Visuelle Modellierung agentenbasierter Systeme

D I S S E R T A T I O N

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)
im Fach Informatik

an der
Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

von
Dipl.-Inform. Dipl.-Phys. Ralph Depke
geboren am 13.06.1965 in Hannover

Datum der mündlichen Prüfung: 20. Dezember 2004

Zusammenfassung

Diese Arbeit beschreibt einen Ansatz zur visuellen Modellierung von Softwaresystemen, die aus autonom und zielgerichtet (proaktiv) handelnden, mit ihrer Umgebung kommunizierenden Software-Bausteinen, sogenannten Software-Agenten, aufgebaut werden.

Immer mehr Softwaresysteme zeichnen sich durch Verteiltheit, Dynamik und Offenheit aus. Software-Agenten sollen die Erstellung solcher Systeme erleichtern. Agentenbasierte Systeme sind modular aufgebaut, robust und dynamisch anpassbar. Der Begriff des Software-Agenten ist durch die Kerneigenschaften Autonomie, Proaktivität und strukturierte Interaktion bestimmt. Anhand dieser Eigenschaften werden Agenten von Objekten und Komponenten abgegrenzt. Die softwaretechnische Entwicklung agentenbasierter Systeme verlangt nach einer geeigneten Modellierungssprache und einem Prozessmodell, das die Verwendung der Sprache regelt. Für beides existieren bislang nur unzureichende Vorschläge. Diese Arbeit trägt zur Lösung dieser Problematik wie folgt bei:

Ein neues abstraktes und formales Modell agentenbasierter Systeme zeigt, dass die Kerneigenschaften präzise dargestellt werden können. Im Unterschied zu vorhandenen Modellen lässt sich nicht nur ein einzelner Agent, sondern ein System von Agenten (Multiagentensystem) modellieren. Wichtige Architekturmodelle agentenbasierter Systeme sind mit dem abstrakten Agentenmodell verträglich. Die wenigen Ausdrucksmittel des abstrakten Modells motivieren eine Verfeinerung in Form eines neuen Modells von gekoppelten I/O-Automaten.

In der praktischen Softwareentwicklung wird die Modellierungssprache Unified Modeling Language (UML) eingesetzt. Die neue Sprache AML zur Entwicklung agentenbasierter Systeme ist deshalb durch ein UML-Profil definiert. Die Semantik von AML-Modellen ist im semantischen Bereich der Graphtransformation definiert, wobei der allgemeine semantische Bereich auf einen adäquaten agentenorientierten Bereich eingeschränkt wird. Dort sind Konstituenten zu den Kernkonzepten von Agenten definiert. Die durch AML beschriebenen agentenbasierten Systeme sind mit dem neuen Automatenmodell für Multiagentensysteme verträglich.

Konstituenten für Proaktivität sind Ziele und Strategien. Gezeigt wird in dieser Arbeit, wie das Erreichen von Zielen durch Anwendung von Strategien mit der Methode des Model Checking überprüft werden kann.

Das neue Prozessmodell APM beschreibt, wie die Sprache AML zur Modellierung agentenbasierter Systeme eingesetzt werden kann.

Abstract

This dissertation describes an approach to the visual modeling of software systems which are composed of autonomous and proactive software entities interacting with their environment. These entities are named software agents.

More and more software systems are distributed, dynamic and open with respect to their environment. Software agents shall ease the development of such systems. Agent based systems provide a modular structure. They are robust and can be adapted dynamically to changing requirements. The notion of software agent relies on the fundamental properties of autonomy, proactivity and structured interaction. According to these properties agents are distinguished from objects and components. For the development of agent based systems an appropriate modeling language and a process model describing the correct usage of the language are demanded. For both aspects only unsufficient or incomplete approaches exist. This thesis contributes to the solution in the following way:

A new abstract and formal model of agent based systems shows, that the fundamental properties can be expressed precisely. Differently from existing models not only a single agent but a multi agent system can be modeled. The most important architectural models of agent-based systems are compatible with the abstract model. The rather restricted structure of the abstract model motivates the refinement by a model of coupled I/O-automata.

The Unified Modeling Language is widely used in industrial software development. Therefore, the new language AML (Agent Modeling Language) for the development of agent-based systems is defined as a UML profile. The semantics of AML models is defined in the semantic domain of graph transformation. The semantic domain is restricted to an adequate agent-oriented domain. There the constituents of the fundamental agent properties are defined. Agent based systems which are described using AML are shown to be compatible with I/O-automata model of multi agent systems.

Constituents for proactivity are goals and strategies. In this thesis, it is shown that the reachability of goals through the application of strategies can be checked by applying a model checking method.

The new process model APM describes, how the language AML can be applied to the modeling of agent-based systems.

Dank

Ich danke Gregor Engels dafür, mir diese Arbeit ermöglicht und sie immer unterstützt zu haben. In vielen Diskussionen schärfte er meinen kritischen Blick dafür, wissenschaftliche Standards anzuwenden. Seine konstruktive Kritik half mir sehr dabei, diese Arbeit abzuschließen.

Für die Übernahme des zweiten Gutachtens zu dieser Arbeit danke ich Wilhelm Schäfer.

Für sehr viele anregende Diskussionen danke ich ganz besonders Reiko Heckel. Aus unserer langjährigen Zusammenarbeit entsprangen viele inhaltliche Fortschritte.

Jochen Küster danke ich für die fruchtbare Zusammenarbeit im Zusammenhang und in Folge seiner Diplomarbeit.

Allen Mitarbeitern der Arbeitsgruppe Datenbank- und Informationssysteme danke ich für das überaus angenehme Arbeitsklima.

Ich danke meiner Frau Antje und unseren Kindern für ihre besondere Geduld und lange Unterstützung. Meinen Eltern danke ich für ihre stetige Begleitung.

Contents

Chapter 1

Einleitung

Software wird immer komplexer und erfordert deshalb verstärkt ein ingenieursmäßiges Vorgehen bei deren Entwicklung. In den letzen 15 Jahren hat sich zur Beherrschung der Komplexität in der Softwareentwicklung das Paradigma der Objektorientierung in den Sprachen und Methoden durchgesetzt. Aktuelle und zukünftige Systeme, wie beispielsweise die aufkommenden Ad-Hoc-Netzwerke, zeichnen sich durch immer mehr Dynamik und Offenheit aus. Die Entwicklung derartiger Systeme soll auf der Basis spezifischer Konzepte wie der der Softwareagenten erleichtert werden. Gegenstand der aktuellen Forschung sind geeignete Sprachen und Methoden für die Entwicklung agentenbasierter Systeme. Damit einhergehende Probleme bilden den Ausgangspunkt dieser Arbeit.

Nach der einleitenden Motivation werden der Problemkreis dieser Arbeit abgegrenzt und darauf aufbauend die Ziele der Arbeit formuliert.

1.1 Motivation

Professionelle Softwareentwicklung ist in der heutigen, vielfach durch Software geprägten Welt enorm wichtig. Die wachsenden Anforderungen an Quantität und insbesondere Qualität von Software verlangen nach systematischen Techniken zur Softwareproduktion. Derartige Anforderungen sind im Rahmen der Industrialisierung von den klassischen Ingenieurdisziplinen für Hardwareprodukte wie Maschinen, Gebäude oder elektrische Geräte erfüllt worden.

In den klassischen Ingenieurdisziplinen werden ausgereifte und genormte Verfahren zur Planung, Erzeugung und Wartung von Produkten eingesetzt. In den unterschiedlichen Phasen werden verschiedene Sichten auf ein Produkt festgelegt, die unterschiedliche Merkmale des Produktes wiedergeben. Beispiele sind Konstruktionszeichnungen und Stücklisten. Mit Hilfe von Werkzeugen, wie beispielsweise CAD-Systemen, werden Routinearbeiten unterstützt und Fehler vermieden. So können Stücklisten aus Konstruktionszeichnungen erzeugt werden, wodurch Konsistenzfehler ausgeschlossen sind. Weiterhin gibt es in den klassischen Ingenieurdisziplinen fest vorgegebene Entwicklungsprozesse, die Ingenieuren allgemein bekannt sind, in der Praxis erprobt sind und breit verwendet werden.

Softwaretechnik ist die Informatikdisziplin, die sich mit ingenieurmäßigen Techniken zur Produktion von Software befasst. Software unterscheidet sich von Hardware vor allem durch die problemlose Reproduzierbarkeit von Produkten. Deshalb ist Produktion von Software überwiegend mit Entwicklung von Software gleichgesetzt. Heutige Software besteht aus sehr vielen, miteinander dynamisch verknüpften Teilen. Diese Komplexität von Software geht vielfach über das Maß hinaus, was bei klassischen "Hardware"-Produkten, wie Automobilen, Gebäuden oder Mikrowellen vorliegt.

Parnas definierte Softwaretechnik im Jahr 1975 in [87] wie folgt: "Software engineering is multi-person construction of multi-version software."

Nicht triviale Software wird also in der Regel von mehr als einer Person entwickelt. Diese arbeitsteilige Entwicklung von Komponenten, die zu größeren Systemen kombiniert werden, bedingt, dass Software-Ingenieure über die Leistungen ihrer Komponenten kommunizieren müssen. Dieser Austausch von Informationen über Software erfordert angemessene Sprachmittel und die Abstraktion von Details der Implementierung von Komponenten. Vielmehr müssen sich Software-Ingenieure auf der Ebene der Spezifikation austauschen, auf der beschrieben wird, *was* die Komponenten leisten. Software-Ingenieure sind wechselseitig Kunden, die ein fremdes Produkt in Form einer Komponente in ihrem eigenen Produkt benutzen möchten.

Der zweite Teil im obigen Zitat berührt das charakteristische Merkmal von Software, dass Software in der Regel einem Evolutionsprozess ausgesetzt ist. Dieses Merkmal rührt daher, dass einerseits die Anforderungen an Software und die Fähigkeiten der zugrunde liegenden Hardware in den letzten Jahrzehnten immer mehr gewachsen sind und dass andererseits Software sehr leicht änderbar ist. Für die Softwaretechnik resultiert daraus die Notwendigkeit, den Prozess der Softwareentwicklung zu strukturieren.

Der spätere Benutzer eines Softwaresystems formuliert seine Anforderungen in einer ihm verständlichen Sprache. Das ist mindestens die natürliche Sprache. Aus dieser informellen Beschreibung wird zusammen mit dem Softwareentwickler eine strukturierte Anforderungsdefinition, das Pflichtenheft, gewonnen, die sowohl für den späteren Benutzer als auch für den Softwareentwickler verständlich ist. Zwischen der Anforderungsbeschreibung und dem Softwaresystem, das gemäß den Anforderungen funktionieren soll, besteht eine große Distanz hinsichtlich der Beschreibungsmittel, der Details, etc.

Es ist schwierig, unmittelbar aus dem Pflichtenheft ein Softwaresystem zu programmieren, das den Anforderungen genügt. Die Kluft zwischen Pflichtenheft und Implementierungsdokumenten wird durch Dokumente überbrückt, die durch die Analyse- und Entwurfsaktivität gewonnen werden. Das Architekturdokument als Ergebnis der Analyseaktivität beschreibt Struktur und Verhalten des Softwaresystems auf grobgranularer Ebene, wobei die Anforderungen des Pflichtenheftes berücksichtigt sind. Die Entwurfsaktivität hat zum Ziel, anhand des Architekturdokumentes ein verfeinertes Modell des Softwaresystems zu erzeugen, in dem alle Anforderungen des Pflichtenheftes berücksichtigt sind. Im Unterschied zum Architekturdokument werden die durch die Implementierungs- und Einsatzplattform gesetzten Randbedingungen berücksichtigt.

Das Paradigma der Objektorientierung hat sich in den vergangenen 15 Jahren in vielen

Bereichen der Softwareentwicklung durchgesetzt. Die Vorteile der Integration von Daten und Funktionen, der Unterscheidung zwischen Instanzen und zugehörigen Klassen und der Mechanismen wie Vererbung haben zum breiten Einsatz zugehöriger Technologien geführt. Verschiedene Arten von Systemen wie Informationssysteme, Realzeitsysteme oder eingebettete Systeme werden objektorientiert entwickelt (vgl. [55]).

Ein wichtiger Fortschritt der Softwaretechnik der letzten 10 Jahre besteht in der Entwicklung von objektorientierten Modellierungssprachen und -methoden wie FUSION, OMT, etc. Mit der Standardisierung der Unified Modeling Language (UML) durch die Object Management Group (OMG) im Jahre 1997 wurde die Inflation der Modellierungssprachen beendet, da sich UML in den folgenden Jahren sowohl in der Forschung als auch der Industrie weitgehend durchsetzte. Weiterhin sind in den letzten Jahren umfangreiche Prozessmodelle für die Entwicklung objektorientierter Systeme wie der Rational Unified Process (RUP) und Catalysis auf der Basis der Modellierungssprachen UML entstanden, die den Ablauf der Softwareentwicklung und die Ergebnisse in Form von Modellen standardisieren.

Die Softwaretechnik für die Entwicklung objektorientierter Systeme hat insgesamt einen hohen Entwicklungsstand erreicht. Die weitergehende Vernetzung von Rechnern führt zu neuartigen Anwendungen, für die neue softwaretechnische Fragestellungen entstehen.

Neuartige Anwendungen. Heutige und kommende Softwaresysteme sind in der Regel verteilt und operieren nebenläufig. Im Unterschied zu klassischen objektorientierten Systemen existiert kein globaler Kontrollfluss mehr. Beispiele solcher Systeme sind Workflow Management Systeme und Grid Systeme [1, 26].

Für solche Systeme ist es notwendig, dass sie zur Laufzeit flexibel mit anderen Systemen, die während der Systementwicklung noch nicht bekannt waren, kommunizieren und zusammen funktionieren müssen. Das erfordert, dass solche Systeme autonom darüber entscheiden, ob und wie sie mit anderen Systemen kooperieren. Die Entscheidungen in einem solchen System werden im Hinblick auf eigene Ziele getroffen.

Systeme, die fähig sind, mit a priori unbekannten Systemen sinnvoll zusammenzuwirken, sind *offen*. Offene Systeme haben "öffentlich" sichtbare Schnittstellen, über die a priori unbekannte Komponenten zur Laufzeit integriert werden (vgl. Coulouris et al. [25]). Die Möglichkeit der dynamischen Integration von Komponenten fördert die Skalierbarkeit offener Systeme. Andererseits können Komponenten, die den aktuellen Anforderungen nicht oder nicht mehr genügen, auch leichter durch geeignetere ersetzt werden. Dadurch wird die Robustheit des Systems erhöht (vgl. Coulouris et al. [25]).

Komponentenbasierte Systeme beruhen auf der Idee der Kopplung von Softwarebausteinen zur Laufzeit. Für solche Systeme liegt der Schwerpunkt auf der Beschreibung von Schnittstellen. Die in dynamischen und unvorhersehbaren Umgebungen wichtigen Eigenschaften der Autonomie und der darauf aufbauenden Zielorientierung werden weniger berücksichtigt.

Konzepte für Softwareagenten und darauf basierende Systeme integrieren gerade de-

rartige Eigenschaften und grenzen sich dadurch von "einfachen" komponentenbasierten Systemen ab.

Agenten. Die Konzepte und Technologien für agentenbasierte Systeme werden in zunehmendem Maße attraktiv für viele Bereiche der Softwareproduktion. Vorteile des Konzeptes "Agent" wie verbesserte Skalierbarkeit, Robustheit und Proaktivität von Softwarearchitekturen sind Anforderungen in vielen heutigen und noch mehr in zukünftigen Systemen. Agenten bilden seit einiger Zeit die Basis für neue Architekturen von Workflow Management Systemen (WFMS). Derartige Systeme werden aufgrund ihrer Charakteristika als prädestiniert für den Einsatz von Agenten angesehen. Agentenbasierte Workflowsysteme sind schon Realität und werden durch Firmen wie *lost wax* als Produkte verkauft [73]. In Kapitel 9 wird dieser Anwendungsbereich weiter diskutiert. Zunächst sind aber die Eigenschaften von Agenten genauer zu betrachten.

Primär sind agentenbasierte Systeme reaktiv. Reaktivität schließt hier ein, dass Agenten ihre Umgebung beobachten und auf Zustände oder Zustandsänderungen reagieren. Spezifische Merkmale des Konzeptes Agent werden in einer vielfach zitierten Arbeit von Franklin und Graesser abgegrenzt [51]. Die drei Kernkonzepte für Agenten sind dabei *Proaktivität*, *Autonomie* und *strukturierte Interaktion* (vgl. auch Weiss [103]).

Unter *Proaktivität* ist hier zu verstehen, dass die Aktionen oder Reaktionen eines Agenten durch seine Ziele und die daraus abgeleiteten Strategien bestimmt sind. Proaktivität eines Agenten fordert notwendigerweise dessen zeitlich kontinuierliche Aktivität. Agenten sollen ihre Ziele von sich aus kontinuierlich verfolgen und nicht erst dann, wenn sie von außen dazu angestoßen werden. Da ein System in der Regel aus einer Vielzahl von Agenten besteht, die alle ihre Ziele verfolgen, agieren die Agenten nebenläufig.

Autonomie (von griech. *autos*, selbst, und *nomos*, Gesetz) ist ein Konzept, das Agenten die Fähigkeit zuspricht, selbständig und unabhängig Entscheidungen über auszuführende Aktivitäten zu treffen. Agenten kapseln und kontrollieren nicht nur ihren Zustand, sondern auch ihr Verhalten. Die Reaktion auf Wahrnehmungen ihrer Umgebung wiederholt sich nicht stereotyp, sondern hängt vom inneren Zustand und der damit verbundenen Entscheidung über eine Aktvität des Agenten ab. Die Kapselung von Zustand *und* Verhalten führt dazu, dass Verhalten von Agenten nur innerhalb festgelegter Schranken vorhersagbar ist. Neben der Entscheidungsautonomie existiert eine zweite Facette des Konzeptes Autonomie: die Handlungsautonomie. Charakteristisch für diese Facette ist, dass Aktivitäten von Agenten nicht nur als Reaktion auf äußere Wahrnehmungen eingeleitet werden, sondern dass Agenten spontan aktiv werden können. Das impliziert aber nicht, dass das Verhalten von Agenten zufällig ist. Das spontane Verhalten ist wiederum abhängig vom inneren Zustand eines Agenten.

Strukturierte Interaktion geht über einfache Kommunikation wie den Aufruf von Methoden auf Objekten hinaus. Dadurch, dass das Verhalten von Agenten flexibler und unbestimmter als das von Objekten ist, wird ein Konzept benötigt, um verschiedene Agenten so zu koordinieren, dass ein aus vielen Agenten bestehendes System (Multiagentensystem) das geforderte Verhalten aufweist. Mehrere Agenten werden durch Interaktion koor-

diniert. Diese Art der Koordination ist durch Interaktionsmuster eingeschränkt. In Kommunikationsprotokollen werden die Reihenfolge und die Art der Nachrichten zwischen den beteiligten Agenten festgelegt. Agenten kooperieren, indem sie derartigen Kommunikationsprotokollen folgen.

Die genannten Eigenschaften sind spezifisch für Agenten. Weder objektorientierte noch komponentenorientierte Methoden der Softwaretechnik berücksichtigen die Spezifika von Agenten. Die Entwicklung agentenbasierter Systeme bedarf der eigenen softwaretechnischen Unterstützung. Dieses Problem bildet den Ausgangspunkt dieser Arbeit.

1.2 Das Problem

Mit einer zunehmenden Realisierung von agentenbasierten Systemen oder der Integration von Agenten in bestehende Systeme stellt sich die Frage, wie derartige Systeme entwickelt werden sollen. Seit einiger Zeit wird diese Fragestellung unter dem Stichwort *Agent-Oriented Software Engineering* allgemein diskutiert (vgl. Wooldridge und Jennings [105, 71]). Um zu einer Lösung zu kommen, ist es notwendig, die wesentlichen Konzepte des Begriffs Agent zu berücksichtigen.

Die Probleme in der agentenorientierten Softwaretechnik sind folgende:

- Der Begriff Agent wird unterschiedlich interpretiert. Unklar ist, welche Aspekte von Agenten in Bezug auf die Softwaretechnik wichtig sind.

- In vorhandenen Ansätzen zu einer Softwaretechnik für Agenten (AOSE) werden die Anforderungen an eine Modellierungssprache bzw. ein Prozessmodell wenig diskutiert (vgl. [7, 58, 27]).

- Bezogen auf konkrete Sprachen zur Modellierung von Softwaresystemen ist zu klären, welche Agentenkonzepte unterstützt werden. Die Unified Modeling Language (UML) als aktuelle Standardmodellierungssprache ist der primäre Kandidat für diese Fragestellung.

- Die Syntax und Semantik vorhandener Sprachen zur Modellierung agentenbasierter Systeme sind unvollständig und unpräzise definiert (vgl. Kapitel 5). Um die Konzepte von Agenten umsetzen zu können, ist ein Systemmodell erforderlich. Die Semantik der Sprache muss zum Systemmodell passen, um die notwendige Eindeutigkeit der Sprache zu gewährleisten und die Überprüfung von Modellen durch Simulation und Verifikation zu ermöglichen. Der Bezug zwischen Syntax und Semantik, bzw. der Sprachstruktur ist in vorhandenen Ansätzen unklar (vgl. Kapitel 5).

- Vergleichbar mit den Problemen bei Modellierungssprachen sind auch die Syntax und Semantik von vorhandenen Prozessmodellen unzureichend definiert.

Durch Entwicklung der Modellierungssprache AML auf der Basis von UML und des Prozessmodells APM, in dem diese Sprache verwendet wird, können Systeme entwickelt werden, die flexibel, proaktiv und autonom sind und strukturierte Interaktion zwischen Systemteilen erfordern. Damit können speziell auch agentenbasierte Systeme mit angemessenen softwaretechnischen Mitteln entwickelt werden. In diesem Sinn trägt diese Arbeit dazu bei, zur Lösung der Probleme bei der agentenorientierten Softwareentwicklung beizutragen.

Ausgehend von den beschriebenen Problemen werden die Ziele dieser Arbeit gesetzt.

1.3 Ziele und Aufbau der Arbeit

Eine wesentliche Vorgabe dieser Arbeit ist, dass auf dem heutigen Standard der Softwareentwicklung aufgebaut wird. Für die Beschreibung von Modellen bildet die Unified Modeling Language (UML) den aktuellen Ausgangspunkt. Die genannten Probleme werden wie folgt behandelt:

- In Kapitel 2 wird der Begriff Agent eingegrenzt. Dazu werden die wesentlichen Eigenschaften von Agenten, wie Proaktivität, Autonomie und strukturierte Interaktion genauer gefasst. Der zweite Beitrag zur Begriffsklärung besteht in einer Abgrenzung zu Objekten und Komponenten. Der Unterschied zwischen klassischen objektorientierten oder komponentenbasierten Systemen und agentenorientierten Systemen wird in Kapitel 3 herausgearbeitet.

- Die Anforderungen an ein agentenorientiertes Modellierungsverfahren werden in Kapitel 4 auf Grundlage des präzisierten Agentenbegriffes erarbeitet.

- In Kapitel 5 wird der aktuelle Stand der agentenorientierten Softwaretechnik (AOSE) herausgearbeitet und es wird deutlich, dass die Entwicklung von Systemen mit integrierten Agentenkonzepten bisher unzureichend softwaretechnisch unterstützt wird.

- Die oben genannten Konzepte von Proaktivität, Autonomie und strukturierter Interaktion in UML sind zu identifizieren und gegebenenfalls zu integrieren. Da UML eine erweiterbare Sprache ist, wird für die Integration auf die Erweiterungsmechanismen zurückgegriffen. Im Hinblick auf die Eindeutigkeit der Modelle und die Intention, aus Modellen auch Implementierungen abzuleiten, wird die Semantik der Modelle in einem adäquaten semantischen Bereich definiert. In Kapitel 6 wird die Spracharchitektur eingeführt. Der semantische Bereich der Graphtransformation bildet die Grundlage. Ein UML-Profil legt die Syntax der neuen Sprache AML fest.

- Ein wesentliches Problem besteht darin zu überprüfen, ob ein Ziel eines Agenten mit Hilfe einer Strategie erreicht werden kann, wobei Ziele und Strategien durch Modelle gegeben sind. In Kapitel 7 wird ein Verfahren zur Überprüfung des Zusammenhangs zwischen Zielen und Strategien auf der Basis von Model Checking vorgestellt.

- In Kapitel 8 wird das neue Prozessmodell APM für die agentenorientierte Softwareentwicklung vorgestellt. Es gliedert Prozesse in Aktivitäten zur Anforderungsspezifikation, zur Analyse und zum Entwurf. Artefakte der einzelnen Prozessschritte sind Teilmodelle, die in der Modellierungssprache AML formuliert werden.
- Wie aus einem UML-basierten Modell eines agentenbasierten Systems eine Implementierung abgeleitet werden kann, wird in Kapitel 9 an dem Beispiel einer Ausführungsplattform für Agenten demonstriert.

Durch Entwicklung der Modellierungssprache AML auf der Basis von UML, des Verfahrens zur Modellüberprüfung auf der Basis von Model Checking und des Prozessmodells APM zeigt diese Arbeit, wie agentenbasierte Systeme mit angemessenen softwaretechnischen Mitteln modelliert werden können.

Chapter 2

Agent: Begriff und Eigenschaften

Fundamental und spezifisch für den Begriff des Agenten sind die Konzepte *Autonomie*, *Strukturierte Interaktion* und *Proaktivität* [105, 51, 103]. Autonomie bedeutet für Agenten, dass sie große Freiheitsgrade haben, über ihr Verhalten zu entscheiden. Aufgrund ihrer Autonomie sind Agenten wenig aneinander gekoppelt. Damit in einem Multiagentensystem Agenten koordiniert operieren, ist Interaktion zwischen den Agenten notwendig. Diese Interaktion ist komplex, falls die durch Integration der Agenten zu schaffende Funktion des Multiagentensystems nicht trivial ist. Agentenorientierte Systeme sind also dadurch geprägt, dass sie ihr funktionales Systemverhalten durch strukturierte Interaktion der Agenten erlangen. Proaktiv verhalten sich Agenten, weil sie über ausgeprägte Fähigkeiten zu spezifischen Reaktionen und zu zielgerichtetem Handeln besitzen.

Da in dieser Arbeit Verfahren zur Entwicklung agentenorientierter *Systeme* untersucht werden sollen, sind immer Multiagentensysteme und nicht nur einzelne Agenten Gegenstand der Betrachtung.

Systeme sind agentenorientiert, wenn die drei spezifischen Agentenkonzepte vorhanden sind. In diesem Kapitel wird jedes dieser Konzepte genauer untersucht. Insbesondere sollen die Möglichkeiten zu einer strengen und formalen Definition der Konzepte untersucht werden. Zur Abgrenzung und Verdeutlichung des Begriffs Agent soll zuerst gezeigt werden, dass agentenbasierte Systeme zur Klasse der reaktiven Systeme gehören.

Reaktivität benennt die Fähigkeit eines Agenten, seine Umgebung fortlaufend wahrzunehmen und auf Veränderungen in angemessener Zeit zu reagieren [51, 105]. Dabei wird der Agent seinen eigenen Zustand und den seiner Umgebung verändern. Reaktivität ist ein Konzept, das nicht nur Agenten verwenden. Vielmehr sind auch beispielsweise Objekte reaktiv, indem sie auf Methodenaufrufe mit Veränderungen des eigenen Zustands reagieren. Bei der Abarbeitung von Methoden ruft ein Objekt selbst Methoden eines anderen Objektes auf, wodurch der Zustand dieser Objekte verändert wird. Die zugreifbaren Objekte eines Objektes können als dessen Umgebung aufgefasst werden.

Reaktive Systeme ähneln funktionalen Systemen, da ebenfalls aus gegebenen Eingaben Ergebnisse berechnet werden. Ein funktionales System berechnet eine Funktion, d.h. aus einem endlichen Eingabewort wird durch eine Folge von Operationen ein Ausgabewort erzeugt. Darüberhinaus führt ein reaktives System eine *andauernde* Interaktion mit seiner

Umgebung durch. Das Verhalten eines solchen Systems ist dabei nicht nur von der aktuellen Eingabe sondern von der gesamten Historie der Eingaben abhängig [101, 102]. Ein reaktives System benötigt einen eigenen (inneren) Zustand, der als Speicher für die Historie dient. Da ein reaktives System nach seinem Start andauernd interaktiv ist, gibt es für die Länge der Eingabenhistorie keine obere Schranke. Im Unterschied dazu muss ein funktionales System nur mit endlichen Eingaben und nicht mit einer potentiell unbeschränkten Anzahl von Daten aus der Vergangenheit umgehen.

Anhand von zwei Beispielen soll der Unterschied verdeutlicht werden. Das Steuerungssystem eines Roboters ist reaktiv. Es muss laufend die Sensoren überwachen und auf Änderungen durch Betätigen von Aktuatoren reagieren. Auf der anderen Seite ist ein Compiler ein Beispiel für ein funktionales, nicht-reaktives System. Aus einer Eingabe, einem Programmtext, wird eine Ausgabe, ein ausführbares Programm, erzeugt.

Im nächsten Schritt werden die fundamentalen Agentenkonzepte genauer betrachtet.

2.1 Proaktivität

Agenten verhalten sich *proaktiv*, d.h. die Aktionen oder Reaktionen eines Agenten sind auch durch seine Ziele und die daraus abgeleiteten Strategien bestimmt [51, 105]. Ein Ziel eines Agenten ist eine Kombination eines bestimmten inneren Zustands mit einem Zustand der Umgebung des Agenten, der durch (pro-)aktives Handeln des Agenten erreicht werden soll. Eine Strategie ist eine Einschränkung der Handlungsmöglichkeiten eines Agenten auf die Handlungen, die zu einem oder mehreren Zielen führen. Agenten sollen ihre Ziele von sich aus kontinuierlich verfolgen und nicht erst dann, wenn sie von außen dazu angestoßen werden. Das Verhalten eines Agenten ist also laufend darauf gerichtet, ein Ziel zu erreichen. Der Agent besitzt deshalb während seiner gesamten Lebenszeit einen eigenen Kontrollfluss (thread of control). Multiagentensysteme bestehen aus einer Vielzahl von Agenten, die alle ihre Ziele kontinuierlich verfolgen. In solchen Systemen agieren die Agenten mithin nebenläufig.

Konkrete Ziele eines Agenten können mit bestimmten (inneren) Zuständen des Agenten identifiziert werden. Eine Strategie ist dann ein Entscheidungsverfahren über Handlungen des Agenten, die dem Erreichen der vorgegebenen Zielzustände dient. Durch die Strategie muss das Agentenverhalten nicht vollständig festgelegt sein. Eine derartige Strategie bildet eine Spezifikation für Agentenverhalten, das durch ein realisiertes agentenorientiertes System respektiert werden muss. Wie die Auswahl von mit der Strategie kompatiblen Aktionen des Agenten bewirkt wird, ist durch den Agenten gekapselt.

2.2 Autonomie

Gemäß Franklin, Graesser, Weiss und Wooldridge übt ein Agent Kontrolle über seinen eigenen Zustand und insbesondere über seine eigenen Aktionen aus [51, 103, 105]. Mit dieser Fähigkeit von Agenten, selbständig und unabhängig Entscheidungen über

auszuführende Handlungen zu treffen, wird das Konzept der *Autonomie* wirksam. Die Kapselung des Entscheidungsverfahrens über Verhalten bewirkt, dass das Verhalten eines Agenten nach außen nur eingeschränkt sichtbar ist.

Als Beispiel für das Konzept der Autonomie wird oft ein Thermostat angeführt, der die Temperatur eines Heizkörpers regelt, ohne von außen gesteuert zu sein. Als Beispiel für ein autonomes Softwaresystem sei der Softwaredämon *xbiff* unter dem Betriebssystem UNIX genannt, der eingehende E-Mails registriert und den Benutzer unter bestimmten Bedingungen über den Eingang informiert. Die Entscheidung darüber, den Benutzer zu informieren, liegt bei dem Softwaredämon und wird nicht unmittelbar von außen beeinflusst. Diese autonome Entscheidung kann mittelbar dann beeinflusst werden, wenn der Softwaredämon die Möglichkeit bietet, das Verhalten mindestens partiell zu konfigurieren.

Anhand dieser Beispiele ist zu sehen, dass autonomes Verhalten einerseits andauernde Aktivität nach sich zieht. Autonomes Handeln schließt ein, dass der Beginn von Aktivitäten von einem Agenten selbst festgelegt werden kann. Typisch ist, dass Veränderungen des inneren Zustands oder der äußeren Umgebung zu Aktivitäten führen. Aber auch das einfache Eintreten eines Zeitpunktes kann eine Aktivität auslösen. Andererseits sind die Veränderungen der Umwelt sichtbare Begleiterscheinungen der autonomen Operationen. Die Beeinflussung der Heizkörpertemperatur ist eine autonome Aktion des Thermostats.

Aus der obigen Betrachtung ergibt sich die Frage, wodurch Aktionen eines Agenten angestoßen werden. Das kann sowohl durch den Agenten selbst als auch in Reaktion auf Veränderungen der Umgebung passieren. Das Zusammenspiel der äußeren und inneren Faktoren bestimmt das Verhalten eines Agenten. Der Thermostat reagiert auf Veränderungen der Raumtemperatur. Seine Aktionen zielen darauf, die Temperatur konstant zu halten. Damit bestimmen sowohl äußere Umstände, die von Agenten weniger beeinflusst werden können, als auch der innere Zustand, der mit durch Ziele und Strategien des Agenten bestimmt ist, das Verhalten eines Agenten. Für Softwareagenten werden die äußeren Umstände, d.h. der Zustand der Umgebung, in Form von Nachrichten wahrnehmbar. Auf eingehende Nachrichten kann synchron oder asynchron reagiert werden. Asynchrone Kommunikation ist für Agenten typischer. Eingehende Nachrichten werden gepuffert und der Agent entscheidet, wann und wie sie abgearbeitet werden.

2.3 Strukturierte Interaktion

Strukturierte Interaktion geht über einfache Kommunikation, wie sie der Aufruf von Methoden auf Objekten darstellt, hinaus. Dadurch, dass das Verhalten von Agenten flexibler und unbestimmter als das von Objekten ist, wird ein Konzept benötigt, um verschiedene Agenten so zu koordinieren, dass ein aus vielen Agenten bestehendes System (Multiagentensystem) gefordertes Verhalten aufweist [103].

Koordination unter Agenten erfordert deren Interaktion. Durch Interaktion von Agenten wird allerdings deren Autonomie eingeschränkt. Konkret besteht die Einschränkung darin, dass Wahrnehmungen und Aktionen für verschiedene Agenten gekoppelt werden. Diese Kopplungen bilden für eine Kollaboration von Agenten mögliche und strukturierte

Formen des Informationsaustausches. Derartige Kombinationen von Wahrnehmungen und Aktionen können komplex sein und werden als *Protokolle* bezeichnet. Indem Agenten ihr autonomes Verhalten partiell auf Protokolle einschränken, wird eine kontrollierte Koordination zwischen Agenten möglich. Das Verhalten ist in Multiagentensystemen nicht nur in den Agenten gebündelt, sondern verteilt sich auch auf die benutzten Protokolle.

Protokolle bilden Schnittstellen von Multiagentensystemen. Falls die Protokolle eines Multiagentensystems öffentlich bekannt und benutzbar sind, dann ist das System offen und kann zur Laufzeit erweitert werden. Die Konsequenzen von Offenheit und Interaktion im Hinblick auf die Beschreibung solcher Systeme wird von Peter Wegnerdiskutiert [101, 102].

Wegners zentrale These lautet, dass offene und damit interaktive Systeme nicht vollständig algorithmisch beschreibbar sind. Als Berechnungsmodell für Algorithmen nimmt Wegner Turingmaschinen an. Turingmaschinen haben die Eigenschaft, dass sie keine Eingaben akzeptieren, während sie eine Berechnung ausführen. Die externe Welt wird während der Berechnung ausgeschlossen.

Für Wegner stellt sich das Problem, dass in offenen, interaktiven Systemen die Kommunikation über öffentliche Schnittstellen zu einem beliebigen Zeitpunkt stattfinden kann. Mit der Kommunikation sind Eingaben für Systemkomponenten verbunden. Eingaben einer Komponente können zu jeder beliebigen Zeit auftreten, auch dann, wenn die Komponente mit Berechnungen beschäftigt ist. Deshalb stellen für Wegner Turingmaschinen kein ausreichendes Modell für interaktive Systeme dar und eine vollständige, algorithmische Spezifikation ist nicht möglich.

Wegner spricht in [101, 102] Interaktionsmaschinen als Ersatz für Turingmaschinen für die Beschreibung interaktiver, offener Systeme an. Da er weitgehend auf eine Formalisierung der Interaktionsmaschinen verzichtet, kann für den weiteren Fortgang dieser Arbeit nicht auf dieses Modell zurückgegriffen werden. Stattdessen sollen im Folgenden existierende Modelle für Agenten und für verteilte Systeme untersucht und zu einem abstrakten Modell für Multiagentensysteme weiterentwickelt werden.

2.4 Beziehungen zwischen den Konzepten

Zwischen den Konzepten Proaktivität, Autonomie und strukturierter Interaktion bestehen Abhängigkeiten, die im Folgenden aufgezeigt werden sollen.

Um sich reaktiv zu verhalten, brauchen Agenten nur zum Zeitpunkt der Reaktion selbst aktiv zu sein. Außerhalb dieser Reaktion benötigt ein Agent keinen Kontrollfluss. Proaktivität erfordert im Unterschied dazu einen fortlaufenden Kontrollfluss, um den gesetzten Zielen nachzugehen. Als Konsequenz benötigen Agenten laufend einen eigenen Kontrollfluss. Sie sind andauernd aktiv. Bezogen auf die Aktivität von Agenten ergänzen sich die Konzepte der Reaktivität und Proaktivität.

Um proaktiv zu sein, muss ein Agent befähigt sein, über sein Verhalten zu entscheiden. Nur so hat er die Freiheitsgrade, sich seinem Ziel zu nähern. Damit impliziert Proaktivität die Entscheidungsautonomie von Agenten. Daneben ist ein Agent auch handlungsau-

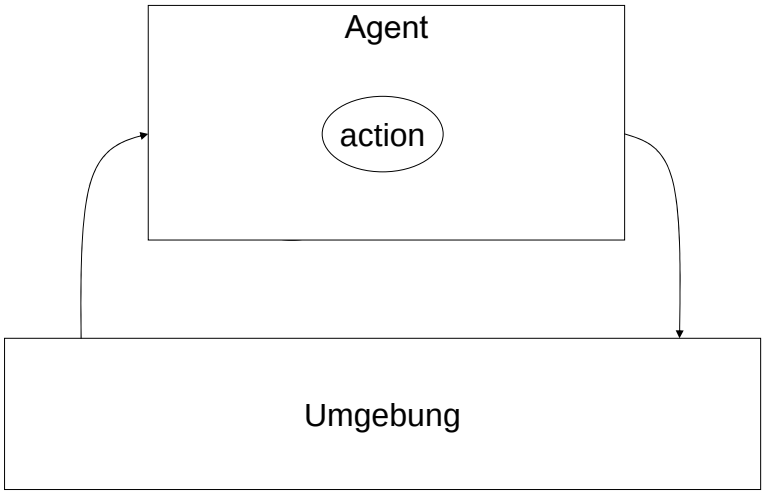

Figure 2.1: Abstrakte Architektur

tonom, d.h. er kann ohne Anstoß (Trigger) von außerhalb des Agenten aktiv sein. Der Begriff der Handlungsautonomie deckt sich mit dem der fortlaufenden Aktivität.

Die strukturierte Interaktion ist das Komplement zur Autonomie eines Agenten. Autonomie impliziert Freiheitsgrade eines Agenten, die durch strukturierte Interaktion eingeschränkt werden. In Bezug auf das abstrakte Modell wird das mögliche Verhalten in Teilen auf das für die Teilnahme an strukturierter Interaktion notwendige Verhalten eingeschränkt.

2.5 Ein abstraktes Modell für Agenten

Um die eingeführten Konzepte zu präzisieren, sollen sie an einem abstrakten Modell demonstriert werden. Ausgangspunkt der weiteren Betrachtung ist das abstrakte Modell von Wooldridge [103] für einzelne Agenten in einer Umgebung.

Abbildung 2.1 zeigt das Wooldridge-Modell eines Agenten und seiner Umgebung, mit der er durch Kommunikation in Verbindung steht. In einem Multiagentensystem mit N Agenten gehören für jeden einzelnen Agenten die $N-1$ anderen Agenten zur Umgebung. Die Umgebung eines Agenten wird als eine abzählbare Menge S von Umgebungszuständen modelliert. Zu jedem Zeitpunkt befindet sich die Umgebung in genau einem dieser Zustände. Die Handlungsmöglichkeiten eines Agenten sind durch eine abzählbare Menge A von (abstrakten) Aktionen gegeben.

$$action : S^* \rightarrow A \tag{2.1}$$

$$env : S \times A \rightarrow \mathcal{P}(S) \tag{2.2}$$

Nachdem der Agent eine neue Aktion $a_n \in A$ ausgehend von der Zustandsfolge $s_0, \ldots, s_n$ mit $s_i \in S$ berechnet hat, geht die Umgebung in einen der Folgezustände aus $env(s_n, a_n)$

über. Falls *env* nur auf einelementige Mengen abbildet, ist *env* deterministisch. Aus dem Folgezustand und den vorangegegangenen Zuständen wird mit *action* die nächste Aktion des Agenten berechnet. In diesem Systemmodell agieren Agent und Umgebung alternierend. Ein Zustandsverlauf (trace) ist eine Folge von Zuständen $s_i \in S$ und Aktionen $a_i \in A$,

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} s_n \xrightarrow{a_n} \tag{2.3}$$

die durch die folgenden beiden Bedingungen eingeschränkt ist:

$$\forall n \in \mathbb{N} : a_n = action(s_0, s_1, \ldots, s_n) \tag{2.4}$$

und

$$\forall n \in \mathbb{N} - \{0\} : s_n \in env(s_{n-1}, a_{n-1}) \tag{2.5}$$

Das charakteristische Verhalten eines Agenten in einer Umgebung ist die Menge aller möglichen Zustandsverläufe. Agenten zeichnen dadurch aus, dass sie in der Regel unbeschränkt lange aktiv sind. In diesem Fall sind die Zustandsverläufe unendliche Folgen.

Problematisch am Modell von Wooldridge ist das Verhältnis zwischen Agent und Umgebung. Die in Gleichung 2.4 formulierte Bedingung besagt, dass der Agent eine Aktion a_n im allgemeinen erst berechnen kann, wenn die Umgebung einen Zustand s_n erreicht hat. Mit der zweiten Bedingung in Gleichung 2.5 wird in diesem Modell vorausgesetzt, dass ein Folgezustand s_n erst dann bestimmt werden kann, wenn der Agent seine nachfolgende Aktion a_{n-1} berechnet hat. Durch die Bedingungen sind die Berechnung von Folgeaktionen des Agenten und die Bestimmung von Folgezuständen der Umgebung eng, d.h. synchron gekoppelt.

In einem Multiagentensystem gehören zur Umgebung ebenfalls Agenten, die autonom und insbesondere nebenläufig agieren. Das hat zur Folge, dass auch Zustandsübergänge stattfinden können, die nicht von Aktionen des betrachteten Agenten abhängen und die asynchron zu diesem erfolgen können. Für ein Multiagentensystemmodell muss die synchrone Kopplung zwischen Agent und Umgebung aufgegeben werden. Das Modell von Wooldridge ist damit für ein Multiagentensystem ungeeignet und muss durch ein Modell ersetzt werden, in dem die Kopplung der Agenten und die Interaktion mit der sonstigen Umgebung genauer beschrieben wird. Zuvor soll allerdings das Modell einzelner Agenten verfeinert werden.

Die abstrakte Agentenarchitektur soll anhand der bisherigen Begriffsbildung für Agenten verfeinert werden. Agenten nehmen ihre Umgebung wahr und haben einen inneren Zustand. Diese Aspekte werden, Wooldridge [105] folgend, in verfeinerte Modelle aufgenommen. Die erste Entscheidung besteht nach Wooldridge darin, die Wahrnehmung der Umgebung von der Entscheidung über Aktionen des Agenten zu trennen, siehe Abbildung 2.2. Ein Subsystem zur Wahrnehmung rezipiert Zustände der äußeren Umgebung und bildet zugehörige Wahrnehmungsinhalte, sogenannte Perzepte. Durch die Wahrnehmung werden äußere Zustände im Hinblick auf gewählte Merkmale klassifiziert. So ist die Menge der Perzepte in der Regel viel kleiner als die Menge der Zustände der

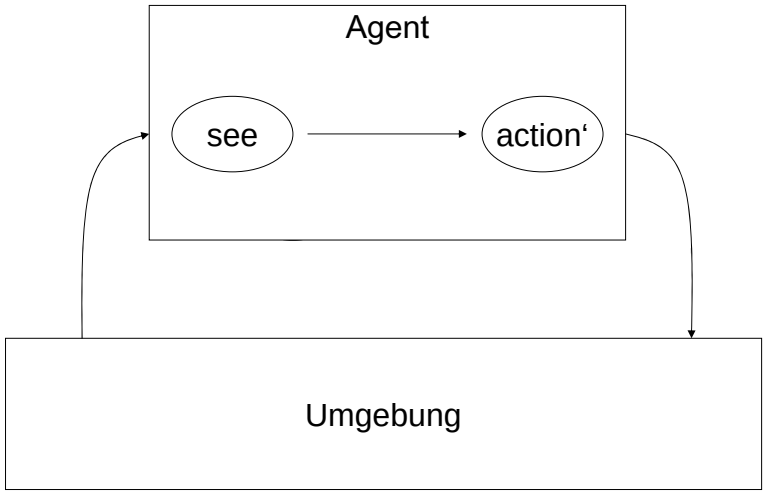

Figure 2.2: Perzeption bei Agenten

äußeren Umgebung. Durch die Wahrnehmung wird festgelegt, auf welche Eingaben überhaupt reagiert wird oder auf welche Eingaben in gleicher Weise reagiert wird. Dadurch wird insbesondere auch eine Vereinfachung der Abbildung *action* erreicht.

Die Wahrnehmungseindrücke eines Agenten sind Elemente der Menge P von Perzepten. Die Wechselwirkung mit der Umgebung wird für jeden Agenten durch die Abbildung *see* beschrieben.

$$see : S \rightarrow P \tag{2.6}$$

Mit *see* wird die Abbildung see^* wie folgt definiert:

$$see^* : S^* \rightarrow P^* \tag{2.7}$$

$$(s_1, \ldots, s_n) \mapsto (see(s_1), \ldots, see(s_n)) \tag{2.8}$$

Das Verhalten des Agenten wird durch die Abbildung $action'$ auf Basis der Perzepte beschrieben:

$$action' : P^* \rightarrow A \tag{2.9}$$

Durch Verknüpfung der Abbildungen see^* und $action'$ entsteht die Abbildung $action = action' \circ see^*$. Wenn für *see* die identische Abbildung angenommen wird, wird deutlich, dass die Zerlegung von *action* in die Abbildungen *see* und $action'$ keine Einschränkung des Systemmodells darstellt.

Die Abbildung *see* klassifiziert Umgebungszustände als Perzepte. So können verschiedene Umgebungszustände zum gleichen Perzept führen. Dadurch kann die Größe des Zustandsraums P^*, auf dem das Verhalten des Agenten zu definieren ist, reduziert werden, und die Definition der Abbildung $action'$ vereinfacht sich.

In diesem noch sehr allgemeinen Modell ist die interne Struktur von Agenten durch die Abbildungen *see* und $action'$ repräsentiert. Das Verhalten des Gesamtsystems

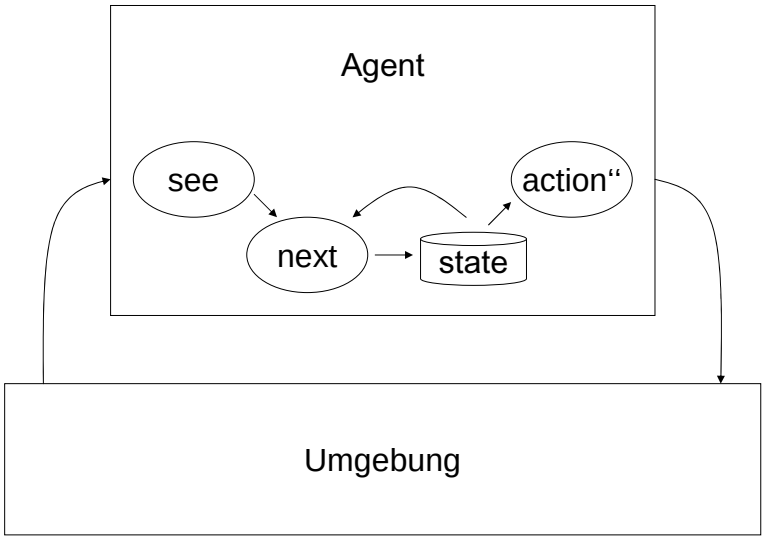

Figure 2.3: Architektur von Agenten mit innerem Zustand

ist, insbesondere im Hinblick auf die Realisierung eines Agentensystems, in den Agenten lokalisiert. Reaktivität von Agenten erfordert, dass lokales Agentenverhalten von einem lokalen, inneren Zustand abhängt. Aufgrund möglicher verschiedener innerer Zustände verhalten sich Agenten bei zweimaliger gleicher Wahrnehmung (durch *see*) unterschiedlich. Agenten haben ein “Gedächtnis”.

Die Abhängigkeit der Aktionen eines Agenten von der Historie der Umgebungszustände ist durch die inneren Zustände in der Regel indirekt gegeben: Die Wahrnehmungen aus P führen zu Veränderungen des inneren Zustands eines Agenten, und Aktionen werden aus dem inneren Zustand berechnet. Die Abbildung 2.3 illustriert diesen Zusammenhang. Die Folge der Umgebungszustände reflektiert sich damit in der Folge der inneren Zustände.

Die inneren Zustände können so definiert werden, dass sie (auch) alle vorangegangenen Wahrnehmungen speichern. Bei jedem Zustandsübergang kann die Folge der durchlaufenen Zustände um den verlassenen Zustand verlängert und im betretenen Zustand gespeichert werden. Die Aktionen eines Agenten hängen vom inneren Zustand des Agenten ab. Mit dieser Festlegung von Zuständen sind die Aktionen auch von der Historie der inneren Zustände des Agenten abhängig. Damit wird deutlich, dass durch die Einführung innerer Zustände das allgemeine Modell, in dem Aktionen aus Umgebungszustandsfolgen berechnet werden, nicht eingeschränkt wird.

Die Menge der inneren Zustände wird mit I bezeichnet. Die Abbildung *see* bleibt in dem verfeinerten Modell unverändert. Die Aktionen resultieren aus dem inneren Zustand, wodurch sich *action* ändert. Zusätzlich ist die Abbildung *next* erforderlich, durch die die Übergangsfunktion eines Automaten definiert wird:

$$action'' : I \rightarrow A \tag{2.10}$$

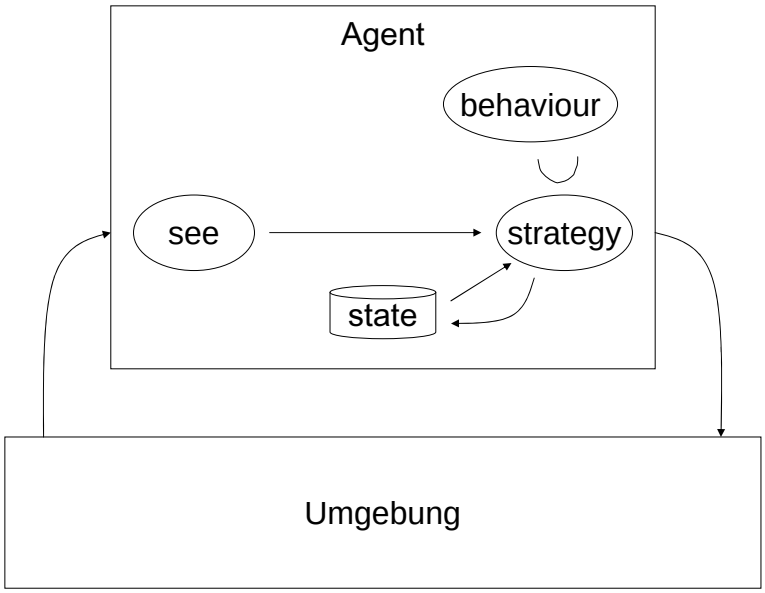

Figure 2.4: Agenten mit Strategie

$$next : I \times P \to I \tag{2.11}$$

Autonomie und Proaktivität. Durch die bisherigen Verfeinerungen lässt sich reaktives Verhalten von Agenten abstrakt modellieren. Autonomie und Proaktivität von Agenten sind bisher unzureichend repräsentiert. Diese Merkmale sollen in einem verfeinerten Modell repräsentiert werden.
Autonomie zeigt sich darin, dass Agenten selbst entscheiden, wie sie auf Wahrnehmungen reagieren. Die Entscheidungsautonomie zeigt sich darin, dass die Auswahl der Aktionen und Folgezustände lokal beim Agenten erfolgt. Mögliche Entscheidungen eines Agenten sind also Mengen von Paaren von Aktionen und Folgezuständen, die aus einer Wahrnehmung und dem aktuellen Zustand resultieren (siehe Abb. 2.4). Mathematisch wird das durch die Abbildung *behaviour* erfasst:

$$behaviour : I \times P \to \mathcal{P}(I \times A) \tag{2.12}$$

die die Abbildungen *next* und *action*″ integriert und verallgemeinert. Soll das Verhalten deterministisch sein, d.h. die Bilder der Abbildung *behaviour* sind nur einelementig, dann kann *next* als Projektion des jeweils einen Elementes der Bilder von *behaviour* auf die erste Komponente aufgefasst werden. Die Abbildung *action*″ entsteht entsprechend durch Projektion auf die zweite Komponente.

Die Entscheidungsautonomie wird besonders deutlich, wenn für einen inneren Zustand $i \in I$ alle Wahrnehmungen $p \in P$ zum gleichen Verhalten führen, d.h. für alle Paare $p_m, p_n \in P$ gilt $behaviour(i, p_m) = behaviour(i, p_n)$. In diesem Fall ist die Reaktion des Agenten unabhängig vom wahrgenommenen Perzept.

Neben der Entscheidungsautonomie besitzt ein Agent auch Handlungsautonomie, d.h. er agiert unabhängig von *äußeren* Wahrnehmungen, so dass diese nicht als Auslöser für Aktionen erforderlich sind. Dieses Merkmal lässt sich in das Modell integrieren, indem Aktionen von Agenten als eigene Perzepte, d.h. als interne Aktionen wirken. Diese internen Aktionen lösen weitere, unter Umständen wieder interne Aktionen aus.

Agenten sind proaktiv, weil sie eine Strategie umsetzen, mit der sie gegebene Ziele erreichen sollen. Eine Strategie bestimmt die Aktionen eines Agenten unter Berücksichtigung der eigenen Wahrnehmungen und des inneren Zustands, um einen oder eine Folge von Zielzuständen zu erreichen. Die Abbildung *strategy* ist so konstruiert, dass sie die genannten Eigenschaften einer Strategie repräsentiert:

$$strategy : I \times P \to \mathcal{P}(I \times A) \tag{2.13}$$

Strategien müssen das mögliche Verhalten eines Agenten berücksichtigen. Der Definitionsbereich von *strategy* muss in demjenigen von *behaviour* enthalten sein. Die Bilder von *strategy* sind ebenfalls Bilder von *behaviour*. Zusammengefasst muss folgende Konsistenzbedingung erfüllt sein:

$$strategy \subseteq behaviour \tag{2.14}$$

Eine Strategie eines Agenten ist deterministisch, wenn die Bilder der Abbildung *strategy* einelementig sind. Dann ist das Modell direkt mit den zuvor entwickelten Modellen verträglich. Im anderen Fall sind Folgezustände und Aktionen nicht eindeutig bestimmt, und das Agentenverhalten ist im abstrakten Modell nichtdeterministisch. Dieses Modell geht über das bisherige deterministische hinaus.

Strukturierte Interaktion. Für die strukturierte Interaktion mit anderen Agenten sind Protokolle notwendig, die mögliche Abläufe von Interaktionen beschreiben. Protokolle sind für Agenten Einschränkungen ihres autonomen Verhaltens. Im abstrakten Modell lassen sich Protokolle als Restriktionen der Abbildung *behaviour* auffassen. Für Zustände, von denen aus ein Protokoll abgearbeitet werden soll, muss das durch *behaviour* beschriebene Verhalten eines Agenten dem Verhalten des Protokolls solange entsprechen, bis das Protokoll abgearbeitet ist.

In den bisherigen Modellen wurde genau *ein* Agent berücksichtigt, der mit einer abstrakt durch eine Zustandsmenge S spezifizierten Umgebung kommuniziert. Da das Ziel darin besteht, Multiagentensysteme zu modellieren, werden die Agenten jetzt explizit in das abstrakte Modell eingefügt. Für jeden einzelnen Agenten gehören die anderen Agenten des Multiagentensystems also nicht (mehr) zur Umgebung. Damit ändert sich der Begriff Umgebung. Für ein 2-Agentensystem ergibt sich die Architektur in Abbildung 2.5.

Die Struktur einzelner Agenten bleibt erhalten. Agenten waren allerdings bisher in der Lage, jeden Zustand der Umgebung wahrzunehmen, da die Abbildung *see* nach Wooldridge total ist. Existieren mehrere Agenten, so wären nach dem bisherigen Umgebungsbegriff für einen betrachteten Agenten die inneren Zustände der anderen Agenten Teil des Umgebungszustandes. Demnach könnte jeder Agent die inneren Zustände anderer

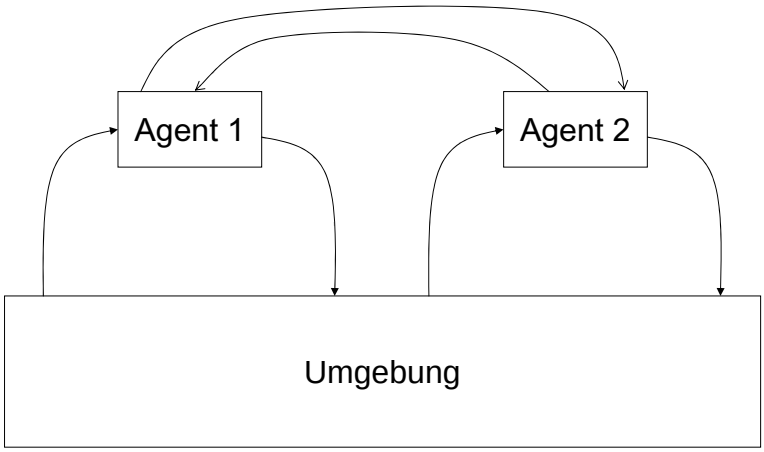

Figure 2.5: Abstrakte Architektur eines Systems mit zwei Agenten

Agenten wahrnehmen. Diese mangelnde Kapselung der inneren Zustände widerspricht dem Konzept, dass die Umgebung von einem Agenten nur dessen Aktionen wahrnimmt.

Der geänderte Umgebungsbegriff wird dadurch formalisiert, dass die Menge der Umgebungszustände S für Multiagentensysteme ab sofort alle inneren Zustände der Agenten ausschließt. Die Wahrnehmung eines jeden Agenten Ag_i bezieht sich dann auf die Umgebungszustände aus S und die Aktionen aus A_j der anderen Agenten Ag_j für $j \neq i$. Die Definitionsbereiche S_i der Abbildungen see_i jedes Agenten $Ag_i, i \in \{1 \dots n\}$ sind entsprechend modifiziert: $S_i = S \cup \bigcup_{j \in \{1 \dots n\} - \{i\}} A_j$.

Nachdem das abstrakte Modell formuliert ist, wird jetzt der Zusammenhang mit etablierten Agentenmodellen diskutiert.

2.6 Verträglichkeit des abstrakten Modells

Bisher wurde gezeigt, dass die Konzepte der Autonomie und strukturierten Interaktion durch das abstrakte Modell realisiert wurden. Proaktivität ist im abstrakten Modell in der Abbildung *strategy* repräsentiert. Im folgenden soll anhand der wichtigsten beiden speziellen Architekturen gezeigt werden, dass diese etablierten Architekturen sich unter die rein aus Konzepten deduzierte abstrakte Architektur subsumieren lassen.

Im nächsten Abschnitt wird auf die Belief-Desire-Intention Architektur (BDI-Architektur) als ein Beispiel einer speziellen Architektur eingegangen und der Zusammenhang zur eingeführten abstrakten Architektur hergestellt. Wichtige andere Beispiele für bisher entwickelte spezielle Architekturen sind geschichtete Architekturen, reaktive Architekturen und weitere logikbasierte Architekturen (vgl. [72]).

Auf die Frage der Ausdrucksmächtigkeit der speziellen Architekturen, d.h. ob die abstrakte Architektur durch eine spezielle Architektur simuliert werden kann, wird nicht eingegangen. Die Aufgabe dieses Abschnittes besteht darin zu zeigen, dass die eingeführte

abstrakte Architektur tatsächlich eine geeignete Abstraktion etablierter Architekturen darstellt.

2.6.1 Die Belief-Desire-Intention Architektur

Die Belief-Desire-Intention Architektur ist bedeutend, da auf ihrer Basis bereits Anwendungen entwickelt worden sind, die praktisch eingesetzt wurden und werden. Für diese Architektur gibt es Bibliotheken zur vereinfachten Entwicklung von agentenorientierten Systemen. Für die Programmiersprache JAVA existiert beispielsweise JACK [64].

Die Architektur basiert auf den Konzepten Belief, Desire und Intention. Die Annahmen (Beliefs) der Menge Bel eines Agenten dienen der Bewertung der Wahrnehmungen des Agenten. Nimmt der Agent neue Perzepte wahr, dann werden die Annahmen des Agenten revidiert, und die Abbildung brf (beliefs revision function) bildet aktualisierte Beliefs:

$$brf : \mathcal{P}(Bel) \times P \rightarrow \mathcal{P}(Bel) \tag{2.15}$$

Die Menge der Intentionen (Intentions) Int bestimmt die Handlungen eines Agenten. Intentionen repräsentieren die konkreten Absichten eines Agenten und sind durch die Abbildung $execute$ unmittelbar mit den Aktionen der Menge A verknüpft:

$$execute : \mathcal{P}(Int) \rightarrow A \tag{2.16}$$

Wesentlich für die BDI-Architektur ist die Verknüpfung zwischen Annahmen und Intentionen. Die revidierten Annahmen und Intentionen legen die Optionen (Desires) für neue Intentionen eines Agenten fest:

$$options : \mathcal{P}(Bel) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Des) \tag{2.17}$$

Auf der Basis der revidierten Annahmen, der aktuellen Intentionen und Optionen wird über die nächsten Intentionen des Agenten entschieden. Neue Intentionen werden mit Hilfe der Abbildung $filter$ bestimmt:

$$filter : \mathcal{P}(Bel) \times \mathcal{P}(Des) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Int) \tag{2.18}$$

Für diese Abbildung wird verlangt, dass die neuen Intentionen entweder alte Intentionen oder mit $option$ bestimmte Wünsche sind. Aus den mit $filter$ berechneten Intentionen wird mit $execute$ eine Agentenaktion berechnet.

Durch Komposition der eingeführten Abbildungen kann die Abbildung $strategy$ gebildet werden:

$$strategy((B, I), p) = ((brf(B, I), I'), execute(I')) \tag{2.19}$$

mit

$$I' = filter(brf(B, I), options(brf(B, I), I), I) \tag{2.20}$$

Die Abbildung *strategy* in 2.19 hat die gleiche Signatur wie die Abbildung *strategy* in Abschnitt 2.5. Die erste Ergebniskomponente der *strategy*-Abbildung in 2.19 enthält den Folgezustand und die zweite Komponente den externen Effekt des Ausführungsschrittes. Die Bedeutung der Ergebniskomponenten entspricht derjenigen der abstrakten *strategy*-Abbildung. Das BDI-Modell ist damit eine Spezialisierung des in Abschnitt 2.5 eingeführten abstrakten Modell.

2.6.2 Logik-basierte Architekturen

Im Rahmen logik-basierter Architekturen treffen Agenten Entscheidungen durch logische Deduktion von Aktionen. Zu diesem Zweck arbeitet jeder Agent als Theorembeweiser. Zu klären ist, wie sich dieser Ansatz in das abstrakte Modell einbetten lässt. Dazu wird die *strategy*-Abbildung jetzt so definiert, dass sie den Vorgang des Theorembeweisens kapselt.

Der interne Zustand eines Agenten ist durch eine Menge d von Aussagen in Form logischer Formeln der Prädikatenlogik erster Stufe gegeben (vgl. Weiss [103], Seite 43). Diese Aussagen bilden das Wissen, das ein Agent von seiner Umgebung hat. Durch die Wahrnehmung der äußeren Umgebung ändert sich der interne Zustand, d.h. die entsprechende Menge d von Aussagen. Die Menge I aller internen Zustände ist allgemein die Menge aller in Prädikatenlogik formulierbarer Mengen von Aussagen. Im konkreten Anwendungsfall ist I passend eingeschränkt.

Seien nach wie vor S die Menge der Umgebungszustände, P die Menge der Perzepte und A die Menge der Aktionen. Bezogen auf das abstrakte Modell ändert sich an der Wahrnehmung und an den Zustandsübergängen nichts, d.h. die Wahrnehmungsabbildung bleibt unverändert: $see : S \rightarrow P$. Die Zustandsübergangsfunktion zum Überführen einer Menge von Aussagen in eine durch die Wahrnehmung beeinflusste neue Menge von Aussagen behält die Form $next : P \times I \rightarrow I$.

Die Abbildung $strategy : I \rightarrow A$ ist für die Auswahl einer Aktion zuständig. Sie wird mit Hilfe logischer Deduktion auf der Basis der Aussagen in d gebildet. Für eine Menge ρ von Ableitungsregeln, einen Zustand d, d.h. eine Menge von Aussagen in Form logischer Formeln und eine Formel ϕ arbeitet ein Agent als Theorembeweiser. Er versucht, ϕ aus d auf der Basis von ρ abzuleiten. Formal schreibt man $d \vdash_\rho \phi$, falls die Ableitung gelingt.

Das Ergebnis einer Strategie soll eine Aktion sein. Für jede Aktion $a \in A$ wird deshalb die Formel $Do(a)$ eingeführt. Leitet ein Agent, ausgehend von einem Zustand d eine solche Formel ab, dann bildet die Aktion a das Ergebnis der *strategy*-Abbildung. Formal heißt das: $strategy(d) = a$, falls $d \vdash_\rho Do(a)$. Falls keine Formel $Do(a)$ abgeleitet werden kann, ist *strategy* nicht definiert. Da aus d u.U. verschiedene Formeln der Form $Do(a)$ abgeleitet werden können, hängt das Ergebnis vom konkreten Vorgehen des Theorembeweisers ab (vgl. Weiss [103], Seite 44).

Ein Beispiel für ein logik-basiertes System ist Concurrent METATEM [49].

Damit ist gezeigt, dass logik-basiertes Agentenverhalten sich in das in Abschnitt 2.5 eingeführte abstrakte Modell als ein weiterer Spezialfall einfügt.

2.6.3 Reaktionsarchitekturen

Entgegen dem Ansatz der logik-basierten Architekturen verzichten Reaktionsarchitekturen auf eine symbolische Repräsentation in der Zustandsbeschreibung. Das wesentliche Konzept dieser Architekturen besteht in der engen Kopplung von Wahrnehmung und Aktionen. Eine gegebene Wahrnehmung führt ohne aufwendige Transformationen, und insbesondere ohne eine symbolverarbeitende Schicht, direkt zu einer Aktion. Dieser Zusammenhang wird oft in sogenannten situation-action-Regeln (SA-Regeln) erfasst.

Eine spezielle reaktive Architektur ist die Subsumtionsarchitektur von Brooks [15]. Die SA-Regeln werden Schichten zugeordnet, die mit Prioritäten versehen sind, welche mit höheren Schichten abnehmen. Anhand der Prioritäten wird zwischen konkurrierenden Regeln ausgewählt. Der Effekt dieser Strukturierung besteht darin, dass Regeln in niederen Schichten vor solchen in höheren Schichten bevorzugt werden. Von Brooks wurde diese Architektur zur Robotersteuerung eingesetzt [14]. In niederen Schichten befinden sich beispielsweise in diesem Fall priorisierte Regeln, um Hindernisse zu umsteuern.

Die Wahrnehmung und die Berechnung von Zustandsübergängen ist gegenüber dem abstrakten Modell unverändert. Die *strategy*-Abbildung berücksichtigt bei der Berechnung von Aktionen die Prioritäten der verschiedenen Schichten. (vgl. Abbildung *action* in Weiss [103], Seite 50).

Die Subsumtionsarchitektur von Brooks fügt sich damit auch in das in Abschnitt 2.5 eingeführte abstrakte Modell als ein weiterer Spezialfall ein. Insgesamt ist damit gezeigt, dass das abstrakte Modell tatsächlich von verschiedenen wichtigen Agentenarchitekturen korrekt *abstrahiert*. Dieser wichtigen Bedeutung des abstrakten Modells steht der Nachteil entgegen, dass das Modell für die Beschreibung spezifischer Systeme zu aufwändig ist. So muss der völlig unspezifische Zustand immer wieder so verfeinert werden, dass zwischen extern sichtbaren Nachrichten und dem inneren Zustand von Agenten unterschieden wird. Dieses Problem tritt bei dem nachfolgend eingeführten Modell agentenbasierter Systeme auf der Basis von I/O-Automaten nicht auf.

2.7 Modellierung von Agenten durch I/O-Automaten

In Kapitel 6 soll gezeigt werden, dass das auf der dort vorgestellten Modellierungssprache basierende Systemmodell mit dem abstrakten Modell verträglich ist. Dadurch wird die Konsistenz der verschiedenen Beschreibungsebenen gesichert.

Problematisch ist die hohe Abstraktion des abstrakten Modells. Notwendig ist stattdessen ein detaillierter strukturiertes Modell agentenbasierter Systeme, um den (asynchronen) Nachrichtenaustausch zwischen Agenten, den inneren Zustand und die Kopplung von Agenten leichter als mit dem eingeführten abstrakten Modell beschreiben zu können.

Im Bereich der verteilten Systeme werden I/O-Automaten dazu verwendet, das Verhalten einzelner Systemkomponenten zu beschreiben [25]. In solchen Systemen ist das Senden und Empfangen von Nachrichten und das Erfassen der inneren Zustandsübergänge wichtig

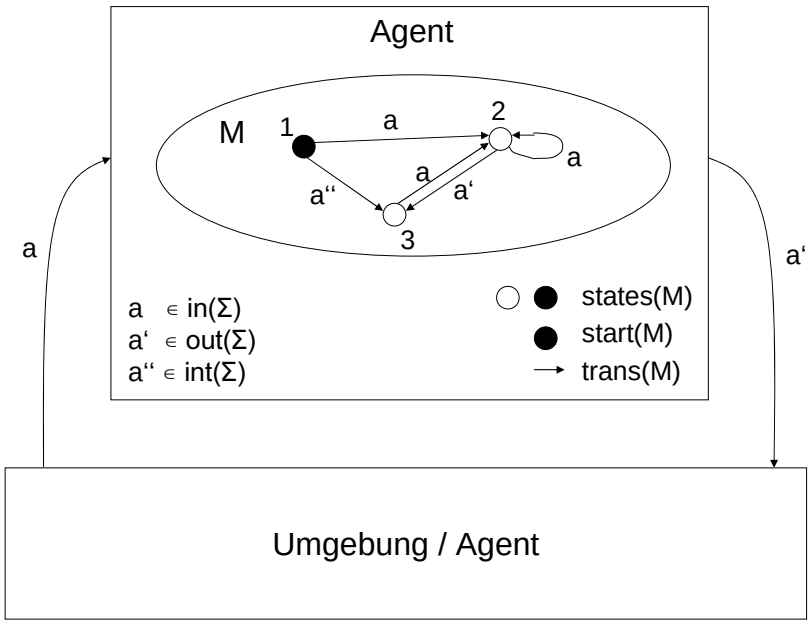

Figure 2.6: I/O-Automat M beschreibt Agentenverhalten

und läßt sich durch I/O-Automaten gut beschreiben. Die Kopplung der Komponenten zum Gesamtsystem wird durch Produktbildung der Komponentenautomaten beschrieben. Die abstrakte Architektur in Abbildung 2.5 bildet insbesondere auch ein verteiltes System, da Agenten nebenläufig agieren und deshalb auch auf verschiedene Prozessoren und in einem Netzwerk verteilt werden können. Insgesamt sind I/O-Automaten ein geeignetes Beschreibungsmittel für agentenbasierte Systeme.

Für verteilte Systeme existieren neben I/O-Automaten [25] andere etablierte mathematische Modelle wie Petri-Netze [88], oder Prozessalgebren [9]. Im Zusammenhang mit dem abstrakten Modell sind I/O-Automaten hier am besten geeignet.

2.7.1 I/O-Automaten

Nach Lynch [25] hat ein I/O-Automat M eine Signatur Σ, die Eingabe- und Ausgabeaktionen sowie interne Aktionen enthält. Mit den paarweise disjunkten Mengen $in(\Sigma)$, $out(\Sigma)$ und $int(\Sigma)$ gilt $\Sigma = (in(\Sigma), out(\Sigma), int(\Sigma))$. Die unterschiedliche Rolle der Aktionen wird bedeutsam, wenn I/O-Automaten gekoppelt werden (siehe Seite 24) $acts(\Sigma)$ ist die Vereinigung aller Aktionen von Σ, d.h. $acts(\Sigma) = in(\Sigma) \cup int(\Sigma) \cup out(\Sigma)$. Extern sichtbare Aktionen sind in der Menge $ext(\Sigma) = in(\Sigma) \cup out(\Sigma)$ vereinigt. Lokal kontrollierte Aktionen sind solche in $local(\Sigma) = out(\Sigma) \cup int(\Sigma)$.

Ein I/O-Automat M ist ein 5-Tupel, das aus den folgenden Komponenten besteht (vgl. Abb. 2.6):

- $sig(M) = \Sigma$, eine Signatur

- $states(M)$, eine (nicht notwendigerweise endliche) Menge von Zuständen
- $start(M)$, die Anfangszustände, die eine nichtleere Teilmenge von $states(M)$ bilden
- $trans(M)$, eine *Zustandsübergangsrelation*, für die $trans(M) \subset states(M) \times acts(sig(M)) \times states(M)$ gilt; verlangt wird, dass für alle $s \in states(M)$ und für alle $a \in in(sig(M))$ ein Tripel $(s, a, s') \in trans(M)$ existiert
- $tasks(M)$, eine *tasks partition*, die eine Äquivalenzrelation auf $local(sig(M))$ mit einer höchstens abzählbaren Menge von Äquivalenzklassen ist

Für die Zustandsübergangsrelation $trans(M)$ ist gefordert, dass jede Eingabe eines Automaten in jedem Zustand verarbeitet wird (input enabling). Dass keine Eingabe "geblockt" werden kann, ist eine fundamentale Annahme dieses Automatenmodells. Sie ist im Zusammenhang mit der Komposition von I/O-Automaten wichtig.

Die fünfte Komponente der Definition von I/O-Automaten ist als abstrakte Beschreibung des Begriffs Kontrollfluss (thread of control) zu verstehen. Für die Äquivalenzklassen (tasks) lassen sich Fairness-Bedingungen definieren, die garantieren, dass jede Klasse bei der Ausführung berücksichtigt wird. Fairness-Bedingungen werden in dieser Arbeit nicht betrachtet. Deshalb wird die fünfte Komponente obiger Definition im weiteren Verlauf nicht mehr berücksichtigt.

Die möglichen Ausführungen eines Automaten sind alternierende, nicht notwendigerweise endliche Folgen $\alpha_n = (s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n)$ von Zuständen und Aktionen, so dass jedes Tripel $(s_k, a_{k+1}, s_{k+1}) \in trans(M)$ mit $k \geq 0$, d.h. eine Transition von M ist. Ist s_0 ein Startzustand, so ist die Folge α_n eine *Ausführung* des Automaten M, sonst ein *Ausführungsfragment*.

In vielen Fällen interessiert nur das beobachtbare Verhalten eines Automaten. Für eine Ausführung α eines Automaten M ist $trace(\alpha)$ die Teilfolge der externen Aktionen von α. Die Menge solcher Teilfolgen aller Ausführungen eines Automaten M heißt $traces(M)$.

Der I/O-Automat in Abbildung 2.6 illustriert das Verhalten: die Aktionen führen zu Zustandsübergängen, wobei für jede Eingabeaktion (hier a) und jeden Zustand ein Übergang definiert ist. Ein Teil eines Ablaufs sieht beispielsweise folgendermaßen aus: Ist der Automat im Anfangszustand 1, dann geht er mit der Eingabeaktion a in den einzig möglichen Folgezustand 2 über. Von dort geht er mit der Ausgabeaktion in den Zustand 3 über, etc. Die Rolle von Ein- und Ausgabeaktionen wird bei der Komposition von I/O-Automaten deutlich.

Komposition von I/O-Automaten. Das Verhalten der Bestandteile eines verteilten Systems wird durch I/O-Automaten beschrieben. Die Integration der Bestandteile zu einem Gesamtsystem erfolgt durch Komposition der I/O-Automaten. Das wesentliche Konzept der Kopplung von I/O-Automaten besteht darin, die Ausgabeaktion a eines Automaten mit den gleichnamigen Eingabeaktionen a anderer Automaten zu identifizieren. Wenn also der eine Automat einen Zustandsübergang mit der Ausgabeaktion a vollführt,

führen alle anderen Automaten mit Eingabeaktion a synchron einen Zustandsübergang durch.

Dieses Kopplungskonzept hat Konsequenzen hinsichtlich der Konstruktion des zusammengesetzten Automaten. Damit interne Aktionen tatsächlich nur zu lokalen Zustandsübergängen eines Automaten führen, wird verlangt, dass für eine Menge zu komponierender Automaten die internen Aktionen jedes Automaten disjunkt zu den Aktionen aller anderen Automaten sind. Weiterhin verlangt Lynch, dass alle Ausgabeaktionen voneinander verschieden sind, damit jede Aktion von genau einer Komponente kontrolliert wird. Eine wichtige Konsequenz dieser Forderung ist, dass Ausgabeaktionen nur mit den Eingabeaktionen anderer Automaten übereinstimmen können. Formal lauten die Verträglichkeitsbedingungen für eine endliche Menge von Signaturen Σ_i, Σ_j mit $i, j \in N$ ($|N|$ endlich) mit $i \neq j$:

- $int(\Sigma_i) \cap acts(\Sigma_j) = \emptyset$
- $out(\Sigma_i) \cap out(\Sigma_j) = \emptyset$
- Keine Aktion ist in unendlich vielen Mengen $acts(\Sigma_i)$ enthalten.

Die Eingabeaktionen in $in(\Sigma)$ des Produktautomaten sollen aus den Eingabeaktionen der Komponenten bestehen, die nicht auch Ausgabeaktionen sind. Dadurch ist sicher gestellt, dass Aktionen, die der internen Kommunikation zwischen den Komponentenautomaten dienen (können), nicht als Eingabeaktionen des Produktautomaten verwendet werden. Die Aktionen aus der Menge $hidden(\Sigma) = \bigcup_{i \in N} in(\Sigma_i) \cap \bigcup_{i \in N} out(\Sigma_i)$ sollen keine Eingabeaktionen des Produktautomaten sein. Sie dienen der Kommunikation zwischen den Komponentenautomaten und sind deshalb als interne Aktionen des Produktautomaten aufzufassen.

Für eine endliche Familie $(M_i)_{i \in N}$ von I/O-Automaten, deren Signaturen die genannten Verträglichkeitsbedingungen erfüllen, ist der zusammengesetzte I/O-Automat M auf der Grundlage der angeführten Argumente wie folgt definiert. Die Signatur Σ ist das Produkt $\Pi_{i \in N} \Sigma_i$ von Signaturen Σ_i, wobei gilt:

- $out(\Sigma) = \bigcup_{i \in N} out(\Sigma_i)$
- $int(\Sigma) = \bigcup_{i \in N} int(\Sigma_i) \cup hidden(\Sigma)$
- $in(\Sigma) = \bigcup_{i \in N} in(\Sigma_i) - hidden(\Sigma)$

Die Menge der Zustände $states(M)$ ist das Produkt $\Pi_{i \in I} states(M_i)$ der Komponentenzustandsmengen $states(M_i)$, wobei die Menge der Startzustände $start(M)$ das Produkt $\Pi_{i \in N} start(M_i)$ der Komponentenstartzustände $start(M_i)$ ist. Die Übergangsrelation $trans(M)$ ist die Menge der Tripel (s, a, s'), so dass für alle $i \in N$, falls $a \in acts(M_i)$, dann $(s_i, a, s_i') \in trans(M_i)$, sonst $s_i = s_i'$.

Für eine Aktion wird also in jedem Komponentenautomaten geschaltet, der die Aktion akzeptiert. Die Rolle von Ein- und Ausgabeaktionen ist damit dadurch definiert, dass das Schalten eines Komponentenautomaten für eine Ausgabeaktion mit dem Schalten anderer Komponentenautomaten mit gleich lautenden Eingabeaktionen korrespondiert.

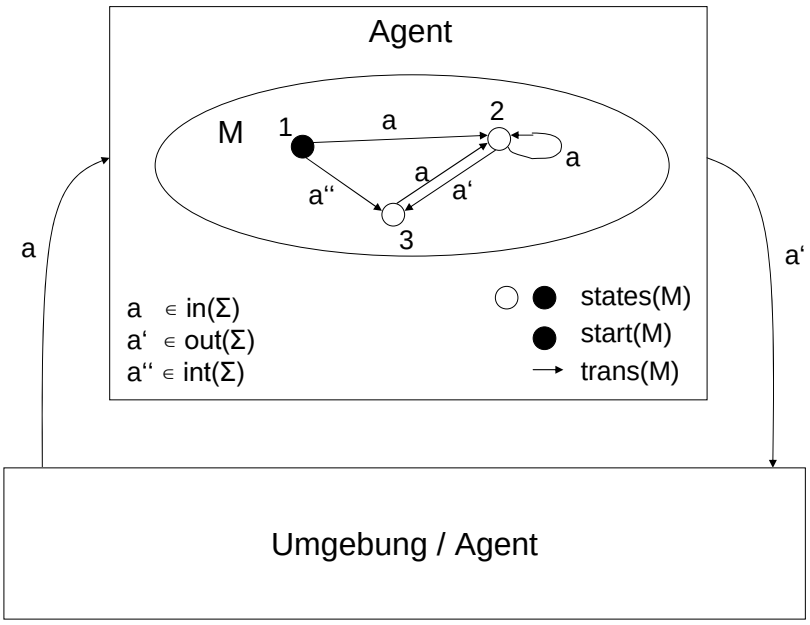

Figure 2.7: I/O-Automat, aus abstraktem Modell abgeleitet

2.7.2 Beschreibung von Multiagentensystemen mit I/O-Automaten

Das vorgestellte abstrakte Modell für einzelne Agenten wird jetzt durch einen I/O-Automaten ausgedrückt. Durch Komposition der I/O-Automaten einzelner Agenten wird ein I/O-Automatenmodell für ein Multiagentensystem gewonnen.

In Kapitel 6 wird gezeigt werden, dass das auf Graphtransformation basierende Systemmodell der zu entwickelnden Modellierungssprache AML auf das I/O-Automatenmodell abgebildet werden kann. Diese Abbildung belegt dann die Angemessenheit des Systemmodells zur Modellierung agentenbasierter Systeme, da transitiv auch die Verträglichkeit mit dem abstrakten Agentenmodell gezeigt sein wird.

Ein einzelner Agent wird als I/O-Automat Ag definiert, dessen Verhalten durch die Abbildung *strategy* bestimmt ist. Die Signatur $sig(Ag)$ des I/O-Automaten Ag ist die disjunkte Vereinigung der Eingabeaktionen $in(Ag)$, der internen Aktionen $int(Ag)$ und der Ausgabeaktionen $out(Ag)$. Im abstrakten Modell sind demgegenüber die nicht notwendigerweise disjunkten Mengen der Perzepte P und Aktionen A gegeben (vgl. Abb. 2.8). Der Durchschnitt werde als $P' = P \cap A$ bezeichnet. Die Menge P' ist also durch Perzepte gegeben, die gleichzeitig Ausgabeaktionen sind. Derartige Perzepte werden als interne Aktionen des I/O-Automaten Ag aufgefasst: $int(Ag) = P'$. Die nicht in P' enthaltenen Perzepte werden als Eingabeaktionen $in(Ag)$ des I/O-Automaten Ag aufgefasst: $in(Ag) = P - P'$. Ausgabeaktionen sind dann Aktionen des abstrakten Modells, die nicht schon interne Aktionen sind: $out(Ag) = A - P' = A - P$.

Die Menge $states(Ag)$ der Zustände des Automaten Ag ist durch $I \cup \{0\} \times I$ gegeben,

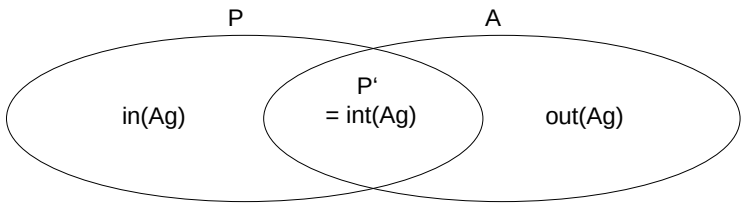

Figure 2.8: Perzepte (P, P') und Aktionen (A) im Verhältnis zur I/O-Automatensignatur

wobei I die Menge der internen Zustände eines Agenten ist. Die Menge $start(Ag)$ der Anfangszustände des Automaten Ag ist ebenfalls I, da das abstrakte Modell keine Einschränkung vorsieht. Für die vollständige Definition des I/O-Automaten Ag sind die möglichen Zustandsübergänge für die verschiedenen Aktionen und Zustände anzugeben. Die Übergangsrelation wird dadurch *spezifiziert*, dass für Aktionen und Bedingungen an den Zustand der resultierende Effekt eines Zustandsübergangs und damit auch der Folgezustand und eine resultierende Aktion angegeben wird. Mit der Spezifikation wird das Aufzählen von einzelnen Zustandsübergängen vermieden.

Die Zustandsübergänge werden auf Basis der *strategy*-Abbildung des abstrakten Modells definiert. Durch diese Abbildung wird in einem Zustand eine Eingabe akzeptiert, eine Ausgabeaktion bestimmt und der Zustand gewechselt.

- *Eingabeaktionen und interne Aktionen:* $p \in in(Ag) \cup int(Ag) = P$

  Vorbedingung: Ag ist im Zustand $i \in states(Ag) = I$ und es existieren $a \in A \cup P'$ und $i' \in states(Ag)$ mit $(i', a) \in strategy(i, p)$.

  Effekt: Ag geht in den Zustand $(0, i')$ über.

- *Ausgabeaktionen:* $a \in out(Ag) = A - P'$

  Vorbedingung: Ag ist im Zustand $(0, i') \in states(Ag)$ und es existiert $a \in A$ und $i \in states(Ag)$ mit $(i', a) \in strategy(i, p)$.

  Effekt: Ag geht in den Zustand i' über.

Ein Beispiel für einen einfachen I/O-Automaten zeigt die Abbildung 2.7.

2.7.3 Kommunikation der I/O-Automaten

Durch die Produktbildung sind die Agenten eng gekoppelt: sobald ein Agent eine Ausgabeaktion produziert, die bei anderen Agenten Eingabeaktionen darstellen, sind die anderen Agenten aktiviert und verarbeiten die Aktion. In verteilten Systemen kommunizieren die beteiligten Prozesse oft nicht in derart eng gekoppelter Weise, sondern

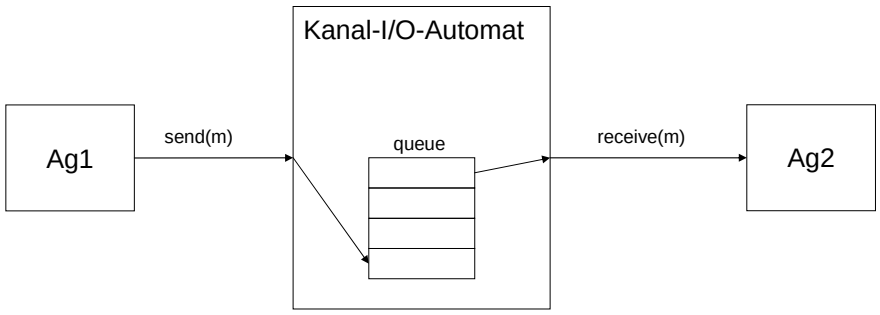

Figure 2.9: Kanal-I/O-Automat zur asynchronen Kopplung zweier Agenten

benutzen dazwischen geschaltete Puffer, die eine Menge von Nachrichten speichern können. Der empfangende Prozess entnimmt dem Puffer Nachrichten unabhängig von der Sendeaktion.

Diese asynchrone Art der Kommunikation kann durch Kanal-I/O-Automaten modelliert werden, die zwischen die kommunizierenden Agenten "geschaltet" werden (vgl. Lynch [25], siehe Abb. 2.9). Um das "Zwischenschalten" zu erklären, muss die Funktionsweise der Kanal-I/O-Automaten geklärt werden.

Sei B ein Nachrichtenalphabet. Ein Kanal-I/O-Automat C hat die Eingabesignatur mit Aktionen $send(m), m \in B$, die an C gesendet werden können, und die Ausgabesignatur mit Aktionen $receive(m), m \in B$, die von anderen I/O-Automaten von C aus empfangen werden können. Die Zustände bestehen aus einem anfangs leeren FIFO-Speicher *queue* mit Elementen aus B. Auf eine detailierte Formalisierung von *queue* wird hier verzichtet. Für einen Kanal-I/O-Automat sind folgende Zustandsübergänge spezifiziert, durch die Eingabeaktionen der Form $send(m)$ in den Puffer gestellt und durch die Ausgabeaktionen der Form $receive(m)$ dem Puffer entnommen werden:

- *Eingabeaktionen:* $send(m)$

  Vorbedingung: keine

  Effekt:

  Füge m in *queue* ein.

- *Ausgabeaktionen:* $receive(m)$

  Vorbedingung: m ist die erste Nachricht in *queue*.

  Effekt: Entferne das erste Element aus *queue*.

Ein Kanal-I/O-Automat wird zwischen zwei Agenten-I/O-Automaten $Ag1$ und $Ag2$ geschaltet, indem das Nachrichtenalphabet des Kanal-I/O-Automaten so gewählt wird,

dass genau die zwischen $Ag1$ und $Ag2$ ausgetauschten Nachrichten enthalten sind. Damit ist klar, wie I/O-Automaten respektive die dadurch modellierten Agenten asynchron kommunizieren können.

2.8 Fazit

Der Begriff Agent wurde im Einklang mit der Literatur auf wenige wesentliche Konzepte reduziert. Als Kernkonzepte von Agenten wurden Autonomie, strukturierte Interaktion und Proaktivität identifiziert. Im Hinblick auf die Problemstellung dieser Arbeit wurde ein abstraktes Systemmodell entwickelt, das die Kernkonzepte erfasst. Im Unterschied zu vorhandenen Modellen lässt sich im vorgestellten abstrakten Modell nicht nur ein einzelner Agent, sondern ein System von Agenten (Multiagentensystem) modellieren.

Die Einsatzmöglichkeiten eines abstrakten Modells sind wegen der wenigen Ausdrucksmittel beschränkt. Das abstrakte Modell wurde deshalb durch ein Automatenmodell verfeinert, das auf den für verteilte Systeme verwendeten I/O-Automaten basiert. Einzelne Agenten sind als spezielle I/O-Automaten modelliert, die dann durch Produktbildung zu einem Systemautomaten für ein Multiagentesystem zusammengesetzt werden. An den Beispielen der BDI-Architektur und der Schichtenarchitektur wurde demonstriert, dass sich wichtige Architekturmodelle in das beschriebene Agentenmodell einordnen lassen.

In der praktischen Softwareentwicklung wird die Modellierungssprache UML eingesetzt. Um die visuelle Modellierung agentenbasierter Systeme durch eine UML-basierte Sprache zu ermöglichen, werden im folgenden Kapitel die Anforderungen an eine solche Sprache und ein darauf aufbauendes Prozessmodell formuliert, die die dargestellten Konzepte agentenbasierter Systeme berücksichtigen. Die vorhandenen Vorschläge zu visuellen Modellierungssprachen und Prozessmodellen werden in Kapitel 5 untersucht. Da die existierenden Ansätze unzureichend sind, wird in Kapitel 6 ein verfeinertes Agentensystemmodell und die UML-basierte Modellierungssprache AML vorgestellt, die den formulierten Anforderungen genügen. In Kapitel 8 verdeutlicht das Prozessmodell APM, wie die Sprache AML zur Modellierung agentenbasierter Systeme eingesetzt werden kann.

Chapter 3

Agentenorientierung im Vergleich zu anderen Paradigmen

Die Charakteristika von Agenten befördern ebenso wie die Paradigmen der objektorientierten oder der komponentenorientierten Softwaretechnik die Entwicklung von jeweils spezifischen Arten von Systemen. Das vorherrschende Paradigma in der Softwareentwicklung, die *Objektorientierung*, hat sich ab ca. 1980 im Zusammenhang mit der Entwicklung komplexer grafischer Benutzungsschnittstellen durchgesetzt (vgl. Kay [75]). Das Paradigma der Komponentenorientierung wurde ab ca. 1990 für die Entwicklung verteilter Systeme propagiert (vgl. Szyperski [98]). In diesem Kapitel wird herausgearbeitet, für welche spezifische Art von Systemen das Paradigma *Agentenorientierung* geeignet ist und wie es sich von den etablierten Paradigmen abhebt.

3.1 Agentenorientierte Softwareentwicklung

Softwareentwicklung ist gemäß Brooks inhärent schwierig (vgl. [13]). Software hat bestimmte essentielle Charakteristika, die es schwierig machen, effiziente, robuste und korrekte Software zu produzieren. Die wesentliche Hürde ist die Komplexität von Software. Die Disziplin der Softwaretechnik hat das Ziel, die inhärente Komplexität von Software zu verstehen und zu beherrschen.

Komplexe Systeme und insbesondere Softwaresysteme zeichnen sich dadurch aus, dass sie aus einer so großen Zahl von Teilen vielschichtig zusammengesetzt sind, dass ihre Struktur und ihr Verhalten schwierig zu erfassen, zu beschreiben und zu entwickeln sind. Um zu verstehen, welche Faktoren die Komplexität von Software beeinflussen, ist eine eher abstrakte Sicht auf Softwaresysteme nützlich. Ein Softwaresystem kann beispielsweise als System kommunizierender I/O-Automaten verstanden werden, das in eine Umgebung eingebunden ist und das Ausgaben produziert, die die Umgebung beeinflussen (vgl. Kapitel 2). Die Umgebung eines Softwaresystems ist oft selbst Software wie im Fall des Internet. Demgegenüber beeinflussen bei eingebetteten Systemen Aktionen den Zustand der physischen Umgebung. Beispielsweise verändert ein Thermostat den Zufluss zu einem

Heizkörper. In einer abstrakten Sicht gibt es wesentliche Faktoren, die die Komplexität beeinflussen (vgl. [43]):

- *Die Eigenschaften der Umgebung:* Die Umgebung kann dynamisch oder statisch, zugreifbar oder nicht modifizierbar, deterministisch oder nichtdeterministisch sein. Im allgemeinen sind dynamische, nicht zugreifbare und nichtdeterministische Umgebungen für die Softwareentwicklung am schwierigsten.

- *Die Interaktion mit der Umgebung:* Eine übliche Form der Interaktion besteht darin, Nachrichten zu empfangen, in einer Eingabewarteschlange abzulegen und dann sequentiell zu verarbeiten. Andererseits werden Nachrichten an Komponenten in der Umgebung gesendet. Daneben gibt es Formen der Umgebung, die als gemeinsamer Speicher funktionieren, auf den mit bestimmten Funktionen lesend oder schreibend zugegriffen wird. Das Konzept des *tuplespace* ist dafür ein Beispiel (vgl. [3]).

Traditionell haben sich Methoden der Softwaretechnik bei der Konstruktion funktionaler Systeme in einer statischen, zugänglichen und deterministischen Umgebung als erfolgreich erwiesen. Heutige Systeme sind allerdings eher in eine dynamische, unzugängliche und nichtdeterministische Umgebung eingebettet. Die Konstruktion und Produktion solcher Systeme ist problematisch. Die für die Softwaretechnik wichtige Fragestellung ist, wie mit solchen komplexen Systemen umgegangen werden kann. Grundlegende Techniken für den Umgang mit komplexen Systemen sind nach Booch, Ghezzi, Lopes und Parnas (vgl. [11, 55, 65, 87]):

- *Dekomposition (Modularisierung):* Diese Bezeichnungen meinen die Zerlegung eines Systems in mehrere Module, d.h. Teile. Die Modularisierung soll in der Art geschehen, dass zwischen den Modulen möglichst wenig Interaktion stattfindet (vgl. Ghezzi und Parnas [55, 87]). Diese Eigenschaft wird als "minimale Kopplung" von Modulen bezeichnet. Innerhalb eines Moduls soll dagegen die Interaktion zwischen den Bestandteilen höher sein als zwischen Modulen. Das ist die Eigenschaft der "maximalen Kohäsion". Modularisierung ist ein Prinzip, durch das möglichst unabhängig funktionierende Bestandteile eines Systems entstehen sollen. Module können so relativ unabhängig voneinander verändert werden. Durch das Hinzufügen neuer Module und deren Kopplung an bereits vorhandene Module ist ein modularisiertes System gut erweiterbar.

- *Abstraktion:* Darunter wird die Vereinfachung eines Problems verstanden (vgl. Booch [11]). Dabei werden Details, die nicht zum Kern des Problems gehören, aus der weiteren Betrachtung des Problems vorerst ausgeblendet und erst bei der verfeinerten Problembeschreibung und -lösung berücksichtigt. Abstraktion ist damit ein Prinzip, das der Lösung eines Problems in verschiedenen sequentiellen Schritten dient. Durch Abstraktion wird die Portabilität verbessert, da beispielsweise für die Anpassung einer Problemlösung an eine neue Systemumgebung lediglich die letzten Verfeinerungsschritte eines Problemlösungsverfahrens erneut auszuführen sind.

- *Trennung nach Aufgaben (separation of concerns):* Darunter wird gemäß Ghezzi et al. [55] und Lopes et al. [65] die Aufteilung eines Problems in unterschiedliche Aspekte verstanden. Bei der Lösung des Problems werden die Aspekte getrennt behandelt. Die Lösung des Problems wird oft vereinfacht, da jetzt einfachere Teilprobleme (Aufgaben) zu lösen sind. Ausgehend von den Lösungen für die Aspektprobleme wird die Lösung für das Gesamtproblem zusammengesetzt. Änderungen auf einzelnen Aspekten wirken sich auf andere Aspekte nur indirekt aus. Für die Erweiterung um einen neuen Aspekt ist dessen Integration mit den anderen Aspekten zu beschreiben.

Der Zusammenhang zwischen komplexen Systemen und agentenorientierter Softwaretechnik wird von Jennings diskutiert (vgl. [71]). Ein Ansatz für die Untersuchung agentenorientierter softwaretechnischer Verfahren besteht darin, mit Hilfe von Softwaremetriken die Leistungsfähigkeit der Verfahren bei der Entwicklung komplexer Systeme zu messen. Bis heute sind allerdings keine derartigen empirischen Untersuchungen erfolgt. Aus diesem Grund soll im folgenden qualitativ argumentiert werden, warum agentenorientierte Softwaretechnik (AOSE) für die Entwicklung komplexer Systeme geeignet ist.

Jennings bildet die Hypothese der Adäquatheit des agentenorientierten Ansatzes [71]: "Agent-oriented approaches can significantly enhance our ability to model, design and build complex, distributed software systems." Die obige Hypothese teilt sich unter Berücksichtigung der grundlegenden Techniken für den Umgang mit komplexen (und verteilten) Systemen in drei Teile auf:

- Agentenorientierte Dekomposition ist eine effektive Methode, um den Problembereich eines komplexen Systems zu zerlegen.
- Agentenorientierte Abstraktionen sind ein geeignetes Mittel für die Modellierung von Systemen.
- Der agentenorientierte Ansatz zur Behandlung von Aspekten ist für komplexe Systeme geeignet.

Die Plausibilität dieser drei Teilhypothesen zur Zerlegung, Abstraktion und Organisation resultiert aus den in Kapitel 2 dargestellten Eigenschaften von Agenten:

Dekomposition. Wie bereits angeführt, setzen sich komplexe Systeme aus Subsystemen zusammen. Im agentenorientierten Ansatz können die Subsysteme mit einzelnen Agenten oder Gruppen von Agenten identifiziert werden. Autonomie impliziert nach den Ausführungen des letzten Kapitels, dass Agenten einen eigenen Kontrollfluss haben. Diese Lokalisierung des Kontrollflusses reduziert die Kopplung der Subsysteme, da die Funktionen der Subsysteme unabhängig voneinander aktiviert werden können.

Agenten entscheiden autonom, welche Operationen zu welcher Zeit ausgeführt werden. Sie orientieren sich an den zu verfolgenden Zielen. Durch die strukturierte Interaktion zwischen Agenten werden die Kontrollflussabhängigkeiten zwischen Subsystemen

realisiert. Die Interaktion in agentenorientierten Systemen erfolgt auf der Basis von Protokollen bedarfsgesteuert und konform mit den Zielen der beteiligten Agenten. Agenten kommunizieren mit anderen Agenten also lediglich, um ihre eigenen Ziele oder gemeinsame Ziele zu erreichen. Dadurch wird eine vergleichsweise lose Kopplung der Subsysteme erreicht.

Insgesamt werden die Kriterien an eine vorteilhafte *Zerlegung* eines Systems erfüllt.

Abstraktion. Wichtige agentenorientierte *Abstraktionen* bilden die Begriffe Agent, Protokoll, Ziel und Strategie. Auf der anderen Seite bestehen komplexe Systeme aus Subsystemen mit Interaktionen. Zu klären ist, wie agentenorientierte Abstraktionen komplexe Systeme beschreiben können.

Zwischen den Begriffen Subsystem und Agent existiert eine Korrespondenz, d.h. Subsysteme werden als Agenten identifiziert. Beziehungen zwischen Subsystemen werden durch Protokolle ausgedrückt: Durch die Zuordnung von Protokollen zu bestimmten Agententypen wird festgelegt, welche Agenten durch Protokolle miteinander kommunizieren und damit in Beziehung treten können.

Ziele beschreiben für Agenten die Anforderungen, die Agenten erfüllen sollen. Durch den Begriff der Strategie werden operationale Spezifikationen bezeichnet, mit denen zugeordnete Ziele erreicht werden sollen. Strategien sind damit Verfeinerungen von Zielen.

Insgesamt ist zu sehen, dass die agentenorientierten Abstraktionen sinnvoll dazu verwendet werden können, Elemente komplexer Systeme zu beschreiben.

Trennung nach Aufgaben (separation of concerns). Ein Ziel, das ein Agent erreichen soll, wird mit Hilfe einer Strategie verfolgt. Eine Strategie setzt sich aus Teilaufgaben zusammen, deren Lösung zum Erreichen des Ziels führen. Im allgemeinen kommuniziert ein Agent zum Erreichen eines Ziels mit anderen Agenten und erledigt dadurch Teilaufgaben. Jede Teilaufgabe wird durch Anwendung eines Protokolls gelöst. Jedes Protokoll ist also eine Problemlösungsstrategie für eine spezielle Aufgabe. Durch Integration verschiedener Protokolle in einer Strategie eines Agenten werden die Lösungen von Teilaufgaben zu einer Lösung der Gesamtaufgabe vereinigt, d.h. das Ziel zur Strategie wird erreicht.

Nachdem qualitativ gezeigt wurde, dass das Paradigma der Agentenorientierung sich für die Entwicklung komplexer Systeme eignet, stellt sich die Frage, für welche Systemklassen Agenten besonders geeignet sind.

Zeitlich wechselnde Anforderungen ergeben sich in offenen, dynamischen Systemen. Entsprechend den Anforderungen können neue Agenten erzeugt werden und über die Protokollkommunikation in ein vorhandenes Agentensystem eingebettet werden. Der in den Agenten lokalisierte Kontrollfluss erlaubt eine gute Skalierung der Systemgröße. Die inkrementelle Entwicklung ist beispielsweise bei robusten Systemen notwendig: unzureichende Agenten werden (im Betrieb) durch besser funktionierende ersetzt.

Das Paradigma der Agentenorientierung erleichtert also tendenziell die Entwicklung und den Betrieb offener, dynamischer und robuster Systeme.

3.2 Agenten und objektorientierte Softwareentwicklung

Heute herrscht das objektorientierte Paradigma in der Softwareentwicklung vor. Es wird sowohl objektorientiert programmiert als auch modelliert. Objektorientierte Modelle lassen sich für verschiedene Aktivitäten des Softwareentwicklungsprozesses wie Anforderungsdefinition, Analyse und Entwurf verwenden (vgl. Kapitel 4). Das Vorgehen zur Entwicklung von Modellen und deren Verfeinerung hin zu Software wird durch Prozessmodelle genauer beschrieben. Bis heute sind verschiedene Prozessmodelle, wie z.B. der Rational Unified Process (RUP), Catalysis, etc. entwickelt worden. RUP und Catalysis werden in Kapitel 5 analysiert.

Für die Verwendung des Paradigmas der Objektorientierung gibt es nach Abadi und Cardelli verschiedene Gründe [2]. Der Hauptgrund besteht in der Ähnlichkeit zwischen Softwaremodellen, inklusive Implementierungen, und Modellen der physischen Welt. Viele Gesichtspunkte der physischen Welt lassen sich als Objekte verstehen. Objekte sind abgeschlossene Einheiten, die sichtbare und unsichtbare Merkmale (Attribute) und ein spezifisches Verhalten aufweisen. Diese Eigenschaften physischer Objekte finden sich in Software-Objekten wieder. Durch die Gewohnheit, mit objektorientierten Modellen der physischen Welt umzugehen, stellt es eine Erleichterung dar, auch Software durch solche Modelle zu beschreiben.

Objekte sind die kleinsten Einheiten in einem objektorientierten Softwaresystem. Sie haben eine Identität, Attribute und Methoden. Objekte haben einen Zustand, der als Belegung von speziellen Zustandsvariablen aufgefasst wird. Das Verhalten eines Objektes wird durch seine Methoden bestimmt. Klassen fassen Objekte mit gleichen Attributen, Assoziationen und Methoden zusammen. Eine Klasse bestimmt den Typ eines Objektes.

Ein besonderer softwaretechnischer Vorteil liegt in der Gruppierung von Attributen und Methoden in Objekten. Die Schnittstelle eines Objektes besteht aus Signaturen für Attribute und Methoden. Erst durch die Kopplung von Daten und zugehörigen Funktionen (Methoden) ist dafür gesorgt, dass mit den Daten in einem intendierten Sinn umgegangen wird, wobei die Art des Umgangs durch die Methoden gegeben ist.

Das Konzept der Vererbung befördert einerseits die Wiederverwendung, z.B. bei Frameworks, in denen in verschiedenen Kontexten von Framework-Klassen geerbt wird. Andererseits erlaubt Vererbung auch die Erweiterung von Objekten durch weitere Attribute und Methoden. Dabei können nicht nur neue Attribute und Methoden eingeführt werden, sondern es können auch bereits existierende Methoden überschrieben werden, um das Verhalten des Objektes zu verändern und nicht nur zu erweitern. Ein wesentliches objektorientiertes Konzept besteht darin, dass die "neuen" Objekte an Stelle der "alten" verwendet werden können, da die Schnittstelle durch Vererbung lediglich erweitert wird.

Damit kann der Kontext des Objektes wiederverwendet werden.

Objekte sind für die Beherrschung von Komplexität entsprechend den zuvor aufgestellten Kriterien geeignet:

- *Zerlegung*: Objekte sind die Einheiten der Zerlegung. Sie bündeln und kapseln Daten und Verhalten im Sinne des Parnasschen Prinzips der maximalen Kohäsion.
- *Abstraktion*: Die Schnittstellen von Objekten in Form von Signaturen für Attribute und Methoden bilden das primäre Abstraktionsmittel. Der Mechanismus der Vererbung erlaubt es, vom Verhalten spezifischer Objektklassen zu abstrahieren und nur auf die allgemeinen, in einer Oberklasse der spezifischen Klassen isolierten Methoden und die dort definierten Attribute Bezug zu nehmen.
- *Trennung nach Aufgaben*: Einzelne Objekte werden oft für dezidierte Aufgaben vorgesehen. Dabei wird die Aufgabe dann allerdings oft durch auf mehrere Klassen verteilte Funktionen erfüllt (Scattering). Um die Anzahl von Klassen zu begrenzen, ist es andererseits üblich, dass eine Klasse Funktionen zu mehreren Aufgaben realisiert (Tangling) [20].

Vergleich mit Agenten. Sowohl Agenten als auch Objekte kapseln Daten und verändern diese Daten durch Operationen (vgl. Wooldridge [105]). Typische Objekte sind passiv. Daneben existieren aber auch aktive Objekte mit einem eigenen Kontrollfluss (thread of control). Agenten sind demgegenüber immer aktiv (vgl. Wooldridge [105] Seite 4, Burmeister [16] Seite 3). Das folgt aus den Konzepten Proaktivität und Autonomie, die beide erfordern, dass Agenten von sich aus aktiv werden. Allerdings weisen aktive Objekte keine weiteren Merkmale von Agenten auf.

Nach Abadi und Cardelli kapselt ein Objekt seinen Zustand und hat Autonomie über den Zustand [2]. Abhängig vom Zustand erfolgt die Auswahl einer Operation zur Reaktion auf eine eingehende Nachricht (vgl. Abbildung 3.1). Die Aktivierung und die Auswahl von Methoden wird in Modellen mit Hilfe von Statecharts modelliert. In Programmiersprachen wie C++ oder JAVA existiert eine derartige Verhaltenskontrolle nicht. Jedes Objekt kann jede öffentliche Methode eines jeden anderen Objektes zu jeder Zeit aufrufen, und der Aufruf führt zur Abarbeitung der Methode. Die zustandsabhängige Verhaltensauswahl kann für Objekte implementiert werden, indem ein Objekt lediglich eine Methode zum Empfangen von Nachrichten (als Parameter) besitzt. In der Methode wird dann über die Auswahl des Verhaltens anhand des Status von Zustandsvariablen entschieden.

Ein Agent besitzt Entscheidungsautonomie, d.h. er kontrolliert auch die Auswahl seines Verhaltens. Im Unterschied zu Objekten ist allerdings nicht der Zustand eines Agenten entscheidend für die Verhaltensauswahl, sondern das zu erreichende Ziel und eine damit einhergehende Strategie. Konstituenten des Konzeptes Proaktivität wie Ziel oder Strategie finden sich bei Objekten nicht.

Als weiteres Konzept ist die strukturierte Interaktion zu diskutieren. In offenen Systemen kann die Umgebung nicht vollständig kontrolliert werden (vgl. Kapitel 1). Wichtig

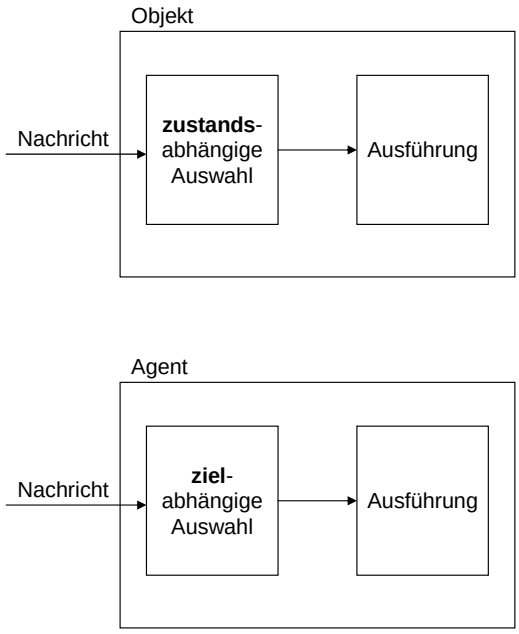

Figure 3.1: Objekt vs. Agent

sind deshalb Entscheidungen zur Laufzeit über die Art und Existenz von Interaktionen. Die Interaktion ist deshalb notwendigerweise flexibel. Erreicht wird Flexibilität dadurch, dass zur Laufzeit über die Verwendung von Interaktionsverfahren entschieden wird. Protokolle sind derartige Verfahren, die den Nachrichtenfluss zwischen mehreren Instanzen zur Abwicklung der Interaktion beschreiben. Durch ein Protokoll ist für jede der beteiligten Instanzen eine jeweilige Funktion bestimmt. Das Protokoll integriert die einzelnen Funktionen zu einer gemeinsamen. Der Effekt ist eine Koordination der am Protokoll beteiligten Instanzen.

Vorteilhaft sind Protokolle dahingehend, dass Scattering und Tangling vermieden werden. So wird der Nachteil vermieden, dass die Aufgabe der Interaktion auf einzelne Kommunikationspartner verteilt gelöst wird und dadurch schwierig zu verfolgen (traceability) und schlecht nachvollziehbar (comprehensibility) ist. Protokolle sind außerdem gut wiederverwendbar und stellen eine sinnvolle Möglichkeit dar, die Aufgabe der Interaktion von anderen Aufgaben zu trennen (separation of concerns) (vgl. Clarke [20], Depke et al. [39]).

Die objektorientierten Konzepte für die Beschreibung von Protokollen sind wenig entwickelt. Darauf wird im Zusammenhang mit der objektorientierten Modellierung in UML innerhalb von Kapitel 5 weiter eingegangen. Im Zusammenhang mit Protokollen eingeführte Konzepte wie das der Rolle machen einen bedeutenden Unterschied zwischen Objekten und Agenten aus (vgl. Jennings [71]).

Die Unterschiede zwischen Objekten und Agenten lauten kurz zusammengefasst:

- Agenten sind so handlungsautonom wie aktive Objekte. Die Entscheidungsau-

tonomie von Agenten ähnelt konzeptionell Objekten mit zustandsabhängiger Verhaltensauswahl beim Empfang von Nachrichten.

- Agenten verhalten sich proaktiv. Objekte kennen demgegenüber weder Ziele noch Strategien.

- Strukturierte Interaktion, z.B. in Form von Protokollen wird von Objekten konzeptionell wenig unterstützt.

3.3 Agenten und komponentenbasierte Softwareentwicklung

Die Konzepte der komponentenbasierten Softwareentwicklung sind mit dem agentenorientierten Ansatz verwandt. Die Gemeinsamkeiten und Unterschiede sollen herausgestellt werden.

Heute wird Software immer mehr aus vorhandenen Komponenten zusammengesetzt. Die Motivation für komponentenbasierte Software besteht in der softwaretechnischen Qualität der *Wiederverwendung* (vgl. Szyperski [98]). Diese resultiert aus den wesentlichen Eigenschaften von Komponenten (vgl. [98], Seite 34):

- Komponentenbasierte Systeme werden durch Komposition aufgebaut, und Komponenten sind die Einheiten der Komposition.

- Die unabhängige Entwicklung und Verwendung von Komponenten ist möglich.

- Komponenten haben verbindlich spezifizierte Schnittstellen und lediglich explizite Kontextabhängigkeiten.

Komponenten dienen der Beherrschung von Komplexität. Mit Bezug auf die am Anfang des Kapitels aufgeführten Kriterien leisten sie folgendes:

- *Zerlegung*: Die Komposition eines Systems aus unabhängigen Subsystemen, den Komponenten, gehört zu den essentiellen Eigenschaften des Ansatzes.

- *Abstraktion*: Komponenten werden als "black boxes" angesehen. Für andere Komponenten sind nur die Schnittstellen sichtbar.

- *Trennung nach Aufgaben*: Die Komponierbarkeit zweier Komponenten hängt von der Verträglichkeit der Schnittstellen ab. Durch Strukturierung der Schnittstellen sind mögliche Kompositionen und damit auch die Organisation von Systemen festgelegt.

Vergleich mit Agenten. Agenten sind genauso wie Komponenten die Einheiten der Verwendung (deployment). Sie sind ebenso zu Systemen komponierbar. Agenten weisen im Vergleich zu Komponenten folgende Unterschiede auf:

- Komponenten sind weder autonom noch proaktiv.
- Die Interaktion zwischen Komponenten basiert auf dem Aufruf von Funktionen oder dem Austausch von Nachrichten (vgl. Szyperski [98]). Agenten kommunizieren durch Protokolle, d.h. in wesentlich stärker strukturierter Weise.

3.4 Fazit

Als Fazit bleibt festzuhalten, dass Agenten für die Entwicklung komplexer Systeme vorteilhaft verwendet werden können. Besonders geeignet ist das Paradigma der Agentenorientierung für die Entwicklung und den Betrieb offener, dynamischer und robuster Systeme, wie sie beispielsweise durch Ad-Hoc-Netzwerke gegeben sind. Die dynamische Komposition von Agenten zu Systemen ist in diesen Kontexten von Vorteil.

Agenten haben Gemeinsamkeiten mit Objekten und Komponenten. Sie bündeln Zustand und Verhalten ebenso wie Objekte. Agenten sind dadurch ähnlich zu Komponenten, dass sie genauso zu Systemen komponiert werden. Agenten unterstützen genauso wie Komponenten, aber im Unterschied zu Objekten, die strukturierte Interaktion.

Die Unterschiede zwischen Agenten und Objekten bzw. Komponenten bestehen insbesondere darin, dass Agenten Anforderungen in Zielen und Strategien berücksichtigen. Autonomes Verhalten ist als essentielles Konzept ebenfalls Agenten im wesentlichen vorbehalten.

Chapter 4

Anforderungen an ein agentenorientiertes Modellierungsverfahren

Die Entwicklung agentenbasierter Systeme erfordert ein Enwicklungsverfahren, das auf die spezifischen Konzepte von Agenten zugeschnitten ist. Im Fokus dieser Arbeit besteht die Aufgabe darin, die Modellierung agentenbasierter Systeme zu ermöglichen. Zu diesem Zweck soll ein Modellierungsverfahren entwickelt werden.

Das Ziel dieses Kapitels besteht darin, die Anforderungen an ein solches Modellierungsverfahren zu entwickeln. Modellierungsverfahren lassen sich in allgemeiner Weise strukturieren (vgl. Engels et al. [47]). Anhand der allgemeinen Bestandteile werden die Anforderungen an ein agentenorientiertes Modellierungsverfahren entwickelt.

4.1 Überblick

Modellierung ist im wesentlichen ein Konzept der Abstraktion sowohl von Eigenschaften der realen Welt, in die ein Softwaresystem sinnvoll eingebettet werden soll, als auch von dem Softwaresystem selbst, von dessen systembedingten Details abgesehen werden soll (vgl. Abb. 4.1 und Engels [47]). Modelle sollen eine Brücke zwischen realer Welt und Software bauen. Um Modelle ausdrücken und notieren zu können, wird eine Modellierungssprache benötigt.

Die Existenz einer Sprache impliziert nicht ihre angemessene Verwendung. Hinzu kommen pragmatische Regeln, die angeben, wie und wofür die Sprache einzusetzen ist. Für Modellierungssprachen ist anzugeben, auf welche Weise Modelle und Sprachkonstrukte schrittweise eingesetzt werden sollen, um von einer Problemstellung in der realen Welt zu einem Softwaresystem zu kommen. Eine weitere Abstraktion besteht darin, das Einzelfallvorgehen zu verallgemeinern und ein Prozessmodell zu formulieren.

Für die Entwicklung eines Modellierungsverfahrens sind detailiertere Anforderungen notwendig. Von Engels et al. werden in [47] zahlreiche Anforderungen an objektori-

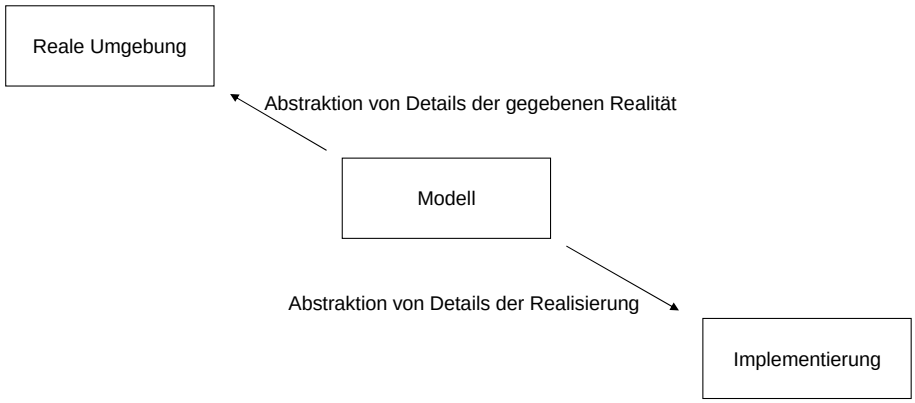

Figure 4.1: Die Rolle von Modellen

entierte Modellierungsverfahren genannt, die weithin allgemein genug sind, um für die meisten Modellierungsverfahren zu gelten. Die Anforderungen basieren auf einer Strukturierung des Gesamtverfahrens in *Sprachstruktur*, *Modellkonstituenten*, *Modellkomposition*, *Modellierungsprozess*, *Modellüberprüfung* und *Softwareentwicklungsprozess*. Diese Bestandteile sollen im folgenden diskutiert werden. Ausgehend von den in Kapitel 2 dargestellten Konzepten werden dann die Anforderungen an eine agentenorientierte Modellierungssprache formuliert.

4.2 Modellierungssprache

Zunächst werden die Anforderungen an die Sprachstruktur, die Modellkonstituenten und die Modellkomposition eines agentenorientierten Modellierungsverfahrens entwickelt.

4.2.1 Sprachstruktur

Modelle dienen der Beschreibung von Systemen und deren Verhalten. Um in Modellen von einzelnen Ausprägungen von Systemen und von konkreten Abläufen zu abstrahieren, wird zwischen Instanz- und Typebene unterschieden. Modellinstanzen werden durch statische Modelle beschrieben. Die Veränderungen von Modellinstanzen werden durch dynamische Modelle beschrieben.

Bei diskreten Systemen wird die Systemdynamik durch Folgen von Zuständen beschrieben. Zustände charakterisieren dabei den aktuellen Ausführungszustand des Systems. Der Zustand eines modellierten Systems ist durch die Instanz eines statischen Modells gegeben. Da die meisten realen (technischen) Systeme einen Zustand haben, aus dem heraus sie sich entwickeln, soll ein solcher Anfangszustand zu jedem Modell existieren.

Die Konzepte agentenbasierter Systeme sollen sich in Syntax und Semantik der Modellierungssprache widerspiegeln. Der semantische Bereich und die Semantik als Abbildung

der Syntax in den semantischen Bereich sind so zu definieren, dass die Modelle mit dem abstrakten Agentenmodell aus Kapitel 2 verträglich sind.

Modellierungssprachen können sich auf eine bestimmte Anwendungsdomäne, wie beispielsweise Echtzeitsysteme beschränken, oder sie können allgemein und domänenübergreifend sein. Die in dieser Arbeit zu entwickelnde agentenorientierte Modellierungssprache soll nicht auf spezifische Anwendungsdomänen beschränkt sein.

Damit ergeben sich folgende allgemeine Anforderungen:

- Modelle haben Instanzen, die den Ausführungszustand eines Systems beschreiben.
- Ein statisches Modell beschreibt den strukturellen Aufbau von Systemen.
- Ein dynamisches Modell spezifiziert mögliche Änderungen von Systemzuständen.
- Zustände sind mit Instanzen gleichzusetzen.
- Statisches und dynamisches Modell bestimmen die Zustandsentwicklung von Systemen, ausgehend von einem Ausgangszustand.
- Konzepte agentenbasierter Systeme werden integriert.
- Die Verträglichkeit mit dem abstrakten Agentenmodell ist sichergestellt.
- Es gibt keine Beschränkung auf eine Anwendungsdomäne.

Um die Konzepte agentenbasierter Systeme in Modellen verwenden zu können, wird im nächsten Abschnitt erarbeitet, durch welche Konstituenten die Konzepte erfasst werden können.

4.2.2 Modellkonstituenten

Nach Engels et al. [47] bilden *Konstituenten* die Bestandteile eines Modells, durch die wesentliche Anforderungen erfasst werden. Modellkonstituenten sind beispielsweise views, subjects, aspects, pattern und frameworks. Für agentenbasierte Systeme sind Konstituenten zu den Konzepten Autonomie, strukturierte Interaktion und Proaktivität erforderlich (vgl. Kapitel 2).

Autonomie. In einem *autonomen* System entscheiden Agenten selbst über ihr Verhalten. Verhaltensmodelle für Agenten müssen ausdrücken können, dass Agenten sowohl auf der Basis ihrer Wahrnehmung der Umgebung als auch davon unabhängig über eigene Aktionen entscheiden. Diese *Entscheidungsautonomie* von Agenten zeigt sich darin, dass sie Entscheidungen anhand verschiedener Freiheitsgrade wie externer Nachrichten, des Umgebungszustands und des inneren Zustands treffen können. Diese Freiheitsgrade, ob, worauf und wie auf etwas reagiert wird, sind bei Agenten stark ausgeprägt. Konstituierend für diese Entscheidungsautonomie sind *interne, flexible Kontrollelemente* von Agenten.

Weiterhin bedingt Autonomie, dass Agenten auch unabhängig von einem äußeren Stimulus aktiv sein können und sich nicht rein reagierend verhalten. Diese *Handlungsautonomie* einzelner Agenten impliziert Nebenläufigkeit in einem System von Agenten. Eine Konstituente für Autonomie im Sinne von unabhängiger Aktivität ist deshalb der eigene *Ausführungsfaden* (thread) eines Agenten, der durch seine Aktivitätenfolge gegeben ist.

In einer Sprache zur Modellierung agentenbasierter Systeme müssen die Konstituenten von Autonomie repräsentiert sein:

- Handlungsautonomie: Jeder Agent besitzt einen eigenen Ausführungsfaden, aus dem ein von anderen Agenten unabhängiger Kontrollfluss resultiert.
- Entscheidungsautonomie: Es muss flexible Kontrollelemente geben, die die unterschiedlichen Freiheitsgrade von Agenten berücksichtigen. Das Verhalten wird anhand der eingehenden Nachrichten, des Umgebungszustands und des inneren Zustands in variabler und komplexer Weise beeinflusst.

Strukturierte Interaktion. Die *Interaktion* zwischen Agenten beschränkt sich nicht auf den isolierten Aufruf von Operationen, wie das beispielsweise bei Objekten der Fall ist. Interaktion zwischen Agenten ist strukturiert und dient der Koordination der beteiligten Agenten.

Bereits in Kapitel 2 wurde der Begriff *Protokoll* eingeführt. Protokolle sind Schemata für koordiniertes Verhalten von Agenten und beschreiben insbesondere einen wechselseitigen Austausch von Nachrichten. Die möglichen Nachrichtenfolgen eines Protokolls sind i.d.R. nicht an bestimmte Agententypen gebunden. Notwendig ist deshalb eine Protokollbeschreibung, die a priori unabhängig von den Agenten ist, die das Protokoll einsetzen.

Gemäß Holzmann muss ein Protokoll folgende Anforderungen erfüllen (vgl. [62], Kapitel 2):

- Die Funktion (Semantik) des Protokolls muss eindeutig beschrieben sein.
- Die Voraussetzungen über die Umgebung müssen definiert sein.
- Das "Vokabular" und der Aufbau (die Syntax) der Nachrichten sind festzulegen.
- Prozedurale Regeln (das Verhalten) für den Nachrichtenaustausch sind anzugeben, z.B. zur Synchronisation von Sender und Empfänger oder zur Deadlock-Vermeidung

Proaktivität. Ein sich *proaktiv* verhaltender Agent leitet Operationen ein, ohne notwendigerweise durch seine Umgebung dazu angestoßen zu werden (vgl. Kapitel 2). Stattdessen soll durch eine Folge von Operationen ein *Ziel* erreicht werden.

Anforderungen an Ziele als Konstituenten für proaktives Verhalten sind gemäß Wooldridge (vgl. [103]):

- Die Voraussetzungen für das Erreichen eines Zieles müssen formuliert werden.

- Der Effekt, der mit dem Erreichen eines Zieles verbunden ist, muss beschrieben werden.

Eine weitere Konstituente proaktiven Verhaltens ist die *Strategie*. Eine Strategie eines Agenten ist gemäß Kapitel 2 durch ein spezifisches lokales Verhalten gegeben. Die Ziele und Strategien eines Agenten stehen zueinander in Beziehung: Eine *Gewinnstrategie* ist eine Strategie, die bei Erfüllung der Voraussetzungen eines Zieles bei ihrer Anwendung den Effekt des Zieles nach sich zieht. Die in Kapitel 2 vorgestellte BDI-Strategie ist ein (abstraktes) Beispiel für eine Gewinnstrategie, die die in der Menge *Desires* gegebenen Ziele erreicht.

In der Spieltheorie beschreiben Strategien die Interaktion von "Spielern" zum Erreichen von individuellen oder gemeinsamen Zielen (vgl. [52]). Hier beschreibt eine *Kommunikationsstrategie* das Verhalten mehrerer Agenten und ihre Interaktion. Eine Kommunikationstrategie ist eine Gewinnstrategie, falls die Agenten durch ihre Anwendung ein gemeinsames Ziel erreichen.

Anforderungen an Strategien als Konstituenten für proaktives Verhalten sind gemäß Wooldridge (vgl. [103]):

- Eine Strategie soll das Verhalten eines oder mehrerer Agenten beschreiben.
- Jeder Strategie ist (mindestens) ein Ziel zugeordnet, das durch ihre Anwendung erreicht werden kann.
- Bezüglich der zugeordneten Ziele ist für eine Strategie zu zeigen, dass sie Gewinnstrategie ist.

Nachdem die Konstituenten und damit die Anforderungen an Modellelemente einer agentenorientierten Modellierungssprache vorgestellt sind, ergeben sich Anforderungen an die Komposition von Konstituenten.

4.2.3 Modellkomposition

Durch Modellkomposition werden Modellkonstituenten zu größeren Modellen integriert. Zu klären ist, auf welche Weise einzelne Konstituenten gleichen oder verschiedenen Typs strukturell zusammengesetzt werden können. Modellkomposition beinhaltet die Aufgabe der Integration von Modellkonstituenten in Modellen.

Demgegenüber wird in einem Modellierungsprozess beschrieben, wie Modelle zu erzeugen sind, mit anderen Modellen in Beziehung stehen oder aus ihnen hervorgehen. Da Modelle oft Teile anderer Modelle werden oder auch selbst verändert werden, müssen diese Arten der Veränderung beschrieben werden. Der erste Vorgang ist die horizontale Skalierung von Modellen, während Verfeinerungen eines Modells als vertikale Skalierung bezeichnet werden (vgl. Engels et al. [47]). Der Modellierungsprozess wird erst im Abschnitt 4.3 behandelt.

Autonomie und Strukturierte Interaktion. Protokolle schränken die Autonomie von Agenten dadurch partiell ein, dass das Verhalten der Agenten Protokollverhalten integriert. Diese Modifikation des Verhaltens erfolgt durch die Bindung von Protokollen an Agenten. Bezogen auf einzelne Agenten können mehrere Protokolle integriert werden. Aus diesen Zusammenhängen folgen als Anforderungen:

- Die Bindung von Protokollen an Agenten und die damit einhergehende Verhaltensmodifikation der Agenten ist zu beschreiben.
- Ein Verfahren zur Bindung mehrerer Protokolle an einen Agenten ist festzulegen.

Autonomie und Proaktivität. Ein proaktiver Agent ist bestrebt, Ziele zu erreichen. Das eigenständige Streben nach einem Ziel mit Hilfe einer Strategie ist nur möglich, falls der Agent selbst aktiv ist. Weiterhin muss er in der Lage sein, unter verschiedenen Möglichkeiten für eigenes Verhalten entsprechend einer Strategie so auszuwählen, dass er seinem Ziel näher kommt. Damit ist andererseits die Entscheidungsautonomie eingeschränkt. Ein proaktiver Agent zeigt also alle Eigenschaften autonomen Verhaltens. Als Anforderungen ergeben sich:

- Proaktives Verhalten wird nur für handlungsautonome Agenten modelliert.
- Die Art der Einschränkung von Entscheidungsautonomie durch Strategien muss festgelegt sein.

Strukturierte Interaktion und Proaktivität. In Multiagentensystemen sind Ziele nicht notwendigerweise einzelnen Agenten zugeordnet, sondern auch mehreren kooperierenden Agenten. Die gemeinsamen Ziele werden in Kommunikationsstrategien, d.h. durch *Interaktionsprotokolle* verfolgt (vgl. [50]). Durch Abarbeiten eines Protokolls erreichen die an dem Protokoll beteiligten Agenten ein mit dem Protokoll verknüpftes gemeinsames Ziel. Für einen einzelnen Agenten geht das Abarbeiten eines Protokolls mit dem Verfolgen eines lokalen Ziels einher. Als Anforderungen ergeben sich:

- Die Dekomposition von Kommunikationszielen in lokale Ziele ist zu beschreiben.
- Die Auswirkung der Protokollbindung auf lokale Stratgien ist zu berücksichtigen.

4.3 Modellierungsprozess

Um zu einem sinnvollen Ausgangspunkt für die Entwicklung eines agentenorientierten Prozessmodells zu kommen, sollen Anforderungen an Softwareprozessmodelle entwickelt werden. In den folgenden beiden Abschnitten stehen die Grundlagen für Softwareprozessmodelle, die Fundamentalaktivitäten und zugehörige Artefakte im Mittelpunkt. Danach werden die allgemeinen und die agentenspezifischen Anforderungen an ein agentenorientiertes Prozessmodell entwickelt.

4.3.1 Grundlagen

Ein Softwareentwicklungsprozess (kurz: Prozess) ist ein Vorgang, der der Konstruktion und Wartung von Software dient. Das Ziel des Prozesses besteht darin, ein Softwaresystem zu erstellen, das den Erwartungen aller Nutzer des Systems gerecht wird. Prozesse sind in Projekte eingebettet, in denen organisatorische und zeitliche Randbedingungen berücksichtigt werden. Ein Prozess wird durch Faktoren beeinflusst, wie die Zahl der Projektbeteiligten, die eingesetzten Systemumgebungen für Entwicklung und Ausführung des Softwareproduktes, die verwendeten Modellierungs- und Programmiersprachen, etc. Die genannten Faktoren sind bei der Strukturierung eines Prozesses je nach Erfordernis des Projektes zu berücksichtigen.

An einem Softwareentwicklungsprozess sind i.d.R. mehrere Personen beteiligt, die in verschiedenen Teilvorgängen Ergebnisse in Form von Dokumenten produzieren (vgl. Sommerville [96]). Ein solcher Arbeitsprozess, auch Workflow genannt, wird durch ein *Prozessmodell* systematisiert. Derartigen Modellen liegen einige grundlegende Begriffe zugrunde. Einzelne Vorgänge werden als *Aktivitäten* bezeichnet, bei denen Ergebnisse vorhergehender Aktivitäten zu neuen Ergebnissen verarbeitet werden. Die Ergebnisdokumente werden allgemein als *Artefakte* bezeichnet.

Organisatorische Einheiten wie Personen oder Gruppen von Personen übernehmen die Bearbeitung von Aktivitäten. Um von konkreten organisatorischen Einheiten im Prozessmodell zu abstrahieren, werden Aktivitäten *organisatorische Rollen* zugeordnet. Diese Rollen können dann in Anwendung des Prozessmodells den konkreten organisatorischen Einheiten zugeordnet werden. In der weiteren Diskussion stehen inhaltliche Fragen bezüglich der Aktivitäten und Artefakte im Mittelpunkt. Die Frage, wie organisatorische Rollen im Rahmen des Projektmanagements zugeordnet werden, wird in dieser Arbeit ausgeklammert.

Zwischen Aktivitäten bestehen sowohl Daten- als auch Kontrollabhängigkeiten, die im Prozessmodell ausgedrückt werden müssen. Der Abschluss einer oder mehrerer Aktivitäten wird in Softwareentwicklungsprozessen oft als *Meilenstein* bezeichnet.

Die soeben beschriebenen Begriffe sind nicht spezifisch für Software, sondern finden in vielen (Ingenieur-)Disziplinen im Rahmen des Projektmanagements Anwendung, um systematisch eine Aufgabe, wie beispielsweise eine Produkterstellung, zu bewältigen. Für Softwareprozessmodelle ist genauer zu klären, welche spezifischen Aktivitäten und Artefakte im Rahmen von agentenorientierter Softwareentwicklung notwendig sind.

Artefakte von Aktivitäten sind vor allem Modell- oder Implementierungsdokumente, die unterschiedliche Beschreibungen für das zu erstellende System bilden (vgl. Jacobsen [70]).

In Softwareentwicklungsprozessen gibt es fundamentale, aufeinander aufbauende Aktivitäten. Diese entstehen kanonisch dadurch, dass ein gegebenes Problem zu strukturieren, zu analysieren und innerhalb der durch die Systemumgebung gesetzten Randbedingungen zu lösen ist. Daraus ergeben sich die grundlegenden, voneinander abgegrenzten Fundamentalaktivitäten in Softwareentwicklungsprozessen, nämlich Anforderungsspezifikation, Analyse, Entwurf, Implementierung und schließlich die Verwendung und die

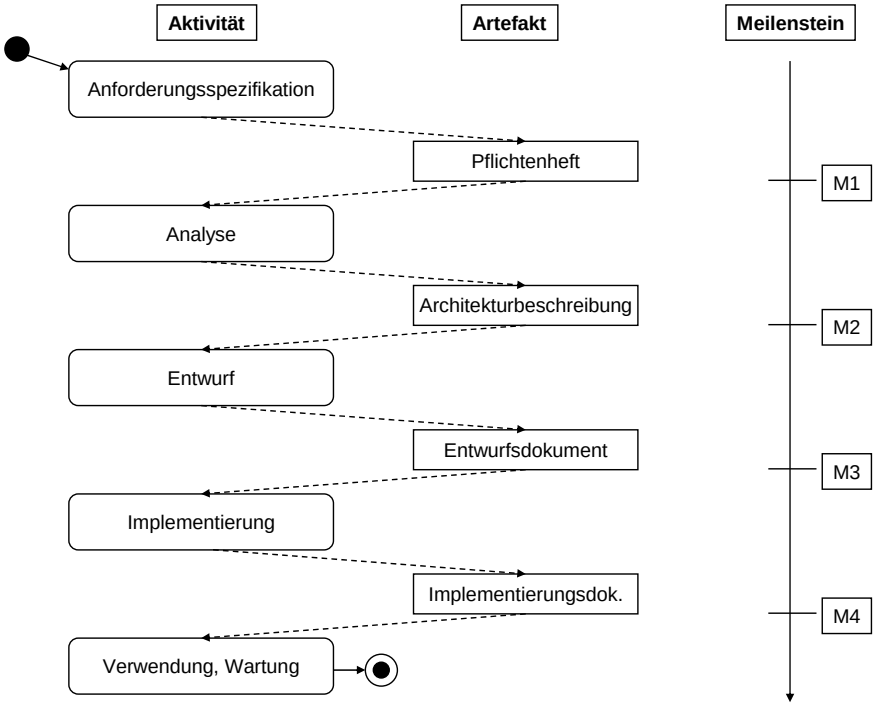

Figure 4.2: Fundamentalaktivitäten und Artefakte

Wartung der Software (vgl. Sommerville [96], siehe Abb. 4.2).

4.3.2 Fundamentalaktivitäten und Artefakte

Um den Erfordernissen von Systemnutzern möglichst nahe zu kommen, ist zu Beginn der Entwicklung immer zu klären, welche neuen Anforderungen an ein vorhandenes oder neu zu erstellendes Softwaresystem existieren. Dazu wird in einer ersten Aktivität ein Modell des Problembereichs erstellt, in dem Informationen über das zu bearbeitende *Geschäftsfeld* strukturiert werden. Die Dynamik des Geschäftsfeldes wird in einem *Geschäftsprozessmodell* festgehalten. Diese Aktivität wird als Ist-Analyse bezeichnet. In eine solche Ist-Analyse kann auch ein schon existierendes und zu erweiterndes Softwaresystem eingeschlossen sein. Darauf aufbauend wird das Soll-Konzept entwickelt, durch das *neue* Anforderungen an das System in Form von Produktfunktionen und -charakteristika festgelegt werden. Die Aktivitäten Ist-Analyse und Soll-Konzept-Erstellung bilden zusammen die *Anforderungsspezifikation*, deren Ergebnis das *Pflichtenheft* ist.

Die Anforderungen sind aus Benutzersicht formuliert und werden im Hinblick auf ihre Eindeutigkeit, Konsistenz und Vollständigkeit in einem weiteren Schritt analysiert. Im Zuge der Aktivität *Analyse* wird ein Analysemodell aufgebaut, das eine abstrakte Form des Softwaresystems bildet und dabei die im Soll-Konzept vorgegebenen Anforderungen berücksichtigt und insbesondere geforderte Abläufe und Strukturen integriert. Dabei

werden in den Anforderungen vorhandene Inkonsistenzen und Redundanz eliminiert. Im Unterschied zu Modellen der Anforderungsspezifikation werden in Analysemodellen notwendigerweise Konzepte verwendet, um *Softwaresysteme* zu strukturieren, wie beispielsweise Funktionen, Module, Klassen, Kollaborationen, etc. Die auf Basis dieser Konzepte beschriebenen Strukturen und Abläufe stellen ein erstes Modell für die *Architektur* des Softwaresystems dar (vgl. [54]). Es abstrahiert aber noch vollständig von den Randbedingungen der konkreten Systemumgebung. Nur falls diese Randbedingungen als Anforderungen in der Anforderungsspezifikation genannt sind, werden sie auch im Analysemodell berücksichtigt. Das entstehende Dokument wird als *Architekturbeschreibung* bezeichnet.

Um zu einer Implementierung des Softwaresystems zu kommen, ist eine Verfeinerung des Analysemodells zu einem Entwurfsmodell durch die Aktivität *Entwurf* notwendig. Der Zweck eines Entwurfsmodells besteht darin, Entscheidungen über die Strukturierung der Implementierung zu treffen und dabei Randbedingungen der Systemumgebung zu berücksichtigen. Die Dynamik des Systems soll möglichst vollständig und detailliert im entstehenden *Entwurfsdokument* beschrieben sein.

Aufbauend auf dem Entwurfsmodell werden in der separaten Aktivität *Implementierung* Implementierungsdokumente in einer Programmiersprache erstellt, aus denen ausführbare Software durch einen Compiler erzeugt wird.

Eine weitere wichtige Aktivität ist die schon genannte *Wartung* von Softwaresystemen, die während der Nutzungsdauer der Software erfolgt. Diese Aktivität dient dazu, während der Nutzung der Software erkannte Fehler zu beheben, die Software an neue oder veränderte Systemumgebungen oder funktionale Anforderungen anzupassen, etc.

Das *Testen* von Software gehört ebenso zu den wichtigen Aktivitäten. Mit Tests sollen Modelle oder Implementierungsdokumente darauf überprüft werden, ob sie die in der Anforderungsspezifikation geforderten Anforderungen erfüllen. Testaktivitäten beziehen sich deshalb auf Artefakte der Aktivitäten Analyse, Entwurf, Implementierung und Wartung. Tests finden entweder als Teilaktivitäten der genannten Aktivitäten statt, wie beispielsweise Komponententests, oder sie sind unabhängige Aktivitäten, die nach Abschluss von Artefakte produzierenden Aktivitäten beginnen, wie beispielsweise Integrationstests.

Diese Arbeit beschränkt sich auf die Aktivitäten Anforderungsspezifikation, Analyse und Entwurf. Diese Aktivitäten und die in ihnen erstellten Dokumente sollen zuerst den Anforderungen genügen, die durch allgemeine, interne Softwarequalitäten gegeben sind. Darauf geht der nächste Abschnitt ein. Danach werden in Abschnitt 4.3.4 die agentenspezifischen Anforderungen an Aktivitäten und Dokumente präsentiert.

4.3.3 Allgemeine Anforderungen: Interne Softwarequalitäten

Interne Softwarequalitäten beziehen sich nach Ghezzi et al. respektive Meyer [55, 83] auf die Entwicklung und die Wartung von Modellen. Insbesondere für die initiale Entwicklung von Software sind die *Verständlichkeit* und die *Überprüfbarkeit* der Modelle wichtig. Für die Wartbarkeit sind *Änderungsfreundlichkeit*, *Erweiterbarkeit* und *Portabilität* wichtig.

Die *Verständlichkeit* der während der Entwicklung erzeugten Dokumente wird heute durch visuelle Modellierungssprachen wie UML verbessert. Die *Überprüfbarkeit* von Modellen hängt von der Verknüpfung des Entwurfsmodells mit den Anforderungen ab, die dem Pflichtenheft zu entnehmen sind. So ist eine formale Verifikation möglich, falls die Anforderungen ebenfalls formal genug formuliert sind. Falls das Pflichtenheft allerdings eher exemplarisch formuliert ist, so werden daraus Testfälle konstruiert, denen Analyse- und Entwurfsmodell genügen müssen.

Besonders wichtig sind die mit der Wartung zusammenhängenden Softwarequalitäten. Da Softwareentwicklung i.d.R. iterativ und inkrementell verläuft, sind zur Wartung gehörige Aktivitäten auch schon bei der erstmaligen Erstellung eines Softwaresystems notwendig. Durch *Änderungsfreundlichkeit* wird erreicht, dass sich Änderungen in Teilen eines Entwurfsmodells wesentlich nur in diesen Teilen auswirken. *Erweiterbare* Entwurfsmodelle ermöglichen die Anpassung an zusätzliche Anforderungen, ohne das Ausgangsmodell grundlegend zu verändern. *Portabilität* unterstützt die Modifikation eines Entwurfsmodells im Hinblick auf veränderte Systemumgebungen.

Um die internen Produktqualitäten Änderungsfreundlichkeit, Erweiterbarkeit und Portabilität zu fördern, haben sich im Laufe der letzten Jahrzehnte die wichtigen Entwurfsprinzipien *Modularisierung*, *Abstraktion* und *separation of concerns* herausgebildet. Diese Prinzipien sind bereits in Kapitel 3 im Zusammenhang mit der Beherrschung komplexer Systeme diskutiert worden. Dass die genannten Produktqualitäten eng mit der Beherrschung komplexer Systeme verknüpft sind, ist evident.

Die Akzeptanz von Prozessmodellen ist in der industriellen Praxis bisher eher gering (vgl. Fugetta [53]). Die Inflexibilität und Einschränkung der Entwickler verbunden mit der mangelnden Unterstützung beim Umgang mit konkreten Dokumenten in herkömmlichen Prozessen führte nach Fugetta dazu, dass Software oft ohne derartige Werkzeuge entwickelt wird. Demgegenüber werden Werkzeuge wie Konfigurationsmanagementsysteme, die den Umgang mit Artefakten erleichtern und damit dem Entwickler Routinetätigkeiten abnehmen, vielfach eingesetzt. Daraus folgt für Softwareprozessmodelle und zugehörige Werkzeuge, dass interne Softwarequalitäten darin weitgehend automatisiert gewährleistet werden sollten.

Konkret hängen die Verwendbarkeit und die Überprüfbarkeit von Modellen sowie auch die Werkzeugunterstützung eng mit dem Grad der Formalisierung von Modellen zusammen. Prozessmodelle sollten deshalb beschreiben, wie UML-Diagramme für die Bildung von Modellen in den grundlegenden Aktivitäten verwendet werden. Dabei ist insbesondere festzulegen, welche syntaktischen und semantischen Beziehungen zwischen verschiedenen Diagrammen bestehen sollen. Die Dynamik des Prozesses ist durch die Bildung von Diagrammen unter Berücksichtigung vorhandener Diagramme bestimmt. Die statischen und dynamischen Beziehungen sollen formalisiert werden, um eine Werkzeugunterstützung des Prozesses zu ermöglichen.

Die allgemeinen Aufgaben und Anforderungen lassen sich wie folgt zusammenfassen:

- Definition der Aktivitäten Anforderungsspezifikation, Analyse und Entwurf mit den Ergebnisdokumenten Pflichtenheft, Architekturbeschreibung und Entwurfsdoku-

ment

- Einsatz visueller Modellierungssprachen für bessere Verständlichkeit der Modelle
- Verwendung der Entwurfsprinzipien Separation of Concern, Abstraktion und Modularisierung für Wartbarkeit und Beherrschung komplexer Systeme
- Formalisierung zur automatisierten Gewährleistung von Softwarequalitäten

Die bisherige allgemeine Diskussion von Anforderungen an ein Prozessmodell wird im nächsten Abschnitt im Hinblick auf die agentenorientierte Softwareentwicklung verfeinert.

4.3.4 Anforderungen an ein agentenorientiertes Prozessmodell

Im Fokus dieser Arbeit steht die modellbasierte Entwicklung agentenbasierter Systeme. Daraus resultiert die Konzentration auf die Aktivitäten Anforderungsspezifikation, Analyse und Entwurf mit den zugehörigen Dokumenten Pflichtenheft, Architekturbeschreibung und Entwurfsdokument. Die wesentliche Aufgabe besteht darin zu klären, welche Konstituenten in den unterschiedlichen Dokumenten berücksichtigt werden sollen. Das erste Dokument ist das Pflichtenheft.

Pflichtenheft. Die Aufgabe des Pflichtenheftes für agentenbasierte Systeme ist es, das Entwicklungsziel, den Einsatzbereich und Anforderungen an die Software zu beschreiben (vgl. Ghezzi [55]).

Das Entwicklungsziel ist durch eine verbale Beschreibung der Aufgaben des Systems gegeben. Der Einsatz- oder Problembereich gliedert sich in Geschäftsfeld und Geschäftsprozesse. Die Anforderungen bestehen aus Produktfunktionen und Produktcharakteristiken. Eine Produktfunktion dient dem Erledigen einer Aufgabe und damit dem Erreichen des Entwicklungsziels. Die Nutzer des Systems werden als Aktoren bezeichnet.

Durch den Begriff *Use Case* werden für eine Produktfunktion Vor- und Nachbedingungen, wesentliche Funktionsmerkmale, beteiligte Aktoren, die Beziehungen zu anderen Use Cases und evtl. der Ablauf beschrieben (vgl. Jacobsen [70]). Ein Use Case ist mit einem funktionalen Ziel verbunden, nämlich der Produktfunktion. Dieses funktionale Ziel wird im Kontext agentenbasierter Systeme als Konstituente proaktiven Verhaltens angesehen (vgl. Abschnitt 4.2.2). Die Aufgaben des Pflichtenheftes für agentenbasierte Systeme sind insgesamt allerdings nicht agentenspezifisch:

- Das Entwicklungsziel verbal beschreiben
- Das Geschäftsfeld und die Geschäftsprozesse beschreiben
- Aus den verbalen Entwicklungszielen Produktfunktionen ableiten
- Die Randbedingungen der Produktfunktionen in Use Cases zusammenfassen

Architekturbeschreibung. Aufbauend auf dem Pflichtenheft wird die agentenbasierte Architektur des Systems beschrieben. Die Aufgabe der Analyse ist es, Produktfunktionen zu analysieren und eine Softwarearchitektur zu erstellen (vgl. Ghezzi [55]):

- Die Produktfunktionen sind anhand zu definierender Kriterien zu gruppieren. Alle Produktfunktionen einer Gruppe sollen möglichst genau einer Komponente, d.h. einem Agenten zugewiesen werden.

- Für Produktfunktionen, die mehreren Agenten zugeordnet werden müssen, ist die erforderliche Interaktion zu beschreiben. Im Kontext agentenbaasierter Systeme erfolgt Interaktion durch Protokolle (vgl. Abschnitt 4.2.2). Falls für eine nichtlokale Produktfunktion ein Protokoll bekannt ist, wird dieses den beteiligten Komponenten zugeordnet.

- Die agentenbasierte Softwarearchitektur soll das zu entwickelnde Softwaresystem in Bezug auf die Anforderungen eindeutig und vollständig beschreiben, d.h. alle Produktfunktionen und Use Cases müssen Agenten zugeordnet werden.

Im Entwurf wird diese Architekturbeschreibung zu einem Entwurfsdokument verfeinert.

Entwurfsdokument. Die Aufgabe des Entwurfs ist es, im Grobentwurf die Komponentenstruktur, die Komponenteninteraktion und Protokolle für die Komponentenbenutzung festzulegen. Im Feinentwurf werden die Struktur und das Verhalten der Komponenten weiter verfeinert (vgl. Ghezzi [55]). Der Grobentwurf entfällt für agentenbasierte Systeme dadurch, dass bereits in der Analyse Komponenten in Form von Agenten eingeführt wurden.

Ein agentenbasiertes System wird gewöhlich mit Hilfe einer Agentenplattform implementiert, die die nebenläufige Ausführung von Agenten und ihre Interaktion unterstützt (vgl. Kapitel 9). Agenten sind die einzigen Komponenten in einem agentenbasierten Softwaresystem, d.h. der Entwurf und die Implementierung eines agentenbasierten Systems reduzieren sich auf die Entwicklung verschiedener Arten von Agenten und auf die Verwendung vorhandener oder die Ausarbeitung neuer Protokolle.

Genauer ergeben sich die folgenden Aufgaben:

- Im Feinentwurf wird für neu zu entwicklende Protokolle das Verhalten der einzelnen Agenten in Bezug auf das Protokoll zu beschreiben.

- Für lokal von einem Agenten zu realisierende Produktfunktionen ist eine detaillierte Verhaltensbeschreibung anhand der Anforderungen gegebener Use Cases auszuarbeiten.

- Verschiedene Protokollverhalten jedes Agenten sind zu einem Gesamtverhalten zu integrieren, das konform zu den Anforderungen des Pflichtenheftes ist.

4.4 Fazit

Die Sprachstruktur von Modellierungssprachen wurde diskutiert. Der Begriff der Modellkonstituente wurde als Träger wesentlicher Konzepte einer Modellierungssprache identifiziert. Auf Basis der Grundlagen aus Kapitel 2 wurden für die Konstituenten der Konzepte Autonomie, Strukturierte Interaktion und Proaktivität Anforderungen festgelegt. Die entwickelten Anforderungen an Modellkonstituenten und deren Komposition sind in Tabelle 4.3 zusammengefasst.

Die Verwendung einer Modellierungssprache wird in einem Prozessmodell beschrieben. Interne Softwarequalitäten führen zu generellen Anforderungen an ein Prozessmodell. Auf agentenbasierte Systeme werden die Anforderungen dadurch spezialisiert, dass für die fundamentalen Aktivitäten Anforderungsbeschreibung, Analyse und Entwurf beschrieben wird, welche Aspekte der Konstituenten agentenbasierter Systeme zu erstellen sind. Methodisch werden dabei generelle Konstituenten von Prozessmodellen mit Agenten-Konstituenten identifiziert. Der Begriff der Produktfunktion ist ein solches Bindeglied. Wenn Produktfunktionen nur durch Vorbedingungen und Effekte spezifiziert sind, dann werden sie in der Anforderungsbeschreibung als Ziel aufgefasst. Die Anforderungen an Aktivitäten eines Modellierungsprozesses zur Entwicklung aller Konstituenten agentenbasierter Systeme sind in den Tabellen der Abbildung 4.4 zusammengefasst. Die obere Tabelle enthält allgemeine und die untere spezifischere Anforderungen an einen agentenspezifischen Modellierungsprozess.

Konzepte	**Konstituenten**	**Anforderungen**
Autonomie	Ausführungsfaden von Agenten (Handlungsautonomie)	- Jeder Agent besitzt einen eigenen Ausführungsfaden, aus dem ein von anderen Agenten unabhängiger Kontrollfluss resultiert.
	Flexible Kontrollelemente (Entscheidungsautonomie)	- Das Verhalten wird anhand der eingehenden Nachrichten, des Umgebungszustands und des inneren Zustands in variabler und komplexer Weise beeinflusst
Strukturierte Interaktion	Protokolle	- Die Funktion (Semantik) eines Protokolls muss eindeutig beschrieben sein. - Die Voraussetzungen über die Umgebung müssen definiert sein. - Das ``Vokabular" und der Aufbau (die Syntax) der Nachrichten sind festzulegen. - Prozedurale Regeln (das Verhalten) für den Nachrichtenaustausch sind anzugeben
Proaktivität	Ziele	- Voraussetzungen für das Erreichen eines Zieles angeben. - Effekt, der mit dem Erreichen eines Zieles verbunden ist, beschreiben.
	Strategien	- Eine Strategie soll das Verhalten eines oder mehrerer Agenten beschreiben. - Jeder Strategie ist (mindestens) ein Ziel zugeordnet, das durch ihre Anwendung erreicht werden kann. - Bezüglich der zugeordneten Ziele ist für eine Strategie zu zeigen, dass sie Gewinnstrategie ist.
Komposition		**Anforderungen**
Autonomie & Interaktion		- Die Bindung von Protokollen an Agenten und die damit einhergehende Verhaltensmodifikation der Agenten ist zu beschreiben. - Ein Verfahren zur Bindung mehrerer Protokolle an einen Agenten ist festzulegen.
Autonomie & Proaktivität		- Ein proaktiver Agent muss handlungsautonom, d.h. insbesondere aktiv sein. - Die Art der Einschränkung von Entscheidungsautonomie durch Strategien muss festgelegt sein.
Interaktion & Proaktivität		- Dekomposition von Kommunikationszielen in lokale Ziele beschreiben. - Auswirkung der Protokollbindung auf lokale Strategien berücksichtigen.

Figure 4.3: Anforderungen an agentenspezifische Modellkonstituenten und deren Komposition

Generelle Anforderungen an ein agentenorientiertes Prozessmodell	
Anforderungsspezifikation	
Analyseaktivität	
Entwurfsaktivität	
Einsatz visueller Modellierungssprachen	
Verwendung der Entwurfsprinzipien Separation of Concern, Abstraktion und Modularisierung	
Formale Beschreibung der Entwicklungsschritte	
Entwicklungsschritte für Konstituenten zu Konzepten von Agenten	Autonomie
	Strukturierte Interaktion
	Proaktivität

Fundamentalaktivitäten → Dokumente	**Anforderungen**
Anforderungsspezifikation → Pflichtenheft	- Entwicklungsziel, Geschäftsfeld und die Geschäftsprozesse beschreiben - Produktfunktionen ableiten - Randbedingungen der Produktfunktionen in Use Cases zusammenfassen
Analyse → Architekturbeschreibung	- Produktfunktionen anhand zu definierender Kriterien gruppieren. - Nicht gruppierbare Produktfunktionen mehreren Gruppen zuweisen - Zu jeder Gruppe einen Agenten bilden - Für Produktfunktionen mit bekannten Kommunikationsstrategien in Form von Protokollen diese an Agenten binden
Entwurf → Entwurfsdokument	- Für neu zu entwickelnde Protokolle das Verhalten der einzelnen Agenten in Bezug auf das Protokoll beschreiben. - Für lokal von einem Agenten zu realisierende Produktfunktionen Verhaltensbeschreibung anhand der Use Cases ausarbeiten. - Verschiedene Protokollverhalten jedes Agenten zu einem Gesamtverhalten integrieren, das konform zu Use Cases ist.

Figure 4.4: Anforderungen an einen agentenspezifischen Modellierungsprozess

Chapter 5

Ansätze für agentenorientierte Softwareentwicklung

Im letzten Kapitel sind Kriterien entwickelt worden, denen eine agentenorientierte Modellierungssprache und ein zugehöriges Prozessmodell genügen sollten. In diesem Kapitel wird gezeigt, dass keiner der bisher entwickelten, oftmals zitierten Ansätze die essentiellen Kriterien erfüllen. Untersucht werden zu diesem Zweck die Kernsprache UML ohne Profil-Erweiterungen, AgentUML [7], MaSE [104], TROPOS [84], GAIA [108] und MAS-CommonKADS [67]. Die Ansätze beinhalten z.T. nicht nur eine Modellierungssprache sondern auch ein Prozessmodell. Für die etablierten Prozessmodelle Rational Unified Process und Catalysis wird untersucht, inwieweit sie im Kontext agentenorientierter Softwareentwicklung verwendet werden könnnen.

5.1 UML

In den letzten 15 Jahren hat sich das Paradigma der Objektorientierung durchgesetzt. Neben objektorientierten Programmiersprachen wie C++ und JAVA haben sich auch objektorientierte Modellierungssprachen, aufbauend auf frühen Vorschlägen wie OMT (vgl. Rumbaugh [93]), OOSE (vgl. Jacobson [70]) und der Booch-Methode (vgl. Booch [11]), in Gestalt der Unified Modeling Language (UML) durchgesetzt (vgl. [86]). Die Unified Modeling Language (UML) hat den Anspruch, für Modellierungsaufgaben universell einsetzbar zu sein. Nach einem Überblick über die konkrete und abstrakte Syntax wird diskutiert, inwieweit sich Konstituenten agentenbasierter Systeme mit UML modellieren lassen.

Konkrete Syntax: Diagrammarten und ihre typische Verwendung.

Use Cases, auch Anwendungs- oder Nutzfälle genannt, beschreiben für den Benutzer relevante Funktionen des Systems. Use Case Diagramme dienen dazu, Systemfunktionen vergleichsweise abstrakt zu beschreiben, wobei die Nutzer der Funktionen in Form von

Akteuren (actors) modelliert werden.

Bleiben die Use Cases und die Implementierungsdiagrammtypen außen vor, so lassen sich die übrigen Diagrammtypen der UML den Aufgaben der Beschreibung der statischen Struktur und des Verhaltens eines Softwaresystems zuordnen. Die statische Struktur wird durch Klassendiagramme beschrieben. Durch die Object Constraint Language können hier zusätzliche Angaben gemacht werden.

Verhaltensdiagramme werden unterschieden nach ihren Aufgaben: Statecharts und die ihnen ähnlichen Aktivitätendiagramme dienen der dynamischen Modellierung und beschreiben mögliche Abläufe im System. Statecharts sind eine Erweiterung von Zustandsdiagrammen. Sie definieren die mögliche Folge von Zuständen und Operationsaufrufen in Reaktion auf das Eintreten von Ereignissen. Sie können dem System, Teilen davon oder einzelnen Klassen zugeordnet werden. Statecharts sind insbesondere dann erforderlich, wenn Operationen nicht in jedem Zustand ausführbar sind und wenn auf gleiche Ereignisse zustandsabhängig verschieden reagiert werden muß.

Die zweite Gruppe von Verhaltensdiagrammen besteht aus Sequenz- und Kollaborationsdiagrammen. Bei ihnen liegt der Schwerpunkt darauf zu beschreiben, welche Daten verändert werden und wie die Interaktion zwischen Objekten durch den Austausch von Nachrichten über die Zeit erfolgt. Verwendet werden dabei Objekte und Methoden von Klassen aus Klassendiagrammen.

Abstrakte Syntax: Sprachstruktur.

Visuelle Modellierungssprachen wie UML haben eine grafische konkrete Syntax. Die nichtsequentielle Ausprägung der Sprache hat dazu geführt, dass die abstrakte Syntax nicht wie bei textuellen Sprachen durch eine kontextfreie Grammatik und zusätzliche kontextsensitive Regeln, sondern durch ein Metamodell definiert wird. Modelle sind Instanzen des Metamodells und bilden die Worte der Sprache. Neben dem Metamodell existieren in UML Restriktionen für Modelle, die durch Bedingungen in der Object Constraint Language (OCL) formuliert sind. Diese Einschränkungen entsprechen kontextsensitiven Bedingungen für textuelle kontextfreie Sprachen.

Das Metamodell von UML 1.5 ist mit einfachen Klassendiagrammen, einer Teilsprache von UML selbst, notiert [86]. Die Sprachdefinition besteht weiterhin aus kontextsensitiven Wohlgeformtheitsregeln, ausgedrückt durch Regeln in der Sprache OCL. Rein verbale Aussagen zur "Semantik" in der UML-Spezifikation beschreiben partiell den Umgang mit Modellen, also die (informelle) Pragmatik von UML. Das Fehlen einer formalen Semantik von UML führt zu einigen Problemen. So ist der Gegenstandsbereich der Modelle unklar, d.h. es wird nicht festgelegt, *was* modelliert wird. Da semantische Beziehungen zwischen Teilmodellen fehlen, ist deren Konsistenz unbestimmt. Weitere Probleme bestehen darin, dass die rein verbale Beschreibung inhärent mehrdeutig ist.

Ein weiteres Merkmal von UML ist die Möglichkeit zur Erweiterung der Sprache. Mit Stereotypen, sogenannten *tagged values* und Bedingungen können Modellelemente spezifisch modifiziert werden. Die Erweiterungen einer Kernsprache können weiter strukturiert werden [47]. Domänenspezifische Erweiterungen beinhalten Sprachelemente, die für bes-

timmte Anwendungsbereiche charakteristisch sind. Anwendungsspezifische Erweiterungen gehen über domänenspezifische hinaus und erfolgen im Kontext einzelner Anwendungen.

Eignung für die Agenten-Konstituenten-Modellierung.

Entsprechend den Kriterien aus Kapitel 4 werden Sprachkonstrukte benötigt, um die Agentenkonzepte strukturierte Interaktion, Autonomie und Proaktivität zu unterstützen. Dazu ist im Zusammenhang mit UML zu untersuchen, inwieweit angebotene Verhaltensdiagramme verwendet werden können.

Objektverhalten, dessen Flexibilität daraus resultiert, dass die Auslösungsmechanismen für Reaktionen komplex sind, kann in UML mit Event-Condition-Action-Regeln (ECA-Regeln) an Transitionen von Statecharts modelliert werden. Diese werden in der Form *event-signature [guard-condition] / action-expression* an Transitionen notiert (vgl. [86], Seite 3-146). Events (*event-signature*) sind beispielsweise der Aufruf von Methoden, Conditions (*[guard-condition]*) können durch OCL-Ausdrücke gegeben sein und eine Action (*action-expression*) kann wiederum eine Methoden (einer anderen Klasse) sein, die ggfs. aufgerufen wird. Die Auswertung der Regeln erfolgt immer in der Reihenfolge Event - Condition - Action. Bei den ECA-Regeln können einzelne Teile fehlen, z.B. muß kein Event auftreten. Nicht modellierbar sind Regeln, bei denen eine Vorbedingung getestet wird und danach erst auf Ereignisse reagiert wird (vgl. Reggio [91]). Insofern sind die Möglichkeiten zur Modellierung von Entscheidungsabläufen eingeschränkt. Entscheidungsautonomes Verhalten kann damit nicht uneingeschränkt modelliert werden.

Agenten sind handlungsautonom und deshalb aktiv, d.h. Agenten führen Zustandsübergänge auch ohne die Aktivierung durch ein extern erzeugtes Ereignis aus. Handlungsautonom sind in UML lediglich aktive Objekte, die einen eigenen Kontrollfluß besitzen. Mit Statecharts ist es möglich, das Verhalten solcher Objekte zu beschreiben.

Agenten koordinieren sich durch Kommunikationsprotokolle. Damit einher gehen auch Veränderungen der lokalen Struktur und des Zustands der Komponente. Rollen stellen ein Konzept dar, um das lokale Verhalten und zugehörige Zustandsveränderungen von Agenten im Zusammenhang mit Protokollen zu beschreiben. In UML fehlen die Mittel, derartige Kommunikationsprotokolle auf der Basis eines adäquaten Rollenkonzeptes zu beschreiben (vgl. [28, 36]).

Sprachmittel für Ziele und Strategien sind in UML nicht vorhanden. Es existieren allerdings UML-Profile, die Ziele als Stereotype für Use Cases einführen. Die entsprechenden Modellierungsansätze TROPOS und MAS-CommonKADS werden in nachfolgenden Abschnitten dieses Kapitels diskutiert.

Insgesamt wird keines der Agentenkonzepte Autonomie, Proaktivität und strukturierte Interaktion von UML ausreichend unterstützt. Als Konsequenz sind verschiedene Modellierungssprachen und -prozessmodelle vorgeschlagen worden.

5.2 AgentUML

Ein vielfach zitierter Vorschlag für eine agentenorientierte UML-Erweiterung ist AgentUML [7]. Darin werden Protokolldiagramme zur Beschreibung strukturierter Interaktion eingeführt. Für die Agentenkonzepte Autonomie und Proaktivität werden keine neuen Sprachelemente vorgeschlagen.

Die Organisation *Foundation for Intelligent Physical Agents* (FIPA) verfolgt seit 1996 das Ziel, Spezifikationen für interoperable Agentensysteme zu produzieren. AgentUML ist als visuelle Notation für Protokollbeschreibungen in einen FIPA-Standard eingeflossen (vgl. [89]).

Die Protokolldiagramme aus AgentUML erweitern UML-Sequenzdiagramme. Die Erweiterungen bestehen darin, dass sowohl Lebenslinien als auch Interaktionspfeile verzweigen können. An den Verzweigungspunkten wird einer der booleschen Operatoren AND, XOR oder OR notiert. Dadurch sollen parallel oder alternativ ausgeführte Lebenslinien und Aktivitäten zusammen mit entsprechenden Nachrichtenflüssen ausgedrückt werden. Das Beispiel eines English-Auction-Protokolls in Abbildung 5.1 zeigt die Erweiterungen. Das Protokolldiagramm im Beispiel ist als Template notiert, das mit Ein- und Ausgabeparametern versehen ist. Anhand der so definierten Schnittstelle von Protokolldiagrammen sollen Protokolle ineinander verschachtelt werden. Ein weiteres Merkmal von Protokolldiagrammen ist, dass Kardinalitäten an ein- und auslaufenden Interaktionspfeilen zu dem Zweck angebracht werden können, die Anzahl der Sender bzw. Empfänger eines Methodenaufrufs festzulegen.

Eignung für die Agenten-Konstituenten-Modellierung

Die UML-Erweiterung der Sequenzdiagramme zu Protokolldiagramme ist nur informell definiert. Weder wird der Erweiterungsmechanismus von UML verwendet noch wird die Erweiterung auf der Ebene der abstrakten Syntax auf der Basis des UML-Metamodells beschrieben.

AgentUML fehlt weiterhin eine formale Semantik, durch die das durch Protokolldiagramme beschriebene Verhalten eindeutig beschrieben würde.

Insgesamt ist AgentUML eine auf die Syntax einer Diagrammart beschränkte und mit Mängeln behaftete UML-Erweiterung zur Beschreibung strukturierter Interaktion. Da andere Agentenkonzepte wie Autonomie und Proaktivität nicht berücksichtigt werden, erfüllt AgentUML nicht die in Kapitel 4 erarbeiteten Anforderungen.

5.3 MaSE

Wood et al. [104] führen das Prozessmodell "Multi-agent Systems Engineering" (MaSE) ein, das zur Notation von Modellteilen UML verwendet. Es besteht aus einer Analyse- und einer darauffolgenden Entwurfsaktivität (vgl. Abbildung 5.2). Die Analyseaktivität besteht aus drei einzelnen Schritten: Festlegen von Zielen, Formulieren von Use Cases

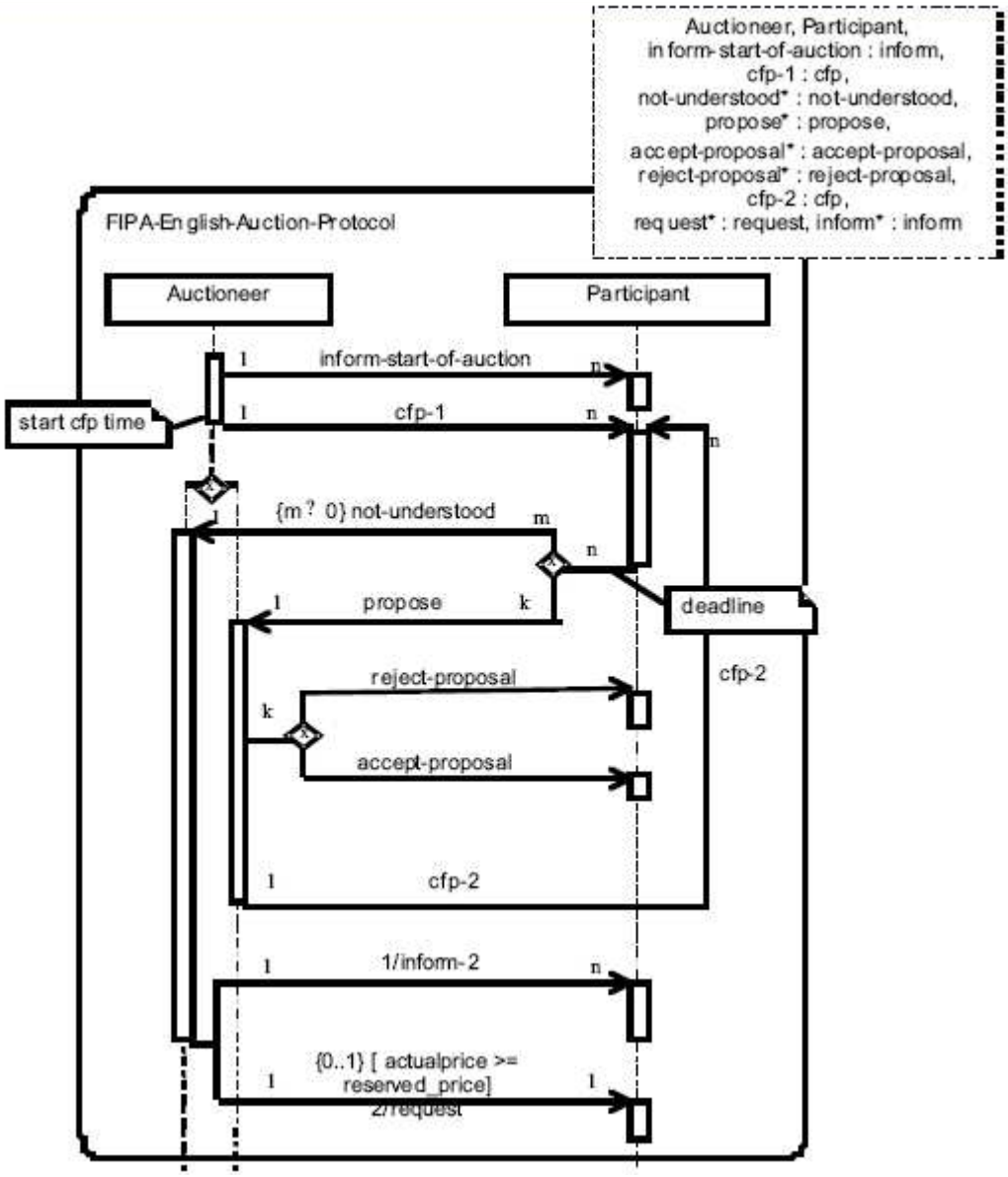

Figure 5.1: Das English-Auction-Protokoll in AgentUML-Darstellung (aus [7])

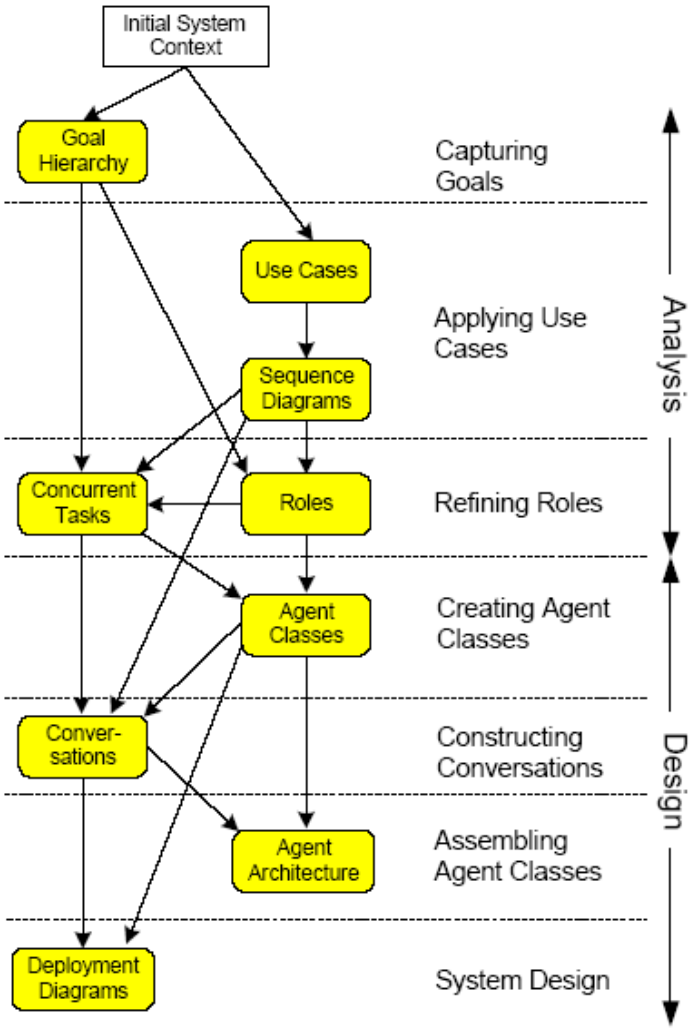

Figure 5.2: MaSE-Aktivitäten (aus [104])

und Verfeinern von Rollen. Die Entwurfsaktivität umfasst die vier Schritte Erzeugen von Agentenklassen, Konstruieren von Interaktionen, Zusammenfügen von Agentenklassen und Systementwurf.

Die Analyseaktivitäten. Ausgangspunkt sind in MaSE textuell kurz umschriebene Anforderungen, die festlegen, *was* das System erreichen soll. Die Anforderungen werden in verschieden detailierte Ziele gegliedert. Ziele werden in Form von Ziel-Hierarchie-Diagrammen notiert und strukturiert. Ziele in höheren Stufen der Hierarchie dienen der Zusammenfassung oder Partitionierung von Teilzielen.

Use Cases werden in MaSE als Folgen von Ereignissen angesehen, die Beispiele dafür bilden, wie ein System sich verhält. Use Cases werden aus der Anforderungsbeschreibung gewonnen. Sie werden durch ein Sequenzdiagramm genauer beschrieben, indem

für die identifizierbaren Teilnehmer von Interaktionen in der Anforderungsbeschreibung Rollen eingeführt werden, zwischen denen ein Interaktionsszenario abläuft. Die einzelnen Interaktionen sind die Ereignisse des zugehörigen Use Case.

In der Analyse-Phase wird das Rollenmodell weiter verfeinert. Dazu werden Ziele Rollen zugeordnet. Zu jedem Ziel erhält eine Rolle ein Ausführungsmodell (Task) in Form eines Petri-Netzes, das das zielgerichtete Verhalten der Rolle beschreibt. Nebenläufige Tasks einer Rolle können untereinander oder mit Tasks anderer Rollen kommunizieren.

Die Entwurfsaktivitäten. Rollen sind die Einheiten, aus denen Agentenklassen zusammengesetzt werden. Eine Rolle umfaßt die Funktionen, für die sie zum Erreichen zugeordneter Agentenziele verantwortlich ist. Falls die Ziele von Rollen im Ziel-Hierarchie-Diagramm in Beziehung stehen oder falls zwischen den Rollen eine intensive Kommunikation abläuft, werden solche Rollen zu Agentenklassen zusammengefasst. In einem Klassendiagramm werden die Agentenklassen zusammen mit den ihnen zugeordneten Rollen in Form von Klassen und Kompositionsbeziehungen notiert.

Die Szenarien aus der Analyse werden in Protokollen (Conversations) integriert, die die Interaktion zwischen zwei Agenten durch Statecharts beschreiben. Die Kommunikation mit Protokollen muss verträglich mit den Tasks der Rollen der beteiligten Agentenklassen sein. Gegebenenfalls sind die Tasks zu erweitern oder zu verändern.

Der Entwurf setzt sich mit der Verwendung einer vorhandenen oder der Entwicklung einer neuen objektbasierten Softwarearchitektur fort. Komponenten einer Architektur bestehen entweder elementar aus Objektklassen mit Attributen und Methoden oder sind komplex strukturiert. Als mögliches Vorgehen wird vorgeschlagen, die Tasks von Rollen der Agenten durch einzelne Komponenten zu realisieren. Agenten setzen sich so aus einer Zahl von Komponenten zusammen. Die Komponenten werden in einem Klassendiagramm notiert.

Der letzte Schritt der Entwurfsphase besteht darin, Agenten zu instanziieren und ihre Verteilung in einer gegebenen Hardwarearchitektur festzulegen. Das Resultat dieses Systementwurfs wird in einem Deployment-Diagramm notiert.

Eignung für die Agenten-Konstituenten-Modellierung

UML-Diagramme werden vielfältig eingesetzt. Entscheidungsautonomie Tasks durch Petri-Netze Handlungsautonomie ist durch aktive gegeben Protokolle entsprechen den conversations, allerdings wird deren funktionales Verhalten (Datenänderunge) wegen der Verwendumg von Statecharts nicht erfasst. Ziele werden explizit beschrieben, bleiben aber formal unklar. Strategien sind durch die Tasks von Rollen gegeben. Der Zusammenhang zwischen Strategien (Tasks, Petri-Netze) und Protokollen (Statecharts) bleibt unklar.

Bewertung des Prozessmodells Die fundamentalen Aktivitäten werden explizit beschrieben. Die Entwicklungsschritte und Zusammenhänge zwischen Teilmodellen werden genannt. Die Schritte betreffen alle Agenten-Konstituenten.

5.4 GAIA

Ein Konzept für die Modellierung mit Rollen schlagen Wooldridge, Jennings und Zambonelli vor [108]. Sie konzentrieren sich auf Rollenmodelle, um Organisationsstrukturen und Verhaltensregeln für Multiagentensysteme auszudrücken. Interaktionen und zugehörige Protokolle sind auf der Ebene der Rollen definiert. Sie bestimmen implizit die Arten sozialen Verhaltens eines Agenten innerhalb einer Organisation von Agenten. Soziale Beziehungen zwischen Rollen werden in einem Rollenmodell definiert. Durch ihre Rollen bekommen Agenten eine wohldefinierte Stellung in einer Organisation.

Gemäß Wooldridge et al. [106] sind einer Rolle Verantwortlichkeiten, Zugriffsrechte, Aktivitäten und Protokolle zugeordnet, die in spezifischen Rollenschemata definiert sind.

Verantwortlichkeiten beschreiben Funktionen einer Rolle. Sie umfassen Lebendigkeitsbedingungen wie die Ausführung einer vorgegebenen Folge von Protokollen, wobei Protokolle wiederum die Interaktion zwischen Rollen festlegen. Protokolle werden mit Hilfe von *Protokollregeln* festgelegt, die durch Rollen beachtet werden müssen. Die Notation von Lebendigkeitsbedingungen leitet sich aus Lebenszyklusausdrücken aus FUSION ab [5]. Sicherheitsbedingungen sind einschränkende Bedinungungen an das Agentenverhalten in Form von Prädikaten.

Zugriffsrechte identifizieren die Ressourcen, die von einer Rolle dazu benötigt werden, Verantwortlichkeiten zu erfüllen.

Ein Protokoll ist eine Beschreibung von Interaktion zwischen Rollen. Ein einzelnes Protokoll enthält die Nachrichten, die zwischen Rollen ausgetauscht werden. Es ist durch seinen Einsatzzweck, die initiierende und antwortende Rolle, Ein- und Ausgaben und den Ablauf, der von der initiierenden Rolle ausgeht, spezifiziert.

Ein Rollenschema wird vorgeschlagen, in dem Eigenschaften von Rollen beschrieben werden (vgl. Abb. 5.3).

Analyseaktivität. Diese Aktivität besteht aus den folgenden Schritten:

1. Identifikation von Schlüsselrollen und informelle Beschreibung jeder Rolle

2. Identifikation von Protokollen, die mit jeder Rolle assoziiert sind und Modellierung eines Rolleninteraktionsmodells

3. Ausarbeitung des Rollenmodells auf der Basis des Interaktionsmodells

Das Rollen- und das Interaktionsmodell werden während der Analyseaktivität erzeugt. Das Rollenmodell besteht aus Rollenschemata und das Interaktionsmodell spezifiziert die Interaktion zwischen Rollen und basiert auf Interaktionsdiagrammen aus FUSION (vgl. [5] pp. 198-203).

Das Vorgehen kann gut mit dem coffee filler-Beispiel von Wooldridge et al. illustriert werden [106]. Das Ziel besteht darin, eine Kaffeekanne voll Kaffee zu halten und

Role Schema: <Name der Rolle>	
Description	<Kurze Beschreibung der Rolle>
Protocols	<Namen der Protokolle, an denen die Rolle beteiligt ist>
Permissions	<Mit der Rolle verknüpfte Rechte>
Responsibilities	
Liveness	<Lebendigkeitsbedingungen>
Safety	<Sicherheitsbedingungen>

Figure 5.3: Rollenschema in Gaia

Interessenten darüber zu informieren, wenn frischer Kaffee aufgebrüht wurde. Die Verantwortlichkeiten (responsibilities) können wie folgt zusammengefasst werden:

- Falls die Kaffeekanne leer ist, brühe neuen Kaffee auf.
- Falls frischer Kaffee aufgebrüht ist, informiere alle Interessenten.

Aus diesen textuellen Anforderungen (responsibilities) kann die folgende Lebendigkeitseigenschaft abgeleitet werden:

$$CoffeeFiller = (Fill.InformWorkers.CheckStock.AwaitEmpty)^{\omega}$$

Das heißt, die CoffeeFiller-Rolle führt zu einer unbeschränkten Wiederholung der Protokolle *Fill*, *InformWorkers*, *CheckStock* und *AwaitEmpty*. Eine Sicherheitseigenschaft besteht darin, dass die Kaffeefüllmenge nie negativ ist. Ausgedrückt wird das durch das Prädikat:

$$coffeeStock \geq 0$$

Das gesamte Rollenschema für die CoffeeFiller-Rolle ist in Abbildung 5.4 dargestellt.

Entwurfsaktivität.

Verschiedene Rollen bilden ein Rollenmodell. Ausgehend vom Rollenmodell werden die folgenden drei Modelle entwickelt:

1. Ein Agentenmodell, das verschiedene Agententypen des Systems enthält.

Role Schema: CoffeeFiller	
Description	This role involves ensuring that coffee is kept filled, and informing the workers when fresh coffee has been brewed.
Protocols	Fill, InformWorkers, CheckStock, AwaitEmpty
Permissions	reads supplied coffeeMaker // name of coffee maker coffeeStatus // full or empty changes coffeeStock // stock level of coffee
Responsibilities	
Liveness	CoffeeFiller = (Fill.InformWorkers.CheckStock.AwaitEmpty)
Safety	coffeeStock >= 0

Figure 5.4: Rollenschema für CoffeeFiller (aus [106])

2. Ein Dienstemodell, das die mit Agentenrollen verknüpften Dienste angibt.

3. Ein Bekanntschaftsmodell (acquaintance), das mögliche Interaktionen (Verbindungen) zwischen Agenten festlegt.

Im Agentenmodell sind eine oder mehrere Rollen zu einem Agententyp vereint. Agententypen enthalten die Kardinalität ihrer Instanzen. Eine Vererbungsbeziehung ist nicht definiert, da gemäß Wooldridge et al. das Agentensystem grobgranular beschrieben ist und lediglich eine kleine Anzahl von Rollen enthält.

Ein Dienst eines Agenten entspricht einer Methode eines Objektes. Dienste werden aus Protokollen des Rollenschemas abgeleitet. Das Dienstemodell identifiziert Eingaben, Ausgaben, Vor- und Nachbedingungen für jeden Dienst, den ein Agent anbietet. Ein- und Ausgaben sowie Vor- und Nachbedingungen hängen von den (permissions) und (responsibilities) im Rollenschema eines Agenten ab. Beispielsweise besitzt der Dienst CheckStock des CoffeeFiller-Agenten eine Vor- und Nachbedingung, dass die Kaffeefüllmenge nicht negativ sein muss, da dies eine Sicherheitsbedingung des CoffeeFiller ist.

Eignung für die Agenten-Konstituenten-Modellierung

Der Ansatz ist textorientiert, d.h. visuelle Sprachen wie UML werden nicht verwendet. Das Konzept der Autonomie wird nicht durch Sprachmittel unterstützt. Strukturierte Interaktion kann in Form von auf einem Rollenkonzept basierenden Protokollen formal beschrieben werden. Ziele sind in Form von Responsibilities modelliert.

Bewertung des Prozessmodells Die fundamentalen Aktivitäten werden nicht explizit beschrieben. Das Bestimmen von Rollen und ihre Aggregation zu Agenten wird

unterstützt.

5.5 MAS-CommonKADS

Das Prozessmodell MAS-CommonKADS von Iglesias et al. [67] ist ein komponentenbasierter Ansatz zur Entwicklung agentenbasierter Systeme. Die erste Aktivität der Analyse behandelt die Konzeptbildung, in der Use Cases anhand von Benutzeranforderungen festgelegt werden. Präzisiert werden die Use Cases durch Sequenzdiagramme. In diesem Zusammenhang werden Rollen festgelegt.

Danach wird eine Anforderungsbeschreibung entwickelt, die aus folgenden Modellen besteht.

- Das Agentenmodell enthält die Charakteristika von Agenten, wie Fähigkeiten zur Schlußfolgerung, Sensoren, Effektoren, Services, Agentengruppen und Hierarchien (Organisationsmodell).
- Das Aufgabenmodell (task model) beschreibt die Aufgaben, die die Agenten ausführen können und schließt Ziele, Zerlegungen und Problemlösungsmethoden ein.
- Das Erfahrungsmodell definiert die Informationsquellen, die von Agenten für das Erreichen ihrer Ziele benötigt werden.
- Das Organisationsmodell definiert die Organisation einer Menge von Agenten.
- Das Koordinationsmodell beschreibt die Kommunikation zwischen Agenten in Form von Protokollen und notwendigen Fähigkeiten.

Die Modelle werden innnerhalb von drei Aktivitäten entwickelt: der Konzeptionsaktivität, der Analyse und dem Entwurf.

Konzeption. Iglesias et al. verwenden Use Cases, um die Interaktionen zwischen dem System und seinen Benutzern zu beschreiben. Message sequence charts (MSCs) dienen zur Beschreibung des Verhaltens der Use Cases. In [68] ist die Konzeptionsaktivität genauer ausgearbeitet. Die User-Environment-Responsibility-Technik (UER-Technik) kombiniert die durch Benutzer, Umgebung und Verpflichtungen (responsibilities) gesteuerte Analyse eines agentenbasierten Systems. Die Benutzer gesteuerte Analyse basiert auf Use Cases und zugehörigen MSCs. Die umgebungsgesteuerte Analyse zielt auf die Identifikation relevanter Objekte in der Umgebung und die Aktionen und Reaktionen des Agenten, welche durch sogenannte Reaction Cases dargestellt werden. Die durch Verpflichtungen (auch Ziele genannt) gesteuerte Analyse erfordert die Erkennung und Benennung von Zielen, die das System ohne direkte Interaktion mit dem Benutzer erfüllen soll. Ziele werden durch Goal Cases repräsentiert. Die Notation basiert auf der für Use Cases in UML.

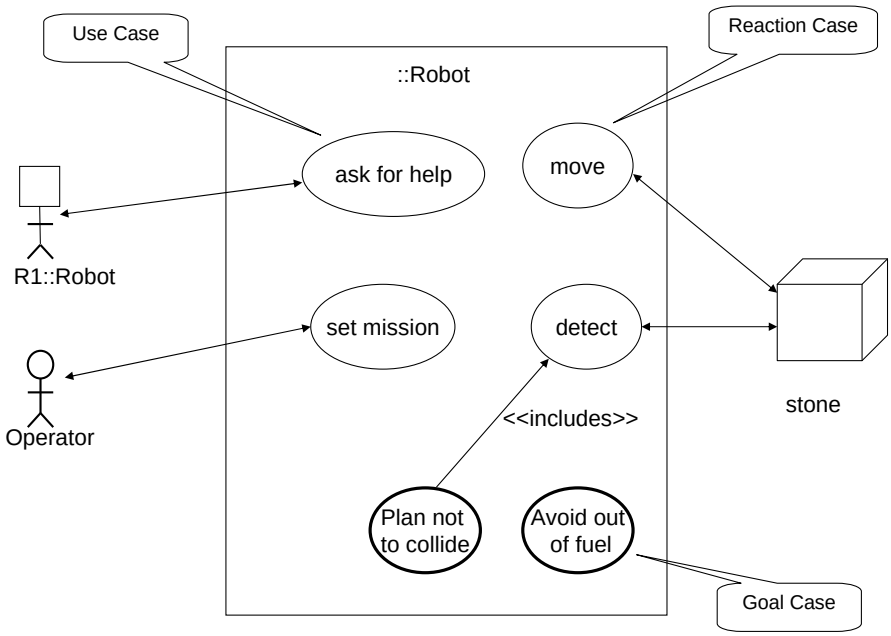

Figure 5.5: Ein UER-Diagramm (aus [79])

Die Abbildung 5.5 zeigt ein Beispiel der UER-Technik. Die traditionellen Use Cases (ask for help und set mission) werden durch den Roboter und den Operator des Roboters benutzt. Ein Teil der Umgebung ist durch den Stein modelliert, der durch den Roboter-Agenten manipuliert werden kann. Der goal case plan not to collide schließt den Reaction Case detect (stone) ein. Das Beispiel ist [79] entnommen.

Die Analyse. Während der Analyseaktivität werden verschiedene Modelle konstruiert (siehe oben). Im Agentenmodell werden Agenten identifiziert und beschrieben. Iglesias et al. schlagen bereits aus der oo-Modellierung bekannte Techniken zur Identifizierung von Agenten vor. In einer modifizierten Version von CRC-Karten werden Ziele, Pläne, Wissen, Kollaborationen und Dienste für jeden Agenten spezifiziert. Zusätzlich werden Use Cases angewandt, um zu beschreiben, wie ein Agent andere Agenten im System verwendet.

Das Koordinationsmodell identifiziert Kommunikationsszenarien zwischen Agenten, Ereignissen (Nachrichten) und Protokollen. Szenarien werden durch MSCs modelliert. Die Repräsentation von Ereignissen und Kommunikationspartnern wird durch sogenannte Ereignis-Fluß-Diagramme beschrieben, die aus Agenten und Nachrichten bestehen und UML-Kollaborationsdiagrammen ähneln. Das interne Verhalten der Agenten während der Interaktion wird durch SDL-Zustandsdiagramme beschrieben.

Das Erfahrungsmodell dient der Modellierung von Wissen. Dieses Wissen gliedert sich in Wissen über den Anwendungsbereich, Inferenzwissen und Aufgabenwissen. Das Anwendungswissen wird durch ein OMT-Objektmodell modelliert. Das Inferenzwissen umfasst

die Inferenzschritte, die zum Lösen einzelner Aufgaben notwendig sind. Das Aufgabenwissen beschreibt, wie Inferenzschritte zum Lösen einzelner Aufgaben zusammengesetzt werden müssen.

Das Organisationsmodell ermöglicht, strukturelle Beziehungen zwischen Agenten mit dem OMT-Objektmodell zu modellieren. Die Aggregationsbeziehung modelliert die Gruppierung von Agenten. Im Unterschied zu Objekten werden für Agenten auch Ziele, Annahmen (beliefs), Pläne, Dienste, Sensoren und Effektoren modelliert.

Eignung für die Agenten-Konstituenten-Modellierung In der Konzeptionsphase werden erweiterte Use Cases entwickelt, durch die die Umgebung, die Reaktion von Agenten und ihre Ziele beschrieben werden. Die weitere Modellierung von Agenten erfolgt in der Analysephase mit Hilfe von CRC-Karten in rein textueller Form. Ziele, Pläne, Teilnehmer und Dienste von Kollaborationen werden notiert. Visuelle Sprachen werden für die Beschreibung der Interaktion (message sequence charts) und des Organisationsmodells (OMT-Klassendiagramm) verwendet.

Die Autonomie der Agenten wird nicht explizit modelliert. Der Zusammenhang zwischen Zielen und Verhaltensbeschreibungen zum Erreichen von Zielen (Strategien) bleibt unklar. Die Interaktion ist wenig strukturiert. Unklar ist auch, wie die UER-Technik das im Entwurf entstehende Agentenmodell beeinflusst.

Bewertung des Prozessmodells Die Beziehungen zwischen den einzelnen Modellen sind unklar. Damit ist kein sinnvolles Prozessmodell erkennbar.

5.6 TROPOS

TROPOS ist ein agentenorientierter Softwareentwicklungsprozess, der die Aktivitäten *Frühe Anforderungsbeschreibung*, *Späte Anforderungsbeschreibung*, *Architekturentwicklung* und *(detaillierter) Entwurf* unterscheidet (vgl. Myopoulos [84]). Der Schwerpunkt liegt auf der Beschreibung und Analyse von Anforderungen, d.h. auf der Aktivität der späten Anforderungsbeschreibung.

TROPOS besitzt die Konstituenten Aktor, Rolle, Ziel, Abhängigkeit und Ressource. Aktoren sind physische Systeme oder Benutzer, Softwareagenten oder Rollen. Eine Rolle ist eine abstrakte Charakterisierung des Verhaltens eines Aktors in einem speziellen Kontext. Rollen werden von Agenten eingenommen. Bei Zielen wird zwischen harten und weichen Zielen unterschieden. Harte Ziele haben klare Erfüllungskriterien, weiche Ziele haben das nicht. Zwischen Aktoren bestehen Abhängigkeiten. Eine Abhängigkeit zwischen zwei Aktoren besagt, dass ein Aktor von einem anderen in Bezug auf ein Abhängigkeitsmerkmal (dependum) abhängt. Diese Merkmale können weiche und harte Ziele, Ressourcen oder Aufgaben sein. Die genannten Konstituenten werden in den vier Hauptaktivitäten des Prozesses verwendet.

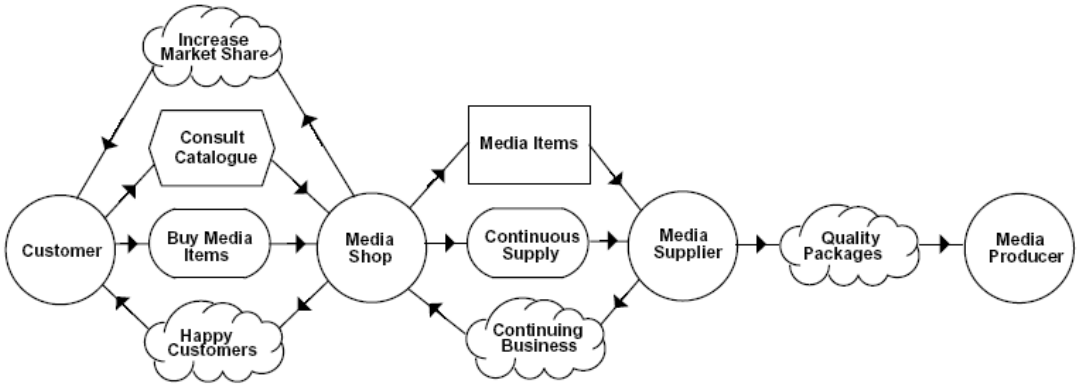

Figure 5.6: i*-Modell (aus [84])

Frühe Anforderungsbeschreibung. In dieser Aktivität werden die verschiedenen Interessenten (stakeholder) zusammen mit ihren Interessen identifiziert. Die Interessenten werden als Aktoren und ihre Interessen werden als Ziele repräsentiert. Die Aktoren und Ziele werden in einem Abhängigkeitsmodell dargestellt. Ein Ziel ist dabei ein Abhängigkeitsmerkmal eines das Ziel verfolgenden Aktors von einem anderen Aktor, falls der zweite Aktor zum Erreichen des Ziels beiträgt. Abbildung 5.6 zeigt ein Beispiel eines solchen Modells. Aktoren sind als Kreise, Ressourcen als Rechtecke, harte Ziele als Ovale, weiche Ziele als Wolken und Aufgaben (tasks) als Sechsecke dargestellt. Der Aktor Media Shop bildet das zu erstellende Softwaresystem.

Späte Anforderungsbeschreibung. In dieser Aktivität sollen die Abhängigkeiten zwischen Aktoren und zwischen ihren Zielen beschrieben werden. Das *strategische Abhängigkeitsmodell* (strategic dependency model) beschreibt die Beziehungen zwischen Aktoren. Das *strategische Erklärungsmodell* (stategic rationale model) dient dazu, Ziele für die jeweiligen Aktoren zu analysieren.

Das strategische Abhängigkeitsmodell verfeinert das Abhängigkeitsmodell aus der frühen Anforderungsbeschreibung. Dabei werden einerseits die Abhängigkeiten in einer textuellen, formalen Spezifikation dadurch verfeinert, dass Bedingungen für die Erfüllung von Abhängigkeiten formuliert werden können. Andererseits werden Ressourcen und Aufgaben als weitere Abhängigkeitsmerkmale zwischen Aktoren eingeführt.

Das zu erstellende Software-System ist als ein eigener Aktor, im Beispiel der Aktor Media Shop, eingeführt und auf die Aktoren der Interessenten bezogen. Diese kennzeichnen die Verpflichtungen gegenüber der Umgebung, auch das, was das System von Aktoren in der Umgebung erwarten kann.

Im strategischen Erklärungsmodell werden Ziele der Aktoren verfeinert. Insbesondere werden die Ziele des System-Aktors analysiert und mit neuen, detaillierteren Zielen, Aufgaben und Ressourcen in Beziehung gesetzt. Neue Arten von Beziehungen beschreiben Zerlegungen oder Means-End-Beziehungen. Ziele oder Teilziele können an andere Aktoren

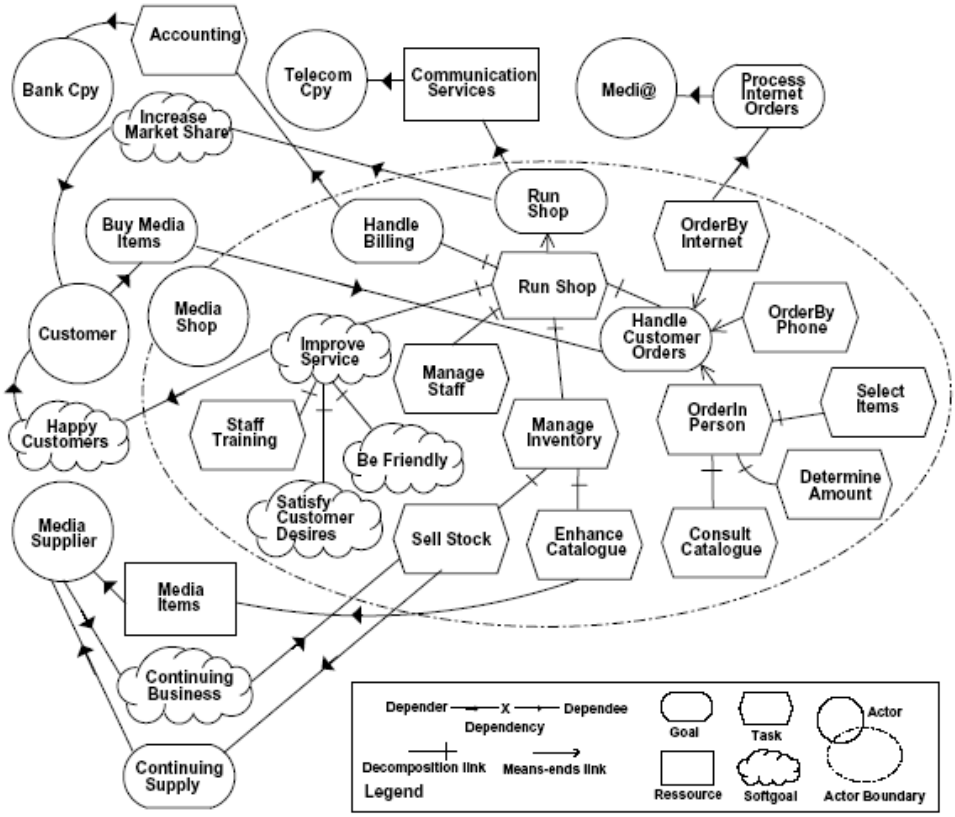

Figure 5.7: Integriertes Abhängigkeits- und Erklärungsmodell (aus [84])

delegiert werden, wofür ein weiterer Beziehungstyp eingeführt wird.

Das Abhängigkeits- und das Erklärungsmodell werden auch integriert in einem Diagramm dargestellt. Ein solches Diagramm ist in Abbildung 5.7 dargestellt.

Um die Modelle notieren zu können, wird für Tropos eine eigene Modellierungssprache als UML-Profil definiert. In dieser Sprache werden Aktoren, harte und weiche Ziele, Ressourcen und Aufgaben als Stereotype von UML-Modellelementen dargestellt (vgl. die frühe Anforderungsbeschreibung). Stereotype werden auch für die verschiedenen Arten von Abhängigkeiten eingeführt. Verschiedene Elemente sind in Abbildung 5.7 enthalten.

Architekturentwicklung und detaillierter Entwurf. Es werden weitere Systemaktoren eingeführt, die für Teilziele und Teilaufgaben verantwortlich sind, die dem System zugeordnet sind. Im detaillierten Entwurf werden die Systemaktoren weiter verfeinert. Die Kommunikation und Koordination der Aktoren wird spezifiziert. Zu diesem Zweck werden Klassen- und Sequenzdiagramme verwendet.

Eignung für die Agenten-Konstituenten-Modellierung Der Schwerpunkt des Vorschlages liegt auf der Anforderungsbeschreibung. Ziele und damit zusammenhängende Strategien in Form von Tasks werden vergleichsweise detailliert visuell beschrieben. Die eingesetzen visuellen Modelle ähneln syntaktisch Use-Case-Diagrammen. Autonomie von Agenten und Interaktion werden nicht explizit modelliert.

Bewertung des Prozessmodells Die Beziehungen zwischen den einzelnen Modellen sind im wesentlichen nur für die Erklärungsmodelle der Anforderungsbeschreibung deutlich.

5.7 Agentenbasierter Modellierungsansatz für Realzeitsysteme

Im Schwerpunktprogramm "Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen" sowie insbesondere dem Sonderforschungsbereich "Selbstoptimierende Systeme des Maschinenbaus" der Deutschen Forschungsgemeinschaft entsteht ein ingenieurwissenschaftlicher Vorschlag zur Anwendung des Agentenparadigmas im Bereich mechatronischer Realzeitsysteme. Funktionsgruppen mechatronischer Systeme mit inhärenter Teilintelligenz werden aus Sicht der Informationsverarbeitung als Softwareagenten aufgefasst. Ein komplexes mechatronisches System, das auf dem Zusammenwirken mehrerer Funktionsgruppen beruht, wird als ein Multiagentensystem aufgefasst.

In [57, 77] wird ein agentenorientierter Ansatz zur Kontrolle und Optimierung mechatronischer Systeme vorgeschlagen, bei denen isoliert arbeitende mechatronische Komponenten verknüpft werden. Da mechatronische Systeme oft sicherheitskritisch sind, scheint die flexible Verknüpfung von Komponenten ungeeignet. Durch einen Satz von Regeln

in Form von Pattern und Rollen wird für geregelte Interaktion gesorgt. Eine kompositionale, auf Model Checking basierende Verifikationstechnik ermöglicht, die Zuverlässigkeit des Gesamtsystems zu überprüfen. Problematisch ist dabei, dass auf Model Checking basierende Techniken für die Überprüfung der Korrektheit von Software für komplexe, verteilte und eingebettete Systeme nur begrenzt einsetzbar sind. In der Arbeit von Giese et. al. wird das dabei auftretende Problem der Zustandsraumexplosion adressiert [56]. Für eine Teilmenge des UML 2.0 Komponentenmodells wird ein für die Domäne der Steuerungssoftware mechatronischer Systeme spezifisches formales semantisches Modell zusammen mit einem Entwurfsverfahren eingeführt. Die Schritte des Entwurfsverfahrens beschreiben, wie aus domänenspezifischen Teilmodellen des Systemverhaltens (Pattern, Statecharts) das komplexe Softwaresystem zusammengesetzt werden kann. Die Korrektheit der Pattern kann einzeln überprüft werden, da sie lediglich einfaches Kommunikationsverhalten und eine feste Anzahl von Kommunikationsrollen besitzen. Durch eine strenge syntaktische Komposition wird garantiert, dass für das entstehende System die Verifikationsresultate der Pattern erhalten bleiben.

Bewertung Der Schwerpunkt des Vorschlages liegt auf der semantisch fundierten, UML-basierten Verhaltensbeschreibung autonomer Teilsysteme und der Beschreibung von deren Kopplung. Durch Pattern und Rollen wird strukturierte Interaktion zwischen Agenten beschrieben. Die Begriffe Ziel und Strategie werden nicht explizit verwendet. Statt Zielen sind zeitbehaftete Bedingungen angegeben, denen Verhaltensbeschreibungen für Komponenten (Statecharts) genügen müssen. Die Überprüfung der Verhaltensbeschreibungen gegebenüber den Bedingungen erfolgt durch Model Checking.

Der Ansatz besitzt Gemeinsamkeiten mit dem in dieser Arbeit vorgestellten Modellierungsverfahren. Es existieren Konstituenten für Autonomie und strukturierte Interaktion. Konstituenten für Proaktivität fehlen. Parallelen sind darin zu sehen, dass Model Checking zur Überprüfung von Verhaltensbeschreibungen verwendet wird. Ein wesentlicher Unterschied besteht darin, dass keine agentenspezifische Modellierungssprache formal definiert wird. Ein Entwurfsverfahren wird skizziert.

5.8 Objektorientierte Modellierungsprozesse

Es soll untersucht werden, inwieweit vorhandene Prozessmodelle, und dabei insbesondere aktuelle objektorientierte Prozesse wie RUP und Catalysis für die agentenorientierte Softwareentwicklung genutzt werden können.

Ausgehend von frühen Vorschlägen wie dem Wasserfallmodell bis hin zu objektorientierten Softwareentwicklungsprozessen finden sich die fundamentalen Aktivitäten Anforderungsspezifikation, Analyse, Entwurf, Implementierung, Wartung und Testen immer wieder (vgl. Jacobson [70]). Im klassischen Wasserfall-Modell werden diese Aktivitäten nacheinander und u.U. iteriert durchgeführt (vgl. Sommerville [96]). Problematisch ist an diesem Modell, dass veränderte Anforderungen für nachfolgende Aktivitäten ein hohes Risiko darstellen, da viele Arbeitsergebnisse u.U. verworfen werden müssen. Die

wesentliche Erkenntnis von Boehm [10] bestand darin, eine iterative und *inkrementelle* Vorgehensweise zu fordern. In dem von Boehm vorgeschlagenen Spiralmodell werden die grundlegenden Aktivitäten sequentiell mehrfach durchlaufen. Zwischen zwei Durchläufen werden die aktuellen Randbedingungen des Projektes festgestellt, Risiken neu bewertet und die folgende Iteration entsprechend modifiziert. Ein weiteres wichtiges Prozessmodell ist das V-Modell, das ebenfalls auf den beschriebenen Fundamentalaktivitäten basiert (vgl. [40]).

Die bisher genannten Prozessmodelle strukturieren den Softwareentwicklungsprozess lediglich in allgemeiner Weise. Es gibt kein grundlegendes Konzept zur Strukturierung von Modellen oder Implementierungsdokumenten. Folgerichtig werden auch weder eine Modellierungs- noch eine Programmiersprache festgelegt. Dieses Fehlen führt zu dem Nachteil, dass die grundlegenden Aktivitäten nur sehr allgemein beschrieben sind. Wie die Aufgaben der grundlegenden Aktivitäten konkret bewältigt werden können, ist unklar.

In diesem Zuge sind objektorientierte Vorgehensmodelle, wie OOSE (vgl. Jacobsen [70]), FUSION [5], Catalysis [41], Rational Unified Process (RUP) [69], u.a. vorgeschlagen worden. In den objektorientierten Prozessmodellen Catalysis und RUP wird UML zur Notation von Modellen verwendet. Durch Verwendung objektorientierter Konzepte in Softwareprozessmodellen kommen die Vorteile der Objektorientierung zum Tragen (vgl. Kapitel 3). Die durchgängige Verwendung der objektorientierten Konzepte in den grundlegenden Aktivitäten Anforderungsspezifikation, Analyse, Entwurf, Implementierung, Testen und Wartung führt dazu, dass Abhängigkeiten zwischen Modellen und Modellteilen in unterschiedlichen Aktivitäten besser verfolgbar sind (traceability). Ein Beispiel sind Klassen des Analyse- und Entwurfsmodells, bei denen die Verfeinerung des Analysemodells durch die Entwurfsaktivität u. a. an der zunehmenden Zahl von Attributen und Operationen in Entwurfsklassen erkennbar ist.

Um objektorientierte Konzepte konkret verwenden zu können, wurden objektorientierte Modellierungssprachen wie UML entwickelt, mit denen Modelle formuliert werden. Die Modelle, d.h. die Artefakte verschiedener Aktivitäten werden in der gleichen Sprache formuliert. Daraus resultiert der Vorteil, dass die Beziehungen zwischen unterschiedlichen Modellen leichter erkennbar sind. Ein Bruch dieser Homogenitätseigenschaft erfolgt heute noch während der Implementierung, da ausführbare Modelle i.d.R. in objektorientierten Programmiersprachen formuliert werden. Eine einheitliche Sprache zur Beschreibung von Modellen hat den Vorteil, dass die Werkzeugunterstützung einfacher ist, da auf die Transformation zwischen unterschiedlichen Repräsentationen verzichtet werden kann.

Wesentliche Merkmale der objektorientierten Prozessmodelle Rational Unified Process und Catalysis werden im folgenden herausgestellt. Dabei wird insbesondere gezeigt, welche Konzepte und Sprachmittel angeboten werden, um mit UML objektorientierte Teilmodelle in den grundlegenden Aktivitäten von Softwareentwicklungsprozessen zu bilden. Bezogen auf die Modellierung agentenbasierter Systeme mit UML werden RUP und Catalysis hinsichtlich derjenigen Kriterien für Softwareprozessmodelle aus Kapitel 4 untersucht, die von Agentenkonzepten unabhängig sind.

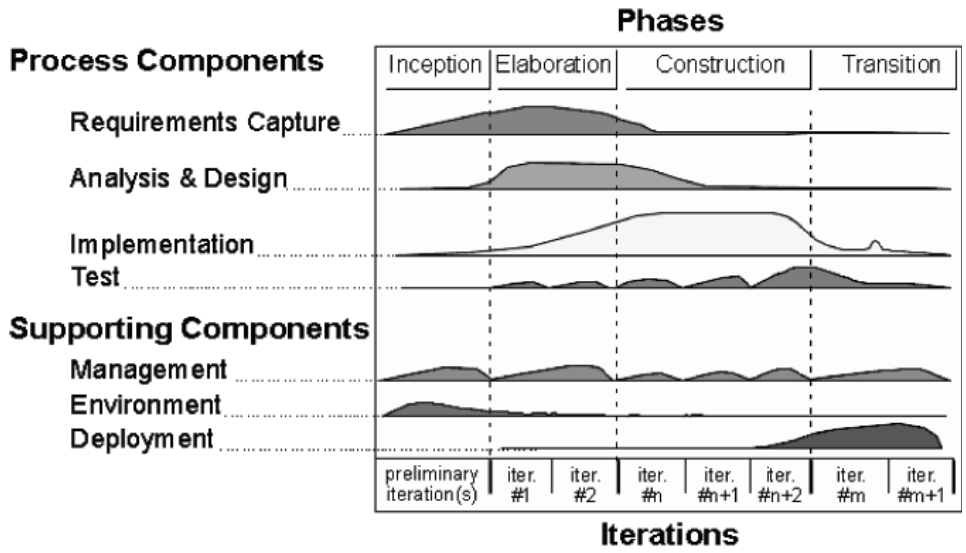

Figure 5.8: Disziplinen (Aktivitäten) und Phasen des Unified Process

Rational Unified Process

Der *Rational Unified Process (RUP)* (kurz: Unified Process oder UP) ist ein Vorgehensmodell der Firma IBM (vormals von Rational Inc., vgl. [69]). Der Unified Process ist nach Angabe seiner Autoren "iterativ, inkrementell, phasenorientiert, architekturzentriert und Use-Case-gesteuert". Im folgenden sollen die wichtigsten Konzepte des Prozesses kurz vorgestellt werden.

Der zeitliche Ablauf des Unified Process ist durch die vier aufeinander folgenden *Phasen* Anfangsphase (Inception), Ausarbeitungsphase (Elaboration), Konstruktionsphase (Construction) und Übergang in die Nutzung (Transition) geprägt. Jede Phase endet mit einem Meilenstein, zu dem bestimmte Artefakte erstellt sein müssen. Die vier Phasen bilden zusammen den gesamten Lebenszyklus eines Produktes.

Mit *Disziplinen* sind in RUP die wesentlichen Aufgaben eines Softwareentwicklungsprozesses bezeichnet. Während der vier Phasen sind die Projektteilnehmer in den Disziplinen Geschäftsprozessmodellierung, Anforderungsspezifikation, Analyse, Entwurf, Implementierung und Test (Business modeling, requirements specification, analysis, design, implementation, test, deployment, configuration and change management, project management) tätig.

In Abbildung 5.8 sind die Disziplinen des Unified Process gegenüber den Phasen dargestellt. Die Höhen der Graphen geben Auskunft über die typischerweise zu erbringende Leistung zu Zeitpunkten der einzelnen Disziplinen.

Die Disziplinen haben in den Phasen unterschiedliches Gewicht. In der Anfangsphase nehmen die Aktivitäten Anforderungsanalyse und das Projektmanagement einen großen Raum ein. In der Ausarbeitungsphase wird die Anforderungsdefinition vollendet. Für die Entwicklung eines konkreten Systems wird in dieser Phase der Entwurf ausgearbeitet. Zusätzlich wird mit der Entwicklung von Prototypen begonnen. Während der Entwurf in

der Konstruktionsphase abgeschlossen wird, nimmt die Implementierung die meiste Zeit in Anspruch. Parallel zur Implementierung werden laufend Tests durchgeführt. Hauptaufgabe der letzten Phase, des Übergangs in die Nutzung, ist die Verteilung des Produktes an die Kunden.

Um von konkreten Personen oder Teams von Personen zu abstrahieren, sind die einzelnen Tätigkeiten innerhalb der Disziplin *Rollen* zugeordnet. Ein Projektteilnehmer oder ein Team kann verschiedene Rollen einnehmen. Jeder Rolle einer Disziplin ist eine Menge von *Aktivitäten* zugeordnet. Die Arbeitsergebnisse von Aktivitäten werden als *Artefakt* bezeichnet. Artefakte sind Dokumente, d.h. natürlich- und programmiersprachliche Texte, Diagramme und ausführbare Programmfragmente. Jede Aktivität erzeugt aus vorgegebenen Artefakten, die Ergebnisse vorangegangener Aktivitäten sind, neue Artefakte als Arbeitsergebnisse.

Für jede Disziplin werden die Aktivitäten zu einem *workflow* verknüpft. Jeder workflow wird durch ein UML-Aktivitätendiagramm beschrieben. Ein workflow enthält Schleifen, die verlassen werden können, falls Austrittsbedingungen, die sich auf auf Eigenschaften von in der Schleife veränderten Artefakten beziehen, erfüllt sind. Indem solche Schleifen mehrfach durchlaufen werden, wachsen Artefakte des Prozesses *inkrementell* und *iterativ*.

Iteratives und inkrementelles Vorgehen entspringt der Idee, ein Bündel von abhängigen Aktivitäten in eine Folge von Bündeln gleicher Aktivitäten zu unterteilen, wobei jede komplexe Aktivität in eine Sequenz von gleichartigen Teilaktivitäten mit kleineren Änderungen zerlegt wird. Mit dieser Aufteilung soll dem Risiko entgegengewirkt werden, dass in einer Folge von komplexen Aktivitäten die Abhängigkeiten zwischen den Aktivitäten zu Problemen führen.

Use Cases bilden das wesentliche Konzept, um funktionale Anforderungen an das zu konstruierende System zu erfassen. Jeder Use Case besteht aus einer Beschreibung der Funktion, die das zu entwickelnde System liefern soll. Die Anforderungen an die Funktion wird festgelegt, indem Eingabe- und zugehörige Ausgabewerte vorgegeben werden. Für jede Kombination von Ein- und Ausgaben können Nebenbedingungen angegeben werden. Durch eine solche Spezifikation sind nicht nur Anforderungen festgelegt, sondern auch gleichzeitig Testfälle für die verschiedenen zu entwickelnden Modelle fixiert. Daneben strukturieren Use Cases den Unified Process, indem sie sukzessive im Analyse-, Entwurfs- und Implementierungsmodell verfeinert und realisiert werden. Durch diese Verfeinerungsbeziehungen (trace) sind die Modelle explizit voneinander abhängig. Änderungen an Modellen ziehen notwenigerweise Änderungen an abhängigen Modellen nach sich.

Für das wichtige Problem, die Interaktion mit dem Benutzer zu modellieren, werden die genannten Verfeinerungsschritte ausgehend von Use Cases konkretisiert. Dazu werden im Klassendiagramm des Analysemodells die stereotypisierten Klassen boundary, control und entity class eingeführt, um die Benutzerinteraktion zu strukturieren. Das Use Case Verhalten wird verfeinert durch Sequenz- und Aktivitätendiagramme beschrieben. Für das Problem der Benutzerinteraktion wird also gezeigt, wie UML-Diagrammen zu benutzen sind, um verfeinerte Modelle zu produzieren.

Eine *Softwarearchitektur* beschreibt eine Menge von Softwarekomponenten und die Verknüpfung der Komponenten durch Konnektoren. Softwarearchitekturen entstehen durch die Verwendung von Architekturstilen und Architekturmustern bei der Entwicklung von Modellen. Aus Stilen, die generische Vorlagen für die Struktur von Gesamtmodellen wie beispielsweise Systemmodellen bilden, und Mustern, die generische Vorlagen für partielle Modellstrukturen bilden, entstehen durch Fixierung von Parametern konkrete Architekturen und Modellteile.

Bewertung. Der Unified Process wird von seinen Autoren als architekturzentriert charakterisiert. Die Architektur des Softwaresystems soll im Rahmen der Disziplinen Analyse und Entwurf bis zum Ende der Elaborationsphase erstellt werden. Die Architektur soll dabei aus den wichtigsten Use Cases abgeleitet werden. Ein Verfahren oder Hinweise dazu, wie aus Use Cases die Softwarearchitektur abgeleitet werden kann, werden nicht angegeben.

Im Prozessmodell Unified Process sind die Begriffe und Aktivitäten verbal und strukturiert beschrieben. Die Zuordnung von Artefakten und Aktivitäten zu Rollen und Ablaufbeschreibungen (Workflows) wird auch visuell durch UML-Klassen- und UML-Aktivitätendiagramme beschrieben. Eine präzise und formale Beschreibung der Begriffe und Aktivitäten existiert im Unified Process allerdings nicht.

Im Unified Process kommen die Entwurfsprinzipien Separation of Concern, Abstraktion und Modularisierung sehr wenig zum Tragen, da kaum auf die Semantik und Pragmatik von Modellen eingegangen wird. Allein die Verwendung von UML-Klassen- und Sequenzdiagrammen für die Strukturierung der Benutzerinteraktion bildet ein Beispiel für die pragmatische Anwendung der Entwurfsprinzipien.

In Bezug auf die agentenorientierte Softwareentwicklung kann der Unified Process einen Rahmen für die Projektorganisation bilden. Eine Vorlage für eine detaillierte und formale Beschreibung von Aktivitäten stellt der Unified Process nicht dar und kann damit in dieser Hinsicht kein Ausgangspunkt für die Formulierung eines agentenorientierten Prozessmodells sein.

Damit ist der Überblick über die Struktur und wichtige Konzepte des Unified Process abgeschlossen. Mit dem Prozessmodell Catalysis wird im folgenden ein zweites aktuelles Beispiel beschrieben, in dem ebenfalls UML-Diagramme für die Beschreibung von Teilmodellen verwendet werden.

Catalysis

In Catalysis gibt es drei wesentliche Konzepte ("Prinzipien") zur Modellkonstruktion: Abstraktion, Präzision und Komposition [41]. *Abstraktion* soll den Modellierungsprozess steuern, indem fortlaufend mehr Details zu Modellen hinzugefügt werden. *Präzision* steht für die Forderung der Autoren, neben verbalen Verhaltensbeschreibungen auch zugehörige formale Spezifikationen zu erstellen. *Komposition* beinhaltet die Forderung, dass möglichst solche Modellelemente eingesetzt werden, die sich untereinander leicht verknüpfen lassen.

Catalysis basiert auf drei Arten von Modellen, die wesentliche Artefakte der Entwicklung darstellen: Typmodelle, Kollaborationen und Frameworks. Alle drei Arten von Modellen dienen dazu, Struktur- und Verhaltensbeschreibungen zu formalisieren und damit zu präzisieren. Die Modelle werden auf verschiedenen Aktivitätsebenen verfeinert, so dass abstrakte Modelle dahingehend vervollständigt werden, dass sie ausführbar sind. Darauf wird unten noch genauer eingegangen.

Die im folgenden kurz beschriebenen Teilmodelle Typmodell, Kollaboration und Framework werden mit UML-Diagrammen notiert. Catalysis verwendet Klassendiagramme, Objektdiagramme, Use-Case-Diagramme, Statecharts und Sequenzdiagramme. In Catalysis werden diese Diagrammarten durchgängig auf allen Abstraktionsebenen eingesetzt.

Ein *Typmodell* beschreibt das nach außen sichtbare Verhalten einer Komponente. Die Notation als dreiteiliger Kasten folgt dem Klassendiagramm in UML-Notation. Der obere Teil bezeichnet den Typ, der mittlere Teil listet die Attribute auf und der untere Teil spezifiziert die Operationen des Typs. Die Attribute im Typmodell dienen ausschießlich dem besseren Verständnis der Operationen und müssen nicht implementiert werden (vollständige Datenkapselung). Die Operationen werden nicht nur durch Signaturen beschrieben, sondern auch mittels Vor- und Nachbedingungen auf den Typattributen sowie durch Invarianten. Typmodelle stellen damit ein wichtiges Mittel für die komponentenbasierte Softwareentwicklung dar: falls eine Komponente einem bestimmten Typ entspricht, kann diese Komponente an allen Stellen eingesetzt werden, an denen dieser Typ benötigt wird. Auch können zwei Komponenten gleichen Typs gegeneinander ausgetauscht werden. Der Test von Komponenten gegen ein Typmodell ist umso präziser, je detaillierter die Semantik der Operationen des Typmodells spezifiziert wurde.

Kollaborationen beschreiben, wie einzelne Komponenten miteinander kommunizieren. Eine Komponente wird durch ein Klassendiagramm beschrieben. In einer Kollaboration nehmen Objekte Rollen ein und stehen abhängig von ihren Rollen über Aktionen mit anderen Objekten in Verbindung. Das Verhalten einer Kollaboration aufgrund von Interaktion zwischen den beteiligten Objekten wird durch ein Kollaborationsdiagramm beschrieben. Das Resultat einer Interaktion wird durch einen *snapshot* beschrieben, indem in einer Kollaboration Objekte und Verbindungen ausgezeichnet werden, die vor der Interaktion bereits existierten.

Um wiederkehrende ähnliche Lösungen für gleichartige Probleme in Modellen einfach wiederverwenden zu können, werden in Catalysis *Frameworks* eingeführt. Das sind UML-Packages aus Typmodellen oder Kollaborationen mit Platzhaltern für generische Modellelemente.

Catalysis sieht drei verschiedene Aktivitätsebenen vor: solche zum Erstellen eines Geschäftsmodells (Anwendungsebene), zum Komponentenentwurf (Komponentenebene) und solche zu Objektentwurf und Komponentenkonfiguration (Implementierungsebene). Aktivitäten in diesen Ebenen sind von Aktivitäten in vorausgehenden Ebenen abhängig. Durch aufeinanderfolgende Aktivitäten in verschiedenen Ebenen werden Geschäftsmodelle sukzessive verfeinert. Damit bilden die drei Aktivitätsebenen Stufen unterschiedlicher

Abstraktion.

Auf der *Anwendungsebene* liegt der Schwerpunkt in der Modellierung des Anwendungsbereiches in der Sprache des Kunden (Geschäftsmodell). Die einzelnen Objekte auf dieser Ebene sind Personen, Systeme und Aufgaben mit denen der Kunde bei seiner Arbeit in direkten Kontakt gelangt. Diese Abstraktionsebene soll den Geschäftsprozess des Unternehmens modellieren und als Grundlage der Anforderungsspezifikation dienen.

Auf der *Komponentenebene* wird das Geschäftsmodell soweit spezifiziert, dass die Funktionen auf einzelne Komponenten verteilt werden können. Hier wird spezifiziert, wer bestimmte Anforderungen des Systems erfüllen muss. Die Spezifikation geschieht durch Typmodelle, Operationsspezifikationen und Zustandsdiagramme, die das nach außen sichtbare Verhalten der Komponenten beschreiben. Komposition wird durch das Zusammensetzen von Komponenten (pluggable parts) an dafür vorgesehenen Schnittstellen erreicht. Vorteile sind Adaptierbarkeit und Wiederverwendung.

Die *Implementierungsebene* beschreibt, wie die Komponente aus kleineren, interagierenden Einheiten zusammengesetzt wird, so dass die gewünschte Funktionalität erreicht wird. Die entstehenden Einheiten werden wieder mittels Methoden der Komponentenebene spezifiziert und solange weiter zerteilt bis man sie direkt auf Objekte der Programmiersprache oder schon vorhandene Komponenten abbilden kann.

Bewertung. Das Prozessmodell Catalysis zeichnet sich dadurch aus, dass die Entwurfsprinzipien Abstraktion und Modularisierung (in Form der Komposition) propagiert und unterstützt werden. Typmodelle und Kollaborationen dienen der Abstraktion. So dienen Kollaborationen der objektunabhängigen Beschreibung von Interaktionen und können im Agentenkontext zur Modellierung von Protokollen eingesetzt werden. Komponenten und Frameworks dienen der Modularisierung von Systemen.

Modelle werden mit Hilfe von UML formuliert und auf drei Aktivitätsebenen erstellt, die die Fundamentalaktivitäten Anforderungsspezifikation, Analyse und Entwurf beinhalten. Die Anwendungsebene dient der Anforderungsbeschreibung und auf der Komponentenebene werden diese Anforderungen analysiert und bestimmten Komponenten zugeordnet. Die Implementierungsebene umfasst Entwurfsaktivitäten, wie die Verfeinerung von Komponenten. Die Entwicklungsschritte von einer Ebene zur nächsten Ebene werden durch die Angabe von Beziehungen zwischen verschiedenen UML-Diagrammen präzise und vergleichsweise formal beschrieben.

5.9 Fazit

In AgentUML, Mase, TROPOS und MAS-KommonKads wird UML im Ansatz dazu verwendet, agentenspezifische Konstituenten zu modellieren. In keinem Fall wird UML, z.B. in Form eines Profils, umfassend an die Erfordernisse agentenorientierter Modellierung angepasst. Die Beschreibungen und Definitionen von Syntax und Semantik einer agentenorientierten Modellierungssprache sind insgesamt für alle Ansätze unzureichend.

Konzept	Konstituente	UML	AgentUML	Mase	GAIA	MAS-KommonKads	TROPOS
Visuelle Modellierung	UML	+	0	+	-	0	0
Autonomie	Ausführungs-faden	+	+	+	-	-	-
	Flexible Kontroll-elemente	0	0	0	-	-	-
Strukturierte Interaktion	Protokoll	0	+	+	+	0	-
Proaktivität	Ziel	-	-	0	+	0	+
	Strategie	-	-	0	0	-	+

Figure 5.9: Vergleich von (agentenorientierten) Modellierungssprachen

Prozessmodelle beschreiben die Pragmatik von Sprachen. Mase ist ein Prozessmodell, das alle fundamentalen Aktivitäten eines Prozesses enthält. Dabei werden die Konstituenten schrittweise entwickelt. TROPOS überzeugt mit einer detaillierten Anforderungsbeschreibung, in der Ziele wichtige Modellierungsinstrumente bilden. Unzureichend ausgearbeitet sind allerdings die Analyse- und Entwurfsaktivitäten. In MAS-KommonKads existieren viele Teilmodelle, deren Zusammenhang aber weitgehend unklar bleibt. Das Prozessmodell Unified Process unterstützt im wesentlichen lediglich das Projektmanagement auf der Basis von UML. Das Prozessmodell Catalysis zeigt demgegenüber, dass für ein a priori zugrunde gelegtes Paradigma wie das der Objektorientierung wesentlich konkretere Prozessschritte in Form von präzisen Entwicklungsschritten beschrieben werden können. Die verschiedenen Prozessmodelle erfüllen die Anforderungen insgesamt in unterschiedlicher Weise und jeweils unvollständig.

In den Tabellen 5.9 und 5.10 sind die Ergebnisse dieses Kapitels zusammengefasst.

Es wird deutlich, dass weder eine Modellierungssprache noch ein Prozessmodell existieren, die die in Kapitel 4 entwickelten Kriterien erfüllen.

Das negative Ergebnis der Untersuchung bildet die Motivation dafür, eine neue UML-basierte Modellierungssprache (vgl. Kapitel 6) und ein zugehöriges Prozessmodell (vgl. Kapitel 8) zu erarbeiten, die den geforderten Kriterien genügen.

Anforderungen		**Mase**	**GAIA**	**MAS-KommonKads**	**TROPOS**	**RUP**	**Catalysis**
Anforderungsspezifikation		+	+	+	+	+	+
Analyseaktivität		+	+	0	0	+	+
Entwurfsaktivität		+	-	0	0	+	+
Einsatz visueller Modellierungssprachen		+	-	+	+	+	+
Verwendung der Entwurfsprinzipien Separation of Concern, Abstraktion und Modularisierung		0	0	0	0	0	+
Formale Beschreibung der Entwicklungsschritte		0	0	-	0	0	0
Entwicklungsschritte für Konstituenten zu Konzepten von Agenten	Autonomie	0	-	-	-	-	-
	Strukturierte Interaktion	0	+	0	-	-	0
	Proaktivität	0	0	0	+	-	-

Figure 5.10: Vergleich von (agentenorientierten) Prozessmodellen

Chapter 6

Eine Modellierungssprache für Agentenorientierte Softwareentwicklung (AOSE)

Aus dem Zusammenhang zwischen Modellkonstituenten und Modellen in Dokumenten ergaben sich die Anforderungen an eine Modellierungssprache, in der sich die charakteristischen Eigenschaften von Agenten wie Autonomie, strukturierte Interaktion und Verhaltensflexibilität widerspiegeln (vgl. Kapitel 2). Im Kapitel 4 wurden die Anforderungen an eine agentenorientierte Modellierungssprache herausgearbeitet. In Kapitel 5 ist gezeigt worden, dass bis heute keine geeignete Modellierungssprache für agentenbasierte Systeme existiert, die diese genannten Anforderungen erfüllt. In diesem Kapitel wird die agentenorientierte Modellierungssprache *Agent Modeling Language* (AML) entwickelt, die den geforderten Kriterien genügt.

Als semantischer Bereich für die Sprache AML dienen Graphen und darauf operierende Graphtransformationen (siehe Abschnitt 6.2). In diesem formal definierten semantischen Bereich wird ein Systemmodell für agentenbasierte Systeme präzise beschrieben. Dieses Systemmodell ist mit dem im Kapitel 2 entwickelten abstrakten Systemmodell verträglich. Syntaktisch ist die Sprache AML mit der etablierten Sprache UML verträglich. Die Syntax der Sprache AML wird in Abschnitt 6.5.1 mit Hilfe der Erweiterungsmechanismen von UML als ein UML-Profil definiert. Die mit der Modellierungssprache AML erzeugbaren Modelle werden auf das Systemmodell im semantischen Bereich abgebildet.

Eigenschaften von AML-Modellen resultieren aus denen des formalen Systemmodells. Die Konstituenten der Konzepte Autonomie, strukturierte Interaktion und Proaktivität werden im Systemmodell repräsentiert. Die Möglichkeiten zur Überprüfung von Modellen auf die Erfüllung von Eigenschaften werden in Kapitel 7 diskutiert. Dieses Kapitel basiert auf vorausgehenden Arbeiten (vgl. [32, 29, 38, 36, 37, 35, 30, 33, 34]).

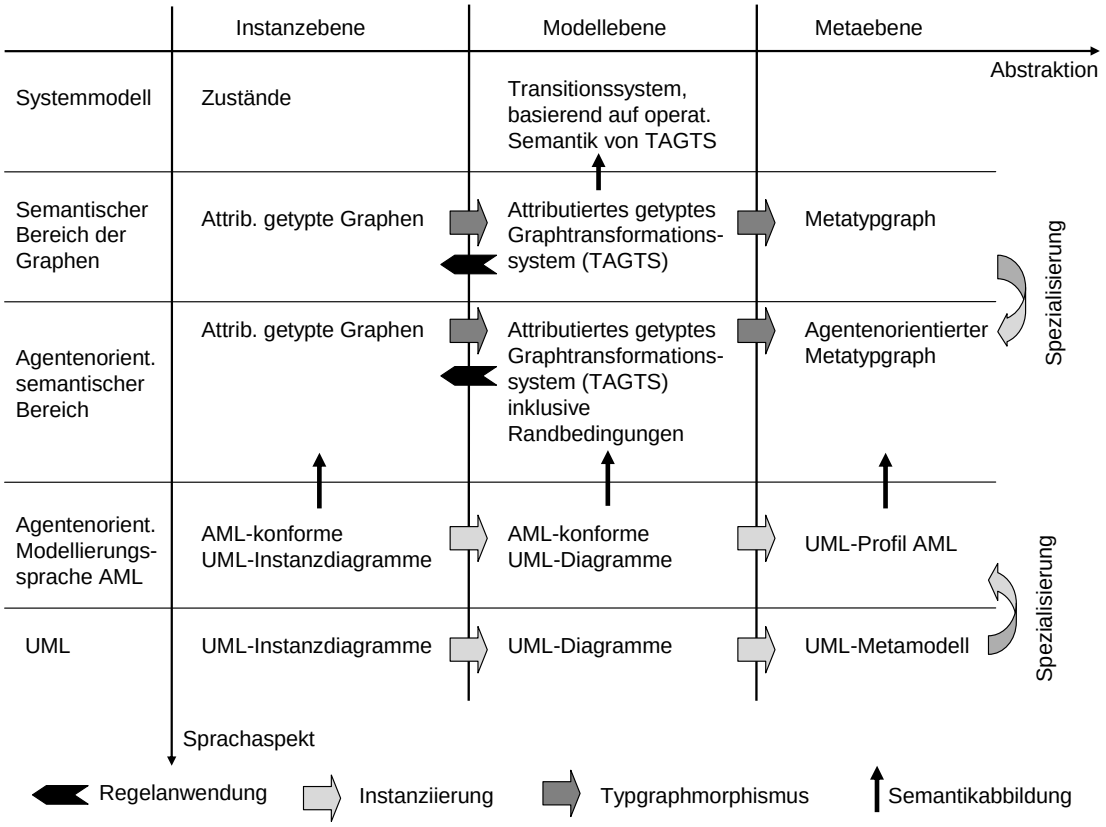

Figure 6.1: Übersicht über die Spracharchitektur

6.1 Sprachstruktur

Die agentenorientierte Modellierungssprache AML soll auf der etablierten Sprache UML aufbauen und durch eine formale Semantik präzise beschrieben werden. Der Sprache AML liegen als semantischer Bereich Graphen und darauf operierende Graphtransformationen zugrunde.

Die Struktur der Sprache ist durch drei Abstraktionsebenen gegeben (siehe Abb. 6.1). Auf der Instanzebene werden die Ausprägungen des Systemzustands beschrieben. Auf der Modellebene werden erlaubte Instanzen und deren Veränderungen durch strukturelle und dynamische Modelle spezifiziert. Die Metaebene dient der Sprachbeschreibung und schränkt die Modelle der Modellebene auf die gewünschten Elemente der Sprache ein.

Orthogonal zu den Abstraktionsebenen existiert die Unterscheidung zwischen der Syntax einer Sprache und dem semantischen Bereich, in dem die Konzepte der Sprache mathematisch präzise beschrieben werden. Die Semantik der Sprache ist eine Abbildung der Syntax in einen semantischen Bereich. Die Abbildung erstreckt sich über alle drei Abstraktionsebenen (siehe Abb. 6.1).

Die Syntax einer Sprache setzt sich in der Regel aus der konkreten und der abstrakten Syntax zusammen. In der konkreten Syntax wird die Notation der Sprache festgelegt.

Demgegenüber spiegelt die abstrakte Syntax die Struktur der Sprache wider und ist oft durch eine homogene, vergleichsweise einfache Repräsentation gegeben. Für textuelle Sprachen ist das oft die Backus-Naur-Form. Für visuelle Sprachen wie UML dient ein Metamodell der Sprachbeschreibung. Worte der Sprache sind durch Instanzen des Metamodells gegeben. Sätze bestehen aus Instanzen, die gemäß den strukturellen Zusammenhängen im Metamodell zusammengesetzt sind. Die Aufteilung in die konkrete und die abstrakte Syntax hat den Vorteil, dass die Semantik nur für die abstrakte Syntax definiert wird. Die Beziehung zwischen abstrakter und konkreter Syntax ist unabhängig davon festgelegt.

Die abstrakte Syntax der visuellen Modellierungssprache UML wird auf der Metaebene durch Klassendiagramme im UML-Metamodell notiert (vgl. UML-Spezifikation [86]). Die Diagramme der konkreten Syntax werden auf der Instanz- und Modellebene verwendet. Sie werden zur abstrakten Repräsentation auf Instanzen der Klassendiagramme des UML-Metamodells abgebildet. Diagramme der Instanzebene sind Instanzen von Diagrammen der Modellebene. Das gleiche gilt zwischen Diagrammen der Modell- und der Metaebene. Die Struktur von UML ist also verträglich mit der Unterscheidung in drei Abstraktionsebenen (siehe Abb. 6.1).

Um Erweiterungen von UML zu definieren, können Modellelemente als Stereotype ausgezeichnet werden. Inhaltlich müssen die Modellelemente vorgegebenen Bedingungen genügen oder sind durch Parameter (tags) erweitert (vgl. [86]). Eine Menge von in dieser Weise ausgezeichneten Modellelementen bildet ein UML-Profil. Für die nachfolgend definierte UML-basierte Modellierungssprache AML ist die Syntax der Diagramme durch ein UML-Profil definiert.

Im semantischen Bereich der Graphen und Graphtransformation sind Modelle durch getypte attributierte Graphtransformationssysteme (TAGTS) gegeben, die aus einem Typgraphen, Graphtransformationsregeln, einer Menge von Startgraphen und aus Randbedingungen bestehen. Typgraphen der Modellebene dienen der Typisierung von Graphen der Instanzebene. Diese Typgraphen sind selbst wiederum durch einen Typgraphen auf der Metaebene, den Metatypgraphen, typisiert. Dieser Metatypgraph schränkt die Typgraphen auf die gewünschten Elemente der Sprache ein.

Durch die Anwendung einer Graphtransformationsregel auf einen Graphen entsteht ein neuer Graph, d.h. der Graph wird transformiert. Durch die Graphtransformationsregel wird eine Graphtransformation induziert. Die Semantik eines Graphtransformationssystems ist durch die von Graphtransformationsregeln induzierten Graphtransformationen auf Graphen, ausgehend von den Startgraphen, gegeben.

Im Systemmodell werden Graphen als Zustände eines Transitionssystems aufgefasst. Übergänge zwischen Zuständen korrespondieren mit Graphtransformationen zwischen den Zustandsgraphen. Transitionssysteme bilden Eingaben für einen (abstrakten) Interpreter, der als Ausgaben Abläufe zum Systemmodell erzeugt.

Der agentenorientierte semantische Bereich leitet sich vom semantischen Bereich der Graphen ab. Im Unterschied zum allgemeinen semantischen Bereich sollen Modelle in Form von TAGTS die Konstituenten agentenorientierter Systeme verwenden. Deshalb

wird der agentenorientierte Metatypgraph auf Sprachelemente für die Konstituenten eingeschränkt. Weiterhin müssen Typgraphen der Modellebene Randbedingungen wie beispielsweise Kardinalitätseinschränkungen erfüllen, die sich aus den Eigenschaften der Konstituenten ergeben.

Die Semantik von AML ist durch Abbildungen von AML-Modellelementen auf geeignete Elemente des agentenorientierten semantischen Bereichs auf der Meta- und der Modellebene festgelegt. Die Elemente des UML-Profils AML werden in Beziehung zu Elementen des agentenorientierten Metatypgraphen gesetzt. Randbedingungen an Graphtransformationssysteme im agentenorientierten semantischen Bereich reflektieren sich in Bedingungen, die in AML an AML-Modelle gestellt sind.

Die Spracharchitektur wird in den folgenden Abschnitten für die verschiedenen Sprachaspekte entsprechend Abbildung 6.1 ausgearbeitet. Der fundamentale semantische Bereich der Graphtransformation wird im folgenden Abschnitt beschrieben. In Abschnitt 6.3 wird die Semantik von Graphtransformationssystemen und darauf aufbauend das Systemmodell diskutiert. Darauf aufbauend werden in Abschnitt 6.4 die Konstituenten von Agenten in einem eingeschränkten agentenorientierten semantischen Bereich, aufbauend auf Graphtransformation repräsentiert. In Abschnitt 6.5 wird die Sprache AML definiert und die Abbildung in den agentenorientierten semantischen Bereich festgelegt.

6.2 Graphen und Graphtransformation als semantischer Bereich

Im semantischen Bereich der Graphen werden die für die agentenorientierte Modellierung notwendigen Konzepte präzise erfasst. Das Ziel besteht darin, einen sogenannten *Typgraphen* auf der Metaebene, im folgenden *Metatypgraph* genannt, für die Sprache AML zu formulieren. Das wird ab Abschnitt 6.4.2 schrittweise erfolgen. Zur Vorbereitung dient die folgende Einführung in den semantischen Bereich der Graphen.

Knoten von Instanzgraphen werden beispielsweise für Agenten stehen, die über Attribute verfügen. Weitere grundlegende Konstituenten agentenbasierter Systeme wie Rolle und Datenobjekt werden als Knoten von Graphen aufgefasst werden.

Ein gerichteter und unmarkierter *Graph* ist ein Tupel $G = \langle G_V, G_E, src^G, tar^G \rangle$ mit einer Knotenmenge G_V, einer Kantenmenge G_E und Abbildungen $src^G : G_E \to G_V$ und $tar^G : G_E \to G_V$, die jeder Kante einen Ausgangs- und Zielknoten zuordnen.

Die Typisierung eines Graphen wird durch eine strukturverträgliche Abbildung in einen *Typgraphen* beschrieben, dessen Knoten und Kanten Typen für den Ausgangsgraphen bilden [23]. Um den Begriff des Typgraphen präzise einführen zu können, wird zuerst die Typisierungsbeziehung in Form eines *Graphmorphismus* formalisiert.

Ein *Graphmorphismus* $f : G \to H$ ist ein Paar von Abbildungen $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$, die die Ausgangs- und Zielknoten von Kanten bewahren, d.h. $src^H \circ f_E = f_V \circ src^G$ und $tar^H \circ f_E = f_V \circ tar^G$.

Knoten und Kanten eines Graphen G werden durch einen Graphmorphismus $m : G \to TG$ Typen aus einem *Typgraphen* TG zugeordnet [23]. Für ein Element x aus G heißt $m(x) = t$ sein Typ, und x wird oft in der Form $x : t \in G$ notiert. Das Paar $\langle G, m\rangle$ heißt *TG-getypter Graph*. Ein Morphismus $g : G \to G'$ heißt *TG-getypter Graphmorphismus* zwischen den TG-getypten Graphen $\langle G, m\rangle$ und $\langle G', m'\rangle$, falls $m' \circ g = m$ gilt.

Damit wird der Typ von Knoten und Kanten unter getypten Graphmorphismen erhalten. Typgraphen existieren sowohl auf der Modell- als auch auf der Metaebene. Ein Typgraph auf Modellebene dient der Typisierung von Instanzgraphen auf der Instanzebene. Ein solcher Typgraph ist selbst wiederum durch den Metatypgraphen typisiert. Diese Typgraphbeziehungen vermitteln zwischen den drei Abstraktionsebenen des semantischen Bereichs der Graphen in der vorgestellten Spracharchitektur (vgl. Abb. 6.1). Der nächste Schritt besteht darin, Attribute von Graphelementen zu definieren.

Attributierte Graphen. Graphen sind attributiert, falls deren Knoten oder Kanten mit Elementen eines abstrakten Datentyps (wie Strings oder natürliche Zahlen) verknüpft (koloriert) sind [81, 21]. Für die weitere Betrachtung genügen Graphen mit kolorierten Knoten. Mathematisch werden abstrakte Datentypen als Algebren über geeigneten Signaturen repräsentiert.

Abstrakte Datentypen dienen der Integration von Daten und Operationen. Mathematisch wird dazu eine Algebra über einer Signatur verwendet. Eine *mehrsortige Signatur* $\Sigma = \langle S, OP\rangle$ besteht aus einer Menge S von Typsymbolen, den *Sorten*, und einer Familie von Mengen von Operationssymbolen $OP = (OP_{w,s})_{w\in S^*, s\in S}$, die durch ihre Argumente und ihren Ergebnistyp indiziert sind. Statt $op \in OP_{s_1,\ldots,s_n,s}$ wird auch $op : s_1 \ldots s_n \to s$ geschrieben.

Eine Algebra für eine Signatur Σ, kurz Σ-Algebra, $A = \langle D, (op^D)_{op\in OP}\rangle$ besteht aus einer Familie $D = (D_s)_{s\in S}$ von Trägermengen für die Typsymbole in S und einer Familie von Operationen $op^D : D_{s_1} \times \ldots \times D_{s_n} \to D_s$ für jedes Operationssymbol $op : s_1 \ldots s_n \to s$ in Σ.

Die Integration von Graphen zur Repräsentation von Modellelementen mit Algebren zur Repräsentation abstrakter Datentypen führt zum Begriff der *attributierten Graphen*. Entsprechend [24] sollen Attribute als Verknüpfungen zwischen Knoten und Attributwerten angesehen werden. Ausgehend von einer Signatur $\Sigma = \langle S, OP\rangle$ besteht ein attributierter Graph (G, A) über Σ aus einem Graphen $G = \langle V, E, src, tar\rangle$ und einer Σ-Algebra A, für die gilt: $\|A\| \subseteq V$ mit $\|A\| = \uplus_{s\in S} D_s$ und $\forall e \in E : src(e) \notin \|A\|$ [1]. Von Knoten aus $\|A\|$ darf also keine Kante ausgehen. Attribute sollen nur lokal eindeutig sein. Um gleiche Attribute für verschiedene Knoten verwenden zu können, wird gefordert, dass die Kanten e zu Knoten aus $\|A\|$ eine spezielle Form haben, nämlich $e = (v, e')$. Die Menge E lässt sich wie folgt darstellen: $E = E_0 \uplus V \times E'$. Um die Zuordnung eines Attributes zu einem Knoten v auszudrücken, sollen diese Kanten weiterhin folgender Bedingung genügen: $\forall e \in E.(e = (v, e') \wedge tar(e) \in \|A\|) \Rightarrow v = src(e)$.

Datenwerte werden nach dieser Definition durch *Datenknoten* $d \in \|A\|$ eines Graphen

[1] $\uplus D_s$ ist die disjunkte Vereinigung der D_s.

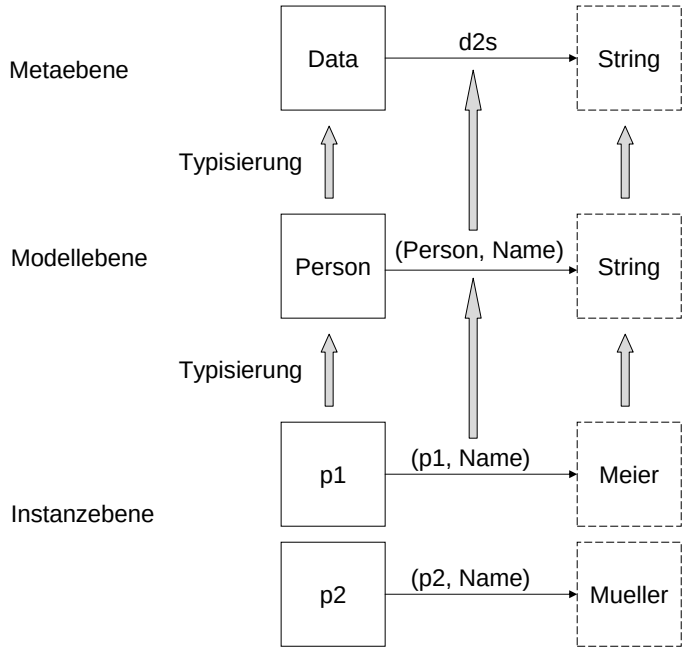

Figure 6.2: Beispielgraphen auf den drei Abstraktionsebenen

repräsentiert. Die anderen Knoten des Graphen sind *Entitätsknoten* $v \in V - \|A\|$, die mit Datenknoten durch Kanten $a \in E$ mit $src(a) = v$ und $tar(a) = d$ verbunden sind. Derartige Kanten heißen *Attribute* während Kanten zwischen Entitätsknoten *Links* genannt werden. Datenknoten haben keine ausgehenden Kanten.

Um auch attributierte Graphen typisieren zu können, ist der Begriff des Morphismus zu erweitern. Ein Σ-*Homomorphismus* $f_A : A_1 \to A_2$ zwischen zwei Σ-Algebren A_1 und A_2 ist eine Familie von Abbildungen $f_A = (f_s)_{s \in S}$, die verträglich mit den Operationen der beiden Algebren ist.

Ein *attributierter Graphmorphismus* $f : (G_1, A_1) \to (G_2, A_2)$ ist ein Paar $f = (f_G, f_A)$ aus einem Graphmorphismus $f_G = (f_V, f_E) : G_1 \to G_2$ und einem Σ-Homomorphismus $f_A : A_1 \to A_2$, so dass folgende Bedingungen gelten:

- $\|f_A\| \subseteq f_V$, wobei $\|f_A\| = \cup_{s \in S} f_s$
- $A_{1s} = f_V^{-1}(A_{2s})$ für alle $s \in S$

Die erste Bedingung besagt, dass die Abbildung der Datenknoten in der Abbildung der Graphknoten enthalten ist. Die zweite Bedingung fordert, dass Attribute aus A_1 nach A_2 abgebildet werden und dass andere Knoten aus V_1 *nicht* nach A_2 abgebildet werden. Also werden Entitätsknoten auf Entitätsknoten und Datenknoten auf Datenknoten abgebildet.

Ein *attributierter Typgraph* ist ein attributierter Graph $< TG, Z >$ über Σ, wobei Z die finale Σ-Algebra $Z = \langle D, (op^D)_{op \in OP} \rangle$ mit $D_s = \{s\}$ für alle $s \in S$ und $op^D : \{s_1\} \times \ldots \times \{s_n\} \to \{s\}$ für jedes Operationssymbol $op : s_1 \ldots s_n \to s$ in Σ ist. Für

eine gegebene Signatur Σ hat eine *finale* Σ-Algebra die Eigenschaft, dass für jede andere Σ-Algebra ein eindeutiger Morphismus auf diese finale Algebra existiert (vgl. [44]). Alle finalen Algebren zu einer Signatur Σ sind isomorph. Die finale Algebra Z ist hier so gewählt, dass die Elemente von Z die Sorten der Signatur repräsentieren, die in TG als Typen für Datenknoten eingefügt sind. Dadurch bewirkt der Σ-Homomorphismus eines Typgraph-Morphismus die Typisierung der Attribute. Aus dieser Konstruktion folgt, dass bei der Typisierung eines Typgraphen die Σ-Algebra konstant bleibt.

Die Kanten zwischen Entitäts- und Datenknoten (Attribute) enthalten als zweite Komponente den Attributnamen. Die Typisierung dieser Kanten durch den Typgraph-Morphismus erhält den Attributnamen nicht notwendigerweise. Diese Eigenschaft ist allerdings zu fordern, da Attributnamen auf Instanzebene mit denjenigen auf Modellebene übereinstimmen sollen. Generell muss also jeder Typgraph-Morphismus von der Instanz- auf die Modellebene auf Attributen bezüglich der zweiten Komponente, d.h. des Attributnamens, die identische Abbildung sein.

In Abbildung 6.2 sind die formalen Zusammenhänge an einem Beispiel dargestellt.

- Signatur Σ:
  $\Sigma = \langle S, OP \rangle$ mit $S = \{STRING\}$ und $OP = (CONCAT_{(STRING,STRING),STRING})$

- Σ-Algebra:
  $A = \langle D, (concat_{(STRING,STRING),STRING}) \rangle$ mit
  $D = (Strings_{STRING})$, wobei
  $Strings = \{a, b, \ldots, z, A, B, \ldots, Z\}^*$ und
  $concat : Strings \times Strings \rightarrow Strings, (v, w) \mapsto vw$

- Finale Algebra:
  $Z = \langle \{STRING\}, (concat^Z_{(STRING,STRING),STRING})) \rangle$ mit
  $concat^Z : \{STRING\} \times \{STRING\} \rightarrow \{STRING\}$,
  $(STRING, STRING) \mapsto STRING$

Formal wird das Beispiel in Abbildung 6.2 wie folgt notiert:

- Graph auf der Instanzebene:
  $G_I =< V_I, E_I, src_I, tar_I >$ mit
  $V_I = \{p1, p2\} \cup Strings$,
  $E_I = \{(p1, Name), (p2, Name)\}$,
  $src_I = \{((p1, Name), p1), ((p2, Name), p2)\}$ und
  $tar_I = \{((p1, Name), Meier), ((p2, Name), Mueller)\}$

- Typgraph auf der Modellebene:
  $TG_M =< V_M, E_M, src_M, tar_M >$,
  $V_M = \{Person, STRING\}$,
  $E_M = \{(Person, Name)\}$,
  $src_M = \{((Person, Name), Person)\}$ und
  $tar_M = \{((Person, Name), STRING)\}$

Typgraphmorphismus zwischen Instanz- und Modellebene:
$g_{TG} : G_I \rightarrow TG_M$,

$p1$	$\mapsto$	$Person$
$Meier$	$\mapsto$	$STRING$
$p2$	$\mapsto$	$Person$
$Mueller$	$\mapsto$	$STRING$
$(p1, Name)$	$\mapsto$	$(Person, Name)$
$(p2, Name)$	$\mapsto$	$(Person, Name)$

- Typgraph auf der Metaebene:
  $TG_{Meta} =< V_{Meta}, E_{Meta}, src_{Meta}, tar_{Meta} >$ mit
  $V_{Meta} = \{Data, String\}$,
  $E_{Meta} = \{d2s\}$,
  $src_{Meta} = \{(d2s, Data)\}$ und
  $tar_{Meta} = \{(d2s, String)\}$

  Typgraphmorphismus zwischen Modell- und Metaebene:
  $g_{TG_{Meta}} : TG_M \rightarrow TG_{Meta}$,

$Person$	$\mapsto$	$Data$
$String$	$\mapsto$	$String$
$(Person, Name)$	$\mapsto$	$d2s$

Als Fazit ist festzuhalten, dass sowohl die Zuordnung von Attributen zu Elementen von Graphen als auch deren Typisierung durch die Theorie der Graphtransformation mathematisch präzise beschrieben werden kann. Damit steht ein wohlfundierter semantischer Bereich für Modelle bereit.

Dynamisches Modell. Ein dynamisches Modell ist eine Menge von Graphtransformationsregeln. Graphtransformationsregeln bestehen aus einem Paar von Graphen, der linken und rechten Seite. Die Wirkung von Graphtransformationsregeln auf Instanzgraphen zur Beschreibung eines Zustandswechsels wird für Graphtransformationsregeln durch den Begriff der *Transformation* geleistet. Durch eine Transformation ist erklärt, wie Regeln auf einen Ausgangszustand angewendet werden, um einen Nachfolgezustand zu erreichen. Der Zusammenhang zwischen Graphtransformationsregeln und Graphtransformationen wird im Rahmen des *double pushout*-Ansatzes (DPO-Ansatzes) mengentheoretisch beschrieben [45]:

DPO-Ansatz. Eine *Graphtransformation* aus einem Vorzustand G in einen Nachzustand H, beschrieben durch $r(o) : G \Longrightarrow H$, ist durch einen Teilgraphisomorphismus $o : L \cup R \rightarrow G \cup H$ gegeben, *occurrence* genannt, so dass

- $o(L) \subseteq G$ und $o(R) \subseteq H$, d.h. die linke Seite der Regel ist in den Vorzustand und die rechte Seite ist in den Nachzustand eingebettet:

- $o(L \setminus R) = G \setminus H$ und $o(R \setminus L) = H \setminus G$. Also wird genau der Teil von G gelöscht, auf den die Elemente von L abgebildet werden, die nicht zu R gehören, und in symmetrischer Weise wird genau der Teil von H hinzugefügt, auf den die Elemente von R abgebildet werden.

Dieses ist eine mengentheoretische Präsentation von Graphtransformation nach dem DPO-Ansatz [45] (siehe [92] für einen Überblick), der auf injektive Ansätze beschränkt ist. In seiner allgemeinen Form wird der DPO-Ansatz kategorientheoretisch durch zwei verbundene *Push-Out*-Diagramme definiert (vgl. Ehrig et al. [45]).

Der beschriebene DPO-Ansatz stellt eine Einschränkung der Anwendbarkeit einer Transformationsregel in einem gegebenen Zustand dar: eine Regel, die eine Instanz löscht, darf nicht angewendet werden, falls dadurch eine Kante ohne Quell- oder Zielknoten erhalten bleibt. Diese *dangling condition* ist das konservative Verfahren, um sicher zu stellen, dass die resultierende Struktur wieder ein wohldefinierter Graph ist, d.h. dass die Abhängigkeit zwischen Kanten und ihren Quell- und Zielknoten bewahrt wird. Es werden nur die explizit in der Regel spezifizierten Teile des Kontextes gelöscht (vgl. [45]). Durch diese Eigenschaft eignet sich der DPO-Ansatz für die Beschreibung von Verhalten in Form von Zustandsänderungen, ohne unbeabsichtigte Seiteneffekte nach sich zu ziehen.

6.3 Graphtransformationssysteme und das Systemmodell

Für die Formulierung von Modellen im semantischen Bereich der Graphtransformation wird der Begriff des Graphtransformationssystems benötigt. Ein solches System bündelt einen Typgraphen, eine darüber getypte Regelmenge, eine Menge von getypten Startgraphen und eine Menge von Bedingungen, denen alle zu dem Graphtransformationssystem gebildeten Graphen entsprechen sollen. Formal besteht folgende Definition:

Ein *getyptes (attributiertes) Graphtransformationssystem* (TAGTS) $\mathcal{G} = \langle TG, RS, SG, C \rangle$ besteht aus einem Typgraphen TG, einer Menge RS von Graphtransformationsregeln $p : L \to R$ über TG, einer Menge SG von Startgraphen und einer Menge struktureller Bedingungen C über TG, wie Kardinalitätsbedingungen, Prädikate, etc.

Graphtransformationssysteme sind Elemente der Modellebene, die durch die Regeln Abläufe in einem System *spezifizieren*. Um Abläufe diskutieren zu können, ist der Begriff des Systems zu klären. Die Details einer speziellen Implementierungsplattform sollen nicht berücksichtigt werden. Stattdessen soll ein allgemeines Systemmodell wesentliche Eigenschaften zu implementierender Systeme reflektieren (vgl. Abb. 6.1). Das Systemmodell legt dazu formal fest, wie ein System aufgebaut ist und wie es sich verhält. Ein Systemmodell bildet eine mathematisch präzise Beschreibung eines Systems und bestimmt dadurch genaue Anforderungen an einen Interpreter. Dieser berechnet aus dem Systemmodell als Eingabe einen Ablauf des Systems als Ausgabe.

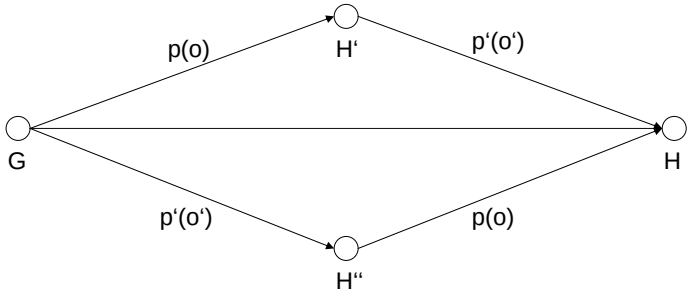

Figure 6.3: Zustände und Übergänge im Transitionssystem $T_{\mathcal{G}}$

Durch sukzessive Anwendung von Regeln auf die Startgraphen eines Graphtransformationssystems $\mathcal{G}$ werden neue Graphen abgeleitet. Die Typisierung der Regeln sorgt dafür, dass auch die abgeleiteten Graphen korrekt typisiert sind. Weiterhin sollen Ableitungen nur möglich sein, falls entstehende Graphen allen Bedingungen aus C entsprechen. Alle auf diese Weise entstehenden Graphen werden als Zustände des Systemmodells aufgefasst. Die Abläufe eines solchen Systems sind durch die Ableitungen, d.h. Folgen von Graphtransformationen ausgehend von Startzuständen gegeben. Die Menge der Graphtransformationen, die in Ableitungen vorkommen, bildet die operationale Semantik eines Graphtransformationssystems.

Die operationale Semantik eines gegebenen Graphtransformationssystems $\mathcal{G}$ ist formal die Menge aller möglichen Graphtransformationen $G \Longrightarrow_{RS} H$, die durch Anwendung der Regeln aus RS entstehen, ausgehend von einem abgeleiteten Graphen G: $opSem(\mathcal{G}) = \{G \Longrightarrow_{RS} H | (\exists p : L \to R \in RS)\ (\exists o : L \cup R \to G \cup H)\ p(o) : G \Longrightarrow H \ \wedge\ S \Longrightarrow^*_{RS} G \ \wedge\ S \in SG\}$. Die Menge SG kann insbesondere auch aus allen über TG getypten Graphen bestehen.

Die Theorie der Graphtransformationssysteme liefert ein Berechnungsmodell mit ähnlichen Eigenschaften wie S/T-Petrinetze (vgl. [6]). Folgen von Transformationsschritten werden zu partiellen Ordnungen abstrahiert, wobei alle Folgen, die sich nur durch die Anordnung der unabhängigen Schritte unterscheiden, identifiziert werden (vgl. [6]). Wesentlich ist die Eigenschaft, dass zwei Transformationsschritte unabhängig sind, d.h. in beliebiger Reihenfolge oder parallel ausgeführt werden können, falls auf gemeinsame Ressourcen nur lesend zugegriffen wird. Diese Eigenschaft von Transformationsschritten heißt *Konfluenz*.

Die operationale Semantik enthält keine Transformationen, die durch parallele Anwendung von Regeln entstehen. In einer erweiterten operationalen Semantik bestehen direkte Übergänge zwischen Graphen auch dann, wenn eine konfluente Ableitung aus dem Ausgangsgraphen in den Zielgraphen existiert: $opSem'(\mathcal{G}) = opSem(\mathcal{G}) \cup \{G \Longrightarrow_{par} H | G \Longrightarrow^+_{RS} H\}$. In dieser modifizierten Semantik ist paralleles Verhalten durch die neuen direkten Übergänge repräsentiert. In Abbildung 6.3 sind die Übergänge in der erweiterten operationalen Semantik für zwei unabhängige Graphtransformationen dargestellt. Da in der erweiterten Semantik sowohl sequentielle als auch parallele Übergänge beschrieben

werden, wird diese Semantik ausgewählt, um das nebenläufige Verhalten eines Systems adäquat zu beschreiben.

Die aus einem Graphtransformationssystem resultierenden Graphtransformationen und die gebildeten Graphen legen ein Systemmodell in Form eines markierten Transitionssystems (labeled transition system, lts) fest. Ein Transitionssystem T ist durch $T = \langle Q, L, Q_0, \rightarrow \rangle$ mit der Menge der Zustände Q, einem endlichen Alphabet L, einer Menge von Startzuständen $Q_0 \subseteq Q$ und einer Zustandsübergangsrelation $\rightarrow \subseteq Q \times L \times Q$ definiert. Die übliche Schreibweise für $(q, l, q') \in \longrightarrow$ ist $q \xrightarrow{l} q'$.

Das Transitionssystem $T_{\mathcal{G}}$ zu einem Graphtransformationssystem $\mathcal{G}$ besteht aus der Zustandsmenge $Q = \{G | G \text{ ist über } TG \text{ getypter Graph }\}$, dem Alphabet $L = opSem'(\mathcal{G})$, der Anfangszustandsmenge $Q_0 = SG$ und der Übergansrelation $\longrightarrow$ mit $q \xrightarrow{l} q'$ gdw. $q, q' \in Q$, $l \in L$ und $l : q \Longrightarrow q' \in opSem'(\mathcal{G})$.

Die Ausführung von Zustandsübergängen ist Aufgabe eines (abstrakten) Interpreters. In jedem Zustand wählt der Interpreter unter mehreren, im Transitionssystem möglichen Übergängen einen Übergang aus und führt die zugehörigen Modifikationen am Graphen des Ausgangszustands aus. Im Ergebnis befindet sich das System in dem Nachfolgezustand, der durch das Transitionssystem angegeben ist. Die Art der Auswahl hängt vom geforderten Interpreterverhalten ab, welches zum Verhalten des zu implementierenden Systems konform sein soll. Sollen beispielsweise rein sequentielle Abläufe modelliert werden, dann wählt der Interpreter keine Übergänge zu parallelen Regelanwendungen. Nichtdeterministisches Systemverhalten kann durch eine nichtdeterministische Auswahl eines Übergangs modelliert werden.

Der vorgestellte semantische Bereich der Graphtransformation wird jetzt so eingeschränkt, dass agentenorientierte Konstituenten berücksichtigt werden.

6.4 Modellkonstituenten im agentenorientierten semantischen Bereich

Aus den grundlegenden Eigenschaften von Agenten sind in Kapitel 4 notwendige Konstituenten für ein Agentenmodell abgeleitet worden. In diesem Abschnitt werden die Konstituenten im Rahmen der vorgestellten Spracharchitektur präzise erfasst. Im ersten Schritt werden dazu die Konstituenten im semantischen Bereich der Graphen repräsentiert.

6.4.1 Der agentenorientierte semantische Bereich

Auf der *Instanzebene* werden Systemzustände als Graphen definiert. Alle Elemente der Instanzebene sind Knoten eines Graphen. Attribute sind Kanten von Entitätsknoten, die beispielsweise Container-Objekte für Daten (Datenobjekte) repräsentieren, zu Datenknoten, die Attributwerte darstellen. Änderungen von Attributwerten erfolgen dadurch, dass eine Attributkante vom Entitätsknoten auf einen anderen Attributwert zeigt. Für alle

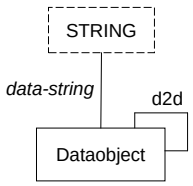

Figure 6.4: Der Metatypgraph TG_{Data}

möglichen Attributwerte des Attributwertebereichs existieren in jedem Instanzgraphen zugehörige Knoten. Für unendliche Attributwertemengen existieren also unendlich viele Datenknoten. Für praktische Probleme stellt das keine Einschränkung dar, da immer nur auf endlich viele Attributwerte Bezug genommen wird. Die restlichen, unendlich vielen Attributwerte werden nicht berücksichtigt (vgl. das Beispiel in Abschnitt 6.2).

Jeder Instanz eines Knotens oder einer Kante auf der Instanzebene ist auf der *Modellebene* zur Typisierung ein Knoten bzw. eine Kante in einem attributierten Typgraphen zugeordnet. Die Namen der Knoten und Kanten bezeichnen den Typ. Diese Struktur wird aus dem semantischen Bereich der Graphtransformation unmittelbar übernommen. Jedem Attributwert der Instanzebene ist auf der Modellebene ein Knoten zugeordnet, der durch eine Sorte, d.h. einen Datentyp bezeichnet ist. O.B.d.A. wird hier nur der Datentyp STRING verwendet. Die Typisierungsabbildung wird dahingehend eingeschränkt, dass die zweite Komponente von Attributbezeichnern, d.h. die eigentlichen Attributnamen, durch die Typisierung erhalten bleiben (vgl. das Beispiel in Abschnitt 6.2). Dadurch wird erreicht, dass ein Entitätsknoten auf Instanzebene genau die Attribute hat, die auf Modellebene vorgegeben sind.

Auf der *Metaebene* werden die Elemente der Sprache definiert. Knoten und Kanten eines *Metatypgraphen* sind Metatypen für Typen im Modellgraphen entsprechend dem vorgestellten semantischen Bereich der Graphtransformation. Formal werden die elementaren Sprachelemente der agentenorientierten Modellierungssprache in einem attributierten Metatypgraphen TG_{Data} erfasst.

Elementare Knoten des Metatypgraphen TG_{Data} sind der Entitätsknoten Dataobject und der Datenknoten STRING, der die einzige verwendete Sorte bezeichnet (vgl. Abbildung 6.4). Datenknoten sind gestrichelt umrandet. Kanten werden von einem Entitätsknoten zum Datenknoten STRING dann gezogen, wenn der Entitätsknoten Attribute vom Typ STRING besitzen soll. In TG_{Data} besagt die Kante data-string, dass Datenobjekte Attribute des Typs String besitzen können. Die Kante d2d ermöglicht, Kanten zwischen Datenobjekten ziehen zu können und diese damit in Beziehung zueinander zu setzen.

6.4.2 Autonomie

Konstituierend für die Autonomie eines Agenten sind gemäß Kapitel 4 *interne, flexible Kontrollelemente* für die Entscheidungsautonomie von Agenten und der eigene *Ausführungsfaden* für die Handlungsautonomie von Agenten, durch den nebenläufiges Ver-

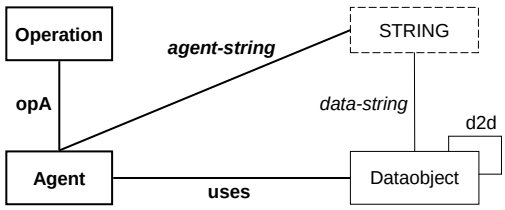

Figure 6.5: Der Metatypgraph TG_{Agent}

halten in Bezug auf andere Agenten beschrieben wird.

Der interne Zustand von Agenten. Agenten kontrollieren sowohl ihren Zustand als auch ihr Verhalten. Die internen Zustandsvariablen eines Agenten lassen sich in zwei Gruppen aufteilen. Zur einen Gruppe gehören die Variablen, die während der gesamten Lebensdauer eines Agenten existieren. Das sind die *Attribute* eines Agenten. Zur anderen Gruppe gehören die Variablen, die vom Agenten selbst während seiner Laufzeit erzeugt, genutzt und entfernt werden können. Diese Gruppe wird benötigt, damit ein Agent Daten in flexibler Weise speichern und mit anderen Agenten austauschen kann. Diese Variablen sind als Attribute in *Datenobjekten* gebündelt.

Die soeben diskutierten Konstituenten für autonomes Verhalten von Agenten werden im Metatypgraph TG_{Agent} erfasst (vgl. Abbildung 6.5). Grundlegende Metatypen sind Agent und Operation. Verbindungen zwischen Agenten und Datenobjekten werden durch die Kante uses typisiert. Attribute von Agenten werden durch die Kante agent-string ausgedrückt.

Agentenoperationen und Graphtransformationsregeln. Das Verhalten eines Agenten ist durch dem Agenten zugeordnete Graphtransformationsregeln gegeben. Jede Graphtransformationsregel wird durch einen eindeutigen Operationsnamen identifiziert. Im Typgraphen eines Agentenmodells sind die Operationsnamen mit Agententypen durch Kanten verbunden. Im Metatypgraphen TG_{Agent} sind deshalb Operationsnamen durch den Knoten Operation typisiert und die Kanten von Operationsnamen zu Agententypen durch die Kante opA typisiert. Zu jedem Operationsnamen gibt es genau eine opA-Kante, d.h. genau einen Agententyp.

In einem Graphtransformationssystem (auf Modellebene) muss zu jedem Operationsnamen im Typgraphen genau eine Graphtransformationsregel existieren. Die Zuordnung von Graphtransformationsregeln zu Agententypen führt zur Forderung, dass die in der linken und rechten Regelseite enthaltenen Elemente folgenden Einschränkungen unterliegen:

- Nur Attribute des Agenten, dem die Graphtransformationsregel zugeordnet ist, dürfen verändert werden.

- Nur die Datenobjekte dürfen durch eine Graphtransformationsregel erzeugt oder gelöscht werden, oder deren Attribute dürfen verändert werden, die vom Agenten der Regel aus erreichbar sind.

Die Handlungsautonomie eines Agenten induziert die Frage, inwieweit der Agent unabhängig und damit nebenläufig zu anderen Agenten agieren kann. Indem Operationen von Agenten durch Graphtransformationsregeln beschrieben werden, ist nebenläufiges Verhalten in dem beschriebenen Systemmodell integriert. Die in Abschnitt 6.3 gewählte operationale Semantik von Graphtransformationssystemen beschreibt sowohl sequentielle als auch parallele Anwendungen von Graphtransformationsregeln. Unterschiedliche Agenten können also Operationen nebenläufig ausführen. Insbesondere kann aber auch ein einzelner Agent für einen geeigneten Regelsatz mehrere Operationen nebenläufig ausführen. Dieser Aspekt wird besonders im nächsten Abschnitt wichtig, da ein Agent auch mehrere Protokolle abarbeiten kann. Dieses Abarbeiten kann auch nebenläufig erfolgen, falls die Regeln der unterschiedlichen Protokolle das zulassen.

Die Möglichkeit, unterschiedliche Graphtransformationsregeln mit gleicher linker Seite in einem Zustand, in dem dieses Muster auftritt, anzuwenden, korrespondiert mit unterschiedlichen Zustandsübergängen im Systemmodell. Ein (abstrakter) Interpreter kann eine nichtdeterministische Auswahl treffen, um nichtdeterministisches Verhalten von Agenten zu beschreiben. Diese Auswahl bildet die Entscheidung für eine Operation und ihre Durchführung, d.h. die Anwendung der zugehörigen Graphtransformationsregel. Die Entscheidungsautonomie eines Agenten manifestiert sich damit im Verhalten des (abstrakten) Interpreters.

Im nächsten Schritt werden die Konstituenten für strukturierte Interaktion im agentenorientierten semantischen Bereich repräsentiert.

6.4.3 Strukturierte Interaktion

Die Modellierungssprache AML soll das Interaktionsverhalten von Agenten beschreiben können. Um die Wiederverwendung von oftmals auftretenden Interaktionsmustern zu unterstützen, wird die Interaktion zwischen Agenten durch Protokolle beschrieben. Protokolle bilden Vorlagen (templates) für koordiniertes Verhalten von Agenten (vgl. z.B. [50]). Die Wiederverwendbarkeit erfordert, dass die Beschreibung von Protokollen nicht an konkrete Agenten gebunden ist. Stattdessen werden *Rollen* an Stelle der Agenten in Protokollen eingesetzt. Rollen bündeln die Zustandsinformation und das Verhalten, durch die die lokale Beteiligung an einem Protokoll beschrieben werden kann. Die strukturierte Interaktion zwischen Agenten wird also durch Protokolle zwischen Rollen beschrieben, welche von Agenten eingenommen werden.

Viele Anforderungen an Rollen sind von Kristensen und von Gottlob et al. in einem objektorientierten Kontext erarbeitet worden (vgl. [78, 60]). Die Anforderungen sind allgemein genug, um auch hier gültig zu sein (vgl. Depke et al. [37]). Ein Rollenkonzept sollte folgende Eigenschaften von Rollen unterstützen:

- *Sichtbarkeit*: Die Sichtbarkeit und der Zugriff auf einen Agenten ist auf Attribute der Rolle oder solche des Agenten selbst beschränkt.
- *Abhängigkeit der Lebensdauer*: Die Lebensdauer einer Rolle hängt von der des Agenten ab, zu dem sie gehört.
- *Schwache Identität*: Rollen haben eine von einem Agent abhängige Identität und bilden mit diesem eine Einheit. Daraus folgt, dass eine Rolle nicht gleichzeitig zu verschiedenen Agenten gehören kann, sondern zu genau einem.
- *Dynamizität*: Während der Lebenszeit eines Agenten kann er Rollen annehmen und ablegen.
- *Multiplizität*: Mehrere Instanzen eines Rollentyps dürfen gleichzeitig zu einem Agent existieren.

Die genannten Eigenschaften sind für Rollen von Agenten aus folgenden Gründen notwenig: Genauso wie für Agenten ist auch für Rollen die Unterscheidung zwischen Typ- und Instanzebene sinnvoll, um durch *Rollentypen* Rollen mit ähnlichen Eigenschaften und ähnlichem Verhalten zusammenzufassen. *Rolleninstanzen* sind aus mehreren Gründen notwendig. Zum einen können sie Zustandsinformationen tragen, die notwendig sind, damit Rollen als Teilnehmer an Protokollen den Zustand der Interaktion speichern können. Zum anderen kann ein Agent mehrere Rolleninstanzen benötigen (*Multiplizität*), falls zwei oder mehr Rollen des gleichen Typs von einem einzelnen Agenten eingenommen werden können.

Eine Rolleninstanz ist genau einem Agenten zugeordnet, da sie die Interaktion des Agenten in einem Protokoll abwickeln soll. Dieser Agent ist der *Basisagent* der Rolle. Auf Instanzebene hängt die Identität der Rolle von der des Agenten ab. Aufgrund dieser Funktion ist auch die Lebensdauer der Rolleninstanz durch die Lebensdauer des Agenten begrenzt. Eine derartige dynamische Integritätsbedingung ist beispielsweise auch bei schwachen Entitäten in Entity-Relationship-Modellen zu finden (vgl. Heuer/Saake [61]). Die *schwache Identität* und *Lebensdauerabhängigkeit* sind also charakteristisch für Rollen von Agenten.

Dynamizität resultiert daraus, dass ein Protokoll erst zur Laufzeit benötigt werden kann und die zugehörigen Rolleninstanzen erst vor Beginn der Abarbeitung des Protokolls erzeugt werden können.

Rollen haben die Funktion, im Kontext von Protokollen Schnittstellen für Agenten zu bilden. Rollen bestimmen damit die *Sichtbarkeit* von Eigenschaften und den Zugriff auf den inneren Zustand von Agenten.

Die beschriebenen charakteristischen Merkmale *Instanziierung*, *Multiplizität*, *schwache Identität*, *Lebensdauerabhängigkeit*, *Dynamizität* und *Sichtbarkeit* bilden damit einen Satz von Anforderungen an die Modellkonstituente Rolle, die in einer agentenorientierten Modellierungssprache erfüllt sein sollen.

Der Metatypgraph TG_{Agent} (siehe Abb. 6.5) wird entsprechend um einen Knoten Role erweitert, der durch eine Kante role-of mit dem Knoten Agent verbunden ist (siehe

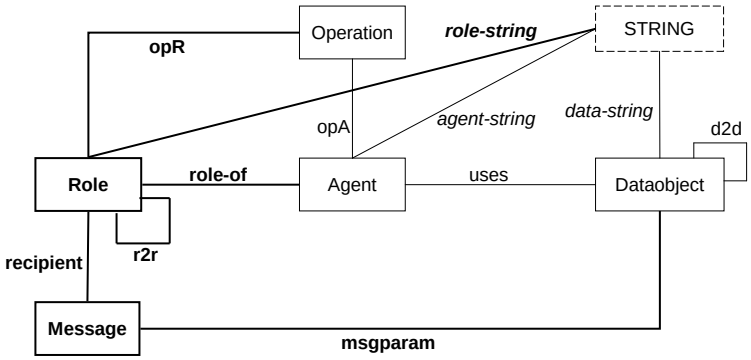

Figure 6.6: Der Metatypgraph TG_{AOSE}

Abb. 6.6). Rollen können genau wie Agenten Attribute haben, die den Zustand der Interaktion speichern. Die Knoten Role und STRING sind deshalb durch die Kante role-string miteinander verbunden. Im Unterschied zu Agenten können Rollen Nachrichten empfangen. Ein entsprechender Knoten Message ist durch die Kante recipient mit Role verbunden. Operationen von Rollen sollen Nachrichten senden und empfangen. Rollen, die einander Nachrichten schicken können, sind durch die Kante r2r verbunden. Für die Verknüpfung von Operationen und Rollen existiert die Kante opR im Metatypgraph TG_{AOSE}.

Die Eigenschaften von Rollen führen zu einer Reihe von Randbedingungen, die von Graphtransformationssystemen (auf Modellebene) erfüllt sein müssen:

- Ein- und Ausgangsgrade (Kardinalitäten) von Knoten für Kanten eines bestimmten Typs sind bisher nicht im Metamodell enthalten. Minimale und maximale Kardinalitäten sind als zusätzliche Bedingungen für einen Typgraphen anzugeben.

- Die Bedingungen für die role-of Kante ergeben sich wie folgt: Die schwache Identität von Rollen impliziert, dass jede Rolleninstanz mit genau einer Basisagenteninstanz durch eine role-of Kante verbunden ist. Auf der anderen Seite können mehrere Rolleninstanzen mit derselben Instanz eines Basisagenten verbunden sein, wie das die Eigenschaft der Multiplizität verlangt.

- Zu jeder Operation im Typgraphen muss eine Graphtransformationsregel existieren, die diese Operation realisiert.

- Graphtransformationsregeln zu Operationen in Rollen müssen Nachrichten senden oder empfangen. Beim Senden wird einer oder mehreren Rollen eine Nachricht angehängt. Beim Empfangen wird lokal eine Nachricht gelöscht. Bei beiden Arten von Operationen wird ansonsten lediglich der lokale Zustand modifiziert.

Protokolle sind auf Modellebene durch ein Graphtransformationssystem $\mathcal{G}_{Protocol} = \langle TG_p, RS_p, SG_p, C_p \rangle$ gegeben. Der Typgraph TG_p des Protokolls ist durch den Metatypgraph TG_{AOSE} getypt. In Typgraphen von Protokollen kommen allerdings keine Agententypen vor, da Protokolle unabhängig von Agenten rein rollenbasiert definiert werden.

Die Regeln in RS_p operieren lediglich auf durch TG_p getypten Rollen und Datenobjekten. Da Regeln den Nachrichtenaustausch zwischen Rollen beschreiben sollen, enthält jede Regel mindestens eine Rolle. Die Menge der Startgraphen SG_p ist leer, da Protokolle nur verwendet werden, nachdem sie in einen Kontext von Agenten eingebunden sind. Die Menge C_p enthält beispielsweise Kardinalitätsbedingungen für Verknüpfungen zwischen Elementen.

Der Kontext ist durch ein System-Graphtransformationssystem $\mathcal{G} = \langle TG, RS, SG, C \rangle$ gegeben, das Agenten enthält, die gemäß Protokollen kommunizieren sollen. Kommunikation ist nach dem Binden von Protokollen möglich. Das Binden von Protokollen wird durch Integration des Protokoll-Graphtransformationssystems $\mathcal{G}_{Protocol}$ in das System-Graphtransformationssystem $\mathcal{G}$ erreicht. Dazu werden die Typgraphen, die Mengen der Startgraphen und die Mengen der Bedingungen zu TG', SG' und C' vereinigt. Für jede Regel des Protokolls wird für jede Rolle ein zugehöriger Agententyp aus dem Typgraphen TG festgelegt, der die Regel anwenden soll. Agenten entsprechenden Typs werden in die Regeln eingefügt und durch eine role-of-Kante mit den zugehörigen Rollen verbunden. Die neuen Regeln bilden zusammen mit den Regeln aus RS die Regelmenge RS'. Im Ergebnis entsteht das System-Graphtransformationssystem $\mathcal{G}' = \langle TG', RS', SG', C' \rangle$.

Die Anforderungen aus Kapitel 4 an strukturierte Interaktion sind vollständig erfüllt:

- Die Funktion (Semantik) eines Protokolls ist durch das Verfahren zur Bindung von Protokoll-Graphtransformationssystemen eindeutig beschrieben.
- Die Voraussetzungen über die Umgebung sind durch das System-Graphtransformationssystem gegeben.
- Das "Vokabular" und der Aufbau (die Syntax) der Nachrichten sind durch den Typgraphen des Protokoll-Graphtransformationssystems bestimmt.
- Prozedurale Regeln (das Verhalten) für den Nachrichtenaustausch sind durch die Graphtransformationsregeln eines Protokoll-Graphtransformationssystems festgelegt.

Im nächsten Schritt werden die Konstituenten für Proaktivität im semantischen Bereich repräsentiert.

6.4.4 Proaktivität

Die Konstituenten für Proaktivität sind Ziele und Strategien (vgl. Kapitel 4). Aus Sicht der Softwaretechnik korrespondiert ein Ziel mit einer funktionalen Anforderung, die durch das Verhalten eines oder mehrerer Agenten zu erfüllen ist. Traditionell werden solche Anforderungen separat von der Software in einem Anforderungsdokument angegeben. Im Unterschied dazu kontrollieren bei zielgerichtetem (proaktivem) Verhalten die Ziele selbst zur Laufzeit die Abläufe in einem System.

Ausgehend von einem Zustand, in dem die Vorbedingung für ein Ziel erfüllt ist, definieren die Nachbedingungen für ein Ziel einen *geforderten* Zustand, den ein Agent

durch eigenes Verhalten zu erreichen trachtet. Eine mit einem Ziel verbundene Strategie bildet eine Zerlegung in Teilziele, die vom Agenten direkter realisiert werden können. Die unmittelbare Ausführung einer Operation zum Erreichen eines Zieles kann dadurch verhindert sein, dass der Agent für die Ausführung auf Informationen eines anderen Agenten angewiesen ist und deshalb mit diesem erst kommunizieren muss. Für eine Strategie ist zu überprüfen, dass sie ein avisiertes Ziel erreicht, d.h. eine Gewinnstrategie ist (vgl. Anforderungen in Kapitel 4).

Im Kontext von Graphtransformationen dienen Graphtransformationsregeln der Modellierung von Zielen und Strategien (vgl. Depke et al. [38, 29, 32]). Eine Graphtransformationsregel repräsentiert die Vor- und Nachbedingung für eine Operation, die in einem Vor- und in einem Nachzustand erfüllt sein müssen. Die Modellierung von Zielen durch Graphtransformationsregeln liefert eine operationale Interpretation für das Erreichen von Zielen: Ein Ziel ist erreicht, falls ein Zustand erreicht wird, in dem die rechte Seite der Graphtransformationsregel matcht. Die linke Seite matcht den Ausgangszustand, von dem aus das Ziel erreicht werden soll. Die in Abschnitt 6.3 vorgestellte nebenläufige Semantik von Graphtransformation erlaubt, dass unabhängige Ziele in beliebiger Reihenfolge oder parallel verfolgt werden können.

Um überprüfen zu können, ob Strategien zugehörige Ziele realisieren, müssen die Begriffe formalisiert werden. Eine Strategie hat keinen Startgraph.

Strategie. Eine *Strategie* ist ein Graphtransformationssystem $\mathcal{G} = \langle TG, RS, \emptyset, C \rangle$ (vgl. Seite 91).

Transformationsfolge. Eine Transformationsfolge $s = (G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n)$ zu einem Graphtransformationssystem $\mathcal{G}$, kurz $G_0 \overset{s}{\Longrightarrow}^*_{\mathcal{G}} G_n$, ist eine Folge von aufeinander folgenden Transformationen, die die Regeln von $\mathcal{G}$ so verwenden, dass alle Graphen $G_0, \ldots, G_n$ die Bedingungen in C erfüllen. Die Menge aller Transformationsfolgen in $\mathcal{G}$ wird durch $\mathcal{G}^*$ bezeichnet. Neu erzeugte Elemente bekommen wieder unverbrauchte Namen, d.h. solche, die zuvor nicht in Transformationsfolgen verwendet wurden. Dadurch ist für alle $0 \leq i < j \leq n$ der Durchschnitt $G_i \cap G_j$ wohldefiniert und repräsentiert den Teil der Struktur, der durch die Transformation von G_i nach G_j erhalten bleibt.

Diese Eigenschaft ist für die Definition von Zielen wichtig. Ein Ziel ist eine Graphtransformationsregel, deren linke Seite in den Anfangszustand und deren rechte Seite in den Endzustand einer Transformationsfolge eingebettet werden kann. Dabei sollen Elemente, die sowohl in der linken als auch der rechten Regelseite vorkommen, auf Teile des Anfangs- und Endzustands abgebildet werden, die über die *ganze* Transformationsfolge hinweg erhalten werden, d.h. nicht etwa zwischendurch gelöscht und dann als neue Elemente mit altem Namen wieder erzeugt wurden.

Ziel. Ein *Ziel* ist eine Graphtransformationsregel $p : L \to R$. Eine Transformationsfolge $G_0 \overset{s}{\Longrightarrow}^*_{\mathcal{G}} G_n$ erfüllt eine Graphtransformationsregel p, falls alle die Effekte, die durch p

gefordert sind, durch die Folge realisiert werden. Das ist der Fall, falls ein Graphhomomorphismus $o : L \cup R \to G_0 \cup G_n$, genannt *occurrence*, existiert, für den gilt

- $o(L) \subseteq G_0$ und $o(R) \subseteq G_n$, d.h. die linke Regelseite ist in den Vorzustand und die rechte Seite ist in den Nachzustand der Regel eingebettet und
- $o(L \setminus R) \subseteq G_0 \setminus G_n$ und $o(R \setminus L) \subseteq G_n \setminus G_0$, d.h. wenigstens der Teil von G_0 wird gelöscht, der durch Elemente von L gematcht wird, die nicht zu R gehören und symmetrisch dazu wird der Teil von G_n hinzugefügt, der durch neue Elemente in R gematcht wird.

Gewinnstrategie. Auf dieser formalen Basis kann die Beziehung zwischen Zielen und Strategien definiert werden: Eine Zielregel p sollte durch ein Graphtransformationssystem $\mathcal{G}$ erfüllt werden, das durch eine Menge von Regeln R, einen Typgraphen TG und eine Menge von Bedingungen C gegeben ist. Das impliziert, dass alle Transformationsfolgen in $\mathcal{G}$, von einem Zustand, der die Vorbedingung von p erfüllt, zu einem Zustand führen, der die Nachbedingung von p erfüllt.

Formal ist eine Strategie $\mathcal{G} = \langle TG, RS, \emptyset, C \rangle$ eine *Gewinnstrategie* für eine Zielregel $p : L \to R$, falls für alle Graphen G_0, die die Bedingungen C erfüllen, folgendes gilt: Falls die linke Seite von p in G_0 durch den Graphhomomorphismus $o_L : L \to G_0$ gematcht wird, dann existiert für alle Transformationsfolgen $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ in $\mathcal{G}$ ein $i \in \{0, \ldots, n\}$, so dass $G_0 \Rightarrow^*_{\mathcal{G}} G_i$ die Zielregel p durch das matching $o : L \cup R \to G_0 \cup G_i$ erfüllt und dabei o_L erweitert.

In Kapitel 7 wird ein Verfahren auf der Basis von *Model Checking* vorgeschlagen, mit dem überprüft werden kann, ob eine Strategie tatsächlich zu einem Ziel führt, d.h. ob sie eine *Gewinnstrategie* ist.

Die Begriffe Ziel und Strategie sollen jetzt in den Kontext von Agentenmodellen eingebettet werden. Strategien, die mehrere Agenten betreffen, sind *Kommunikationsstrategien*. Im folgenden sollen allein die im letzten Abschnitt vorgestellten Protokolle als Kommunikationsstrategien dienen. Die Bindung eines Protokolls an eine Menge von Agenten stellt die Anwendung einer Kommunikationsstrategie dar. Das Ziel eines Protokolls kann im einfachsten Fall durch eine triviale Regel gegeben sein, die auf der linken und der rechten Seite nur die am Protokoll beteiligten Rollen vor Beginn bzw. nach Ende der (erfolgreichen) Protokollabarbeitung enthält. Nichttriviale Zielregeln spezifizieren darüber hinaus Anforderungen an den zu erreichenden Zustand. Eine Zielregel wird dadurch gebunden, dass entsprechend der Bindung des zugehörigen Protokolls die in der Regel enthaltenen Rollen mit Agenten durch eine role-of-Beziehung verknüpft werden. Ein Beispiel für ein Protokoll und das zugehörige Ziel wird auf syntaktischer Ebene nach Einführung der Syntax der Sprache AML in Abschnitt 6.5 vorgestellt.

Neben Kommunikationsstrategien für eine Menge von Agenten besitzen einzelne Agenten lokale Ziele und Strategien. Für ein lokales Ziel enthält die Zielregel genau einen Agenten sowohl auf der linken als auch auf der rechten Regelseite. Weiterhin sind Rollen

und Datenobjekte erlaubt, mit denen der Agent verbunden sein darf. Eine lokale Strategie zu einem lokalen Ziel hat die Aufgabe, die Ergebnisse verschiedener Protokollabarbeitungen eines Agenten durch zusätzliche Operationen so zu integrieren, dass das lokale Ziel erreicht wird.

Auf formaler Ebene ist eine Kommunikationsstrategie durch ein Graphtransformationssystem $\mathcal{G}_p$ eines Protokolls gegeben. Die Protokolle werden in der Menge *protocols*, die lokalen Strategien in der Menge *localstrategies* versammelt. Die durch Graphtransformationsregeln repräsentierten Ziele sind in der Menge *goals* vereinigt. Der Zusammenhang zwischen Zielen und Strategien wird durch die Abbildung *strategy4goal* hergestellt:

$$\begin{aligned} \mathit{strategy4goal} : \mathit{goals} &\rightarrow \mathit{protocols} \cup \mathit{localstrategies} \\ p &\mapsto \mathcal{G} \end{aligned}$$

Damit sind Ziele und Strategien semantisch repräsentiert. Nach der Diskussion, wie sich die Komposition von Konstituenten im agentenorientierten semantischen Bereich reflektiert, wird der agentenorientierte semantische Bereich danach zusammengefasst dargestellt.

6.4.5 Komposition von Konstituenten

Hinsichtlich der Komposition von Konstituenten ist zu diskutieren, inwieweit die Anforderungen an die Komposition von Konstituenten aus Kapitel 4 im semantischen Bereich berücksichtigt werden.

Autonomie und strukturierte Interaktion. Durch die Abarbeitung von *Protokollen* ist die *Autonomie* der beteiligten Agenten mit Absicht partiell eingeschränkt. Im semantischen Bereich wird diese kontrollierte Einschränkung dadurch wirksam, dass Agenten auf den Eingang von Nachrichten warten. Die linken Seiten der Graphtransformationsregeln zu gebundenen Protokollen enthalten Nachrichten. Eine solche Regel kann erst dann angewendet werden, wenn eine in der Regel vorkommende Rolle des betrachteten Agenten die Nachricht angehängt bekommen hat. Die so bewirkte Synchronisation von Agenten schränkt die nebenläufige Ausführung von Agenten ein.

Autonomie und Proaktivität. Wie bereits in Kapitel 4 herausgestellt, ist die *Autonomie* eine notwendige Bedingung für *Proaktivität*. Im semantischen Bereich zeigt sich diese Beziehung dadurch, dass Strategien aus Operationen eines Agenten bestehen, die dieser autonom ausführt. Das zur Strategie gehörige Ziel wird erreicht, falls die Strategie eine Gewinnstrategie ist.

Proaktivität und strukturierte Interaktion. Die Interaktion zwischen Agenten ist notwendig, um Ziele zu erreichen, die die einzelnen Agenten für sich alleine nicht erreichen können. Die Zielregel eines Protokolls enthält deshalb Rollen, die unterschiedlichen

Agenten zugeordnet werden. Das zugehörige Protokoll bildet eine Kommunikationsstrategie zum Erreichen des Ziels. Der Austausch von Nachrichten und zugehörigen Parametern soll dafür sorgen, dass alle beteiligten Agenten ihren Zustand so verändern, dass sie sich dem spezifizierten Zielzustand nähern. Ein Protokoll zeigt dieses Verhalten, falls es eine Gewinnstrategie bezüglich der Zielregel darstellt.

6.4.6 Zusammenfassung

Die Sprachelemente zu Konstituenten agentenbasierter Systeme im semantischen Bereich der Graphtransformation sind in Abbildung 6.7 dargestellt. Auf der Metaebene befindet sich der Metatypgraph TG_{AOSE} der Sprache AML. Durch ihn wird die Struktur der Sprache im agentenorientierten semantischen Bereich beschrieben.

Die Modellebene enthält das System-Graphtransformationssystem $GTS_{SYS} = \langle TG, RS, SG, C \rangle$, das die Struktur und das Verhalten des agentenbasierten Systems beschreibt. Die Struktur ist durch den Typgraphen TG beschrieben, der selbst durch den Typgraphen TG_{AOSE} getypt ist. Für GTS_{SYS} gelten folgende Bedingungen:

- Ein- und Ausgangsgrade (Kardinalitäten) von Knoten für Kanten eines bestimmten Typs sind als zusätzliche Bedingungen in C anzugeben.
- Jeder Operationsname in TG ist genau einem Agententyp zugeordnet.
- Zu jeder Operation im Typgraphen TG muss genau eine Graphtransformationsregel gleichen Namens in RS existieren, die diese Operation realisiert.
- Nur Attribute des Agenten, dem in TG eine Graphtransformationsregel aus RS zugeordnet ist, dürfen durch die Regel verändert werden.
- Nur die Datenobjekte dürfen durch eine Graphtransformationsregel erzeugt oder gelöscht werden, oder deren Attribute dürfen verändert werden, die vom die Regel besitzenden Agenten aus erreichbar sind.
- Graphtransformationsregeln zu Operationen mit Rollen müssen Nachrichten senden oder empfangen. Beim Senden wird einer oder mehreren Rollen eine Nachricht angehängt. Beim Empfangen wird lokal eine Nachricht gelöscht. Bei beiden Arten von Operationen wird ansonsten lediglich der lokale Zustand modifiziert.
- Die Bedingungen für die *role-of* Kante ergeben sich wie folgt: Die schwache Identität von Rollen impliziert, dass jede Rolleninstanz mit genau einer Basisagenteninstanz durch eine *role-of* Kante verbunden ist. Mehrere Rolleninstanzen können mit derselben Instanz eines Basisagenten verbunden sein.
- Die Lebensdauer einer Rolle hängt von der des Agenten ab, zu dem sie gehört.
- Während der Lebenszeit eines Agenten kann er Rollen annehmen und ablegen.

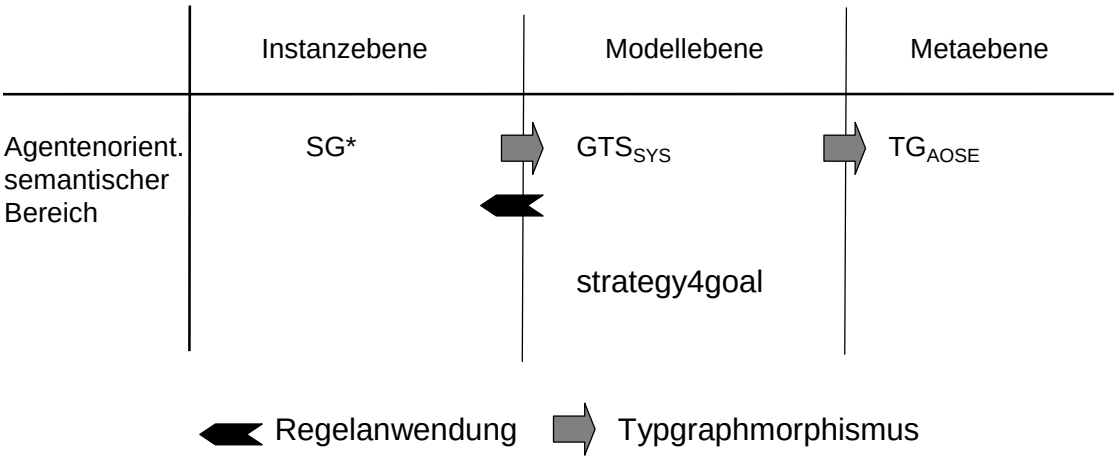

Figure 6.7: Bestandteile des agentenorientierten semantischen Bereichs

Die Beziehung zwischen Zielen und Strategien wird durch eine Abbildung repräsentiert. Die Abbildung *strategy4goal* ordnet Zielen Strategien zu. Alle Typgraphen und Regeln von Strategie-Graphtransformationssystemen der Modellebene sind dabei selbst durch den Typgraphen TG_{AOSE} getypt.

Auf der Instanzebene beschreiben die aus den Startgraphen in SG mit den Regeln des System-Graphtransformationssystems GTS_{SYS} abgeleiteten Graphen verschiedene Zustände. Die resultierende Menge aus abgeleiteten Graphen ist mit SG^* bezeichnet.

Nach Etablierung der Konstituenten im semantischen Bereich besteht der nächste Schritt darin, passende syntaktische Repräsentanten einzuführen.

6.5 Die Modellierungssprache AML

Zunächst wird das UML-Profil zu AML definiert und der Bezug zum semantischen Bereich und sonstigen Anforderungen geklärt. Darauf aufbauend werden zum Profil konforme Diagrammarten eingeführt. Die Diskussion anderer Ansätze für UML-basierte Sprachen zeigt die Adäquatheit der in dieser Arbeit entwickelten Spracharchitektur. Abschließend wird gezeigt, dass AML-Modelle mit dem in Kapitel 2 entwickelten Automatenmodell verträglich sind.

6.5.1 Das UML-Profil zu AML

Die Syntax von AML basiert auf einem Profil der Modellierungssprache UML. Wie bereits am Anfang des Kapitels erläutert, besteht ein Profil aus einer Menge von Stereotypen für UML-Modellelemente. Das Profil leitet sich aus den Elementen des Metatypgraphen TG_{AOSE} ab (siehe Abb. 6.6). Für jeden Knoten im Metatypgraphen wird ein Stereotyp im Profil eingefügt. Jedes Stereotyp steht in Bezug zu einer Basismetaklasse aus

dem UML-Metamodell. Eine Beschreibung erklärt den Zweck des Stereotyps und Bedingungen schränken seine Anwendung ein. Die Bedingungen des Stereotyps übertragen Informationen aus dem semantischen Bereich in die Syntax der Sprache AML.

Für jedes Stereotyp wird der Zusammenhang mit den Anforderungen und die Semantikabbildung auf den Metatypgraphen von AML diskutiert:

Stereotyp	⟨⟨ Agent ⟩⟩
Basisklasse	Class
Oberklasse	N/A
Beschreibung	Klassen mit Stereotyp ⟨⟨ Agent ⟩⟩ repräsentieren Agentenklassen.
Bedingungen	Die Klasse ist aktiv, d.h. sie besitzt einen eigenen Kontrollfluss: `self.isActive = true` Agenten bestehen nur aus Attributen und Operationen. Nachrichten gehören nicht dazu, da Agenten lediglich mit Hilfe von Protokollen strukturiert kommunizieren: `self.feature->forAll(oclIsTypeOf(Operation) or oclIsTypeOf(Attribute))` Durch `Operation.specification` ist der Name einer Graphtransformationsregel gegeben.
Abbildung	Abbildung von ⟨⟨ Agent ⟩⟩ auf Agent in TG_{AOSE}. Attribute werden auf agent-string abgebildet. Operationen werden auf Operation abgebildet und durch die Kante opA angebunden.

Stereotyp	⟨⟨ Data ⟩⟩
Basisklasse	Class
Oberklasse	N/A
Beschreibung	Daten werden außer in Attributen auch in Datenobjekten, d.h. Instanzen von Klassen mit Stereotyp ⟨⟨ Data ⟩⟩ gehalten.
Bedingungen	Die Klasse enthält keine Methoden, sondern nurAttribute: `self.feature->forAll(oclIsTypeOf(Attribute))`
Abbildung	Abbildung von ⟨⟨ Data ⟩⟩ auf Dataobject in TG_{AOSE}. Attribute werden auf data-string abgebildet.

Stereotyp	⟨⟨ Role ⟩⟩
Basisklasse	Class
Oberklasse	N/A
Beschreibung	Die Interaktion stützt sich auf Rollen. Klassen mit Stereotyp ⟨⟨ Role ⟩⟩ enthalten Attribute, Operationen und Nachrichten.
Bedingungen	Rollen enthalten Attribute, Operationen und Nachrichten: `self.feature->forAll(oclIsTypeOf(Operation) or oclIsTypeOf(Attribute) or oclIsTypeOf(Reception))` Mindestens eine Operation und eine Nachricht existieren, damit die Rolle an einer Interaktion teilnehmen kann: `self.feature->exists(oclIsTypeOf(Operation) and and self.feature->exists(oclIsTypeOf(Attribute) and self.feature->exists(oclIsTypeOf(Reception)`
Abbildung	Abbildung der ⟨⟨ Role ⟩⟩-Klasse auf Role in TG_{AOSE}. Attribute werden auf role-string abgebildet. Operationen werden auf Operation abgebildet und durch die Kante opR angebunden. Nachrichten werden auf Reception abgebildet und durch recipient angebunden.

Stereotyp	⟨⟨ role-of ⟩⟩
Basisklasse	Association
Oberklasse	N/A
Beschreibung	Eine Rolle wird durch eine Assoziation vom Stereotyp ⟨⟨ role-of ⟩⟩ an einen Agenten gebunden.
Bedingungen	Eine ⟨⟨ role-of ⟩⟩-Assoziation hat die gleichen Einschränkungen wie die Kompositionsbeziehung in UML: `let self.aggregation = composite` Beziehungen verlaufen immer zwischen zwei Klassen mit Stereotyp ⟨⟨ Role ⟩⟩ und ⟨⟨ Agent ⟩⟩: `self.associationEnd->one(c | c.stereotype = Role) and self.associationEnd->one(c | c.stereotype = Agent)`
Abbildung	Abbildung der ⟨⟨ role-of ⟩⟩-Assoziation auf den Kantentyp role-of in TG_{AOSE}.

Stereotyp	⟨⟨ Message ⟩⟩
Basisklasse	Reception
Oberklasse	N/A
Beschreibung	Eine Nachricht mit Parametern wird zwischen Rollen ausgetauscht.
Bedingungen	Parameter sind von der Klasse Data: `self.parameter->forAll(oclTypeOf(Data))`
Abbildung	Abbildung von ⟨⟨ Message ⟩⟩ auf den Knotentyp Message in TG_{AOSE}. Eine Nachricht kann Daten als Parameter besitzen, was durch den Kantentyp msgparam zum Knotentyp Data ausgedrückt wird.

Stereotyp	⟨⟨ Protocol ⟩⟩
Basisklasse	Package
Oberklasse	N/A
Beschreibung	Ein Protokoll integriert strukturell Rollen und Datenobjekte, die als Parameter von Nachrichten zwischen Rollen ausgetauscht werden. Das Paket wird als Template eingesetzt, indem der Template-Mechanismus von UML verwendet wird.
Bedingungen	Ein Protokoll enthält wenigstens zwei Role-Klassen.
Abbildung	-

Stereotyp	⟨⟨ LHS ⟩⟩
Basisklasse	Package
Oberklasse	N/A
Beschreibung	Das Paket enthält die linke Regelseite einer Regel. In den Teilpaketen befinden sich Objektdiagramme, bestehend aus Instanzen des Agentenklassendiagramms zum Typgraph des betrachteten Graphtransformationssystems.
Bedingungen	Die linke Regelseite darf nicht leer sein.
Abbildung	-

Stereotyp	⟨⟨ RHS ⟩⟩
Basisklasse	Package
Oberklasse	N/A
Beschreibung	Das Paket enthält die rechte Regelseite einer Regel.
Bedingungen	-
Abbildung	-

Stereotyp	⟨⟨ Rule ⟩⟩
Basisklasse	Class
Oberklasse	N/A
Beschreibung	Graphtransformationsregeln bestehen aus einer linken und einer rechten Regelseite. Das Paket enthält zwei mit den Stereotypen ⟨⟨ LHS ⟩⟩ (left hand side) und ⟨⟨ RHS ⟩⟩ (right hand side) markierte Teilpakte für die linke und die rechte Regelseite. Die gemeinsamen Elemente der linken und rechten Regelseite werden nur im Teilpaket der linken Regelseite definiert. Im Teilpaket der rechten Regelseite werden für gemeinsame Elemente die Namen der linken Regelseite verwendet.
Bedingungen	Identische Elemente der linken und rechten Seite werden nur einmal auf der linken Seite definiert und dann auf der anderen Seite referenziert. Nachrichtenaustausch erfolgt nur zwischen Rollen.
Abbildung	-

Die soeben definierten Sterotype des UML-Profils AML dienen in verschiedenen Diagrammarten der Modellierung agentenbasierter Systeme.

6.5.2 Diagramme

Aus Modellelementen des UML-Profils von AML werden Diagramme für die Modellierung agentenbasierter Systeme gebildet. Das Agentenklassendiagramm definiert ein strukturelles Modell des Systems. Dessen Dynamik ist durch ein Paket von Regeldiagrammen gegeben, die der Notation von Graphtransformationsregeln dienen. Ein Protokoll ist ein parametrisiertes Paket, das aus einem Klassendiagramm für die Struktur und einem Paket mit Regeldiagrammen für das Verhalten des Protokolls besteht. Protokolle werden syntaktisch an die Agenten gebunden, die das Protokoll verwenden. Ein Ziel und die zugehörige Strategie sind durch Regeldiagramme beschrieben, die in einem Paket gruppiert werden.

Beispiele verdeutlichen den Einsatz von Diagrammen. Diese Vorgehensweise entspricht derjenigen in der UML-Spezifikation.

Agentenklassendiagramm. Modelle werden im semantischen Bereich durch ein TAGTS angegeben, das aus einem Typgraphen und einer Menge von Graphtransformationsregeln besteht. Der Typgraph wird durch ein *Agentenklassendiagramm* notiert, in dem lediglich zum UML-Profil von AML konforme Klassen vorkommen. Es dürfen alle Stereotype außer ⟨⟨ Protocol ⟩⟩ und ⟨⟨ Rule ⟩⟩ verwendet werden. Durch Verwendung der Stereotype des UML-Profils von AML werden die Elemente des Agentenklassendiagramms syntaktisch korrekt typisiert. Dabei sind insbesondere alle zum Profil gehörigen Randbedingungen einzuhalten. Dadurch wird erreicht, dass ein syntaktisch korrektes

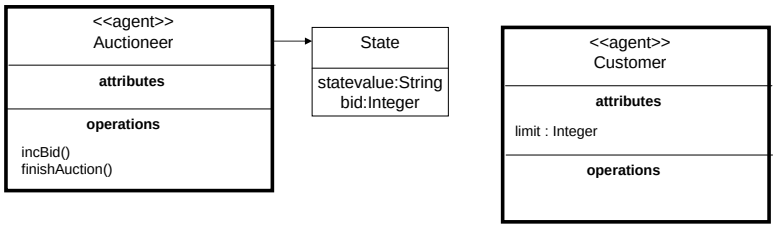

Figure 6.8: Ein Agentenklassendiagramm

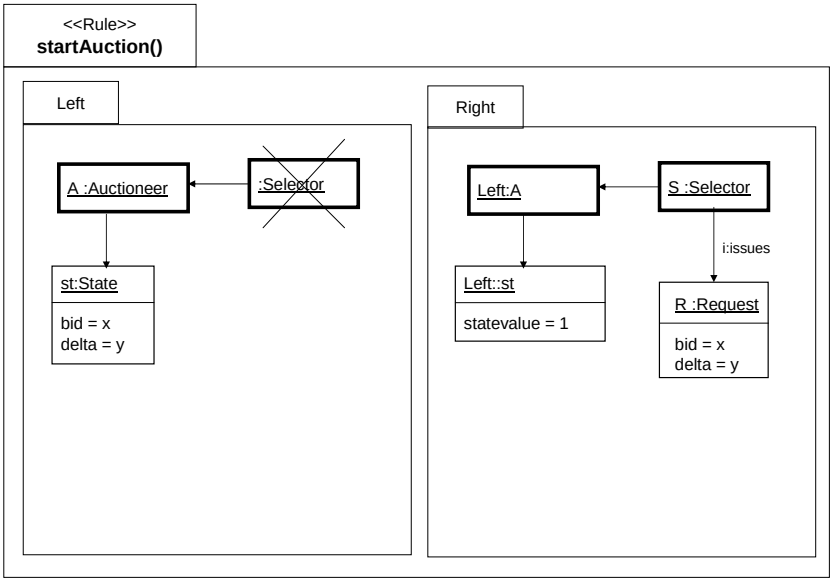

Figure 6.9: Ein Regeldiagramm

Agentenklassendiagramm einen Typgraphen notiert, der durch den AML-Metatypgraph getypt ist. Ein Beispiel zeigt Abbildung 6.8.

Regeldiagramm. Eine Graphtransformationsregel wird durch ein stereotypisiertes Paketdiagramm notiert. Zwei Pakete mit Namen Left und Right werden für die linke und rechte Seite einer Graphtransformationsregel in das Regelpaket eingefügt. Die Pakete für die linke und rechte Seite enthalten Agentenklassendiagramme mit Elementen aus dem Profil AML. Ein Beispiel zeigt Abbildung 6.9.

Diagramme für Protokolle und Ziele. Protokolle werden auf Basis des UML-Profils von AML notiert. Der Template-Mechanismus von UML wird genutzt, um die Bindung von Protokollen zu beschreiben. Das Binden eines Protokolls entspricht syntaktisch der Verwendung eines Templates. Der statische Anteil eines Protokolls wird durch ein

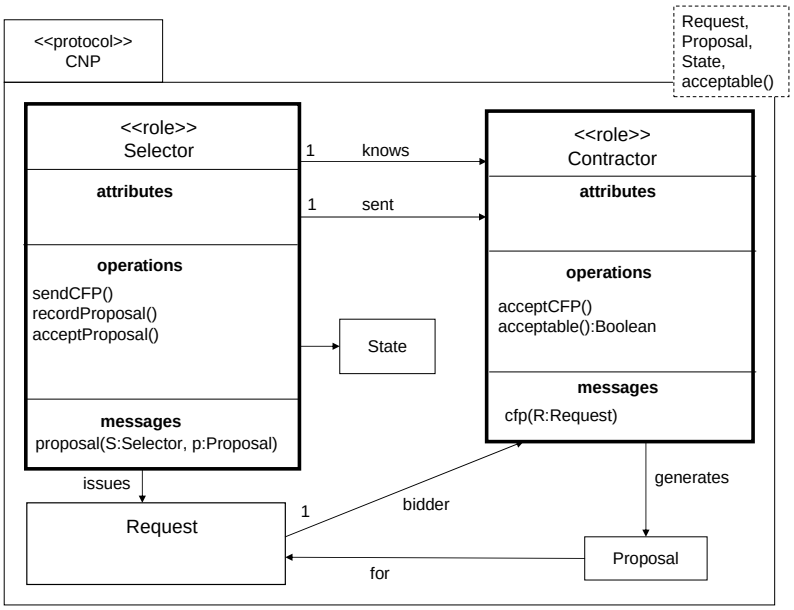

Figure 6.10: Das Protokolldiagramm des *contract net*-Protokolls

Klassendiagramm in einem Template-Paket beschrieben. Der dynamische Anteil eines Protokolls besteht aus den Regeldiagrammen des Protokolls.

Protokolldiagramme definieren Kommunikationsstrategien. Zugehörige Zielregeln abstrahieren von der Interaktion und beschreiben den Zusammenhang zwischen Anfangs- und Endzustand eines agentenbasierten Systems. Durch die Bindung eines Protokolls werden aus Zielen und Strategien für Rollen solche der Agenten, die das Protokoll verwenden. Zu den Diagrammen für Struktur und Verhalten kommen Regeldiagramme für Ziele, um ein Protokoll vollständig zu definieren

Das Binden von Protokollen Protokolle werden im Verlauf der Entwicklung eines agentenbasierten Systems gebunden (vgl. Kapitel 8). Anhand eines Beispiels soll hier gezeigt werden, wie ein gegebenes Protokoll syntaktisch an eine Menge von Agenten, die das Protokoll nutzen sollen, gebunden wird. Als Beispiel dient die Anwendung *Englische Auktion* [50]. Dabei kommuniziert ein Agent, der Auktionator, mit mehreren anderen Agenten, den Kunden. Das Ziel besteht darin, den am höchsten bietenden Kunden zu bestimmen.

Der Auktionator versucht, das Ergebnis zu maximieren, indem er wiederholt die Akzeptanz eines schrittweise erhöhten Gebots abfragt. Dieses wiederholte Frage-Antwort-Muster kann durch ein Interaktionsprotokoll, das *contract net*-Protokoll, im Rahmen der englischen Auktion realisiert werden. Im folgenden werden die Sprachelemente von AML anhand des *contract net*-Protokolls demonstriert.

Das *Contract Net*-Protokoll (CN-Protokoll) ist ein typisches Interaktionsmuster für die multilaterale Kommunikation in agentenbasierten Systemen [95]. In dieses Protokoll

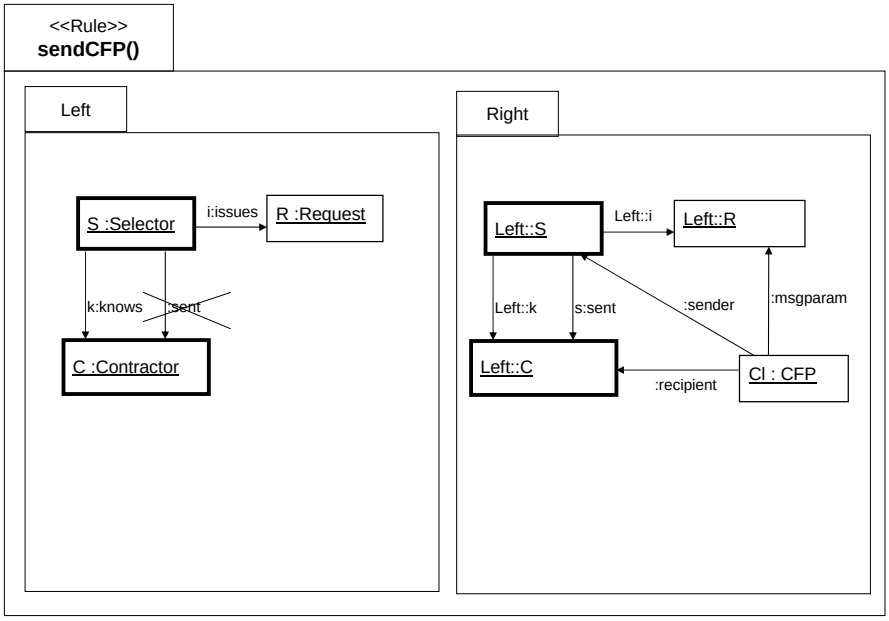

Figure 6.11: Die Regel sendCFP()

sind zwei Arten von Rollen involviert, der selector-Rollentyp und der contractor-Rollentyp. Eine selector-Rolle fordert Vorschläge von am Protokoll beteiligten contractor-Rollen an, indem sie eine entsprechende Anfragenachricht (call for proposal) verschickt. Die contractor-Rollen, die die Anfrage erhalten, können durch Senden eines Vorschlags antworten. Die selector-Rolle wählt unter den erhaltenen Antworten der contractor-Rollen die zuerst erhaltene aus und schickt der zugehörigen contractor-Rolle eine Akzeptanznachricht.

Das Graphtransformationssystem $\mathcal{G}$ zu der Strategie ist durch das Klassendiagrammin Abbildung 6.10 gegeben, wodurch der Typgraph TG notiert wird. Die Randbedingungen in C sind durch die Kardinalitätseinschränkungen des Klassendiagramms gegeben. Die Regeln der Menge R des *Contract Net*-Protokolls sind in den Abbildungen 6.11, 6.12, 6.13 und 6.14 dargestellt. Die globalen Vor- und Nachbedingungen des Protokolls sind in der Zielregel in Abbildung 6.15 spezifiziert.

Die Verwendung eines Protokolls für einen spezifischen Zweck ist eine Entscheidung, die während des Softwareentwurfs getroffen wird. Ein Protokoll wird dadurch verwendet, dass es an ein gegebenes Modell gebunden wird. Das Binden ist ein rein syntaktischer Mechanismus der Makroexpansion: Die Parameter eines Protokoll-Templates werden mit den Modellelementen eines gegebenen Klassendiagramms belegt (vgl. Fig. 6.8). Danach werden die Modellelemente des Protokolls in das Klassendiagramm eingefügt, und die Rollenklassen werden mit Agentenklassen durch eine *role-of* Beziehung verbunden.

Die Graphtransformationsregeln werden in der folgenden Weise expandiert: jede Rolle in einer Regel wird durch die *role-of* Beziehung mit einem Agenten verbunden. Die Beziehung von Rolle und Agent muss konsistent zu dem expandierten Klassendiagramm

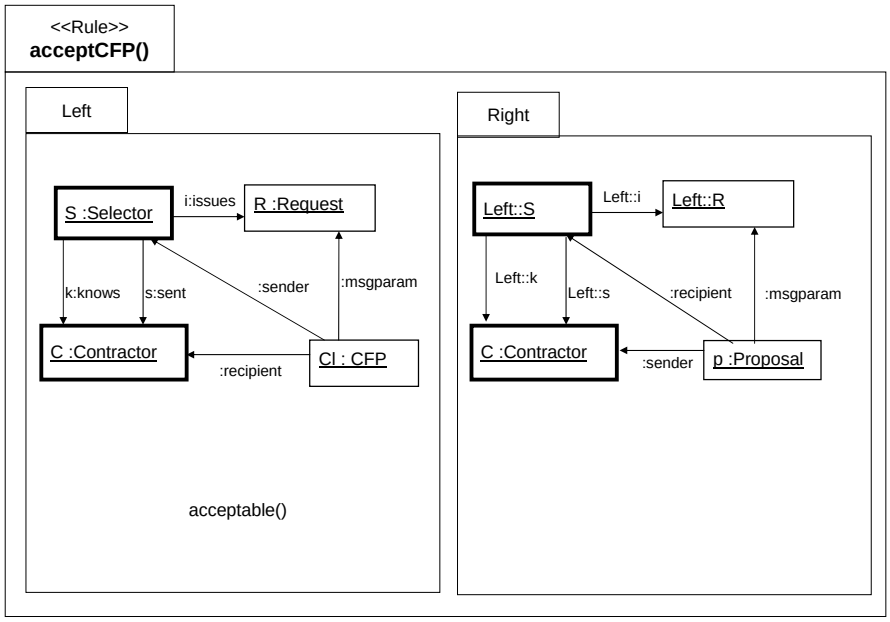

Figure 6.12: Die Regel acceptCFP()

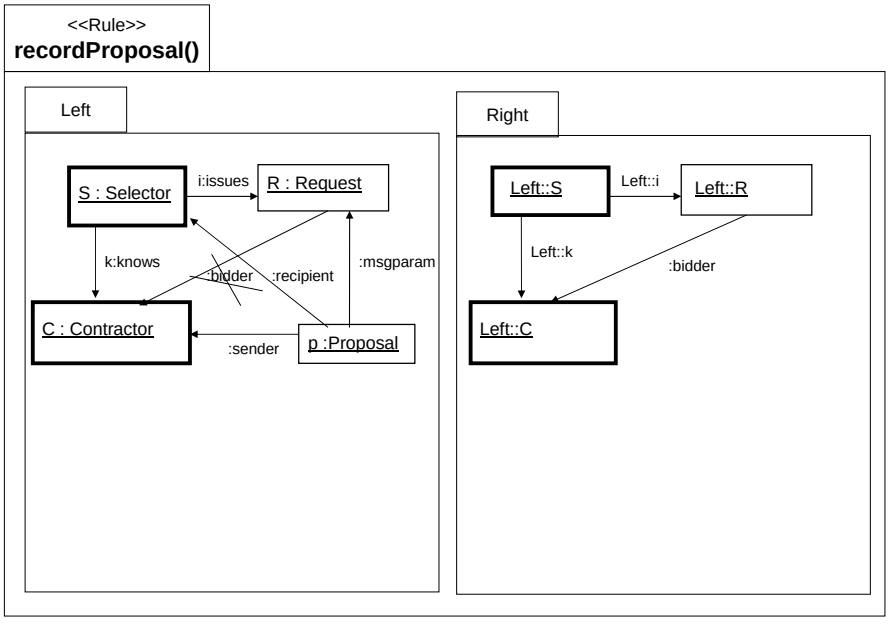

Figure 6.13: Die Regel recordProposal()

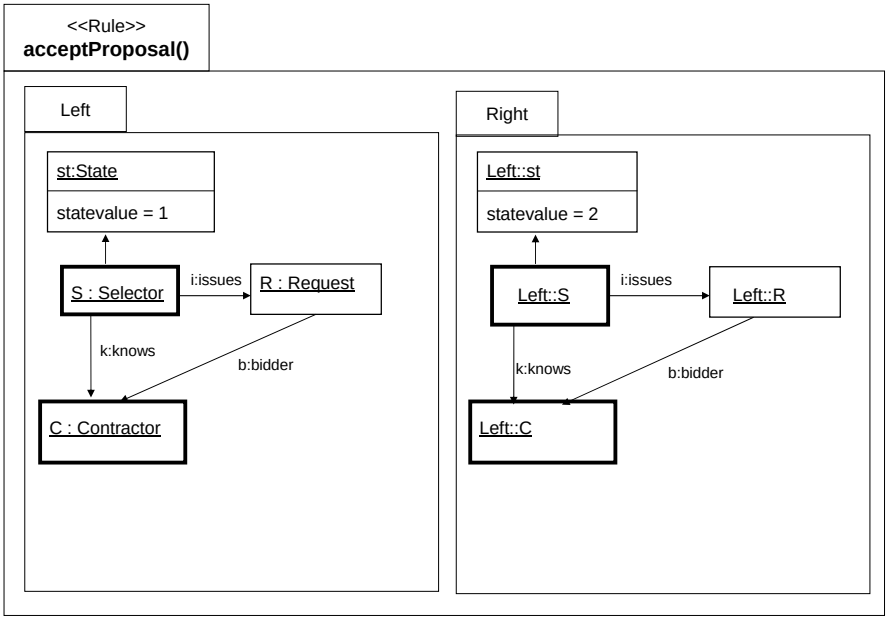

Figure 6.14: Die Regel acceptProposal()

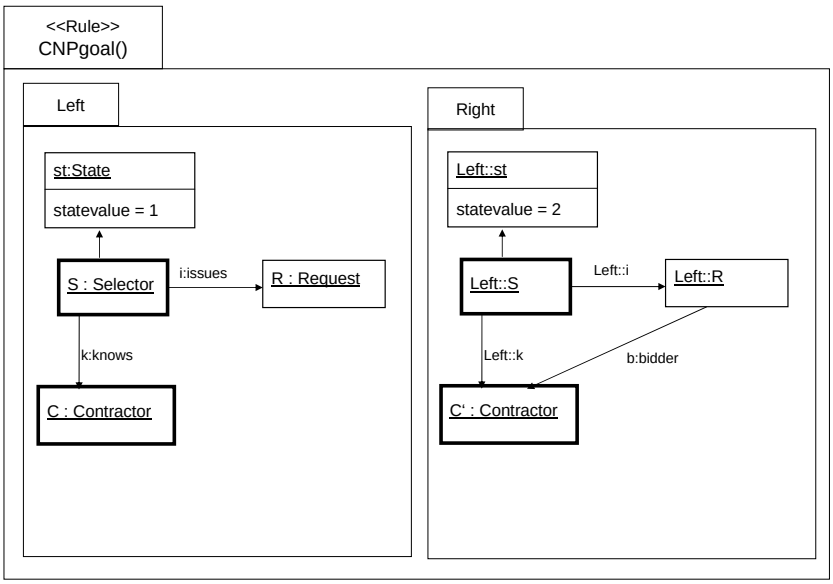

Figure 6.15: Die Zielregel des CN-Protokolls

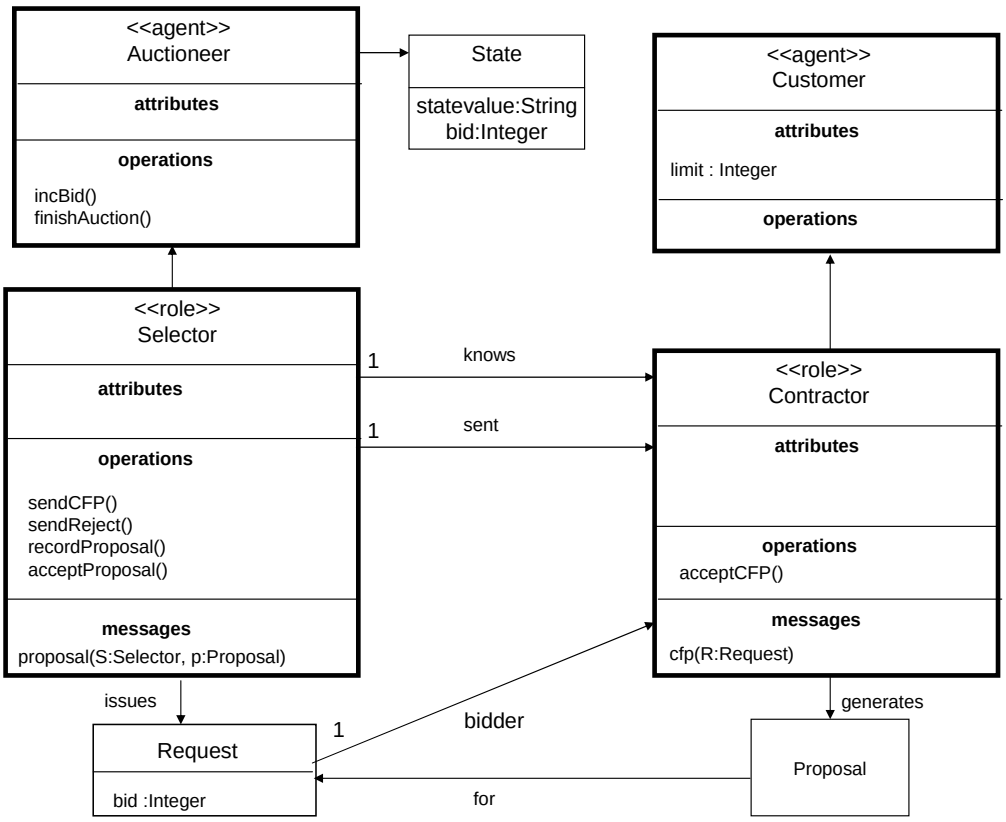

Figure 6.16: Integriertes Agentenklassendiagramm

sein. Protokollparameter in jeder Regel werden mit den Modellelementen belegt, die bereits im Protokoll-Paketdiagramm fixiert worden sind. Die expandierten Graphtransformationsregeln beschreiben jetzt das Protokollverhalten der Agenten, an die die Rollen gebunden worden sind.

Im Beispiel wird das *contract net*-Protokoll im Rahmen einer englischen Auktion instanziiert. Dazu wird die selector-Rolle an die Agentenklasse auctioneer und die contractor-Rolle an die Agentenklasse customer gebunden. Die Parameter des Protokoll-Paketdiagramms werden mit den Datenobjekten request und proposal instanziiert. Als Ergebnis entsteht das Agentenklassendiagramm in Abbildung 6.16. Das Binden der Protokollregeln erfolgt analog. Für die Regel acceptCFP() ist das Resultat in Abbildung 6.17 zu sehen.

Das Binden eines Protokolls im syntaktischen Bereich durch Verwendung eines Templates korrespondiert im semantischen Bereich mit dem Binden von Protokollrollen an Agenten durch role-of-Kanten. Bezogen auf die Protokollbindung ist Graphtransformation ein geeigneter semantischer Bereich, da die notwendigen Änderungen des Systemverhaltens allein durch strukturelle Änderungen von Graphtransformationsregeln erreicht werden. Die Notwendigkeit zu einer Änderung der operationalen Semantik von TAGTS,

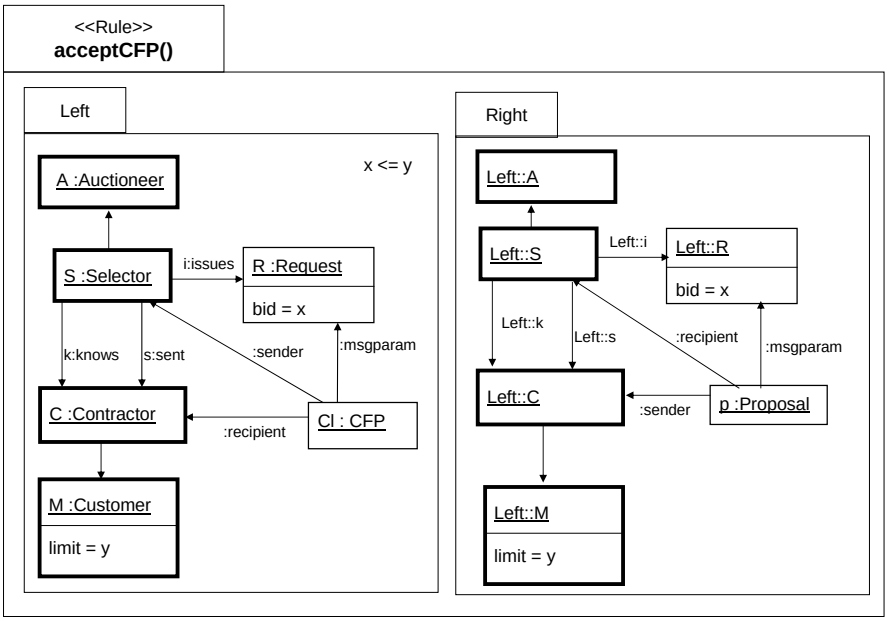

Figure 6.17: Die gebundene Regel acceptCFP()

wie von Depke et al. in [37] vorgestellt, besteht nicht.

6.5.3 Andere Ansätze für UML-basierte Spracharchitekturen

Verschiedene andere Ansätze für eine UML-basierte Spracharchitektur existieren, mit denen neben der Syntax auch die Semantik definiert wird. Wichtige Ansätze werden nachfolgend eingeordnet.

MOF. Der verbreitete Standard für die Definition von Modellierungssprachen ist die Sprache Meta Object Facility (MOF) [85]. MOF führt 4 Sprachebenen ein. Die abstrakte Syntax auf jeder Ebene wird durch Klassendiagramme auf der darüberliegenden Ebene definiert. Damit dienen Modelle selbst der Definition von Modellierungssprachen, wodurch die Vokabel Metamodellierung erklärt ist. Auf jeder Ebene von MOF können konkrete Syntaxrepräsentationen für die definierten Sprachelemente eingeführt werden. Für UML als bekannteste MOF-basierte Sprache sind zahlreiche unterschiedliche Diagramme definiert. Nachteilig an MOF ist das Fehlen der Unterstützung, semantische Bereiche anzugeben und eine Semantik für MOF-Sprachelemente zu definieren. Weiterhin ist die Instanzrelation zwischen verschiedenen Ebenen nicht präzise definiert (vgl. pUML [48]). Gerade dieses Problem wird durch die vorgestellte Spracharchitektur vermieden: es werden sowohl ein semantischer Bereich, nämlich Graphtransformation, als auch präzise Typ-Instanz-Abhängigkeiten durch die Typgraph-Beziehung angegeben.

Meta-Modeling Framework. Die Definition der Modellierungssprache UML 1.5 [86] ist nur partiell präzise und formal, wodurch sich zahlreiche Probleme wie Mehrdeutigkeit, Inkonsistenz, Unvollständigkeit, etc. ergeben (vgl. Kapitel 5). Als ein systematisches Verfahren haben Clark et al. [12, 17] die Spracharchitektur MMF (Meta Modeling Facility) und darauf aufbauend die Sprache MML (Meta Modeling Language) vorgeschlagen. Dabei wird MOF rekursiv um eine Folge von Verfeinerungsebenen auf der Basis von Paketen erweitert, bei der auf jeder Ebene die Struktur der vorhergehenden Ebene und ein Instanziierungsmechanismus definiert werden. Dieser Mechanismus legt auch die (denotationelle) Semantik der Ebene fest, durch die die Konstrukte der vorhergehenden Ebene erklärt werden. Das Verhalten der Konstrukte wird allerdings nicht berücksichtigt, so dass dieser Ansatz hier ungeeignet ist.

Metamodeling mathematics. Varro et al. schlagen eine mengenbasierte Semantik für UML-Modelle vor [100]. Die Dynamik von Modellen wird durch Graphtransformatiossysteme beschrieben, die durch UML-Diagramme notiert werden.. Neu ist die mathematische Fundierung der Konzepte Instanziierung und Verfeinerung sowohl für für statische als auch für dynamische Modelle.

Dieser Ansatz ähnelt dem in dieser Arbeit vorgestellten Vorschlag. Ein wesentlicher Unterschied besteht darin, dass mengen- und ordungstheoretische Begriffe verwendet werden. Die hier vorgestellte Spracharchitektur verwendet demgegenüber Begriffe der Theorie der Graphtransformation, wie die Typisierung durch Typgraphen und Attributierung durch Algebren. Der Vorschlag von Varro stellt eine denkbare Alternative zur vorgestellten Spracharchitektur dar. Gründe, aus denen dieser Ansatz vorzuziehen wäre, sind allerdings nicht erkennbar.

DMM. Graphtransformation dient weiterhin als Mittel, um eine graphische operationale Semantik für UML zu definieren: einerseits für einen Interpreter [46] und andererseits für einen Compiler [59] von UML-Diagrammen. Für das in dieser Arbeit behandelte Problem der Sprachdefinition ist der Ansatz noch nicht verwendbar, da der Regelsatz zur Beschreibung der Ausführung von UML-Diagrammen sich bisher auf Statecharts beschränkt.

6.6 Verträglichkeit mit dem abstrakten Agentenmodell

Um zu zeigen, dass gemäß AML definierte Modelle agentenbasierter Systeme mit dem abstrakten Systemmodell aus dem Kapitel 2 verträglich sind, wird das Systemmodell so auf ein Produkt von I/O-Automaten abgebildet, dass das Verhalten des Produktautomaten dem des Systemmodells gleicht.

Ein AML-Modell wird im semantischen Bereich durch ein Graphtransformationssystem $\mathcal{G} =< TG, RS, SG, C >$ mit einem Typgraphen TG, einer Regelmenge RS, einer Menge SG von Anfangsgraphen und einer Menge von Bedingungen C definiert. Dabei ist TG durch den Metatypgraphen TG_{AOSE} getypt. Der Typgraph TG und die Regelmenge

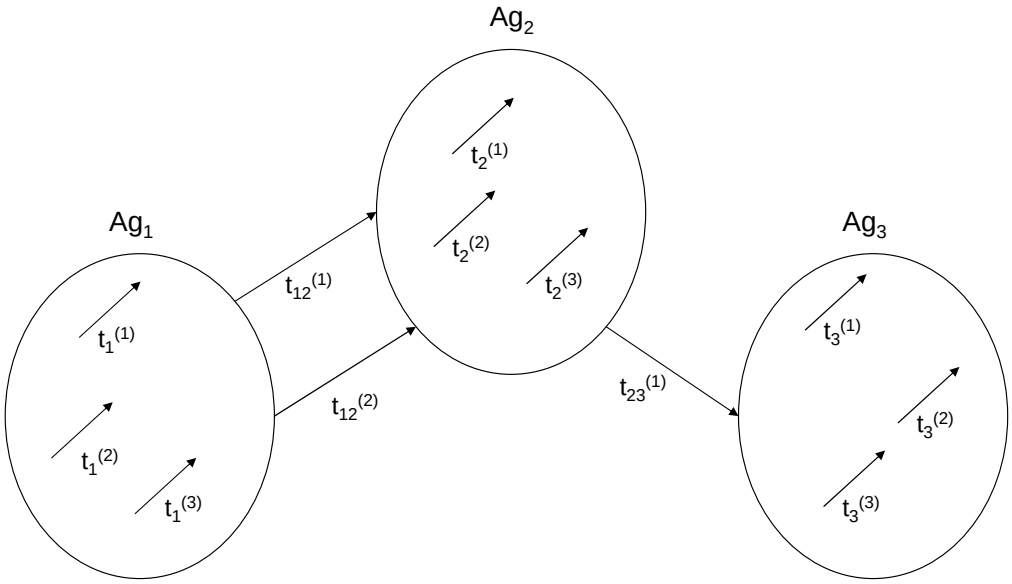

Figure 6.18: Partitionierung der Transitionen eines Transitionssystems

RS können aus einem UML-Modell abgeleitet sein, das dem AML-Profil entspricht (vgl. Abschnitt 6.5.1).

Ein aus $\mathcal{G}$ abgeleitetes Systemmodell ist durch ein Transitionssystem $T_{\mathcal{G}} = \langle Q, \longrightarrow, Q_0 \rangle$ gegeben. Die Regeln eines Agenten führen zu zwei Arten von Transitionen auf Zustandsgraphen: entweder werden lokale Daten verändert oder ein Agent sendet eine Nachricht. Die Transitionen in $\longrightarrow$ lassen sich in die lokalen Transitionen $\longrightarrow_{Ag_i} \subseteq \longrightarrow$ einzelner Agenten Ag_i mit $\longrightarrow_{Ag_i} = \{t_i^1, ..., t_i^{(n_i)}\}$ und die Transitionen $\longrightarrow_{Ag_i,Ag_j} \subseteq \longrightarrow$ $(i \neq j)$ mit $\longrightarrow_{Ag_i,Ag_j} = \{t_i^1, ..., t_i^{(n_i)}\}$ einordnen, für die zwei Agenten Ag_i und Ag_j Zustandsänderungen dadurch erfahren, dass zwischen ihnen eine Nachricht ausgetauscht wird (vgl. Abb. 6.18). Der Austausch einer Nachricht geschieht dabei in zwei Schritten. Zuerst erfolgt eine Transition, durch die dem empfangenden Agenten eine Nachricht angehängt wird. Entweder unmittelbar oder nach einer Folge anderer Transitionen erfolgt eine Transition, die die Nachricht löscht und den internen Zustand des empfangenden Agenten ändern kann.

Definition der I/O-Automaten zu Agenten. Der nächste Schritt besteht darin, zu einem Agenten Ag_i einen I/O-Automaten M_i so zu definieren, dass das Automatenprodukt das Verhalten des Agentensystems repräsentiert. $states(M_i)$, die Menge der Zustände eines Agenten Ag_i, ist zum einen durch die Menge Q der Zustände gegeben, die Quelle oder Ziel von Transitionen in $\longrightarrow_{Ag_i}$ sind. Zum anderen gehören die Quellen von Transitionen aus $\longrightarrow_{Ag_i,Ag_j}$ zu $states(M_i)$. Die Menge der Anfangszustände des I/O-Automaten M_i zum Agenten Ag_i ist durch $start(M_i) = SG \cap states(M_i)$ gegeben.

Die Signatur Σ_i von M_i setzt sich aus den Mengen $in(\Sigma_i)$, $int(\Sigma_i)$ und $out(\Sigma_i)$ zusammen (vgl. Kapitel 2). Ihre Vereinigung ergibt $acts(\Sigma_i)$. Die Eingabeaktionen aus $in(\Sigma_i)$ bestehen aus den Transitionen, die aus Graphtransformationsregeln hervorgehen, die eine Nachricht enthalten, die der Agent von anderen Agenten empfangen kann, d.h. $in(\Sigma_i) = \bigcup_{j \in N} \longrightarrow_{Ag_j, Ag_i}$.

Die Ausgabeaktionen aus $out(\Sigma_i)$ bestehen aus den Transitionen, die aus Graphtransformationsregeln hervorgehen, die eine Nachricht enthalten, die der Agent an andere Agenten verschicken kann, d.h. $out(\Sigma_i) = \bigcup_{j \in N} \longrightarrow_{Ag_i, Ag_j}$.

Die Menge der internen Aktionen $int(\Sigma_i)$ besteht aus den Transitionen, die nur den inneren Zustand eines Agenten Ag_i ändern, d.h. $int(\Sigma_i) = \longrightarrow_{Ag_i}$.

Im Automatenmodell werden die lokalen Transitionen durch interne Aktionen angestoßen. Die Transitionen zum Senden und Empfangen einer Nachricht gehen von einem Zustand des sendenden Agenten aus und ändern den Zustand des empfangenden Agenten. Diese Transitionen resultieren aus Graphtransformationsregeln des sendenden Agenten, die Nachrichten an empfangende Agenten anhängen.

Die *Zustandsübergangsrelation* eines I/O-Automaten M_i ist durch $trans(M_i)$ mit $trans(M_i) \subseteq states(M_i) \times acts(\Sigma_i) \times states(M_i)$ gegeben.

Für eine Aktion $t \in int(\Sigma_i)$ überführt $trans(M_i)$ den Quellzustand von t in den Zielzustand von t. Falls $t \in in(\Sigma_i)$ ist, dann ist der Zielzustand gleich dem Quellzustand. Für $t \in out(\Sigma_i)$ ist der Zielzustand gleich dem Quellzustand. Diese beiden Übergänge resultieren daraus, dass entsprechende Transitionen aus Graphtransformationsregeln resultieren, die lediglich eine Nachricht senden und den Zustand sonst nicht ändern.

Der Produktautomat. Die Aktionen des Produktautomaten M ergeben sich entsprechend der Definition in Kapitel 2. Insbesondere bestehen die internen Aktionen von M damit auch aus den Aktionen, die zwischen Agentenautomaten M_i ausgetauscht werden.

Der Zustandsraum $states(M)$ des Automatenproduktes ist das kartesische Produkt der Zustandsräume $states(M_i)$ der einzelnen Agenten (vgl. Kapitel 2). Zwischen der Zustandsübergangsrelation $trans(M)$ und den einzelnen Zustandsübergangsrelationen $trans(M_i)$ besteht folgender Zusammenhang: $(s, t, s') \in trans(M)$, falls $(s_i, t, s'_i) \in trans(M_i)$ oder $s_i = s'_i$. Die Interpretation ist wie folgt: Ist $t = t_i^{(j)}$, dann erfolgt lediglich ein Übergang in M_i. Für $t = t_{i,j}^{(k)}$ erfolgen Übergänge in M_i und M_j. Das Resultat der Produktbildung zeigt damit genau das Verhalten des Systemmodells.

Die Konstruktion des Produktautomaten zeigt, dass das auf Graphtransformation basierende Agenten-Systemmodell konform zu dem in Kapitel entwickelten Automatenmodell ist und damit indirekt auch mit dem abstrakten Modell agentenbasierter Systeme verträglich ist.

6.7 Fazit

In diesem Kapitel sind die Spracharchitektur, die Syntax und die Semantik der Modellierungssprache AML zur Entwicklung agentenbasierter Systeme in Übereinstimmung mit den Anforderungen aus Kapitel 4 entwickelt worden. Die neuartige Spracharchitektur mit drei Abstraktionsebenen, bestehend aus Instanz-, Modell- und Metaebene und othogonal dazu mit den beiden Sprachebenen der Syntax und Semantik ermöglichte, die Sprache AML präzise zu definieren.

Für die Konzepte Proaktivität, Autonomie und strukturierte Interaktion wurden Konstituenten wie Ziel, Strategie, Ausführungsfaden, Kontrollelement und Protokoll eingeführt. Diese Konstituenten wurden durch spezifische Sprachkonstrukte in AML repräsentiert.

Der semantische Bereich der Graphtransformation hat sich als geeignet für die semantische Repräsentation von AML-Modellen erwiesen. Ausgehend von den Konstituenten agentenbasierter Systeme wurden Bedingungen an Modelle formuliert. Neu eingeführt wurde das Konzept des Metatypgraphen zur Typisierung von Modell-Typgraphen, wodurch der allgemeine semantische Bereich auf einen spezifisch agentenorientierten Bereich eingeschränkt wurde. Zum einen wurde ein Metatypgraph TG_{AOSE} entwickelt, der die Typgraphen von Modell-Graphtransformationssystemen einschränkt. Zum anderen wurde eine Menge von Randbedingungen formuliert, die aus den Eigenschaften der Konstituenten resultieren.

Die Syntax der Sprache AML wurde durch ein UML-Profil definiert. Für die Elemente des Metatypgraphen wurden Stereotype von UML-Modellelementen definiert. Die Randbedingungen des semantischen Bereichs finden sich in Bedingungen an die Modellelemente wieder.

Durch die Abbildung von Modell-Graphtransformationssystemen in eine Menge von I/O-Automaten wurde gezeigt, dass AML-Modelle mit den abstrakteren Modellen agentenbasierter Systeme aus Kapitel 2 verträglich sind.

Mit der Modellierungssprache AML ist es möglich, semantisch wohldefinierte Modelle agentenbasierter Systeme zu entwickeln. Die Vorgehensweise für eine solche Entwicklung wird im Kapitel 8 beschrieben.

Chapter 7

Modellüberprüfung

Modelle sollen gegebenen Anforderungen genügen. Eine wesentliche Aufgabe der modellbasierten Softwareentwicklung besteht neben der Modellerstellung darin, Modelle auf Erfüllung von Anforderungen zu überprüfen. Die Anforderungen sind in der Sprache AML durch Ziele gegeben, die von Agenten durch das Verfolgen von Strategien erreicht werden sollen. Ein zu überprüfendes Modell ist hier durch eine Strategie gegeben. Das generelle Problem besteht darin zu überprüfen, ob ein Ziel mit Hilfe einer Strategie erreicht werden kann, d.h. ob eine Gewinnstrategie vorliegt. In Kapitel 6 ist gezeigt worden, wie die Begriffe Ziel, Strategie und Gewinnstrategie mit Hilfe von Graphtransformationsregeln und Graphtransformationssystemen präzisiert werden können. Die Zusammenhänge sind in den ersten drei Zeilen der Tabelle in Abbildung 7.1 dargestellt.

Für die Überprüfung von Modellen stehen allgemeine formale Verfahren wie Model Checking zur Verfügung. Modelle sind dabei durch Kripkestrukturen gegeben, die aus Transitionssystemen abgeleitet werden können. Eine zu überprüfende Eigenschaft ist durch eine temporal-logische Formel gegeben. Model Checking ermöglicht zu überprüfen, ob ein gegebenes Modell eine solche Eigenschaft erfüllt.

Eigenschaften in Form von Zielen und Modelle in Form von Strategien sind entsprechend dem vorherigen Kapitel durch Graphtransformationsregeln und Graphtransformationssysteme gegeben. Der Ansatz dieses Kapitels besteht darin, die derart gegebenen Modellbestandteile und die zugehörigen Anforderungen in die Eingabesprache eines Model Checkers zu überführen.

In Kapitel 6 ist gezeigt worden, dass Graphtransformationssysteme ein Transitionssystem spezifizieren, dessen Zustände Graphen sind und dessen Transitionen Graphtransformationsschritte sind. Problematisch für die Untersuchung des durch das Transitionssystem gegebenen Zustandsraums ist dessen Größe, die abhängig von den betrachteten Variablen und deren Wertebereichen ist. Von Varro ist gezeigt worden, dass die Graphtransformationssysteme so auf Transitionssysteme abgebildet werden können, dass der entstehende Zustandsraum möglichst klein wird [99]. Dadurch ist es möglich, Strategie-Graphtransformationssysteme auf das Erreichen von Zielzuständen bezüglich eines gegebenen Ausgangszustands zu testen.

In diesem Kapitel werden dazu zwei Probleme gelöst. Zum einen wird gezeigt, wie

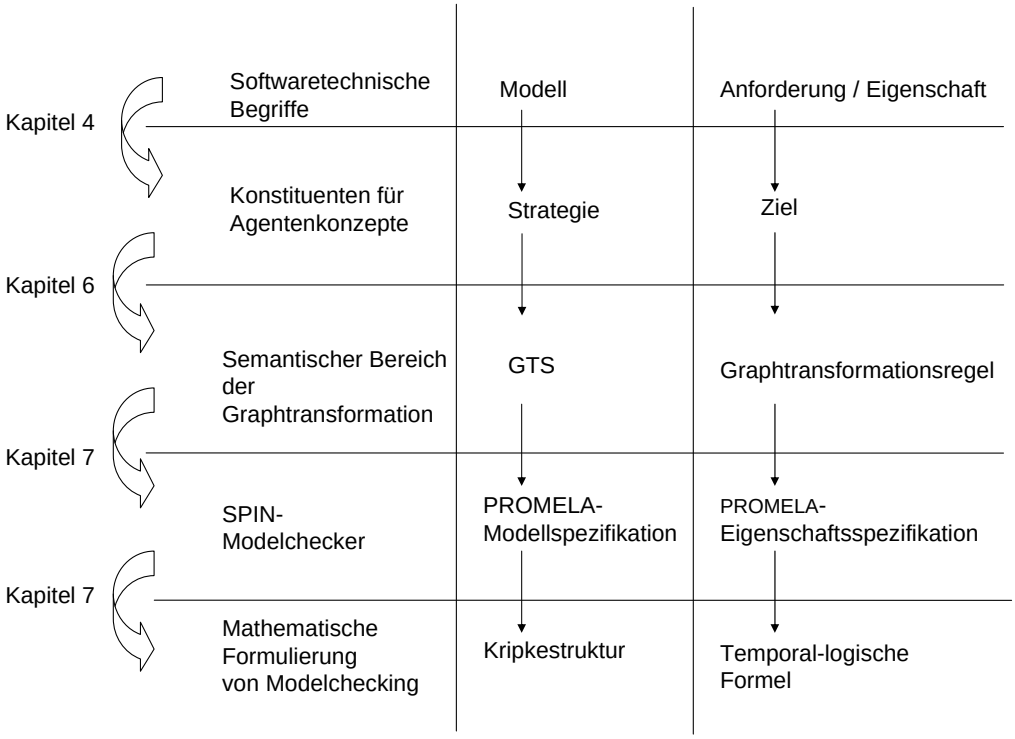

Figure 7.1: Übersicht

eine Strategie in die *Spezifikation* eines Transitionssystems, respektive einer Kripkestruktur abgebildet werden kann. Dabei bildet die Modellbeschreibungssprache PROMELA des Model Checkers SPIN den Bildbereich der Abbildung. In dieser Sprache werden Kripkestrukturen spezifiziert. Zum anderen wird ausgearbeitet, wie aus einer Zielregel eine temporal-logische Formel abgeleitet werden kann, deren Erfülltheit dem Erreichen des Ziels entspricht. Durch den Model Checker SPIN wird die Erfülltheit der Formel im gegebenen PROMELA-Modell überprüft und so indirekt überprüft, ob die vorgegebene Strategie zum betrachteten Ziel führt.

Zur Einführung folgen zwei Abschnitte über die Konzepte von Model Checking und über den Model Checker SPIN. Darauf aufbauend wird ein Algorithmus präsentiert, mit dem Strategien in SPIN-Modellspezifikationen übersetzt werden können. Danach wird gezeigt, wie aus einer Zielregel eine temporal-logische Formel gebildet wird, deren Gültigkeit im gegebenen Modell mit Hilfe von SPIN überprüft wird. Vorarbeiten zu diesem Kapitel finden sich in [32].

7.1 Model Checking

Im Rahmen der modellbasierten Softwareentwicklung werden Modelle erstellt, die einen Satz gegebener Anforderungen erfüllen sollen. Das Problem dieses Ansatzes besteht darin,

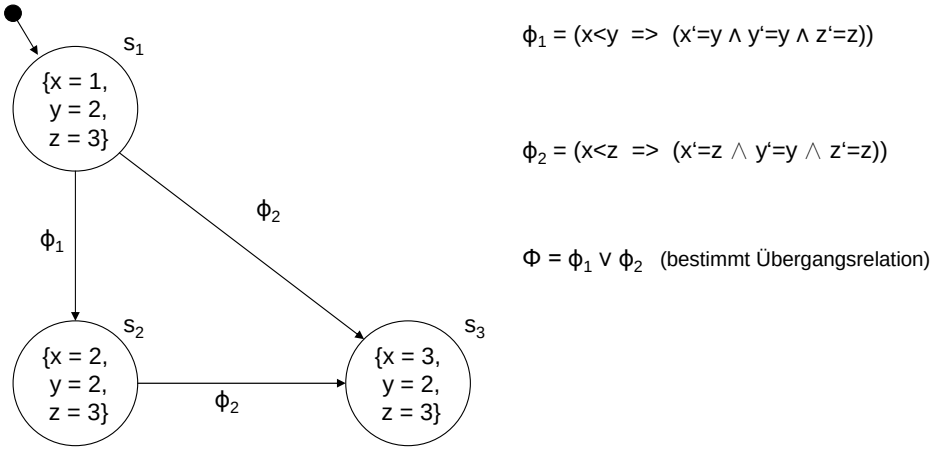

Figure 7.2: Beispiel-Transitionssystem

die Korrektheit eines Modells hinsichtlich der Anforderungen zu überprüfen. Model Checking ist eine automatische Technik, um endliche Zustandssysteme, also insbesondere endliche Transitionssysteme, bezüglich formal definierter Anforderungen zu verifizieren. Anforderungen sind durch temporal-logische Formeln ausgedrückt. Ein effizientes Suchverfahren auf dem Zustandsraum des Transitionssystems prüft, ob die Anforderungen erfüllt werden.

Zuerst werden die wesentlichen Konzepte von Model Checking vorgestellt (vgl. Kindler, Merz, Clarke, et al. [76, 82, 19]).

Zustandsbeschreibung. Der Zustandsraum wird in abstrakter Form durch eine Kripkestruktur beschrieben. Eine Kripkestruktur $M = (S, S_0, R, L)$ über einer Menge atomarer Aussagen AP besteht aus einer endlichen Zustandsmenge S, einer Menge von Anfangszuständen $S_0 \subseteq S$, einer (totalen) Übergangsrelation $R \subseteq S \times S$ und einer Abbildung $L : S \to 2^{AP}$, durch die jedem Zustand eine Menge atomarer Aussagen zugewiesen wird. Ein Pfad π in einer Kripkestruktur ist eine Folge von Zuständen $\pi = (s_1, s_2, ...)$, für die $(s_i, s_{i+1}) \in R$ für alle $i \in \mathbb{N}$ gilt.

Ein Beispiel für eine Kripkestruktur ist in Abbildung 7.2 dargestellt. Formal besteht die Kripkestruktur M_{Max} aus $S = \{s_1, s_2, s_3\}$, $S_0 = \{s_1\}$, $R = \{(s_1, s_2), (s_2, s_3), (s_1, s_3)\}$ und $L = \{(s_1, \{x = 1, y = 2, z = 3\}), (s_2, \{x = 2, y = 2, z = 3\}), (s_3, \{x = 3, y = 2, z = 3\})\}$.

Für konkrete Modelle ist der endliche Zustandsraum oft durch eine Menge von Belegungen $\sigma : V \to \{1, 2, \ldots, n\}$ von Variablen einer endlichen Menge V gegeben. Die Wertemenge der n ersten natürlichen Zahlen ist hier o.B.d.A. gewählt worden. Andere endliche Mengen sind ebenfalls zulässig. Im Beispiel sind die Variablenbelegungen durch die Abbildung L der Kripkestruktur M_{Max} gegeben. Die Belegungen der Variablen charakterisieren den Zustand eindeutig.

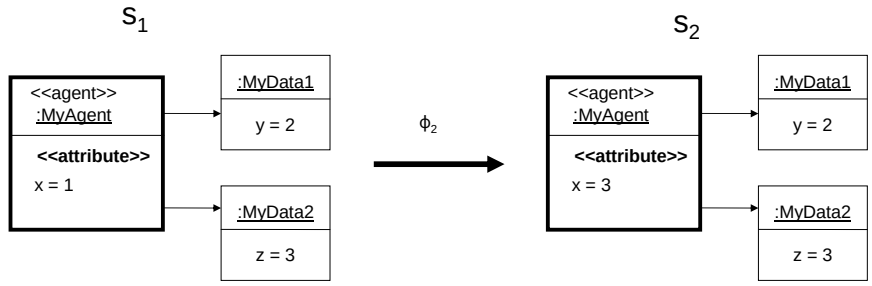

Figure 7.3: Zustandsübergang im Systemmodell

Um Zustandsübergänge durch Änderungen von Variablenbelegungen ausdrücken zu können, wird zu der Menge V eine Menge V' gestrichener Variablen gebildet, d.h. zu jedem $v \in V$ existiert genau ein $v' \in V'$. Die Zustandsübergangsrelation des Transitionssystems wird jetzt mit Hilfe der Variablen aus V und V' definiert, indem gestrichene Variablen als Träger der Werte im neuen Zustand angesehen werden. Der Zusammenhang mit dem vorherigen Zustand wird durch eine Disjunktion von Formeln für gestrichene und ungestrichene Variablen hergestellt, die jeweils bestimmte Übergänge spezifizieren. Das Resultat ist eine *Zustandsübergangsformel* ϕ, die der Zustandsübergangsrelation entspricht. Zwischen zwei Belegungen σ und σ' findet ein Zustandsübergang statt, falls mit diesen Belegungen ϕ erfüllt ist. Für die Zustandsübergangsrelation R gilt: $R = \{(\sigma', \sigma) | (\sigma', \sigma) \models \phi\}$. R ist also genau dann für zwei durch die Belegungen σ und σ' gegebenen Zustände erfüllt, wenn die Formel ϕ für diese Belegungen erfüllt ist.

Im Beispiel legt die Regel $\phi_1 = (x < y \Rightarrow (x' = y \wedge y' = y \wedge z' = z))$ den Übergang zwischen s_1 und s_2 fest. Die Regel $\phi_2 = (x < z \Rightarrow (x' = z \wedge y' = y \wedge z' = z))$ legt den Übergang zwischen s_1 und s_3 bzw. zwischen s_2 und s_3 fest.

Diese Sicht entspricht gut der Vorstellung eines ablaufenden Programms: Programmschritte überführen eine Belegung σ der Programmvariablen in V in eine nachfolgende Belegung σ' der (gestrichenen) Programmvariablen in V'. Das Programm entspricht dann der Formel ϕ mit $\phi = \phi_1 \vee \phi_2$. In einer imperativen Programmiersprache würde ϕ durch eine case-Anweisung mit unterschiedlichen Variablenzuweisungen für die verschiedenen Fälle realisiert werden. Im Kontext von Modellierungssprachen entsprechen Zustandsänderungen dem Übergang zwischen zwei Instanzdiagrammen. In Abbildung 7.3 ist ein Beispiel dargestellt.

Ausführungen und ihre temporal-logischen Eigenschaften. Die Ausführungen einer Kripkestruktur bestehen aus allen Pfaden, die von einem Anfangszustand ausgehen. Alle Ausführungen werden für einen Anfangszustand in einem *Computation Tree* zusammengefasst. Für das Beispiel ergibt sich der Baum in Abbildung 7.4. Die Ausführungen bestehen aus den Pfaden $\pi_1 = (s_1, s_2, s_3)$ und $\pi_2 = (s_1, s_3)$.

Atomare Formeln aus elementaren Gleichungen über Zustandsvariablen wurden bisher zur Charakterisierung von Zuständen verwendet. Durch boolesche Verknüpfungen von

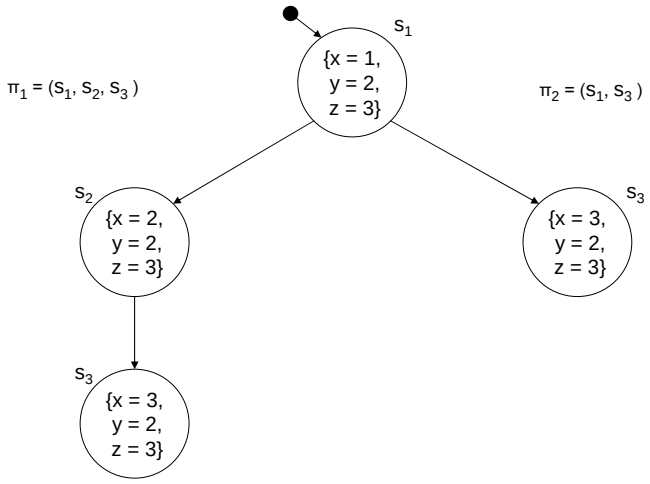

Figure 7.4: Computation Tree des Beispiels

atomaren Formeln werden *Zustandsformeln* gebildet, die auf komplexeren Zustandsmengen erfüllt sein können. Um komplexe Formeln aus elementaren Gleichungen zu vermeiden, sind nachfolgend auch Ungleichungen über Zustandsvariablen erlaubt. Diese können (bei endlichen Wertemengen) in Disjunktionen von Gleichungen über Zustandsvariablen expandiert werden.

Temporale Operatoren dienen dazu, die Gültigkeit einer Zustandsformel entlang eines Pfades in einer Kripkestruktur zu spezifizieren. Diese Operatoren produzieren aus Zustandsformeln Pfadformeln. Die entstehenden Formeln werden auf einem Pfad interpretiert. Die Pfadformel $G\phi$ für eine Zustandsformel ϕ ist auf einem Pfad π erfüllt, falls sie in allen Zuständen des Pfades (globally) erfüllt ist. Die Pfadformel $F\phi$ ist auf einem Pfad π erfüllt, falls sie in (mindestens) einem Zustand des Pfades (finally) erfüllt ist.

Die *Pfadquantoren* E und A dienen dazu, für einen Zustand s und eine Pfadformel ϕ über verschiedene Pfade zu quantifizieren. Die Zustandsformel $E\phi$ verlangt, dass es vom Zustand s aus einen Pfad gibt, auf dem ϕ gilt. $A\phi$ verlangt, dass von s aus auf allen Pfaden die Pfadformel ϕ gilt.

Als Beispiel sei eine Pfadformel $\phi = F(x \geq y)$ gegeben. ϕ ist auf einem Pfad erfüllt, falls in mindestens einem Zustand des Pfades $x \geq y$ gilt. Bezogen auf die vorgestellte Kripkestruktur gelten auf den Pfaden π_1 und π_2 sowohl $\pi_1 \models \phi$ als auch $\pi_2 \models \phi$, da $x \geq y$ im Zustand s_3 erfüllt ist (vgl. Abbildung 7.4). Also ist ϕ auf allen Pfaden erfüllt, die vom Anfangszustand s_1 ausgehen. Die Quantifizierung über alle Pfade ergibt die Zustandsformel $A\phi = AF(x \geq y)$. Sie ist im Zustand s_1 erfüllt, d.h. $s_1 \models A\phi$. Die Zustandsformeln $EF(x = y)$ ist im Zustand s_1 erfüllt, da auf dem Pfad π_1 im Zustand s_2 die Formel $x = y$ erfüllt wird.

Abhängig von den Möglichkeiten, Formeln zu bilden, entstehen unterschiedliche temporale Logiken. Die temporale Logik CTL^* (Computation Tree Logic) erlaubt das Bilden von Formeln in der allgemeinen Form, wie soeben beschrieben und erlaubt damit insbeson-

dere, Aussagen über einzelne Pfade zu spezifizieren. Um überprüfen zu können, ob durch eine Strategie ein Ziel erreicht wird, wird von einzelnen Pfaden abstrahiert, die vom Ausgangszustand ausgehen. Wichtig ist, ob ein spezifizierter Zielzustand auf *allen* möglichen Pfad erreicht wird. Eine temporale Logik, die von einzelnen Pfaden abstrahiert und sich allein auf Aussagen über Zustände beschränkt, ist CTL, die durch Einschränkung von CTL^* auf Zustandsformeln entsteht.

Auch CTL ist noch zu allgemein, da nur die Zustandsformeln wichtig sind, die über *alle* Pfade quantifizieren. Die temporale Logik LTL (Linear Temporal Logic) ist eine auf Zustandsformeln der Form $A\phi$ eingeschränkte Teilmenge von CTL^*. LTL Model Checking behandelt also die Frage, ob eine Pfadformel ϕ ausgehend von einem Zustand s auf allen von s ausgehenden Pfaden erfüllt ist, d.h. ob gilt $s \models A\phi$. Damit ist LTL die für die gegebene Problemstellung passende Logik. Der Pfadquantor A am Anfang einer Formel wird gewöhnlich unterdrückt. Ein Beispiel für eine LTL-Formel ist $\phi_{Max} = F(x \geq y \wedge x \geq z)$. In einem System, das ϕ_{Max} im Anfangszustand erfüllt, enthält x irgendwann einen mindestens so großen Wert wie y und z.

LTL Model Checking behandelt die Frage, ob für eine Kripkestruktur im Anfangszustand s die Formel $A\phi$ gilt, d.h. ob $s \models A\phi$. Diese Fragestellung ist äquivalent dazu zu überprüfen, ob $s \models \neg E(\neg\phi)$, bzw. ob $s \not\models E(\neg\phi)$ gilt, d.h. ob ein Zustand von s ausgehend existiert, in dem $\neg\phi$ nicht gilt. Algorithmen zum LTL Model Checking beantworten genau diese Frage und liefern ggf. den Zustand zurück, in dem $\neg\phi$ nicht gilt. Auf entsprechende Algorithmen und Datenstrukturen wird an dieser Stelle nicht eingegangen, siehe dazu [18].

Um Modelle durch Model Checking zu überprüfen, wird das Model Checking Werkzeug SPIN eingesetzt, das auf der Logik LTL basiert. In Abschnitt 7.3 wird gezeigt, dass LTL Model Checking für die zu Beginn dieses Kapitels aufgeworfene Problemstellung geeignet ist.

7.2 Der Model Checker SPIN

Der Model Checker SPIN akzeptiert die textuelle Sprache PROMELA zur Beschreibung von Modellen und Eigenschaften (vgl. [63]). Die Spezifikation eines Modells ähnelt einem Programm in einer höheren Programmiersprache.

Syntax. Die Syntax von PROMELA ist an die der Programmiersprache C angelehnt. Es existieren Sprachelemente, um Zustände und Verhalten von Modellen zu spezifizieren.

Zustände sind durch Belegungen von Variablen gegeben. Variablen sind nachfolgend vom Typ bool mit dem Wertebereich $\{0, 1\}$ oder int mit dem Wertebereich $\{-2^{31}, ..., 2^{31} - 1\}$ (vgl. [63]). Statt einzelner Variablen können auch Reihungen (Arrays) deklariert werden.

Das Verhalten des Modells wird in *Prozessen* spezifiziert. Die Prozesse sind vergleichbar mit Prozeduren in höheren Programmiersprachen. Sie sind Verhaltensspezifikationen und bestehen aus einem Kopf mit dem Namen des Prozesses und einem Rumpf, der eine

Folge von Anweisungen enthält. Anweisungen sind vor allem Zuweisungsoperationen für Variablen. Daneben sind Bedingungen in Form von Vergleichsoperationen auf Variablen spezielle Anweisungen, die ausgeführt werden, falls sie wahr sind. Die Ausführung einer Bedingung wird solange blockiert, bis sie erfüllt ist. In diesem Fall schreitet die Ausführung mit der nächsten Anweisung fort. Anweisungen sind durch die (äquivalenten) Separatoren ; oder -> getrennt. Letzterer wird als "syntaktischer Zucker" oft dazu verwendet, um eine Folge von Bedingungen von einer nachfolgenden Folge von Zuweisungen zu separieren. Durch die gesamte Folge aus Bedingungen und Zuweisungen wird dann eine *Reaktionsregel* definiert.

Durch das Schlüsselwort atomic ist es möglich, mehrere Anweisungen so zu gruppieren, dass sie als unteilbare Einheit, also transaktional ausgeführt werden. Für die Steuerung des Kontrollflusses existieren Anweisungen zur Mehrfachauswahl (if), zur Wiederholung (do) und zum unbedingten Sprung (goto). Der spezielle Prozess init wird zu Beginn ausgeführt. Weitere Prozesse müssen durch diesen Prozess in einer run-Anweisung angestoßen werden. Als Beispiel dient eine PROMELA-Spezifikation der Kripkestruktur M_{Max}:

```
/* PROMELA-specification of transition system M_{Max} */

/* Variables */
int x;
int y;
int z;

/* Initial process */
init
{
/* Initialization */
x = 1;
y = 2;
z = 3;

/* Transition specifications */
do
    /* phi_1 */
    :: atomic { x<y -> x=y; }
    /* phi_2 */
    :: atomic { x<z -> x=z; }
od
}
/* end of specification */
```

Die Verwendung der Sprachkonstrukte wird anhand der Semantik und der überprüfbaren Eigenschaften deutlich.

Semantik. Die Zustandsmenge S einer zu einer PROMELA-Spezifikation gehörigen Kripkestruktur M wird durch die in der Spezifikation verwendeten Variablen und die Kontrollzustände für alle Anweisungen zur Steuerung des Kontrollflusses aufgespannt. Kontrollzustände repräsentieren die aktuell ausgeführte Anweisung, bei Fallunterscheidungen den gewählten Fall und bei Schleifendurchläufen die Zahl der Durchläufe. Im obigen Beispiel spannen die drei Variablen x, y und z den Zustandsraum S auf. Der Anfangszustand ist durch die Initialisierung zu $x = 1$, $y = 2$ und $z = 3$ gegeben.

Die Übergangsrelation R leitet sich aus der Verhaltensspezifikation ab, die durch die Prozesse gegeben ist. Einzelne Anweisungen eines PROMELA-Modells führen zu Änderungen an Variablen. Sie tragen damit zur Definition einer Zustandsübergangsformel ϕ bei. In obigem Beispiel einer PROMELA-Spezifikation sind entsprechende Anweisungen zu den Formeln ϕ_1 und ϕ_2 eingefügt. Insgesamt ergibt sich die Zustandsübergangsformel $\phi = \phi_1 \vee \phi_2$.

Die sukzessive Ausführung von Anweisungen liefert eine Ausführungsfolge, die aus den Zuständen besteht, die nacheinander durch Ausführung der Anweisungen entstehen. Zu berücksichtigen sind nebenläufig ablaufende Prozesse. Die Fälle einer Mehrfachauswahl müssen sich nicht immer gegenseitig ausschließen. SPIN berücksichtigt bei der Überprüfung von Eigenschaften alle alternativen Fälle. Weiterhin können mehrere Prozesse definiert sein, die nebenläufig ablaufen. Ausführungsfolgen des Systems entstehen deshalb durch Verschachtelung (Interleaving) der Ausführungsfolgen einzelner Prozesse. Alle terminierenden und zyklischen Ausführungsfolgen definieren das Systemverhalten eines Modells. Da der Zustandsraum und die Verhaltensbeschreibung jedes PROMELA-Modells endlich ist, entstehen keine nichtzyklischen, unendlichen Ausführungsfolgen. Zyklische Ausführungsfolgen werden erkannt.

Eigenschaften von Modellen. Neben der Beschreibung von Modellen ist die Definition von Eigenschaften in Form temporaler Formeln notwendig. In die Spezifikation werden Bedingungen zum Test von Eigenschaften eingebaut. Diese Eigenschaften sollen in den Zuständen, die mit Ausführungen bis zu dieser Stelle der Spezifikation verträglich sind, erfüllt sein. Verschiedene Ausprägungen von Eigenschaften sind durch Zusicherungen, Invarianten, Deadlocks und zeitliche Anforderungen (temporal claims) gegeben.

Eine Zusicherung hat als Argument eine boolesche Bedingung. Eine Zusicherung in Form einer assert-Anweisung ist immer ausführbar und kann an beliebiger Stelle in einer Promela-Spezifikation stehen. Die Bedingung soll an der Stelle, an der die assert-Anweisung eingefügt ist, erfüllt sein. Invarianten des Systemverhaltens lassen sich mit Hilfe von Zusicherungen im Prozess init angeben.

Anweisungen können durch Label mit dem Präfix end markiert werden. Dadurch wird der vor der Anweisung erreichte Zustand als Endzustand ausgewiesen, d.h. an dieser Stelle wird der Prozess als regulär terminiert angesehen. *Deadlocks* sind Zustände, in denen keine weiteren Zustandsübergänge möglich sind und die keine regulären Endzustände sind.

Die bisher vorgestellten Eigenschaften lassen sich durch Analyse der Ausführungsfolgen

überprüfen. Für darüber hinaus gehende temporale Aussagen bietet SPIN auf temporaler Logik basierende Sprachelemente.

Eine *negative temporale Behauptung* ist durch eine Folge von Anweisungen gegeben, die in keiner Ausführung auftreten sollen. Eine solche Behauptung soll auf den Ausführungsfolgen überprüft werden. SPIN überprüft, ob diese Folge in einem gegebenen Modell für irgendeine Ausführungsfolge verletzt wird und liefert im zutreffenden Fall die zugehörige Ausführungsfolge mit den entsprechenden Zuständen als Gegenbeispiel. Eine solche temporale Behauptung entspricht einer temporal-logischen Pfadformel ϕ. Formal überprüft der Model Checker somit die LTL-Zustandsformel $A\phi$ für den Anfangszustand des Modells. Die Pfadoperatoren werden in PROMELA-LTL-Formeln wie folgt notiert: aus A wird $\Box$ respektive `[]` (always) und aus F wird $\Diamond$ respektive `<>` (eventually). Für das obige Beispiel ist die Pfadformel durch $\phi =$ `Maximum` gegeben:

```
#define Maximum ( <>(x >= y && x >= z))
```

Der Model Checker benötigt als Eingabe allerdings eine negierte temporale Aussage und findet gegebenenfalls Gegenbeispiele. Hier soll also überprüft werden, ob `notMaximum` verletzt wird:

```
#define notMaximum  ([](x<y) || [](x<z))
```

Auf Basis der Zielsprache PROMELA wird jetzt gezeigt, wie durch Graphtransformationssysteme gegebene Strategien in SPIN-Modellspezifikationen übersetzt werden können.

7.3 Von Strategien zu Modellspezifikationen in SPIN

Nach einem Vorschlag von Varro lässt sich aus einem Graphtransformationssystem mit Anfangszustand ein Transitionssystem ableiten (vgl. [99]). Der vorgeschlagene Algorithmus wird hier auf die Übersetzung eines Graphtransformationssystems in die Eingabesprache PROMELA des Model Checkers SPIN hin angepasst.

Eine Strategie ist durch ein Graphtransformationssystem $\mathcal{G} = \langle TG, RS, \emptyset, C\rangle$ gegeben. Weiterhin ist ein Anfangsgraph G_0 gegeben, der die Situation repräsentiert, in der die Strategie angewendet wird. Der Zustandsgraph G_0 respektiert die Bedingungen C von $\mathcal{G}$ und ist durch den Typgraphen TG getypt. Der Zustandsgraph G_0, der Typgraph TG und die Regelmenge RS werden zur Spezifikation eines Transitionssystems verwendet. Für das Graphtransformationssystem aus Kapitel 6 ist ein Beispiel für einen Anfangsgraphen G_0 in Abbildung 7.5 gegeben.

In der Eingabesprache PROMELA des LTL Model Checkers SPIN werden Zustände durch Reihungen (Arrays) der Typen `bool` und `int` ausgedrückt. Um Instanzgraphen zu repräsentieren, werden boolesche Arrays für Agenten, Rollen, Datenobjekte und Links des Typgraphen TG definiert, durch die die Zugehörigkeit des entsprechenden Elementes zum Zustand ausgedrückt wird. Die Indizes dieser Instanzvariablen repräsentieren die Identitäten der korrespondierenden Elemente. Die deklarierten Variablen werden entsprechend dem Anfangszustand, der durch G_0 gegeben ist, initialisiert.

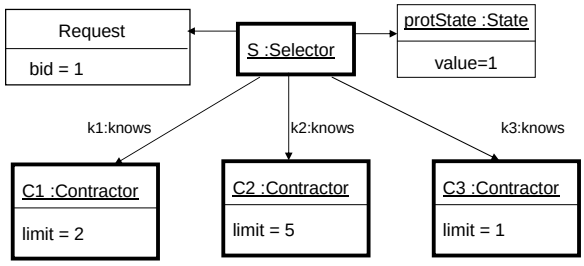

Figure 7.5: Beispielgraph G_0 eines Anfangszustand

Graphtransformationsschritte repräsentieren die Änderungen konkreter Elemente des Zustands. Das steht in Kontrast zu der abstrakten Spezifikation dieser Änderungen mit Hilfe von Regeln. Eine einzelne Regel führt zu verschiedenen Transformationsschritten, die als Instanziierung dieser Regel angesehen werden können. In der operationalen Semantik des DPO-Ansatzes besteht diese Instanziierung im Matching der linken Regelseiten im aktuellen Zustandsgraphen. Dieses Matching wird von Model Checkern wie SPIN nicht angeboten. Die Instanziierung erfolgt daher statisch als Teil der Übersetzung in die Eingabesprache PROMELA. Aus jeder Regel aus RS wird eine Menge von Transitionen für alle möglichen Instanziierungen dieser Regel. Diese Vorgehensweise wird dadurch eingeschränkt, dass der von einem Model Checker betrachtete Zustandsraum beschränkt ist. Daraus resultiert eine feste obere Grenze für die Zahl der Instanziierungen einer Regel.

Der betrachtete Zustandsraum kann abhängig von der Größe des Typgraphen TG und des Anfangszustands G_0 so groß sein, dass Model Checking nicht möglich ist. Die Idee von Varro besteht darin, die Größe des zu untersuchenden Zustands zu verkleinern, indem die Repräsentation auf solche Typen beschränkt wird, deren Instanzen tatsächlich durch eine der Regeln verändert werden, d.h. den dynamischen Teil des Zustands (vgl. [99]). Der statische Teil, der für die Regelanwendung nur gelesen aber nie modifiziert wird, ist lediglich explizit als Teil des Anfangsgraphen G_0 gegeben.

So werden in Strategien keine Agenten erzeugt oder gelöscht. Gewöhnlich werden insbesondere in Kommunikationsstrategien keine Rollen erzeugt oder gelöscht. Allein die Werte von Attributen von Agenten oder Rollen werden verändert. Durch Regeln werden während der Anwendung einer Strategie Datenobjekte oder Links zwischen Elementen erzeugt oder gelöscht. Eine obere Grenze für die Zahl der Datenobjekte kann durch Bilden aller aus G_0 resultierenden Graphtransformationen, die in einen Endzustand führen, bestimmt werden. Genau diese Aufgabe übernimmt aber ein Model Checker. Deshalb wird das Problem hier pragmatisch dadurch gelöst, dass für Datenobjekte, deren Attribute und Links die in SPIN maximal möglichen Array-Größen gewählt werden. Sollten die Größen nicht ausreichen, bricht der Model Checker mit einer Fehlermeldung ab und die Strategie ist aufgrund der Ressourcenbeschränkung nicht überprüfbar. Unter praktischen Gesichtspunkten liegt aber kein gravierendes Problem vor, da einzelne Strategien beschränkt komplex sind und eingeschränkt viele Datenobjekte erzeugen. Varro klammert diese Problematik aus (vgl. [99]).

Transitionen werden in PROMELA als Reaktionsregeln erfasst. Für jede Graphtransformationsregel wird die linke Regelseite bezüglich des statischen Teils des Zustands ausgewertet. Dieser entspricht dem statischen Teil des Anfangszustands G_0. Daraus resultiert eine feste Zahl von Transitionsspezifikationen für jeden Match der linken Regelseite. Die Vorbedingung jeder korrespondierenden Reaktionsregel besteht aus einer Konjunktion von atomaren Aussagen bezüglich der dynamischen Zustandsvariablen, die das Matching auf dem dynamischen Teil des Zustands repräsentieren. Im Zuweisungsteil der Reaktionsregel werden hauptsächlich Attribute mit neuen Werten belegt. Weiterhin wird ein Element dadurch erzeugt, dass eine unbenutzte Instanzvariable auf `true` gesetzt wird. Das Löschen eines Elementes wird durch das Setzen der zugehörigen Instanzvariable auf `false` realisiert.

Die Übersetzung eines Graphtransformationssystems in eine PROMELA-Spezifikation entsprechend dem beschriebenen Vorgehen ist durch folgenden Algorithmus gegeben. Er basiert auf dem allgemeinen Algorithmus, der von Varro vorgeschlagen wurde (vgl. [99]). Der Algorithmus ist an die Problemstellung dadurch angepasst, dass anfangs G0 explizit in einen statischen und dynamischen Anteil aufgespalten wird, dass nur die in Strategien auftretenden dynamischen Anteile in der Variablendeklaration berücksichtigt werden und dass die Spezifikation der Transitionen auf PROMELA zugeschnitten ist.

```
ALGORITHMUS AgentGTS2Promela
EINGABE: Graphtransformationssystem G = <TG, RS, G0, C>
AUSGABE: PROMELA-Spezifikation
METHODE:
BEGIN
    // Gebe mittels PRINT-Anweisungen eine PROMELA-Spezifikation aus.

    // Schritt 1: Analyse von G0

    Spalte G0 in statischen und dynamischen Anteil G0s und G0d auf,
    wobei zu G0s alle Elemente von G0 gehören, die nicht durch eine
    Regel aus RS modifiziert werden.

    // Schritt 2: Deklarationen

    // Die Elementeanzahlen der ARRAYs sind dadurch bestimmt, wieviele
    // Elemente des jeweiligen Typs dynamisch durch Regeln aus RS
    // erzeugt oder gelöscht werden.

    Die Elemente aus G0d werden typweise von Null an durchnummeriert.

    // Deklaration von 1D-ARRAYs boolescher Zustandsvariablen
    // für jeden Objekttyp Ok (k=1, ... , r) in TG,
    // der zu dynamischen Elementen in G0 gehört.
```

```
  PRINT (’’ bool O1[CntO1], ... , Or[CntOr]; ’’);

  // Deklaration von 2D-ARRAYs boolescher Zustandsvariablen
  // für jede Assoziation ASSOCi (i=1, ... , m) in TG

  PRINT (’’ bool ASSOC1[CntAss11,CntAss12], ...,
             ASSOCm[CntAssm1,CntAssm2]; ’’);

  // Ein 1D-ARRAY von Zustandsvariablen für jedes Attribut
  // ATTRIBi (i=1, ... , n) eines Agenten, einer Rolle oder
  // eines Objektes in TG.
  // Alle Attribute haben einen Aufzählungstyp <enum type> aus
  // {bool, int}

  PRINT (’’ <enum type> ATTRIB1[CntAtt1], ... ,
               <enum type> ATTRIBn[CntAttn]; ’’);

// Schritt 2: Initialisierung der deklarierten Variablen entsprechend
// dem durch G0 gegebenen Zustand

  Für jedes dynamische Element von G0 wird die entsprechende Variable mit dem
  zugehörigen Wert belegt

// Schritt 3: Erzeugung von Reaktionsregeln für Graphtransformationsschritte

  // Bilden der linken Seiten der Reaktionsregeln

  FOR EACH Graphtransformationsregel P:L-->R in RS DO
    IF exists match in G0s THEN
    FOR EACH Matching des dynamischen Anteils in P DO

    1: Erzeuge atomare Bedingungen B(L), die die Belegung von Elementen
       aus L in Bezug auf die Zustandsvariablen testen.

    2: Erzeuge Aktionen A(R) für R, d.h. Zuweisungen zu Zustandsvariablen,
       die Änderungen gemäß R ausführen.

    3: Erzeuge eine transaktionale Reaktionsregel, die auf der
       linken Seite eine Konjunktion der Bedingungen aus
       B(L)={b1(L), b2(L), ... , bm(L) }
       und auf der rechten Seite eine Folge von Aktionen aus
```

```
        A(R)={a1(R), a2(R), ... , an(R)} enthält, d.h.:
            PRINT('' ATOMIC{(b1(L) && b2(L) && ... && bm(L)) -->
                     (a1(R);a2(R); ... ;an(R))} '');
      END
    END
  END
END
```

Zur Illustration wird der Algorithmus auf das in Kapitel 6 vorgestellte Beispiel angewandt.

Zustandsrepräsentation. Zu Beginn werden die Variablen deklariert (Zeilen 1 bis 3 des Algorithmus). G_0 besteht aus einer Anzahl noOfContractors von Agenten des Typs contractor. Die Anzahl der Vorschläge ist auf noOfProposals beschränkt. Im Beispielzustand existieren höchstens drei contractors und drei proposals. Die booleschen Arrays cfp und prop werden für die dynamischen Objekte der Typen Proposal und CFP eingeführt. Das dynamische Erzeugen oder Löschen durch die Anwendung von Graphtransformationsregeln kann so repräsentiert werden.

```
bool cfp[noOfContractors];
bool prop[noOfProposals];
```

Alle anderen Instanzen von Rollen und Datenobjekten sind bereits im Anfangszustand enthalten und werden während der Ausführung des Protokolls weder erzeugt noch gelöscht. Der Startzustand ist ebenfalls durch die Variablenwerte statischer Klassen bestimmt: eine Variable limit wird für jede contractor-Rolle eingeführt und die Variable statevalue korrespondiert mit den Attributen des Objektes protState (vgl. Abb. 7.5).

```
int statevalue;
int limit[noOfContractors];
```

Für die dynamisch sich ändernden links existieren weitere boolsche Arrays:

```
bool sent[noOfProposals];
bool cfpSender[noOfProposals];
bool cfpRecipient[noOfProposals][noOfContractors];
bool propSender[noOfProposals][noOfContractors];
bool propRecipient[noOfProposals];
bool bidder[noOfContractors];
```

Alle Arrays bilden zusammen den Zustandsraum des Systems, in dem das contract-net-Protokoll untersucht wird. Die Arrays werden gemäß dem Startzustand G_0 initialisiert (Initialisierungsteil des Algorithmus, vgl Abb. 7.5). Beispielsweise werden die limit-Werte gesetzt und die Links sent werden mit false initialisiert.

```
#define CNPstart 1
```

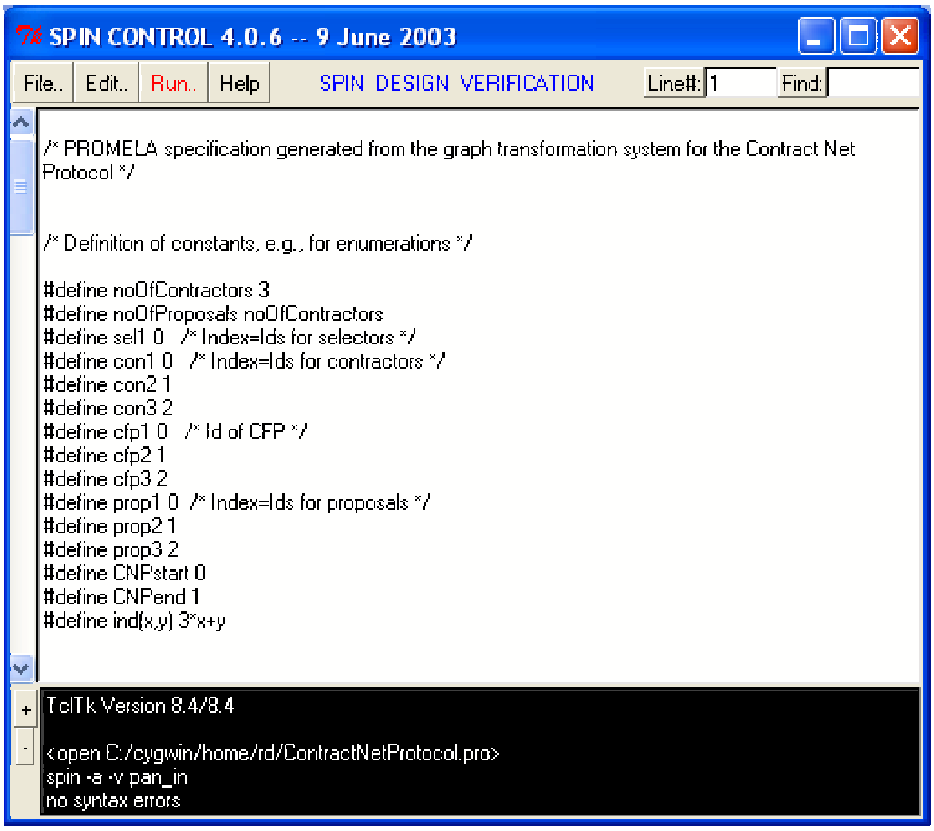

Figure 7.6: Screenshot von xspin mit geladener PROMELA-Spezifikation

```
#define CNPend 2
statevalue=1;
limit[1]=2;
limit[2]=5;
...
sent[1]=false;
sent[2]=false;
...
```

Reaktionsregeln. Für die Regel sendCFP (vgl. Abb. 6.11) entsteht die Reaktionsregel für den Match, der durch die Rolle 1 des Rollentyps Contractor und die Nachricht 1 des Typs cfp gegeben ist.

```
/* sendCFP */
:: atomic { msgreceived[1]==false ->
        msgreceived[1]=true; cfp[1]=true;
        cfpSender[1]=true; cfpRecipient[1][1]=true;  }
```

Das zu analysierende Modell ist vollständig in eine zugehörige PROMELA-Spezifikation eines Transitionssystems für den Model Checker SPIN übersetzt worden (vgl. Anhang A) [1]. In Abbildung 7.6 ist ein Screenshot des Werkzeugs xspin zu sehen, nachdem eine Syntaxüberprüfung der beschriebenen Spezifikation durchgeführt worden ist.

Im nächsten Schritt werden die in SPIN zu analysierenden Eigenschaften aus der Zielregel abgeleitet.

7.4 Transformation von Zielen in SPIN

Nach dem Erzeugen eines Modells besteht die nächste Aufgabe darin, den Begriff einer Gewinnstrategie für eine Zielregel $p : L \Rightarrow R$ nach SPIN zu übersetzen. Die linke Seite L ist eine Bedingung für den initialen Zustand des Systems. Falls diese Bedingung erfüllt ist, muss für eine Gewinnstrategie die rechte Seite R für jede Ausführung des Protokolls irgendwann erfüllt sein. Da jedoch L und R nicht disjunkt sind, sollten die Matches von L und R auf den geteilten Entitäten übereinstimmen.Um diese Bedingung zu übersetzen, müssen für jeden Match von L alle Matches von R in den erreichbaren Zuständen des Modells berücksichtigt werden.

In PROMELA können Modelleigenschaften durch Korrektheitsaussagen ausgedrückt werden. Ausgehend von einfachen Aussagen aus booleschen Bedingungen für Zustandsvariablen werden zugehörige Formeln mit Hilfe temporaler Operatoren gebildet. Eine Zielregel dient als Ausgangspunkt für die Konstruktion einer temporalen Formel für die Korrektheitsaussage.

Genau wie bei der Übersetzung von Strategien sind im wesentlichen die sich ändernden Teile des Zustands zu berücksichtigen. Nur die sich verändernden Teile der Zielregel müssen dynamisch gematcht werden, während die invarianten Teile nur im Anfangszustand gematcht werden müssen. Letzteres wird bei der Übersetzung der Zielregel erledigt, wobei für jeden einzelnen Match eine eigene Formel erzeugt wird. Alle entstehenden Formeln sind dann disjunktiv zu verknüpfen.

Für jeden Match $o_L(L)$ im Anfangszustand G_0 wird ein nachfolgender Match $o_j(R)$ benötigt. Dieser Zusammenhang wird durch eine temporal-logische Formel ausgedrückt, in der die Eigenschaft "nachfolgender Match" durch temporal-logischen Operator $\diamond$ (eventually) ausgedrückt wird.

$$goal_i \quad = \quad (o(L) \Rightarrow \diamond(o_1(R) \vee \ldots \vee o_k(R)))$$

Die verschiedenen Matches $o_1, \ldots, o_k$ von R sind nummeriert, wodurch die Matches mit den gewählten Matches o_L von L verträglich sind.

Die Formeln für die verschiedenen Matches von L sind durch in einer Disjunktion verbunden:

$$goal \quad = \quad (goal_1 \vee \ldots \vee goal_n)$$

[1] Es wurden SPIN in der Version 4.0.6 und die grafische Benutzungsschnittstelle xspin in der Version 4.2.0 benutzt (vgl. `www.spinroot.com`)

Im Beispiel hat die Disjunktion nur ein einzelnes Mitglied, da es nur einen Match der Regel p gibt (vgl. Abb. 6.15). Die rechte Seite von p matcht drei verschiedene Auftreten der contractor-Rolle. Dabei gehören die contractor-Rollen zum invarianten Teil des Zustands.

In PROMELA müssen die Matches durch Variablen ausgedrückt werden, die im vorherigen Abschnitt im Zusammenhang mit der Übersetzung von Strategien eingeführt wurden. Dabei müssen wieder nur die dynamischen Matches berücksichtigt werden:

```
#define occL (statevalue==cnpStart)
#define occR1 (statevalue==cnpEnd && bidder[con1]==true)
#define occR2 (statevalue==cnpEnd && bidder[con2]==true)
#define occR3 (statevalue==cnpEnd && bidder[con3]==true)
```

Aus diesen Definitionen wird eine temporal-logische Formel erstellt, die als Eingabe für den Model Checker SPIN dient:

```
#define CNPgoal (occL --> <>(occR1 || occR2 || occR3))
```

Der Model Checker benötigt als Eingabe eine negierte temporale Aussage.

```
#define notCNPgoal  (occL && [](!occR1) && [](!occR2) && [](!occR3))
```

Der Model Checker SPIN bestätigt die Gültigkeit der temporalen Formel `CNPgoal` (vgl. Screenshot in Abb. 7.7). Das Überprüfen der gegebenen Kommunikationsstrategie (Contract-Net-Protokoll) in Bezug auf die Eigenschaft CNPgoal ist damit gelungen. Daraus resultiert, dass die Strategie das Ziel in der durch G_0 gegebenen Situation erreicht. Allerdings wird nicht allgemein bewiesen, dass $\mathcal{G}$ eine Gewinnstrategie bezüglich p ist. Model Checking erlaubt lediglich das Testen eines gegebenen Zustandsmodells mit bestimmtem Anfangszustand.

7.5 Andere Ansätze zum Model Checking agentenbasierter Systeme

Die Überprüfung zielgerichteten Verhaltens durch Model Checking wurde in verschiedenen Ansätzen untersucht.

Von Wooldridge et al. wird ebenfalls der Model Checker SPIN für die Verifikation agentenbasierter Systeme verwendet [107]. Im Unterschied zum hier vorgeschlagenen Ansatz beruht die Verhaltensbeschreibung lediglich auf einer modifizierten BDI-Logik, die in PROMELA-Modelle und temporal-logische Aussagen in SPIN übersetzt wird.

Mocha ist ein Vorschlag und ein Werkzeug zum modularen Model Checking agentenbasierter Systeme [4]. Im allgemeinen besteht ein System aus lose gekoppelten Teilsystemen. Ein Zustandsübergangsgraph des Gesamtsystems bildet das zu analysierende Modell. Da Model Checking auf der umfassenden Exploration eines Zustandsraums beruht

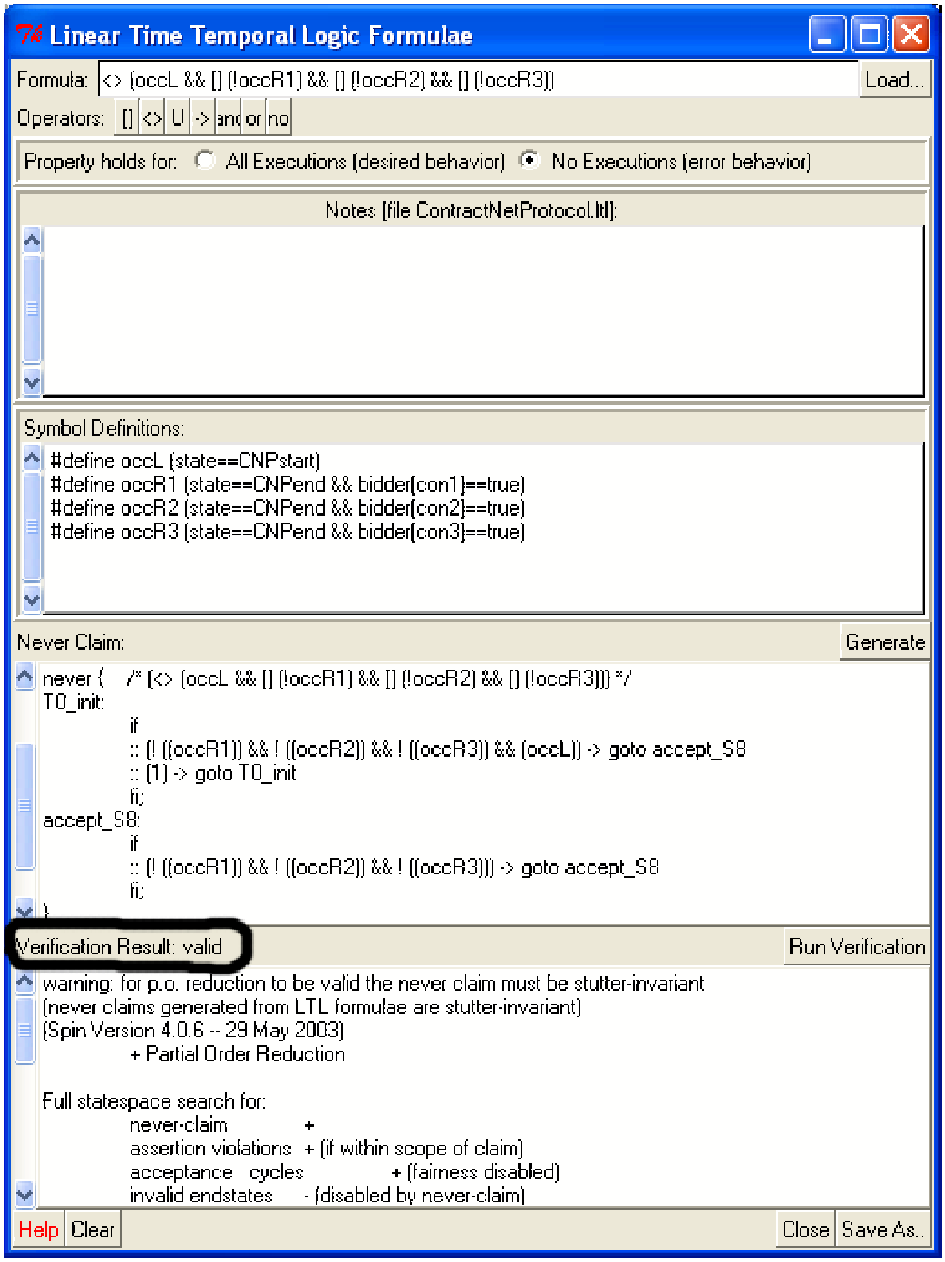

Figure 7.7: Screenshot der Verifikation der PROMELA-Spezifikation des Contract-Net-Protokolls

und die Größe des Zustandsraums eines Modells exponentiell mit der Größe der Modellbeschreibung wächst, ist die Skalierbarkeit ein wichtiges Problem.

Mocha besitzt ein hierarchisches System kommunizierender Module. Durch die Modularisierung wird die Manipulation unstrukturierter Zustandsübergangsgraphen vermieden. Eine ähnliche Situation liegt allerdings im hier vorgestellten Vorschlag ebenfalls vor: jede Regel einer Strategie kann als eine Zielregel aufgefasst werden, die durch eine Strategie zu verfeinern ist. In dieser Weise wird das Model Checking von Zielerreichbarkeit hierarchisch zerlegt. Da lokale Variablen innerhalb einer Strategie nicht die Erreichbarkeit anderer Ziele beeinflussen, wird so die Größe des zu untersuchenden Zustandsraums reduziert.

Alle vorgestellten Ansätze zum Model Checking benutzen textuelle Sprachen, die die Spezifikation von Zustandsübergangssystemen und die Formulierung von logischen Formeln zur Anforderungsdefinition erlauben. Im Unterschied können hier visuelle Spezifikationen auf der Basis von UML und Graphtransformation sowie durch Zielregeln gegebene Anforderungen analysiert werden.

7.6 Fazit

Agenten werden durch Anforderungen, die durch Ziele gegeben sind, kontrolliert. Agenten streben diese Ziele an, indem sie Strategien anwenden. Das Problem besteht darin, eine Strategie daraufhin zu überprüfen, ob sie ein gegebenes Ziel erreicht. Diese Frage ist insbesondere für Kommunikationsstrategien wichtig, die durch Protokolle für Interaktion zwischen Agenten gegeben sind. Ein Ziel ist durch eine Graphtransformationsregel und ein Protokoll durch ein Graphtransformationssystem mit Verhaltensregeln für die beteiligten Agenten gegeben.

Die Anwendbarkeit einer Graphtransformationsregel hängt von der Struktur des betrachteten Systemzustandes ab. Darum kann die Frage, ob eine Kommunikationsstrategie ein bestimmtes Ziel erreicht, nur für einen gegebenen Initialzustand beantwortet werden. Dabei muss die Anzahl der Elemente beschränkt werden, um einen endlichen Zustandsraum zu garantieren. Unter diesen Voraussetzungen ist ein Modell in eine Spezifikation eines Transitionssystems in der Eingabesprache PROMELA des Model Checkers SPIN übersetzt worden. Gezeigt wurde, wie eine Zielregel in eine zugehörige temporal-logische Formel übersetzt werden kann. Damit konnte die Erreichbarkeit des Ziels automatisch in SPIN überprüft werden.

Der beschriebene Ansatz zur Modellüberprüfung ist skalierbar, da zum einen lediglich der dynamische Anteil eines möglicherweise umfangreichen Ausgangszustands berücksichtigt wird. Zum anderen können einzelne Regeln von Strategien wieder als Zielregeln aufgefasst werden, die durch Strategien verfeinert werden. Andererseits ist die in dieser Art mögliche Skalierung problematisch, da vorausgesetzt wird, dass die bei der Abarbeitung der verschiedenen Teilstrategien entstehenden Zwischenzustände nicht wechselwirken.

Chapter 8

Ein Prozessmodell für AOSE

Aufbauend auf der im letzten Kapitel entwickelten Modellierungssprache wird das Agentenorientierte Prozessmodell (APM) für die agentenorientierte Softwareentwicklung erarbeitet. Die Anforderungen aus Kapitel 4 werden dabei berücksichtigt. Vorarbeiten zu diesem Kapitel finden sich in [38].

8.1 Übersicht über APM

Ein agentenorientierter Modellierungsprozess wird entsprechend den Anforderungen in drei typische, aus der objektorientierten Modellierung bekannte Aktivitäten aufgeteilt: Anforderungsspezifikation, Analyse und Entwurf (vgl. [38, 96, 93]). Diese Aktivitäten bestehen aus verschiedenen grundlegenden Teilaktivitäten mit den Ergebnisdokumenten Pflichtenheft, Architekturbeschreibung und Entwurfsdokument (vgl. Abb. 8.1). Die Ergebnisdokumente im APM enthalten Diagramme, die das Paradigma der Agentenorientierung entsprechend den Anforderungen aus Kapitel 4 berücksichtigen.

In der Anforderungsspezifikation wird zwischen Ist-Zustand und Soll-Konzept unterschieden. Der Ist-Zustand ist durch das strukturelle Modell für Begriffe des Problembereichs und durch das Geschäftsprozessmodell, das Funktionen und Abläufe im Problembereich beschreibt, gegeben. Das Soll-Konzept betrifft Funktionen des Geschäftsprozessmodells, die durch Produktfunktionen des zu erstellenden Softwaresystems realisiert werden sollen. Produktfunktionen, für die lediglich Informationen über ihre Voraussetzungen und den zu erreichenden Zustand bekannt sind, bilden Ziele, die durch geeignet zu wählende Strategien erreicht werden sollen. Die Beschreibungen von Ist-Zustand und Soll-Konzept bilden zusammen das Pflichtenheft.

Wie das System aufgebaut sein soll, wird in der Analyse herausgearbeitet. In der Grobanalyse werden Produktfunktionen und Nutzer zu Paketen gruppiert. Für die Pakete wird eine geeignete Zahl von Agenten eingeführt, die für ausgewählte Nutzer und Produktfunktionen zuständig sind. In der Feinanalyse werden für Ziele, zu denen Strategien bekannt sind, zugeordnete Strategien an eingeführte Agenten gebunden.

Im Entwurf werden für Ziele ohne bekannte Strategien, die mehrere Agenten betreffen,

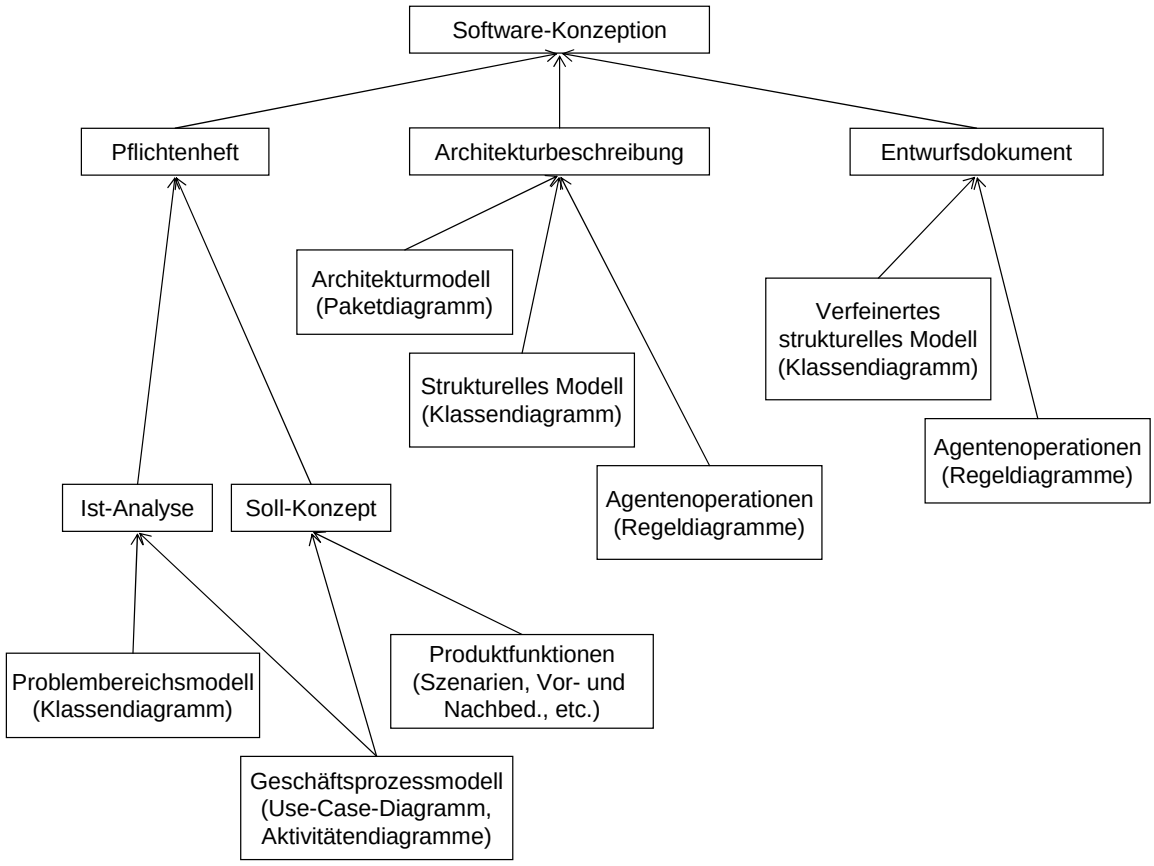

Figure 8.1: Dokumente des Prozessmodells APM

neue Protokolle entwickelt. Die neuen Protokolle werden dann ebenfalls gebunden. Damit sind für alle Ziele Strategien eingeführt. Im letzten Schritt wird das lokale Verhalten jedes Agenten verfeinert, indem das aus den Protokollbeteiligungen resultierende Verhalten bezüglich gebundener Rollen so integriert wird, dass lokale Ziele erreicht werden.

Der DAG (directed acyclic graph) in Abbildung 8.1 enthält alle im Verlauf der Entwicklung produzierten Dokumente. Durch die Pfeile wird beschrieben, welche Dokumente in andere Dokumente integriert werden. Zwischen den verschiedenen Dokumenten bestehen inhaltliche Abhängigkeiten. Die Dokumente der Architekturbeschreibung hängen von denen des Pflichtenheftes ab, und das Entwurfsdokument kann erst vollendet werden, wenn das Architekturdokument erstellt ist. Diese Vorrang-Beziehungen übertragen sich auf die Teildokumente. Weiterhin bestehen zwischen Teildokumenten weitere Abhängigkeiten, die nachfolgend geklärt werden. Beispielsweise basiert die Festlegung von Produktfunktionen auf einem vorhandenen Problembereichsmodell.

Jedes Dokument wird in einer eigenen Aktivität bearbeitet. Diese Aktivitäten zur Erstellung der verschiedenen Dokumente sind voneinander entsprechend den Dokument-Abhängigkeiten ebenfalls voneinander abhängig. Ein Durchlauf durch den DAG in Ab-

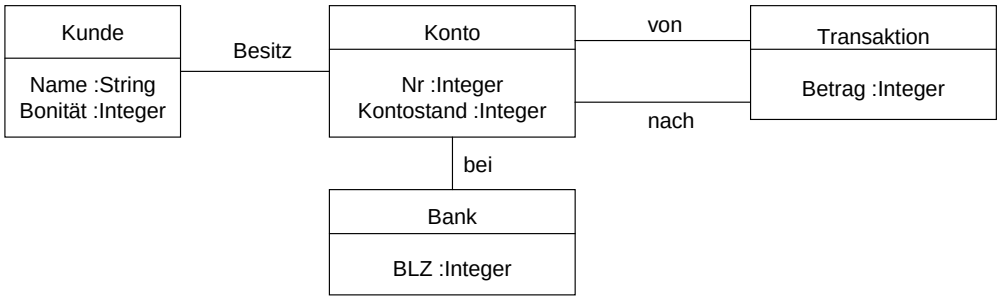

Figure 8.2: Klassendiagramm des Problembereichs

bildung 8.1 in preorder-Reihenfolge berücksichtigt die Abhängigkeiten, stellt aber nicht die einzige Möglichkeit einer Abfolge dar. Beispielsweise sind Iterationen in Teilbäumen denkbar. In dieser Arbeit sollen allerdings nicht Fragen des Projektmanagement, d.h. wer wann welche Aktivität übernimmt, diskutiert werden. Der Focus liegt stattdessen auf den Inhalten der Dokumente und den damit einhergehenden Einzelaktivitäten des APM.

8.2 Anforderungsspezifikation

In der Anforderungsspezifikation werden das Geschäftsprozessmodell, das Problembereichsmodell und die Spezifikation von Produktfunktionen entwickelt. Das Geschäftsfeld ist durch Geschäftsprozesse gegeben, die in Use-Case- und Aktivitätendiagrammen beschrieben werden. Der Problembereich wird durch ein Klassendiagramm modelliert. Die Produktfunktionen werden verschieden modelliert.

Als Anwendungsbeispiel für ein Geschäftsfeld dient ein Teilbereich des Bankwesens. Das Problembereichsmodell erfasst die Begriffe aus der Bankenwelt und deren Beziehungen. Der Problembereich wird durch das Klassendiagramm in Abbildung 8.2 beschrieben. Es enthält Klassen für die typischen Begriffe Kunde, Konto, Bank, Transaktion und die passenden Assoziationen dazwischen.

Das Geschäftsfeld ist durch ein Use-Case-Diagramm zur Beschreibung der Geschäftsprozesse durch Use Cases und deren Nutzer durch Akteure gegeben. Geschäftsprozesse im Bankbereich sind die Einzahlung von Bargeld, die Erstellung eines Kontoauszugs und die Überweisung von einem Kundenkonto auf ein Konto bei einer Fremdbank (vgl. Abb. 8.3). Überweisungen mit minimalen Kosten werden dadurch getätigt, dass für die Abwicklung eine Clearingstelle ausgesucht wird, die solche Transaktionen zwischen Banken mit den vergleichsweise geringsten Kosten durchführt.

In Abbildung 8.4 ist für die Geschäftsprozesse Überweisung und Überweisung mit minimalen Kosten eine Verfeinerung durch elementare Geschäftsprozesse dargestellt. Das Aktivitätendiagramm gibt an, in welcher Reihenfolge die elementaren Geschäftsprozesse ablaufen müssen, um den zugehörigen Geschäftsprozess zu realisieren. Elementare Geschäftsprozesse können selbst dadurch verfeinert werden, dass Szenarien dessen erfol-

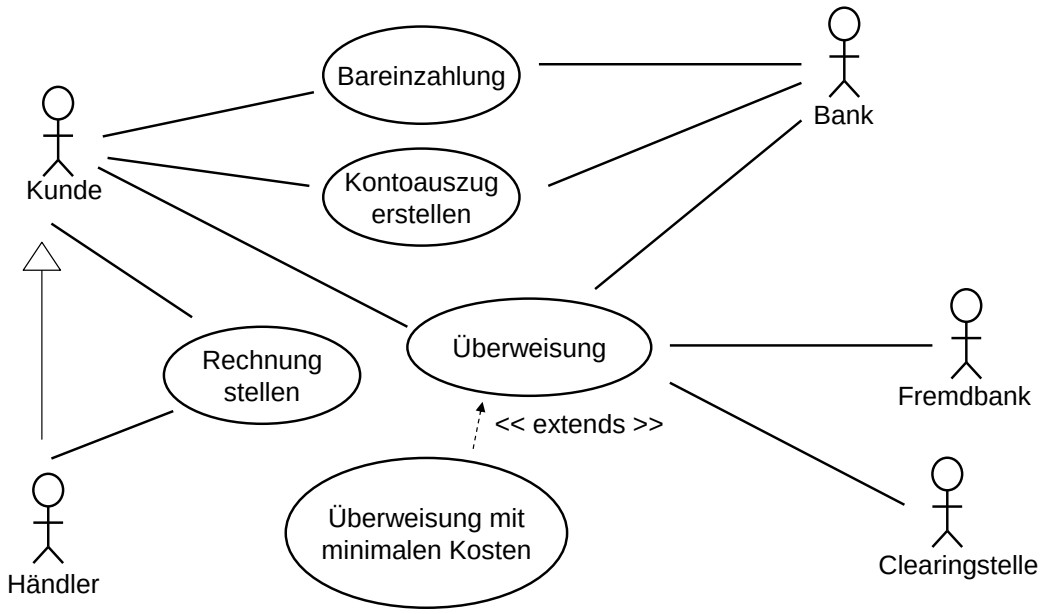

Figure 8.3: Use-Case-Diagramm für Geschäftsprozessmodell

greiche Abarbeitung beschreiben. Weiterhin können auch Szenarien für Misserfolgsfälle gegeben sein. Für den elementaren Geschäftsprozess Transaktion erstellen ist in Abbildung 8.5 ein Sequenzdiagramm angegeben, das Erfolgs- und Misserfolgsfälle integriert.

Elementare Geschäftsprozesse, bei denen das Verfahren zum Erreichen der benannten Funktion a priori unklar ist, werden durch das Stereotyp ⟨⟨ goal ⟩⟩ als Ziele ausgewiesen. Bei einem Ziel sind lediglich die Voraussetungen und der zu erreichende Zielzustand spezifiziert. In Abbildung 8.4 ist der elementare Geschäftsprozess Clearingstelle mit günstigsten Konditionen ermitteln als Ziel gekennzeichnet. In einer Notiz vom Stereotyp ⟨⟨ conditions ⟩⟩ sind Vor- und Nachbedingungen aufgeführt. Im Beispiel besteht die Vorbedinung darin, dass eine nichtleere Menge CS von Clearingstellen existiert. Die Nachbedingung fordert, dass eine Clearingstelle c mit $c \in CS$ existiert, deren Kosten minimal bezüglich der Kosten aller Clearingstellen in CS ist.

Geschäftsprozesse sollen durch das Softwaresystem automatisiert werden. Dazu werden elementare Geschäftsprozesse durch Produktfunktionen realisiert. Die Produktfunktionen sollen die vorgegebenen Erfolgs- oder Misserfolgsszenarien berücksichtigen. Für als Ziele ausgezeichnete elementare Geschäftsprozesse ist als Produktfunktion eine Strategie zu wählen oder auszuarbeiten, die die Vorbedingungen des Ziels erfüllen und den gewünschten Effekt bewirken. Die Frage der Realisierung von Produktfunktionen wird in Analyse und Entwurf behandelt.

Für die Produktfunktion Transaktion erstellen ist in Abbildung 8.5 ein Sequenzdiagramm gezeigt, das die Schritte bis zum erfolgreichen Erstellen einer Transaktion angibt.

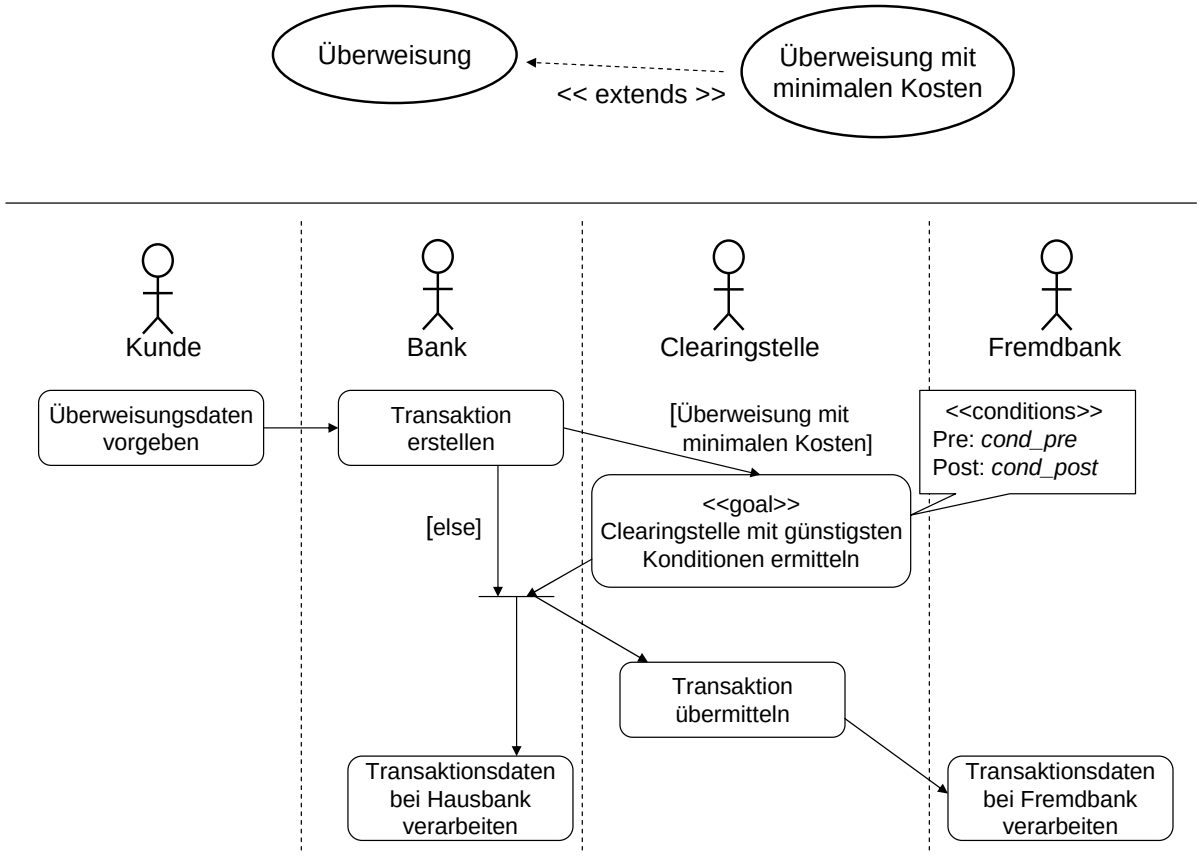

Figure 8.4: Elementare Geschäftsprozesse zu den Geschäftsprozessen Überweisung und Überweisung mit minimalen Kosten (Use-Case-Diagramm und Aktivitätendiagramm)

8.3 Analyse

In der Analyse wird modelliert, *was* das System tun soll, um die gegebenen Anforderungen zu erfüllen. Entwurfsentscheidungen über das detaillierte Verhalten von Agenten oder die Aufteilung der Software auf Hardwarekomponenten sollen vermieden werden. Im Entwurf soll danach die Frage beantwortet werden, *wie* das System funktioniert. Ähnlich wie in der objektorientierten Analyse wird zwischen Grob- und Feinanalyse unterschieden. In der Grobanalyse wird das System modularisiert, und Agenten werden eingeführt. In der Feinanalyse werden Struktur und Verhalten auf der Ebene von Zielen und Strategien verfeinert.

Grobanalyse

In der Grobanalyse soll das System zuerst strukturiert werden. Dazu werden Pakete gebildet, die eng verknüpfte Funktionen zusammenfassen sollen. Bedingungen für das

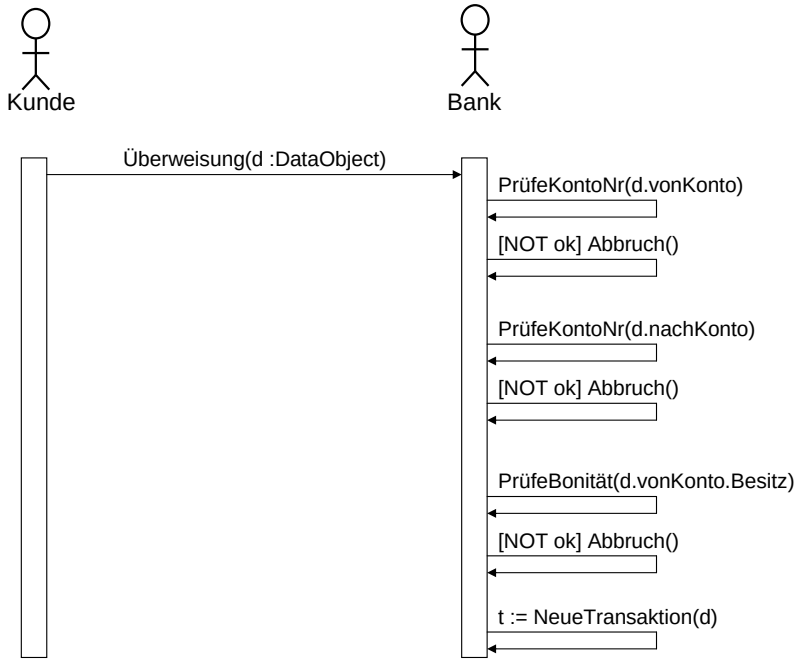

Figure 8.5: Sequenzdiagramm zur Produktfunktion Transaktion erstellen

Gruppieren von Produktfunktionen in Paketen sind:

- Produktfunktionen gruppieren, die nur einen Nutzer betreffen.
- Produktfunktionen mit gemeinsamen Teilfunktionen gruppieren.
- Jede Produktfunktion wird in genau ein Paket gruppiert.
- Jedes Paket enthält mindestens eine Produktfunktion.

Die Anwendung der Kriterien auf die Bankanwendung führt zur Bildung der Pakete in Abbildung 8.6. Nutzer sind mit einer oder mehreren Produktfunktionen von Paketen durch eine Kante vom Stereotyp trace verbunden. Gemäß den aufgestellten Kriterien enthält jedes Paket mindestens eine Produktfunktion. In einem agentenbasierten System werden Produktfunktionen durch Agenten realisiert, so dass jedem Paket mindestens ein Agent zuzuordnen ist. Zur Einführung von Agenten in die verschiedenen Pakete dienen folgende Kriterien:

- Für jedes Paket mindestens einen Agent einführen.
- Jede Produktfunktion muss von einem Agenten realisiert werden.
- Fakultativ: für unabhängige Produktfunktionen unterschiedliche Agenten einführen.

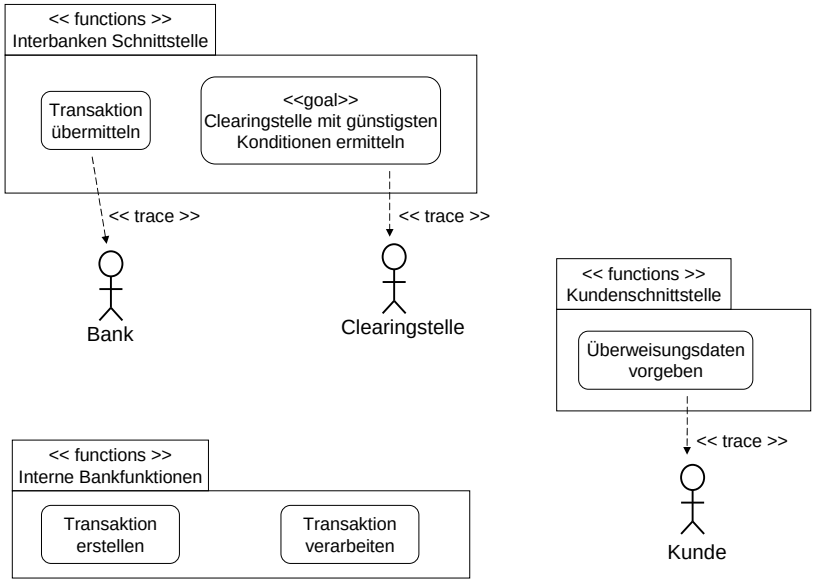

Figure 8.6: Gruppierung Produktfunktionen in Pakete

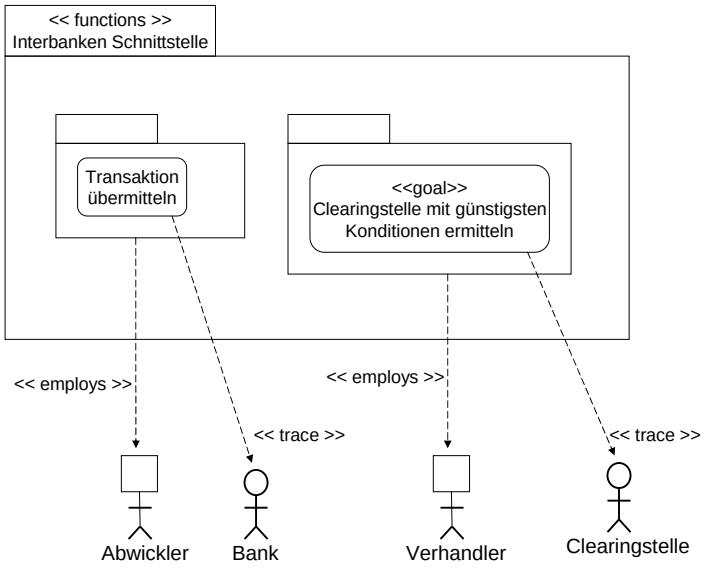

Figure 8.7: Zuordnung von Agenten zu Paketen

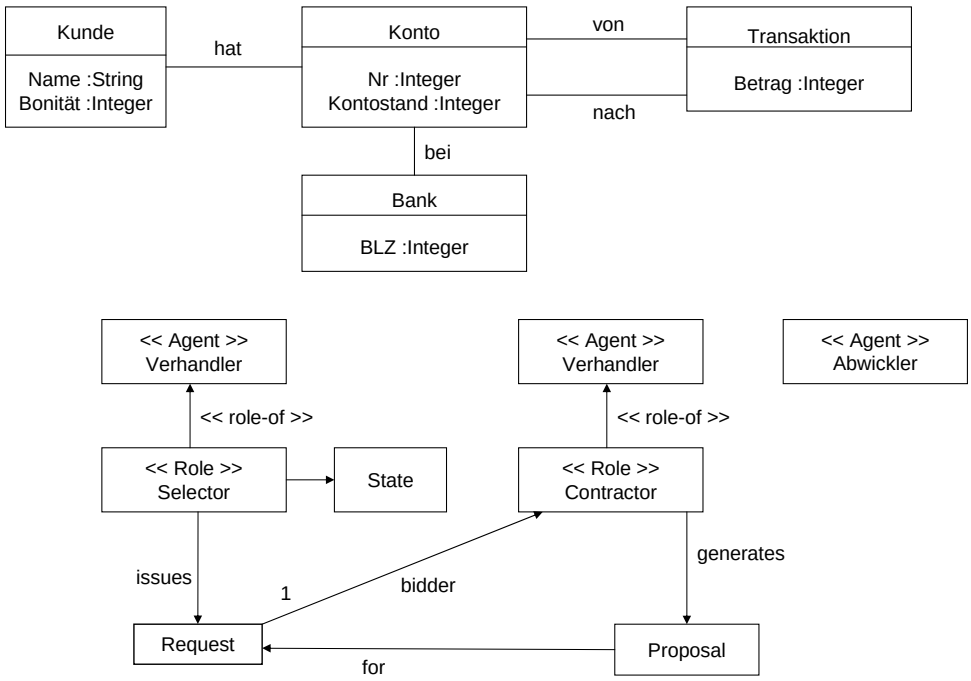

Figure 8.8: Analyseklassendiagramm des Banksystems

Zur Beschreibung agentenbasierter Systeme werden die Pakete mit als Agenten stereotypisierten Akteuren (mit quadratischen Köpfen) durch eine Kante mit Stereotyp ⟨⟨ employs ⟩⟩ verbunden. Damit werden die Anforderungen hinsichtlich der Verteilung von Funktionen auf verschiedene Agenten berücksichtigt. Ist ein Agent nur für einen Teil der Produktfunktionen eines Paketes zuständig, so werden die ihm zugeordneten Produktfunktionen in einem eigenen, dem Paket untergeordneten Paket gruppiert. Das Ergebnis ist ein verfeinertes Paketdiagramm, das ein (abstraktes) strukturelles Modell des Systems bildet. Das Diagramm in Abbildung 8.7 enthält die Agenten Verhandler und Abwickler für das Paket Interbanken Schnittstelle.

Feinanalyse

Eine Verfeinerung des Modells wird dadurch erreicht, dass Zielen Strategien zugeordnet werden. In der Analyse können als Ziele charakterisierte Geschäftsprozesse durch Produktfunktionen verfeinert werden, falls für sie bereits Strategien existieren. Neue Strategien werden erst im Entwurf entwickelt. Die zur Strategie gehörigen strukturellen Modellelemente werden in das gegebene Modell integriert, und die Verhaltensbeschreibungen werden an bestimmte Agenten gebunden.

Das Ziel Clearingstelle mit günstigsten Konditionen ermitteln wird durch die Kommunikationsstrategie *Contract-Net-Protokoll* erreicht (vgl. Kapitel 6). Die Frage, ob eine Strategie ein Ziel erreichen kann, wird mit dem in Kapitel 7 vorgestellten Verfahren zur

Modellüberprüfung beantwortet. Als Ergebnis der Bindung der Kommunikationsstrategie ergibt sich das Analyseklassendiagramm in Abbildung 8.8. Das Verhalten des Agenten Verhandler ist durch die zur Strategie gehörigen Regeln gegeben. Sie entsprechen denjenigen aus Kapitel 6, allerdings mit entsprechend modifizierter Bindung.

8.4 Entwurf

Das Entwurfsmodell arbeitet das Analysemodell weiter aus und konzentriert sich auf die Frage, *wie* das System funktionieren soll, d.h. das Verhalten der Agenten wird vollständig beschrieben. Das hat zur Folge, dass nicht mehr die globale Sicht auf das System im Blickpunkt ist, wie während der Analyse, sondern die lokale Sicht, die dann die Basis für eine Implementierung bildet.

Für nichtlokale Ziele, zu denen kein Protokoll als geeignete Kommunikationsstrategie bekannt ist, muss ein passendes Protokoll ausgearbeitet werden. Fragen des Protokollentwurfs wurden in Kapitel 7 diskutiert. Im wesentlichen sind Syntax, Semantik und Pragmatik von Protokollen festzulegen. Die Sprache AML legt folgendes Vorgehen nahe:

- Rollen für die Protokollbeteiligten einführen
- In Sequenzdiagrammen den Nachrichtenfluss zwischen den Rollen beschreiben
- Beschreiben, wie Rollen nach Erhalt bzw. vor dem Versand einer Nachricht operieren sollen, Attribute von Rollen und Datenobjekte einführen
- Lokale Operationen, sowie das Senden und das Empfangen von Nachrichten durch Graphtransformationsregeln beschreiben

Im nächsten Schritt besteht die Aufgabe des Entwurfs darin, das lokale Verhalten der Agenten vollständig zu beschreiben. Dazu müssen zu lokalen Zielen der Agenten Strategien ausgearbeitet werden. An einem Beispiel wird das passende Vorgehen demonstriert.

Für das lokale Ziel Transaktion verarbeiten des Agenten BackofficeAgent wird eine Strategie skizziert. Laut Vorbedingung ist eine Transaktion gegeben, die laut Nachbedingung zu einer Kontobewegung auf dem adressierten Konto führen soll. Dieses Verhalten ist durch eine Zielregel ausgedrückt (vgl. Abbildung 8.9). Eine lokale Strategie muss vom zu belastenden Konto einen entsprechenden Betrag abheben, den Betrag dem Zielkonto gutschreiben und dann die Transaktion löschen. Diese drei Operationen sind verfeinert zu beschreiben. Insbesondere ist dafür Sorge zu tragen, dass der Vorgang atomar abläuft, d.h. eventuelle Fehler in der Verarbeitung sind dahingehend zu berücksichtigen, dass der Ausgangszustand hergestellt wird.

Mit den in den bisherigen Schritten des Entwurfs hinzugekommenen strukturellen Modellelementen wird das Analyseklassendiagramm zu einem Entwurfsklassendiagramm erweitert.

Zuletzt ist im Rahmen des Entwurfs die Verteilung der Agenten mit ihren Rollen und der Datenobjekte auf verschiedene Rechenknoten einer Hardwarearchitektur festzulegen.

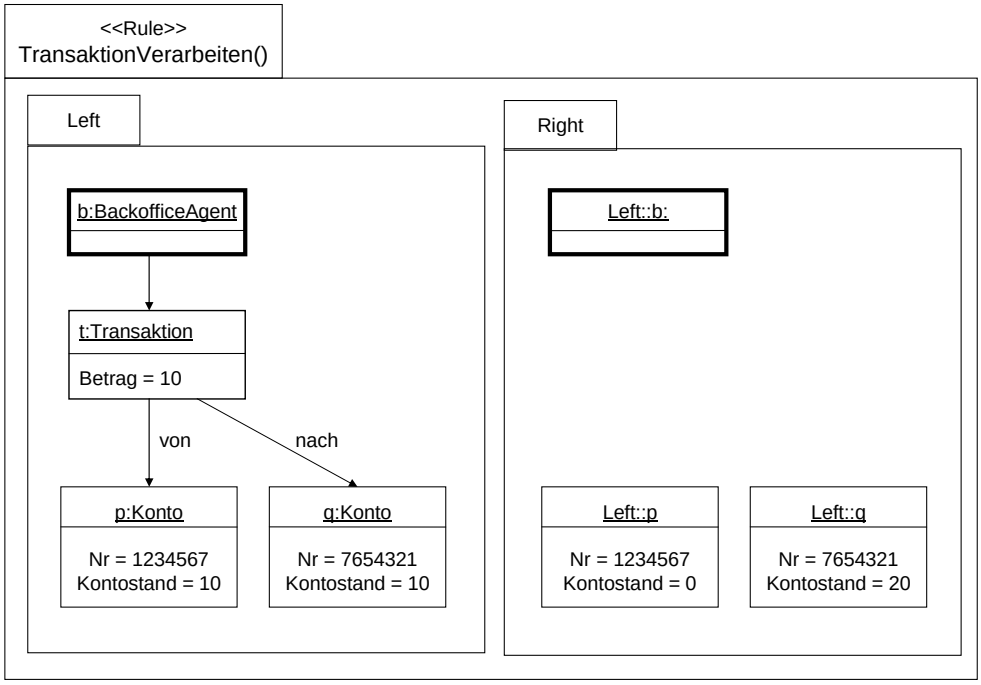

Figure 8.9: Zielregel zum lokalen Ziel Transaktion verarbeiten

Diese Problematik befindet sich allerdings nicht im Fokus dieser Arbeit und wird nicht weiter diskutiert.

8.5 Abhängigkeiten zwischen Artefakten

Die Artefakte in Pflichtenheft, Architekturbeschreibung und Entwurfsdokument bauen aufeinander auf. Dieses betrifft einerseits Klassen- und Paketdiagramme zur Beschreibung struktureller Modelle und andererseits Use-Case-Diagramme, Regeldiagramme und Sequenzdiagramme zur Verhaltensbeschreibung.

Abhängigkeiten bestehen zwischen den Klassendiagrammen des Problembereichsmodellls, des Analysemodells und des Entwurfsmodells. Das Analysemodell verfeinert das Problembereichsmodell und das Entwurfsmodell verfeinert das Analysemodell. Das Paketdiagramm des Architekturmodells wird wesentlich aus den im Pflichtenheft gegebenen Produktfunktionen abgeleitet.

Die Produktfunktionen werden aus dem Use-Case-Diagramm des Geschäftsprozessmodells und den Aktivitätendiagrammen der elementaren Geschäftsprozesse gewonnen. Produktfunktionen sollen elementare Geschäftsprozesse realisieren und sind einerseits durch Sequenzdiagramme für Erfolgs- und Misserfolgsfallszenarien und andererseits durch Vor- und Nachbedingungen für Ziele spezifiziert. Danach werden sowohl in der Analyse als auch im Entwurf Agentenoperationen durch Regeldiagramme notiert. Die Operationen

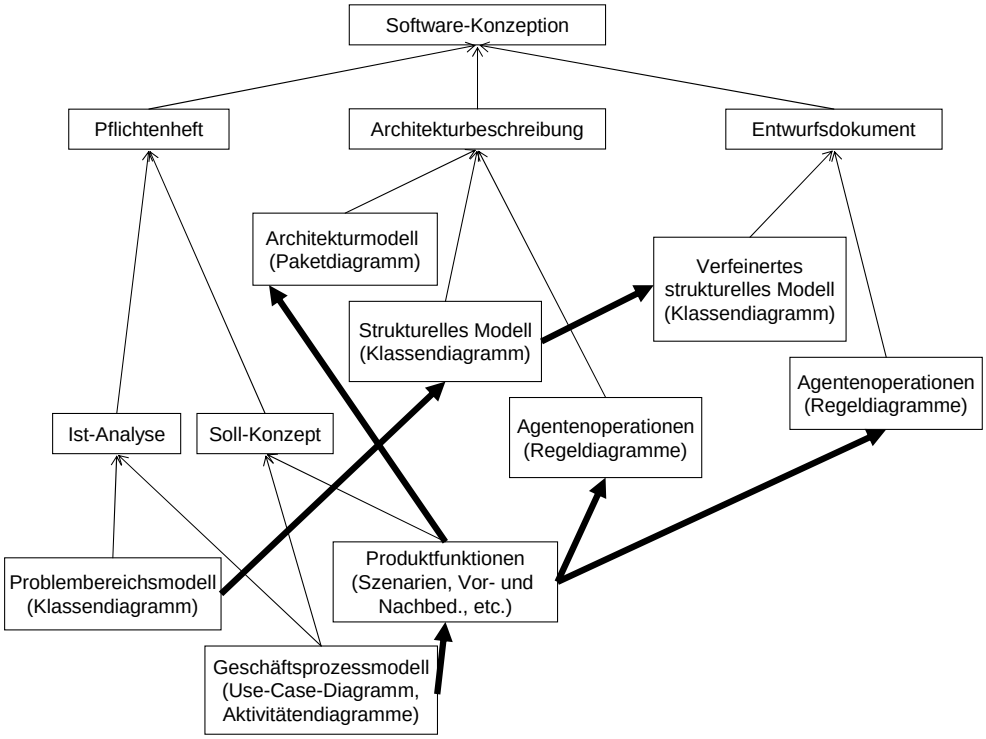

Figure 8.10: Abhängigkeiten zwischen Artefakten des Prozessmodells APM

realisieren Strategien und dienen dazu, das durch Sequenzdiagramme gegebene Verhalten zu realisieren.

Die Abhängigkeiten zwischen den verschiedenen Artefakten des Prozessmodells APM sind in Abbildung 8.10 durch fette Pfeile notiert. Die Abhängigkeiten schränken die Reihenfolge bei der Erstellung von Dokumenten ein, d.h. ein Projektablaufplan muss diese Randbedingungen berücksichtigen.

8.6 Fazit

In diesem Kapitel ist die Verwendung der Sprache AML demonstriert worden. In Anlehnung an etablierte objektorientierte Prozessmodelle wurde ein einfaches agentenorientiertes Prozessmodell (APM) entwickelt, das die Verwendung (Pragmatik) der Sprache AML beschreibt. Das Prozessmodell APM erfüllt die Anforderungen aus Kapitel 4. Dazu gehören die präzise Beschreibung der produzierten Artefakte und des Zusammenhangs der Artefakte. Strategien, wie beispielsweise Protokolle, können durch das in in Kapitel 7 vorgestellte Verfahren zur Modellüberprüfung während des Entwurfs verifiziert werden.

Chapter 9

Eine agentenbasierte Plattform für Informationssysteme

In diesem Kapitel werden drei Ziele verfolgt: zum einen soll für einen Anwendungsbereich (domain) die Nützlichkeit des agentenbasierten Ansatzes gezeigt werden. Zum anderen soll deutlich werden, wie eine UML-basierte Sprache zur Modellierung agentenbasierter Systeme verwendet werden kann. Last but not least soll eine Methode zur Implementierung agentenbasierter Systeme vorgestellt werden.

Den Anwendungsbereich bilden verteilte Internet-basierte Informationssysteme. Sie gewinnen einen immer höheren Stellenwert und weite Verbreitung. Unternehmensportale und Workflow-Managementsysteme sind aktuelle Beispiele derartiger Anwendungsplattformen. Diese müssen im Hinblick auf ihren Einsatz in gegebenen Anwendungsbereichen spezialisiert werden. Dazu ist es notwendig, die Struktur der Daten des Anwendungsbereichs und die ablaufenden Prozesse problemorientiert zu beschreiben. Beispiele für Prozesse sind Workflows unter Benutzerbeteiligung, aber auch nicht interaktive Abläufe wie komplexe Transaktionen in verteilten Buchungssystemen. Ein in einer Diagrammsprache wie UML beschriebenes Informationsmodell liefert eine auch dem Anwender verständliche Beschreibung von Datenflüssen und Abläufen. Für die Realisierung resultiert das Problem, das Modell innerhalb der gewählten Plattform in eine lauffähige Anwendung zu übertragen.

Im folgenden Abschnitt werden zunächst die Anforderungen an typische Informationssysteme anhand eines Beispiels abgeleitet. Wichtige Begriffe sind dabei Prozess und Workflow. Warum Informationssysteme mit Softwareagenten vorteilhaft realisiert werden können, wird in Abschnitt 9.2 erläutert. Danach wird gezeigt, wie Workflows auf der Basis von UML und AML modelliert werden können. In Abschnitt 9.4 wird eine Plattform für die Workflow-Verarbeitung vorgestellt. Als Anwendungsbeispiel der Plattform dient das Workflow-Management, das im Rahmen einer Vorlesung notwendig ist.

9.1 Verteilte Informationssysteme

Als Anwendungsbeispiel dient ein Informationssystem einer Universität. Es ist in seiner aktuellen Form oft nicht personalisiert und verursacht bei den betroffenen Benutzern (Professoren, Mitarbeitern und Studenten) einen hohen Aufwand. Für die meisten Studenten bedeutet dies tagtägliches Suchen in verschiedenen Webseiten, um an relevante Informationen wie z.B. Übungszettel, Skripten, Klausurtermine etc. zu gelangen.

Auf der anderen Seite müssen Professoren und ihre Mitarbeiter für jede neue Veranstaltung in Form von Vorlesungen, Seminaren und Projektgruppen neue Webseiten erstellen und diese mühsam von Hand pflegen. Ferner müssen Sie die Teilnehmerlisten für Übungen oder Klausuren erstellen und verwalten und ausserdem die aktuellen Veranstaltungsressourcen wie z.B. Übungszettel und Musterlösungen von Hand auf den Webseiten zur Verfügung stellen.

Dieses Szenario zeigt auf, welche Anforderungen an ein webbasiertes Informationssystem zur Unterstützung des Vorlesungsbetriebs zu stellen sind. Benötigt werden Funktionen für Management- und Erstellungsprozesse in der Universität, wozu z.B. die Pflege persönlicher Seiten, die Erstellung von Dokumenten, das Management von Vorlesungen mit Übungen und die (automatische) Aktualisierung von Webseiten gehören. Eine detailliertere Liste mit Anforderungen ist in [90] aufgeführt.

Für die durch die genannten Anforderungen abgegrenzte Problemklasse werden heute Workflow-Management-Systeme eingesetzt. Zentral sind in solchen Systemen die Begriffe Prozess und Workflow, auf die im folgenden eingegangen wird.

9.1.1 Prozesse und Workflows

Bei den Abläufen in einer Organisation kann man die auftretenden Prozesse in Material-, Informations- und Geschäftsprozesse klassifizieren [1] Materialprozesse bewirken den Transport von materiellen Dingen und sind von der weiteren Betrachtung ausgeschlossen. Zuerst soll der Unterschied zwischen Informations- und Geschäftsprozessen deutlich gemacht werden. Informationsprozesse nehmen Daten aus verschiedenen Quellen, wie Datenbanken oder auch von Benutzern entgegen, verarbeiten diese Daten, leiten sie weiter oder speichern sie. Informationsprozesse sind oft durch Softwaresysteme automatisiert oder werden von Anwendern mit Rechnerunterstützung bearbeitet. Als Beispiel sei wieder der Vorlesungsbetrieb genannt, dessen Informationsprozess weitgehend auf ein Softwaresystem übertragen werden soll. Geschäftsprozesse (engl.: business processes) sind umfassende Prozesse, die in unmittelbarem Zusammenhang zu einem Organisationsziel stehen und einen messbaren Beitrag zum Erreichen des Zieles leisten sollen. Geschäftsprozesse sind übergeordnete Prozesse, die durch Material- und Informationsprozesse realisiert werden [1]. Die Geschäftsprozesse werden a priori unabhängig von Informationsystemen beschrieben. Gesammelt werden die funktionalen Anforderungen, die sich aus dem Organisationsziel ableiten. Informationssysteme sind lediglich ein Mittel der Umsetzung und Automatisierung der Prozesse.

Geschäftsprozesse sind zum einen permanente Prozesse, die sich durch eine niedrige

Schwankung und eine daraus resultierende hohe Standardisierung der Arbeitsabläufe auszeichnet. Permanente Prozesse werden auch als Routineprozesse bezeichnet. Andererseits handelt es sich bei Geschäftsprozessen um temporäre Prozesse, die eine große Zahl von möglichen Vorgehensweisen bzw. Abläufen umfassen. Temporäre Prozesse werden auch als Projekte bezeichnet. Für beide Arten von Geschäftsprozessen sollen die darin enthaltenenen Informationsprozesse so weit wie möglich automatisiert werden.

Um Informationsteilprozesse eines Geschäftsprozesses auf ein Softwaresystem übertragen zu können, müssen Aufgaben, Abläufe und Verantwortlichkeiten genau, eindeutig und vollständig beschrieben werden. Eine detaillierte Beschreibung von Informationsprozessen, den darin enthaltenen Aktivitäten sowie den Ablaufbeziehungen der Aktivitäten untereinander bezeichnet man als Workflow. Die Workflow-Management-Coalition (WFMC) definiert in ihrem Referenzmodell den Begriff Workflow wie folgt [22]: "The computerized facilitation or automation of a business process, in whole or part." Häufig wird der Begriff aber nicht nur für die Beschreibung eines Vorgangs sondern auch für den Vorgang selbst benutzt. Ist eine explizite Unterscheidung nötig, wird von Workflow-Modell auf der einen Seite und Workflow-Instanz auf der anderen Seite gesprochen.

Workflow-Modelle bestehen hauptsächlich aus Ablaufdefinitionen, in denen Aktivitäten beschrieben werden. Eine Aktivität ist eine elementare, unmittelbar ausführbare Operation oder selbst ein möglicherweise komplexer Ablauf vieler untergeordneter Aktivitäten. Unterschiedliche Aktivitäten können sequentiell, nebenläufig oder selektiv ausgeführt werden. Ihre Ausführung kann von Vorbedingungen und Ereignissen abhängen. Aktivitäten modifizieren passive Objekte, die oft als Dokumente bezeichnet werden. Aktivitäten werden von zugeordneten Personen oder Softwarekomponenten ausgeführt. Im Modell wird von den konkreten Instanzen oft abstrahiert und stattdessen auf Rollen zurückgegrifffen. Eine Rolle gruppiert verschiedene Aktivitäten in der Weise, dass sie gemeinsam einer ausführenden Instanz zugeordnet werden können.

Instanzen von Workflow-Modellen werden Software gestützt ausgeführt. Auf diese Aufgabe sind die Workflow-Management-Systeme spezialisiert.

9.1.2 Workflow-Management-Systeme (WFMS)

Falls Workflows durch Informationssysteme unterstützt werden sollen, indem zum Beispiel die Folge der einzelnen Aktivitäten gesteuert und verantwortliche Personen oder Komponenten für die Aktivitäten zugewiesen werden, benötigt man ein Workflow-Management-System (WFMS). Ein solches System wird im Referenzmodell der Workflow-Management-Coalition definiert als [22]:

"A system that defines, creates and manages the execution of workflows through the use of software running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications."

Charakteristisch für WFMS ist, dass sie heterogen sind. Sie sind mit verschiedenen Anwendungen gekoppelt und an der Abarbeitung eines Workflows sind verschiedene Benutzer beteiligt. Die Benutzer und die System- und Softwarekomponenten arbeiten

kooperativ im Rahmen eines Workflows. Der Zusammenhang zwischen verschiedenen Komponenten kann durch häufige Interaktion geprägt sein und die Komponenten greifen häufig auf Daten zu. Dieses Verhalten führt zu einer hohen Komplexität der Workflows. Die Operationen der beteiligten Komponenten sind typischerweise nicht vollständig unter der Kontrolle des WFMS, d.h. diese Komponenten arbeiten in Bezug auf das WFMS autonom. Darüberhinaus kann das System offen sein, wenn vor Abarbeitung des Workflows nicht bekannt ist, welche Komponenten teilnehmen. Stattdessen wird der Workflow während der Abarbeitung adaptiert, sobald neue Komponenten hinzukommen. Die Abarbeitung von Workflows kann sich über lange Zeitspannen (Tage, Wochen) hinziehen.

9.2 Agenten für Workflow-Managementsysteme

Softwareagenten, oft einfach Agenten genannt, bilden seit einiger Zeit eine Basis für Architekturen von Workflow-Management-Systemen (WFMS). Derartige Systeme werden aufgrund ihrer Charakteristika als prädestiniert für den Einsatz von Agenten angesehen. Die verteilte und nebenläufige Bearbeitung von Workflows durch autonome Instanzen in einer offenen Umgebung mit a priori unvollständig bekannten Komponenten ist dabei das wichtigste Merkmal, das für den Einsatz von Agenten spricht.

Ihre Flexibilität, Autonomie und strukturierte Interaktion machen Agenten für bestimmte Problem- und Systemklassen geeignet:

- In Multiagentensystemen sind die nebenläufig ablaufenden Agenten auf verschiedene (entfernte) Rechenknoten verteilbar.
- Agenten homogenisieren Systeme, indem Anwendungen als Agenten gekapselt werden.
- Der Datenaustausch zwischen Anwendungen wird auf Agentenkommunikation zurückgeführt und dadurch homogenisiert.
- Agentenbasierte Systeme sind skalierbar, indem von Agenten eines Typs die erforderliche Anzahl von Instanzen im Netz verteilt erzeugt wird.
- Die Robustheit nimmt durch den Einsatz von geklonten Agenten für ausfallgefährdete Agenten zu.

Workflow-Management-Systeme profitieren von den Vorteilen von Agenten [66, 74, 31]. WFMS werden in der Regel in einer verteilten Umgebung eingesetzt. Insbesondere, wenn das WFMS skalierbar sein soll, ist es sinnvoll, auch das WFMS zu verteilen. Beides gelingt mit Agenten. Üblicherweise müssen WFMS zur Laufzeit mit verschiedenartigen Workflows, an denen wechselnde Personen und Softwarekomponenten beteiligt sind, zurechtkommen. Das ist insbesondere bei temporären Prozessen (Projekten) der Fall. An

dieser Stelle können die an einem Workflow beteiligten Komponenten flexibel durch Agenten realisiert werden. Weiterhin sind WFMS in der Regel stark mit anderen Systemkomponenten gekoppelt. Diese Heterogenität kann durch Kapselung externer Komponenten mit Agenten gemindert werden.

Um Agenten in WFMS einsetzen zu können, ist zu klären, wie zugehörige Workflow-Modelle aufgebaut sind und welche Architektur für ein derartiges WFMS geeignet ist. Zunächst soll die Modellierung von Workflows auf der Basis der Modellierungssprache Unified Modeling Language (UML) und des in Kapitel 6 eingeführten UML-Profils AML erörtert werden.

9.3 Agentenbasierte Modellierung von Workflows mit UML in BUSSARD

UML hat sich in den letzten Jahren als Industriestandard durchgesetzt. Damit stellt sich die Frage nach der Eignung von UML für die Workflow-Modellierung. Diese Frage wurde positiv beantwortet [86, 42]. Für Workflow-Modelle wurden von den vorhandenen Diagrammarten lediglich Klassen- und Aktvitätendiagramme verwendet.

In dem agentenbasierten Workflow-Mangementsysem BUSSARD werden UML-Klassendiagramme und UML-Aktivitätendiagramme zur Beschreibung von Workflows verwendet [90]. Das System BUSSARD wird unten genauer beschrieben. Mit diesem System wurde gezeigt, dass UML auch für die Beschreibung von ausführbaren UML-Agentenmodellen geeignet ist. Im Unterschied zur Sprache AML wird in BUSSARD die Semantik der Diagramme durch ihre Implementierung festgelegt. Am Ende dieses Abschnitts werden die Unterschiede diskutiert.

BUSSARD-Modelle basieren auf den Konzepten Objekt, Agent, Rolle und Prozess. Der Anwendungsbereich wird durch ein UML-Klassendiagramm beschrieben. Dem Benutzer können verschiedene Rollen zugeordnet werden, die im Prozessablauf benutzerspezifische Aktivitäten ausüben. Charakteristisch für das Modell und die entwickelte Plattform ist, dass Rollen von Software-Agenten übergenommen werden, die autonom die entsprechenden Aktivitäten ausführen und gegebenenfalls mit dem Benutzer kommunizieren. Prozesse werden durch UML-Aktivitätendiagramme modelliert, die beschreiben, welche Rollen Aktivitäten in welcher Reihenfolge oder parallel übernehmen. Neben dem Kontrollfluss wird auch der Objektfluss zwischen verschiedenen Aktivitäten modelliert. Zur Abarbeitung der modellierten Prozesse in der von uns entwickelten Anwendungsplattform werden die Modelle in XML-Dokumente abgebildet. Die aus den Aktivitätendiagrammen erzeugten XML-Dokumente werden von den Agenten interpretiert und verteilt abgearbeitet.

Ein Aktivitätendiagramm ist gemäß der UML-Spezifikation [86] ein besonderes Zustandsdiagramm, siehe Abb. 9.1. Die Zustände werden als Aktivitäten bezeichnet, da im Zustand selbst Operationen ausgeführt werden. Eine Aktivität wird grafisch durch ein an den Ecken abgerundetes Rechteck dargestellt. Die Aktivitäten sind durch Tran-

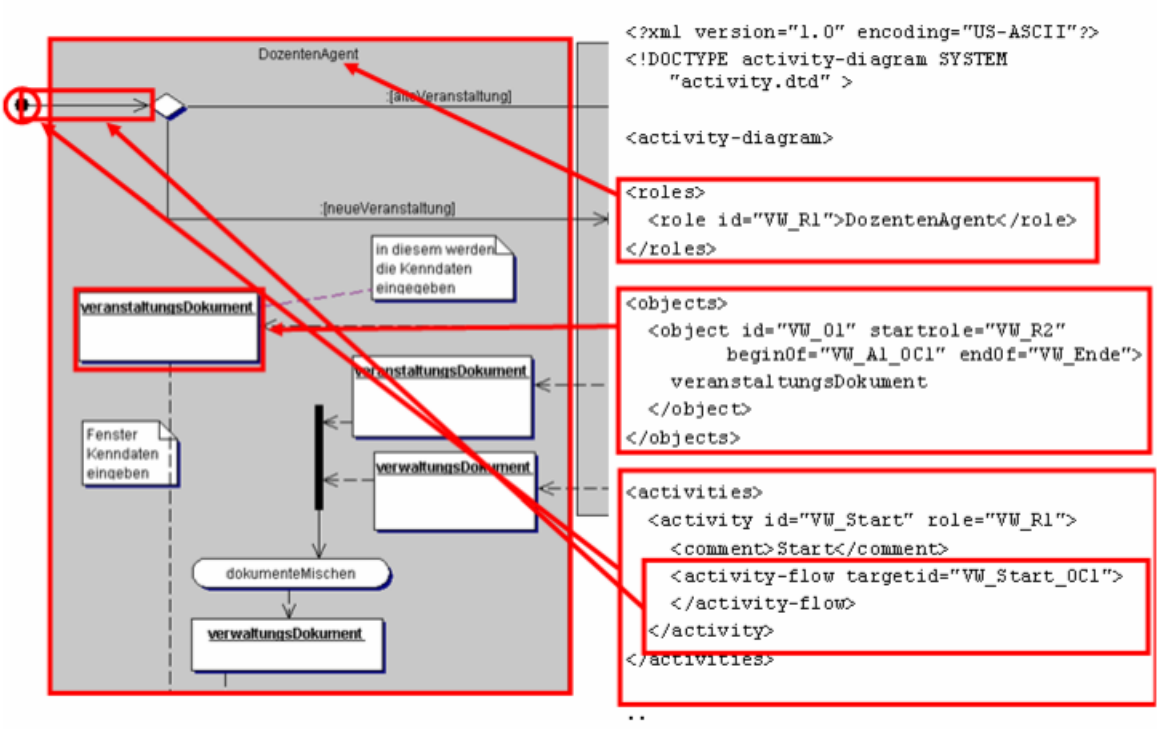

Figure 9.1: Workflow-Aktivitätendiagramm und zugehöriges ACTIVITYML-Dokument

sitionen, dargestellt durch Pfeile, verbunden. Eine Aktivität wird ausgeführt, sobald alle ihre Eingangsaktivitäten abgeschlossen sind. Eine einzelne Aktivität kann selbst für ein komplettes Aktivitätendiagramm stehen, das bei Eintritt in die Aktivität ausgeführt wird. Sobald eine letzte Subaktivität des untergeordneten Diagramms abgearbeitet ist, ist auch der übergeordnete Aktivitätszustand beendet. Weitere Kontrollflusselemente sind die selektive Verzweigung und die Vereinigung von Kontrollflüssen, dargestellt durch Rauten. An Verzweigungen entscheiden boolesche Bedingungen in eckigen Klammern (guards) über den Folgezustand. Nebenläufigkeit wird durch Fork- und Joinzustände mit mehreren aus- bzw. einlaufenden Transitionen durch schwarze Balken notiert. An Fork-Zuständen wird in mehrere Aktivitäten verzweigt, die nebenläufig ausgefürt werden. An Join-Zuständen werden Kontrollflüsse wieder vereinigt und synchronisiert. Weiterhin kann auch der Datenfluss zwischen Aktivitäten durch Objekte dargestellt werden, die mit den Aktivitäten durch gestrichelte Linien verbunden sind, zwischen denen sie fließen. Last but not least können Aktivitäten in sogenannten Bahnen (swimlanes) gruppiert werden. Dadurch können Aktivitäten den ausführenden Komponenten zugeordnet werden.

In UML-Aktivitätendiagrammen sind alle für Workflow-Modelle notwendigen Kontrollflusselemente vorhanden [42]. Sequentielle, verzweigte, nebenläufige und synchronisierte Abläufe können beschrieben werden. Die mögliche hierarchische Strukturierung von Aktivitäten in UML-Aktivitätendiagrammen erfüllt ebenfalls eine Anforderung an die Workflow-Modellierung. Der Fluss von Dokumenten in einem Workflow-System kann

durch Objektflüsse zwischen einzelnen Aktivitäten modelliert werden. Durch Bahnen ist die Möglichkeit gegeben, die Verantwortung für Aktivitäten bestimmten Rollen zuzuordnen.

An einem Beispiel soll gezeigt werden, wie mit UML-Aktiviätendiagrammen Workflows beschrieben werden können. Im linken Teil von Abb. 9.1 ist ein Ausschnitt aus einem Aktivitätendiagramm zu sehen, das Aktivitäten im Rahmen der Vorlesungsverwaltung enthält. Der Ausschnitt zeigt eine Bahn (swimlane), deren Aktivitäten von einem der Bahn zugeordneten Agenten, hier dem DozentenAgenten, ausgeführt werden. Nach Eintritt in den Workflow spaltet sich der Kontrollfluss selektiv auf. Sobald dieser über eine Grenze zwischen zwei Bahnen verläuft, wird der Agent der Zielbahn dazu angestoßen, die entsprechende Zielaktivität auszuführen. Weiterhin sind Objektflüsse für die Objekte Veranstaltungs- und Verwaltungsdokument zu sehen. Durch einen Objektfluss ist immer ein Kontrollfluss impliziert. In Abb. 9.1 ist zu sehen, dass die implizierten Kontrollflüsse für das Ankommen des Veranstaltungsdokuments und des Verwaltungsdokuments an einem Join-Balken synchronisiert werden. Darauf wird die Aktivität dokumenteMischen vom DozentenAgent eingeleitet.

BUSSARD vs. AML. Strukturelle Modelle agentenbasierter Anwendungen sind sowohl in BUSSARD als auch in AML durch Klassendiagramme gegeben. Zu BUSSARD existiert allerdings kein UML-Profil mit einer festgelegten Semantik. Agenten-Klassen werden dort als Klassen aktiver Objekte aufgefasst, bei denen zwischen Operationen, die allein der Agent ausführt, und Nachrichten, die von anderen Agenten empfangen werden können, unterschieden wird. Diese Unterscheidung findet sich konzeptionell ebenso in AML-Klassendiagrammen. Allerdings können dort allein Rollen, die mit Agenten verknüpft sind, Nachrichten empfangen. In BUSSARD sind Rollen quasi in Agenten integriert. Damit zusammenhängend existiert in BUSSARD kein Konzept von Protokollen, durch die strukturierte Interaktion auf einer von konkreten Agenten unabhängigen Basis wie der der Rollen beschrieben werden kann.

Als Verhaltensmodell verwendet BUSSARD Aktivitätendiagramme. In einzelnen Bahnen (swimlanes) wird das lokale Verhalten von Agenten beschrieben. Kontroll- und Objektflüsse über die Grenzen von Bahnen hinweg beschreiben die Interaktion zwischen Agenten. Obwohl sich dieser Ansatz von den auf Graphregeln und Graphtransformationen basierenden AML-Verhaltensmodellen unterscheidet, ist er nicht grundsätzlich verschieden. Aktivitätendiagramme sind im wesentlichen Kontrollstrukturen für Operationen. Zum einen können Operationen durch Graphtransformationen simuliert werden, indem die Veränderungen an einem Datenzustand durch Anwendung einer Operation explizit durch einen Zustand vor und einen Zustand nach Anwendung der Operation beschrieben werden. Die Operation selbst entspricht einer Graphregel. Zum anderen können die Kontrollstrukturen durch Graphregeln simuliert werden. Die Simulation wird kurz skizziert: Jede Aktivität (activity-state) eines Aktivitätendiagramms erhält eine eindeutige Nummer. Alle Agenten-Klassen erhalten ein Attribut activity, in dem für jeden Agenten die Nummer seiner aktuellen Aktivität abgelegt ist. Die Graphregel jeder Opera-

tion wird so erweitert, dass auf der linken Regelseite das Attribut activity mit der Nummer der zur Operation gehörigen Aktivität belegt sein muss. Auf der rechten Regelseite wird das Attribut activity mit der Nummer der nachfolgenden Aktivität belegt. Verzweigungen in Aktivitätendiagrammen können durch entsprechende Anwendungsbedingungen von Graphregeln realisiert werden.

Weiterhin ist in einer Diplomarbeit die Plattform BUSSARD um einen Graphtransformationsansatz erweitert worden [97]. Dabei wurde die Sprache ActivityML (vgl. folgender Abschnitt) um Elemente erweitert, mit denen Graphregeln angegeben werden können. Der Interpreter für die Sprache ActivityML wurde so erweitert, dass von den Graphregeln induzierte Graphtransformationen transaktional nach einem Zwei-Phasen-Sperrprotokoll angewendet werden können.

Nach dem Überblick über die in BUSSARD verwendeten Sprachen und dem Vergleich zwischen BUSSARD- und AML-Modellen werden jetzt die Architektur und die verwendeten Technologien der Plattform vorgestellt, die die Workflow-Modelle ausführt.

9.4 Softwarearchitektur der Agentenplattform BUSSARD

Am Anfang der Systementwicklung waren Entscheidungen zur Architektur und zu den zu verwendenden Sprachen zu treffen. Auf der Grundsatzentscheidung für eine agentenbasierte Architektur aufbauend wurden verschiedene Agentenplattformen, wie beispielsweise IBM Aglets [80] und IKV Grasshopper [8] evaluiert. Da diese Plattformen sehr stark auf Mobilität von Agenten ausgerichtet waren und diese Agenteneigenschaft von eher untergeordnetem Interesse war, wurde eine eigene Implementierung bevorzugt.

Da eine verteilte Internet-basierte Plattform entwickelt werden sollte, wurde als Programmiersprache Java ausgewählt. Für Java existiert die Laufzeitbibliothek J2EE, die Internet-basierte Server-Anwendungen unterstützt. Weiterhin wurde festgelegt, die eXtensible Markup Language (XML) zu verwenden.

9.4.1 Die Rolle von XML

Die Vorteile von XML konnten auf dreierlei Weise eingebracht werden: zum Kodieren von Nachrichten, die Agenten untereinander austauschen, als Format für die Workflows, die von den Agenten interpretiert werden und zur Repräsentation der von den Agenten verarbeiteten Dokumente im System.

Nachrichten in XML zu kodieren hat den Vorteil, dass in die Nachrichten XML-Dokumente als Daten aufgenommen werden können. Das ist deshalb wichtig, da sowohl Workflow-Beschreibungen als auch durch die Workflows verarbeitete Dokumente zwischen den Agenten ausgetauscht werden. Ein ähnlicher Ansatz wird mit der XML-Sprache SOAP verfolgt [94].

Workflow-Beschreibungen sind XML-Dokumente in der Sprache ActivityML. Hier wird die Möglichkeit von XML genutzt, neue spezifische Formate bzw. Sprachen definieren

zu können und die zugehörigen Dokumente durch einen Parser auf Gültigkeit prüfen zu können. Dokumente im XML-Format zwischen den Agenten auszutauschen hat den Vorteil, dass einzelne Agenten mit Daten in einer homogenen Struktur umgehen können. Schliesslich wurde auch die Möglichkeit genutzt, XML-Dokumente mittels XSL-Stylesheets in HTML-Dokumente zu transformieren und so per Web-Browser zu präsentieren.

9.4.2 Die Sprache ActivityML

In der XML-Sprache ActivityML sind die Workflow-Modelle beschrieben. Die Sprache ist in der Art entworfen, dass Workflow-Aktivitätendiagramme in ihr repräsentiert werden können. An dem Ausschnitt eines Workflow-Aktivitätendiagramms und des zugehörigen ActivityML-Dokuments in Abbildung 9.1 wird die Transformation deutlich. Das Element `role` ist einem Agenten zugeordnet, der Aktivitäten einer Bahn (swimlane) ausführt. Das Element `object` repräsentiert ein zwischen zwei Aktivitäten übertragenes Dokument. Für das Element `activity` wird seine Bahn über einen Verweis auf ein `role`-Element und seine ausgehenden Transitionen als Folge von `activity-flow`-Elementen gespeichert.

Ausgehend von dem eben dargestellten Beispiel soll anhand der DTD der Sprache ActivityML gezeigt werden, wie die Abbildung zwischen Workflow-Aktivitätendiagrammen und ActivityML-Dokumenten allgemein erfolgt.

```
<!-- DTD der XML-Sprache ActivityML ->
<!ELEMENT activity-diagram (diagramname, comment?, start, activities,
                        decisions?, objects?, roles?, synchronizations?)>
<!ELEMENT diagramname   (#PCDATA)>
<!ELEMENT comment       (#PCDATA)>
<!ELEMENT start         (activity-flow)>
<!ATTLIST start
          role          IDREF   #REQUIRED>
<!ELEMENT activities    (activity+)>
<!ELEMENT activity      (comment?, name?, activity-flow)>
<!ATTLIST activity
          id            ID      #REQUIRED
          role          IDREF   #REQUIRED
          sub-activity  CDATA   #IMPLIED>
<!ELEMENT decisions     (decision+)>
<!ELEMENT decision      (comment?, activity-flow+)>
<!ATTLIST decision
          id            ID      #REQUIRED
          role          IDREF   #REQUIRED>
<!ELEMENT objects       (object+) >
<!ELEMENT object        (#PCDATA) >
<!ATTLIST object
```

```
          id            ID      #REQUIRED
          startrole     IDREF   #REQUIRED
          beginOf       IDREF   #REQUIRED
          endOf         IDREF   #REQUIRED>
<!ELEMENT roles         (role+)>
<!ELEMENT role          (name)>
<!ATTLIST role
          id            ID      #REQUIRED>
<!ELEMENT synchronizations      (synchronization+)>
<!ELEMENT synchronization       (comment?, (activity-flow)+)>
<!ATTLIST synchronization
          id            ID      #REQUIRED
          role          IDREF   #REQUIRED>
<!ELEMENT activity-flow (guard-condition?, object-item*)>
<!ATTLIST activity-flow
          target-id     IDREF   #REQUIRED>
<!ELEMENT object-item   EMPTY>
<!ATTLIST object-item
          id            IDREF   #REQUIRED>
<!ELEMENT guard-condition       (#PCDATA)>
<!ELEMENT name          (#PCDATA)>
```

Das Wurzelelement `activity-diagram` schachtelt neben seinem Namen die Startaktivität `start` und Folgen der Elemente `activity`, `decision`, `object`, `role` und `synchronization`. Durch das Element `activity` werden Aktivitäten repräsentiert. Das Unterelement `name` enthält den Namen einer auszuführenden Methode des Agenten, der der Aktivität zugeordnet ist. Das Element `activity-flow` repräsentiert eine Transition und verweist auf die nachfolgenden Knoten im Workflow, d.h. eine `activity`, `decision` oder `synchronization`. Stimmt das Attribut `role` der Zielaktivität nicht mit dem Attribut `role` der aktuellen Aktivität überein, dann delegiert der Agent diese Aktivität an einen passenden Agenten. Schliesslich enthält das Attribut `sub-activity` den Dateinamen eines ActivityML-Dokumentes, das den untergeordneten Workflow repräsentiert.

Das Element `decision` steuert die Verzweigung zu mehreren alternativen Aktivitäten. Die zugeordneten `activity-flow`-Elemente verweisen auf diese Alternativen und besitzen in diesem Fall `guard-condition`-Elemente. Das Element `role` verweist auf denjenigen Agenten, der die Bedingungen prüft.

Im Element `guard-condition` ist der Name einer Methode gespeichert. Dieses Element soll nur mit einem Transitionselement `activity-flow` eines `decision`-Elements verbunden sein. Die enthaltene Zeichenkette ist der Name einer Methode, die eine Bedingung im Zusammenhang mit dem `decision`-Element testet.

Das Element `synchronization` ermöglicht die Verzweigung des Aktivitätenflusses zur parallelen Bearbeitung von Aktivitäten durch mehrere Agenten (`fork`) und die spätere Zusammenführung der Kontrollflüsse durch ein weiteres `synchronization`-

Element (`join`). Das Unterelement `activity-flow` gibt die Ziele für den Fall fork an. Das Attribut `role` benennt den Agenten, der im Fall `fork` per Nachrichtenversand die Teilaufgaben an andere Agenten bzw. an Klone vergibt.

Nach diesem Überblick über den Zusammenhang zwischen Workflow-Aktivitätendiagrammen und ActivityML-Dokumenten stellt sich die Frage, wie die Plattform strukturiert ist, innerhalb derer die ActivityML-Dokumente abgearbeitet werden.

9.4.3 Die Architektur des WFMS

Die Internet-Agentenplattform baut auf drei Klassen von Agenten auf: Benutzeragenten, Agenten für Koordination und Agenten zur Ressourcenkapselung. In Abbildung 9.2 sind Dozenten- und Studentenagenten spezielle Benutzeragenten (UserAgent). Agenten dieser Klasse sind in der Lage, über ein Servlet mit dem Benutzer, für den der Agent Stellvertreter im System ist, zu kommunizieren. Manageragenten gehören zur Klasse der für Koordination zuständigen Agenten. Ein Manageragent initiiert Workflows in Form von ActivityML-Dokumenten und koordiniert die Abarbeitung dahingehend, dass den Bahnen (swimlanes) im Workflow konkrete Agenten für die Abarbeitung der zugehörigen Aktivitäten zugewiesen werden. Die verfügbaren Ressourcen werden durch Datenbank-Agenten und PublisherAgenten (siehe Abb. 9.2) gekapselt. Für neue Ressourcen können weitere Agententypen integriert werden. Durch die Kapselung wird erreicht, dass die Kommunikation zwischen allen Agenten in Form von XML-Dokumenten erfolgen kann.

Die verschiedenen Agenten wirken wie folgt zusammen. Der Benutzer kommuniziert mit der Plattform, indem er auf einer Webseite Eingaben tätigt, d.h. eine Anfrage (technisch: per HTTP-Request) an ein Servlet sendet, die in Form einer Nachricht an den entsprechenden Benutzeragenten weitergeleitet wird. Dieser kommuniziert entsprechend dem Workflow, in den er eingebunden ist, mit Ressourcenagenten wie dem Datenbank-Agent und dem PublisherAgent oder mit anderen Benutzeragenten, die in den Workflow involviert sind. Danach wird eine Nachricht im XML-Format mit Ergebnissen an das Servlet zurückgeschickt. Das Servlet wendet auf das enthaltene XML-Ergebnisdokument ein XSL-Stylesheet zur Produktion eines HTML-Dokuments an und schickt es an den Web-Browser zurück, wodurch der Benutzer das Ergebnis seiner Interaktion (technisch HTTP-Request) erhält.

Alle Agenten benötigen Grundfunktionen zur Kommunikation und Steuerung. Agenten sind als eine Java-Klasse implementiert und erben von einer abstrakten Klasse ActivityAgent, die Funktionen zur Kommunikation und Interpretation von ActivityML-Dokumenten bietet. Zur Kommunikation dienen XML-Nachrichtendokumente, die ein spezielles Element für den Typ der Nachricht wie zum Beispiel Speichern und beliebige XML-Dokumente als Parameter enthalten. Die integrierten Dokumente dienem dem Datenaustausch. Dabei handelt es sich nicht nur um Anwendungsdaten sondern auch um ActivityML-Dokumente, die Agenten zwecks Abarbeitung zugestellt werden können. Zur Verarbeitung der Workflows in Form von ActivityML-Dokumenten ist in jedem Agenten ein ActivityML-Interpreter implementiert. Einzelne elementare Aktivitäten (ohne

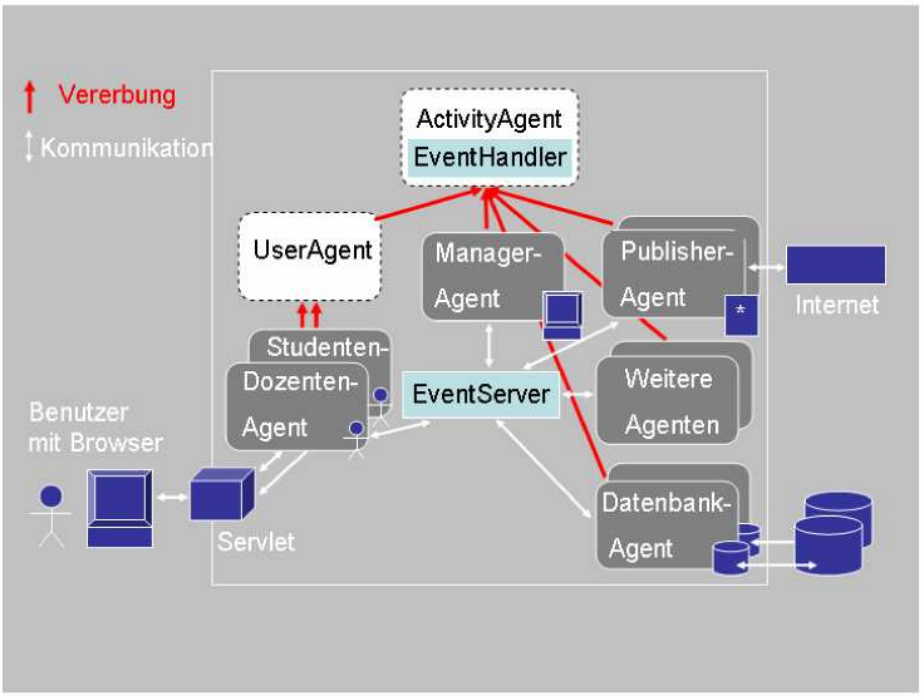

Figure 9.2: Architektur der Internet-Agentenplattform

Subaktivitäten) werden durch den Aufruf von lokalen Methoden implementiert. Eine Aufteilung des Kontrollflusses (`fork`) wird durch Klonen von zu den Nachfolgeaktivitäten passenden Agenten erreicht. Die Synchronisation und Vereinigung von Kontrollflüssen (`join`) erfordert das Warten eines Agenten auf passende Nachrichten anderer Agenten der zu synchronisierenden Kontrollflüsse. Entscheidungen (`guards`) an Verzweigungspunkten des Kontrollflusses werden mittels Methoden mit boolschen Rückgabewerten getroffen.

Die Kommunikation zwischen Agenten wird über EventHandler und EventServer abgehandelt. Der Activity-Agent und damit jeder Agent im System enthält einen EventHandler, der mit einem EventServer kommuniziert. Der EventServer nimmt Nachrichten von den EventHandlern der Agenten entgegen, puffert diese und leitet sie an einen bestimmten oder auch mehrere Agenten weiter.

Skalierbarkeit wird dadurch erreicht, dass sich für Events im Rahmen der Systemressourcen beliebig viele Agenten registrieren können und dann von einem Manageragenten für die Abarbeitung von Workflows ausgewählt werden können. Durch Erzeugen von Agenten auf schwach ausgelasteten Knoten eines Netzwerks kann Lastverteilung erreicht werden. Eine konkrete Strategie wurde allerdings (noch) nicht implementiert.

Zur Spezialisierung des Activity-Agent werden weitere Methoden implementiert, die bei Nachrichten eines spezifischen Typs ausgeführt werden. Zum Beispiel reagiert ein Datenbankagent auf die Nachricht, etwas zu speichern, mit der Ablage eines als Parameter an die Nachricht gehängten XML-Dokuments in einer Datenbank.

Damit ist der Überblick über die Internet-Agentenplattform BUSSARD abgeschlossen. Detaillierte Informationen finden sich unter [90].

9.5 Fazit

Im Anwendungsbereich der Workflow-Managementsysteme ist anhand der agentenbasierten Plattform BUSSARD gezeigt worden, dass Agenten zur Realisierung von Workflow-gesteuerten Informationssystemen geeignet sind. Die Struktur und das Verhalten der Agenten wird in BUSSARD durch UML-Klassen- und Aktivitätendiagramme modelliert. Diese UML-Modelle bilden zusammen bezüglich der Ausführung vollständige Spezifikationen und werden in die XML-Sprache ACTIVITYML abgebildet. ACTIVITYML-Dokumente werden durch die Plattform BUSSARD direkt ausgeführt. Insgesamt ist so gezeigt worden, dass eine Implementierung agentenbasierter Systeme unmittelbar aus Entwurfsmodellen abgeleitet werden kann.

Es wurde diskutiert, dass die den Struktur- und Verhaltensmodellen zugrunde liegenden Konzepte von BUSSARD und AML zu einem beträchtlichen Teil übereinstimmen. Insofern kann trotz der syntaktischen Unterschiede gefolgert werden, dass BUSSARD einen Ansatz darstellt, in dessen Rahmen auch AML-Modelle implementiert werden können.

Chapter 10

Abschluss

Eine Zusammenfassung der Ergebnisse und ein kurzer Ausblick auf die weitere Entwicklung im Bereich agentenbasierter Systeme schließen diese Arbeit ab.

10.1 Ergebnisse

Diese Arbeit beschreibt einen neuen Ansatz zur visuellen Modellierung agentenbasierter Systeme. Aufbauend auf den Agenten-Kernkonzepten Proaktivität, Autonomie und strukturierte Interaktion sind ein abstraktes Modell agentenbasierter Systeme, ein verfeinerndes Automatenmodell, die UML-basierte Sprache AML, ein auf Graphtransformation basierender semantischer Bereich, ein auf Model Checking gestützter Ansatz zur Modellüberprüfung, das Prozessmodell APM und eine Ausführungsplattform für Agenten entwickelt worden.

Im einzelnen sind folgende Ergebnisse erzielt worden:

- Als Kernkonzepte von Agenten wurden Autonomie, strukturierte Interaktion und Proaktivität identifiziert. Ein neues abstraktes Modell für Multiagentensysteme erfasst die Kernkonzepte präzise und formal. Das abstrakte Modell wurde wegen der geringen Ausdrucksmittel durch ein Automatenmodell verfeinert, das auf den für verteilte Systeme verwendeten I/O-Automaten basiert. Einzelne Agenten sind als spezielle I/O-Automaten modelliert, die durch Produktbildung zu einem Systemautomaten für ein Multiagentesystem zusammengesetzt werden. Für wichtige Agentenarchitekturen wie die BDI-Architektur und die Schichtenarchitektur wurde demonstriert, dass sie sich in das neue Agentenmodell einordnen lassen.

- Die Sprache AML zur agentenorientierten Modellierung wurde definiert. Für die Konzepte Proaktivität, Autonomie und strukturierte Interaktion wurden Konstituenten wie Ziel, Strategie, Ausführungsfaden, Kontrollelement und Protokoll eingeführt. Diese Konstituenten wurden durch spezifische Sprachkonstrukte in AML repräsentiert. Die Syntax der Sprache AML wurde durch ein UML-Profil definiert.

Randbedingungen für die Verwendung der Konstituenten finden sich in Bedingungen an einzelne AML-Modellelemente wieder.

- Der semantische Bereich der Graphtransformation hat sich als geeignet für die Definition der Semantik von AML-Modellen erwiesen. Neu eingeführt wurde das Konzept des Metatypgraph zur Einschränkung des allgemeinen semantischen Bereichs auf einen spezifisch agentenorientierten Bereich. Die Semantik von AML wurde durch eine Abbildung auf Elemente des Metatypgraphen definiert. Durch die Abbildung von Modell-Graphtransformationssystemen in eine Menge von I/O-Automaten wurde gezeigt, dass AML-Modelle mit dem abstrakten Modell agentenbasierter Systeme verträglich sind.

- Auf der Basis von Model Checking wurde der Zusammenhang zwischen Zielen und Strategien von Agenten behandelt, d.h. eine Strategie wird daraufhin überprüft, ob sie ein gegebenes Ziel erreicht. Strategien wurden in Spezifikationen eines Transitionssystems in der Eingabesprache PROMELA des Model Checkers SPIN übersetzt. Gezeigt wurde, dass eine Zielregel in eine gleichwertige temporal-logische Formel übersetzt werden kann. Damit konnte die Erreichbarkeit des Ziels durch Überprüfen der Spezifikation bezüglich der temporal-logischen Formel in SPIN entschieden werden.

- In Anlehnung an etablierte objektorientierte Prozessmodelle wurde ein agentenorientiertes Prozessmodell (APM) entwickelt, das die Verwendung (Pragmatik) der Sprache AML beschreibt. Dazu gehören die genaue Beschreibung der produzierten Artefakte und des Zusammenhangs der Artefakte. Das Verfahren zur Modellüberprüfung wurde in APM verwendet.

- Anhand der agentenbasierten Plattform BUSSARD ist gezeigt worden, dass eine Implementierung eines agentenbasierten Systems aus einem (Entwurfs-)Modell abgeleitet werden kann.

Der beschriebene Ansatz zur visuellen Modellierung agentenbasierter Systeme zeichnet sich dadurch aus, dass seine Grundlagen klar abgegrenzt und fundiert sind und dass die darauf aufbauende Sprache hinsichtlich ihrer Syntax, Semantik und Pragmatik präzise und zusammenhängend beschrieben wurden.

10.2 Ausblick

Der beschriebene Ansatz kann in verschiedener Hinsicht erweitert werden. Durch weitere Formalisierung des Prozessmodells APM hinsichtlich der Transformation von Modellen können die Anforderungen an ein Entwicklungswerkzeug noch genauer gefasst werden. Ein solches Werkzeug, das die Verwendung des Prozessmodells APM unterstützt, würde auch zur mit APM verknüpften Modellüberprüfung eingesetzt. Als Konsequenz ist ein Model Checker an das Entwicklungswerkzeug anzubinden.

Generell stellt sich die Frage nach der Relevanz von Software-Agenten. Die identifizierten Kernkonzepte sind fundamental, und ihr Nutzen im Zusammenhang mit der Entwicklung offener, robuster und anpassbarer Systeme ist grundsätzlich evident. Umso deutlicher stellt sich die Frage, warum agentenbasierte Systeme nur langsam Verbreitung finden. Zum einen fehlte die softwaretechnische Unterstützung für die Entwicklung derartiger Systeme. Diesen Mangel hilft diese Arbeit zu lindern. Zum anderen wird der Begriff *Agent* vielfältig interpretiert. Merkmale wie Mobilität oder künstliche Intelligenz werden Agenten attributiert. Die unterschiedlichen und zum Teil unklaren Interpretationen haben dazu geführt, dass lediglich Teilkonzepte von Agenten Eingang in neue Technologien gefunden haben. Diese Entwicklung wird sich fortsetzen. Beispielsweise bietet das Konzept der Proaktivität Potential dafür, die Anforderungen an ein System als Ziele und Strategien in dedizierten Software-Bausteinen zu lokalisieren. Der Begriff *Agent* selbst wird sich dabei vermutlich nur eingeschränkt durchsetzen.

Bibliography

[1] Wil Van Der Aalst and Kees Van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, Cambridge, MA, USA, 2004.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, 1986.

[4] R. Alur, L. de Alfaro, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Kirsch, and B. Wang. Mocha: A model checking tool that exploits design structure. In *Proc. 23rd Intl. Conference on Software Engineering (ICSE'2001)*, pages 835–836, Los Alamitos, CA, 2001. IEEE Computer Society Press.

[5] P. Arnold, D. Coleman, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The FUSION Method*. Prentice Hall International, 1994.

[6] P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Concurrent semantics of algebraic graph transformation. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*, pages 107 – 188. World Scientific, 1999.

[7] B. Bauer, J.P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In P. Ciancarini and M.J. Wooldridge, editors, *Proc. 1st Int. Workshop on Agent-Oriented Software Engineering (AOSE 2000), Limerick, Ireland, June 2000*, volume 1957 of *LNCS*, pages 91–104. Springer-Verlag, Berlin, 2001.

[8] C. Bäumer, M. Breugst, S. Choy, and T. Magedanz. Grasshopper — A universal agent platform based on OMG MASIF and FIPA standards. In Ahmed Karmouch and Roger Impley, editors, *First International Workshop on Mobile Agents for Telecommunication Applications (MATA'99)*, pages 1–18, Ottawa, Canada, October 1999. World Scientific Publishing Ltd.

[9] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1–3):109 – 137, 1984.

[10] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[11] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, 1994.

[12] S. Brodsky, T. Clark, S. Cook, A. Evans, and S. Kent. Feasibility study in rearchitecting UML as a family of languages using a precise oo meta-modeling approach, University of York, Department of Computer Science, 2000. `http://www.cs.york.ac.uk/puml/mmf/mmf.pdf`.

[13] F. P. Brooks. *The Mythical Man-Month: Essays in Software Engineering.* Addison-Wesley Publishing Company, 20th anniversary edition, 1995.

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

[15] Rodney A. Brooks. Intelligence without reason. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, August 1991. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[16] B. Burmeister. Models and methodology for agent-oriented analysis and design. In Klaus Fischer, editor, *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems.* DFKI Document D-96-06, 1996.

[17] Tony Clark, Andy Evans, and Stuart Kent. The metamodelling language calculus: Foundation semantics for UML. In Heinrich Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *LNCS*, pages 17–31. Springer-Verlag, 2001.

[18] E. M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001.

[19] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[20] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. In *Proc. 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'99)*, pages 325–339. ACM Press, 1999.

[21] I. Claßen and M. Löwe. Scheme evolution in object–oriented models. In M. Wirsing, editor, *ICSE–17 Workshop on Formal Methods Application in Software Engineering Practice*, Seattle, Washington, April 1995.

[22] Workflow Management Coalition. The Workflow Reference Model (Version 1.1), 1995. `http://www.wfmc.org/standards/docs/tc003v11.pdf`.

[23] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.

[24] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter 3, pages 163–246. World Scientific, 1997.

[25] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 3rd edition, 2000.

[26] Ewa Deelman and Carl Kesselman. Grid computing. *Scientific Programming*, 10(2):101–102, 2002.

[27] S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3), 2001.

[28] R. Depke, G. Engels, and J. M. Küster. On the integration of roles in the UML. Technical Report tr-ri-01-214, University of Paderborn, August 2000.

[29] R. Depke, J.H. Hausmann, and R. Heckel. Design of an agent-based modeling language based on graph transformation. In M. Nagl and J. Pfaltz, editors, *2nd Int. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003), Charlottesville, VA, USA, Sep./Oct. 2003. Revised selected and invited papers*. Vol. 3062 of LNCS. Springer-Verlag, Berlin, 2004.

[30] R. Depke and R. Heckel. Formalizing the development of agent-based systems using graph processes. In *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques*, pages 419–426. Carleton Scientific, 2000. Waterloo, Ontario, Canada 2000.

[31] R. Depke and R. Heckel. Modellierung von Prozessen mit UML und Realisierung durch eine Internet-Agentenplattform. In *Tagungsband 9. Kolloquium Software-Entwicklung für Internet und Intranet*, pages 112–124, Ostfildern, Deutschland, September 2001. Technische Akademie Esslingen.

[32] R. Depke and R. Heckel. Modeling and analysis of agents' goal-driven behavior using graph transformation. In H. D. Ehrich, J. J. Meyer, and M. D. Ryan, editors,

Objects, Agents and Features - Structuring Mechanisms for Contemporary Software, Revised Papers. Int. Seminar, Dagstuhl Castle, Germany, February 2003. Vol. 2975 of LNCS. Springer-Verlag, Berlin, 2004.

[33] R. Depke, R. Heckel, and J. M. Küster. Integrating visual modeling of agent-based and object-oriented systems. In *Proc. Fourth Int. Conference on Autonomous Agents (AGENTS-2000)*, pages 82–83, Barcelona, Spain, June 2000. ACM Press.

[34] R. Depke, R. Heckel, and J. M. Küster. Modeling agent-based systems with graph transformation and UML: From requirement specification to object-oriented design. In H. Ehrig and G. Taentzer, editors, *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra'2000), Berlin, Germany*, March 2000.

[35] R. Depke, R. Heckel, and J. M. Küster. Agent-oriented modeling with graph transformation. In P. Ciancarini and M. J. Wooldridge, editors, *Proc. 1st Int. Workshop on Agent-Oriented Software Engineering (AOSE 2000), Limerick, Ireland, June 2000*, volume 1957 of *LNCS*, pages 105–120. Springer-Verlag, Berlin, 2001.

[36] R. Depke, R. Heckel, and J. M. Küster. Improving the agent-oriented modeling process with roles. In *Proc. Fifth Int. Conference on Autonomous Agents (AGENTS-2001)*, Montreal, Canada, June 2001. ACM Press.

[37] R. Depke, R. Heckel, and J. M. Küster. Roles in agent-oriented modeling. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3):281–302, 2001.

[38] R. Depke, R. Heckel, and J. M. Küster. Formal agent-oriented modeling with graph transformation. *Science of Computer Programming*, 44(2):229–252, 2002.

[39] R. Depke and K. Mehner. "Separation of Concerns" mit Rollen, Subjekten und Aspekten. In K. Mehner, M. Mezini, E. Pulvermüller, and A. Speck, editors, *Proc. Aspektorientierung - Workshop der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung*, pages 1–7, Paderborn, Germany, May 2001. Technical Report tr-ri-01-223, University of Paderborn.

[40] W. Dröschel and M. Wiemers. *Das V-Modell 97.* Oldenbourg-Verlag, 1999.

[41] D. F. D'Souza and A. C. Wills. *Catalysis: Component and Framework-Based Development.* Addison-Wesley Publishing Company, 1997.

[42] Marlon Dumas and Arthur H. M. ter Hofstede. UML activity diagrams as a workflow specification language. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 76–90. Springer-Verlag, 2001.

[43] G. Weiss (ed.). *Multiagent Systems.* MIT Press, Cambridge, MA, 1999.

[44] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications 1: Equations and initial semantics*, volume 6 of *EACTS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[45] H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[46] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000, York, UK*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer-Verlag, 2000.

[47] Gregor Engels and Luuk Groenewegen. Object-oriented modeling: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE-00): The Future of Software Engineering*, pages 103–116, Limerick, Ireland, June 4–11 2000. ACM Press.

[48] A. Evans and S. Kent. Core meta modelling semantics of UML: The pUML approach. In R. France and B. Rumpe, editors, *Proc. UML'99*, volume 1723 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, September 1999.

[49] M. Fisher. Concurrent METATEM – A language for modeling reactive systems. In *Proceedings of Parallel Architectures and Languages Europe (PARLE)*, Lecture Notes in Computer Science, vol 694. Springer-Verlag: Heidelberg, Germany, June 1993.

[50] Foundation for Intelligent Physical Agents (FIPA). Agent communication language. In *FIPA 97 Specification, Version 2.0,* `http://www.fipa.org`. FIPA, 1997.

[51] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Proc. ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193 of *LNAI*, pages 21–36. Springer-Verlag, August 12–13 1997.

[52] D. Fudenberg and J. Tirole. *Game Theory.* The MIT Press, Cambridge, Massachusetts, 1991.

[53] Alonso Fugetta. Software process: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE-00): The Future of Software Engineering*, pages 25–34, Limerick, Ireland, June 4–11 2000. ACM Press.

[54] David Garlan. Software architecture: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE-00): The Future of Software Engineering*, pages 91–102, Limerick, Ireland, June 4–11 2000. ACM Press.

[55] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, 1991.

[56] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[57] H. Giese, S. Burmester, F. Klein, D. Schilling, and M. Tichy. Multi-agent system design for safety-critical self-optimizing mechatronic systems with uml. In B Henderson-Sellers and J Debenham, editors, *OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies*, pages 21–32, Anaheim, CA, USA, Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia, 2003.

[58] F. Giunchiglia, J. Mylopoulos, and A. Perini. The TROPOS software development methodology: processes, models and diagrams. In *Proc. of the 1st Int. Conference on Autonomous Agents and Multiagent Systems*, pages 35–36. ACM Press, 2002.

[59] Martin Gogolla, Paul Ziemann, and Sabine Kuske. Towards an Integrated Graph Based Semantics for UML. In Paolo Bottoni and Mark Minas, editors, *Proc. ICGT Workshop Graph Transformation and Visual Modeling Techniques (GT-VMT'2002).* Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier (October 2002), 2002.

[60] Georg Gottlob, Michael Schrefl, and Brigitte Rock. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[61] A. Heuer and G. Saake. *Datenbanken – Konzepte und Sprachen, 2., aktualisierte und erweiterte Auflage.* MITP-Verlag, Bonn, 2000.

[62] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, 1991.

[63] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.

[64] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - summary of an agent infrastructure. In *Proc. Fifth Int. Conference on Autonomous Agents (AGENTS-2001)*, pages 443–448, Montreal, Canada, June 2001. ACM Press.

[65] W. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.

[66] M. N. Huhns and M. P. Singh. Managing heterogeneous transaction workflows with co-operating agents. In Nicholas R. Jennings and Michael J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, pages 219–239. Springer-Verlag: Heidelberg, Germany, 1998.

[67] C. A. Iglesias, M. Garijo, J. C. González, and J. R. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Proc. 4th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, volume 1365 of *LNAI*, pages 313–328. Springer-Verlag, 1998.

[68] Carlos A. Iglesias and Mercedes Garijo. UER technique: Conceptualisation for agent oriented development. In *Proceedings of the 3rd World Multiconference on Systemics, Cybernetics and Informatics (SCI'99) and 5th International Conference on Information Systems Analysis and Synthesis (ISAS'99)*, Orlando (USA), July 1999.

[69] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley, 1999.

[70] I. Jacobson, M. Christerson, P. Jonsson, and G. G. Overgaard. *Object-Oriented Software Engineering, A Use Case Driven Approach.* Addison-Wesley, Reading, Massachusets, 1992.

[71] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[72] Nicholas R. Jennings and Michael J. Wooldridge, editors. *Agent Technology: Foundations, Applications, and Markets.* Springer-Verlag: Heidelberg, Germany, 1998.

[73] N.R. Jennings. Lost wax agent framework, September 2004. `http://www.lostwax.com/agents/framework/`.

[74] Gregor Joeris. Decentralized and flexible workflow enactment based on task coordination agents. In *Proc. of the 2nd Int'l. Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2000 @ CAiSE*00)*, pages 41–62, Stockholm, Sweden, 2000.

[75] Alan Curtis Kay. *The reactive engine.* PhD thesis, Dept. of Electrical Engineering, Computer Science, University of Utah, 1969.

[76] E. Kindler. *Model checking.* Materialien zur Vorlesung im Wintersemester 2003/4, Universität Paderborn, 2003. `http://wwwcs.upb.de/cs/kindler/Lehre/WS03/MC/index.html`.

[77] F. Klein and H. Giese. Separation of concerns for mechatronic multi-agent systems through dynamic communities. In Ricardo Choren, Alessandro Garcia, Carlos

Lucena, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications*, Lecture Notes in Computer Science. Springer-Verlag, December 2004.

[78] Bent B. Kristensen. Object Oriented Modeling with Roles. In *Proceedings of the 2nd International Conference on Object-Oriented Information Systems (OOIS'95), Dublin, Ireland, 1995*, pages 57–71, London, 1996. Springer-Verlag.

[79] J.M. Küster. Visual modeling of agent-based systems: A role-oriented approach using the UML. diploma thesis, University of Paderborn, Germany, 2000.

[80] Danny B. Lange and Mitsuru Oshima. *Programming and deploying Java mobile agents with Aglets.* Addison-Wesley, Reading, MA, USA, 1999.

[81] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep, M. J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[82] Stephan Merz. Model checking: A tutorial overview. *Lecture Notes in Computer Science*, vol. 2067, Springer-Verlag, Berlin, 2001.

[83] B. Meyer. *Object-Oriented Software Construction, Second Edition.* The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[84] J. Mylopoulos, M. Kolp, and J. Castro. UML for agent-oriented software development: The tropos proposal. *Lecture Notes in Computer Science*, 2185:422–438, 2001.

[85] Object Management Group. Meta object facility (MOF) specification, September 1999. `http://www.omg.org`.

[86] Object Management Group. *OMG Unified Modeling Language Specification Version 1.5*, March 2003. `http://www.omg.org`.

[87] D. L. Parnas. *Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs*, volume 23 of *Lecture Notes in Computer Science*. Springer-Verlag, 1975.

[88] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[89] Stefan Poslad and Patricia Charlton. Standardizing agent interoperability: The FIPA approach. *Lecture Notes in Computer Science*, 2086:98–113, 2001.

[90] Projektgruppe ABS. Dokumentation zum Projekt "Entwicklung agentenbasierter Systeme mit UML und Java", Universität Paderborn, 2000. `http://wwwcs.upb.de/cs/ag-engels/ag_dt/Courses/Lehrveranstaltungen/WS9900/ABS/index.html`.

[91] G. Reggio and E. Astesiano. An extension of UML for modelling the nonpurely-reactive behavior of active objects. In H. Ehrig, G. Engels, F. Orejas, and M. Wirsing, editors, *Dagstuhl Seminar No. 00411 on "Semi-Formal and Formal Specification Techniques for Software Systems"*, number 288 in Dagstuhl Seminar Series, October 2000.

[92] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations.* World Scientific, 1997.

[93] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–oriented Modelling and Design.* Prentice–Hall, 1991.

[94] S. Seely and K. Sharkey. *SOAP: Cross Platform Internet Development Using XML.* Prentice Hall PTR, Upper Saddle River, New Jersey, 2001.

[95] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *IEEE Transaction on Computers*, number 12 in C-29, pages 1104–1113, 1980.

[96] Ian Sommerville. *Software Engineering.* Addison-Wesley, 6th edition, 2001.

[97] A. Stiewe. Ein Beitrag zur Integration agentenspezifischen Verhaltens in ein Informationssystem. Diplomarbeit, Universität Paderborn, 2002.

[98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 1998.

[99] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 3(2):85–113, May 2003.

[100] Dániel Varró and András Pataricza. Metamodeling mathematics: A precise and visual framework for describing semantics domains of UML models. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, volume 2460 of *LNCS*, pages 18–33, Dresden, Germany, September 30 – October 4 2002. Springer-Verlag.

[101] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, 1997.

[102] Peter Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, 20 February 1998.

[103] G. Weiß. Agentenorientiertes Software Engineering. *Informatik Spektrum*, 24(2):98–101, 2001.

[104] M. Wood and S. A. DeLoach. An Overview of the Multiagent Systems Engineering Methodology. In P. Ciancarini and M.J. Wooldridge, editors, *Proc. 1st Int. Workshop on Agent-Oriented Software Engineering (AOSE 2000), Limerick, Ireland, June 2000*, volume 1957 of *LNCS*, pages 207–222. Springer-Verlag, Berlin, 2001.

[105] M. Wooldridge. Agent-based software engineering. In *IEEE Proceedings on Software Engineering*, volume 144, pages 23–37, October 1997.

[106] M. J. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[107] Michael Wooldridge, Michael Fisher, Marc-Philippe Huget, and Simon Parsons. Model checking multi-agent systems with MABLE. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 952–959. ACM Press, July 2002.

[108] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational Abstractions for the Analysis and Design of Multi-Agent Systems. In P. Ciancarini and M.J. Wooldridge, editors, *Proc. 1st Int. Workshop on Agent-Oriented Software Engineering (AOSE 2000), Limerick, Ireland, June 2000*, volume 1957 of *LNCS*, pages 235–252. Springer-Verlag, Berlin, 2001.

Index

Appendix A

PROMELA-Spezifikation für das Contract-Net-Protokoll

```
/* Promela-specification derived from the graph transformation system
   for the contract net protocol */


/* Definition of constants, e.g., for enumerations */
#define noOfContractors 3
#define noOfProposals noOfContractors
#define sel1 0   /* Index=Ids for selectors */
#define con1 0   /* Index=Ids for contractors */
#define con2 1
#define con3 2
#define cfp1 0   /* Id of CFP */
#define cfp2 1
#define cfp3 2
#define prop1 0  /* Index=Ids for proposals */
#define prop2 1
#define prop3 2
#define CNPstart 0
#define CNPend 1
#define ind(x,y) 3*x+y



/* Declaration of State Variable Arrays */
```

```
/* classes */
bool cfp[noOfContractors];
bool prop[noOfProposals];


/* variables of static classes: values are constant */
int bid;
int state;
int limit[noOfContractors];
bool msgreceived[noOfContractors];


/* variables of dynamic classes */
bool answer[noOfProposals];     /* Argument prop1 bis prop3 */

/* associations */
bool cfpSender[noOfProposals];  /* Argument cfp */
bool cfpRecipient[9];           /* Argumente cfp, con */
bool propSender[9];             /* Argumente prop, con */
bool propRecipient[3];          /* Argument prop */
bool bidder[3];                 /* Argument con */



init
{
/* Initialization */

bid=1;
state=CNPstart;
limit[con1]=2;
limit[con2]=3;
limit[con3]=4;
msgreceived[con1]=false;
msgreceived[con2]=false;
msgreceived[con3]=false;
answer[prop1]=false;
answer[prop2]=false;
answer[prop3]=false;
cfpSender[cfp1]=false;
cfpSender[cfp2]=false;
cfpSender[cfp3]=false;
```

```
cfpRecipient[ind(cfp1,con1)]=false;
cfpRecipient[ind(cfp1,con2)]=false;
cfpRecipient[ind(cfp1,con3)]=false;
cfpRecipient[ind(cfp2,con1)]=false;
cfpRecipient[ind(cfp2,con2)]=false;
cfpRecipient[ind(cfp2,con3)]=false;
cfpRecipient[ind(cfp3,con1)]=false;
cfpRecipient[ind(cfp3,con2)]=false;
cfpRecipient[ind(cfp3,con3)]=false;
propSender[ind(prop1,con1)]=false;
propSender[ind(prop1,con2)]=false;
propSender[ind(prop1,con3)]=false;
propSender[ind(prop2,con1)]=false;
propSender[ind(prop2,con2)]=false;
propSender[ind(prop2,con3)]=false;
propSender[ind(prop3,con1)]=false;
propSender[ind(prop3,con2)]=false;
propSender[ind(prop3,con3)]=false;
propRecipient[prop1]=false;
propRecipient[prop2]=false;
propRecipient[prop3]=false;
bidder[con1]=false;
bidder[con2]=false;
bidder[con3]=false;
cnpEnd=false;



/* Transitions, i.e., guarded commands for different
   graph transformation rules */
do
        /* sendCFP */
        :: atomic { msgreceived[con1]==false -> msgreceived[con1]=true;
           cfp[cfp1]=true; cfpSender[cfp1]=true;
           cfpRecipient[ind(cfp1,con1)]=true;  }
        :: atomic { msgreceived[con2]==false -> msgreceived[con2]=true;
           cfp[cfp2]=true; cfpSender[cfp2]=true;
           cfpRecipient[ind(cfp2,con2)]=true;  }
        :: atomic { msgreceived[con3]==false -> msgreceived[con3]=true;
           cfp[cfp3]=true; cfpSender[cfp3]=true;
           cfpRecipient[ind(cfp3,con3)]=true;  }
```

```
/* acceptCFP */
:: atomic { cfp[cfp1]==true && cfpSender[cfp1]==true &&
   cfpRecipient[ind(cfp1,con1)]==true && limit[con1]>=bid ->
   cfp[cfp1]=false; cfpSender[cfp1]=false;
   cfpRecipient[ind(cfp1,con1)]=false; prop[prop1]=true;
   propSender[ind(prop1,con1)]=true; propRecipient[prop1]=true;
   answer[prop1]=true; }

:: atomic { cfp[cfp2]==true && cfpSender[cfp2]==true &&
   cfpRecipient[ind(cfp2,con2)]==true && limit[con2]>=bid ->
   cfp[cfp2]=false; cfpSender[cfp2]=false;
   cfpRecipient[ind(cfp2,con2)]=false; prop[prop2]=true;
   propSender[ind(prop2,con2)]=true; propRecipient[prop2]=true;
   answer[prop2]=true; }

:: atomic { cfp[cfp3]==true && cfpSender[cfp3]==true &&
   cfpRecipient[ind(cfp3,con3)]==true && limit[con3]>=bid ->
   cfp[cfp3]=false; cfpSender[cfp3]=false;
   cfpRecipient[ind(cfp3,con3)]=false; prop[prop3]=true;
   propSender[ind(prop3,con3)]=true; propRecipient[prop3]=true;
   answer[prop3]=true; }

/* recordProposal */
:: atomic { prop[prop1]==true && propSender[ind(prop1,con1)]==true &&
   propRecipient[prop1]==true && answer[prop1]==true &&
   !bidder[con1]==true && !bidder[con2]==true &&
   !bidder[con3]==true -> prop[prop1]=false;
   propSender[ind(prop1,con1)]=false; propRecipient[prop1]=false;
   answer[prop1]=false; bidder[con1]=true; }

:: atomic { prop[prop2]==true && propSender[ind(prop2,con2)]==true &&
   propRecipient[prop2]==true && answer[prop2]==true &&
   !bidder[con1]==true && !bidder[con2]==true &&
   !bidder[con3]==true -> prop[prop2]=false;
   propSender[ind(prop2,con2)]=false; propRecipient[prop2]=false;
   answer[prop2]=false; bidder[con2]=true; }

:: atomic { prop[prop3]==true && propSender[ind(prop3,con3)]==true &&
   propRecipient[prop3]==true && answer[prop3]==true &&
   !bidder[con1]==true && !bidder[con2]==true &&
   !bidder[con3]==true -> prop[prop3]=false;
   propSender[ind(prop3,con3)]=false; propRecipient[prop3]=false;
```

```
            answer[prop3]=false; bidder[con3]=true; }

         /* acceptProposal */
         :: atomic { bidder[con1]==true -> state=CNPend; end1:break; }
         :: atomic { bidder[con2]==true -> state=CNPend; end2:break; }
         :: atomic { bidder[con3]==true -> state=CNPend; end3:break; }
od
}
```