Technische Universität
Darmstadt
Fachbereich Informatik

Fraunhofer Institut für
Graphische Datenverarbeitung

Diplomarbeit

Über den komponenten-orientierten Entwurf von mobilen Agenten und deren Verhalten

von

Jan Hävecker
Matrikelnummer 861304

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Graphisch-Interaktive Systeme
Fraunhoferstraße 5
64283 Darmstadt

Prüfer: Prof. Dr.-Ing. J.L. Encarnação

Betreuer: Dipl.-Ing. Ulrich Pinsdorf

Aufgabenstellung für die Diplomarbeit des
Herrn cand.-Inform. Jan Hävecker
Matrikel-Nr. 861304

Thema: "Über den komponenten-orientierten Entwurf von mobilen Agenten und deren Verhalten"

Mobile Agenten sind Programme, die im Auftrag eines Benutzers über ein Netzwerk migrieren können, um in dessen Auftrag bestimmte Aufgaben selbstständig zu erledigen. *SeMoA* ist eine Plattform für mobile Software-Agenten. Die Plattform konzentriert sich vor allem auf den Aspekt der Sicherheit mobiler Agenten, inklusive dem Schutz der Agenten vor so genannten *malicious hosts*. Ein weiteres wichtiges Feature von SeMoA ist die Interoperabilität mit anderen Agenten- und Komponentenstandards wie beispielsweise *Aglets* und *JADE*. SeMoA kann daher auch als Application Server betrachtet werden. Der SeMoA Server selbst ist vollständig in Java implementiert und auch die Agenten sind in der Regel in Java geschrieben. SeMoA steht als *Open Source* zur Verfügung.

Bisher bietet SeMoA sehr wenig Hilfestellung bei der Implementierung der Agenten selbst bzw. deren Verhalten. Jeder Agent muss entweder von Grund auf neu geschrieben oder per *Copy&Paste* aus dem Programmcode von bestehenden Agenten zusammengestellt werden. Dadurch ist die strukturierte Wiederverwendbarkeit des Programmcodes der Agenten stark eingeschränkt. Dies hat zur Folge, dass der Aufwand für die Entwicklung von Agenten relativ hoch ist.

Zunächst soll untersucht werden, welche Methoden zur Modellierung und Implementierung von Agenten und deren Verhalten schon in anderen Agentenplattformen oder Forschungsarbeiten entwickelt wurden. Es existiert bereits eine Reihe von Arbeiten, welche die Entwicklung von (mobilen) Agenten erleichtern. Diese reichen von Methodologien über generische Architekturen bis hin zu konkreten Frameworks. Jeder dieser Ansätze hat spezifische Eigenschaften, welche den Einsatz im Sinne dieser Arbeit oder im Rahmen von SeMoA erschweren oder unmöglich machen. Vor allem die Modularität der jeweiligen Verhaltenselemente und der daraus resultierende Grad der Wiederverwendbarkeit ist bisher noch verbesserungsfähig.

Die Aufgabe besteht nun darin, ein Konzept zu entwickeln, mit dessen Hilfe die Programmierung von (mobilen) Agenten vereinfacht werden kann. Vorstellbar ist die Entwicklung eines Frameworks, eines Konstruktionswerkzeuges, einer Modellierungssprache für mobile Agenten, oder eine Kombination dieser Möglichkeiten. Oberste Priorität hat dabei die Wiederverwendbarkeit der Resultate, die das Konzept liefert. Wünschenswert ist die Entwicklung einer Komponenten-Technologie, welche die Modularisierung des Agenten und dessen Verhalten ermöglicht. Die einzelnen Komponenten sollen so gestaltet werden können, dass sie in unterschiedlichen Kontexten wiederverwendbar sind. Außerdem soll die Interoperabilität der Komponenten berücksichtigt werden; sowohl innerhalb der SeMoA Plattform, als auch zwischen verschiedenen Plattformen für Agenten.

Das entwickelte Konzept soll zunächst auf einer abstrakten Ebene modelliert werden. Dies kann beispielsweise mit Hilfe einer graphischen Modellierungssprache wie UML geschehen. Auf jeden Fall muss eine konkrete Umsetzung (Implementierung) des Konzepts für die SeMoA Plattform bereitgestellt werden. Diese Umsetzung soll so gestaltet sein, dass sie direkt im Entwicklungsprozess von Agenten angewendet werden kann.

Die Lösung der gestellten Aufgabe, insbesondere die Literaturrecherche und -auswertung, erfolgt nach wissenschaftlichen Methoden. Für die Umsetzung der entwickelten Lösung, sind die Konzepte des objekt-orientierten Software Engineerings anzuwenden.

Darmstadt, den 1. Mai 2004

Dipl.-Ing. Ulrich Pinsdorf

Prof. Dr.-Ing. J.L. Encarnação

Selbständigkeitserklärung

Hiermit erkläre ich an Eides statt, dass ich diese Arbeit eigenhändig und nur mit den angegebenen Hilfsmitteln angefertigt habe.

Jan Hävecker Darmstadt, November 2004

Danksagung

Besonderer Dank gebührt den Mitarbeitern des Fraunhofer Instituts, vor allem Dipl.-Ing. Ulrich Pinsdorf und Dipl.-Inform. Jan Peters für die wertvollen Anregungen und Ratschläge.

Bedanken möchte ich mich weiterhin bei allen, die mich während der Anfertigung dieser Arbeit unterstützt haben. Dazu gehören unter anderem Margot, Claus und Anne Hävecker, Nadine Karner, Swen Aussmann, Dennis Bartussek, Fabian Huschka und Jason Kafka.

Jan Hävecker Darmstadt, November 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Mobile Software-Agenten sind Programme, die im Auftrag eines Anwenders bestimmte Aufgaben selbstständig für diesen erledigen können. Agenten werden *mobil* genannt, wenn sie das Wirtssystem, auf dem sie ausgeführt werden, verlassen können um auf einem anderen System ihren Auftrag fortzuführen. Man spricht dabei von der *Migration* eines Agenten von einem System auf ein anderes. Die Hauptanwendungsgebiete für Technologien, die auf dem Konzept des mobilen Agenten basieren, sind Computer-Netzwerke beziehungsweise verteilte Systeme. Das Vordringen des Internet in viele Bereiche des Lebens und die neusten Errungenschaften in der Telekommunikation und drahtlosen Netzwerk-Technologien stellen eine weitläufige und allgegenwärtige Infrastruktur für die Bereitstellung von Diensten dar. Diese Infrastruktur lässt sich unter anderem von mobilen Agenten nutzen. Um die Ausführung eines Agenten auf einem Computer-System zu ermöglichen, muss in diesem System eine spezielle Software-Plattform bereitgestellt werden.

Viele Forschungs- und Entwicklungs-Bestrebungen der Vergangenheit konzentrieren sich auf die Bereitstellung von Software-Infrastrukturen, deren Aufgabe es ist, das Entwerfen von konkreten Anwendungen zu ermöglichen, welche auf mobilen Agenten basieren. Es gibt jedoch relativ wenige Ansätze, welche die Entwicklung des Agenten selbst und dessen Verhalten betreffen. Bisher bietet SeMoA sehr wenig Hilfestellung bei der Implementierung der Agenten beziehungsweise deren Verhalten. Jeder Agent muss entweder von Grund auf neu geschrieben oder per *Copy&Paste* aus dem Programmcode von bestehenden Agenten oder anderen System-Komponenten zusammengestellt werden. Dadurch ist die strukturierte *Wiederverwendbarkeit* des Programmcodes der Agenten stark eingeschränkt. Es entstehen keine unabhängigen *Software-Komponenten*, die sich in verschiedenen Kontexten wiederverwenden lassen. Eine Folge ist, dass der Aufwand für die Entwicklung von Agenten relativ hoch ist.

Im wissenschaftlichen Umfeld existieren einige Arbeiten, welche die Entwicklung von (mobilen) Agenten behandeln. Diese reichen von allgemeinen *Methodologien* über generische Architekturen bis hin zu konkreten *Frameworks*. Ein Problem der bekannten Methodologien ist, dass diese in der Regel von einem *top-down* Entwurf ausgehen, diesen jedoch nicht nahe genug an die Implementierungsebene annähern. Wie sich die entstandenen Modelle implementieren lassen, ist häufig unklar oder der vorgeschlagene Weg genügt nicht den Ansprüchen eines wiederverwendbaren Ansatzes. Problematisch bei Architekturen und Frameworks ist häufig, dass diese nicht auf die Mobilität eines Agenten ausgerichtet oder zu sehr an eine bestimmte Agenten-Plattform gekoppelt sind.

Im Großen und Ganzen lässt sich feststellen, dass ein allgemeiner Mangel an Wiederverwendbarkeit bezüglich der Entwicklung von mobilen Agenten und deren Verhalten besteht. Dieser Mangel bezieht sich sowohl auf die Wiederverwendbarkeit der Methoden und Techniken selbst, als auch auf die Resultate, die durch deren Anwendung entstehen.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es, ein Konzept zu entwickeln mit dessen Hilfe sich mobile Agenten und deren Verhalten entwerfen lassen.

Im Rahmen dieser Arbeit wird dem Verhalten des Agenten die zentrale Rolle in dessen Architektur zugeschrieben. Es wird also in den meisten Fällen nicht zwischen dem Agenten und dem Verhalten des Agenten unterschieden. Der Grund dafür ist, dass nach Meinung des Autors, die Architektur eines Agenten fast vollständig durch dessen Verhalten bestimmt wird oder zumindest, unter Verwendung geeigneter Entwurfsmethoden, in dieser Weise bestimmt werden kann.

Das zu entwickelnde Konzept soll im speziellen auf *mobile* Agenten zugeschnitten sein und sowohl den Entwurf von mobilen Agenten, als auch deren Implementierung unterstützen. Dabei soll darauf geachtet werden, dass sich die erstellten Modelle tatsächlich auf klar vorgegebenem Wege in eine lauffähige Implementierung überführen lassen.

Oberste Priorität hat die Wiederverwendbarkeit der Resultate, die das Konzept liefert. Die einzelnen Komponenten sollen so gestaltet werden können, dass sie in unterschiedlichen Kontexten wiederverwendbar sind.

Insgesamt ergeben sich drei zentrale Fragen, die durch diese Arbeit beantwortet werden sollen:

- Wie kann ein mobiler Agent aus wiederverwendbaren Komponenten konstruiert werden?
- Wie sehen diese Komponenten aus und welche Schnittstelle haben sie gemeinsam?
- Wie muss eine Architektur beschaffen sein, die das Verstehen und Warten von Agentenverhalten erleichtert?

Zur Demonstration der Anwendbarkeit des entwickelten Konzeptes soll eine exemplarische Umsetzung für SeMoA bereitgestellt werden.

1.3 Überblick

Der in dieser Arbeit vorgestellte Lösungsansatz basiert auf dem Konzept des *hierarchischen Zustandsautomaten*. Eine geeignete Interpretation diese Konzeptes ermöglicht es die Paradigmen des *komponenten-orientierten Software Engineerings* und der *zustandsorientierten Programmierung* (beziehungsweise der Modellierung) in einem auf Agenten bezogenen Kontext zu vereinen. Die Anwendung des Lösungsansatzes schlägt zum Einen ein Brücke zwischen der Modellebene bekannter Methodologien für den Entwurf von Agenten (-Systemen) und der Implementierungsebene; zum Anderen wird eine unabhängige Anwendung des entwickelten Ansatzes im Kontext mobiler Agenten ermöglicht.

Die vorliegende Arbeit ist in drei Teile gegliedert:

Teil I ist der Darstellung der Grundlagen gewidmet, welche das Fundament dieser Arbeit bilden. Enthalten sind dort theoretische Grundlagen, die zum Verständnis der Arbeit benötigt werden, sowie die Ergebnisse der Recherche, welche der Konzeption des Lösungsansatzes vorangestellt war. *Kapitel 2* gibt eine Einführung in das Gebiet der Software-Agenten. Dabei geht unter anderem um die Definition des Begriffs Agent und Agenten-System, sowie um eine Darstellung der Plattform SeMoA. In *Kapitel 3* wird eine Einführung in die *Unified Modelling Language* (*UML*) gegeben, da insbesondere die *Zustandsdiagramme* für diese Arbeit von großer Bedeutung sind. *Kapitel 4* stellt einige bekannte Architekturen und Methodologien für die Entwicklung von Agenten vor. Auf dieser Grundlage werden in *Kapitel 5* schließlich Arbeiten vorgestellt, welche thematisch mit der vorliegenden Arbeit verwandt sind.

Teil II beschäftigt sich mit der Gestaltung und Umsetzung des Konzepts das zur Entwicklung von mobilen Agenten geschaffen wurde. Die Gliederung ist stark am Entwurfsprozess eines Softwaresystems orientiert. Dabei beschreibt *Kapitel 6* die Anforderungen, die an das entwickelte Konzept gestellt wurden. In *Kapitel 7* wird der Lösungsansatz vorweggenommen um das Verständnis der nachfolgenden Kapitel zu erleichtern. *Kapitel 8* stellt eine Analyse des Konzepts bezüglich der Anforderung und der verwandten Arbeiten dar. In *Kapitel 9* und *Kapitel 10* wird schließlich die Umsetzung des Konzepts für SeMoA als Entwurf und Implementierung vorgestellt.

Teil III stellt die Resultate dieser Arbeit in geschlossener Form vor. *Kapitel 12* enthält eine Bewertung und *Kapitel 11* eine Zusammenfassung des vorgestellten Konzepts. Abschließend erfolgt in *Kapitel 13* ein Ausblick auf weiterführende Ideen zum behandelten Thema.

Teil I

Grundlagen

Kapitel 2

Agenten

Um eine Grundlage für das Verständnis des Inhalts der vorliegenden Arbeit zu schaffen, wird in diesem Kapitel ein kurze Einführung in das Thema Agenten gegeben. Nach *Jennings* sind die essentiellen Konzepte der agenten-orientierten Computerwissenschaft: *Agent*, *Agenten-Organisationen*, *Interaktion* und *Umgebung* [34]. Diese Konzepte werden im Folgenden vorgestellt. Weiterhin werden die Unterschiede zwischen Agenten im Allgemeinen und mobilen Agenten erläutert. Es wird auch eine kurze Einführung in das agenten-orientierte Software Engineering gegeben. Letztlich erfolgt eine kompakte Übersicht der SeMoA Plattform.

2.1 Definition

Der Begriff Agent stammt vom lateinischen Wort *agere* ab, was soviel bedeutet wie handeln oder wirken, und bezeichnet ursprünglich einen Geschäftsträger im politischen Sinne. Agenten stehen im Zentrum der Forschung einer Reihe von wissenschaftlichen Disziplinen. Dazu gehören Ökonomie [46], Philosophie [12], Kognitionswissenschaft [68] und Informatik [71]. In der Informatik wird dieser Begriff etwa seit Ende der 70er Jahre verwendet [7], es existiert jedoch bis heute keine allgemeingültige Definition des Konzepts Agent. Eine recht allgemein gehaltene Definition nach *Wooldridge* lautet beispielsweise:

> *Ein Agent ist ein Computersystem, das sich ein einer Umgebung befindet, und in der Lage ist autonome Handlungen in dieser Umgebung durchzuführen, um seine Ziele zu erreichen.*[1]

Als Beispiel für Agenten aus dem Bereich der Informatik lassen sich unter anderem Figuren in Computerspielen nennen, welche durch eine künstlichen Intelligenz gesteuert werden. Ein konkretes Beispiel für einen solchen Agenten ist der *Quakebot*, der von *Lent* und *Laird* vorgestellt

[1] Dieses Zitat ist eine Übersetzung durch den Autor, vergleiche Originaltext [75].

wurde [36]. Auch eine *Web Spider*, ein Programm, das Daten im Internet sammelt um Indizes für Suchmaschinen zu erstellen, kann als Agent betrachtet werden. Diese werden beispielsweise von *Google*[2] eingesetzt. Eine sehr greifbare Ausprägung des Agentenbegriffs stellen autonome Roboter dar. Als konkretes Beispiel sei hier *Sony's AIBO*[3] genannt, der sehr eindrucksvoll im Roboter-Fußball [58] eingesetzt wird.

Die genannte Definition des Agentenbegriffs von *Wooldridge* und *Jennings* ist nicht wirklich vollständig. Erstens sagt sie nichts über den Typ der Umgebung aus, in der sich der Agent befindet. Agenten sind in vielen verschiedenen Umgebungstypen zu finden; beispielsweise die physische Welt, das Internet, andere Agenten, ein Betriebssystem oder all das in verschiedenen Kombinationen. Zweitens wurde die Autonomie nicht definiert. Autonomie steht wie der Begriff des Agenten selbst für ein Konzept, das schwer eindeutig definiert werden kann.

Allein innerhalb der Informatik gibt es für Software-Agenten eine Vielzahl von unterschiedlichen Definitionen des Agentenbegriffs. Bezeichnungen für einige Typen von Agenten sind beispielsweise: *autonome Agenten*, *intelligente Agenten*, *rationale Agenten*, *mobile Agenten*, *virtuelle Agenten*, *Informationsagenten*, etc. Welche Bedeutung diese verschiedenen Bezeichnungen haben ist nicht ohne weiteres klar. Es wäre daher immer nötig, eine Definition des Konzepts mit anzugeben, das hinter dem jeweiligen Agentenbegriff steht. Darum werden Agenten üblicherweise anhand ihrer Eigenschaften charakterisiert.

Ein bekanntes Schema dafür stammt von *Wooldridge* und *Jennings* [75]. Dabei wird zwischen einem starken und einem schwachen Agentenbegriff (*notion of agency*) unterschieden. Der *schwache Agentenbegriff* umfasst die Eigenschaften:

Autonomie: Die Fähigkeit ohne direkte Steuerung (Intervention) zu funktionieren. Autonome Agenten können selbst entscheiden welche Handlungen sie unter welchen Bedingungen durchführen. Sie sind dadurch in der Lage unabhängig mit verschiedenen Situationen umzugehen.

Soziale Fähigkeit: Gestattet Agenten, mit anderen Agenten und möglicherweise auch mit Menschen zu interagieren. Dies kann durch die Verwendung einer speziellen Sprache für die Kommunikation zwischen Agenten geschehen (*agent-communication language*).

Reaktiviktät: Reaktive Agenten können ihre Umgebung wahrnehmen und auf Änderungen in dieser Umgebung reagieren.

Proaktivität: Ein proaktiver Agent verhält sich in einer zielgerichteten Art und Weise. Diese Agenten reagieren nicht einfach auf ihre Umgebung, sondern ergreifen selbst die Initiative, um ein Ziel zu erreichen.

[2] http://www.google.de
[3] http://www.sony.net/Products/aibo/index.html

Diese Charakteristiken werden auch als die *Essenz des Agententums* bezeichnet. Agenten derart zu gestalten, dass diese Charakteristiken auf sie zutreffen, ist eine sehr komplexe Aufgabe. Dies wird anschaulich von *Wooldridge* in [74, S. 32-33] dargestellt.

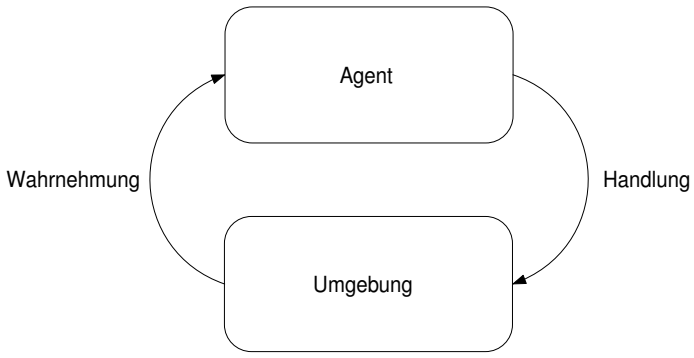

Abbildung 2.1: Die Interaktion zwischen Agent und Umgebung.

Abbildung 2.1 vermittelt einen abstrakten Überblick über das Zusammenspiel zwischen Agent und Umgebung [74]. Der Agent nimmt Informationen über die Umgebung wahr und führt Handlungen durch, die die Umgebung beeinflussen.

Der *starke Agentenbegriff* erweitert den schwachen Begriff um mentale Konzepte wie Glaube, Verlangen, Intention, Wissen etc. und soll an dieser Stelle nicht näher erläutert werden. In Abschnitt 4.1.3 werden solche Agenten vorgestellt.

Eine andere Charakterisierung für das Konzept des Agenten findet man bei *Etzioni* und *Weld* [15]. Zu deren Schema gehören neben der Automomie:

Zeitliche Kontinuität: Agenten sind keine einmaligen Berechnungen, die eine einzelne Eingabe auf eine einzelne Ausgabe abbilden und dann terminieren, sondern sind über einen längeren Zeitraum aktiv.

Persönlichkeit: Der Agent besitzt eine glaubhafte Persönlichkeit und einen emotionalen Zustand, um effektive Interaktion zu ermöglichen.

Kommunikationsfähigkeit: Dies entspricht der Eigenschaft der *sozialen Fähigkeit* nach *Wooldrige* und *Jennings*.

Anpassungsfähigkeit: Der Agent passt sich Vorgaben durch einen Benutzer an. Außerdem kann er sich Änderungen in seiner Umgebung anpassen.

Mobilität: Sie gestattet dem Agenten zwischen verschiedenen Rechnern, Systemarchitekturen und Plattformen transportiert zu werden.

Andere Charakteristika für Agenten, die nach *Luck* und *d'Inverno* oft implizit oder explizit vorausgesetzt werden, sind [40]:

Aufrichtigkeit: Agenten sind ehrlich.

Wohlwollen: Agenten tun was sie sollen.

Rationalität: Agenten arbeiten auf optimale Art und Weise um ihre Ziele zu erreichen.

Die Agenten, um die es in dieser Arbeit geht, haben alle die Eigenschaft, dass sie mobil sind (zumindest potentiell). Die Frage ist nun, ob man allein aufgrund der Mobilität Schlussfolgerungen ziehen kann, durch die sich gewisse Rahmenbedingungen für den Rest des Agenten ergeben. Dies wäre besonders für die Modellierung und Programmierung von mobilen Agenten im Sinne dieser Arbeit interessant. Anhand einer Aussage von *Luck* und *d'Inverno* [40] lässt sich feststellen, dass das nicht der Fall ist:

> *Die gemeinsame Eigenschaft von allen mobilen Agenten ist zwar die Mobilität; betrachtet man aber das Anwendungsgebiet der mobilen Agenten genauer, so wird man feststellen, dass diese Mobilität lediglich andere, zentralere Charakteristiken verstärkt.*[4]

2.2 Verhalten und Umgebung

Das Kernproblem, das ein Agent zu lösen hat, besteht darin herauszufinden, wie er sich verhalten muss, um seine gegeben Aufgaben am besten zu bewältigen. Er muss also entscheiden können, welche Aktionen er wann durchzuführen hat (Autonomie). Die Komplexität dieses Entscheidungsprozesses kann durch eine Reihe verschiedener Eigenschaften der Umgebung, in der sich der Agent befindet, beeinflusst werden. *Wooldridge* führt hierzu aus:

> *Intelligentes, rationales Verhalten (eines Agenten) ist auf natürliche Weise mit der Umgebung verbunden, in der sich ein Agent befindet — intelligentes Verhalten ist nicht köperlos, sondern ist ein Produkt der Interaktion zwischen Agent und Umgebung.*[5]

Es existiert ein Schema von *Russell* und *Norvig* mit dessen Hilfe sich die Eigenschaften der Umgebung klassifizieren lassen [63]:

Zugänglich oder unzugänglich: In einer zugänglichen Umgebung kann der Agent vollständige, genaue und aktuelle Informationen über den Zustand der Umgebung abrufen. Viele komplexere Umgebungen wie beispielsweise die physische Welt oder das Internet sind unzugänglich. Je zugänglicher eine Umgebung ist, desto einfacher ist es Agenten zu konstruieren, die in ihr operieren.

[4] Die Charakterisierung von Agenten durch ihre Eigenschaften ist also nicht ganz unproblematisch. *Luck* und *d'Inverno* stellen in [40] einen Ansatz dar die Charakterisierung von Agenten zu formalisieren.

[5] Dieses Zitat ist eine Übersetzung durch den Autor, vergleiche Originaltext [9, 74].

Deterministisch oder nicht-deterministisch: In einer deterministischen Umgebung hat jede Aktion einen konstanten, reproduzierbaren Effekt. Es gibt keine Unsicherheit bezüglich der Zustandsänderung, die durch eine Aktion hervorgerufen wird. Nicht-deterministische Umgebungen stellen die größte Herausforderung für die Konstruktion von Agenten dar.

Episodisch oder nicht-episodisch: In einer episodischen Umgebung lassen sich die Erfahrungen des Agenten in einzelne Episoden einteilen. Jede Episode besteht aus einer Wahrnehmung und einer darauf folgenden Handlung. Diese Episoden sind stets unabhängig; das heißt eine Episode hängt nicht davon ab, welche Handlungen in vorangegangenen Episoden stattgefunden haben.

Statisch oder dynamisch: Eine statische Umgebung kann ihren Zustand ausschließlich durch Handlungen des Agenten ändern. In einer dynamischen Umgebung existieren andere Prozesse, die Veränderungen der Umgebung herbeiführen können. Die physische Welt ist im höchsten Maße dynamisch.

Diskret oder kontinuierlich: Eine Umgebung ist diskret, falls es eine fest und endliche Anzahl von Handlungen und Wahrnehmungen in ihr gibt. Ein Schachspiel wäre ein Beispiele für eine diskrete Umgebung (es gibt eine feste Anzahl möglicher Bewegungen bei jedem Zug), Taxifahren ein Beispiel für eine kontinuierliche (Geschwindigkeit und Position ändern sich kontinuierlich).

Die komplexeste, allgemeine Klasse von Umgebungen bilden jene, die unzugänglich, nicht-deterministisch, nicht-episodisch, dynamisch und kontinuierlich sind. Die einfachste Klasse ist also zugänglich, deterministisch, episodisch, statisch und diskret. Wie *Russell* und *Norvig* jedoch feststellen, nützt es wenig, dass eine konkrete Umgebung deterministisch ist, wenn sie vom Aufbau und Umfang her so komplex ist, dass sie dadurch als quasi nicht-deterministisch zu behandeln ist [63]. Welche Eigenschaften die Umgebung eines Agenten in der Praxis aufweist, hängt von dem verwendeten Agenten-System und dem spezifischen Anwendungsgebiet ab.

Bei dem Entwurf von Agenten ist es wichtig zu berücksichtigen, in welcher Umgebung der Agent arbeiten soll. In statischen Umgebungen, die sich nicht im Verlauf der Zeit ändern, wäre proaktives Verhalten beispielsweise leichter zu implementieren als in dynamischen Umgebungen.

2.3 Mobile Agenten

Das Konzept des mobilen Agenten, so wie er in dieser Arbeit verwendet wird, ist eine spezielle Ausprägung von dem Konzept des *mobilen Codes*[6]. Der Begriff mobiler Code kommt aus dem Gebiet der verteilten Systeme und steht dort für den Paradigmenwechsel vom Versenden von

[6] Ein Patent von *General Magic* [54]

Daten (*data shipping*) zum Versenden von Funktion (*function shipping*). In traditionellen *Client-Server Systemen* werden ausschließlich Daten zwischen Prozeduren versendet, die bereits auf dem Client oder dem Server (statisch) existieren. In Systemen, die mit mobilem Code arbeiten, wird die Prozedur selbst versendet. Dadurch kann die Prozedur an der Stelle arbeiten, an der die Daten tatsächlich vorliegen. Mit Hilfe dieser *lokalen Interaktion* kann unter Umständen die Komplexität (Datenverkehr, Zeitaufwand) der Aufgabe reduziert werden.

Abbildung 2.2 stellt den Austausch von Daten im Client-Server Paradigma dar, Abbildung 2.3 illustriert das Versenden von Prozeduren [77].

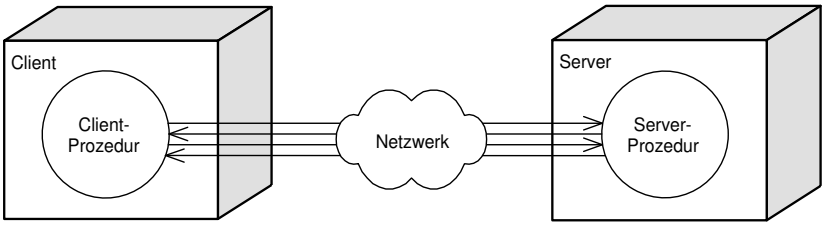

Abbildung 2.2: Das Versenden von Daten in klassischen Client-Server Systemen.

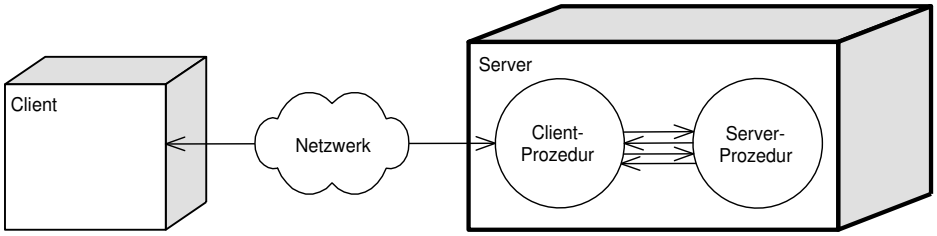

Abbildung 2.3: Das Versenden von Prozeduren im Paradigma des mobilen Codes.

Mobiler Code sollte jedoch nicht ausschließlich als Mittel zu Lastverteilung in verteilten Systemen betrachtet werden. Einige weitere Potentiale von mobilem Code bestehen in der dynamischen Erweiterung der Funktionalität, der Verstärkung der Autonomie und der Verbesserung der Fehlertoleranz bei beliebigen Diensten.

Während mobiler Code ein recht weitläufiger Begriff ist, gibt es für die Metapher des mobilen Agenten einige Eigenschaften, die essentiell für alle mobilen Agenten sind. Die zentralen Charakteristiken für mobile Agenten sind *Autonomie*, *Mobilität* und *Persistenz*, wobei Persistenz eng mit Mobilität verwandt ist. Wenn ein mobiler Agent sich von einem Ort zu einem anderen bewegt (*Migration*), müssen die internen Zustände und Daten erhalten werden um am Zielort weiter verwendet werden zu können.

Aus dieser Perspektive betrachtet setzt sich ein mobiler Agent prinzipiell aus drei Komponenten zusammen [20]:

Code: Der Programm-Code, welcher die Fähigkeiten und das Verhalten des Agenten definiert.

Daten: Die benötigten Daten, die der Agent mit sich führt um seine Aufgabe erfüllen zu können.

Zustand: Der interne Verarbeitungszustand (*state*) des Agenten.

2.3.1 Mobilität

Der Hauptunterschied zwischen mobilem Code und mobilen Agenten ist neben der Autonomie die Form der Mobilität. Nach *Picco* [55] sind eigentlich nur solche mobilen Einheiten als mobile Agenten zu betrachten, bei denen alle drei Bestandteile (Code, Daten, Zustand) bei der Migration übermittelt werden. Vor allem der Zustand des Agenten spielt eine besondere Rolle, der dieser für die Migration fixiert werden muss. Bei mobilem Code wird häufig nur der Code selbst übertragen.

Es lassen sich zwei grundsätzliche Arten der Mobilität unterscheiden: die *starke Mobilität* und die *schwache Mobilität* [72]. Bei der starken Mobilität wird die Ausführung des Agenten an der aktuellen Instruktion angehalten und nach der Migration exakt an der gleichen Stelle fortgesetzt. Dazu ist es nötig, den *execution stack* und die *heap Daten* des Prozesses zu erhalten (*serialisieren*). Bei der schwachen Mobilität wird der Ausführungszustand des Agenten nicht in die neue Umgebung transferiert.

Ist es aus bestimmten Gründen nicht möglich, Zustand des Agentenprozesses zur Laufzeit zu erhalten[7], so ist es nicht ganz einfach, starke Mobilität umzusetzen. Eine Möglichkeit den internen Zustand des Agenten dennoch zu erhalten, wäre die Verwendung von Wiedereintritts-Methoden und expliziten Zustandsvariablen im Agenten selbst. Dieser Ansatz kann in Anwendungen mit mehreren Prozessen jedoch sehr komplex und fehleranfällig werden [7].

2.3.2 Vorteile

Es gibt einige relative Vorteile, die mobile Agenten Systeme gegenüber traditionellen Client-Server Systemen bieten. Relativ deshalb, weil keiner dieser Vorteile ausschließlich mit mobilen Agenten zu erreichen ist. Alles, was mit mobilen Agenten umgesetzt wird, kann auch mit anderen Entwurfsprinzipien wie Client-Server erreicht werden. Die Summe dieser relativen Vorteile lässt sich jedoch mit Hilfe von mobilen Agenten leichter nutzen. Zu den genannten Vorteilen gehören:

Lokale Interaktion: Ein wesentlicher Vorteil von mobilem Code. Der Vorteil vom Versenden von Funktion gegenüber dem Versenden von Daten wurde bereits vorgestellt (siehe Abschnitt 2.3).

[7] Dies ist beispielsweise in der Java VM Architektur nicht möglich, da diese keinen Zugriff auf den *execution stack* gestattet

Erhöhte Flexibilität: Ein Client greift auf die Ressourcen, die ein Server anbietet, über eine vordefinierte Menge von Diensten zu. Die Schnittstellen dieser Dienste sind ebenfalls vordefiniert und müssen dem Client bekannt sein. Es ist im klassischen Client-Server Paradigma schwer bis unmöglich diese Schnittstellen dynamisch an neue Anforderungen anzupassen. Hier können mobile Agenten benutzt werden um die Schnittstellen auf der Client- und/oder Server-Seite dynamisch zu aktualisieren.

Verbesserte Fehlertoleranz: Während der Interaktion in klassischen Client-Server Systemen kann es vorkommen, dass sich der Zustand einer Berechnung über ganze Teile des Systems erstreckt. Dadurch wird es erschwert, einen auftretenden Fehler zu lokalisieren und zu beheben. Da mobile Agenten den Code, der für die Lösung einer Aufgabe benötigt wird, mit sich führen, ist der Zustand der Interaktion lokal im Agenten vorhanden.

Protokoll-Kapselung: In konventionellen Systemen sind Daten typischerweise passive Elemente, die von anderen aktiven Elementen des Systems verarbeitet werden. Netzwerk-Pakete beispielsweise enthalten Daten, die an jedem Knoten des Netzwerks, den sie passieren, verarbeitet und weitergeleitet werden. Unter der Verwendung von mobilen Agenten werden die Daten zu aktiven Elementen, die die benötigte Logik zur ihrer Interpretation selbst mitbringen. Ein Netzwerk-Paket würde dann das verwendete Routing-Protokoll mit sich führen, während es durch das Netzwerk wandert.

Unterstützung für Offline-Operationen: Mobile Agenten wandern durch Netzwerke, suchen Informationen und erledigen selbstständig Aufgaben für ihren Auftraggeber. Bei der Rückkehr präsentieren sie die Früchte ihrer Arbeit. In der Zwischenzeit ist es für den Nutzer nicht nötig, den Fortschritt der Arbeit des Agenten zu überwachen. Dies macht mobile Agenten besonders nützlich in mobilen Umgebungen, da keine permanente Netzwerkverbindung aufrecht erhalten werden muss. Eine Verbindung ist außer beim Versenden und Empfangen des Agenten nicht nötig.

Unterstützung für schwache Endgeräte: Mobile Endgeräte[8] verfügen oft über relativ wenig Speicher und Rechenkapazität. Es bietet sich daher an, Aufgaben, die viel Rechenzeit oder Speicherplatz benötigen, mit einem mobilen Agenten an einen Ort zu schicken, an dem mehr Ressourcen zur Verfügung stehen. Dort kann die Aufgabe dann in kürzerer Zeit gelöst werden.

Gute Skalierbarkeit: Die bessere Skalierbarkeit von (mobilen) Agenten-Systemen basiert auf dem verwendeten Kommunikations-Paradigma. Im Gegensatz zu Systemen, die mit *remote procedure calls* (*RPC*) arbeiten, wird in mobilen Agenten Systemen das Versenden von Nachrichten (*messaging*) praktiziert.

Robuste Fern-Interaktion: Eine Schwäche der Client-Server Systeme, die RPC verwenden, ist die reduzierte Verlässlichkeit der Kommunikationswege in *Wide Area Networks*. Hier

[8] Gemeint sind PDAs, Handys etc.

ist das Versenden von Nachrichten, wie es in mobilen Agenten-Systemen angewendet wird von Vorteil, da dies inhärent asynchron abläuft. Der Empfänger muss nicht erst antworten bevor der Sender eine weitere Nachricht schicken kann. Weiterhin müssen Agenten zwangsläufig mit der Situation umgehen können, dass ein Server oder Dienst nicht erreichbar ist. Idealer Weise sollten sie dann ihr Ziel selbstständig auf einem alternativen Weg erreichen können.

Die eigentliche Stärke von mobilen Agenten liegt in der Kombination der relativen Vorteile. Wie *Harrison et al.* feststellen, ist „die Gesamtheit der Vorteile von mobilen Agenten überwältigend stark“, weil sie ein ein umfassendes, offenes und allgemeines Framework für die Entwicklung und Personalisierung von Netzwerkdiensten bieten können [30]. Die Alternativen zu mobilen Agenten können diese zwar in relativen Vorteilen überbieten, es gibt jedoch keine einzelne Alternative, die all die Funktionalität unterstützt, die von einem System für mobile Agenten geboten wird.

Eine zusammenfassende Darstellung der Vorteile von mobilen Agenten Systemen gegenüber traditionellen Client-Server Systemen findet sich in [10]. In [55] finden sich einige konkrete Beispiele für die erfolgreiche Anwendung von mobilen Agenten.

2.4 Multi-Agenten-Systeme

Multi-Agenten-Systeme (MAS) bieten eine Umgebung für Gruppen von Agenten, die zusammenarbeiten um ihre Ziele zu erreichen (Organisation). Die Gruppe der Agenten kann dabei statisch oder dynamisch zusammengesetzt sein, oder aus einer Kombination von statisch und dynamisch. Um ein bestimmtes Ziel zu erreichen werden typischerweise die Gruppen so zusammengestellt, dass sich jeweils auf bestimmte Aufgaben spezialisierte Agenten gegenseitig unterstützen können.

Für die Koordination beim Erreichen der Ziele steht Kommunikation im Vordergrund (soziale Fähigkeit), sei es zwischen Agenten oder Menschen und Agenten. Dafür ist eine gemeinsame Sprache notwendig, die von allen Beteiligten verstanden wird. In [18] wird eine solche *Agent Communication Language* (*ACL*) definiert.

Multi-Agenten Systeme sind als *Middleware* zu verstehen, mit deren Hilfe konkrete Anwendungen in Bereichen wie E-Commerce, intelligenten Assistenz Systemen, Multi-Media etc. implementiert werden können.

2.5 Agenten-Orientiertes Software Engineering

Das Hauptziel des AOSE ist die Entwicklung von Methodologien und Werkzeugen, die es ermöglichen agenten-basierte Software mit möglichst geringen Entwicklungs- und Wartungskos-

ten zu erstellen. Im agenten-orientierten Software Engineering (AOSE) stellt die Metapher des Agenten die zentrale Einheit für die Kapselung von Funktionalität und Verantwortlichkeiten dar. Zusätzlich soll die entstehende Software flexibel, leicht zu benutzen, skalierbar und von hoher Qualität sein. Bei der Entwicklung von Multi-Agenten Systemen sind die Konzepte, die verwendet werden um mit der Komplexität von verteilten Systemen umzugehen, im Prinzip die gleichen wie in anderen Software-Umgebungen: Dekomposition, Abstraktion, Organsation [4, 35]. Es müssen daher wiederverwendbare agenten- und problem-spezifische Abstraktionen gefunden werden. Der Code des Multi-Agenten-Systems muss in wiederverwendbare Komponenten unterteilt werden. Es werden Werkzeuge für die Konstruktion, Überwachung und Verbesserung von Frameworks für Multi-Agenten Systeme benötigt.

Agenten-orientierte Programmierung[9] (AOP) kann als eine Erweiterung der objekt-orientierten Programmierung (OOP) angesehen werden. Die objekt-orientierte Programmierung seinerseits kann als Nachfolger der prozeduralen Programmierung betrachtet werden. In der objekt-orientierten Programmierung ist das zentrale Konzept das Objekt. Ein Objekt ist eine logische Kombination von Datenstrukturen und deren zugehörigen Funktionen. Objekte werden hauptsächlich als Abstraktionen für *passive* Objekte der physischen Welt benutzt. Agenten können eine gute Abstraktion für *aktive* Objekte darstellen. Ein weiterer Unterschied zwischen agenten-orientierter Programmierung und objekt-orientierter Programmierung ist, dass Objekte in der Regel extern kontrolliert werden (*whitebox control*). Dies steht im Gegensatz zu Agenten, bei denen aufgrund ihres autonomen Verhaltens die Kontrolle eher im Agenten selbst liegt (*black-box control*).

2.6 Das Projekt SeMoA

Der Projektname *SeMoA* steht für *Secure Mobile Agents*. Es handelt sich dabei um eine erweiterbaren und offenen Application Server für sichere Multimedia und E-Commerce Anwendungen, bei dem eine Unterstützung für mobile Agenten ein zentraler Teil des Systems ist. SeMoA wird am *Fraunhofer Institut für Graphische Datenverarbeitung* in der Abteilung *Sicherheitstechnologie für Graphik und Kommunikationssysteme* entwickelt.

SeMoA ist prinzipiell zwar als Multi-Agenten-System zu verstehen, der Aspekt der Kommunikation beziehungsweise der sozialen Fähigkeit von Agenten steht jedoch nicht im Vordergrund. Zentral ist vielmehr das Konzept der Mobilität, so wie es in Abschnitt 2.3 besprochen wurde.

SeMoA konzentriert sich vor allem auf den Aspekt der Sicherheit mobiler Agenten, inklusive dem Schutz der Agenten vor so genannten *malicious hosts*[10] [60]. Die Reiseroute eines mobilen Agenten umfasst in der Regel eine Reihe von Servern. Diese können unter Umständen von konkurrierenden Anbietern betrieben werden. Ein *malicious host* könnte versuchen den Agenten

[9] Die agenten-orientierte Programmierung bildet ein Teilgebiet des AOSE.

[10] Dies bedeutet etwa soviel wie „boshaftes Wirts-System“.

zu überwachen, zu manipulieren, Daten zu stehlen oder ihn als trojanisches Pferd zu verwenden, um Angriffe auf Server konkurrierender Anbieter zu starten. Andererseits muss ein System für mobile Agenten auch vor boshaften Agenten (*malicious agents*) geschützt werden. Boshafte Agenten könnten versuchen, anderen Agenten zu schaden oder unautorisierten Zugriff auf das System zu erhalten. Ein vernünftiges Sicherheitsmodell ist also absolut notwendig und wird von SeMoA zur Verfügung gestellt.[11]

Ein weiteres wichtiges Feature von SeMoA ist die Interoperabilität mit anderen Plattformen [56, 57], wie beispielsweise *Aglets* und *JADE*. Agenten in SeMoA werden nicht vom Server direkt verwaltet sondern durch die Verwendung einer Abstraktion zwischen Server und Agent, dem so genannten *Lifecycle*. Ein *Lifecycle* ist eine Kapselung (*wrapper*), welche den Agenten umschließt. Der *Lifecycle* stellt die SeMoA Umgebung als die natürliche Umgebung des Agenten dar, und umgekehrt den Agenten der fremden Plattform als SeMoA Agenten. Verschiedene Umgebungen für Agenten können parallel emuliert werden, was prinzipiell die Interaktion zwischen heterogenen Agententypen ermöglicht.

2.6.1 Die Architektur von SeMoA

Grundlage des SeMoA-Systems ist einen minimaler System-Kern, der sich flexibel erweitern und leicht an ein beliebiges Anwendungsgebiet anpassen lässt. In der Architektur von SeMoA wird zwischen Agenten und Diensten unterschieden. Agenten können Dienste im System registrieren und abfragen. Die Aufgaben des SeMoA-Servers sind das Laden und Installieren von Agenten im System, das Senden von Nachrichten an Agenten, das Entfernen von Agenten aus dem System, die Separierung der Agenten und die Bewahrung von deren Anonymität, sowie das Verwalten der Dienste.

Agenten werden durch ein Verzeichnis verwaltet, in dem der Name des Agenten auf eine privates Objekt innerhalb des Servers abgebildet wird. Dieses Objekt wird als der *Kontext* des Agenten (*agent context*) bezeichnet. Der Kontext kontrolliert den *Lebenszyklus* (*lifecycle*) des Agenten, den er repräsentiert und verwaltet Referenzen zu verschiedenen Objekten die mit dem Agent assoziiert sind.

Zur Verwaltung der Dienste existiert ein globales Verzeichnis, das *Environment*[12]. Dadurch lassen sich beliebige Objekte mit Hilfe eines bekannten Namens veröffentlichen und auffinden (Erhalt der Schnittstelle). Die Namen sind hierarchisch strukturiert, ähnlich wie in einem Dateisystem. Für die Manipulation des *Environments* steht in SeMoA eine *shell* und ein graphisches Werkzeug (*Envision*) zur Verfügung.

Die meisten Dienste und System-Komponenten in SeMoA können leicht durch andere Implementierungen ersetzt werden. In vielen Fällen geschieht dies einfach durch die Implementierung

[11] Für Details über das Sicherheitsmodell sei auf [60] verwiesen.

[12] Das *Environment* ist aus der Perspektive des Agenten als dessen Umgebung zu betrachten, wie sie in Abschnitt 2.2 diskutiert wurde.

der geeigneten Schnittstelle.

Der SeMoA-Server selbst ist vollständig in Java implementiert und auch die Agenten können in Java geschrieben werden. SeMoA steht als *Open Source* zur Verfügung[13]. Nähere Details über SeMoA finden sich in [61].

2.6.2 Mobile Agenten in SeMoA

Im Gegensatz zu anderen, vergleichbaren Plattformen für mobile Agenten existiert in SeMoA keine Schnittstelle oder Basisklasse für Agenten. Ein Agent wird statt dessen einfach mit einem Ausführungsprozess assoziiert. Dieser dedizierte Prozess führt ausschließlich den Code, des an ihn gebundenen Agenten aus. Für die Mobilität des Codes beziehungsweise des Agenten ist es lediglich notwendig, dass dieser serialisiert werden kann.

Technisch bedeutet dies, dass ein Agent durch eine Klasse dargestellt ist, welche zumindest die Java Schnittstellen `java.lang.Runnable` und `java.io.Serializable` implementiert.

Die Schnittstelle `Runnable` definiert eine Methode mit dem Namen `run`. Diese Methode stellt Haupt- oder Startprozedur des Agenten dar. Wenn ein Agent nach eine Migration sein Zielsystem erreicht hat, passiert dieser Zunächst eine Reihe von Sicherheits-Schichten, in denen der Agent überprüft wird. Der Agent wird schließlich gestartet indem SeMoA einen Java *Thread* zur Verfügung stellt, welcher die `run` Methode des Agenten ausführt. Alle Handlungen, die ein Agent von nun an ausführt, gehen von dieser Methode aus. Ist das Ende der Methode erreicht, terminiert der Agent.

Um die Mobilität eines Agenten zu ermöglichen, ist es notwendig, dass dieser in eine persistente Form transformiert, also dass er serialisiert werden kann. Damit dies möglich ist, genügt es einfach, die Schnittstelle `Serializable` zu implementieren. Die der eigentliche Vorgang der Serialsierung und De-Serialisierung, wird automatisch von Java und SeMoA übernommen.

Dadurch, dass SeMoA weder eine Schnittstelle noch eine Basisklasse für Agenten anbietet, wird eine maximale Flexibilität bezüglich der Form erreicht, in der ein Agent vorliegen kann. Nachteilig ist jedoch, dass das Konzept des Agenten, in dieser Form, wenig greifbar ist. Erfahrungsgemäß bereitet dies gerade Anfängern, die sich zum ersten Mal mit SeMoA beschäftigen gewisse Schwierigkeiten.

2.6.3 Das Starten von Agenten

Ein Agent wird in Form eines *Java Archives* (JAR) transportiert, das den serialisierten Zustand des Agenten enthält. Bevor der Agent im System aufgesetzt und eine seiner Klassen mit der lokalen JVM verbunden werden, muss der Agenten zunächst eine Reihe von Sicherheitsfiltern

[13] http://www.semoa.org

passieren. Unter anderem wird dabei die digitale Signatur des Agenten überprüft und dessen *Byte Code* analysiert. Sobald der Agent für den Server zugelassen worden ist, werden eigens eine *Thread Group* und ein *Class Loader* für den Agenten erzeugt. Innerhalb der *Thread Group* wird ein *Thread* für die Ausführung des Agenten verwendet.

Vor dem Start des Agenten wird dessen System-Typ (SeMoA, JADE, etc.) und dessen Agenten-Typ (Java, Shell-Script, etc.) abgefragt. Aufgrund dieser Informationen stellt SeMoA einen geeigneten *Lifecycle* für den Agenten zur Verfügung.[14] Dieser kapselt die Instanz des Agenten und übersetzt zwischen dem *Lifecycle* eines SeMoA-Agenten und dem *Lifecycle* des ursprünglichen Agenten-Systems (falls diese verschieden sind).[15] Insbesondere erzeugt der *Lifecycle* alle Komponenten, die benötigt werden, um dem Agenten zu suggerieren, dass er sich in einem System seines ursprünglichen Typs befindet. Durch die Verwendung eines *Lifecycle* wird die Repräsentation des Agenten vollständig vom SeMoA-Kern entkoppelt.

[14] Es wird das Entwurfsmuster *Abstract Factory* verwendet [21, S. 87-95].
[15] Hier kommt das Entwurfsmuster *Adapter* zum Einsatz [21, S. 139-150].

Kapitel 3

Die Unified Modelling Language

Die *Unified Modelling Language* oder kurz *UML* [5, 19, 62, 70], ist eine visuelle Modellierungssprache, die in dieser Arbeit eine wichtige Rolle spielt. Daher werden an dieser Stelle ein knapper Überblick sowie eine Einführung in die relevanten Teile der Sprache gegeben.

3.1 Überblick

UML ist eine vielseitig einsetzbare, visuelle Modellierungssprache. Sie wurde entwickelt, um den Entwurf und die Konfiguration von Software-Systemen zu erleichtern, sowie deren Verständlichkeit und Wartbarkeit zu verbessern. UML definiert Artefakte und Modelle für die Spezifizierung, Visualisierung, Konstruktion und Dokumentation von *diskreten Systemen*. Anwenden lässt sich die Sprache in verschiedensten Entwicklungsmethoden, insbesondere iterativen, objekt-orientierten Entwurfsprozessen, Anwendungsgebieten und Medien.

Mit UML lassen sich Informationen über die *statische Struktur* und das *dynamische Verhalten* eines Systems visuell fixieren. Dabei wird ein System als eine Menge von diskreten Objekten modelliert, die miteinander interagieren, um die Aufgabe zu erfüllen, die von einem Nutzer gefordert wird. Die statische Struktur definiert die Arten von Objekten, die für die Implementierung eines Systems wichtig sind, sowie deren Beziehungen untereinander. Das dynamische Verhalten definiert eine Reihe von Schritten, welche die Objekte innerhalb ihrer Lebenszeit durchlaufen. Außerdem wird die Kommunikation zwischen den Objekten spezifiziert, die für die Erfüllung der Aufgabe des Systems nötig ist.

Ein System wird aus verschiedenen, separaten Perspektiven modelliert, die es ermöglichen das System für verschiedene Zwecke zu interpretieren. Für jede dieser Ansichten des Systems existieren jeweils eigene Modelle, die durch Diagramme beschrieben werden. Die erstellten Modelle selbst werden durch eine spezielle Ansicht, dem Modell-Management, in hierarchischen

Einheiten organisiert. Dabei sind *Pakete* (*packages*) die organisatorischen Einheiten, in die Modelle unterteilt werden. Pakete fassen Modellelemente in Gruppen zusammen und lassen sich beliebig verschachteln, können also selbst wieder Pakete enthalten. Ein System lässt sich so als ein einzelnes Paket darstellen, in dem alle anderen Bestandteile des Modells rekursiv enthalten sind.

Die wichtigsten UML Diagrammtypen, die in dieser Arbeit vorkommen, werden nun kurz vorgestellt:

Klassendiagramm (class diagram): Klassendiagramme sind wohl die bekanntesten und am häufigsten verwendeten Diagramme der UML. Ein Klassendiagramm ist ein Repräsentation der statischen Perspektive und enthält eine Reihe statischer Modellelemente. Dazu gehören die Klassen und Typen sowie deren Inhalt und Beziehungen. Ein Klassendiagramm kann auch Pakete darstellen und Symbole für verschachtelte Pakete enthalten.

Komponentendiagramm (component diagram): Ein Komponentendiagramm stellt die statischen Abhängigkeiten zwischen den einzelnen Software-Komponenten dar. Dazu gehört sowohl der Quellcode als auch binäre und ausführbare Komponenten. In einem Komponentendiagramm werden jedoch keine Instanzen von Komponenten dargestellt, dies ist die Aufgabe von Verteilungsdiagrammen.

Verteilungsdiagramm (deployment diagram): Ein Verteilungsdiagramm beschreibt die Konfiguration von Komponenten und deren Instanzen zur Laufzeit. Verteilungsdiagramme gehören zur statischen Ansicht eines Systems.

Sequenzdiagramm (sequence diagram): Ein Sequenzdiagramm zeigt die Interaktionen zwischen Objekten in ihrer zeitlichen Abfolge. Insbesondere werden die interagierenden Objekte dargestellt, sowie die Reihenfolge der Nachrichten, die dabei ausgetauscht werden. Sequenzdiagramme beschreiben dynamischen Abläufe innerhalb eines Systems.

Zustandsdiagramm (statechart): Dieses Diagramm zeigt einen *Zustandsautomaten*. Ein Zustandsautomat beschreibt dynamischen Aspekte eines Systems und setzt sich aus Zuständen und Transitionen zwischen den Zuständen zusammen. Da dieser Diagrammtyp von besonderer Bedeutung innerhalb der vorliegenden Arbeit ist, wird er später noch ausführlich diskutiert (siehe Abschnitt 3.2).

Um Missverständnisse zu vermeiden, erfolgt an dieser Stelle eine Auflistung von Punkten, die beschreiben was UML nicht ist:

Eine Programmiersprache: Spezifikationen, die mit UML erstellt wurden, sind nicht direkt ausführbar. Mit Hilfe von Werkzeugen lässt sich jedoch aus den erstellten Spezifikationen Code in verschiedensten Programmiersprachen teilweise automatisch erzeugen. Weiterhin lassen sich via *reverse engineering* UML-Modelle aus bestehenden Programmen erzeugen.

Eine Formale Sprache: UML wurde nicht für den Einsatz von Methoden entworfen, die auf einer stark formalisierbaren Semantik basieren; wie beispielsweise die *automatische Verifikation* von Systemkomponenten.[1]

Eine spezialisierte Sprache: UML wurde nicht konzipiert, um damit Modelle für sehr spezielle Gebiete, wie den Entwurf von graphischen Benutzeroberflächen, VLSI-Schaltkreisen oder regelbasierten, künstlichen Intelligenzen zu erstellen. Auch für die Modellierung von *kontinuierlichen Systemen*, wie man sie zum Beispiel in der Physik findet, wurde UML nicht entworfen.

Die UML ist heute einer der wichtigsten Standards zur graphischen Repräsentation des Entwurfs von fast allen Teilen eines objekt-orientierten Systems. UML wird jedoch seit einiger Zeit auch in ganz anderen Anwendungsbereichen, wie beispielsweise der Modellierung von Geschäftsprozessen eingesetzt. Die Beliebtheit von der Sprache ist vor allem auf eine gute Balance zwischen Erlernbarkeit und Ausdrucksstärke zurückzuführen.

3.2 Zustandsdiagramme

Das Konzept der Zustandsdiagramme, wie es in UML eingesetzt wird, ist stark an der Arbeit von *David Harel* [29] orientiert. Ein Zustandsdiagramm stellt einen *Zustandsautomaten* (*state machine*) dar. Ein solcher Automat wird durch einen Graph repräsentiert, der sich aus *Zuständen* (*states*) und *Transitionen* (*transitions*) zusammen setzt.

Mit Hilfe von Zustandsautomaten lässt sich beschreiben, wie sich ein Objekt innerhalb eines Systems verhält, insbesondere, wie sich das Verhalten im Laufe der Zeit verändert. Ein Zustandsautomat lässt sich auch als Modell verstehen, das alle möglichen Abläufe, die im Leben eines Objektes auftreten können, darstellt. Jedes Objekt wird dabei als eine isolierte Einheit betrachtet, das nur bestimmte, vordefinierte *Ereignisse* (*events*) wahrnehmen und auf diese reagieren kann. In diesem Sinne wird alles, was ein Objekt beeinflussen kann durch Ereignisse modelliert, sei es die Kommunikation zwischen verschiedenen Objekten oder das Auftreten von Veränderungen in der Umgebung. Ereignisse werden zwischen Objekten oder der Umgebung und Objekten wie Nachrichten versendet und empfangen. Wenn ein Objekt ein Ereignis registriert, reagiert es in einer Weise auf das Ereignis, die vom aktuellen Zustand abhängt. Die Reaktion kann das Ausführen einer *Aktion* (*action*) oder das Wechseln des Zustands (Auslösen einer Transition) zur Folge haben.

Der Zustandsautomat der UML lässt sich als eine Erweiterung des *endlichen Automaten* (*finite state machine*) betrachten. Ein endlicher Automat ist ein mathematisches Modell einer einfachen Rechenmaschine, das in der Informatik, insbesondere der Berechenbarkeitstheorie und

[1] Zu diesem Zweck ist beispielsweise die Sprache *Z* besser geeignet [67].

dem Compilerbau verwendet wird. Der Zusatz endlich leitet sich aus der Tatsache ab, dass die Anzahl der inneren Zustände eines solchen Automaten stets endlich ist. Ein endlicher Automat liest eine Folge von Symbolen aus einem Eingabealphabet ein und gibt, je nach Struktur des Automaten, eine bestimmte Folge von Symbolen aus einem Ausgabealphabet aus. Beim Lesen der Eingabe und Schreiben der Ausgabe geht der Automat in Schritten vor, die durch die Zustände definiert sind. In jedem Schritt wird das jeweils nächste Symbol der Eingabe eingelesen. Abhängig von dem gelesenen Symbol und dem aktuellen Zustand wird eine Folge von Symbolen aus dem Ausgabealphabet ausgegeben und der Automat geht in einen neuen Zustand über.

Man unterscheidet zwischen den zwei Typen endlicher Automaten: *Moore* und *Mealy*. Bei einem Moore-Automat hängt die Ausgabe des Automaten ausschließlich von seinem Zustand ab; bei einem Mealy-Automat hängt die Ausgabe sowohl von seinem Zustand, als auch von dem aktuellen Symbol der Eingabe ab. Grundsätzlich kann man weiterhin zwischen *deterministischen* und *nichtdeterministischen* endlichen Automaten unterscheiden. Bei deterministischen endlichen Automaten gibt es beim Zustandsübergang nur eine Möglichkeit für den Folgezustand (pro Eingabe), bei nichtdeterministischen endlichen Automaten sind mehrere Übergänge möglich.

In UML wird das Modell des endlichen Automaten durch die Konzepte Hierarchie und Nebenläufigkeit (*concurrency*), sowie durch das auf Ereignissen basierende Kommunikationsprinzip erweitert [5, 29]. Mit den Zustandsdiagrammen der UML lassen sich sowohl Moore-, als auch Mealy-Automaten modellieren. Ein Zustandsautomat, in dem *alle* Aktionen Transitionen zugeordnet werden, ist vom Typ Mealy; ein Zustandsautomat, in dem *alle* Aktionen mit Zuständen assoziiert werden, ist vom Typ Moore. In der Praxis werden in der Regel Zustandsautomaten konstruiert, die eine Mischung aus beiden Typen sind. Die Zustandsautomaten der UML sind nichtdeterministisch.

3.2.1 Elemente

Dieser Abschnitt beschreibt die Elemente eines Zustandsdiagramms im Detail.

Aktionen. Eine Aktion ist eine kurze, atomare Berechnung, deren Dauer im Verhältnis zu der Dauer der Ereignisse des gesamten Systems vernachlässigbar ist. Eine Aktion kann nicht angehalten werden, sondern wird im Ganzen vollzogen. Das Versenden einer Nachricht, der Aufruf einer Methode, oder das Erzeugen oder Zerstören eines Objektes etc. sind Beispiele für Aktionen. Man kann Aktionen auch als diskrete Schritte auffassen, aus denen sich ein Verhalten zusammen setzt.

Eine Aktion kann aus einer Reihe von anderen Aktionen zusammengesetzt sein. Auch eine solche Sequenz von Aktionen kann nicht unterbrochen oder terminiert werden, bevor diese nicht vollständig beendet worden ist.

Aktionen werden Transitionen oder Zuständen zugeordnet und werden entsprechend ausgeführt, falls eine Transition ausgelöst oder ein Zustand aktiviert oder deaktiviert wird.

Ereignisse. Ein Ereignis repräsentiert eine Begebenheit, die für das Objekt von Bedeutung ist. Ereignisse treten zu bestimmten Zeitpunkten an bestimmten Orten auf; sie haben keine Dauer. Es ist möglich, dass Ereignisse gewisse *Parameter* haben. Über diese Parameter lässt sich eine bestimmte Instanz eines Ereignisses charakterisieren. Einige Grundtypen von Ergeignissen lassen sich unterscheiden:

Änderung: Ein Ereignis kann die Änderung eines internen oder externen Zustandes signalisieren. Solche Ereignisse können auch vom Objekt selbst versandt werden.

Anfrage: Ereignisse können Anfragen zwischen Objekten repräsentieren. Anfragen erfordern eine Antwort.

Signal: Ereignisse können asynchrone Benachrichtigungen zwischen Objekten repräsentieren. Diese erfordern keine Antwort.

Zeit: Ereignisse können symbolisieren, dass ein bestimmter, absoluter Zeitpunkt erreicht ist oder, dass eine relative Menge Zeit verstrichen ist.

Das Konzept Ereignis lässt sich gut mit dem Konzept Klasse vergleichen. Ereignisse und Klassen repräsentieren jeweils einen gewissen Typus, von dem sich konkrete Instanzen bilden lassen. Klassen haben Attribute, die in jeder Instanz der Klasse andere Werte annehmen können. Ereignisse haben Parameter. Wie Klassen lassen sich auch Ereignisse in Hierarchien anordnen, denen Verallgemeinerung oder Erweiterung zu Grunde liegt.

Zustände. Ein Zustand repräsentiert einen Zeitraum im Leben eines Objekts. Zustände lassen sich auf drei komplementäre Arten charakterisieren: Ein Zustand kann ein Zeitraum sein, in dem das Objekt auf das Auftreten eines bestimmten Ereignisses wartet, oder in dem eine bestimmte Aktion von dem Objekt ausgeführt wird. Außerdem lassen sich Zustände über die Werte von Attributen des Objekts charakterisieren.

Es lassen sich neben den einfachen Zuständen zwei weitere Typen unterscheiden: *zusammengesetzte Zustände* (*composite states*) und *Pseudozustände* (*pseudostates*). Ein zusammengesetzter Zustand beinhaltet eine Reihe von Teilzuständen, die entweder der Reihe nach (*sequential substates*) oder parallel (*concurrent substates*) aktiviert werden. Zu den Pseudozuständen gehören die *Start-* und *Endzustände* (*initial state, final state*). Diese dienen zum initialisieren beziehungsweise terminieren des Kontrollflusses innerhalb eines Zustandsautomaten.

Ein Zustand verfügt über mehrere Attribute:

Name: Zustände können Namen haben, oder anonym sein. Es werden häufig Nomen oder kurze Phrasen gewählt, die aus dem Vokabular des Anwendungsbereichs stammen, für den das beschriebene System entworfen wird.

Eintritts- / Austrittsaktion (*entry/exit action*): Die Eintrittsaktion wird beim aktivieren eines Zustands ausgeführt, die Austrittsaktion beim deaktivieren (verlassen) eines Zustands.

Interne Transitionen (*internal transitions*): Eine interne Transition verhält sich wie ein gewöhnliche Transition, mit dem Unterschied, dass eine interne Transition keinen definierten Zielzustand hat und kein Zustandswechsel bei ihrer Auslösung stattfindet. Das bedeutet, dass keine Eintritts- oder Austrittsaktionen ausgeführt werden. Dies unterscheidet eine interne Transition von einer Selbst-Transition, bei der eine Transition von einem Zustand zu diesem selbst definiert ist.

Teilzustände (*substates*): Ein Zustand, der über Teilzustände verfügt wird als zusammengesetzter Zustand bezeichnet. Ein zusammengesetzter Zustand kann auch als Zustandsautomat betrachtet werden. Ein Zustand, der über keine Teilzustände verfügt wird als einfacher Zustand bezeichnet.

Verzögerte Ereignisse (*deferred events*): Ein Zustand verwaltet eine Liste von Ereignissen, die bei ihrem Auftreten weder bearbeitet werden noch verworfen werden sollten. Falls ein solches Ereignis in dem aktuellen Zustand auftritt, so wird es erst vom nächsten aktiven Zustand verarbeitet.

Aktivitäten (*activities, do-activities*): Während sich ein Objekt in einem bestimmten Zustand befindet kann es eine fortwährende Aktivität ausüben. Diese Aktivität wird nach dem Ausführen der Eintrittsaktion gestartet. Falls währenddessen eine Transition ausgelöst wird, so wird die Aktivität unterbrochen, die Austrittsaktion wird ausgeführt und der aktive Zustand wird deaktiviert.

Ein einfacher Zustand wird in der UML als ein Rechteck mit runden Ecken dargestellt. Startzustände werden durch einen Punkt dargestellt und Endzustände durch einen Kreis, der einen Punkt enthält. Zusammengesetzte Zustände umschließen in der Darstellung ihre Teilzustände.

Transitionen. Eine Transition stellt die Reaktion eines Objektes auf ein Ereignis dar. Jede Transition beginnt und endet jeweils in einem Zustand. Diese Zustände werden als *Quell-* und *Zielzustand* bezeichnet. Eine Transition die in einem Zustand beginnt, definiert die Reaktion des Objekts, das sich in diesem Zustand befindet, auf ein Ereignis. Tritt dieses Ereignis auf, so wird die Transition *ausgelöst* (*fired*). Dies hat einen Wechsel des aktiven Zustands des Automaten zum Zielzustand zur Folge.

Die Attribute einer Transition im Detail:

Auslösendes Ereignis (trigger event): Ein Ereignis eines bestimmten Typs, das für das Auslösen einer Transition verantwortlich sein kann. Tritt das auslösende Ereignis ein *und* ist die Wachbedingung erfüllt, so wird die Transition vollzogen.

Wachbedingung (guard condition): Eine Bedingung, die wahr oder falsch sein kann. Die Transition kann nur vollzogen werden, wenn diese Bedingung wahr ist.

Aktion: Die Aktion die ausgeführt wird, falls die Transition ausgelöst wurde.

Zielzustand: Der potentielle nächste aktive Zustand des Zustandsautomaten. Dieser Zustand wird aktiviert, falls die Transition vollzogen worden ist.

Eine Transition wird in der UML durch einen soliden Pfeil dargestellt, der zwischen dem Quell- und dem Zielzustand der Transition eingezeichnet ist. Der Pfeil lässt sich optional durch einen Transitionsbezeichner beschriften, auf dem bestimmte Attribute der Transition genannt werden können.

3.2.2 Beispiel: Ein Geldautomat

Das in Abbildung 3.1 dargestellte Zustandsdiagramm zeigt das Verhalten eines Geldautomaten [5, S. 251]. Im Startzustand *Leerlauf* wartet der Automat darauf, dass entweder eine Geldkarte eingeführt oder der Wartungs-Modus akitviert wird.

Durch das Ereignis *warten* wird der Zustand *Wartung* aktiviert. Dieser ist für technische Wartungsarbeiten des Geldautomaten gedacht, beispielsweise das Nachfüllen der Bargeldvorräte. Ist die Wartung abgeschlossen, so wird der Zustand automatisch verlassen (die Transition hat weder ein auslösendes Ereignis noch eine Wachbedingung) und der Zustand *Leerlauf* wird aktiviert.

Das Einführen einer Karte in den Automaten wird durch das Ereignis *karteEingeführt* dargestellt. Durch dieses Ereignis wird vom Zustand *Leerlauf* aus eine Transition zum Zustand *Aktiv* ausgelöst. Im dem zusammengesetzten Zustand *Aktiv* werden die Transaktionen eines Kunden bearbeitet. Zunächst wird die Karte durch die Eintrittsaktion *karteLesen* eingelesen. Im Startzustand *Prüfen* werden die Kundendaten überprüft. Im nächsten aktiven Teilzustand *Auswählen* kann der Kunde eine Transaktion wählen, die ausgeführt werden soll. Nachdem eine Transaktion im Zustand *Verarbeiten* durchgeführt wurde, hat der Kunde die Wahl, entweder eine weitere Transaktion durchzuführen oder den Vorgang zu beenden. Diese Möglichkeiten werden durch die Ereignisse *weiter* und *fertig* dargestellt. Tritt das Ereignis *fertig* auf, so wechselt der Automat in den Zustand *Ausgeben* und händigt dem Kunden das Geld aus oder zeigt die gewünschten Daten an. Abschliessend wird automatisch in den Zustand *Leerlauf* gewechselt und die Austrittsaktion *karteAuswerfen* gibt die Karte an den Kunden zurück.

Während der Automat sich im Zustand *Aktiv* oder einem seiner Teilzustände befindet, kann das Auftreten des Ereignisses *abbruch* jederzeit für ein Verlassen des Zustands sorgen. In diesem Fall kehrt der Automat in den Zustand *Leerlauf* zurück.

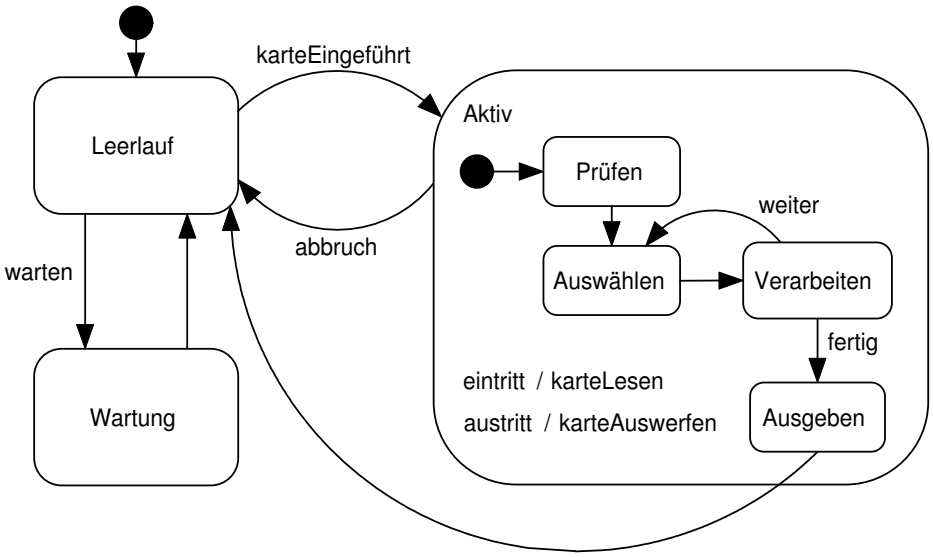

Abbildung 3.1: Das Verhalten eines Geldautomaten als Zustandsdiagramm.

Kapitel 4

Entwicklung von Agenten

Im Folgenden sollen bestehende Ansätze zur Modellierung und Implementierung von Agenten vorgestellt werden. Es wird dabei zwischen *Architektur-Typen*, welche sich in der Vergangenheit bewährt haben, und *Methodologien*, also systematischen Vorgehensweisen für den Entwurf, unterschieden. Sowohl die Architektur-Typen, als auch die Methodologien, sind nicht an eine bestimmte Plattform gebunden und beziehen sich eher auf allgemeine Systeme für Software-Agenten, als auf jene für mobilen Agenten.

4.1 Architektur-Typen

In diesem Abschnitt findet sich eine kurze Übersicht über Architektur-Typen mit deren Hilfe das Verhalten von Agenten implementiert werden kann. Es handelt sich dabei nicht um konkrete Architekturen in dem Sinne, dass sie unmittelbar in der hier beschriebenen Form einsetzbar sind. Die vorgestellten Typen sind eher als Abstraktionen von unterschiedlichen Familien konkreter Architekturen zu betrachten. Die Darstellung orientiert sich an [74].

4.1.1 Logik-Basierende Architekturen

In logik-basierenden Architekturen wird das Verhalten des Agenten durch *logische Deduktion* bestimmt. Verhalten durch logische Deduktion zu modellieren ist ein traditionelle Ansatz in Systemen für *Künstliche Intelligenz*. In diesem Fall liegen die Beschreibung des gewünschten Agentenverhaltens sowie die Beschreibung der Umwelt des Agenten in einer symbolischen Repräsentation vor; beispielsweise als Formeln der *Prädikaten-Logik erster Stufe* [37]. Die vorliegenden Formeln werden syntaktisch manipuliert (Deduktion) um das Verhalten zu bestimmen. Dies stellt den Entscheidungsfindungsprozess des Agenten dar. Es wird dabei also aus der symbolischen Repräsentation des gewünschten Agenten-Verhaltens, das tatsächliche Verhalten,

unter Berücksichtigung der Umgebung, abgeleitet. Abbildung 4.1 zeigt ein Schema dieses Entscheidungsfindungsprozesses.

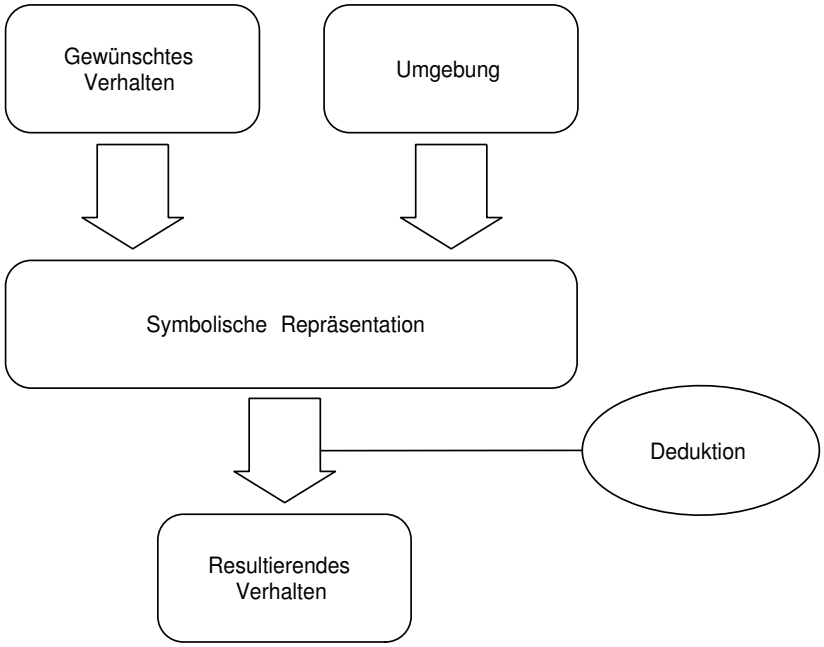

Abbildung 4.1: Verhaltensmodellierung in logik-basierten Architekturen.

Der Vorteil von Systemen, die auf reiner Logik basieren ist, dass die Spezifikation des Verhaltens in einer eleganten Form vorliegt, die eine klare (logische) Semantik aufweist. Außerdem liegt die Spezifikation in einer direkt ausführbaren Form vor. In anderen Systemen wird die Spezifikation häufig in mehreren Schritten bearbeitet, bis schließlich eine ausführbare Implementierung vorliegt.

Nachteilig sind jedoch die Eigenschaften bezüglich der *Rechtzeitigkeit*, mit der ein Ergebnis durch Deduktion gewonnen werden kann. Syntaktische Manipulation ist relativ aufwendig; es kann vorkommen, dass sich die Umgebung des Agenten schneller ändert, als dieser zu einer Entscheidung kommt, die den alten Zustand der Umgebung betrifft. Dies ist vor allem in komplexen, dynamischen Umgebungen ein Problem. Im Falle der Anwendung von Prädikaten-Logik erster Stufe kann es sogar sein, dass die Entscheidungsfindung des Agenten gar nicht terminiert. Das Problem der Rechtzeitigkeit kann behoben werden indem man sich von einer exklusiven Verwendung von logischen Repräsentationen für die Beschreibung des Verhaltens und der Umgebung absieht und gemischte Modelle benutzt. Dadurch verliert man jedoch einen der größten Vorteile einer logik-basierten Architektur: die Eleganz.

Ein anderes Problem ist, dass die Modellierung des Verhaltens durch logische Formeln nicht sehr intuitiv ist. In der Regel ist eine recht spezielle mathematische Grundbildung die Voraussetzung um sich in dieser Form ausdrücken zu können.

4.1.2 Reaktive Architekturen

In reaktiven Architekturen liegt das Verhalten des Agenten in einer Art direkten Abbildung von Situation zu Handlung vor (vgl. Abbildung 2.1). Reaktiv daher, weil der Zwischenschritt des expliziten Nachdenkens über eine gegebene Situation in der Regel wegfällt. Das bedeutet, dass eine reaktive Architektur kein zentrales Weltmodell enthält und außerdem keine komplexen (eventuell symbolischen) Verarbeitungsschritte für die Bestimmung einer Handlung durchgeführt werden. Natürlich muss ein gewisser Zeitrahmen für die Berechnung der Entscheidungen eingeräumt werden, dieser sollte jedoch im Vergleich zum Zeitrahmen des Umgebenden Systems zu vernachlässigen sein. Man kann daher feststellen, dass ein reaktives System rechtzeitig auf Ereignisse reagieren muss.

Es gibt zwei Prinzipien, die reaktiven Architekturen zu Grunde liegen [9]:

- Intelligentes Verhalten ist ein Produkt der Interaktion zwischen Agent und Umgebung.
- Intelligentes Verhalten entsteht durch die Interaktion von einer Reihe einfacherer Grundverhalten.

Die Vorteile von reaktiven Systemen liegen in der Einfachheit des Konzepts und der geringen Komplexität der Berechnungen, die zur Entscheidungsfindung benötigt werden. Nach *Wooldridge* sind die positiven Eigenschaften von reaktiven Architekturen Einfachheit, Ökonomie, Eleganz und Robustheit gegen Fehler [77].

Von Nachteil ist, dass rein reaktive Systeme nicht lernfähig sind. Alle Erfahrungen und Informationen müssen in den Verhaltensbausteinen fest kodiert werden. Da in der Regel kein Umweltmodell gebildet wird, müssen ausreichend viele Informationen aus der lokalen Umgebung des Agenten verfügbar sein.

Das Gesamtverhalten geht *irgendwie* aus der Interaktion der einzelnen Verhalten hervor (*emergentes Verhalten*); wie das genau geschieht ist prinzipiell nicht vorgegeben. Es ist daher ohne die Verwendung weiterer Hilfsmittel nicht klar zu erkennen, wie ein Agent durch pure Reaktivität umgesetzt werden kann. Der Zwischenschritt in Form eines Hilfs-Konzeptes oder einer Methode fehlt.

Zustandsautomaten können als ein solches Konzept betrachtet werden, mit dessen Hilfe sich reaktive Architekturen modellieren und implementieren lassen. Dieser Punkt wird in Abschnitt 8.2 noch eingehend diskutiert werden.

4.1.3 BDI-Architekturen

BDI steht für *belief-desire-intention* (*Glaube-Wunsch-Intention*). In diesen Architekturen wird das Verhalten des Agenten durch die Manipulation von Datenstrukturen bestimmt, welche die

Konzepte Glaube, Wunsch und Intention des Agenten repräsentieren. Diese Architekturen haben ihre Wurzeln in der Philosophie. Die Entscheidungsfindung des Agenten wird dabei an der des Menschen orientiert: Moment für Moment wird entschieden welche Handlungen durchgeführt werden sollen um die gesetzten Ziele zu erreichen. Dabei müssen zwei Fragen beantwortet werden:

- *Was* sind die Ziele, die erreicht werden sollen?
- *Wie* werden diese Ziele erreicht?

Der Vorgang der Beantwortung dieser Fragen wird in BDI-Architekturen durch Funktionen modelliert. Aufgrund der Ansichten des Agenten über sich und seine Umgebung (Glaube), ergeben sich Wünsche, die der Agent erfüllen möchte. Aus den Wünschen gehen die Intentionen hervor, die den Agenten dazu bringen Handlungen auszuführen, durch die er seine Ziele erreichen kann. Insgesamt spielen die Intentionen eine tragende Rolle. Sie sorgen für die Stabilität der Entscheidungsfindung und fokussieren die „Überlegungen“ des Agenten.

Die Funktionsweise einer BDI-Architektur wird nun am Beispiel der *IRMA* (*Intelligent Resource-bounded Machine Architecture*) vorgestellt, welche von *Bratman*, *Israel* und *Pollack* vorgestellt wurde [6]. Die Bausteine der Architektur sind:

- Eine Menge von aktuellen *Glaubens-Daten*, welche die Informationen repräsentieren, die ein Agent über seine Umwelt besitzt.
- Eine Menge von aktuellen *Optionen* (*Wünschen*), die die möglichen Handlungsweisen des Agenten repräsentieren.
- Eine Menge von *Intentionen*. Sie repräsentieren die Fokussierung des Agenten auf ein bestimmtes Ziel, das erreicht werden soll.
- Eine *Glaubens-Revisions-Funktion* (`grf`), die aufgrund der gegebenen *Glaubens-Daten* und dem *sensorischen Input* des Agenten eine Menge von neuen Glaubens-Daten generiert.
- Eine *Options-Generations-Funktion* (`ogf`), welche die Optionen bestimmt die dem Agenten offen stehen (seine *Wünsche*). Die Berechnung erfolgt auf Grundlage der aktuellen Glaubens-Daten und der aktuellen Intentionen.
- Eine *Filter-Funktion* (`filter`), die den Willen des Agenten repräsentiert. Sie bestimmt die *Intentionen* des Agenten auf Grundlage der aktuellen Glaubens-Daten, Wünsche und Intentionen.
- Eine *Ausführungs-Funktion* (`ausführen`), die eine auszuführende Handlung bestimmt. Die geschieht auf Grundlage der Intentionen.

Abbildung 4.2 illustriert den Ablauf der Entscheidungsfindung innerhalb der IRMA-Architektur [74]. Wie man erkennen kann, enthält das System einige Rückkopplungen, welche verschiedene Regel-Werke zwischen den jeweiligen Elementen bilden: Der Agent nimmt Informationen über seine Umwelt und sich selbst auf (`Wahrnehmung`). Dies stellt die Eingabedaten des Systems dar, aus denen schließlich eine Handlung abgeleitet werden soll. Diese Daten werden mit dem bisherigen `Glauben` des Agenten durch die `grf` kombiniert und ergeben eine neue Menge von Glaubens-Daten. Diese fließend sowohl in die Funktion `filter` als auch in `ogf` ein. Die `ogf` erzeugt eine Menge von `Wünschen`, welche in `filter` einfließen. Diese Funktion generiert aus ihren Eingaben die `Intentionen`, welche wiederum in `filter`, `ogf` und die Funktion `ausführen` eingebracht werden. Die Funktion `ausführen` bestimmt dann die Handlung die ausgeführt wird.

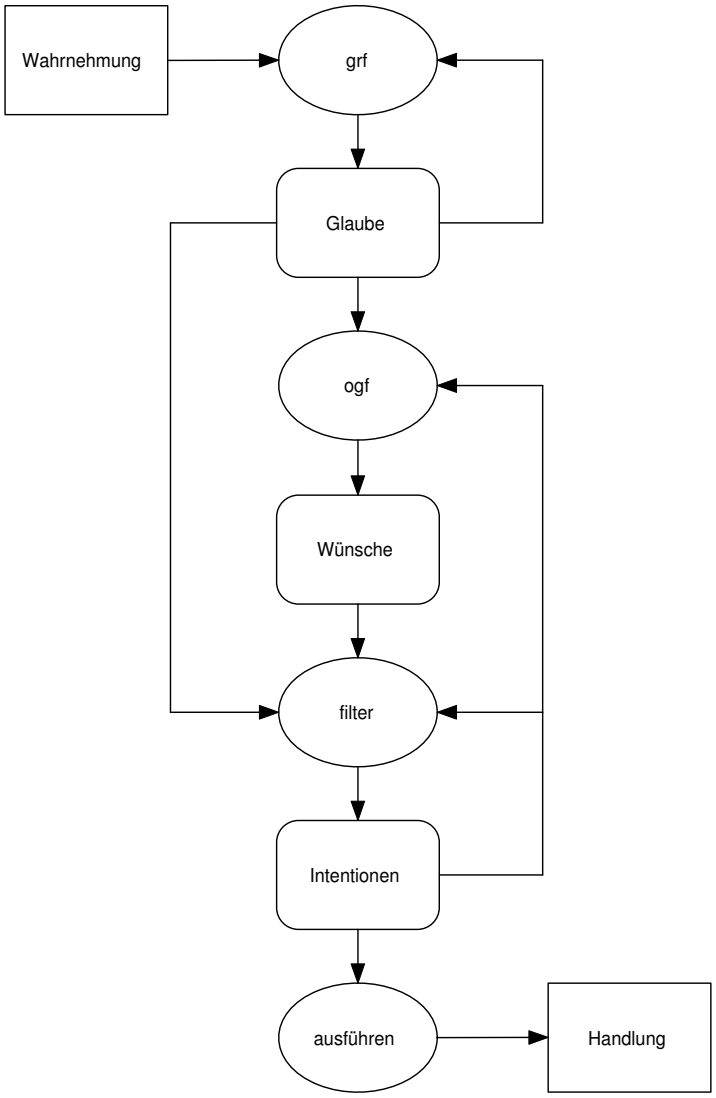

Abbildung 4.2: Die Entscheidungsfindung durch eine BDI-Architektur.

Glaube, Wunsch und Intention werden in konkreten Implementierungen oft durch logische Formeln repräsentiert.

Eine große Herausforderung in BDI-Architekturen ist es, die Richtige Balance zu finden mit der die Intentionen verfolgt werden sollen. Wenn der Agent seine Intentionen zu oft ändert, zum Beispiel falls ein Ziel nicht auf Anhieb erreicht werden kann, ist es möglich, dass er nur wenige oder gar keine seiner Ziele wirklich erreicht. Wenn der Agent seine Intentionen selten oder nie ändert, ist er unter Umständen nicht in der Lage, auf Änderungen in der Umgebung zu reagieren und erreicht womöglich deshalb seine Ziele nicht. Oder anders ausgedrückt: es muss eine Balance zwischen *proaktivem* und *reaktivem* Verhalten gefunden werden (vgl. Abschnitt 2.1).

Das Konzept, auf dem BDI-Architekturen basieren, ist jedoch recht intuitiv. Die Überlegungen darüber, was zu tun ist und wie es zu tun ist sind leicht nachvollziehen. Ein informales Verständnis der Konzepte Glaube, Wunsch und Intention ist bei den meisten Menschen vorhanden. Weiterhin liefert das Modell eine Form von *funktionaler Dekomposition*. Es ist daher leicht nachvollziehbar, welche Subsysteme benötigt werden, um einen BDI-Agenten zu konstruieren.

Problematisch ist jedoch die Implementierung der einzelnen Funktionen in effizienter Weise. Dies trifft im besonderen auf die Verwendung von logischen Formeln zur Implementierung der BDI-Elemente zu (vgl. Abschnitt 4.1.1). Insgesamt lässt sich noch festhalten, dass der Aufwand, der betrieben werden muss um ein BDI-System zu implementieren relativ hoch ist. Für Agentenverhalten von niedriger Komplexität ist sicherlich ein reaktiver Ansatz günstiger zu realisieren (vgl. Abschnitt 4.1.2).

4.2 Methodologien

Eine Methodologie lässt sich als wiederverwendbarer Prozess für die Analyse und den Entwurf eines Systems auffassen. Durch die Anwendung einer Methodologie, wird aus einem groben Entwurfskonzept oder einer Idee, Schritt für Schritt ein detaillierteres Modell entwickelt. Während der Anwendung eines solchen Prozesses, werden in der Regel gewisse Artefakte vom Entwickler erstellt, die das Modell fixieren. Ein guter Überblick existierender Methodologien findet sich in [32, 69]. Im folgenden Abschnitt werden einige für diese Arbeit relevante Ansätze diskutiert. Die hier vorgestellten Methodologien sind iterative *top-down* Verfahren für die Modellierung und Entwicklung von agenten-basierten Systemen.

Ein häufig verwendetes Konzept in Methodologien für agenten-basierte Systeme, ist das der *Rolle*. Eine Rolle lässt sich als eine abstrakte Beschreibung der Funktion auffassen, die ein Agent in einem System übernimmt. Man könnte auch den Begriff des Amtes verwenden, das von dem Agenten ausgeübt wird. Agenten werden bestimmten Rollen zugeordnet. Dabei kann ein Agent mehrere Rollen übernehmen und die Zuordnung der Rollen kann sich zu einem späteren Zeitpunkt ändern. Ein Agent kann also zwischen verschiedenen Rollen hin und her wechseln.

4.2.1 Gaia

Die Gaia[1] Methodologie für agenten-orientierte Analyse und Entwurf wurde von *Wooldridge*, *Jennings* und *Kinny* vorgestellt [76]. Gaia ist eine Methodologie, die die Entwicklung von Agenten sowohl auf der *Mikro-Ebene* als auch auf der *Makro-Ebene* unterstützt. Auf der Mikro-Ebene geht es um den Entwurf der Struktur des Agenten selbst, währen auf der Makro-Ebene der Entwurf der Agenten-Gesellschaft und Organisationen im Vordergrund steht. Dabei werden vor allem der Aspekt der Autonomie und der problemlösende Charakter von Agenten berücksichtigt, sowie die Interaktion und Organisation von Agenten. Durch die Verwendung von Gaia können Software-Entwickler Spezifikationen für Agenten-Systeme entwickeln, die leicht implementiert werden können.

Die Idee ist, von einem abstrakten Verständnis des geforderten Systems und dessen Struktur in mehreren hierarchischen Schritten auf eine konkretere Spezifikation des Systems hinzuarbeiten. Das niedrigste Abstraktionsniveau, das nach der Anwendung von Gaia erreicht wurde, kann dann durch Anwendung von traditionellen Entwurfstechniken in eine geeignete Implementierung umgesetzt werden. Zu diesen Techniken gehören zum Beispiel Ansätze aus dem objekt-orientierten Software Engineering. Abbildung 4.3 illustriert die hierarchischen Verarbeitungsschritte von Gaia [76].

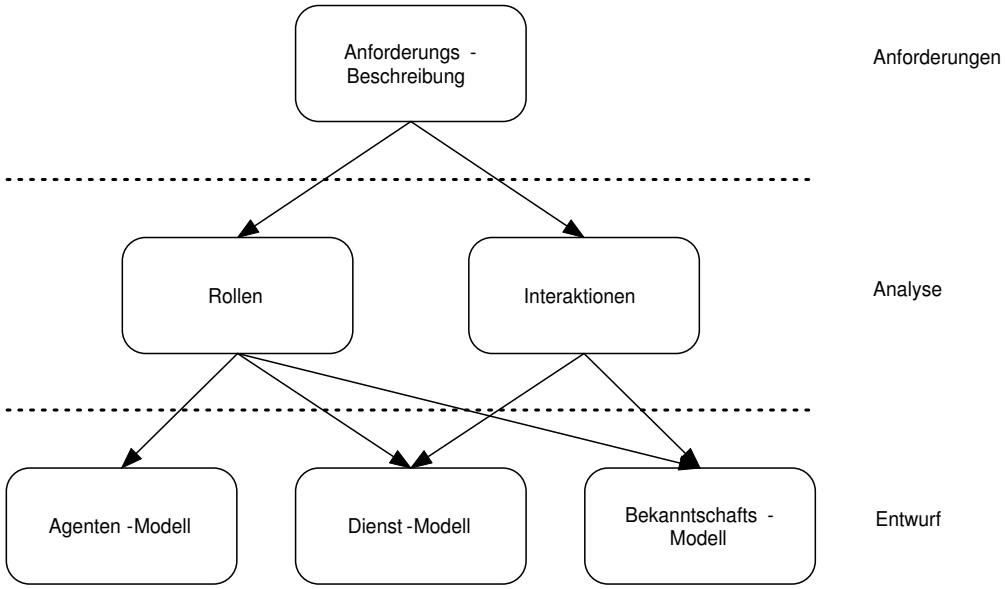

Abbildung 4.3: Die Beziehungen zwischen den Modellen Gaias.

Gaia lässt sich in zwei Phasen unterteilen: *Analyse* und *Entwurf*. In der Analyse-Phase geht es darum ein Verständnis für die Organisation des Systems zu entwickeln. Diese Organisation wird durch eine Menge von Rollen dargestellt, welche in einer bestimmten Weise mit einander

[1] In der griechischen Mythologie die Erde in Göttergestalt.

Interagieren. In der Analyse werden als erstes die *Rollen* innerhalb des Systems identifiziert. Danach werden die *Interaktionen* zwischen diesen Rollen definiert.

Eine Rolle besitzt in Gaia vier Attribute:

Verantwortlichkeiten: Sie bestimmen die Aufgabe der Rolle innerhalb des Systems. Es liegt in der Verantwortung einer Rolle, etwas positives zu dem System beizutragen (*liveness*) oder zu verhindern, dass etwas passiert, das einen negativen Effekt auf das System hat (*safety*).

Befugnisse: Diese regeln welcher Handlungen einer Rolle gestattet sind und welche nicht. Insbesondere werden Zugriffsrechte auf Informationen geregelt.

Aktivitäten: Dies sind Aufgaben, die eine Rolle unabhängig erledigt, ohne dabei mit anderen Rollen zu interagieren.

Protokolle: Sie stellen spezifische Muster der Interaktion zwischen Rollen dar. Es wird der Ablauf der Interaktion zwischen Rollen geregelt, die ein gemeinsames Ziel erreichen wollen. Eine spezifische Klasse stellen beispielsweise Kommunikations-Protokolle dar.

Gaia stellt formale Operatoren und Vorlagen zur Verfügung, um Rollen und deren Attribute zu repräsentieren. Außerdem existieren Schemata für die Interaktion zwischen Rollen.

In der Entwurfs-Phase von Gaia wird zunächst das *Agenten-Modell* spezifiziert. Das Modell dokumentiert die verschiedenen *Agenten-Typen*, die zur Entwicklung des Systems verwendet werden, sowie die *Agenten-Instanzen*, welche diese Typen zur Laufzeit realisieren werden. Dafür werden die erstellten Rollen bestimmten Agenten-Typen zugeordnet. Das kann in einem eins zu eins Verhältnis geschehen, es ist jedoch auch möglich, dass ein Agent mehrere Rollen übernimmt. Es entsteht ein gerichteter, azyklischer Graph, in dem die Knoten die Agenten-Typen repräsentieren und die Blätter die Agenten-Rollen. Ein Agent setzt sich dabei aus denn Rollen seiner Nachfahren zusammen.

Beispiel 4.1 *In Abbildung 4.4 wird ein solcher Graph dargestellt: Agenten-Typ 1 hat die Nachkommen Typ 2 und Typ 3, das heißt Typ 1 setzt sich aus den Rollen A, B und C zusammen.*

Für jeden Agenten-Typ wird schließlich die Anzahl der Instanzen notiert, die später zur Laufzeit erzeugt werden sollen.

Anschließend wird das *Dienst-Modell* bestimmt, das benötigt wird, um eine Rolle in einem oder mehreren Agenten zu erfüllen. Das Dienst-Modell beschreibt die konkreten Funktionen eines Agenten. Diese Funktionen lassen sich in etwa mit den Methoden der Objekte aus der objekt-orientierten Programmierung vergleichen. Dienste stehen jedoch nicht in gleicher Weise anderen Agenten zu Verfügung, wie das Methoden von Objekten für andere Objekte tun. Ein

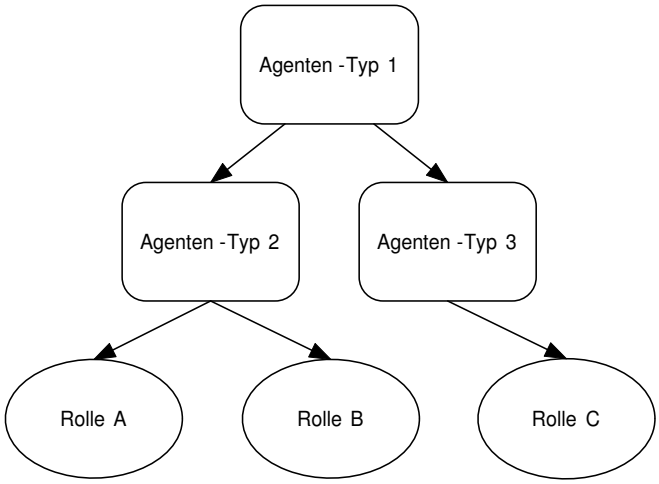

Abbildung 4.4: Beispiel der Hierarchie der Agenten-Typen und Rollen in Gaia.

Dienst definiert eine zusammenhängende Aktivität, die ein Agent ausübt. Jede der Aktivitäten, die während der Erstellung der Rollen identifiziert wurden, wird auf einen Dienst abgebildet. Für jeden Dienst werden die Eingaben, Ausgaben, *pre-* und *post-conditions* festgelegt.

Der letzte Schritt besteht darin, ein *Bekanntschafts-Modell* zu erstellen, das die Kommunikation zwischen den Agenten repräsentiert. Dabei werden lediglich die Kommunikationsverbindungen zwischen den Agenten definiert, nicht jedoch die Nachrichten, die zwischen ihnen versendet werden sollen. Das Modell besteht aus einem gerichteten Graphen, in dem die Knoten die Agenten-Typen repräsentieren und die Kanten die Kommunikationswege zwischen den Agent-Typen darstellen.

Eine wichtige Beobachtung ist, dass die Fähigkeiten der Agenten sowie deren Organisation, also ihre Beziehungen untereinander, zur Kompilierzeit festgelegt werden. Sie sind also während der Laufzeit *statisch*. Da Gaia in dieser Weise konzipiert wurde, ist es für den Einsatz in unzugänglichen und dynamischen Umgebungen weniger geeignet. In zugänglichen oder statischen Umgebungen hat Gaia sich jedoch als brauchbarer Ansatz erwiesen.

Wie eingangs erwähnt, arbeitet die Anwendung von Gaia nicht auf eine implementierungsfähige Modellierung des Agenten-Systems hin. Die Analyse-Modelle werden nur soweit auf ein niedrigeres Abstraktionsniveau transformiert, dass die Umsetzung auf eine Implementierung durch traditionelle Entwicklungsmethoden durchgeführt werden kann. Wie genau die Dienste, Rollen und Interaktionen eines Agenten erfüllt werden, also wie das spezifizierte Verhalten umgesetzt wird, ist nicht Bestandteil des Entwurfs mit Gaia. Nach Meinung der Autoren hängt dies von dem konkreten Anwendungsgebiet des entworfenen Systems ab.

4.2.2 MaSE

Die *Multiagent Systems Engineering Methodology* (MaSE) wurde von *Wood* und *DeLoach* vorgestellt [73]. Bezüglich des Anwendungsgebietes und der allgemeinen Verwendbarkeit ist diese Methodologie sehr ähnlich zu *Gaia*. Auch MaSE dient zur Entwicklung von allgemeinen Multi-Agenten-Systemen. Anders als Gaia geht diese Methodologie jedoch Stärker auf die Implementierung eines Systems ein. Der Entwicklungsprozess wird von der ersten System-Spezifikation bis zur konkreten Implementierung der einzelnen Agenten von MaSE unterstützt. Die Entwurfsmodelle werden zum größten Teil in Diagrammen der UML fixiert.

In der Analyse-Phase werden die Entwurfsziele des Systems festgelegt, aus welchen sich die Anwendungsfälle (*Use Cases*) ergeben. Diese Anwendungsfälle werden durch Sequenzsdiagramme weiter verfeinert. Schließlich werden die im System vorhandenen Rollen bestimmt. Die Aufgaben, welche von einer Rolle übernommen werden können, werden durch Zustandsdiagrammen spezifiziert.

In der Design-Phase werden die Agenten-Klassen definiert und zusammengestellt, dabei kommen spezielle Klassendiagramme für Agenten, sowie Zustands- und Verteilungsdiagramme zum Einsatz. Weiterhin wird die Kommunikation zwischen Agenten durch Zustandsdiagramme beschrieben. Aus den Klassen- und Zustandsdiagrammen werden schließlich die Verteilungsdiagramme für das vollständige System abgeleitet.

Für die Implementierung der Agenten steht ein Werkzeug, das *agentTool*, zur Verfügung, mit dessen Hilfe sich aus den MaSE-Modellen ausführbare Programme erzeugen lassen. Das Ziel der Methodologie ist es, diese Generierung von Code soweit wie möglich zu automatisieren; insbesondere soll es später einmal möglich sein, Code direkt aus den erstellten Verteilungsdiagrammen zu erzeugen.

Auch die Beschränkungen von MaSE sind sehr ähnlich wie jene von Gaia: Das entwickelte System muss abgeschlossen sein, das heißt alle Fähigkeiten der Agenten sowie deren Beziehungen untereinander sind statisch zur Laufzeit festgelegt. Weiterhin müssen alle externen Schnittstellen, die während der Interaktion der Agenten eingesetzt werden, durch Agenten gekapselt sein. Dynamische Systeme, in denen Agenten zur Laufzeit erzeugt, zerstört oder bewegt werden können, werden nicht von MaSE berücksichtigt; insbesondere können also keine Systeme für mobile Agenten entworfen werden. Ein weiterer Punkt ist, dass die Kommunikation zwischen Agenten in einem eins-zu-eins Verhältnis stattfinden muss. Ein gleichzeitiges Versenden von Nachrichten zwischen mehreren Parteien (*multicasting*) ist also nicht möglich.

4.2.3 Agenten-UML

Die *Universal Modeling Language* (*UML*) [5] dient zur graphischen Repräsentation des Entwurfs von fast allen Teilen eines objekt-orientierten Systems. Es gibt einige Ansätze zur Modellierung von Agenten (-Systemen), die UML verwenden.

Man kann zwischen zwei grundlegenden Vorgehensweisen unterscheiden. Erstens, die direkte Nutzung von UML um agenten-orientierte Modelle in objekt-orientierte Modelle zu transformieren. Dies geht in der Regel mit einer veränderten Interpretation der Semantik der UML einher. Der Vorteil ist, dass Entwickler vorhandene UML-basierte Werkzeuge benutzen können um die Spezifikationen für ihr System zu erstellen. Außerdem können sie ihre Erfahrung im Entwurf von objekt-orientierten Systemen direkt einsetzen. Zweitens, die Erweiterung von UML durch speziell an Agenten angepasste, graphische Konstrukte und anschließende Nutzung zur Modell-Transformation. Der Vorteil hier ist, dass die UML-Modifikation für das Anwendungsgebiet maßgeschneidert werden kann.

Odell, *Parnuak* und *Bauer* schlagen ein Modell mit drei Schichten zur Repräsentation von *Agenten-Interaktions Protokollen* (*AIPs*) vor [50], das eine erweiterte UML als Beschreibungssprache verwendet. AIPs werden als Entwurfsmuster definiert, die sowohl die Kommunikation zwischen Agenten repräsentieren, als auch Rahmenbedingungen für die Inhalte der verwendeten Nachrichten spezifizieren. Die Kommunikation wird dargestellt, als eine Reihe zulässiger Nachrichten, die zwischen Agenten ausgetauscht werden können. Das AIP-Modell ist keine Methodologie an sich, sondern eine Beschreibungssprache für Agenten-Interaktionen, die unabhängig von einem bestimmten Entwurfsprozess ist. Dieser Ansatz benötigt Änderungen an der UML selbst und nicht nur an deren Semantik. Folgende UML-Bestandteile wurden angepasst: Pakete, Templates, Sequenzdiagramme, Kollaborationsdiagramme, Aktivitätsdiagramme und Zustandsdiagramme.

Die drei AIP-Schichten sind hierarchisch angeordnet: Die erste Schicht beinhaltet die zweite und diese wiederum die dritte Schicht. In der *ersten Schicht* werden die Typen der Protokolle mit Hilfe von Paketen und Templates definiert. Ein Paket repräsentiert ein Protokoll als eigenständige Einheit. Derart verpackt, kann ein Protokoll als konfigurierbare und wiederverwendbare Komponente verstanden werden. Wie die Konfiguration des Protokolls zu erfolgen hat, wird über ein Template festgelegt. Dieses bestimmt, wie die variablen Protokoll-Parameter zur Laufzeit gesetzt werden sollen.

Die *zweite Schicht* repräsentiert die Interaktionen zwischen Agenten durch die Verwendung von Sequenz-, Kollaborations- , Aktivitätsdiagrammen, sowie Zustandsdiagrammen. Hier wird im Prinzip eine Art Skelett des Protokolls konstruiert. Es wird festgelegt, wer mit wem Kommuniziert und welche Nachrichten verschickt und empfangen werden sollen.

In der *dritten Schicht* wird das interne Verhalten der Agenten festgelegt. Hier werden die Prozesse spezifiziert, die innerhalb des Agenten ablaufen um die Protokolle zu implementieren an denen der Agent teilnimmt. Dabei kommen Aktivitätsdiagramme und Statecharts zum Einsatz.

Abbildung 4.5 stellt die drei Schichten im Zusammenhang dar: Die erste Schicht wird dabei durch ein Protkoll-Paket dargestellt, das durch ein Template parametrisiert wird. Schicht zwei wird durch ein Sequenzdiagramm dargestellt, in dem zwei Agenten-Rollen mit einander kommunizieren. Die dritte Schicht wird durch ein Zustandsdiagramm verkörpert, dass die Interna

einer der beiden Rollen definiert. Die Darstellung ist sehr abstrakt gewählt und die Änderungen der UML-Syntax durch AIP gehen nicht aus der Abbildung hervor.[2]

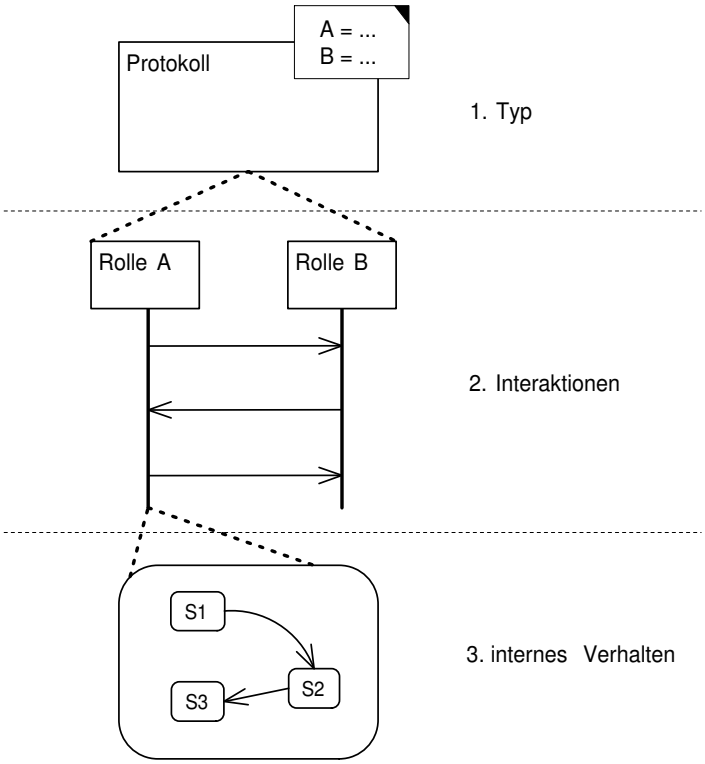

Abbildung 4.5: Die Hierarchie der Schichten in AIP.

In einer Erweiterung von AIP wird UML noch weiter modifiziert, um noch mehr Aspekte des agenten-orientierten Entwurfs zu berücksichtigen. Dieser Ansatz wird als *Agent UML* (*AUML*) bezeichnet [49]. AUML unterstützt unter Anderem eine übersichtlichere Darstellung Agenten-Rollen, also den verschiedenen Rollen, die ein Agenten während der Interaktion mit anderen Agenten übernimmt.

Weiterhin wurde der Begriff der Mobilität mit in die Verteilungsdiagramme aufgenommen. Zwei Dinge werden vorgeschlagen:

- *at-home* Markierungen, die das Start- oder Heimat-System des Agenten bezeichnen.
- Kennzeichnen von Migrationspfaden durch entsprechende Pfeile *(mobile*).

Das folgende Beispiel illustriert die Integration der Mobilität:

Beispiel 4.2 *Abbildung 4.6 stellt ein Verteilungsdiagramm dar in dem die Mobilität zum Ausdruck kommt. Zu sehen sind zwei Systeme: Ein Internet-Server auf dem sich Einkäufe tätigen*

[2] Details zu AIP findet man in [50].

lassen und der Laptop eines Anwenders. Der Anwender möchte mittels eines mobilen Agenten auf diesem Server einkaufen. Der Einkaufs-Agent wird daher auf beiden Systemen dargestellt, auf dem Laptop jedoch mit der at-home Markierung. Der Migrationspfad des Agenten wird durch einen Doppel-Pfeil mit der Markierung mobile dargestellt. Der Einkauf selbst wird durch die beiden Agenten getätigt: Dem Einkaufs-Agent des Anwenders und dem Verkaufs-Agent, der sich auf dem Server befindet.

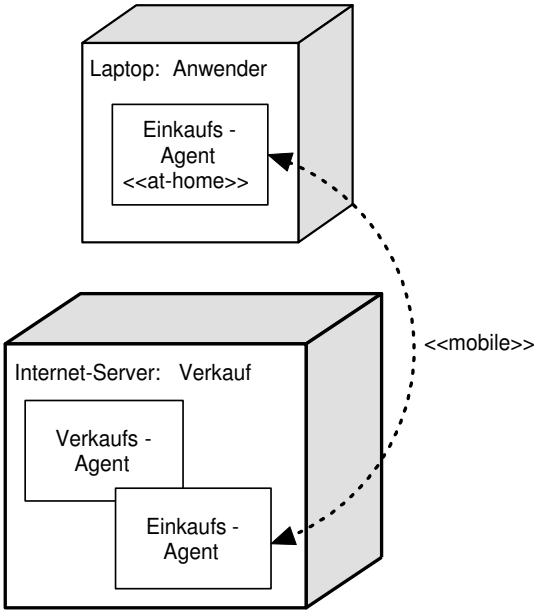

Abbildung 4.6: Die Integration der Mobilität in Verteilungsdiagrammen.

Kapitel 5

Verwandte Arbeiten

In diesem Kapitel werden einige existierende Lösungen für die Modellierung und Implementierung von Agenten vorgestellt, die mit dem in dieser Arbeit entwickelten Ansatz relativ eng verwandt sind. Jede der Arbeiten wird bezüglich der Zielsetzung dieser Arbeit untersucht und bewertet. Ein umfassender Überblick über weitere Ansätze, die im Rahmen von Multi-Agenten-Systemen entwickelt wurden, findet sich in [17].

5.1 ADK

Das *ADK* (*AgentBean Development Kit*) stellt ein Framework bereit, das die komponentenbasiert Konstruktion von mobilen Agenten unterstützt [26]. Die Idee des ADK ist, Agenten aus einer Reihe von wohldefinierten Klassen von Software-Komponenten zusammenzusetzen. Die Konzeption dieser Komponenten soll eine modulare Definition erlauben, die Konstruktion von Agenten erleichtern und eine Wiederverwendung der Komponenten ermöglichen. Diese Komponenten implementieren das *Java Bean* Konzept und lassen sich in drei Kategorien einteilen:

Navigatoren (*navigational components*): Die Komponenten dieser Kategorie sind verantwortlich für die Bestimmung und Ausführung der Reiseroute des Agenten. Diese Route kann dabei entweder statisch in Form einer festen Liste gegeben sein, oder vom Agenten zur Laufzeit bestimmt werden.

Ausführer (*performers*): Die Aufgabe dieser Komponenten ist es, bestimmte Aufgaben auszuführen, die an den jeweiligen Punkten der Reiseroute definiert sind. Es lassen sich mehrere solcher Komponenten verbinden um eine gegeben Aufgabe zu erfüllen.

Berichter (*reporters*): Diese Kategorie enthält Komponenten, welche dafür sorgen, dass die vom Agenten erzielten Resultate an der vorgegeben Stelle eintreffen. Eine solche Kom-

ponente könnte beispielsweise Nachrichten an den Anwender des Agenten über das Netzwerk senden, oder diese Nachrichten sammeln und bei der Rückkehr übergeben.

Die Interaktion zwischen einzelnen Komponenten geschieht in einer reaktiven Weise. Ein Ereignis, das von einer Komponente generiert wird, kann in einer anderen Komponente eine Handlung auslösen.

Abbildung 5.1 stellt das Schema der Interaktion der drei Kategorien dar: Nach jeder Migration sendet der *Navigator* das Ereignis *migrationAbgeschlossen* an den *Ausführer*. Nachdem diese die spezifizierten Aktionen abgeschlossen hat, wird das dem *Berichter* durch das Ereignis *aktionAusgeführt* mitgeteilt. Der *Berichter* kann dann die angefallenen Resultate der Aktionen weiterleiten oder für den späteren Gebrauch speichern.

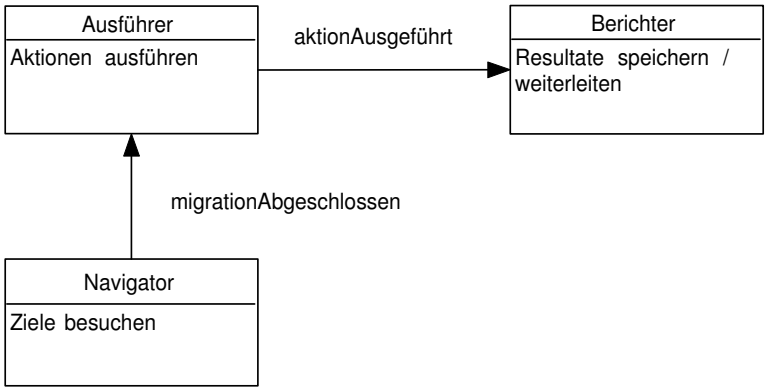

Abbildung 5.1: Die Komponenten-Kategorien des ADK im Zusammenhang.

Konkrete Ausprägungen der einzelnen Komponenten-Typen werden vom Entwickler in einer beliebigen Weise implementiert. Außer der vorgeschriebenen Verwendung des Bean-Konzeptes hat das ADK keinen Einfluss auf diesen Vorgang. Ist die Implementierung abgeschlossen, so lassen sich die Komponenten mit dem ADK bearbeiten. Aufgrund der Anwendung des Bean-Konzeptes lassen sich die vorhandenen Komponenten innerhalb einer *BeanBox* auf graphischem Wege zusammenstellen [27]. Wenn die Konstruktion abgeschlossen ist, erstellt das ADK automatisch den Code des Agenten. Zunächst werden die Beans instanziert und konfiguriert, anschließend werden die Kommunikationsverbindungen zwischen diesen Beans initialisiert.

Die Einteilung in die drei vorgeschlagenen Kategorien stellt eine starke (und beabsichtigte) Einschränkung der Komplexität und der Anwendungsgebiete für die erstellten Agenten dar. Nach *Gschwind et al.* ist das ursprüngliche Anwendungsgebiet von ADK-basierten, mobilen Agenten, das Management und die Wartung von Netzwerken und Systemen [26]. Aufgrund der Flexibilität des Ansatzes lässt sich das ADK auch auf anderen Gebieten einsetzen. Diese müssen nach Meinung des Autors jedoch einen hohen Verwandtschaftsgrad zu dem ursprünglichen Anwendungsgebiet aufweisen. Außerdem sollte auch die benötigte Komplexität der Agenten etwa auf dem gleichen Niveau liegen.

Der Modellierungs-Aspekt des ADK setzt auf einem recht niedrigen Abstraktionsniveau an, dennoch wird der Weg bis zur Implementierung der Komponenten nicht vollstänig überbrückt. Außer der Verwendung von Java Beans als Implementierungsgrundlage gibt es kein Konzept, das die Modellierung oder Implementierung der einzelnen Komponenten unterstützt.

Die Verwendung von Java Beans stellt einen bekannten und anerkannten Ansatz zum komponenten-orientierten Entwurf von Software dar. Die dadurch gewonnene Konfigurierbarkeit und die Möglichkeit zur graphischen Konstruktion machen den gesamten Ansatz sehr praktikabel.

Das ADK ist konzeptionell an keine bestimmte Agenten-Plattform gebunden, sondern wurde im Gegenteil, explizit im Sinne der Interoperabilität, der entstehenden Komponenten entworfen. Da außerdem Java als Sprachgrundlage verwendet wird, lässt sich der Ansatz prinzipiell sehr gut auf SeMoA übertragen.

5.2 JADE

JADE (*Java Agent Developement Framework*) ist ein Multi-Agenten-System, das von TILAB entwickelt wird. Eines der Hauptziele dieser Plattform ist es, die leichte Implementierung von Multi-Agenten Gesellschaften zu ermöglichen, deren Agenten gemäß der FIPA-Spezifikationen[1] interagieren. Wie auch SeMoA ist JADE vollständig in Java implementiert und als *Open Source* verfügbar[2].

JADE stellt sowohl ein Framework für die Modellierung des Verhaltens von Agenten, als auch eine Agenten-Basisklasse zur Verfügung. Im Folgenden soll beides näher betrachtet werden.

5.2.1 Das Behaviour-Framework

Das *Behaviour-Framework* von JADE ist eine Hierarchie von Hilfsklassen, die einen reaktiven Charakter aufweisen. Im Behaviour-Framework lässt sich jede Handlung eines Agenten als ein eigenes Objekt modellieren. Für die Implementierung von Handlungen steht eine abstrakte Basisklasse bereit. Die Granularität der Handlungen ist dabei dem Programmierer des Agenten überlassen. Es lassen sich sowohl einzelne Methodenaufrufe als auch ganze Kommunikationsprotokolle in einem Behaviour-Objekt unterbringen.

In JADE existiert eine abstrakte Basisklasse (`Agent`), die jeder JADE Agent erweitern muss[3], der das Framework nutzen soll. Diese Klasse besitzt eine `addBehaviour` Methode über die sich Behaviour-Objekte dynamisch zu einem Agenten hinzufügen lassen. Ein hinzugefügtes

[1] http://www.fipa.org

[2] http://jade.tilab.com

[3] Die Subtypen-Beziehung zwischen einem konkreten Agenten und der `Agent` Klasse wird jedoch vom eigentlichen JADE-System nicht gefordert.

Behaviour wird dann zusammen mit anderen eventuell hinzugefügten Behaviour-Objekten von einem *Scheduler* verwaltet, der in die Klasse `Agent` integriert ist.

Der Scheduler simuliert eine semi-parallele Ausführung der einzelnen Behaviour-Objekte nach dem *round-robin* Schema. Der Scheduler wird verwendet, da in JADE jedem Agenten konzeptionell nur ein Prozess zur Verfügung steht. Multi-Prozess Agenten lassen sich natürlich implementieren, dies wird jedoch nicht direkt vom Framework unterstützt. Der Scheduler ruft der Reihe nach auf jedem Behaviour-Objekt des Agenten die `action` Methode auf. Sobald diese Methode beendet ist, wird die Methode `done` vom Scheduler aufgerufen um zu herauszufinden ob die Handlung vollständig abgeschlossen wurde. Falls ja, so wird das Behaviour-Objekt aus der Scheduler Liste gelöscht, ansonsten arbeitet der Scheduler weiter mit diesem Objekt und `action` wird früher oder später erneut aufgerufen[4]. Abbildung 5.2 illustriert den Kontrollfluss zwischen Agent, Scheduler und Behaviour.

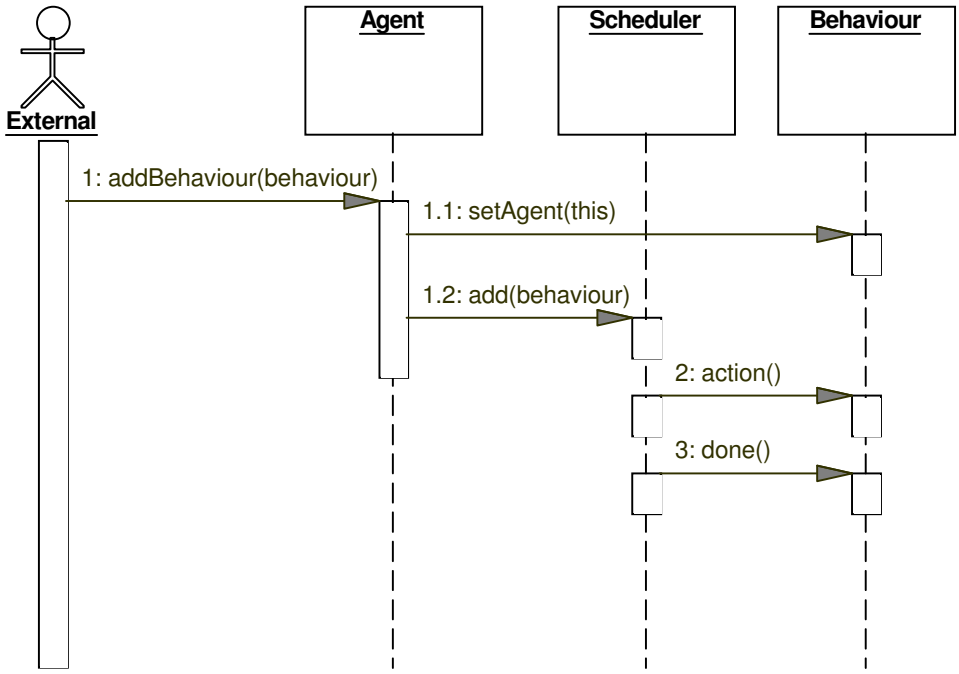

Abbildung 5.2: Ein UML Sequenzdiagramm

Neben den atomaren Behaviour Objekten unterstützt JADE außerdem die Komposition von Behaviour Objekten[5]. Dadurch lassen sich komplexe Handlungen aus einfacheren zusammensetzen. Im Framework sind einige Subklassen von `Behaviour` verfügbar, die ihrerseits Behaviour

[4] Da es in Java nicht möglich ist auf den *execution stack* zuzugreifen, gibt es keine Möglichkeit ein Behaviour während der Ausführung seiner `action` Methode zu unterbrechen und später fortzusetzen.

[5] Es wird das Entwurfsmuster *Composite* verwendet [21, S. 163-173].

Objekte enthalten können und diese nach einem bestimmten Schema ausführen. Abbildung 5.3 gibt eine Übersicht über die wichtigsten Klassen aus dem *Behaviour Framework*.

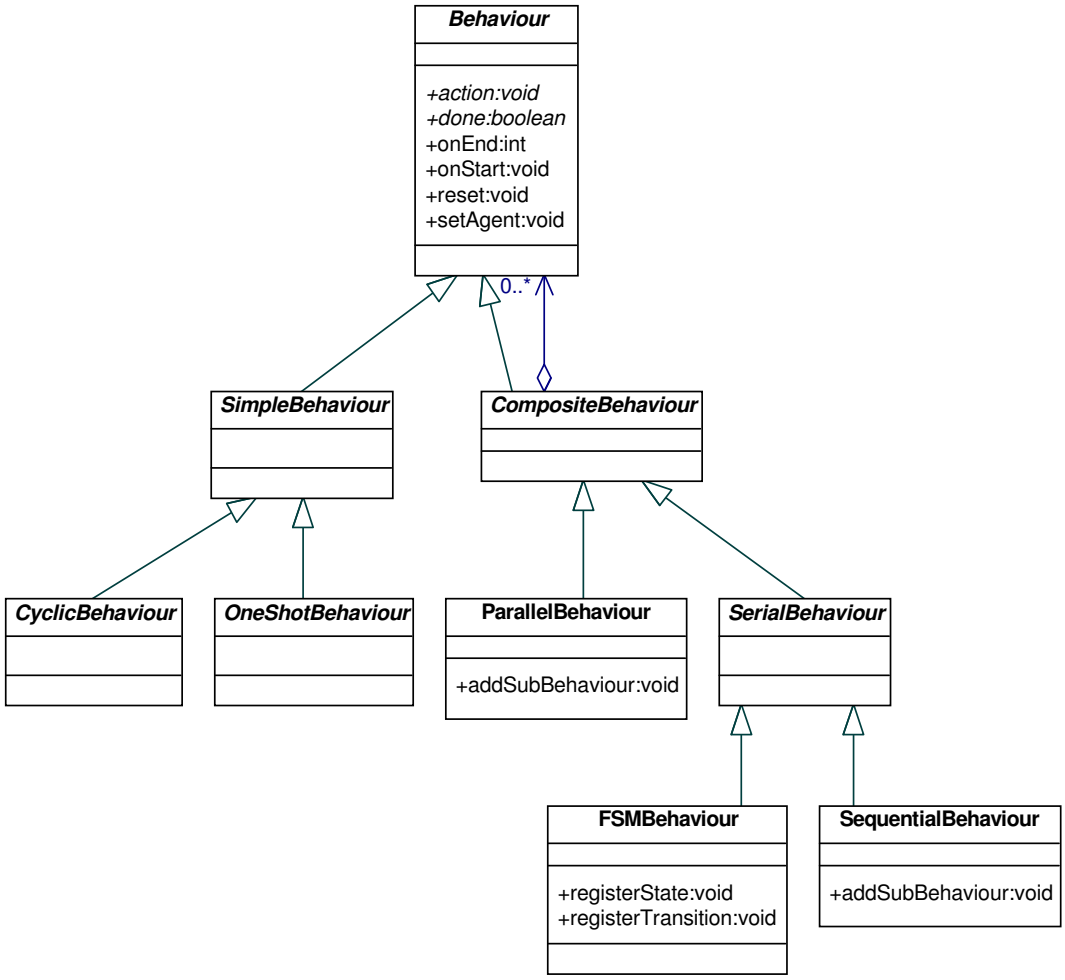

Abbildung 5.3: Ein vereinfachtes Klassendiagramm für das *Behaviour Framework*.

Behaviour: Diese abstrakte Klasse dient als Basis für die Implementierung der Handlungen von Agenten. Sie definiert verschiedene Methoden, deren Semantik ähnlich den Methoden des UML-Konzeptes *Zustand* ist. Die Methoden `onStart` und `onEnd` werden beim Aktivieren und Deaktivieren des Behaviours aufgerufen. Die `action` Methode verkörpert den Kern des Behaviours. Hier wird der Programmcode spezifiziert, welcher die Handlung darstellt. Die `reset` Methode dient zum Wiederherstellen des Ausgangszustandes eines Behaviours. Weiterhin sind einige Methoden für das Setzen und Abrufen eines Datenspeichers vorhanden. Dieser Datenspeicher kann nützlich sein für den Austausch von Daten zwischen Behaviour-Objekten. Diese Klasse definiert auch die Schnittstelle, die für die Verwendung in einem Scheduler benötigt wird.

SimpleBehaviour: Dient als abstrakte Basisklasse für atomare Handlungen.

OneShotBehaviour: Eine atomare Handlung, die nur einmal ausgeführt werden darf.

CyclicBehaviour: Eine atomare Handlung, die kontinuierlich ausgeführt werden muss.

CompositeBehaviour: Diese abstrakte Klasse dient als Basis für Handlungen, die sich aus einer Reihe von anderen Behaviours zusammengesetzten. Die eigentlichen Operationen die von diesem Behaviour ausgeführt werden sollen, werden nicht in dieser Klasse definiert, sondern in den enthaltenen Behaviour-Objekten. Das `CompositeBehaviour` kümmert sich nur um das Scheduling der Kinder, nach einem vorgegebenen Schema.

SerialBehaviour: Ein zusammengesetztes Behaviour, das seine Kinder in einer gewissen (eventuell dynamischen) Reihenfolge ausführt.

SequentialBehaviour: Ein zusammengesetztes Behaviour, das seine Kinder in einer festen vorgegebenen Reihenfolge ausführt. Sie kann verwendet werden, wenn eine komplexe Handlung durch eine Reihe von atomaren Schritten dargestellt werden kann.

ParallelBehaviour: Ein zusammengesetztes Behaviour, das seine Kinder parallel (nebenläufig) ausführt und erst terminiert, wenn eine bestimmte Bedingung für eines der Kinder erfüllt worden ist. Sie kann verwendet werden, wenn eine komplexe Handlung als eine Ansammlung von alternativen Operationen dargestellt werden kann.

FSMBehaviour: Diese Klasse ist ein zusammengesetztes Behaviour, dessen Kinder einen Zustandsautomaten bilden. Die Kinder bilden die Zustände des Automaten und die Zustandsübergänge (*transitions*) werden vom Programmierer definiert. Da jedes Kind auch ein Behaviour ist, ist es möglich hierarchische Automaten zu definieren (Zustände enthalten Zustände). Die Klasse `FSMBehaviour` hat die Aufgabe den nächsten Ausführungszustand zu ermitteln. Bei der Registrierung eines Zustands wird ein Name mit dem zugehörigen Behaviour in Verbindung gebracht. Wenn alle Zustände definiert worden sind, werden die Übergänge zwischen ihnen definiert. Jeder Übergang bekommt einen Wert zugewiesen, der dessen Ereignisnummer darstellt. Nachdem Start- und End-Zustände markiert worden sind, ist der Automat fertig zur Ausführung. Wenn ein Zustand abgearbeitet ist liefert die `onEnd` Methode die Ereignisnummer zurück, die den nächsten auszuführenden Zustand bestimmt.

Aufgrund des verwendeten *Multitasking* Modells in Form des Schedulers muss unbedingt vermieden werden, dass in einem Behaviour eines Agenten eine Endlosschleife entsteht. Außerdem ist es nötig darauf zu achten, dass Operationen, die innerhalb einer `action` Methode ablaufen, nicht zu aufwändig sind. Der Grund dafür ist, dass sich die Verarbeitung einer `action` Methode nicht unterbrechen lässt; ist die Ausführung der Methode zu aufwendig, so kommt das Scheduling aus dem Gleichgewicht. Allerdings nur bezogen auf die Behaviours eines Agenten, andere Agenten laufen in anderen Prozessen und können daher unabhängig fortfahren.

Nach *Fonseca et al.* ist die Programmierung des Verhaltens von Agenten mit Hilfe einer Hierarchie von Hilfsklassen im Grunde sinnvoll. Die von JADE angebotenen Klassen führen jedoch zu keiner sehr guten Wiederverwendbarkeit von Code, da sie zu primitiv sind [17]. Der endliche Automat der Klasse `FSMBehaviour` ist ein Schritt in die richtige Richtung, die Umsetzung durch JADE ist allerdings noch nicht mächtig genug.

Es ist nicht klar, welche Syntax und Semantik den Zustandsautomaten zu Grunde liegt, die sich mittels der Klasse `FSMBehaviour` definieren lassen. Vermutlich wurden einige Teile der UML Zustandsdiagramme verwendet.

Das Framework berücksichtigt die Mobilität von Agenten nicht explizit, ein Symbol für die Migration lässt sich also nicht im Modell des Agentenverhaltens finden.

Die Implementierung des Behaviour-Frameworks ist sehr eng mit dem JADE-System verwoben. Eine Anwendung des Frameworks innerhalb der SeMoA-Plattform ist daher nach Meinung des Autors nicht ohne größeren Aufwand zu realisieren.

5.2.2 Das Projekt HSMBehaviour

Das *HSMBehaviour* ist eine Erweiterung des Behaviour-Frameworks von JADE, die es ermöglicht, auf UML basierende, hierarchische Zustandsautomaten aus JADE Behaviours zu konstruieren und im Rahmen von JADE auszuführen. HSMBehaviour wird in einer Zusammenarbeit der *University of California* in Santa Cruz und der *University of Utah* als Teil des Projektes *SCATE* (*Santa Cruz Agent Technology & Environments Research*) entwickelt. Das HSMBehaviour befindet sich zur Zeit noch in der Entwicklung.[6] Soweit das möglich ist, soll aufgrund des hohen Verwandtheitsgrades zur vorliegen Arbeit dennoch eine Darstellung und Analyse gegeben werden.

HSMBehaviour erweitert die Klasse `FSMBehaviour` von JADE in vielerlei Hinsicht. Es wird zum einen eine genauere Kontrolle über die Reihenfolge der Ausführung von Behaviours ermöglicht. Zum anderen wird die flexible Dekomponierbarkeit von Behaviours verbessert. Das Framework stellt in der vorliegenden Form laut den Autoren *Griss et al.* einen Kompromiss zwischen dem Verhaltensmodell von JADE und dem *CoolAgent* dar [23, 24].

Abbildung 5.4 stellt das Framework als Klassendiagramm dar. Es wurden bezüglich der Schnittstellen starke Vereinfachungen vorgenommen; viele der öffentlichen Methoden, sowie sämtliche Konstruktoren werden nicht dargestellt.

HSMBehaviour: Die Klasse `HSMBehaviour` ist ein zusammengesetztes JADE-Behaviour (`CompositeBehaviour`), das als Zustandsautomat betrachtet werden kann. Dieser verwaltet eine Reihe von Teilzuständen, welche vom Typ `Behaviour` sind. Zustände

[6] Der Quellcode ist frei verfügbar unter http://www.cse.ucsc.edu/research/agents/hsm.

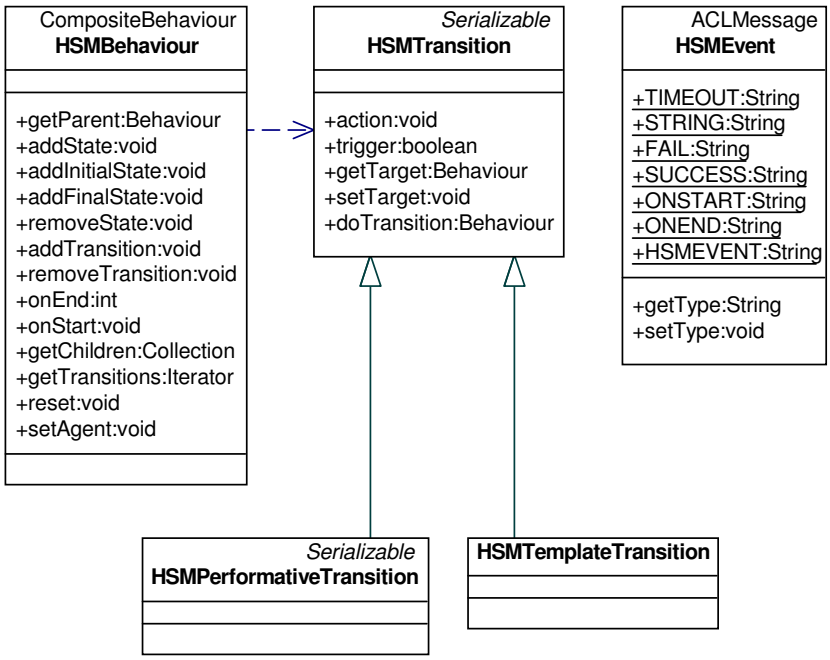

Abbildung 5.4: Ein vereinfachtes Klassendiagramm für das *HSMBehaviour Framework*.

lassen sich durch die Methode `addState` hinzufügen. Genau ein Zustand wird als Startzustand registriert (`addInitialState`); eine beliebige Anzahl von Zuständen lässt sich als Endzustand registrieren (`addFinalState`). Zustände lassen sich jeweils mehrfach registrieren, das heißt ein einziger Zustand kann Start- und Endzustand zugleich sein.

Sobald ein `HSMBehaviour` von JADE ausgeführt wird (durch die Methode `action`), ruft dieses die Methode `action` auf dem aktuellen Zustand auf. Diese Methode des Zustands wird so lange aufgerufen, bis die Methode `done` signalisiert, dass die Ausführung des Behaviours abgeschlossen ist. Ein neuer Zustand wird erst aktiviert, wenn dies durch das explizite Auslösen einer Transition signalisiert wird.

HSMEvent: Das `HSMEvent` wurde als Sub-Klasse von `ACLMessage` definiert, um die Nachrichten des Systems und die internen Nachrichten des Agenten zu vereinheitlichen. Daher können Ereignisse vom Typ `HSMEvent` durch den Agenten in üblicher weise verwaltet werden (`receive`, `post` und `putback`). Ereignisse werden JADE-typisch durch *MessageTemplates* miteinander verglichen.

HSMTransition: Diese Klasse definiert eine Transition, welche über die Attribute *Aktion*, *Trigger (auslösendes Ereignis)*, *Quell-* und *Zielzustand* verfügt. Eine Transition vom Typ `HSMTransition` wird über die Methode `addTransition` zum Zustandsautomaten hinzugefügt.

Zum Entwickeln und Testen von Zustandsautomaten steht eine graphische Entwicklungsumgebung, der *HSMEditor*, zur Verfügung. Dieser Editor gestattet eine graphische Konstruktion und die Simulation des Kontrollflusses eines hierarchischen Zustandsautomaten, der durch ein HSMBehaviour spezifiziert wird. Diese Zustandsautomaten lassen sich persistent im XML-Format speichern (inklusive Ausführungszustand). Die XML-Darstellung kann wiederum als Parameter für ein geeignetes Behaviour (`HsmPersistentBehaviour`) verwendet werden.

Die Zustände und Transitionen, inklusive der hierarchischen Beziehungen der Zustände, lassen sich mittels des Editors erzeugen und zu einer vollständigen visuellen Repräsentation des Automaten verbinden. Jeder Zustand muss mit dem Code assoziiert werden, der das Verhalten bestimmt; zu diesem Zweck kann jeder Zustand ein Behaviour referenzieren, welches dann bei der Ausführung des Automaten geladen und an die korrekte Stelle eingefügt wird. Die Referenzierung geschieht durch den Klassen-Namen des Behaviours.

Zur Simulation steht eine spezielle Klasse bereit (`HsmSimulationBehaviour`), die XML-Definitionen oder manuell programmierte Zustandsautomaten entgegen nimmt und die Ausführung des Automaten in einem Fenster graphisch darstellt. Die Funktionsweise des Simulators ist zu vergleichen mit der eines Debuggers: Die Ausführung lässt sich jederzeit anhalten und auch in einzelnen Schritten abarbeiten.

Die Verwendung von UML-basierten, hierarchischen Zutandsautomaten für die Modellierung des Verhaltens von Agenten ist ein sehr guter Ansatz (siehe Kapitel 8), der das ursprüngliche Framework von JADE stark erweitert. Die Verfügbarkeit eines graphischen Werkzeuges für die Erstellung und das Testen von Automaten rundet den Ansatz ab.

Das Ziel für die Syntax und Semantik der Automaten ist klar definiert (UML), dies wurde zum aktuellen Zeitpunkt jedoch noch nicht vollständig realisiert. Zudem ist die Abbildung von UML auf die Implementierung nicht sehr intuitiv: Es existiert nur für die Elemente Zustand, Transition und Ereignis eine Repräsentation durch Java-Klassen. Durch die Verwendung des Editors wird dies jedoch kompensiert.

In HSMBehaviour findet keine konsequente Trennung von Modell und Kontrollfluss statt. Zwar ist ein Zustandsautomat in der XML-Form ein reines Modell, wird dieser jedoch in eine Laufzeit-Repräsentation überführt, so wird ein Teil des Programmablaufs durch Automaten selbst gesteuert. Die Klasse `HSMBehaviour` übernimmt die Steuerung des Kontrollflusses, ist jedoch gleichzeitig ein Teil des Modells.

Für die Anwendung in SeMoA ist HSMBehaviour nicht geeignet, das es zu sehr auf JADE bezogen ist. Dies ist beispielsweise auf die Typisierung der Teilzustände als `Behaviour` oder die Erweiterung der `ACLMessage` durch das `HSMEvent` zurückzuführen.

5.3 XABSL

XABSL steht für *Extensible Agent Behavior Specification Language* [38]. Es handelt sich dabei um eine Sprache zur Beschreibung des Verhaltens von Agenten, die im *Roboter-Fußball* vom *GermanTeam* [58] in der *Sony Four Legged League*[7] eingesetzt wird. Die Agenten sind in diesem Fall *Aibo* Roboter; sie sind die Spieler im Roboter-Fußball. XABSL wurde von *Martin Loetzsch* an der Berliner *Humbold Universität* entwickelt und ist als Open Source verfügbar [8].

XABSL wurde entwickelt um insbesondere die Spezifikation von sehr komplexen Verhalten zu vereinfachen. Außerdem soll der Entwurf von stark reaktivem Verhalten sowie von Verhalten die über einen langen Zeitraum ablaufen, besonders unterstützt werden.

Die Architektur von XABSL basiert auf hierarchischen Zustandsautomaten. Mit XABSL werden im Prinzip Hierarchien von Verhaltens-Modulen beschrieben. Diese Module werden als *Optionen* bezeichnet. Grundsätzlich ist das Verhalten eines Agenten aus einer Reihe von Optionen zusammengesetzt. Die verwendeten Optionen bilden einen *Optionen-Graph*. Dieser ist azyklisch, gerichtet und besitzt eine Wurzel. Die Blätter dieses Graphen werden als *Basisverhalten* bezeichnet; sie sorgen für die Ausführung der Handlungen des Agenten, können also auch als seine Grundfähigkeiten betrachtet werden.

Das Gesamtverhalten eines Agenten wird durch die Wurzel repräsentiert. Die Aufgabe der inneren Konten des Graphen, also der Optionen, ist es zu entscheiden welches der Basisverhalten als nächstes ausgeführt werden soll. Abbildung 5.5 stellt den vereinfachten Optionen-Graph eines Roboter-Torwarts dar [38]. Die Darstellung bedient sich der üblichen Bezeichnungen des Fußballspiels und sollte daher selbsterklärend sein.

Der Entscheidungsfindungsprozess der Optionen ist durch Zustandsautomaten definiert. Wie üblich besteht der Automat aus Zuständen und Transitionen sowie einem ausgezeichneten Startzustand. Es gibt jedoch keine speziellen Endzustände. Statt dessen ist es möglich, von jedem Zustand aus eine Transition zu definieren, die den Zustandsautomaten verlässt und in einer Option oder einem Basisverhalten endet. Abbildung 5.6 stellt den Zustandsautomaten der Option *spielen* dar.

Die Entscheidung eines Zustands, über das Auslösen der nächsten Transition, wird durch einen *Entscheidungs-Baum* modelliert. Die Blätter des Baums repräsentieren die Transitionen zu anderen Zuständen des Automaten. Die inneren Knoten des Baums sind Konditionale Ausdrücke, die aus Variablen oder Funktionen der Umgebung des Agenten, Sensordaten oder Nachrichten von anderen Agenten zusammengesetzt sind. Anstelle von konkreten Daten werden Symbole oder Variablen für jede dieser Funktionen verwendet. Abbildung 5.7 zeigt den Entscheidungs-Baum des Zustands *gehe-zum-Ball*.

[7] http://www.openr.org/robocup
[8] http://www.ki.informatik.hu-berlin.de/XABSL

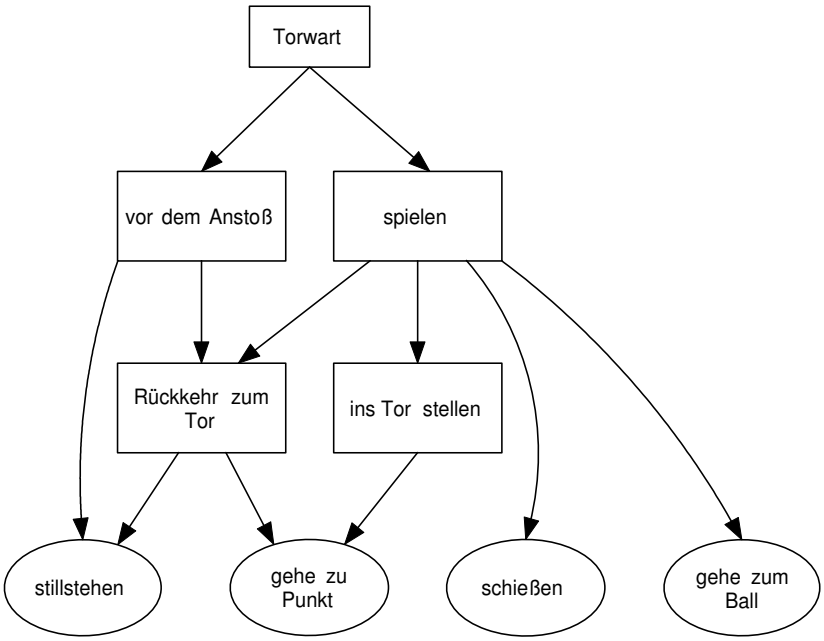

Abbildung 5.5: Der Optionen-Graph eines vereinfachten Torwarts.

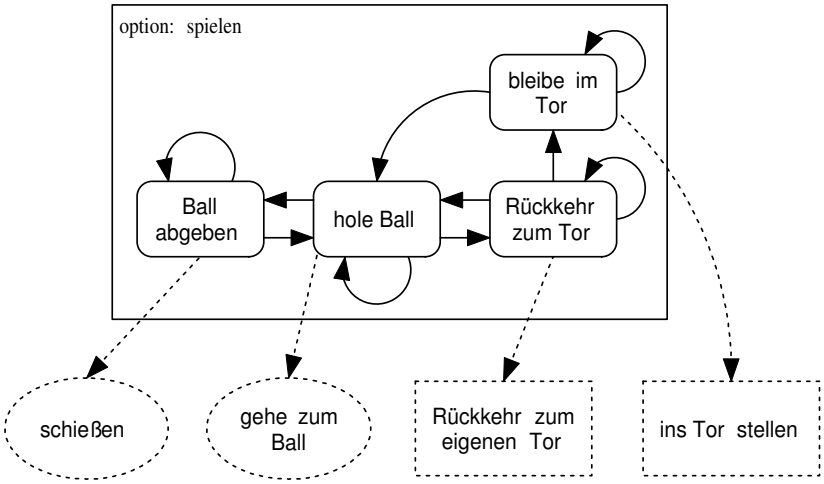

Abbildung 5.6: Der Zustandsautomat der Option *spielen*.

XABSL ist ein *XML 1.0* Dialekt [8], der in *XML Schema* [16] spezifiziert ist. Alle Bausteine des Verhaltens eines Agenten, außer den Basisverhalten, werden in diesem XML-Dialekt beschrieben. Für die Implementierung des Basisverhaltens wird eine Sprache benutzt, die direkt mit dem Betriebsystem des Roboters zusammenarbeiten kann; in diesem Fall wird *C++* verwendet. Auch die Symbole, die vom Verhalten bei der Interaktion mit den Komponenten des Roboters und seiner Umgebung eingesetzt werden, müssen durch diese Sprache definiert werden.

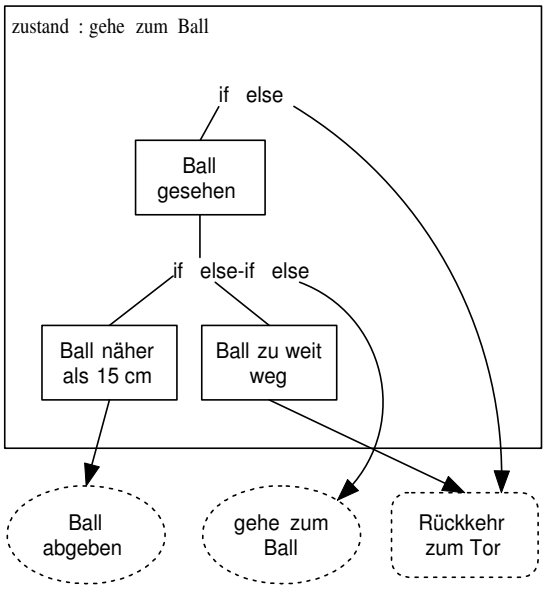

Abbildung 5.7: Der Entscheidungs-Baum des Zustands *gehe-zum-Ball*.

Die XML-Definitionen werden zunächst in einen *Übergangs-Code* übersetzt, der dann von der *XabslEngine* ausgeführt werden kann. Dieser Schritt ist nötig, um von verwendeten XML-Bibliotheken unabhängig zu werden, da diese nicht zwangsläufig auf jeder Roboter-Plattform zur Verfügung stehen.

Ein Zusatznutzen der Verwendung von XML ist, dass die XML-Daten auch leicht in andere Darstellungen überführt werden können. Dadurch lassen sich beispielsweise automatisch Beschreibungen und Diagramme für die implementierten Graphen und Zustandsautomaten generieren.

Abbildung 5.8 stellt die Verarbeitungsschritte der XML-Definitionen durch XABSL dar. Mittels XSLT-Transformationen, lässt sich aus dem XML-Dokument sowohl der Übergangs-Code generieren, als auch die Dokumentation des Roboter-Verhaltens.

Die Unterteilung in drei Schichten und die Verwendung von hierarchischen Zustandsautomaten ermöglichen die Modellierung von sehr komplexen Roboter-Verhalten. Die Entscheidungsfindung lässt sich kurzfristig und reaktiv, aber auch sehr langfristig und zielgerichtet konstruieren.[9]

XABSL verwendet eine eigene Form von hierarchischen Zustandsautomaten. Die Syntax und Semantik der verwendeten Automaten ist durch die Spezifikation in XML Schema klar definiert.

Die Gewährleistung der Modularität der einzelnen Verhaltenselemente ist Sache des Entwicklers und wird nicht direkt von XABSL unterstützt. Problematisch sind nach Meinung des Autors

[9] Die Leistungsfähigkeit von XABSL wurde zuletzt bei den internationalen Weltmeisterschaften im Roboter-Fußball 2004 in Portugal unter Beweis gestellt. Das *GermanTeam* belegte den 1. Platz und ist somit der amtierende Weltmeister.

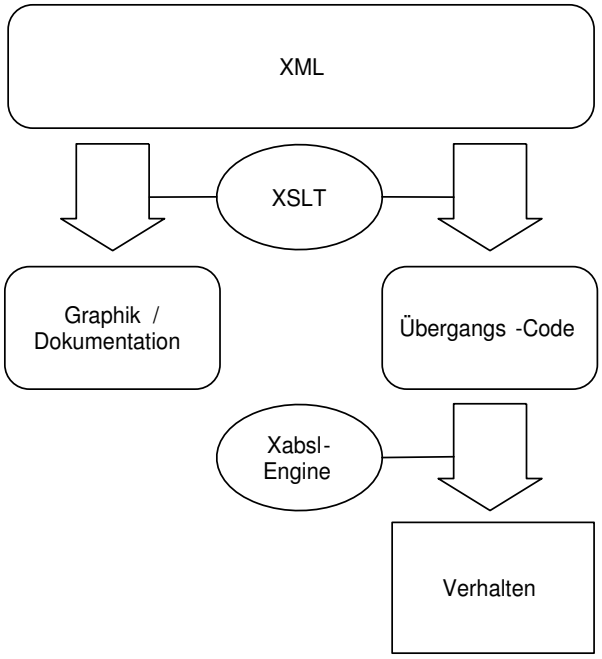

Abbildung 5.8: Ein Überblick der XML-Verarbeitung in XABSL.

vor allem die Transitionen, welche frei zwischen allen Elementen und Schichten eines Modells definiert werden dürfen.

Durch die Verwendung von XML als Beschreibungssprache für das Verhalten, ist ein hohes Maß an Flexibilität gegeben. Außerdem wird eine saubere Trennung zwischen der Beschreibung des Verhaltensmodells und der Ausführung des Modells erreicht. Die Verwendung von Übergangs-Code sorgt für eine sehr gute Portabilität der erzeugten Komponenten.

Da die atomaren Verhalten und die *XabslEngine* in C++ implementiert sind, ist eine Anwendung von XABSL in SeMoA nicht direkt möglich. Um dies zu erreichen müsste die *XabslEngine* in Java portiert und dahingehend angepasst werden, dass die atomaren Verhalten auch in Java implementiert werden können.

Teil II

Realisierung

Kapitel 6

Anforderungen

If you reuse code, you'll save a load, but if you reuse design, your future will shine.
— Ralph Johnson

In diesem Kapitel werden die Anforderungen für das zu entwickelnde Konzept diskutiert. Es wird unterschieden zwischen Anforderungen, die nach der Meinung des Autors im Allgemeinen für eine Lösung zur Modellierung und Implementierung von mobilen Agenten erfüllt werden müssen und solchen Anforderungen, welche die Umsetzung des Konzepts für die SeMoA-Plattform betreffen.

Im Folgenden wird dargestellt, wie die Anforderungen lauten und weshalb diese im einzelnen gestellt werden. Eine genaue Analyse der Anforderungen, unter Berücksichtigung der verwandten Arbeiten, wird in Kapitel 8 durchgeführt. Dabei werden die Grundlagen des Lösungsansatzes entwickelt, der beschreibt wie die Anforderungen zu erfüllen sind. Wie die Anforderungen letztlich durch diesen Ansatz erfüllt werden, wird in Kapitel 7 dargestellt und in Kaptitel 9 im Detail entwickelt.

6.1 Konzeptionelle Anforderungen

Das Konzept soll die Entwicklung von Agenten ermöglichen, welche die Charakteristiken *Autonomie*, *soziale Fähigkeit*, *Reaktivität* oder *Proaktivität* aufweisen können. Die *Mobilität* spielt zudem eine besondere Rolle, da sie für mobile Agenten auf jeden Fall gegeben sein muss.

In dem Konzept sollen zwei aufeinander folgenden Phasen separat unterstützt werden und zur Anwendung kommen: *Modellierung* und *Implementierung*. Die Idee ist, dass das Konzept eine Sprache zur Beschreibung der Architektur eines Agenten enthält, mit der diese Architektur unabhängig von der Implementierung modelliert werden kann. Dies lässt sich zurückführen auf

die *Architecture Description Languages* (*ADLs*), die seit einigen Jahren bei dem Entwurf von Software-Systemen angewendet werden [42].

In der Modellierungsphase soll der Agent zunächst auf einer abstrakten Ebene entworfen werden. Der Entwurf erfolgt in einer sprachlichen Form, die für den Menschen leicht verständlich ist und daher auch für Kommunikation zwischen Entwicklern gewinnbringend eingesetzt werden kann. Aus Gründen der Verständlichkeit ist eine im hohen Maße graphisch-orientierte Repräsentation der Sprache wünschenswert. Es sind außerdem Richtlinen für die ziel-orientierte Verwendung der Sprache anzugeben. Sprache und Richtlinien lassen sich als *Entwurfsmethode* (*Methodologie*) zusammenfassen. Diese Methode soll speziell auf das Anwendungsgebiet der mobilen Agenten zugeschnitten sein.

In der Implementierungsphase wird, mit Hilfe eines zu definierenden *Prozesses*, aus dem Entwurf des Agenten eine für Computer lesbare und ausführbare Darstellung (Software) generiert. Eine Voraussetzung dafür ist die Verwendung einer *Komponenten-Technologie*, welche die Abbildung des modularisierten Agenten in wiederverwendbare Software-Komponenten ermöglicht.

Durch die strikte Trennung und explizite, separate Unterstützung beider Phasen soll eine gute Wiederverwendbarkeit erreicht werden; sowohl bezüglich des Entwurfs, als auch bezüglich der Implementierung.

Insgesamt sind also drei Bestandteile zu finden, aus denen sich das Konzept zusammensetzt:

- Eine Entwurfsmethode für mobile Agenten.
- Ein Prozess, mit dem sich aus dem Entwurf eine ausführbare Form erzeugen lässt.
- Eine Komponenten-Technologie, die den Prozess unterstützt.

Abbildung 6.1 stellt dar, wie eine Entwurfsidee durch die Verwendung einer Methodologie, eines Prozesses und einer Komponenten-Technologie in eine konkrete Implementierung überführt wird.

Oberste Priorität hat die *Wiederverwendbarkeit* und *Erweiterbarkeit* der Resultate, welche die Anwendung des Konzepts liefert. Durch Wiederverwendbarkeit wird gewährleistet, dass sich der Entwurf und die daraus resultierende Software, in Teilen oder als Ganzes, in anderen Kontexten wiederverwenden lassen. Erweiterbarkeit meint, dass der Entwurf und die daraus resultierende Software sich leicht an veränderte Anforderungen, also Änderungen der Spezifikation, anpassen lässt. Ist eine gute Wiederverwendbarkeit und Erweiterbarkeit gewährleistet, so wird der Aufwand verringert, der für die Entwicklung und Wartung von mobilen Agenten betrieben werden muss.

Die Erfüllung beider Kriterien lässt sich durch *Modularität* erreichen. Es muss möglich sein, durch die Anwendung des Konzepts die Resultate so zu gestalten, dass sie sich in einzelne,

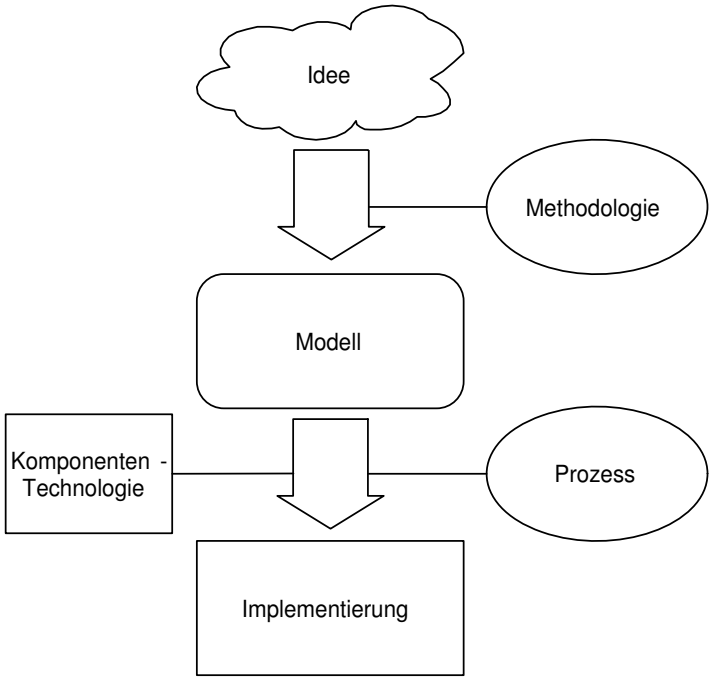

Abbildung 6.1: Von der Idee zur Implementierung.

unabhängige Elemente aufteilen lassen. Diese Elemente kapseln gewisse Geheimnisse, die nur für sie selbst verfügbar sind, und dürfen nur über klar definierte Schnittstellen miteinander interagieren; sie werden als *Module* bezeichnet. Module können hierarchisch zusammengesetzt werden, es gibt also Module, die aus anderen Teilmodulen bestehen. Modularität soll sowohl für den Entwurf, als auch für die resultierende Implementierung gelten.

Es bleibt zu klären, wie sich Modularität erreichen lässt. *Meyer* [43] gibt fünf Kriterien an, die bei der Anwendung einer Entwurfsmethode erfüllt werden müssen, um die Modularität der resultierenden Elemente zu gewährleisten:

Dekomponierbarkeit: Eine Entwurfsmethode erfüllt modulare Dekomponierbarkeit, falls sie die Zerlegung eines Elementes in eine kleine Anzahl von Teilelementen unterstützt, die eine geringere Komplexität aufweisen. Diese Teilelemente müssen soweit von einander unabhängig sein, dass es möglich ist, sie einzeln zu bearbeiten und zu verändern ohne dabei andere Teilelemente zu berücksichtigen.

Komponierbarkeit: Eine Entwurfsmethode erfüllt modulare Komponierbarkeit, falls sie die Erstellung von Elementen ermöglicht, welche frei miteinander kombiniert werden können und diese Kombinationen ihrerseits neue Elemente bilden.[1] Idealerweise lassen sich diese

[1] Man beachte, dass Komponierbarkeit nicht zwangsläufig aus der Dekomponierbarkeit folgt. Die Resultate einer Dekomposition müssen nicht notwendiger Weise miteinander kombiniert werden können.

zusammengesetzten Elemente sogar in einem anderen Kontext anwenden, als der für den die Basiselemente ursprünglich entwickelt wurden. Dieses Kriterium ist sehr stark an das der Wiederverwendbarkeit gebunden.

Verständlichkeit: Eine Entwurfsmethode erfüllt modulare Verständlichkeit, falls sie dabei hilft Elemente zu erstellen, die unabhängig von anderen Elementen verstanden werden können, mit denen das betrachtet Element in einem Zusammenhang steht. Schlimmstenfalls sollen nur einige wenige andere Elemente untersucht werden.

Kontinuität: Eine Entwurfsmethode erfüllt modulare Kontinuität, falls eine kleine Änderung der Spezifikation auch nur die Änderung einer angemessen kleinen Anzahl von Elementen, welche die Spezifikation implementieren, nach sich zieht.

Schutz: Eine Entwurfsmethode erfüllt modularen Schutz, falls sich das Auftreten eines abnormen Ereignisses oder Zustandes und dessen Behandlung nur innerhalb eines Elementes abspielt. Andere Elemente sollen möglichst nicht davon betroffen sein; im schlimmsten Fall nur einige wenige.

Diese Kriterien müssen in erster Linie von der zu entwerfenden Methodologie berücksichtigt werden. Ist dies der Fall, so lässt sich die dadurch gewonnene Modularität in die Implementierung übertragen. Es lassen sich dann sowohl Teile des Entwurfs eines Agenten als auch Teile der Implementierung unabhängig verändern und wiederverwenden.

Von besonderer Bedeutung ist die modulare Dekomponierbarkeit. Durch die Erfüllung dieses Kriteriums fällt es leicht die Belange[2], die berücksichtigt werden müssen um das Anwendungsziel zu erreichen, auf die einzelnen Teilmodule zu verteilen. Die *Separierung der Belange* (*separation of concerns*) in Softwaresystemen ist der Schlüssel, um mit wachsender Komplexität umzugehen [13, 53]. Je mehr sich ein Modul auf einen einzelnen oder nur auf wenige Belange konzentriert, um so wahrscheinlicher ist es, dass dieses Modul in unterschiedlichen Kontexten wiederverwendet werden kann. Ein Modul ist umso wiederverwendbarer, je weniger Abhängigkeiten an den Benutzungskontext bestehen; dies ist genau dann der Fall, wenn sich dieser Teil der Software nur auf einen einzelnen Belang konzentriert. Entwurfsmethoden, die modulare Dekomponierbarkeit erfüllen, sind in häufig *top-down* Verfahren (siehe Abschnitt 4.2).

Die eingesetzte Entwurfsmethode, muss besonders an die spezifischen Charakteristiken von mobilen Agenten angepasst werden (siehe Abschnitt 2.1). Durch diese spezielle Anpassung soll ein ein minimaler Overhead beim Entwurf gewährleistet werden. Die Methode soll so gestaltet sein, dass es möglich ist, diese sowohl als Bindeglied zwischen anderen Entwurfsmethoden und der Implementierung zu verwenden, als auch die Methode unabhängig für den Entwurf einzusetzen. Weiterhin muss auf eine leichte Anwendbarkeit der Methode geachtet werden. Letztlich

[2] Ein Belang (*concern*) ist jede kohärente Angelegenheit in einer bestimmten Problemdomäne. Die Komplexität eines Belanges ist beliebig.

soll dem Entwickler eines Agenten dadurch gestattet werden seine mentalen Modelle, die das Entwurfsziel beschreiben, möglichst direkt in Software-Komponenten zu übersetzten.

Die Komponenten-Technologie, die zum Einsatz kommt um die entworfenen Modelle zu implementieren, soll so gestaltet sein, dass es möglich ist, diese auch losgelöst vom Rest des Konzeptes einzusetzen. Die dadurch gewonnenen Flexibilität trägt dazu bei, die Wiederverwendbarkeit und Erweiterbarkeit des Konzeptes selbst (und nicht nur der Resultate) zu gewährleisten. Ist die Komponenten-Technologie unabhängig von der Entwurfsmethode und dem Prozess, so kann sie theoretisch auch mit anderen Entwurfsmethoden eingesetzt werden. Ein Entwickler ist dann beispielsweise in der Lage, Agenten mit Hilfe der Komponenten-Technologie und unter Verwendung von traditionellen objekt-orientierten Entwurfsmethoden zu erstellen. Die Technologie soll leicht erlernbar und einfach zu benutzen sein. Daher ist es wichtig, dass ein griffiger und aussagekräftiger Komponentenbegriff verwendet wird, der die Eigenschaften und die Anwendbarkeit einer Komponente repräsentiert.

Ein weiteres wichtiges Kriterium, welches durch das zu entwickelnde Konzept berücksichtigt werden muss, ist die *Interoperabilität*. Die entwickelten Agenten sollen zwischen Agenten-Plattformen verschiedener Hersteller migrieren und auf diesen Plattformen ausgeführt werden können, ohne dass der Agent neu kompiliert werden muss. Ein Agent soll weiterhin in der Lage sein, mit anderen Agenten verschiedener Hersteller zu interagieren und zu kommunizieren. Interoperabilität zielt vor allem auf die Verbreitung und die industrielle Akzeptanz von Systemen für mobile Agenten ab. Selbst wenn die Entwicklungsmethode Interoperabilität nicht direkt ermöglichen oder verbessern sollte, so muss zumindest gewährleistet sein, dass es auf anderer Ebene weiterhin möglich ist, Interoperabilität bei gleichzeitiger Verwendung des entwickelten Konzepts umzusetzen.

6.2 Anforderungen an die Implementierung

Da SeMoA als Plattform für die prototypische Umsetzung des Konzepts gewählt wurde, ergeben sich einige Anforderungen, die in diesem Kontext erfüllt werden müssen.

Alle Bestandteile von SeMoA sind ausschließlich in der Programmiersprache Java implementiert. Es gehört zu den Richtlinien der Plattform, dass auch alle zukünftigen Erweiterungen in Java geschrieben werden müssen. Der Grund für diese Einschränkung ist, dass SeMoA momentan mit Hilfe des *JDK* (*Java Development Kit*) übersetzt, und in eine ausführbare Form gebracht werden kann; es werden keine anderen Werkzeuge benötigt. Alles was man braucht um eine lauffähige SeMoA-Plattform zu erstellen, ist der frei verfügbare Quellcode, das JDK, sowie einige Java Pakete, die nicht im JDK enthalten sind. Dadurch wird der Entwicklungsprozess sehr einfach und übersichtlich gehalten. Außerdem ist das gesamte Projekt in allen Teilen jederzeit änderbar. Alles was man beherrschen muss, ist die Software-Entwicklung mit Java. Das soll auch zukünftig so bleiben.

Da es verschiedene Compiler und auch verschiedene *Virtual Machines* für die Java Sprache gibt, muss noch erwähnt werden, dass nur Bestandteile des JDK von *Sun Microsystems* in Frage kommen, oder solche Varianten, die zu einhundert Prozent kompatibel zum JDK sind.[3] Zum Übersetzen des Java-Codes wird also das Programm `javac` eingesetzt und zum Ausführen des *Byte Codes*, das Programm `java`. Aus Gründen der Vermarktung von SeMoA, soll keine Erweiterung derzeit eine höher Versionsnummer des JDK als 1.4 benötigen. Gegebenenfalls lassen sich jedoch benötigte Bibliotheken als separate Pakete einbinden.

Übertragen auf die Implementierung des Konzepts heißt das, dass die ausführbaren Programmteile alle in Java geschrieben sein müssen. Davon betroffen sind jedoch nur die ausführbaren Teile. Das heißt, dass Konfigurationsdateien und andere Daten, die vom Programm benötigt werden, in jeder Form vorliegen können, solange eine Bearbeitung mit Hilfe von Java möglich ist[4].

Die Konsequenzen dieser Einschränkungen sind, dass keine Erweiterungen von Java, wie beispielsweise *AspectJ*[5], *HyperJ*[6] oder *MultiJava*[7], für die Implementierung eingesetzt werden können. Es ist auch nicht möglich über das *Java Native Interface* (*JNI*) auf plattformabhängigen Code zuzugreifen. Ausserdem können neue Sprachkonstrukte und Pakete, die seit der Version 1.4 in Java integriert wurden, nicht genutzt werden. Davon betroffen sind etwa *Generics* oder das `java.util.concurrent` Paket.

Ein weiter wichtiger Punkt ist die gute Dokumentation und Lesbarkeit des Java-Codes, der für die Umsetzung des Konzeptes entsteht. Das SeMoA-Projekt definiert zu diesem Zweck einen eigenen *Codestyle* [59]. Diese Forderung beruht auf der Tatsache, dass Softwaresysteme circa achtzig Prozent ihres Lebenszyklus in der Wartung verbringen. Dabei werden die Wartungsarbeiten sehr oft nicht vom ursprünglichen Autor des Codes durchgeführt. Durch die gute Dokumentation und Lesbarkeit soll gewährleistet werden, dass ein fremder Entwickler sich mit möglichst wenig Aufwand in dem Code zurechtfindet, diesen verstehen kann, und in die Lage versetzt wird, Änderungen vorzunehmen.

[3] Dazu gehört beispielsweise *jikes*.
[4] Als Beispiel sei hier XML genannt.
[5] http://eclipse.org/aspectj
[6] http://www.alphaworks.ibm.com/tech/hyperj
[7] http://multijava.sourceforge.net

Kapitel 7

Lösungsansatz

You know you have achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away. — Antoine de Saint-Exupery

In diesem Kapitel wird das Konzept für den komponenten-orientierten Entwurf von mobilen Agenten vorgestellt. Es ist das Ziel, einen Überblick zu gewähren, der das Verständnis der nachfolgenden Kapitel erleichtert. Zunächst werden die globalen Zusammenhänge erläutert, anschließend wird das Konzept jeweils aus der Perspektive der drei Bestandteile *Entwurfsmethode*, *Prozess* und *Komponenten-Technologie* dargestellt.

Die Darstellung erfolgt weitestgehend ohne Begründung für die Wahl des Ansatzes und die einzelnen Entwurfsentscheidungen, da diese in Kapitel 8 vorgestellt werden. Eine detaillierte Darstellung des Entwurfs, auf dem das Konzept basiert, erfolgt in Kapitel 9.

7.1 Das Gesamtkonzept

Das in dieser Arbeit entwickelte Konzept besteht aus drei Teilen: Einer Entwurfsmethode für mobile Agenten, einem Prozess, mit dem sich aus dem Entwurf eine ausführbare Form erzeugen lässt und einer Komponenten-Technologie, die in dem Prozess zum Einsatz kommt (vgl. Abschnitt 6.1.).

Den Kern des Konzepts bilden *hierarchische Zustandsautomaten.*[1] Diese dienen als Sprache zur Beschreibung des Agentenverhaltens. Die verwendete Art der Automaten ist stark an den Zustandsautomaten orientiert, wie sie in den *Zustandsdiagrammen* der UML verwendet werden. Viele Teile der Syntax und Semantik der UML Zustandsautomaten wurden übernommen, es gibt jedoch auch einige Anpassungen, die speziell für das Anwendungsgebiet der mobilen

[1] Eine ausführliche Begründung für die Wahl der hierarchischen Zustandsautomaten findet sich in Kapitel 8.

Agenten vorgenommen wurden. Dazu gehört insbesondere ein neuer Typ von Transition, die *Migrations-Transition*, durch die sich die Mobilität eines Agenten gut abbilden lässt. Weiterhin wurden einige bewusste Einschränkungen der Ausdrucksmöglichkeiten gegenüber UML vorgenommen, um die Modularität der Resultate und die damit verbundene Wiederverwendbarkeit zu verbessern.

Der Einsatz des Konzepts gestaltet sich folgendermaßen: Ein Entwickler spezifiziert durch Anwendung der zustandsorientierten Modellierung das Verhalten des Agenten in Zustandsdiagrammen. Die Modellierung kann an zwei verschiedenen Abstraktionsebenen angesetzt werden. Im ersten Fall wird der gesamte Agent ausschließlich durch Zustandsdiagramme dargestellt, das heißt der Entwurf beginnt auf der höchsten Abstraktionsebene mit Zustandsautomanten und endet mit diesen auch in der Implementierung. Die zustandsorientierten Modellierung überführt also die Entwurfsidee *direkt* in die Implementierung. Abbildung 7.1 stellt die Entwicklung eines Agenten auf die direkte Art dar. Im Vergleich mit der Abbildung 6.1 erkennt man, dass als Entwurfsmethode die zustandsorientierte Modellierung und als Modell Zustandsdiagramme gewählt wurden.

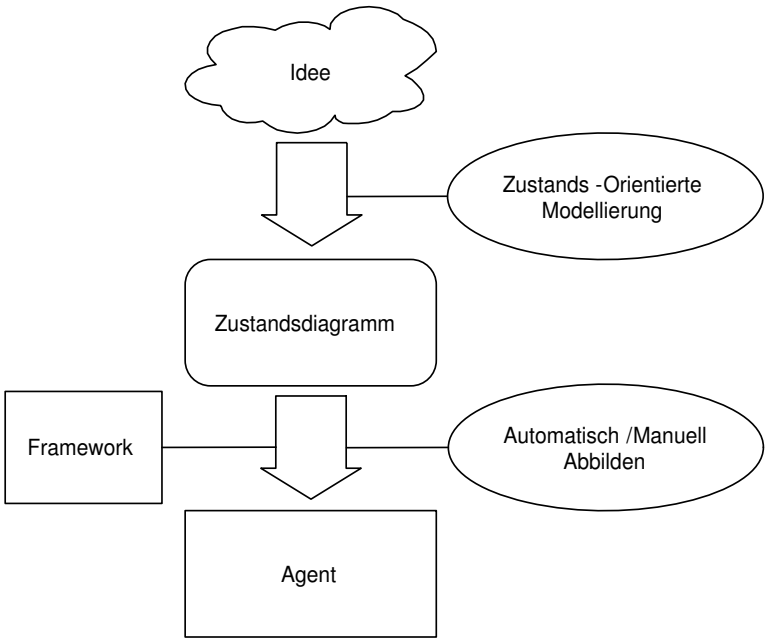

Abbildung 7.1: Der Lösungsansatz (direkt).

Im zweiten Fall wird der Entwurf des Agenten zunächst mit einer anderen geeigneten Entwurfsmethode begonnen, welche auf ihrer untersten Stufe kein implementierungsfähiges Niveau erreicht. Anschliessend wird das erstellte Zwischen-Modell durch Zustandsdiagramme weiter verfeinert bis eine Implementierung leicht möglich ist. Die zustandsorientierten Modellierung überführt also die Entwurfsidee nur *indirekt* in die Implementierung. Abbildung 7.2 stellt die Entwicklung eines Agenten auf die indirekte Art dar.

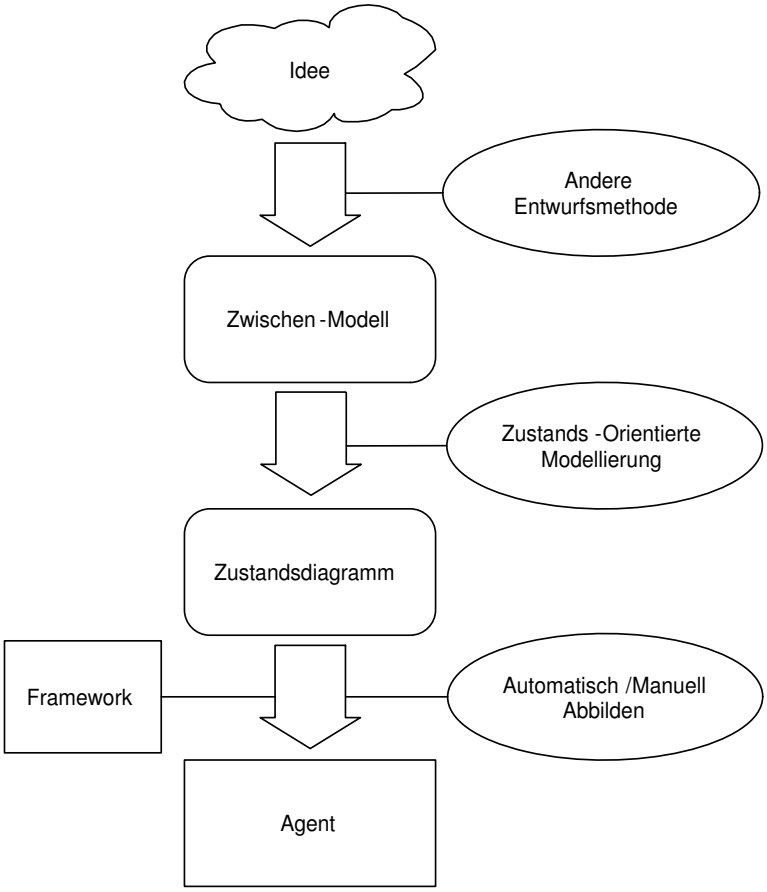

Abbildung 7.2: Der Lösungsansatz (indirekt).

Die verwendeten Zustandsdiagramme lassen sich auf mehreren Wegen erzeugen. Vorstellbar ist, dass die Diagramme entweder abstrakt, mit Hilfe eines UML-fähigen Programms spezifiziert, in einer Metasprache wie XML beschrieben oder direkt in Java-Code implementiert werden.[2]

Wenn die Automaten nicht in Java vorliegen, müssen die abstrakten Definitionen zunächst in Java abgebildet werden. Dies geschieht entweder automatisch, unter Verwendung eines Werkzeugs, oder manuell durch den Entwickler. Die Laufzeitelemente eines Automaten lassen sich sowohl statisch zur Kompilierzeit, in Form von geeigneten Klassendefinitionen, als auch dynamisch zur Laufzeit, durch Instanzierung konfigurierbarer Komponenten, erzeugen. Aufgrund der höheren Flexibilität ist der Einsatz von konfigurierbaren Komponenten vorzuziehen. In diesem Erzeugungsprozess kommt ein speziell entwickeltes *Framework* zum Einsatz, das es gestattet die Übersetzung der Zustandsdiagramme in Java-Code mit minimalem Aufwand durchzuführen.

Ist das Verhalten des Agenten in dieser Form erstellt, so kann das fertige Verhaltensmodul in einen konkreten, ausführbaren SeMoA-Agenten integriert werden. Dafür steht ein weiteres Fra-

[2] Der Implementierungsteil dieser Arbeit unterstützt nur die direkte Umsetzung in Java, dies jedoch in einer sehr praktikablen Weise. Siehe auch Kapitel 13.

mework bereit, das unter anderem eine spezielle *Basisklasse* für Agenten zur Verfügung stellt, die mit Zustandsautomaten parametrisiert werden kann.

7.2 Die Entwurfsmethode

Aufgabe der Entwurfsmethode ist es, aus einer Entwurfsidee, die sich als ein mentales Modell im Kopf eines Entwicklers befindet, ein implementierungsfähiges Modell zu erstellen. Die hier vorgestellte Methode ist ein *top-down* Verfahren, das schrittweise ein Modell auf Grundlage von Zustandsautomaten generiert. Das Vorgehen ist dabei sehr stark an die zustandsorientierte Modellierung angelehnt, wie sie in der UML vorgeschlagen wird. Es handelt sich also nicht um eine komplett neu definierte Methodologie, die ausschließlich auf eigens entwickelten Konzepten aufbaut, sondern um einen Ansatz, der sich an Bewährtem orientiert.

Das zentrale Konzept der Entwurfsmethode sind hierarchische Zustandsautomaten. Diese eignen sich sowohl für die Beschreibung von sehr einfachem als auch von sehr komplexem Verhalten (vgl. Abschnitt 5.3). Die Entwurfsmethode wurde so gestaltet, dass sie sich als eigenständiges Verfahren anwenden lässt. Sie ist jedoch auch so implementierungsnah anwendbar, dass sie sehr gut eine Brücke zwischen einer eher abstrakten Methodologie, wie beispielsweise *Gaia*, und einer konkreten Umsetzung der damit erstellten Modelle schlagen kann.

Zunächst werden die Grundlagen der hier verwendeten Zustandsautomaten dargestellt, anschliessend erfolgt ein Überblick über die zustandsorientierte Modellierung, wie sie in der Entwurfsmethode angewandt wird.

7.2.1 Hierarchische Zustandsautomaten

Die Grundlage der Entwurfsmethode sind Zustandsdiagramme beziehungsweise Zustandsautomaten. Diese dienen als Sprache für die Spezifikation des Verhaltens von mobilen Agenten. Die Syntax und Semantik der Zustandsdiagramme, wie sie hier eingesetzt werden, ist stark an der der UML orientiert. Wie in der UML setzten sich Zustandsdiagramme aus Zuständen und Transitionen zwischen den Zuständen zusammen. Diese bilden einen Graph, in dem die Zustände die Knoten darstellen und die Transitionen die Kanten, welche die Zustände mit einander verbinden. Ein hierarchischer Aufbau von Zuständen wird unterstützt, das heißt Zustände können Teilzustände enthalten. Zustände sind also ein rekursives Konzept. Insbesondere sind die Begriffe Zustand und Zustandsautomat austauschbar. Ein Zustandsautomat ist streng genommen ein zusammengesetzter Zustand, ein Zustand kann wiederum einen ganzen Zustandsautomaten repräsentieren. Daher kann ein vollständiger Zustandsautomat als *Black-Box* in Form eines einzelnen Zustandes in einen Zustandsautomaten eingefügt werden.

Der Zustandsautomat selbst definiert den Kontrollfluss, der das Verhalten des Agenten bestimmt. Dabei reagiert er auf gewisse Ereignisse und auf die Erfüllung gewisser Bedingungen, die im

Entwurf spezifiziert wurden. Ein Zustand wird genau dann geändert wenn eine der ausgehenden Transitionen des Zustands ausgelöst wird. Eine Transition wird ausgelöst falls ein auftretendes Ereignis als Auslöser (*trigger*) für diese Transition definiert ist und außerdem die Wachbedingung (*guard condition*) dieser Transition erfüllt ist. Ist keine Wachbedingung definiert, so wird die Transition auch ausgelöst. Es gibt allerdings auch Transitionen die kein auslösendes Ereignis benötigen sondern nur von der Erfüllung einer Bedingung abhängig sind. Die eigentlichen vom Agenten durchgeführten Handlungen, sind in Aktionen gekapselt, welche beim Erreichen eines Zustands oder dem Auslösen einer Transition ausgeführt werden.

Es gibt jedoch gewisse Einschränkungen gegenüber der UML, die im Rahmen der Entwurfsmethode beachtet werden müssen. Nicht gestattet ist der Einsatz von:

- Nebenläufigen Teilzuständen (*concurrent substates*)
- Transitionen, die über Zustandsgrenzen hinweg definiert sind (*boundary crossing transitions*)
- Aktivitäten (*activities* oder auch *do-activities*)

Die Begründungen für diese Einschränkungen werden in Kapitel 8 dargelegt.

Eine Erweiterung der UML ist ein neuer Typ von Transition, die *Migrations-Transition*, durch die sich die Mobilität eines Agenten gut abbilden lässt. Wird eine solche Transition ausgelöst, so ändert der Agenten seinen Ausführungsort bevor der neue Zustand abgearbeitet wird. Eine Migrations-Transition wird in Diagrammen durch einen Pfeil dargestellt, dessen Basis „gestrichelt“ ist. Wie bei gewöhnlichen Transitionen lassen sich auch hier auslösende Ereignisse und Wachbedinugen definieren, die das Auslösen der Transition steuern.

7.2.2 Zustandsorientierte Modellierung

Zentral bei der Anwendung der Entwurfsmethode ist die Dekomposition des Verhaltens in separate Teilverhalten und die Identifikation von Zuständen innerhalb dieser Verhaltensbausteine.

Zunächst muss geklärt werden, welche Rolle die Zustände eines Automaten selbst im Bezug zu dem Verhalten des Agenten spielen. Im Kontext mobiler Agenten gibt es nach Meinung des Autors drei mögliche Modellierungsperspektiven, die innerhalb eines Modells auch kombiniert werden können:

- Zustände repräsentieren einen Zeitraum im Leben des Agenten (ursprüngliche Definition)
- Zustände stellen die Rollen des Agenten dar, die dieser verkörpert
- Zustände repäsentieren die Orte, an denen sich der Agent zu einer gegebenen Zeit befindet

Jede dieser Herangehensweisen interpretiert das Konzept des Zustands ein wenig anders. Je nach Anwendungsfall ist die eine oder andere Art vorzuziehen, letzten Endes werden jedoch in allen drei Fällen Zustandsautomaten generiert, die aus den gleichen Element-Typen aufgebaut sind. Die unterschiedlichen Perspektiven haben je nach Art und Komplexität des Verhaltens des Agenten spezifische Vorzüge. So ist im Falle eines stationären Agenten, dessen Aufgabe es ist, bestimmte Nachrichten zu empfangen und zu beantworten, eine Modellierung durch Rollen und Zustände sicherlich angemessener, als ein am Standort des Agenten orientiertes Modell. Andererseits lässt sich ein mobiler Agent, dessen Aufgabe es ist, innerhalb eines statischen Netzwerkes gewisse Statusinformationen zu sammeln, durchaus gut in einem Modell beschreiben, das an den einzelnen Aufenthaltsorten des Agenten orientiert ist. In vielen Fällen dürfte die günstigste Lösung eine Kombination der drei vorgestellten Perspektiven sein.

Als erstes wird nun die reine Modellierung durch Zustände dargestellt, da diese den Kern der Entwurfsmethode bildet. Rolle und Ort werden letztendlich auch in Form von Zustandsautomaten beziehungsweise Zuständen repräsentiert, das Vorgehen ist daher ähnlich.

Bei der zustandsorientierten Modellierung ist die Berücksichtigung der reaktiven Eigenschaften des Agenten von besonderer Bedeutung. Die Ereignisse, auf die ein Agent reagieren kann, sowie die Reaktionen des Agenten auf diese Ereignisse müssen definiert werden. Außerdem sind die Auswirkung der vorangegangenen Ereignisse auf das aktuelle Verhalten des Agenten zu spezifizieren. Insgesamt wird dadurch die Reihenfolge der Ereignisse, auf die ein Agent reagieren kann, durch das Modell bestimmt.

Der Vorgang der Modellierung lässt sich in mehrere Schritte unterteilen. Anfangs wird das gesamte Verhalten des Agenten durch einen einzigen Zustand repräsentiert. In den folgenden Schritten wird dieser Zustand in Teilzustände untergliedert, so lange bis die gewünschte Granularität erreicht ist. Man kann dieses Vorgehen auch als eine Partitionierung des Verhaltens interpretieren. Die Verarbeitungsschritte sind im einzelnen:

1. Identifikation der Teilzustände
2. Festlegen des Startzustands und der Endzustände
3. Definition der Ereignisse, die erkannt werden sollen
4. Hinzufügen der Transitionen zwischen den Zuständen unter Verwendung der Ereignisse
5. Definition von Wachbedingungen auf den Transitionen
6. Definition der Aktionen, die durchgeführt werden sollen
7. Identifikation der Eintritts- und Ausgangsaktionen
8. Für alle Teilzustände Schritt 1 durchführen

Generell sollte darauf geachtet werden, dass der entstandene Zustandsautomat keine Überflüssigen Zustände oder Transitionen enthält; er sollte so einfach wie möglich und so komplex wie nötig sein. Es ist auch zu berücksichtigen, dass eine gute Balance zwischen hierarchischer Verschachtelung und planarer Organisation der Zustände eines Automaten gefunden wird. Außerdem sollten die Zustände und Transitionen aus dem Vokabular des Systems benannt werden um die Verständlichkeit zu erhöhen. Bei der Definition der Aktionen muss schon im Voraus auf eine gute Balance des benötigten Zeit- und Resourcenaufwands geachtet werden, welche diese voraussichtlich in der Implementierung benötigen werden.

Das folgende Beispiel illustriert einen simplen mobilen Agenten dessen Verhalten durch einen Zustandsautomaten spezifiziert ist:

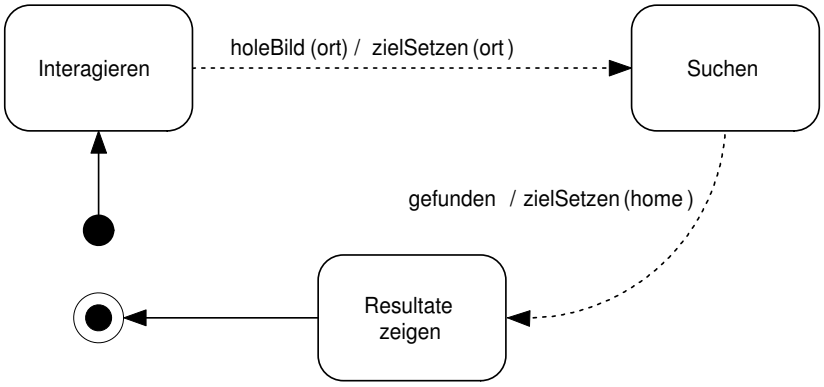

Abbildung 7.3: Das Zustandsdiagramm eines simplen mobilen Agenten.

Beispiel 7.1 *Der in Abbildung 7.3 dargestellte Zustandsautomat definiert das Verhalten eines mobilen Agenten, dessen Aufgabe es ist, eine vom Nutzer geforderte Information auf einem entfernten System zu suchen und dem Nutzer zu überbringen.*

Im Zustand Interagieren *fragt der Agent den Anwender nach der gewünschten Information und nach dem Ort, an dem die Suche stattfinden soll. Sind beide Angaben gemacht worden, so wird das Ereignis* holeBild *generiert. Der Parameter des Ereignisses ist der Ort, an dem gesucht werden soll. Die Migrations-Transition zwischen* Interagieren *und* Suchen *hat als Attribute das auslösende Ereignis* holeBild *und die Aktion* zielSetzten*. Die Aktion setzt den Parameter* ort *des Ereignisses als das Mirgations-Ziel des Agenten. Nach dem Auslösen der Transition migriert der Agent an den Zielort und der Zustand* Suchen *wird zum aktiven Zustand des Automaten.*

Am Zielort angekommen, sucht der Agent die angegebene Information. Ist diese gefunden, so wird das Ereignis gefunden *erzeugt. Dieses Ereignis löst eine Migrations-Transition zum Zustand* Resultate zeigen *aus. Das Ziel der Migration ist das Heimat-System* home *des Agenten. Dort angekommen wird der Zustand* Resultate zeigen *aktiv und der Anwender bekommt das Ergebnis der Suche präsentiert.*

Ähnlich wie bei der zustandsorientierten Modellierung wird auch bei der Modellierung durch Rollen eine Dekomposition der Rolle in Teilrollen vorgenommen. Eine atomare Rolle wird durch einen Zustandsautomaten repräsentiert, daher lässt sich eine Rolle als eine stärkere Abstraktion den Verhaltens ansehen, als dies ein Zustand ist. Es ist jedoch trotzdem möglich, dass ein Zustand sich aus mehreren Rollen zusammen setzt. Der Wechsel einer Rolle wird, in Analogie zur Transition zwischen Zuständen, ebenfalls durch eine solche Transition definiert.

Das folgende Beispiel illustriert eine zustandsorientierte Modellierung anhand der Dekompositon der Rollen eines Agenten:

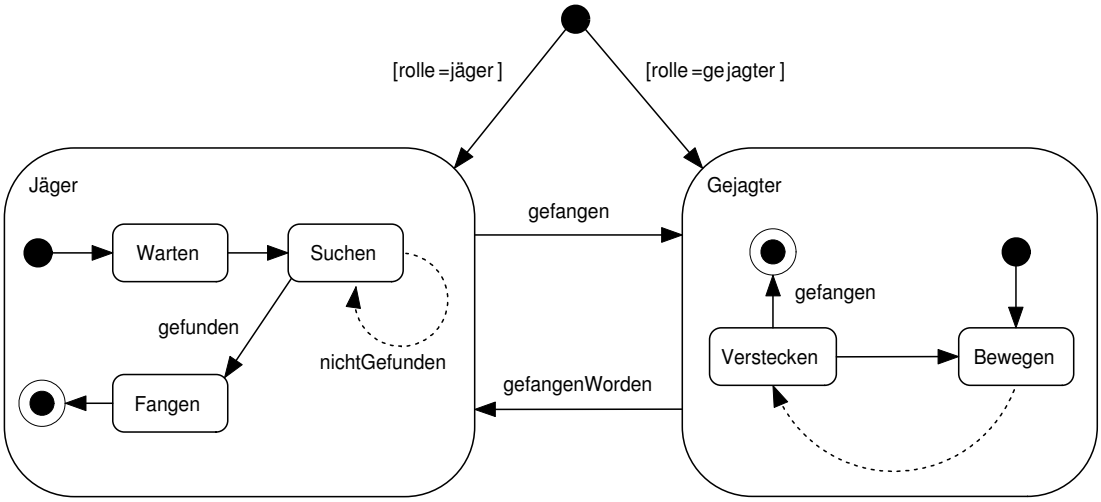

Abbildung 7.4: Das Zustandsdiagramm eines Agenten, der nach Rollen dekomponiert wurde.

Beispiel 7.2 *Abbildung 7.4 zeigt das Zustandsdiagramm eines mobilen Agenten, der das Spiel* Fangen und Verstecken *implementiert. Das Spiel wird von zwei Agenten gespielt, welche vom gleichen Typus sind und die Rolle des Jägers und des Gejagten dynamisch tauschen. Das Verhalten beider Agenten ist also durch den gleichen Zustandsautomaten beschrieben. Während des Spiels migrieren die Agenten durch ein begrenztes Netzwerk.*

Beim Start der Agenten wird angegeben, welche Rolle der jeweilige Agent übernehmen soll. Dies wird durch den Parameter rolle *ausgedrückt, welcher von den Wachbedingungen der vom Startzustand ausgehenden Transitionen ausgewertet wird.*

In der Rolle des Jägers wartet der Agent zunächst eine gewisse Zeit um dem Gejagten einen Vorsprung zu gewähren. Anschließend wechselt er in den Zustand Suchen*, in dem das lokale System durchsucht wird. Wurde der Gejagte gefunden, so wird das Ereignis* gefunden *erzeugt, welches einen Zustandswechsel nach* Fangen *auslöst. Andernfalls wird durch das Ereignis* nichtGefunden *eine Migrations-Transition ausgelöst welche den Jäger auf ein anderes System innerhalb des Netzwerks bewegt. Im Zustand* fangen *bekommt der Gejagte mitgeteilt, dass er gefangen*

worden ist, außerdem wird der Endzustand des Zustands Jäger *aktiviert und das Ereignis* gefangen *wird erzeugt. Der Agent in der Rolle* Jäger *schlüpft nun in die Rolle* Gejagter.

In Rolle des Gejagten begibt sich der Agent zunächst auf ein anderes System. Dies wird durch den Zustand Bewegen *und die Migrations-Transition, die zum Zustand* Verstecken *führt, ausgedrückt. Ist die Migration vollzogen so versteckt sich der Agent für eine bestimmte Zeit auf dem lokalen System. Wird der Gejagte vom Jäger gefangen, während er sich versteckt, so wird ihm dies durch das Ereignis* gefagen *mitgeteilt. In diesem Fall wird der Zustand* Gejagter *beendet und das Ereignis* gefangenWorden *wird generiert, welches einen Wechsel der Rolle von* Gejagter *zu* Jäger *zur Folge hat. Falls der Gejagte nicht entdeckt wird, so wird der Zustand* Bewegen *aktiviert und der Agent ändert erneut seine Position innerhalb des Netzwerks.*

Bei der orts-orientierten Modellierung des Verhaltens kommt es zu keiner Dekomposition der Orte, diese bilden daher eine flaches Netzwerk ohne Hierarchie. Der Wechsel eines Ortes wird durch eine Mirgations-Transition dargestellt. Ein Ort wird durch einen Zustandsautomaten repräsentiert. Dieser Automat kann gegebenenfalls wieder eine beliebig komplexe Hierarchie aufweisen.

Das folgende Beispiel illustriert eine zustandsorientierte Modellierung anhand der Dekompositon der Rollen eines Agenten:

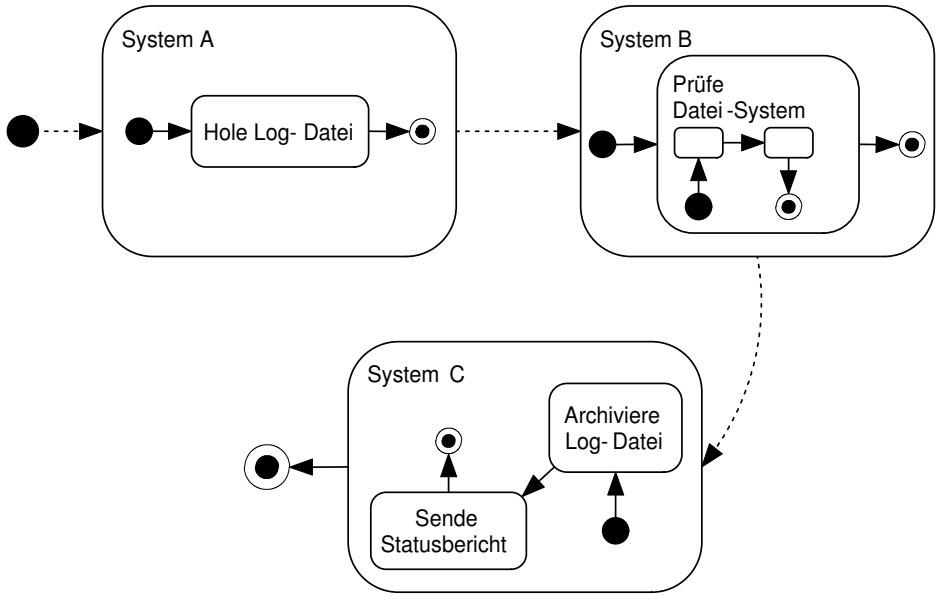

Abbildung 7.5: Das Zustandsdiagramm eines Agenten, der nach Orten dekomponiert wurde.

Beispiel 7.3 *In Abbildung 7.5 wird der Zustandsautomat eines mobilen Agenten dargestellt, der eine Reihe von Wartungsarbeiten auf verschiedenen Systemen durchführt. Jedes der besuchten*

Systeme wird durch einen einzelnen Zustand dargestellt. Je nach Komplexität der Aufgabe ist jeder dieser Zustände wiederum unterschiedlich zusammengesetzt.

Der Agent beginnt seine Reise auf dem System A, auf dem er eine Log-Datei abholt. Danach migriert er auf das System B. Dort wird das Datei-System überprüft, was durch einen nicht näher spezifizierten Zustandsautomaten dargestellt ist. Auf dem letzten besuchten System (C) wird die mitgebrachte Log-Datei archiviert und ein Statusbericht wird an den Anwender gesendet. Ist dies abgeschlossen, so wird der Agent terminiert.

7.3 Der Prozess

Der Prozess, der den Entwurf in die Implementierung überführt, ist aufgrund des verwendeten Modells für das Verhalten (den Zustandsautomanten) sehr einfach gehalten. Im Prinzip werden die Elemente aus der Entwurfsebene direkt auf Elemente der Implementierungsebene abgebildet. Diese Abbildung findet im Verhältnis eins zu eins statt. Das heißt, Modellelemente wie Agent, Zustandsautomat, Zustand, Transition, Aktion etc. haben eine explizite Repräsentation durch ein Objekt, das zur Laufzeit existiert. Aufgrund der Architektur der verwendeten Komponenten-Technologie ist es nicht notwendig, eine explizite Repräsentation aller Modellelemente als Java-Klasse zu erzeugen. Es müssen lediglich Instanzen der Klassen erstellt werden, welche die Elemente der Entwurfsebene repräsentieren. Diese können dann zur Laufzeit konfiguriert und miteinander in Beziehung gesetzt werden, so dass das Modell korrekt repräsentiert wird.

Sofern die auszuführenden Aktionen schon als wiederverwendbare Komponenten vorliegen, können diese auch einfach instanziert und in den Zustandsautomaten eingebunden werden. Ansonsten müssen die Aktionen eigens in Form von Klassen für den Zustandsautomaten erstellt werden.

Letztendlich muss also für jedes Modellelement eines Zustandsautomaten ein entsprechendes Laufzeitobjekt erzeugt werden. Dabei ist es egal, ob dies direkt durch ein Java Programm geschieht oder indirekt durch eine Meta-Definition, die ihrerseits von einem Programm umgesetzt wird.

7.4 Die Komponenten-Technologie

Die Implementierung von mobilen Agenten, deren Verhalten durch hierarchische Zustandsautomaten spezifiziert ist, wird im Rahmen des Gesamtkonzeptes durch ein Java Framework unterstützt. Außerdem wird eine Integration des Frameworks in SeMoA bereitgestellt. Das Framework besteht aus zwei Teilen:

- Einer Sammlung von Java-Klassen, die es ermöglichen eine *Datenstruktur* zu konstruieren, welche einen Zustandsautomaten repräsentiert
- Einem *Interpreter* für diese Datenstruktur

Die Klassen der Datenstruktur sind so entworfen worden, dass sich die Elemente eines Zustandsautomaten eins zu eins auf diese Klassen abbilden lassen. Es gibt also Repräsentationen für Zustände, Transitionen, Aktionen, Ereignisse und Bedingungen. Die Klassen selbst dienen fast alle ausschließlich dazu, die repräsentative Datenstruktur eines Zustandsautomanten zu bilden. Eine Ausnahme bilden die Aktionen, in denen die eigentliche Applikationslogik gekapselt ist. Die Datenstruktur lässt sich als eine Hierarchie von Zuständen auffassen, die einen gerichteten, azyklischen Graphen bilden. Unter Verwendung des *Java Beans* Konzeptes [27] wurden die Klassen als eigenständige Komponenten implementiert, die sich auch zur Laufzeit konfigurieren lassen.

Der Interpreter dient zur Steuerung des Kontrollflusses, der in einem Zustandsautomaten abläuft. Der Interpreter regelt, welcher Zustand zu einem gegebenen Zeitpunkt aktiv ist, wann eine Transition stattfindet und welche Aktionen in welcher Reihenfolge ausgeführt werden.

Im Allgemeinen ist die Voraussetzung für die zustandsorientierte Modellierung eine Definition des Kontextes, in dem sich der Zustandsautomat befindet. Der Kontext stellt eine Ansammlung von Funktionalität und Informationen dar, auf welche die Aktionen, die im Rahmen des Automaten ausgeführt werden, Zugriff haben. Er wird also jeweils durch den Agenten und durch die Umgebung des Agenten bestimmt. Das Framework stellt zu diesem Zweck eine Kontext-Komponente zur Verfügung, welche die Rolle dieses Kontextes übernimmt. Es handelt sich dabei um eine Art von Verzeichnis, in dem sich beliebige Java-Objekte unter einem eindeutigen Namen ablegen lassen. Über diesen Namen lassen sich die Objekte wieder abrufen und so zwischen verschiedenen Aktionen und Bedingungen eines Zustandsautomaten lokal verteilen. Dies ist beispielsweise dann sinnvoll, wenn eine Bedingung den Wert einer Variable benötigt, die in der Aktion eines Zustands gesetzt wurde. Über den Kontext lässt sich der Austausch der Variable entkoppeln, so dass die Bedingung keine Referenz der Aktion benötigt.

Um mit den Anforderungen umzugehen, die durch die Mobilität eines Agenten entstehen, wird eine besondere Art der Transition unterstützt: Die *Migrations-Transition*. Durch diese Transition lässt sich der Aspekt der Mobilität in einem erweiterten Zustandsdiagramm ausdrücken und direkt in die Implementierung abbilden.

Für die Integration in SeMoA wurde ein eigenes Framework entwickelt, das unter anderem eine Basisklasse für Agenten bereitstellt. Diese Basisklasse lässt sich durch einen beliebigen Zustandsautomaten parametrisieren, der das Verhalten des Agenten bestimmt. Der interne Interpreter des Agenten steuert dabei den Ablauf des Zustandsautomanten, um das Verhalten zu realisieren. Ein weiterer wichtiger Beitrag ist die Anbindung der Zustandsautomaten an das Kommunikationsystem von SeMoA. Dafür steht ein spezieller Adapter bereit, der System-Nachrichten

in Ereignisse übersetzt, die der Interpreter für den Automaten des Agenten verarbeiten kann. Insgesamt ist das Framework als erweiterbare Grundlage für die Entwicklung von SeMoA-Agenten gedacht. Es bietet eine Reihe von vorgefertigten, häufig benötigten Elementen für die Implementierung dieser Agenten. Dazu gehören beispielsweise wiederverwendbare Aktionen wie die *Migrationsaktion*, welche für die Kopplung an eine Migrations-Transition gedacht ist. Diese Aktion setzt ein neues Migrationsziel für den Agenten bevor dieser migriert.

Abschließend sei noch angemerkt, dass die Entwurfsmethode und die Komponenten-Technologie sich auch unabhängig von einander einsetzten lassen. Der Kern der Komponenten-Technologie, das Framework für die Erstellung von Zustandsautomaten, ist insbesondere so gestaltet, dass er sich auch in ganz anderen Bereichen der Software-Entwicklung einsetzten lässt. Er kann als generische Unterstützung für die zustandsorientierte Programmierung in Java betrachtet werden. Für alle Elemente des Frameworks existieren jeweils separate Schnittstellen, so dass sich die Implementierungen der Komponenten mit wenig Aufwand austauschen lassen. Insbesondere lässt sich für bestimmte Anwendungsfälle ein neuer Interpreter schreiben, der die Semantik der Zustandsautomaten geeignet anpasst, ohne den Rest des Frameworks ändern zu müssen.

Kapitel 8

Analyse

> *All truths are easy to understand once they are discovered; the point is to discover them.* — Galileo Galilei

Wie in Kapitel 7 bereits vorgestellt, wird in dieser Arbeit ein auf hierarchischen Zustandsautomaten basierender Ansatz zur Modellierung und Implementierung von mobilen Agenten entwickelt. In diesem Kapitel soll eine Begründung für die Wahl von Zustandsautomaten auf Basis der UML als Modellierungsgrundlage gegeben werden. Dabei wird untersucht, in wie weit der vorgestellte Ansatz den Anforderungen genügt, die in Kapitel 6 gestellt wurden.

Es gibt zwei zentrale Fragen, die im Rahmen dieser Analyse beantwortet werden sollen:

- Ist zustandsbasiertes Agentenverhalten ein nützliches Paradigma?
- Welche Unterstützung sollte eine Plattform für die zustandsorientierte Agenten-Programmierung bieten?

Zunächst wird die Ausgangslage der Arbeit dargestellt. Begonnen wird mit einer Zusammenfassung der Anforderungen und einer anschließenden Betrachtung inwiefern andere Ansätze diese Anforderungen erfüllen. Daraus ergeben sich einige Schwachstellen existierender Arbeiten, welche durch diese Arbeit beseitigt werden sollen. Anschließend wird die zustandsorientierte Modellierung bezüglich der Anforderungen untersucht. Aus dieser Betrachtung ergeben sich einige zusätzliche Rahmenbedingungen für die Modellierung durch Zustandsautomaten, welche auf eine Verbesserung der Modularität der Modellelemente, sowie der daraus resultierenden Implementierung abzielen. Diese Rahmenbedingungen werden abschließend dargestellt und bereiten den Übergang von der Analyse zum Entwurf des entwickelten Konzeptes, der in Kapitel 9 erarbeitet wird.

8.1 Die Ausgangslage

In Kapitel 6 wurde die Entwicklung eines Konzeptes gefordert, mit dessen Hilfe sich mobile Agenten und deren Verhalten derart erstellen lassen, dass sowohl der Entwurf des Agenten, als auch die Implementierung in hohem Maße wiederverwendbar und erweiterbar ist. Wiederverwendbarkeit und Erweiterbarkeit sollen durch Modularität der Resultate gewährleistet werden, die durch die Anwendung der geforderten Methodologie und des zugehörigen Implementierungsprozesses entstehen. Zur besseren Bewertung der Modularität wurden fünf Kriterien vorgestellt:

- Dekomponierbarkeit
- Komponierbarkeit
- Verständlichkeit
- Kontinuität
- Schutz

Wie in Abschnitt 2.6 dargestellt, bietet SeMoA keine Unterstützung für die Entwicklung von Agenten. Es existiert kein Konzept für den Entwurf von Agent und auch keine Unterstützung seitens der Software. Es existiert kein Framework auf dessen Grundlage sich mobile Agenten konstruieren lassen, auch eine Basisklasse für Agenten ist nicht vorhanden.

In Kapitel 4 wurde gezeigt, dass einige interessante Ansätze für die Modellierung von Agenten existieren. Es wurde jedoch auch klar, dass der Übergang von der Modellebene zur Implementierung häufig nicht berücksichtigt wird.

In Kapitel 5 wurde eine Reihe von Ansätzen für die Modellierung und Implementierung von Agenten, beziehungsweise mobilen Agenten, beschrieben. Als viel versprechender Ansatz hat sich die Modellierung des Verhaltens durch Zustandsautomaten herausgestellt. Als Gründe dafür sind die Einfachheit, Ökonomie, Eleganz und der graphische Bezug des Ansatzes zu nennen. Weiterhin sorgt die weite Verbreitung der UML in vielen Bereichen der Informatik für einen hohen Bekanntheitsgrad der Zustandsdiagramme und der darin verwendeten Konzepte. Diese Punkte sprechen für die Anwendbarkeit und Verständlichkeit der zustandsorientierten Modellierung.

Was bisher noch nicht ausreichend berücksichtigt wird, ist die Modularität der Resultate, die durch die Modellierung mittels Zustandsautomaten entstehen. Dies wird in diesem Kapitel noch im Detail diskutiert werden. Ein weiterer Punkt ist die Darstellung der Mobilität von Agenten. Diese wird durch Zustandsdiagramme nicht explizit unterstützt.

Diese Schwachstellen lassen sich nach Meinung des Autors durch das entwickelte Konzept beheben. Um das darzustellen, wird nun die zustandsorientierte Modellierung im Detail betrachtet.

8.2 Zustandsorientierte Modellierung

In diesem Abschnitt wird die Anwendbarkeit der Verhaltensmodellierung durch hierarchische Zustandsautomaten analysiert. Dazu betrachten wir zunächst einige inhärente Eigenschaften der zustandsorientierten Modellierung im Hinblick auf die gestellten Anforderungen. Anschließend wird die Anwendbarkeit im Kontext von mobilen Agenten untersucht.

8.2.1 Modularität

Um zu klären, in welchem Ausmaß die zustandsorientierten Modellierung Modularität unterstützt, wird nun die Umsetzung der fünf Kriterien der Modularität nach *Meyer* [43] in dieser Entwurfstechnik untersucht.

Modulare Dekomponierbarkeit. Die Dekomposition von Zuständen in Teilzustände ist eines der zentralen Konzepte der hierarchischen Zustandsautomaten. Die Modellierungsmethode an sich zielt darauf ab, dynamisches Verhalten in Teile zu zerlegen und diese Teile zueinander in Bezug zu setzen. Teilzustände haben per Definition eine geringere Komplexität, als der Zustand der sie enthält, dieser Teil des Kriteriums wird daher direkt unterstützt. Nicht ganz so selbstverständlich ist jedoch die Umsetzung der Unabhängigkeit der einzelnen Teilzustände gegeben. Bloße Dekomposition von Zuständen sorgt nicht a priori für Unabhängigkeit. Dies zeigt sich besonders deutlich im Falle von Transitionen, die zwischen zwei Zuständen definiert sind, welche sich auf unterschiedlichen Hierarchie-Ebenen befinden.

Beispiel 8.1 *Abbildung 3.1 auf Seite 33 zeigt einen Zustandsautomaten, der eine grenzüberschreitende Transition (border crossing transition) enthält. Diese ist zwischen den Zuständen* `Ausgeben` *und* `Leerlauf` *definiert.*

In diesem Fall kann der Zustand, dessen Grenze durch die Transition verletzt wird, nicht unabhängig von dem anderen Zustand, der außerhalb des geschnittenen Zustands liegt, wiederverwendet werden. Um dies zu vermeiden, muss eine zusätzliche Beschränkung für die Entwurfsmethode eingeführt werden, welche das Verwenden von grenzüberschreitenden Transitionen verbietet.

Sind die Teilzustände jedoch einmal so gestaltet, dass sie tatsächlich unabhängig verändert werden können, so wird die modulare Dekomponierbarkeit insgesamt durch die zustandsorientierte Modellierung in hohem Maße unterstützt. Gleichzeitig wird dadurch auch die Erfüllung des Kriteriums der modularen Komponierbarkeit erleichtert.

Modulare Komponierbarkeit. Auch die modulare Komponierbarkeit ist ein zentrales Konzept der zustandsorientierten Modellierung: Mehrere Zustände lassen sich zusammensetzen um einen neuen Zustand zu bilden; ganze Zustandsautomaten lassen sich prinzipiell aus bestehenden Zuständen, Transitionen, Aktionen und Bedingungen konstruieren. Da der Entwurf in der Regel *top-down*, also vom Abstrakten hin zum Konkreten, stattfindet, ist die Erfüllung des Kriteriums an eine saubere modulare Dekomposition gebunden, welche unabhängige Teilzustände hervorbringt. Wird die Dekomposition in dieser Form durchgeführt, so ist dadurch auch die Komponierbarkeit der Modellelemente erfüllt. Bezüglich der Modellebene ist es also nicht sehr schwierig, eine gute modulare Komponierbarkeit durch Anwendung der zustandsorientierten Modellierung zu erreichen.

Soll jedoch die Implementierung eines Zustands in einem fremden Kontext wiederverwendet werden, so muss ein Entwickler mit Vorsicht zur Tat schreiten. Die Implementierung der funktionstragenden Elemente, also den Aktionen und Bedingungen, muss sorgsam, mit möglichst wenigen Abhängigkeiten und unter der Vermeidung von Seiteneffekten in den Programmteilen geschehen. Ansonsten kann trotz einer guten modularen Komponierbarkeit der Modellelemente das Kriterium in der Implementierung nicht erfüllt werden.

Modulare Verständlichkeit. Die Verständlichkeit von Zustandsdiagrammen ist sicherlich der Hauptgrund für deren Verwendung in Theorie und Praxis. Allein durch die Tatsache, dass Zustandsautomaten sich gut durch Diagramme beschreiben lassen, also durch eine speziell dafür ausgelegte Symbolsprache, die nur wenige textuelle Elemente enthält, ist viel zur Verständlichkeit beigetragen. Dies lässt sich dadurch erklären, dass der menschliche Wahrnehmungsapparat sehr stark auf die Aufnahme und Interpretation von visuellen Informationen ausgelegt ist.[1] Eine graphische Repräsentation bietet jedoch nicht nur Vorteile bei der Aufnahme von Modellen sondern auch das Ausdrücken von mentalen Modellen in graphischen Repräsentationen wird dadurch unterstützt. Eine wichtige Schlussfolgerung daraus ist, dass auch die Kommunikation zwischen mehreren Entwicklern, die gemeinsam an einem Modell arbeiten, verbessert werden kann. Von einer abstrakten Perspektive aus betrachtet lässt sich dieser Vorgang als Synchronisation der jeweilige mentalen Modelle der Teilnehmer auffassen.

Der Knackpunkt der modularen Verständlichkeit ist das Verständnis von unabhängigen Elementen, die isoliert betrachtet und verstanden werden wollen. Dies ist sicherlich für Zustandsdiagramme gegeben: Zustände sind die Repräsentation von unabhängigen Zeiträumen und Handlungen die innerhalb dieser Zeiträume ausgeführt werden. Alles was für das Verständnis eines Zustandes benötigt wird, ist ausschließlich im Rahmen der Definition des Zustandes vorhanden. Die Zusammenhänge zwischen verschiedenen Zuständen werden explizit durch Transitionen modelliert, daher lassen sich diese Zusammenhänge unabhängig von den Zuständen betrachten.

[1] Der menschliche Wahrnehmungsapparat ist Gegenstand der psychologischen Forschung, in der in den letzten Jahren große Fortschritte auf diesem Gebiet gemacht wurden [22, 41].

Auch in der Transition sind alle relevanten Elemente, wie das auslösende Ereignis, die Wachbedingung oder die Aktion, lokal definiert und können ohne den Rest des Automaten in Betracht zu ziehen verstanden werden. Auf einer höheren Ebene gilt dies auch für ganze Zustandsautomaten, die vollständig über die in ihnen enthaltenen Zustände und Transitionen definiert sind.

Zusammenfassend lässt sich also sagen, dass die modulare Verständlichkeit durch die zustandsorientierte Modellierung sehr stark unterstützt wird.

Modulare Kontinuität. Um die Erfüllung dieses Kriteriums durch die zustandsorientierte Modellierung beurteilen zu können, muss zunächst definiert werden, was in diesem Kontext als kleine Änderung der Spezifikation zu verstehen ist. Ein kleine Änderung in einem Zustandsautomaten ist das Ändern oder Hinzufügen einer Transition oder eines Zustands. Unter der Änderung einer Transition kann ein Ersetzen des auslösenden Ereignisses, das Anpassen der Wachbedinung oder das Ändern der Spezifikation der Transitions-Aktion verstanden werden. Eine Änderung eines Zustands bezeichnet ein Ersetzten oder Ändern der Spezifikation der Zustands-, Eintritts- oder Austritts-Aktionen oder ein Aufbrechen des Zustands in mehrere Teilzustände.

Wie nun die Implementierung auf solche kleinen Änderungen der Spezifikation reagiert, hängt von der Art der Abbildung der Modellelemente in die Implementierungsebene ab. Idealer Weise findet diese Abbildung im Verhältnis eins zu eins statt, das heißt ein Modellelement wird durch genau ein Element in der Implementierung repräsentiert und umgekehrt. In diesem Fall sind nur genau die Teile der Implementierung betroffen, die auch im Modell geändert wurden.

Dieses Kriterium wird also nicht unmittelbar von der Modellierung durch Zustandsautomaten selbst gefördert, sondern hängt von dem Prozess ab, der das Modell in die Implementierung überführt.

Modularer Schutz. In dem zustandsorientierten Modell werden alle möglichen Zustände explizit erfasst und im Modell dargestellt, die im Leben eines Agenten auftreten können. Weiterhin werden alle Reaktionen auf sämtliche Ereignisse, die berücksichtigt werden sollen, auf entsprechende Transitionen abgebildet. Abnorme Ereignisse und Zustände werden von der Modellierung ausgeschlossen. Die Behandlung dieser Fälle bleibt also der Implementierung eines Zustandsautomaten überlassen. Wie das konkret geschieht, hängt stark von der verwendeten Programmiersprache und dem angewandten Programmierstil ab. Im Rahmen dieser Arbeit ist die Sprache Java, und die Vorgehensweise bei der Programmierung richtet sich nach den Methoden und Konzepten des objekt-orientierten Software Engineerings.

Dieses Kriterium wird daher nach Meinung des Autors nicht sonderlich durch zustands-orientierte Modellierung gefördert, es wird jedoch auch nicht im negativen Sinne beeinflusst.

8.2.2 Im Kontext mobiler Agenten

Um beurteilen zu können, in welchem Ausmaß sich zustandsorientierte Modellierung im Gebiet der mobilen Agenten anwenden lässt, werden im Folgenden typische Verhaltensmerkmale mobiler Agenten unter diesem Gesichtspunkt diskutiert.

In Agenten-Kontext lässt sich das Verhaltensmodell als Vermittler zwischen der Umgebung des Agenten und dem Agenten selbst betrachten. Abbildung 8.1 stellt diesen Zusammenhang dar: Der Agent hat ein gewisses Vorhaben, das durch den Zustandsautomaten in eine Aktion umgesetzt wird, wenn alle Bedingungen dafür erfüllt sind. Die Aktion hat eine Auswirkung auf die Umgebung welche sich als Ereignis repräsentieren lässt. Dieses Ereignis wird vom Zustandsautomaten registriert und stellt somit eine Wahrnehmung des Agenten dar.

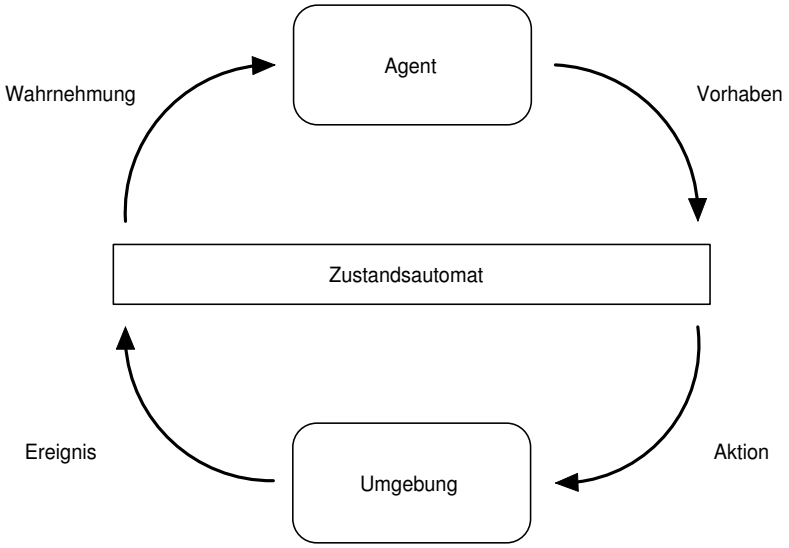

Abbildung 8.1: Der Zustandsautomat als Vermittler zwischen Agent und Umgebung.

Betrachten wir zunächst die Verhaltensmerkmale mobiler Agenten. In Abschnitt 2.1 wurden die essentiellen Charakteristika von Agenten vorgestellt. Inklusive der für mobile Agenten geforderten Mobilität lauten diese: Autonomie, Kommunikationsfähigkeit, Reaktivität, Proaktivität und Mobilität.

Prinzipiell lassen sich alle diese Verhaltensmerkmale sicherlich durch Zustandsautomaten modellieren. Es bleibt zu klären wie leicht das im jeweiligen Fall geschehen kann.

Am offensichtlichsten ist dies wohl bei der Reaktivität zu erkennen. Es ist ein inhärentes Konzept der zustandsorientierten Modellierung, dass Handlungen als Reaktionen auf das Auftreten gewisser Ereignisse erfolgen können. Schließlich ist der Wechsel von Zuständen vor allem durch das Auslösen von Transitionen durch bestimmte Ereignisse definierbar. Reaktives Verhalten lässt sich also direkt in ein zustandsorientiertes Modell abbilden. Aus dieser Perspektive

wird deutlich, dass die Modellierung auf Basis von Zustandsautomaten eine Form von reaktiver Architektur hervorbringt (siehe Abschnitt 4.1.2).

Aus dieser Erkenntnis lässt sich auch die Umsetzung der Kommunikationsfähigkeit ableiten. Eine simple Art der Kommunikation zwischen zwei Agenten lässt sich durch zwei Zustandsautomaten modellieren, die Nachrichten in Ereignisse kapseln und diese über einen externen Vermittler austauschen. Die Zustände des jeweiligen Automaten werden in diesem Fall als Schritte in einem Kommunikationsprotokoll interpretiert, das beide Parteien abarbeiten. Es können auf diese Weise auch komplette Verhandlungsprotokolle durch Zustandsautomaten beschrieben werden.

Wurde bisher stark mittels des Ereignis-Konzepts argumentiert, so ergibt sich die Proaktivität wenn man Ereignisse gewissermaßen vernachlässigt: Ein Zustandsautomat, auf dem keine auslösenden Ereignisse definiert sind und der einen vorgegebenen Algorithmus abarbeitet, stellt eine sehr einfache Form von zielgerichtetem Verhalten dar; vorausgesetzt der Algorithmus ist selbst zielgerichtet definiert. Proaktivität ist in diesem Fall implizit gegeben. Auch allgemein lässt sich festhalten, dass in Modellen, die auf Zustandsautomaten basieren, das Ziel nicht unmittelbar in der Darstellung zu erkennen ist, auch wenn es tatsächlich vorhanden ist. Anders ist dies beispielsweise in BDI-Architekturen, in denen es Elemente gibt, die ein Ziel explizit repräsentieren (siehe Abschnitt 4.1.3).

Etwas problematisch ist die Einschätzung der Autonomie. Dies ist zum größten Teil in der schlechten Definierbarkeit des Begriffs selbst begründet. Autonomie ist vermutlich die Eigenschaft, die am stärksten mit mobilen Agenten in Verbindung gebracht wird. Dennoch wird Autonomie oft einfach als gegeben vorausgesetzt, das heißt Agenten sind per Definition autonom. In diesem Fall ist Autonomie schon dadurch gegeben, dass der Agent ein aktives Objekt ist, also über einen dedizierten Ausführungsprozess verfügt, und in der Lage ist, ohne Rückfragen eine spezifizierte Aufgabe zu erledigen. Für Andere ist Autonomie jedoch ein wichtige, aber schwer zu realisierende Eigenschaft, der besondere Aufmerksamkeit gewidmet werden muss. Nach dieser Interpretation bedeutet Autonomie die absolute Unabhängigkeit, sowohl von anderen Agent als auch vom Nutzer des Agenten, außerdem ist der Agent in der Lage sein Verhalten selbst zu formen und seiner Umgebung anzupassen. Nach *Luck* und *d'Inverno* [39] ist dieser strenge Begriff der Autonomie praktisch nicht zu realisieren. Der schwächere Autonomiebegriff kann jedoch sicherlich durch eine Reihe von Ansätzen realisiert werden. Dazu gehört auch die Steuerung des Verhaltens durch Zustandsautomaten.

Bleibt zu klären wie die Umsetzung der Mobilität stattfindet. Mobilität ist explizit nicht durch die Elemente eines Zustandsautomaten erfasst. Wird der Vorgang einer Migration als Handlung interpretiert, lässt sich diese in Form einer Aktion in Zustandsautomaten abbilden. Dies ist jedoch aus Gründen der Verständlichkeit nicht optimal: Die Mobilität ist ein zentrales Konzept des mobilen Agenten, es ist sehr wichtig zu verstehen wann ein Ortswechsel stattfindet, wenn man das Verhalten des Agenten nachvollziehen will. Außerdem ist mit der Migration stets ein

Einstellen der Aktivität des Agenten verbunden, sowie ein Speichern des aktuellen Ausführungszustands, so dass der Agent nach der Migration an der entsprechenden Stelle seine Arbeit wieder aufnehmen kann. Dies wird durch eine Aktion nicht deutlich genug abgebildet, da eine Aktion kein graphisches Element ist. Die Migration sollte daher durch ein eigenes visuelles Element im Zustandsautomaten repräsentiert sein.

8.3 Konsequenzen für den Lösungsansatz

Wie im vorangegangenen Abschnitt deutlich wurde, lassen sich viele modulare Kriterien und Eigenschaften des Verhaltens von mobilen Agenten gut durch Zustandsdiagramme, die auf der UML basieren, abbilden. Für eine geeignete Umsetzung einiger dieser Kriterien müssen jedoch besondere Rahmenbedingungen durch den Lösungsansatz geschaffen werden. Diese sollen im Folgenden vorgestellt werden.

8.3.1 Modifikationen der UML

Als Konsequenz der vorangegangen Analyse ergeben sich einige Änderungen an der Syntax und Semantik der Zustandsdiagramme der UML. Dadurch entsteht eine Art Dialekt dieser Modellierungssprache, der die Grundlage des Lösungsansatzes bildet. Die Änderungen werden nun im Einzelnen vorgestellt.

Um die Mobilität eines Agenten gut in Zustandsdiagrammen repräsentieren zu können, wurde das Hinzufügen eines speziellen graphischen Elementes angesprochen. Dieses Element wird als *Migrations-Transition* bezeichnet. Eine Migrations-Transition wird ähnlich verwendet und verfügt über die gleichen Attribute wie eine gewöhnliche Transition, es kann also ein auslösendes Ereignis und eine Wachbedingung auf ihr definiert werden. Eine Migrations-Transition wird zwischen einem Quell- und einem Zielzustand definiert, wobei der Quellzustand nach dem Auslösen der Transition deaktiviert wird und der Zielzustand der aktive Zustand des Automaten wird. Bei dem Wechsel des Zustands durch eine Migrations-Transition wird jedoch gleichzeitig ein Wechsel des Ausführungsortes des Agenten vollzogen.[2] Die Migrations-Transition wird durch einen Transitionsspfeil dargestellt, dessen Basis „gestrichelt“ ist (vgl. Abschnitt 7.2.1).

Für die Umsetzung der Dekomponierbarkeit wird gefordert, dass keine *grenzüberschreitenden Transitionen* in Zustandsautomaten definiert werden dürfen.[3] Dies kann auf der Modellierungsebene nur durch ein simples Verbot dieser Art der Transition umgesetzt werden, das heißt der Entwickler muss sich der Problematik bewusst sein und seine Modelle entsprechend erstellen.

[2] In der Implementierung bedeutet dies, dass die Ausführung des Automaten angehalten werden muss, damit dieser serialisiert und mit dem Agenten an den Bestimmungsort geschickt werden kann.

[3] Siehe auch [66].

Auf der Implementierungsebene ist es jedoch möglich ein gegebenes Modell daraufhin zu überprüfen, ob eine solche Transition vorliegt oder nicht. Dies wird dem Entwickler dann mitgeteilt, so dass dieser seinen Entwurf überarbeiten kann.

Nach Meinung des Autors sind auch Anpassungen nötig, die nicht direkt mit den genannten Kriterien in Zusammenhang stehen. Konkret ist damit das Entfernen von nebenläufigen Teilzuständen (*concurrent substates*) und Aktivitäten (*activities*, auch *do-activity*) aus der Modellierungssprache gemeint.

Nach *Simons* ist das Konzept der Aktivität ein Element, welches das Modell des Zustandsautomaten unnötig kompliziert macht [66]. Unnötig deshalb, weil Aktivitäten vollständig durch Dekomposition des betreffenden Zustands in Teilzustände beziehungsweise Sub-Zustandsautomaten modelliert werden können. Aktivitäten sind also eigentlich als Teil des Verhaltens, das durch einen Sub-Zustandsautomaten repräsentiert wird, anzusehen. Weil die Dekomposition eines der Kernkonzepte der zustandsorientierten Modellierung ist, sollte diese der Verwendung von Aktivitäten vorgezogen werden.

Auf der Ebene der Modellierung mag das Konzept der nebenläufigen Teilzustände sehr nützlich sein, weil es gewisse parallel ablaufende Prozesse gut darstellen kann. Nach Meinung des Autors eignet sich jedoch gerade die zustandsorientierte Modellierung dazu, parallele Prozesse in anderer Form zu modellieren. Dieser alternative Ansatz basiert auf dem Paradigma der reaktiven Programmierung und ist besser für die für Implementierung von Modellen geeignet, als nebenläufige Teilzustände. In der reaktiven Programmierung werden Systeme betrachtet deren Verhalten sich aus Reaktionen auf bestimmte Ereignisse zusammen setzen (vgl. 4.1.2). Reaktiven System setzten sich aus Komponenten zusammen, die parallel arbeiten. Diese nebenläufigen Komponenten kommunizieren und synchronisieren sich durch das versenden von globalen Nachrichten (*broadcasting*) beziehungsweise Ereignissen.

Übertragen auf die zustandsorientierte Modellierung sind diese Komponenten mit unabhängigen Zustandsautomaten gleich zu setzten. Die Automaten interagieren, indem sie untereinander Ereignisse empfangen und generieren. Es werden also keine Synchronisationspunkte wie bei der Modellierung durch nebenläufige Teilzustände benötigt, da diese Interaktionen asynchron ablaufen. Dies passt außerdem sehr gut zu der Annahme, dass Agenten als aktive Objekte betrachtet werden können (vgl. Abschnitt 2.5): Nach *Booch* ist die Modellierung der Verhaltens eines Objektes durch aktive Objekte der Modellierung durch nebenläufige Teilzustände unter bestimmten Umständen vorzuziehen [5, S. 254]. Dies ist genau dann der Fall, wenn die nebenläufigen Teile des Verhaltens nicht vom Zustand des jeweiligen anderen Verhaltens abhängen, sondern eher von speziellen Informationen, welche geeignet durch Nachrichten repräsentiert werden können.

Insgesamt kann durch die Unterstützung der Modellierung von nebenläufigen Prozessen ein hohes Maß an Verständlichkeit und Robustheit geschaffen werden. Ein Entwickler kann es auf diese Weise vermeiden, sich mit der direkten nebenläufigen Programmierung zu beschäftigen.

Probleme wie korrupte Daten, *deadlocks*, *missed notifications*, etc. fallen also nicht mehr an, da keine explizite Synchronisation oder Wartemechanismen mehr benötigt werden.

8.3.2 Bezüglich der Implementierung

Wie bei der Diskussion der fünf Kriterien der Modularität bereits angemerkt wurde, ist es günstig für die modulare Kontinuität, wenn die Umsetzung eines zustandsorientierten Modells in eine Implementierung in einem Verhältnis von eins zu eins stattfindet. Ist die zu Grunde liegende Software-Architektur derart gestaltet, dass dies möglich ist, so ergibt sich ein weiter Vorteil: Der Prozess, der ein Modell in die Implementierung überführt, ist trivial. Das hat nicht nur zur Folge, dass der Entwickler weniger Aufwand betreiben muss, die Implementierung selbst wird dadurch auch leichter verständlich. Ein Entwickler muss also nur die Konzepte der zustandsorientieten Modellierung beherrschen und wird dadurch in die Lage versetzt zu modellieren, das Modell zu implementieren und außerdem vorliegende fremde Implementierungen schnell zu begreifen, ohne das ursprüngliche Modell zu kennen.

Kapitel 9

Entwurf

> *A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.* — Douglas Noel Adams

In diesem Kapitel wird der Entwurf der Komponenten-Technologie für die zustandsorientierte Entwicklung vorgestellt. Innerhalb des Gesamtkonzeptes, das in dieser Arbeit entwickelt wurde, ist es die Hauptaufgabe der Technologie, die Abbildung von Modellen, die auf Zustandsautomaten basieren, auf wiederverwendbare Software-Komponenten zu ermöglichen.

9.1 Drei Schichten

Die Komponenten-Technologie setzt sich aus drei Schichten zusammen, die auf einander aufbauen. Jede dieser Schichten wird durch ein eigenes Paket repräsentiert. Die unterste Schicht bildet ein Framework mit dessen Hilfe sich generische Zustandsautomaten in Software-Komponenten abbilden lassen. Dieses Framework trägt den Namen JHSM, was für *Java Hierarchical State Machines*[1] steht.

Die folgende Schicht stellt ein Framework für die Entwicklung des Verhaltens von Agenten dar, das auf JHSM aufbaut. Der Name des Frameworks lautet AMoA und steht für *Agent Modelling Architecture*[2]. AMoA implementiert die in Kapitel 8 vorgestellen Änderungen gegenüber der auf UML basierenden Zustandsdiagrammen beziehungsweise Zustandsautomaten. Das Framework ist unabhängig von SeMoA und ist daher auch für den Einsatz in anderen Plattformen geeignet.

Die letzte Schicht ist eine Anbindung von AMoA an die SeMoA-Plattform. Im Zentrum steht dabei eine Basisklasse für SeMoA Agenten, die mit Zustandsautomaten parametrisierbar ist, welche mit JHSM erstellt wurden. Diese Schicht hat keinen speziellen Namen.

[1] Dies bedeutet in etwa „hierarchische Zustandsautomaten in Java“.

[2] Eine Architektur für die Modellierung von Agenten.

Abbildung 9.1 stellt die drei Schichten dar. Die Perspektive ist so gewählt, dass eine Schicht jeweils auf der darunter liegenden Schicht aufbaut. Die folgenden Abschnitte befassen sich mit diesen Schichten im Detail.

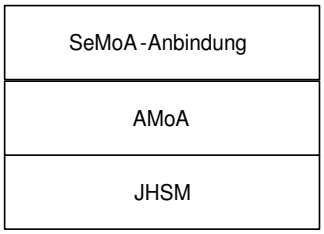

Abbildung 9.1: Die drei Schichten der Komponenten-Technologie.

9.2 Hierarchische Zustandsautomaten in Java

Das JHSM Framework stellt den Kern der gesamten Komponenten-Technologie dar. Es wurde so gestaltet, dass sich das Modell eines Zustandsautomaten eins-zu-eins auf die Elemente des Frameworks abbilden lässt. Daher existiert für jedes der Elemente *Zustand*, *Transition*, *Ereignis*, *Aktion* und *Bedingung* eine Repräsentation durch eine Schnittstelle (*interface*). In der entsprechenden Reihenfolge lauten die Namen der Schnittstellen: `State`, `Transition`, `Event`, `Action` und `Condition`. Diese Schnittstellen repräsentieren die *Basiselemente* eines Zustandsautomaten. Aus der Implementierung der Basiselemente lässt sich dynamisch eine *Datenstruktur* generieren, welche den Zustandsautomanten verkörpert. Die Ausführung des Automaten wird durch einen speziellen *Interpreter* vorgenommen, der die Datensturktur interpretiert. Der Interpreter wird durch die Schnittstelle `Interpreter` repräsentiert.

Das Framework wurde so konzipiert, dass die einzigen Elemente deren Implementierung später anwendungsspezifischen Code enthalten wird, durch die Schnittstellen `Action`, `Condition` und `Event` repräsentiert sind. Der Rest der Basiselemente kann ohne Änderung in verschiedensten Szenarien wiederverwendet werden.

Für die für die Einbettung der Aktionen und Bedingungen in einen Ausführungskontext steht eine spezielle Abstraktionsschicht, die *Kontextschicht*, zur Verfügung (vgl. Abschnitt 7.4). Diese Schicht stellt den Kontext des Zustandsautomaten aus der Perspektive der Aktionen und Bedingungen dar und wird durch die Schnittstelle `Context` repräsentiert.

Für den Austausch der Ereignisse steht eine weitere Abstraktionsschicht, die *Ereignisschicht*, zur Verfügung, welche durch die Schnittstelle `EventBus` repräsentiert wird.

Insgesamt lassen sich innerhalb von JHSM also wiederum drei Schichten identifizieren: Ereignisschicht, Basiselemente und Kontextschicht. Diese Schichten werden in Abbildung 9.2 abstrakt dargestellt.

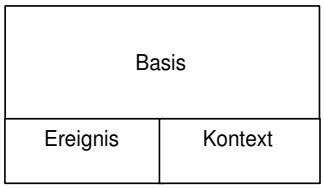

Abbildung 9.2: Die drei Schichten von JHSM.

Der Aufbau der einzelnen Schichten wird nun im Detail erläutert. Im Anschluss daran wird die Rolle des Interpreters sowie der Gesamtzusammenhang der drei Schichten dargestellt.

9.2.1 Die Ereignisschicht

Den Kern der Ereignisschicht stellt der `EventBus` dar. Dieser liefert die auftretende Ereignisse an den Zustand, in dem der Zustandsautomat sich gerade befindet. Diese Ereignisse können ihren Ursprung entweder im Automaten selbst oder außerhalb des Automaten oder Agenten haben. Der `EventBus` ist über den `Context` für jede Aktion verfügbar. Durch den Einsatz des `EventBus` werden die Elemente des Automaten beim Versenden und Empfangen von Ereignissen entkoppelt.

Abbildung 9.3 zeigt ein Klassendiagramm, dass die Schnittstellen der Ereignisschicht zu einander in Bezug setzt.

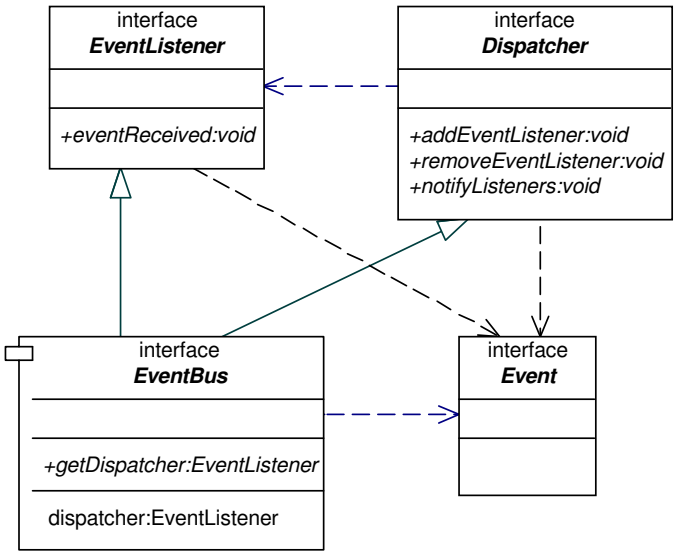

Abbildung 9.3: Die Schnittstellen der Ereignisschicht.

Die Beschreibung der Schnittstellen der Ereignisschicht:

Event: Diese Schnittstelle repräsentiert ein Ereignis. Sie dient nur zur Typisierung und definiert keinerlei Methoden oder Properties[3].

EventListener: Die Schnittstelle `EventListener` stellt den Aspekt des Empfangens von Ereignisse dar. Diese Ereignisse werden als Parameter an die Methode `eventReceived` übergeben. Die Schnittstelle wird neben dem `EventBus` auch von `Interpreter` erweitert.

Dispatcher: Die Schnittstelle `Dispatcher` stellt den Aspekt des Versendens von Ereignissen dar. Zu diesem Zweck verwaltet ein `Dispatcher` eine Liste von `EventListener` Objekten. Über die Methoden `addEventListener` und `removeEventListener` lassen sich diese hinzufügen und entfernen. Die Methode `notifyListeners` nimmt ein `Event` als Parameter entgegen und sendet dies an alle registrierten `EventListener`.

EventBus: Der EventBus erweitert die beiden Schnittstellen `EventListener` und `Dispatcher`. Über die Methode `getDispatcher` lässt sich ein geeigneter `EventListener` erhalten, der gegebenenfalls ein Ereignis an den aktuellen Zustand des Automaten (weiter) sendet.

Die Ereignisschicht enthält das Entwurfsmuster *Observer* [21, S. 293-303]. Dabei wird das *Subject* durch `Dispatcher` dargestellt und der *Observer* durch `EventListener`.

9.2.2 Die Basiselemente

Die hierarchischen Beziehungen innerhalb eines Zustandsautomaten lassen sich als *Teil-von-Relationen* mit Hilfe des *Composite* Entwurfsmusters [21, S. 163-173] darstellen. Abbildung 9.4 veranschaulicht dies. Der oberste zusammengesetzte Zustand der Hierarchie stellt den gesamten Zustandsautomaten dar. Dieser selbst wird nicht interpretiert sondern nur die in ihm enthaltenen Teilzustände.

Die Schnittstelle `CompositeState` (zusammengesetzter Zustand) stellt Methoden zur Verfügung, mit denen sich eine solche Zustandshierarchie aufbauen und verwalten lässt.

Um die gemeinsame Funktionalität der Basiselemente `State` und `Transition` zu kapseln steht eine weitere Schnittstelle `Component` zur Verfügung. Ein solche `Component` kann als strukturtragende Komponente eines Zustandsautomaten verstanden werden.

Die Schnittstellen `CompositeState`, `Component` und `State` bilden gemeinsam das Entwurfsmuster *Composite*. Die Rollenverteilung dabei ist recht offensichtlich: Es wird *Composite* durch `CompsiteState`, *Leaf* durch `State` und *Component* durch `Component` dargestellt. Die Nachkommen eines `CompsiteState` sind jedoch vom Typ `State` und nicht `Component`; dies stellt eine kleine Variation des Musters dar.

[3] Gemeint sind hier *Java Bean Properties*.

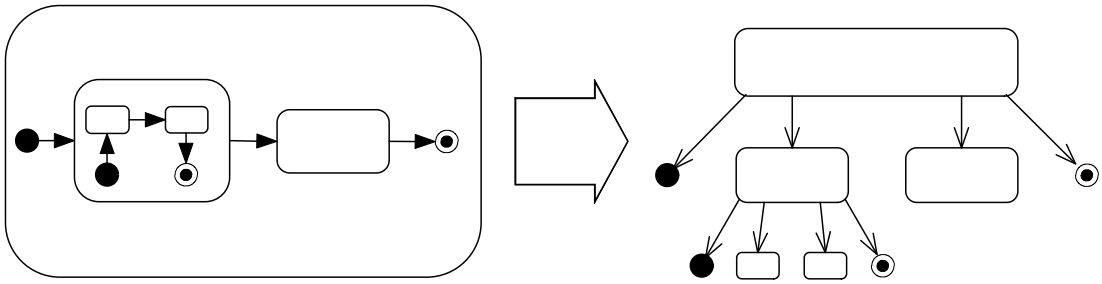

Abbildung 9.4: Die Anwendung des Entwurfsmusters *Composite*.

Die Schnittstellen der Basiselemente sind in Abbildung 9.5 in einem Klassendiagramm zu sehen. Um das Zusammenspiel der einzelnen Elemente näher zu erläutern werden nun die Schnittstellen vorgestellt:

Component: Kapselt die gemeinsame Funktionalität der Basiselemente `State` und `Transition`. `Component` kann als strukturtragende Komponente eines Zustandsautomaten verstanden werden. Zur Definition der Hierarchie steht die *Bean Property*[4] `parent` zur Verfügung. Durch `parent` wird der Vorgänger in der Hierarchie, also das übergeordnete Element, referenziert.

Da Aktionen sowohl auf Zuständen als auch auf Transitionen definiert werden können, wird eine `action` Property vom Typ `Action` zur Verfügung gestellt. Die mit der Property assoziierte Aktion kann über die Methode `action` ausgeführt werden. Als Parameter erwartet die Methode den `Context` des Zustandsautomaten.

Durch die `name` Property lässt sich ein Name der Komponente spezifizieren, dieser hat jedoch keine Bedeutung bei der Ausführung des Automaten, sondern dient nur zum Debugging.

State: Repräsentiert einen Zustand. Die Schnittstelle `Component` wird durch `State` erweitert. Die Eintritts- und Austrittsaktionen eines Zustands werden auf die Properties `entry` und `exit` abgebildet.

Jeder Zustand verwaltet eine Menge von Transitonen, für welche er jeweils den Quellzustand darstellt; die Transitionen gehen also von diesem Zustand aus zu einem anderen, dem Zielzustand, über. Diese Menge von Transitionen wird auf die Property `transitions` abgebildet. Über die Methoden `addTransition` und `removeTransition` lassen sich Transitionen zu `transitions` hinzufügen oder entfernen.

Ein Zustand verwaltet weiterhin eine Menge von verzögerten Ereignissen, welche nicht fallen gelassen werden, obwohl der Zustand keine Transitionen definiert, die durch diese

[4] Ein Attribut einer Klasse, das über *getter* und *setter* Methoden gelesen und gesetzt werden kann. Im Fall von `parent` also durch `getParent` und `setParent`. Wird eine *Property* auf einem Interface definiert, so muss die implementierende Klasse für die Bereitstellung des Attributes sorgen.

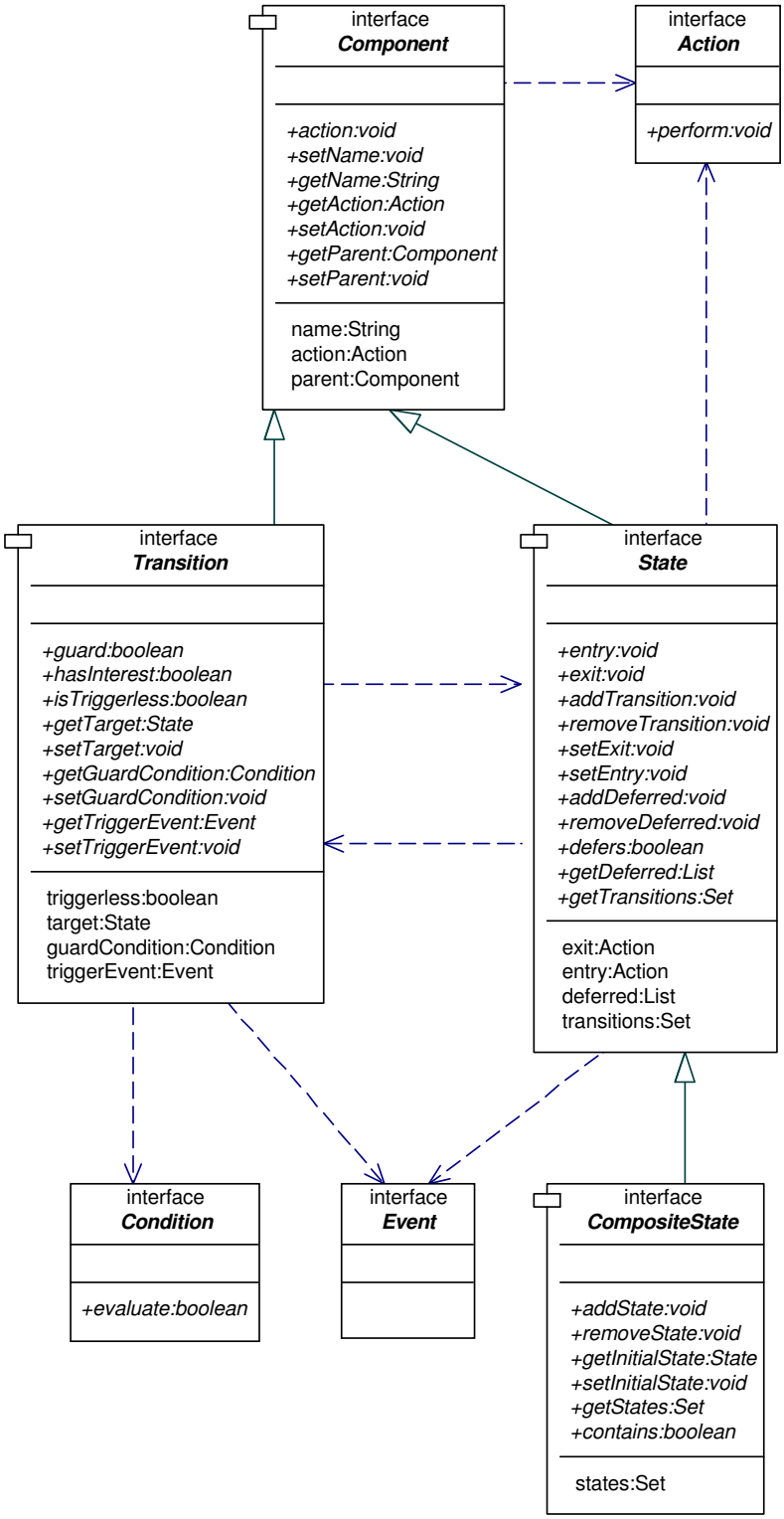

Abbildung 9.5: Die Schnittstellen der Basiselemente.

Ereignisse ausgelöst werden. Die Menge dieser Ereignisse wird auf die Property `deferred` abgebildet. Über die Methode `defers` lässt sich herausfinden ob der Zustand ein gegebenes Ereignis verzögert. Als Parameter der Methode wird ein `Event` erwartet, der Rückgabewert ist ein `boolean`.

CompositeState: Ein zusammengesetzter Zustand. Kann einen einzelnen Zustand oder einen ganzen Zustandsautomaten darstellen. `CompositeState` erweitert `State`. Ein `CompositeState` verfügt über eine Reihe von `State` Objekten: Dies sind die Teilzustände, welche als dessen Nachkommen in der Zustandshierarchie zu betrachten sind. Die Teilzustände werden auf die Property `state` abgebildet. Mit den Methoden `addState` und `removeState` lassen sich Zustände zu `states` hinzufügen und entfernen. Durch die Methode `contains` lässt sich ermitteln, ob ein gegebener Zustand ein Teilzustand des `CompositState` ist. Als Parameter der Methode wird ein `State` erwartet, der Rückgabewert ist ein `boolean`.

Der Startzustand des Automaten wird auf die Property `initialState` abgebildet.

Transition: Repräsentiert eine Transition. Die Schnittstelle `Component` wird durch `Transition` erweitert. Eine Transition kennt nur ihren Ziel- nicht den Quellzustand. Der Zielzustand wird auf die Property `target` abgebildet.

Die Wachbedingung der Transition wird auf die Property `guardCondition` und das auslösende Ereignis auf die Property `triggerEvent` abgebildet. Die Methode `guard` gibt an ob die Wachbedingung erfüllt ist oder nicht. Als Parameter wird der `Context` erwartet, der Rückgabewert ist ein `boolean`. Die Methode `hasInterest` gibt an ob ein gegebenes Ereignis als `triggerEvent` definiert wurde. Der Parameter ist ein `Event`, der Rückgabewert ein `boolean`. Wie bereits erwähnt, dient die Klasse `Event` ausschließlich zur Typisierung von Ereignissen. Für den Vergleich zweier Ereignisse ist daher in diesem Fall ein Vergleich der Klassen beider Ereignisse durch `equals` ausreichend. Die Methode `isTriggerless` liefert einen `boolean` zurück, der angibt ob ein auslösendes Ereignis auf der Transition definiert wurde oder nicht.

Action: Repräsentiert eine Aktion. Dies ist der Träger der Funktionalität, die in einem Zustand oder beim Wechsel eines Zustandes ausgeführt werden soll. Der Code der durch die Aktion ausgeführt werden soll wird in der Methode `perform` untergebracht. Der Context des Zustandsautomaten wird als Paramter von `perform` erwartet.

Condition: Repräsentiert eine Bedingung. Die Methode `evaluate` wertet die Bedingung aus und gibt einen `boolean` zurück.

9.2.3 Die Kontextschicht

Über den Kontext lassen sich zum Einen Daten teilen, zum Anderen lässt sich darüber das aktuelle Ereignis (im Fall des Auslösens einer Transition) und die Ereignisschicht (`EventBus`) re-

ferenzieren. Die Kontextschicht wird ausschließlich durch die Schnittstelle `Context` gebildet. Zusammen mit der Datenstruktur und dem Interpreter bildet der Kontext das Entwurfsmuster *Interpreter* [21, S. 243-255].

`Context` wird unmittelbar nur in `Action` und `Condition` verwendet. Abbildung 9.6 illustriert das Umfeld von `Context` in einem Klassendiagramm.

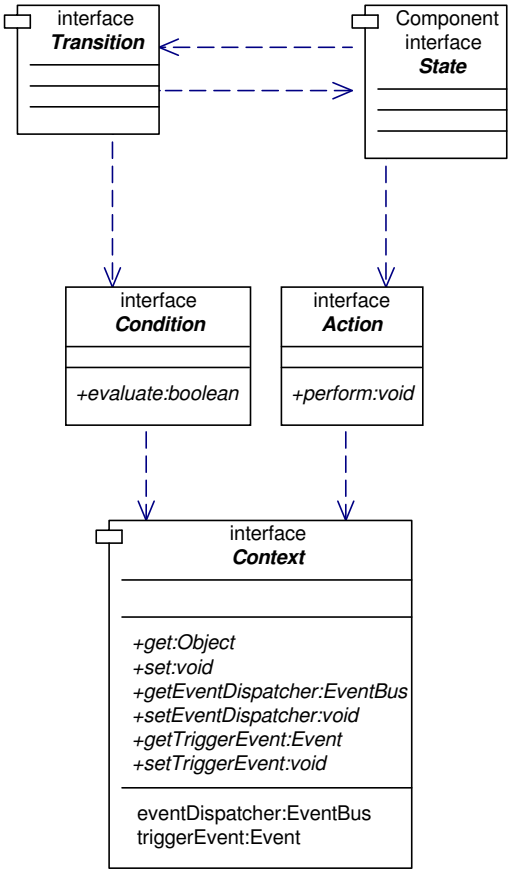

Abbildung 9.6: Die Schnittstellen der Kontextschicht.

Die Beschreibung der Schnittstelle:

Context: Repräsentiert den Kontext eines Zustandsautomaten. Ein `Context` arbeitet ähnlich wie ein Verzeichnis oder ein Wörterbuch. Über die Methoden `get` und `set` lassen sich beliebige Objekte unter bestimmten Namen im Kontext ablegen und auch wieder abrufen. Die Namen werden dabei durch `String` Objekte repräsentiert. Die `get` Methode erwartet eine `String` als Parameter und liefert ein `Object` zurück, die `set` Methode erwartet einen `String` und ein `Object` als Parameter.

Über die Property `eventDispatcher` lässt sich auf eine Referenz des `EventBus` erhalten und setzten.

Durch die Property `triggerEvent` lässt sich die Referenz des Ereignisses, das momentan verarbeitet wird abrufen. Falls kein Ereignis für das Auslösen der Transition benötigt wurde, wird diese Property auf `null` gesetzt.

9.2.4 Der Interpreter

Aufgabe des Interpreters ist es, den Kontrollfluss eines gegebenen Zustandsautomaten zu steuern. Der Interpreter bestimmt den aktiven Zustand des Automaten, nimmt Ereignisse entgegen und leitet diese an den aktiven Zustand weiter, wertet Bedingungen aus, entscheidet wann eine Transition ausgelöst wird und führt gegebenenfalls Aktionen aus.

Der `Interpreter` bildet mit `State`, `CompositeState` und `Context` das Entwurfsmuster *Interpreter* [21, S. 243-255]. Abbildung 9.7 stellt dies im Zusammenhang dar.

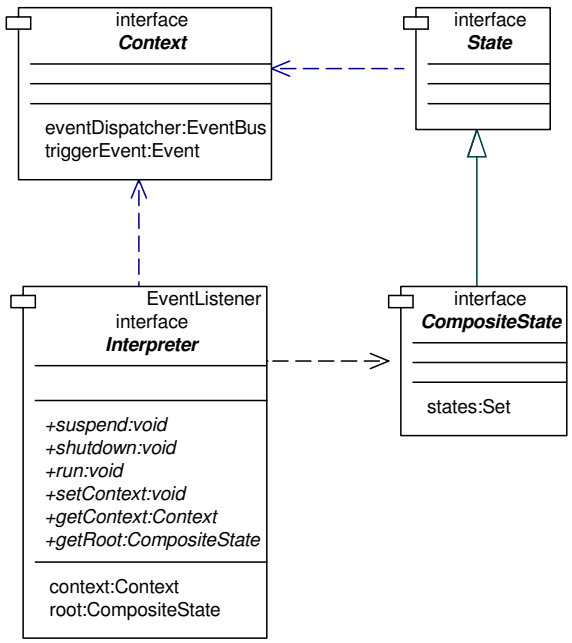

Abbildung 9.7: Der Interpreter und dessen Umfeld.

Die Beschreibung der Schnittstelle:

Interpreter: Ein Interpreter für Zustandsautomaten. Der `Interpreter` lässt sich starten durch das Ausführen der `run` Methode, `suspend` hält die Ausführung des `Interpreters` an und kann mit erneutem Aufruf von `run` wieder fortgesetzt werden. Der Aufruf von `shutdown` stoppt und beendet den Interpreter endgültig.

Die Property `root` stellt den Zustandsautomaten dar, der interpretiert wird. Dies ist der oberste zusammengesetzte Zustand der Zustandshierarchie.

Über die Property `context` lässt sich der `Context` des Zustandsautomanten festlegen, der interpretiert wird.

9.2.5 Das Gesamtbild

Setzt man die Elemente der drei Schichten zu einander in Bezug, so ergibt sich ein Bild, dass in Abbildung 9.8 dargestellt ist. Die Struktur des abgebildeten `CompositeState` ist mit der aus Abbildung 9.4 identisch. Die Attribute der Zustände und Transitionen sind jeweils exemplarisch hinzugefügt worden. Start- und Endzustände sind nicht dargestellt. Man beachte, dass der Interpreter hier nicht aufgeführt ist. Dieser bildet gewissermaßen eine Hülle, welche die drei dargestellten Schichten enthält.

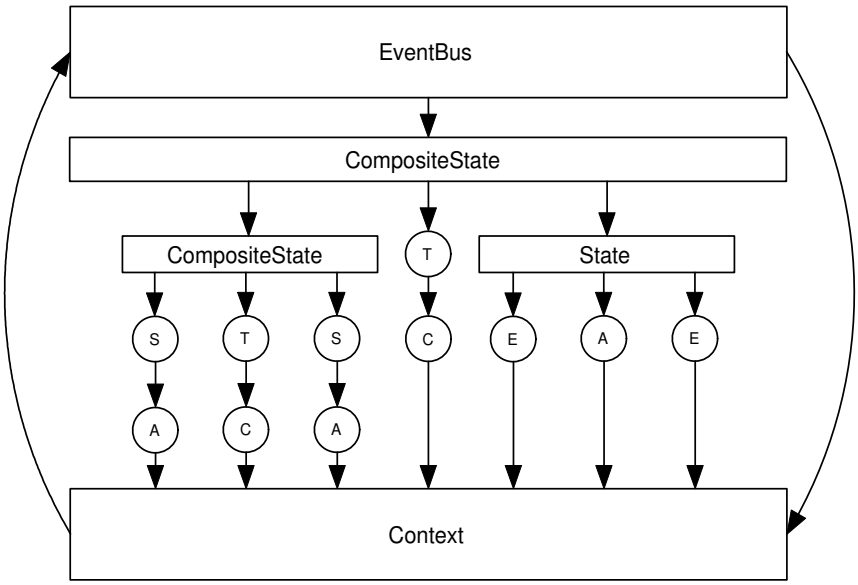

Abbildung 9.8: Die Elemente von JHSM in hierachischem Zusammenhang. Dabei steht *S* für State, *T* für Transition, *C* für Condition, *A* für Action und *E* für entry beziehungsweise exit.

Um den Kontrollfluss zu verdeutlichen der bei der Interpretation eines Zustandsautomaten entsteht, wird hierzu ein Sequenzdiagramm in Abbildung 9.9 gegeben. Die Darstellung ist sehr abstrakt und spielt sich in einem simplen, idealisierten Szenario ab. Betrachtet wird nur die Interpretation des Startzustands eines Zustandsautomaten, sowie das auslösen einer einzigen Transition, die zu einem unbekannten Zustand führt und auf der kein auslösendes Ereignis definiert ist. Nach dem Erreichen des unbekannten Zustands wiederholen sich im Prinzip die dargestellten Schritte. Die Interaktion ist auf der Ebene von Klassen veranschaulicht, das heißt, dass für die drei unterschiedlichen Aktionen keine eigene Repräsentation vorhanden ist.

1: Der Interpreter beginnt die Interpretation des Zustandsautomaten, der durch einen `CompositeState` gegeben ist, indem er den Startzustand des Automaten abfragt. Dieser

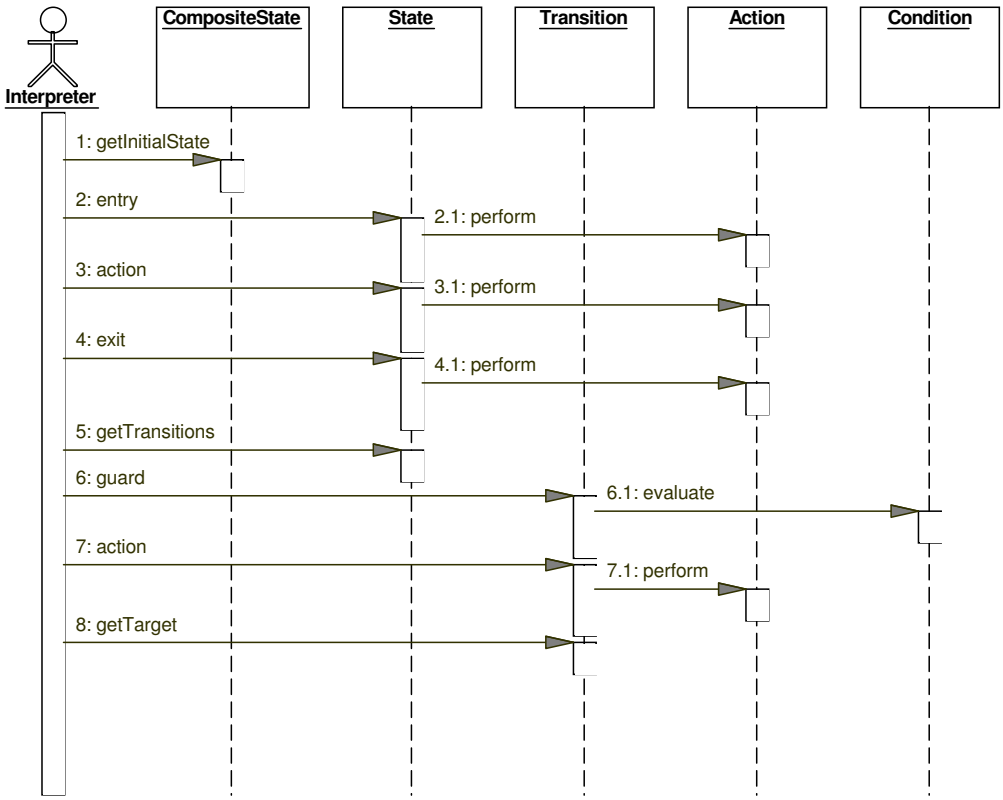

Abbildung 9.9: Der schematische Kontrollfluss der Interpretation.

Zustand ist nun der aktive Zustand des Automaten.

2-4: Auf dem aktiven Zustand werden nacheinander die `entry`, `exit` und `action` Methoden aufgerufen. Diese führen jeweils auf der mit der jeweiligen Handlung assoziierten `Action` die Methode `perform` aus.

5: Der Interpreter betrachtet alle ausgehenden Transitionen, die er über `getTransitions` vom aktiven Zustand erhält (in diesem Fall nur eine).

6: Auf der Transition wird die Wachbedingung durch den Aufruf von `guard` überprüft. Diese ruft wiederum `evaluate` auf der entsprechenden `Condition` auf: Es wird angenommen, dass diese erfüllt ist.

7: Die Transition wird ausgelöst, daher wird die `action` Methode ausgeführt.

8: Der Interpreter ermittelt den Zielzustand der Transition durch die Methode `getTraget` und die Schritte 2-7 wiederholen sich.

Alle Schnittstellen des Frameworks sind noch einmal zusammen in Abbildung 9.10 dargestellt. Da die Schnittstellen im einzelnen schon vorgestellt wurden enthält die Abbildung nur noch die Zusammenhänge zwischen den Elementen.

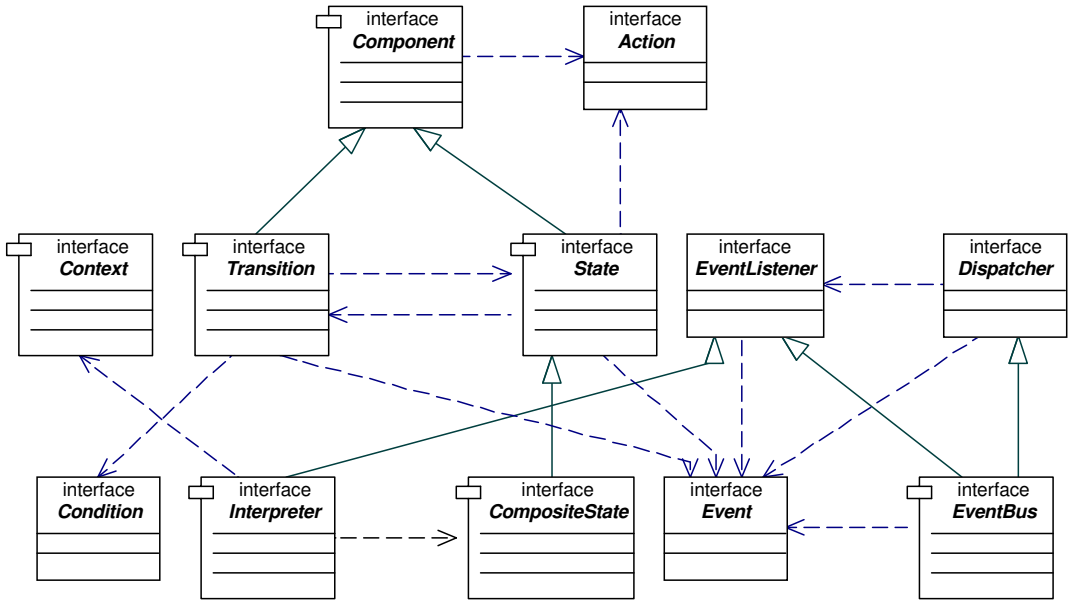

Abbildung 9.10: Alle Schnittstellen von JHSM im Zusammenhang.

JHSM wurde ganz bewusst sehr unabhängig von einem bestimmten Anwendungskontext gestaltet. Es ist sowohl unabhängig von SeMoA als auch von mobilen Agenten oder Agenten im Allgemeinen. JHSM sollte sich daher in allen Fällen anwenden lassen, in denen sich das komplexe, dynamische Verhalten eines beliebigen Objekts gut durch Zustandsautomaten beschreiben lässt.

9.3 Eine Architektur für die Modellierung von Agenten

Die Aufgabe des AMoA Frameworks ist es, eine Modellierung des Verhaltens von mobilen Agenten zu ermöglichen, so wie diese in Kapitel 7 vorgestellt wurde. Dafür ist es nötig das Framework JHSM zu erweitern, damit auch der Aspekt der Mobilität direkt in der Implementierung abgebildet werden kann (vgl. Kapitel 8).

Zu diesem Zweck wird die `MigrationTransition` als Basiselement eines Zustandsautomaten eingeführt. Diese stellt neben dem Wechsel eines Zustands auch einen Wechsel des Ausführungsortes des Agenten dar (vgl. Abschnitt 8.3.1). Die Klasse `MigrationTransition` implementiert `Transition`, definiert jedoch keine eigenen Methoden oder Properties, sondern dient nur zur Identifikation des syntaktischen Elements der Migrations-Transition.

Um die erweiterten Zustandsautomaten interpretieren zu können wird eine Erweiterung des Interpreters aus JHSM benötigt. Diese wird durch die Klasse `AMoAInterpreter` zur Verfügung gestellt. Dieser Interpreter implementiert die Schnittstelle `Interpreter`. Durch dessen Verwendung wird es ermöglicht eine Migrations-Transition in einem Zustandsautomaten zu verwenden, so dass diese korrekt interpretiert wird.

AMoA wurde so gestaltet, dass es unabhängig von SeMoA verwendet werden kann. Es ist also als ein allgemeines Framework für die Modellierung des Verhaltens von mobilen Agenten anzusehen, das sich auch im Kontext anderer Plattformen anwenden lässt. Mit AMoA wurde ein Grundstein für ein solches Framework gelegt, auf dem in Zukunft noch aufgebaut werden kann. Einige mögliche Erweiterungen werden in Kapitel 13 vorgeschlagen.

Um AMoA in einer beliebigen Plattform nutzen zu können, muss eine Integration des Frameworks in das jeweilige Agenten-Konzept der Plattform stattfinden. Im Rahmen des Ausführungsmodells eines Agenten muss ein Interpreter mit einem gegebenen Zustandsautomaten initialisiert und anschließend gestartet werden. Vor einer Migration muss die Interpretation gestoppt werden, so dass der aktuelle Ausführungszustand des Automaten serialisiert werden kann. Nach der Migration muss die Ausführung des Interpreters fortgesetzt werden.

9.4 Anbindung an SeMoA

Um das Framework AMoA komfortabel in SeMoA nutzen zu können, steht eine spezielle Umsetzung dafür innerhalb von SeMoA bereit. Diese beinhaltet:

- Zwei Basisklassen für mobile Agenten
- Eine Anpassung an die Ereigniskommunikation von SeMoA
- Pakete zum Sammeln von generischen und wiederverwendbaren Aktionen und Ereignissen

Die einzelnen Punkte werden in den folgenden Abschnitten besprochen.

9.4.1 Basisklassen

Es werden zwei Basisklassen zur Verfügung gestellt: `AbstractAgent` und `HSMAgent`. Während erstere eine ganz allgemeine Basisklasse darstellt, welche nur die Grundprinzipien eines SeMoA Agenten implementiert, ist letztere eine spezielle Agenten-Klassen, die für die Verwendung mit AMoA entworfen wurde. Es folgt die Beschreibung der Klassen:

AbstractAgent: Die abstrakte Klasse `AbstractAgent` implementiert die Java Schnittstellen `java.lang.Runnable`, `java.io.Serializable` und `DE.FhG.IGD.semoa.agent.Resumable`. Die Bedeutung der ersten beiden Schnittstellen wurde bereits in Abschnitt 2.6.2 beschrieben, letztere wird von dem *Lifecycle-Konzept* von SeMoA verwendet. Agenten, die `Resumable` implementieren, werden über Fehler bei der Migration oder Fehler bei dem Anhalten des Agenten informiert. In einem solchen Fall wird der Agent neu aufgesetzt und die Methode `resume` wird mit einem entsprechenden Fehler-Code als Parameter aufgerufen. Der Agent ist dann in der Lage, anhand des Fehler-Codes zu entscheiden wie er reagiert.

HSMAgent: Die Klasse `HSMAgent` erweitert `AbstractAgent` und stellt eine Verbindung zu AMoA beziehungsweise JHSM her. Die Klasse implementiert die Schnittstelle `Context` und stellt somit den Kontext eines Zustandsautomaten dar. Folglich stellt sie die Properties `eventDispatcher` und `triggerEvent` zur Verfügung. Durch die Methode `setBehavior` lässt sich das Verhalten des Agenten festlegen. Als Parameter wird ein `CompositeState` erwartet, welcher als Zustandsautomat interpretiert wird. Für die Interpretation stellt `HSMAgent` einen internen `Interpreter` zur Verfügung. Die Initialisierung und Verbindung von Kontext, Zustandsautomat, EventBus und Interpreter findet automatisch statt. Beim Aufruf von `setBehavior` werden die entsprechenden Objekte für den Interpreter, den EventBus und den Kontext instanziert und untereinander bekannt gemacht (referenziert). Der Agent kann anschließend durch den Aufruf der Methode `run` gestartet werden.

9.4.2 Kommunikation

Um Systemereignisse von SeMoA für den *HSMAgent* nutzbar zu machen, wurde ein Adapter entwickelt, der das Konzept des *EventReflectors* auf AMoA überträgt. Dieser `EventReflectorAdapter` wird beim EventRelector von SeMoA registriert und nimmt Nachrichten von diesem entgegen. Diese Nachrichten werden als `SemoaEvent` gekapselt und dem EventBus des Zustandsautomaten übergeben. Es kommt dabei das Entwurfsmuster `Adapter` zum Einsatz [21, S. 139-150].

Es folgt eine Beschreibung der Schnittstellen dieser beiden Klassen:

EventReflectorAdapter: Die Klasse `EventReflectorAdapter` implementiert die Schnittstelle `DE.FhG.IGD.util.Listener`. Diese wird vom EventReflector bei der Registrierung erwartet. Die Methode `notifiedOf` nimmt Nachrichten in Form eines `Object` entgegen, das als ein `SemoaEvent` an den Zustandsautomaten weitergeleitet wird.

SemoaEvent: Eine Wrapper-Klasse für Nachrichten des EventReflectors. Im Konstruktor der Klasse wird ein Object übergeben, welches über die Methode `getReflectedEvent` wieder abgerufen werden kann.

9.4.3 Sammelpakete

Um zu vermeiden, dass bestimmte Elemente eines Zustandsautomaten, die im Kontext von SeMoA häufig auftreten immer wieder neu implementiert werden müssen, stehen Pakete bereit in denen solche Elemente abgelegt werden können. Momentan stehen beispielsweise Ereignisse bereit, die eine Migration des Agenten und das Terminieren des Agenten veranlassen sollen. Weiterhin ist eine Aktion vorhanden, die vor dem Terminieren eines Agenten aufgerufen werden sollte um sicherzustellen, dass der Agent korrekt vom SeMoA-System beendet wird.[5]

9.4.4 Gesamtbild

Abbildung 9.11 stellt AMoA im Kontext von SeMoA durch ein Komponentendiagramm dar. Dargestellt ist ein Agent innerhalb des SeMoA-Systems. Die beiden SeMoA-Dienste, die einem Agenten mindestens bekannt sein müssen um seine Umgebung zu beeinflussen und mit anderen Agenten oder Diensten in Kontakt zu treten sind das *Environment* und der *EventReflector*.

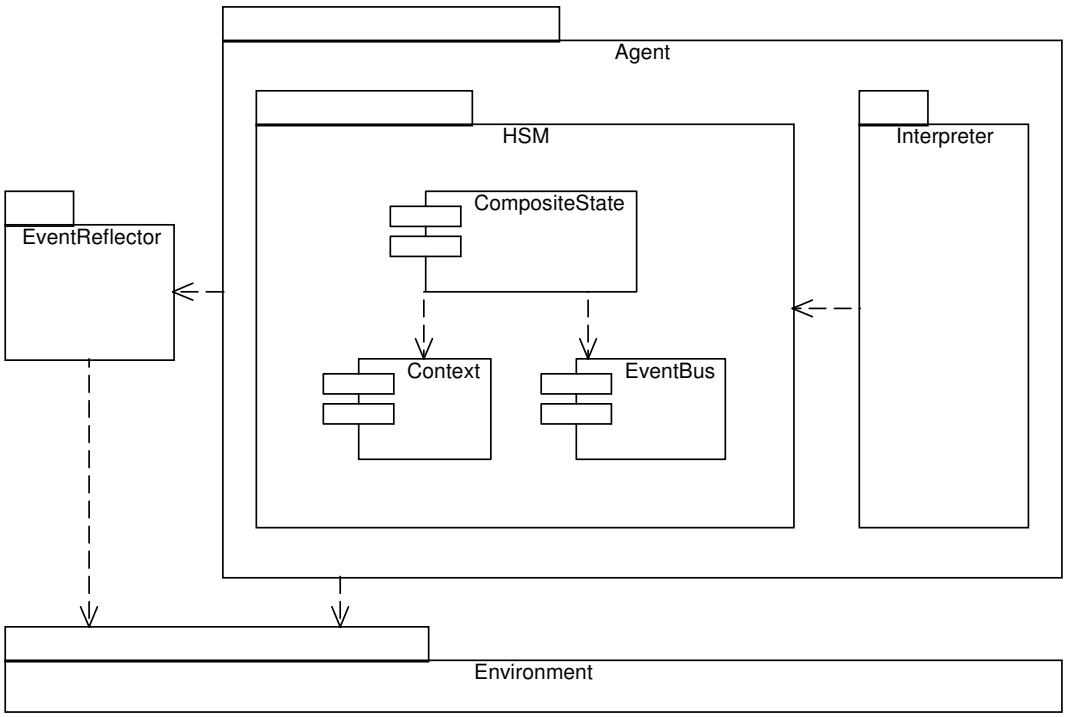

Abbildung 9.11: Die Komponeten von AMoA im Kontext SeMoAs.

Wie man sehen kann, kapselt der Agent den hierarchischen Zustandsautomaten, der sein Verhalten bestimmt, und den Interpreter, der für die Ausführung des Automaten verantwortlich ist.

[5] Es ist dafür nötig das *ticket* des Agenten auf `null` zu setzten.

Die Komponente *HSM* ist nicht als Softwarekomponente vorhanden sondern grenzt nur innerhalb der Abbildung der Teil des Automaten gegenüber dem *Interpreter* ab. Der Zustandsautomat selbst wird durch den *CompositeState* repräsentiert. Der *CompositeState* kennt nur seinen *Context* und den *EventBus*. Man kann leicht erkennen, dass die Kopplung [43] zwischen den Komponenten minimal und ausschließlich von einseitiger Art ist.

Kapitel 10

Implementierung

> *The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.* — Kent Beck

In diesem Kapitel finden sich die wichtigsten Details der Implementierung des in Kapitel 9 vorgestellten Entwurfs. Zunächst wird das allgemeine Vorgehen bei der Programmierung in Java knapp dargestellt. Anschließend wird eine Reihe von Hilfsmitteln vorgestellt, welche für die erfolgreiche Umsetzung des Entwurfs eigens erstellt werden mussten. Abschließend wird die Integration der fertigen Lösung in SeMoA erläutert.

10.1 Das Vorgehen bei der Implementierung

Der Entwurf des Lösungsansatzes wurde in der Programmiersprache Java unter der Verwendung des JDK 1.3 von *Sun Microsystems* implementiert. Als Entwicklungsumgebung wurde die *NetBeans IDE*[1] verwendet. Für Automatisierung einiger Entwicklungsvorgänge wurde *ANT*[2] eingesetzt.

Der Gesamte Entwicklungsprozess wurde test-getrieben (*test-driven*) gestaltet. Das heißt, es wurden zuerst Testfälle erdacht und implementiert und erst danach wurde die Programmierung des Frameworks weiter vorangetrieben. Dieser Ablauf wird häufig mit „*code a little, test a little*“ bezeichnet. Als Hilfmittel bei diesem Vorgehen wurde das Framework *JUnit*[3] genutzt. Zunächst wurde der grobe Rahmen des Frameworks entwickelt. Dazu gehören die Implementierungen der Schnittstellen der Basiselemente und eine Rohfassung des Interpreters, der über eine minimale Funktionalität verfügt.

[1] http://www.netbeans.org
[2] http://ant.apache.org
[3] http://www.junit.org

Das Testen gestaltet sich in diesem Kontext folgendermaßen. Aus den Basiselementen werden mehrere konkrete Zustandsautomaten konstruiert. Ein Testfall wird aus dem Automaten und einem Test-Treiber gebildet, der den Automaten von dem Interpreter ausführen lässt und gegebenenfalls den Ablauf der Interpretierung durch das Erzeugen von Ereignissen steuert. Jeder Interpretationsschritt wird automatisch protokolliert und nach dem Ende des Test mit einem Protokoll verglichen, dass die erwarteten Schritte des Test enthält. Sind diese beiden Protokolle identisch, so ist der Test positiv.

Falls ein Test negativ ausfällt, hat der Interpreter nicht korrekt gearbeitet und muss geändert werden. Nach jeder dieser Änderungen muss jedoch gewährleistet sein, dass auch die bereits absolvierten Tests noch bestanden werden. Durch die vollständige Automatisierung des Testvorgangs kann dies in wenigen Sekunden überprüft werden. Auf diese Weise wurde das Framework schrittweise dem Lösungsansatz angenähert.

10.2 Die Hilfsmittel

Um das Testen in der beschriebenen Weise durchzuführen, wird ein Test-Werkzeug benötigt, das die Protokollierung und automatische Prüfung der Interpretationsschritte ermöglicht. Zu diesem Zweck wurde die Klasse `TestProtocol` entwickelt. Diese Klasse bietet eine sehr simple Schnittstelle an, über die sich einzelne Schritte in Form von Strings eintragen lassen. Diese werden in einer internen Liste gespeichert. Im Konstruktor der Klasse wird ein Protokoll übergeben, das die erwarteten Schritte enthält. Über die Methode `evaluate` lässt sich überprüfen ob die interne Liste mit den erwarteten Schritten übereinstimmt. Die Methode liefert entsprechend `true` oder `false` zurück.

Über den EventBus eines Zustandsautomaten können jederzeit Ereignisse eintreffen. Die Lieferung der Ereignisse kann durch mehrere parallele Threads gleichzeitig geschehen. Für die Synchronisation dieser Threads mit dem Thread des Agenten wurde eine Hilfsklasse entwickelt, welche die sonst übliche Kombination aus `synchronized` Blöcken und `wait-notify` Mechanismen in sich kapselt.[4] Mit Hilfe des `BooleanLock` ist es möglich den Zugriff auf einen Block Code zu steuern. Die wird erreicht durch das kapseln einer boolschen Variable, die auf sichere Weise von mehreren Threads manipuliert werden kann. Beliebige Threads können gleichzeitig den Wert der Variable setzten oder auslesen und auf eine Änderung der Variable warten. Intern wird der `wait-notify` Mechanismus verwendet um die Warte-Funktion zu realisieren.

Da Agenten und all ihre Attribute vollständig serialisierbar sein müssen um migrieren zu können, bereitet die Anwendung von *Logging* einige Probleme. Da keine Logging-API bekannt ist, deren Klassen serialisierbar sind[5], musste eine eigene API für diesen Zweck entwickelt werden.

[4] Dies wurde inspiriert von [31, S. 407-427].

[5] Dies trifft auch auf die Logging-API von SeMoA zu.

Diese API ist minimal jedoch erweiterbar gehalten. Durch die Implementierung des Entwurfsmusters *Abstract Factory* ist es möglich beliebige Logger-Typen zu werden.

10.3 Die Integration in SeMoA

Da die Implementierung des Entwurfs als Sub-Modul von SeMoA umgesetzt wurde, ist keine Installation nötig. Es müssen sich lediglich die drei Pakete des Frameworks an den richtigen stellen innerhalb des Verzeichnisbaumes von SeMoA befinden. Der SeMoA-Kern selbst ist unter `DE.FhG.IGD.semoa` zu finden. Da AMoA und JHSM unabhängig von SeMoA sind, werden diese unter `DE.FhG.IGD.amoa` beziehungsweise `DE.FhG.IGD.jhsm` abgelegt, Die Anbindung von SeMoA an AMoA befindet sich in `DE.FhG.IGD.semoa.amoa`.

Teil III

Resultate

Kapitel 11

Zusammenfassung

Was nicht auf einer einzigen Manuskriptseite zusammengefaßt werden kann, ist weder durchdacht noch entscheidungsreif. — Dwight David Eisenhower

In dieser Arbeit wird ein Konzept vorgestellt, durch dessen Verwendung sich mobile Agenten und deren Verhalten in einer wiederverwendbaren Art und Weise entwerfen lassen. Die Zielsetzung der Arbeit wird durch einen zustandsorientierten Ansatz erfüllt, der speziell auf mobile Agenten zugeschnitten ist. Das Konzept setzt sich zusammen aus einer zustands-orientierten Entwurfsmethode und einem komponenten-orientierten Framework für die Implementierung der entworfenen Modelle.

Die zentrale Anforderung, welche an das Konzept gestellt wird, ist die Wiederverwendbarkeit der Resultate, die durch die Anwendung des Konzepts entstehen. Dies wird durch eine geeignete Interpretation des Konzeptes des Zustandsautomaten umgesetzt. Der Lösungsansatz wird durch ein spezielles Java-Framework im Rahmen von SeMoA nutzbar gemacht.

Es wird eine Einführung in die Grundlagen gegeben, die zum Verständnis dieser Arbeit benötigt werden. Dazu gehört ein Überblick über die wichtigsten Themen der agenten-orientierten Computerwissenschaft, sowie eine knappe Einführung in die UML beziehungsweise die Zustandsdiagramme der UML. Es werden auch einige bestehende Ansätze zur Modellierung und Implementierung von (mobilen) Agenten vorgestellt und untersucht. Die Konzeption und die Umsetzung des durch diese Arbeit vorgestellten Ansatzes werden ausführlich dargestellt. Dabei werden zunächst die Anforderungen an das zu entwickelnde Konzept formuliert und begründet, gefolgt von einer Analyse der Anforderungen bezüglich der zustands-orientierten Modellierung. Aus dieser Analyse wird schließlich der Lösungsansatz abgeleitet. Der Ansatz wird zunächst auf konzeptioneller Ebene beschrieben und anschließend in Form einer technischen Beschreibung der für die Implementierung relevanten Teile entworfen. Abschließend wird die konkrete Umsetzung des Konzepts für SeMoA vorgestellt.

Kapitel 12

Bewertung

Bescheidenheit ist die ungesündeste Form der Selbstbewertung. — Sir Peter Ustinov

Das in dieser Arbeit vorgestellte Konzept zur Modellierung und Implementierung von mobilen Agenten und deren Verhalten stellt eine vielseitig einsetzbare und erweiterbare Lösung dar. Der Grund für die Wahl der Zustandsautomaten als Modellierungsgrundlage liegt zum einen in der leichten Erlernbarkeit und in der Popularität von UML. Zum Anderen wird durch die Modellierung mit Zustandsautomaten deren natürliche Unterstützung der *Modularität* ausgenutzt. Wie in der vorangegangen Analyse gezeigt wurde, hat zustandsorientierte Modellierung eine inhärente positive Auswirkung auf viele der modularen Kriterien. Dies gilt insbesondere für modulare *Dekomponierbarkeit*, *Komponierbarkeit* und *Verständlichkeit*. Weiterhin eignet sie sich gut, um das Verhalten eines mobilen Agenten abzubilden, so dass dieser die agententypischen Charakteristika aufweist. Besonders offensichtlich ist dies im Falle der Eigenschaften *Reaktivität*, *Kommunikationsfähigkeit* und *Mobilität* zu erkennen. Wie sich herausgestellt hat, konnten diese günstigen Eigenschaften der zustandsorientierte Modellierung durch eine Anpassung der Elemente der Modellierung, sowie eine optimale Unterstützung des Gesamtkonzeptes durch eine geeignete Software-Architektur noch verstärkt werden. Besonders hervorzuheben ist die Integration des Konzeptes der Mobilität in die Zustandsdiagramme, durch das die Verständlichkeit des Verhaltensmodells stark begünstigt wird.

Da der Aspekt der Hierarchie in der gewählten Modellierungssprache stark ausgeprägt ist, lässt sich das Abstraktionsniveau der Modellebene fast beliebig wählen. Dies hat zur Folge, dass das entwickeltet Konzept sowohl eigenständig einsetzbar ist, aber auch in Verbindung mit anderen Entwurfsmethoden verwendet werden kann. Durch diese Flexibilität des Lösungsansatzes ist eine vielseitige und praxisnahe Anwendung möglich.

Die zustandsorientierte Modellierung löst jedoch nicht alle Probleme eines Entwicklers für Agenten vollständig. Wie die Dekomposition eines Verhaltens in Zustände genau zu geschehen

hat, ist von Fall zu Fall verschieden und muss je nach Anwendungsziel individuell vom Entwickler vorgenommen werden. Dies lässt sich vergleichen mit der Bildung von konkreten Mengen von Objekten und deren Schnittstellen in der objekt-orientierten Programmierung. Weiterhin ist auch bei der Implementierung der Funktionstragenden Elemente (*Aktion* und *Bedingung*) eines Zustandsautomaten noch immer die übliche Problemstellung der objekt-orientierten Programmierung zu bewältigen.

Es lässt sich feststellen, dass die zustandsorientierte Modellierung eine sehr gute Möglichkeit darstellt, das Verhalten von mobilen Agenten abzubilden. Durch die gute Unterstützung der Modularität wird die Wiederverwendbarkeit der Resultate, sowohl der Implementierung als auch des Modells, gefördert.

Das im Rahmen des Konzeptes entwickelte Framework ermöglicht die direkte Abbildung eines zustandsorientierten Modells auf wiederverwendbare Software-Komponenten. Durch die Verwendung des *Bean-Konzeptes* wird eine gute Verständlichkeit und Anwendbarkeit des Komponenten-Modells erreicht. Die Aufteilung in drei unabhängige Schichten sorgt dafür, dass sich nicht nur die mit Hilfe des Frameworks erstellten Resultate wiederverwenden lassen, sondern auch Teile des Frameworks selbst. So ist *JHSM* als ein generisches Framework für die zustandsorientierte Programmierung in Java zu betrachten. Da Modell und Ausführung durch die implementierung des Entwurfsmusters *Interpreter* sauber getrennt wurden, ist eine Erweiterung und eine Anwendung des Frameworks in vielen unterschiedlichen Kontexten möglich. Das Framework *AMoA* lässt sich als ein auf mobile Agenten bezogenen Ansatz für die Modellierung des Verhaltens betrachten, der an keine spezielle Plattform gebunden ist und daher auch in verschiedenen Bereichen wiederverwendet werden kann.

Die Anbindung des Framworks an SeMoA wurde so realisiert, dass diese leicht benutzbar ist. Durch die Bindung der Konzepte des Frameworks an eine *Basisklasse* wird das Verständnis des Lösungsansatzes und das Konzept des mobilen Agenten in SeMoA stark gefördert.

Gegenüber verwandten Arbeiten auf dem Gebiet der Verhaltensmodellierung für (mobile) Software-Agenten lässt sich hervorheben, dass dieser Ansatz besonderen Wert auf die Modularität der Modell- und Software-Elemente legt, welche durch dessen Anwendung entstehen. Zu diesem Zweck wurde die Syntax der Zustandsdiagramme der UML angepasst. Als wichtigste Modifikationen lassen sich das Verbot von *grenzüberschreitenden Transitionen*, die Integration der Mobilität und die Entfernung von *nebenläufigen Teilzuständen* nennen. Die Entfernung der nebenläufigen Teilzustände hat außerdem zur Folge, dass die Modellierung des Verhaltens sich stärker in die Richtung von reaktiven, ereignis-orientierten Modellen entwickelt. Dies hat insbesondere den Vorteil, dass die üblichen Probleme der nebenläufigen Programmierung nicht mehr auftreten.

Sicherlich gibt es noch viele Möglichkeiten, den gegeben Stand der Arbeit zu erweitern. Nahe liegend wäre beispielsweise die Einführung einer Metasprache oder eines Datenformates für die unabhängige Definition der Zustandsautomaten. Darauf aufbauend wäre auch ein Werkzeug

für die graphische Erstellung und Manipulation der Automaten sehr wünschenswert. Diese und andere weiterführende Ideen werden in Kapitel 13 vorgestellt.

Insgesamt stellt das in dieser Arbeit entwickelte Konzept eine vielseitig einsetzbare und erweiterbare Grundlage für die Entwicklung von mobilen Agenten dar, die in dieser Form bisher nicht existiert.

Kapitel 13

Ausblick

Das Ende eines Dinges ist der Anfang eines anderen. — Leonardo da Vinci

In diesem Ausblick werden einige Ideen welche die zukünftige Entwicklung und Anwendung des in dieser Arbeit vorgestellten Konzeptes betreffen diskutiert.

Zunächst einmal wäre es wichtig das vorgestellte Konzept durch den Einsatz in einem realen Projekt zu evaluieren. Dies ist zum einen als ein erweiterter Test zu verstehen, der die Anwendbarkeit des Konzeptes und insbesondere des entwickelten Frameworks untermauert. Nebenbei könnte jedoch während der Evaluation auch einen ausreichende Menge an wiederverwendbaren Komponenten erstellt werden, die für das Umfeld der SeMoA-Plattform häufig benötigt werden. Diese Komponenten sollten dann die Realisierung von zukünftigen Projekten erleichtern. Ideal wäre ein Anwendungsfall, in dem auch der Einsatz einer Methodologie wie *Gaia* oder *MaSE* sinnvoll ist. Das Konzept könnte dann sowohl eigenständig, als auch im Verbund mit einer globalen Entwurfsmethode evaluiert werden. Nach einer solchen Evaluierung werden sich sicherlich einige Detail des Konzeptes und dessen Implementierung ändern, es ist jedoch zu erwarten, dass der Lösungsansatz insgesamt gestärkt aus dieser Untersuchung hervorgeht und zukünftig ein noch größeres Potential bezüglich seiner Anwendbarkeit darstellt.

Interessant wäre es, die Anbindung des Frameworks an eine andere Agenten-Plattform wie beispielsweise *JADE* oder *Tracy* durchzuführen. Neben einer Demonstration der Machbarkeit würde dies unter Umständen auch zur Verbreitung und Anwendung des Konzeptes unter den Benutzern dieser Plattformen führen. Der Aufwand der Anbindung sollte aufgrund der Struktur des Frameworks sehr gering ausfallen. Den anspruchsvollsten Teil einer solchen Unternehmung stellt sicherlich die Implementierung von anschaulichen Beispielen dar.

Ein Aspekt des Konzeptes, der in Zukunft näher untersucht werden sollte, ist die automatischen Verifikation von zustandsorientierten Modellen. Der Entwicklungsprozess würde durch die Verwendung von automatischer Verifikation verkürzt werden und die resultierende Software wäre

sicherlich auch robuster. Eine weiterführende Lektüre zu diesem Thema stellen die Arbeiten von *Simons* [66] und von *Bianco et al.* [3] dar. In diesem Rahmen, aber auch im allgemeinen, wäre eine formale Definition der Syntax und Semantik der verwendeten Zustandsdiagramme nötig. Mit Hilfe einer Formalisierung der Zustandsdaigramme könnten außerdem automatisch Simulationen und Testfälle erzeugt werden. Dadurch wäre eine starke Unterstützung für die Qualitätssicherung in der Agenten-Entwicklung gegeben.

Zu den wünschenswerten Erweiterungen des Konzepts gehört auch die Definition einer Meta-Sprache für Zustandsautomaten, sowie die Modellierung von Agenten durch ein graphisches Werkzeug. Als Vorbild dient in dieser Hinsicht *XABSL* wegen der fruchtbaren Verwendung von XML für die Definition und Visualisierung der Automaten. Die Arbeiten *ADK* und das Projekt *HSMBehaviour* sind gute Beispiele für die erfolgreiche Implementierung eines graphischen Werkzeugs. Die Vorteile dieser Ansätze wurden im einzelnen schon in Kapitel 5 dargestellt und sollen daher an dieser Stelle nicht wiederholt werden.

Ein weiterer spannender Aspekt ist die konzept-unabhängige Anwendung von JHSM zur allgemeinen zustandsorientierten Programmierung in Java. Diese Anwendung von JHSM wurde bereits angeregt und es wäre sicherlich interessant in naher Zukunft zu überprüfen ob nicht auch andere Software-Komponenten, die keine Agenten verkörpern, auf der Grundlage von hierarchischen Zustandsautomaten implementiert worden sind. In [64] findet sich eine Diskussion der zustandsorientierten Programmierung unter praktischen Gesichtspunkten.

Nach Meinung des Autors steht einer fruchtbaren Verwendung und Erweiterung des vorgestellten Konzeptes nichts im Wege.

Teil IV

Anhang

Anhang A

Glossar

Debugging — Ein aus dem Englischen stammender Begriff und bedeutet wörtlich übersetzt soviel wie „entwanzen“ (von bug: Wanze). Gemeint ist das Auffinden, Diagnostizieren und Eliminieren von Fehlern in Hardware und vor allem von Programmfehlern in Software.

Framework — Ein Begriff aus der objekt-orientierten Programmierung. Im Gegensatz zur Programmbibliothek besteht ein Framework zusätzlich aus einem Hauptprogramm, das die globale Steuerung übernimmt. Wörtlich übersetzt bedeutet Framework (Programm-) Gerüst, -Rahmen oder -Skelett. Dies drückt aus, dass die grobe Architektur bereits vorgegeben ist, und dass nur noch an ganz bestimmten Stellen applikationsspezifischer Code „eingehängt“ wird. Die eigentliche Applikation umfasst also kein Hauptprogramm mehr sondern wird von Framework-Komponenten aus aufgerufen.

Java — Eine objekt-orientierte Programmiersprache von SUN Microsystems. Java-Anwendungen laufen ohne jegliche Änderung auf den meisten Plattformen, wie z. B. auf einem PC mit Windows98/NT oder OS/2, auf einem Mac unter MacOS, oder auf einer SunSPARC-Workstation unter Linux.

Methodologie — (deutsch auch: Methodenlehre) Die Lehre von den Methoden, den wissenschaftlichen Verfahren. Als solche ist Methodologie „Meta-Wissenschaft“ und somit Teildisziplin der Wissenschaftstheorie (Epistemologie). Im Englischen und Französischen ist die Unterscheidung „Methodologie“ versus „Methodik“ jedoch unbekannt. Der Einfluss des Amerikanischen trägt zu einem unpräzisen Sprachgebrauch im Deutschen bei: Unter „Methodologie“ wird auch eine Sammlung von Methoden verstanden, ein Methodenbündel. Im Rahmen dieser Arbeit ist der unpräzise Begriff gemeint. Methodologie bezeichnet hier also eine Methode beziehungsweise eine Sammlung von Methoden; oder genauer gesagt: eine Entwurfsmethode für Software-Systeme.

Middleware — Bezeichnet in der Informatik anwendungs-unabhängige Technologien, die Dienstleistungen zur Vermittlung zwischen Anwendungen anbieten, so dass die Komplexität

der zu Grunde liegenden Applikationen und Infrastruktur verborgen wird.

Migration — (von lat.: migratio: Wanderung, Auszug) Bezeichnet in der Regel einen Wechsel des Ortes. Im Kontext mobiler Agenten ist mit Migration der Wechsel des Ausführungssystems des Agenten gemeint.

Open Source — Steht für quelloffen, einerseits in dem Sinne, dass der Quelltext eines Programms frei erhältlich ist, andererseits für „offene Quelle“, also dass ein Werk frei zur Verfügung steht. Software gilt als Open Source, wenn sie bestimmte Kriterien erfüllt, die in ihrer Open-Source-Lizenz geregelt sind.

Overhead — Als Overhead werden in der Datenübertragung Daten bezeichnet, die nicht primär zu den Nutzdaten zählen, sondern als Hilfsdaten zur Übermittlung oder Speicherung benötigt werden. Zum Overhead zählt somit beispielsweise ein vom Empfänger zum Sender zurückgeschickter Überprüfungscode um die Korrektheit der übertragenen Daten sicherzustellen. Im Kontext einer Methode kann der Begriff auch mit „Mehraufwand“ übersetzt werden.

Reverse Engineering — Bezeichnet den Vorgang, aus einem bestehenden, fertigen System durch Untersuchung der Strukturen, Zustände und Verhaltensweisen die Konstruktionselemente zu extrahieren. In der Informatik ist damit oft die Rückgewinnung des Quellcodes oder einer vergleichbaren Beschreibung aus Binärcode, beispielsweise von einem ausführbaren Programm oder einer Programmbibliothek gemeint.

Prädikaten-Logik — Die wichtigsten Teilgebiete der elementaren formalen Logik sind die klassische Aussagen- und Prädikatenlogik. Während in der Aussagenlogik Aussagen (d.h. wahrheitsfähige Sätze) nicht weiter analysiert werden und nur die verschiedenen Junktoren, die Aussagen miteinander verknüpfen, relevant sind, beruht die Prädikatenlogik auf einer genaueren Unterscheidung zwischen verschiedenen Ausdruckskategorien wie Termen, Funktoren, Prädikatoren und Quantoren.

Scheduler — (engl.: Disponent, Planer) Ein elementarer Teil von Betriebssystemen, die Multitasking unterstützen. Werden auf einem Computer mehrere Prozesse gleichzeitig ausgeführt, so muss das Betriebssystem die vorhandenen Ressourcen auf die verschiedenen Prozesse aufteilen. Diese Aufteilung sollte so optimal wie möglich erfolgen, wobei allerdings unterschiedliche Ziele verfolgt werden können: Auslastung der Betriebsmittel, Schnelle Antwortzeiten, Durchsatz.

SeMoA — Eine Plattform für mobile Agenten, welche am „Fraunhofer Institut für Graphische Datenverarbeitung“ in der Abteilung „Sicherheitstechnologie für Graphik und Kommunikationssysteme“ entwickelt wird. Neben der Bereitstellung der Infrastruktur für mobile Agenten, konzentriert sich SeMoA vor allem auf den Aspekt der Sicherheit, sowie auf die Interoperabilität mit anderen Agenten- und Komponenten-Standards.

Shell — Eine Software, welche eine Zeile Text in der Kommandozeile einliest, diesen Text als Kommando interpretiert und ausführt, beispielsweise durch das Starten weiterer Programme.

Anhang B

Akronyme

ADK	Agent Development Kit
AMoA	Agent Modelling Architecture
Aglet	Agile Applet (Agent)
AUML	Agent Unified Modelling Language
JADE	Java Agent Development Framework
JDK	Java Development Kit
JHSM	Java Hierarchical State Machines
MaSE	Multiagent Systems Engineering
SeMoA	Secure Mobile Agents
UML	Unified Modelling Language
XABSL	Extensible Agent Behavior Specification Language
XML	Extensible Markup Language

Literaturverzeichnis

[1] Toshiaki Arai and Frieder Stolzenburg. Multiagent systems specification by UML statecharts aiming at intelligent manufacturing. Technical Report 12-2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.

[2] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - a FIPA compliant agent framework. In *4th International Conference on Practical Applications of Intelligent Agents and Multi-Agent Systems*, pages 97–108, London, April 1999.

[3] V. D. Bianco, L. Lavazza, and M. Mauri. A formalization of UML statecharts for real-time software modeling. In H. Ehrig, B. J. Krämer, and A. Ertas, editors, *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*. Society of Design and Process Science, 2002. www.sdpsnet.org.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings Pub. Co., 1994.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley Longman Inc., 1st edition, October 20 1998. ISBN 0-201-57168-4.

[6] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. In *Computational Intelligence*, volume 4, pages 349–355, 1988.

[7] Peter Braun. Über die Migration bei mobilen Agenten. Jenaer Schriften zur Mathematik und Informatik Math/Inf/99/13, Friedrich-Schiller-Universität Jena, Institut für Informatik, April 1999.

[8] T. Bray, J. Paoli, C.M. Sperberger-McQueen, and E. Maler. W3C recommendation: Extensible markup language (XML) 1.0 (second edition). http://www.w3.org/TR/REC-xml, 2000.

[9] R. A. Brooks. Elephants don't play chess. In P. Maes, editor, *Designing Autonomous Agents*, pages 3–15, Cambridge, MA, 1990. The MIT Press.

[10] Tobias Butte. Base technologies for the development of agent-based distributed applications. http://www.informatik.tu-darmstadt.de/BS/Lehre/Sem01/proceedings/, 2001. ISVIS 2001, TU Darmstadt.

[11] T. De Wolf and T. Holvoet. Adaptive behaviour based on evolving thresholds with feedback. In *Proceedings of the AISB'03 Third Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS)*, pages 91–96, 2003.

[12] D.C. Dennet. *The Intentional Stance*. The MIT Press: Cambridge, 1987.

[13] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[14] Doron Drusinsky and David Harel. On the power of bounded concurrency I: finite automata. *J. ACM*, 41(3):517–539, 1994.

[15] O. Etzioni and D. Weld. Intelligent agents on the internet: Fact, fiction and forecast. *IEEE Expert*, 10(4):44–49, 1995.

[16] D.C. Fallside. W3C recommendation: XML schema part 0: Primer. http://www.w3.org/TR/xmlschema-0/, 2001.

[17] S. Fonseca, M. Griss, and R. Letsinger. Agent behaviour architectures - a MAS framework comparison. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, International Conference on Autonomous Agents, pages 86 – 87, Bologna, Italy, 2002. ACM Press (New York, NY, USA). ISBN:1-58113-480-0.

[18] Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. http://www.fipa.org/, 2002. Document number: SC00061G.

[19] Martin Fowler. *UML Distilled. Applying the Standard Object Modeling Language*. Addison Wesley Longman, 1997.

[20] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley Pub. Co., 1995. ISBN 0-201-63361-2.

[22] E.B. Goldstein. *Sensation and perception, (5th edition)*. Belmont, CA: Wadsworth, 1999.

[23] Martin Griss, Steven Fonseca, Dick Cowan, and Robert Kessler. Smartagent: Extending the JADE agent behavior model HPL 2002-18. In *AOSE workshop*, Orlando Florida, 2002. SCI.

[24] Martin Griss, Steven Fonseca, Dick Cowan, and Robert Kessler. Using UML state machines models for more precise and flexible JADE agent behaviors HPL 2002-298(R). In *AAMAS AOSE workshop*, Bologna, Italy, July 2002.

[25] Martin Griss, Robert Kessler, Brian Remick, and Ryan Delucchi. SCATE: Hierachical state machine (HSM) for JADE. http://www.cse.ucsc.edu/research/agents/hsm/, April 2004. Website.

[26] T. Gschwind, M. Feridun, and S. Pleisch. ADK - building mobile agents for network and systems management from reusable components. In *Proceedings 1st International Symposium on Agent Systems and Applications and 3rd International Symposium on Mobile Agents (ASA/MA '99)*, pages 13–21, Los Alamitos, 1999. IEEE Computer Society Press.

[27] G. Hamilton. Java Beans. Sun Microsystems, http://java.sun.com/beans, July 1997.

[28] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw Hill, 1998.

[29] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[30] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile agents: Are they a good idea? Technical report, T. J. Watson Research Center, Yorktown Heights, New York, 1995.

[31] P. Hyde. *Java Thread Programming*. Sams Publishing, 1999.

[32] Carlos Iglesias, Mercedes Garrijo, and José Gonzalez. A survey of agent-oriented methodologies. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.

[33] Ivar Jacobson, Grady Booch, and James Rambaugh. *The Objectory Software Developement Process*. Addison Wesley Longman, 1998.

[34] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 177(2):277–296, 2000.

[35] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Comms. of the ACM*, 44(4):35–41, 2001.

[36] M. van Lent and J. Laird. Developing an artificial intelligence engine. In *Proceedings of the Game Developers' Conference*, pages 577–588, San Jose, CA, 1999.

[37] Y. Lesperance, H.J. Levesque, F. Lin, D. Marcu, R. Reiter, and R.B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J.P. Müller, and M. Tambe, editors, *Intelligent Agents H*, volume 1037 of *LNAI*, pages 331–346. Springer-Verlag: Berlin, Germany, 1996.

[38] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence, Padova, Italy, 2004. Springer. to appear.

[39] M. Luck and M. d'Inverno. Autonomy: A nice idea in theory. In *Intelligent Agents VII: Proceedings of the Seventh International Workshop on Agent Theories, Architectures and Languages*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2001.

[40] Michael Luck and Mark d'Inverno. A conceptual framework for agent definition and development. *The Computer Journal*, 44(1):1–20, 2001.

[41] D. Marr. *Vision*. San Francisco: W. H. Freeman, 1982.

[42] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[43] B. Meyer. *Object-Oriented Software Construction, second edition*. Prentice Hall, 1997.

[44] R. Montanari, G. Tonti, and C. Stefanelli. A policy-based mobile agent infrastructure. In *Proceedings of the 3rd IEEE International Symposium on Applications and the Internet*. IEEE Computer Society Press, October 2003.

[45] Jan Murray. Specifying agents with UML in robotic soccer. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, International Conference on Autonomous Agents, pages 51 – 52, Bologna, Italy, 2002. ACM Press (New York, NY, USA). ISBN 1-58113-480-0.

[46] J. Von Neumann and O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton University Press: Princeton, 1944.

[47] Christoph Niederberger and Markus H. Gross. Towards a game agent. Technical Report 377, Institute of Visual Computing, ETH Zürich, 2002.

[48] J. Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, May-June 2002.

[49] J. Odell, H.V.D Parunak, and B. Bauer. Extending UML for agents. In *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, Austin, Texas, 2000.

[50] J. Odell, H.V.D Parunak, and B. Bauer. Representing agent interaction protocols in UML. *The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, pages 121–140, 2000.

[51] James Odell, H. Van Dyke Parunak, and Mitch Fleischer. The role of roles in designing effective agent organizations. In *Software Engineering for Large-Scale Multi-Agent Systems*, volume 2603 of *Lecture Notes on Computer Science*, pages 27–28. Springer, Berlin, 2003.

[52] James Odell, H. Van Dyke Parunak, Mitch Fleischer, and Sven Breuckner. Modeling agents and their environment. In *Agent-Oriented Software Engineering (AOSE) III*, volume 2585 of *Lecture Notes on Computer Science*, pages 16–31. Springer, Berlin, 2002.

[53] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[54] U.S. Patent and Trademark Office. System and method for distributed computation based upon the movement, execution and interaction of processes in a network, 1997. US patent no. 5603031.

[55] Gian Pietro Picco. Mobile agents: An introduction. *Journal of Microprocessors and Microsystems*, (25):65–74, 2001. Invited contribution to a special issue on mobile agents.

[56] Ulrich Pinsdorf. A formal approach for interoperability between mobile agent systems and component based architectures. In *Proceedings of 11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2004)*. IEEE Computer Society Press. Brno, Czech Republic, May 2004.

[57] Ulrich Pinsdorf and Volker Roth. Mobile agent interoperability patterns and practice. In *Proceedings of Ninth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2002)*, Computer Graphics Edition, pages 238–244, University of Lund, Lund, Sweden, April 2002. Institute of Electrical and Electronics Engineers, IEEE Computer Society Press. ISBN 0-7695-1549-5.

[58] T. Röfer, I. Dahm, U. Düffert, J. Hoffmann, M. Jüngel, M. Kallnik, M. Lötzsch, M. Risler, M. Stelzer, and J. Ziegler. German-team 2003. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence, Padova, Italy, 2004. Springer. to appear.

[59] Volker Roth. *The $SeMoA^{TM}$ Code Conventions*. Fraunhofer IGD, Darmstadt, Germany, November 2000. Version 0.1.

[60] Volker Roth and Mehrdad Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., März 2001. IEEE Computer Society.

[61] Volker Roth, Ulrich Pinsdorf, Jan Peters, and Peter Ebinger. SeMoA whitepaper. unpublished. Available at http://www.semoa.org/.

[62] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modelling Language Reference Manual*. Addison Wesley, 1999. ISBN 020130998X.

[63] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995. ISBN: 0-13-360124-2.

[64] Miro Samek and Paul Y. Montgomery. State-oriented programming. *Embedded Systems Programming*, pages 22–43, August 2000.

[65] Olivier Sigaud and Pierre Gérard. Being reactive by exchanging roles: An empirical study. In *Balancing Reactivity and Social Deliberation in Multi-Agent Systems, From RoboCup to Real-World Applications (selected papers from the ECAI 2000 Workshop and additional contributions)*, Lecture Notes In Computer Science, pages 150 – 174, London, UK, 2001. Springer-Verlag. ISBN:3-540-42327-3.

[66] Anthony J.H. Simons. On the compositional properties of UML statechart diagrams. *Electronic Workshops in Computing: Rigorous Object-Oriented Methods 2000*, 2000.

[67] J. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[68] S.P. Stich. *From Folk Psychology to Cognitive Science*. The MIT Press: Cambridge, 1983.

[69] Amund Tveit. A survey of agent-oriented software engineering. Proc. of the First NTNU CSGS Conference (http://www.amundt.org), May 2001.

[70] Unified modeling language (UML), version 1.4. Specification formal/20010967, Object Management Group, 2001. Available from URL `http://www.omg.org/`.

[71] W. van der Hoek and W. Wooldrige. Towards a logic of rational agency. *Logic Journal of the IGPL*, 11(2):133–157, 2003.

[72] Giovanni Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis, Politecnico Di Milano, 1997.

[73] Mark F. Wood and Scott A. DeLoach. An overview of the multiagent systems engineering methodology. In P. Ciancarini and M. Wooldrige, editors, *Proceedings of the First International Workshop on Agent-Oriented Software Engineering, June 2000*, volume 1957 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, January 2001.

[74] M. Wooldridge. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter Intelligent Agents, pages 27–73. MIT Press: Cambridge, 1999.

[75] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2), 1995.

[76] M. Wooldridge, N.R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *International Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.

[77] Michael Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons (Chichester, England), 2002. ISBN 0 47149691X.

[78] Michael Wooldridge and Nicholas R. Jennings. Pitfalls of agent-oriented development. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 385–391, New York, 9–13, 1998. ACM Press.