Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Diplomarbeit

Eine Simulationsunterstützung für Agentenplattformen

Hinnerk Gildhoff

1gildhof@informatik.uni-hamburg.de
h.gildhoff@web.de
Studiengang Informatik
Matr.-Nr. 5408508
Fachsemester 10

Erstgutachter: Prof. Dr. W. Lamersdorf
Zweitgutachter: Dr. D. Moldt

In seeking the unattainable, simplicity only gets in the way.
– *Alan Perlis*

Danksagung

Für die technische Unterstützung möchte ich mich bei Alexander Pokahr und Lars Braubach bedanken, die mir jederzeit mit Rat und Tat zur Seite gestanden haben. Für die seelische Unterstützung geht mein Dank an Stefanie Maack, die mehr als meine Freundin ist. Des Weiteren bedanke ich mich bei allen, die mir bei der sprachlichen Verbesserung der Arbeit geholfen haben, insbesondere bei meiner Schwester Maike Gildhoff.

Hinnerk Gildhoff

Inhaltsverzeichnis

1 Einleitung

In der heutigen Welt entstehen immer komplexere, verteilte Systeme, die mit traditionellen Ansätzen fortwährend schwieriger beherrschbar sind (vgl. [TvM07]). Des Weiteren steigt das Bedürfnis nach selbstorganisierenden Systemen, die robuster, flexibler und dennoch ausgesprochen dynamisch sind. Das Paradigma der Multiagentensysteme setzt genau an dieser Problematik an.

Die fundamentale Idee hinter Multiagentensystemen besteht im Grunde aus der Annahme, dass sich ein komplexes System aus weitgehend selbstständigen (intelligenten) Einheiten zusammensetzt, die stetig miteinander interagieren. Somit liegt ein sich ständig änderndes System vor. Jeder Agent übernimmt hierbei eine spezielle Rolle. Die Konzepte der Agententechnologie sollen helfen, neue komplexe Systeme zu entwickeln oder bestehende komplizierte Systeme besser beherrschbar zu machen.

Die Simulationstechnologie unterstützt die Analyse komplexer, dynamischer Systeme. Anhand von Experimenten an einem Modell können Aussagen über das Verhalten des Originalsystems getroffen und verschiedene Verhaltensmuster erkannt werden. Das Anwendungsgebiet der Simulationen ist ebenso wie bei der Agententechnologie sehr weitläufig und wird in den unterschiedlichsten Branchen angewendet. Es gibt viele verschiedene Gründe, um Simulationen anzuwenden. So besteht zum Beispiel die Möglichkeit, Hypothesen, die das Originalsystem betreffen, an einem Simulationsmodell zu überprüfen.

Die Gemeinde der Simulationstechnologie ist immer auf der Suche nach neuen Methoden, um die Klasse der modellierbaren Systeme zu erweitern (vgl. [Klü01]). Die Agententechnologie bietet hierbei eine neue Möglichkeit der Simulation, die sogenannte Multiagentensimulation. Diese wird hauptsächlich eingesetzt, um Hypothesen an einem Modell zu überprüfen. Die Agententechnologie kommt hierbei zum Einsatz, um zum Beispiel dem Modellierer ein intuitiveres Modellieren von Gesellschaften zu ermöglichen. Folglich muss die Simulation nicht ausschließlich auf Objekten basieren, sondern kann auch aus Agenten bestehen, die gerade durch ihre Autonomie und ihr selbständiges Handeln eine neue Art der Simulation ermöglichen. Insbesondere bei Simulationen, die sich aus einer Reihe von Individuen zusammensetzen, bietet sich die Implementierung von Agenten an, um das individuelle Verhalten auf einem abstrakten Niveau festzulegen, ohne es genau definieren zu müssen.

Die Multiagentensimulation zeichnet sich vor allem durch die Nutzung von Agenten für die Simulationstechnologie aus. In [Yil06] wird hierfür der Begriff agentenbasierte Simulation (Agent-based Simulation) oder agentenunterstützte Simulation (Agent-supported

Simulation) verwendet. Die agentenbasierte Simulation bezieht sich hierbei auf Simulationen, dessen Modellverhalten durch Agenten erzeugt wird. Im Gegensatz dazu werden bei der agentenunterstützenden Simulation die Agenten nur als Hilfsmittel eingesetzt, um zum Beispiel den Simulationsanwender bei der Analyse der Simulationsergebnisse zu unterstützen.

Allerdings zeigt nun auch die Gemeinde der Agentenentwickler Interesse an der Simulationstechnologie, da die diesbezüglichen Konzepte auch bei der agentenbasierten Softwareentwicklung eingesetzt werden können, was in [Yil06] auch als Agentensimulation (Agent Simulation) bezeichnet wird. Diese unterschiedlichen Interessen (siehe Abbildung 1.1) teilen den Bereich der Multiagentensimulation in zwei Teilbereiche ein, die jeweils von einer der beiden Technologien genutzt werden. Durch eine bessere Zusammenarbeit beider Fachdisziplinen erhoffen sich beide Seiten den Gewinn neuer Erkenntnisse.

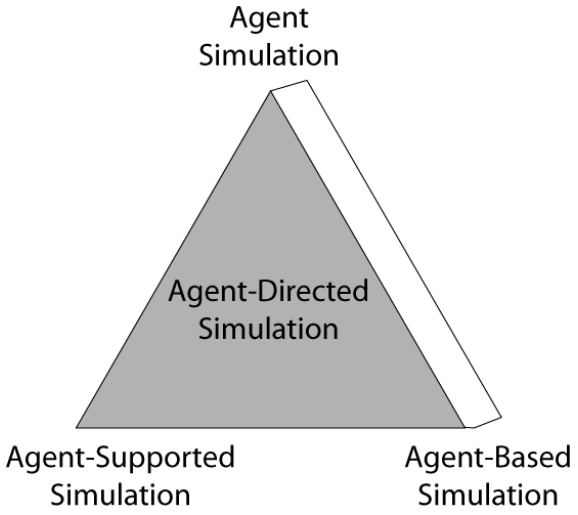

Abbildung 1.1: Agent-Directed Simulation (aus [Yil06])

Mit dieser Arbeit soll die Zusammenarbeit beider Technologien vorangetrieben werden, indem die Konzepte der Simulationstechnologie bei der agentenbasierten Softwareentwicklung angewendet (agent simulation) und die dadurch entstehenden Vorteile praktisch veranschaulicht werden. Anstatt domänenspezifische Simulationen zu entwerfen, die nicht als reale Softwarelösungen eingesetzt werden, wird die Simulationstechnologie bei dem Entwicklungsprozess eines Multiagentensystems als Hilfsmittel eingesetzt, um das Multiagentensystem besser verstehen und optimieren zu können. Das Multiagentensystem soll nach einer iterativen Entwicklungs- und Simulationsphase so getestet und optimiert sein, dass eine optimale Problembewältigung erreicht werden kann. Zudem können Simulationsabläufe dazu genutzt werden, die Vorteile einer agentenbasierten Lösung zu verdeutlichen und somit zu einer Verbreitung der Multiagententechnologie beitragen.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist der Entwurf einer Simulationsunterstützung für Agentenplattformen. Agentenplattformen stellen eine Infrastruktur bereit, auf der Multiagentensysteme ausgeführt werden können. Die Simulationsunterstützung soll die bestehende Infrastruktur erweitern, um Multiagentensysteme in verschiedenen Betriebsmodi ausführen zu können. Folglich kann das Design der Agenten und des gesamten Multiagentensystems besser verstanden, validiert und optimiert werden. Hierbei bleibt die Simulationsunterstützung aber nur ein Hilfsmittel, um den Entwicklungsprozess von praktisch anwendbaren Multiagentensystemen zu unterstützen.

Agentenbasierte Simulationsframeworks, wie zum Beispiel SeSAm [KHF+05] oder Swarm [Gro98], unterscheiden sich in diesem Punkt von den hier verfolgten Zielen. Sie stellen Werkzeuge (Frameworks) dar, deren Ziel die agentenbasierte Modellierung (ABM) und die agentenbasierte Simulation (ABS) ist. Andere agentenbasierte Simulationsframeworks haben sich insbesondere auf die agentenbasierte soziale Simulation (ABSS) spezialisiert, wie zum Beispiel Repast [Tea06] oder NetLogo [Net06]. Hierbei soll die agentenbasierte Simulationen von Gesellschaften, zur Erklärung von sozialen Phänomenen beitragen.

In dieser Arbeit geht es jedoch darum, den Entwurfsprozess von praktisch anwendbaren Multiagentensystemen durch die Simulationstechnologie zu unterstützen. Der Grund für die Erstellung des Multiagentensystems ist praktischer Natur und dieses endet nicht mit seiner Simulation und deren Erkenntnissen, sondern zielt darauf ab, die entworfene Software (ein Multiagentensystem) in der Praxis einzusetzen. Beim Entwurf der Simulationsunterstützung soll darüber hinaus darauf geachtet werden, dass die Agentenplattform alle nicht benötigten Simulationsmechanismen vor dem Anwender verbirgt, damit dieser im optimalen Fall ein Multiagentensystem wie zuvor entwerfen kann, ohne vor neue Anforderungen gestellt zu werden.

Des Weiteren wird innerhalb dieser Arbeit eine Simulationsunterstützung für eine konkrete Agentenplattform realisiert und die Funktionsweise anhand einer Beispielanwendung verdeutlicht. Nach der Realisierung wird eine Schlussbetrachtung durchgeführt und insbesondere darauf eingegangen, welche Vorteile sich durch die Simulationsunterstützung ergeben. Zudem soll geklärt werden, wie allgemeingültig die Simulationsunterstützung ist.

1.2 Struktur der Arbeit

Im zweiten Kapitel werden in erster Linie die benötigten Grundlagen vorgestellt. Hierzu gehört ein Grundverständnis von Agenten und Multiagentensystemen sowie eine allgemeine Einführung in den Bereich der Simulationen. Hierbei wird die Simulationstechno-

logie aufgrund ihrer großen Reichweite und ihrer unterschiedlichen Methoden auf einen Teilbereich eingegrenzt. Darüber hinaus wird ein kurzer Einblick in die Multiagentensimulation vermittelt, ein erster Ansatz, um die Erkenntnisse und Vorteile beider Technologien zu vereinen.

Im dritten Kapitel werden die Anforderungen an die Simulationsunterstützung definiert. Danach werden verschiedene Varianten vorgestellt, mit denen eine Simulationsunterstützung verwirklicht werden kann. Nachdem die unterschiedlichen Implementierungsmöglichkeiten gegenübergestellt wurden, wird der vielversprechenste Implementierungsansatz ausgewählt und weiterentwickelt. Anhand dessen wird ein allgemeingültiges Konzept für die Simulationsunterstützung von Agentenplattformen aufgestellt. Hierbei werden konkrete Komponenten genannt, die für eine Simulationsunterstützung zwingend erforderlich sind.

Im vierten Kapitel werden die zuvor genannten Konzepte verwirklicht. Zudem wird die Simulationsunterstützung am Beispiel von Jadex implementiert. Jadex ist eine konkrete Agentenplattform, die sich durch eine BDI-Architektur auszeichnet (siehe [PBL03]). Diese Agentenplattform wird mit der allgemeinen Simulationsarchitektur ausgestattet, wobei auch eine Schnittstelle für Erweiterungen bereitgestellt wird.

Die Anwendbarkeit der Simulationsunterstützung wird im fünften Kapitel durch ein konkretes Beispiel demonstriert. Das Anwendungsbeispiel ist ein agentenbasiertes Verkehrsleitsystem, welches nach der Entwicklung als reale Software eingesetzt werden soll. Anhand dieses Beispiels werden die Vorteile verdeutlicht, die durch die Simulationsunterstützung entstehen, und es wird aufgezeigt, welche Simulationsmodi sich für dieses Beispiel besonders eignen.

Im sechsten Kapitel werden die Resultate der Arbeit zusammengefasst. Abschließend erfolgt ein Ausblick über mögliche Erweiterungen der Simulationsinfrastruktur.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Themengebiete dieser Arbeit vorgestellt. Als erstes wird das Thema *Agententechnologie* genauer betrachtet, wobei unter anderem die Begriffe *Agent*, *Multiagentensystem* und *Agentenplattform* geklärt werden. Danach werden die Grundlagen der *Simulationstechnologie* vorgestellt und dabei wird das Gebiet der Simulation auf einen Teilbereich, der für diese Arbeit entscheidend ist, eingegrenzt.

Ein Grundverständnis beider Technologien ist essentiell für die Vereinigung von Simulations- und Agententechnologie, wie am Ende des Kapitels am Beispiel der *Multiagentensimulation* gezeigt wird.

2.1 Agententechnologie

Die Agententechnologie ist ein junges Forschungsgebiet, das sich bei dem Entwurf von komplexen Softwaresystemen bewiesen hat, da sie die Komplexität solcher Systeme bereits auf der Konzeptebene handhabbar macht (siehe [Jen01]). Die Komplexität muss beherrschbar und verwaltbar sein, damit diese Systeme in ihrem vollen Funktionsumfang genutzt werden können.

2.1.1 Agent

Im Lexikon [LI81] wird der Begriff *Agent* als "Vermittler von Geschäften für Dritte" beschrieben. Innerhalb der Agententechnologie wird der Begriff *Agent* als Synonym für *Softwareagent* gebraucht. Eine universale Definition konnte sich in dem jungen Forschungsgebiet noch nicht durchsetzen, allerdings wird die Definition von Wooldridge ([Woo02], S. 15) von der Gemeinde der Agentenforscher prinzipiell anerkannt, da sie sich auf das Wesentliche beschränkt:

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

In dieser Definition werden die zwei wichtigsten Eigenschaften eines Agenten deutlich. Als erstes gilt, dass ein Agent autonom handelt. Er entscheidet für sich selbst, welche Aktion er als nächstes ausführen möchte, um seine eigenen Ziele zu erreichen, ohne extern kontrolliert zu werden.

Des Weiteren existiert ein Agent in einer Umgebung und steht in einer ständigen Interaktion mit dieser. Eine Umgebung kann verschiedene Eigenschaften haben. Jeweils nach diesen Eigenschaften, werden von den Agenten gewisse Fähigkeiten verlangt, damit sie in der Umgebung adäquat agieren können. Dies erklärt auch, warum keine universelle

Definition für Agenten existiert. Umgebungen mit unterschiedlichen Eigenschaften führen zu Agenten mit verschiedenen Fähigkeiten. Rusell und Norvig klassifizieren Umgebungen durch vier Eigenschaften (nach [RN95]):

Accessible versus inaccessible. In einer zugänglichen Umgebung hat ein Agent den vollständigen, akkuraten Zugriff auf den aktuellen Zustand der Umgebung. Für die meisten Umgebungen trifft diese Behauptung allerdings nicht zu. Meistens hat der Agent nur einen eingeschränkten Zugriff auf die Umgebung. Umgebungen, die nicht zugänglich sind, kommen bei Multiagentensystemen kaum vor.

Deterministic versus non-deterministic. In einer deterministischen Umgebung hat jede Aktion genau einen garantierten Effekt. In einer nichtdeterministischen Umgebung kann eine Aktion zu anderen Ergebnissen führen und somit ist nicht sicher, welcher Zustand nach einer Aktion eintritt.

Static versus dynamic. Eine statische Umgebung lässt sich nur durch Aktionen von Agenten verändern. Im Gegensatz dazu kann eine dynamische Umgebung durch Prozesse beeinflusst oder verändert werden, die außerhalb der Kontrolle von Agenten liegen.

Discrete versus continuous. Eine Umgebung ist diskret, wenn es eine feste, endliche Anzahl von Aktionen und Wahrnehmungen gibt. Ansonsten ist die Umgebung kontinuierlich.

Anhand dieser Eigenschaften können sämtliche Umgebungen eines Multiagentensystems eingestuft werden. Eine Umgebung, die nicht zugänglich, nichtdeterministisch, dynamisch und kontinuierlich ist, gehört zu den komplexesten Umgebungen. Je nach Einstufung der Umgebung werden von den Agenten verschiedene Eigenschaften gefordert, damit sie in der kategorisierten Umgebung nicht scheitern. Padgham und Winikoff ([PW04], S.3) nennen sieben grundlegende Eigenschaften für Agenten. Neben den bereits erwähnten Eigenschaften der Autonomie eines Agenten und das sich ein Agent immer in einer Umgebung befindet (situated), zeichnet sich dieser durch folgende, weitere Eigenschaften aus:

Robust: Agenten werden meistens in Umgebungen eingesetzt, die nicht vollständig zugänglich sind. In diesen Umgebungen kann sich der Agent nie sicher sein, ob der gewünschte Effekt einer Aktion eintritt. Er muss mit einem Misserfolg sowie einem unerwarteten Effekt umgehen können und sich aus einem Fehlerzustand selber wieder befreien können.

Flexibel: Ein Agent hat eine Fülle von Aktionen parat, mit denen er flexibel auf die Umgebung reagieren kann.

Reaktiv: Ein Agent muss auf signifikante Änderungen in der Umgebung rechtzeitig reagieren können und seine Aktionen und Ziele ständig der sich ändernden Umgebung anpassen.

Proaktiv: Agenten arbeiten zielorientiert, indem sie die Initiative übernehmen, um ihre eigenen Ziele zu erreichen.

Sozial: Agenten sind fähig mit anderen Agenten (oder Menschen) zu interagieren, um ihre eigenen Ziele zu erreichen.

Die Qualität (oder auch Intelligenz) eines Agenten wird vor allem durch die Mischung aus reaktivem und proaktivem Verhalten bestimmt (nach [Woo02]). Ein Agent soll systematisch probieren, seine Ziele zu erreichen. Allerdings darf er dabei nicht nur einseitig seine Ziele verfolgen, sondern muss auch ständig seine Umgebung beobachten, um auf Änderungen in dieser Umgebung reagieren zu können. Falls er aber zu oft reagiert und sein Ziel nicht lange genug verfolgt, wird er dieses auch nicht erreichen. Die Mischung aus reaktivem und proaktivem Handeln muss also genaustens abgewogen werden.

2.1.2 Multiagentensystem

Das Besondere an der Agententechnologie wird nicht nur durch die Eigenschaften und die Architektur eines Agenten bestimmt, sondern vor allem durch den Entwurf eines Systems, in dem sich mehrere Agenten befinden, die miteinander kommunizieren und interagieren können. Solch ein System nennt man Multiagentensystem (MAS). Wooldridge beschreibt ein Multiagentensystem wie folgt ([Woo02], S.3):

> A multiagent system is one that consists of a number of agents, which *interact* with one another, typically by exchanging messages through some computer network infrastructure.

Die Interaktionsform zwischen den einzelnen Agenten ist entscheidend für die Arbeitsweise des Multiagentensystems. Jeder Agent ist fähig, unabhängige Entscheidungen zu treffen. Folglich kann der Agent für sich selbst entscheiden, welche Aktion er ausführen möchte, damit er seine eigenen Ziele erreicht. Dem Agenten muss nicht explizit mitgeteilt werden, was er tun soll. Er geht autonom vor. Prinzipiell haben die Agenten eines Multiagentensystems sehr unterschiedliche Ziele und Motivationen. Damit Agenten ihre eigenen Ziele erreichen können, müssen sie mit anderen Agenten des Multiagentensystems erfolgreich interagieren. Hierzu benötigen die Agenten Fähigkeiten, um miteinander kooperieren, koordinieren und verhandeln zu können.

Zudem benötigt jeder Agent eine gewisse Art von Kontrolle (Einflussmöglichkeit) auf seine unmittelbare Umgebung. Allerdings hat nicht jeder Agent einen gleich großen Einflussbereich, und die Einflussbereiche verschiedener Agenten können sich überschneiden. Dies kann zu Abhängigkeitsverhältnissen und verschiedenen Interaktionstypen zwischen Agenten führen, wie in Abbildung 2.1 veranschaulicht wird. Der Entwickler des

Multiagentensystems muss sich über die verschiedenen Interaktionstypen der Agenten im Klaren sein, damit er die Vorgänge innerhalb des Multiagentensystems versteht.

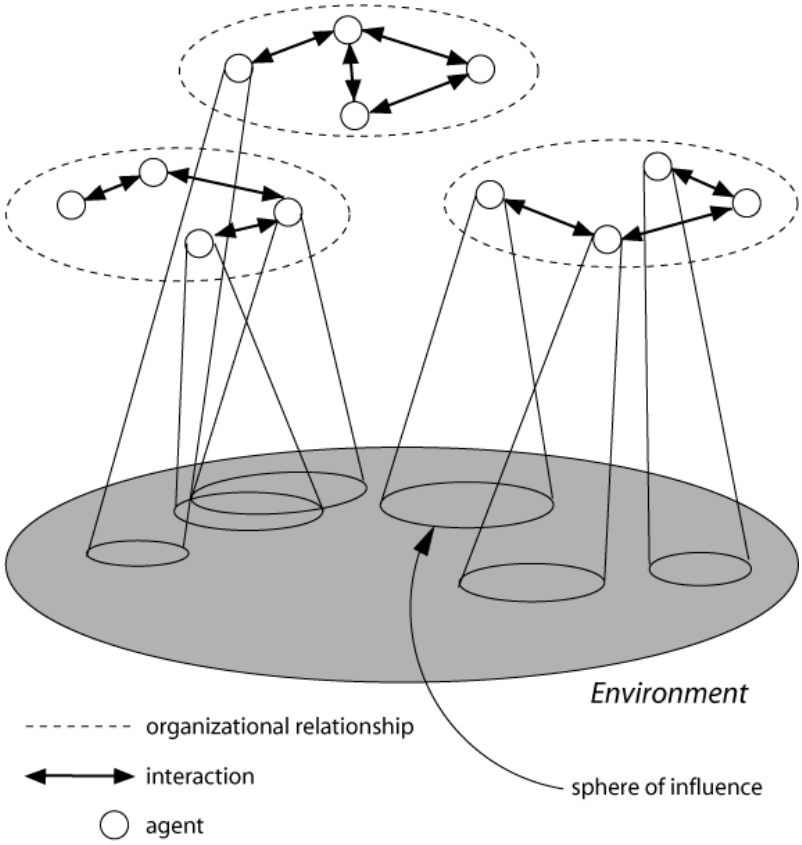

Abbildung 2.1: Typische Struktur eines Multiagentensytems (aus [Woo02])

Die Agententechnologie befasst sich somit nicht nur mit dem Design einzelner Agenten, sondern insbesondere auch mit einem System, das aus mehreren Agenten besteht. Nach [Woo02] ist gerade die soziale Eigenschaft ein wichtiges Merkmal für die Einzigartigkeit und die Intelligenz der Spezies Mensch. Sie kann nur durch eine Sprache auf höchster Ebene und durch die Fähigkeiten zum Kooperieren, Koordinieren und Verhandeln erreicht werden. In einem Multiagentensystem dienen menschliche Gesellschaften als Vorbilder, um zum Beispiel das soziale Verhalten in einer Gesellschaft von selbstorganisierenden Agenten optimieren zu können. Hierfür wird eine allgemeine Agentensprache[1] entwickelt, damit Agenten auf höchstem Niveau miteinander kommunizieren und ihre mentalen Zustände anderen Agenten oder auch Menschen mitteilen können.

2.1.3 Agenten und Objekte

Die agentenorientierte Softwareentwicklung kann als Weiterentwicklung der Objektorientierung verstanden werden. Laut [Jen01] gibt es offensichtliche Ähnlichkeiten zwischen

[1] Die *ACL* (Agent Communication Language).

Objekten und Agenten, wie zum Beispiel die Anwendung des *Geheimnisprinzips* (information hiding). Objekte kapseln genau wie Agenten einen Zustand und können in diesem Zustand Methoden anwenden oder Aktionen ausführen. Zudem können nicht nur Agenten, sondern auch Objekte durch den Austausch von Nachrichten interagieren.

Dennoch gibt es eine Reihe von signifikanten Unterschieden zwischen Objekten und Agenten. Wooldridge nennt drei Unterschiede ([Woo02], S.25-26), wobei der wichtigste im Grad der Autonomie liegt. Ein Objekt kann seinen inneren Zustand verstecken, bietet aber zugleich Methoden für andere Objekte an, damit diese auf den inneren Zustand des Objekts zugreifen können. Der Aufruf einer Methode führt zur Ausführung einer Aktion des Objekts. Folglich wird das Verhalten eines Objekts extern kontrolliert. Dies ist ein Unterschied zum Verhalten der Agenten. Der Agent hat nicht nur die Kontrolle über seinen internen Zustand, sondern auch über sein Verhalten. Ein externer Agent kann einen anderen Agenten nur mit einer Nachricht informieren, dass dieser eine Aktion ausführen soll. Der informierte Agent entscheidet aber selbst, ob er die Aktion ausführt oder nicht. Somit kontrollieren Agenten im Gegensatz zu Objekten, ihr Verhalten selber.

Darüber hinaus werden Agenten von Objekten anhand ihrer flexiblen, autonomen Handlungsmöglichkeiten unterschieden. Ein Agent kann mit verschiedenen (reaktiven, proaktiven oder sozialen) Eigenschaften modelliert werden. Diese Eigenschaften führen zu einem flexiblen Verhalten, das zwar für Objekte nachgebildet werden kann, aber nicht explizit in deren objektorientierten Modellen unterstützt wird.

Des Weiteren zeichnet sich ein Multiagentensystem durch eine unabhängige Ausführung von Agenten aus. Die parallele Ausführung mehrerer Agenten, die gemeinsam an der Lösung eines Problems arbeiten, ist ein elementarer Bestandteil des Agentenparadigmas. Somit können Agenten immer aktiv sein, während Objekte nur bei Methodenaufrufen Berechnungen durchführen. Objekte können zwar auch nebenläufig ausgeführt werden, aber entscheidend ist laut ([Woo02], S.26):

> ... in the standard object model, there is a single thread of control in the system.

Gängige objektorientierte Programmiersprachen wie zum Beispiel *Java*, unterstützen zwar die Nebenläufigkeit, aber sie erfassen nicht das Konzept der autonomen Entitäten, wodurch sich die nebenläufige Ausführung von Agenten anders auswirkt, als es bei nebenläufigen Objekten der Fall ist.

2.1.4 Agentenarchitekturen

Agentenarchitekturen sind Softwarearchitekturen für Agenten. Sie legen den inneren Aufbau und die äußere Struktur von Agenten fest, wobei insbesondere die innere Struktur eines Agenten im Fokus der Architekturentwicklung liegt. Hierzu gehört, wie der Zustand aktualisiert und wie die nächste Aktion des Agenten bestimmt wird. Die Agen-

tenarchitektur ist somit für das Verhalten eines Agenten zuständig. Allerdings gibt es eine Vielzahl an unterschiedlichen Agentenarchitekturen. Prinzipiell lassen sich drei Klassen von Agentenarchitekturen unterscheiden (vgl. [Sud04]), die Klasse der *reaktiven*, der *deliberativen* und der *hybriden* Agentenarchitekturen.

Reaktive Agenten

Die Entwicklung von reaktiven Agenten wurde sehr stark durch die Verhaltenspsychologie beeinflusst. In [Mül96] werden reaktive Agenten als Agenten beschrieben, die ihre Entscheidungen zur Laufzeit treffen, ohne die Entscheidung zum Beispiel von intern gespeicherten Informationen abhängig zu machen. Sie reagieren schnell und handeln nach einfachen Regeln. Reaktive Agenten besitzen keinerlei Informationen über ihre Umwelt und verfügen insbesondere über kein internes Modell der Welt. Ihr Verhalten wird direkt durch die sensorische Eingabe (Wahrnehmung) bestimmt. Eine Wahrnehmung des Agenten kann somit zu der sofortigen Ausführung einer Aktion führen. Diese Art der Agentenarchitektur führt zu robusten Agenten, die sich besonders für Umgebungen eignen, welche für Agenten schwer zu handhaben sind. Da auf ein komplexes, internes Verarbeiten von Informationen verzichtet wird, gilt die Implementierung eines reaktiven Agenten als weniger komplex als die Entwicklung von deliberativen oder hybriden Architekturen.

Rodney Brooks entwickelte 1986 die *Subsumptionsarchitektur* (siehe [Bro86]), die zu den bekanntesten Vertretern der reaktiven Agentenarchitekturen zählt. Anstatt den Weg der klassichen AI[2] zu verfolgen und ein System in voneinander unabhängige, funktionale Verarbeitungseinheiten zu zerlegen, wird eine aktivitätsorientierte Dekomposition des Systems durchgeführt. Daraus resultieren unabhängige Module (*behaviour modules*), die Aktivitäten produzieren sollen und die nicht mit einer Abstraktion der Welt, sondern direkt mit der realen Welt verbunden sind. Die Architektur wird insbesondere in der Robotik verwendet. Die Verhaltensmodule erhalten auf direktem Wege die Sensordaten und prüfen die eigene Vorbedingung auf Erfüllung. Falls die Vorbedingung erfüllt ist, produziert das jeweilige Modul eine spezifische Aktion.

Deliberative Agenten

Im Gegensatz zu den reaktiven Agenten weisen die deliberativen Agenten eigenes Wissen auf. Der Ansatz beruht auf der Aussage der *physical symbol system hypothesis* von Simons und Newells (siehe [NS76]), nach der Agenten eine interne Repräsentation der Welt aufrechterhalten müssen. Zudem besitzen sie einen expliziten mentalen Zustand. Sowohl die Welt als auch der Zustand eines Agenten werden intern durch Symbole abgebildet. Durch die Suchmöglichkeiten innerhalb der internen Symbolmenge und durch Operationen auf diesen Symbolen soll intelligentes Handeln ermöglicht werden.

[2] Künstliche Intelligenz (Artificial Intelligence).

Die BDI-Architektur (Beliefs Desires Intentions) ist der bekannteste Ansatz für die Entwicklung von deliberativen Agenten. Der Philosophieprofessor Michael Bratman entwickelte 1987 (siehe [Bra87]) ein ursprünglich philosophisch gedachtes Modell zur Beschreibung von rationalem Verhalten anhand der Konzepte *Beliefs* (Glaube), *Desires* (Wünsche) und *Intentions* (Absichten).

Beliefs drücken den Glauben eines Agenten aus. Es sind Informationen oder auch Annahmen über die Umwelt eines Agenten. Diese Annahmen müssen nicht notwendigerweise auf die Wirklichkeit zutreffen. Sie entsprechen nur dem Glauben des individuellen Agenten und stellen die subjektive Weltanschauung eines Agenten dar.

Desires entsprechen den Wünschen eines Agenten. Sie beschreiben zukünftige Weltzustände, die der Agent erfüllt haben möchte. In den *Desires* können auch Wünsche enthalten sein, die nur alleine, aber nicht zusammen erfüllbar sind (Konflikt). Des Weiteren ist es möglich einen Wunsch zu haben, von dem der Agent nicht glaubt, dass er verwirklicht werden kann.

Intentions formulieren, im Gegensatz zu den *Desires*, aktive Vorhaben des Agenten. Hier werden die Absichten eines Agenten beschrieben, die aus den *Desires* abgeleitet wurden und an die der Agent glaubt. Der Agent verfolgt eine Absicht so lange, bis er diese erreicht, oder aufgegeben hat.

Rao und Georgeff haben 1995 (siehe [RG95]) das philosophische Konzept von Bratman in einer formalen Theorie spezifiziert, um ein ausführbares Modell für Agenten zu erreichen. Das Modell wurde mit den Konzepten der *Goals* (Ziele) und *Plans* (Pläne) ausgestattet, die die Konzepte der *Desires* und *Intentions* konkretisieren.

Die *Goals* entsprechen den aktuellen Zielen eines Agenten. Die Ziele werden üblicherweise durch mehrere *Beliefs* dargestellt und sie bilden eine Teilmenge der *Desires*, da bei den Zielen keine Konflikte auftreten dürfen. Ein Plan enthält mehrere *Intentions* und bildet eine konkrete Ausführungsbeschreibung, mit der ein bestimmtes Ziel oder auch Teilziel erreicht werden kann. Anhand dieser beiden Konzepte entsteht ein ausführbares BDI-Modell, dass die konkreten Konzepte der *Beliefs*, *Goals* und *Plans* benutzt. Das *Procedural Reasoning System* (PRS) (Georgeff und Lansky, 1987) gehört zu den bekanntesten und ersten Systemen, die das ausführbare BDI-Modell implementiert haben.

Hybride Agenten

In [Mül96] wird bemängelt, dass reaktive Agenten trotz ihrer Vorteile ein großes Defizit haben. Sie können nicht zielorientiert arbeiten. Deliberative Agenten arbeiten dagegen zielorientiert, aber ihr Handeln basiert auf allgemeinen Schlussfolgerungsmechanismen, die nicht gut gesteuert werden können und die kaum reaktives Handeln ermöglichen. Um diese Restriktionen aufzuheben und die Vorteile beider Architekturen nutzen zu

können gibt es hybride Agenten.

Ein naheliegender Ansatz ist nach [WJ94] einen hybriden Agenten aus zwei (oder mehreren) Subsystemen zu entwerfen. Der Agent besteht somit mindestens aus einem deliberativen und einem reaktiven Subsystem. Der deliberative Teil erlaubt dem Agenten einen Zugriff auf das intern gespeicherte Weltmodell sowie die zeitintensive Entwicklung von Strategien, um eigene Ziele zu erreichen. Im Gegensatz dazu ist der reaktive Teil eines Agenten für eine direkte und schnelle Reaktion auf Ereignisse in der Umwelt verantwortlich. Diese Art der Strukturierung führt meistens zu einem Schichtenmodell (layerd architecture), wie es zum Beispiel in [Mül96] vorgestellt wird. Hier werden die verschiedenen Funktionalitäten eines Agenten in zwei oder mehreren Schichten hierarchisch organisiert.

Eine hybride Technik wird immer dann eingesetzt, wenn die Stärken anderer Techniken vereint und maximiert sowie ihre Schwächen minimiert werden sollen, um so den bestmöglichsten Ansatz für die eigenen Zwecke zu erreichen (nach [JW98]). Diese Regel gilt auch für hybride Agentenarchitekturen.

2.1.5 Gründe für Agenten

Die Agententechnologie ist nur für einen speziellen Anwendungsbereich geeignet und soll somit nicht in beliebigen Bereichen eingesetzt werden. In diesem Abschnitt soll kurz erläutert werden, anhand welcher Kriterien der Entwickler herausfindet, ob ein Lösungsansatz mittels des Agentenparadigmas vorteilhaft ist oder ob vielleicht doch besser traditionelle Paradigmen eingesetzt werden sollen, wie zum Beispiel die Objektorientierung.

Padgham und Winikoff nennen in [PW04] drei Eigenschaften, die eine Applikation haben kann, damit sich der Einsatz von Agenten lohnt, und stellen dar, welche Vorteile aus dem Einsatz von Agenten entstehen.

Agenten werden vor allem in Applikationen eingesetzt, die hochgradig komplex und dezentralisiert sind, da sie in erster Linie die Kopplung des Gesamtsystems reduzieren. Agenten arbeiten nicht nur autonom, sondern sie sind auch robust gegenüber Fehlern und können reaktiv und proaktiv handeln. Zudem müssen Agenten nicht ständig (extern) kontrolliert werden, sondern sie übernehmen Eigenverantwortung, um ein Ziel zu erreichen, und agieren dabei sehr flexibel. Durch diese Art der Selbstorganisation wird die Kopplung innerhalb des Systems reduziert und das Gesamtsystem wird beherrschbar. In diesem Sinne werden Agenten auch gerne als Wrapper für Altsysteme (Legacy Systems) eingesetzt (nach [PW04]), um diese leichter mit neuen Technologien verknüpfen zu können, anstatt das ganze System ersetzen zu müssen.

Zudem bieten sich Agenten generell in Umgebungen an, die sehr anspruchsvoll sind (dynamisch, unvorhersehbar, unzuverlässig) und in denen es immer wieder zu Fehlern

kommen kann. Die NASA verwendete zum Beispiel 1998 im Projekt *Deep Space 1*[3] ein Agentensystem, welches für die Planung und Ausführung von Aktivitäten des Raumfahrzeuges verantwortlich war.

Darüber hinaus biete sich der Einsatz von Agenten insbesondere in Bereichen an, in denen der Agent als Stellvertreter für den Menschen agieren soll. Durch die verschiedenen Verhaltensformen der Agenten, in Bezug auf problemorientiertes Handeln, wirken sie menschenähnlicher. Zu diesem Gebiet gehören zum Beispiel die Bereiche der Simulationen sowie des Entertainments (nach [PW04]).

2.1.6 Agentenplattform

In der Definition eines Multiagentensystems wurde bereits darauf hingewiesen, dass eine Netzwerkinfrastruktur (computer network infrastructure) bereitgestellt werden muss, damit Agenten wie gewünscht ablaufen und interagieren können. Die Infrastruktur wird durch eine sogenannte Agentenplattform bereitgestellt. Die FIPA (Foundation for Intelligent Physical Agents)[4] beschreibt eine abstrakte Architektur (siehe [FIPA02a]) für Agentenplattformen, im Sinne einer Menge von Schnittstellen, die eine FIPA-konforme Agentenplattform implementieren muss.

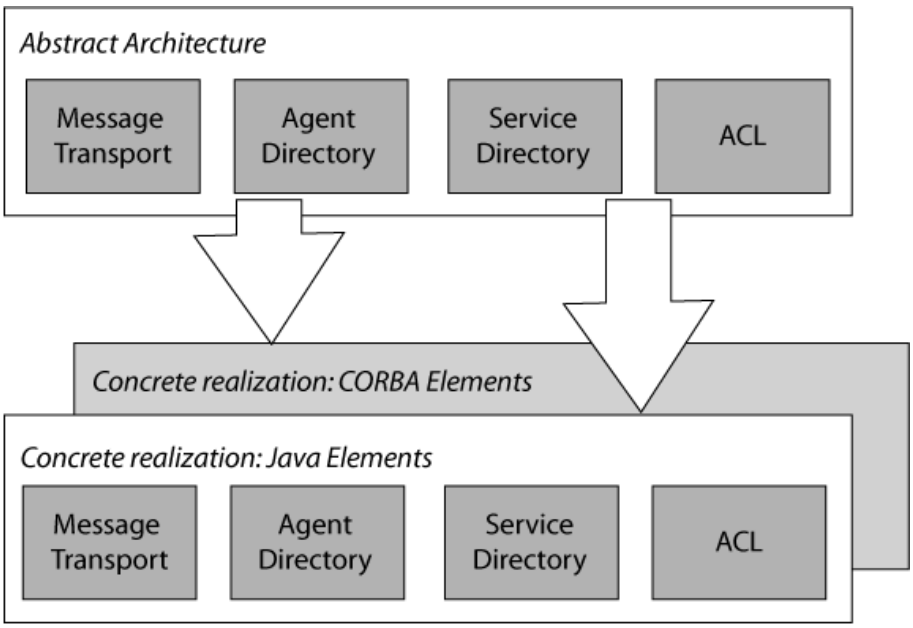

Abbildung 2.2: FIPA Abstract Architecture (aus [FIPA02a])

Die abstrakte Architektur zeichnet sich durch drei Hauptkomponenten aus. Die ADS-Komponente (Agent Directory Service) ist für die Verwaltung der Agenten zuständig. Sie wird meistens selbst als ein spezieller Agent realisiert. Dieser Agent benötigt einen direkten Zugriff auf die Funktionen der Agentenplattform und ist für die Verwaltung

[3] Ausführliche Projektinformationen unter `http://nmp.nasa.gov/ds1/`.
[4] Weitere Informationen unter http://www.fipa.org.

aller weiteren Agenten des Systems verantwortlich. Zu den Verwaltungsaufgaben des ADS gehören insbesondere die Erzeugung und Initialisierung sowie das Entfernen von Agenten.

Damit Agenten ihre Dienstleistungen anderen Agenten zur Verfügung stellen können und selber auch in der Lage sind, andere Dienste finden und nutzen zu können, gibt es die SD-Komponente (Service Directory). Diese Komponente bietet Funktionen an, um Dienstleistungen registrieren, deregistrieren, finden und modifizieren zu können. Genau wie beim ADS wird meistens auch die SD-Komponente als ein gesonderter Agent realisiert. Aufgrund der Implementierung als Agent passen ADS und SD auf natürliche Art und Weise in das Umfeld eines Multiagentensystems.

Das Nachrichtentransportsystem MTS (Message Transport Service) bildet die dritte fundamentale Komponente einer jeden FIPA-konformen Agentenplattform. Diese Komponente ist für den Nachrichtenaustausch zwischen den lokalen Agenten einer Agentenplattform verantwortlich. Zudem kümmert sie sich um den Transport von Nachrichten zwischen zwei oder mehreren voneinander getrennten Agentenplattformen, wobei nur gefordert wird, dass jede Agentenplattform die Spezifikation der FIPA erfüllt.

Die ACL-Komponente (Agent Communication Language) ist für die Agentenkommunikation zuständig. Insbesondere das Format der Nachrichten wird durch den FIPA-Standard ACL festgelegt. Dies ist eine universelle Sprache für die Kommunikation zwischen Agenten, die den Aufbau einer Agentennachricht definiert (siehe [FIPA02b]).

Der FIPA-Standard schafft Interoperabilität von verschiedenen Agenten und Agentenplattformen. Zudem reduziert er den Entwicklungsaufwand und erzeugt Gemeinden (Communities). Allerdings ist eine Standardisierung mühsam und benötigt üblicherweise viel Zeit. Deswegen sollten nur die elementaren Komponenten und Strukturen eines Systems standardisiert werden.

Im nächsten Abschnitt werden die Grundlagen der Simulationstechnologie vorgestellt.

2.2 Simulationstechnologie

Die Simulationstechnologie beschäftigt sich mit der Bildung einer allumfassenden Theorie für die Modellierung und Simulation komplexer Systeme (nach [Zei76]). Die erstellten Konzepte der Beschreibung, Modellierung, Validierung, Simulation und Erforschung sollen in eine abstrakte Form gebracht werden, die den unterschiedlichsten Anwendungsgebieten zu Nutze kommt. Die Computersimulation hat nicht zuletzt durch den Technologiefortschritt im Software- und Hardwarebereich eine bedeutende Stellung in der Analyse und Modellierung von dynamischen Systemen eingenommen. Ihre entwickelten algorithmischen Lösungs-, Software- und Architekturkonzepte helfen bei der Erstellung geeigneter Modellierungssoftware. Obgleich die Simulationstechnologie in den un-

terschiedlichsten Gebieten eingesetzt wird, bleiben die im folgendem Kapitel vorgestellten Konzepte und Grundbegriffe erhalten.

2.2.1 Modellierung und Simulation

Die Simulation dient als ein Instrument zur Analyse und Modellbildung komplexer Systeme. Page ([Pag91], S.7) schreibt:

> Man spricht ganz allgemein von *Simulation*, wenn bei der Systemanalyse ein Modell an die Stelle des Originalsystems tritt und Experimente am Modell durchgeführt werden.

Ein System (Originalsystem) bezeichnet ein Konstrukt, dessen Komponenten in einer Beziehung stehen, so dass sie sich von ihrer Umwelt abgrenzen (siehe Abbildung 2.3). Durch die Beziehungen der einzelnen Komponenten untereinander erhält das System einen inneren Aufbau, der als Struktur bezeichnet wird. Im Bereich der Simulation wird das System von seiner Umwelt durch eine einer besonderen Fragestellung abgegrenzt. Das System entspricht einem Teil der realen Welt, der im Interesse des Simulationsanwenders ist. Ein System kann natürlich oder künstlich sein. Die reale Existenz des Systems ist nicht erforderlich. Systeme, die für die Zukunft geplant sind, können auch im Interesse des Simulationsanwenders liegen.

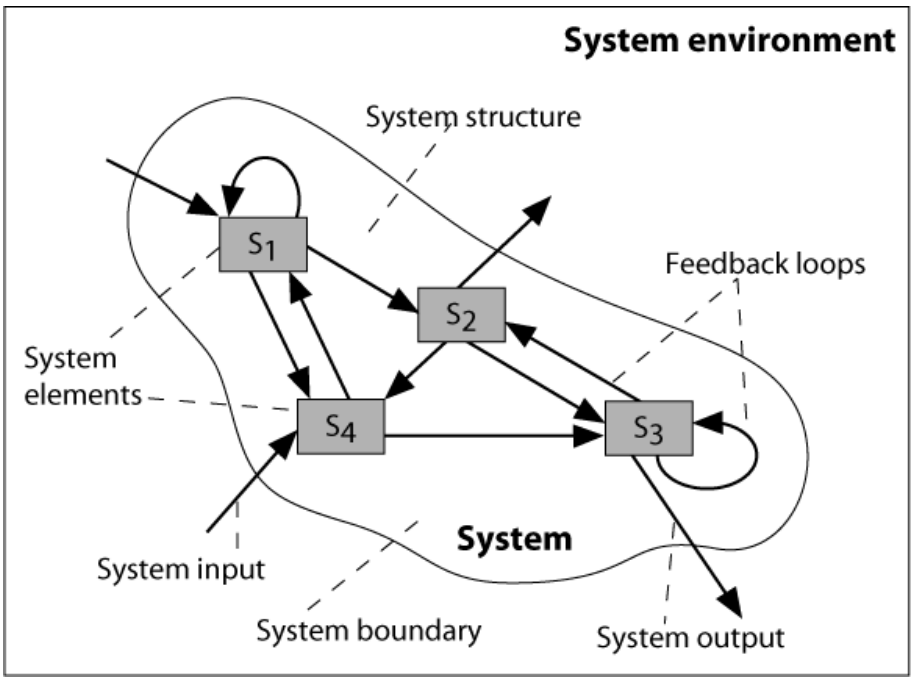

Abbildung 2.3: Grundlegende Systembegriffe (aus [PK05])

Modelle helfen ein System und seine Wechselwirkungen besser zu verstehen. Mittels Abstraktion und Idealisierung wird das Originalsystem auf ein Modell abgebildet. Das

Modell ist die Simplifizierung des Originals. Erst durch die vereinfachte Darstellung des Originals werden komplexe Systeme beherrschbar. Die surjektive Abbildung vernachlässigt alle Eigenschaften des Systems, die nicht im Interesse des Modellierers liegen. Somit können verschiedene Fragestellungen für ein und dasselbe System auch zu unterschiedlichen Modellen führen. Falls das reale System über eine Struktur verfügt, besitzt auch das Modell einzelne Komponenten, die in einer Wechselwirkung stehen. Die Komponenten des Modells bestehen aus einem Zustand und einer Menge von Transformationsregeln. Die Transformationsregeln entsprechen Methoden, die einen Zustandswechsel zu einer bestimmten Zeit herbeiführen können.

Damit die Experimente am Modell auch zu aussagekräftigen Ergebnissen führen, muss das Modell vor der Simulationsphase validiert werden. Eine detaillierte Analyse des Originalsystems, ist die Vorraussetzung für die korrekte Abbildung des Modells. Hierfür werden ausreichende Daten des Originalsystems gesammelt, damit in mehreren Iterationen die Ergebnisse des Modells mit denen des Originalsystems verglichen werden können. Das Modell wird in mehreren Zyklen getestet und kalibriert, bis das Originalverhalten dem Modellverhalten ausreichend entspricht.

In der Computersimulation berechnet der Computer das Verhalten des Originalsystems anhand eines ausführbaren Modells. Richard M. Fujimoto definiert die Computersimulation (in [Fuj00], S. 4) als:

> A computer simulation is a computation that models the behaviour of some real or imagined system over time.

Die Idee wird von Zeigler [Zei76] veranschaulicht. Das Originalsystem entspricht einer Quelle von Verhaltensdaten. Die interessanten Verhaltensdaten werden in einem Verlauf über die Zeit betrachtet. Das Modell bietet eine Menge von Instruktionen an, um die Verhaltensdaten zu konstruieren. Modelle können durch verschiedene Formalismen ausgedrückt werden, wie zum Beispiel durch Differentialgleichungen oder ereignisdiskrete Formalismen. Diese verschiedenen Formen der Modellbeschreibungen, stellen alle Anweisungen für "jemand oder etwas" bereit, um Daten generieren zu können. Der Computer ist heutzutage derjenige, der die Anweisungen des Modells ausführt und Daten generiert.

In der Simulation ist das Modell von seinem Simulator (Observer) getrennt (siehe Abbildung 2.4), da dasselbe Modell von unterschiedlichen Simulatoren ausgeführt werden kann. Des Weiteren ermöglicht es eine Portierung und Interoperabilität auf hohem Abstraktionsniveau (nach [Zei76]).

Zwischen den verschiedenen Relationen der drei Hauptkomponenten (System, Modell und Simulator) sind nach [ZPK99] zwei Beziehungen hervorzuheben (siehe Abbildung 2.4). Die *Modellierungsrelation* (representation) bestimmt die Gültigkeit des Modells. Die Beziehung zwischen Modell und System bestimmt, mit welchem Grad das Modell das

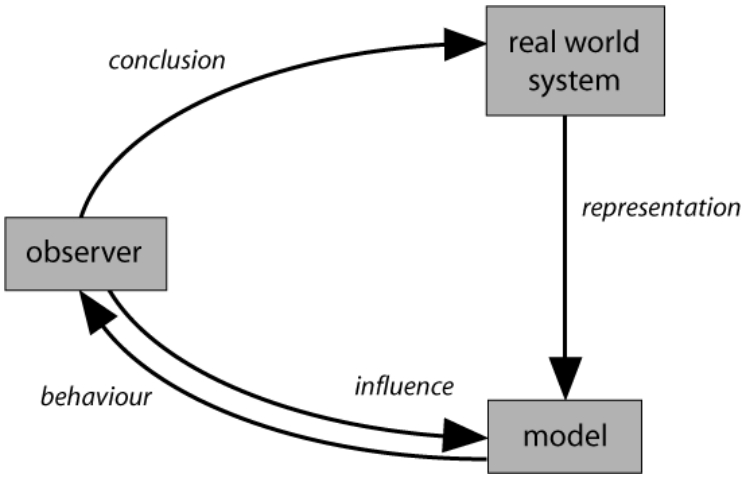

Abbildung 2.4: System, Modell und Experimentator (aus [PK05])

System präsentiert. Für den Simulationsanwender ist es ausreichend, wenn das Modell nur das interessante Verhalten des Originalsystems gültig wiedergibt. Jegliche Art von anderweitigem Verhalten des Originals, das in keiner Verbindung mit dem interessanten Verhalten steht, kann weggelassen oder vernachlässigt werden. Die *Simulationsrelation* (influence) definiert die Korrektheit, mit der der Computer die Anweisungen des Modells ausführt. Hierbei geht es um eine korrekte Modellausgabe (behaviour), die anhand der aktuellen Modelleingabe und dem initialen Zustand des Modells berechnet wird.

2.2.2 Zeit

Die Zeit spielt eine elementare Rolle in der Simulationstechnologie. Laut [Fuj00] berechnet die Computersimulation das Verhalten eines Modells innerhalb einer bestimmten Zeit. Hierfür muss das Modell seinen Zustand repräsentieren können und einen Mechanismus für einen Zustandswechsel bereitstellen sowie eine Repräsentation der Zeit besitzen. Der Computer kann den Zustand eines Modells durch Variablen repräsentieren und mittels Variablenzuweisungen können Zustandswechsel realisiert werden.

Die Zeit hat innerhalb der Simulationsbranche viele unterschiedliche Notationen. Fujimoto schlägt folgende Zeit-Definitionen vor ([Fuj00], S. 27):

Physikalische Zeit (physical time): Die Zeit im physikalischen System (Originalsystem).

Simulationszeit (simulation time): Die Simulation nutzt diese Abstraktion, um die physikalische Zeit modellieren zu können.

Systemzeit (wallclock time): Die Zeit, die während der Simulationsausführung vergeht.

Die physikalische Zeit und die Systemzeit funktionieren im Grunde wie die bekannte, konventionelle "Zeit". Die Simulationszeit benutzt hingegen ein eigenes Konzept, das nur im Bereich der Simulationen existiert ([Fuj00], S.28):

Simulation time is defined as a totally ordered set of values where each value represents an instant of time in the physical system being modeled. Further, for any two values of simulation time T_1 representing physical time P_1, and T_2 representing P_2, if $T_1 < T_2$, then P_1 occurs before P_2, and $(T_2 - T_1)$ is equal to $(P_2 - P_1) * K$ for some constant K. If $T_1 < T_2$, then T_1 is said to occur *before* T_2, and if $T_1 > T_2$, then T_1 is said to occur after T_2.

Der Fortschritt der Simulationszeit kann eine direkte Verbindung zur Systemzeit haben, muss es aber nicht. Simulationsausführungen, in denen sich die Simulationszeit genau wie die Systemzeit verhält, werden als Ausführung in *Realzeit* (real-time) bezeichnet. Diese Ausführungszeit wird zum Beispiel in virtuellen Umgebungen benutzt (siehe [Fuj00]).

Die *skalierte Realzeit* (scaled real-time) ist eine Variation der *Realzeit*. Diese Ausführungsart schreitet um einen konstanten Faktor schneller oder langsamer als die Systemzeit voran. Die Ausführung in Realzeit ist somit ein Spezialfall der skalierten Realzeit, in dem der Skalierungsfaktor auf *eins* gesetzt ist. Die skalierte Ausführung ist nützlich, um uninteressante Teilbereiche einer Simulation überspringen (schnelle Ausführung) und interessante Teilbereiche besser analysieren zu können (langsame Ausführung). Beide Ausführungsmodi benutzen nach ([Fuj00], S.29) die Mapping-Funktion 2.1, um die Systemzeit in die Simulationszeit zu transformieren:

$$T_s = W2S(T_w) = T_{start} + Scale * (T_w - T_{wStart}) \tag{2.1}$$

T_w ist die aktuelle Systemzeit und T_{start} ist die Simulationszeit, zu der die Simulation gestartet ist, wohingegen T_{wStart} die Systemzeit präsentiert, die beim Start der Simulation vorlag. $Scale$ ist der Skalierungsfaktor, der die Zeit schneller oder langsamer ablaufen lässt.

In analytischen Simulationen, die nicht Menschen als Komponente innerhalb der Simulation aufweisen, ist die Simulationszeit prinzipiell nicht an die Systemzeit gebunden. Die Berechnungen eines Simulationsschrittes kann von Millisekunden bis Stunden reichen. Die Simulationszeit schreitet erst nach der beendeten Berechnung weiter, egal wie lange die Berechnung dauert. Somit ist die Simulationszeit völlig von der Systemzeit entkoppelt. Diese Art der Ausführung wird als *so-schnell-wie-möglich* (as-fast-as-possible) bezeichnet (nach [Fuj00]).

2.2.3 Modellklassifikation

Die Klassifikation von Modellen kann durch eine Vielzahl von unterschiedlichen Kriterien vorgenommen werden. Laut [Pag91] können Modelle nach der Art der *Untersuchungsmethode*, nach dem *Abbildungsmedium*, nach dem *Verwendungszweck* und schließlich nach Art der *Zustandsübergänge* klassifiziert werden.

In Abbildung 2.5 werden Modelle nach der Art der Zustandsübergänge klassifiziert.

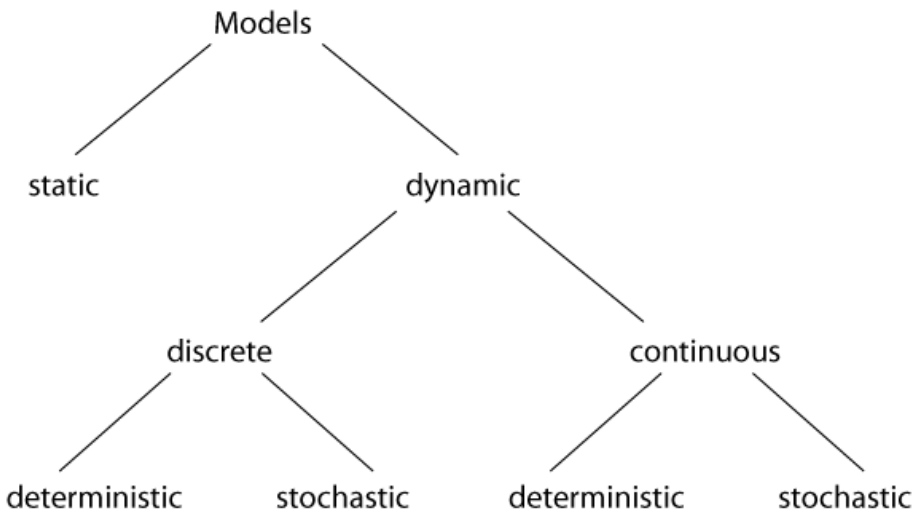

Abbildung 2.5: Modellklassifikation nach Art der Zustandsübergänge (aus [PK05])

Ein *statisches* Modell hat keinen Zeitbezug und es treten keine Zustandsänderungen auf, somit gibt es auch kein interessantes Systemverhalten zu beobachten. Damit sind diese Modelle nicht weiter relevant für die in dieser Arbeit festgelegte Zielsetzung der Simulationsunterstützung. *Dynamische* Modelle sind wiederum interessant für die Simulation, da sie von einer Zeit abhängig sind, die Zustandsänderungen im Modell auslöst. Die Zustandsänderungen in einem dynamischen Modell können wiederum auf zwei verschiedene Arten ausgelöst werden.

Auf der einen Seite gibt es dynamische, *kontinuierliche* Modelle, deren Zustandsänderungen mittels höherer Mathematik auf kontinuierliche Art und Weise berechnet werden. Hierbei lassen sich die Zustandsvariablen und ihre Änderungen zum Beispiel durch stetige Funktionen beschreiben. Auf der anderen Seite gibt es die *diskrete* Modellierung dynamischer Modelle. In diesem Modell werden Zustandsvariablen nur zu bestimmten diskreten Zeitpunkten geändert.

In ([Pag91], S.6) wird ein Modell als *deterministisch* bezeichnet, "wenn seine Reaktion auf eine bestimmte Eingabe, ausgehend von einem bestimmten Zustand, eindeutig festgelegt ist." Ist dies nicht der Fall, wird das Modell als *stochastisch* bezeichnet, da die Reaktion des Modells nur durch eine Wahrscheinlichkeitsverteilung angegeben werden kann.

Modelle können auch anhand der Zeitpräsentation im Modell klassifiziert werden (siehe Abbildung 2.6). Diese Art verdeutlicht die drei grundlegenden Simulationsarten (continiuous, time-stepped, event-driven), die nun im folgenden genauer betrachtet werden sollen.

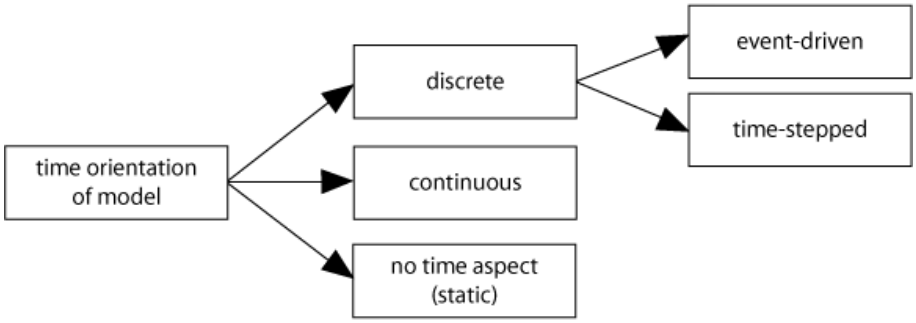

Abbildung 2.6: Modellklassifikation nach Art der Zeitpräsentation (aus [PK05])

Kontinuierliche Modellierung

Kontinuierliche Modelle gehören historisch betrachtet zu den ersten Simulationsmodellen ihrer Art. Sie bestehen aus einem System von Differentialgleichungen, mit einer freien Zeitvariable. Die Simulation besteht im Wesentlichen darin, die Differentialgleichungen mit Hilfe von numerischen Integrationsverfahren zu lösen (siehe [ZPK99]). Diese Modelle eignen sich besonders für Systeme mit stetigen Zustandsänderungen, also Systeme, deren Zustand sich kontinuierlich verändert. Somit gibt es in kontinuierlichen Modellen auch keinen expliziten Nachfolgezustand. Es wird eine Ableitungsfunktion benutzt, die die Änderungsrate der Zustandsvariablen spezifiziert. Zu jedem Zeitpunkt auf der Zeitachse ist nur die Änderungsrate der gegebenen Zustände bekannt. Von dieser Information ausgehend wird der Zustand zu jedem Punkt in der Zukunft berechnet. Diese Modelle werden unter anderem bei der Untersuchung komplexer physikalischer Vorgänge eingesetzt, zum Beispiel bei der Wettervorhersage.

Diskrete Modellierung

Aufgrund der speziellen Anwendungsgebiete der kontinuierlichen Modellierung beschäftigt sich diese Arbeit ausschließlich mit der diskreten Modellierung von dynamischen Modellen. In der diskreten Modellierung berechnet eine Zustandsübergangsfunktion anhand des aktuellen Zustands und des Einflusses der Umgebung den Nachfolgezustand zur nächsten Zeitinstanz (siehe [ZPK99]). Diskrete Modelle werden wiederum in Zeitmodelle und Ereignismodelle unterteilt.

Diskrete Zeitmodelle

Diskrete Zeitmodelle sind die verständlichsten von allen dynamischen Modellen. Sie zeichnen sich durch einen einfachen Simulationsalgorithmus aus, in dem die Zeit in regelmäßigen, diskreten Schritten voranschreitet (*time steps*). Jeder Zeitsprung bewirkt eine Neuberechnung des Systemzustandes. Innerhalb einer Simulationsrunde werden der Nachfolgezustand und die Ausgabe durch den aktuellen Zustand des Systems und die

Eingabe der Umgebung bestimmt. Das diskrete Zeitmodell benötigt also zwei Funktionen (siehe [ZPK99]), eine Funktion für die Berechnung des Nachfolgezustandes und die zweite für die Ausgabe des Modells.

Die δ-Funktion (state transition function) berechnet den neuen Zustand des Modells, zum Zeitpunkt $t+1$, anhand dem alten Zustand $q(t)$ und der aktuellen Eingabe $x(t)$, zum Zeitpunkt t:

$$q(t+1) = \delta(q(t), x(t)) \tag{2.2}$$

Die λ-Funktion (output function) berechnet die Ausgabe, zum Zeitpunkt t, mittels Zustand $q(t)$ und der Eingabe $x(t)$, zum Zeitpunkt t:

$$y(t) = \lambda(q(t), x(t)) \tag{2.3}$$

Der Simulationsalgorithmus 1, stellt die einfachste Form eines diskreten Zeitmodels dar. Er durchläuft ein definiertes Zeitintervall $[T_i, T_f]$ und führt pro Zeitinstanz die Berechnung der δ- und der λ-Funktionen aus.

$t \leftarrow Ti$;
while $t <= T_f$ **do**
  $y(t) \leftarrow \lambda(q(t), x(t))$;
  $q(t+1) \leftarrow \delta(q(t), x(t))$;
end

Algorithmus 1 : Diskreter Simulationsalgorithums

Anhand dieses simplifizierten Simulationsalgorithmus wird verdeutlicht, dass diskrete Zeitmodelle zu jedem Zeitpunkt in genau einem Zustand sind und immer eine Ausgabe pro Zustandswechsel produzieren. Des Weiteren definieren sie immer genau, wie sich dieser Zustand ändern kann. Im Falle von endlich vielen Zuständen und endlichen Eingaben kann so die komplette Tabelle der möglichen Zuständsänderungen berechnet und tabellarisch veranschaulicht werden.

Der Algorithmus scheint auf den ersten Blick sehr einfach zu sein, aber seine abstrakte Natur versteckt eine Fülle potentieller Komplexität. Bei einer Kopplung mehrerer Komponenten innerhalb des Systems (siehe cellular automata [ZPK99], S.41) kann ein komplexes unvorhersagbares Verhalten entstehen. Der globale Zustand des Systems besteht dann aus der Ansammlung aller lokalen Zustände. Der Algorithmus für solche Multikomponentensysteme liest in einem Simulationsschritt alle Komponenten, wendet jeweils die *state transition*-Funktionen auf den lokalen Zustand an und speichert das Ergebnis in einer Kopie. Wenn die Funktion auf alle Komponenten angewendet wurde, wird der globale Systemzustand hergestellt und die Zeit springt zum nächsten Zeitpunkt. In diesem Fall ist entscheidend, dass zu jedem diskreten Zeitpunkt die δ-Funktion auf alle Komponenten ausgeführt wird, egal, ob sich wirklich der Zustand der lokalen Kompo-

nente ändert oder nicht.

Anwendungsgebiete für die Ebenen der diskreten Zeitmodelle finden sich besonders für Systeme mit sprunghaften Zustandsänderungen in regelmäßigen Zeitabständen (Bedienungssysteme) (siehe [ZPK99]).

Diskrete Ereignismodelle

In der Regel kann davon ausgegangen werden, dass sich pro Zeitschritt nur wenige Komponenten innerhalb eines Multikomponentensystems ändern. Somit führt das diskrete Zeitmodell relativ viele unnötige Berechnungen durch. Um dieses Problem zu lösen und eine effizientere Berechnung zu erreichen, wurden die diskreten Ereignismodelle entwickelt. Ein Ereignis entspricht in diesem Modell genau einer Zustandsänderung. Die dem zugrunde liegende Idee besagt, dass sich die Modellausführung nur um die interessanten Punkte in der Zeit kümmern braucht (die Ereignisse). Die Neuberechnung der Zustandsvariablen wird auf das Eintreffen eines bestimmten Ereignisses gelegt. Dieser Ansatz erfordert von dem Simulationsalgorithmus spezielle Fähigkeiten, um zukünftige Ereignisse finden und einplanen zu können. Diskrete Ereignismodelle bieten somit eine genauere Abbildung bei geringerem Rechenaufwand als das zeitgesteuerte diskrete Modell, bei dem die Neuberechnungen des Systemzustandes zu regelmäßigen Zeitpunkten stattfindet [PLC00].

Diskrete Ereignismodelle können in verschiedenen Varianten (parallel oder hierarchisch) auftreten. An dieser Stelle genügt das Verständnis des traditionellen diskreten Ereignismodells, wie es in [ZPK99] logisch definiert ist. Es bildet die Grundlage aller weiteren Varianten. Die Spezifikation des traditionellen, diskreten Ereignissystems ist folgende Strukur:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

- X ist die Menge der Eingabevariablen
- S ist eine Menge von Zuständen
- Y ist die Menge der Ausgabevariablen
- $\delta_{int} : S \to S$ ist die interne Transitionsfunktion
- $\delta_{ext} : Q \times S \to S$ ist die externe Transitionsfunktion, wobei $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ die totale Zustandsmenge ist und e die Zeit, die seit der letzten Transition vergangen ist.
- $\lambda : S \to Y$ ist die Ausgabefunktion
- $ta : S \to R^{+}_{0,\infty}$ ist die Menge der positiven reellen Zahlen mit 0 und ∞.

Ein Simulator braucht solch eine Systemspezifikation, um das Modell ausführen zu können und so das beobachtete Verhalten des Originalsystems reproduzieren zu können. Dabei lassen sich die auszuführenden Operationen des Simulators exakt beschreiben. Das System befindet sich zu jedem Zeitpunkt in genau einem Zustand $s \in S$. Falls kein externes Ereignis aufgetreten ist, bleibt das System in seinem Zustand s, für die Restzeit $ta(s)$. Falls die Restzeit abgelaufen ist ($e = ta(s)$), ohne dass ein externes Ereignis auftritt, erzeugt das System von sich aus eine Ausgabe $y \in Y$ und wechselt in den neuen Zustand $\delta_{int}(s)$. Falls ein externes Ereignis $x \in X$ auftritt, bevor die Restzeit abgelaufen ist, wechselt das System in einen neue Zustand, der mittels $\delta_{ext}(s, e, x)$ berechnet wird. Der neue Zustand s' des Systems wird also entweder durch die interne oder die externe Transitionsfunktion festgelegt und das System bleibt maximal für eine Dauer von $ta(s')$ in diesem Zustand. Danach kann die Ausführung wieder von vorne beginnen.

Der Vollständigkeit halber muss erwähnt werden, dass es noch verschiedene Kombinationen der drei zuletzt genannten Grundmodelle gibt. Hierzu gehört unter anderem die Kombination von kontinuierlichen und diskreten Modellen. Die Kombinationsformen sind allerdings, genau wie die kontinuierliche Simulation, nur für spezielle Anwendungsdomänen interessant. Im weiteren Verlauf der Arbeit werden nur die diskreten Modelle verwendet. Der Algorithmus 1 und die Formalismen der diskreten Ereignismodelle, sollen als Grundlage für die spätere Simulationsarchitektur genutzt werden.

2.2.4 Hierarchische Systemspezifikation

Die hierarchische Systemspezifikation ist für jedes Framework die Basis zur Modellierung und Simulation. Sie bietet Methoden an, um existierende Realsysteme analysieren sowie Modelle ableiten und entwerfen zu können.

In einer hierarchischen Architektur ist jede Ebene mit Informationen über das System (Originalsystem) vertraut, die auf darunterliegenden Ebenen nicht bekannt sind. Diese Art der Abstraktion hilft, die Komplexität eines Systems in den Griff zu bekommen. Das Systemframework von Klir [Kli85] definiert vier grundlegende Ebenen:

Ebene	Name	Beschreibung
0	Quelle	Welche Variablen sollen beobachtet und gemessen werden
1	Daten	Gesammelte Daten des Quellsystems
2	Generator	Mittel um Daten für das Datensystem zu generieren
3	Struktur	Gekoppelte Komponenten formen ein generatives System

Tabelle 2.1: Systemframework von Klir (siehe [Kli85])

Ebene Null legt einen Ausschnitt der realen Welt fest, den der Simulationsanwender beobachten und somit auch modellieren möchte. In dieser niedrigen Ebene definiert der Modellierer, welche Variablen für ihn interessant sind und wie sie gemessen werden sollen. Ebene eins entspricht einer Sammlung aller angefallenen Daten, die aus den Beo-

bachtungen und Messungen des Quellsystems stammen. Diese Daten können durch verschiedene Regeln erzeugt werden. Der Generator auf Ebene zwei legt die Formalismen fest und stellt somit die Mittel bereit, um die Daten zu erzeugen. Spricht der Simulationsanwender über Modelle, ist in diesem Falle der Generator gemeint. Wird hingegen über ein System gesprochen, ist Ebene drei gemeint. Diese Ebene stellt ein spezifisches, generatives System bereit, das über genügend Informationen verfügt, um die beobachteten Daten von Ebene eins selbständig generieren zu können.

Das beschriebene Framework von Klir stellt eine einheitliche Perspektive bereit, um wichtige Strukturkonzepte voneinander unterscheiden zu können. Aus dieser Perspektive ergeben sich drei wichtige Strukturkonzepte:

1. Die Systemanalyse geht von einem bereits existierenden oder hypothetischen System aus, dessen Struktur bekannt ist. Das beobachtete Verhalten des Systems soll anhand der bekannten Struktur verstanden und erklärt werden können (Top-down Analyse). Eine bedeutende Form der Systemanalyse ist die Computersimulation. Hier werden Daten anhand der bereitgestellten Modellinstruktionen erzeugt und analysiert.

2. Bei der Systemdeduktion existiert ein System, dessen Struktur dem Anwender unbekannt ist. Anhand des allgemein beobachtbaren Systemverhaltens soll eine spezielle Struktur abgeleitet werden, die das gleiche Verhalten aufweist und auch die gleichen Daten generiert (Bottom-up Ansatz). Dieser Ansatz wird bei der Konstruktion von Modellen angewendet.

3. Beim Systemdesign werden Strukturen untersucht, um ein komplett neues System oder ein bereits existierendes zu gestalten. Meist existiert das Quellsystem noch nicht, sondern soll gebaut und mit bestimmten Funktionalitäten ausgestattet werden (Bottom-up Ansatz). Das Konzept des Systemdesigns wird, genau wie die Systemdeduktion, bei der Modellkonstruktion eingesetzt.

Dieser Abschnitt wurde nur eingebaut, um die Modellierung und Simulation vollständig abschließen zu können. Im weiteren Verlauf der Arbeit werden die vorgestellten Strukturkonzepte nicht explizit verwendet, da nicht der Entwurf und die Modellierung eines Multiagentensystems, sondern die Simulation von Multiagentensystemen im Vordergrund steht (Systemanalyse).

2.2.5 Gründe für Simulationen

Für den Einsatz von Simulationen nennt [Zei76] eine Reihe von Gründen. In erster Linie ist der Mensch daran interessiert, die Funktionsweise von realen oder zukünftigen Systemen zu verstehen. Naturwissenschaftler experimentieren an Modellen, um Erkenntnisse über das Originalsystem zu gewinnen sowie aufgestellte Hypothesen zu validieren oder Prognosen über das Verhalten von hypothetischen Systemen zu ermöglichen.

Zudem sind Experimente an dem Originalsystem in vielen Fällen nicht zweckmäßig oder gar nicht erst möglich. In folgenden Fällen wird daher auf die Simulation zurückgegriffen (vgl. [Sau99]):

1. Es soll das Verhalten eines Realsystems untersucht werden, das (noch) nicht existiert.
2. Experimente mit dem Realsystem wären zu gefährlich oder zu teuer.
3. Das Realsystem würde durch Experimente zerstört.
4. Es soll eine Prognose für die zukünftige Entwicklung des Realsystems erstellt werden.
5. Die Vorgänge laufen in der Realität zu schnell oder zu langsam ab.

Durch die Parametrisierung der Modelle kann die Simulation mit unterschiedlichen Parametern durchgeführt werden. Hiervon erhofft sich der Akteur, gewisse Operationen oder Prozessabläufe innerhalb des Modells optimieren oder validieren zu können. Verschiedenste Änderungen des Modells oder der Umgebungen lassen sich simulieren und seine Effekte können in Ruhe beobachtet und analysiert werden. Dies ist mit den meisten Originalsystemen nicht durchführbar.

Häufig sind Experimente am Originalsystem auch zu teuer, zu gefährlich oder sogar unmöglich. Experimente am Originalsystem sind, im Gegensatz zu Computersimulationen, nicht zerstörungsfrei. Ein Originalsystem kann nach Beendigung eines Experiments verändert vorliegen, aufgrund dessen das Experiment nicht noch einmal wiederholt werden kann. Im Bereich der Computersimulationen wird das Modell einfach wieder in den Anfangszustand zurückgesetzt und das Experiment kann wiederholt werden.

In Simulationsprogrammen hat auch die Zeit eine besondere Bedeutung. Vorgänge können in der Realität zu schnell oder gar zu langsam ablaufen. Beide Extrema führen dazu, dass ein Verhalten nicht analysiert werden kann. Die Ausführung der Modellinstruktionen liegt jedoch in der Hand des Benutzers und kann somit auf entsprechende Extremsituationen eingestellt werden.

Mit Hilfe der Simulationstechnologie können die Untersuchung und die Experimentierung an interagierenden Komponenten eines komplexen Systems realisiert werden oder es wird die Analyse auf ein Subsystem des gesamten Systems eingeschränkt.

Die gewonnen Daten einer Computersimulation lassen sich effizienter weiterverarbeiten, schneller interpretieren und sie können automatisch auf das Wichtige beschränkt und zusammengefasst werden. Das gesammelte Wissen, gewonnen aus verschiedenen

Simulationsabläufen und dem Design des Modells, ist von hohem Wert für das Originalsystem. Diese Ergebnisse helfen das Originalsystem zu verbessern, zu verstehen oder zu schützen.

2.2.6 Simulationsframework

Ein Framework besteht aus einer Menge von verschiedenen Klassen, die für eine bestimmte Aufgabe miteinander verknüpft werden. Es ist kein lauffähiges Programm, sondern es unterstützt und vereinfacht lediglich die Entwicklung von Individualsoftware. Die vorgeschriebene Architektur des Frameworks enthält unter anderem abstrakte Klassen und Schnittstellen, die der Anwender implementieren muss, damit eine lauffähige, individuelle Applikation entsteht (siehe [GHJV98]).

Ein Simulationsframework bietet eine Architektur für Simulationsprogramme an. Durch diese vorgegebene Architektur vereinfacht sich der Entwurf verschiedener Simulationsprogramme. Das Simulationsframework definiert lediglich die Basiselemente und ihre Beziehungen untereinander (vgl. 2.4), die auf die grundlegenden Systemebenen von [Kli85] zurückzuführen sind.

Die Art des Simulationsmodells führt zu simulationsspezifischen Anforderungen. Diskrete Simulatoren haben eine Reihe von gemeinsamen Anforderungen an Funktionen und Datenstrukturen, deren Implementierung durch die Art der Simulation bestimmt wird. Page verdeutlicht in [PLC00] typische Komponenten, die auch in der Software entsprechend umgesetzt werden müssen:

Zustandsvariablen: Sie repräsentieren den Zustand einzelner Modellkomponenten zu einem bestimmten Zeitpunkt. Der Zustand des gesamten Modells ergibt sich aus der Menge aller Komponentenzustände.

Simulationsuhr: Die Simulationsuhr ist eine Abstraktion der realen Zeit, die die aktuelle Simulationszeit des Modells anzeigt. Die Simulationszeit ist für viele Komponenten des Modells eine wichtige Information und muss daher leicht zugänglich sein.

Ereignisverwaltung: Jedes Ereignis, das für einen späteren Simulationszeitpunkt vorgemerkt werden soll, wird in einer zeitlich geordneten Liste geführt (Ereignisliste). Hierbei spielt die Verwaltung der Ereignisliste eine entscheidende Rolle. Neue Ereignisse müssen richtig in die Liste eingereiht werden und ausgeführte Ereignisse müssen entsprechend entfernt werden.

Mechanismus für Zustandsänderungen: Änderungen im Modell werden durch eine Prozedur realisiert, die einzelne Modellkomponenten vom aktuellen Zustand in deren Nachfolgezustand überführt.

Warteschlangen: Für viele Modelltypen können Warteschlangen interessant sein. Insbesondere bei Bedien- und Wartesystemen sind die Wartezeiten von Kunden wichtige Größen.

Zufallsgenerator: Die Erzeugung synthetischer Zufallszahlen ist ein zentrales Element der diskreten stochastischen Simulation, denn sie wird zur Modellierung zufällig eintretender Ereignisse benötigt. Um Aktivitäten untersuchen zu können, deren Ergebnisse nicht genau vorausgesagt werden können, wird ein Bereich der möglichen Ergebnisse angegeben. Somit kann der Zufallsgenerator verschiedene Ergebnisse anhand des definierten Wertebereichs generieren und zufällige Aktivitäten in die Simulation mit einbeziehen.

Initialisierungsroutine: Diese Prozedur initialisiert beim Start der Simulation das Modell und schafft somit für jeden Simulationsablauf die gleichen Voraussetzungen.

Statistische Auswertung: Ein Simulator sollte je nach Fragestellung mehrere Modellvariablen beobachten und Statistiken über diese ausgeben. Hiezu gehören statistische Größen, wie zum Beispiel Extrema, Mittelwerte und Standardabweichungen.

Reportgenerator: Die Ergebnisse der statistischen Auswertung können durch eine Reihe unterschiedlichster Routinen realisiert und ausgegeben werden. In der Regel erfolgt die Ausgabe in einer Datei und gleichzeitig am Bildschirm in tabellarischer Form oder in anspruchsvolleren Grafiken.

2.3 Multiagentensimulation

Die Multiagentensimulation bezeichnet die Ausführung eines Multiagentenmodells in einer virtuellen Umwelt (siehe [Glü05]). Das Multiagentenmodell besteht dabei aus interagierenden aktiven und passiven Einheiten, genau wie das Originalsystem. Gerade solche komplexen dezentralen Systeme stellen laut [Klü01] ein Problem für die Simulationsbranche dar und können mit traditionellen Modellierungstechniken nicht mehr adäquat abstrahiert werden.

Dies gilt zum Beispiel für die Deregulierung der Energiewirtschaft. Die komplexen Abhängigkeiten der globalen Energieindustrie (Strom, Gas, Wasser, ...) und die verstärkte Privatisierung führen immer mehr zu einem dezentralen System mit vielen individuellen Anbietern. Das System wird immer komplexer und ist als Ganzes nur noch schwer beherrschbar, was durch regelmäßige Pannen immer mehr deutlicher wird (siehe [MN05]).

Die Agententechnolgie bietet in solchen Fällen ein optimales Mittel an, um komplexe, verteilte Systeme aus autonomen und interagierenden Agenten abzubilden. Die so entstandene Multiagentensimulation wendet die Konzepte der Multiagentensysteme auf die der Simulation an.

2.3.1 Multiagentenmodell

Die Simulationsbranche hat laut Klügl [Klü01] hauptsächlich zwei Probleme, die unter Zuhilfenahme von Multiagentensystemen besser gelöst werden können:

- Modellbildung und Simulation: Bei der Modellbildung komplexer realer Systeme darf das Modell nicht zu stark vereinfacht werden, weil die Experimentierphase sonst nur triviale Antworten auf die gestellten Fragen liefert. Das Modell wird somit zu weit abstrahiert und sein Verhalten entspricht nicht mehr dem Originalsystem. Es fehlt die richtige Modellierungsmethode. In diesem Fall helfen Multiagentenmodelle. Falls interagierende autonome Einheiten im Originalsystem auftauchen, werden diese im Modell als Agenten dargestellt. Das Verhalten der individuellen Einheiten wird somit nicht wegabstrahiert, sondern bleibt durch die Agenten im Modell erhalten.
- Komplexe mathematische Modelle: Diese Modelle können durch höhere Mathematik beschrieben werden, schrecken aber viele potenzielle Simulationsanwender aufgrund der komplizierten mathematischen Mittel ab. Für viele interessante Systeme fehlt eine Standardmethode, die leicht angewendet werden kann, ohne komplexe mathematische Modelle programmieren zu müssen. Für individuenbasierte Originalsysteme bieten sich Modelle an, die ebenfalls auf Individuen basieren. Diese Abbildung ist für den Modellierer leichter verständlich und kann zu einem erhöhten Einsatz der Simulationstechnologie in den unterschiedlichsten Domänen beitragen.

Multiagentenmodelle unterscheiden sich von anderen Modellen hauptsächlich durch ihre aktiven Komponenten. Die aktiven Komponenten werden auch als *simulierter Agent* bezeichnet. Der simulierte Agent zeichnet sich durch verschiedene Merkmale aus. Die genannten Eigenschaften von Klügl [Klü01], sollen den Begriff nicht einschränken, sondern dem Modellierer als Hilfestellung dienen, um die Agenten in seinem System zu finden.

Ein (simulierter) Agent, ...

- ... verändert nicht nur sich selbst, sondern wirkt auch auf seine Umwelt ein und bleibt in dieser persistent. Aktionen treten dabei nicht nur als passive Antworten auf Umwelteinflüsse auf.
- ... agiert in Relation zu seiner Umwelt. Er verändert sie nicht nur, sondern kann auch Informationen über sie beziehen.
- ... besitzt einen beschränkten Wahrnehmungs- und Aktionsradius (Lokalität des Verhaltens).
- ... verfügt über ein nichttriviales Verhaltensreportoire. Dieses Verhalten ist auf einer höheren Ebene beschreibbar.

Für die Simulationsanwendung ist das nichttriviale Verhalten eines Agenten entscheidend. Im Gegensatz zu einem Objekt ist der Agent abstrakter gehalten, der nicht nur seinen internen Zustand kapselt, sondern auch sein Verhalten selber bestimmt und somit auch innerhalb einer Simulation die Verantwortung für sein Handeln übernimmt.

Das globale Verhalten eines Multiagentenmodells ergibt sich aus dem lokalen Verhalten aller vorhandenen Agenten, deren Interaktion untereinander und dem Einfluss der simulierten Umgebung. Die Interaktionsform der simulierten Agenten wird dabei durch die Struktur des Originalsystems bestimmt und muss somit genaustens analysiert werden, damit das Modellverhalten dem des Originalsystems entspricht.

2.3.2 Einsatzgebiete der Multiagentensimulation

Durch die stark gestiegenden Rechnerkraft werden Multiagentensimulationen in einer Vielzahl von unterschiedlichen Anwendungsgebieten möglich. Die Rechnerkraft ermöglicht groß-skalierbare Multiagentensimulationen, die vor ein paar Jahren noch nicht möglich waren.

Die Simulation von sozialen Systemen (social simulation) oder des künstlichen Lebens (artifical life), greift häufig auf Multiagentenmodelle zurück (siehe [Klü00]). Hierzu gehören nicht nur Modelle menschlicher Gesellschaften, sondern zum Beispiel auch Gesellschaftsstrukturen innerhalb der Tierwelt. Gesellschaftliche Modelle sind hochgradig selbstorganisierend, skalierbar, robust und offen. Solch eine Gesellschaft, bestehend aus einer Vielzahl von Individuen, kann idealerweise als Multiagentensystem modelliert werden. Auftretende gesellschaftliche Phänomene sollen durch die Multiagentensimulation besser verstanden und erklärt werden können.

Weitere Anwendungsklassen der Multiagentensimulation finden sich unter anderem in der Planung von Städten, Verkehrsflüssen oder der Simulation von Katastrophen sowie der Verwaltung von Versorgungsketten (supply chain management). Diese Klassen von Multiagentensimulation zeichnen sich dadurch aus, dass sie von der Simulationsbranche eingeführt wurden, um das Originalsystem besser verstehen und somit auch optimieren zu können. Das entworfene Multiagentenmodell wird jedoch nicht in der Realität eingesetzt, sondern dient nur der Analyse und hilft neue Erkenntnisse über das Original zu finden.

Durch die Weiterentwicklung der Multiagentensimulation werden Techniken, Beobachtungsmethoden und Modelle bereitgestellt, die eine tiefere Einsicht in die funktionsweise komplexer Modelle und vor allem der darunterliegenden Mechanismen ermöglichen. Die ersten Techniken wurden innerhalb der Simulationsgemeinde entwickelt und auch nur in diesem Bereich angewendet, statt zum Beispiel bei der Lösung von technischen Softwareproblemen mitzuhelfen, die bei der Entwicklung von komplexen Multiagentensystemen auftreten können.

Die Gemeinde der Agentenentwickler erstellt real einsetzbare Multiagentensysteme, die konkrete Probleme lösen sollen. Hierfür müssen große, skalierbare Systeme (Hunderte von komplexen Agenten) getestet werden. Meistens fehlt allerdings ein Grundverständnis der Simulationstechnologie, wodurch die Experimente scheitern.

Ungefähr seit dem Jahrtausendwechsel 1999-2000 (siehe [HENR03]) findet ein regerer Austausch zwischen beiden Technologien statt, damit beide Diziplinen voneinander lernen können und sowohl die Agententechnologie von der Simulationstechnolgie verbessert eingesetzt werden kann als auch umgekehrt. Somit sollen auch real einsetzbare Multiagentensysteme (Softwarelösungen) simuliert und getestet werden, wodurch sich auch der Anwendungsbereich der Multiagentensimulation vergrößert. Es werden nicht nur abstrakte Modelle entworfen und getestet, um Rückschlüsse auf das Originalsystem zu ziehen, sondern auch Multiagentensysteme simuliert, die nach einer erfolgreichen Validierung das Originalsystem selbst darstellen. Dieser Unterschied wird in Kapitel 5 deutlich. Das hier vorgestellte Multiagentensystem ist nicht nur ein Modell, um den Verkehrsfluss des Originals besser planen zu können. Das Ziel ist, ein Multiagentensystem (Multiagentenmodell) zu entwickeln, das nach mehreren Simulationsabläufen als praktisches Verkehrsleitsystem am Originalschauplatz integriert wird.

2.4 Zusammenfassung

Die Grundlagen beinhalten eine Einführung in zwei große Themengebiete. Einerseits wird die Agententechnologie vorgestellt, wobei insbesondere die Eigenschaften eines Agenten im Vordergrund stehen, genau wie das Zusammenspiel mehrerer Agenten in einem Multiagentensystem. Andererseits wird eine Einführung in das Gebiet der Simulationen geliefert, wobei dieses Gebiet auf den speziellen Teilbereich der diskreten Simulation eingeschränkt wird.

Danach wird auf den Bereich der Multiagentensimulationen eingegangen, ein erster Versuch, die beiden unterschiedlichen Technologien zu vereinen und auf die möglichen Vorteile einer solchen Vereinigung einzugehen.

Des Weiteren soll durch die Einführung in die Multiagentensimulation das Ziel der Arbeit besser verdeutlicht werden. Nicht nur die Simulationsbranche hat Vorteile durch den Einsatz der Agententechnologie. Die Methoden der Simulationsgemeinde können auch erfolgreich bei der Entwicklung von konkreten Muiltiagentensystem eingesetzt werden.

Im nächsten Kapitel werden mögliche Ansätze vorgestellt, um die Konzepte der Simulationstechnologie in eine Agentenplattform zu integrieren. Zudem wird eine allgemeine Simulationsarchitektur entworfen, die die Grundlage für spätere Implementierungen bildet.

3 Entwicklung einer allgemeinen Simulationsarchitektur

In diesem Kapitel wird eine allgemeine Simulationsarchitektur für Agentenplattformen ausgearbeitet. Zuerst werden verwandte Simulationsprogramme vorgestellt und anschließend werden die Anforderungen an die eigene Simulationsplattform definiert. Danach werden zwei verschiedene Realisierungsmöglichkeiten miteinander verglichen und eine dieser Möglichkeiten ausgewählt. Im Anschluss an die Auswahl wird die allgemeine Simulationsarchitektur entworfen, ihre einzelnen Komponenten spezifiziert und deren Zusammenspiel veranschaulicht.

3.1 Verwandte Simulationsprogramme

Es gibt eine Vielzahl von Frameworks, die eine agentenbasierte Simulation unterstützen. Diese Werkzeuge wurden in erster Linie entworfen, um den Anwender beim Modellieren von Multiagentenmodellen zu unterstützen. In Anlehnung an [GB02] gibt es mindestens zwei unterschiedliche Arten von solchen Werkzeugen, die nun vorgestellt werden sollen.

Auf der einen Seite gibt es Simulationsframeworks, die eine Bibliothek (API) von Simulationsroutinen zur Verfügung stellen. Folglich kann der Programmierer auf die Methoden der API zurückgreifen und braucht keine eigenen Routinen für die Simulation zu entwerfen. Diese Art wird zum Beispiel von folgenden Simulationsframeworks verwendet:

REPAST stellt eine Java-API bereit, damit Programmierer eigene Simulationsumgebungen entwerfen können. Zudem bietet die API Routinen an, um Agenten in sozialen Netzwerken zu erstellen und Daten von den Simulationsabläufen automatisch verarbeiten zu können. Des Weiteren wird eine Möglichkeit für die problemlose Erstellung von Benutzeroberflächen bereitgestellt (siehe [Ced02]).

Swarm bietet eine API zum Modellieren an. Hier ist die Basiseinheit der Simulation ein *swarm*, womit eine Ansammlung von Agenten gemeint ist, die einen Ablaufplan von mehreren Aktionen ausführen (siehe [Gro98]).

Diese Bibliotheken ermöglichen dem Programmierer sein eigenes Simulationsmodell zu erstellen, allerdings haben sie auch ihre Grenzen, da zum Beispiel keine eigenen Agenten ohne weiteres entworfen und in das bestehende Multiagentensystem integriert werden können. Zudem erfordert die Bibliothek ein solides Grundverständnis der verwendeten Programmiersprache, die in diesem Fall Java lautet.

Auf der anderen Seite gibt es Simulationsframeworks, die die Modellierung von einfachen Modellen durch direkte Manipulation oder "visuelle Programmierung" erlauben. Ein Modell kann durch Modellierung und einfache Befehle erstellt und ausgeführt werden, ohne dass beim Anwender eine Programmiersprache vorausgesetzt wird. Zu den bekanntesten Vertretern dieser Art gehören zum Beispiel folgende:

StarLogo ist eine programmierbare Modellierungsumgebung um das Verhalten von dezentralisierten Systemen, wie zum Beispiel Ameisenkolonien, zu erforschen. Es wurde speziell für Studenten entworfen (siehe [Sta06]).

SeSam ist ein java-basiertes Simulationsframework, indem Agentenverhalten durch grafische UML-Diagramme spezifiziert wird. Anhand einer erheblichen Anzahl von primitiven Objekten kann der Anwender eine Simulation grafisch entwerfen, ohne die Syntax der darunterliegenden Programmiersprache kennen zu müssen (vgl. [Klü01]).

Dieser Abschnitt verschafft einen Überblick über agentenbasierte Simulationsprogramme. Der Einblick soll bei den aufzustellenden Anforderungen der Simulationsunterstüzung helfen und als Wegweiser für die eigene Simulationsunterstüzung einer Agentenplattform dienen.

3.2 Anforderungsanalyse

Eine Agentenplattform mit Simulationsunterstützung muss als Erstes alle grundlegenden Dienste einer Agentenplattform bereitstellen, damit die Agenten in einer akkuraten Umgebung agieren können. Hierzu gehören vor allem ein Nachrichtentransportsystem (MTS), eine Komponente für die Verwaltung der Agenten (ADS) und ein Dienstvermittler (DF), wie in der abstrakten FIPA-Architektur gefordert wird (siehe Abbildung 2.2).

Neben diesen fundamentalen Komponenten sind zusätzliche Anforderungen nötig, damit Multiagentensysteme simuliert werden können. Gefordert wird insbesondere ein integriertes Zeitsystem. Das Zeitsystem muss die Verwaltung der System- und Simulationszeit übernehmen. Erst durch die Realisierung einer eigenen Zeitkomponente kann die Zeit auf eigene Art und Weise definiert und verwaltet werden. Die Verwaltung der Zeit bestimmt den Betriebsmodus der Agentenplattform. In dieser Arbeit werden neben dem Normalbetrieb drei Simulationsmodi gefordert (scaled real-time, event-driven und time-stepped).

Somit kann ein Multiagentensystem in verschiedenen Betriebsmodi ausgeführt werden, in denen die Zeit von einem abstrakten Zeitsystem verwaltet wird. Das Zeitsystem bestimmt, wann die nächste Aktion ausgeführt werden soll. Die Aktion selber soll nicht durch die Simulationsplattform, sondern weiterhin durch die interne Struktur des Agenten bestimmt werden. Der Agent wird in seiner Entscheidungsfreiheit, welche Aktionen

er ausführt, nicht beeinflusst.

Eine zeitlich kontrollierbare Ausführung eines Multiagentensystems reicht aber noch nicht aus, um von einer Multiagentensimulation sprechen zu können. Innerhalb der Simulation spielt die Wiederholbarkeit eines Experimentes eine entscheidende Rolle. Die Ergebnisse eines Simulationsablaufes können nur als gültig betrachtet werden, wenn sie unter gleichen Bedingungen wiederholbar sind. Zufällige Ergebnisse, die nicht wiederholbar sind, würden nicht zur Erkenntnis des zu untersuchenden Systems beitragen. So können anhand einer wiederholbaren Simulation zum Beispiel verschiedene Agentenstrategien miteinander verglichen werden.

Die dritte Anforderung bezieht sich auf die Entwicklung eines Szenarios. Multiagentensysteme laufen immer in einer Umgebung ab. Die Umgebung kann sich ständig verändern und immer wieder zu unerwarteten Zuständen führen. Um Multiagentensysteme auf solche Herausforderungen vorzubereiten, sollen sie in veränderbaren virtuellen Umgebungen getestet werden. Jeder Simulationsablauf in einer virtuellen Umgebung ist in dieser Arbeit die Ausführung eines Szenarios. Die Szenarien sollen für den Simulationsanwender eine benutzerfreundliche Möglichkeit bereitstellen, um die unterschiedlichsten virtuellen Umgebungen beschreiben zu können, in denen das entworfene Multiagentensystem getestet werden kann. Ein Szenario soll auf einfache Art und Weise durch einen Simulationsanwender erstellbar sein und die technischen Anforderungen sollen dabei so gering wie möglich gehalten werden.

Die drei zuvor genannten Anforderungen sind essenziell für die Bereitstellung einer Simulationsunterstützung. Darüber hinaus soll zusätzlich eine komfortable Administratormöglichkeit für den Simulationsanwender bereitgestellt werden. Der administrative Bereich soll die Verwaltung der Simulation ermöglichen. Diesbezüglich wird eine grafische Oberfläche gefordert, um die Art des Simulationsbetriebes auswählen und einzelne Simulationsabläufe verwalten zu können. Des Weiteren sind eine Vielzahl von Simulationsoptionen denkbar, wie zum Beispiel der manuelle oder automatische Betrieb und die Einstellung der Ausführungsgranularität.

Da die Simulationsplattform durch eine Vielzahl weiterer Komponenten ausgebaut werden kann, die jedoch nicht zwingend benötigt werden, wird ebenfalls die Bereitstellung einer Schnittstelle gefordert. Die Schnittstelle soll eine zukünftige Erweiterung der Simulationsplattform erleichtern, indem sie einen allgemeinen Zugriff auf die Informationen eines jeden Simulationsablaufes bietet.

Nachdem die allgemeinen Anforderungen an die Simulationsunterstützung dargelegt wurden, wird im nächsten Abschnitt auf verschiedene Möglichkeiten eingegangen, wie

man die Simulationsunterstützung innerhalb einer Agentenplattform am besten umsetzen kann.

3.3 Arten der Simulationsunterstützung

Eine Agentenplattform kann im Grunde auf zwei verschiedene Arten mit einer Simulationsunterstützung ausgestattet werden. Beide Varianten haben eine koordinierbare Ausführung der Agentenaktionen zum Ziel. Der erste Ansatz steuert die Agentenaktionen mit einem zentralen Koordinator, der in der Anwendungsebene implementiert wird. Der zweite Ansatz befasst sich hingegen mit einer Lösung im Kern der Agentenplattform. Dieses Unterkapitel beschreibt und vergleicht beide Ansätze und veranschaulicht ihre Vor- und Nachteile. Anhand dieser wird ein Ansatz ausgewählt, der im weiteren Verlauf der Arbeit verwendet wird.

3.3.1 Simulationsunterstützung auf Anwendungsebene

Der erste Ansatz beschäftigt sich mit der Implementierung einer Simulationsunterstützung auf Anwendungsebene. In [BPL+06b] wird eine zentrale Middleware-Komponente vorgeschlagen, ein Dienst, der die Kontrolle über den Prozessablauf übernimmt und somit verantwortlich für die Synchronisation der einzelnen Agentenaktionen in Bezug auf die globale Zeit ist. Der Dienst wird als FIPA-konformer Agent realisiert, da er so auf natürliche Art und Weise optimal in die Umgebung eines Multiagentensystems passt und daher eine Simulationsunterstützung für Agentenplattformen entworfen werden kann, ohne die Plattform selbst verändern zu müssen. Dieser Agent fungiert als ein zentraler Koordinator.

Der Koordinator ist verantwortlich für eine wiederholbare Ausführung der Simulation. Er nutzt eine globale Liste von Zeitpunkten, um die Aktivitäten der Agenten zu planen. Agenten, die an dem Simulationsvorgang teilnehmen möchten, müssen sich zunächst bei dem Koordinator registrieren. Sie können sich aber jederzeit wieder deregistrieren. Somit muss der Koordinator sehr flexibel sein, da sich ständig neue Agenten an- und abmelden können. Sollte ein Fehler innerhalb eines Agenten auftreten und dieser für den Koordinator nicht mehr funktionsfähig sein, darf nicht die ganze Simulation zusammenbrechen. Der Koordinator muss robust genug sein, um mit solchen Ereignissen umgehen zu können.

Agenten, die sich beim Koordinator registriert haben, melden jede ihrer Aktionen an und warten auf eine Bestätigung seinerseits, dass sie ihre Aktion ausführen dürfen. Der Koordinator erhält somit eine Vielzahl an Nachrichten, wann welcher Agent eine Aktion ausführen möchte. Er benachrichtigt stets den Agenten, der am Anfang seiner globalen Zeitliste steht und wartet, bis die Aktion des Agenten ausgeführt wurde. Dann springt die Uhr zum nächsten Zeitpunkt und es wird der nachfolgende Agent benachrichtigt. Natürlich ist aus Gründen der Robustheit auch ein Timeout für Nachrichten vorgesehen.

Falls der Agent keine Bestätigung bezüglich der von ihm auszuführenden Aktion innerhalb einer bestimmten Zeit sendet, wird ein Fehler angenommen und der Koordinator schreitet zum nächsten Zeitpunkt fort.

Der beschriebene Protokollablauf entspricht prinzipiell einer synchronen Kommunikation. Der oben genannte Ansatz zur Simulationsunterstützung vermeidet zwar Deadlocks und zu viele Nachrichtensendungen an den zentralen Koordinator, muss aber teilweise eine asynchrone Kommunikation zulassen, um die Autonomie der Agenten gewährleisten zu können. Die Autonomie der individuellen Agenten gehört zu den wichtigsten Eigenschaften der Agentenphilosophie. Agenten können aufgrund von Veränderungen in der Umwelt oder anhand von Informationen, die sie durch andere Agenten erhalten, ihre eigene Weltanschauung verändern. Somit können bereits angekündigte Aktionen eines Agenten revidiert werden, indem der Agent den Koordinator benachrichtigt und dieser den Zeitpunkt der widerrufenen Agentenaktion entfernt. Um diese Freiheitsmöglichkeiten in das Synchronisationsprotokoll mit einzubeziehen, können Agenten, die aufgrund anderer Agentenaktionen tätig werden, den Koordinator benachrichtigen und die eigene Aktionsausführung beanspruchen. Festzuhalten bleibt, dass Agenten immer nur eine Aktion ausführen, wenn der Koordinator sie benachrichtigt oder wenn andere Agentenaktionen sie auslösen und der Koordinator diese zwischenzeitliche Unterbrechung genehmigt.

Dieser leichtgewichtige Ansatz ermöglicht wiederholbare Simulationsabläufe und zeichnet sich vor allem durch seine Einfachheit, Interoperabilität und einen agentenfreundlichen Mechanismus aus. Allerdings kann die zentralisierte Architektur einen Engpass aufweisen und einen *single point of failure* darstellen, da der Koordinator für die Verarbeitung aller Nachrichten verantwortlich ist. Zudem muss jeder Agent das Synchronisationsprotokoll und seine Zeitaspekte implementieren, wodurch der Anwendungsentwickler spezifische Simulationsanforderungen erfüllen muss, um die Simulationsunterstützung nutzen zu können.

3.3.2 Simulationsunterstützung auf Plattformebene

Der zweite Ansatz setzt auf der Plattformebene an. Anstatt die Anwendungsebene mit einer speziellen Komponente auszustatten, soll die Plattform so verändert werden, dass dem Anwendungsprogrammierer die Simulationsunterstützung durch die Agentenplattform bereitgestellt wird. Aus diesem Grund muss sich der Anwendungsprogrammierer nicht um simulationsspezifische Implementierungen kümmern, die durch die Nutzung der Simulationskomponente entstehen. Multiagentensysteme können mit Hilfe der Agentenpllattform weiterhin so entworfen werden wie bereits vor der Entwicklung der integrierten Simulationsunterstützung.

Das System wird also auf niedrigster Ebene verändert. Hierfür ist ein gutes Verständnis der Agentenplattform erforderlich, um die entsprechenden Basiskomponenten des Systems verändern zu können. Entscheidend ist die Änderung der Ausführungskomponente. Die Ausführungskomponente wird vom *Agent Management System* genutzt, um Agentenaktionen durchzuführen. Diese Komponente muss für die Simulationsunterstützung um verschiedene Betriebsmodi erweitert werden. Ein Multiagentensystem soll im Normalbetrieb oder im Simulationsbetrieb ausgeführt werden können, wobei der Simulationsbetrieb wiederum unterschiedliche Simulationsmodi bereitstellt. Damit die Ausführungskomponente eine wiederholbare Ausführung gewährleisten kann, wird wie in gängigen Simulationprogrammen (siehe Abschnitt 3.1), nur ein Ausführungsprozess implementiert und das Multiagentensystem somit sequentiell ausgeführt.

Die Ausführungskomponente arbeitet immer mit einer Zeitkomponente zusammen, um wiederholbare Simulationen in unterschiedlichen Zeitsystemen durchführen zu können. Nur aufgrund der Simulationsunterstützung wird eine alleinstehende Zeitkomponente benötigt. Die zeitliche Ausführung spielt eine zentrale Rolle in der Simulation. Je nach Simulationsmodus schreitet die Zeit auf unterschiedliche Art und Weise voran und es werden nur bestimmte Aktionen ausgeführt. Die Zeitkomponente bestimmt, wann eine Aktion ausgeführt wird, delegiert die Ausführung aber an eine eigenständige Ausführungskomponente, die alle aktuell auszuführenden Agentenaktionen nacheinander abarbeitet.

Der Simulationsbetrieb erfordert noch eine letzte Komponente, die Szenariokomponente. Diese bildet im Gegensatz zu den beiden anderen eine Schnittstelle zwischen dem Simulationsanwender und der Simulationsausführung. Anhand der Szenariokomponente kann der Simulationsanwender die Simulation initialisieren und steuern. Der Simulationsanwender muss in einer festgelegten Metasprache ein Multiagentenszenario definieren. Dieses ist erforderlich für die Initialisierung eines Simulationsablaufes und wird im weiteren Verlauf der Arbeit noch veranschaulicht.

Die drei Komponenten bilden die minimalen Anforderungen an eine Agentenplattform mit Simulationsunterstützung und können noch durch zusätzliche Komponenten erweitert werden, wie zum Beispiel durch eine Komponente zur Analyse der Simulationsergebnisse. Für die Realisierung dieser Architektur ist ein hoher, einmaliger Implementierungsaufwand der Simulationsunterstützung notwendig.

An dieser Stelle wird die Agentenplattform JACK[1] erwähnt, da für diese Agentenplattform bereits eine Simulationsunterstützung in Form von *JACK Sim* entworfen wurde. Laut [Ltd05] ist *JACK Sim* ein Framework für den Entwurf und die Ausführung von wie-

[1] Weitere Informationen unter http://www.agent-software.com/.

derholbaren, agentenbasierten Simulationen, wobei das Agentenverhalten durch JACK oder *JACK Teams* realisiert werden muss. *JACK Sim* besteht aus drei grundlegenden Komponenten: Eine Komponente für das zu simulierende Multiagentenmodell, eine weitere für die Verwaltung der Zeit und eine Komponente für die Visualisierung der Modellausführung.

Allerdings wurde der Ansatz nicht vollständig auf Plattformebene umgesetzt. Die Zeitkomponente basiert unter anderem auf speziellen Zeit-Agenten die für eine wiederholbare Ausführung verantwortlich sind. Die drei genannten Komponenten werden für den Anwender nur teilweise transparent gehalten. Der Anwender steht immer noch in der Verantwortung ein Multiagentensystem mit speziellen Zeit-Agenten auszustatten. Danach können alle Agenten eines Multiagentensystems simuliert werden, die sich bei der Zeitkomponente registrieren und die eine spezielle Schnittstelle sowie eine Zeit-*Capability* implementieren (siehe [Ltd05], S. 9).

Im nächsten Abschnitt wird ein direkter Vergleich zwischen beiden Ansätzen gezogen und ein Ansatz ausgewählt.

3.3.3 Zusammenfassung und Auswahl eines Ansatzes

Die Implementierung auf Anwendungsebene ist eine schnelle und probate Lösung. Sie ist agentenfreundlich und zeichnet sich, wie bereits erwähnt, durch ihre Einfachheit und Interoperabilität aus. Insbesondere die Interoperabilität spricht für diesen Ansatz. Agenten verschiedenster Plattformen können durch das Simulationsprotokoll miteinander über Gateway-Agenten verbunden werden und an einer gemeinsamen Simulation teilnehmen. Bei der Implementierung auf Plattformebene ist diese Möglichkeit nur gegeben, wenn alle beteiligten Agentenplattformen mit einer Simulationsunterstützung auf Plattformebene ausgestattet und zusätzlich durch Gateway-Agenten verbunden werden.

Nachteile der Implementierung auf Anwendungsebene entstehen hauptsächlich durch die zentrale Stellung des Koordinators und durch das Synchronisationsprotokoll zur Simulationsausführung. Der Koordinator ist für die Verarbeitung aller Synchronisationsnachrichten verantwortlich und kümmert sich um die Einhaltung des Protokollablaufes. Der Anwendungsentwickler muss alle Agenten, die simuliert werden sollen, mit dem Synchronisationsprotokoll ausstatten. Dieser Aufwand ist bei einem Multiagentensystem mit n Agenten ungefähr n mal so hoch wie die einmalige Entwicklung eines Multiagentenszenarios für die Simulation auf Plattformebene. Des Weiteren muss der Simulationsanwender keine Erfahrung mit der Entwicklung von speziellen Multiagentensystemen haben. Diese Arbeit kann getrennt von dem Anwendungsentwickler durchgeführt werden.

Der Ansatz auf Plattformebene verlangt zudem keine doppelte Realisierung der Agen-

ten. Der Simulationsanwender kann das Verhalten des Agenten mittels Simulationsunterstützung simulieren, untersuchen und denselben Agenten in der später fertiggestellten Softwareversion einsetzen. Die Simulationsunterstützung auf Anwendungsebene erfordert sowohl die Entwicklung eines Simulationsagenten, der das Synchronisationsprotokoll implementiert und nur innerhalb von Simulationen eingesetzt wird, als auch eine weitere Version dieses Agenten, der in einem normal betriebenen Multiagentensystem eingesetzt werden kann.

Aufgrund der überwiegenden Vorteile für den Ansatz auf Plattformebene wird in dieser Arbeit eine Simulationsunterstützung auf Plattformebene realisiert. Die in diesem Ansatz vorliegende komponentenbasierte Architektur ist verständlich und austauschbar und lässt sich gut weiterentwickeln. Des Weiteren können Fehler im Kern des Systems früher abgefangen und verarbeitet werden. Die Ausführung auf Anwendungsebene wäre aufgrund des Nachrichtenversands anfälliger für Fehler und auch ineffizienter. Darüber hinaus ist ein schneller Wechsel zwischen den einzelnen Betriebsmodi nur im Systemkern realisierbar.

Im weiteren Verlauf dieses Kapitels werden die grundlegenden Konzepte einer Simulationsarchitektur für Multiagentensysteme vorgestellt, die die zuvor definierten Anforderungen an eine Simulationsunterstützung auf Plattformebene erfüllen sollen. Die Konzepte werden so allgemein wie möglich gehalten, damit sie auf eine möglichst große Klasse von Agentenplattformen anwendbar sind.

3.4 Simulationsarchitektur

Im Folgenden Abschnit werden die Konzepte für eine Agentenplattform mit Simulationsunterstützung erstellt. In Abbildung 3.1 ist eine allgemeine Architektur mit typischen Simulationskomponenten abgebildet. Der Simulationsanwender nimmt über die Administratorkomponente Einfluss auf die Simulationsplattform. Besonders die Szenariokomponente unterliegt der Eingabe des Anwenders. Hier wird ein benutzerdefiniertes Szenario geladen und mit Hilfe des speziellen Agenten *Scenario Mananger* (SCM) ausgeführt.

Der Betriebsmodus wird durch die Zeitkomponente bestimmt und die wiederholbare Simulationsausführung wird durch das Zusammenspiel der Ausführungs- und der Zeitkomponente gewährleistet. Neben diesen Hauptkomponenten gibt es des Weiteren noch die Historienkomponente, die für die Aufzeichnung der Simulationsaktivitäten verantwortlich ist, und weitere optionale Komponenten, wie zum Beispiel eine Komponente für die automatische Analyse der Historiendaten. Im weiteren Verlauf dieses Kapitels werden alle Komponenten aus Abbildung 3.1 ausführlich vorgestellt.

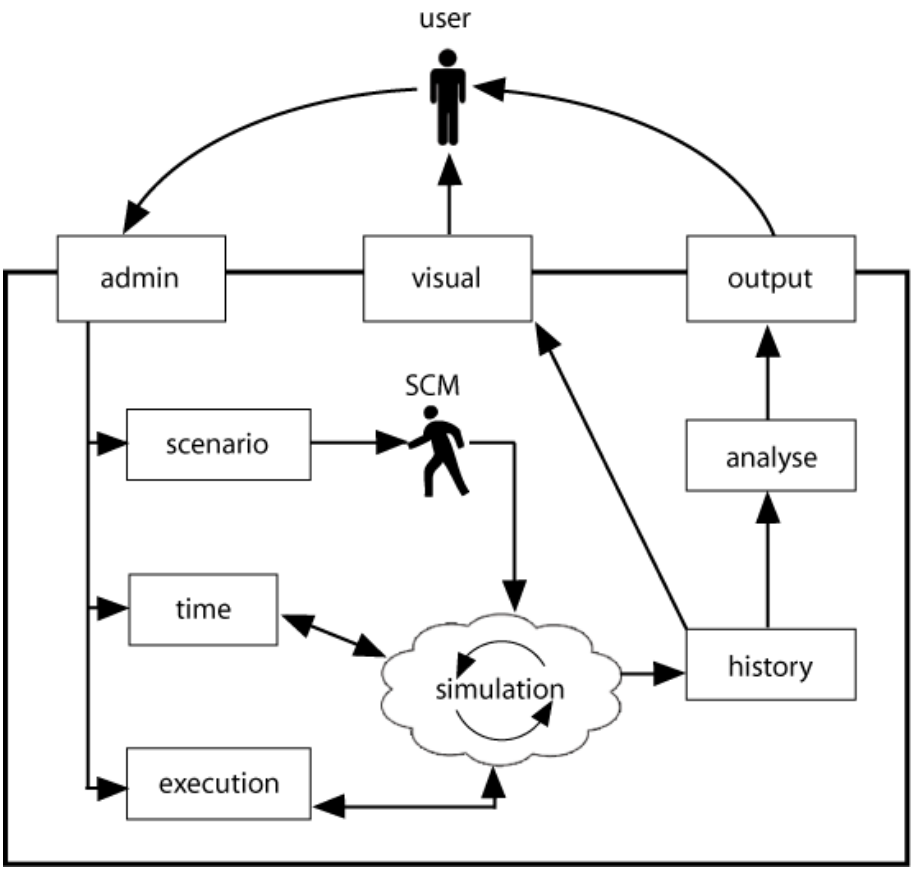

Abbildung 3.1: Zusammenspiel der Simulationskomponenten

3.4.1 Ausführungskomponente

Agentenplattformen haben nicht nur die Basiskomponenten der abstrakten FIPA-Architektur (siehe Abbildung 2.2) gemeinsam, sondern auch eine Ausführungskomponente. Gemeint ist ein Bestandteil des Systems, das sich zur Laufzeit um die Ausführung der Agentenaktionen kümmert. Diese Ausführungskomponente liegt im Fokus der Simulationsinfrastruktur. Läuft ein Multiagentensystem in der Laufzeitumgebung der Agentenplattform ab, führt die Ausführungskomponente die Agentenaktionen zu bestimmten Zeiten durch. Die Aktionen können seriell oder parallel ausgeführt werden. Wird die Agentenplattform um eine Simulationsunterstützung erweitert, muss vor allem die Ausführungskomponente ausgebaut werden. Anstatt die Agenten nur im Normalbetrieb ausführen zu können, werden zusätzliche Simulationsbetriebe gefordert.

Die Ausführungskomponente kann auf einem oder auf mehreren Prozessen basieren. Für die in dieser Arbeit entwickelte Simulationsunterstützung wird ein serieller Betrieb durch einen *Thread*[2] gewährleistet. Auch der Normalbetrieb des Multiagentensystems basiert auf einem Thread. Somit ist die Wiederholbarkeit eines Simulationsablaufs nur von einer Instanz der Ausführungskomponente abhängig und Multiagentensysteme, die im

[2]Ein leichtgewichtiger Prozess.

Normalbetrieb der Simulationsplattform ausgeführt werden, können keine Anomalien aufweisen, die nicht auch im Simulationsbetrieb beobachtet werden können. Abbildung 3.2 stellt den groben Programmablauf der Ausführungskomponente in einem Flussdiagramm dar.

Ein Ausführungsmodus, der auf mehr als einem Thread basiert, würde den Programmablauf der Ausführungskomponente parallel bearbeiten. Dies kann zwar zu einer Leistungsoptimierung bei entsprechender Hardware führen, verstößt jedoch gegen die Wiederholbarkeitsanforderung eines Simulationsablaufs. Als Lösung könnte man zwar den parallen Betrieb von Agenten synchronisieren, dies würde aber die Vorteile der parallelen Ausführung mindern.

Die Standard-Ausführungskomponente wird beim Starten der Agentenplattform erzeugt, genau wie die dazu passende Zeitkomponente. Je nach Betriebsmodus der Plattform kann sich die Ausführungskomponente in speziellen Teilen unterscheiden. Ihre Hauptaufgabe, eine wiederholbare Ausführung von Agentenaktionen, gilt aber für alle speziellen Ausführungsarten. In Abbildung 3.2 wird der dazu benötigte allgemeine Programmablauf dargestellt, der auf alle Spezialisierungen zutrifft.

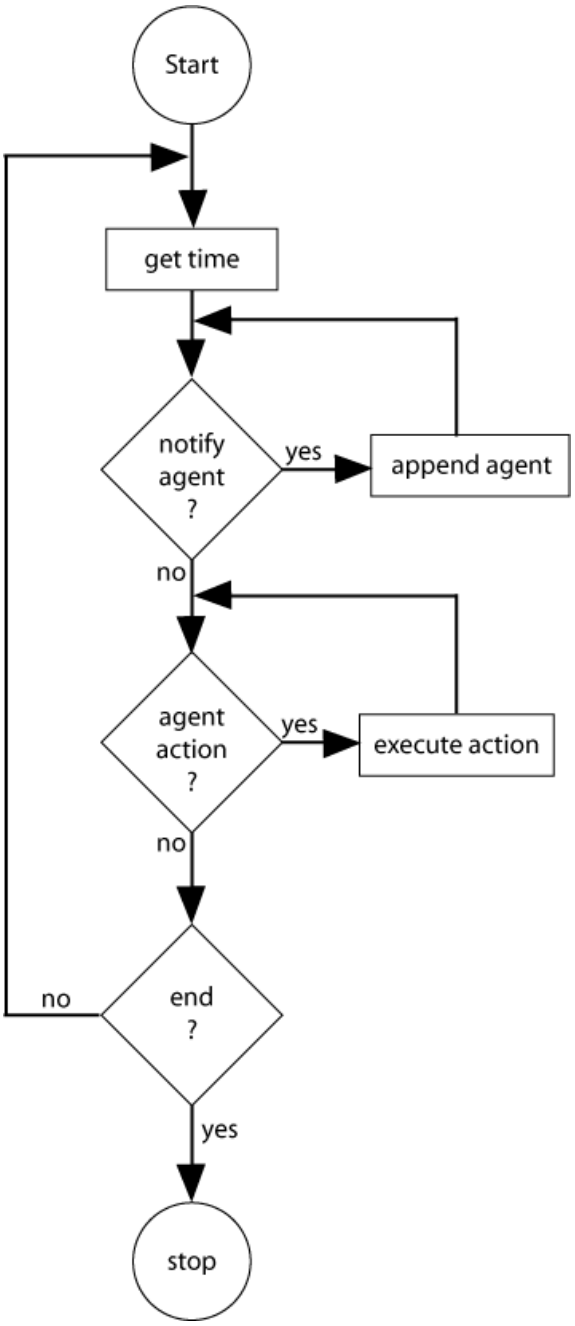

Abbildung 3.2: Programmablauf der Ausführungskomponente

Der Thread der Ausführungskomponente geht nach seiner Erzeugung sofort in die Hauptschleife über und durchläuft diese so lange, bis der Simulationsablauf gestoppt wird. In der ersten Aktion der Hauptschleife (get time) wird die Zeit der Agentenplattform abgefragt, wobei der genaue Zeitstempel durch die Zeitkomponente bestimmt und zurückgegeben wird. Der Wert des Zeitstempels bestimmt, ob ein Agent oder mehrere Agenten mit der Ausführung einer Aktion an der Reihe sind (notify agent?). Agenten, die eine Aktion ausführen dürfen, werden in einer wiederholbaren Reihenfolge benachrichtigt und in eine Ausführungswarteschlange eingereiht (append agent). Agenten, die zum selben Zeitpunkt eine Aktion ausführen wollen, werden immer in der gleichen Rei-

henfolge benachrichtigt. Falls Agent *a* vor Agent *b* erzeugt wird, wird bei gleichem Zeitstempel immer auch Agent *a* vor Agent *b* benachrichtigt. Somit ist die Wiederholbarkeit nicht gefährdet. Im nächsten Schritt wird die Ausführungswarteschlange abgearbeitet (agent action?). Agenten, die sich in dieser Warteschlange befinden, dürfen genau eine Agentenaktion ausführen (execute action). Falls der Simulationsablauf nach dieser Abarbeitung immer noch nicht zu Ende ist (end?), wird die Hauptschleife erneut durchlaufen.

3.4.2 Zeitkomponente

Die Ausführungskomponente ist für die wiederholbare Ausführung von Agentenaktionen verantwortlich. Hierfür benötigt sie die aktuelle Plattformzeit, die durch die Zeitkomponente verwaltet wird. Die Zeitkomponente bestimmt den genauen Zeitstempel des Anwendungsprogramms und gibt diesen auf Anfrage zurück. Im Normalbetrieb wird die Systemzeit zurückgegeben. In den Simulationsmodi wird die Fortschreitung der Zeit hingegen auf eine abstraktere Art bestimmt. Je nach Simulationsverfahren schreitet die Zeit auf unterschiedliche Art und Weise voran. Hierfür benötigt die Agentenplattform eine weitere Präsentation der Zeit, die Simulationszeit. Die Simulationszeit wird je nach Simulationsbetrieb auf unterschiedliche Art und Weise verwaltet.

In dieser Arbeit werden drei verschiedene Simulationsmodi implementiert, wie aus Abbildung 3.3 zu entnehmen ist. Hierbei gilt, dass ein Simulationsmodus im Grunde aus dem Zusammenspiel eines konkreten Zeit- und Ausführungssystems resultiert. Im folgenden Abschnitt werden die drei unterschiedlichen Zeitsysteme vorgestellt.

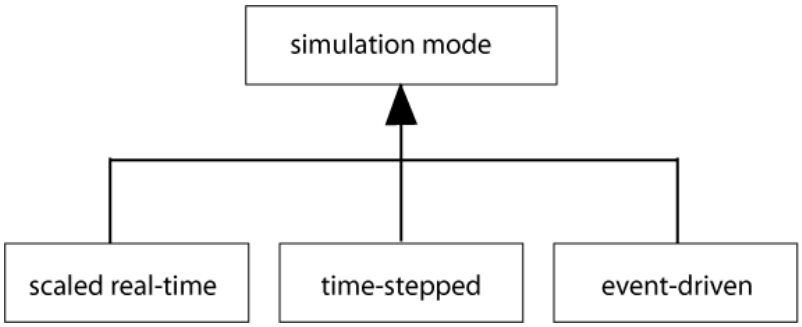

Abbildung 3.3: Simulationsmodi der Zeitkomponente

Im skalierten Zeitsystem (scaled real-time) entspricht die Simulationszeit einer skalierten Systemzeit. Der Normalbetrieb ist ein Spezialfall des skalierten Zeitsystems, bei dem der Skalierungsfaktor *eins* ist, und wodurch folglich die Systemzeit zurückgegeben wird.

Die Berechnung der Simulationszeit findet immer zu diskreten Zeitpunkten statt. Ein Skalierungsfaktor, der größer als *eins* ist, lässt die Zeit und somit auch die Simulation immer schneller voranschreiten, je größer der Faktor wird. Ist der Skalierungsfaktor kleiner als *eins*, wird die Zeit und somit auch die Simulation langsamer, je näher der Faktor gegen *null* geht. Ein Skalierungsfaktor von *null* würde dem Pausieren der Simulation ent-

sprechen, was hier aber auf eine andere Art gelöst werden soll. Folglich wird der Wert null, genau wie negative Skalierungsfaktoren, nicht zugelassen.

In dem zeitgesteuerten Zeitsystem (time-stepped) schreitet die Zeit in benutzerdefinierten Zeitschritten voran. Der Simulationsbenutzer definiert am Anfang der Simulation einen Zeitschritt, der innerhalb eines Simulationsablaufes verändert werden kann. Die Zeit springt so schnell wie möglich (as-fast-as-possible) von einem zum nächsten Zeitpunkt. In der Regel gilt für den zeitgesteuerten Simulationsmodus, dass ein Zeitsprung mehr als eine Agentenaktion auslöst, im Gegensatz zu dem ereignisgesteuerten Simulationsmodus, in dem immer nur genau eine Agentenaktion pro Zeitsprung ausgeführt wird.

Das ereignisgesteuerte Zeitsystem (event-driven) zeichnet sich durch eine ereignisgesteuerte Ausführung aus. Alle zukünftigen Ereignisse werden anhand ihrer Zeitstempel in eine globale Warteschlange einsortiert. Die Warteschlange wird so schnell wie möglich (as-fast-as-possible) abgearbeitet, indem immer das erste Ereignis entfernt und für die Ausführung vorbereitet wird. Zur Vorbereitung gehört nur, die Simulationszeit auf den Zeitstempel des Ereignisses zu setzen. Danach kann das Ereignis ausgeführt werden. Dieser Ablauf wird so lange wiederholt, bis die Warteschlange leer ist. Demzufolge ist die ereignisgesteuerte Implementierung der Zeitkomponente nicht nur für die Zeitverwaltung verantwortlich, sondern sie muss sich auch um eine Verwaltung der Ereignisse kümmern.

3.4.3 Szenariokomponente

Damit ein Multiagentensystem simuliert werden kann, muss eine Agentenplattform mit einer Szenariokomponente ausgestattet werden. Die Simulation erfordert eine Simulationsumgebung, in der das Multiagentensystem getestet wird. Diese Simulationsumgebung wird durch den Simulationsanwender spezifiziert, damit das Multiagentensystem in unterschiedlichen Szenarien getestet werden kann.

Die Simulationsumgebung wird durch die Szenariokomponente verwaltet. Zu den Verwaltungsaufgaben gehört das Initialisieren, Stoppen und Ausführen eines Szenarios, wobei jedes Szenario eine individuelle Simulationsumgebung beschreibt. Der Simulationsanwender definiert Szenarien (Multiagentenszenarien) in einer deskriptiven Sprache und die Szenariokomponente setzt mit Hilfe des FIPA-konformen SCM-Agenten die Simulationsumgebung um.

Multiagentenszenario

Ein Multiagentensystem zeichnet sich vor allem durch mehrere Agenten aus, die sich in einer Umgebung befinden. Dabei befindet sich ein Agent in ständiger Interaktion mit seiner Umwelt, um auf Änderungen in dieser reagieren zu können. Die Umgebung kann das

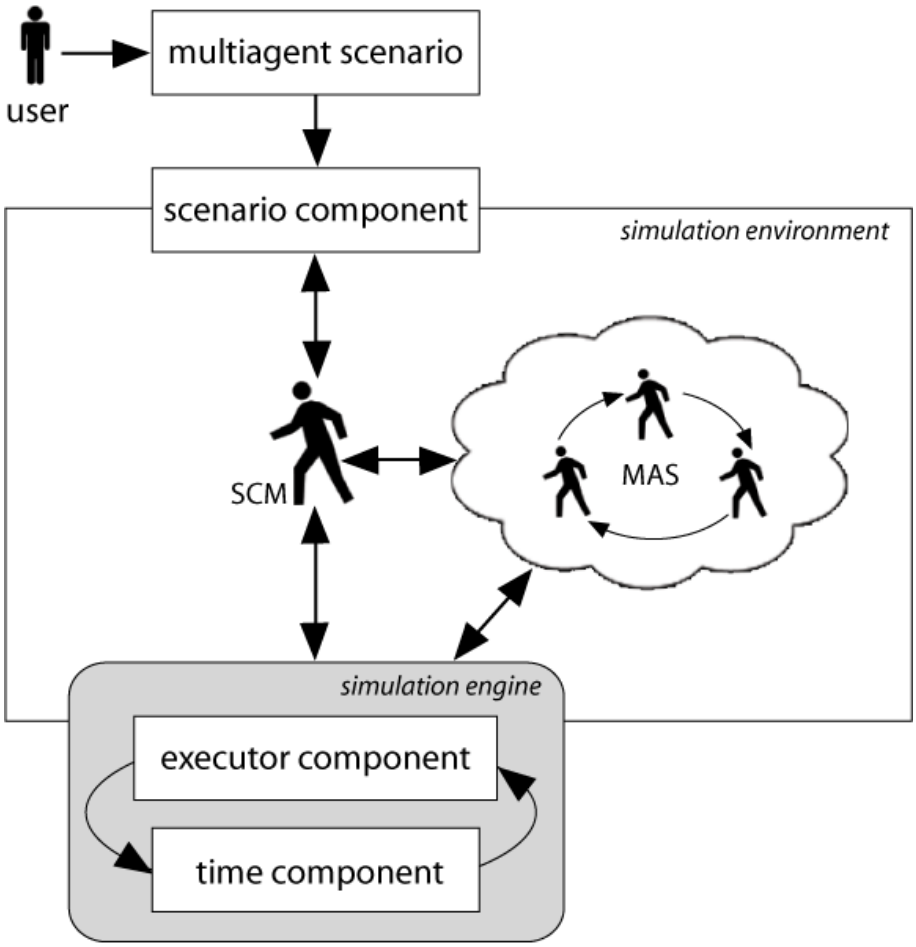

Abbildung 3.4: Grundlegender Simulationsablauf

Verhalten der Agenten beeinflussen, genau wie Agenten Änderungen in der Umgebung hervorrufen können. Die subjektive Umgebung eines Agenten setzt sich aus allen anderen Agenten sowie weiteren externen Objekten zusammen, die Einfluss auf den Agenten nehmen können, ohne das sich der Agent dessen bewusst ist.

Die Simulation von Multiagentensystemen benötigt eine steuerbare Simulationsumgebung, die es ermöglicht, das Verhalten der Agenten unter verschiedensten Bedingungen genaustens zu testen und zu analysieren. Der Simulationsanwender spezifiziert die Simulationsumgebung durch die Beschreibung eines Szenarios. In dem Szenario wird angegeben, welche Agenten für einen Simulationsablauf gebraucht werden, wie diese Agenten initialisiert werden und ob das beschriebene Multiagentensystem durch definierte Ereignisse beeinflusst werden soll (durch den SCM). Somit können unterschiedlichste Multiagentenszenarien entwickelt werden, um einzelne Simulationsabläufe in einem Multiagentensystem durchführen zu können. Die Beschreibung eines Multiagentenszenarios stellt folglich eine Menge von Informationen und Anweisungen bereit, die eine Szenariokomponente für die Ausführung einzelner Simulationsabläufe benötigt. Die Szenariokomponente ist somit ein Teil der Umwelt aller Agenten, die jedoch nur für die Simulation benötigt wird. Wird das Multiagentensystem in der ”Realität” eingesetzt,

übernimmt die reale Umgebung die Rolle der Simulationsumgebung.

Entwicklung eines Multiagentenszenarios

Die Simulationsunterstützung für Agentenplattformen ist auf Plattformebene realisiert. Ein Grund dafür ist, die Klasse der simulierbaren Multiagentensysteme so groß wie möglich zu halten, ohne den Simulationsanwender mit spezifischen Simulationsanforderungen zu überhäufen.

Die Simulationsarchitektur dieser Arbeit erfordert vom Simulationsanwender die Spezifikation eines Multiagentenszenarios nach einer bestimmten Struktur. Um die Entwicklung eines Multiagentenszenarios so benutzerfreundlich wie möglich zu halten und somit eine Vielzahl von Anwendern aus interdiziplinären Bereichen das Simulieren zu ermöglichen, werden folgende Anforderungen an den Entwicklungsprozess eines Multiagentensystems gestellt.

Die Entwicklung eines Multiagentenszenarios soll...

1. ... kaum spezifische Vorkenntnisse verlangen.
2. ... mit geringem Zeitaufwand möglich sein.
3. ... plattformunabhängig sein.
4. ... flexibel und erweiterbar sein.

Aufgrund dieser Anforderungen liegt es nahe, die Beschreibung eines Multiagentenszenarios mittels XML[3] durchzuführen. Die Struktur ist meist selbsterklärend und erfordert kaum technisches Grundwissen vom Simulationsanwender, um XML benutzen oder lernen zu können. Eine XML-Datei kann mit beliebigen Editoren erstellt werden und nach einer kleinen Einführung lassen sich in kürzester Zeit die unterschiedlichsten Szenarien definieren. Demnach wären alle vier Aspekte der Anforderungen gewährleistet, da laut [Con05] (Punkt eins) XML auch "herkömmliche Fallen vermeidet, wie sie in anderen Sprachkonstruktionen auftreten: XML ist erweiterbar, plattformunabhängig und unterstützt Internationalisierung / Lokalisierung und Unicode".

Vom deklarativen zum ausführbaren Szenario

Das deklarativ spezifizierte Multiagentenszenario (XML), wird durch die Szenariokomponente in ein Format umgewandelt, das der SCM (siehe Abbildung 3.4) ausführen kann (mittels einer imperativen Sprache). Hierfür muss ein geeigneter XML-Parser bereitgestellt werden, der die enthaltenen Informationen des XML-Dokumentes in ein entsprechendes Laufzeitobjekt transformiert, ohne dass wichtige Informationen verloren gehen.

[3] Weitere Informationen unter http://www.w3.org/XML/.

Der Parser wird von der Agentenplattform bereitgestellt und muss dem Simulationsanwender nicht vertraut sein. Der Simulationsanwender muss nur eine wohlgeformte und gültige XML-Datei entwickeln, die beim Start eines Simulationsablaufes automatisch geparst, transformiert und ausgeführt werden kann.

Damit die Modellierung der semi-strukturierten Daten vom Parser problemlos abgearbeitet werden kann, wird eine XML-Schema-Definition benutzt. So wird nicht nur die Struktur von XML-Dokumenten spezifiziert, sondern es kann auch der Inhalt von Elementen und Parametern festgelegt werden. Die XML-Datei (Beschreibung des Szenarios) wird im weiteren Verlauf dieser Arbeit als *Scenario Definition File* (SDF) bezeichnet. Ein SDF kann in allen bereitgestellten Simulationsmodi ohne Einschränkung ausgeführt werden.

Technisch betrachtet besteht die SDF-Datei aus einer Beschreibung von Agenten und Ereignissen. Beim Starten der Simulation werden die spezifizierten Agenten des SDF erzeugt und entsprechend angegebener Parameter mit einem initialen Zustand versehen. Die Erzeugung der Agenten entspricht der Abarbeitungsreihenfolge der Baumstruktur des XML-Dokumentes. Die Einhaltung der Reihenfolge ist wichtig für die Wiederholbarkeit der Simulationsabläufe, da so die Agentenaktionen, entsprechend dieser Initialisierungsreihenfolge und dem jeweiligen Zeitstempel der Aktion, ausgeführt werden können (siehe Abbildung 3.4.1).

Multiagentenszenarien können mit Ereignissen ausgestattet sein, müssen es aber nicht. Die Ausstattung ist abhängig von dem entworfenen Multiagentensystem. Es gibt Agenten, die automatisch weitere Ereignisse hervorbringen. Andere sind wiederum passiv, bis ein bestimmtes Ereignis eintritt, auf dass sie gewartet haben. Je nach Art der Agenten und des gewünschten Szenarios ist es dem Simulationsanwender überlassen wie er das Multiagentensystem mit Ereignissen beeinflussen möchte. Mit *Ereignissen* sind an dieser Stelle ausschließlich Nachrichten gemeint. Die Nachrichten werden nach den Beschreibungen des SDF vom SCM dynamisch generiert und zu den gewünschten Zeitpunkten verschickt. Hierbei können Nachrichten jedoch nur an zuvor definierte Agenten verschickt werden. Dieser Nachrichtenversand spiegelt somit Ereignisse einer Umwelt wieder, auf die die Agenten reagieren können.

3.4.4 Optionale Komponenten

Die bereits vorgestellten Komponenten bilden die Grundlage einer Simulationsunterstützung für Agentenplattformen. Sie sind zwingend erforderlich für einen Simulationsbetrieb. Allerdings gibt es noch weitere Komponenten, die für diese Arbeit nicht zwangsläufig erforderlich sind, aber in der Simulationstechnologie als typische Komponenten bezeichnet und von einer Simulationssoftware gefordert werden (nach [PK05]). In diesem Abschnitt werden die optionalen Komponenten kurz vorgestellt und es wird auf

eine Integration in die bestehende Simulationsarchitektur eingegangen.

Analysekomponente

Eine Analysekomponente unterstützt den Simulationsanwender bei der Auswertung der Ergebnisse. Jede Simulation wird aufgrund einer bestimmten Fragestellung durchgeführt und so können, je nach Fragestellung, unterschiedliche Modellvariablen im Interesse des Simulationsanwenders liegen. Von einem Simulator wird erwartet, dass er Möglichkeiten anbietet, um einzelne Modellvariablen zu beobachten, und Statistiken über diese erstellt ([PLC00], S.28):

> Neben den elementaren statistischen Größen, wie Extrema, Mittelwerte und Standardabweichung, sollten auch zeitlich gewichtete Statistiken ermittelt werden können.

Die Analysekomponente ermöglicht eine automatische Analyse und Aufbearbeitung der Ergebnisse. Daher muss der Simulationsanwender nicht immer wieder die gleichen mathematischen Formeln auf die resultierenden Daten anwenden, sondern er erhält bereits ein subanalysiertes Ergebnis der Simulation. Zudem ermöglicht die Analysekomponente automatische Vergleiche von unterschiedlichen Simulationsabläufen sowie eine Option, um benutzerdefinierte Analysefunktion definieren und hinzufügen zu können. Am Ende eines jeden Simulationsablaufes werden dann die analysierten Ergebnisse präsentiert, wofür eine Ausgabekomponente benötigt wird.

Die Analysekomponente untersucht hauptsächlich die Zustandsvariablen der Agenten, ihre Aktionen und die Interaktionen zwischen einzelnen Agenten und folgende Zustandsänderungen. Im Grunde muss hierfür eine Historie über alle Agentenaktionen eines Simulationsablaufes geführt werden. Dabei reicht es nicht aus, die einzelnen Agentenaktionen aufzuzeichnen, sondern es müssen auch die Zustandsvariablen der Agenten bei jeder Agentenaktion neu dokumentiert werden. Diese Aufzeichnung wird von einer Historienkomponente erledigt. Die Analysekomponente benutzt die gespeicherten Daten für die Analyse und leitet die Ergebnisse an die Ausgabekomponente weiter.

Ausgabekomponente

Die Ausgabekomponente ist für die Ausgabe und Präsentation der Ergebnisse der Analysekomponente zuständig. Dabei werden sowohl die Ausgabe auf dem Bildschirm präsentiert als auch deren Ergebnisse in einer Datei gespeichert. Die Bildschirmausgabe unterliegt keinen Einschränkungen. Die Art der Ausgabe reicht von einer tabellarischen Form bis hin zu komplexen grafischen Ausgaben, wie zum Beispiel mehrdimensionalen Diagrammen.

Visualisierungskomponente

Eine Visulaisierungskomponente bietet eine allgemeingültige Darstellung des Simulationsablaufes an. Anstatt für jedes neue Multiagentensystem eine eigene visuelle Unterstützung des Simulationsablaufes programmieren zu müssen, kann diese Komponente alle Simulationsabläufe auf eine universale Art und Weise darstellen. Ein Simulationsablauf kann visuell dargestellt werden, indem zum Beispiel Agentenaktionen mit grafischen Mitteln zur Laufzeit dargestellt werden.

Die drei zuletzt genannten Komponenten wurden nur der Vollständigkeit halber vorgestellt. Sie werden im Rahmen dieser Arbeit nicht implementiert, da sie nicht im Fokus der eigentlichen Aufgabenstellung liegen. Die bereits erwähnte Historienkomponente wird im nächsten Abschnitt vorgestellt und im späteren Verlauf dieser Arbeit auch implementiert, da sie, im Hinblick auf die drei optionalen Komponenten, eine praktische Schnittstelle für spätere Erweiterungen bietet.

3.4.5 Historienkomponente

Die Historienkomponente protokolliert alle wichtigen Simulationsereignisse. Beim Start der Simulation werden alle Agenten und ihre internen Zustände festgehalten. Tritt ein neues Ereignis auf, wird genaustens aufgezeichnet, was für ein Ereignis aufgetreten ist und welche Folgen es nach sich zieht. Hierzu gehören Veränderungen innerhalb der Agenten oder auch innerhalb des Multiagentensystems.

Die Historienkomponente ist somit während eines Simulationsablaufes ständig aktiv und zeichnet alle relevanten Daten auf. Da dieser Vorgang sehr rechenintensiv ist, sollte innerhalb der Administratorkomponente eine Auswahlmöglichkeit bereitgestellt werden, um die Historienaufzeichnung an- oder auschalten zu können.

3.4.6 Administratorkomponente

Die Adminstratorkomponente ist zuständig für die Verwaltung aller Simulationsabläufe. Diese ist nicht zwingend erforderlich, um eine Agentenplattform mit Simulationsunterstützung anzubieten, wird jedoch empfohlen, da sie dem Simulationsanwender eine benutzerfreundliche Administratormöglichkeit bietet.

Zu den allgemeinen Administratorfunktionen gehören eine Auswahlmöglichkeit für den Simulationsbetrieb sowie Funktionen, um einen Simulationsablauf verwalten zu können. Die Auswahlmöglichkeit bietet alle Simulationstechniken an, die in einer entsprechenden Zeit- und Ausführungskomponente realisiert sind. Zu den Verwaltungsfunktionen gehört das Laden eines Multiagentenszenarios genauso wie das Starten, Pausieren und Stoppen einer Simulation.

Des Weiteren soll, unabhängig von dem aktuellen Betriebsmodus, der aktuelle Fortschritt der System- und Simulationszeit präsentiert werden. Zudem sollen der ereignisgesteuerte und der zeitgesteuerte Simulationsmodus eine Option anbieten, um die Simulation manuell oder automatisch ausführen zu können. Wird die Simulation automatisch betrieben, durchläuft der Ausführungsalgorithmus ununterbrochen die Hauptschleife (siehe Abbildung 3.2) und führt alle Agentenaktionen so schnell wie möglich aus, bis die Simulation gestoppt wird. Im manuellen Betrieb wird nach jeder Runde der Hauptschleife gestoppt und der Benutzer muss manuell den nächsten Rundendurchlauf bestätigen. Eine Simulation, die automatisch oder schrittweise durchlaufen werden kann, bietet dem Simulationsanwender den Vorteil, den Vorgang innerhalb eines Multiagentensystem genaustens untersuchen zu können. Im Normalbetrieb (skalierten Simulationsbetrieb) würde ein manueller Betrieb nicht sinnvoll sein, da die Systemzeit (skalierte Systemzeit) stetig voranschreitet und es somit in einem Schritt zur Ausführung aller Agentenaktionen kommen könnte.

Jeder Simulationsmodus stellt weitere spezielle Einstellungen zur Verfügung. Der skalierte Simulationsmodus bietet eine Funktion an, um den Skalierungsfaktor während oder vor Beginn eines Simulationsablaufes ändern zu können. Das Gleiche gilt für die Veränderung des Zeitschrittes im zeitgesteuerten Simulationmodus. Der ereignisgesteuerte Simulationsmodus bietet eine Auswahl der Ausführungsgranularität an. Diese Ausführungsgranularität unterscheidet zwei Arten der Granularität. Die grobe Ereignisausführung führt automatisch alle Agentenaktionen mit gleichen Zeitstempeln aus. Die feinere Ausführung unterscheidet hingegen Agentenaktionen mit gleichen Zeitstempeln. Somit kann der feine, ereignisgesteuerte Simulationsmodus im manuellen Betrieb eine Agentenaktion nach der anderen ausführen, auch wenn die Agentenaktion den gleichen Zeitstempel hat.

Zudem werden für alle Betriebsmodi zukünftige und vergangene Agentenaktionen in einer Tabelle aufgelistet und dem Simulationsanwender präsentiert, damit dieser weiß, welche Agentenaktionen als nächstes durchgeführt werden und welche bereits der Vergangenheit angehören.

3.5 Zusammenfassung

In diesem Kapitel werden die Anforderungen an die allgemeine Simulationsunterstützung aufgelistet. Die Anforderungen können prinzipiell auf zwei unterschiedliche Arten erfüllt werden, wobei in dieser Arbeit die Implementierung auf Plattformebene gewählt wird. Danach wird die allgemeine Simulationsarchitektur vorgestellt, mit deren Hilfe die Simulationsunterstützung erreicht werden kann.

Die Simulationsarchitektur besteht aus mehreren Komponenten, die die Grundlage der Simulationssoftware bilden. Des Weiteren wird die Architektur mit optionalen Komponenten ausgestattet, die den Aufbau vervollständigen und so zu einer voll funktionsfähigen Simulationsunterstützung für Agentenplattformen beitragen sollen.

Nachdem das Konzept einer Simulationsarchitektur für Agentenplattformen veranschaulicht worden ist, soll im nächsten Kapitel die Implementierung an einer konkreten Agentenplattform beschrieben werden. Als Agentplattform wird Jadex gewählt, das am Arbeitsbereich VSIS (Verteilte Systeme & Informationssysteme) des Fachbereiches Informatik der Universität Hamburg entwickelt wird.

Jadex wird mit der objektorientierten Programmiersprache Java[4] entwickelt und ist ein Vertreter der BDI-Modelle für Agenten. Es zeichnet sich durch die Implementierung einer BDI *Reasoning-Engine* aus, die vom Rest der Agentenplattform entkoppelt ist und sich anhand einer Schnittstelle in beliebige Agentenplattformen integrieren lässt. Hierfür müssen nur bestimmte Basisdienste innerhalb eines sogenannten *Adapters* implementiert werden. Dies wird im nächsten Kapitel detailliert dargestellt.

[4] Siehe http://java.sun.com/.

4 Implementierung der Simulationsplattform

Die im vorherigen Kaptiel definierten Komponenten einer Simulationsarchitektur werden nun am Beispiel von Jadex implementiert. Damit wird gezeigt, dass sich die zuvor abstrakt definierte Simulationsarchitektur in der Praxis umsetzen lässt. Nachdem die Simulationsplattform realisiert ist, werden die Fähigkeiten und Garantien der Architektur zusammengefasst und den zuvor definierten Anforderungen gegenübergestellt.

4.1 Jadex

Jadex (JADE Extension) ist eine BDI Reasoning-Engine, die anfänglich als Erweiterung der Agentenplattform JADE (Java Agent Development Framework) entwickelt wurde (siehe [PBL03]). Jade legt den Schwerpunkt auf die Entwicklung einer Agentenplattform und deren Basisdienste, wie zum Beispiel den Transport von Nachrichten und die Verwaltung der Agenten über ein ADS. Hierzu gehört die Umsetzung einer Ablaufumgebung oder auch Middleware für Agenten, die die Standards der FIPA realisiert. Des Weiteren überzeugt JADE durch Stabilität, viele zusätzliche Hilfsfunktionen zum *Debuggen* und eine große Gemeinde, die beständig an der Weiterentwicklung der Plattform arbeitet.

Jadex macht sich die gegebene Infrastruktur von JADE zunutze und entwickelt auf dieser eine BDI Reasoning-Engine, die die interne Architektur der Agenten repräsentiert. Hierzu gehört der interne Zustand eines Agenten sowie die Umsetzung von den Konzepten für rationales Handeln. Zudem kann Jadex auch als ein Software-Framework betrachtet werden, mit dem komplexe, zielorientierte Agenten entworfen werden können, die den Konzepten eines BDI-Modells folgen und auf beliebigen Agentenplattformen ablaufen können.

4.1.1 Adapter

Jadex legt den Schwerpunkt auf die BDI-Architektur der Agenten und nicht auf die typischen Basisdienste einer Agentenplattform. Aus diesem Grund wird die *Reasoning*-Schicht mit einer Adapterschicht lose gekoppelt. Durch diese lose Koppelung kann Jadex mit den unterschiedlichsten Agentenplattformen integriert werden oder auch mit anderer Unternehmenssoftware (Enterprise Systems) zusammenarbeiten, wie in [Lin06] am Beispiel von *Java EE*[1] gezeigt wurde.

[1] Enterprise Edition.

Damit eine Agentenplattform mit dem BDI-System von Jadex gekoppelt werden kann, wurde das *Adapter* Entwurfsmuster aus der Softwareentwicklung verwendet. Für jede bestehende Agentenplattform, wie zum Beispiel JADE, muss ein Adapter entwickelt werden, der die Koppelung zwischen Agentenplattform und Jadex realisiert. Das hierfür benutzte Adapter-Entwurfsmuster wird in ([GHJV98], S.139) wie folgt definiert:

> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Die Definition bezieht sich auf die Koppelung von zwei Klassen mittels eines Adapters. In Jadex wird die BDI Reasoning-Engine über den gleichen Mechanismus in eine Plattform integriert, wie in Abbildung 4.1 dargestellt wird.

Jadex zeichnet sich durch drei Schichten aus: Die grundlegende Middlewareschicht (hier *Host Platform*), eine Adapterschicht (*Adapter Layer*) und die BDI Reasoning-Engine (gekapselt im *Jadex Agent*). Die *Host Platform* kann nur den Agenten der eigenen Plattform ausführen (hier *JADE Agent*). Damit dieser Agent auch auf die BDI-Funktionen des *Jadex Agent* zugreifen kann, wird der *Jadex Agent* von einem *Adapter Agent* eingehüllt, indem wichtige Dienste des *Jadex Agent* und des *Adapter Agent* über eine Schnittstelle bereitgestellt werden.

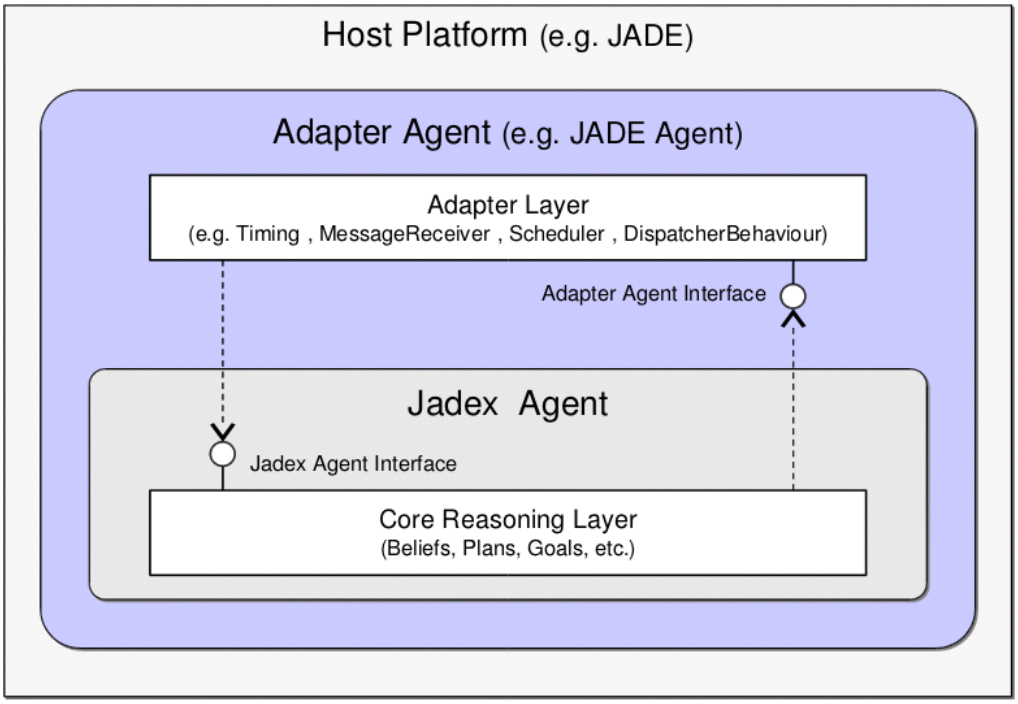

Abbildung 4.1: Jadex Integration (aus [BPL05b])

In [BPL05b] wird zusammenfassend festgehalten, dass die Schnittstelle des *Jadex Agent* die BDI-Verfahren bereitstellt, damit der Agent aufgrund seiner eigenen Weltanschauung schlussfolgern (reasoning) kann. Die Schnittstelle des *Adapter Agent* stellt hingegen die

technischen Hilfsmittel bereit, um Agenten benachrichtigen zu können und dem Agenten einen eigenen Nachrichtenversand zu ermöglichen. Somit lassen sich beliebige Agentenplattformen mit der BDI Reasoning-Engine von Jadex erweitern, obwohl beide Seiten immer nur über einen Adapter miteinander verbunden sind. Insofern werden komplexe Abhängigkeiten vermieden und die Kopplung bleibt überschau- und austauschbar.

Die vorgestellte Integration von Basisdiensten einer Agentenplattform mit den Mechanismen des BDI-Modells von Jadex bietet sich besonders für die Implementierung einer Simulationsunterstützung an. Auf der einen Seite müssen die Basisdienste der zugrunde liegenden Middlewareschicht um eine Simulationsinfrastruktur erweitert werden, und auf der anderen Seite kann die BDI Reasoning-Engine ohne Änderungen übernommen werden. Eine Veränderung in der Middlewareschicht hat keine Einwirkung auf die Realisierung der BDI-Kernfunktionen. Das BDI-Modell kann unabhängig weiterentwickelt werden und muss sich keiner speziellen Middleware anpassen, solange die geforderten Basisdienste bereitgestellt werden.

Somit werden Simulationsabläufe erreicht, die auf BDI-Agenten basieren. Zudem wird die Simulationsunterstützung auf Middlewareebene realisert, indem ein geeigneter Adapter (Simulation Adapater Agent) entworfen wird, der sowohl die Dienste der neuen Simulationsinfrastruktur sowie die Methoden der Reasoning-Engine implementiert.

4.1.2 Plattformadapter

Jede Middleware (Host Platform) muss grundlengende Dienste für Jadex bereitstellen. Hierzu gehört eine ADS-Komponente für die Verwaltung der Agenten, eine SD-Komponente für die Auffindung von Dienstleistungen sowie eine ACL-Komponente für den FIPA konformen Nachrichtenversand. Gegenwärtig gibt es drei Adapter für Jadex, die eine solche Ablaufumgebung ermöglichen (nach [BPL06a]). Hierzu gehört JADE, ein experimenteller Adapter für *Diet*[2] und eine alleinstehende Plattform (Standalone Plaform).

Der Plattformadapter stellt die Verbindung zwischen der Middleware und der Jadex Reasoning-Engine her. Der Plattformadapter muss drei grundlegende Schnittstellen implementieren, die im Kern von Jadex festgelegt sind. Hierzu gehört der *IAgentAdapter*, *IMessageAdapter* und ein *IMessageEventAdapter*. Der *IAgentAdapter* wird vom *Jadex Agent* benötigt, um grundlegende Methoden auf dem spezifischen *Platform Agent* aufzurufen, wie zum Beispiel die Benachrichtigung von Agenten. Die Schnittstelle *IMessageAdapter* wird benötigt, damit Nachrichten von der externen Middleware auch vom *Jadex Agent* verarbeitet werden können. Im Gegensatz dazu wird die Schnittstelle *IMessageEventAdapter* vom *Jadex Agent* für den eigenen Nachrichtenversand benötigt.

Des Weiteren muss ein Plattformadapter eine sogenannte *Plan Library* bereitstellen. In dieser Bibliothek befinden sich Funktionen der ADS- und SD-Komponente, die mittels

[2]Weitere Informationen unter http://diet-agents.sourceforge.net/.

Capabilities[3] umgesetzt werden. Hierzu gehört zum Beispiel das Erstellen, Suchen und Zerstören von Agenten.

Die alleinstehende Plattform (Standalone Platform) benötigt keine weitere Middleware, sondern stellt selbst alle essenziellen Dienste einer Agentenplattform bereit und bietet dem Programmierer die vollständige Kontrolle über die Ausführung von Agenten an. Dies beinhaltet vor allem die Kontrolle über die leichtgewichtigen Prozesse (Threads) eines Agenten. Die Plattform kann jedem Agenten einen neuen Thread zuweisen und somit auf Nebenläufigkeit setzen, oder eine abgestufte Art von Nebenläufigkeit benutzen, bis hin zu einem globalen Thread und ergo eine serielle Ausführung von Agenten. Die Realisierung der Agentenausführung wird somit dem Programmierer überlassen.

Aufgrund der Kontrollmöglichkeiten über alle Schichten der Agentenplattform wird die Simulationsplattform die alleinstehende Plattform als Grundlage benutzen und um die erforderten Komponenten erweitern, woraus eine Plattform mit Simulationsunterstützung resultiert. Dabei wird darauf geachtet, dass der Kern von Jadex mit allen bisherigen Plattformadaptern kompatibel bleibt und die BDI Reasoning-Engine nicht grundlegend verändert werden muss.

4.1.3 Kernsystem von Jadex

Das Kernsystem von Jadex bildet den elementaren Bestandteil dieser Agentenplattform, auf den jeder Plattformadapter angewiesen ist. Zu diesem Kern gehört nicht nur die BDI Reasoning-Engine, sondern auch eine Reihe von Hilfsmitteln und weiteren Dienstprogrammen. Der Anwendungsentwickler braucht zum Beispiel eine Möglichkeit, um BDI-Agenten auf einfache Art und Weise spezifizieren zu können, damit das Kernsystem anhand dieser Informationen in der Lage ist, ein lauffähiges Objekt zu erstellen. Zu diesem Zweck wird JiBX[4] benutzt, welches die sogenannten ADF-Dateien (Agent Description Files) an Java Objekte bindet (siehe [BPL06a], Kapitel 4.), die als BDI-Agenten in der Reasoning-Engine verarbeitet werden.

Neben weiteren Hilfsmitteln, verschiedenen Konfigurationsmöglichkeiten und allgemein definierten Fähigkeiten der Agenten (*Capabilities*) werden mehrere Schnittstellen oder auch abstrakte Klassen definiert, die in jede Adapterschicht implementiert werden müssen. Sie stellen die Verbindung zwischen dem Kern von Jadex und allen Adapterschichten her. Im folgenden Abschnitt werden die wichtigsten Schnittstellen untersucht und auf notwendige Änderungen (oder Erweiterungen) im Kern von Jadex eingegangen, damit als nächstes die Simulationsplattform entworfen werden kann.

Jadex ermöglicht die Erstellung und Benutzung von BDI-Agenten für entsprechend adaptierte Middleware. Ein BDI-Agent von Jadex wird in der Klasse *RBDIAgent* definiert, die das Interface *IJadexAgent* implementiert (siehe Abbildung 4.2). Jeder Adapter be-

[3] Weitere Informationen finden sich zum Beispiel in [BPL05a].
[4] Weitere Informationen unter http://jibx.sourceforge.net/.

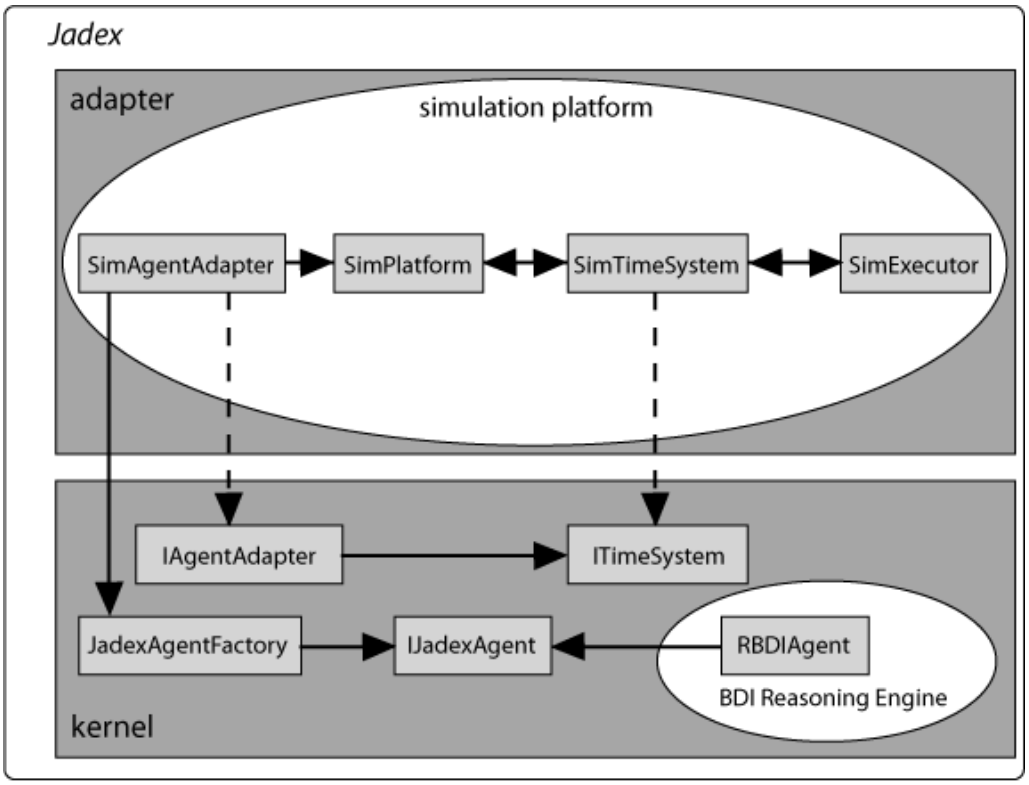

Abbildung 4.2: Simulationsplattform

nutzt die gleiche Fabrik-Methode der Klasse *JadexAgentFactory*, um einen Agenten vom Typ *IJadexAgent* zu erzeugen. Die Fabrik erzeugt Agenten als Laufzeitobjekte und kann entscheiden, von welchem konkreten Typ dieser Agent ist. Das Fabrik-Entwurfsmuster überlässt der Fabrik die Entscheidung, welche Klasse zu instanziieren, zu konfigurieren und schließlich zurückzugeben ist. In Frage kommen alle Klassen, die die Schnittstelle *IJadexAgent* implementieren.

Die Agentenfabrik und die Struktur der BDI-Agenten müssen nicht geändert werden. Die Fabrik im Kern von Jadex kümmert sich weiterhin um die Erzeugung von korrekten BDI-Agenten und sollte somit von jeder Adapterschicht (auch vom Simulationsadapter) benutzt werden.

Andere wichtige Schnittstellen betreffen den Nachrichtenversand. Auf der einen Seite existiert die Schnittstelle *IMessageAdapter*, um externe Nachrichten an einen Jadex-Agenten zu schicken, und auf der anderen Seite wird die Schnittstelle *IMessageEvent-Transport* bereitgestellt, damit die Jadex-Agenten ihre eigenen Nachrichten über beliebige Transportwege verschicken können. Der Nachrichtenversand muss, genau wie der Erzeugungsprozess der Agenten, auch nicht verändert werden, da er keinen Einfluss auf die kommende Simulationsarchitektur hat. Die Nachrichten können weiterhin wie zuvor verschickt werden.

Die Schnittstelle *IAgentAdapter* ist für die zu entwickelnde Simulationsunterstützung entscheidend. Diese Schnittstelle verlangt die konkrete Implementierung eines Plattformagenten (SimAgentAdapter), der alle Anforderungen der Simulationsplattform erfüllt und das Kernsystem von Jadex mit der Simulationsplattform koppelt (siehe Abbildung 4.2). Als Grundlage wird der Plattformagent der alleinstehenden Plattform verwendet. Diese Implementierung enthält bereits alle grundlegenden Methoden, wie zum Beispiel den Nachrichtenversand und muss nur noch entsprechend den neu hinzukommenden Simulationsanforderungen angepasst werden.

Zu den wichtigsten Simulationsanforderungen gehört die Einbindung einer eigenen Zeitkomponente. Die Einbindung der Zeitkomponente ist die einzige Erweiterung im Kern von Jadex, die unumgänglich ist und auf alle anderen Adapter nachhaltige Auswirkung hat. Die Zeitkomponente muss im Kernsysten von Jadex verankert werden, damit jegliche Agentenaktivitäten[5], die an eine zeitliche Ausführung gebunden sind, über eine zentrale Komponente laufen und von dieser individuell (je nach Ablaufumgebung) gesteuert werden können. Dabei muss sowohl der Kern von Jadex, als auch die Adapterschicht auf die zentrale Zeitkomponente zugreifen können. Aus diesem Grund wird im Kernsystem die Schnittstelle *ITimeSystem* definiert, die von jedem Adapter zusätzlich implementiert werden muss. Im Falle der Simulationsplattform wird diese Aufgabe durch das *SimTimeSystem* erledigt (siehe Abbildung 4.2).

Solche Agentenaktionen wurden normalerweise im Kern von Jadex und in der Adapterschicht zu Zeitpunkten der Systemzeit ausgeführt. Für Simulationen hat dies in einer eigenen Zeitkomponente zu geschehen (SimTimeSystem), damit die Zeit auf eigene Art und Weise verwaltet werden kann. Folglich wird die Plattformzeit nicht notwendigerweise durch die Systemzeit berechnet. Die Schnittstelle *ITimeSystem* im Kern von Jadex delegiert die Verwaltung der Plattformzeit an die Adapterschicht, wodurch diese auch die zeitliche Ausführungskontrolle über alle Agentenaktionen erhält. Infolgedessen stellt die notwendige Implementierung eines eigenen, komplexen Zeitsystems für den Simulationsbetrieb kein Problem mehr dar. Die Simulationsplattform kann ihr eigenes Zeitsystem entwerfen und bestehende Plattformadapter beschränken sich bei der Implementierung ihres eigenen Zeitsystems auf die minimale Erfüllung der Anforderungen.

Die Zeitkomponente *ITimeSystem* wird über den zentralen Agentenadapter *IAgentAdapter* jeder Adapterschicht zugänglich gemacht. Hierfür muss die Schnittstelle *IAgentAdapter* um eine neue Methode erweitert werden:

```
1 public ITimeSystem getTimeSystem();
```

Die Schnittstelle *ITimeSystem* ist im Gegensatz zum *IAgentAdapter* eine äußerst leichtgewichtige Schnittstelle. Sie fordert nur die Implementierung einer Methode:

```
1 public long getTime();
```

[5] Wie zum Beispiel das Warten eines Agenten, auf eine Nachricht oder ein anderes Ereignis.

Die Methode gibt die genaue Zeit (Zeitstempel) der aktuellen Plattform zurück. Der Kern von Jadex sowie alle Plattformadapter benutzen nur noch das *ITimeSystem*, um zeitliche Abhängigkeiten zu verarbeiten. Wenn die zu koppelnde Plattform keinen Simulationsbetrieb benötigt und die Zeitkomponente so einfach wie möglich implementiert werden soll, wird die Systemzeit zurückgegeben:

```
1 public long getTime()
2 {
3      System.currentTimeMillis();
4 }
```

Dies sind alle nötigen Veränderungen im Kern von Jadex, damit eine Simulationsplattform mit dem Kernsystem von Jadex gekoppelt werden kann. Der Kern muss nur um die Schnittstelle *ITimeSystem* erweitert werden, wodurch bestehende Adapter schnell und einfach an das neue Kernsystem gekoppelt werden können.

Im folgenden Abschnitt wird die konkrete Implementierung der Adapterschicht erläutert, die die grundlegende Simulationsarchitektur repräsentiert.

4.2 Adapterschicht der Simulationsplattform

Die Adapterschicht der Simulationsplattform basiert auf der alleinstehenden Plattform von Jadex und benötigt somit keine weitere Middleware, da sie bereits alle erforderlichen Basisfunktionen anbietet.

Im Abschnitt 4.1.3 wurde bereits auf den Agentenadapter *SimAgentAdapter* eingegangen. Wie aus Abbildung 4.2 zu entnehmen ist, hat dieser Adapter keinen direkten Zugriff auf das Zeitsystem, sondern greift anhand der Klasse *SimPlatform* auf das Zeitsystem zu. Die Klasse *SimPlatform* ist der Einstiegspunkt für die Erzeugung der Simulationsplattform. Hier werden grundlegende Dienstleistungen, wie zum Beispiel der Nachrichtenversand, vorbereitet sowie der Start der grafischen Oberfläche eingeleitet und die fundamentalen Komponenten der abstrakten FIPA-Architektur erzeugt (siehe Abbildung 3.4).

Die Simulationsplattform wird beim Starten standardmäßig mit dem Normalmodus initialisiert. Dieser Modus beruht auf einem Zeitsystem, dass immer die aktuelle Systemzeit als Plattformzeit zurückgibt. Allerdings gehört zu der Initialisierunsphase nicht nur die Erstellung eines speziellen Zeitsystems der abstrakten Zeitkomponente (SimTimeSystem), sondern auch die Erzeugung eines entsprechenden Threads, der für die wiederholbare Ausführung der Agentenaktionen zuständig ist.

Die Zeitkomponente *SimTimeSystem* ist, genau wie die Ausführungskomponente *SimExecutor*, eine abstrakte Klasse, die nicht erzeugt werden kann. Beide Komponenten haben für jeden Simulationsbetrieb genau eine Spezialisierung implementiert (siehe Abbildung 4.3). Es werden jeweils die vier Betriebsmodi *normal*, *scaled real-time*, *time-stepped* und

event-driven implementiert. Wird ein konkretes Zeitsystem beim Starten der Simulationsplattform erzeugt oder während des Betriebes geändert, garantiert die Simulationsplattform auch eine Änderung der speziellen Ausführungskomponente. Somit muss bei der Änderung des aktuellen Betriebsmodus bedacht werden, dass vorherige Threads zur Ausführung nicht weiter benötigt und deshalb gestoppt werden müssen, um keine unnötigen Berechnungen durchzuführen.

4.2.1 Realisierung unterschiedlicher Betriebsmodi

Die Realisierung der vier unterschiedlichen Betriebsmodi entsteht im Grunde aus der Interaktion zwischen der Zeit- und der Ausführungskomponente (siehe Abbildung 4.3). Durch das Zusammenspiel beider Komponenten wird eine wiederholbare Berechnung eines Multiagentensystems erreicht, die je nach Betriebsmodus auf unterschiedliche Art und Weise abläuft.

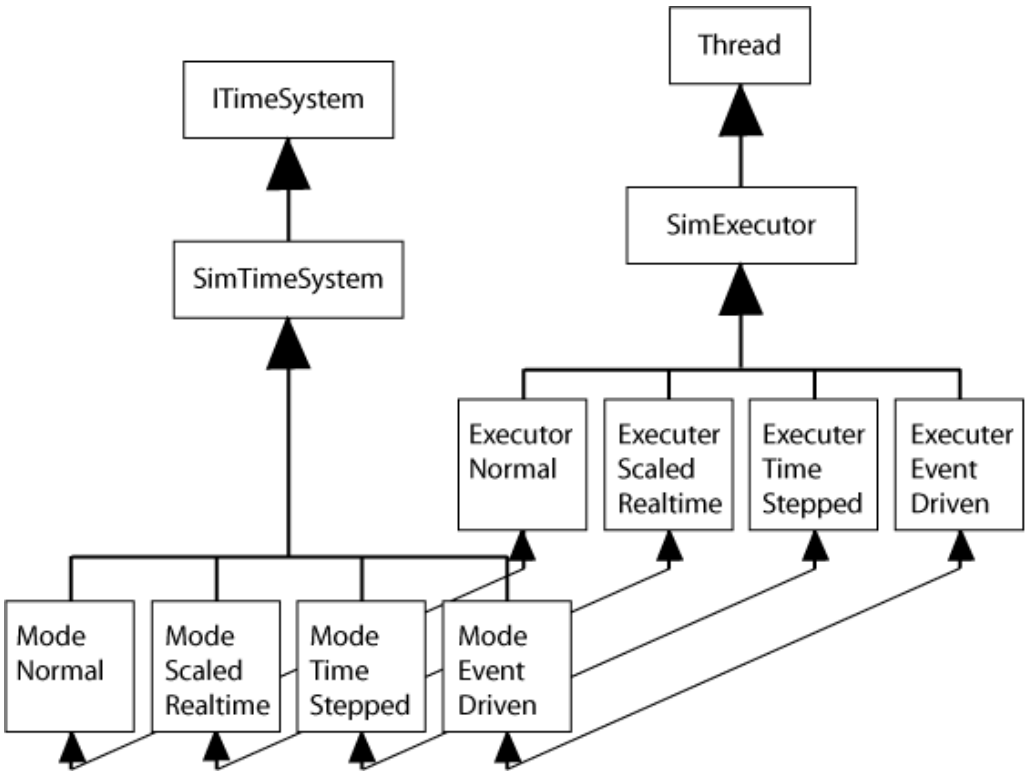

Abbildung 4.3: Zusammenspiel der Zeit- und Ausführungskomponente

In der Simulationsplattform wird die Schnittstelle *ITimeSystem* von der abstrakten Klasse *SimTimeSystem* implementiert. *SimTimeSystem* ist abstrakt, weil die Methode *getTime()* abstrakt gehalten wird. Dies ermöglicht spezielle Zeitsysteme, die die Zeit auf eigene Art und Weise verwalten können. Zudem bietet *SimTimeSystem* weitere, konkrete Methoden an, die für alle Unterklassen benötigt werden, wie zum Beispiel der Zugriff auf den aktuellen Thread der Ausführungskomponente (Ausführungsprozess), der von dem aktuellen Zeitsystem erzeugt wird. Wichtig ist hierbei, dass die Simulationsplattform im-

mer nur einen Ausführungsprozess zulässt. Wird ein neues Zeitsystem initialisiert und der entsprechende Ausführungsmodus gestartet, muss der vorherige Ausführungsprozess bereits beseitigt sein.

Alle Ausführungsprozesse werden von der abstrakten Klasse *SimExecutor* abgeleitet. Diese Klasse beinhaltet eine Reihe von Methoden und Einstellungen, die für alle konkreten Ausführungssysteme gültig sind. Hierzu gehört unter anderem die Überprüfung, ob die Simulation automatisch oder manuell ausgeführt werden soll, ob die Simulation pausiert oder gestoppt wird sowie die Implementierung des allgemeinen Ausführungsprozesses (siehe Algorithmus 2).

```
  /** Erzeuge Prozess                                          **/
1 while Ausführung läuft do
2     if keine Pause then
3         if automatische Ausführung then
4             NotifyAgents();
5         else
6             WaitForUserAction();
7         end
8         ExecuteAgents();
9     end
10 end
  /** Beende Prozess                                           **/
```

Algorithmus 2 : Allgemeiner Ausführungsalgorithmus

Allein die Methode *NotifyAgents()* (Zeile vier) wird in der Klasse *SimExecutor* abstrakt gehalten und muss von jeder konkreten Ausführungskomponente realisiert werden. Anhand dieser Methode wird berechnet, wie die Zeit fortschreitet und welcher Agent oder Agenten nach dem Fortschritt der Zeit benachrichtigt werden, da sie als nächstes eine Agentenaktion ausführen dürfen. Benachrichtigte Agenten tragen sich somit unverzüglich in die Ausführungsschlange der abstrakten Ausführungskomponente (SimExecutor) ein. Hierbei tragen sich die Agenten in der gleichen Reihenfolge ein, wie sie benachrichtigt werden.

Falls der Simulationsbetrieb nicht automatisch ausgeführt werden soll, ruft der Algorithmus 2 keine Methode auf, um die Zeit fortschreiten zu lassen und um neue Agenten zu benachrichtigen (Zeile sechs), sondern wartet auf die Interaktion des Simulationsanwenders. Erst durch die Benutzerinteraktion findet ein Zeitsprung statt, der Agentenaktionen nach sich ziehen kann.

Nachdem sich kein Agent, ein Agent oder mehrere Agenten in die *Ausführungsschlange* der Ausführungskomponente eingetragen haben, wird in Zeile acht überprüft, ob

die Ausführungsschlange leer ist. Ist dies nicht der Fall werden alle wartenden Agenten nacheinander ausgeführt, bis die Ausführungsschlange wieder leer ist. Danach kann der Ausführungsalgorithmus wieder von vorne beginnen.

Im weiteren Verlauf des Abschnittes wird auf die spezielle Benachrichtigung von Agenten eingegangen (Algorithmus 2, Zeile 4), zu der auch die Berechnung der fortschreitenden Simulationszeit gehört. Für diese spezielle Methode wird eine weitere Warteschlange in der Klasse *SimExecutor* bereitgestellt. Diese *Agentenschlange* besteht aus Referenzen auf alle Agenten, die an der Simulation teilnehmen. Ihre Reihenfolge entsteht aus der Erzeugungsreihenfolge der Agenten und bestimmt die Sequenz der Benachrichtigungen und somit auch die Ausführungsreihenfolge der Agentenaktionen. Anhand dieser global festgelegten Ordnung wird in jedem Betriebsmodus[6] eine wiederholbare Ausführung garantiert, solange die Simulationsabläufe gleich initialisiert werden.

Normalbetieb

Der Normalbetrieb wird durch die Zusammenarbeit des Zeitsystems *ModeNormal* und der Ausführungskomponente *ExecutorNormal* gewährleistet (siehe Abbildung 4.3). Der Ausführungsalgorithmus 2 ruft in Zeile vier die Benachrichtigungsmethode *NotifyAgents()* des Normalbetriebes auf, der in Algorithmus 3 veranschaulicht wird.

```
  /** Benachrichtigung von Agenten im Normalbetrieb          **/
1 if Agentenschlange nicht leer then
2 |   simtime ← GetPlatformTime();
3 |   foreach Agent in Agentenschlange do
4 |   |   next ← GetNextAction(Agent);
5 |   |   diff ← (next − simtime);
6 |   |   if diff <= 0 then
7 |   |   |   Notify(Agent);
8 |   |   end
9 |   end
10 end
  /** Weiter mit allgemeinem Ausführungsalgorithmus 2        **/
```

Algorithmus 3 : Benachrichtige Agenten im Normalbetrieb

In diesem Algorithmus wird als erstes die Plattformzeit anhand der Zeitkomponente abgefragt (Zeile zwei), die in diesem Modus nichts anderes als die aktuelle Systemzeit ist und in Form eines Zeitstempels zurückgegeben wird. Danach werden alle Agenten in einer globalen Reihenfolge durchlaufen und der Zeitstempel ihrer nächsten Agentenaktion abgefragt (Zeile vier). Falls eine Agentenaktion einen kleineren (oder gleichen) Zeitstempel hat als der aktuelle Zeitstempel der Plattform wird der Agent benachrichtigt und in

[6] Die Simulationsplattform benutzt nur serielle Ausführungsbetriebe.

die Ausführungsschlange der Ausführungskomponente eingetragen (Zeile sieben). Nach der Benachrichtigung aller nötigen Agenten wird die Kontrolle zurück an den Ausführungsalgorithmus in Zeile acht gegeben, damit die benachrichtigten Agenten der Reihe nach ausgeführt werden können.

Skalierter Simulationsbetrieb

Das skalierte Ausführungssystem *ExecutorScaledRealtime* benutzt zur Benachrichtigung der Agenten ebenso den Algorithmus 3, der bereits vom Normalbetrieb benutzt wird. Der Unterschied beider Modi liegt nur in der Implementierung der Methode *GetPlatformTime()* (Zeile zwei). Anstatt die Systemzeit zurückzubekommen, wird im skalierten Zeitsystem *ModeScaledRealtime* folgende Gleichung verwendet (vgl. mit 2.1)

$$simTime = startTime + fix + scale * (sysTime - startSysTime) \tag{4.1}$$

Das Zeitsystem des skalierten Simulationsbetriebes benutzt den Skalierungsfaktor *scale*, der jederzeit geändert werden kann, solange er in einem definiert Bereich von $]0, X]$ liegt. Falls der Skalierunsfaktor eins ist, wird die Systemzeit zurückgegeben, genau wie es im Normalbetrieb der Fall ist. Für ein Faktor größer als eins gilt, dass die Simulationszeit immer schneller voranschreitet, je näher sie dem Wert X kommt. Der Wert X bildet die obere Grenze des Skalierungsfaktors und kann durch den Benutzer festgelegt werden. Diese obere Grenze sollte von dem Simulationsanwender je nach Anwendungsszenario mit Bedacht gesetzt werden, da laut ([Fuj00], S.29) gilt:

> it is easy to slow down a simulation, but often difficult to speed one up!

Zudem bringt zum Beispiel ein schneller Zeitfortschritt nichts, wenn der Computer mit der Berechnung der Simulationsergebnisse nicht mithalten kann.

Wird der Skalierungsfaktor hingegen kleiner als eins und nähert sich der Null an wird die Simulation immer langsamer. Dies stellt kein Problem für den Rechner dar. Im Gegensatz zu einer Steigerung der Geschwindigkeit können Ausführungen von Agentenaktionen ohne Probleme verzögert ausgeführt werden. Die Berechnung der Agentenaktion selber bleibt immer gleich, unabhängig davon, ob die Zeit schneller oder langsamer läuft. Die untere Grenze von Null wird nie erreicht. Dies würde eine Pausierung der Simulation bewirken. Allerdings wird die Pausierung für alle Simulationsmodi gefordert und unter Zuhilfenahme der Ausführungskomponente realisiert. Daher wurde hier bereits die Null als nicht zulässig definiert.

Der Wert *sysTime* stellt immer die aktuelle Systemzeit dar. Wird von diesem Wert die Systemzeit zu Beginn der Simulation abgezogen (*startSysTime*), bleibt die Anzahl der Millisekunden übrig, die innerhalb des aktuellen Simulationsablaufes bereits vergangen sind (Fortschritt der Simulation). Das Produkt aus dem Fortschritt der Simulation und dem Skalierungsfaktor zeigt somit den skalierten Fortschritt der Simulation an. Damit aber

nicht nur die Anzahl der vergangenen Millisekunden seit Beginn der Simulation angezeigt wird, addiert man den Wert *startTime* dazu. Der Zeitstempel *startTime* entspricht einem genauen Datum. Somit kann nicht nur die vergangene Simulationszeit angezeigt werden, sondern auch ein konkretes Datum angegeben werden.

In der Gleichung 4.1 ist der Wert von *startTime* identisch mit dem von *sysTime*. Allerdings ist dies keine Notwendigkeit. Der Wert *startTime* könnte bei der Initialisierung der Simulation zum Beispiel ein Datum aus der Vergangenheit oder Zukunft sein. Folglich könnte der Simulationsanwender eine Simulation starten, dessen Startpunkt in der Vergangenheit oder Zukunft liegt. Für diese Arbeit ist es ausreichend, den Zeitstempel der Simulation automatisch auf die reale Zeit zu setzen. Der Fortschritt der Zeit wird mittels *sysTime* berechnet und bleibt gleich, unabhängig vom Zeitpunkt, an dem die Simulation startet.

Schließlich ist noch der Wert von *fix* zu erklären. Dieser Wert ist bei der Initialisierung des skalierten Simulationsmodus Null und wird immer dann neu berechnet, wenn der Skalierungsfaktor zur Laufzeit geändert wird. Diese Änderung bewirkt einen Zeitsprung in die Zukunft oder Vergangenheit (siehe Abbildung 4.4), der nicht erwünscht ist und infolgedessen korrigiert werden muss. Falls der Simulationsanwender den Skalierungsfaktor zur Laufzeit ändert, soll die Simulation genau ab dem letzten Simulationszeitpunkt (vor dem Wechsel) weiterschreiten. Nur das Tempo des Weiterschreitens soll verändert werden können, ohne das ein Zeitsprung[7] auftritt. Der Wert *fix* berechnet die Differenz des Zeitsprungs und addiert diesen positiven oder negativen Wert auf die Simulationszeit.

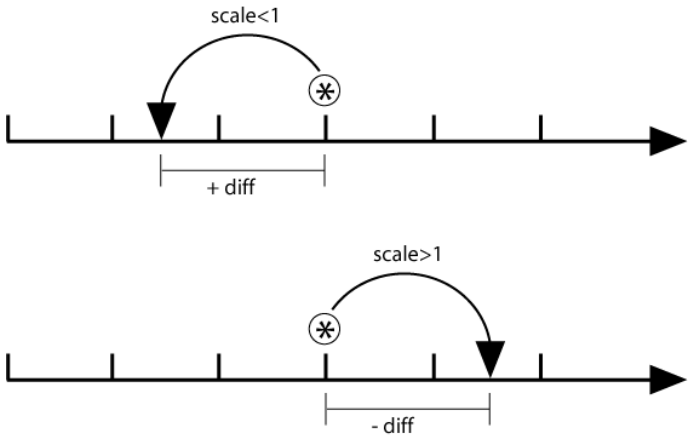

Abbildung 4.4: Änderung des Skalierungsfaktors zur Laufzeit und seine Folgen

Die skalierte Simulationsausführung soll genau wie der Normalbetrieb nur automatisch (siehe Zeile drei in Algorithmus 2) berechnet werden. Ein manueller Betrieb würde keinen Sinn ergeben, da beide Modi zu abhängig von der Systemzeit sind und diese immer weiterläuft, unabhängig davon, ob die Methode *NotifyAgents()* aufgerufen wird oder

[7] Dieser Zeitsprung tritt nur auf, wenn der Skalierungsfaktor zur Laufzeit geändert wird.

nicht. In den Simulationsmodi *time-stepped* und *event-driven* ist die Simulationzeit vollständig von der Systemzeit entkoppelt[8] und schreitet auf eigene Weise voran. Hier macht eine Unterscheidung zwischen manuellem und automatischem Betrieb durchaus Sinn, wie im folgenden verdeutlicht wird.

Simulation in diskreten Zeitschritten

Das Zeitsystem *ModeTimeStepped* wird für den Simulationsmodus *time-stepped* benötigt. Es basiert im Wesentlichen auf einem Zeitschritt, der vom Simulationsanwender zur Laufzeit ständig verändert werden kann. Beim Starten dieses Simulationsmodus beträgt der Wert laut Voreinstellung 1000 Millisekunden, was wiederum einer Sekunde entspricht.

Das Zeitsystem gibt beim Aufruf der Methode *GetTime()* immer die aktuelle Simulationszeit zurück, die beim Starten dieses Simulationsmodus mit der Systemzeit initialisiert wird. Die beiden vorherigen Betriebsmodi haben beim Abfragen der Plattformzeit auch zu einer Änderung dieser Plattformzeit geführt. Dies gilt für die beiden letzten Simulationsbetriebe (time-stepped, event-driven) nicht mehr. Bei dem diskreten Zeitmodell wird die Simulationszeit nur über die Methode *NextTimepoint()* (Algorithmus 4 Zeile eins) verändert. Beim Aufruf dieser Methode wird die neue Simulationszeit aus der Summe der aktuellen Simulationszeit und der Größe des Zeitschrittes neu berechnet (siehe Gleichung 4.2). Danach wird der gleiche Ausführungsalgorithmus wie zuvor verwendet.

$$simTime = simTime + timestep \tag{4.2}$$

Der resultierende Algoritmus 4, der die Agenten im Simulationsbetrieb *time-stepped* benachrichtigt, unterscheidet sich nur durch den expliziten Sprung zum nächsten Zeitpunkt (Zeile eins) von Algorithmus 3 und nur durch die Art und Weise, wie die Simulationszeit berechnet wird.

```
  /** Benachrichtige Agenten                                    **/
1 NextTimepoint();
2 Aufruf von Algorithmus 3;
  /** Weiter mit Ausführungsalgorithmus 2                       **/
```

Algorithmus 4 : Benachrichtigung von Agenten im Modus Time-Stepped

Ereignisgesteuerte Simulation

Die Simulationszeit des Simulationsmodus *event-driven* wird wie die anderen Simulationsmodi bei der Initialisierung mit der aktuellen Systemzeit in der Klasse *ModeEventDriven* belegt. Bei einer Anfrage nach der aktuellen Plattformzeit wird die aktuelle Simulationszeit unverändert zurückgegeben.

[8] Die Systemzeit wird nur zur Initialisierung benutzt.

Die Änderung der Simulationszeit findet nur durch den expliziten Methodenaufruf von *SetNewSimTime(simTime)* (Zeile vier im Ausführungsalgorithmus 5) in der Klasse *ExecutorEventDriven* statt, der als Parameter die neue Simulationszeit fordert. Die neue Simulationszeit entspricht immer dem Zeitpunkt des nächstliegenden Ereignisses. Folglich wird für diesen Simulationsbetrieb die Benachrichtigung der Agenten auf eine etwas andere Weise durchgeführt. Anstatt über die globale Agentenschlange zu iterieren und jeweils zu prüfen, ob ein Agent eine Aktion ausführen darf, wird in dem Algorithmus 5 nur der nächstliegende Agent pro Simulationsrunde benachrichtigt. Dies ist immer der Agent, dessen Ereignis den kleinsten Zeitstempel hat. Somit wird eine Ereigniswarteschlange eingeführt, die alle zukünftigen Agentenereignisse nach dem jeweiligen Zeitstempel der Aktion sortiert. Der Algorithmus springt immer zum nächstliegenden Ereignis, setzt die Simulationszeit auf den Zeitstempel des Ereignisses und führt schließlich das Ereignis aus.

```
   /** Benachrichtige Agenten                                    **/
 1 if Ereigniswarteschlange nicht leer then
 2     event ← GetFirstEvent(Ereigniswarteschlange);
 3     eventTime ← GetTime(event);
 4     SetNewSimTime(eventTime);
 5     agent ← GetAgent(event);
 6     Notify(agent);
 7     Remove(event);
 8     if Führe alle Aktionen eines Zeitpunktes aus then
 9         foreach tmpEvent in Ereigniswarteschlange do
10             tmpEventTime ← GetTime(tmpEvent);
11             if eventTime == newEventTime then
12                 tmpAgent ← GetAgent(tmpEvent);
13                 Notify(tmpAgent);
14                 Remove(tmpEvent);
15             else
16                 break;
17             end
18         end
19     end
20 end
   /** Weiter mit Ausführungsalgorithmus 2                       **/
```

Algorithmus 5 : Benachrichtigung von Agenten im Modus Event-Driven

In Zeile acht erkennt man eine Abweichung der oben beschriebenen Ausführung. Bis jetzt wurde gesagt, dass pro Simulationsrunde immer nur ein Agent ausgeführt wird (Zeile zwei bis sieben). Dies entspricht einer agentenorientierten Ausführung. Allerdings

kann es vorkommen, dass mehr als ein Agent zu dem selben Zeitpunkt eine Aktion ausführen möchte. In diesem Fall ermöglicht der Algorithmus 5 eine Unterscheidung zwischen agentenorientierter oder zeitlicher Ausführung. Bei der agentenorientierten Ausführung wird immer nur ein Agent pro Runde abgearbeitet. In der zeitlichen Ausführung werden innerhalb einer Runde alle Agenten der Reihe nach benachrichtigt, die den gleichen Zeitstempel vorweisen (Zeile acht bis 19). Allerdings wird diese Unterscheidung nur im manuellen Ausführungsbetrieb deutlich.

4.2.2 Historie des Simulationsbetriebes

Die Historienkomponente soll möglichst alle agentenspezifischen Aktivitäten aufzeichnen, die im Interesse eines Simulationsanwenders liegen könnten. Diese Informationen bilden eine wichtige Schnittstelle für mögliche Weiterentwicklungen der Simulationsplattform, wie bereits in Abschnitt 3.4.4 erwähnt wurde. Des Weiteren sollen diese Informationen in der administrativen Oberfläche genutzt werden, um den aktuellen Simulationsverlauf auf einfache Art und Weise darzustellen.

In einem Simulationsablauf sollen alle wichtigen Simulationsereignisse aufgezeichnet werden. Es gibt eine Reihe von unterschiedlichen Ereignissen, die auftreten können. Die Abgrenzung mehrerer Ereignisse voneinander ist keine selbstverständliche Aufgabe, da es keine Definition eines Simulationsereignisses innerhalb der Agententechnologie gibt. In dieser Arbeit werden drei Arten von Simulationsereignissen unterschieden: Eine Simulationsaktion, eine Agentenaktion und eine Agendaaktion.

Der Begriff Simulationsaktion steht für die Ausführung mehrerer Agentenaktionen, die alle zu dem selben Zeitpunkt ausgeführt werden sollen. Eine Agentenaktion setzt sich wiederum aus einer Reihe von kausal abhängigen Agendaaktionen eines Agenten zusammen, dessen Nacheinanderausführung zum selben Zeitpunkt stattfinden soll. Um diese beiden Definitionen zu verstehen, wird im folgenden kurz das Agenda-Konzept der BDI Agenten erklärt, wodurch sich auch der Begriff Agendaaktion erklärt.

Jeder Agent besitzt eine eigene Agenda. Diese Agenda ist als eine Baumstruktur aufgebaut, wobei jeder Knoten des Baumes einer Agendaaktion entspricht. Mögliche Agendaaktionen sind zum Beispiel die Ausführung eines Planschrittes oder die Reaktion auf ein BDI-Ereignis. Der BDI-Interpreter arbeitet die Agenda und somit die interne Baumstruktur eines Agenten ab. Je nach Strategie des Interpreters werden bestimmte Agendaaktionen ausgewählt und ausgeführt, genauso wie einzelne Agendaaktionen entfernt werden können, falls sie nicht mehr aktuell sind.

Anhand dieser Definition lassen sich alle drei Simulationsereignisse klar voneinander abgrenzen. Alle Agendaaktionen eines Agenten gehören zu einer übergeordneten Agentenaktion, die wiederum zu einer Simulationsaktion gehört, solange kein Zeitsprung zwischen den einzelnen Agendaaktionen auftritt. Ein Zeitsprung tritt zum Beispiel auf, wenn

der Agent auf einen bestimmten Zeitpunkt, auf das Eintreffen einer Nachricht oder auf ein anderes Ereignis wartet.

Die Aufzeichnung der Historie wird auf zwei unterschiedliche Arten durchgeführt. Auf der einen Seite werden die gröberen Agentenaktionen aufgezeichnet und auf der anderen Seite wird jede einzelne Agendaaktion einer Agentenaktion erfasst. Aufgrund der Ausführungszeiten können zudem einzelne Agentenaktionen zu einer Simulationsaktion gruppiert werden und zum Beispiel in einer grafischen Oberfläche als zusammengehörig dargestellt werden.

Die Aufzeichnung der Ereignisse findet in der Simulationskomponente *SimExecutor* statt. Bevor eine Agendaaktion ausgeführt wird, wird sie entsprechend aufgezeichnet. Falls die aktuelle Agendaaktion von einem anderen Agenten ausgeführt werden soll, als dem, der die letzte Agendaaktion ausgeführt hat, wird des Weiteren eine neue Agentenaktion vermerkt. Der Start einer Agentenaktion ist genau einem Zeitpunkt zugeordnet. Tritt dieser Zeitpunkt auf, wird der Agent über das Erreichen des Zeitpunktes informiert und führt eine Reihe von Agendaaktionen aus, die sich mit keinen Agendaaktionen eines anderen Agenten überschneiden.

Die Aufzeichnung einer Aktion, egal ob Agenten- oder Agendaaktion, beinhaltet immer das Datum und den Zeitstempel der Aktion sowie den Namen des Agenten und zusätzlich noch den Namen der genauen Agendaaktion. Die Agentenaktion erhält folglich den Namen der ersten Agendaaktion.

Die vorgestellten Granularitätsformen eines Simulationsereignisses werden auch bei den unterschiedlichen Ausführungsgranularitäten berücksichtigt. So liegt es nahe, eine Simulation auch auf drei verschiedene Arten durchlaufen zu können, solange ein manueller Ausführungsbetrieb vorliegt. Entweder wird als Ausführungschritt eine Simulationsaktion, eine Agentenaktion oder eine Agendaaktion gewählt. Die manuelle Ausführung einzelner Agendaaktionen wurde nicht implementiert. Der Grund liegt hierfür vor allem in der hohen Anzahl der Agendaaktionen und der daraus resultierenden Benutzerunfreundlichkeit. Zudem gibt es viele Standard-Agendaaktionen, die für den Simulationsanwender weniger interessant sind. Ein Beispiel einer Agentenaktion mit zugehörigen Agendaaktionen verdeutlicht dies:

$$Agentenaktion(eventX) \Longleftrightarrow ProcessEventAction(event = eventX) \wedge ExecutePlanStepAction(plan = planY) \wedge PlanCleanupAction()$$

Die Agentenaktion, die aufgrund des Ereignisses *eventX* ausgelöst wird, entspricht drei Agendaaktionen, wobei die Agendaaktion *PlanCleanupAction()* für den Simulationsanwender irrelevant ist. In allen Betriebsmodi werden sämtliche Formen von Simulationsereignissen aufgezeichnet. Bei der Ausführung der Simulation, wird die unterschiedliche Ausführungsgranularität nur im Simulationsmodus *event-driven* benutzt. Allerdings bietet dieser Modus aus oben genannten Gründen nur die Einstellung an, um von einer

Simulationsaktion (oder Agentenaktion) zur nächsten zu springen.

4.2.3 Szenarioverarbeitung

Ein Szenario muss individuell vom Simulationsanwender entworfen werden. Hierbei sollte ein Szenario folgende Aufgaben erfüllen:

- Beschreibe Agenten, die an der Simulation teilnehmen sollen.
- Beschreibe Ereignisse, die auf das Multiagentensystem einwirken sollen.
- Stelle einen Zufallsgenerator bereit, um zufällige Ereignisse erzeugen zu können (nutze z.B. eine Normalverteilung).

Das Szenario wird in einem XML-File spezifiziert, wobei die genaue Struktur durch ein XML-Schema festgelegt wird (Listing 4.1, Zeile fünf). Zudem hilft das XML-Schema nicht nur bei der Einhaltung eines wohlgeformten XML-Dokumentes, sondern überprüft auch, ob das spezifizierte Szenario SDF-gültig ist.

Beispiel 4.1 stellt eine gültige SDF-Datei dar. An der SDF-Datei wird deutlich, dass ein Szenario aus drei Teilen besteht. Der erste Teil, die Beschreibung des Szenarios (Listing 4.1, Zeile sieben), ist eine textuelle Beschreibung, die den Inhalt des optionalen *description*-Tags darstellt.

Beschreibung der Szenarioagenten

Der zweite Teil ist für jedes Szenario zwingend erforderlich. Innerhalb des erforderlichen *agents*-Tags, werden eine Reihe von Agenten definiert, die in dem Multiagentensystem agieren sollen und somit auch an der Simulation teilnehmen (Listing 4.1, Zeile neun bis 28). Ein Agent wird durch das *agent*-Tag repräsentiert. Der Agent zeichnet sich durch das Attribut *name* aus, das den eindeutigen Namen des Agenten innerhalb der Agentenplattform darstellt. Zudem besteht das *agent*-Tag aus drei Sub-Tags (type, configuration, arguments), wovon nur das *type*-Tag zwingend erforderlich ist.

Das *type*-Tag bestimmt den Typ des Agenten und verweist auf ein ADF-File, das die genaue Beschreibung und Funktionsweise des Agenten definiert. Weitere Informationen über die Spezifikation von Jadex Agenten finden sich zum Beispiel in Kapitel vier von [BPL06a].

Listing 4.1: Beispiel eines *Scenario Defition File* (SDF)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <scenario xmlns="http://jadex.sourceforge.net/jadex"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://jadex.sourceforge.net/jadex
5 ␣␣http://www.gildhoff.com/jadex/scenario.xsd">
6
```

```
7  <description>load traffic light scenario</description>
8
9  <agents>
10   <agent name="TrafficLightOne">
11     <type>.../agents/trafficlight/TrafficLight.agent.xml</type>
12     <configuration>default</configuration>
13     <arguments>
14       <argument import="...agents.trafficlight.*">
15         new TrafficLight("TLOne", TrafficLight.STATE_RED)
16       </argument>
17     </arguments>
18   </agent>
19   <agent name="TrafficLightTwo">
20     <type>.../agents/trafficlight/TrafficLight.agent.xml</type>
21     <configuration>default</configuration>
22     <arguments>
23       <argument import="...agents.trafficlight.*">
24         new TrafficLight("TLTwo", TrafficLight.STATE_GREEN)
25       </argument>
26     </arguments>
27   </agent>
28 </agents>
29
30 <messages seed="1" maxRange="5000" repeat="10">
31   <message name="inform_car_arrvied">
32     <receiver>TrafficLightOne@lars</receiver>
33     <waitFor>50</waitFor>
34     <parameters>
35       <parameter>
36         <name>performative</name>
37         <value import="...fipa.SFipa">SFipa.INFORM</value>
38       </parameter>
39     </parameters>
40   </message>
41   <message name="inform_car_arrvied">
42     <receiver>TrafficLightTwo@lars</receiver>
43     <waitFor>150</waitFor>
44     <parameters>
45       <parameter>
46         <name>performative</name>
47         <value import="...fipa.SFipa">SFipa.INFORM</value>
48       </parameter>
49     </parameters>
50   </message>
51 </messages>
52
```

```
53 </scenario>
```

Das *configuration*-Tag ist optional. Falls das Tag nicht gesetzt ist, wird der Standardwert *default* angenommen. Dieser Wert beschreibt, in welchem Modus der Agent initialisiert werden soll.

Ferner können für die Initialisierung eines Agenten gewisse Argumente nötig sein. Diese Argumente werden im *arguments*-Tag angegeben. Es können beliebig viele Argumente übergeben werden. Argumente sind Ausdrücke der Programmiersprache Java, die in einem Textformat spezifiziert werden und beim Laden des Szenarios entsprechend ausgeführt werden sollen. Falls hierbei eine bestimmte Java-Klasse importiert werden muss, wird diese im Parameter *import* angegeben.

Beschreibung der Szenarioereignisse und der Zufallsgenerator

Der dritte und letzte Hauptteil eines jeden Szenarios ist der Bereich zwischen den optionalen *messages*-Tags (Listing 4.1, Zeile 30 bis 51). In diesem Bereich werden Nachrichten definiert, die zu gewissen Zeitpunkten abgeschickt werden sollen. Diese Nachrichten kommen somit aus einer abstrakten virtuellen Umgebung, in der das Agentensystem getestet werden soll. Das Szenario bestimmt, wann welche Nachrichten in das Multiagentensystem eingebracht werden. Die Nachrichten sollen somit das Verhalten einer möglichen Umgebung wiederspiegeln, die Einfluss auf das Multiagentensystem hat. Ein einmal definiertes SDF-File kann schnell und unkompliziert modifiziert werden und somit das zu entwickelnde Multiagentensystem unter einer Vielzahl von unterschiedlichen Einflüssen testen. Hierfür muss nur der Bereich *messages* geändert werden und es entsteht sofort ein anderes Szenario.

Im Rahmen dieser Arbeit wurde nur ein einfacher Zufallsgenerator für den Nachrichtenversand eingebaut. Das *messages*-Tag kann mit drei optionalen Attributen ausgestattet werden. Das Attribut *repeat* gibt an, wie oft die spezifizierten Nachrichten versendet werden sollen und die beiden anderen Attribute *seed* und *maxrange* werden von dem Zufallsgenerator benutzt, um die Zeitpunkte der wiederholbaren Nachrichten zu bestimmen. Ist nur der Wert *repeat* angegeben, wird ein standardisierter Zufallsgenerator benutzt.

Der Wert *seed* sorgt dafür, dass der Generator immer mit demselben Wert initialisiert wird. Bei gleichbleibendem Initialisierungswert wird somit garantiert, dass immer dieselben Reihenfolgen von Zufallszahlen zurückgegeben werden, wodurch das Multiagentenszeanrio mit seinem Verhalten wiederholbar[9] bleibt. Der Wert *maxrange* entspricht einer oberen Grenze für den Zufallsgenerator. Seine generierten Zahlen sind im Breich $[0, maxrange]$.

[9] Als Erweiterung sind Zufallsgeneratoren denkbar, die keine wiederholbaren Folgen von Zufallszahlen generieren.

Damit der SCM die spezifizierten Nachrichten auch zu dem gewünschten Zeitpunkt versenden kann, müssen im SDF die Nachrichten mit einen Namen und einer Empfänger-Adresse ausgestattet sein. Des Weiteren können Nachrichten mit beliebig vielen Parametern versehen sein, die im optionalen Sub-Tag *parameters* spezifiziert werden. Dies ist notwendig, damit der Agent die *Sprechakttheorie* einhalten kann. Hierbei handelt es sich um eine Kommunikationstheorie, die die Semantik von Nachrichtentypen festlegt und somit einen komplexen Nachrichtenversand innerhalb der Agententechnologie erst ermöglicht. Obendrein ermöglichen die Parameter eine flexible Möglichkeit für den Benutzer, um Nachrichten mit eigenen Daten versehen zu können, wie zum Beispiel benutzerdefinierte Objekte.

Das Sub-Tag *waitFor* bestimmt den Sendezeitpunkt der Nachricht. Der Sendezeitpunkt ist die Summe aus dem Startwert der Simulationszeit und dem angebenen Wert von *waitFor*. Hat die erste spezifizierte Nachricht zum Beispiel einen Wartewert von 1500, wird diese Nachricht 1500 Millisekunden nach dem Start der Simulation verschickt. Hat die zweite Nachricht einen Wartewert von 2000, wird sie 2000 Millisekunden nach dem Start der Simulation verschickt und entsprechend 500 Millisekunden nach Versendung der ersten Nachricht. Allerdings werden diese absoluten Zeitpunkte nur bei der ersten Versendung einer Nachricht genutzt.

Wurden alle spezifizierten Nachrichten wie beschrieben verschickt, wird die Nachrichtenverschickung so oft wiederholt, wie im Attribut *repeat* angegeben wurde. Vor Beginn jeder Wiederholungsrunde wird der Wert *waitFor* für alle Nachrichten durch einen Zufallsgenerator neu bestimmt. Der zufällig generierte Wert, aus dem Bereich $[0, maxrange]$, wird dabei auf die bereits vergangene Simulationszeit addiert, damit der neue Sendezeitpunkt in der Zukunft liegt, oder zumindest der aktuellen Simulationszeit entspricht.

Szenarioausführung

Das SDF kann als eine Typspezifikation für eine Klasse von instanziierten Szenarien gesehen werden. Beim Laden der SDF-Datei wird die gleiche Technik wie beim Laden einer ADF-Datei verwendet. Der Ladevorgang erzeugt für jedes XML-Element (z.B. scenario, agents, message,...) genau ein Java-Objekt. Die Verbindung von XML-Elementen und Java-Objekten wird mittels JiBX[10] realisiert. Wird ein geladenes Szenario durch den Simulationsanwender gestartet, wird das Szenario-Objekt an den SCM-Agenten übergeben. Dieser reagiert auf die Übergabe, indem er alle beschriebenen Agenten der Reihe nach erzeugt und anschließend den Nachrichtenversand vorbereitet.

Der Ablaufplan 4.5 der Szenarioausführung achtet insbesondere auf einen wiederholbaren Nachrichtenversand. Hierbei bereitet vor allem die Implementierung des Simulationsmodus *time-stepped* Schwierigkeiten. In diesem Modus können benutzerdefinierte Zeitsprünge stattfinden, die die Szenarioverwaltung zu einem vermehrten Nachrichten-

[10] Weitere Informationen unter http://jibx.sourceforge.net/.

versand veranlassen. Anstatt pro Zeitpunkt nur eine Nachricht verschicken zu müssen, kann es in diesem Modus eine Reihe von Nachrichten geben, die aufgrund des Zeitsprunges alle zum selben Zeitpunkt versendet werden sollen.

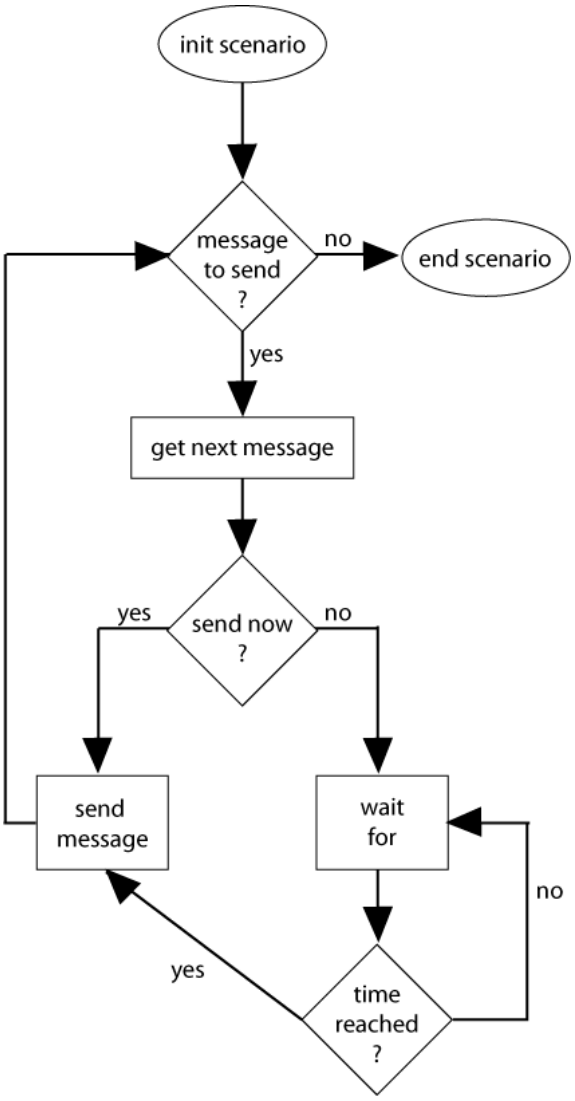

Abbildung 4.5: Ablaufplan der Szenarioausführung

Um einen Nachrichtenversand mit wiederholbaren Folgen von Agentenaktionen garantieren zu können, achtet die Szenariokomponente explizit darauf, in welcher Reihenfolge die Nachrichten verschickt werden und welche bereits verschickt wurden. Somit haben Zeitsprünge in der Simulation keinen negativen Einfluss auf die Reihenfolge der Agentenausführung.

4.3 Administrative Oberfläche

Die administrative Oberfläche (siehe 4.6) gibt dem Simulationsanwender die Kontrolle über die Simulationsplattform. Anhand dieser Oberfläche kann der Anwender Multi-

agentenszeanrios laden und einzelne Simulationsabläufe des Szenarios verwalten. Ein Simulationsablauf kann vom Anwender gestartet, gestoppt oder pausiert werden.

Des Weiteren kann der Anwender für jeden Simulationsablauf einen der vorhandenen Simulationsmodi auswählen. Nachdem eine Simulation gestartet wurde, kann der Simulationsmodus nicht mehr geändert werden, es sei denn der Anwender stoppt den Simulationsablauf.

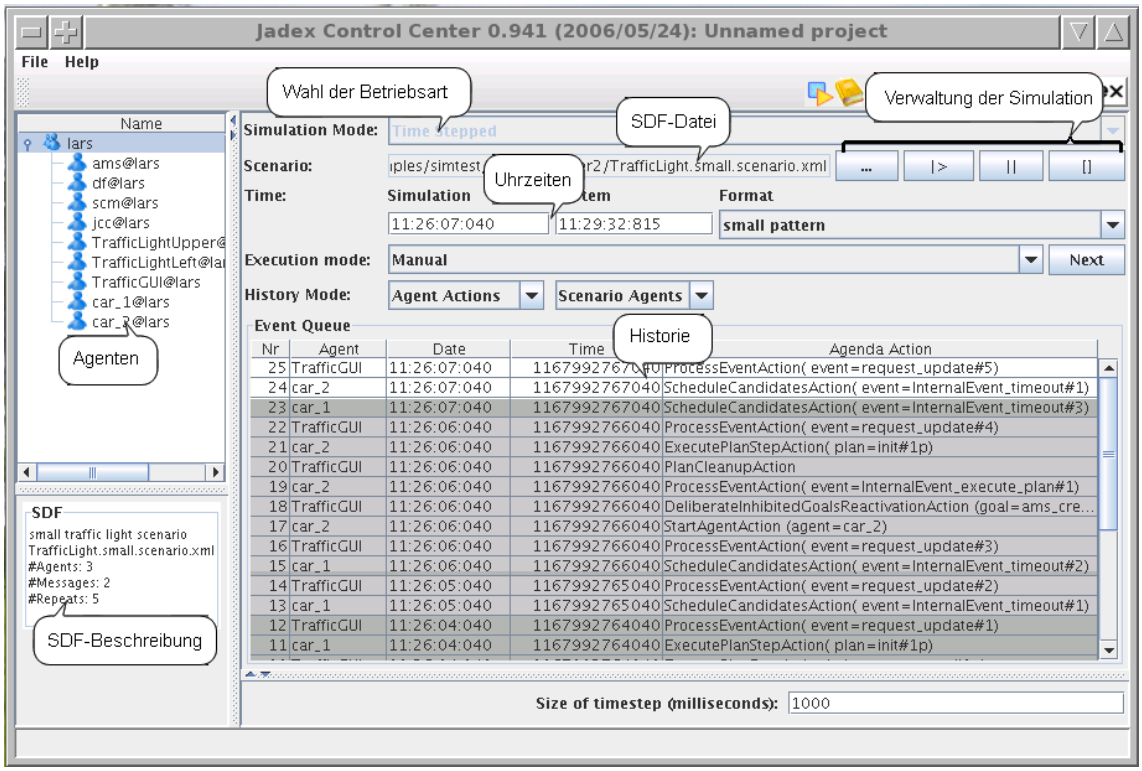

Abbildung 4.6: Jadex Simulationsplattform im Modus *time-stepped*

Die Oberfläche stellt zwei verschiedene Uhrzeiten dar. Zum einen die Simulationszeit und zusätzlich noch die Systemzeit. Beide Zeiten werden automatisch aktualisiert, wenn ein neues Ereignis eintritt. Zudem wird jedes Simulationsereignis in einer Tabelle dargestellt. Hierbei kann der Anwender zwischen zwei verschiedenen Perspektiven wählen, wie es bereits in der Historienkomponente 4.2.2 vorgeschlagen wurde. Es gibt eine Ansicht für die Auflistung aller Agendaaktionen und eine weitere Perspektive für die der Agentenaktionen. Aktionen die zum gleichen Zeitpunkt stattfinden, werden farblich gruppiert. Darüber hinaus kann der Anwender einen Filter benutzen, um entweder nur Aktionen der speziellen Szenarioagenten oder aller Agenten des Multiagentensystems aufzuzeichnen, wozu auch die Systemagenten *SD*[11], *ADS*[12], *JCC* und *SCM* gehören.

Jeder Simulationsmodus bietet eine spezielle Funktion für den Anwender an. Falls der Simulationsmodus *event-driven* oder *time-stepped* ausgewählt wurde, kann der Anwen-

[11] Wird in Jadex *Directory Facilitator* (DF) genannt.
[12] Wird in Jadex *Agent Management System* (AMS) genannt.

der zwischen einem manuellen oder automatischen Betrieb wählen. Zudem kann der Anwender im Modus *scaled real-time* den Skalierungsfaktor und die obere Grenze des Skalierungsfaktor zur Laufzeit ändern und bei *time-stepped* die Größe des Zeitsprungs dynamisch einstellen. Bei *event-driven* kann der Anwender des Weiteren zwischen zwei verschiedenen Ausführungsgranularitäten wählen. Entweder wird ein Simulationsereignis oder eine Agentenaktion als Ausführungseinheit gewählt.

4.4 Zusammenfassung

Der Kern des Jadex-Systems, die BDI Reasoning-Engine, ist lose mit der Adapterschicht gekoppelt. Dies erleichtert die Entwicklung einer eigenen Simulationsinfrastruktur innerhalb der Adapterschicht, die die Simulationsanforderungen erfüllt und trotzdem noch mit BDI-Agenten gekoppelt werden kann, ohne deren interne Struktur anpassen zu müssen.

Die Agenten bestimmen ihre Aktionen selber. Die Simulationsinfrastrukur legt nur den Zeitpunkt fest, wann der Agent seine Aktion ausführen darf. Dabei sind alle Zeitpunkte zeitlich geordnet und die Agentenaktionen werden entsprechend der Reihenfolge nacheinander ausgeführt. Unter der Annahme, dass die BDI Reasoning-Engine bei gleichen Vorausetzungen immer dieselbe Agentenaktion auswählt, garantiert die entworfene Simulationsplattform eine wiederholbare Simulationsausführung, die durch das Zusammenspiel der Ausführungs- und der Zeitkomponente gewährleistet wird.

Damit ein Multiagentensystem auf die unterschiedlichsten Szenarien getestet werden kann, wurde das SDF-Format entwickelt. Die Beschreibung eines Szenarios basiert auf einer XML-Struktur, die teilweise mit spezifischen Details der Programmiersprache Java oder der Agententechnologie ausgestattet wird. Durch diesen Kompromiss hat der Anwender die Möglichkeit, ein Szenario auf einfache Art und Weise entwickeln und komplexere Anweisungen mit Hilfe von Java integrieren zu können.

Die Anforderungen an eine administrative Komponenten konnten in Jadex problemlos umgesetzt werden. Auch die Historienkomponente konnte erfolgreich realisiert werden. Die Historienkomponente zeichnet nicht nur alle Agentenaktionen auf, sondern benutzt auch unterschiedliche Granularitätsformen für die Aufzeichnungen. Somit stehen genügend Informationen für zukünftige Erweiterungen bereit.

5 Anwendungsbeispiel

In diesem Kapitel wird die Anwendbarkeit der Simulationskonzepte auf agentenbasierte Softwareentwicklung anhand eines praxisnahen Beispiels demonstriert. Als erstes wird ein Anwendungsfall bestimmt, in dem ein Multiagentensystem als sinnvoller Problemlöser in Betracht kommt. Nach der Auswahl einer geeigneten Domäne kann das Multiagentensystem mittels der Agentenplattform implementiert werden. Aufgrund der vorhandenen Simulationsunterstützung wird ein iterativer Entwurfsprozess angewendet. Einerseits wird das Multiagentensystem entworfen und weiterentwickelt, wozu insbesondere das Design der Agenten gehört und andererseits wird eine virtuelle Umgebung entwickelt, um bereits entworfene Agentenfähigkeiten in unterschiedlichen Szenarien testen zu können. Anhand einzelner Simulationsabläufe und den daraus gewonnenen Erkenntnissen kann danach wiederum das Verhalten der Agenten weiterentwickelt oder entsprechend angepasst werden.

In Entwicklungsteams werden diese Phasen meistens von denselben Personen abwechselnd durchlaufen. Allerdings ist es auch denkbar, dass sich ein Team von Agentenentwicklern nur um das Design der Agenten kümmert und parallel dazu eine Simulationsteam die Analyse der Simulationsabläufe vornimmt.

Am Ende eines iterativen Entwurfsprozess ergibt sich ein analysiertes Multiagentensystem dessen Vor- oder Nachteile dem Agentenentwickler und Simulationsanwender verdeutlicht wurden. Somit fällt die Entscheidung leichter, ob das Multiagentensystem in der realen Umgebung eingesetzt werden soll. Das Multiagentensystem wurde in einer Vielzahl unterschiedlicher Szenarien getestet und somit erscheint es nicht mehr als ein komplexes System, dessen Verhalten schwer vorauszusagen oder zu verstehen ist, sondern als das richtige Softwaresystem, um ein konkretes Problem auf die bestmöglichste Art zu lösen.

Wenn das System in der realen Umgebung implementiert wird, startet die dritte Phase des Entwicklungsprozesses. Hierbei werden die entworfenen Agenten nicht mehr verändert. Das Multiagentensystem bekommt nur eine neue Umgebung zugewiesen. Anstatt in einer Simulationsumgebung abzulaufen und auf Ereignisse des spezifizierten Szenarios zu reagieren, wird das System in die reale Umgebung integriert. Zukünftige Ereignisse kommen also aus der realen Umgebung, die je nach Anwendungsgebiet aus den verschiedensten Strukturen bestehen kann und nun mit dem Multiagentensystem verbunden werden muss. Danach kann das Multiagentensystems in der Praxis verwendet werden.

5.1 Auswahl einer Domäne

Das gesuchte Anwendungsbeispiel soll aus einer Domäne kommen, in der Systeme existieren, die wichtige und typische Eigenschaften eines Multiagentensystems aufweisen. Hierzu gehört insbesondere die Annahme, dass das System aus individuellen Entitäten besteht, die auf komplexe Art und Weise miteinander interagieren können, so dass sich diese komplexe Interaktion nicht ohne weiteres in einem zentralisierten System abbilden lässt. Zudem sollte das System in einer dynamischen Umgebung liegen, die unerwartete Ereignisse produzieren kann, damit das zukünftige Multiagentensystem auch ausgiebig getestet werden kann.

Neben der Anforderung, dass das Originalsystem wichtige Konzepte eines Multiagentensystems aufweisen soll, wird natürlich noch die praktische Anwendbarkeit des zu entwerfenden Multiagentensystems verlangt. Im Gegensatz zu einer agentenbasierten Simulation soll nicht einfach nur ein Modell des Originalsystems entworfen und simuliert werden. Das zu entwickelnde Multiagentensystem kann als Alternative eines Originalsystems entwickelt und später für dieses auch eingesetzt werden.

Die Anforderungen treffen somit auf Softwaresysteme zu, die sich mit speziellen Problemlösungen auseinandersetzen und in denen der Einsatz eines Multiagentensystems als vorteilhaft angesehen werden kann. Des Weiteren kommen Ausnahmedomänen in Frage, die aufgrund ihrer Komplexität mit klassischen Softwarekonzepten noch nicht beherrschbar sind.

Zutreffende Szenarien finden sich zum Beispiel in den Bereichen Verkehr, Transport, Logistik, Fertigung, Ingenieurwesen und noch vielen anderen Gebieten. Diese Arbeit wird als Beispielanwendung ein agentengestütztes Verkehrsleitsystem entwickeln. Die Behörde für Stadtentwicklung und Umwelt in Hamburg arbeitet bereits seit 2004 an der Verbesserung des Verkehrsflusses mit Hilfe von computergestützten Verkehrsleitsystemen. Die Optimierung der Ampelanlagen erhöht nicht nur die durchschnittliche Geschwindigkeit und vermindert die Umweltverschmutzung, sondern erzielt auch einen volkswirtschaftlichen Gewinn durch Kraftstoffersparnisse (nach [fSuU05]).

Laut Olaf Koch[1], von der Behörde fr Stadtentwicklung und Umwelt in Hamburg, besteht das aktuelle Verkehrsleitsystem aus einem hierachischen Modell in drei Ebenen. An oberster Stelle steht ein globaler Strategierechner, der alle Teilnetze einer Stadt auf oberster Ebene koordiniert. Ein Teilnetz besitzt wiederum einen lokalen Strategierechner, der die Verwaltung aller Ampelanlagen innerhalb seines Bereiches übernimmt. Der lokale Strategierechner koordiniert im Schnitt 15 Ampelanlagen in einem ungefähren Umkreis

[1] Die Informationen entstammen einem Telefoninterview mit Herrn Olaf Koch (Dezember 2006).

von 1000 Metern, wobei sich jede Ampelanlage (eine Kreuzung) anhand von Detektoren und Anforderungstaster (z.B. für Fußgänger) prinzipiell selber verwaltet. Der jeweils übergeordnete Strategierechner ist für die Koordination des Verkehrsflusses zwischen den einzelnen Ampelanlagen zuständig.

Das Hauptproblem des Systems sind anfallende Verwaltungsaufgaben. Das System wird ständig überwacht und muss teilweise manuell parameterisiert werden. Hierdurch wurde schnell erkannt, dass eine flächendeckende Einführung für den Gesamtbreich Hamburg nicht möglich ist. Es fallen zu viele Verwaltungsaufgaben an und das System wird für den Menschen nicht mehr beherrschbar. Falsche Eingaben in die Strategierechner können fatale Auswirkungen auf den Verkehrsfluss haben, genau wie der Ausfall eines Strategierechners

Das komplexe Verkehrsleitsystem kann auch als ein Multiagentensystem betrachtet werden, in dem jede Ampel als eine individuelle Entität einer komplexen Umgebung angesehen wird und in der die Ampeln miteinander um die nächste Grünphase verhandeln. Zudem kann die Simulationsunterstützung bei einer Entscheidung hilfreich sein, ob das agentenbasierte Verkehrsleitsystem die Probleme der aktuell verwendeten Technologie adäquat lösen kann (zum Beispiel durch Selbstverwaltung und Robustheit gegenüber Ausfällen) und somit vielleicht auch das flächendeckende Szenario besser beherrschbar macht.

Allerdings sei hiermit darauf hingewiesen, dass bei dem Entwurf der Beispielanwendung dieser Arbeit nicht die optimale Strategiefindung im Vordergrund steht und schon gar nicht die Entwicklung eines besseren Verkehrsleitsystems als das Bestehende. Das ausgewählte Szenario soll vielmehr verdeutlichen, in wie weit die Simulationsunterstüzung den Entwurfsprozess eines Multiagentensystem verbessert, zur Verständlichkeit des Systems beiträgt und den praktischen Einsatz in unterschiedlichen Anwendungsgebieten effizient unterstützt.

5.2 Agentenbasiertes Verkehrsleitsystem

Die verkehrsbasierte Beispielanwendung beschränkt sich auf die Einführung einer Kreuzung. Im Vordergrund steht das Design der Ampel-Agenten. Sie gehören zu dem Teil des Multiagentensystems, das in die Realität direkt übertragen werden soll.

In der realen Umgebung sollen, genau wie bei dem bestehenden Verkehrsleitsystem, Detektoren und Anforderungstaster verwendet werden, um die Ampel-Agenten über den aktuellen Verkehrsfluss informieren zu können. Allerdings werden in der Entwicklungsphase des Multiagentensystems die Autos auch als Agenten realisiert, die somit die Arbeit der Detektoren übernehmen und die Ampeln über entsprechende Verkehrsdaten informieren können (Nachrichten).

Das Multiagentensystem beinhaltet aber nicht nur zwei Agententypen (Ampel, Auto), sondern benutzt auch einen weiteren Hilfsagenten (GUI-Agenten), der den Simulationsablauf visuell darstellt und als Generator für Auto-Agenten eingesetzt wird. Im SDF (siehe Abbildung 5.1) wird somit nur die Erzeugung von zwei Ampelagenten (die jeweils mit einem anderen Zustand initialisiert werden) und einem Manager-Agent beschrieben (Zeile 2-25). Die Erzeugung der Autos wird durch die Nachrichten *createCar* festgelegt (Zeile 27-38) und durch den Zufallsgenerator mehrfach wiederholt.

Listing 5.1: Eine Szenariobeschreibung für das Verkehrsleitsystem

```
1  ...
2  <agents>
3   <agent name="TrafficLight1">
4    <type>..trafficlight/TrafficLight.agent.xml</type>
5    <configuration>default</configuration>
6    <arguments>
7      <argument import="...agents.trafficlight.*">
8      new TrafficLight("TrafficLightUpper", TrafficLight.STATE_RED)
9      </argument>
10   </arguments>
11  </agent>
12  <agent name="TrafficLight2">
13   <type>..trafficlight/TrafficLight.agent.xml</type>
14   <configuration>default</configuration>
15   <arguments>
16     <argument import="...agents.trafficlight.*">
17     new TrafficLight("TrafficLightLeft", TrafficLight.STATE_GREEN)
18     </argument>
19   </arguments>
20  </agent>
21  <agent name="TrafficManager">
22   <type>..gui/TrafficGUI.agent.xml</type>
23   <configuration>default</configuration>
24  </agent>
25 </agents>
26
27 <messages seed="1" maxRange="5000" repeat="2">
28  <message name="createCar">
29  <receiver>TrafficGUI@lars</receiver>
30  <waitFor>3000</waitFor>
31  <parameters>
32   <parameter>
33    <name>performative</name>
34    <value import="jadex.adapter.fipa.SFipa">SFipa.INFORM</value>
35   </parameter>
```

```
 </parameters>
 </message>
</messages>
...
```

Der Ampel-Agent (siehe Abbildung 5.2) bekommt zur Initialisierung ein Java-Objekt vom Typ *TrafficLight* übergeben, indem wichtige Informationen für die Ampel gespeichert werden. Hierduch ist sich der Agent über seinen eigenen Zustand (state) bewusst und weiß genau, wieviele Autos aktuell bei ihm passieren möchten (numberOfCars). Der Ampel-Agent ist auch darüber informiert, wieviele Autos gerade unwiderruflich passieren und sich noch in einer gefährlichen Kollisionszone befinden (numberOfLeavingCars). Der Ampel-Agent wird nicht nur über das Eintreffen eines Autos informiert (informCarArrvied), sondern auch über Autos, die die Kreuzung verlassen (informCarLeft).

Des Weiteren kann ein Ampel-Agent *A* die Nachricht *change* von einem anderen Ampel-Agenten *B* erhalten, wenn *B* die Grünphase für sich beansprucht. In dieser Beispielanwendung bekommt der Ampel-Agent *B* den Zustand Grün, wenn bei ihm mehr Autos als bei *A* warten. Dies Verhandlung ist nur Beispielhaft und kann natürlich noch erweitert werden, damit zum Beispiel ein Auto nicht unendlich lange warten muss (fairness).

Die Nachricht *ask* wird hingegen von *A* nach *B* gesendet, wenn *A* den Zustand Grün hat und soeben eines seiner wartenden Autos die Kreuzung passiert. Mit dieser Nachricht wird *B* informiert, dass er die Verhandlungsphase nochmal anstoßen soll. Ansonsten wird die Verhandlungsphase immer von dem Ampel-Agenten angestoßen, der den Zustand Rot hat und bei dem soeben ein Auto eingetroffen ist.

Listing 5.2: Auszug aus dem ADF eines Ampel-Agenten

```
...
<beliefs>
 <belief name="tl" class="TrafficLight">
  <fact>(TrafficLight)$args[0]</fact>
 </belief>
 <belief name="state" class="int">
  <fact evaluationmode="dynamic">$beliefbase.tl.getState()</fact>
 </belief>
 <belief name="numberOfCars" class="int">
  <fact evaluationmode="dynamic">$beliefbase.tl.getNumberOfCars()</fact>
 </belief>
 <belief name="numberOfLeavingCars" class="Integer">
  <fact>0</fact>
 </belief>
</beliefs>

<plans>
 <plan name="inform_car_arrvied">
```

```
19   <body>new PCarArrived()</body>
20   <trigger>
21    <messageevent ref="informCarArrvied"/>
22   </trigger>
23  </plan>
24  <plan name="inform_car_left">
25   <body>new PCarLeft()</body>
26   <trigger>
27    <messageevent ref="carLeft"/>
28   </trigger>
29  </plan>
30  <plan name="receive_request_state_change">
31   <body>new PCalculateStateChange()</body>
32   <trigger>
33    <messageevent ref="change"/>
34   </trigger>
35  </plan>
36  <plan name="ask_for_state_change_again">
37   <body>new PAskAgainForStateChange()</body>
38   <trigger>
39    <messageevent ref="ask"/>
40   </trigger>
41  </plan>
42 </plans>
43 ...
```

In Abbildung 5.1 wird ein beispielhaftes Verhalten des Multiagentensystems dargestellt. Hierbei handelt es sich um ein einfaches Szenario, das auf sechs Agenten beschränkt ist. Hierzu gehören zwei Ampel-Agenten, ein Manager-Agent und drei Auto-Agenten. Der SCM ist exklusiv, da er in jedem Simulationsbetrieb vorhanden ist und die Umsetzung des Szenarios verwirklicht, sobald der Simulationsanwender den Befehl zum Start der Simulation gibt. Danach initialisiert der SCM die Simulation, indem er die spezifizierten Agenten aus dem SDF der Reihe nach erzeugt.

Nachdem die drei spezifizierten Agenten erzeugt wurden, wartet der SCM auf die Sendezeitpunkte der Nachrichten. In diesem Fall hat der Simulationsanwender eine Nachricht vom Typ *createCar* definiert, die zwei mal wiederholt werden soll. Diese drei Nachrichten werden zu bestimmten Zeitpunkten an den Verkehrsmanager geschickt, damit dieser ein Auto-Agent erzeugt. Der Auto-Agent wird mit individuellen Parametern versehen und in das Multiagentensystem integriert, damit er auf eigene Art und Weise auf die Kreuzung zusteuert. Das Multiagentenszenario gilt als beendet, sobald die letzte Nachricht des SDF verschickt wurde. Der Simulationsablauf ist allerdings noch nicht beendet und wird vom Simulationsanwender genaustens beobachtet. Die Simulation ist erst zu Ende, wenn alle Auto-Agenten das Multiagentensystem verlassen haben.

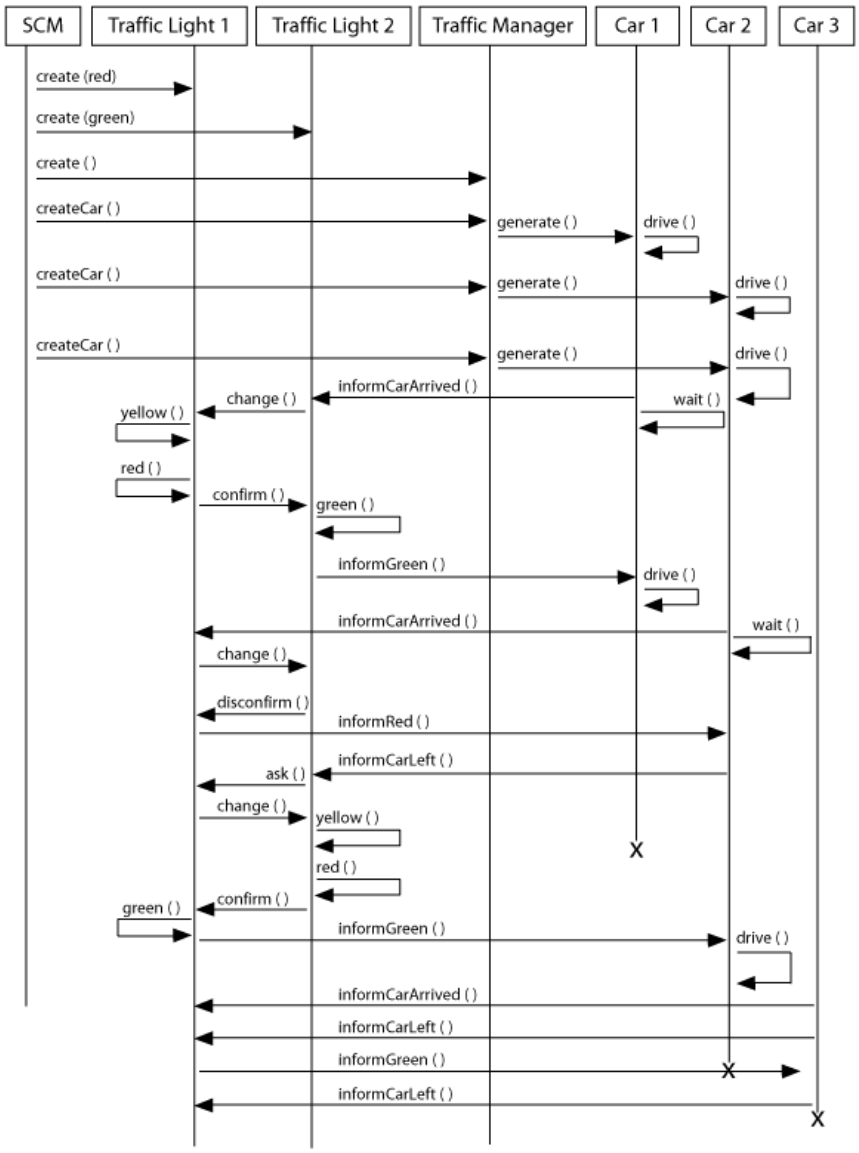

Abbildung 5.1: Beispielhaftes Interaktionsdiagramm der Ampel-Agenten

Das Auto *Car 1* trifft als erstes auf eine Ampel. Es informiert die Ampel *Traffic Light 2* über das Eintreffen und wartet auf eine Antwort der Ampel. Da *Traffic Light 2* den Zustand rot besitzt und mindestens ein Auto auf grün wartet, erkundigt sich der Ampel-Agent bei seinem Nachbarn *Traffic Light 1* nach einem Zustandswechsel. *Traffic Light 1* hat den Zustand grün und hat keine Informationen über mögliche Autos. Deswegen ändert er sofort seinen Zustand, von grün über gelb auf rot. Danach führt *Traffic Light 2* den bestätigten Wechsel aus und sendet die Information grün an das Auto weiter, welches somit über die Kreuzung fahren darf.

Das Auto *Car 2* erreicht als nächstes die Ampel *Traffic Light 1*. Nun fragt dieser Ampel-Agent seinen Nachbarn nach einem Zustandswechsel. Allerdings hat der Nachbar gerade einem Auto erlaubt über grün zu fahren und noch keine Bestätigung erhalten, dass das Auto die Kreuzung verlassen hat. Somit wird der Zustandswechsel abgelehnt. *Traffic*

Light 1 gibt somit den Zustand rot an das Auto Nummer zwei zurück, woraufhin dieses auf grün warten muss.

Mittlerweile hat das erste Auto die Kreuzung verlassen und teilt dies der zuständigen Ampel *Traffic Light 2* mit. Dieser Ampel-Agent erinnert seinen Nachbarn an die zuvor gescheiterte Frage nach dem Zustandswechsel und teilt ihm mit, die Anfrage zu wiederholen, da sich sein Zustand soeben verändert hat. Daraufhin wiederholt *Traffic Light 1* die Anfrage nach einem Zustandswechsel, der bestätigt wird und woraufhin das wartende Auto über den neuen Zustand grün informiert wird. Das Eintreffen des Autos *Car 3* geschieht zur Grünphase von *Traffic Light 2*, wodurch dieses keine Probleme hat die Kreuzung zu passieren.

Um die Vorgänge dieser Beispielanwendung besser präsentieren und beobachten zu können, erzeugt der Manager-Agent die grafische Oberfläche 5.2. Diese grafische Oberfläche ist nicht zwingend erforderlich, da das Verhalten der Agenten auch anhand der gespeicherten Agentenaktionen in der Historienkomponente abgelesen werden kann. Hier kann sogar eine genauere Analyse durchgeführt werden, da nicht jede Agentenaktion eine grafische Veränderung auslöst. Aber solange noch keine allgemeingültige visuelle Komponente existiert, bietet sich eine individuell entworfene Visualisierung der Ereignisse an, um die Simulationsvorgänge zu verdeutlichen und wodurch zum Beispiel auch die Fehlersuche unterstützt wird.

5.3 Auswertung

Das entwickelte Verkehrsleitsystem dient als Beispiel, um die Anwendbarkeit und Vorteile einer Simulationsunterstützung für Agentensysteme aufzuzeigen. Das entworfene Multiagentensystem kann sich nicht mit gängigen Verkehrsleitsystemen messen, wie sie heutzutage verwendet werden. Allerdings scheint das Agentenparadigma durchaus für diesen Anwendungsfall geeignet zu sein. Durch die dezentrale Lösung und die Verwendung von autonom arbeitenden Agenten kann die Stabilität des Systems gesteigert werden. Es gibt keinen zentralen Rechner mehr, der durch einen Ausfall die Funktionsweise des gesamten Systems negativ beeinflussen kann.

Die Vorteile der Simulationsunterstützung wurden beim iterativen Entwurfsprozess sofort deutlich. Durch den schrittweisen Entwurf der Agenten und einer anschließenden Simulation werden Probleme im Design des Agenten sofort deutlich und können sogleich behoben werden. Am Anfang ist das Design des Agenten noch übersichtlich und durch einen schnellen Wechsel der beiden Phasen (Entwicklung und Simulation) werden Fehler sofort erkannt. Werden Agenten jedoch erst getestet wenn sie bereits ein komplexes Verhalten aufweisen, dann ist es deutlich schwieriger die entscheidenden Fehler zu finden.

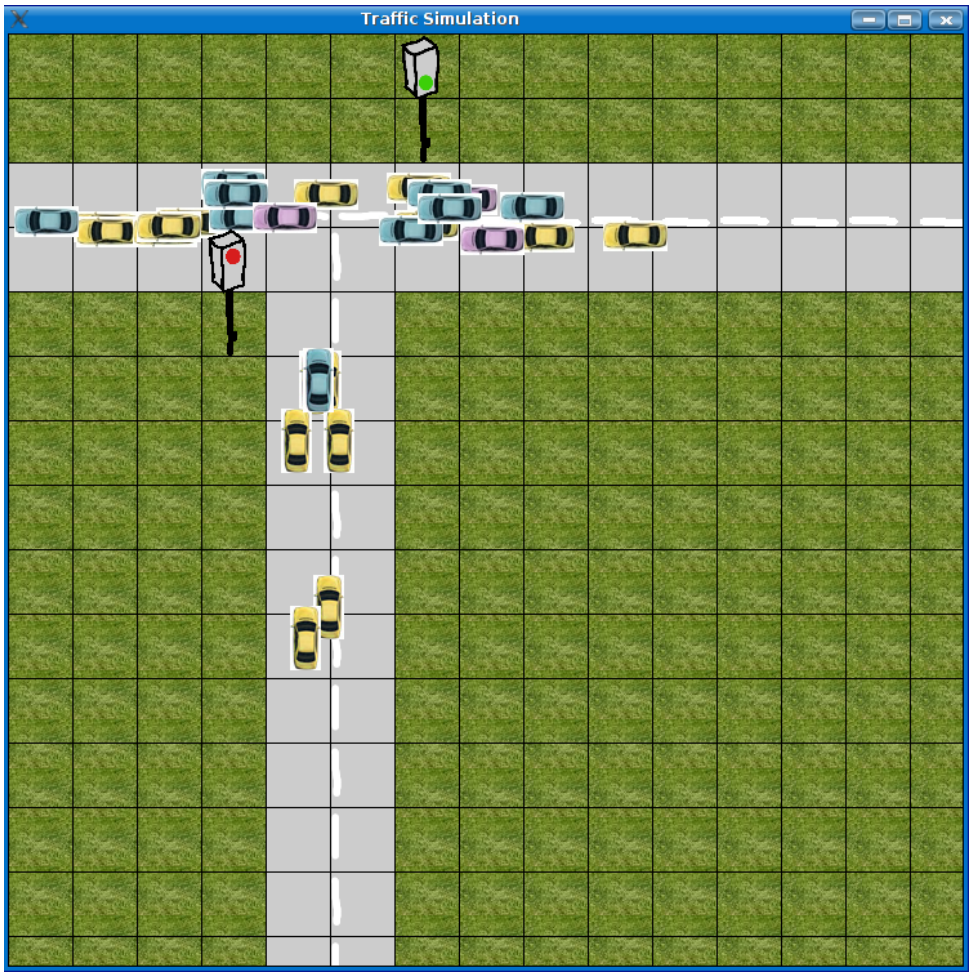

Abbildung 5.2: Visuelle Darstellung der Verkehrssimulation

Die unterschiedlichen Simulationsmodi erlauben die zeitlich gesteuerte Ausführung eines Multiagentensystems. Das Anwendungsbeispiel lässt sich in allen vier Betriebsmodi ausführen. Die beiden Betriebsmodi *time-stepped* und *event-driven* bieten bei der Analyse des Multiagentensystem die besten Möglichkeiten. Sie sind vollständig von der Systemzeit entkoppelt und stellen somit die besten Methoden dar, um das Multiagentensystem schrittweise zu durchlaufen. Die anderen beiden Betriebsmodi haben das Problem, dass selbst bei einer Pausierung der Ausführung die Zeit immer weiterläuft und in der nächsten Ausführungsrunde ein großer Zeitsprung stattfindet.

Der Simulationsmodus *time-stepped* hat den Vorteil, dass der Simulationsanwender mit regelmäßigen Zeitsprüngen die Simulation fortschreiten lassen kann. Allerdings muss der Zeitschritt je nach dem Multiagentenszenario richtig gewählt werden. Ein zu großer Zeitsprung kann die Ausführung von Agentenaktionen zeitlich stark nach hinten verschieben und ein zu kleiner Zeitsprung bewirkt in vielen Zeitsprüngen gar keine Aktion. Folglich muss die richtige Mitte gefunden werden. In dem Anwendungsbeispiel scheint

ein guter Zeitsprung bei 1000 Millisekunden zu liegen. Dies entspricht der Erzeugungszeit der meisten Folgeereignisse, wie zum Beispiel der Zustandswechsel einer Ampel von Gelb auf Rot oder der Fahraktion eines Autos, da der Auto-Agent jede Sekunde eine Fahraktion auslöst (abgesehen von wartenden Autos), die das Auto um wenige Meter fortbewegt.

Der Simulationsmodus *event-driven* ist auch in dieser Beispielanwendung sehr hilfreich. In diesem Modus werden alle Ereignisse (je nach Ausführungsgranularität) einzeln ausgeführt. Allerdings kann dies im manuellen Betrieb ein Nachteil sein, wenn der Simulationsanwender zum Beispiel viele unwichtige Ereignisse einzeln durchlaufen muss. Hier bietet sich dann wieder *time-stepped* an. Am besten eignet sich *event-driven*, wenn die Ereignisse in unregelmäßigen Abständen auftreten. Diese kann man in der Beispielanwendung durch ein Szenario verdeutlichen, in denen die Autos in unregelmäßigen Abständen erzeugt werden. Allerdings bleibt das Problem der Folgeaktionen eines Autos (tritt jede Sekunde auf), die auch hier durchlaufen werden müssen. Folglich gilt, dass sich für dieses Szenario der Modus *time-stepped* am besten eignet, solange der Zeitschritt in einem adäquaten Bereich liegt.

Die erfolgreiche Realisierung des Anwendungsbeispiels hat auch verdeutlicht, dass die Beschreibung mittels SDF nicht immer ausreicht, um eine geeignete Simulationsumgebung für das Multiagentensystem bereitzustellen. Die Simulationsumgebung wird zwar zentral durch das SDF beschrieben, aber es benötigt in den meisten Fällen auch zusätzliche Hilfsmittel oder Agenten innerhalb des Multiagentensystems, die nur für die Realisierung der virtuellen Umgebung gebraucht werden, wie zum Beispiel der Auto-Agent.

In dem agentenbasierten Verkehrsleitsystem besteht die Simulationsumgebung vor allem aus dem Manager- und dem Auto-Agenten. Es ist zwar denkbar, dass beschriebene Verhalten aus Abbildung 5.1 ohne den Manager-Agent und die Autos zu erreichen, indem im SDF keine Nachrichten an den den Verkehrsmanager geschickt, sondern diese sofort an die Ampeln weitergeleitet werden. Allerdings würde dies nicht unbedingt zur Verständlichkeit des Multiagentensystems beitragen. Die Handlungen der Ampel-Agenten können am besten verstanden werden, wenn der Simulationsanwender die Autos in Aktion sieht und nicht nur abstrakt definierte Nachrichten beobachtet.

Das nächste Kapitel fasst abschließend diese Arbeit zusammen und bewertet das Erreichte im Vergleich zu den gesetzten Zielen. Nach diesem Fazit wird zudem ein Ausblick auf mögliche Erweiterungen der Simulationsplattform gegeben. Hierzu gehören Ideen und Verbesserungsvorschläge, mit denen die Simulationsunterstützung für Agentenplattformen noch effizienter umgesetzt werden kann.

6 Zusammenfassung und Ausblick

Abschließend wird die Arbeit zusammengefasst und die Simulationsunterstützung sowie die Umsetzung der einzelnen Betriebsmodi bewertet. In dem nachfolgenden Ausblick werden Möglichkeiten zur Weiterentwicklung der Simulationsunterstützung aufgezählt.

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wird ein allgemeines Konzept vorgestellt, um Agentenplattformen mit einer Simulationsunterstützung auszustatten. Ziel dieser Ausstattung ist es, die agentenbasierte Softwareentwicklung durch den Einsatz der Simulationstechnologie zu unterstützen. Die Simulation von Multiagentensystemen soll den Entwicklern helfen, komplexe Multiagentensysteme besser verstehen und analysieren zu können. Am Ende eines iterativen Entwicklungsprozesses (von Agentendesign und Simulation) soll ein optimiertes Multiagentensystem stehen, das durch mehrfache Testläufe und Validierungen in Bezug auf den zukünftigen Einsatzort ohne Bedenken eingesetzt werden kann.

Die Simulationsunterstützung wird auf Plattformebene entworfen. Somit ist die Implementierung der Simulationsinfrastruktur vor dem Anwender verborgen. Agenten können wie zuvor entworfen werden, ohne dass der Anwender zum Beispiel Rücksicht auf spezifische Simulationsprotokolle nehmen muss. Allein die Beschreibung eines Szenarios wird vom Anwender gefordert, sobald er sein Multiagentensystem simulieren möchte. Allerdings ist die Beschreibung einzelner Szenarien aufgrund des XML-Formates klar verständlich und kann schnell erlernt werden.

Zudem setzt sich die Simulationsinfrastruktur aus weiteren Komponenten zusammen, wobei insbesondere die Ausführungs- und die Zeitkomponente eine wichtige Rolle einnehmen. Die Ausführungskomponente basiert auf einem Thread und führt wiederholbare Agentenaktionen nach den Regeln der Zeitkomponente aus.

Die Realisierung der einzelnen Simulationsmodi basiert auf den theoretischen Grundlagen der Simulationstechnologie. Hierzu gehört insbesondere die Verwaltung der Zeit, die in dieser Arbeit nach gängigen Methoden umgesetzt wurde. Die Realisierung der Wiederholbarkeit und die Ausführung der einzelnen Agentenaktionen basiert auf einem seriellen Betrieb und einem eigenen Ausführungsalgorithmus. Hierfür wird eine eigene Definition von Agentenaktionen eingeführt und zum anderen muss die Annahme erfüllt sein, dass die Agenten unter gleichen Voraussetzungen immer dieselbe Aktion auswählen. Erst dann kann mit den Konzepten dieser Arbeit Wiederholbarkeit garantiert werden (Für Jadex gilt diese Annahme).

Der Betriebsmodus *Normalbetrieb* ist nicht zwingend erforderlich, denn die Realisierung von *scaled real-time* beinhaltet diesen Modus, solange der Skalierungsfaktor *eins* ist. Allerdings ist *scaled real-time* ein Simulationsmodus, der noch stark mit der Systemzeit gekoppelt ist und sich nur teilweise sinnvoll auf die Ausführung von Multiagentensystemen anwenden lässt. Ein Beispiel für eine sinnvolle Anwendung von *scaled real-time* ist, die Simulationsgeschwindigkeit bei interessanten Teilbereichen langsamer und bei uninteressanten Teilbereichen der Simulation schneller voranschreiten zu lassen. Jedoch können diese interessanten oder auch uninteressanten Teilbereiche noch effizienter mit den Simulationsmethoden *time-stepped* oder *event-driven* untersucht werden. Sie sind nicht von der Systemzeit abhängig und können manuell ausgeführt werden.

Die Simulationsunterstützung auf Plattformebene ermöglicht eine Simulation aller Multiagentensysteme, die mit der Agentenplattform entworfen werden können. Jedoch ist die Anwendung der Simulationsinfrastruktur nicht immer sinnvoll, da nicht jede Anwendung eine Simulation benötigt.

Die Simulationsunterstützung wird in Jadex transparent für den Anwender gehalten. Er kann sie nutzen, brauch es aber nicht. Jadex stellt nicht nur gängige Funktionen einer Agentenplattform bereit, sondern erweitert diese auch um einen Mechanismus für Multiagentensimulationen. Somit können Multiagentensysteme nicht nur entworfen, sondern zugleich auch simuliert werden.

6.2 Ausblick

Die entworfene Simulationsplattform von Jadex kann mit einer Reihe von weiteren Komponenten ausgestattet werden (siehe Abschnitt 3.4.4), die zum Standard von gängigen Simulationsprodukten gehören. Die Historienkomponente bietet die hierfür benötigte Schnittstelle an. Sie zeichnet alle Agentenaktionen auf, so dass zum Beispiel eine mögliche visuelle Komponente diese Daten graphisch verarbeiten kann. Weitere Komponenten sind zum Beispiel für die automatische Analyse der Simulationsergebnisse oder für die Visualisierung des Simulationsablaufes zuständig.

Die Szenariokomponente wurde realisiert und ist funktionsfähig. Sie bietet sich besonders für Erweiterungen an. Jedoch wird die Simulationsumgebung meistens nicht nur durch das SDF beschrieben, sondern sie besteht auch aus einer Reihe von Objekten und Agenten, die zusätzlich entworfen werden, wie zum Beispiel die Autos in dem agentenbasierten Verkehrsleitsystem. Hier wäre es denkbar eine Reihe von universalen Umgebungsdiensten in Form von Objekten und Agenten bereitzustellen, die je nach Szenario genutzt werden können. Diese Umgebungsdienste sollen typische und immer wieder benötigte Fähigkeiten einer Umgebung so beschreiben, dass sie von unterschiedli-

chen Multiagentensystemen benutzt werden können. Folglich können immer komplexere Umgebungen entwickelt werden und der einfache Simulationsanwender müsste nicht zwangsweise eine eigene Simulationsumgebung entwerfen, sondern könnte auf bereits existierende Umgebungsdienste zurückgreifen.

In diesem Zusammenhang ist auch ein Modellierungswerkzeug für Umgebungen denkbar. Der Simulationsanwender braucht keine fest definierte Simulationsumgebung auszuwählen, sondern kann mit Hilfe eines Modellierungswerkzeuges seine eigene Simulationsumgebung gestalten, indem er vorgegebene Entitäten auswählt und gegebenenfalls miteinander in Verbindung setzt.

Andere Erweiterungsmöglichkeiten finden sich in der Realisierung des Nachrichtenversandes. Der Nachrichtenversand bildet die Grundlage für die Agentenkommunikation. In der Simulationsplattform von Jadex wird ein lokaler Nachrichtenversand als Methodenaufruf implementiert. Folglich vergeht keine Zeit bei einem lokalen Nachrichtenversand. Interessant wäre es jedoch, wenn der Simulationsanwender die Versanddauer und Verlustrate von Nachrichten in dem administrativen Bereich einstellen könnte. Diese Einstellungsmöglichkeiten einzelner Simulationsabläufe sind insbesondere für reale Einsatzumgebungen interessant, in denen der Nachrichtenversand zwischen den Agenten problematisch ist und nicht immer garantiert werden kann. Dies gilt insbesondere für Multiagentensysteme, deren Agenten an physikalisch unterschiedlichen Orten interagieren.

Als nächster Schritt wäre somit auch eine Erweiterung der Simulationsunterstützung auf verteilte Agentenplattformen denkbar. In diesem Fall reicht die lokale Simulationsunterstützung einer Agentenplattform nicht mehr aus. Ein wiederholbarer Simulationsablauf, der auf mehreren Agentenplattformen arbeitet, muss auf globaler Ebene zum Beispiel durch *Gateway*-Agenten (siehe [KNB+03]) synchronisiert werden.

Jadex zeichnet sich vor allem durch die lose gekoppelte BDI Reasoning-Engine aus, die durch die Adapterschicht mit verschiedenen Agentenplattformen verknüpft werden kann. Allerdings basiert die Realisierung der Simulationsplattform auf der alleinstehenden Plattform von Jadex und in der Adapterschicht wurde nur die Realisierung der Zeitkomponente gefordert. Alle weiteren Komponenten wie zum Beispiel die Ausführungskomponente sind komplett in der alleinstehenden Simulationsplattform implementiert. Somit können adaptierte Agentenplattformen die Simulationsunterstützung nicht nutzen, da ihre Ausführungskomponente zum Beispiel keine Wiederholbarkeit garantieren kann. Ein wiederholbarer Simulationsablauf wird im Rahmen dieser Arbeit nur durch die alleinstehende Simulationsplattform von Jadex garantiert.

Abkürzungsverzeichnis

ABM ..... Agent Based Modelling
ABS ..... Agent Based Simulation
ABSS ..... Agent Based Social Simulation
ACL ..... Agent Communication Language
ADF ..... Agent Description File
ADS ..... Agent Directory Service
AI ..... Artificial Intelligence
API ..... Application Programming Interface
BDI ..... Belief Desire Intention
FIPA ..... Foundation for Intelligent Physical Agents
JADE ..... Java Agent Development Framework
Jadex ..... JADE Extension
MAS ..... Multi-agent System
MTS ..... Message Transport Service
NASA ..... National Aeronautics and Space Administration
PRS ..... Procedural Reasoning System
Repast ..... Recursive Porus Agent Simulation Toolkit
SCM ..... Scenario Manager
SD ..... Service Directory
SeSAm ..... Shell for Simulated Agent Systems
VSIS ..... Verteilte Systeme & Informationssysteme
XML ..... Extensible Markup Language

Abbildungsverzeichnis

Literaturverzeichnis

[BPL05a] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. *Proceedings of the Third International Workshop on Programming Multi-Agent Systems (ProMAS'05)*, pages 99–114, 2005.

[BPL05b] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI-Agent System combining Middleware and Reasoning. *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168, 9 2005.

[BPL06a] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: User Guide, 2006. `http://vsis-www.informatik.uni-hamburg.de/projects/jadex/jadex-0.94x/userguide/`.

[BPL+06b] L. Braubach, A. Pokahr, W. Lamersdorf, Krempels, and Woelk. A generic time management service for distributed multi-agent systems. *Applied Artificial Intelligence*, 2 2006.

[Bra87] M. Bratman. Intentions, plans and practical reason. *Harvard University Press, Cambridge, MA,*, 1987.

[Bro86] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23, April 1986.

[Ced02] L.-E. Cederman. Endogenizing geopolitical boundaries with agent-based modeling. *Proc. Natl. Acad. Sci.*, 2002.

[Con05] World Wide Web Consortium. XML in 10 Punkten, 03 2005. `http://www.w3c.de/Misc/XML-in-10-points.html`.

[FIPA02a] Foundation for Intelligent Physical Agents. FIPA Abstract Architecture Specification, December 2002. `http://www.fipa.org/specs/fipa00001/SC00001L.html`.

[FIPA02b] Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification, December 2002. `http://www.fipa.org/specs/fipa00061/SC00061G.html`.

[fSuU05] Behörde für Stadtentwicklung und Umwelt. Intelligente Ampeln machen den Verkehr schneller und flüssiger. hamburg.de, 2 2005. `http://fhh.hamburg.de/stadt/Aktuell/pressemeldungen/2005/februar/03/2005-02-03-bsu-ampeln.html`.

[Fuj00] R. M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons Inc., 2000.

[GB02] N. Gilbert and S. Bankes. Platforms and methods for agent-based modeling. *Proc. Natl. Acad. Sci.*, 2002.

[GHJV98] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Inc., 15th edition, 1998.

[Glü05] Steffen Glückselig. Holonische Multiagentensimulation. Bayerische Julius-Maximilians-Universität. Institut für Informatik., März 2005.

[Gro98] Swarm Development Group. *Swarm User Guide*, 1998. `http://www.swarm.org/`.

[HENR03] D. Hales, B. Edmonds, E. Norling, and J. Rouchier. *Multi-Agent-Based Simulation 3*. Springer-Verlag, july 2003.

[Jen01] N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM, Vol.44, No. 4*, pages 35–41, 2001.

[JW98] N. R. Jennings and M. J. Wooldridge. *Agent Technology: Foundations, Applications and Markets*. Springer Verlag, 1998.

[KHF+05] F. Klügl, R. Herrler, M. Fehler, C. Triebig, and G. Andriotti. *More than a simple Tutorial about SeSAm*, June 2005. `http://www.simsesam.de/`.

[Kli85] G. J. Klir. *Architecture of Systems Complexity*. Sauders, 1985.

[Klü00] F. Klügl. *Aktivitätsbasierte Verhaltensmodellierung und ihre Unterstützung bei Multiagentensimulationen*. Bayerische Julius-Maximilians-Universität Würzburg, 2000.

[Klü01] F. Klügl. *Multiagentensimulation*. Addison-Wesley Verlag, 2001.

[KNB+03] K.-H. Krempels, J. Nimis, L. Braubach, A. Pokahr, R. Herrler, and T. Scholz. Entwicklung intelligenter Multi-Multiagentensysteme - Werkzeugunterstützung, Lösungen und offene Fragen. *Informatik 2003 - 33. Jahrestagung der GI*, 2003.

[LI81] München Lexikographisches Institut. *Der Große Knaur*. Lexikographisches Institut, München, 1981.

[Lin06] S. Linstaedt. Integration von Agentenplattformen in Middleware - am Beispiel von Jadex und Java EE. Universität Hamburg. Fakultät für Mathematik, Informatik und Naturwissenschaften, 04 2006.

[Ltd05] Agent Oriented Software Pty. Ltd. JACK Intelligent Agents - JACK Sim Manual, 6 2005. http://www.agent-software.com.

[MN05] C. M. Macal and M. J. North. Tutorial on agent-based modeling and simulation. *Winter Simulation Conference*, 2005.

[Mül96] J. P. Müller. *The design of intelligent agents: A layered approach*. Springer, 1996.

[Net06] NetLogo. *NetLogo User Manual*, 3.1.3 edition, 2006. http://ccl.northwestern.edu/netlogo/.

[NS76] A. Newell and H. A. Simon. Computer science as empirical enquiry: Symbols and search. *Communictaions of the ACM*, pages 113–126, 1976.

[Pag91] Bernd Page. *Diskrete Simulation*. Springer-Verlag, 1991.

[PBL03] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP - in search of innovation (Special Issue on JADE)*, 2003.

[PK05] B. Page and W. Kreutzer. *The Java Simulation Handbook - Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 2005.

[PLC00] B. Page, T. Lechler, and S. Claassen. *Objektorientierte Simulation in Java*. Libri Books on Demand, 2000.

[PW04] L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems - A Practical Guide*. John Wiley & Sons Inc., 2004.

[RG95] A. S. Rao and M. P. Georgeff. BDI-Agents: From theory to practice. *First International Conference on Multi-Agent Systems (ICMAS-95)*, 1995.

[RN95] A. J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[Sau99] T. Sauerbier. Theorie und praxis von simulationssystemen, 1999.

[Sta06] StarLogo. Getting started, 12 2006. http://education.mit.edu/starlogo/gettingstarted/getting_started.html.

[Sud04] J. Sudeikat. Betrachtung und Auswahl der Methoden zur Entwicklung von Agentensystemen. Hamburg, Universität, FB Informatik, 2004.

[Tea06] Repast Development Team. Repast Homepage - Recursive Porus Agent Simulation Toolkit, December 2006. http://repast.sourceforge.net/.

[TvM07] A. S. Tanenbaum and S. v. Maarten. *Distributed systems : principles and paradigms*. Pearson/Prentice Hall, 2007.

[WJ94] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, pages 35–41, October 1994.

[Woo02] M. J. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons Inc., 2002.

[Yil06] L. Yilmaz. Agent-Directed Simulation, 12 2006. `http://www.eng.auburn.edu/~yilmaz/ADS.html`.

[Zei76] Bernard P. Zeigler. *Theory of Modeling and Simulation*. John Wiley & Sons Inc., first edition, 1976.

[ZPK99] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, second edition, 1999.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____