (mit oder ohne Einigungssmarkierung) der Verhandlung anzukündigen.

Die explizite Spezifikation von Verhandlungszuständen stellt den Versuch dar, ein generisches Format zu verwenden, das für jede Verhandlung sinnvoll sein kann. Von Protokoll zu Protokoll kann dieses Format durch entsprechende einschränkende Einträge im Metaprotokoll (vgl. 6.2.1.3) reduziert werden.

Beispiele

Der Zustand einer Auktion besteht aus dem aktuellen Angebot, dem Anbieter (bei Nichtanonymität) und der Anzahl der noch vorhandenen Teilnehmer. Ein Zustandswechsel liegt vor, wenn ein neues Angebot vorliegt oder wenn das Timeout abläuft.

Der Zustand der "Reisverhandlung" (s. Abschnitt 5.2.3.5) besteht aus 0 bis max. 6 aktuellen Angeboten (jeder der 3 Teilnehmer darf max. 2 Angebote gleichzeitig vorlegen) und den Relationen der Teilnehmer zu den Angeboten. Die Verhandlung ist in dem Moment erfolgreich beendet, wenn alle 3 einem Angebot zugestimmt haben.

Der Vorteil der expliziten gegenüber der impliziten Definition besteht in der Repräsentationsunabhängigkeit. Diese Art der Definition ist deshalb auf der richtigen Abstraktionsstufe angesiedelt. Ein Teilnehmer muss nicht verstehen, was überhaupt eine Markierung oder Stellenbelegung ist. Ein prinzipieller Nachteil ist, dass genau durch diese Abstraktion Information verlorengehen kann.

Wie schon in Abschnitt 5.2.3.4 erwähnt, gibt es sinnvolle Protokolle, in denen nicht jeder Zustand allen Beteiligten bekannt gemacht werden soll. Ein generischer Ansatz, dieses Problem zu lösen, basiert auf den Rolleninformationen und sieht wie folgt aus:

1. *Bzgl. der Teilnehmer*: Die Rolle eines Teilnehmers legt fest, welche Rollen er sehen darf und auch, ob er die Gesamtzahl der Teilnehmer erfahren kann.

2. *Bzgl. der Angebote*: Die Rolle legt auch fest, welche Adressierungsarten erlaubt sind. Aufgrund der Adressierung eines Angebots kann die Protokoll–Engine herausfinden, wem der entsprechende Verhandlungszustand zugestellt werden darf.

3. *Bzgl. der Relation Teilnehmer–Angebot*: Wenn ein Angebot an einen Teilnehmer adressiert ist, dann sieht er gemäß der Adressierungsart, wer oder

wie viele wie zu diesem Angebot abgestimmt haben. Die Sichtbarkeit dieser Relation wird jedoch durch 1. eingeschränkt.

Um jederzeit alle beschriebenen Zustände an alle Agenten übermitteln zu können, werden während der Verhandlung in jedem Protokollmodul interne Tabellen gepflegt, die diesen Zustand widerspiegeln. Für jeden beschriebenen Zustand, z.B. bzgl. der vorgelegten Angebote, wird eine Tabelle von Teilnehmern und Agenten erzeugt:

	Agent1	Agent2	Agent3	Agent4
Angebot 1	x			
Angebot 2		x		
Angebot 3		x		
Angebot 4			x	

Tabelle 6.3: Gemeinsame Struktur der Zustandstabellen

Vertragstemplates

Um die Einhaltung der Semantik von gültigen Verträgen als eines wichtigen Aspekts der Protokollkonformität zu unterstützen, werden sogenannte *Vertragtemplates* als strukturierte Schablonen zum Ausfüllen eines gültigen Vertrags eingeführt. In der Initialisierungsphase einer Verhandlung (vgl. Abschnitt 5.3.2.1) muss bei der Festlegung eines OOPAMELA–Verhandlungsprotokolls auch ein Vertragstemplate spezifiziert bzw. ausgewählt werden. Alle Agenten kommunizieren über diese Vertragtemplates miteinander. Sie können zwischen den Agenten ausgetauscht werden. Damit die Protokoll–Engine alle Änderungen an den Verträgen prüfen kann, werden sie über die Protokollmodule ausgetauscht. So können alle Veränderungen an diesen Templates durch die Protokoll–Engine verifiziert werden.

Die Steuerungseinheit ist prinzipiell in der Lage, mit jeder Art von Vertragstemplates zu arbeiten. Die Überprüfung aller Änderungen an diesen Vertragtemplates wird von einer externen Komponente (siehe Abschnitt 6.2.3.2) vorgenommen. Damit ein Vertrag von der Protokoll–Engine vernünftig eingesetzt werden kann, müssen folgende Bedingungen erfüllt sein:

- Ein Vertragstemplate muss die Methode *Clone()* implementieren.
- Ein Vertragstemplate muss serialisierbar sein, so dass es als Java–Objekt verschickt werden kann (vgl. Abschnitt 6.1.2.2).
- Ein Vertragstemplate sollte zumindest den Gegenstand der Verhandlung beschreiben können.

Die Modellierung der Vertragstemplates für automatisierte Verhandlungen durch Softwareagenten orientiert sich an dem Vertragsmodell des COSMOS–Projektes [MGT+99], wobei diejenigen Bestandteile, die z.Z. nicht von Agenten interpretiert werden können (wie z.B. Vertrags*klauseln* im juristischen Sinne), weggelassen wurden.

6.2.3.4 Das Kontrollmodul

Das Kontrollmodul ist für die Kontrolle der Protokoll–Engine verantwortlich. Es ist für das Starten einer Verhandlung und das Einbinden der Agenten zuständig. Das Kontrollmodul kann am Verhandlungsende die Verhandlung wieder stoppen. Für die Kontrolle einer Verhandlung gibt es zwei Schnittstellen. Auf der einen Seite gibt es eine Schnittstelle, die vom Broker aus benutzt werden kann, um eine Verhandlung zu initiieren. Auf der anderen Seite existiert eine Benutzerschnittstelle, um diese Funktionalität von Hand aus zu bedienen.

Die Schnittstelle für den Broker

Damit ein Broker innerhalb der Verhandlungsarchitektur (s. Abschnitt 5.3.2) eine Verhandlung initiieren kann, wurde die in Abbildung 6.16 gezeigte Schnittstelle mit folgenden Methoden implementiert:

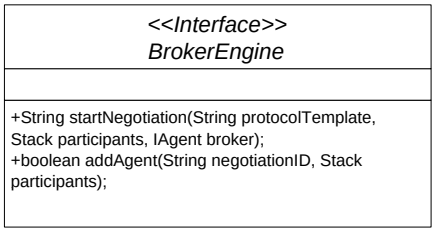


Abbildung 6.16: Die Schnittstelle für den Broker

Die Methode *startNegotiation()* erhält als Parameter ein Verhandlungsprotokoll, einen Stack mit den Voyager–Referenzen aller für diese Verhandlung angemeldeten Teilnehmern und die Voyager–Referenz des Brokers. Daraufhin erzeugt die Protokoll–Engine die Protokollmodule für jeden Verhandlungsteilnehmer, liest das Verhandlungsprotokoll ein, erzeugt die entsprechenden Petrinetze und startet die Verhandlung. Jede Verhandlung erhält eine eindeutige Identifizierung, um sich später auf sie zu beziehen. Die Referenz des Brokers wird dafür benutzt, den Metaverhandlungszustand an den Broker zu übermitteln. Für den Fall, dass ein Teilnehmer während einer Verhandlung ausscheidet, ist es seine Aufgabe, neue Teilnehmer zu finden und diese der Steuerungseinheit zuzuführen. Dazu wird die Methode *addAgent()* benutzt, die es erlaubt, Verhandlungsteilnehmer während einer laufenden Verhandlung der Steuerungseinheit zuzuführen.

Die Benutzerschnittstelle

Um manuell auf die Funktionen des Kontrollmoduls zugreifen zu können, wurde — hauptsächlich zu Testzwecken und für Vorführungen — eine GUI–Komponente für die Protokoll–Engine entworfen (siehe Abbildung 6.17).

Mit Hilfe dieser Schnittstelle ist es möglich, direkt auf die Funktionen der Protokoll–Engine zuzugreifen. Im folgenden werden die Befehle kurz erläutert. Sie können für das Ausprobieren und Testen von Verhandlungsprotokollen benutzt werden. Diese Benutzerschnittstelle kann im Moment folgende Befehle verarbeiten (diese Menge kann aber noch erweitert werden):

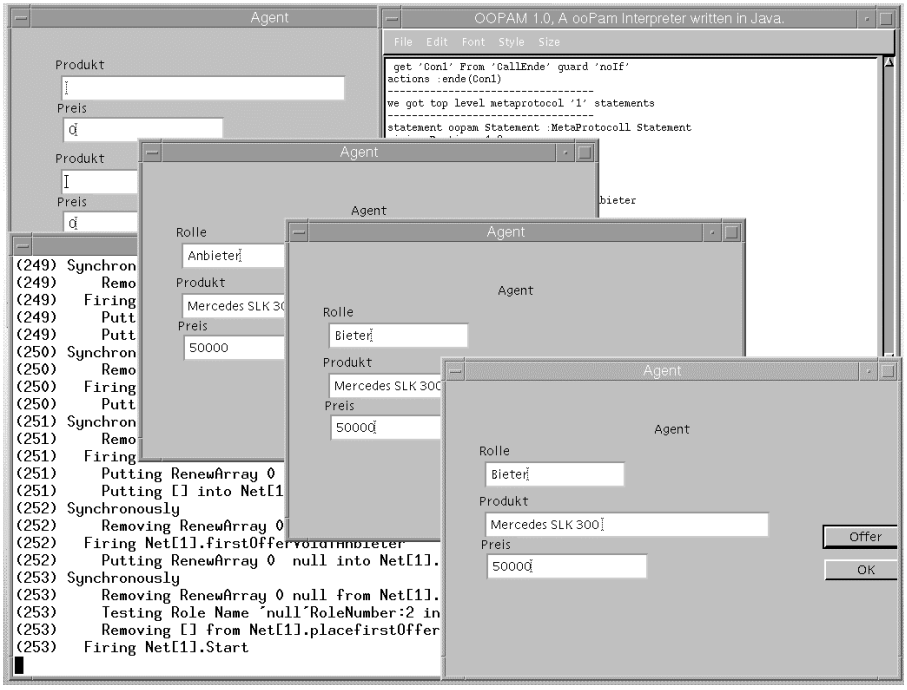


Abbildung 6.17: Die Benutzerschnittstelle des Kontrollmoduls

- **load "protocolname"**: veranlasst das Laden eines persistent abgelegten, vollständig parametrisierten Verhandlungsprotokolls in die Engine. Die entsprechenden Referenz- und Petrinetze werden sofort erzeugt.
- **run**: startet eine Verhandlung
- **prot "count"**: erzeugt eine beliebige Anzahl von Protokollmodulen, in diese muss aber noch eine Voyager–Referenz geladen werden.
- **connect "IPaddress, IPportnumber"**: verbindet das Protokollmodul mit einem Voyager–Agenten unter der angegebenen Adresse.
- **contract**: erzeugt ein neues Vertragstemplate. Die Verhandlung basiert aufgrund dieses Vertragstemplates.
- **bye:** beendet die Steuerungseinheit.
- **auk:** erzeugt 4 Protokollmodule und verbindet sie mit der Protokoll–Engine und 4 Agenten unter den Adressen: *localhost:7600*, *localhost:7700*, *localhost:7500* und *localhost:80000*. Dieser Befehl wird nur für Demonstrationszwecke benutzt.

Mit Hilfe dieser Befehle ist es möglich, die Protokoll–Engine als Steuerungseinheit für Verhandlungen direkt einzusetzen, ohne auf weitere Unterstützungsdienste der Gesamtarchitektur angewiesen zu sein. Verhandlungen, deren Protokolle oder die Agenten können so einfach getestet und entworfen werden.

6.3 Realisierung von Verhandlungsstrategien

Zu einer Automatisierung von Verhandlungen, die über eine systemtechnische Unterstützung der Verhandlungsteilnehmer hinausgehen soll, gehört zweifelslos die Bereitstellung konkreter Verhandlungsstrategien, die von einem elektronischen Agenten genutzt werden können, um den Inhalt der mit anderen Teilnehmern ausgetauschten Verhandlungsnachrichten zu generieren. Wie groß das Potenzial automatisierter Verhandlungssysteme wirklich ist, hängt offensichtlich in entscheidendem Masse davon ab, ob formale Strategien gefunden werden, die sich in der realen Praxis bewähren können.

Im Gegensatz zu den beschriebenen Verhandlungsprotokollen und deren Realisierungstechniken ist es jedoch unmöglich (oder zumindest äußerst unwahrscheinlich), ein generisches Spezifikationsformat zu finden, mit dem sich jede Verhandlungsstrategie beschreiben ließe, wenn dieses Format mehr (verhandlungsspezifische) Semantik als eine universelle Programmiersprache bieten soll. Dies liegt daran, dass Verhandlungsstrategien (fast) so vielfältig sein können wie Algorithmen, die aus einer Eingabe (dem erhaltenen Angebot) eine syntaktisch korrekte Ausgabe (dem Gegenangebot) erzeugen können. Außer den allgemeinen Formalisierungskriterien wie etwa Nutzenfunktion, Wissensbasis, Protokollkonformität... (vgl. Abschnitt 5.2.2.2) lassen sich nämlich kaum weiter gehende Gemeinsamkeiten zwischen möglichen Strategien feststellen.

Deshalb läuft jeder Versuch, konkrete Verhandlungsstrategien zu realisieren, letztendlich darauf hinaus, einen einzelnen Algorithmus bzw. eine Klasse von Algorithmen zur Berechnung von Verhandlungsaktionen zu entwerfen und zu implementieren. Dementsprechend wird im folgenden zunächst kurz auf Klassifikationskriterien für potenzielle Algorithmen eingegangen und anschließend ein Framework zur Anwendung von Genetischen Algorithmen als Verhandlungsstrategien (siehe auch [TWL00, WTL00, Wol00]) vorgestellt.

6.3.1 Klassifikationskriterien

Die genannte Vielfalt von potenziellen Algorithmen zur Realisierung von Verhandlungsstrategien führt auch dazu, dass nur relativ allgemeine Kriterien zu deren Klassifikation aufgestellt werden können. Im Prinzip kann eine Verhandlungsstrategie mit jedem Algorithmus, der syntaktisch korrekte Verhandlungsaktionen bzw. -nachrichten berechnet, implementiert werden. In der Fachliteratur sind eine entsprechend große Vielzahl von sehr unterschiedlichen Algorithmen für — meist ziemlich spezifische — Verhandlungsprobleme vorgeschlagen worden (vgl. Abschnitt 5.2). Deshalb wird hier nicht versucht, eine erschöpfende Klassifikation durchzuführen, sondern gewisse Kriterien herauszustellen, an Hand derer eine bestimmte Strategie bzgl. ihrer Eignung für eine Anwendungsumgebung generell bewertet werden kann (s. a. [TGML98]).

Mathematisch–analytische Strategien Dabei wird ein Ansatz gewählt, der mit Hilfe eines festen analytischen Verfahrens ein möglichst optimales Verhandlungsangebot berechnet. Solche Ansätze werden beispielsweise aus der Spieltheorie abgeleitet.

Heuristisch–evolutionäre Strategien Bei einem solchen Vorgehen wird versucht, mit Hilfe von lernenden Systemen oder durch eine Art von evolutionärem Vorgehen möglichst gute Verhandlungsnachrichten zu berechnen.

Im Gegensatz zu den mathematisch–analytischen Strategien ist aber das konkrete Verfahren, um zu solchen Verhandlungsnachrichten zu kommen, nicht fest eingebaut, sondern man erhofft sich, dass durch einen Lerneffekt ein solches Verfahren entsteht. Damit setzt dieses Verfahren eine Ebene höher an: Es gibt keinen festen Algorithmus an, um zu Verhandlungsnachrichten zu kommen, sondern ein Verfahren, um zu solch einem Algorithmus zu kommen.

Lokale Strategien sind solche, die bei der Berechnung der Verhandlungsaktionen nicht auf die Kooperation mit anderen Verhandlungsteilnehmern angewiesen sind. D.h. jede lokale Strategie arbeitet für sich allein und nur zum eigenen Nutzen.

Verteilte Strategien Im Gegensatz zu lokalen Strategien machen verteilte Strategien Gebrauch von der Kooperation mit anderen Verhandlungsteilnehmern. Dabei sind zwei Formen unterscheidbar: Kooperation zwischen gewissen Teilnehmern an der *selben* Verhandlung, bspw. um *Koalitionen* zu bilden, und Kooperation zwischen Teilnehmern *verschiedener* (gleichzeitiger) Verhandlungen.

Damit bilden diese Kriterien zwei gegensätzliche Paare (mathematisch–analytisch vs. heuristisch–evolutionär und lokal vs. verteilt), die zwei unterschiedliche Dimensionen zur Charakterisierung von Strategien darstellen. Weitere Kriterien — bspw. ob die Strategie explizit von früher gewonnener Verhandlungserfahrung Gebrauch macht (wie z.B. in [ZW00] beschrieben), oder ob eine Strategie jeweils nur an einer Verhandlung oder gleichzeitig an mehreren (vgl. konzeptionellen Ansatz in [BK00]) beteiligt ist — sind zwar denkbar, jedoch lassen sich die meisten von ihnen entweder unter die oberen Kriterien subsummieren — z.B. kann die gleichzeitige Teilnahme an mehreren in wechselseitiger Beziehung stehenden Verhandlungen als eine Sonderform der Kooperation zwischen verschiedenen Verhandlungen und damit als eine Form der verteilten Strategien betrachtet werden — oder sie beziehen sich auf spezielle Aspekte, die nicht für alle Verhandlungsstrategien in Betracht zu ziehen sind.

Analytische versus evolutionäre Strategien

In der Literatur existieren sowohl analytische als auch evolutionäre (bzw. nicht–analytische i.a.) Verfahren als Vorschläge für automatische Verhandlungsstrategien, wobei die Zahl der analytischen Verfahren weit überwiegt (siehe z.B. Überblick in [Seb92]). Selbst wenn man nicht nur evolutionsorientierte Verfahren, sondern auch andere dynamische Strategien, die bspw. auf *fuzzy constraints* basieren (siehe z.B. [KB00]), zu den nicht–analytischen Verfahren hinzuzählt, ändert sich dadurch das zahlenmäßige Verhältnis zwischen beiden Kategorien kaum. Der Hauptgrund dafür liegt in der langen Tradition der *Spieltheorie*, aus der die meisten analytischen Verfahren stammen, als einer Disziplin zur Erforschung formaler Strategien für bzw. strategischen Verhaltens von menschlichen Verhandlungsführern.

Analytische und evolutionäre Strategien basieren auf sehr unterschiedlichen Prinzipien und haben im Vergleich zueinander sowohl Vor- als auch Nachteile. Während erstere von relativ festen bzw. *statischen* mathematischen Modellen Gebrauch machen, benutzen letztere ein sehr *dynamisches* Berechnungsmodell,

das auf Basisprinzipien der Evolution basiert. Bezüglich analytischer Verfahren gibt es bereits relativ ausgeklügelte Techniken, die als Strategien für spezifische Verhandlungsprobleme genutzt werden können. Beispielsweise wird in [ZS96] eine auf der Bayes–Methode basierte Technik zur Schätzung der Akzeptanzschwelle des Kontrahenten (in einer bilateralen Verhandlung) vorgeschlagen. Diese Technik demonstriert auch, dass obwohl die zu Grunde liegende Berechnungsmethode statisch ist, ein gewisser Lerneffekt dadurch erreicht werden kann, dass die benutzte Wissensbasis während der Verhandlung dynamisch aktualisiert wird.

Mit evolutionären Strategien kann jedoch im allgemeinen eine wesentlich größere Dimension bezüglich des Lerneffekts erzielt werden, da nicht nur die verwendete Datenbasis durch Evolutionstechniken angepasst werden kann, sondern auch die auf den Daten operierenden Algorithmen selbst, was auch als evolutionsbasierte Programminduktion [Jac97] bezeichnet wird. Daher sind evolutionäre Strategien prinzipiell erheblich anpassungsfähiger und es ist zumindest anzunehmen, dass sie für eine dynamische Umgebung wie die eines elektronischen Dienstemarktes besser geeignet sein könnten als analytische Strategien. Diese Annahme muss jedoch durch eine konkrete Realisierung und entsprechende Experimente gestützt werden. Das im folgenden dargestellte Framework wurde als ein erster Schritt zu diesem Zweck entwickelt.

6.3.2 Ein Framework zur Anwendung von Genetischen Algorithmen

6.3.2.1 Basisprinzipien Genetischer Algorithmen

Genetische Algorithmen sind aus der Motivation entstanden, die Prinzipien der natürlichen Evolution im Bereich der Informatik zu nutzen. Dabei werden meist mit Hilfe sogenannter *genetischer Operatoren* neue Objekte erzeugt. Diese müssen dann — wie in der Natur — ihre Tauglichkeit, die in der Fachliteratur als *Fitness* bezeichnet wird, unter Beweis stellen. In der Natur ist die Fitness meistens eine Umschreibung für die Überlebenswahrscheinlichkeit eines Individuums und beeinflusst damit direkt die Anzahl der Nachkommen. Bei genetischen Algorithmen muss hingegen eine eigene Fitness definiert werden, also kann die Fitness bei der Lösung eines Optimierungsproblems beispielsweise der Funktionswert der zu optimierenden Funktion sein. Diese künstlich erzeugte Fitness muss wie in der Natur die Überlebenswahrscheinlichkeit der Individuen beeinflussen.

Wesentlicher Vorteil der genetischen Algorithmen ist, dass die wenigen (vereinfachten) Prinzipien natürlicher Evolution (Reproduktion, Crossover und Mutation sowie eine Fitness–Funktion), auf denen sie basieren, ein sehr großes Problemlösungspotenzial besitzen, da sie im Laufe der Zeit immer mehr auf bestimmte Anforderungen und Probleme spezialisierte Individuen und Spezies hervorbringen. Dabei sind genetische Algorithmen insbesondere für komplexe Optimierungsprobleme geeignet. Das Übertragen der Ideen aus der Genetik in andere Gebiete findet nicht nur in der Informatik statt, sondern auch auf solchen Gebieten wie der Soziologie (siehe bspw. [Daw96]).

Evolutionsstrategien

Grundlegend für die Anwendung von genetischen Algorithmen ist also zunächst die Definition einer *Fitnessfunktion*, die jedem Individuum einen Wert zuordnet, der repräsentiert, wie gut das Individuum die zu optimierende Funktion erfüllt. Dieser beeinflusst dann die Überlebenswahrscheinlichkeit des einzelnen Individuums. Diese Fitnessfunktion spiegelt außerdem die jeweilige Anwendungsdomäne wider. Das bedeutet, dass in ihr kodiert ist, wie gut ein Individuum in dieser Domäne die jeweilige Funktion erfüllt. *Reproduktion* ist dann einfach das Vervielfältigen eines Genoms und *Mutationen* sind als "Fehler", die mit einer bestimmten Wahrscheinlichkeit während des Kopiervorgangs auftreten, zu modellieren. Idealerweise soll Reproduktion zusammen mit der Selektion durch die Fitness die Übernahme besonders guter Individuen aus einer Generation in die nächste ermöglichen, während Mutation neue ähnliche Individuen erzeugen soll, so dass immer auch andere, bisher nicht betrachtete Individuen untersucht werden. *Crossover* schließlich stellt das Vermengen des Genoms zweier Individuen dar, aus dem dann ein neues Genom erzeugt wird. Dadurch soll es ermöglicht werden, Teile zweier unterschiedlicher Genome zu kombinieren, so dass zwei gute Teillösungen aus zwei unterschiedlichen Genomen in ein einziges Genom integriert werden können. Genetische Algorithmen könnte man jedoch auch nur mit Mutation als genetischem Operator arbeiten lassen, da dadurch immer wieder neue Individuen entstehen, die dann mit den anderen Individuen verglichen werden können. Mit Crossover ist es zwar auf der einen Seite möglich, durch die Kombination von Teillösungen sehr gute neue Individuen zu erzeugen, auf der anderen Seite ist es aber durchaus wahrscheinlich, dass durch solche Crossover–Operationen auch Teillösungen zerstört werden und sehr schlechte Nachkommen erzeugt werden. Crossover und die Optimierung der Ergebnisse dieses Operators sind dementsprechend eine sehr wichtige Frage in der Forschung im Bereich genetischer Algorithmen.

Trotz der relativ einfachen Prinzipien genetischer Algorithmen gibt es bei der Implementierung zahlreiche Detailfragen zu beantworten. Zunächst ist eine wichtige Frage, wie die Auslese und das Erzeugen der Generationen konkret erfolgen soll. Der einfachste Fall ist die (1+1) Strategie. Dies bedeutet, dass von dem einen Elternindividuum durch Mutation ein Nachkomme erzeugt wird. Diese beiden Individuen werden dann mit Hilfe der Fitnessfunktion bewertet und das fittere Individuum ist dann das Elternindividuum für die nächste Generation. Eine logische Erweiterung dieser Evolutionsstrategie ist die $(1 + \lambda)$–Evolutionsstrategie. Dabei werden dann aus dem Elternindividuum λ Individuen erzeugt, diese werden alle durch die Fitnessfunktion bewertet, und schließlich wird aus diesem Individuen das fitteste als neues Elternindividuum gewählt. Alternativ kann man auch eine $(1, \lambda)$–Evolutionsstrategie verwenden. Diese unterscheidet sich von der $(1+\lambda)$–Strategie dadurch, dass nur eines der Nachkommen das neue Elternindividuum sein kann; das alte Elternindividuum wird aus der Population entfernt. Wenn man statt eines einzigen Elternindividuum mehrere verwendet, kommt man zu den $(\mu+\lambda)$– bzw. (μ, λ)–Evolutionsstrategien. Dabei werden dann aus den μ Eltern durch Mutation und Reproduktion λ Kinder erzeugt, die dann jeweils einer Mutation unterzogen werden. Natürlich muss dann bei einer (μ, λ)–Evolutionsstrategie $\lambda > \mu$ gelten.

Es gibt nun noch die Möglichkeit, Rekombinationsoperatoren wie z.B. das Crossover zu definieren. Rekombination bedeutet dabei, dass die Genome meh-

rerer Individuen kombiniert werden, um ein neues Genom zu erhalten — Crossover ist also eine Ausprägung eines solchen Operators. Allgemein kann man dann von $(\mu/\rho, \lambda)$–Evolutionsstrategien sprechen. ρ kennzeichnet dabei die Größe der Subpopulation, auf welcher der Rekombinationsoperator arbeitet. Im Fall von Crossover, also dem Austausch von Teilen des Genoms zweier Individuen, würde $\rho = 2$ gelten. Die Evolutionsstrategie folgt dann entsprechend einer $(\mu/2, \lambda)$–Strategie. Konkret bedeutet dies, dass zunächst aus der μ Individuen großen Elterngeneration $2 \cdot \lambda$ Individuen erzeugt werden. Auf jeweils zwei von diesen wird dann der Rekombinationsoperator angewendet, der dann ein neues Individuum ergibt. Dieses wird dann noch der Mutation unterzogen. So ergeben sich wieder λ Individuen, von denen die μ fittesten dann die nächste Elterngeneration stellen. Würden hier nicht nur die fittesten Individuen aus der Nachkommengeneration in Betracht gezogen, sondern auch die alte Elterngeneration, wäre man bei einer $(\mu/2 + \lambda)$–Strategie.

Es stellt sich nun die Frage, wie man von den μ Eltern auf die λ Kinder kommt. Üblicherweise verwendet man hierzu eine fitnessproportionale Selektion (*Roulette–Wheel*–Selektion). Jedes einzelne Individuum aus der Elterngeneration bekommt eine bestimmte Wahrscheinlichkeit zugeordnet, die direkt proportional zur Fitness dieses Elternindividuums ist. Basierend auf diesen Wahrscheinlichkeiten wird für jedes der μ Individuen entschieden, von welchem Elternindividuum es abstammen soll. Neben der Reduzierung der μ Individuen der Kindergeneration auf die λ Individuen der Elterngeneration findet hier also indirekt eine zweite Selektion statt.

Eine tiefer gehende, theoretisch fundierte Erklärung dafür, warum solche Evolutionsstrategien bei komplexen Optimierungsaufgaben sinnvoll einsetzbar sind, bietet das sogenannte *Schematheorem*. Dieses Theorem führt zunächst den Begriff des *Schemas* als eine Untermenge einer Population ein. Es werden dann solche Schemata unterschieden, die Individuen mit vorteilhaften Merkmalen umfassen und solche mit weniger vorteilhaften Merkmalen. Dann wird dargestellt, wie die einzelnen Schemata abhängig von der Güte der jeweiligen Individuen Crossover und Mutation "überleben". Dies ergibt dann die eigentliche Aussage des Schematheorems, nämlich dass Schemata mit guten Merkmalen in der Population sehr stark wachsen und somit der Suchraum effizient durchsucht wird. Eine detaillierte formale Ausarbeitung dieses für genetische Algorithmen sehr grundlegenden Theorems findet sich u.a. in [Jac97, S. 190ff].

Genetische Programmierung

Bei genetischer Programmierung handelt es sich um ein Verfahren, bei dem die Prinzipien genetischer Algorithmen nicht dazu verwendet werden, bestimmte Daten zu finden, die eine Funktion möglichst optimal erfüllen, sondern hier werden Algorithmen bzw. Programme "gezüchtet", die eine bestimmte Funktionalität aufweisen. Typischerweise wird dabei allerdings nicht versucht, Algorithmen für klassische Probleme wie Suche, Sortieren usw. zu finden, sondern es geht meistens um Algorithmen für komplexe Problemstellungen. Dieses Verfahren wurde von Koza zum ersten Mal erwähnt [Koz92]. Mittlerweile gibt es eine große Anzahl von Publikationen in diesem Feld. Die wichtigste Grundlage für die vorliegende Arbeit war [BNKF98], worin eine gute Einführung in das Feld der genetischen Programmierung gegeben wird.

Genetische Programmierung kann im wesentlichen als eine Anwendung von

genetischen Algorithmen auf andere Datenstrukturen angesehen werden. Damit bleiben also die wesentlichen Prinzipien — Mutation, Crossover und Reproduktion — erhalten, aber es stellt sich die Frage, wie man die Datenstrukturen, also in diesem Falle Programme bzw. Algorithmen, darstellen kann. Dabei muss vor allem beachtet werden, dass die Datenstrukturen genetische Operatoren zulassen müssen. Eine Veränderung an dem Programmcode muss also möglichst immer wieder zu einem gültigen Programm führen. Dieses ist bei klassischen Programmiersprachen meist nicht der Fall, da bereits die Änderung eines einzigen Zeichens im Programmcode nicht nur einfach die Semantik des Programms ändern kann, sondern das ganze Programm syntaktisch ungültig werden lassen kann. Man muss also alternative Darstellungsweisen für die Algorithmen finden.

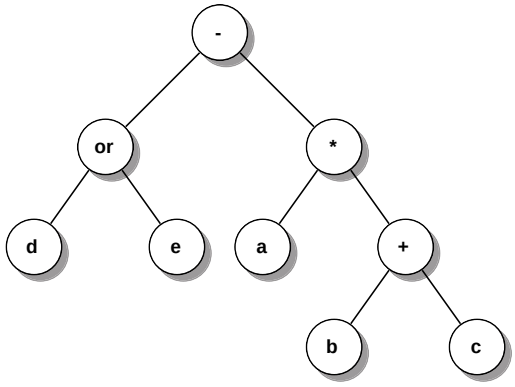


Abbildung 6.18: Darstellung eines Genoms als Baum

Als erste Alternative bietet sich dabei eine graphenbasierte Darstellungsweise an. So kann man z.B. eine Darstellung des Programmcodes als Bäume wählen. Abbildung 6.18 zeigt ein Beispiel für einen solchen Baum, der in diesem Fall die Berechnungsvorschrift $(d \text{ or } e) - (a \cdot (b + c))$ darstellt. Diese ähneln dann den *Parsetrees*, die ein Compiler oder Interpreter als Ergebnis des Parsevorgangs eines klassischen Quellcodes anlegt. Bei dieser Darstellungsweise sind die inneren Knoten des Baumes die einzelnen Operatoren und die Blätter die Variablen oder Konstanten. Mögliche Mutationsoperatoren wären also an den Blättern eine beliebige andere Konstante oder Variable einzusetzen oder an einem inneren Knoten einen anderen Operator einzusetzen. Auf jeden Fall bleibt der Baum dann aber die Darstellung eines gültigen Algorithmus. Neben solchen einfachen Mutationsoperatoren sind auch komplexere Operatoren dankbar wie z.B. Permutation, also der Tausch der Kinderknoten eines Knotens. Es gibt auch Mutationsoperatoren, die eine Auswirkung auf ganze Subbäume haben: Man kann einen Subbaum durch einen einfachen Knoten oder einen neuen, zufälligen Subbaum ersetzen.

Es gibt auch zahlreiche verschiedene Möglichkeiten, einen Crossoveroperator zu definieren. Man kann dabei zunächst einfach den Austausch eines Teils des Graphen eines Individuums mit einem Teil eines Graphen eines anderen Individuums implementieren. Komplexere Crossover–Operatoren könnten beispielsweise auch die Position des Subbaums im Baum mit einbeziehen und nur Subbäume an identischen oder zumindest ähnlichen Stellen gegeneinander austauschen. Eine weitere Möglichkeit ist es, Crossover nicht zwischen zwei ver-

schiedenen Individuen zu definieren, sondern auf einem einzigen. In diesem Fall werden zwei Subbäume des Individuums gegeneinander ausgetauscht. Allerdings widerspricht dies der eigentlichen Motivation, einen Crossover–Operator einzusetzen. Der Crossover–Operator soll eigentlich dazu dienen, gute Teillösungen in der Population zu verbreiten. Dieses findet nicht statt, wenn der Crossover–Operator nur auf einem Individuum arbeitet.

Neben diesen Bäumen gibt es auch andere Graphenarten als Repräsentationen der Genome. Einige Repräsentationen erlauben neben Bäumen auch Graphen, die Zyklen enthalten. Dabei wird während der Ausführung der Graph durchlaufen, und an jedem Knoten wird eine Aktion ausgelöst und dann an Hand einer Verzweigungsentscheidung eine der herauslaufenden Kanten gewählt. Eine weitere Möglichkeit ist es, den Graphen selber tatsächlich nur als Genotyp zu betrachten und daraus einen Phänotyp zu entwickeln. Dieses setzt die Analogie zur Natur noch weiter fort: Hier wird auch aus dem Genotyp (also der DNA) ein komplettes Individuum (also der Phänotyp) erzeugt. Bei genetischer Programmierung repräsentiert der Graph (Genotyp) dann eine "Anleitung" zum Erzeugen des eigentlichen Programms (Phänotyp).

```
b=b+c
a=a*b
d=d or e
a=d - a
```

Abbildung 6.19: Ein lineares Genom in einem Zwei–Adress–Maschinencode

Eine andere Möglichkeit, die Genome darzustellen, ist ein lineares Genom. Abbildung 6.19 zeigt ein entsprechendes Beispiel, das ebenfalls die Berechnungsvorschrift $(d \text{ or } e) - (a \cdot (b + c))$ darstellt. Diese Repräsentation lehnt sich an normalen Programmcode, insbesondere Assembler–Sprachen, an. Im wesentlichen wird also ein bestimmter Satz von Operatoren zur Verfügung gestellt, die wieder mit Konstanten und Variablen parametrisiert werden können. Diese werden jedoch nicht in einem Baum angeordnet, sondern linear wie in einem normalen Programmcode. Allerdings wird durch die sehr einfache Syntax und die kleine Anzahl von möglichen Operatoren verhindert, dass Mutationen auf dem Programmcode zu ungültigen Programmen führen. Mutation kann dabei das Ändern eines Operators, aber auch das Ändern der Variablen oder Konstanten des Operators sein. Crossover kann als Austausch eines einzelnen oder mehrerer Operatoren mit den zugehörigen Parametern definiert werden.

Genetische Programmierung stellt also prinzipiell eine Unterklasse der genetischen Algorithmen dar. Während genetische Algorithmen allgemein versuchen, beliebige Objekte in einem bestimmten Kontext zu optimieren, versucht genetische Programmierung, automatisch Algorithmen bzw. Programme für bestimmte Einsatzgebiete zu finden. Entsprechend besteht die wesentliche Frage bei genetischer Programmierung darin, wie die Algorithmen dargestellt werden können. Dabei kann man lineare und graphenbasierte, insbesondere baumbasierte Repräsentationen unterscheiden. Beide lassen sich direkt oder indirekt auf die normale Repräsentation von Programmen als Programmtexte zurückführen.

6.3.2.2 Anwendbarkeit für Verhandlungsstrategien

Wie bereits am Ende von Abschnitt 6.3.1 erwähnt, wird in dieser Arbeit von der Hypothese ausgegangen, dass genetische Algorithmen, insbesondere genetische Programmierung, auf Grund ihrer "natürlichen" Anpassungsfähigkeit eine gute Alternative zu analytischen Verfahren im Hinblick auf die Realisierung von Verhandlungsstrategien darstellen. Inwieweit diese Hypothese tatsächlich Bestand hat, kann bei dem aktuellen Stand der Forschung noch nicht klar beantwortet werden, da es bisher nur vereinzelte Arbeiten gegeben hat, die sich mit der Anwendung von genetischen Algorithmen auf konkrete Verhandlungsprobleme beschäftigen. Einen der konkretesten Versuche in dieser Richtung stellt die Dissertation von J. Oliver [Oli96] dar, deren Ansatz und wesentliche Ergebnisse im folgenden zusammengefasst werden.

In der Arbeit von Jim Oliver wurden die Prinzipien von genetischen Algorithmen auf relativ einfache, bilaterale Verhandlungsszenarien angewendet. Wesentliche Voraussetzung ist dabei zunächst, dass jeder einzelne Agent eine Nutzenfunktion hat, die jedem möglichen Verhandlungsergebnis einen Wert zuordnet, der ein Mass dafür ist, wie gut das Ergebnis für den jeweiligen Agenten ist. Basierend auf dieser Nutzenfunktion werden dann Verhandlungsstrategien gebildet. Eine solche Strategie ist in Abbildung 6.20 dargestellt.

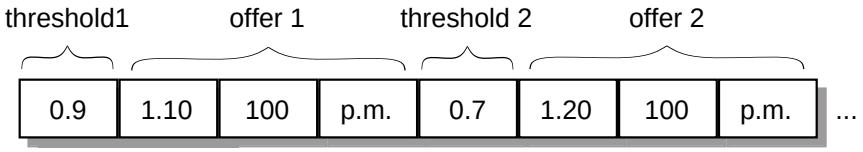


Abbildung 6.20: Beispiel eines linearen Genoms als Verhandlungsstrategie aus [Oli96]

Ein Angebot, das von dem Verhandlungspartner gemacht worden ist, wird zunächst mit Hilfe der Nutzenfunktion bewertet. Der Agent vergleicht dann anschließend den ermittelten Nutzenwert mit dem Schwellwert 1 (*threshold 1*). Wenn der Nutzenwert diesen überschreitet, nimmt er das Angebot an, sonst macht er ein Gegenangebot (in diesem Fall als Tupel (1.10, 100, p.m.) dargestellt). Dieses Gegenangebot wird dann entweder von der Gegenseite akzeptiert, oder der Agent bekommt ein neues Angebot, das er dann wieder mit Hilfe der Nutzenfunktion bewertet, und dann, falls es den Schwellenwert 2 übertrifft, annimmt oder sonst wieder mit einem Gegenangebot beantwortet. Dieses Vorgehen wird für insgesamt drei Runden durchgeführt, d.h. jeder Agent hat die Möglichkeit, 3 Angebote vorzulegen. Wenn die Agenten dann nicht zu einer Einigung gekommen sind, wird die Verhandlung als erfolglos abgebrochen.

Um diese Strategien nun durch genetische Algorithmen zu optimieren, sieht Oliver eine Population von 20 Agenten vor, die zunächst jeder eine zufällige Strategie zugewiesen bekommen. Anschließend müssen die Agenten gegeneinander in Verhandlungen antreten. Dabei werden 20 Verhandlungen durchgeführt, so dass durchschnittlich zwar jeder Agent an zwei Verhandlungen teilnimmt, aber man kann dennoch nicht sicher sein, dass wirklich jeder Agent an mindestens einer solchen Verhandlung teilnimmt. Danach werden die Strategien der Agenten dann den üblichen genetischen Operatoren unterworfen und eine neue

Generation von Strategien gebildet. Der Vorgang wiederholt sich dann für insgesamt 20 Generationen. Um Mittelwerte bilden zu können, wird das gesamte Experiment dann zehnmal mit jeweils neuen zufälligen Strategien wiederholt.

Der verwendete genetische Algorithmus implementiert dabei das Crossover so, dass zwei Eltern ausgewählt werden und dann ein Kind gebildet wird, bei dem jeder Teil des Genoms von einem der beiden Eltern kommen kann. Der Elternteil, von dem jeweils übernommen wird, wird dabei für jedes Genomteil neu und unabhängig zufällig ausgewählt. Bei der Mutation wird einfach ein Wert in dem Genom durch einen neuen zufälligen Wert ersetzt. Bei der Übernahme in eine neue Generation wird eine Strategie mit der Wahrscheinlichkeit 0,5 einem Crossover und mit der Wahrscheinlichkeit 0,05 einer Mutation unterzogen.

Oliver wendet diese Verfahrensweisen auf mehrere unterschiedliche Szenarien an. Eine Gemeinsamkeit dieser Szenarien ist, dass es immer um bilaterale Verhandlungen geht und auch immer um mehrere verschiedene Verhandlungsgegenstände wie zum Beispiel Preis, Farbe und Lieferdatum. Daher sind die Angebote auch immer Tupel, die den Wert von jedem dieser Verhandlungsgegenstände repräsentieren. Oliver hat in seiner Arbeit diese Vorgehensweise auf verschiedene Szenarien angewendet. Diese definieren dann jeweils eine Nutzenfunktion für die beiden verhandelnden Agenten. Folgende Szenarien wurden verwendet:

No Conflict Hierbei haben die Agenten identische Nutzenfunktionen, d.h. für beide ist das optimale Verhandlungsergebnis identisch.

Pure Distributive Bargaining In diesem Szenario ist jeder Vorteil für den einen Agenten ein Nachteil für den anderen. Dies ähnelt dem Teilen einer Torte: Wenn einer mehr bekommt, bekommt der andere automatisch weniger.

Simple Integrative Bargaining Hier sind die Nutzenfunktionen zwar wieder entgegengesetzt, aber unterschiedliche Verhandlungsgegenstände sind für die Verhandlungspartner unterschiedlich wichtig. Hier gilt es also, einen Kompromiss zu finden, der die Prioritäten der einzelnen Verhandlungspartner möglichst gut erfüllt.

Divorce Hier geht es wieder um verschiedene Verhandlungsgegenstände, die auch wieder unterschiedlich priorisiert sind. Ein besonderes Merkmal dieses Szenarios ist es, dass es 3125 verschiedene mögliche Verhandlungsergebnisse gibt, und dieses Szenario somit außerordentlich komplex ist. Im einzelnen geht es dabei um die verschiedenen Besitztümer, die das Paar während seiner Ehe angesammelt hat. Die einzelnen Punkte sind dabei die Betreuung der Kinder, das Bargeld, der Grundbesitz, das Vermögen und das Haus. Bei jedem dieser Punkte gibt es in diesem Modell fünf verschiedene Vereinbarungen, so dass sich insgesamt $5^5 = 3125$ Möglichkeiten für einen Kompromiss ergeben.

International Business Transaction Hier gibt es "nur" 256 verschiedene mögliche Verhandlungsergebnisse. Es wird nämlich über den Preis, die Lieferfrist, die Währung und den Gerichtsstandort verhandelt. Bei jedem dieser Punkte wird von vier möglichen Ergebnissen ausgegangen, so daß

sich insgesamt $4^4 = 256$ Möglichkeiten ergeben. Ein Vorteil dieses Szenarios ist, dass Daten darüber vorliegen, wie gut Menschen in dieser Verhandlung sind. Außerdem können hier unterschiedliche Nutzenfunktionen für die Agenten verwendet werden. Dabei sind die Prioritäten der Agenten immer gleich, d.h. für den Käufer ist es immer am besten, in Dollar zu zahlen, dann kommen andere harte Währungen, der Euro und schließlich die ungarische Währung. Für den Verkäufer hingegen ist die ungarische Währung immer am besten, dann kommt der Euro, Dollar und schließlich andere harte Währungen. Während in einem Szenario der Käufer dem Dollar einen Nutzen von 0,1 zuordnet und dann den anderen harten Währungen 0,02, dem Euro 0,01 und der ungarischen Währung 0,0, können diese konkreten Werte in einem anderen Szenario 0,03, 0,02, 0,01 und 0,0 sein. Es bleibt also die Rangfolge erhalten, aber die konkreten Werte ändern sich.

Anhand dieser Szenarien versucht Oliver fünf anspruchsvolle Fragen zu beantworten. Die konkreten Fragestellungen wurden wie folgt bearbeitet:

1. ***Können Agenten gute Verhandlungsergebnisse erzielen?*** Um diese Frage zu beantworten, werden die Strategien, die mit dem genetischen Algorithmus optimiert wurden, mit zufälligen Strategien verglichen. Im Ergebnis sind die Agenten, deren Strategien mit genetischen Algorithmen optimiert wurde, tatsächlich besser als die zufälligen Strategien. Damit kann man davon ausgehen, dass die genetisch optimierten Strategien gute Ergebnisse in dem Sinne erreichen, dass sie besser als zufällige Strategien sind.

2. ***Können Agenten generische Strategien lernen?*** Hierfür wird das Szenario der International Business Transaction verwendet. Dabei werden fünf verschiedene Nutzenfunktionen für die Käufer aufgestellt, welche die entsprechenden Rangfolgen einhalten, aber unterschiedliche konkrete Nutzenwerte erzeugen. Es wird dann einmal der genetische Algorithmus für jeden dieser Käufertypen mit jeweils eigenen Verkäufern durchgeführt. Dabei werden sowohl Käufer als auch Verkäufer durch den genetischen Algorithmus optimiert. Dies entspricht also dem herkömmlichen Szenario, in dem beide Agenten mit genetischen Algorithmen optimiert werden und fixe Nutzenfunktionen haben, d.h. jeder Agent kann sich genau auf die Nutzenfunktion des anderen Agenten optimieren.

   In einem zweiten Durchgang werden die so optimierten Käuferstrategien noch einmal verwendet, aber es wird eine komplett neue Population von Verkäufern erzeugt. Die Verkäufer müssen in diesem Lauf abwechselnd mit unterschiedlichen Käuferstrategien aus diesem Pool verhandeln, die natürlich auch unterschiedliche Nutzenfunktionen haben. Somit kann sich der Verkäufer in diesem zweiten Experiment nicht mehr individuell auf eine einzige Nutzenfunktion hin optimieren, sondern muss mit verschiedenen Nutzenfunktionen zurecht kommen. Am Ende werden dann die Ergebnisse des ersten Laufs, in dem der Verkäufer auf einen Käufer optimiert wird, mit denen des zweiten Laufs verglichen, in dem der Verkäufer mit mehreren unterschiedlichen Käufern zurecht kommen muss.

Eine statistische Auswertung ergibt, dass die generischen Strategien nicht nur besser als zufällige Strategien sind, sondern sogar nicht unterscheidbar von den speziellen Strategien.

3. ***Entwickeln Agenten effektive Strategien?*** Mit dieser Frage soll herausgefunden werden, ob Agenten Strategien entwickeln, die für sie selber gute Ergebnisse erzielen. Zur Beantwortung dieser Frage wurde zunächst überprüft, ob die Agenten mit der Zeit einen besseren Nutzen erreichen. Dies lässt sich nicht zeigen, da die Werte zwar steigen, aber ein starkes Rauschen haben, so dass zum Teil auch wieder Verschlechterungen feststellbar sind. Außerdem wurden die Ergebnisse der Verhandlungen auch mit dem Mittelwert aller möglichen Verhandlungsergebnisse verglichen, und dabei wurde festgestellt, dass immer wieder Ergebnisse erzielt werden, die schlechter als der Durchschnitt aller möglichen Verhandlungsergebnisse sind.

   Als letztes wurde überprüft, ob in distributiven Spielen mehr Verhandlungen ergebnislos beendet werden als bei integrativen. Distributive Spiele sind solche Szenarien, in denen der Vorteil für einen Verhandlungspartner auch gleichzeitig ein Nachteil für den anderen Verhandlungspartner bedeutet, also zum Beispiel das das Szenario "Pure Distributive Bargaining". Integrative sind solche Spiele, bei denen ein Kompromiss zwischen den Prioritäten der beiden Verhandlungspartner gefunden werden muss, also beispielsweise die "Simple Integrative Bargaining", "Divorce" und "International Business Transaction" Szenarien. Die höhere Rate von ergebnislosen Verhandlungen sollte sich bei distributiven Szenarien ergeben, da es bei einem distributiven Spiel günstiger sein kann, die Verhandlung ergebnislos abzubrechen als dem Verhandlungspartner ein gutes Verhandlungsergebnis zu ermöglichen, was bei distributiven Spielen automatisch für einen selbst ein schlechtes Verhandlungsergebnis ist. Dies wurde schließlich bestätigt — wenn auch je nach Szenario unterschiedlich stark. Damit muss allerdings insgesamt die Frage, ob Agenten wirklich effektive Strategien entwickeln, zumindest zum Teil negativ beantwortet werden.

4. ***Kann man die Agenten ausbeuten/ausnutzen?*** Hier geht es um die Frage, ob man einen Agenten ausbeuten kann, dessen Strategie mit Hilfe eines genetischen Algorithmus optimiert wird. Dazu lässt man solche Agenten mit Agenten verhandeln, die nur Angebote annehmen, die mindestens 70% des maximalen Nutzens haben. Hier zeigt sich, dass Agenten tatsächlich unter diesen Bedingungen ausgenutzt werden können. Außerdem zeigt sich, dass die Summe der Nutzen beider Agenten besser wird.

   Als ein weiterer Fall, der auch etwas allgemeiner ist, wird untersucht, wie stark beim Akzeptieren von Angeboten der Schwellenwert der Agenten überschritten wird. Wenn dieser Abstand zwischen Schwellenwert und Nutzen des Verhandlungsergebnisses groß ist, kann der Agent ausgebeutet werden, da er auch ein schlechteres Ergebnis akzeptiert hätte. Der Abstand zwischen dem Nutzen des Verhandlungsergebnisses und dem *threshold* sollte sinken und idealerweise Null erreichen. Der Agent "weiss" dann praktisch, was er erwarten kann und kann dann entsprechend nicht mehr ausgebeutet werden. Hier zeigte sich zwar ein Sinken der Differenz, aber Null wurde dabei noch nicht einmal annähernd erreicht.

5. ***Wie gut verhandeln Agenten im Vergleich zu Menschen?*** Hier wurden die Ergebnisse von verhandelnden Agenten mit den von Menschen erreichten Ergebnissen verglichen. Zum Einsatz kam dabei das "Divorce" und das "International Business Transaction" Szenario, bei dem einige Daten über die Ergebnisse bei menschlichen Verhandlungen vorliegen. Es wurden zunächst einmal die erzielten Nutzenwerte für die jeweiligen Agenten und die Summe dieser Nutzenwerte mit den Werten der Menschen verglichen. Hier ergab sich kein signifikanter Unterschied, obwohl die Menschen etwas besser waren. Bei der Anzahl der ergebnislos abgebrochenen Verhandlungen sind die Menschen je nach Szenario besser oder schlechter als die Agenten. Aber bei der Anzahl der Verhandlungen, in denen in den entscheidenden Bereichen die optimalen Kompromisse gefunden wurden, sind die Agenten deutlich besser als die Menschen. Das bedeutet, dass sie eher dazu in der Lage sind, die entscheidenden Punkte herauszufinden und dort Kompromisse zu erzielen.

Insgesamt kann Olivers Arbeit als ein erster Schritt gesehen werden, genetischen Methoden überhaupt auf praxisnahe Weise für Verhandlungen einzusetzen. Seine Arbeit ist im Gegensatz zu vielen anderen spieltheoretischen Aufgabestellungen wie etwa das berühmte *Iterated Prisoner's Dilemma* [Pou93] deutlich von realen Verhandlungsszenarien inspiriert. Der wesentliche Punkt, der hier zur Frage stand, ist, ob Agenten, die auf genetischen Algorithmen basieren, überhaupt Verhandlungsstrategien lernen können. Und dass diese grundlegende Frage insgesamt positiv beantwortet wurde, ist ein guter Grund dafür, den Ansatz von Oliver weiterzuentwickeln.

Allerdings unterliegen Olivers Experimente — und damit die von ihm erzielten Ergebnisse — auch klaren Einschränkungen. Die erste ist, dass nur sehr kleine Generationen und nur wenig Durchläufe der Berechnungen vorgenommen wurden. So ist es beispielsweise gar nicht sicher, ob in jeder Generation auch wirklich jeder Agent mindestens einmal an einer Verhandlung teilnimmt. Da jedoch bei der statistischen Auswertung dies zumindest zum Teil wieder berücksichtigt wurde, kann man an der Arbeit zumindest auf dieser Basis nur eingeschränkte Kritik üben.

Eine sehr wichtige Frage ist jedoch, ob in Olivers Arbeit wirklich das Lernen von Strategien bearbeitet wird. Ein ganz wesentlicher Aspekt von Strategien sollte die Generizität sein. Eine Verhandlungsstrategie gibt einem — ähnlich einem Algorithmus — eine allgemeine Vorschrift an die Hand, wie man mit einem Verhandlungspartner umgehen kann, so dass die eigenen Ziele beachtet werden. Sie sollte also idealerweise unabhängig von dem jeweiligen Verhandlungspartner sein. In Olivers Arbeit werden jedoch die Agenten immer auf bestimmte Verhandlungspartner trainiert. Oliver selber gibt zwar an, dass Agenten generische Strategien lernen können, aber auch dort verwendet er nur Agenten mit ähnlichen Nutzenfunktionen als Verhandlungspartner. Überspitzt könnte man sagen, dass Olivers "ideale" Strategie aus genau einem Angebot besteht: dem optimalen Kompromiss. Da meistens auch die Strategien beider Agenten mit Hilfe eines genetischen Algorithmus optimiert werden, würde sich die Verhandlung dann letztendlich auf ein Angebot (den optimalen Kompromiss) und das anschließende Akzeptieren des Angebots reduzieren. In diesem Zusammenhang noch von Verhandlungen und Strategien zu sprechen, erscheint schon fast übertrieben.

Weil also keine absolut generischen Strategien entstehen, ergibt sich bei Oli-

ver ein weiteres Problem: Man kann unter diesen Voraussetzungen kaum in realen Umgebungen mit Agenten arbeiten, da zunächst eine Trainingsphase notwendig ist und zwar jedes Mal, wenn sich die Strategie der Verhandlungspartner oder ihre Nutzenfunktion ändert. Oliver hätte dabei insbesondere Fragen nach der Länge dieser Trainingsphasen und den dabei entstehenden ungünstigen Verhandlungsergebnissen nachgehen sollen. Schließlich sind Olivers lineare Verhandlungsstrategien kritikwürdig, da sie keine Entscheidungen unterstützen. Diese wären wahrscheinlich für generische Strategien auch eine wesentliche Voraussetzung. Außerdem bearbeitet Oliver in seiner Arbeit nur bilaterale Verhandlungen, die ein sehr einfaches Verhandlungsszenario darstellen.

6.3.2.3 Realisierung des GA–Frameworks

Im Rahmen dieser Arbeit ist ein Framework zur Anwendung von genetischen Algorithmen (kurz: GA–Framework) entstanden. Auch bei der Entwicklung dieses Frameworks steht die Generizität der Mechanismen im Vordergrund des Entwurfs und der Implementierung (vgl. Abschnitt 1.4.1 und 4.1.1), d.h. das Framework soll nicht nur zur Realisierung von Verhandlungsstrategien dienen, sondern prinzipiell auch für andere Anwendungsdomänen nutzbar sein. Ähnlich wie beim Plug-in–Mechanismus (vgl. Abschnitt 5.3.1.2) kennzeichnet der Begriff des Frameworks hier eine Sammlung von generischen Klassen, die ein Gerüst bilden, in das der Entwickler nur noch an bestimmten Stellen spezifische Anwendungsklassen einfügen muss, um eine fertige Applikation zu erzeugen. Im folgenden wird die Konzeption und das Implementationsdesign des GA–Frameworks beschrieben.

Endliche Automaten als Datenstruktur

Ein entscheidender Aspekt beim Einsatz von genetischen Algorithmen ist die Wahl der Datenstrukturen, auf den die genetischen Verfahren operieren. Wie im letzten Abschnitt (6.3.2.2) beschrieben wurde, gibt es bereits einen Ansatz, genetische Algorithmen mit linearen Genomen für automatische Verhandlungen zu verwenden. Im Rahmen der vorliegenden Arbeit sollen jedoch endliche Automaten als Basisdatenstruktur verwendet werden.

Hauptgrund für den Einsatz von endlichen Automaten ist, dass sie im Gegensatz zu den linearen Genomen aus der Arbeit von Oliver Entscheidungen innerhalb der Strategien erlauben. Ein lineares Genom, wie Oliver es vorschlägt, durchläuft lediglich seinen vordefinierten Weg, ohne dass es möglich ist, basierend auf den Angeboten des Verhandlungspartners Entscheidungen zu fällen und abhängig von diesen Verhandlungsangeboten unterschiedliche Gegenangebote zu machen. Man könnte statt endlicher Automaten auch kompliziertere Strukturen verwenden, die eine noch größere Ausdrucksmächtigkeit haben. Es gilt hier jedoch, eine sinnvolle Abwägung zwischen der Ausdrucksmächtigkeit des verwendeten Modells und der damit einhergehenden Vergrößerung des Suchbereiches zu treffen.

Es stellt sich nun die Frage, welche Art von endlichen Automaten betrachtet wird und welche Anforderungen berücksichtigt werden müssen. Endliche Automaten akzeptieren üblicherweise reguläre Mengen. Bei der Anwendung auf Verhandlungsszenarien reicht es nicht aus, einfach eine bestimmte Menge zu akzeptieren, sondern jede Eingabe (=Verhandlungsangebot) muss mit einer Ausgabe

beantwortet werden. Also muss ein endlicher Automat mit Ausgabe verwendet werden. Diese unterscheidet man in Mealy– und Moore–Automaten [HU79]. Mealy–Automaten haben eine bestimmte Ausgabe bei jeder Kante, Moore–Automaten bei jedem Zustand. Beide sind äquivalent zueinander: Für jeden Automaten des einen Typs kann man effektiv einen Automaten des anderen Typs konstruieren, der genau dasselbe leistet. In der vorliegenden Arbeit ist die Wahl jedoch auf Mealy–Automaten gefallen, da Mutationen wie das Ändern der Ausgabe an einer Kante nur zu einer relativ kleinen Änderung führt. Bei den Verhandlungsstrategien bedeutet dies, dass nur bei einem bestimmten Zustand mit einer bestimmten Eingabe eine andere Ausgabe erzeugt wird. Würden Moore–Automaten verwendet und damit die Ausgabe an einen Zustand gebunden, würde in vielen Situationen ein anderes Gegenangebot erzeugt und dies könnte in der einen oder anderen Situation ein Vorteil sein. In vielen Situationen wird die veränderte Ausgabe aber wahrscheinlich negative Folgen haben. Es erscheint also sinnvoll, die Ausgabe an die Kanten zu binden, so dass der Mutationsoperator eine feinere Granularität bekommt.

Zusätzlich sollen die verwendeten Automaten noch folgende Eigenschaften haben:

1. **Einen Anfangszustand**: Damit ist der Start der Strategie deterministisch festgelegt und die Berechnung der Fitnesswerte wird somit vereinfacht, weil die Strategie sich einheitlicher verhält. Es wird auch klar werden, dass ein Anfangszustand eine wesentliche Erleichterung für die Definition des Crossover–Operators ist.

2. **Keinen Endzustand**: Das Ende der Verhandlung sollte stattdessen durch eine bestimmte Ausgabe oder durch die Verhandlungskontrolle erfolgen.

3. **Vollständigkeit**: Es muss für jeden Zustand und jedes Eingabesymbol mindestens eine Kante geben. Anderenfalls wäre es denkbar, dass der endliche Automat während der Verhandlung anhalten muss, da er ein Eingabesymbol erhält, dass er nicht verarbeiten kann.

4. **Determinismus**: Für jeden Zustand und jedes Eingabesymbol darf es maximal eine Kante geben. Dies führt dazu — wie schon das Vorhandensein nur eines Anfangszustandes — dass die Strategien sich einheitlicher verhalten und somit eine Fitnessberechnung einfacher beziehungsweise aussagekräftiger wird[7].

Die letzten beiden Eigenschaften führen zusammen zu der Voraussetzung, dass es für jeden Zustand und jedes Eingabesymbol genau eine Kante gibt. Die genetischen Operatoren dürfen diese Eigenschaften des endlichen Automaten nicht verletzen. Es darf also durch die genetischen Operatoren kein Automat entstehen, der die hier definierten Eigenschaften nicht hat.

Nachdem die Struktur der hier verwendeten endlichen Automaten geklärt ist, müssen nun die einzelnen genetischen Operatoren auf diesen Strukturen definiert werden. *Reproduktion* ist trivial: Der Automat muss nur kopiert werden. Für *Mutation* gibt es verschiedene Möglichkeiten. Der hier verwendete Mutationsoperator wählt zufällig eine dieser Möglichkeiten und wendet diese dann auf den endlichen Automaten an. Die Möglichkeiten sind im einzelnen:

[7] Bei deterministischen Automaten führt jede Verhandlung zwischen zwei bestimmten Automaten immer zu demselben Ergebnis.

- Ändern des Ziels / Ursprungs einer Kante
- Ändern des Ausgabesymbols / Eingabesymbols einer Kante
- Zustand hinzufügen / löschen
- Kante hinzufügen / löschen

Um die Eigenschaften des endlichen Automaten, insbesondere Determinismus und Vollständigkeit, bei der Anwendung dieser verschiedenen Operatoren zu bewahren, muss für die einzelnen Operatoren zusätzlicher Aufwand betrieben werden, der auf den ersten Blick nicht offensichtlich ist. Wenn beispielsweise der Ursprung oder das Eingabesymbol einer Kante geändert wird, haben einige Eingabesymbole eines bestimmten Zustands keine zugewiesenen Kanten mehr. In diesem Fall müssen andere Kanten diese Lücken auffüllen. Eine weitere Anomalie, die durch Mutation entstehen kann, ist, dass ein Zustand, der einem Automaten hinzugefügt wird, zunächst unerreichbar ist. Erst durch weitere Mutationen, also beispielsweise das Ändern des Ziels einer Kante oder das Hinzufügen einer Kante, wird dieser neue Zustand tatsächlich in den Automaten integriert.

Gemäß der Definition aus [BNKF98] sind die hier definierten Operationen *Punktmutationen*, da sie nur einen atomaren Teil (Zustand oder Kante) des Graphen ändern, also zum Beispiel der Kante neue Symbole zuordnen oder ihr einen anderen Ursprung oder ein anderes Ziel geben. Obwohl es denkbar wäre, auch Mutationen zu definieren, die auf ganzen Teilgraphen arbeiten, wurde dies nicht realisiert. Der Grund dafür liegt zum einen darin, dass große Änderungen mit einer großen Wahrscheinlichkeit relativ schlechten Nachwuchs erzeugen. Zum anderen werden bei Anwendung des Crossover–Operators schon große Teile des Graphen geändert, so dass durch diesen Operator bereits genug Änderungen auf großen Teilen des Graphen vorgenommen werden.

Beim *Crossover–Operator* sollen hauptsächlich gute Teile des Genoms eines Individuums in der Population verbreitet werden. Durch die Kombination vieler guter Teillösungen durch den Crossover–Operator soll letztendlich dann auch eine gute Gesamtlösung entstehen. Sinn des Crossover–Operators besteht allerdings allein darin, Teile eines Genoms überhaupt in der Population verteilen. Es bleibt dem Zufall überlassen, ob dabei gute oder schlechte Teile verbreitet werden; schlechte Teile müssen also im Lauf der Zeit wieder durch die Fitnessfunktion heraussortiert werden. Durch Crossover soll das Genom zweier Individuen gemischt werden. Bei endlichen Automaten wurde dies folgendermaßen erreicht:

1. Selektiere zwei Automaten für das Crossover (*Eltern* und teile die Menge der Zustände der Eltern in zwei Untermengen (siehe Abbildung 6.21)

2. Zustände, die eine Kante in die andere Untermenge haben, werden als *Outputs* bezeichnet. Zustände, die Endzustand einer Kante mit Ursprung in der anderen Untermenge haben, werden *Inputs* genannt.

3. Erzeuge die Zustandsmengen der neuen Automaten (*Kinder*): Sie bestehen jeweils aus der Zustandsuntermenge mit dem Startzustand eines Elternteils und der Zustandsuntermenge ohne Startzustand des anderen Elternteils.

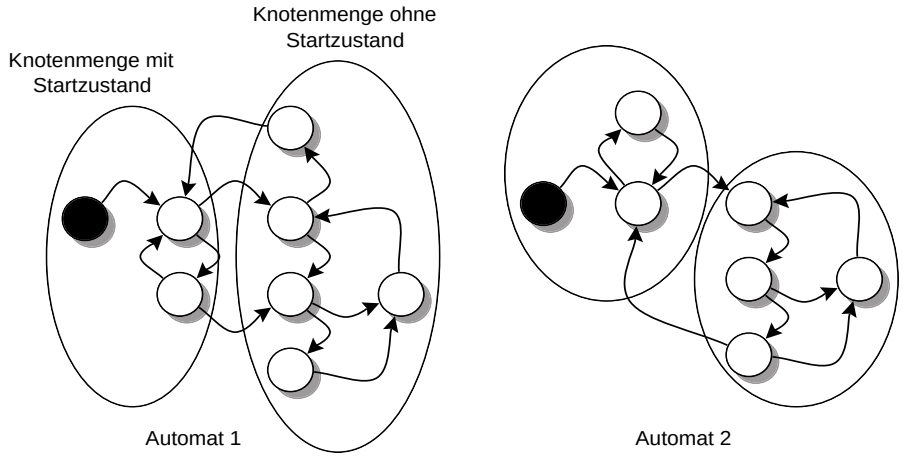


Abbildung 6.21: Crossover: zwei Automaten (*Eltern*) werden jeweils in zwei Untermengen aufgeteilt

4. Verbinde die Outputs und Inputs der beiden Untermengen (siehe Abbildung 6.22).

In der konkreten Implementierung wird später allerdings von den zwei entstandenen Kindern nur eines in die neue Generation aufgenommen.

Implementationsdesign

Es folgt nun die Beschreibung des Implementationsdesigns des GA–Frameworks. Abbildung 6.23 zeigt die Basisklassen dieses Frameworks, der Übersichtlichkeit halber in vereinfachter Form, bei der die Methoden der Klassen ausgeblendet sind.

Die Datenstruktur, die durch die genetischen Algorithmen optimiert werden soll, muss das *Evolvable*–Interface implementieren. Eine solche Datenstruktur muss die Methoden *mutate(), crossover()* und *copy()* implementieren, die jeweils für die genetischen Operatoren Mutation, Crossover und Reproduktion stehen. Hier zeigt sich eine erste Designentscheidung: Alle Operatoren sollen in einer Klasse implementiert werden. Dies ist jedoch oft nicht praktikabel, da dann die entsprechende Klasse sehr komplex werden würde. Außerdem wird so der Austausch einer Implementation eines Mutationsoperators gegen eine andere unnötig verkompliziert. Um hier Abhilfe zu schaffen, wurde die Klasse *AggregateEvolvable* sowie die Interfaces *Data, Mutator* und *Crossover* entworfen. Hier wird also das *Bridge*–Pattern [GHJV95, S. 151] angewendet: Statt die Funktionalität in der Klasse selber zu implementieren, werden Methodenaufrufe an andere Klassen weitergeleitet. Somit können die genetischen Operatoren jeweils in eigenen Klassen implementiert werden. *Mutator* implementiert dabei den Mutationsoperator, *Crossover* den Crossover–Operator und *Data* die Datenstruktur sowie den Reproduktionsoperator.

Durch dieses Design steht es dem Benutzer des GA–Frameworks frei, entweder alle genetischen Operatoren in einer Klasse oder in drei unterschiedlichen Klassen zu implementieren. Für welche Möglichkeit man sich entscheidet, hängt

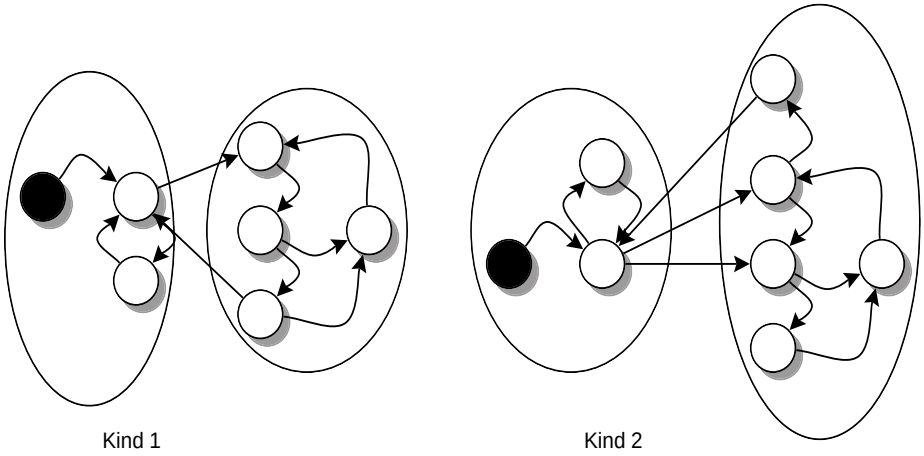


Abbildung 6.22: Crossover: Die durch Vertauschen der Zustandsuntermengen neu entstandenen Automaten (*Kinder*)

dabei im wesentlichen davon ab, wie komplex die Implementierung der Operatoren ist und ob man unterschiedliche Implementationen der jeweiligen Operatoren verwenden will.

Die Klasse *GPEvolver* stellt eine abstrakte Klasse dar, welche die gemeinsamen Teile aller hier verwendeten Varianten des genetischen Algorithmus implementiert. Alle konkreten Implementationen eines genetischen Algorithmus erben also von dieser Klasse und implementieren im wesentlichen die Methode *calculateFitness()*. Der Grund dafür ist, dass die Berechnung der Fitnesswerte in den einzelnen Versionen des genetischen Algorithmus zu unterschiedlich ist, als dass man sie alle in einer Klasse behandeln könnte. Diese Unterschiede sind sogar der Hauptgrund für die Implementierung zusätzlicher Klassen neben diesen Basisklassen. *GPEvolver* implementiert einen (μ, λ)–Algorithmus (siehe Abschnitt 6.3.2.1). Das bedeutet, dass in jeder Generation μ Individuen sind. Aus diesen Individuen werden dann mit Hilfe der genetischen Operatoren λ Individuen erzeugt ($\lambda > \mu$). Es wird dann eine Berechnung der Fitness ausgeführt, und anschließend werden die μ Individuen mit der höchsten Fitness in die nächste Generation übernommen. Auf diese μ Individuen beziehen sich dann auch die Angaben bezüglich der mittleren Fitnesswerte. μ und λ werden in den beiden Instanzvariablen *sizeOfChildGeneration* und *sizeOfParentGeneration* gespeichert. Die Wahrscheinlichkeiten für eine Mutation beziehungsweise ein Crossover sind in den Instanzvariablen *mutationProb* und *crossoverProb* abgelegt. Um die Fitness der einzelnen Elemente zu speichern, verwendet der *GPEvolver* die Klasse *FitnessEvolvable*. Diese bildet eine Kapsel um ein Evolvable. Dabei werden die *mutate(), crossover()* und *copy()* Methodenaufrufe an das interne *Evolvable* weitergeleitet. Darüber hinaus wird eine Methode zum Zugriff auf den Fitnesswert definiert sowie eine Methode, um zwei *FitnessEvolvable* anhand ihrer Fitnesswerte zu vergleichen. Da wie bereits erwähnt die Berechnung der Fitnesswerte in den unterschiedlichen Anwendungsgebieten sehr unterschiedlich sein kann, wird in dieser Klasse noch keine Annahme darüber gemacht, wie der Fitnesswert berechnet wird.

In diesem Klassendiagramm ist ein konkreter Fall, implementiert durch die

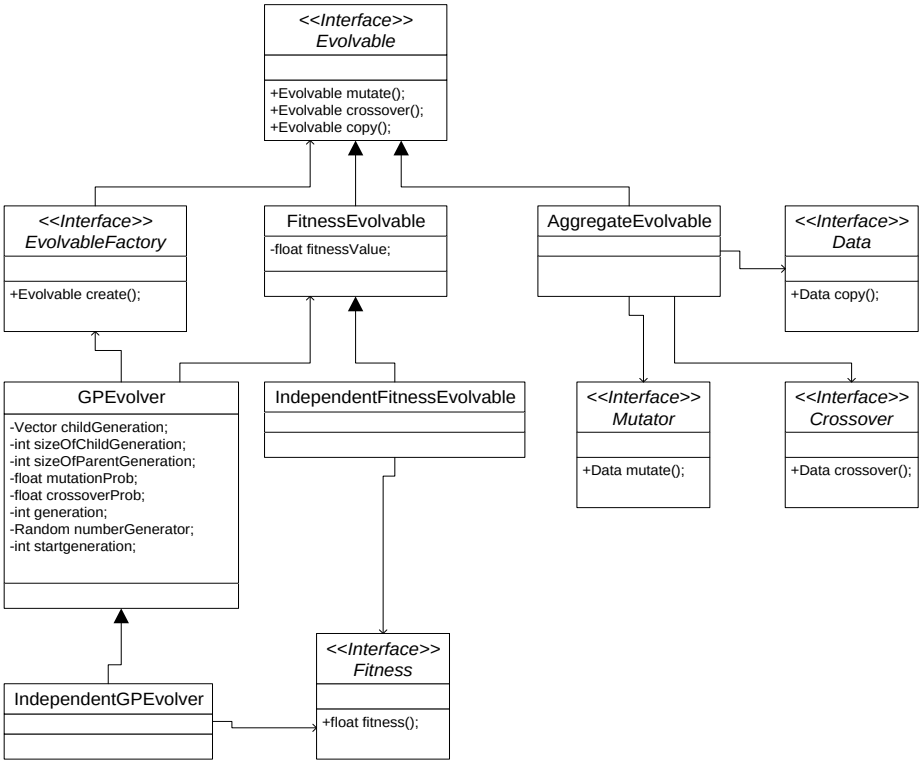


Abbildung 6.23: Übersicht der Basisklassen des GA–Frameworks

Klassen *IndependentFitnessEvolvable* und *IndependentGPEvolver* sowie *Fitness*, aufgeführt: Diese Klassen kommen dann zum Einsatz, wenn die Berechnung der Fitness einzig von den Daten des einzelnen Individuums abhängt. Ein Beispiel wäre die Optimierung einer Funktion: Für jedes Individuum kann man durch Einsetzen in die Funktion eine Fitness ermitteln, ohne andere Individuen zu betrachten. Ein Gegenbeispiel sind Verhandlungen: Hier treten typischerweise zwei Individuen in einer Verhandlung gegeneinander an. Das bedeutet, dass der Fitnesswert eines Individuums abhängig von anderen Individuen ist. *IndependentFitnessEvolvable* enthält dementsprechend eine Referenz auf eine Klasse, die Fitness implementiert. Wenn man dann ein *IndependentFitnessEvolvable* erzeugt, wird im Konstruktor mit Hilfe dieser Klasse der Fitnesswert berechnet und gespeichert. Man kann dann später auf diesen Wert mit *getFitness()* beliebig häufig zugreifen, ohne dass der Wert neu berechnet werden muss. Im *IndependentGPEvolver* wird dann entsprechend statt *FitnessEvolvables IndependentFitnessEvolvables* verwendet. Darüber hinaus enthält die *calculateFitness()* Methode keinen Programmcode, da die Fitnesswerte schon beim Erzeugen der einzelnen Individuen berechnet werden. Die *IndependentFitnessEvolvables* berechnen also die Fitnesswerte selber, ohne dass der *IndependentGPEvolver* eingreift — dies ist der Grund dafür, dass die *calculateFitness()*–Methode wie oben erwähnt leer ist. Wäre die Berechnung der Fitnesswerte nicht unabhängig von den anderen Individuen, müsste in dieser Methode jeweils die *Fitness einer Gruppe* von Individuen berechnet werden.

Details über die Implementation der zusätzlichen Klassen zur Berechnung der Gruppenfitness sowie die Implementation der endlichen Automaten sind in [Wol00] enthalten.

6.3.3 Realisierung von Verhandlungsszenarien

Mit dem im letzten Abschnitt (6.3.2) beschriebenen generischen GA–Framework wurden dann unterschiedliche Verhandlungsszenarien entwickelt, um die Anwendbarkeit von genetischen Algorithmen als Verhandlungsstrategien zu testen. Hierbei stellt sich zunächst die Frage, welche Arten von Verhandlungen mit Hilfe dieses Frameworks realisiert werden sollen. In Abschnitt 5.2.1 wurde bereits versucht, die ganze Bandbreite unterschiedlicher Verhandlungsarten aufzuzeigen. Aus dieser Vielfalt wurden zwei weit verbreitete, jedoch relativ simple Klassen für die konkrete Realisierung ausgewählt, nämlich *bilaterale Verhandlungen* und *Auktionen* als eine typische Ausprägung multilateraler Verhandlungen.

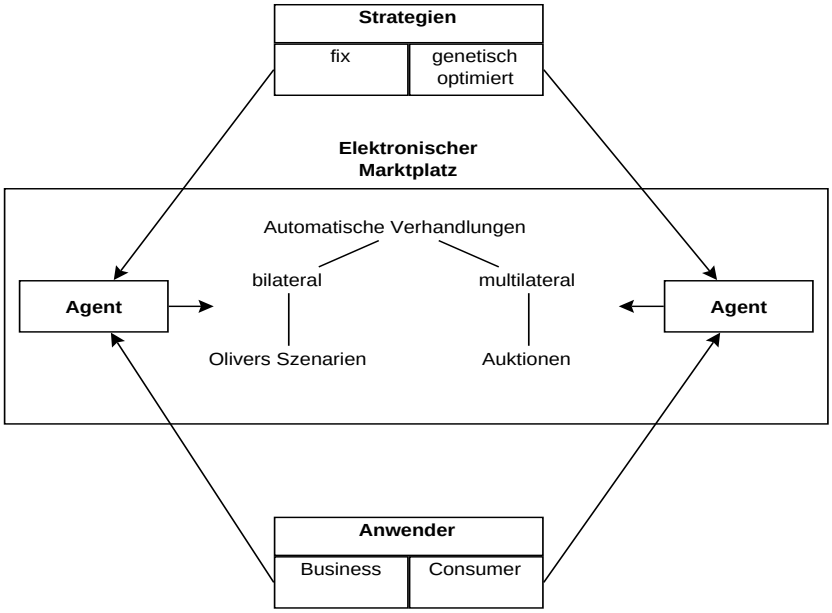


Abbildung 6.24: Konzeption der auf dem GA–Framework basierten Verhandlungsszenarien

Abbildung 6.24 zeigt die allgemeine Konzeption der realisierten Verhandlungsszenarien. Alle Szenarien bauen auf der Simulation eines *elektronischen Marktplatzes* auf, also eines Systems, in dem Agenten autonom Handelstransaktionen durchführen können. In der hier vorliegenden Arbeit werden in dem elektronischen Marktplatz zum einen bilaterale Verhandlungen wie die Szenarien aus der Arbeit von Oliver [Oli96] und zum anderen Auktionen als multilaterale Verhandlungsart zur Preisfindung verwendet. An den Verhandlungen nehmen *Agenten* teil. Sie können in der Verhandlung entweder eine *fixe* oder eine *genetisch optimierte* Strategie verfolgen. Diese unterscheiden sich dadurch, ob sie von einem genetischen Algorithmus optimiert werden oder nicht. Die Agenten vertreten auf dem elektronischen Marktplatz die eigentlichen *Anwender* des

Systems. Dabei handelt es sich um Personen oder Institutionen, die auf dem elektronischen Marktplatz handeln wollen. Bei den Anwendern kann man im wesentlichen *Business–Anwender* und *Consumer* unterscheiden, also Endkunden und Kunden, die selber wiederum Firmen sind. Letztere zeichnen sich in diesem Zusammenhang hauptsächlich dadurch aus, dass sie entweder über große Mengen von Waren verhandeln oder viele einzelne Verhandlungen durchführen müssen, um ihre Waren einzukaufen oder abzusetzen.

6.3.3.1 Bilaterale Szenarien

Bei diesen Szenarien handelt es sich vor allem um die Szenarien aus der Arbeit von Jim Oliver [Oli96], auf die bereits in Abschnitt 6.3.2.2 ausführlich eingegangen wurde. Bezogen auf die in Abbildung 6.24 dargestellte Konzeption bedeutet dies, dass hier nur bilaterale Verhandlungen implementiert werden und auch nur genetisch optimierte Strategien zum Einsatz kommen. Dabei werden jedoch sowohl Anwender aus der Kategorie Business als auch Consumer in Betracht gezogen.

Im Gegensatz zu der Arbeit von Oliver werden hier allerdings endliche Automaten statt der linearen Genome verwendet. Ziel ist es dabei, an Hand eines Vergleichs zwischen der hier vorliegenden Implementation und Olivers Implementation gegebenenfalls auch die Ergebnisse bezüglich des Lernens von Strategien mit Hilfe genetischer Algorithmen aus Olivers Arbeit zu übernehmen. Dabei geht es im wesentlichen um die Fragestellung, ob Agenten gute Verhandlungsergebnisse erzielen können und ob sie effektive Strategien entwickeln können. Entsprechend werden nur die "No Conflict", "Pure Distributive Bargaining", "Simple Integrative Bargaining" und "Divorce" Szenarien (vgl. Abschnitt 6.3.2.2) mit endlichen Automaten implementiert und die Ergebnisse mit Olivers Ergebnissen verglichen. Nicht implementiert wurde also das Szenario der "International Business Transaction", da dieses dem "Divorce" Szenario ähnelt und da ein direkter Vergleich mit Werten für Verhandlungen zwischen Menschen, die für dieses Szenario vorliegen, kein Ziel dieser Arbeit ist.

Bei den implementierten Szenarien werden die selben Parameter wie bei Oliver verwendet. Das bedeutet, dass auch in dieser Versuchsreihe die Populationsgröße 20 beträgt, in jeder Runde 20 Verhandlungen stattfinden und der ganze Vorgang für 20 Generationen wiederholt wird. Das gesamte Experiment wird dann noch mehrfach wiederholt, und es werden die Mittelwerte festgehalten. Diese Wahl von Parametern wurde in Abschnitt 6.3.2.2 kritisiert, da sowohl die Anzahl Generationen als auch die Anzahl Verhandlungen und die Anzahl der Wiederholungen für das gesamte Experiment zu niedrig sind. Dies ist insbesondere deswegen schwer nachvollziehbar, weil eine entsprechende Anpassung der Parameter mühelos möglich ist und auch die Laufzeiten der Simulation sich in erträglichen Grenzen halten. Dennoch wurden diese Parameter beibehalten, um eine Vergleichbarkeit mit Olivers Arbeit zu gewährleisten.

Ein wesentlicher Unterschied zu Olivers Arbeit ist allerdings, dass die Mutations– und Crossover–Wahrscheinlichkeiten nicht übernommen wurden. Dies ist darin begründet, dass hier auch andere Datenstrukturen verwendet werden und somit eine Anpassung dieser Parameter an neue Datenstrukturen gerechtfertigt erscheint. Außerdem ist hier auch wieder Olivers Parameterwahl kritikwürdig, da eine Crossover–Wahrscheinlichkeit von 0,5 und eine Mutations–Wahrscheinlichkeit von 0,05 gewählt wurden. Damit ist zunächst das Problem

verbunden, dass die Mehrzahl der Individuen (55%) bei der Übernahme in die nächste Generation verändert werden. Diese Tatsache kann die Selektion behindern, da eigentlich die weniger fitten Individuen aus der Population durch Selektion heraus genommen werden sollen. Wenn nun aber die fitteren Individuen in der Mehrzahl beim Übergang in die nächste Generation modifiziert werden, dann bedeutet dies, dass sie nicht zwangsläufig die fitteren Individuen bleiben, sondern eben durch diese Veränderungen wesentlich weniger fit geworden sein könnten. Leider gibt die Arbeit von Oliver auch keine umfassende Erklärung für seine Wahl der Parameter; es gibt nicht einmal einen Hinweis darauf, ob Oliver überhaupt für diese Parameter andere Werte ausprobiert hat.

Semantik der endlichen Automaten

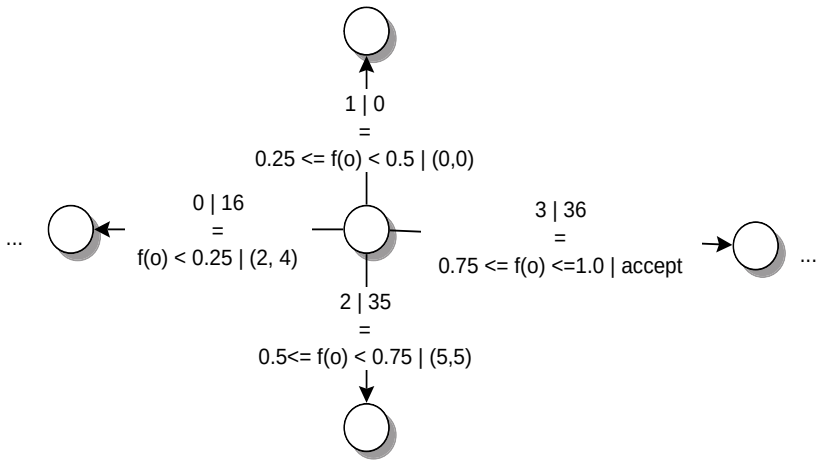


Abbildung 6.25: Beispiel für die Semantik der endlichen Automaten für bilaterale Verhandlungen

Für diese Szenarien muss noch eine Semantik für die Automaten definiert werden. Dazu dient der endliche Automat in Abbildung 6.25 als Beispiel. Dieser Automat hat ein Eingabealphabet mit der Mächtigkeit 4. Entsprechend geht von jedem Zustand auf Grund der Vollständigkeit und des Determinismus des Automaten (vgl. Abschnitt 6.3.2.3) für jedes Eingabesymbol eine Kante aus. In der Abbildung ist exemplarisch ein Zustand mit den entsprechenden vier Kanten gezeigt, die auch alle in vier verschiedene Zustände führen. Dabei hat jedes Eingabesymbol eine eigene Kante. Es wäre auch denkbar, dass mehrere Eingabesymbole identische Kanten verwenden. Um nun dem Eingabealphabet eine Semantik zuzuordnen, wird das Intervall der möglichen Nutzenwerte (in Olivers Beispielen $[0, 1]$) in mehrere gleichgroße Intervalle aufgeteilt. Die Anzahl dieser Intervalle entspricht dabei der Mächtigkeit des Eingabealphabetes. In diesem Beispiel wird also das Intervall $[0; 1]$ in die Subintervalle $[0; 0.25[, [0.25; 0.5[, [0.5; 0.75[$ und $[0.75; 1]$ aufgeteilt, die jeweils den Elementen 0,1,2 und 3 des Eingabealphabetes entsprechen. Wenn dann der Agent sich mit seiner Strategie in dem entsprechenden Zustand befindet, berechnet er den Nutzenwert des eingegangenen Angebots und verfolgt dann die entsprechende Kante. Bei einem Nutzenwert von 0.6 würde zum Beispiel die Kante für das Intervall $[0.5; 0.75[$ ausgewählt.

Dieses Intervall entspricht dann dem Eingabesymbol 3. Die Anzahl Kanten und damit die Anzahl Subintervalle, in die die Fitness unterteilt wird, kann variiert werden.

Als Ausgabesymbol befindet sich an jeder Kante wiederum eine natürliche Zahl, die dann entweder einem Gegenangebot entspricht oder dem Akzeptieren des Angebots. Hier wird eine Abbildung definiert, die jedem möglichen Angebot eine natürliche Zahl zuordnet. Für diese Abbildung f gilt, wenn $\mathcal{A}$ die Menge der Angebote mit einem zusätzlichen Symbol für das Akzeptieren des Angebots der Gegenseite ist:

$$\begin{aligned} \forall n \in [0, |\mathcal{A}| - 1] &: f(n) \in \mathcal{A} \\ \forall n, m \in [0, |\mathcal{A}| - 1], n \neq m &: f(n) \neq f(m) \\ \forall a \in \mathcal{A} : \exists n \in [0, |\mathcal{A}| - 1] &: f(n) = a \end{aligned}$$

Diese Abbildung ist also eine bijektive Abbildung von den natürlichen Zahlen zwischen 0 und der Anzahl der Elemente in $\mathcal{A} - 1$ in die Menge $\mathcal{A}$. Somit kann man den Zahlen, welche die eigentlichen Ausgabesymbole des endlichen Automaten darstellen, jeweils eindeutig ein Angebot als Semantik zuordnen, und es ist außerdem für den Automaten möglich, jedes beliebige Angebot aus der Menge aller Angebote als Gegenangebot zu machen.

Damit ist dann die Semantik sowohl des Eingabealphabetes als auch des Ausgabealphabetes für die endlichen Automaten in diesem Einsatzkontext geklärt. Dabei ist es wichtig, noch einmal darauf hinzuweisen, dass die Automaten selber nur auf natürlichen Zahlen arbeiten und erst außerhalb, also hier in der Fitnessfunktion, diesen Zahlen eine Semantik zugeordnet wird und entsprechend dann die Auslese vorgenommen wird. Mutation und Crossover sowie die Implementierung der eigentlichen Automaten sind davon unabhängig.

Experimente und Ergebnisse

Unter Anwendung des GA–Frameworks wurden spezielle Anwendungsklassen und Werkzeuge entwickelt, um die geschilderten bilateralen Verhandlungsszenarien in Olivers Arbeit durchzuführen. Die dabei gesammelten Ergebnisse werden im folgenden zusammengefasst und bewertet.

Hauptziel der Durchführung von Experimenten, die den Szenarien aus Olivers Arbeit entsprechen, ist es zunächst, möglichst gute Ergebnisse für die Fitnesswerte zu erreichen. Aufgrund des genannten Unterschieds zwischen den zu Grunde liegenden Datenstrukturen wurden dazu nicht Olivers Parameter für die Mutations– und Crossover–Wahrscheinlichkeit übernommen, sondern die einzelnen Szenarien mit unterschiedlichen Werten für diese Parameter ausprobiert, um optimalere zu finden. Dabei wurde zum einen der Fitnesswert nach 20 Generationen ermittelt als auch die Varianz für den Fitnesswert. Diese Varianz berechnet sich als

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

Ansonsten wurden die Parameter von Oliver übernommen, d.h.:

- Populationsgröße 20 (sowohl Kind- als auch Elterngeneration)
- 20 Generationen
- 3 Verhandlungsrunden
- 20 Verhandlungen pro Generation

Zur Ermittlung der Werte wurden nicht nur 10 Läufe gemittelt, wie Oliver dies getan hat, sondern 100 Läufe durchgeführt. Man kann dann bei den einzelnen Szenarien die mittleren Fitnesswerte und ihre Standardabweichung ermitteln. Abbildung 6.26 zeigt die Benutzerschnittstelle des interaktiven Werkzeugs zur Durchführung der bilateralen Verhandlungsszenarien, mit dem alle relevanten Parameter beliebig eingestellt werden können.

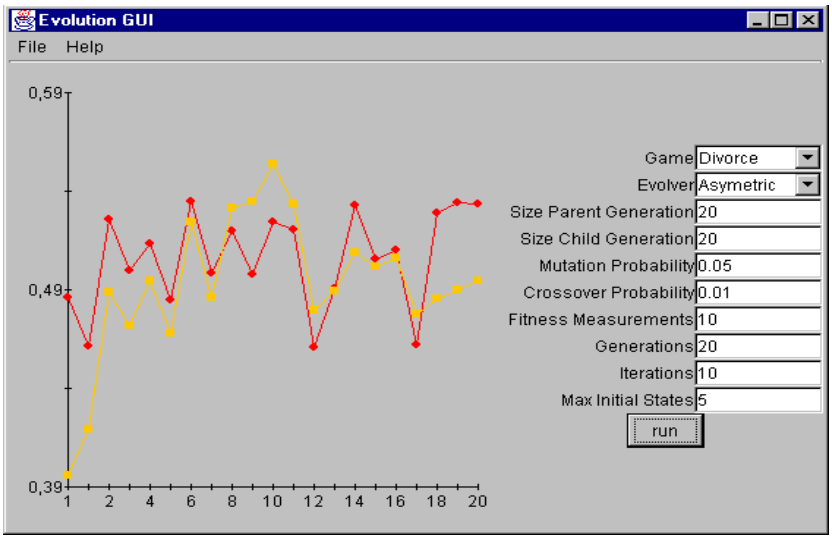


Abbildung 6.26: Interaktives Werkzeug zur Simulation von Verhandlungen

Dabei unterscheiden die meisten Szenarien zwischen zwei verschiedenen Spielern, die jeweils zwei Rollen in der Verhandlung entsprechen, also zum Beispiel dem Ehemann und der Ehefrau im "Divorce" Szenario. Das einzige Szenario, das eine solche Unterscheidung nicht benötigt, ist das "No Conflict" Szenario, da hier beide Spieler identische Nutzenfunktionen und damit auch identische Fitnesswerte haben. Bei diesen Experimenten wurden jeweils vier Kanten verwendet.

Als nächstes wurden die Automaten mit einer unterschiedlichen Anzahl von Kanten pro Zustand versehen. Wie man in Abbildung 6.25 sehen kann, wird das Intervall der möglichen Fitnesswerte ($[0;1]$) in mehrere gleichgroße Intervalle aufgeteilt. Dabei richtet sich die Größe der Intervalle nach der Anzahl der Kanten, die jeder Zustand in dem Automaten haben darf.

Die einzelnen Ergebnisse für die Fitnesswerte und deren Varianzen sind in Anhang E.1 graphisch dargestellt. Dabei sind jeweils die Fitnesswerte für die 20. Generation berücksichtigt worden. Die Fitnesswerte sind normalisiert worden und liegen daher im Intervall $[0;1]$.

Bewertung der Ergebnisse

Im folgenden sollen die im Rahmen der vorliegenden Arbeit erzielten Ergebnisse mit den aus Olivers Arbeit verglichen werden, um daraus Schlüsse für die Anwendbarkeit von genetischen Algorithmen auf bilaterale Verhandlungsszenarien zu ziehen.

An den Graphen in Anhang E.1 lässt sich erkennen, dass die besten Fitnesswerte bei niedrigen Mutations– und hohen Crossover–Wahrscheinlichkeiten erreicht werden. Dies entspricht auch der Wahl, die Oliver getroffen hat (Crossover–Wahrscheinlichkeit 0,5, Mutations–Wahrscheinlichkeit 0,05). Desweiteren ist erkennbar, dass die Varianz in einem sehr niedrigen Bereich liegen und man daher die Ergebnisse als sehr zuverlässig ansehen kann.

Diesbezüglich soll noch darauf hingewiesen werden, dass die Auswahl zum Teil unterschiedlichen Kriterien genügen muss: Beim "Pure Distributive Bargaining"–Szenario sollten zum Beispiel die beiden Fitnesswerte zusammen 1 ergeben und möglichst gleichgroß sein, da es dort darum geht, eine möglichst gerechte Aufteilung zu erhalten. Beim "No Conflict"–Szenario geht es lediglich um den höchsten Fitnesswert, der ja außerdem für beide Verhandlungspartner wegen der Identität der Nutzenfunktion auch gleich ist, bei den "Simple Integrative Bargaining" und "Divorce"–Szenarien um eine möglichst große Summe der Fitnesswerte.

Dieses Ergebnis ist auf den ersten Blick relativ überraschend, da man normalerweise davon ausgeht, dass Crossover nur sehr sparsam eingesetzt werden soll. Einige genetische Verfahren verzichten sogar vollständig auf den Einsatz von Crossover. In diesem Zusammenhang sei noch einmal darauf hingewiesen, dass Crossover eben eigentlich nur dazu dienen soll, gute Teile einzelner Individuen in der Population zu verbreiten. Die Variation der einzelnen Individuen und die Erzeugung komplett neuer Individuen sollte durch Mutation erfolgen. Die Erklärung für dieses Ergebnis liegt wahrscheinlich darin, dass Mutation so, wie sie hier und auch in Olivers Arbeit implementiert ist, immer auf einen sehr kleinen Teil des Genoms wirkt, während Crossover auf große Teile wirkt. Durch Crossover können also sehr schnell große Teile eines Genoms geändert werden und somit eine schnellere und möglicherweise auch bessere Anpassung stattfinden. Außerdem ist es auch nur durch Crossover möglich, eine koheränte Population zu schaffen, d.h. dass alle Individuen ähnliche Fitnesswerte erreichen werden, und da die Individuen bei den Verhandlungen immer wieder auf größtenteils identische Strategien treffen, können sie sich damit schnell an diese Strategien anpassen. Diese Kohärenz könnte auch der Grund dafür sein, dass die Varianz der Fitnesswerte stark abfällt, sobald die Crossover–Wahrscheinlichkeit größer als Null ist.

Tabelle 6.4 zeigt die im Rahmen dieser Arbeit erzielten maximalen Fitnesswerte im Vergleich mit denen aus der Arbeit von Oliver.

Hieran lässt sich gut erkennen, dass die hier erzielten Ergebnisse im Prinzip denen aus Olivers Arbeit gleichen. Damit ist es nun möglich, zumindest einen Teil der Ergebnisse zu übernehmen, die Oliver mit Hilfe seiner Experimente erzielt hat (siehe auch Abschnitt 6.3.2.2). Dazu gehören:

- Die Ergebnisse sind besser als die zufälliger Strategien.
- Agenten können Verhandlungsstrategien mit Hilfe von genetischen Algorithmen lernen.

Szenario	Fitness Agent 1	Fitness Agent 2	Olivers Wert Agent 1	Olivers Wert Agent 2
"No Conflict"	0.86	0.86	0.88	0.88
"Pure Distributive Bargaining"	0.51	0.49	0.54	0.46
"Simple Integrative Bargaining"	0.59	0.57	0.57	0.57
"Divorce"	0.58	0.55	0.57	0.50

Tabelle 6.4: Vergleich der Fitnesswerte mit den aus Olivers Arbeit

Damit hat sich gezeigt, dass endliche Automaten als eine im Verhältnis zu linearen Genomen deutlich komplexere Datenstruktur auch in diesen verhältnismäßig einfachen Szenarien einsetzbar sind. Gleichzeitig werfen diese Ergebnisse die Frage auf, ob es einen Vorteil gibt, wenn man endliche Automaten als Datenstruktur verwendet.

Um diese Frage zu beantworten, kann man sich die Werte der Experimente betrachten, bei denen die Automaten unterschiedlich viele Kanten haben. Wenn man nämlich einem Automaten in jedem Zustand nur eine Kante gibt, kann er nicht mehr zwischen verschiedenen Optionen wählen, sondern muss immer dieser einen Kante folgen. Damit wird er fast gleich mächtig zu den Genomen, die Oliver hat[8]. Diese Experimente geben nämlich Aufschluss darüber, ob die Entscheidungen, die ein Automat prinzipiell fällen kann, in diesem Szenario ein Vorteil sind.

Die Graphen der Ergebnisse zeigen deutlich, dass die Entscheidungen in diesem Szenario *keinen* Vorteil darstellen, denn die Werte für einen Automaten mit nur einer Kante (also einen Automaten ohne Entscheidungen) sind praktisch identisch zu den Werten der Automaten mit mehreren Kanten. Dies ist allerdings auch nicht wirklich verwunderlich, da die hier gewählten Szenarien sehr einfache Szenarien sind. Insbesondere ist es so, dass hier immer wieder Verhandlungspartner aufeinander treffen, die *identische* Nutzenfunktionen haben. Eine Entscheidung hätte nur dann einen Vorteil, wenn es unterschiedliche Verhandlungspartner gäbe und man abhängig davon unterschiedliche Angebote machen würde. Dies führt zu einer sehr grundlegenden Kritik an Olivers Szenarien: Die implizite Annahme, dass während des gesamten Optimierungsprozesses immer die selbe bzw. sehr ähnliche Fitnessfunktionen verwendet werden, ist sehr unrealistisch.

Eine andere Problematik besteht darin, dass genetische Algorithmen immer eine gewisse "Trainingsphase" benötigen, um gute Ergebnisse zu liefern. Dies hängt damit zusammen, dass man bei diesen Algorithmen immer eine Population braucht, auf der zahlreiche Fitnessmessungen gemacht werden. Bei herkömmlichen Problemstellungen ist dies noch nicht kritisch, da Fitnessberechnungen lediglich Zeit kosten. Wenn man aber Verhandlungen betrachtet, kostet eine solche Testphase möglicherweise auch Geld, wenn das Verhandlungsergebnis suboptimal ist.

[8] Der einzige Unterschied ist dann, dass in Olivers Genom ein linearer Weg durch das Genom vorgezeichnet ist, während die hier vorliegenden Automaten auch Schleifen und Ähnliches bilden können.

Für einen praxisnäheren Einsatz von genetischen Algorithmen zur Optimierung von Verhandlungsstrategien ist es also notwendig, diese besonderen Voraussetzungen zu beachten und entsprechende Szenarien auszuwählen.

6.3.3.2 Auktionsszenarien

Für Auktionsszenarien muss zunächst festgelegt werden, welche Art von Auktionen simuliert werden soll. Hierfür wurde eine Auktion mit *versiegelten* Angeboten (s. a. Abschnitt 5.2.1) gewählt, da diese auf Grund ihrer hohen Anonymität und der relativ großen Unabhängigkeit der Teilnehmer voneinander in einer Umgebung wie dem Internet sehr sinnvoll erscheint. Eine solche Auktion fängt damit an, dass der Auktionator das Mindestgebot für eine Ware verkündet. Wenn kein Teilnehmer sich findet, der diesen Preis zahlen will, kann der Auktionator den Preis senken. Andernfalls muss jedes Gebot von den Interessenten höher sein als das letzte. Das schließlich erzielte Höchstgebot ist auch der Preis, den der entsprechende Teilnehmer zahlen muss (und nicht etwa das zweit höchste Gebot wie bei der Vickrey–Auktion). Während der Auktion erfährt jeder Teilnehmer nur, ob sein Angebot das aktuell höchste ist. Die Angebote anderer Teilnehmer erfährt er also nicht. Die Auktion wird nach einer bestimmten Anzahl von Runden beendet.

Genau wie bei den bilateralen Verhandlungsszenarien ist es auch hier notwendig, einen Markt zu simulieren, in dem sich die durch genetische Algorithmen optimierten Strategien bewähren können. In diesem Fall sollen jedoch neben den genetischen Algorithmen auch fixe Strategien zur Anwendung kommen, um den Charakter eines realistischen Marktes etwas näher zu simulieren, als es in der Arbeit von Oliver der Fall war (vgl. Abbildung 6.24).

Für die fixen Strategien kann man auf einfache Strategien für Auktionen zurückgreifen, also die exponentielle oder lineare Erhöhung des Angebots, wie sie z.B. auch in dem System *MarketMaker* [MIT] eingesetzt werden. Diese bieten die Möglichkeit, zum Beispiel eine zögerliche Erhöhung des Angebots zu simulieren, wenn man mit Hilfe eines exponentiellen Verlaufs zunächst das Angebot nur wenig erhöht und erst später wirklich deutliche Erhöhungen macht. Außerdem können diese Strategien auch unterschiedliche Marktverhältnisse simulieren, also beispielsweise einen hohen Marktpreis durch höhere durchschnittliche Angebote und einen niedrigeren Marktpreis durch niedrigere durchschnittliche Angebote. Dabei sollten tatsächlich zufällige Abweichungen von diesen mittleren Preisen vorkommen, wie dies auch in wirklichen Märkten der Fall ist. Dort haben unterschiedliche Kunden unterschiedliche Preisvorstellungen, so dass entsprechend unterschiedliche Preise bei den einzelnen Verhandlungen erzielt werden.

Für die genetischen Strategien muss wiederum die Semantik der endlichen Automaten spezifiziert werden. Dies ist nun relativ einfach: Die endlichen Automaten sind schon für die im letzten Abschnitt beschriebenen Experimente um ein anfängliches Verhandlungsangebot ergänzt worden. Damit lässt sich der erwähnte anfängliche Wert relativ leicht abbilden. Für die späteren Ausgaben wird der Automat mit einem zweielementigen Eingabealphabet versehen (je ein Element für ein erfolgreiches und nicht erfolgreiches Angebot) sowie einem Ausgabealphabet, das dann allerdings einige hundert Elemente haben kann, also beispielsweise die Zahlen von 0-200, die dann eine Gebotserhöhung zwischen 0 und 200 darstellen beziehungsweise eine Mindestgebotssenkung zwischen 0 und 200.

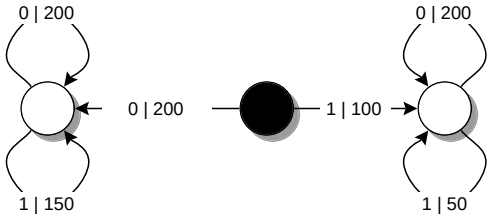


Abbildung 6.27: Eine simple Strategie für Auktionsszenarien als endlicher Automat

Die beispielhafte Strategie in Abbildung 6.27 soll dies veranschaulichen. Dabei soll das Eingabesymbol 0 für ein negatives Ergebnis in der jeweiligen Runde stehen, also ein Gebot, das nicht den Zuschlag bekommt oder ein Mindestgebot, das niemand abgeben will. Das Eingabesymbol 1 wiederum steht für ein positives Ergebnis, also ein Gebot, das zur Zeit das höchste ist, oder ein Mindestgebot, bei dem es Bieter gibt. Im ersten Zug wird unterschieden, ob das Ergebnis der ersten Runde positiv oder negativ war. Falls es positiv war, wird nur eine relativ niedrige Zahl zurückgegeben (100), also das Gebot vorsichtig erhöht beziehungsweise das Mindestgebot vorsichtig gesenkt. Wenn es negativ war, wird ein höherer Wert (200) zurückgegeben, also wird das Gebot dann stärker erhöht beziehungsweise das Mindestgebot stärker erniedrigt. Von diesem ersten Zug hängt auch das Verhalten in den darauf folgenden Runden ab: Wenn man schon in der ersten Runde ein positives Ergebnis erzielt hat, werden bei späteren negativen Ergebnissen niedrigere Werte zurückgegeben. In dieser Strategie findet sich also die Auffassung wieder, nach der man bei einem positiven Ergebnis in der ersten Runde davon ausgehen kann, dass in den späteren Runden positive Ergebnisse auch mit niedrigeren Geboten bzw. Mindestgeboten möglich ist.

Experimente und Ergebnisse

Um mit dem GA–Framework Auktionsszenarien zu realisieren, wurden ebenfalls spezielle Anwendungsklassen und Werkzeuge entwickelt, mit denen dann erste Experimente durchgeführt wurden. Die Konzeption der Experimente und deren Ergebnisse werden im folgenden zusammengefasst.

Die Parameter für den genetischen Algorithmus bzgl. der Auktionsszenarien, die mit dem interaktiven Werkzeug in Abbildung 6.28 flexibel eingestellt werden können, finden sich in Tabelle 6.5. Die Eigenschaften der Bieter und Anbieter finden sich in den folgenden Tabellen 6.6 und 6.7.

Insgesamt wurden folgende Szenarien implementiert:

1. **Zwei genetisch optimierte Strategien** An jeder Auktion nehmen zwei genetisch optimierte Strategien und 50 % (also zwei) fixe Strategien teil.

2. **Eine genetisch optimierte Strategie** An jeder Auktion nimmt eine genetisch optimierte Strategie und 50 % (also eine) fixe teil. Diese Parameter werden auch bei den folgenden Szenarien verwendet.

3. **Keine Wahl** In jedem Zustand haben die Automaten nur noch eine Kante, d.h. sie haben keine Entscheidungsmöglichkeiten mehr.

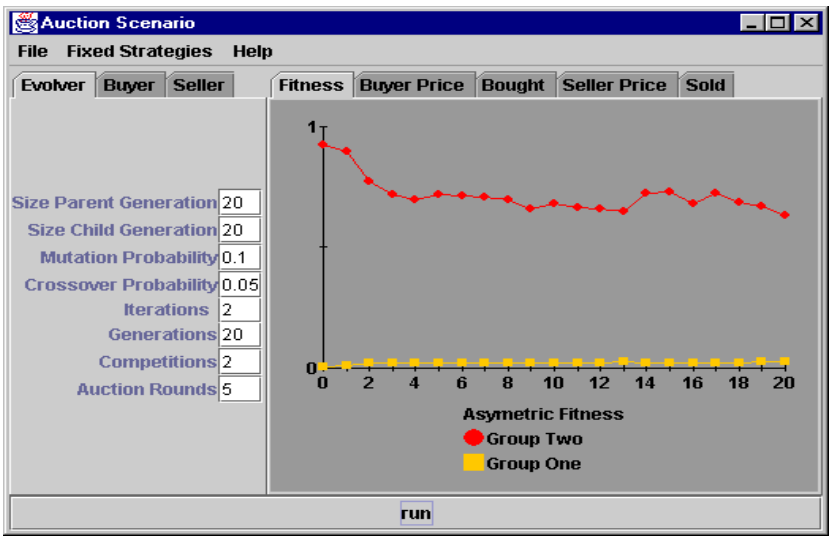


Abbildung 6.28: Interaktives Werkzeug zur Simulation von Auktionen

Parameter	Wert
Generationen	20
Populationsgröße	20
Wiederholungen (für Mittelwerte)	100
Auktionen pro Generation	4
Auktionsrunden	5

Tabelle 6.5: Parameter für den genetischen Algorithmus im Auktionsszenario

4. **Preislimitiertes Szenario** Die Fitnessfunktion für die endlichen Automaten wurde so modifiziert, dass die Ersteigerung für einen Preis *geq* 1300 mit einer Fitness von 0 "bestraft" wird.

5. **Anteilslimitiertes Szenario** Die Fitnessfunktion wird so modifiziert, dass endliche Automaten, die einen überdurchschnittlichen Preis gezahlt haben, eine Fitness von 0 erhalten. Damit soll ein Marktanteil von 50 % erreicht werden.

Die Experimente wurden mit verschiedenen Werten für die Mutations– und die Crossover–Wahrscheinlichkeiten durchgeführt. Dabei beziehen sich diese Werte sowohl auf die Optimierung der Strategien für die Anbieter als auch auf die Optimierung der Strategien für die Bieter. Bei jedem dieser Experimente werden einige Werte gemessen. (Die einzelnen Ergebnisse sind in Anhang E.2 zu finden.)

Beim "Zwei genetisch optimierte Strategien"–Szenario gehört dazu zunächst der Preis, den Bieter mit einer genetisch optimierten Strategie für die jeweilige Ware bezahlen (Abbildung E.22). Dementsprechend wird auch der Preis gemessen, den Bieter mit einer festen Strategie für die jeweiligen Ware zahlen

	genetisch optimierte Strategie	fixe Strategie
Anzahl pro Auktion	2	50 % = 2
Anfangsgebot	0-2000	$N(900, 100)$
Erhöhung pro Runde	0-120	$N(100, 10)$
	FSMs haben am Anfang $\leq$ 10 Zustände	lineare Angebotserhöhung

Tabelle 6.6: Eigenschaften der Bieter, wobei $N(x, y)$ eine Normalverteilung mit Mittelwert x und Standardabweichung y repräsentiert

	genetisch optimierte Strategie	fixe Strategie
Anzahl pro Auktion	2	50 % = 2
anfängliches Mindestgebot	0-1000	$N(1000, 100)$
Senkung pro Runde	0-100	$N(100, 10)$
	FSM haben am Anfang $\leq$ 10 Zustände	lineare Senkung

Tabelle 6.7: Eigenschaften der Anbieter

(Abbildung E.23). Schließlich wird dann gemessen, welcher Teil der Auktionen mit dem Erfolg eines Bieter mit einer genetisch optimierten Strategie endet. Dies ist in Abbildung E.24 dargestellt. Schließlich wird noch die Differenz zwischen den Preisen, die ein Anbieter mit einer genetisch optimierten Strategie erreicht, und den Preisen, die ein Anbieter mit einer festen Strategie erzielt, in Abbildung E.25 dargestellt. Dabei bedeutet eine positive Zahl, dass in diesem Szenario Anbietern mit einer genetisch optimierten Strategie höhere Preise erzielen als solche mit einer fixen Strategie.

Beim nächsten Experiment, dem "Eine genetisch optimierte Strategie"–Szenario, wurden wieder dieselben Werte gemessen, aber diesmal nahmen an jeder Auktion nur jeweils ein Agent mit einer genetisch optimierten Strategie teil. Der Anteil der Agenten mit fixer Strategie blieb bei 50 % und entsprechend nahm an jeder Auktion nur noch ein Agent mit einer fixen Strategie teil. Die entsprechenden Werte finden sich dann in den Abbildungen E.26, E.27, E.28 und E.29. Bei allen folgenden Experimenten nahmen dann auch weiterhin jeweils ein Agent mit einer genetisch optimierten Strategie und ein Agent mit einer festen Strategie an jeder Auktion teil.

Bei dem "Keine Wahl"–Szenario wurde analog zu den bilateralen Szenarien im letzten Abschnitt die endlichen Automaten so modifiziert, dass in jedem Zustand lediglich eine Kante existiert, so dass der endliche Automat nicht mehr zwischen mehreren Kanten entscheiden kann. Die Diagramme mit der Darstellung der jeweiligen Werte finden sich dann in den Abbildungen E.30, E.31, E.32

und E.33.

Im den letzten beiden Szenarien wurde die Fitnessfunktion modifiziert: Einmal wird allen Agenten, die einen Preis über 1300 zahlen, eine Fitness von 0 zugeordnet, obwohl sie etwas ersteigert haben (preislimitiertes Szenario, Abbildungen E.34, E.35, E.36 und E.37). Die andere Modifikation der Fitnessfunktion soll den Marktanteil der Bieter mit genetisch optimierten Strategien reduzieren (anteilslimitiertes Szenario). Dazu wird nur den Agenten eine Fitness > 0 zugeordnet, die einen unterdurchschnittlichen Preis gezahlt haben. Das Optimum für die Population wäre also, dass sie nur bei der Hälfte der Auktionen mit den niedrigsten Preisen erfolgreich bieten. Die entsprechenden Diagramme der gemessenen Werte finden sich in den Abbildungen E.38, E.39, E.40 und E.41.

Bewertung der Ergebnisse

Aus den eben geschilderten Auktionsszenarien und den in Anhang E aufgelisteten graphish–numerischen Ergebnissen lässt sich zunächst erkennen, dass die Strategien der Anbieter (in diesen Szenarien gleich Auktionator) keinen wirklichen Einfluss auf das Ergebnis der Auktion haben. In Bezug auf den Auktionator haben sowohl fixe Strategien als auch solche, die durch einen genetischen Algorithmus optimiert werden, gleich gute Werte erzielt. Angesichts der Semantik von Auktionsprotokollen ist dies aber keineswegs verwunderlich, denn Auktion ist eine Verhandlungsform, bei der allein die Bieter durch ihr gegenseitiges Überbieten den Preis bestimmen. Der Auktionator spielt während des eigentlichen Verhandlungsprozesses keine (semantisch) aktive Rolle. Der einzige Fall, der eine Ausnahme dazu darstellt, besteht darin, dass der Auktionator das Mindestgebot so hoch setzt, dass kein Bieter bereit ist, den Preis zu bezahlen. Dies wurde in den Experimenten jedoch durch entsprechende Einstellung der Strategieparameter verhindert.

In Bezug auf die Strategien der Bieter wurden einige interessante Ergebnisse erzielt. So ergibt sich aus dem "Zwei genetisch optimierte Strategien"–Szenario — unabhängig von der Mutations– und Crossover–Wahrscheinlichkeiten — ein im Vergleich zu den fixen Strategien sehr hoher Preis für die genetisch optimierten Strategien. Dies ist damit kombiniert, dass die genetisch optimierten Strategien praktisch immer die Waren ersteigern, also einen Anteil in der Nähe von 1.0 am Markt haben, da auch in diesen Auktionsszenarien ein sehr enger Zusammenhang zwischen Preis und Marktanteil besteht: Wenn man bereit ist, einen beliebig hohen Preis zu zahlen, dann wird man auch bei allen Auktionen gewinnen. Wenn man allerdings nur einen niedrigeren Preis zahlen möchte, dann wird der Anteil entsprechend sinken.

Der Grund dafür, dass in diesem Szenario die genetisch optimierten Strategien fast ohne Rücksicht auf den Preis versuchen, alles am Markt zu ersteigern, ist in der Fitnessfunktion zu suchen. Das schlechteste Ergebnis für eine genetisch optimierte Strategie ist eine Fitnessbewertung von 0. Dies tritt genau dann ein, wenn die genetisch optimierte Strategie nichts ersteigert. Wenn nun an jeder Auktion zwei genetisch optimierte Strategien teilnehmen, wird der Evolutionsprozess dafür sorgen, dass die Strategien immer höhere Preise zu zahlen bereit sind. Wenn nämlich eine Strategie mit einem niedrigen Maximalgebot auf eine Strategie mit einem höheren Maximalgebot trifft, wird die Strategie mit dem niedrigeren Maximalgebot eine Fitness von 0 aus dieser Auktion erhalten, während die andere Strategie eine Fitness > 0 erhält. Damit verschlechtern sich

die Chancen der Strategie mit niedrigerem Maximalgebot, weiter in der Population vertreten zu sein, und eine Evolution zugunsten höherer Maximalgebote tritt ein.[9]

Die Tatsache, dass die Teilnahme von zwei und mehr Agenten mit genetisch optimierten Strategien an jeder Auktion — die ansonsten aber keinen weiteren Einschränkungen unterliegen — offensichtlich nicht sinnvoll ist, war auch die Hauptmotivation dafür, die anderen Szenarien zu entwickeln.

So nimmt dann im "Eine genetisch optimierte Strategie"-Szenario nur jeweils ein Agent mit einer genetisch optimierten Strategie an jeder Auktion teil. Damit ergibt sich ein niedrigerer Wert für den durchschnittlichen Preis, den die Agenten mit der genetisch optimierten Strategie für die Ware zahlen. Gleichzeitig erniedrigt sich auch der Anteil der für die genetisch optimierten Strategie erfolgreichen Auktionen. Dies entspricht der bereits oben erwähnten Tatsache, dass zwischen Preis und Marktanteil ein wesentlicher Zusammenhang besteht. Dennoch ist der Preis, den genetisch optimierte Strategien zu zahlen bereit sind, höher als jener Preis, den die fixen Strategien im Durchschnitt zahlen. Die noch erfolgreichsten Strategien — also solche, die zu einem relativ niedrigen Preis relativ viele Waren ersteigern — werden dabei in diesem Szenario bei niedrigen Mutationsraten und niedrigen bis mittleren Crossoverwahrscheinlichkeiten erreicht, wie man an den Graphen erkennen kann.

In nächsten — *preislimitiert* genannten — Szenario kann der Anwender über eine Einschränkung des Preises Einfluss auf das Bietverhalten nehmen, was sicherlich sinnvoller ist als die Einschränkung der Teilnehmerzahl. In diesem Fall wurde spezifiziert, dass der Anwender nur 1300 als höchstes Gebot zulassen will. Die Fitnessfunktion wurde deshalb so modifiziert, dass ein Preis > 1300 der Strategie einen Fitnesswert von 0 zuordnet, obwohl etwas ersteigert wurde. Die dabei durch die Evolution entstehenden Strategien führen tatsächlich dazu, dass die mittleren Preise bei allen Mutation– und Crossover–Wahrscheinlichkeiten kleiner als 1300 sind. Die gezahlten Preise sind zwar höher als jene, welche die fixen Strategien zahlen, aber dafür ist der Marktanteil meistens über 50 %, so dass für diesen höheren Preis auch entsprechend mehr erworben wird.

Bei dem *anteilslimitierten* Szenario erfolgt die Steuerung durch den Anwender nicht über den Preis, sondern über den Marktanteil, der erzielt werden soll, in diesem Beispiel 50 %. Nur jene Strategien, die bei den 50 % Auktionen mit den niedrigsten Preisen den Zuschlag erhalten haben, erhalten einen Fitnesswert > 0 dafür. Dieses Szenario ist das anteilslimitierte Szenario. An den entsprechenden Graphen kann man erkennen, dass der Marktanteil in diesen Szenarien immer unter 50 % bleibt, was nicht verwunderlich ist, da durch Mutation und Crossover einige der in der letzten Generation noch erfolgreichen Strategien zu nicht erfolgreichen Strategien gemacht werden. Entsprechend ist der Marktanteil bei niedrigen Mutationsraten größer. Dennoch ist der Preis, den die genetisch optimierten Strategien zahlen, höher als jener, den die fixen Strategien zahlen. Damit ist diese Steuerungsmöglichkeit nicht sehr effektiv, den fixe Strategien erreichen in diesem Szenario mit einem niedrigeren Preis einen

[9] Dies wird auch durch die absolute Höhe der Gebote bestätigt. Der höchste Durchschnittswert beträgt 2279 (bei 10 % Mutationsrate und 50 % Crossoverwahrscheinlichkeit). Dies muß man mit dem maximalen Gebot, das die endlichen Automaten überhaupt erzeugen können, vergleichen. Aus einem maximalen Anfangsgebot von 2000 und 4 Auktionsrunden, in denen die Gebote jeweils maximal um 120 erhöht werden können, ergibt sich ein maximales Gebot für die endlichen Automaten von 2480.

höheren Marktanteil und sind daher offensichtlich überlegen.

Bei dem "Keine Wahl"–Szenario soll eine wichtige Frage, die sich auch bei den bilateralen Verhandlungsszenarien gestellt hat — nämlich, ob die endlichen Automaten mit ihrer Möglichkeit, Entscheidungen zu treffen, einen Vorteil haben gegenüber Datenstrukturen, die diese Möglichkeit nicht haben. Um dieses zu testen, wird ein Experiment durchgeführt, bei dem die Automaten nicht mehr die Möglichkeit haben, unterschiedlich zu reagieren, abhängig davon, ob sie zur Zeit das Gebot gemacht haben, das jetzt den Zuschlag bekommen würde oder nicht. Ein Vergleich der Abbildungen E.26 und E.30 sowie Abbildungen E.28 und E.32 ergeben zunächst keine signifikanten Unterschiede. Ein signifikanter Unterschied ist auch durch eine gemeinsame Darstellung der jeweiligen Werte für den Anteil der für genetische Strategien erfolgreichen Auktionen und die erfolgreichen Gebote nicht ersichtlich. Also kann auch hier wie schon im letzten Abschnitt zwar nicht nachgewiesen werden, dass endliche Automaten aufgrund der größeren Ausdrucksmächtigkeit tatsächlich bessere Strategien erzeugen, aber dies ist eher auf die verhältnismäßig geringe Komplexität der Szenarien zurückzuführen. Ziel bei diesen Szenarien war allerdings auch nicht, die Eignung von endlichen Automaten zu untersuchen, sondern vielmehr die Eignung von genetischen Algorithmen für Auktionen zu zeigen.

Eine weitere Frage, die im Zusammenhang mit der Anwendung von genetischen Strategien im allgemeinen sehr wichtig erscheint, ist, wie schnell sich eine Population auf Marktveränderungen adaptiert. Um diese Frage zu beantworten, wurde ein weiteres Experiment durchgeführt, bei dem die Parameter für den genetischen Algorithmus hauptsächlich so modifiziert werden, dass das Marktverhalten dynamischer wird. Daher nehmen bei diesem Experiment die Bieter nur jeweils an zwei Auktionen in jeder Generation teil und in Generation 10 wird das Verhalten der fixen Strategien abrupt geändert: Der Mittelwert von 900 für das Anfangsgebot wird auf 1500 erhöht. Als Mutationsrate wurde der Wert 0,05 und als Crossover–Wahrscheinlichkeit der Wert von 0,2 gewählt, da diese sich bei den vorherigen Experimenten bewährt haben.

Die Änderungen der Preis und Marktanteile finden sich in den Abbildungen 6.29 und 6.30. Bei dem Diagramm, das die Änderungen des Marktanteils zeigt, kann man sehen, dass es am Anfang des Experiments 3 Generationen dauert, bis sich der Marktanteil auf einen Wert eingestellt hat, der sich dann praktisch nicht mehr ändert. Auf Grund der sprunghaften Steigerung der Gebote der festen Strategien in Generation 10 bricht der Marktanteil zunächst ein — wobei gleichzeitig eine sprunghafte Preissteigerung (in Abbildung 6.30) zu vermerken ist — und gebraucht diesmal 4 Generationen, bis wieder ein stabiles Niveau erreicht wird. Dies gibt einen ungefähren Eindruck davon, wie lange man mit suboptimalen Ergebnissen bei einer plötzlichen Änderung des Marktes zurechtkommen muss. Wichtig ist dabei aber auch, dass die Änderung nach einer Generation, also zwei Auktionen pro Agent, bereits eine deutliche Verbesserung darstellt. Außerdem muss man im realen Einsatz in der Regel eher von einer langsamen, kontinuierlichen Änderung der Preise ausgehen, bei der eine noch schnellere Anpassung der genetischen Strategien angenommen werden kann.

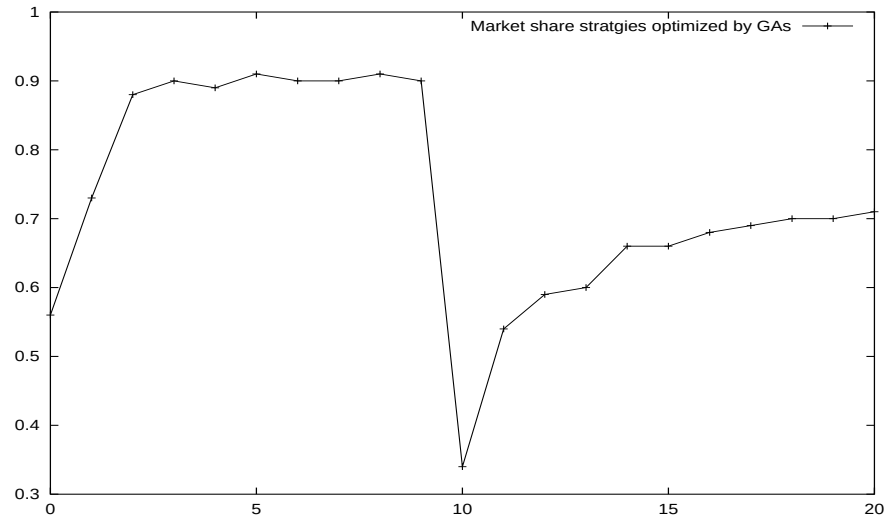


Abbildung 6.29: Anteil der erfolgreichen Auktionen für genetisch optimierte Strategien in Abhängigkeit von der Generation

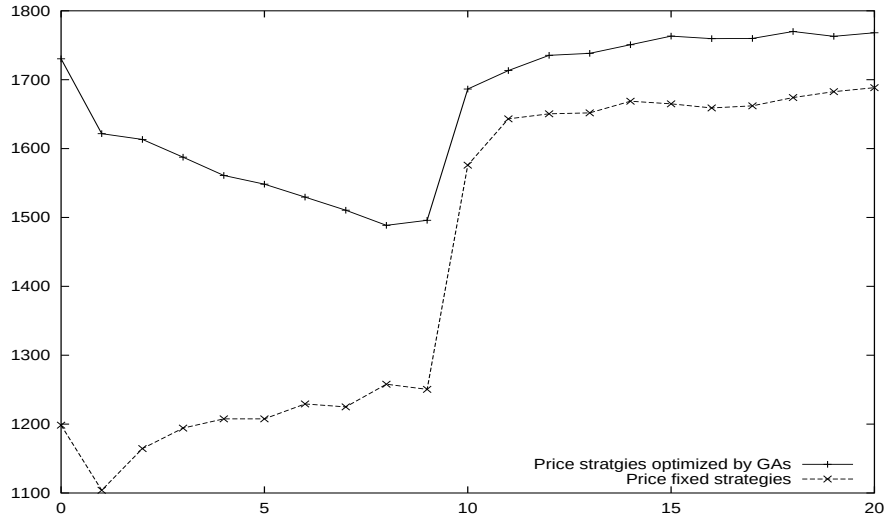


Abbildung 6.30: Preis bei Bietern mit fester Strategie und mit genetischem Algorithmus in Abhängigkeit von der Generation

6.4 Regelgesteuerte automatische Verhandlungen

Obwohl Mechanismen zur Automatisierung von Verhandlungen als eine funktionale Erweiterung der im ersten Teil dieser Arbeit (Kapitel 2 bis 4) dargestellten regelbasierten Steuerungsmechanismen betrachtet werden können (vgl. Abschnitt 5.1), können auch sie in der Praxis ganz erheblich von den Regel-

mechanismen unterstützt werden. Dies ist leicht nachzuvollziehen, wenn man bedenkt, dass in offenen Dienstemärkten autonom verhandelnde Agenten auf die selbe Weise ein anwendungssemantisches (häufig auch finanzielles) Risiko für die Anwender darstellen wie jede andere Art von Agenten, denen man die autonome Durchführung von Handelstransaktionen anvertrauen würde (siehe Abschnitt 4.4.4.1).

In diesem Abschnitt soll deshalb gezeigt werden, wie automatisch ablaufende Verhandlungen sowohl in Bezug auf die verwendeten Protokolle als auch Strategien mit Hilfe der geschilderten Regelmechanismen konkret gesteuert werden können (siehe auch [TKL00] sowie weitere technische Details in [Kun00]). Dazu wird zunächst ein Rule–Konfigurationsmodell eingeführt, mit dem konkrete Rule–Konfigurationen graphisch dargestellt werden können (6.4.1). Dann wird an Hand von Verhandlungsszenarien die Steuerung von Verhandlungsprotokollen (6.4.2) und -strategien (6.4.3) erläutert. Schließlich wird gezeigt, wie das Zusammenspiel von Rule–gestützten Protokoll und Strategie stattfindet (6.4.4).

6.4.1 Rule–Konfigurationsmodell

Da es sich bei den Rule–gestützten Verhandlungsszenarien um komplexere Abläufe handelt, wird für deren Entwurf ein präzises Modell benötigt, um die Konfiguration der Verhandlungsteilnehmer durch Rules darzustellen. Dafür wird das folgende Modell benutzt.

Die Rule–Unterstützung für autonom verhandelnde Agenten basiert auf der Anwendung von Rules und des entsprechenden Aktivierungsmodells (siehe Abschnitt 3.3.3.2). Damit die Rules auch das gewünschte Verhalten an dem Agenten auslösen, müssen sie sowohl in Bezug auf den jeweiligen Rule–Typ, –Bedingungsteil als auch die Ausführungsreihenfolge korrekt konfiguriert werden. Um die Rule–Konfiguration im Zusammenhang mit dem Aktivierungsmodell formal darstellen zu können, ist eine Petrinetz–Darstellung, die um Rollen an den Transitionen erweitert worden ist, gewählt worden. In Abbildung 6.31 ist ein (abstraktes) Beispiel der formalen Darstellung von einer Rule–Konfiguration im Zusammenhang mit dem Aktivierungsmodell dargestellt.

Der RS–DII–Aufrufer wird an der alles beinhaltenden, gestrichelten Transition in einem Oval angegeben (Kontext–Id des Agenten). In der Verfeinerung ist dargestellt, dass vor dem eigentlichen Methodenaufruf ein Event verteilt wird, das jede Rule empfängt (Triggerung der Rules vor dem Methodenaufruf). Dies ist in der Abbildung 6.31 durch die erste schmale hohe Transition dargestellt, die zur selben Zeit mehrere Rules triggert, die dann nebenläufig aktiviert sind. Eine Transition, die eine Rule–Aktion darstellt, wird mit dem Rule–Typ, ggf. dem Modus, dem Bedingungsteil oder der Aktion beschriftet. Auch wenn eine Rule auf das Event nicht reagiert, schaltet seine Transition. Nachdem alle Rules ihre Aktivierung erfolgreich beendet haben, findet eine Synchronisation der nebenläufigen Handlungen statt, was durch die zweite schmale hohe Transition dargestellt ist. Danach findet der Aufruf der Methode statt, dargestellt durch eine Transition, die mit der Methodensignatur beschriftet ist. Im Anschluss daran wiederholt sich der zuvor beschriebene Teil (Triggerung der Rules nach dem Methodenaufruf). Es kann sowohl nach der ersten Synchronisation der Rule–Aktivierung zu einer *RuleException* kommen als auch nach der zweiten Synchronisation. Dadurch wird ein Rollback der gesamten Transaktion veranlasst. Findet keine *RuleException* statt, wird die gesamte Transaktion

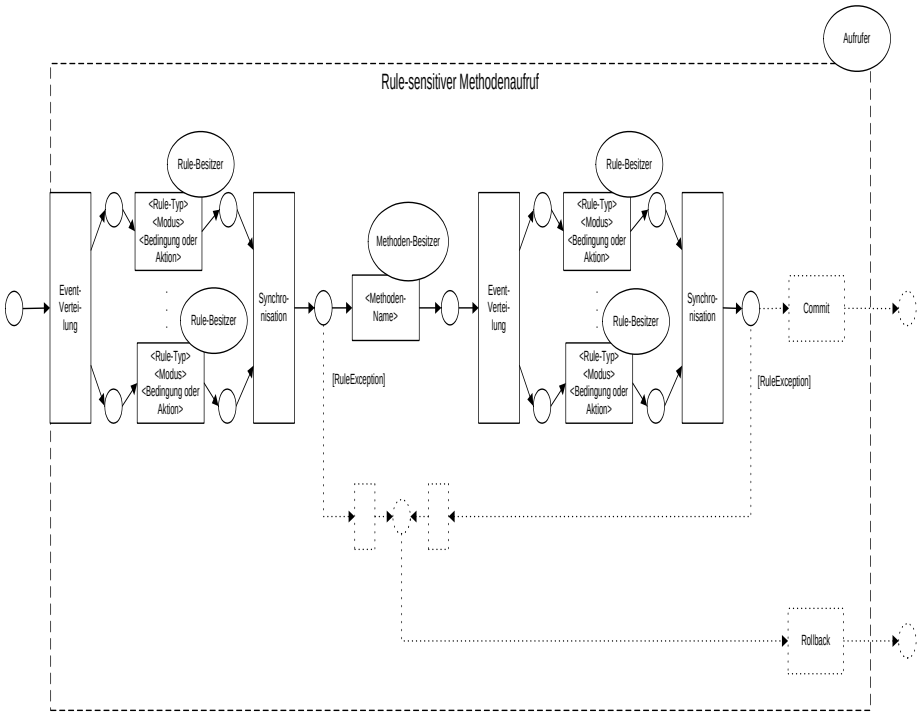


Abbildung 6.31: Formale Darstellung einer Rule–Konfiguration als Petrinetz

durch ein Commit abgeschlossen. Das gepunktete Teilnetz, das die Transaktionalität verdeutlicht, wird im Folgenden weggelassen, da es für die eigentliche Rule–Konfiguration keine Bedeutung hat. Es ist immer implizit durch das Aktivierungsmodell gegeben und kann lediglich bei der Ausführung zur Anwendung kommen. Die Rollen an den Transitionen der Verfeinerung beziehen sich auf die Agenten, an denen die Rules konfiguriert sind, bzw. auf den Agenten, an dem die Methode aufgerufen wird (Ziel–Id).

Der Übersichtlichkeit wegen werden Teile des Netzes in den folgenden Abschnitten weggelassen, sofern sie für die Rule–Konfiguration irrelevant sind. Die beschriebene Darstellung einer Rule–Konfiguration hat den Vorteil, dass alle Konfigurationsparameter direkt ablesbar sind.

6.4.2 Rule–gestützte Verhandlungsprotokolle

Um die Steuerung von Verhandlungsabläufen mit Rules zu demonstrieren, werden hier nicht die in Abschnitt 6.2 dargestellten Petrinetz–basierten Protokolle, sondern einfachere, nachrichtenorientierte Verhandlungsprotokolle (vgl. Abschnitt 6.2.1.1) eingesetzt, da diese sich *vollständig* als Rules spezifizieren lassen, wodurch die Anwendbarkeit von Rules anschaulicher gemacht werden kann.

Prinzipiell wird die Realisierung unterschiedlicher nachrichtenorientierter Verhandlungsprotokolle dadurch erreicht, dass der Aufruf der Methoden an einem an der Verhandlung teilnehmenden Agenten in Abhängigkeit eines für die Verhandlung relevanten Zustands–Property des Agenten mit Hilfe von Rules gesteuert wird. Die Überprüfung, ob der Aufruf einer Methode in einem bestimm-

ten Zustand erlaubt ist, erfolgt durch Regeln vom Typ *Requirement–Rule*. Das Übergehen in einen Folgezustand erfolgt durch *State–Transition–Rules*. Die Rules beider Typen agieren auf dem Zustands–Property des Agenten. Angestoßen durch die Änderung des Zustands–Propertys wird eine Aktivität der Verhandlungsstrategie ausgeführt, die an den jeweiligen Zustand gebunden ist. Um den Austausch von Verhandlungsnachrichten für den Rule–sensitiven Methodenaufrufmechanismus RS–DII (s. Abschnitt 4.3.1.5) direkt zugänglich zu machen, werden die für die Verhandlungsprotokolle relevanten Sprachelemente der Agenten werden als folgende Methoden realisiert:

- void offer(Policy offer, String agentAlias): Ein Angebot (*offer*) wird von einem Agenten (*agentAlias*) unterbreitet. (Ein *Gegenangebot* muss nicht speziell behandelt, sondern kann ebenfalls mit dieser Signatur unterbreitet werden.)

- void accept(String agentAlias): Ein Angebot wird von einem Agenten (*agentAlias*) akzeptiert.

- void abort(String agentAlias): Ein Angebot wird von einem Agenten (*agentAlias*) abgelehnt oder die Verhandlung wird abgebrochen.

Ähnlich wie in Abschnitt 6.3.3 wird in den folgenden Abschnitten wird die Rule–Steuerung des Verhandlungsprotokolls am Beispiel eines bilateralen und eines Auktionsszenarios beschrieben.

6.4.2.1 Bilaterales Szenario

Ein einfaches bilaterales Verhandlungsprotokoll, wie es z.B. auf einem Basar eingesetzt werden kann, lässt sich mit den genannten drei Methoden der Sprachschnittstelle des Agenten wie folgt beschreiben:

*Nachdem ein Verhandlungspartner auf dem Marktplatz ermittelt wurde, wird ihm ein erstes Angebot unterbreitet (*offer()*). Dieser hat daraufhin die Möglichkeit, mit einem Gegenangebot zu reagieren (ebenfalls mit der Methode* offer()*), das Angebot zu akzeptieren (*accept()*) oder das Angebot abzulehnen (*abort()*), und somit für einen vorzeitigen Abbruch der Verhandlung zu sorgen. Wird sein Angebot mit einem Gegenangebot beantwortet (*offer()*), dann hat der Agent ebenso die Möglichkeit, in der oben genannten Weise zu reagieren.*

Die Zustände, in denen sich ein Agent befinden kann und die für die Verhandlung relevant sind, umfassen folgende:

- einen Anfangszustand, in dem er sich befindet, bevor die Verhandlung startet,

- einen Wartezustand, in dem er sich befindet, wenn er auf Antwort des Verhandlungspartners wartet,

- einen Entscheidungszustand, in dem er sich befindet, wenn er das Angebot des Verhandlungspartners evaluiert,

- einen Abbruchzustand, in dem er sich befindet, wenn die Verhandlung abgebrochen wurde (*abort()*), um Aufräumarbeiten durchzuführen und

- einen Vertragsabschlusszustand, in dem er sich befindet, wenn das Angebot angenommen wurde (*accept()*), um den Vertrag abzuschließen.

In Abbildung 6.32 ist ein Zustandsdiagramm (State Diagram) eines Agenten in UML dargestellt, der eine Verhandlung startet[10]). Im Zustandsdiagramm wird die Informationsphase, also das Auffinden des Verhandlungspartners, als abgeschlossen betrachtet. Der Anfangszustand bezieht sich auf den Anfang der Verhandlung. In Abbildung 6.33 ist das Pendant, der passive Agent, dargestellt. Mit den Zustandsdiagrammen wird die Aufrufreihenfolge der Methoden der Sprachschnittstelle, in Abhängigkeit der für das Verhandlungsprotokoll relevanten Zustände des Agenten, dargestellt. Auf die Aktivitäten, die in den Zuständen stattfinden, wird in Abschnitt 6.4.3 eingegangen.

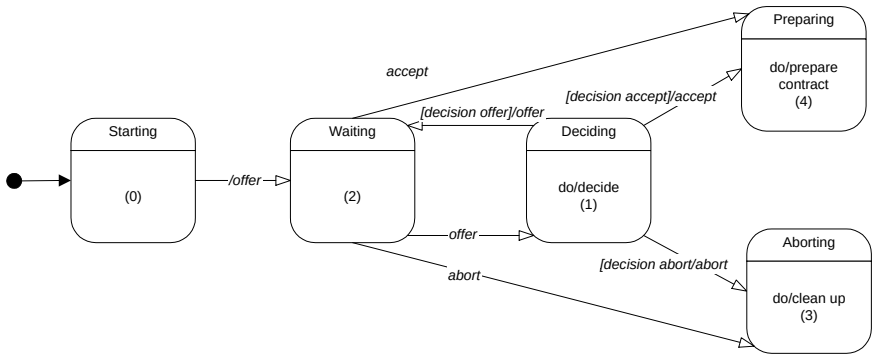


Abbildung 6.32: Zustandsdiagramm des aktiven Agenten bzgl. des bilateralen Verhandlungsprotokolls

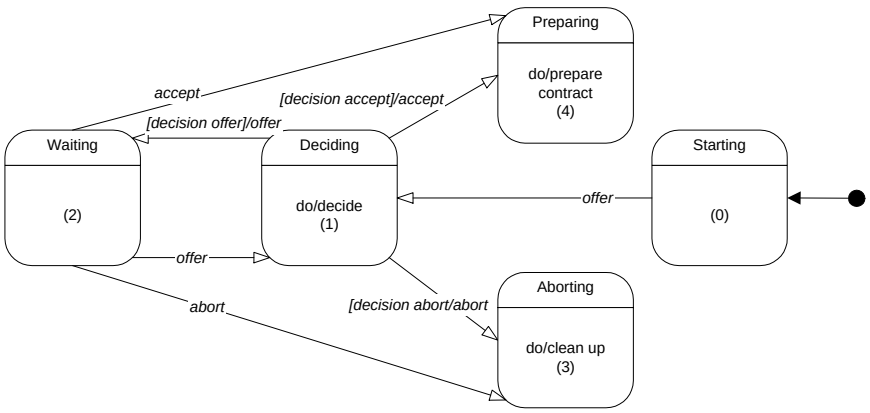


Abbildung 6.33: Zustandsdiagramm des passiven Agenten bzgl. des bilateralen Verhandlungsprotokolls

[10]In UML (siehe z.B. [FS98]) werden die Zustände eines State Diagram als Rechtecke mit abgerundeten Kanten dargestellt, die eine Zustandsbezeichnung mit sich führen und eine Aktivität beinhalten können, die in diesem Zustand stattfinden soll (*do/activity*). Die Kanten sind in der Form: *Event* [*Guard*] / *Action* dargestellt, von denen jeder Teil optional ist.

Die Zustände der beiden Zustandsdiagramme in Abbildung 6.32 und Abbildung 6.33 enthalten eine in Klammern gefasste Ziffer, die den Zustand nummeriert. Mit dieser Nummerierung kann der Zustand über ein Zustands–Property abgebildet werden, über das Rules den jeweiligen Zustand des Agenten überprüfen können. Damit kann vor jedem Methodenaufruf eine Requirement–Rule überprüfen, ob der Aufruf der Methode in dem Zustand gestattet ist. Nach dem Methodenaufruf kann der Zustand mit einer State–Transition–Rule geändert werden. Der Agent registriert nach dem Methodenaufruf den Zustandswechsel des Zustands–Properties und startet die in dem jeweiligen Zustand festgelegte Aktivität. Da ein Rule–sensitiver Methodenaufruf transaktional ist, wird der Zustandswechsel tatsächlich auch erst nach Beendigung des Methodenaufrufs sichtbar, so dass es sich hier um einen sequentiellen Ablauf handelt. Der zur Steuerung des Verhandlungsprotokolls eingesetzte Rule–Modus ist der Internal–Modus. Das bedeutet, dass sich die Rules auf den Agenten selbst beziehen und keine Interaktion mit Rules anderer Agenten benötigen. Dadurch wird auch die Autonomie des Agenten gewährleistet.

Jeder Agent besitzt die Anzahl von Requirement–Rules und State–Transition–Rules, die benötigt werden, um das Verhalten beider Zustandsdiagramme, die des aktiven und die des passiven Agenten, zu steuern. Diese Anzahl von Rules kann auch als Protokollsatz bezeichnet werden, der dem Agenten in eine dafür vorgesehene Protokoll–Set (*Protocol Set*) geladen wird. Im weiteren Verlauf des Abschnittes wird die Rule–Konfiguration für die beiden Teilnehmer an dem bilateralen Verhandlungsszenario beschrieben, wobei *Agent B* die Verhandlung anfängt, also aus Sicht des *Agenten A* der Aufrufer ist.

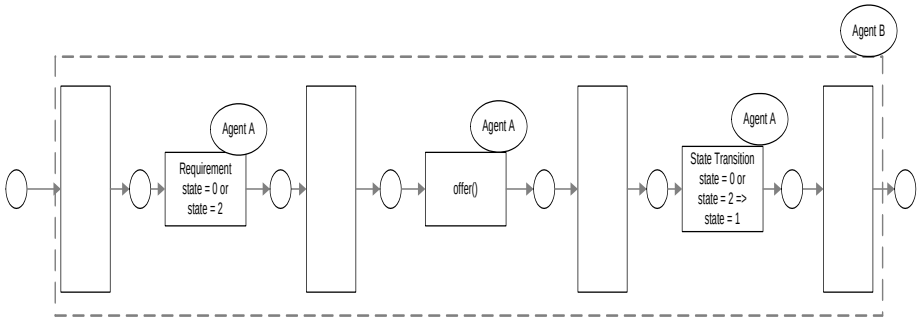


Abbildung 6.34: Rule–Konfiguration für bilaterale Verhandlung (1)

In Abbildung 6.34 sind die Rules dargestellt, die vor und nach dem Aufruf der Methode *offer()* bei dem Agenten A konfiguriert werden müssen. Ruft der Agent B an dem Agenten A die Methode *offer()* Rule–sensitiv auf, dann prüft zuerst eine Requirement–Rule, ob sich der Agent im Anfangszustand (Starting – 0) oder im Wartezustand (Waiting – 2) befindet. Nach dem Methodenaufruf gibt es eine State–Transition–Rule, die den Zustand je nach Ausgangszustand (Starting – 0 oder Waiting – 2) in den Entscheidungszustand (Deciding – 1) bringt. Dadurch wird die Aktivität *decide* angestossen. Entscheidet diese, dass ein Gegenangebot gemacht werden soll, dann wird die Methode *offer()* an dem Agenten B Rule–sensitiv aufgerufen.

Abbildung 6.35 zeigt die State–Transition–Rule, die dafür sorgt, dass der Agent A von dem Entscheidungszustand (Deciding — 1) in den Wartezustand

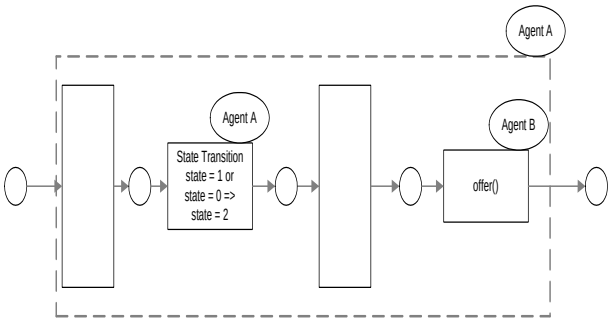


Abbildung 6.35: Rule–Konfiguration für bilaterale Verhandlung (2)

(Waiting — 2) wechselt. Für den Fall, dass der Agent A das erste Angebot macht, wird durch die selbe Rule von dem Startzustand (Starting — 0) auch in den Wartezustand gewechselt.

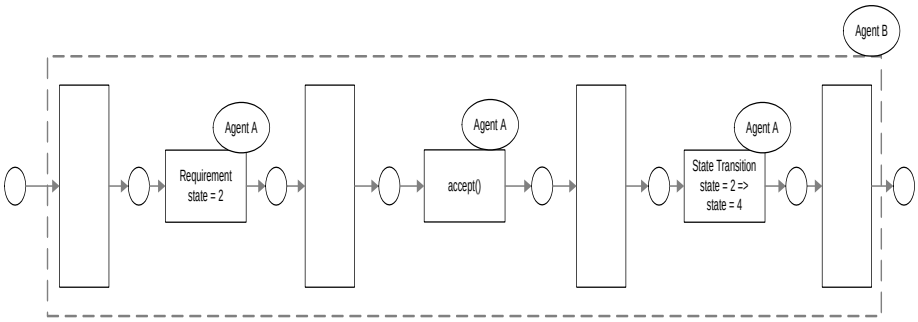


Abbildung 6.36: Rule–Konfiguration für bilaterale Verhandlung (3)

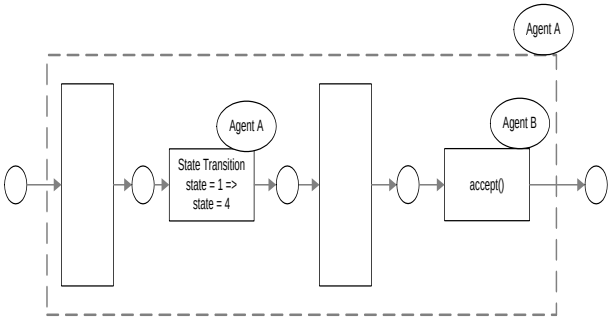


Abbildung 6.37: Rule–Konfiguration für bilaterale Verhandlung (4)

In Abbildung 6.36 sind die Rules dargestellt, die vor und nach dem Rule–sensitiven Aufruf der Methode *accept()* bei dem Agenten A konfiguriert werden müssen. Vor dem Rule–sensitiven Aufruf prüft eine Requirement Rule, ob sich der Agent in dem Wartezustand (Waiting — 2) befindet, und nach dem Methodenaufruf sorgt eine State–Transition–Rule dafür, dass von dem Wartezustand

in den Vertragsabschlusszustand (Preparing — 4) gewechselt wird. Falls der Agent A die Methode *accept()* bei dem Agenten B aufruft, wird der Zustand des Agenten A von dem Entscheidungszustand (Deciding — 1) in den Vertragsabschlusszustand (Preparing — 4) gewechselt (Abbildung 6.37). Das kann aufgrund einer Folgeaktion der Aktivität *decide* in dem Entscheidungszustand (Deciding — 1) geschehen.

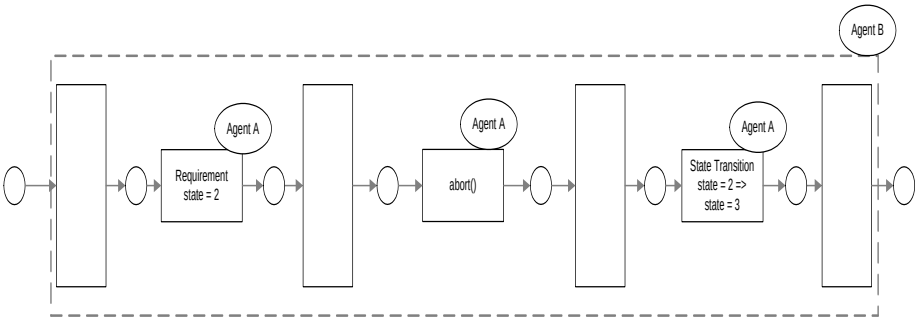


Abbildung 6.38: Rule–Konfiguration für bilaterale Verhandlung (5)

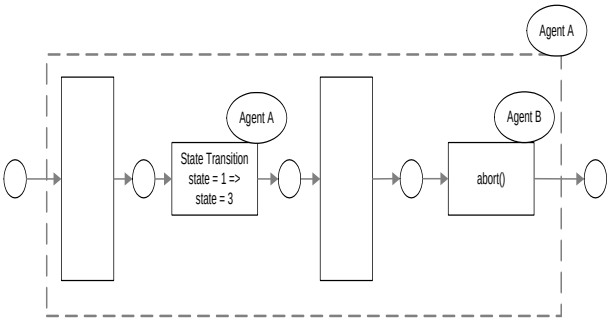


Abbildung 6.39: Rule–Konfiguration für bilaterale Verhandlung (6)

In Abbildung 6.38 sind die Rules dargestellt, die vor und nach dem Rule–sensitiven Aufruf der Methode *abort()* bei dem Agenten A konfiguriert werden müssen. Vor dem Rule–sensitiven Aufruf prüft eine Requirement–Rule, ob sich der Agent in dem Wartezustand (Waiting — 2) befindet, und nach dem Methodenaufruf ändert eine State–Transition–Rule den Wartezustand (Waiting – 2) in den Abbruchzustand (Aborting — 3). Falls der Agent A die Methode *abort()* bei dem Agenten B aufruft, wird von dem Entscheidungszustand (Deciding — 1) in den Abbruchzustand (Aborting — 3) gewechselt (Abbildung 6.39). Das kann aufgrund einer Folgeaktion der Aktivität *decide* in dem Entscheidungszustand (Deciding — 1) geschehen.

Wenn die Rollen der Agenten A und B ausgetauscht werden, dann ergibt sich die Rule–Konfiguration für den Agenten B. Die Rules können mit Hilfe von "Wildcards" so konfiguriert werden, dass der Verhandlungspartner für die Konfiguration nicht bekannt sein muss. Dadurch kann die selbe Rule–Konfiguration zum Verhandeln mit unterschiedlichen Agenten eingesetzt werden.

6.4.2.2 Auktionsszenario

Das Verhandlungsprotokoll für eine einfache öffentliche Auktion (die Englische Auktion) lässt sich mit den im letzten Abschnitt genannten drei Methoden der Sprachschnittstelle des Agenten wie folgt beschreiben:

*Der Auktionator ermittelt über den Broker alle im Marktplatz registrierten Marktteilnehmer. Er sendet jedem der Käufer nacheinander das aktuelle Gebot zu (*offer()*), wenn es sich geändert hat oder nachdem die Auktion gestartet wurde (Erstgebot). Ein Käufer hat die Möglichkeit, ein Gebot zu überbieten (*offer()*) oder abzulehnen (*abort()*). Macht der Käufer ein Gegengebot (*offer()*), so hat der Auktionator die Möglichkeit, das Gegengebot anzunehmen (*accept()*) oder abzulehnen (*abort()*). Ein Käufer kann jederzeit von sich aus ein Gebot abgeben (*offer()*), das von dem Auktionator mit Annahme (*accept()*) oder Ablehnung (*abort()*) beantwortet wird.*

Die Zustände, in denen sich ein Agent befinden kann und die für die Verhandlung relevant sind, umfassen:

- einen Anfangszustand, in dem er sich befindet, bevor auf ein Gebot reagiert wird,
- einen Wartezustand, in dem er sich befindet, wenn er auf Antwort des Verhandlungspartners wartet,
- einen Entscheidungszustand, in dem er sich befindet, wenn er das Angebot des Verhandlungspartners evaluiert,
- einen Abbruchzustand, in dem er sich befindet, wenn das Gebot abgelehnt wurde bzw. kein Gebot gemacht werden soll und
- einen Vertragsabschlusszustand, in dem er sich befindet, wenn das Gebot angenommen wurde (*accept()*).

In der Abbildung 6.40 ist ein Zustandsdiagramm (State Diagramm UML) eines Auktionators dargestellt, der eine Verhandlung startet.

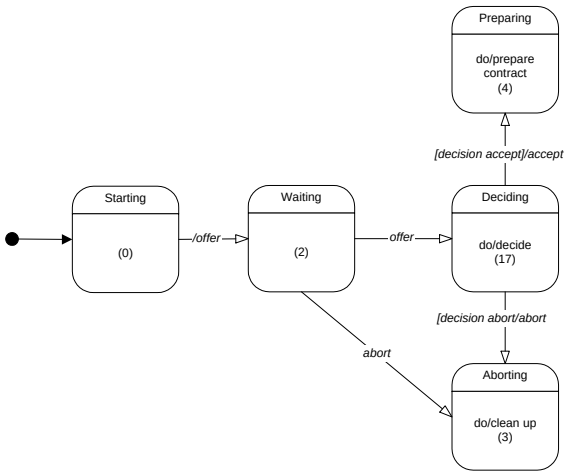


Abbildung 6.40: Zustandsdiagramm für den Auktionator

In dem Zustandsdiagramm wird die Informationsphase, also das Auffinden der Auktionsteilnehmer, als bereits abgeschlossen betrachtet. Ebenfalls hier nicht dargestellt und im weiteren auch nicht behandelt ist die Start- und Endzeit der Auktion sowie ein iterativer Vorgang des Auktionators, um die Auktionsteilnehmer solange über Preisänderungen zu informieren, bis die Auktion zu Ende ist. Es soll hier nur auf die Steuerung der Aufrufreihenfolge eingegangen werden.

In Abbildung 6.41 ist das Zustandsdiagramm des Auktionsteilnehmers dargestellt. Mit diesen Zustandsdiagrammen soll auch hier wieder ausschließlich die Aufrufreihenfolge der Methoden der Sprachschnittstelle in Abhängigkeit der für das Verhandlungsprotokoll Auktion relevanten Agentenzustände verdeutlicht werden.

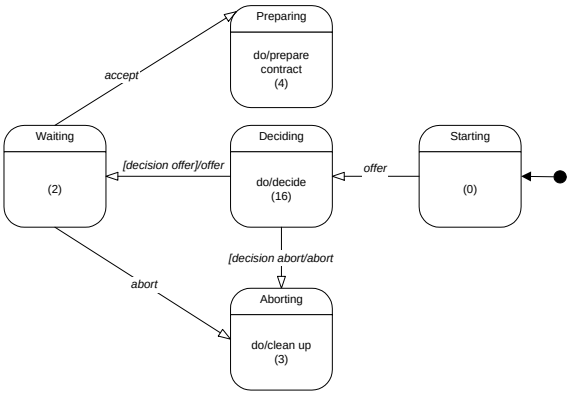


Abbildung 6.41: Zustandsdiagramm für den Auktionsteilnehmer

Hinsichtlich der Aktivitäten, die in den Zuständen stattfinden, sei auch hier wieder auf den Abschnitt 6.4.3 verwiesen.

Je nach Rolle des Agenten (Auktionsteilnehmer oder Auktionator) werden unterschiedliche *Requirement–Rules* und *State–Transition–Rules* benötigt, um das Verhalten des jeweiligen Zustandsdiagramms abzubilden. Im weiteren Verlauf des Abschnittes werden die Rule–Konfiguration des Auktionsteilnehmers und des Auktionators erläutert.

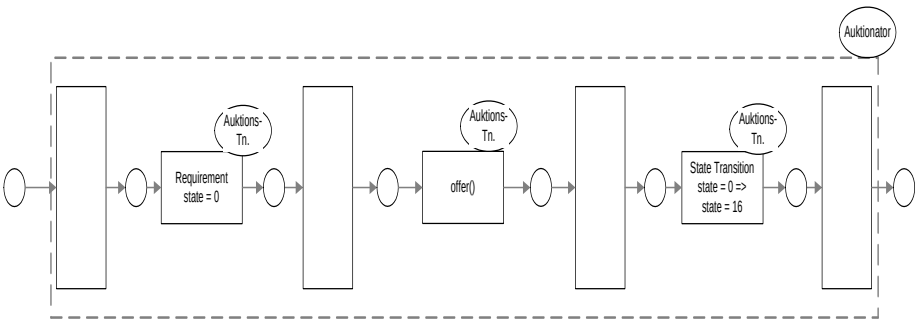


Abbildung 6.42: Rule–Konfiguration für den Auktionsteilnehmer (1)

In Abbildung 6.42 sind die Rules dargestellt, die beim Aufruf der Methode *offer()* bei dem Auktionsteilnehmer konfiguriert sein müssen. Vor dem Aufruf

wird mit einer Requirement–Rule geprüft, ob sich der Agent in dem Startzustand (Starting — 0) aufhält. Erst wenn dies der Fall ist, wird die Methode *offer()* ausgeführt, und im Anschluss daran sorgt eine State–Transition–Rule für den Zustandswechsel von dem Startzustand (Starting — 0) in den Entscheidungszustand (Deciding — 16) des Auktionsteilnehmers. Der Aufruf der Methode *offer()* an dem Auktionsteilnehmer kann als das Informieren über das höchste Angebot aufgefasst werden.

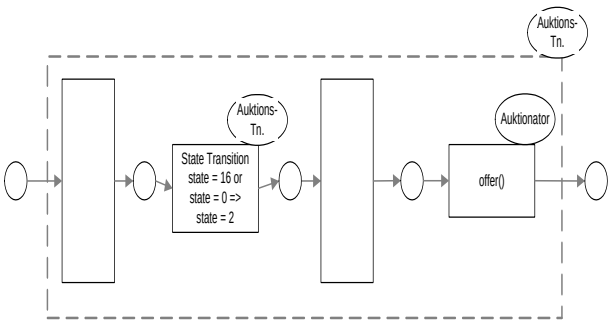


Abbildung 6.43: Rule–Konfiguration für den Auktionsteilnehmer (2)

Wenn sich der Auktionsteilnehmer in dem Entscheidungszustand dazu entscheidet, das Angebot zu überbieten, dann ruft er die Methode *offer()* bei dem Auktionator auf (s. Abbildung 6.43). Vor dem Aufruf sorgt eine State–Transition–Rule dafür, dass der Auktionsteilnehmer sowohl vom Entscheidungszustand (Deciding — 16) als auch vom Startzustand (Starting — 0) in den Wartezustand wechselt. Letzteres kommt vor, falls er von sich aus ein Angebot macht, ohne vorher benachrichtigt worden zu sein.

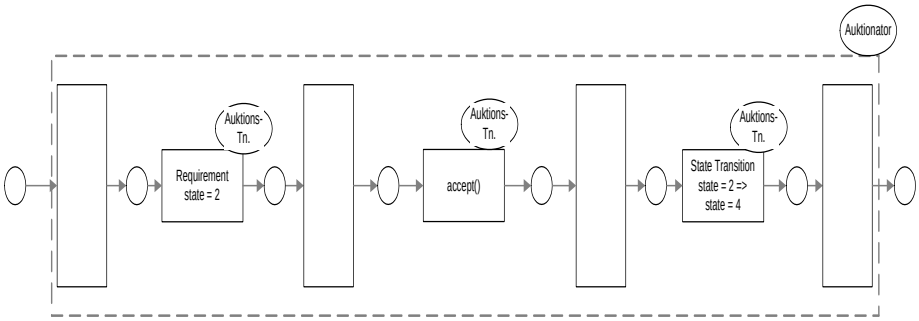


Abbildung 6.44: Rule–Konfiguration für den Auktionsteilnehmer (3)

In Abbildung 6.44 sind die Rules dargestellt, die beim Aufruf der Methode *accept()* bei dem Auktionsteilnehmer konfiguriert sein müssen. Vor dem Aufruf wird mit einer Requirement–Rule geprüft, ob sich der Agent in dem Wartezustand befindet (Waiting — 2), und nach dem Aufruf sorgt eine State–Transition–Rule dafür, dass von dem Wartezustand in den Vertragsabschlusszustand (Preparing — 4) übergegangen wird. Der Vertragsabschlusszustand wird bei der Auktion so interpretiert, dass der Vertrag zwar verbindlich ist, jedoch bis zum Ende der Auktion durch das Überbieten anderer Agenten wieder aufgehoben

werden kann.

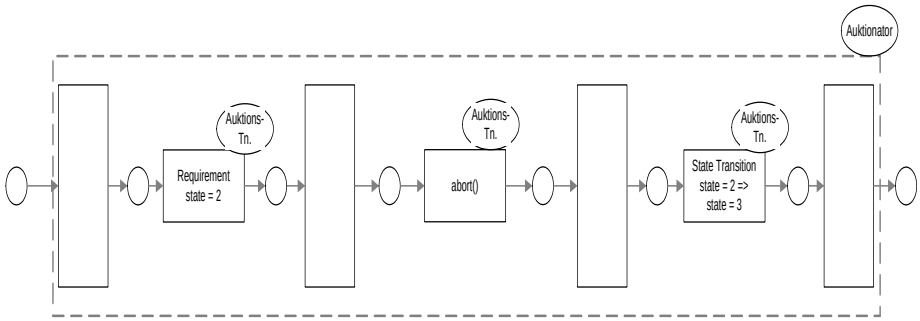


Abbildung 6.45: Rule–Konfiguration für den Auktionsteilnehmer (4)

Sollte der Auktionator das vom Auktionsteilnehmer gemachte Gebot ablehnen und die Methode *abort()* bei ihm aufrufen, wird vor dem Aufruf beim Auktionsteilnehmer mit einer Requirement–Rule geprüft, ob er sich im Wartezustand (Waiting — 2) befindet. Danach sorgt eine State–Transition–Rule dafür, dass von dem Wartezustand in den Abbruchzustand (Aborting — 3) gewechselt wird (Abbildung 6.45).

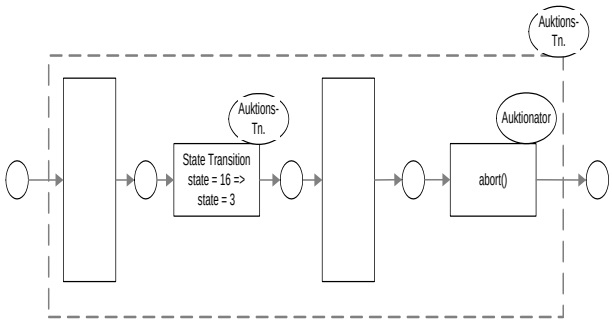


Abbildung 6.46: Rule–Konfiguration für den Auktionsteilnehmer (5)

Für den Fall, dass in dem Entscheidungszustand des Auktionsteilnehmers die Entscheidung auf Abbruch fällt, sorgt vor dem Aufruf der Methode *abort()* an dem Auktionator eine State–Transition–Rule dafür, dass von dem Entscheidungszustand (Deciding — 16) in den Abbruchzustand (Aborting — 3) übergegangen wird (Abbildung 6.46).

Damit ist die Rule–Konfiguration des Auktionsteilnehmers beschrieben. Als nächstes wird nun die Rule–Konfiguration des Auktionators beschrieben.

In Abbildung 6.47 sind die Rules dargestellt, die beim Aufruf der Methode *offer()* bei dem Auktionator konfiguriert sein müssen. Vor dem Aufruf wird mit einer Requirement–Rule geprüft, ob sich der Auktionator in dem Startzustand (Starting — 0) oder in dem Wartezustand (Waiting — 2) befindet. Nach dem Aufruf sorgt eine State–Transition–Rule dafür, dass von dem Wartezustand oder dem Startzustand in den Entscheidungszustand (Deciding — 17) übergegangen wird.

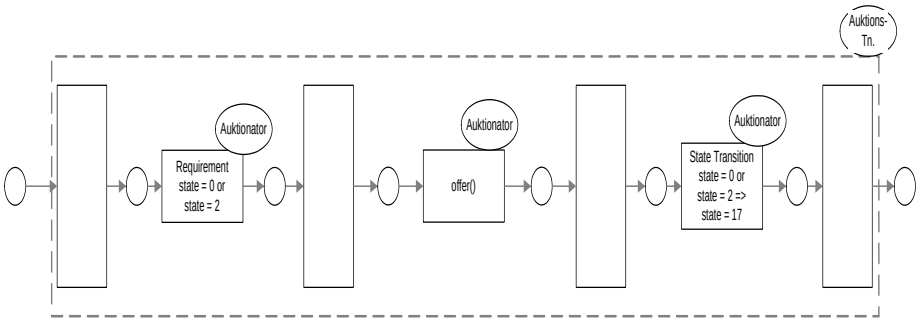


Abbildung 6.47: Rule–Konfiguration für den Auktionator (1)

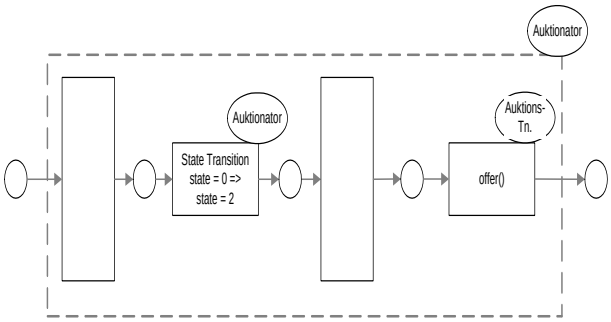


Abbildung 6.48: Rule–Konfiguration für den Auktionator (2)

Wenn der Auktionator den Auktionsteilnehmer über ein neues Gebot informiert, dann ruft er die Methode *offer()* bei dem Auktionsteilnehmer auf. Vor dem Aufruf sorgt eine State–Transition–Rule dafür, dass der Auktionator von dem Startzustand (Starting — 0) in den Wartezustand (Waiting — 2) übergeht (Abbildung 6.48).

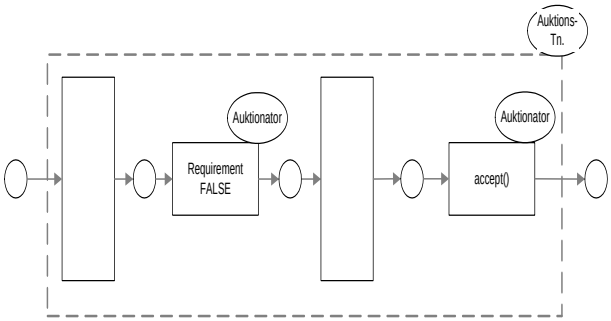


Abbildung 6.49: Rule–Konfiguration für den Auktionator (3)

In Abbildung 6.49 ist der Aufruf der Methode *accept()* an dem Auktionator dargestellt. Dieser Aufruf ist den Auktionsteilnehmern nicht gestattet, was mit einer Requirement–Rule ausgedrückt wird, die als Bedingungsteil lediglich das Literal *FALSE* enthält.

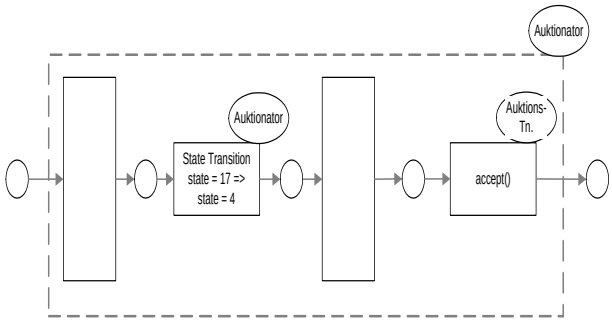


Abbildung 6.50: Rule–Konfiguration für den Auktionator (4)

Akzeptiert der Auktionator ein Gebot, so ruft er die Methode *accept()* beim Auktionsteilnehmer auf. Vor dem Aufruf sorgt eine State–Transition–Rule dafür, dass der Auktionator von dem Entscheidungszustand (Deciding — 17) in den Vertragsvorbereitungszustand (Preparing — 4) wechselt (Abbildung 6.50).

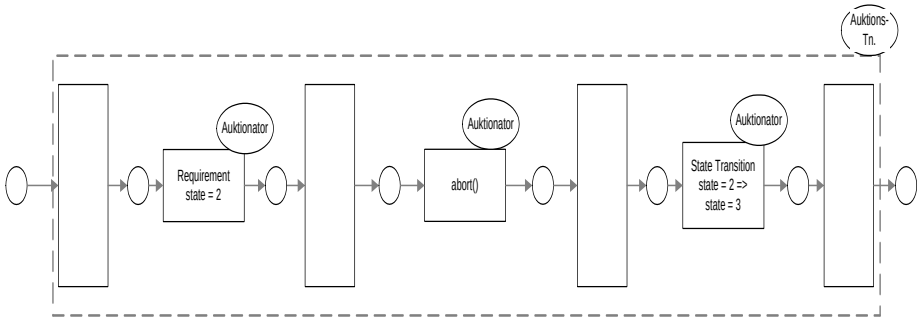


Abbildung 6.51: Rule–Konfiguration für den Auktionator (5)

In Abbildung 6.51 ist der Aufruf der Methode *abort()* an dem Auktionator dargestellt. Vor dem Aufruf prüft eine Requirement–Rule, dass sich der Auktionator in dem Wartezustand (Waiting — 2) befindet, und nach dem Aufruf sorgt einen State–Transition–Rule dafür, dass von dem Wartezustand in den Abbruchzustand (Aborting — 3) gewechselt wird.

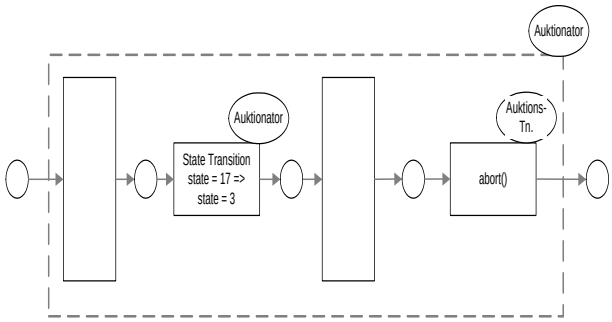


Abbildung 6.52: Rule–Konfiguration für den Auktionator (6)

Wenn der Auktionator ein Gebot ablehnt, dann ruft er die Methode *abort()* beim Auktionsteilnehmer auf. Vor dem Aufruf sorgt eine State–Transition–Rule dafür, dass der Auktionator von dem Entscheidungszustand (Deciding — 17) in den Abbruchzustand übergeht (Abbildung 6.52).

Auch bei der Auktion lassen sich die Rules durch die Verwendung von Wildcards so konfigurieren, dass sie nur den Namen des eigenen Agenten kennen müssen. Dadurch wird erreicht, dass ein Protokollsatz nur einmal geladen werden muss und dann mit jedem Verhandlungspartner zusammenspielt.

6.4.3 Rule–gestützte Verhandlungsstrategien

Aus Anwendersicht kann gerade die Steuerung der Verhandlungsstrategie durch Rules sehr sinnvoll sein, da das Verhandlungsergebnis direkt von der Strategie abhängt. Dabei können Rules vor allem als *Meta*–Strategien eingesetzt werden, d.h. sie verändern nicht die prinzipielle Funktionsweise (den Berechnungsalgorithmus) der Strategie, sondern stellen sicher, dass die Strategie gewisse Randbedingungen nicht verletzt bzw. zu bestimmten Zeitpunkten erfüllen muss. Damit können solche regelbasierte Meta–Strategien für beliebige Strategien wieder verwendet werden.

Typ/Modus:	Requirement/Internal
Trigger:	vor *offer()*
Bedingung:	**Preis / Budget $\leq$ 0,5**

Tabelle 6.8: Beispielregel für den Verhandlungsspielraum des Kunden

Tabelle 6.8 zeigt beispielsweise eine Regel zur Festlegung des Verhandlungsspielraums für einen Agenten, der den Kunden bei einer bilateralen Verhandlung vertritt. Diese Regel besagt, dass die Höhe des von dem Agenten erzeugten Angebots maximal die Hälfte des aktuellen Budgets betragen darf.

Typ/Modus:	Requirement/Internal
Trigger:	vor *offer()*
Bedingung:	**Preis $\geq$ 300**

Tabelle 6.9: Beispielregel für den Verhandlungsspielraum des Anbieters

Dagegen kann der Anbieter von seinem Agenten verlangen, dass der Preis für die entsprechende Ware keinesfalls eine absolute Grenze unterschreiten darf. Eine solche Regel zeigt Tabelle 6.9.

Typ/Modus:	StateTransition/Internal
Trigger:	vor *offer()*
Bedingung:	**Verhandlungsrunde = 5**
Vorbedingung:	Verhandlungsschritt = 4
Nachbedingung:	Verhandlungsschritt = 2

Tabelle 6.10: Beispielregel zur Modifikation des Verhandlungsschrittes

Interessanter sind Regeln, die nicht konstant greifen, sondern Veränderungen des Bietverhaltens während des Verhandlungsprozesses bewirken. Beispielsweise könnte sich ein Teilnehmer an einer Auktion strategisch so verhalten, dass er zu Anfang sein Angebot pro Runde immer um 4 Einheiten erhöht, nach einer gewissen Runde den Verhandlungsschritt jedoch auf 2 begrentzt, um damit das Risiko eines überzogenen Gebots wieder zu reduzieren. Eine solche Regel ist in Tabelle 6.10 dargestellt. Sie bewirkt, dass in der 5. Verhandlungsrunde das die Erhöhung des Angebots von 4 auf 2 Einheiten reduziert wird.

Im folgenden werden die Mechanismen beschrieben, die zu einer reibungslosen Steuerung der Verhandlungsstrategie durch externe Rules benötigt werden. Die semantische Verbindung zwischen Protokoll und Strategie bildet das Zustands–Property, das den für die Verhandlung relevanten Agentenzustand hält. Dieses Property ist in der *Protocol Set* enthalten, da es in den meisten Fällen von den Protokoll–Rules gesetzt wird. Abhängig von dem Wert des Zustands–Propertys wird jeweils eine entsprechende Aktivität der Strategie ausgeführt. Die Aktivitäten im Abbruchzustand (Aborting — 3) und im Vertragsabschlusszustand (Preparing — 4) setzen dabei intern das für die Verhandlung relevante Zustands–Property des Agenten zurück auf den Startzustand (Starting — 0), damit der Agent weiterhin protokollkonform ansprechbar ist.

Wert des Zustands–Propertys	Aktivität
0	keine Aktivität (Starting)
1	Aufruf der Methode *decide()* an der Strategie
11	Aufruf der Methode *firstOffer()* an der Strategie
12	Aufruf der Methode *counterOffer()* an der Strategie
13	Aufruf der Methode *acceptOffer()* an der Strategie
14	Aufruf der Methode *abortOffer()* an der Strategie
15	Aufruf der Methode *counterOfferOrAccept()* an der Strategie
16	Aufruf der Methode *counterOfferOrAbort()* an der Strategie
17	Aufruf der Methode *acceptOrAbortOffer()* an der Strategie
2	keine Aktivität (Waiting)
3	Abbruchaktivitäten (Aborting), wie das Löschen voriger Angebote
4	Aktivitäten zum Abschließen des Vertrages (Preparing)
n	keine Aktivität bzw. frei definierbar

Tabelle 6.11: Zuordnung der Werte des Zustands–Propertys zu Aktivitäten

Die Zuordnung der Werte des Zustands-Properties zu den Aktivitäten ist in Tabelle 6.11 dargestellt. Wenn eine Methode an dem Agenten Rule–sensitiv

aufgerufen werden soll und der Agent sich nicht in dem für das Verhandlungsprotokoll relevanten korrekten Zustand befindet, dann wird eine *RuleException* erzeugt und damit der Aufruf der Methode unterbunden. Damit in einem Fehlerfall das Zustands–Property eines Agenten nicht in einem Zustand bleibt, in dem der Agent nicht mehr ansprechbar ist, kann ein Timeout dafür sorgen, dass nach einer gewissen Zeit das Zustands–Property zurückgesetzt wird.

Um eine Verhandlungsstrategie mit Rules zu steuern, müssen folgende Anforderungen erfüllt sein. Da Strategie mit ihrer Umwelt über die Properties aus den Rule–Sets des Agenten kommuniziert, wird zunächst für eine Strategie eine eigene Rule–Set (*Strategy Set*) definiert. Diese Properties modellieren je nach Strategie die Anzahl der Verhandlungsrunden, die Größenordnung des Verhandlungsschrittes, um den das nächste Angebot erhöht oder erniedrigt wird etc..

Außerdem muss der Agent eine von der Strategie vorgegebene Schnittstelle implementieren, deren Methoden die *Konfigurationspunkte* für Rules darstellen, d.h. sie werden (ausschließlich) dazu benutzt, um Rules triggernde Events auszulösen. Diese Schnittstelle wird so implementiert, dass die Methoden leere Methodenrümpfe aufweisen. Sie dienen lediglich der Konfiguration von Rules, weshalb sie hier als Konfigurationspunkte bezeichnet werden. Innerhalb der Strategie wird dann die jeweilige Methode (Konfigurationspunkt) Rule–sensitiv an dem Agenten aufgerufen. Dadurch werden die an dem Konfigurationspunkt konfigurierten Rules gefeuert, die die Properties der Strategie auswerten bzw. verändern können.

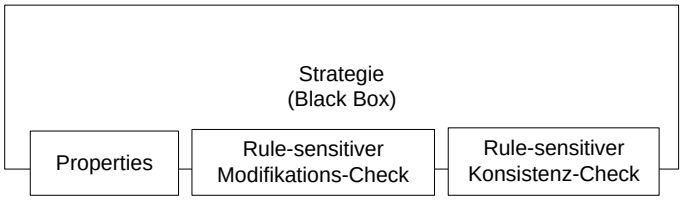


Abbildung 6.53: Schnittstellen einer Rule–sensitiven Strategie

Dabei wird unterschieden zwischen Methoden, die für Konsistenzprüfungen von der Strategie aufgerufen werden, und Methoden, die zum Modifizieren der Properties der Strategie aufgerufen werden. Die Strategie fängt bei konsistenzprüfenden Konfigurationspunkten *RuleExceptions* ab und behandelt diese. Im Gegensatz dazu werden *RuleExceptions* bei modifizierenden Konfigurationspunkten von der Strategie ignoriert. Es können dennoch Rules mit einer pessimistischen Strategie genutzt werden, um Änderungen an Properties — durch Ausnutzung der Eigenschaft der Isoliertheit der Transaktion — nicht durchführen zu lassen. Abbildung 6.53 stellt die Voraussetzungen für eine Rule–gesteuerte Strategie dar.

Beispielsweise kann die Rule–Steuerung der Strategie über eine Schnittstelle erfolgen, die zwei Methoden wie folgt deklariert: *checkConsistency()* und *modifyStrategy()*. In diesem Beispiel ist die Methode *checkConsistency()* dafür vorgesehen, Requirement–Rules zu konfigurieren, um ein Angebot zu prüfen (wie etwa die in Tabelle 6.8 und 6.9). Die Methode *modifyStrategy()* kann für die Konfiguration Property–schreibender Rules benutzt werden, um z.B. mit einer State–Transition–Rule bei Erreichen einer bestimmten Anzahl von Verhand-

lungsrunden ein Abbruch-Property zu setzen, das von der Strategie interpretiert wird und die Verhandlung dann abbricht. Oder es kann nach einer gewissen Anzahl von Verhandlungsrunden der Betrag des Verhandlungsschrittes verändert werden, so dass das nächste Angebot niedriger oder höher ausfällt (vgl. Tabelle 6.10).

6.4.4 Interaktion zwischen Protokoll und Strategie

In diesem Abschnitt sollen nun die Rule–Interaktionen zur Steuerung automatisierter Verhandlungen im Gesamtüberblick dargestellt werden. Dabei soll auch das Zusammenspiel zwischen Protokoll– und Strategie–Rules deutlich gemacht werden.

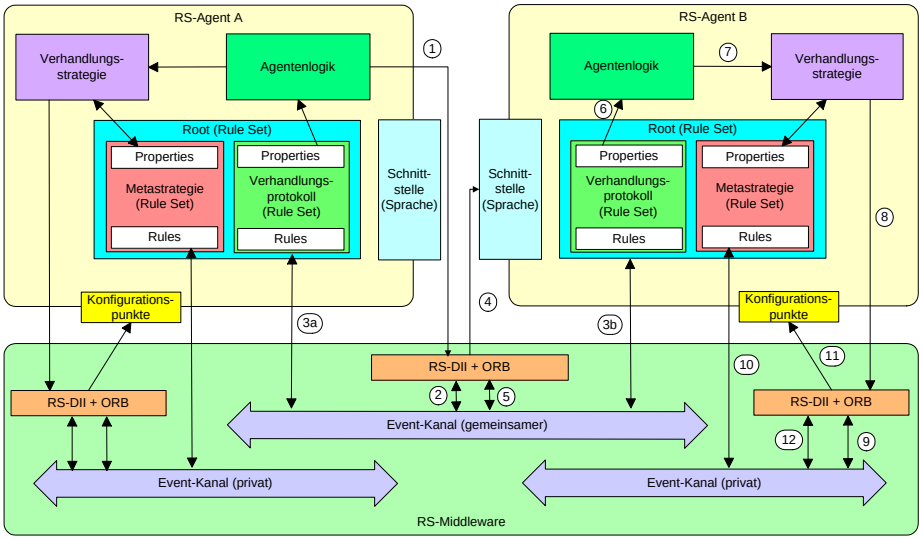


Abbildung 6.54: Steuerung autonom verhandelnder Agenten mit Rules

In Abbildung 6.54 ist die komplette Rule–Unterstützung der Verhandlungsphase und damit die Koordination von Verhandlungsprotokoll und Verhandlungsstrategie dargestellt. Es sind zwei Verhandlungsteilnehmer, Agent A und Agent B, dargestellt, die miteinander über die Rule–sensitive Middleware kommunizieren.

Konzeptionell wird für die Rule–Steuerung der Strategie aus Gründen der Autonomie der Agenten ein privater Kommunikationsmechanismus gewählt, der in Abbildung 6.54 durch einen privaten Event–Kanal hervorgehoben ist. Im Gegensatz dazu wird zur Steuerung des Protokolls ein gemeinsamer Kommunikationsmechanismus benötigt, der durch einen geteilten Event–Kanal dargestellt ist. Die Rule–sensitive Middleware wurde bereits in Abschnitt 3.3.3 beschrieben, so dass hier nur soweit wie nötig auf deren Mechanismen eingegangen wird. Im folgenden soll die Abbildung 6.54 anhand des bilateralen Verhandlungsprotokolls in Abschnitt 6.4.2.1 dargestellt werden.

Der Agent A startet die Kommunikation, indem er dem Agenten B die Nachricht *offer()* über das RS–DII zusendet **(1)**. Der Parameter der Nachricht *offer()* ist ein Bedingungsausdruck, der eine beliebige Anzahl von verhandelbaren Ei-

genschaften enthalten kann. Das RS–DII erzeugt vor dem Methodenaufruf ein Rule–Event, das beiden Agenten über den gemeinsamen Event–Kanal übermittelt wird **(2,3a,3b)**. Ist die Nachricht nicht erlaubt, wird eine *RuleException* ausgelöst, die die weitere Kommunikation stoppt. Ist die Nachricht erlaubt, kann das RS–DII die Nachricht über den ORB (Object Request Broker) an den Agenten B vermitteln **(4)**. Der Agent B merkt sich das Angebot. Das RS–DII verteilt wieder ein Rule–Event über den gemeinsamen Event–Kanal an beide Agenten **(5,3a,3b)**. Dadurch wird das Zustands–Property des Agenten B und eventuell auch das des Agenten A mit einer State–Transition–Rule protokollabhängig geändert. Da der Aufruf über das RS–DII transaktional ist, werden die veränderten Properties erst bei Beendigung des Aufrufs sichtbar. Das Versenden der Nachricht *offer()* über das RS–DII ist nun abgeschlossen.

Die Änderung des Zustands–Propertys wird dem Agenten B von der Rule–Set mitgeteilt **(6)**, dessen Wert besagt, dass ein Angebot eingetroffen ist. Daraufhin wird die mit diesem Zustand verbundene Aktivität ausgeführt, die die Strategie benachrichtigt **(7)**. Die Aufgabe der Strategie besteht nun darin, das Angebot zu prüfen, gegebenenfalls ein Gegenangebot zu machen, anzunehmen oder abzulehnen. Die Strategie ruft zur Überprüfung des Angebots einen Konfigurationspunkt, nämlich die Methode *checkConsistency()* Rule–sensitiv über das RS–DII bei dem Agenten B auf **(8)** und findet damit heraus, ob das Angebot akzeptabel ist. Das RS–DII verteilt wieder vor dem Methodenaufruf ein Rule–Event an den Agenten B, diesmal über den privaten Event–Kanal **(9,10)**. Dadurch wird eine Requirement–Rule bei dem Agenten B gefeuert, die bei Nichtakzeptanz eine *RuleException* auslöst. Falls keine *RuleException* geworfen wird, ruft das RS–DII den Konfigurationspunkt auf **(11)** und verteilt wieder ein Rule–Event, das an die Rules des Agenten B verteilt wird **(12,10)**.

Der Aufruf des ersten Konfigurationspunktes über das RS–DII ist damit abgeschlossen. Wurde das Angebot akzeptiert, dann wird im Anschluss daran die Methode *accept()* bei dem Agenten A aufgerufen. Kam es jedoch zu einer *RuleException* beim Prüfen des Angebotes, dann wird in der Fehlerbehandlung ein neues Gegenangebot erstellt. Der zweite Konfigurationspunkt *modifyStrategy()* wird vor dem Erstellen des Gegenangebotes zum Modifizieren der Strategie–Properties über das RS–DII beim Agenten B aufgerufen. Der Ablauf ist der gleiche wie bei dem ersten Konfigurationspunkt. Im Anschluss daran erzeugt die Strategie die Nachricht *offer()* mit dem Gegenangebot als Parameter, die über das RS–DII dem Agenten A zugesendet wird. Der Agent A durchläuft dann die gleichen Schritte wie zuvor der Agent B. Dieser Vorgang wiederholt sich solange, bis die Agenten entweder zu einer Übereinkunft kommen oder einer der beiden Agenten die Verhandlung abbricht.

6.5 Zusammenfassung

In diesem Kapitel wurde die generische und semantisch offene agentenbasierte Verhandlungsarchitektur aus dem vorherigen Kapitel in ein konkretes und funktional vollständiges Verhandlungssystem umgesetzt. Dies bedeutet, dass alle Mechanismen zu einer Automatisierung von Verhandlungsprozessen konkretisiert werden müssen.

Dazu gehören zunächst agentenspezifische Mechanismen. Es wurde aufgezeigt, wie der so genannte *Plug-in*–Mechanismus zur dynamischen Einbindung

von Anwendungsfunktionalität in ein mobiles Agentengerüst implementiert werden kann. Aus dieser Implementation wird auch deutlicher, dass dieser Mechanismus keineswegs auf die besondere Semantik von verhandelnden Agenten beschränkt, sondern im allgemeinen zur *dynamischen Komposition* von beliebigen (aber anwendungsspezifischen) Agenten verwendet werden kann. Zu den Agentenmechanismen wurde ebenfalls das *Kommunikationsmodul* eingeordnet, da die Kommunikationsfähigkeit zu den grundlegenden Eigenschaften von allen Agenten gehört. Um dieses Modul zu realisieren, wurde KQML als eine standardisierte Agentenkommunikationssprache gewählt und ein entsprechendes KQML–Produkt mit dem Plug-in–Mechanismus integriert.

Als nächstes mussten konkrete *Verhandlungsprotokolle* zur Steuerung verhandelnder Agenten realisiert werden. Da die Anforderungen an solche Protokolle — wie die Analyse im letzten Kapitel deutlich macht — sehr vielfältig sein können, wurde auch hierfür ein generischer Lösungsansatz vorgeschlagen, nämlich die Konzeption einer deklarativen, auf gefärbten Petrinetzen basierten Beschreibungssprache, mit der möglichst viele verschiedene Verhandlungsprotokolle spezifiziert werden können. Im Gegensatz zu rein nachrichtenorientierten Sprachen ist die hier vorgeschlagene *prozessorientierte* Sprache in der Lage, auch *dynamische* Aspekte von automatisierten Verhandlungsprozessen zu behandeln. Aufbauend auf dieser Sprache wurden dann Komponenten zur Generierung und Ausführung von Verhandlungsprotokollen entwickelt, mit denen verhandelnde Agenten bei der Einhaltung eines Protokolls *effektiv* unterstützt werden können.

Desweiteren wurde ein auf Genetischen Algorithmen basiertes Framework zur Realisierung konkreter *Verhandlungsstrategien* dargestellt. Hauptziel der Entwicklung dieses Frameworks war es, effektive, maschinell generierte Strategien zu finden, die sich an eine offene, dynamische Umgebung *adaptieren* können. Dazu wurde zunächst als Framework–Kern eine auf *endlichen Automaten* operierende Bibliothek von genetischen Algorithmen entworfen und implementiert. Darauf aufbauend wurden dann unterschiedliche *Verhandlungsszenarien* — einschließlich bilateraler und Auktionen als einer Form multilateraler Verhandlungen — entwickelt und eine Reihe entsprechender Experimente durchgeführt. Die erzielten Ergebnisse wurden graphisch aufbereitet, analysiert und bewertet.

Schließlich wurde in diesem Kapitel auch detailliert dargestellt, wie die im ersten Teil dieser Arbeit (Kapitel 2 bis 4) behandelten Regelverarbeitungsmechanismen zur *Steuerung automatisierter Verhandlungen* eingesetzt werden können. Dabei wurde insbesondere auch gezeigt, dass sowohl einfache, nachrichtenorientierte Verhandlungsprotokolle als auch ebenfalls relativ einfache (Meta–) Verhandlungsstrategien *ausschließlich* in Form von *Rules* realisiert werden können, wenn gewisse Rahmenbedingungen bezüglich der internen Funktionsweise der Agenten, bspw. dass sie explizite *Konfigurationspunkte* aufweisen, gewährleistet sind. Die Realisierung dieser Bedingungen führten auch hier zu einem Anwendungsframework, das insbesondere das *Zusammenspiel* zwischen Protokoll– und Strategie–Rules festlegt.

Kapitel 7

Resümee und Ausblick

Unseren Umgang mit Wissen und Information brachte T. S. Eliot[1] einmal wie folgt zum Ausdruck:

> *Where is the wisdom we have lost in knowledge?*
> *Where is the knowledge we have lost in information?*

Wenn man nicht weiß, dass diese Zeilen einem Gedicht von 1934 [Eli74] entstammen, könnte man gewiss annehmen, dass sie sich auf den aktuellen Zustand unserer "Informationsgesellschaft" — ein Begriff, der jünger als das Zitat zu sein scheint — beziehen. Trotz oder besser gesagt *mit* allen unbestreitbaren Vorteilen und Fortschritten, die uns die Informationstechnologie beschert, drängt sich auch immer mehr die Gefahr der Informationsüberflutung und der *Unbeherrschbarkeit* von Informationssystemen auf, vor der Kritiker lange gewarnt haben (siehe z.B. [Wei78]). Obwohl in der vorliegenden Arbeit der in diesem Zitat wiederholt auftretende *Wissen*sbegriff nicht — oder zumindest nicht *explizit*[2] — behandelt wird, könnte sie rückblickend als ein Versuch zur Verbesserung der Beherrschbarkeit von Informationssystemen, insbesondere verteilten Anwendungssystemen, gewertet werden.

Hierbei spielt der Charakter der *Interaktionsorientierung* verteilter Anwendungskomponenten in einer *offenen* Umgebung eine zentrale Rolle: Es sollen systemtechnische Mechanismen geschaffen werden, um verteilte Anwendungen in ihrem Interaktionsverhalten zueinander zu steuern und aktiv zu unterstützen, ohne ihre *Autonomie* und ihre Funktionalität zu beeinträchtigen. Dazu gehören zum einen regelbasierte, interaktionsorientierte Steuerungsmechanismen und zum anderen automatisierte Verhandlungsmechanismen, wobei letztere einerseits als funktionale Erweiterung der Regelmechanismen betrachtet werden, andererseits aber auch von diesen gesteuert und unterstützt werden können.

Daher wurde in dieser Arbeit zunächst auf die wesentlichen Aspekte offener verteilter Anwendungen eingegangen, die bei der Konzeption und Realisierung der angestrebten Steuerungsmechanismen zu berücksichtigen waren, wobei das *Interaktionsparadigma* hervorgehoben wurde, da es das Leitmotiv für die Erarbeitung der dargestellten Konzepte und Lösungsansätze darstellt. Da die

[1] Amerikanischer Dichter und Literaturnobelpreisträger von 1948

[2] Sowohl die logikbasierten Regeln zur Steuerung verteilter Anwendungssysteme als auch die auf genetischen Algorithmen basierten Verhandlungsstrategien, die in dieser Arbeit behandelt wurden, können als eine Form *impliziten* Wissens betrachtet werden.

zu behandelnden Mechanismen konkrete Anwendung im Bereich *elektronischer Dienstemärkte* finden sollen, wurde dann der vielfältige technische Hintergrund dieses praxisnahen Gebiets beleuchtet.

Anschließend wurden dann grundlegende Aspekte der *Modellierung* von regelbasierten Steuerungsmechanismen behandelt. Hierzu wurden zunächst logische und mathematische Grundlagen, insbesondere relevante Konzepte der Prädikatenlogik sowie das Simplexverfahren zur Lösung von Ungleichungssystemen, die für eine *generische* Realisierung von Steuerungsverfahren benötigt werden, erläutert. Es wurde dann im einzelnen gezeigt, wie eine auf diesen formalen Grundlagen basierte generische Verarbeitung von Steuerungsregeln theoretisch konzipiert werden kann. Dabei wurden die Verarbeitungsfunktionen auf zwei unterschiedlichen Ausdrucksmächtigkeitsstufen betrachtet, die entsprechend unterschiedlich komplexe Algorithmen benötigen.

Um die Steuerungskonzepte in Mechanismen umzusetzen, die in praxisrelevanten Umgebungen effizient eingesetzt werden können, sind geeignete *Systemarchitekturen* erforderlich. Diesbezüglich stellte sich bei der Analyse bestehender Systeme und Lösungsansätze heraus, dass sowohl eine *zentrale* als auch eine *dezentrale* Architektur zur Anwendung von Regeln sinnvoll sind, da sehr unterschiedliche *praktische* Anforderungen an eine konkrete Umsetzung der konzeptionell völlig generischen Regelkonzepte gestellt werden können. Dementsprechend wurden zwei unterschiedliche Systemarchitekturen präsentiert, wobei jedoch die *dezentrale* dem Konzept der Interaktionsorientierung erheblich besser entspricht, was sich u.a. in der expliziten Unterscheidung zwischen lokalen und entfernten *Aktivierungsmodi* für die Regeln manifestiert.

Bei der konkreten *Implementierung* der architekturellen Konzepte war es wichtig, technische Randbedingungen einer Integration in bestehende Middleware– und Agentenarchitekturen sowie Basistechnologien für elektronische Dienstemärkte zu berücksichtigen, um eine möglichst hohe Interoperabilität mit existierenden Systemen zu gewährleisten. Dies hat auch wesentlichen Einfluss auf die Wahl der Programmiersprache und Softwareprodukte, auf den die beschriebenen Steuerungsmechanismen aufsetzen können. Durch die im Anschluss daran entwickelten *Anwendungsszenarien* wurde der Nachweis erbracht, dass die implementierten Lösungen tatsächlich in unterschiedlichen Anwendungssystemen eingesetzt werden können.

Als eine weitere Form der Interaktionsunterstützung und -steuerung wurden dann automatisierte Verhandlungsmechanismen untersucht. Eine eingehende *Analyse* wurde vorgenommen, um bestehende Konzepte und Anforderungen an eine vollständige Automatisierung von Verhandlungsprozessen deutlich zu machen. Daraus wurde eine *Architektur* selbstständig *verhandelnder Agenten* konzipiert, die auf der Kombination von dynamisch austauschbaren Verhandlungsmodulen und entsprechenden Unterstützungsdiensten basiert.

Um diese Architektur in ein konkretes *Verhandlungssystem* für mobile Softwareagenten umzusetzen, mussten allerdings nicht nur die *Agentenmechanismen* implementiert, sondern vor allem konkrete Lösungsansätze zur Realisierung von *Verhandlungsprotokollen und -strategien* herausgearbeitet werden. Zur Spezifikation beliebiger Verhandlungsprotokolle wurde eine generische, auf gefärbten Petrinetzen basierte *deklarative Sprache* entworfen. Unterschiedliche Tools und Dienste, um die *Erstellung und Generierung* von Protokollen zu unterstützen bzw. um die Agenten bei der *Ausführung* von Verhandlungsprozessen protokollkonform zu steuern, wurden dann vorgeschlagen, realisiert und ausführlich

beschrieben. Für die Realisierung von Verhandlungsstrategien wurde ein Framework zur Anwendung von *genetischen Algorithmen* entwickelt und in einer Reihe von unterschiedlichen *Verhandlungsszenarien* erprobt. Die dabei erzielten Ergebnisse wurden präsentiert und bewertet.

Als **Gesamtfazit** lässt sich festhalten, dass mit dieser Arbeit ein Satz von *generischen Rahmenwerken* zur Steuerung offener verteilter Anwendungssysteme präsentiert wurde. Die von diesen Rahmenwerken angebotenen Regelverarbeitungs- und Verhandlungsmechanismen sind dabei durch einen hohen Grad an *Dynamik*, *Transparenz*, und insbesondere *Interaktionsorientierung* gekennzeichnet. Ihre praktische Anwendbarkeit wurde einerseits durch die Integration in praxisrelevante Systeme und Softwareprodukte und andererseits durch eine Vielzahl von *Anwendungsszenarien* sichergestellt. Darüber hinaus wurde die enge Verbindung zwischen den beiden in dieser Arbeit verfolgten Hauptansätzen zur Steuerung verteilter Anwendungen, nämlich interaktionsorientierter Regelverarbeitung und Automatisierung von Verhandlungsprozessen, durch die Realisierung *regelgesteuerter Verhandlungsszenarien* konkret aufgezeigt.

Hauptsächlich auf Grund des Rahmenwerk–Charakters sowie der Generizität der präsentierten Konzepte und Mechanismen lassen sich jedoch in vielerlei Hinsicht noch mögliche Erweiterungen vornehmen. Am wichtigsten erscheinen dem Autor dabei folgende Aspekte:

- **Mögliche Erweiterungen der Regelverarbeitungsmechanismen**: Die im Rahmen dieser Arbeit entwickelten regelbasierten Steuerungsmechanismen können aus einer allgemeinen Sichtweise als erste Versuche betrachtet werden, logik- und regelbasierte Verfahren für das Gebiet der offenen verteilten Anwendungen konkret nutzbar zu machen. Diesbezüglich bieten die Gebiete Formale Logik und Künstliche Intelligenz eine immense Fülle an Regelverarbeitungsmechanismen, die *prinzipiell* sowohl zur Verbesserung der implementierten generischen Verarbeitungsfunktionen als auch zur Erweiterung der Ausdrucksmächtigkeit der Regeln herangezogen werden können. Eine essenzielle Erweiterung der bestehenden Funktionalität wäre beispielsweise die direkte und orthogonale Integration *modallogischer* Funktionen in die PM– und RS–Bibliotheken. Die Anwendungsszenarien in Abschnitt 4.4.3 zeigen nämlich deutlich, dass die allgemeine Verfügbarkeit von modalen Operatoren zur Spezifikation von Kooperationsanforderungen sehr sinnvoll wäre. Bisher können modale Operatoren jedoch nur durch eine zusätzliche Anwendungsschicht, die eine Reihe von Einschränkungen bezüglich der Semantik dieser Operatoren erfordert, verarbeitet werden. Auch eine Erweiterung der Ausdrucksmächtigkeit um *temporallogische* Operatoren ist ebenso vorstellbar, da die zeitliche Gültigkeit von Regeln bzw. den mit ihnen ausgedrückten Anforderungen dann (direkt) spezifizierbar wäre.

  Eine weitere Erweiterungsmöglichkeit betrifft die Konfiguration von Rules für eine Anwendung bzw. einen Agenten. Wie der letzte Hauptabschnitt des letzten Kapitels (6.4) zeigt, können an einem verhandelnden Agenten prinzipiell sehr komplexe Rule–Konfigurationen vorgenommen werden. Dabei ist zu beachten, dass auf Grund der dynamischen Interaktionen eine neu hinzugefügte bzw. wieder entfernte Rule sich nicht nur auf den Agenten, an dem sie konfiguriert ist, sondern auf das gesamte Anwendungssystem auswirken kann. Dies verdeutlicht den Bedarf an entsprechenden

Werkzeugen, welche die *Entwicklung* von Rule–Konfigurationen semantisch besser unterstützen, als das im Rahmen dieser Arbeit verwirklicht wurde (vgl. Beschreibung der Werkzeuge in Abschnitt 4.4.1 und 4.4.4.2).

- **Mögliche Erweiterungen der Verhandlungsmechanismen**: Da das präsentierte Verhandlungssystem von der konzeptionellen Architektur bis zur Implementation — insbesondere die *Plug-in*–Architektur der Agenten — modular aufgebaut ist, lassen sich grundsätzlich alle Module durch andere Realisierungen ersetzen. Das *Kommunikationsmodul* kann beispielsweise durch solche, die eine andere Sprache als KQML verarbeiten können, ausgetauscht werden. Hierzu bietet sich insbesondere XML als Alternative an, da diese Sprache bereits als neuer Standard im Bereich Kommunikationssprachen für offene verteilte Anwendungen etabliert zu sein scheint. In Bezug auf *Verhandlungsprotokollen* könnte u.a. eine *dezentrale* Steuerung von Verhandlungsprozessen als Alternative zu der zentralen Protokoll–Engine überlegenswert sein. Im dezentralen Fall müsste das (bei dem Agenten befindliche) Protokollmodul allein und auf eine abgekoppelte Weise die Protokollkonformität sicherstellen und die beteiligten Agenten hätten auf diese Weise wesentlich mehr Autonomie. Jedoch gilt eine solche dezentrale Realisierung bei dem gegenwärtigen Kenntnisstand als sehr aufwendig und schwierig.

  In Bezug auf die Realisierung von *Verhandlungsstrategien* sind besonders viele sinnvolle Erweiterungen denkbar. Die gegenwärtig genutzten Strategien basieren auf relativ simplen genetischen Algorithmen (es werden bspw. nur *Punktmutationen* eingesetzt). Sie könnten durch andere Methoden aus dem Gebiet der Genetischen Programmierung ersetzt werden. Desweiteren sollten weitere Verhandlungsszenarien entwickelt und Experimente in einem größeren Umfang durchgeführt werden, um genauere Aussagen über die Praxistauglichkeit von genetischen Algorithmen treffen zu können. Es ist aber offensichtlich, dass prinzipiell auch ganz andere (intelligente) Verfahren für diesen Zweck genutzt werden können. Außerdem erscheint es — unabhängig von spezifischen Algorithmen — für das Strategiemodul der Agentenarchitektur sinnvoll, eine eigene *Strukturierung* so zu schaffen, dass gewisse informationsverarbeitungstechnische Eigenschaften gewährleistet bzw. unterstützt werden. Hierzu wurden bereits im Rahmen dieser Arbeit erste konkrete Ansätze zu einem auf dem Konzept der *Aktoren* basierten Framework, das u.a. die Eigenschaften der *parallelen Nutzung* mehrerer Strategiealgorithmen unter *eingeschränkten Rechenressourcen* unterstützt, entwickelt (siehe [TSGL99, TSL00]).

- **Sicherheitsmechanismen**: Da bei der Realisierung vieler Konzepte in dieser Arbeit mobile Agenten als eine Basistechnologie eingesetzt werden, ergibt sich ein grundsätzliches Sicherheitsrisiko, das bei einem praktischen Einsatz berücksichtigt werden muss. Die Agententechnologie an sich birgt bereits deutliche Sicherheitsgefahren, die Gegenstand eines eigenen Forschungszweiges darstellen (s. z.B. [CGH+95]). Durch die in dieser Arbeit dargestellten *dynamischen* Systemmechanismen — insbesondere den *Plug-in*–Mechanismus für mobile Agenten — ergeben sich jedoch *zusätzliche* potenzielle Sicherheitsprobleme, da die *Modifizierung* eines Agenten in diesem Fall prinzipiell nicht als ein Sicherheitsangriff, sondern als ei-

ne explizite Eigenschaft / Funktionalität angeboten wird. Daraus ergeben sich drei neue Varianten der unerwünschten Verwendung der Agenten:

- Unerlaubter Austausch eines Moduls (d.h. neuer Einsatz, Austausch oder Entfernen eines Moduls)
- Überladen des Agenten mit (unnötigen) Modulen
- Stören eines Moduls durch fremde Aufrufe (*Spamming*)

Um diese Probleme zu lösen, wurde im Rahmen dieser Arbeit ein auf *delegierbaren Capabilities* basiertes Sicherheitsframework konzipiert, das die dynamische und anwendungstransparente Einbindung von Autorisierungsmechanismen beim Zugriff auf Agentenmethoden (insbesondere die *plug()*–Methode) zum Austausch von Modulen) ermöglicht. Details dieses Frameworks sind in [GJ99] enthalten.

- **Weitere Erschließung der Anwendungsgebiete**: Die in der vorliegenden Arbeit beschriebenen Steuerungs- und Verhandlungsmechanismen sind gewollt so generisch modelliert und realisiert worden, dass sie nicht nur für das hier exemplarisch gewählte Gebiet der elektronischen Dienstemärkte nutzbar, sondern im Prinzip auch in weiteren Anwendungsgebieten einsetzbar sind. Als konkretes Beispiel hierfür wurde ein neues Projekt konzipiert, das sich mit der agentenbasierten *Krankenhausorganisation* beschäftigt. In dieses von der DFG im Rahmen des Schwerpunktprogramms "Intelligente Softwareagenten und betriebswirtschaftliche Anwendungsszenarien" geförderte Projekt werden viele der hier dargestellten Konzepte einfließen.

  Auch innerhalb des Gebiets E–Commerce sind weitere Anwendungsmöglichkeiten denkbar. Beispielsweise könnte im Sinne von *E–Business*–Architekturen [BL00] ein (konzeptionelles) Framework zur Unterstützung eines umfassenden Unternehmensorganisationsmodells (einschließlich der Bereiche Unternehmens–Kooperation und Elektronischer Handel) mit den dargestellten Steuerungs- und Verhandlungsmechanismen erstellt werden.

Anhang A

Spezifikationen zum Policy–Manager

A.1 Entscheidungstabellen

In diesem Abschnitt finden sich Entscheidungstabellen für die generische Verarbeitung von Steuerungsregeln in der Basisausdrucksmächtigkeit (s. Abschnitt 2.3.2).

A.1.1 Logische Implikation

Seien $a = (x \lhd a_v)$, $b = (x \rhd b_v)$ atomare Bedingungen nach Definition 2.3.1 mit Relationen $\lhd$ und $\rhd$.

Es gilt $a \rightarrow b$ unter den in Tabelle A.1 aufgelisteten Bedingungen.

$\lhd \setminus \rhd$	$=$	$<$	$\leq$	$>$	$\geq$	$\neq$
$=$	$b_v = a_v$	$b_v > a_v$	$b_v \geq a_v$	$b_v < a_v$	$b_v \leq a_v$	$b_v \neq a_v$
$<$		$b_v \geq a_v$	$b_v \geq a_{v\ -1}$			$b_v \geq a_v$
$\leq$		$b_v > a_v$	$b_v \geq a_v$			$b_v > a_v$
$>$				$b_v \leq a_v$	$b_v \leq a_{v\ +1}$	$b_v \leq a_v$
$\geq$				$b_v < a_v$	$b_v \leq a_v$	$b_v < a_v$
$\neq$						$b_v = a_v$

Tabelle A.1: Logische Implikation zwischen atomaren Bedingungen der Basisausdrucksmächtigkeit

A.1.2 Widerspruch

Es gilt $a \wedge b \equiv FALSE$ unter den in Tabelle A.2 aufgelisteten Bedingungen.

Dabei beziehen sich die kleingedruckten Zusätze $+1$ und -1, wie in Abschnitt 2.3.2.2 erläutert, nur auf ganzzahlige Werte a_v bzw. b_v.

$\lhd \setminus \rhd$	$=$	$<$	$\leq$	$>$	$\geq$	$\neq$
$=$	$b_v \neq a_v$	$b_v \leq a_v$	$b_v < a_v$	$b_v \geq a_v$	$b_v > a_v$	$b_v = a_v$
$<$	$b_v \geq a_v$			$b_v \geq a_v$ $_{-1}$	$b_v \geq a_v$	
$\leq$	$b_v > a_v$			$b_v \geq a_v$	$b_v > a_v$	
$>$	$b_v \leq a_v$	$b_v \leq a_v$ $_{+1}$	$b_v \leq a_v$			
$\geq$	$b_v < a_v$	$b_v \leq a_v$	$b_v < a_v$			
$\neq$	$b_v = a_v$					

Tabelle A.2: Widerspruch zwischen atomaren Bedingungen der Basisausdrucksmächtigkeit

A.1.3 Normierung negierter Atome

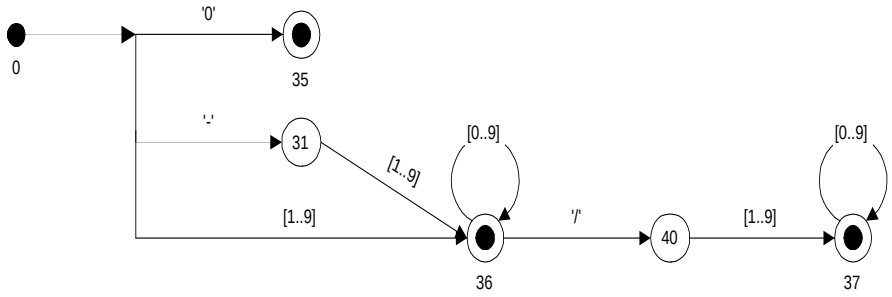
Atom mit Negation	äquivalenter atomarer Ausdruck
$\neg(x = a)$	$x \neq a$
$\neg(x < a)$	$x \geq a$
$\neg(x \leq a)$	$x > a$
$\neg(x > a)$	$x \leq a$
$\neg(x \geq a)$	$x < a$
$\neg(x \neq a)$	$x = a$

Tabelle A.3: Normierung negierter atomarer Bedingungen

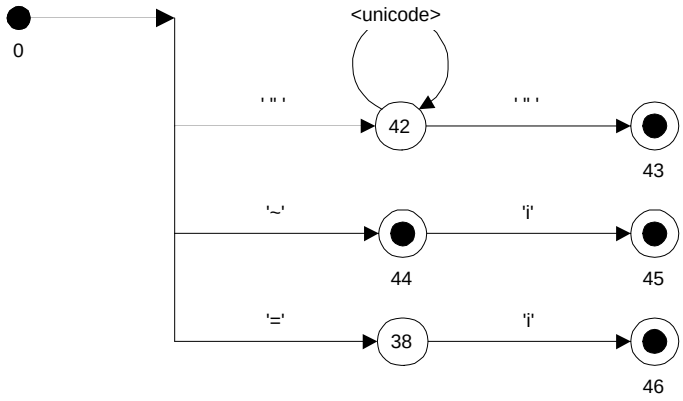
A.2 EPML Parser

A.2.1 Lexikalischer Analyzer

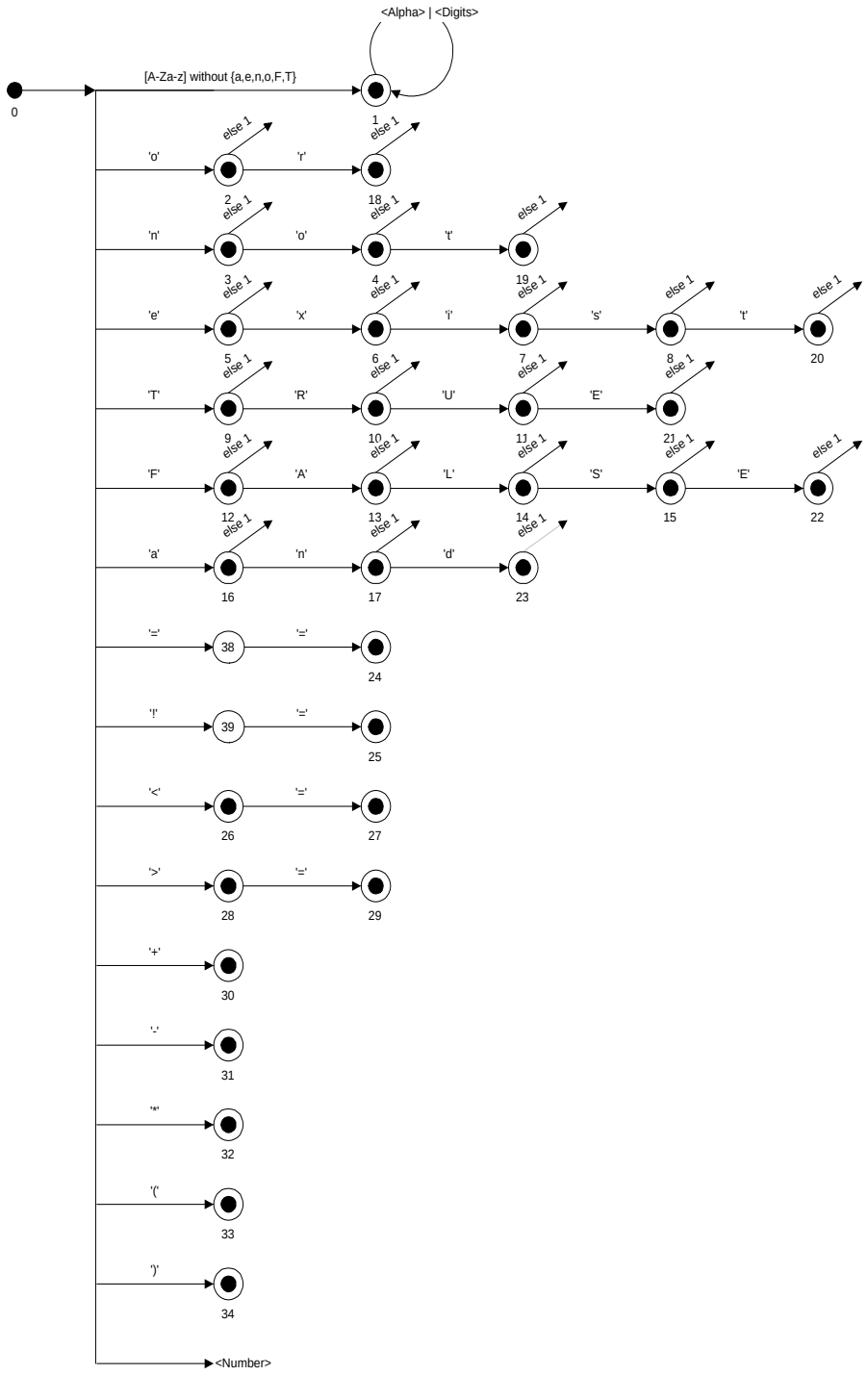
Erkennung von Zahlen



Erkennung von Strings



Erkennung von Zeichen



A.2.2 Syntax

```
<constraint>        = <bool_or>

<bool_or>           = <bool_and> or <bool_or>
                    | <bool_and>

<bool_and>          = <bool_not> and <bool_and>
                    | <bool_not>

<bool_not>          = not ( <bool_compare> )
                    | <bool_compare>

<bool_compare>      = <atomic_expression>
                    | exist <Ident>
                    | TRUE
                    | FALSE

<atomic_expression> = <left_side> <comp> <Number>
                    | <Ident> <string_comp> <string>

<comp>              = == | < | <= | > | >= | /* != */

<left_side>         = <mult> + <left_side>
                    /* | <mult> - <left_side>  */
                    | <mult>

<mult>              = <Number> * <Ident>

<string>            = "<Unicode>"


<string_comp>       = == | != | ~ | =i | ~i
```

Aufschlüsselung der Terminale in Token:

```
<Number>    = -<Digits>
            |  <Digits>
            | -<Digits> / <Digits>
            |  <Digits> / <Digits>

<Digits>    = <Digit> <Digits>
            | <Digit>

<Digit>     = [0-9]

<Ident>     = <Leader> <FollowSeq>

<FollowSeq> = <FollowSeq> <Follow>

<Leader>    = [A-Za-z]

<Follow>    = [A-Za-z0-9]
```

A.3 CORBA–Schnittstellen

A.3.1 PolicyPersistence IDL

```
module PolicyPersistence {

  typedef sequence<string> stringArray;
  typedef string propSetPolicy[2];

  // definition of exceptions:
  exception PPPolicyNameExistsException {};
  exception PPNoSuchPolicyException {};

  // definition of interface:
  interface PPAccessAdaptor {

    // format for policyName == propSet:policyName:
    void set(in string policyName, in string policyConstraint)
    raises(PPPolicyNameExistsException);

    propSetPolicy get(in string policyName)
    raises(PPNoSuchPolicyException);

    // return true if deletion succeed, false otherwise:
    boolean del(in string policyName);
    stringArray getPolicyList();

  };

};
```

A.3.2 PropertyService IDL

```
module PropertyService {

  exception PSAccessException {};
  exception PSDynamicTypeException {};

  // the first is the typename, the second the value:
  typedef sequence<string, 2>typeValue;

  struct PSNameTypeValue {

    string propertyName;
    typeValue value;

  };

  typedef sequence<PSNameTypeValue>PSNameTypeValueArray;
  typedef sequence<string>stringArray;

  interface PSAccessAdaptor {

    boolean isDefined(in string name) raises(PSAccessException);
    typeValue   get(in string name) raises(PSAccessException);
    void set(in string name, in typeValue value)
    raises(PSDynamicTypeException, PSAccessException);
    boolean delete(in string name);
    stringArray getSetNames();
    PSNameTypeValueArray getAllProperties();
    void store();

  };

};
```

A.3.3 ExtendedPolicyManager IDL

```
#include "PropertyService.idl"

module ExtendedPolicyManager {

  // definition of exceptions:
  exception EPMNotInDNFException{};
  exception EPMNotOptimizalableException {};
  exception EPMEmptySolutionSetException {};
  exception EPMUnboundedException {};
  exception EPMDevisionByZeroException {};
  exception EPMNumberFormatException {};
  exception EPMSyntaxException {string reason;};

  // predeclaration of interfaces:
  interface EPMPolicyManager;
  interface EPMPolicy;
  interface EPMBasisSolution;

  // typedefs:
  typedef sequence<string>stringArray;
  typedef sequence<EPMBasisSolution>EPMBasisSolutionArray;
  typedef sequence<double>doubleArray;


  // declaration of interfaces:
  interface EPMPolicyManager {

    EPMPolicy createPolicy(
        in PropertyService::PSAccessAdaptor adaptor,
        in string propertySet,
        in string constraint) // policy-factory
    raises(EPMSyntaxException);

    attribute double EPM_EVALUATION_TOLERANCE;
    attribute float EPM_SIMPLEX_NEARVALUE;

  };
```

```
interface EPMPolicy {

  // logical transformations:
  boolean isLeanDNF();
  void leanify();
  void minimize();
  void normalize();
  boolean isConsistent();
  boolean isTautology();

  // generic functions:
  EPMPolicy unify(in EPMPolicy other)
  raises(EPMNotInDNFException);

  boolean stronger(in EPMPolicy other)
  raises(EPMNotInDNFException);

  boolean equivalent(in EPMPolicy other)
  raises(EPMNotInDNFException);

  boolean evaluate(in PropertyService::PSAccessAdaptor adaptor,
                   in string propertySet)
  raises(PropertyService::PSAccessException,
         PropertyService::PSDynamicTypeException);

  void activate(in PropertyService::PSAccessAdaptor adaptor,
                in string propertySet);

  EPMBasisSolutionArray optimum(in EPMPolicy other,
                                in stringArray maxNames,
                                in boolean minPriority)
  raises(EPMEmptySolutionSetException,
         EPMNotOptimizalableException,
         EPMUnboundedException);

  EPMBasisSolution minOptimum(in EPMPolicy other)
  raises(EPMEmptySolutionSetException, EPMUnboundedException,
         EPMNotOptimizalableException);

  EPMBasisSolution maxOptimum(in EPMPolicy other)
  raises(EPMEmptySolutionSetException, EPMUnboundedException,
         EPMNotOptimizalableException);

  stringArray getAllPropertyNames();

  string toString();

};
```

```
  interface EPMBasisSolution {

    stringArray getPropertyNames();
    doubleArray getValues();
    boolean getMin();
    void setMin(in boolean isMinimum);
    string toString();
    // because there are no null-Values allowed to transport
    // over the wire:
    boolean isNull();

  };

};
```

Anhang B

Schnittstellen der Rule–sensitiven Middleware

B.1 DomainService.idl

```
module DomainService {

 typedef sequence <string> stringArray;

 /*
  * Domain-Service for Rule-Sensitive Objects:
  */
 interface Domain {
  exception AlreadyExists {};
  /*
   * returns the complete number of rules which are included in the
   * members registered in this domain and all sub-domains.
   * If you call this method on the root-domain you get all rules.
   */
  long getNumOfRules();
  /*
   * The name of this domain.
   */
  string getName();
  /*
   * Indicates, if this domain is the root domain.
   * A root domain could not be sub-domain of other domains.
   */
  boolean isRoot();
  /*
   * Registers an object per name/address association.
  */
  void register(in string name,in string address)
    raises(AlreadyExists);
  /*
```

```
   * Deregisters an object per name.
   */
  void deregister(in string name);
  /*
   * No registering is allowing while there are rsdii's which have
   * not called rsdiiEnd(in long rsdiiId) yet.
   */
  void rsdiiStart(in double rsdiiId);
  /*
   * Indicates the end of an rsdii call
   */
  void rsdiiEnd(in double rsdiiId);
  /*
   * Adds a domain to this domain. This makes the added domain a
   * sub-domain of this one.
   */
  void add(in Domain domain) raises(AlreadyExists);
  /*
   * Removes a domain from this domain.
   */
  void remove(in Domain domain);
  /*
   * Returns the number of subdomains which are contained in this
   * domain. (not sub-sub..-domains)
   */
  long getSubDomainSize();
  /*
   * Returns the number of objects which are not domains of this
   * domain. (only objects which are directly part of this domain)
   */
  long getLeafSize();
  /*
   * Resolve the address:
   */
  string resolve(in string name);
  /*
   * returns a string-array of all member in this domain.
   */
  stringArray getLeafs();
 };
};
```

B.2 EventFactory.idl

```
#include "CosTransactions.idl"

module EventFactory {

 /* the interface RuleEvent should be abstract! */
 interface RuleEvent {
  string getContextId();
  double getRsdiiId();
  CosTransactions::Coordinator getCoordinator();
  void setCoordinator(in CosTransactions::Coordinator coordinator);
 };

 /* the interface ReplyRuleEvent should be abstract! */
 interface ReplyRuleEvent : RuleEvent {
 /* constants for the content:
  * getContent() returns:
  * - RULEEXCEPTION followed by a detailed description
  *   of the error if available.
  * - is empty ("") if everything is allright
  * - or contains a Policy-String, if it is a reply of a rule in
  * callback mode.
  */
  const string RULEEXCEPTION = "RuleException";
  string getContent();
 };

 interface InternalReplyRuleEvent : ReplyRuleEvent {
 /* This event is for communicating with
  * the rsdii. Only rules of calling Container
  * have to send this.
  */
 };

 interface ExternalReplyRuleEvent : ReplyRuleEvent {
 /* This event is for the communication of
  * the remote Containers Rules with the Container rules
  * they were calling from.
  * The TargetRuleId string contains the Rule-SetName and the
  * RuleName. "/RuleSet/.../RuleName"
  */
  string getTargetId();
  string getTargetRuleId();
 };
```

```
/* the interface MethodRuleEvent should be abstract! */
interface MethodRuleEvent : RuleEvent {
 string getSignature();
 boolean isBeforeExecution();
 string getTarget();
};


interface BeforeMethodInvocationRuleEvent : MethodRuleEvent {
/* indicates if the MethodRuleEvent was fired before
 * method invocation.
 */
};

interface AfterMethodInvocationRuleEvent : MethodRuleEvent {
/* indicates if the MethodRuleEvent was fired after
 * method invocation.
 */
};

interface FilterRuleEvent : MethodRuleEvent {
 string getUnifyParam();
};


interface EventFactory {
 InternalReplyRuleEvent constructInternalReplyRuleEvent(
   in string contextId,
   in string content,
   in double rsdiiId);

 ExternalReplyRuleEvent constructExternalReplyRuleEvent(
   in string contextId,
   in string content,
   in string targetId,
   in string targetRuleId,
   in double rsdiiId);

 MethodRuleEvent constructMethodRuleEvent(
   in string contextId,
   in string target,
   in string signature,
   in boolean beforeExecution,
   in double rsdiiId);
```

```
  FilterRuleEvent constructFilterRuleEvent(
    in string contextId,
    in string target,
    in string signature,
    in boolean beforeExecution,
    in string unifyParam,
    in double rsdiiId);
 };

};
```

B.3 Marketplace.idl

```
#include "DomainService.idl"

module Marketplace {

 typedef sequence<string>agentAddressArray;
 typedef sequence<string>agentAliasArray;

 interface Market : DomainService::Domain {
 /*
  * Look up all agents who sell the desired item.
  * The searching agent has to give his own id with.
  */
  agentAddressArray lookUp(in string item, in string whoLooksId);
  void enterRoom(in string activeAgentAlias,
                 in agentAliasArray companions);
  void leaveRoom(in string activeAgentAlias);
 };
};
```

B.4 DTD der Rule Base

```
<!ELEMENT RuleBase (RuleSet)*> <!ELEMENT RuleSet
(RuleSet|Property|Policy|
        Requirement|Action|StateTransition|Counter)*>
<!ATTLIST RuleSet name CDATA #REQUIRED>

<!ELEMENT Property EMPTY> <!ATTLIST Property
  name CDATA #REQUIRED
  type (ULong|UShort|UDouble|UFloat|String|Mark) #REQUIRED
  value CDATA #REQUIRED
>

<!ELEMENT Policy (Defaults,TriggerItem*)> <!ELEMENT Requirement
(Defaults,TriggerItem*)>

<!ELEMENT Action (Defaults,TriggerItem*)> <!ATTLIST Action
  signature CDATA #REQUIRED
  rulesensitive (TRUE|FALSE) #REQUIRED
>
<!ELEMENT StateTransition (Defaults,TriggerItem*)> <!ATTLIST
StateTransition
  precondition CDATA #REQUIRED
  postcondition CDATA #REQUIRED
>
<!ELEMENT Counter (Defaults,TriggerItem*)> <!ATTLIST Counter
  counter CDATA #REQUIRED
  increment (TRUE|FALSE) #REQUIRED
>

<!ELEMENT TriggerItem EMPTY> <!ATTLIST TriggerItem
  signature CDATA #REQUIRED
  target CDATA #REQUIRED
  contextId CDATA #REQUIRED
  everytimes (TRUE|FALSE) #REQUIRED
  beforeExecution (TRUE|FALSE)  #REQUIRED
>
<!ELEMENT Defaults EMPTY> <!ATTLIST Defaults
  name CDATA #REQUIRED
  condition CDATA #REQUIRED
  mode (Internal|Filter|Oneway|Callback) #REQUIRED
>
```

Anhang C

Beispiel für die Benutzung des *Plug-in*–Mechanismus

Die Benutzung der beteiligten Klassen des Plug-in–Mechanismus soll anhand des folgenden Beispiels verdeutlicht werden.

Als Beispiel für Quelle und Ziel von Kooperationen dienen zwei Objekte der Klasse *MyObject*. Die Methoden dieser Klasse verrichten keine wirklich zweckmässige Arbeit, sondern sollen nur dokumentieren, dass sie gerufen wurden.

Das Beispiel beinhaltet folgende Schritte:

1. Erzeugung von Quell- und Zielobjekt
2. Erzeugung der *Cooperation*–Objekte
3. Komposition der Objekte
4. Aufruf der Quellmethoden

Erzeugung von Quell- und Zielobjekt Die Erzeugung der Objekte wird durch Voyager geleistet. Dies ist notwendig, weil Voyager den *MessageListener* Dienst anbietet.

```
IMyObject  Source = (IMyObject) Voyager.construct("dynamics.agents.plugexample.MyObject");
IMyObject  Dest1 = (IMyObject)  Voyager.construct("dynamics.agents.plugexample.MyObject");
IMyObject  Dest2 = (IMyObject)  Voyager.construct("dynamics.agents.plugexample.MyObject");
IMyObject  Dest3 = (IMyObject)  Voyager.construct("dynamics.agents.plugexample.MyObject");

Source.Setid("source"); // Namen setzen
Dest1.Setid("dest1");
Dest2.Setid("dest2");
Dest3.Setid("dest3");
```

Erzeugung der *Cooperation*–Objekte Es werden drei *Cooperation*–Objekte erzeugt. Zunächst werden die Methodensignaturen der Quell- und Zielmethoden spezifiziert, dann wird festgelegt, ob die Zielmethode parallel oder seriell zur Quellmethode ausgeführt werden soll. Zwei der *Cooperation*–Objekte brauchen

sich um die Weiterleitung von Parametern nicht zu kümmern. Für das dritte *Cooperation*–Objekt wird spezifiziert, dass eine Weiterleitung des Inputparameters der Methode *messageB()* an dieselbe Methode des Zielobjektes stattfinden soll.

```
Cooperation Coop1 = new Cooperation(); // Coop1 erzeugen

Coop1.setSrcSignature("messageA()V"); // Methodensignaturen spezifizieren
Coop1.setDestSignature("messageA()V");

Coop1.setExecuteInParallel(false); // Zielmethode seriell ausf"uhren

Cooperation Coop2 = new Cooperation(); // Coop2 erzeugen

Coop2.setSrcSignature("messageA()V"); // Methodensignaturen spezifizieren
Coop2.setDestSignature("messageA()V");

Coop2.setExecuteInParallel(true); // Zielmethode parallel ausf"uhren

Cooperation Coop3 = new Cooperation(); // Coop3 erzeugen

Coop3.setSrcSignature("messageB(I)V"); // Methodensignaturen spezifizieren
Coop3.setDestSignature("messageB(I)V");

Coop3.setExecuteInParallel(false); // Zielmethode seriell ausf"uhren

Object[] params = new Object[1]; // Parameterliste erzeugen

params[0] = new Integer(1); // Parametertyp spezifizieren
Coop3.setParameters(params); // der Wert ist ohne Belang
```

Komposition der Objekte Den erzeugten Objekten werden zunächst durch Übergabe der Objektreferenz an die statische *of()*–Methode der Klasse *Pluggability* die Eigenschaft der dynamischen Komposition zugewiesen. Der Rückgabewert ist vom Typ *IPluggable*. Entsprechend der drei erzeugten *Cooperation*–Objekte werden drei Kompositionen erzeugt.

```
IPluggable SourcePlug = Pluggability.of(Source); \\ Kompositionsfaehigkeit erzeugen

IPluggable DestPlug1 = Pluggability.of(Dest1);

IPluggable DestPlug2 = Pluggability.of(Dest2);

IPluggable DestPlug3 = Pluggability.of(Dest3);


SourcePlug.plug(DestPlug1,Coop1); \\ Verbinden von SourcePlug mit DestPlug1
                                  \\ unter Verwendung von Coop1
SourcePlug.plug(DestPlug2,Coop2);

SourcePlug.plug(DestPlug3,Coop3);
```

Aufruf der Quellmethoden Die in den *Cooperation*–Objekten spezifizierten Methoden werden am Quellobjekt aufgerufen.

```
System.out.println("Sending message ......");

 Source.messageA();  // Aufruf des Quellobjektes
 Source.messageB(5);

System.out.println("message sent ......");
```

Programmausgabe Die Ausgabe des Beispielprogramms ermöglicht eine Verfolgung der Ereignisse. Die Ausgabe zeigt, dass zunächst das Objekt gerufen wird, das eine parallele Ausführung spezifiziert hat. Das bedeutet, dass bei Beginn der Ausführung der Quellmethode das Ereignis asynchron an die Zielmethode gemeldet wird.

```
Running dynamics.agents.plugexample.plugexample ...
voyager(tm) 2.0 beta 2, copyright objectspace 1997, 1998
address = 192.168.1.1:1025
constructed

Sending message ......
message event: messageA()V        <-- Source.messageA();

message event starting messageA()V  <-- Message Event zeigt Ausfuehrungsbeginn
selector: 1 parallel Targets found  <-- parallele Ausfuehrung fuer dest2
Current Target : dynamics.agents.plugexample.MyObject@80ccc33  <-- dest2
Current Coop : messageA()V messageA()V    <-- Coop2
calling onewway : dynamics.agents.plugexample.MyObject@80ccc33 messageA()V <-- asynchron


Message A received. ID:source <-- Source.messageA();

Message A received. ID:dest2 <-- Ergebnis von Coop2

message event completed messageA()V <-- Message Event zeigt Ausfuehrungsende
selector: 1 serial Targets found    <-- serielle Ausfuehrung fuer dest1
Current Target : dynamics.agents.plugexample.MyObject@80ccc59 <-- dest1
Current Coop : messageA()V messageA()V  <-- Coop1
calling sync : dynamics.agents.plugexample.MyObject@80ccc59 messageA()V <-- synchron

Message A received. ID:dest1 <-- Ergebnis von Coop1

message event: messageB(I)V <-- Source.messageB(5);

message event starting messageB(I)V  <-- Message Event zeigt Ausfuehrungsbeginn

Message B received. ID: source Param: 5  <-- Source.messageB(5);

message event completed messageB(I)V <-- Message Event zeigt Ausfuehrungsende
selector: 1 serial Targets found     <-- serielle Ausfuehrung fuer dest3
Current Target : dynamics.agents.plugexample.MyObject@80cccd4 <-- dest3
Current Coop : messageB(I)V messageB(I)V  <-- Coop3
calling sync : dynamics.agents.plugexample.MyObject@80cccd4 messageB(I)V <-- synchron

Message B received. ID: dest3 Param: 5 <-- Ergebnis von Coop3

message sent ......
```

Die Programmausgabe zeigt deutlich, dass das Objekt mit der ID 'dest2' die Nachricht über den Aufruf von *messageA()* deutlich vor dem Objekt mit der

ID 'dest1' erhält. Der Grund dafür ist die parallele Ausführung von Objekt 'dest2' zur Quellmethode. Desweiteren ist erkennbar, dass das Objekt 'dest3' den Parameter der Methode *messageB()* korrekt übermittelt bekommt.

Die Klasse *MyObject* Um das Beispiel zu vervollständigen, sei noch die Implementation der Klasse *MyObject* angefügt.

```
public class MyObject implements IMyObject, Serializable
{
          String id;

  public void messageA()
  {
      System.out.println( "Message A received. ID:" + id );
      try
              {
                 Thread.currentThread().sleep(5000);
              }
              catch(Exception e)
              {
                 e.printStackTrace();
              }

  }
  public void messageB(int Param)
  {
      System.out.println( "Message B received. ID: " + id + " Param: " + Param );

  }
  public void messageC(Integer Param,String message)
  {
      System.out.println( "Message C received. ID: " + id + " Param: " + Param
  + "message: " + message);

  }
  public void messageD(String message)
  {
      System.out.println( "Message D received. ID: " + id +  " message: " + message );

  }

  public void Setid(String newId)
  {
      id = newId;
  }
}
```

Anhang D

OOPAMELA–Syntax und –Beispiele

D.1 OOPAMELA–Syntax in EBNF

Non-terminals
Terminals
non-formal description of terminals

```
protocol        := description [includes]* header

description     := /** free description text **/

includes        :=  #include "filename";

header          :=  protocol "protocolname" { mainpart }

mainpart        := interfaces process metaprotocol

interfaces      :=  interfaces { [interfacedecl]* }

process         :=  process { places transitions }

metaprotocol    :=  metaprotocol { metaprotocolentries }

interfacedecl   :=  interface "interfacename" { idl-definition }

places          :=  places: [placedecl]*

transitions     := [transitiondecl]*

placedecl       := type "placename" [ relates to
                     "transitionname"];
```

```
transitiondecl  := trans "transitionname" [uses "interfacename"
                   [[anynumber]][ alias "agentname"] |
                    relates to "agentname"] { transitionbody }

transitionbody  := getdecl [getdecl]*| putdecl [putdecl]*
                   [actions] [timeout]

getdecl         := get [variable[[[anynumber]]] ,variable[
                   [[anynumber]]] *]
                    from "placename" [relationcond]
                   [ if comparedecl];

putdecl         := put [variable[[[anynumber]]]
                   ,variable[[[anynumber]]] *]
                    to "placename" [relationcond];

type            := int | double | float | string |
                   interfacename | anonymous | timeout

actions         := actions: actionlist

comparedecl     := expression comparator expression |
                   (comparedecl && comparedecl)     |
                   (comparedecl || comparedecl)

accesspath      := variable[->attribute|method]+

comparator      := < | <= | == | >= | > | !=

value           := true | false |
                   anynumber | anyconstant | accesspath

actionlist      := variable = expression;

expression      := value|variable|agentcall[CL1] = operator
                   value|variable|agentcall

operator        := *|+|/|%|||&

agentcall       := [system->]method

method          := methodname(parameterlist)

parameterlist   := parameter , parameter*

parameter       := value

relationcond    := [ [ x | booleanexpr ] ]
```

```
booleanexpr     := x comparator relconstant       |
                   (booleanexpr) && (booleanexpr) |
                   (booleanexpr) || (booleanexpr)

relconstant     := { anynumber |  [ this->] instanceNo |
                   [ this->] minInstances | [ this->] maxInstances }
                   [operator relconstant] | (relconstant)

metaprotocolentries := { MinimumParties: anyint; }  |
                       { MaximumParties: anyint; }  |
                       { GlobalTimeout: anyint; }   |
                       { DefaultTimeout: anyint; }  |
                       { constantNoOfParties; }     |
                       { anonymous; }

metarolespec    := rolename { metaroleentry {, metaroleentry }* }

metatransspec   := transname { metatransentry {, metatransentry }* }

metaroleentry   := { MinimumParties: anyint; }          |
                   { MaximumParties: anyint; }          |
                   { knows: rolename , rolename* |  all }; |
                   { newPartiesVoting [rolevoting* | ; ] } |
                   { constantNoOfParties; }

rolevoting      :=  role: { rolename |  all }  percent: percent;

metatransentry  := { Timeout: anyint; }
```

D.2 Ein Verhandlungsprotokoll in OOPAMELA

Dieses Verhandlungsprotokoll beschreibt eine einfache Auktion mit 4 Teilnehmern. Mit Hilfe dieses Protokolls wurden die ersten Agenten an der Protokoll–Engine getestet.

```
protocol "auktionfinal" {

       // Interface section:
  interfaces {
     interface "Anbieter" {
        "Contract firstOffer(void);"
     }
     interface "Bieter1" {
        "Contract make Offer(Contract);"
     }
     interface "Bieter2" {
        "Contract make Offer(Contract);"
     }
     interface "Auktionator" {
        "bool check(Contract, Contract);"
     }
  }

  places {
     Init "Auktion";
     Contract "Start2";
     Contract "Angebot";
     Contract "Angebot3";
     Contract "Angebot4";
     Contract "Place5";
     Contract "Place6";
     Anonymous"Place7";
     bool     "Place8";
     bool     "Place9";
     bool     "Place10";
  }
  trans "Init" {
      get Anonymous from "Auktion";
      put "Con1" to "Start2";
      action "Con1 = new Contract()"
  }
  trans "Start" uses "Anbieter" {
      get "Con1" from "Start2";
      put "Con1" to "Angebot";
      action "Con1=firstOffer()"
  }
  trans "verteilen" {
      get "Con1" from "Angebot";
      put "Con1" to "Angebot3";
      action "Con2=Con1.Clone()"
  }
  trans "holeAngebot1" uses "Bieter1" {
      get "Con1" from "Angebot3";
      put "Con1" to "Place5";
      action "Con1=makeOffer(Con1)"
  }
  trans "holeAbgebot" uses "Bieter2" {
      get "Con2" from "Angebot4";
      put "Con2" to "Place6";
      action "Con2=makeOffer(Con2)"
  }
```

```
   trans "Auktion" uses "Auktionator" {
      get "Con1" from "Place5";
      get "Con2" from "Place6";
      put "bOK" to "Place8";
      action "bOK=check(Con1,Con2)"
   }
   trans "Trans7" {
      get "bOK" from "Place8"
      put "bOK" to "Place9";
      put "bOK2" to "Place10";
      action "bOK2 = new Boolean(bOK.booleanValue())"
   }
   trans "Trans6" {
      get "bOK2" from "Place10" if "bOK2.booleanValue() == false";
      put Anonymous to "Auktion";
   }
   trans "Ende" {
      test "bOK" from "Place9" if "bOK.booleanValue() ==true";
      action ":terminate()"
   }
}

metaprotocol {

        MinimumParties: 4;
        MaximumParties: 4;
        NegotiationTimeout: 20;
        DefaultTimeout: 20;

        Role "Anbieter" {
                MinimumParties: 1;
                MaximumParties: -1;
        }
}
```

D.3 Das Standardreferenznetz in OOPAMELA

Dieses Netz wird standardmässig für die Kommunikation mit den Verhandlungsteilnehmern benutzt. Es steht dem Entwickler jedoch frei, dieses Netz zu verändern, oder eigene Referenznetzte für die Kommunikation zu benutzen.

```
protocol "callNet" {

  // Interface section:
  interfaces {
     interface "Protokollmodul {
       "void callAgent(Contract);"
     }
  }

  places {
     Anonymous "TimerInit";
     Anonymous "TimerEnde";
     Anonymous "callRunning";
     String    "CallFunc";
     String    "CallFunc2";
     Contract  "CloneInit";
     Contract  "CloneNew";
     Contract  "CloneOld";
     Contract  "CallReady";
     Contract  "update1";
     Contract  "update2";
     Role      "Agent";
     Role      "Tester";
     bool      "Control";
     Contract  "TestCon1";
     Contract  "TestCon2";
     Contract  "TestFailed";
     Contract  "CallEnde";
  }
  trans "Start"  {
        put "Con1"    to "CloneInit";
       put Anonymous to "TimerInit";
       put "strFunc" to "CallFunc";
       put "myRole"  to "Agent";
       put "Tester"  to "Tester";
       action ":new(strFunc,Con1,myRole,Tester)"
  }
  trans "dummy" {
       get "strFunc" from "CallFunc";
       put "strFunc" to "CallFunc2";
  }
  trans "Clone" {
       get "Con1" from "CloneInit";
       put "Con1" to "CloneNew";
       put "Con2" to "CloneOld";
       action "Con2 = Con1.Clone()"
  }
  trans "callRole" {
     test "strFunc" from "CallFunc2";
     test "myRole" from "Agent";
     get "Con1" from "CloneNew";
               put "Con1" to "update1";
     put "ConF" to "update2";
     action "ConF=(Contract)
                  myRole.ProtModul.callAgentWith(strFunc, Con1)"
  }
```

```
    trans "update" {
       get "Con1" from "update1";
       get "ConF" from "update2";
       put "Con1" to "CallReady";
       action "Con1.update(ConF)"
    }
    trans "Test" {
       test "strFunc" from "CallFunc2";
       test "Tester" from "Tester";
       get "Con2" from "CloneOld";
       get "Con1" from "CallReady";
                 put "Con1" to "TestCon1";
       put "Con2" to "TestCon2";
       put "bOK" to "Control";
       action "bOK = new Boolean(true)"
    }
    trans "Check1" {
       get "Con1" from "TestCon1";
       test "bOK" from "Control";
       put "Con1" to "CallEnde" if "bOK.booleanValue() == true";
    }
    trans "Check2" {
       get "Con2" from "TestCon2";
       test "bOK" from "Control";
       put "Con2" to "TestFailed" if "bOK.booleanValue() == false";
    }
    trans "CloneFailed"{
       get "Con2" from "TestFailed";
       get "Con1" from "TestCon1";
       get "bOK" from "Control";
       put "Con1" to "CloneNew";
       put "Con2" to "CloneOld";
    }
    trans "Timer"{
       get Anonymous from "TimerInit";
       put Anonymous to "TimerEnde";
       action "EngineTimer.timeout(60)"
    }
    trans "DefaultCon"{
       get Anonymous from "TimerEnde";
       put "Con1" to "CallEnde";
       action "Con1 = new Contract()"
    }
    trans "ende"{
       get "Con1" from "CallEnde";
       action ":ende(Con1)"
    }
}
```

Anhang E

Ergebnisse aus den Experimenten mit dem GA–Framework

E.1 Bilaterale Verhandlungsszenarien

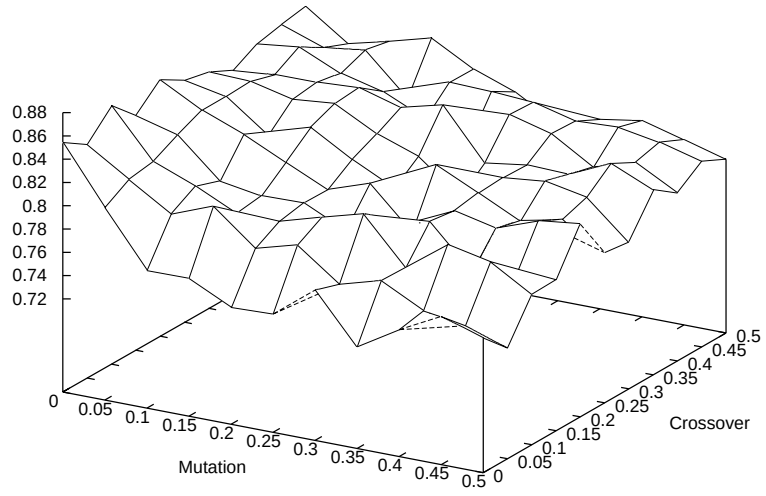


Abbildung E.1: Fitnesswerte für das "No Conflict" Szenario

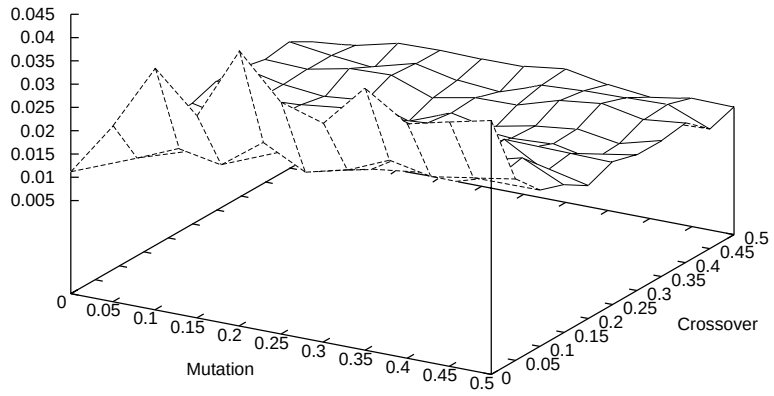


Abbildung E.2: Varianz für die Fitnesswerte im "No Conflict" Szenario

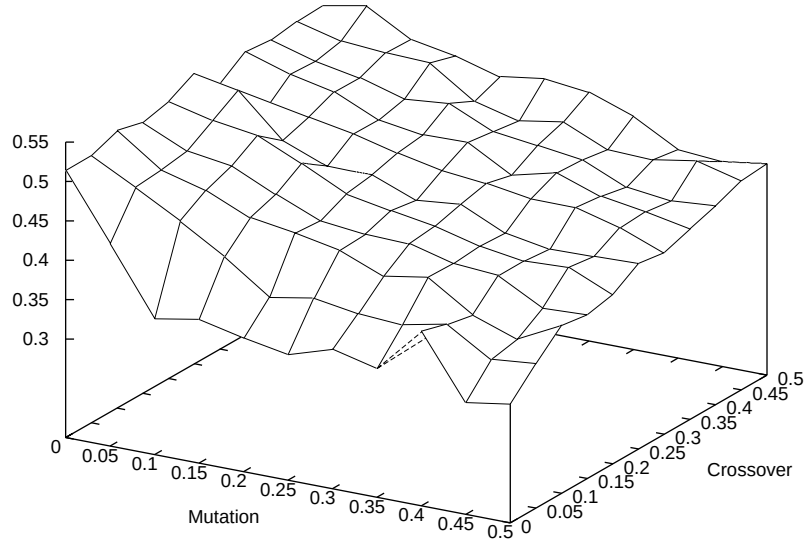


Abbildung E.3: Fitnesswerte für den ersten Spieler im "Pure Distributive Bargaining"–Szenario

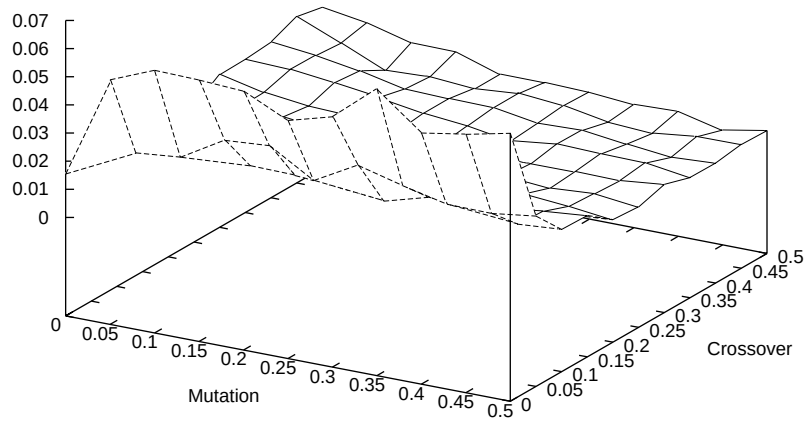


Abbildung E.4: Varianz für die Fitnesswerte des ersten Spielers im "Pure Distributive Bargaining"–Szenario

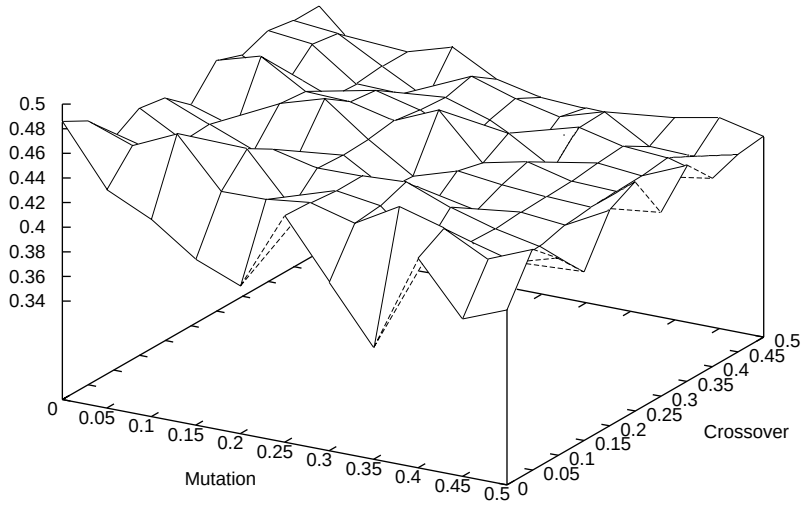


Abbildung E.5: Fitnesswerte für den zweiten Spieler im "Pure Distributive Bargaining"–Szenario

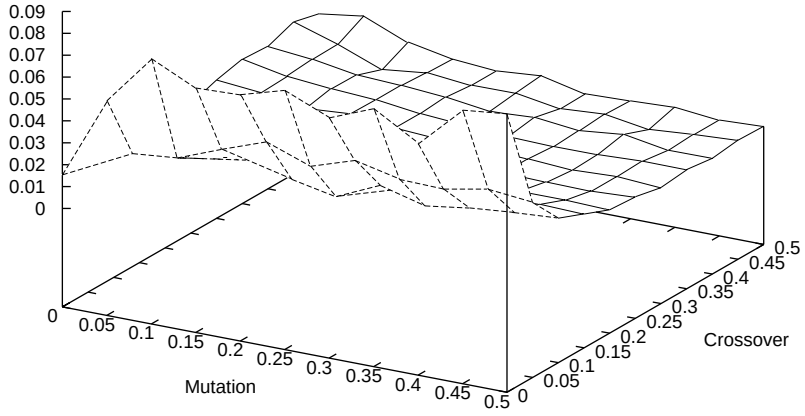


Abbildung E.6: Varianz für die Fitnesswerte des zweiten Spielers im "Pure Distributive Bargaining"–Szenario

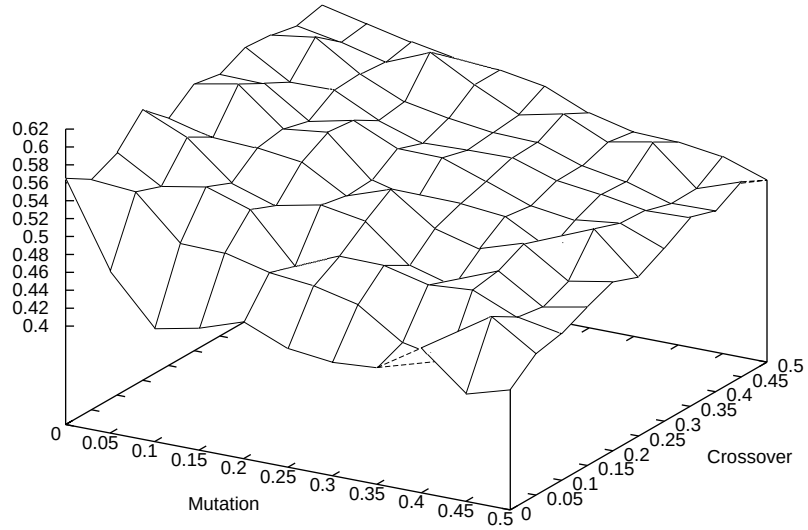


Abbildung E.7: Fitnesswerte für den ersten Spieler im "Simple Integrative Bargaining"–Szenario

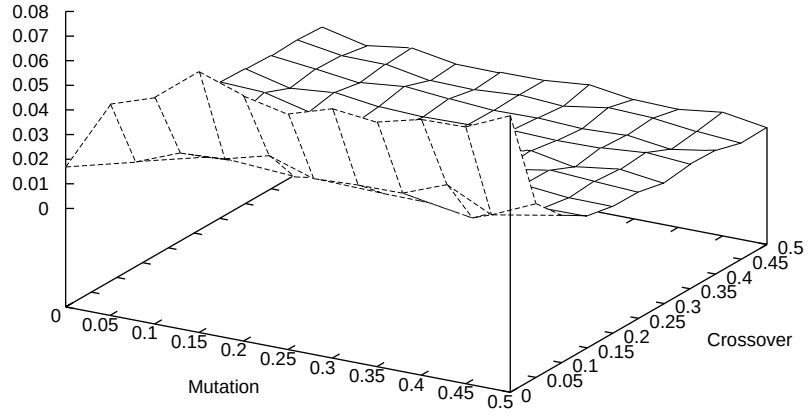


Abbildung E.8: Varianz für die Fitnesswerte des ersten Spielers im "Simple Integrative Bargaining"–Szenario

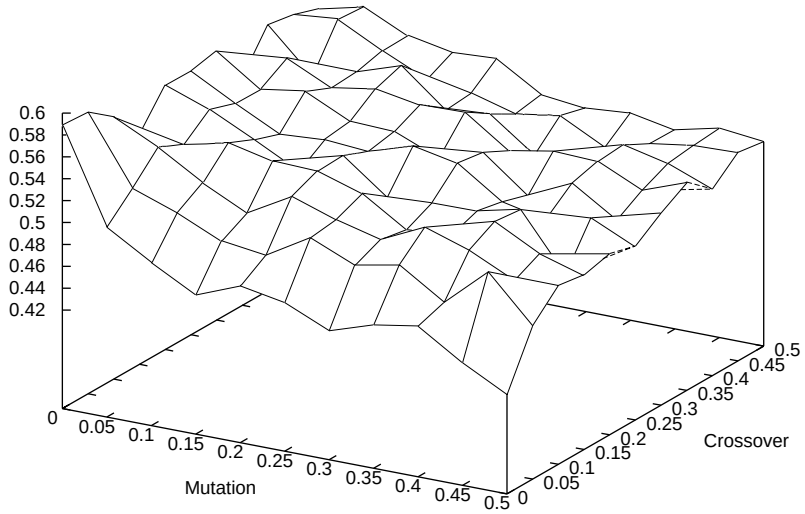


Abbildung E.9: Fitnesswerte für den zweiten Spieler im "Simple Integrative Bargaining"–Szenario

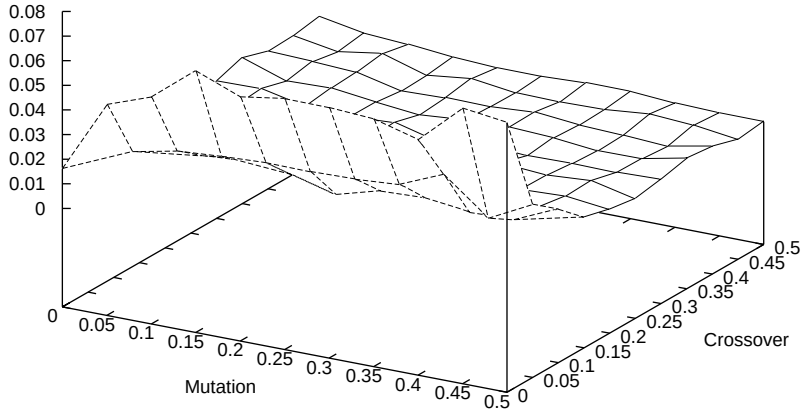


Abbildung E.10: Varianz für die Fitnesswerte des zweiten Spielers im "Simple Integrative Bargaining"–Szenario

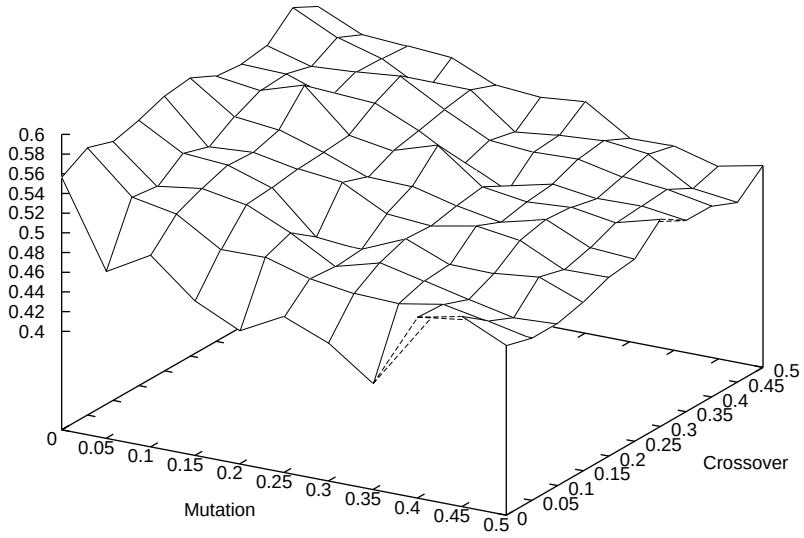


Abbildung E.11: Fitnesswerte für den ersten Spieler im "Divorce"–Szenario

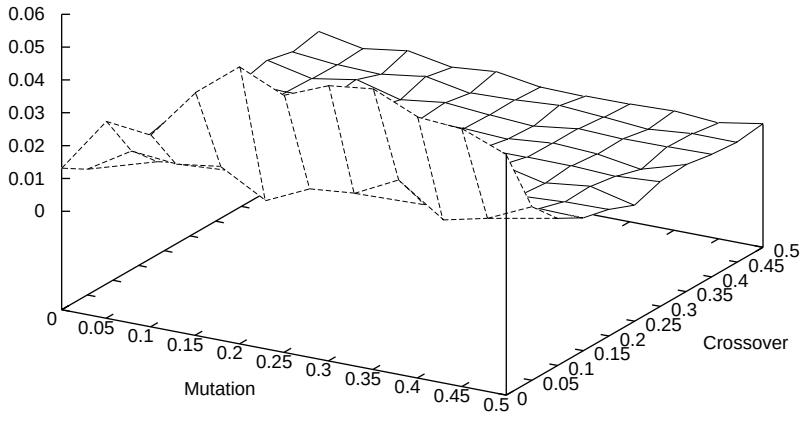


Abbildung E.12: Varianz für die Fitnesswerte des ersten Spielers im "Divorce"–Szenario

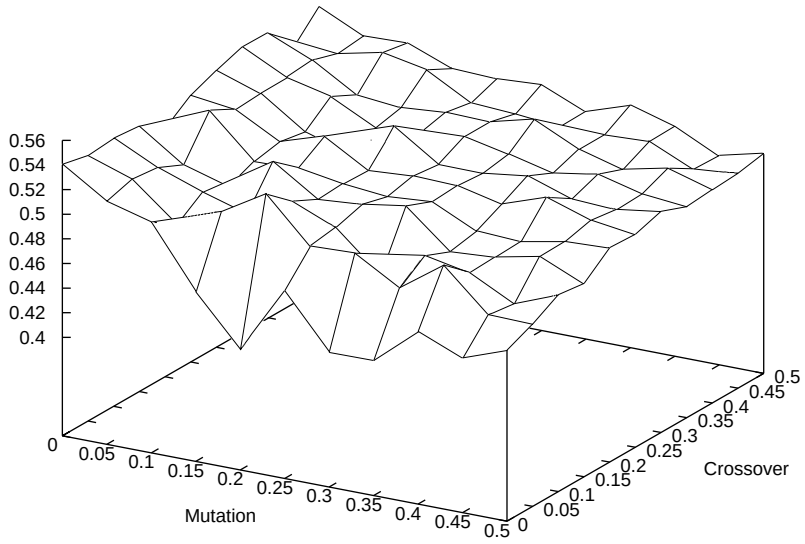


Abbildung E.13: Fitnesswerte für den zweiten Spieler im "Divorce"–Szenario

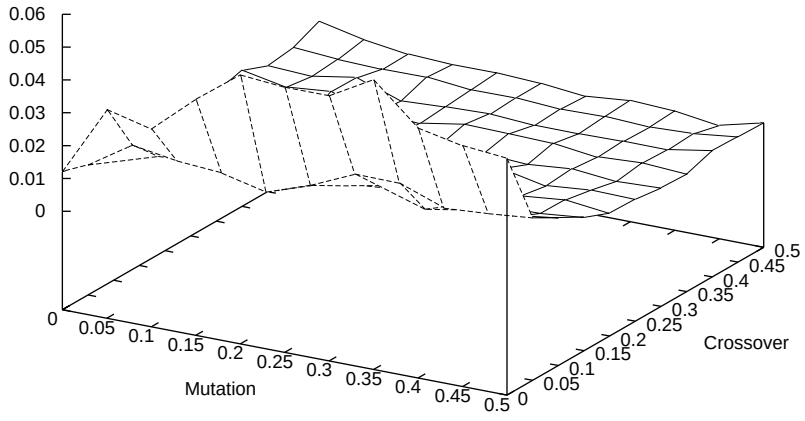


Abbildung E.14: Varianz für die Fitnesswerte des zweiten Spielers im "Divorce"–Szenario

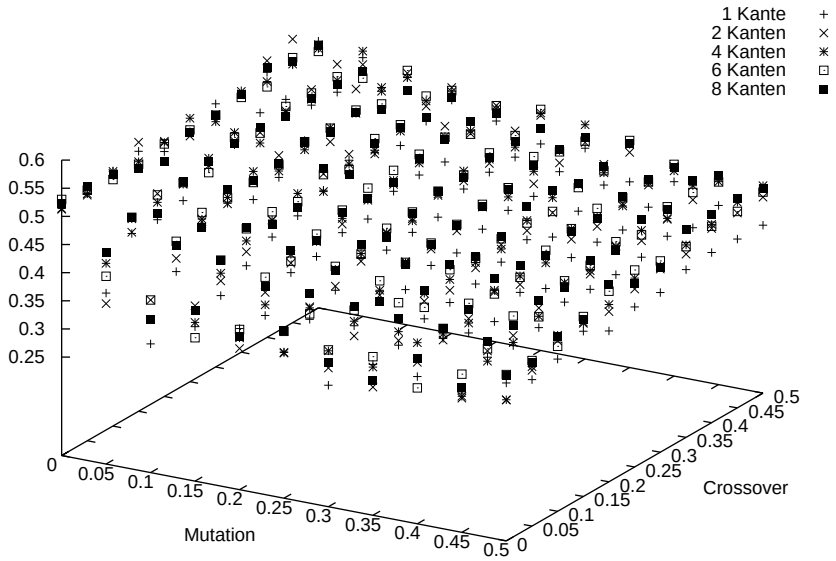


Abbildung E.15: Fitnesswerte für den ersten Spieler im "Pure Distributive Bargaining"–Szenario mit unterschiedlicher Anzahl von Kanten

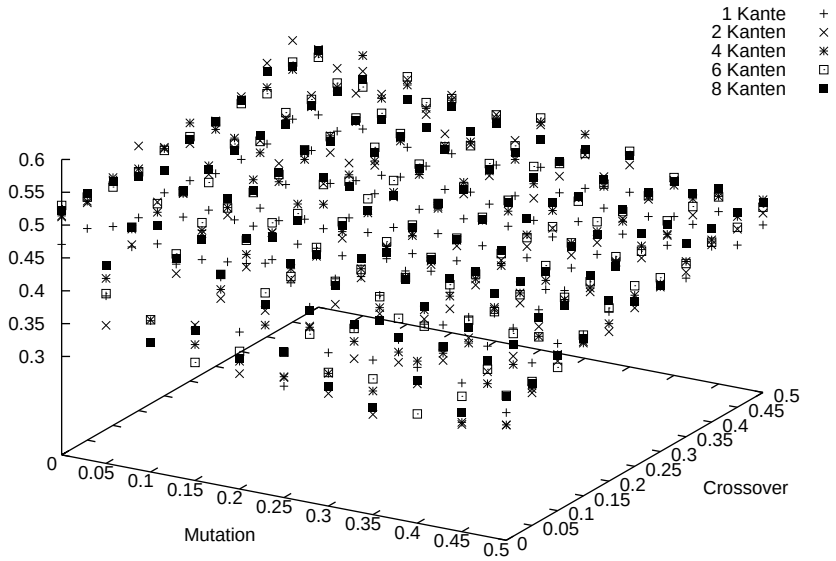


Abbildung E.16: Fitnesswerte für den zweiten Spieler im "Pure Distributive Bargaining"–Szenario mit unterschiedlicher Anzahl von Kanten

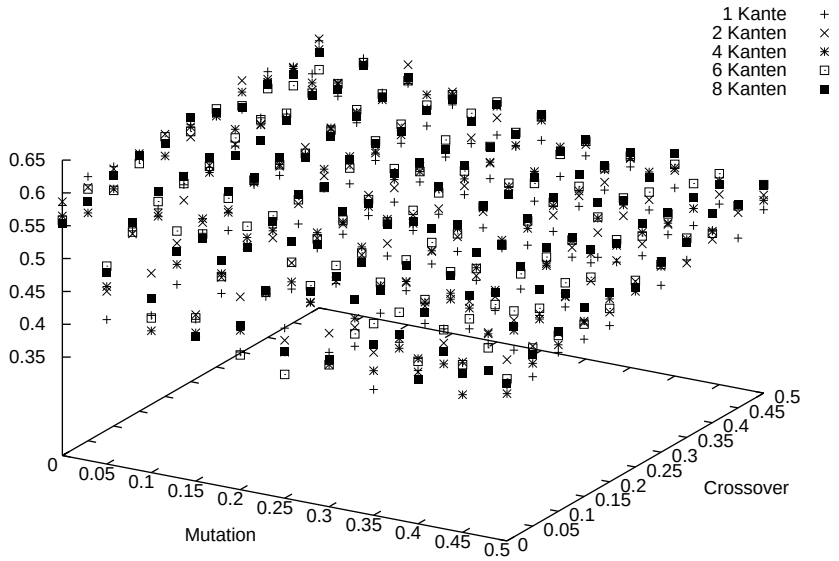


Abbildung E.17: Fitnesswerte für den ersten Spieler im "Simple Integrative Bargaining"–Szenario mit unterschiedlicher Anzahl von Kanten

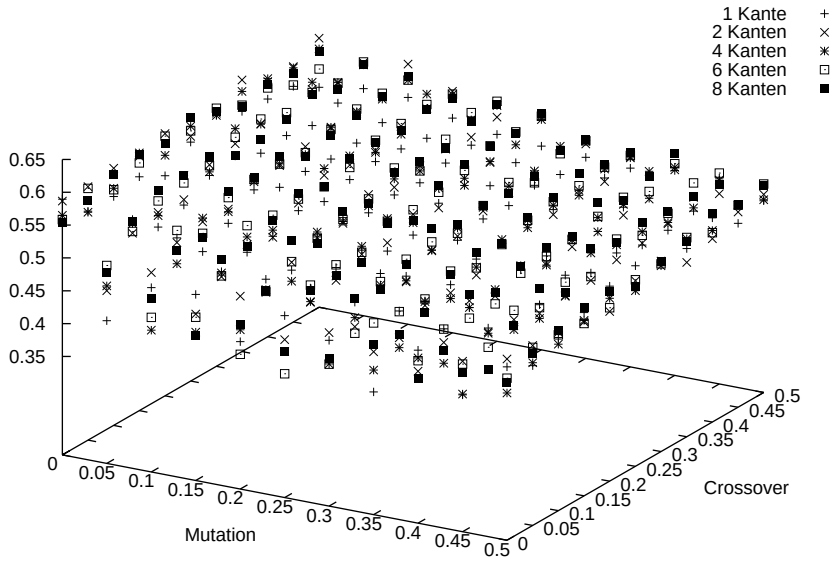


Abbildung E.18: Fitnesswerte für den zweiten Spieler im "Simple Integrative Bargaining"–Szenario mit unterschiedlicher Anzahl von Kanten

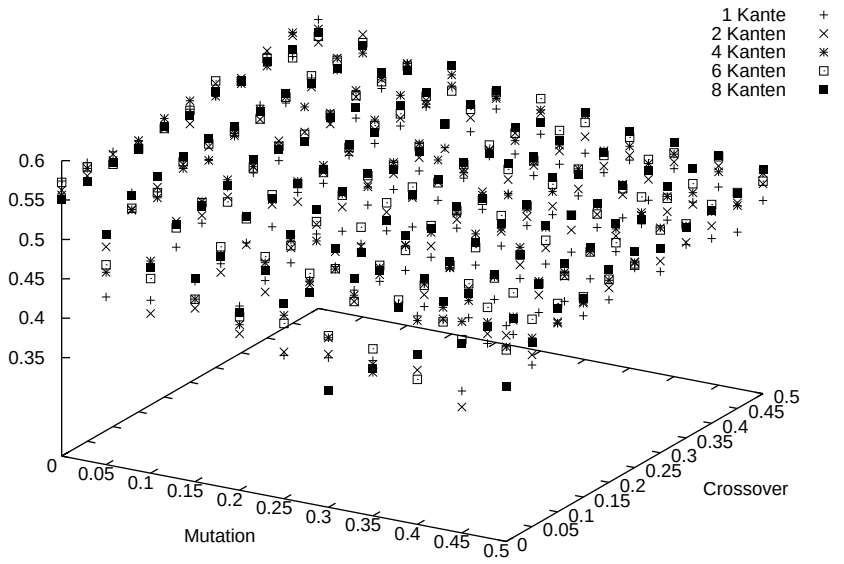


Abbildung E.19: Fitnesswerte für den ersten Spieler im "Divorce"–Szenario mit unterschiedlicher Anzahl von Kanten

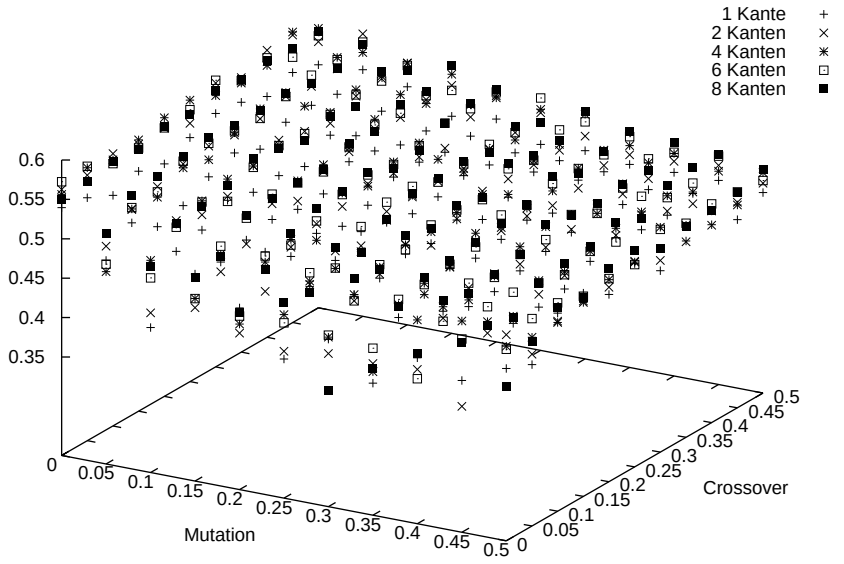


Abbildung E.20: Fitnesswerte für den zweiten Spieler im "Divorce"–Szenario mit unterschiedlicher Anzahl von Kanten

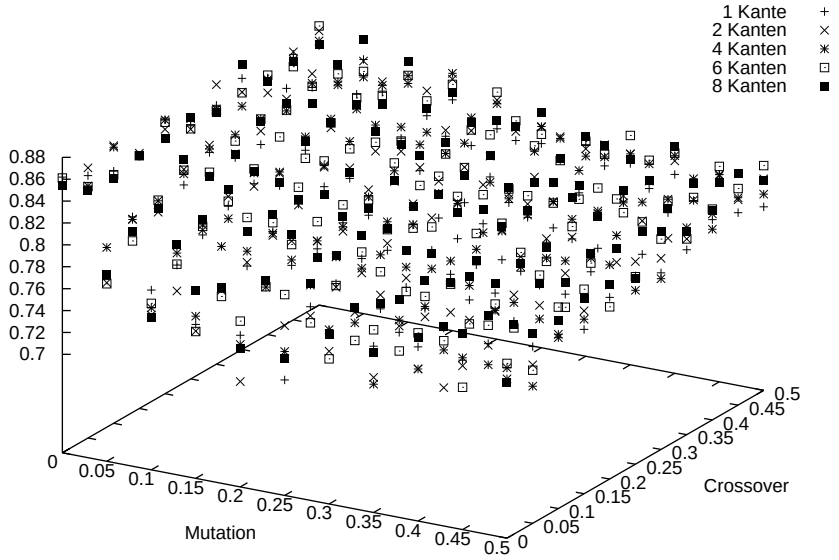


Abbildung E.21: Fitnesswerte für das "No Conflict"–Szenario mit unterschiedlicher Anzahl von Kanten

E.2 Auktionsszenarien

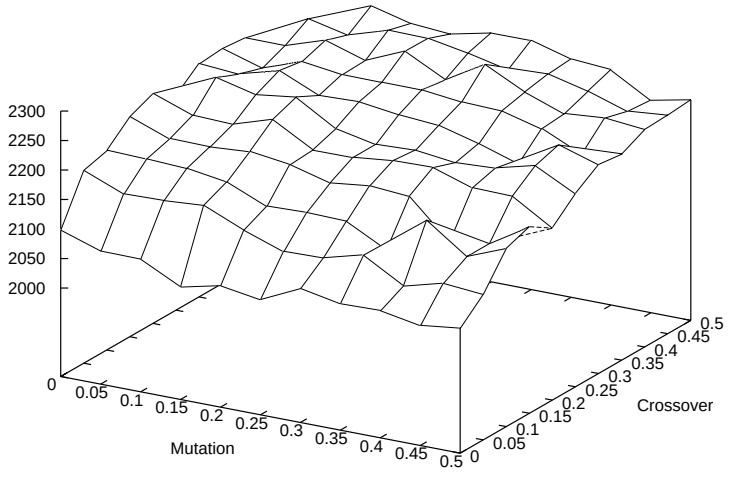


Abbildung E.22: Preis bei Bietern mit genetischem Algorithmus im "Zwei genetisch optimierte Strategien"–Szenario

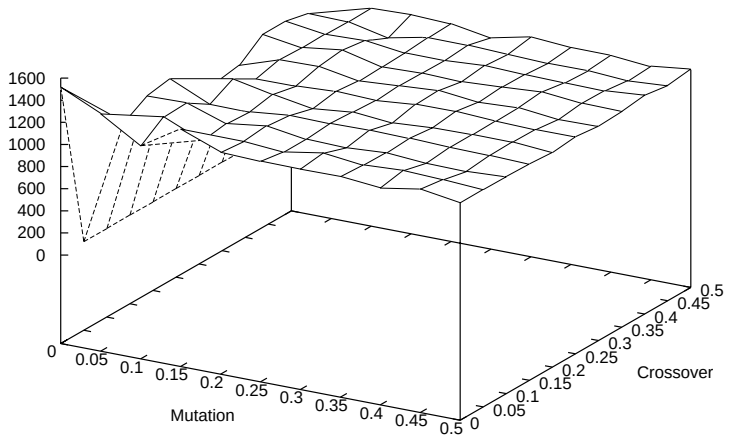


Abbildung E.23: Preis bei Bietern mit fester Strategie im "Zwei genetisch optimierte Strategien"–Szenario

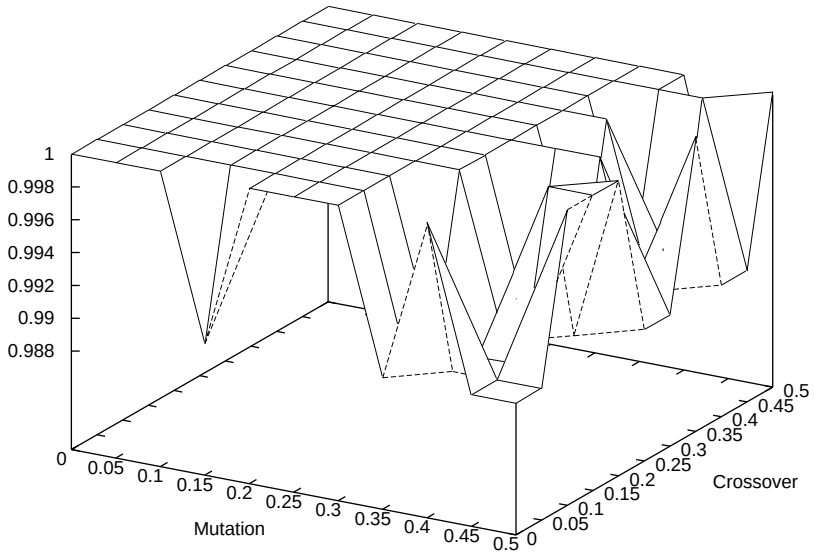


Abbildung E.24: Anteil der erfolgreichen Auktionen für genetisch optimierte Strategien im "Zwei genetisch optimierte Strategien"–Szenario

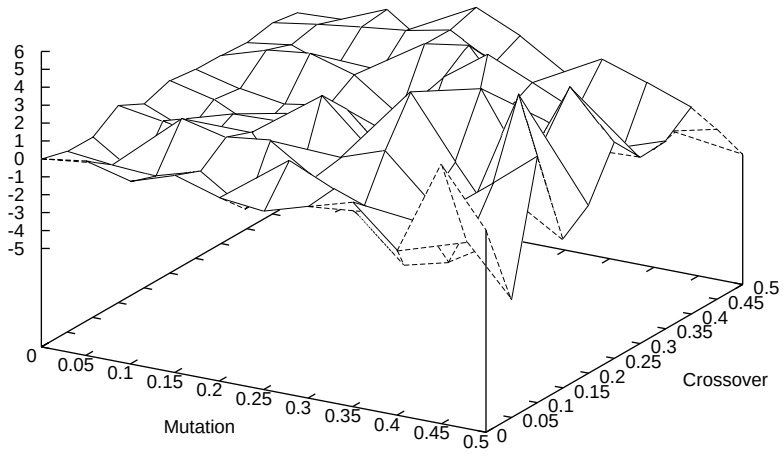


Abbildung E.25: Preisunterschied für genetisch optimierte und fixe Strategien als Anbieter im "Zwei genetisch optimierte Strategien"–Szenario

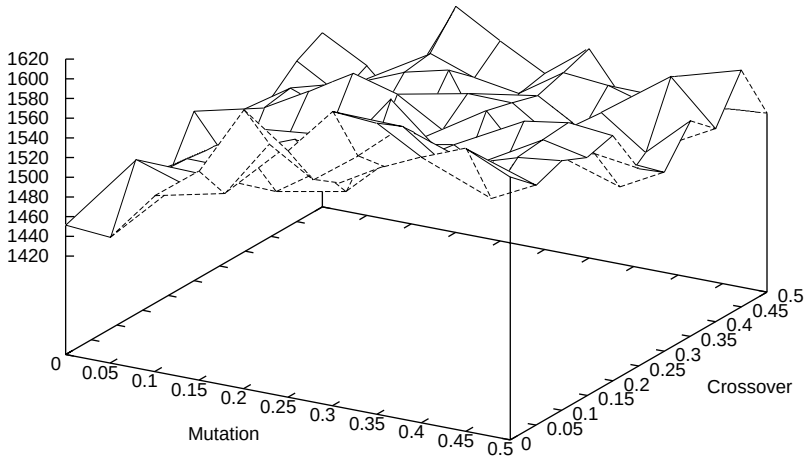


Abbildung E.26: Preis bei Bietern mit genetischem Algorithmus im "Eine genetisch optimierte Strategie"–Szenario

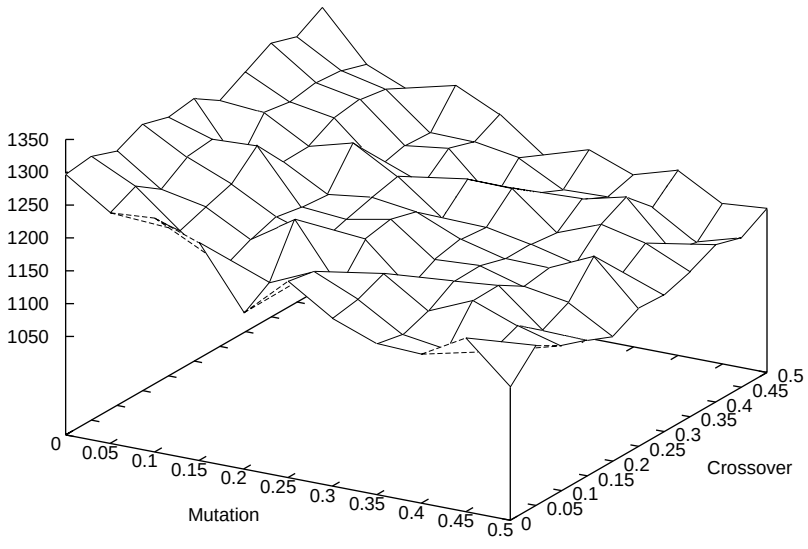


Abbildung E.27: Preis bei Bietern mit fester Strategie im "Eine genetisch optimierte Strategie"–Szenario

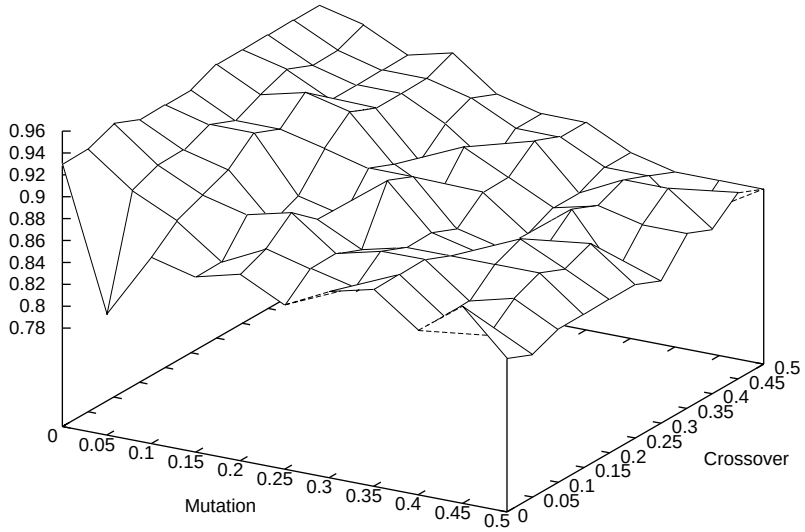


Abbildung E.28: Anteil der erfolgreichen Auktionen für genetisch optimierte Strategien im "Eine genetisch optimierte Strategie"–Szenario

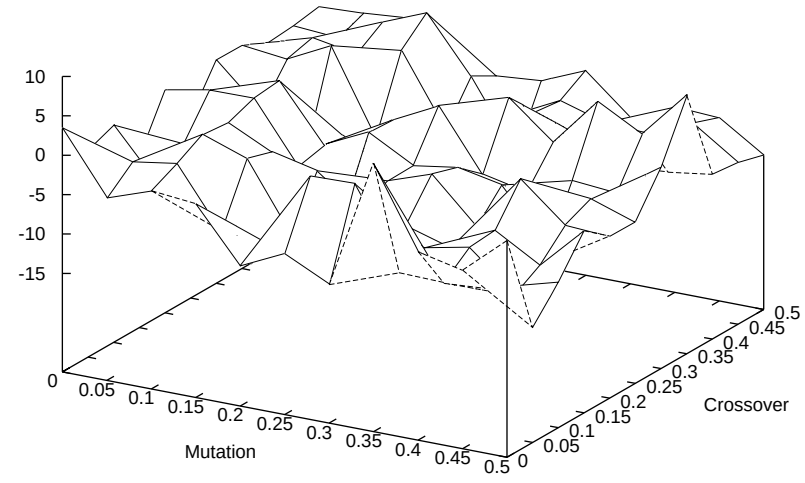


Abbildung E.29: Preisunterschied für genetisch optimierte und fixe Strategien als Anbieter im "Eine genetisch optimierte Strategie"–Szenario

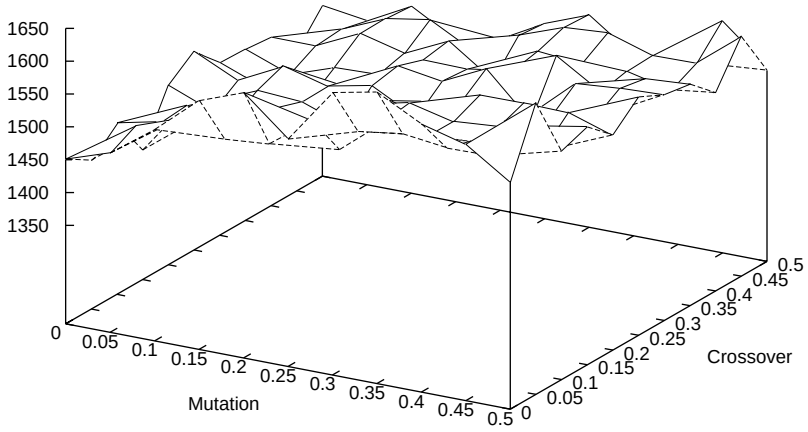


Abbildung E.30: Preis bei Bietern mit genetischem Algorithmus im "Keine Wahl"–Szenario

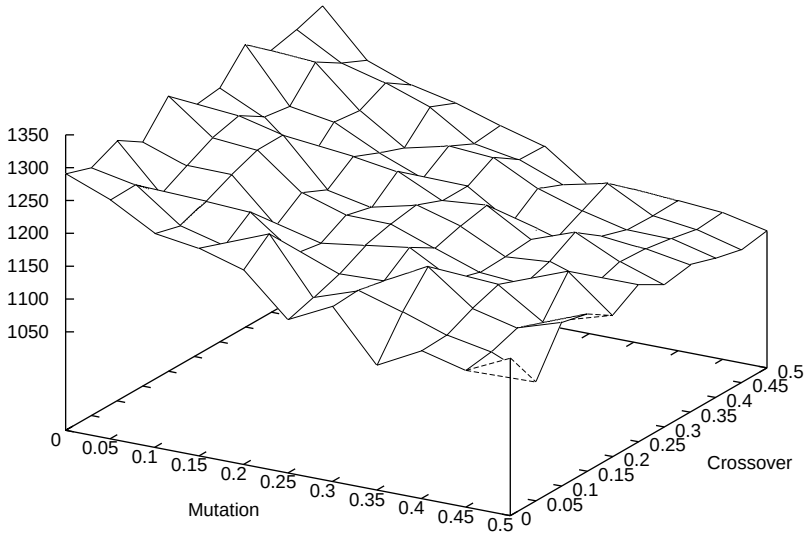


Abbildung E.31: Preis bei Bietern fester Strategie im "Keine Wahl"–Szenario

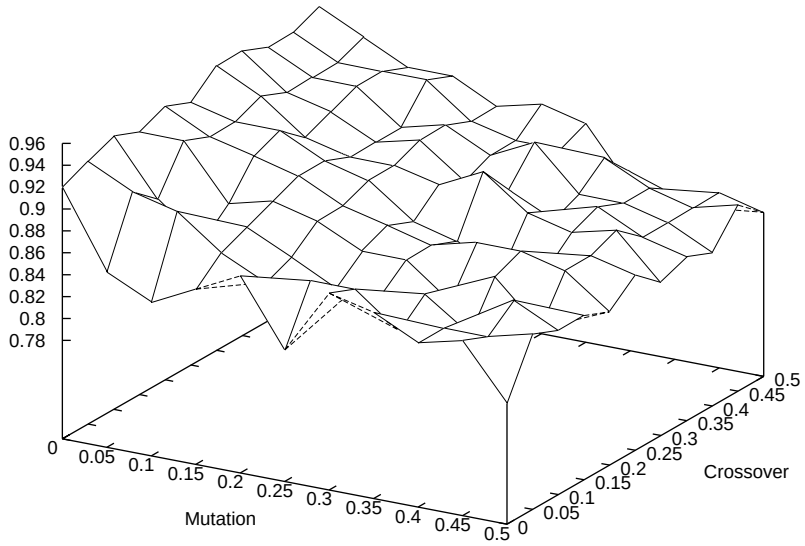


Abbildung E.32: Anteil der erfolgreichen Auktionen für genetisch optimierte Strategien im "Keine Wahl"–Szenario

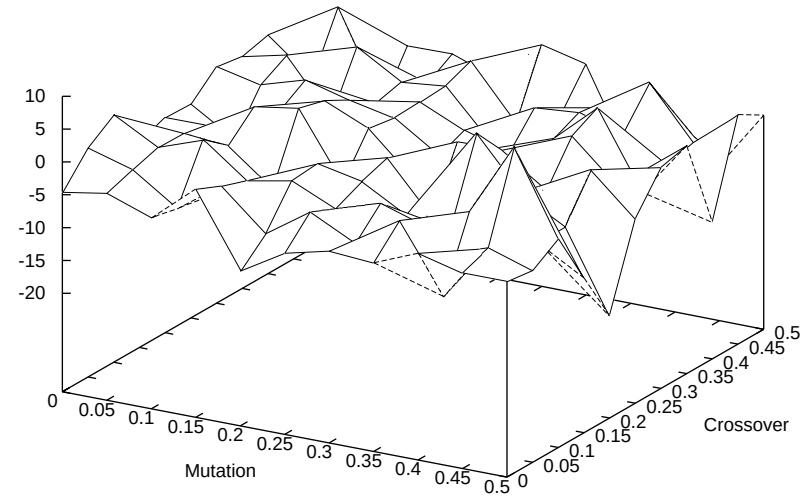


Abbildung E.33: Preisunterschied für genetisch optimierte und fixe Strategien als Anbieter im "Keine Wahl"–Szenario

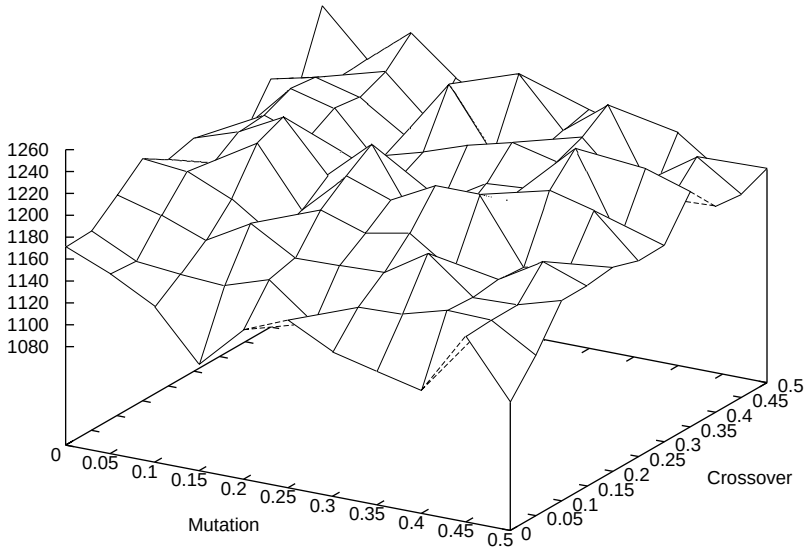


Abbildung E.34: Preis bei Bietern mit genetischem Algorithmus im preislimitierten Szenario

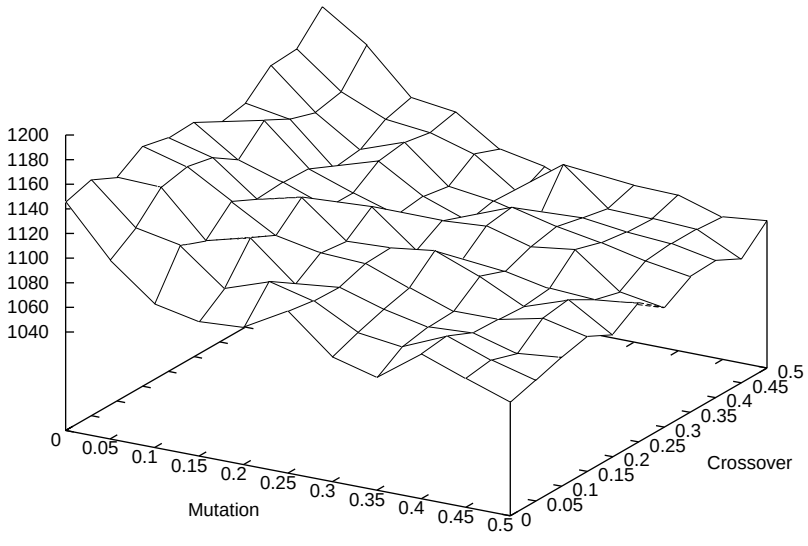


Abbildung E.35: Preis bei Bietern mit fester Strategie im preislimitierten Szenario

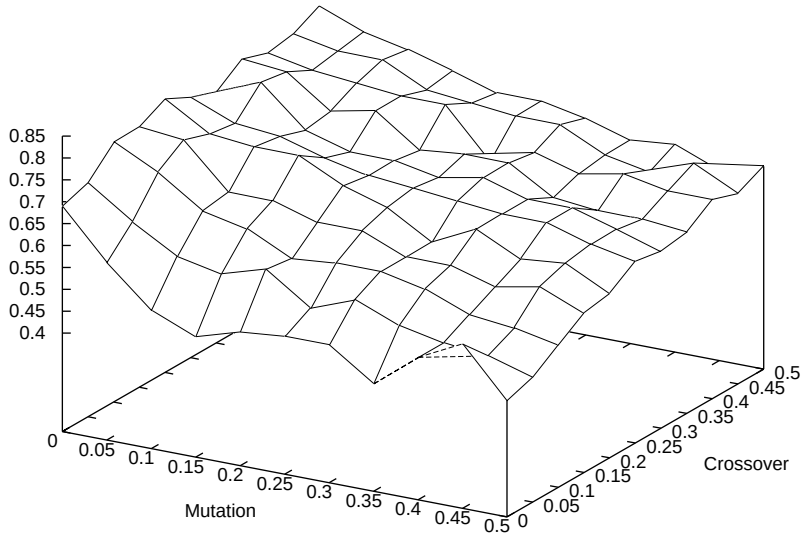


Abbildung E.36: Anteil der erfolgreichen Auktionen für genetisch optimierte Strategien im preislimitierten Szenario

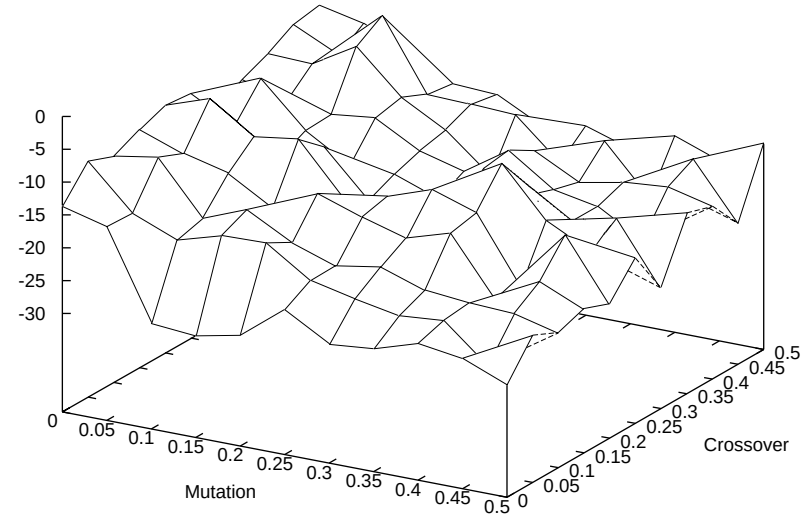


Abbildung E.37: Preisunterschied für genetisch optimierte und fixe Strategien als Anbieter im preislimitierten Szenario

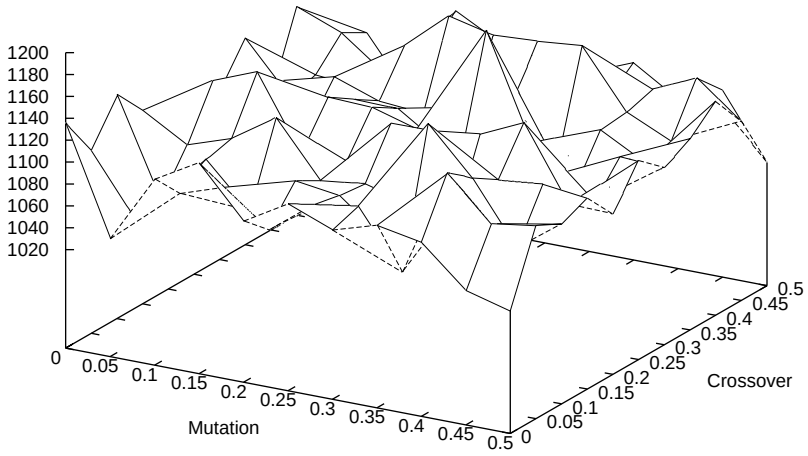


Abbildung E.38: Preis bei Bietern mit genetischem Algorithmus im anteilslimitierten Szenario

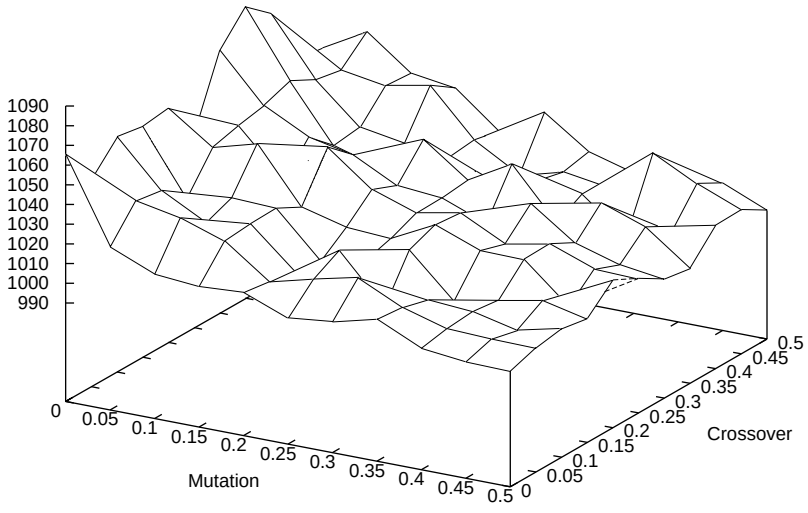


Abbildung E.39: Preis bei Bietern mit fester Strategie im anteilslimitierten Szenario

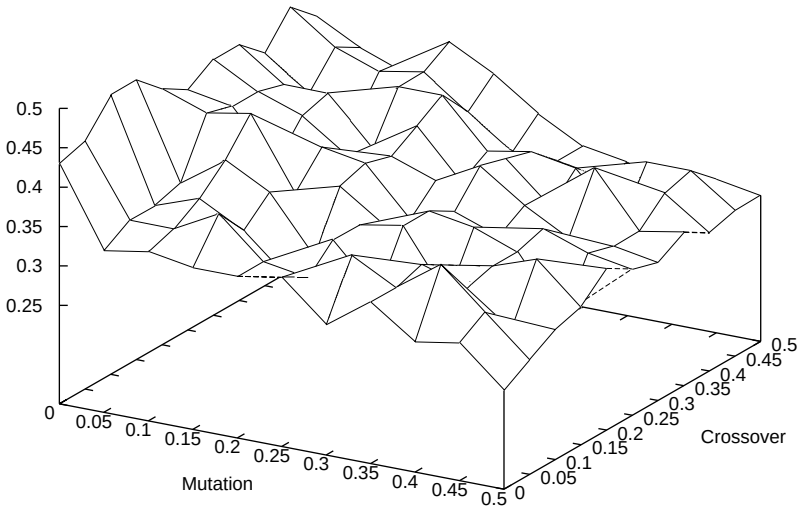


Abbildung E.40: Anteil der erfolgreichen Auktionen für genetisch optimierte Strategien im anteilslimitierten Szenario

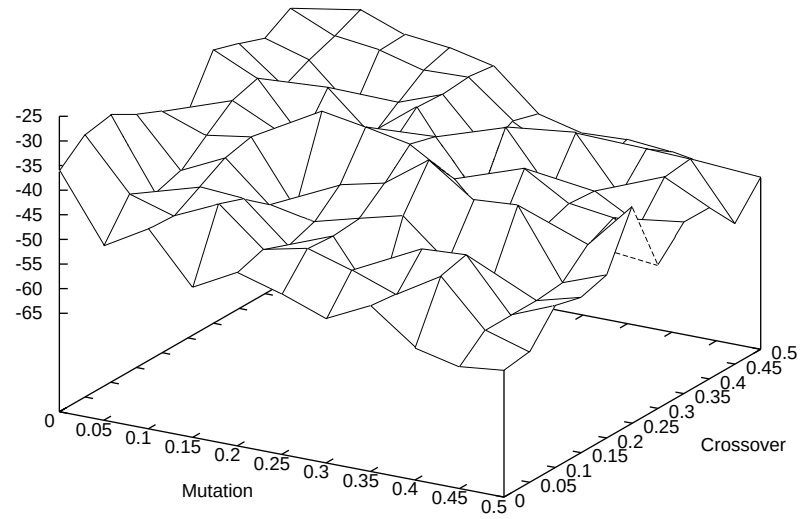


Abbildung E.41: Preisunterschied für genetisch optimierte und fixe Strategien als Anbieter im anteilslimitierten Szenario

Abbildungsverzeichnis

Tabellenverzeichnis

Symbolverzeichnis

A, $\hat{A}$	:	eine Matrix A bzw. $\hat{A}$
A^t	:	die Transponierte der Matrix A. Transponieren bedeutet das Spiegeln der Matrix an der Diagonalen.
$[a]$	:	die Gauß–Klammer: größte ganze Zahl, die kleiner oder gleich a ist.
B	:	der Vektor der Indizes der Basisvariablen;
E	:	Einheitsmatrix: $\begin{Bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{Bmatrix}$;
e	:	$(1, 1, \dots, 1)^t$ (alle Komponenten 1);
$H_{.k}$	:	der k-te Spaltenvektor der Matrix H;
$H_{i.}$	:	der i-te Zeilenvektor der Matrix H;
$\prec$	:	lexikographisch kleiner als;
lex max	:	lexikographisches Maximum;
lex min	:	lexikographisches Minimum;
$Mat(m, n)$	:	Menge der reellen Matrizen mit m Zeilen und n Spalten;
$M(P)$	:	zulässiger Bereich des Problems (P);
$Rg(H)$	:	Rang der Matrix H;
x_N	:	Nichtbasisvektor;
x_B	:	Basisvektor;
z	:	Zielfunktion;

Literaturverzeichnis

[AHU83] Alfred V. Aho, John E. Hopcroft und Jeffrey D. Ullmann. *Data structures and algorithms.* Addison–Wesley, 1983.

[Ame91] P. America. Designing an Object–Oriented Programming Language with Behavioral Subtyping. In J. W. de Baker, W. P. de Roever und G. Rozenberg, Hrsg., *Foundations of Object–Oriented Languages, REX School/Workshop*, Nummer 489 in Lecture Notes in Computer Science, S. 60–90. Springer–Verlag, 1991.

[Aus71] J. L. Austin. *How to do things with words.* Oxford University Press, 1971.

[BB98] J. P. Bigus und J. Bigus. *Constructing Intelligent Agents with Java.* Wiley Computer Publishing, 1998.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison–Wesley, 1999.

[Ber93] P. A. Bernstein. Middleware: An architecture for distributed system services. Technischer Bericht CRL 93/6, Digital Equipment Corp., Cambridge Research Lab, Cambridge, Mass., USA, 1993.

[Bib92] Wolfgang Bibel. *Deduktion: Automatisierung der Logik.* Oldenbourg, 1992.

[BK00] M. Benyoucef und R. Keller. A Conceptual Architecture for a Combined Negotiation Support System. In IEEE, Hrsg., *Proceedings of the 11th International Conference on Database and Expert Systems (DEXA 2000)*, 2000. (Zur Veröffentlichung angenommen).

[BL00] A. Bartelt und W. Lamersdorf. Agent–Oriented Concepts to Foster the Automation of e–Business. In IEEE, Hrsg., *Proceedings of the 11th International Conference on Database and Expert Systems (DEXA 2000)*, 2000. (Zur Veröffentlichung angenommen).

[BM87] E. P. Beisel und M. Mendel. *Optimierungsmethoden des Operation Research*, Band 1. Vieweg, 1987.

[BNKF98] W. Banzhaf, P. Nordin, R. E. Keller und F. D. Francone. *Genetic Programming — An Introduction: On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann Publishers, 1998.

[Bog99] Marko Boger. *Java in verteilten Systemen.* dpunkt–Verlag, 1999.

[BS97] Carrie Beam und Arie Segev. Automated Negotiations: A Survey of the State of the Art. Technischer Bericht, University of California, Berkeley, 1997.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76, Heft 2/3, S. 138–164, 1988.

[Cat94] R. G. G. Cattell. *Object Data Management.* Addison Wesley, 1994.

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris und G. Tsudik. Itinerant agents for mobile computing. Technischer Bericht RC 20010, IBM Research Division, T.J. Watson Research Center, 1995.

[Che80] B. F. Chellas. *Modal logic: an introduction.* Cambridge University Press, 1980.

[Chm98] Alexei Chmelev. Realisierung kommunikationsfähiger mobiler Agenten mit KQML und Voyager. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Januar 1998.

[CM94] W. F. Clocksin und C. S. Mellish. *Programming in Prolog.* Springer–Verlag, 1994.

[Con96] The Unicode Consortium. *The Unicode Standard – Version 2.0.* Addison–Wesley, 1996.

[CW85] L. Cardelli und P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17, Heft 4, S. 471–522, 1985.

[DAR93] DRAFT Specification of the KQML Agent–Communication Language. The DARPA Knowledge Sharing Initiative — External Interfaces Working Group, Juni 1993.

[Daw96] Richard Dawkins. *Das egoistische Gen.* Rowohlt Taschenbuch–Verlag, 1996.

[DG00] F. Dignum und M. Greaves, Hrsg. *Issues in Agent Communication.* Nummer 1916 in Lecture Notes in Artificial Intelligence. Springer, 2000.

[DGG95] K. R. Dittrich, S. Gatziu und A. Geppert. The Active Database System Manifesto: A Rulebase of ADBMS Features. In T. Sellis, Hrsg., *Proc. of the 2nd Intl. Workshop on Rules in Database Systems*, S. 3–17. Springer–Verlag, 1995.

[DM98] S. M. Davis und C. Meyer. *Blur: the speed of change in the connected economy.* Capstone, 1998.

[DS83] R. Davis und R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20, S. 63–109, 1983.

[eBa] eBay. eBay Website. http://www.ebay.de/.

[Eli74] T. S. Eliot. Choruses from "The Rock". In *Collected poems: 1909 – 1962.* Faber and Faber, London, reset and print Ausgabe, 1974.

[EMC99] Hartmut Ehrig, Bernd Mahr und Felix Cornelius. *Mathematisch–strukturelle Grundlagen der Informatik.* Springer–Verlag, 1999.

[FdM98] S. Fischer und H. de Meer. Using Cooperation in QoS Selection to Reduce Service Cost. In Griffel et al. [GTL98], S. 57–67.

[FFMM94] T. Finin, R. Fritzson, D. McKay und R. McEntire. KQML as an Agent Communication Language. In *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94).* ACM Press, November 1994.

[FG96] S. Franklin und A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In J.P. Müller, M.J. Wooldridge und N.R. Jennings, Hrsg., *Intelligent Agents III: Agent Theories, Architectures, and Languages (Proceedings of ECAI'96)*, LNCS. Springer–Verlag, August 1996.

[FIP] FIPA Specifications. http://www.fipa.org.

[Fit96] M. Fitting. *First–order logic and automated theorem proving.* Springer–Verlag, 2 Ausgabe, 1996.

[FMFM94] T. Finin, R. McKay, R. Fritzson und R. McEntire. Kqml: An information and knowledge exchange protocol. In K. Fuchi und T. Yokoi, Hrsg., *Knowledge Building and Knowledge Sharing.* Omsha and IOS Press, 1994.

[FS98] M. Fowler und K. Scott. *UML Distilled, Applying the Standard Object Modeling Language.* Addison–Wesley, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison–Wesley, 1995.

[GJ99] H. Grunwald und M. Jens. Dynamische Sicherheitskonfiguration mobiler Agenten. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Juni 1999.

[Göl97] M. Göllnitz. Generische Verarbeitung von Richtlinien für offene verteilte Systeme. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), September 1997.

[GN87] M. R. Genesereth und N. J. Nilsson. *Logical foundations of artificial intelligence.* Morgan Kaufmann Publishers, 1987.

[Gri98] F. Griffel. *Componentware – Konzepte und Techniken eines Softwareparadigmas.* dpunkt–Verlag, 1998.

[Gro98] Olaf Großler. Ereignisgesteuerte Aktivierung von Richtlinien. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Dezember 1998.

[Gro00] Olaf Großler. Konzeption und Entwicklung einer Steuerungseinheit zur Kontrolle von automatisierten Verhandlungen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Januar 2000.

[GTL98] F. Griffel, T. Tu und W. Lamersdorf, Hrsg. *Electronic Commerce – Proc. International IFIP/GI Working Conference on Trends in Distributed Systems for Electronic Commerce (TrEC'98)*, Band 2. dpunkt–Verlag, Juni 1998.

[Hal88] F. Halsall. *Data communications, computer networks and OSI.* Addison–Wesley, 1988.

[HFFS99] Y. Hoffner, C. Facciorusso, S. Field und A. Schade. Distribution Issues in the Design and Implementation od a Virtual Market Place. In L. Kutvonen, H. König und M. Tienari, Hrsg., *Proc. of the Second Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinky, Finland.* Kluwer Academic Publisher, Juni 1999.

[Hog89] D. Hogrefe. *Estelle, LOTOS und SDL — Standard–Spezifikationssprachen für verteilte Systeme.* Springer–Verlag, 1989.

[Hor89] L. R. Horn. *A natural history of negation.* University of Chicago Press, 1989.

[HSP96] T. H. Harrison, D. C. Schmidt und I. Pyarali. Asynchronous completion token. In *Proceedings of the 3rd Annual Pattern Languages of Programming Conference (PLOP)*, Allerton Park , Illinois, September 1996.

[HU79] J. E. Hopcroft und J. D. Ullmann. *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley, 1979.

[IBM] IBM. JKQML. http://alphaworks.ibm.com.

[INH98] T. Ishida, T. Nishida und F. Hattori. Overview of Community Computing. In Ishida [Ish98], S. 1–11.

[INP] VisiBroker Produktdokumentation. http://www.inprise.com/visibroker/.

[ION] Orbix/OrbixWeb Produktdokumentation. http://www.iona.com/products/orbix/index.html.

[Ish98] T. Ishida, Hrsg. *Community Computing – Collaboration over Global Information Networks.* Wiley, 1998.

[Jac97] C. Jacob. *Principia Evolvica : Simulierte Evolution mit Mathematica.* dpunkt–Verlag, 1997.

[Jen97] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Band 1: Basic Concepts (1992); 2: Analysis Methods (1994); 3: Practical Use (1997) der *EATCS Monographs on Theoretical Computer Science*. Springer–Verlag, 1997.

[KB00] R. Kowalczyk und V. Bui. On Fuzzy E-Negotiation Agents: Autonomous Negotiation with Incomplete and Imprecise Information. In IEEE, Hrsg., *Proceedings of the 11th International Conference on Database and Expert Systems (DEXA 2000)*, 2000. (Zur Veröffentlichung angenommen).

[KF98] Manoj Kumar und Stuart I. Feldman. Business Negotiations on the Internet. IAC Reports, IBM Research Division, T.J. Watson Research Center, 1998.

[KJ98] M. Knapik und J. Johnson. *Developing Intelligent Agents for Distributed Systems*. McGraw–Hill, 1998.

[KMW99] O. Kummer, D. Moldt und F. Wienberg. Symmetric Communication between Coloured Petri Net Simulations and Java-Processes. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, Band 1639 der *Lecture Notes in Computer Science*, S. 86–105, Williamsburg, Virginia, Juni 1999. Springer–Verlag.

[Koz92] J. R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, 1992.

[Kun98] Christian Kunze. Policy–Manager Ergänzungen mit Simplexverfahren. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Dezember 1998.

[Kun00] Christian Kunze. Ein Framework zur Anwendung von Rules in elektronischen Verhandlungsszenarien. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), April 2000.

[Lan99] Christian Langmann. Konzeption und Realisierung von Protokollen zur Steuerung automatisierter Verhandlungen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Januar 1999.

[LC93] G. T. Leavens und Y. Cheon. Extending CORBA IDL to Specify Behavior with Larch. Technischer Bericht TR 93-20, IOWA State University of Science and Technology, Department of Computer Science, August 1993.

[LM98] W. Lamersdorf und M. Merz, Hrsg. *Proc. International IFIP/GI Working Conference on Trends in Distributed Systems for Electronic Commerce (TrEC'98)*, Band 1 der *Lecture Notes in Computer Science*. Springer–Verlag, Juni 1998.

[LO98] D. B. Lange und M. Oshima. *Programming and Deploying Java Mobile Agents With Aglets*. Addison–Wesley, 1998.

[Mar99] Aureliusz Maresz. Rule–Management für offene verteilte Anwendungen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), April 1999.

[Mer96] M. Merz. *Elektronische Dienstemärkte: Modelle und Mechanismen zur Unterstützung von Handelstransaktionen in offenen verteilten Systemen.* Dissertation, Fachbereich Informatik, Universität Hamburg, 1996.

[Mer99a] M. Merz. *Elektronic Commerce – Marktmodelle, Technologien und Anwendungen.* dpunkt–Verlag, 1999.

[Mer99b] M. Merz. *Elektronische Dienstemärkte: Modelle und Mechanismen des Electronic Commerce.* Springer–Verlag, 1999.

[Mey97] B. Meyer. *Object–Oriented Software Construction.* Prentice Hall, 2 Ausgabe, 1997.

[MGT+99] M. Merz, F. Griffel, T. Tu, S. Müller-Wilken, H. Weinreich, M. Boger und W. Lamersdorf. Supporting Electronic Commerce Transactions with Contracting Services. *International Journal on Cooperative Information Systems*, 7, Heft 4, 1999.

[MIT] MIT Media Lab. Marketmaker. http://maker.media.mit.edu/.

[MJ97] K. Müller-Jones. *Koordinierte Dienstnutzung in offenen verteilten Dienstemärkten.* Dissertation, Fachbereich Informatik, Universität Hamburg, ProBusiness, 1997.

[Mül96] J. P. Müller. A Cooperation Model for Autonomous Agents. In J. P. Müller, M. J. Wooldridge und N. R. Jennings, Hrsg., *Intelligent Agents III: Agent Theories, Architectures, and Languages (Proceedings of ECAI'96)*, LNCS. Springer–Verlag, August 1996.

[MMSS94] D.A. Marriot, M. Mansouri-Samani und M. Sloman. Specification of Management Policies. In *Proceedings of the 5th IFIP/IEEE International Workshop on Distributed Systems: Operation and Management*, Toulouse, Oktober 1994. IEEE Press.

[Mof94] J.D. Moffett. Specification of Management Policies and Discretionary Access Control. In Sloman [Slo94a], S. 455–480.

[MPT98] H. De Meer, A. Puliafito und O. Tomarchio. Management of QoS with Software Agents. *Cybernetics and Systems: An International Journal*, 27, Heft 5, S. 499–523, Mai 1998.

[MS93] J. Moffett und M. Sloman. Policy Hierachies for Distributed Systems Management. *IEEE Journal on Selected Areas in Communications*, 11, Heft 9, Dezember 1993.

[MTL96] M. Merz, M. T. Tu und W. Lamersdorf. Dynamic Support Service Selection for Business Transactions in Electronic Service Markets. In O. Spaniol, C. Linnhoff-Popien und B. Meyer, Hrsg., *Proc. Aachen Workshop "Trends in Distributed Systems"*, S. 183–195. Verlag der Augustinus Buchhandlung, Aachener Beiträge zur Informatik, Oktober 1996.

[ND95] O. Nierstrasz und L. Dami. Component–Oriented Software Technology. In Nierstrasz und Tsichritzis [NT95], Kapitel 1, S. 3–28.

[Nie95] O. Nierstrasz. Regular Types for Active Objects. In Nierstrasz und Tsichritzis [NT95], Kapitel 4, S. 99–121.

[NT95] O. Nierstrasz und D. Tsichritzis, Hrsg. *Object–Oriented Software Composition.* The Object–Oriented Series. Prentice Hall, 1995.

[Obj] Voyager Produktdokumentation. http://www.objectspace.com/products/vgrProdDocs.asp.

[ODP94a] ISO/IEC DIS 10746: Information Technology – Open Systems Interconnection – Data Management and Open Distributed Processing – Basic Reference Model of Open Distributed Processing – Part 1: Overview. ISO/IEC, 1994.

[ODP94b] ISO/IEC DIS 10746: Information Technology – Open Systems Interconnection – Data Management and Open Distributed Processing – Basic Reference Model of Open Distributed Processing – Part 3: Architecture. ISO/IEC, 1994.

[ODP95] ISO/IEC DIS 13235: Information Technology – Open Systems Interconnection – Data Management and Open Distributed Processing – Draft ODP Trading Function. ISO/IEC, Juni 1995.

[OHE99] R. Orfali, D. Harkey und J. Edwards. *Client/Server Survival Guide.* John Wiley & Sons, 3 Ausgabe, 1999.

[Oli96] J. R. Oliver. *On Artificial Agents for Negotiation in Electronic Commerce.* Dissertation, The Wharton School, University of Pennsylvania, 1996.

[OMG95] Common Facilities Architecture Revision 4.0. OMG Document 95-1-2, Januar 1995.

[OMG96] Trading Object Service. OMG Document 96-05-06, 1996.

[OMG97] A Discussion of the Object Management Architecture. OMG, Januar 1997.

[OMG98a] Business Object Component Architecture Revision 1.2. OMG Document 98-07-01, 1998.

[OMG98b] CORBAservices: Common Object Services Specification. OMG Document 98-12-09, Dezember 1998.

[OnS] OnSale. OnSale Website. http://www.onsale.com/.

[OV91] M. T. Özsu und P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[Pat99] N. W. Paton, Hrsg. *Active Rules in Database Systems.* Monograph in Computer Science. Springer–Verlag, 1999.

[PB96] A. Puder und C. Burger. New Concepts for Qualitative Trader Cooperation. In A. Schill, C. Mittasch, O. Spaniol und C. Popien, Hrsg., *Proc. IFIP/IEEE International Conference on Distributed Platforms*. Chapman & Hall, 1996.

[PMG95] A. Puder, S. Markwitz und F. Gudermann. Service Trading Using Conceptual Structures. In *Proc. 3rd International Conference on Conceptual Structures (ICCS'95)*, University of California, Santa Cruz, 1995. Springer–Verlag.

[PMGG95] A. Puder, S. Markwitz, F. Gudermann und K. Geihs. AI–Based Trading in Open Distributed Environments. In *Proc. 3rd International Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, 1995. Chapman & Hall.

[Pou93] W. Poundstone. *Prisoner's Dilemma — John von Neumann, Game Theory, and the Puzzle of the Bomb*. Oxford University Press, 1993.

[Rai82] H. Raiffa. *The Art and Science of Negotiation*. Harvard University Press, Cambridge, London, 1982.

[Rhe93] H. Rheingold. *The Virtual Community*. Addison–Wesley, 1993.

[RZ94a] J. Rosenschein und G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiations among Computers*. MIT Press, 1994.

[RZ94b] J. Rosenschein und G. Zlotkin. Designing Conventions for Automated Negotiations. *AI Magazine*, S. 29–46, FALL 1994.

[Sch87] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1987.

[Sch93a] A. Schill. *DCE: Das OSF Distributed Computing Environment*. Springer–Verlag, 1993.

[Sch93b] B. Schmid. Electronic Markets. *EM – Electronic Markets*, 3, Heft 9/10, S. 3–4, 1993.

[Sch98] Martin Schmidt. Realisierung eines generischen und persistenten Eigenschaftsdienstes für offene, verteilte Anwendungen. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), November 1998.

[Seb92] J. K. Sebenius. Negotiation Analysis: A Characterization an Review. *Management Science*, 38, Heft 1, S. 18–38, Januar 1992.

[Sie96] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.

[Sin98] M. P. Singh. Agent Communication Languages: Rethinking the Principles. *Computer*, 31, Heft 12, S. 40–47, Dezember 1998.

[SL95] T. Sandholm und V. Lesser. Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework. In V. Lesser, Hrsg., *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, S. 328–335, San Francisco, Juni 1995. AAAI / MIT Press.

[Slo94a] M. Sloman, Hrsg. *Network and Distributed Systems Management.* Addison-Wesley, 1994.

[Slo94b] M. Sloman. Policy Driven Management for Distributed Systems. *Journal of Network and Systems Management*, 2, Heft 4, 1994.

[Sow84] J. F. Sowa. *Conceptual Structures.* Addison–Wesley, 1984.

[Sow91] J. F. Sowa, Hrsg. *Principles of Semantic Networks.* Morgan Kaufmann Publishers, 1991.

[ST94] M. Sloman und K. Twidle. Domains: A Framework for Structuring Management Policy. In Sloman [Slo94a], S. 433–453.

[Str91] B. Stroustrup. *The C++ programming language.* Addison–Wesley, 2 Ausgabe, 1991.

[TGL99] M. T. Tu, F. Griffel und W. Lamersdorf. Integration of Intelligent and Mobile Agents for E-Commerce – A Research Agenda. In St. Kirn und M. Petsch, Hrsg., *Proc. Workshop "Intelligente Softwareagenten und betriebswirtschaftliche Anwendungsszenarien".* TU Ilmenau, FG Wirtschaftsinformatik 2, Arbeitsbericht Nr. 14, 1999.

[TGML97a] M. T. Tu, F. Griffel, M. Merz und W. Lamersdorf. Generic Policy Management for Open Service Markets. In H. König und K. Geihs, Hrsg., *Proc. of the Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), Cottbus, Germany.* Chapman & Hall, September 1997.

[TGML97b] M. T. Tu, F. Griffel, M. Münke und W. Lamersdorf. Revidierte Projektbeschreibung "Koordinations–Management in Offenen Verteilten Systemen". Technischer Bericht, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Januar 1997.

[TGML98] M. T. Tu, F. Griffel, M. Merz und W. Lamersdorf. A Plug-In Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents. In K. Rothermel und F. Hohl, Hrsg., *Proc. 2. Intl. Workshop on Mobile Agents, MA'98, Stuttgart*, Nummer 1477 in Lecture Notes in Computer Science, S. 222–236. Springer–Verlag, 1998.

[TGML99] M. T. Tu, F. Griffel, M. Merz und W. Lamersdorf. Interaction–Oriented Rule Management for Mobile Agent Applications. In L. Kutvonen, H. König und M. Tienari, Hrsg., *Proc. of the Second Int. Working Conference on Distributed Applications and Interoperable Systems (DAIS'99), Helsinky, Finland*, S. 423–437. Kluwer Academic Publisher, Juni 1999.

[TKL00] M. T. Tu, C. Kunze und W. Lamersdorf. A Rule Management Framework for Negotiating Mobile Agents. In *Proc. of the 4th International Enterprise Distributed Object Computing Conference (EDOC'2000), Makuhari, Japan*, S. 135–143. IEEE, 2000.

[TLGL99] M. T. Tu, C. Langmann, F. Griffel und W. Lamersdorf. Dynamische Generierung von Protokollen zur Steuerung automatisierter Verhandlungen. In K. Beiersdörfer, G. Engels und W. Schäfer, Hrsg., *Proc. 29. Jahrestagung der Gesellschaft für Informatik (Informatik'99)*, Informatik aktuell, S. 50–59. Springer–Verlag, 1999.

[Tsc94] V. Tschammer. *Integration kooperierender Systeme – Architekturen und Dienstplattformen für offene verteilte Systeme.* Dissertation, GMD–Bericht Nr. 220, R. Oldenbourg, 1994.

[TSGL99] M. T. Tu, C. Seebode, F. Griffel und W. Lamersdorf. An Actor–Based Framework for Negotiating Mobile Agents. In Yiming Ye und Jiming Liu, Hrsg., *Proc. IAT'99 Workshop on Agents in Electronic Commerce, WAEC'99, Hong Kong, China*, S. 69–78. Hong Kong Baptist University, 1999.

[TSL00] M. T. Tu, C. Seebode und W. Lamersdorf. DynamiCS: An Actor–Based Framework for Negotiating Mobile Agents. *Electronic Commerce Research Journal*, 2000. (Zur Veröffentlichung angenommen).

[TWL00] M. T. Tu, E. Wolff und W. Lamersdorf. Genetic Algorithms for Automated Negotiations: A FSM–Based Application Approach. In A. M. Tjoa, R. R. Wagner und A. Al-Zobaidie, Hrsg., *Proceedings of the 11th International Conference on Database and Expert Systems (DEXA 2000)*, S. 1029–1033. IEEE, 2000.

[WC96] J. Widom und S. Ceri. *Active Database Systems.* Morgan Kaufmann, 1996.

[Weg96] P. Wegner. Interactive software technology. In A. B. Tucker, Hrsg., *The Computer Science and Engineering Handbook*, Kapitel 113, S. 2440–2463. CRC Press, Dezember 1996.

[Weg97] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40, Heft 5, S. 80–91, Mai 1997.

[Weg98] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, Februar 1998.

[Wei78] J. Weizenbaum. *Die Macht der Computer und die Ohnmacht der Vernunft.* Suhrkamp, 1978.

[WG99a] P. Wegner und D. Goldin. Interaction as a framework for modeling. In P. Chen, J. Akoka, H. Kangassalo und B. Thalheim, Hrsg., *Conceptual Modeling – Current Issues and Future Directions*, Lecture Notes in Computer Science. Springer–Verlag, 1999.

[WG99b] P. Wegner und D. Goldin. Mathematical models of interactive computing. Technischer Bericht, Brown University, Januar 1999.

[Wie95a] G. Wiederhold. Mediation in information systems. *Computing Surveys*, 27, Heft 2, S. 265–267, 1995.

[Wie95b] R. Wies. *Policies in Integrated Network and Systems Management.* Verlag Shaker, Aachen, 1995.

[Wol00] Eberhard Wolff. Genetische Programmiermodelle für automatische Verhandlungen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme (VSYS), Januar 2000.

[WTL00] E. Wolff, M. T. Tu und W. Lamersdorf. Using Genetic Algorithms to Enable Automated Auctions. In K. Bauknecht, S. M. Madria und G. Pernul, Hrsg., *Proceedings of the 1st International Conference on Electronic Commerce and Web Technologies (EC-Web 2000)*, Nummer 1875 in Lecture Notes in Computer Science, S. 389–398. Springer–Verlag, 2000.

[WWC92] G. Wiederhold, P. Wegner und S. Ceri. Towards megaprogramming. *Communications of the ACM*, 35, Heft 11, S. 89–99, November 1992.

[YS95] D. M. Yellin und R. E. Strom. Collaboration specifications and component adapters. Technischer Bericht, IBM T. J. Watson Research Center, 1995.

[ZS96] D. Zeng und K. Sycara. How Can an Agent Learn to Negotiate? In J. P. Müller, M. J. Wooldridge und N. R. Jennings, Hrsg., *Intelligent Agents III: Agent Theories, Architectures, and Languages (Proceedings of ECAI'96)*, LNCS. Springer–Verlag, August 1996.

[ZW00] D. M. Zhang und W. Y. Wong. Reusing Previous Negotiation Experiences in Multi–Agent Negotiation. *Electronic Commerce Research Journal*, 2000. (Zur Veröffentlichung angenommen).