Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Vitalij Freese

Portierbarkeit eines Multiagentensystems am Beispiel von Jadex und Whitestein LS/TS

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Vitalij Freese

Portierbarkeit eines Multiagentensystems am Beispiel von Jadex und Whitestein LS/TS

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Wolfgang Renz
Zweitgutachter : Prof. Dr. Michael Neitzke

Abgegeben am 31. August 2009

Vitalij Freese

Thema der Bachelorthesis

Portierbarkeit eines Multiagentensystems am Beispiel von Jadex und Whitestein LS/TS

Stichworte

Software-Agenten, Multiagentensystem, Simulation, Transportlogistik

Kurzzusammenfassung

Die zunehmende Automatisierung in Bereich Transport und Logistik macht den Einsatz von Multiagentenplattformen zur verhandlungsbasierten Simulation und operativen Steuerung verteilter Systeme zu einem attraktiven Forschungs- und Entwicklungsfeld.

Nichtfunktionale Eigenschaften wie Systemstabilität und Skalierbarkeit hin zu großen Systemleistungen sind entscheidend für die Wahl der Plattform. Da deren Programmiermodelle bislang nicht standardisiert sind, wird hier die Portierbarkeit eines Simulationssystems von der open-source Plattform Jadex auf die kommerzielle Plattform Whitestein LS/TS untersucht. Hier konnte aus einem prototypischen Simulationssystem auf einem Rechner ein skalierbares Simulationssystem für ein Rechnercluster entwickelt werden.

Vitalij Freese

Title of the paper

Portability of a Multi-Agent System using the Example of Jadex and Whitestein LS/TS

Keywords

Software Agents, Multi-Agent System, Simulation, Transport Logistics

Abstract

The increasing degree of automation in the transport and logistics domain makes the usage of multi-agent platforms for simulation and operation of distribute systems an attractive field of research and development.

Non-functional properties like system stability and scalability towards large systems are crucial for the choice of the platform to use. Since the programming models have not yet been standardized so far, the portability of a simulation system from the open source platform Jadex to the commercial platform Whitestein LS/TS is explored here. As result, a prototypical single computer simulation system has been ported and thereby extended to a scalable simulation system suitable to run on a computing cluster.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AML	Agent Modeling Language
AMS	Agent Management System
AO	Agent Orientierten
BDI	Beliefe-Desire-Intention
BE	Business Edition
CAL	Core Agent Layer
DAO	Data Access Object
DF	Directory Facilitator
EE	Enterprise Edition
EJB	Enterprise Java Beans
FIPA	Foundation for inteligent Physical Agents
FOPL	First-Order Predicate Logic
HQL	Hibernate Query Language
HUB	Hauptumschlagbase
IDE	Integrated development environment
JADE	Java Agent Development Framework
Jadex	Java Agent Development Framework Extension
JDBC	Java Database Connectivity
LISA	Library for Interactions and Structured Actions Enginem
LS/TS	Live System Technology Suite
MARGE	Multi-Agent Reasoning based on Goal-oriented Execution
MbFSIP	Multiagentensystem-basiertes Framework zur Simulation logistischer Prozesse
MDAL	Message Dispatching Agent Logic
MMLab	Labor für Multimediale Systeme
OO	Objekt Orientierten
ORM	Objektrelationales Mapping
OQL	Object Query Language
PE	Personal Edition
RE	Rune-time Environment
RMI	Remote Method Invocation
RPC	Remote Procedure Calls

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Die Motivation

Mit „Intelligenten Agenten“ ist ein neues Zeitalter in der Softwareentwicklung eingebrochen. Dank ihrem interdisziplinären Charakter finden Agenten Einfluss in Disziplinen der Wissenschaft und dringen in viele Bereiche unseres Lebens (vgl. [Brenner et al 1997]). Ihr Einsatz findet in der Autoindustrie, Energieversorgung, Internet, Telekommunikation, statt und das ist bei weitern keine vollständige Liste. Agenten übernehmen die Aufgaben eines Menschen in der Industrie als Roboter, sie handeln in unserem Auftrag auf dem elektronischen Markt und fliegen unbemannte Fluggeräte.

Solche Eigenschaften, wie Autonomie der Agenten und ihre interaktive Fähigkeit in einem verteilten *Multiagentensystem (MAS)* haben auch einen Platz in der Simulation gefunden.

Diese Multiagentensimulationsrichtung weckte das wirtschaftliche Interesse insbesondere im Bereich Logistik. So wurde im *Labor für Multimediale Systeme (MMLab)* Rahmen einer Bachelorarbeit im Studiengang Angewandte Informatik am Departement Informatik der Fakultät Technik und Informatik der Hochschule für *Angewandte Wissenschaften Hamburg (HAW)* 2007 mit Unterstützung der Firma *portrix.net GmbH*[1] von W. Ruwinski und D. Timotin ein Multiagentensystem-basiertes Framework zur Simulation logistischer Prozesse entwickelt. Das Ziel der Arbeit war die Untersuchung der Potenziale und Perspektiven von MAS im Bezug auf eine transportwirtschaftliches Planungsproblem und Entwicklung eines Frameworks, das die benötigten Mittel zum Untersuchen des Problems zur Verfügung stellt (vgl. [Ruwinski und Timotin 2007]). Das entwickelte Softwareprodukt wurde mit einem Prototyp abgeschlossen, der eine verteilte Agentenanwendung mit einer Multiagentensimulation, einem Steuerungstool und einem Analysetool zum Auswerten der Simulationsdaten enthält.

In der Simulation wurde das Szenario eines Paketdienstes realisiert, in dem die Pakete unter Aufsicht eines Verhandlungskoordinators mit LKWs handeln, um von einem Umschlagspunkt zu einem anderen transportiert werden. Allerdings enthält die Agentensimulation dieses Prototyps eine gewisse „Grobkörnigkeit“, die aufgrund des hardwaretechnisch begrenzten Agenten entstand (s. Kap. 6 [Ruwinski und Timotin 2007]). Abgesehen von der Anzahl der Agenten im MAS wurde ein permanentes Wachstum der Pakete im System festgestellt, was eine weitere fachliche Untersuchung erfordert.

Die geringe Menge der Agenten und die ständig wachsende Menge der Pakete in der Simulation sind zwei wesentliche motivierende Gründe diese Arbeit fortzusetzen.

1.2 Die Aufgabenstellung

Das Ziel der Arbeit ist es die Portierbarkeit eines implementierten *Multiagentensystem-basierten Frameworks zur Simulation logistischer Prozesse (MbFSlP)* von der *Jadex Standalone Agentenplattform*[2] auf die *Living Systems Technology Suite (LS/TS)* Agentenplattform der Firma

1 http://www.portrix.net (August 2009)
2 http://jadex.informatik.uni-hamburg.de (August 2009)

Whitestein[3] zu analysieren. Der Einsatz der LS/TS Plattform ist eine Vorgabe für diese Arbeit. Die Portierbarkeitsanalyse wird mit Hinblick auf die Realisierungsmöglichkeit und Portierungsaufwand durchgeführt. Für die Analyse werden genommen:

- Das Agentenprogrammiermodell *JADE Extension (Jadex)*, mit dem die *Belief-Diseur-Intention (BDI)* Agenten des Frameworks implementiert wurden;

- von Whitestein entwickelte Programmiermodelle: *Message Dispatching Agent Logic (MDAL), Multi-Agent Reasoning based on Goal-oriented Execution (MARGE), Library for Interactions and Structured Actions Engine (LISA), Semantic Communication (SemCom)*, die einen Einsatz der unterschiedlichen Agentenarchitekturen gewährleisten.

Weiterhin soll die Portierung realisiert werden und mit einem Prototyp abgeschlossen werden, um eine weitere Analyse der logistischen Prozesse der Simulation mit Hinblick auf das Paketwachstum durchzuführen. Die portierte Agentensimulation soll das beschriebene Simulationsszenario (s. Kap. 3.3.7 [Ruwinski und Timotin 2007]) widerspiegeln und die Agentenanwendung soll die gleiche Instrumente zum Parametrisieren der Simulation, Steuern der Simulationsabläufen und Analysieren der Simulationsdaten zur Verfügung stellen. Um maximale Nutzen der Portierung zu gewinnen, soll das Multiagentensystem sowohl lokal auf einer LS/TS Agentenplattform als auch verteilt über mehrere Plattformen ausführbar sein.

Es wird erwartet, dass ein kommerzielles Produkt der Firma Whitestein eine deutlich größere Agentenmenge in einem verteilten Multiagentensystem unterstützt.

1.3 Aufbau der Arbeit

Im ersten Kapitel wurde die Motivation der Bachelorarbeit vorgestellt und die Aufgabenstellung der Bachelorarbeit beschrieben.

Das zweite Kapitel beschafft das Fundament der Arbeit. Dazu gehören die verteilten Systeme und Petrinetze. Mit dem Unterkapitel Software-Lebenszyklus werden Kenntnisse gewonnen, um den Stand des Projektes richtig einzuschätzen. Ein großer Abschnitt des Kapitels wird jedoch der Agenten Paradigma gewidmet, in dem die verschiedene Agenteneigenschaften, sowohl ihre Architekturen als auch das Multiagentensystem und Agentenkommunikation mit ihren Merkmalen, beschrieben werden.

Im dritten Kapitel wird die Arbeit ein „Jadex Multiagentensystem-basiertes Framework zur Simulation logistischer Prozesse“ von W. Ruwinski und D. Timotin vorgestellt. In diesem Teil der Arbeit werden die Architektur des Frameworks, Agentensimulation und ihre Semantik, die Agenten selbst erläutert.

Im Verlauf des vierten Kapitels werden verschiedenen Agentenprogrammiermodellen betrachtet. Am Anfang dieses Abschnittes werden die Grundlagen des Programmiermodells Jadex beschrieben, das zur Implementation des Multiagentensystems des oben erwähnten Frameworks zur Simulation logistischer Prozesse benutzt wurde. Danach wird das Produkt der Firma Whitestein Technology Group AG, ihre Agentenplattform und unterschiedliche Programmiermodelle dieses Produktes vorgestellt.

---

3 http://www.whitestein.com (August 2009)

Im Kapitel fünf wird eine Analyse des Frameworks zur Simulation logistischer Prozesse durchgeführt. Während der Analyse werden die Agenten, ihre Architektur und Multiagentensystem, in dem sie miteinander kommunizieren, unter die Lupe genommen. Im Verlauf dieses Kapitels wird eine Entscheidung zum Auswahl des geeigneten Programmiermodells zum Portieren des Multiagentensystems auf Whitestein Produkt getroffen. Nach der Auswahl wird der Aufwand der bevorstehenden Arbeit festgestellt und die Risikoanalyse der bevorstehenden Portierung durchgeführt. Zum Kapitelschluss werden die Eigenschaften des Frameworks beschrieben, die während des Testens des Frameworks in der Erprobungsphase festgestellt wurden.

Im sechsten Kapitel wird das Design des zu portierenden Frameworks geschildert. Es werden verschiedene Sichten der Architektur im Hinblick auf die Verteilung, Persistenz, Konfiguration und Parametrisierung des Frameworks veranschaulicht. Ebenso werden das Nachrichtenprotokoll mittels Sequenzdiagrammen und das Matchverfahren beschrieben.

Das Kapitel sieben stellt die Realisierung der Portierung des Multiagentensystems auf Whitestein Produkt dar. Die Implementierungsvorgehensweise und die Portierungsregeln werden festgelegt. Es werden auch die Regeln zum Realisieren der Ziele und Techniken zum Nachbilden des proaktiven Agentenverhaltens mittels des MDAL Programmiermodell beschrieben. Es wird sich im Weiteren die Datenbankintegration, XML Konfigurationsparameter und Implementierung der einzelnen Agenten dargestellt.

Das vorletzte Kapitel enthält eine Bewertung der Agentenanwendung nach dem Portierungseinsatz und den durchgeführten Tests des Multiagentensystems mit Hinblick auf die Stabilität, Skalierung und Verteilung.

Im letzten Kapitel wird ein Blick auf die gesamte Arbeit geworfen und eine Zusammenfassung gegeben.

2 Grundlagen

Das Ziel dieser Arbeit ist die Portierbarkeit eins MASs auf eine neue Agentenplattform zu untersuchen. Dies wird am bestens dadurch möglich, die Portierung durchgeführt und das Ergebnis getestet wird. Dazu sollen im ersten Kapitel die theoretischen Grundlagen beschrieben werden, die fürs Erreichen gestellten Ziels notwendig sind.

2.1 Agenten

An erster Stelle der Arbeit werden die Intelligenten Agenten vorgestellt, um zu verstehen was portiert wird. Es wird der Begriff „Intelligenten Agenten" definiert und die Agenteneigenschaften werden soweit dargestellt, wie es für das Verständnis der eingesetzten Agentenplattformen und des portierten Frameworks nötig ist.

In der modernen menschlichen Welt ist das Wort „Agent" sehr bekannt. Die Menschen kennen die Agenten als Händler, Vertreter, die beauftragt sind, ein Teil oder komplette Aufgabe meistens gegen eine Provision zu erledigen. Es kann sich um den Verkauf eines Hauses, Autos oder Aktien an der Börse handeln. Agenten sind in unserem Verständnis die Personen, auf die wir uns verlassen können. Sie sind hoch qualifiziert und garantieren, dass der Auftrag ausgeführt wird.

In der Welt der Wissenschaft gibt es immer noch keine eindeutige Definition für den Begriff „Agent" oder, fachlich richtig zu sein, „intelligenter Agent".[4] Die Agententechnologien befinden sich in einer ständigen Entwicklung und sind als hoch dynamisch eingestuft.

2.1.1 Was ist ein intelligenter Agent?

Einen intelligenten Agenten zu definieren ist äußerst schwierig. Der Versuch einer allgemeinen Definition scheitert nach Meinung R. Zarnekow, weil:

> *„Dies beruht vor allem auf dem interdisziplinären Charakter der Agenten, welcher sich in Einflüssen unterschiedlicher wissenschaftlicher Forschungsrichtungen einerseits und den von der Praxis gestellten Anforderungen anderseits widerspiegelt." [Brenner et al 1997]*

Wooldridge gibt jedoch eine einfache und plausible Definition eines Agenten:

> *„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. " [Wooldridge 1999]*

Einen besseren allgemeinen Überblick von der Menge der existierenden Versuche, einen intelligenten Agenten zu definieren, ist im Kapitel 1.5.1 der Dissertation zu lesen [Raffel 2005].

Trotz der Schwierigkeit einen Agent zu definieren, unterlegen sie alle einer strengen Kategorisierung.

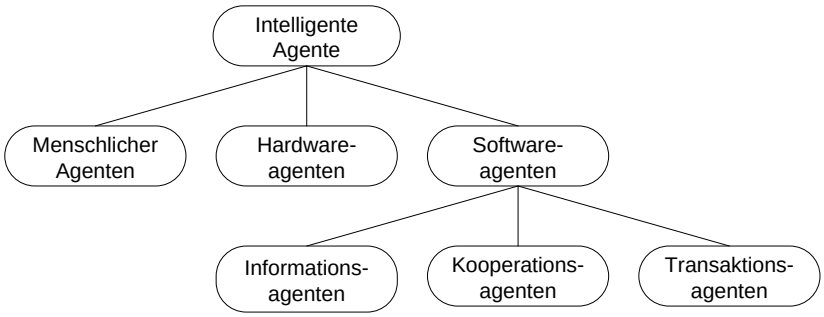
---

4 Im weiteren Verlauf der Arbeit wird kein Unterschied zwischen Agent und intelligenter Agent gemacht.

So sind intelligente Agenten in drei Kategorien aufgeteilt:

- menschliche Agenten, die für uns als Werbeagenten (engl. promotion agent) bezeichnet werden.
- Hardwareagenten, die als Roboter in Industrie- und Servicebereichen sehr verbreitet sind.
- Softwareagenten (s. Abbildung 2.1).

Die Softwareagenten sind keine physikalischen Objekte, sie sind ausschließlich Computerprogramme, die z. B. einen Menschen oder eine Maschine in einer virtuellen Umwelt darstellen. Aufgrund der Herkunft der Softwareagenten, nämlich die Softwarewelt werden die Agenten von Objekten unterschieden. Die beiden Elemente herrschen in Objekt- und Agentenorientierten Welten und besitzen sowohl gemeinsame als auch unterschiedliche Eigenschaften.



Vgl. [Brenner et al 1997]

Abbildung 2.1: Kategorien intelligenter Agenten

Agenten unterstützen genau so wie Objekte das Prinzip der Kapselung (engl. information hiding) (vgl. [Gildhoff 2007]). Sie verbergen ihre Implementierung vor fremden Zugriffen und unerwünschten Einflussfaktoren. Die Kommunikation mit anderen Agenten findet nur über explizit definierte Kommunikationsschnittstellen aber im Gegenteil zu Objekten sind die Agenten im Besitz eines größeren Grades an Autonomie. Ein Objekt ist zwar in der Lage, autonom über seinen internen Zustand zu entscheiden, hat aber keinen Einfluss auf sein Verhalten. Das Ausführen einer Methode eines Objektes passiert zwangsmäßig. Nur ein intelligenter Agent ist in der Lage einen Widerstand zu leisten und eine selbstständige, autonome Entscheidung zu treffen, ob der externer Aufruf (in der Agentenwelt werden anstatt der externen Methoden/Aufrufen die Nachrichten gesendet) ausgeführt werden soll oder nicht. Ein weiterer Unterschied eines Agenten von einem Objekt ist die Fähigkeit unterschiedliche Ziele zu verfolgen. Objekte sind passiv und kommen zu Ausführung nur dann, wenn sie aufgerufen werden, die Agenten sind aktiv und in der Lage von sich allein zur Ausführung aktiviert zu werden (vgl. [Eymann 2000]).

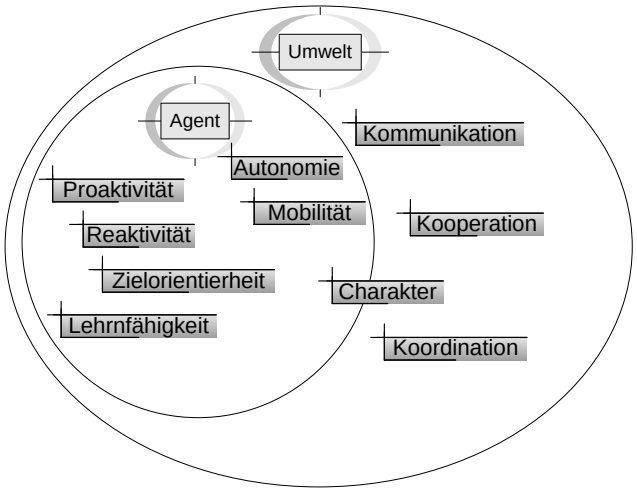
2.1.2 Wozu intelligente Agenten einsetzen?

Was macht die Agenten so interessant? Wo sind die Einflussgebiete der intelligenten Agenten?

Nach der R. Zarnekow Definition (s. o.) findet die Menge der Eigenschaften eines Agenten einen Einfluss in verschiedenen Fachrichtungen statt (s. Abbildung 2.2). Es ist aber nicht unbedingt notwendig, dass ein Agent alle diese Eigenschaften nachweist.

Die Agenten befinden sich in einer permanenten Verbindung mit der Umwelt (engl. environment). Daraus lassen sich zwei Kategorien der Agenteneigenschaften unterscheiden: interne und externe.

Zu den internen Agenteneigenschaften gehören beispielsweise die Autonomie oder Lernfähigkeit, die das „innere Wesen“ eines Agenten bestimmen, und zu den externen gehören, z. B. die, die eine Interaktion zwischen mehreren Agenten oder Agenten und Menschen charakterisieren z. B. Kommunikation (vgl. [Brenner et al 1997]).



Vgl. [Brenner et al 1997]

Abbildung 2.2: Charakteristika intelligenten Agenten

Die Agenteneigenschaften in einzelnen:

Autonomie: Agenten unterliegen keiner (unmittelbaren) Steuerung und Kontrolle durch einen Nutzer. Sie handeln eigenständig, aber „im Sinne ihrer Auftraggeber„ die Auswahl/Planung ihrer Handlung kann (muss aber nicht) nach sehr komplexen Methoden erfolgen [Görz et al 2003].

Reaktivität: Im engeren Sinne wird es als das unmittelbare Reagieren auf Umweltereignisse verstanden. Im Allgemeinen betrifft es die Interaktion mit der Umwelt insgesamt [Görz et al 2003].

Proaktivität: Der Begriff ist verwandt mit Zielgerichtetheit (und wird teilweise gleichwertig benutzt). Eine spezielle Betonung liegt dabei auf der „Eigeninitiative“ des Agenten [Görz et al 2003].

Kommunikation/Interaktion mit einer Umwelt: Agenten nehmen Informationen aus ihrer Umwelt auf (Aufträge, Erfassen von Situationen, Kontrolle ihrer Auktionen). Sie agieren in ihrer Umwelt, um sie (auftragsgemäß) zu beeinflussen [Görz et al 2003].

Kooperation: Agenten arbeiten gemeinsam mit einem Menschen und anderen Agenten um ein globales, gemeinsames Ziel zu erreichen oder eine große, komplexe Aufgabe schnell und effektiv zu erledigen.

Koordination: Agenten arbeiten gemeinsam mit aufeinander abgestimmten Verhalten. Zu diesem Zweck können die Agenten Nachrichten austauschen - explizierte Koordination. Aber auch ohne Kommunikation ist eine Koordination möglich - implizierte Koordination beispielsweise durch das Verändern der Umwelt [Görz et al 2003]. Während der explizierten Form der Koordination klar und einfach ist, verfügt die implizierte Koordination einen deutlich hohen Grad an der

Komplexität, weil der Agent in der Lage ist, das Verhalten eines anderen Agenten oder sogar einer Gruppe der Agenten zu modellieren und daraus seinen nächsten Arbeitsschritt oder eine Entscheidung zu definieren.

Reaktives Verhalten: Damit ist das unmittelbare Reagieren „*Stimulus-Response*“[5] auf die Umwelt gemeint. Es gibt keine Planung, die Mechanismen beruhen auf einfachen Zuordnungen, z. B. in Tabellenform [Görz et al 2003].

Deliberatives Verhalten: „Deliberation“ bezeichnet die explizit modellierte Auswahl von Zielen bzw. Absichten [Görz et al 2003].

Diese längst nicht vollständig aufgeführte und definierte Liste der Agenteneigenschaften definiert Einflussgebiete der intelligenten Agenten in der modernen Wissenschaft. So finden ihren Einsatz die Reaktivität, Proaktivität und Lernfähigkeit eines Agenten in der künstlichen Intelligenz und die Agenten mit Eigenschaften der Kommunikation und Koordination in der verteilten künstlichen Intelligenz. Die Mobilität eines Agenten macht ihn besonders für das Netzwerk interessant (vgl. [Brenner et al 1997]).

Die intelligenten Agenten finden ihren Einsatz fast in jedem Bereich unseres Lebens: Sei es in der letzten Zeit explosionsverbreiteter Internet, E-Commerce, Marketing, Industrie, Logistik, Prozesscontrolling usw. Ihr Einfluss wächst und wird immer größer.

2.1.3 Agentenumgebung

Bei der Betrachtung der Abbildung 2.2 wird festgestellt, dass ein wichtiger Teil der intelligenten Agenten in der Arbeit vernachlässigt wurde. Es wurde nur ein internes Modell eines Agenten, nämlich das System mit allen seinen Eigenschaften beschrieben, die Umgebung oder Umwelt eines Agenten war allerdings fachlich nicht erläutert.

Unter einer Agentenumwelt wird für einen Agenten eine „relevante Umwelt“ gemeint. Das kann eine reale Welt wie z. B. ein abgestürztes Gebäude für einen Rettungsroboter, Internet für einen Informationsagenten - oder eine virtuelle Spielwelt sein. Ein Agent bekommt über ein "Input" - sogenannter Sensor - die aktuelle Information über die Umgebung, und über „Output“ - sogenannter Effektor - verändert der Agent seine Umgebung. Dabei existieren mehrere Ausprägungen der Agentenumgebung [Russell und Norvig 1995]:

Abgeschlossenheit der Umwelt: Die Umwelt ist geschlossen, wenn sie als fixierte Gesamtheit mit festgelegten Situationen betrachtet werden kann. Offene Umwelten erfordern eine höhere Flexibilität der Agenten [Görz et al 2003].

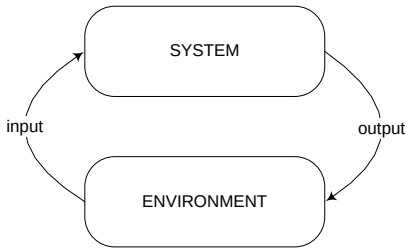
Dynamik der Umwelt: Eine Umwelt heißt „dynamisch“ für einen Agenten, wenn sie sich während des Entscheidungsprozesses eines Agenten ändert, andernfalls ist sie statisch [Görz et al 2003].

5 Jeder Handlung ist ein unmittelbarer „Reflex“ auf ein äußeres „Signal“ [Görz et al 2003].

2.1.4 Klassifikation der Agenten.

Die Agentenumwelt spielt eine große Rolle bei der Klassifizierung der Agenten nach Vidal J.M und Durfee E.H. In ihrem „k-Level Konzept“ unterteilen sie die intelligenten Agenten in drei Levels. Ein wichtiger Aspekt des Konzeptes ist der Prozess, wie das agenteneigene Verhalten auf die veränderte Umgebung oder Verhalten der anderen Agenten angepasst oder abgestimmt wird.

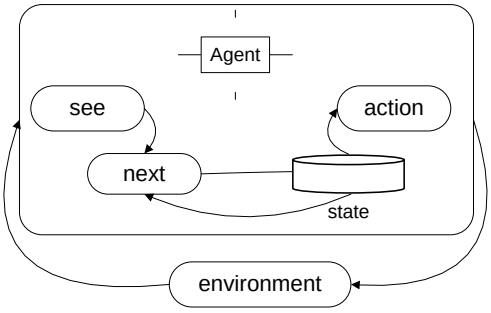
Level-0-Agent: Diese Agenten erhalten einen Anreiz, Eingabe(engl. stimulus), führen eine Auktion(engl. response) aus und werden dafür belohnt (s. Abbildung 2.3). Es sind modelllose Agenten. Die Umwelt stellt für die Klasse der Agenten eine „Black Box“ dar und die anderen Agenten haben keinen Einfluss auf die internen Prozesse eines Agenten (vgl. [Eymann 2003]).



Vgl. [Wooldridge 1999]

Abbildung 2.3: Reaktiver Agent und seine Umgebung

Level-1-Agent: Die Agenten dieses Levels besitzen ein Gedächtnis und ziehen das Verhalten der anderen Agenten in die Entscheidungsprozesse mit ein. Allerdings sind diese Agenten nur in der Lage, ihre eigenen Erfahrungen mit anderen Agenten zu speichern (s. Abbildung 2.4). Aufgrund dieser Daten (vergangenes Verhalten) werden zukünftige Auktionen eines Agenten ausgerechnet.



Vgl. [Wooldridge 1999]

Abbildung 2.4: Agent mit Gedächtnis und die Umgebung nach Wooldridge

Level-2-Agent: Die Agenten dieses Levels verfügen über ein explizites Modell im Vergleich zu den anderen Agenten und sind in der Lage, die Prognosen oder Vorhersagen der Agentenauktion zu generieren. Diese Klasse der Agenten besitzt in der Regel ein symbolisches Modell der Agentenumgebung. Die Agenten sind sehr komplex und benötigen sehr viel Rechenzeit, um ihren nächsten Schritt bzw. die nächste Auktion zu bestimmen. Diese Eigenschaft der Level-2-Agenten macht ihren Einsatz in einer dynamischen Umgebung zu einer besonders schwierigen Aufgabe (vgl. [Eymann 2003]).

Mit der Steigung von Level 0 bis 2 erhöht sich die Agentenkomplexität, die einen direkten Einfluss auf die Portierbarkeit, Zeit und Aufwand der Portierung hat. Deswegen ist es wichtig zu wissen, zu welchem Level die MbFSlP Agenten zugeordnet sind. Jeder Agent realisiert eine bestimmte Architektur, die sich aus Eigenschaften der Agenten ergibt.

2.1.5 Agentenarchitekturen

Grundlegend sind zwei Architekturen aus der Menge der Agenteneigenschaften zu unterscheiden: reaktive- und deliberative Agentenarchitektur.

Reaktive Agent ist der einfachste Agent von allen. Agenten dieser Architektur existieren nach „Stimulus-Response-Architektur" Prinzipien: Eine Auktion wird nur dann erfühlt, wenn ein externer Anreiz existiert. Diese Agenten besitzen kein Gedächtnis. Intern verfügen die Agenten über keine komplexen Entscheidungsmechanismen, stattdessen besitzen sie eine exakt erfasste Liste mit allen Reflexhandlungen/Auktionen auf konkrete Signale/Anreize, die auftreten können. Die Signale/Anreize oder auch Umweltveränderungen, die nicht erfasst wurden, werden nicht bearbeitet, sondern einfach ignoriert. Die Listen mit möglichen Auktionen eines Agenten sind in der Regel als Tabellen oder neuronale Netze realisiert. Das Bestimmen von Agentenaktivitäten wird mit Hilfe einer Vergleichsoperation (Match-Implementation) realisiert. Die reaktiven Agenten können durchaus sehr komplexe Berechnungsoperationen oder Zuordnungsfunktionen implementieren (vgl. [Görz et al 2003]).

Deliberative/Kognitive Agent unterscheidet sich von einem reaktiven Agenten, indem diese Agentenarchitektur über ein explizites, symbolisches Modell der Agentenumwelt besitzt. Dieses Umweltmodell wird in der Regel vorab erzeugt und bildet ein Fundament eines Agentenwissens (vgl. [Brenner et al 1997]). Die deliberative Agenten verfügen auch über ein internes Wissen und ein sehr komplexen Zustandsmodell sogenannter *„mentalen Zustand".* Ein weiterer Unterschied der Agenten dieser Architektur ist die Fähigkeit zur logischen Schlussfolgerung, die auf der Basis eines Umweltmodells und ihrem Agentenzustand inklusive internen Wissens stattfindet (vgl. [Brenner et al 1997]). Die Agenten dieser Architektur gehören zu Level-2-Agenten der Klassifikation (vgl. [Görz et al 2003]).

Zu den deliberativen Agenten gehören zielorientierte Agenten und BDI Agenten.

2.1.6 Zielorientierte Agenten

Die zielorientierten Agenten besitzen ein Umweltmodell, internes Wissen, Ziele und Pläne. Diese Agenten treffen eine Entscheidung um ein Ziel zu verfolgen, das zur aktuellen Zeit am geeignetsten für das Erreichen eines bestimmten Umweltzustandes oder agenteninternes Zustandes ist. Die Ziele werden in einem Agent nebenläufig verfolgt.

> *„In Zielen bzw. der Fähigkeit zur Zielverfolgung drücken sich die Proaktivität und Adaptivität eines Agenten aus." [Pokahr 2007]*

Nach der Zielauswahl folgt ein Verfahren, um einen zweckmäßigen Plan zu finden, der diesen Agenten zum Erreichen seines Ziels näher bringt. Der ganze Entscheidungsprozess wird als *„practical reasoning"* bezeichnet und besteht aus zwei Teilprozessen: „*deliberation"* das Festlegen eines Ziels und „*means-ends reasoning"* zum Bestimmen einer Auktion, nämlich eines Planes (vgl. [Görz et al 2003]).

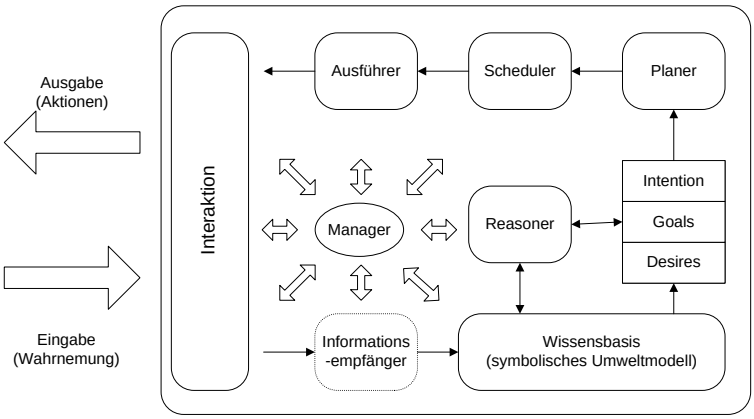
Die zielorientierte Agenten haben allerdings ihre Vorteile und Nachteile. So die Trennung/Aufteilung des „*practical reasoning*" in zweistufigen Entscheidungsprozessen bringt eine bessere Strukturierung nach [Görz et al 2003] und durch eine separate Trennung von Zielen und Plänen findet eine Trennung statt zwischen dem Zustand, der erreicht werden soll und den

Mitteln, die zum Erreichen eines Zustandes zur Verfügung stehen (vgl. [Pokahr 2007]). Allerdings liegt der Schwerpunkt der zielorientierten Agenten in der zweiten Stufe des „*practical reasoning*“ - nämlich „*means-ends reasoning*“. Damit wird die Zieldeliberation vernachlässigt. Der Agent kann ein Ziel auswählen, dem kein Plan, keine verfügbare Fähigkeit in der Planbibliothek vorliegt (vgl. [Görz et al 2003]). Und zum Letzten geht der gesamte Entscheidungsprozess davon aus, dass ein zielorientierter Agent die langfristigen Ziele während der ersten Stufe der Entscheidung auswählt, definiert.

Ein weiterer und sehr wichtiger Aspekt der zielorientierten Agenten, der unbedingt erwähnt soll, ist eine Fähigkeit die Unterziele zu generieren. Das Generieren eines Unterziels geschieht aus einem Plan, konkret implementierter Agentenhandlung und vergleichbar mit einem synchronen objektorientierten Methodenaufruf, allerdings mit einem Unterschied. In der objektorientierter Welt wird nur die adressierte Methode ausgeführt und in der agentenorientierter Welt werden alle möglichen Pläne zu dem Ausführen gebracht, bis das Unterziel erreicht ist. Erst dann wird das Ausführen des übergeordneten aufrufenden Planes fortgesetzt. Diese Fähigkeit macht das Aufbauen von komplexen Zielsystemen möglich, die erst zur Laufzeit expandieren können.

2.1.7 BDI Agenten

Belief Desire Intention Architektur (BDI-Architektur) ist eine der bekanntesten Architekturen aus dem Bereich der deliberativen Agenten (s. Abbildung 2.5). Die BDI Agenten erweitern die zielorientierte Agenten und realisieren beide Teilprozessen des *practical reasoning.*



Quelle [Brenner et al 1997].

Abbildung 2.5: BDI Architektur.

Die grundlegenden architektonischen Elemente der Architektur sind:

- **Überzeugung** (engl. belief) Sie repräsentieren ein internes Wissen eines Agenten über seine Umwelt und sein Zustand. Das Wissen trägt den Charakter einer Annahme, eines Glaubens, dass es sich jeder Zeit als falsch, nicht korrekt oder veraltet aufweisen kann. Das Problem kann z. B. in einer dynamischen Welt auftreten. Aufgrund seiner Überzeugungen modelliert ein intelligenter Agent mögliche Umweltzustände und letztendlich trifft auf dieser Grundlage eine Entscheidung, die seinen nächsten Handlungsschritt und damit auch sein Verhalten bestimmt. Zum Realisieren eines Agentenwissens werden Variablen, relationalen *Daten Banken (DB)* oder Ausdrücke einer prädikatenlogischen Sprache benutzt.

- **Wunsch** (engl. desire) Sie sind direkt aus den Überzeugungen abgeleitet und beschreiben einen zukünftigen Agentenzustand oder einen Zustand eines Umweltmodells, der Agent hofft erreichen zu haben. Ein Agent kann einen Wunsch besitzen, der niemals erreicht werden kann - nicht realistische Wünsche. Er kann auch Wünsche besitzen, die im Konflikt miteinander stehen - einander ausschließende Wünsche, die nur separat erfühlt werden können. Ein Wunsch grenzt einen Bereich der zukünftigen Agentenhandlungen, bestimmt diese aber nicht.

- **Ziel** (engl. goal)[6] und Wunsch werden in meisten Literaturquellen synonym benutzt. Es wird angenommen, dass die Wünsche eines Agenten mit Hilfe von Zielen dargestellt werden. Obwohl die Ziele und Wünsche in der Abbildung 2.5 einzeln betrachtet werden, beschreiben die Autoren T. Eymann, G. Görz, C.R. Rollinger, J. Schneeberger und M. Wooldridge sie als ein Ganzes. Zusammenfassend können die Ziele eines Agenten langfristig/kurzfristig, erreichbar/nicht erreichbar, realistisch/nicht realistisch sein können und werden auch in Hauptziele und Unterziele aufgeteilt.

- **Absicht** (engl. intention) Absichten werden aus Zielen abgeleitet und repräsentieren eine Teilmenge von Zielen, die ein Agent pflichtmäßig entscheidet zu verfolgen. Dabei wird der *„deliberation prozess“* des zweistufigen Entscheidungsprozesses „*practical reasoning*“ noch einmal aufgeteilt. Zum Ersten werden die Ziele ausgewählt, die nach der Meinung eines Agenten relevant und wichtig sind. Dabei gelten die gleichen Regeln wie bei den Wünschen. Und zum Letzten wird es für die Absichten entschieden, die das konkrete Verhalten eines Agenten, zumindest bis das Ziel erreicht oder verworfen wird. Es wird aber eine *„starke Realismus Förderung“* bei der Auswahl von Absichten verlangt [Görz et al 2003].

- **Plan** ist eine konkrete Realisierung von Agentenhandlungen oder Auktionen. Dabei besteht ein Zusammenhang zwischen Intentionen und Plänen: Intentionen bilden Teilpläne des Gesamtplans eines Agenten und die Menge aller Pläne spiegelt wiederum die Intention eines Agenten wieder [Brenner et al 1997].

2.2 Multiagentensystemarchitekturen

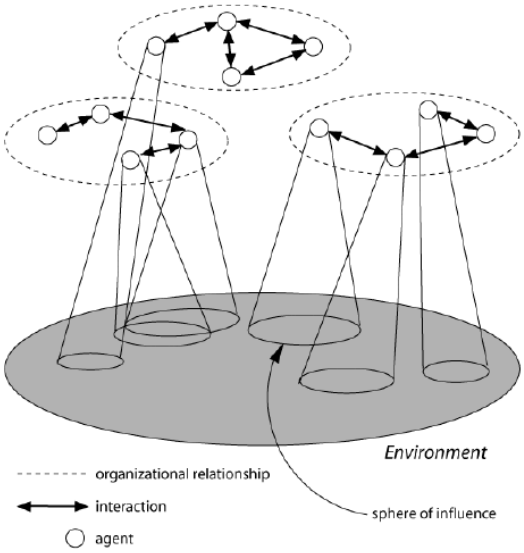
Nachdem die Agenten im Bezug auf eine Instanz besprochen wurden, stellen sich folgenden Fragen: Was stellt eine Gruppe der intelligenten Agenten dar?

Alle diese Fragen werden durch das Studium von MAS beantwortet. Dieser Bereich der Agententechnologie befasst sich mit Koordination, Kooperation, Kommunikation (s. Abbildung 2.2), verteilten Problemlösen und sozialen Fähigkeiten eines Agenten. M.J. Wooldridge beschreibt ein MAS (s.u. Abbildung 2.6):

> *“The idea of a multiagent system is very simple. An agent is a computer system that is capable of independent action on behalf of its user or own. … A multiagent system is one that consists of a number of agents, which interact with one another, typically by exchanging message through some computer network infrastructure.” [Wooldridge 2001]*

6 Im weiteren Verlauf der Arbeit werden die Ziele eines Agenten synonym zu seinen Wünschen benutzt.

Der Schwerpunkt von MAS ist der Einsatz von mehreren intelligenten Agenten in einer Umgebung, einem Gesamtsystem um eine bestmögliche, effiziente Zusammenarbeit zwischen Agenten erreichen und damit ein bestimmtes globales, gemeinsames Ziel optimal verfolgen zu können.



Quelle: [Wooldridge 2001]

Abbildung 2.6: Typische Struktur eines Multiagentensystem nach Wooldridge

Durch die Kooperation kann auch die Not an der Koordination innerhalb der kooperierenden Agentengruppe entstehen. Das Verlangen nach einer kooperativen Arbeit mit anderen Mitgliedern des Systems entsteht meistens dadurch, dass die Agenten nur ein Bruchteil des gesamten Wissens oder Fähigkeiten besitzen. Die Eigenschaften der Agenten sich zu koordinieren und kooperieren verlangt ein weiteres Merkmal der MAS - nämlich die Kommunikation. Sie ist der Grundstein der oben genannten Eigenschaften und wird zum Teil auch durch dezentrales Informationsspeicherung hervorgerufen. Jeder Agent lagert seine neu gewonnene Daten und internen Zustand in seinem eigenen Speicher, Wissensbasis. Ist das Wissen unvollständig und ein Wunsch nach dem Erweitern des Wissens existiert, so kann ein Agent eine Verbindung mit einem anderen Agenten aufbauen und mit ihm kommunizieren, um eigenes Wissen zu vervollständigen.

MAS lassen sich als geschlossene oder offene Systeme beschreiben. Ein offenes System repräsentiert eine Agentengesellschaft, wo neue Agenten dazu kommen können. Ein geschlossenes System ist eine Gesellschaft, wo das nicht zugelassen ist. Durch die Arten, Typen der Agenten in einem System lassen sich MAS in heterogene und homogene Systeme aufzuteilen. Dabei bestehen die homogenen Systeme nur aus einer Agentenart.

Alle Agenten laufen in einem MAS asynchron und bilden eine Menge von nebenläufigen autonomen Elementen. Der Grad der Selbstständigkeit wird in einem MAS durch die Abwesenheit der globalen Kontrolle über Agenten noch einmal erhöht. Ein weiteres Merkmal der Agenten in einem MAS ist sein hoher Eignungsgrad für die verteilte Problemlösung.

Alle o. g. Eigenschaften der einzelnen Agenten oder mehreren autonomen Agenten machen den Agenten möglich in einem MAS eine Gruppe zu bilden, um gemeinsame Aufgabe zu erledigen oder ein gemeinsames Ziel zu verfolgen. Eine Gruppe kann sowohl homogen als auch heterogen sein. Durch die Gruppenbildung der Agenten mit einer festen oder einer dynamischen Anzahl der Mitglieder kann ein Ziel schneller erreicht werden. Der Potenzial einer Gruppe wird jedoch erst durch Koordination der Agenten voll ausgenutzt. Nur mit der Koordinierung kann eine

Agentengruppe eine Aufgabe optimal aufteilen oder das Verfolgen eines gemeinsamen Ziels zentral oder dezentral planen. Die Agentengruppen können wiederum zusammen gefasst werden und eine Organisation[7] darstellen.

2.2.1 Agentenkommunikation

Die Kommunikation zwischen Agenten ist einer der wichtigsten Aspekte der Agentenarchitektur. Nur mit Unterstützung der Kommunikationsmittel ist es möglich die Agenten zu Koordinieren, Kooperieren und ihre Handlungen zu synchronisieren. Erst mit der Fähigkeit zu kommunizieren wird es einer Agentenmenge möglich ein gemeinsames Ziel zu verfolgen, verteiltes Problemlösen zu realisieren oder in einer verteilten Agentensimulation ein Modell der realen Welt zu simulieren. Mittels Nachrichtenaustausch sind die Agenten in der Lage nicht nur einfache Anfragen an einen anderen Agent senden, sonder auch ein Dialog mit ihm bilden und seine Handlung beeinflussen.

Wie es schon erwähnt wurde, erfolgt die Kommunikation in der Agentenwelt mittels Nachrichtenübermittlung, die für eine indirekte oder eine direkte Agentenkommunikation benutzt wird. Eine indirekte Kommunikation erfolgt über ein Blackboardsystem.

> *„Ein Blackboard bietet allen Agenten innerhalb eines Multi-Agentensystems einen gemeinsamen Arbeitsbereich, in dem sie Informationen, Daten und Wissen austauschen können." [Brenner et al 1997]*

Dieses System besteht aus einer zentralen DB. Die Agenten greifen auf das Blackboard zu, um ihre Information zu speichern (veröffentlichen) oder Informationen zu lesen. Sie können auch die relevanten Informationen aus Blackboard ausfiltern. Um eine Informationsveränderung im Blackboard zu verfolgen, greifen die Agenten in regelmäßigen Zeitabständen (Polling) auf das System zu oder sie werden von einem Dispatcher (vgl. [Brenner et al 1997]) oder einem Filteragenten (vgl. [Eymann 2003]) über die Veränderungen informiert.

Unter einer direkten Kommunikation wird eine „Agent zu Agent" Nachrichtenübertragung gemeint. Diese Art der Nachrichtenübermittlung wird mit Broadcast-, Multicast-, und Unicast[8] Nachrichten realisiert.

Um oben beschriebene Kooperation und Koordination zu realisieren, werden Agentensprachen benötigt. Gegenwärtig sind zwei alternative Sprachen in den Agententechnologien gewissermaßen zum Standard geworden: *Agent Communication Language (ACL)* und *Knowledge Query and Manipulation Language (KQML)*. Beide Alternativen realisieren eine Sprechakttheorie, die von J. L. Austin 1962 entwickelt wurde und 1969 von J. Searle erweitert und veröffentlicht wurde.

> *„In der Sprachaktentheorie stellen die ausgetauschten Nachrichten nicht nur Informationen dar, sondern sind gleichzeitig auch Aktionen des Senders gegenüber Empfänger." [Eymann 2000]*

---

7 Organisation - Gesamtheit der Maßnahmen zur Erreichung von Zwecken und Zielen, durch die ein soziales System arbeitsteilig strukturiert wird und die Aktivitäten der zum System gehörenden Menschen, der Einsatz von Mitteln und die Verarbeitung von Informationen geordnet werden. (vgl. [Hill et al 1984])

8 Eine Nachricht wird bei Broadcast an alle Agenten, Multicast an eine Gruppe der Agenten und Unicast an einen Agenten des Multiagentensystems übermittelt.

Ein Sprechakt besteht aus drei Akten: lokutionäre, illokutionäre, perlokutionäre. Die erste Akte stellt eine syntaktische Beschreibung der Nachricht, ihr Namen dar. Die zweite Akte definiert eine absichtliche Handlung des Senders. Die dritte Akte beschreibt eine Auswirkung der Handlung auf die Umwelt. Die Handlung wird weiter hin in fünf Typen der illokutionären Akte aufgeteilt. Eine detaillierte Beschreibung der Theorie ist in [Wooldridge 1999] zu finden.

ACL wurde von *Foundation for intelligent Physical Agents (FIPA)*[9] entwickelt. FIPA ist eine nicht kommerzielle Organisation, die 1996 mit dem Ziel gegründet wurde, industrierelevanten Standards für heterogene und interagierende Agentensysteme festzulegen [Weiß und Jakob 2005]. Die Sprachaktentheorie wird in ACL mit Performativen (illokutionäre Akte) implementiert[10]. Jedes Performativ definiert eine Agentenhandlung, die der Sender einem Empfänger per Netzwerk bekannt macht. Die Agenten sind jedoch autonom und der Empfang einer Nachricht garantiert nicht, dass der Empfänger die Handlung erfühlt.

Die Agenten sind in der Lage aus einzelnen Nachrichten komplizierte Dialoge zu bauen. Diese Fähigkeit der Agenten wird zum ihren Koordinieren eingesetzt, die aufeinander abgestimmtes Verhalten bedeutet (vgl. [Görz et al 2003]). Die Reihenfolge der Nachrichten und der Type der Performativen wird in einem Protokoll definiert. Mit diesen Protokollen werden Standarten für koordiniertes Verhalten zwischen Agenten gewährleistet. FIPA bietet eine Reihe der vordefinierten Protokolle an[11].

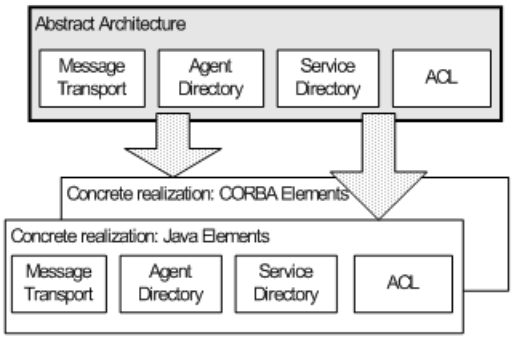
2.2.2 Agentenplattformen

Die Eigenschaft der Agenten in einer Umgebung miteinander zu kommunizieren, macht die Agenten fähig für kooperative Arbeit und für die Verfolgung eines globalen Ziels in einem verteilten Agentensystem. Eine Infrastruktur für ein MAS, die Kommunikations- und Verwaltungsschnittstellen anbietet, wird von einer Agentenplattform (Middleware) zur Verfugung gestellt.

> *„Middleware: Die Middleware enthält hochwertige Dienste zur Verwaltung verteilter Umgebungen und hat dabei insbesondere die Aufgabe einheitliche Schnittstellen bereitzustellen.“ [Brenner et al 1997]*

Zu den einheitlichen Schnittstellen gehören Dienstleistungen zum Verwaltung der Agenten in einer verteilten oder einer lokalen Agentenanwendung, in der die Agenten ihre Dienste anderen Agenten anbieten können oder Dienste anderen Agenten benutzen können. Als eine Schnittstelle zwischen einem Betriebssystem und einem MAS werden auch die Mittel zum Kommunizieren zwischen Agenten zur Verfügung gestellt. FIPA definiert eine abstrakte Architektur, die eine FIPA konforme Agentenplattform mit allgemeinen Schnittstellen beschreibt (s. Abbildung 2.7).

9 http://www.fipa.org (August 2009)
10 http:// www.fipa.org/specs/fipa00037/index.html (August 2009)
11 http://www.fipa.org/repository/standardspecs.html (August 2009)



Quelle www.fipa.org/specs/fipa00001/SC00001L.html, gelesen und kopiert am 20.07.09

Abbildung 2.7: FIPA abstrakte Architektur einer Agentenplattform.

- Die Funktionalitäten der einzelnen Schnittstellen der FIPA konformen Agentenplattform sind:
- *Agent Directory (AD)* oder *Agent Management System (AMS)*. Diese Schnittstelle dient zum Verwalten der Agenten, Kreieren oder Terminieren der Agenten.
- *Directory Service (DS)* oder *Directory Facilitator (DF).* Die Schnittstelle stellt ein Verzeichnis der Dienstleistungen eines Agenten dar, die er veröffentlicht. DF wird auch zum Suchen der Dienstleistungen anderer Agenten benutzt.
- *Message Transport (MT)* Ist für die Nachrichtenübermittlung zwischen Agenten zuständig.
- *Agent Communication Language (ACL)* Ist für die Nachrichtenkommunikation mittels ACL zuständig.

2.3 Verteilte Systeme und Nebenläufigkeit

Im Kapitel 2.1.6 wurde erwähnt, dass die Ziele eines Agenten in einem Agentensystem nebenläufig verfolgt werden können. Kapitel 2.2 beschreibt, dass die Agenten in einem MAS extrem autonom und nebenläufig sind. Das Kapitel 2.2.2 beschreibt, dass eine Agentenplattform Dienste anbietet, die eine Realisierung der verteilten MAS ermöglicht. Aus diesen Gründen werden die Begriffe „Nebenläufigkeit“ und „Verteilung“ im weiteren Verlauf dieses Kapitels definiert.

Die Nebenläufigkeit und die Verteilung sind zwei zusammenhängende Begriffe. So wird ein gleichzeitiger Ablauf von mehreren Aktivitäten auf einer Recheneinheit als ein paralleler oder ein nebenläufiger Ablauf bezeichnet (engl. concurrency). Dabei existiert in der Literatur keine strenge Trennung zwischen Parallelität und Nebenläufigkeit, obwohl ein echter paralleler Ablauf auf einer Maschine mit mehreren Prozessoren und nebenläufiger oder auch ein pseudoparalleler Ablauf auf einer Maschine mit einem Prozessor möglich ist (Vgl. [Oechsle 2007]).

> *„Ein verteiltes System ist eine Sammlung unabhängiger Computer, die den Benutzer wie ein einzelnes kohärentes[12] System erscheint.“ [Tanenbaum und Stehen 2007]*

Ein verteiltes System realisiert im Ganzen eine echte Parallelität.

12 Kohärenz (v. lat. cohaerentia) - Zusammenhang [Wahrig 1999].

Die Darstellung von nebenläufigen, parallelen Prozessen und dynamischen Aktivitäten in einem verteilten System erfolgt mittels Petrinetzen, die sowohl eine hohe Ausdrucksmächtigkeit bietet, als auch die Analysemethoden für Eigenschaften eines Entwurfes bereitstellt, die schon zum Entwurfszeitpunkt Aussagen über das System zulassen z. B. Deadlock-Freiheit. Die Stärke des Petrinetz Konzeptes liegt in der Ressourcenabstraktion eines Systems und in der Modellierung eines Systems auf einer abstrakter und auch auf einer detaillierten Ebene (vgl. [Muscholl 2001]).

2.3.1 Verteilte Systeme

Der Definition eines verteilten Systems entnommen besteht ein verteiltes System aus einer Menge der autonomen Recheneinheiten, die unabhängig von ihrer Art und ihrer Leistung zusammenarbeiten, um ein gemeinsames Ziel zu verfolgen. In einer Rolle des Benutzers kann sowohl ein Mensch als auch ein Programm auftreten. Dieser Benutzer darf nicht mitbekommen, dass er mit einem verteilten System interagiert. Dabei werden alle Aktivitäten dieses System wie z. B. Fehlerbehandlung beim Ausfall eines entfernten Computer, Synchronisierungsmechanismen oder Nachrichtenkommunikation zwischen einzelnen Rechner von dem Benutzer verborgen. Diese Eigenschaft des Systems, sich als eine „virtuelle Einheit“ darzustellen, wird als Transparenz bezeichnet. Diese Transparenz lässt sich in drei verschiedene Arten aufteilen (vgl. [Tanenbaum und Stehen 2007]):

- Zugriffstransparenz verbirgt, dass ein Benutzer auf eine Systemressource mit verschiedenen Betriebssystemen, verschiedenen Rechenarchitekturen.

- Ortstransparenz. Dieser Art der Transparenz verbirgt von dem Benutzer, wo die Ressourcen sich im System befinden.

- Nebenläufigkeitstransparenz ermöglicht mehreren Benutzer zur selben Zeit auf eine Ressource zugreifen. Die parallel ausführbaren Prozesse bleiben unauffällig.

Eine weitere Eigenschaft eines verteilten Systems ist die Skalierbarkeit. Das System kann skalierbar im Bezug auf ihre Große sein. Zum Beispiel können es weitere Systemressourcen oder Benutzer zugefügt werden. Es gibt eine geografische Skalierung, wenn sich die Benutzer und die Ressourcen weit voneinander befinden.

Die verteilten Systeme lassen sich in verteilte Computer- und Informationssysteme klassifizieren. Zu den verteilten Computersystemen gehören die Cluster-Computer. Das System besteht aus einem Hochgeschwindigkeitsnetzwerk und einer Menge gleichen Recheneinheiten, die über dieses Netzwerk miteinander verbunden sind. Ein wesentliches Merkmal dieses System ist der Einsatz eines gleichen Betriebssystems. Verteilte Informationssystem realisieren eine Client-Server Architektur. Zu dieser Klasse gehören *Remote Procedure Calls (RPC)* und *Remote Method Invocation (RMI)*[13]. Sie repräsentieren ein Kommunikationsmodell für direktes Kommunizieren zwischen Anwendungen oder Objekten (vgl. [Tanenbaum und Stehen 2007]).

Die Kommunikation erfolgt in einem verteilten Systems durch die Nachrichtenübermittlung. Die Nachrichten sind ein Grundstein in diesem System unabhängig davon, ob es um eine

13 http://java.sun.com/javase/technologies/core/basic/rmi/ (August 2009)

Synchronisierung der verteilten parallelen Prozesse, einen entfernten RMI-Methodenaufruf oder einen Informationsaustausch handelt.

Es wird zwischen vier Arten der Kommunikation unterschieden, die durch ihre Kombination entstehen:

- Persistente Kommunikation, die dauerhaft gespeichert wird.
- Flüchtige Kommunikation stellt die Nachrichten dar, die mit der Terminierung eines Senders und eines Empfängers verloren gehen z. B. Socket[14].
- Asynchrone Kommunikation: Die Ausführung eines Senders wird nicht unterbrochen.
- Synchrone Kommunikation: Der Sender bleibt blockiert, bis er eine Antwort bekommt.

2.3.2 Grundlagen der Petrinetzen

Petrinetze stellen eine Erweiterung der endlichen Automaten dar. Die Automaten haben einfache Modellform und eine sparsame Verwendung der Modellelemente. Sie bilden große komplexe Modelle im Bezug auf die Zustandsmenge bei der Darstellung von Systemen mit parallelen Prozessen. Die Petrinetze reduzieren die Darstellungskomplexität dieses Modells durch das Aufteilen eines Systemzustandes in mehrere Teilzustände, die sich zum Teil asynchron zu einander verändern können. Der Gesamtzustand eines solchen Petrinetzes wird durch eine Menge der Teilzustände zu einem Zeitpunkt definiert (vgl. [Lunze 2006]).

Ein Model eines Petrinetzes besteht aus folgenden Elementen (s. Abbildung 2.8).

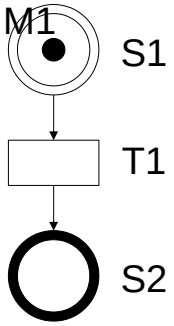


Abbildung 2.8: Petrinetz Modell

- Ein Zustand S1, der als eine „Stelle“ oder „Platz“ bezeichnet wird.
- Ein Zustandsübergang T1, der als eine Transition bezeichnet wird.
- Eine Kante, die in zwei Arten aufgeteilt ist: eine Präkante
- S1->T1 und eine Postkante T1->S2.
- Eine Marke M1, die von S1 über T1 zu S2 läuft.

Die Stellen und die Transitionen sind statische Elemente des Modells. Eine Transition verbindet zwei Stellen und teilt sie in Prästellen und Poststelle. Eine Marke ist jedoch eine dynamische Komponente des Petrinetzes. Sie fließt von einer Prästelle über Transition zu einer Poststelle. Im Bezug auf die Bewegung der Marke wird zwischen Eingangstransition und einer Ausgangstransition unterschieden. Eine Transition oder auch Ereignis genannt, findet nur dann statt, wenn sie aktiviert ist. Unter einer aktivierten Transition wird verstanden, dass alle Prästellen der Transition markiert sind.

Zum Prozess eines Petrinetzes gehören statische und dynamische Modellelemente, Markenfluss und die Bedingungen, die diesen Fluss bestimmen. Ein Prozess besteht aus mehreren Teilprozessen

14 Berkeley sockets application programming interface (API)

die sequenziell oder parallel ablaufen. Jeder dieser Teilprozess wird durch drei Elemente beschrieben [Lunze 2006]:

- Bedingungen, die erfüllt sein müssen, damit der Teilprozess ablaufen kann;
- den Prozessablauf;
- Bedingungen, die nach dem Ablauf eines Prozesses erfühlt werden;

Während der Modellierung werden am häufigsten die folgenden vier Standardsituationen für Teilprozesse eingesetzt:

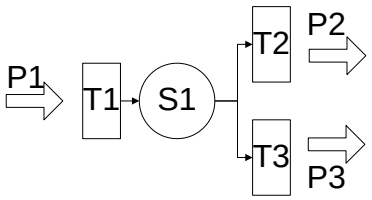


Abbildung 2.9: Auswahl

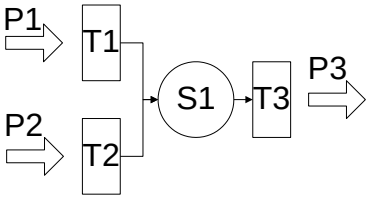


Abbildung 2.10: Begegnung

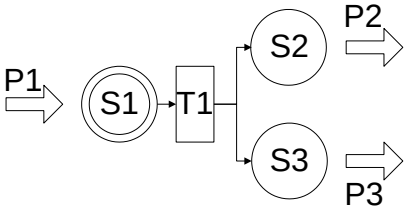


Abbildung 2.12: Aufspaltung

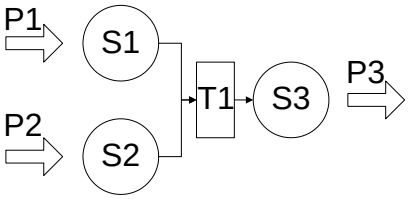


Abbildung 2.11: Synchronisierung

- Auswahl. Das Verhalten des Perinetzes ist in dieser Situation nicht deterministisch. Bei der Auswahl wird ein Teilprozess P1 nur von einer zur Verfügung stehenden Möglichkeiten (Transitionen) P2 oder P3 weiter fortgesetzt. Das Netz überbringt aber keine Information darüber, welche von den beiden Transitionen T2 oder T3 freigeschaltet sollen. Deswegen wird diese Auswahlsituation als ein Konflikt betrachtet (s. Abbildung 2.9).
- Begegnung. Bei dieser Konstellation der Modellelemente wird ein Teilprozess P1 oder P2 von einem folgenden Teilprozess P3 abgelöst. Das Markieren der Stelle S1 kann entweder durch eine Transition T1 oder T2 erfolgen (s. Abbildung 2.10).
- Aufspaltung (engl. fork). Es ist ebenso eine nicht deterministische Prozesssituation, in der ein Teilprozess P1 nach der Transformierung von zwei parallelen Prozessen P2 und P3 fortgesetzt wird (s. Abbildung 2.12).
- Synchronisation (engl. join). Ein Teilprozess P3 wird erst dann zum Ausführen gebracht, wenn alle Prästellen S1 und S2 einer Transition markiert sind und die Teilprozesse P1 und P2 ihre Aktivitäten abgeschlossen haben (s. Abbildung 2.11).

Für eine Abbildung eines realen Systems als Petrinetz ist es zunächst notwendig (vgl. [Rosenstengel und Winand 1991]):

- festzulegen, welche Elemente des Realsystems als Zustände und welche als Ereignisse im Sinne der Petrinetz Theorie definiert werden sollen und,

- Darauf aufbauend, die Informationen über (kausales) Ereignis Zustandszusammenhang zu einem Petrinetz zu fügen.

Das Modellieren eines Petrinetzes erfolgt in drei Stufen. In der Stufe 1 werden die Zustände (Stellen) und die Ereignisse (Transitionen) zusammen mit ihrer Kausalität in einem System festgestellt. In dieser Stufe werden die atomare Prozesse festgelegt, die nur noch eine Menge der unabhängig von einander, lösen Prozessen bilden. In der Stufe 2 wird aus der Menge der atomaren Prozesse ein Prozessnetz aufgebaut. In dieser Modellierungsphase werden die zusammenhängende Merkmale der lösen atomaren Prozessen festgestellt, die dann zu einem Teilprozess gefasst und miteinander verbunden werden. Es entsteht ein Netz mit sequenziell oder parallel ablaufenden Teilprozessen, das nur die statischen Elemente des Petrinetz-Modells befasst. In der Stufe 3 wird das dynamische Verhalten des Netzes definiert. Es werden die Anfangsmarkierungen und die Schaltregel des Petrinetzes definiert. Dadurch werden die Gesamtzustände des Systems definiert und das dynamische Verhalten des Netzes analysiert (vgl. [Rosenstengel und Winand 1991]).

2.4 Softwareengineering-SE

Der Abschnitt der Arbeit werden die theoretischen Grundlagen eines Software-Lebenszyklus beschrieben, die im weiteren Verlauf der Arbeit benutzt werden, um Standpunkt des portierenden Softwareproduktes festzustellen und Risikoanalyse durchzuführen.

2.4.1 Software-Lebenszyklus

> *„Der Software-Lebenszyklus (engl. software life cycle) ist der Prozess der Entwicklung von Softwareprodukten und kennzeichnet alle Phasen und Stadien dieser Produkte von ihrer Entwicklung, Einführung und Wartung bis zu ihrer Ablösung oder Beseitigung.“ [Dumke 2000]*

Eine Software läuft im Laufe ihrer Existenz mehrere Phasen des Lebenszyklus durch, wie Problemdefinition, Anforderungsanalyse, Spezifikation, Entwurf, Implementierung, Erprobung, Auslieferung und Wartung. Dabei ist keine Überraschung, dass ein Softwarestück nach der erfolgreichen Abnahme (Erprobung und Auslieferung) im Verlauf der Zeit auf eine Wartung angewiesen ist. Sie kann aus verschiedenen Gründen hervorgerufen werden wie z. B. eine Erweiterung der Anwendung, um neue Funktionalitäten oder Verbesserung der Softwareeffizienz zu schaffen. Wenn die Veränderungen in der Anwendung sehr umfangreich sind, wird es von einer Migration gesprochen. Der Begriff Migration wird oft als Synonym für Portierung verwendet.

> *„Die Migration ist die Überführung eines Softwaresystems auf eine neue operationale Umgebung. Das kann von der Änderungen der System-Software (insbesondere des Betriebssystem) bis hin zum Wechsel der Plattform reichen.“ [Dumke 2000]*

Die Veränderungen, die während der Wartung vorgenommen werden, haben meistens einen positiven Einfluss auf die Effizienz der Anwendung. Dabei handelt es sich um die Performance und Ressourcenverbrauch.

Der Grad der erforderlichen Wartung wird als Wartbarkeitsgrad bezeichnet und hängt von den folgenden Merkmalen einer Anwendung ab (vgl. [Dumke 2000]):

- Analysierbarkeit: wie schnell und einfach einzelne Softwarekomponente erfasst werden können und ihre Funktionen und Eigenschaften verstanden können.
- Änderbarkeit: wie hoch ist der Aufwand die vorgenommenen Änderungen zu realisieren.
- Stabilität: welche Auswirkungen können die Änderungen im Bezug auf einzelne Anwendungskomponente oder gesamtes Produkt haben.
- Testbarkeit: Wie groß ist der Testaufwand.

Ein weiterer Einflussfaktor der Wartbarkeit ist die Dokumentation der Software, die zu knapp, nicht vollständig oder einfach veraltet aufgrund zahlreicher Modifizierungen der Software sein kann.

Ist es der Fall und die Dokumentation erweitert werden soll oder die Änderungen im Programmcode vorgenommen werden müssen, so wird Reverse Engineering eingesetzt. Reverse Engineering wird zusammen mit der Wartung als Reengineering bezeichnet.

> *„Unter Software-Reengineering werden die Aktivitäten, Maßnahmen und Methoden verstanden, die der Erhöhung des Verständnisses, der Wartbarkeit und der Wiederverwendbarkeit dienen bzw. erst ermöglichen.“ [Dumke 2000]*

Reengineering beinhaltet auch ihre allgemeine Phasen: -Komponente bestimmen; -Komponente verstehen; -Komponenteneigenschaften bestimmen/messen; -Transformationsform auswählen; -Komponente transformieren; -Transformationsergebnis bewerten.

2.5 Agentenorientierte Software Engineering (AOSE)

Ein Agent ist eine Instanz mit seinen Eigenschaften und Aktivitäten, die mithilfe der bereits etablierten Techniken und Methoden des Software Engineering dargestellt werden. Im Kapitel 2.1.2 wurde geschildert, dass ein Objekt sowohl Gemeinsamkeit als auch Unterschiedlichkeit mit einem Agenten hat. R. Depke vergleicht in seiner Arbeit [Depke 2004] einen Agenten mit einer Softwarekomponente[15] und stellt weitere Ähnlichkeiten zwischen Agenten- und Objektwelt dar. G. Weiß und R. Jakob beschreiben, dass die Agentenorientierung besonders gut verträglich mit anderen Ansätzen, Techniken, Methoden ist. Dabei:

> *„… erhebt die agentenorientierte Betrachtungsweise nicht den Anspruch, andere Betrachtungsweisen zu ersetzen oder auszuschließen.“ [Weiß und Jakob 2005]*

15 Softwarekomponente - jede Komponente besitzt eine verbindlich definierte Außenschicht, ihre Schnittstelle. Diese Schnittstelle ist der Vertrag zwischen der Aufrufer und der Komponente. Sie besitzt auch eine nicht verbindliche Innensicht, nämlich die Implementierung selbst. Sie bestehen aus Moduln und/oder anderen Komponenten; jedes Modul ist eine Spezielle Komponente besonders einfacher Bauart. vgl. [Brössler und Siedersleben 2000]

Dies ist entscheidend, um die Agententechnologien mit anderen Konzepten, Ansätze z. B. Objekttechnologien zu kombinieren. So werden die anerkannten Methoden und Techniken des Software Engineering zum Modellieren, Analysieren und Implementieren der agentenorientierten Software eingesetzt.

> *„Agentenorientierte Software Engineering (AOSE) bezeichnet einen Bereich, der an der Schnittstelle zwischen Software Engineering einerseits und Agenten- und Multiagentensystemen andererseits entsteht. Gegenstand von AOSE sind Vorgehensweisen, Methoden, Techniken und Tools für die Erstellung und Handhabung von agentenorientierter Software.“ [Weiß 2001]*

AOSE hat die gleichen Entwicklungsphasen einer Software, wie SE und verläuft den gleichen Lebenszyklus (s. Kap. 2.4.1). Die Agenten brauchen aber aufgrund ihrer Komplexität, die durch ihre Autonomie, ihre Zielgerichtetheit und ihren mentalen Zustand entsteht, neue Methoden und Werkzeugen für einen Entwurf und Realisierung einer Agentenanwendung. Vor allem soll jedoch eine agentenorientierte Denkweise beschaffen werden, um ein Multiagentensystem zu entwerfen.

Heutzutage existieren zahlreiche Methoden, Techniken und Werkzeuge für die Entwicklung der Agentensoftware, die von G. Weiß und R. Jakob in vier Gruppen zusammengefasst wurden (vgl. [Weiß und Jakob 2005]):

- Agententechnologie mit Hinblick auf eine abstrakte Agentenebene z. B. Rollen oder Verfahren zur Koordination oder zum gemeinsamen Zielverfolgen;
- Objektorientierung mit ihren erweiterten für die Agenten Methoden;
- Requirements Engineering mit Bezug auf die Zielorientierung;
- Knowledge Engineering zum Modellieren des Wissens eines Agenten;

So z. B. bei der Modellierung der Agenten des Jadex MbFSlPs wurden Methoden all dieser Gruppen benutzt:

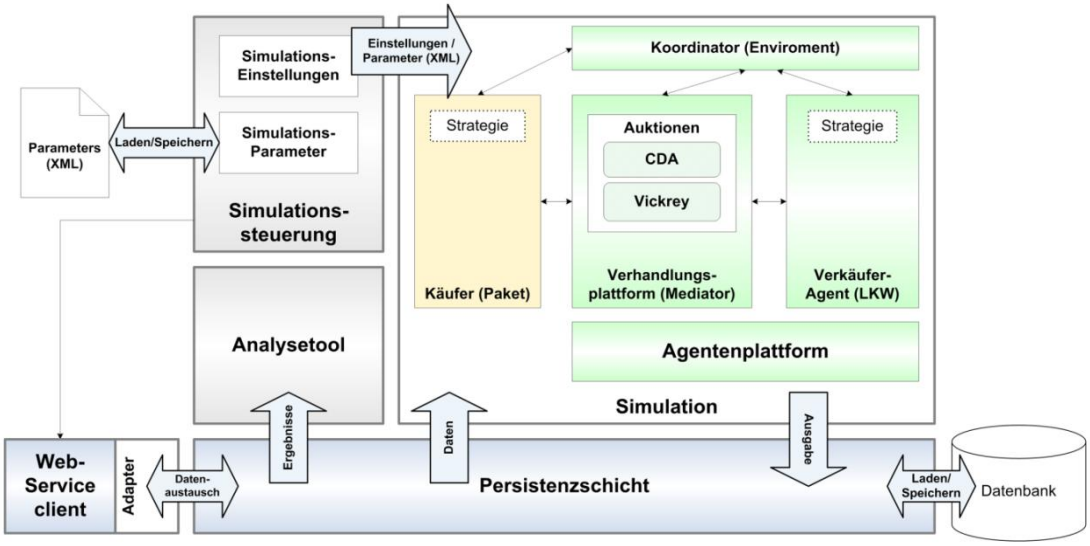
- Es wurde das Wissen eines Agenten modelliert.
- Ihre Rollen, im Zusammenspiel mit anderen Agenten oder Agentengruppen, wurden definiert.
- Agentenaktivitäten, die ein Agent in einer Rolle ausführen kann, wurden festgelegt.
- Es wurden die Agentenziele definiert, die in einer Rolle mittels definierten Aktivitäten erreichen werden sollen.
- Um ein globales Ziel zu verfolgen, wurden die Koordination und Synchronisation der Agenten mittels *Objekt Orientierten (OO)* und *Agent Orientierten (AO)* Mechanismen benutzt.

Eine besondere Herausforderung ist einen Agenten mit OO Techniken oder einen Objekt mit agentenorientiertem Verhalten zu implementieren. So wurden die Paketobjekte der Agentensimulation (s. Kap. 3.3) oder portierte auf eine Agentenplattform mittels eines reaktiven, nachrichtenorientierten Programmiermodells zielorientierte Agenten, deren Zielgerichtetheit mit Konzepten, Techniken und Methoden der Objekttechnologie realisiert (s. Kap. 7.3).

3 Multiagentensystem-basiertes Framework zur Simulation logistischer Prozesse (MbFSlP)

In diesem Abschnitt der Arbeit wird die zuportierende Agentenanwendung vorgestellt. Das Kapitel liefert theoretische Grundlagen eines MbFSlP. Hier wird die Bachelorarbeit der Herren Wladimir Ruwinski und Dennis Timotin dargestellt und die Architektur des Frameworks, MAS und die Semantik der Agentensimulation beschrieben. Das Kapitel befasst sich auch mit der technischen Realisierung des MASs und präsentiert eine Implementierung eines *Continuous Double Auction (CDA)* Verhandlungsverfahrens.

Das Framework stellt zu Verfügung eine Agentensimulationsumgebung zum Simulieren der logistischen Prozesse, ein Tool zum Auswerten der Informationen, die während des Simulierens in eine DB persistent geschrieben werden, und eine Softwarekomponente zum Steuern der Agentensimulation (s. Abbildung 3.1). Das Modell eines logistischen Netzwerks für die Simulation wurde auf der Basis realen Daten erstellt, die von der Firma *portrix.net GmbH* zur Verfügung gestellt wurden. Zum Realisieren der Agentensimulation wurde ein Jadex (s. Kap 4.1) Agentensystem benutzt.



Quelle [Ruwinski und Timotin 2007].

Abbildung 3.1: Architektur des Multiagentensystem-basiertes Frameworks zur Simulation logistischer Prozesse

3.1 Architektur

Die Architektur des Frameworks besteht aus vier Komponenten:

- Simulation. Die Komponente ist auf einem BDI basierten Jadex Agentenplattform aufgebaut und simuliert logistische und marktwirtschaftliche Prozesse.

Simulationssteuerung. Das Teil der Architektur ist für das Konfigurieren der Simulation verantwortlich. Die Steuerungsschnittstelle ermöglicht das Festlegen der Simulationsparameter und

Einstellen der Simulationseinstellungen mit Hilfe eines visuellen Tools oder einem *Extensible Markup Language (XML)*[16] Deskriptor.

- Die Persistenzschicht ist für dauerhaftes Speichern von relevanten Informationen zuständig, die während der Simulation sorgfältig gesammelt wurden. Außerdem stellt diese Schicht vordefinierten Parameter für Softwareagenten und notwendige Simulationsdaten für die Analyse der logistischen Prozesse.

- Analyse Tool dient zum Analysieren, Auswerten und Visualisieren der Informationen, die während einer Simulation gewonnen wurden.

3.2 Multiagentensimulation

Im Framework wurde ein vereinfachtes Model eines Paket- und Brief- Express- Dienst der Firma DHL Vertriebs GmbH & Co. OHG[17] simuliert. Das Ziel der Simulation ist eine Paketlieferung unter bestimmten Voraussetzungen wirtschaftlich günstig zu befördern. Das Befördern einer Lieferung erfolgt mit Hilfe der LKW Transportfahrzeugen von einem Umschlaglager zum Anderen. Dabei trifft jede Liefer- und Transporteinheit ihr eigene, selbstständige Entscheidungen. Diese Entscheidungen werden in einer Verhandlung zwischen einem Transportgut und einem Transportmittel getroffen in der Form eines Gebotes und eines Angebotes. Eine detaillierte Beschreibung dafür ist in der Arbeit zu lesen (s. Kap. 3.3.7 [Ruwinski und Timotin 2007]

Die Agentensimulation ist abstrakt in zwei Teilen gespaltet. Ein Teil stellt ein logistisches, unimodales Netz[18] dar, in dem ein Paket mithilfe eines LKW von einem Ort zum anderen abtransportiert wird. Der andere Teil stellt einen Markt dar, in dem die Pakete mit Trucks unter Beobachtung eines Koordinators handeln, um einen Transportplatz für einen angemessenen profitablen Transportpreis zu gewinnen.

In einem virtuellen logistischen Netz existieren *Hauptumschlagbasen (HUB)*, die alle miteinander verbunden sind und ein Netzwerk bilden, in dem jeder HUB-Knoten eine direkte Verbindung zu allen anderen hat. Jeder HUB besitzt über einen geografischen Punkt, der durch Breite und Länge abgebildet ist, und einer Menge der Wareneingängen und Warenausgängen. Die HUB-Knoten sind in drei Typen aufgeteilt: einen Metropol-, Standard- und Versand-HUB, die nach ihrer Leistung eingestuft sind. Die Verbindung (Route) zwischen zwei HUB-Knoten entspricht einer Luftlinie. Die Pakete werden von einem HUB zum Anderen über eine Route mit einem Truck befördert und sind in zwei Typen Standard- und Express Pakete aufgeteilt, die unterschiedliche Lieferfristen haben. Die Trucks besitzen über eine begrenzte Ladekapazität, die allerdings mit einem Anhänger erhöht werden kann. Die Pakete und die Trucks sind im Besitz eines Ursprungs- und Zielort. Der Ursprungsort eines Trucks repräsentiert eine Transportbasis, wo er als eine Transporteinheit eines Fuhrparkes angemeldet ist.

16 www.w3.org/XML (August 2009)
17 www.dhl.de (August 2009)
18 Ein unimodales Netz ist dadurch gekennzeichnet, dass im Hauptlauf kein Wechsel des Verkehrsträgers stattfindet. Das schließt nicht aus, dass der Hauptlauf mit verschiedenen Verkehrsmitteln (z.B. verschiedenen LKW-Typen) durchgeführt wird, die aber alle zum gleichen Verkehrsträger (Straße, Luft, Wasser) gehören müssen. Vgl. [Buchholz et al 1998]

Die Verhandlung zwischen einem Paket und einem Truck findet in einer HUB Verhandlungsplattform statt, die als ein Mediator in den Verhandlungen auftritt. Der HUB synchronisiert die Aktionen zwischen beiden Parteien. Die Pakete und Trucks sind mit einem Budget ausgestattet. Das Geld in der vorgegebenen Simulationswelt repräsentiert einen Anreiz Faktor, der einen Agenten zu den Aktivitäten (Verhandlungen) anstoßt. Das Ziel jedes Pakets und eines Trucks ist ein Gewinn während der Verhandlung zu erbringen. Die Verhandlungen werden nebenläufig oder parallel in mehreren HUB-Knoten ausgeführt, die durch einen globalen Taktgeber (Environment) synchronisiert werden. Das Environment nimmt die Bereitschaft jeder HUB-Verhandlungsplattform entgegen und sobald sich alle Teilnehmer als bereit gemeldet haben, sendet er ein Signal zum Starten der Verhandlungen. Das Environment teilt die Simulation in vordefinierte Spielrunden. Die Zeit zwischen diesen Runden ist als Transportweg eines Paketes oder Trucks bezeichnet.

3.3 Simulationsteilnehmer: Agent und Objekt

Die Agenten und Objekten, die an der Agentensimulation teilnehmen, haben ihre eigene Merkmale und Eigenschaften (s. Anhang A1 Technische Realisierung der Jadex Agenten). Der Manageragent ist z. B. ein passiver Agent, weil er eine Anweisung befolgt. Sein Ziel ist die Simulation zu starten. Das Ziel dieses Agenten definiert auch seine Rolle: Simulationsstarter. Er trifft keine Entscheidungen und handelt rein reaktiv.

Der Environment-Agent synchronisiert die Simulation. Sein Ziel definiert seine Rolle eines globalen Koordinators. Dieser Agent besitzt proaktive Eigenschaften.

Ein HUB ist im Besitz von mehren Rollen und Zielen. Er stellt eine der anspruchsvollsten Implementierungsstellen der Simulation dar. Der Agent handelt sowohl reaktiv als auch proaktiv. Er ist immer aktiv und diktiert allen anderen Agenten und Objekten (Paket, Truck, Environment), wann sie aktiv sein dürfen und welche Aktivitäten sie ausführen dürfen.

Der Truck-Agenten ist ein passiver Agent. Er reagiert auf die Anweisungen eines Environment- und HUB-Agenten. Seine Handlung ist eher reaktiv als proaktiv, obwohl er beide Eigenschaften in der Implementierung hat.

Das letzte Mitglied der Agentensimulation ist ein Paket. Seine Handlungen sind reaktiv und seine Implementierung stellt ein Imitat eines Agenten dar. Ein Paket ist ein Objekt, der im Simulationsmodell als Java Objekte nur aus Belastbarkeits- und Performancegründen implementiert wurde. Jadex befand sich zu der Zeit in der Entwicklungsphase und war nicht in der Lage die notwendige Menge an Agenten auf einer Maschine zu bearbeiten, verwalten. Die Anzahl der Agenten auf der Jadex-Plattform war maximal auf 200 Instanzen begrenzt (s. Kap. 4.1.1.2 [Ruwinski und Timotin 2007]). Diese Einschränkung brachte die Entwickler der geplanten Paketagenten als Objekte zu implementieren. Trotz der Objektorientierung wurden die Pakete mit Agenten ähnlichen Verhalten realisiert. Das Realisieren erfolgte durch verlagern dieses Verhaltens an einen HUB-Agenten. Pakete besitzen in der Rolle als Transportgut keine Aktivitäten, sie verhalten sich passiv und verfolgen die Anweisungen eines HUB-Agenten oder Truck-Agenten. Aber in der Rolle eines Käufers muss ein Paket an den Verhandlungen teilnehmen, um zum Lieferort abtransportiert zu werden. Die Teilnahme an den Verhandlungen fordert von einem Paket, dass er ein Gebot berechnet und ihn einem HUB-Agenten bekannt macht. Diese Aufgabe wurde aufgrund objektorientierter Lösung eines Paketes von einem HUB-Agenten in der Rolle einer Verhandlungsplattform übernommen. Das Paket selbst ist als ein Java Bean-Objekt implementiert,

um sie in DB in einer persistent schreiben zu können. In diesem Bean-Objekt werden alle simulationsrelevanten Daten untergebracht: Identifikation Nummer des Paketes, Startort und Lieferungsort, Strategie, Preise usw. Das Paket besitzt keine Kommunikationsleitungen mit den Mitgliedern des Simulationsmodells.

3.4 Multiagentensystem

Im Bezug auf das MAS stellt die Agentenanwendung einen recht komplexen Mechanismus dar. Es ist ein geschlossenes und heterogenes Agentensystem. Alle Agenten und Objekte dieses System verfolgen ihre eigenen Ziele. An dieser Stelle ist eine Übersicht der Agenten- und Objektzielen:

- Paket: -Ankommen zum Zielort; -Anhalten der erforderlichen Liefertermine; -Generieren eines Gebotes; -Teilnahme an der Verhandlung; -Profit.

- Truck: -Generieren eines Angebotes; -Teilnahme an der Verhandlung; -Reservierung der Lagerfläche; - in einem HUB für seine Ladung; -Pakete beladen und entladen; -Transportgut transportieren; -Profit.

- HUB: -Die Trucks im Fuhrpark an- und abmelden; -Die Gebote und Angebote anfordern; -Die Verhandlungen zwischen Paketen und Trucks starten und koordinieren; -Die Lagerfläche für eine bevorstehende Lieferung reservieren; -Die Trucks ausladen und beladen; -Die Pakete zwischenlagern.

- Environment: Simulationssynchronisierung.

- Manager: Simulationsstart.

Im Ganzen verfolgen alle Teilnehmer dieses MASs nur ein gemeinsames Ziel, nämlich die Pakete liefern. Um dieses Ziel zu erreichen wurde eine Agentengesellschaft aufgebaut, in der jeder Agentensubjekt eigene Rolle besitz. Hier ist eine Übersicht der Agentenrollen:

- Paket: Transportgut und Käufer

- Truck: Transportmittel und Verkäufer

- HUB: Umschlagspunkt/Lager, Fuhrpark, Verhandlungskoordinator/Verhandlungsplattform.

- Der Environment-Agent besitzt nur eine Rolle eines globalen Koordinators (Simulationstaktgeber).

- Der Manager Agent übernimmt einer Rolle des Simulationsstarters.

In dieser Agentengesellschaft werden Agentengruppen gebildet, z. B.: Truck-Agenten bilden während des Transportwegs eine Gruppe „beladenes Transportmittel“ mit Paketobjekten. Die Agenten arbeiten kooperativ während der Verhandlung, die von einem zentralen Koordinator geleitet wird.

3.5 Verhandlungsverfahren im Multiagentensystem-basierten Framework (MbFSlP)

Ein zentrales Element der MbFSIP Multiagentensimulation ist die Verhandlung. Sie ist das Wesentliche, was die Agenten von Objekten unterscheidet. Die Verhandlungen erfordern Autonomie der Agenten mit ihrer Fähigkeit selbstendige Entscheidungen zu treffen, Kommunikation, Koordination, Synchronisation, eine gewisse Maß der Reaktivität und Proaktivität. Zum Realisieren der Verhandlungsfähigkeiten der Agenten wurde im MAS ein *Continuos Double Auction (CDA)* Auktionsverfahren benutzt (s. Kap. 2.3.4.1 [Ruwinski und Timotin 2007]).). Die Auktion erfolgt innerhalb eines HUBs zwischen Paketen und Trucks.

Um an einer Verhandlung teilnehmen zu dürfen, sollen sich die Pakete und die Trucks bei einer Verhandlungsplattform registrieren. Ihre Registrierung erfolgt durch das Ausladen der Pakete und Anmelden der Trucks in einem HUB, der während der Verhandlung als eine Verhandlungsplattform und ein Verhandlungsmediator auftritt. (s. Kap. 3.3.4.2 und 4.1.1.5 [Ruwinski und Timotin 2007]). Die Verhandlung wird von einem globalen Koordinator (Environment) gestartet und von dem Verhandlungsmediator in mehrere Runden aufgeteilt. Nach dem Start geben alle Pakete einen Gebot und alle Trucks einen Angebot für eine bestimmte Route an den Mediator ab, um einen Vertrag (eine Tour) zwischen Paketen und Trucks abzuschließen. Die Bestimmung einer Route erfolgt mittels strategischen Entscheidungen eines Paketen und eines Trucks (s. Kap. 3.3.4.4 [Ruwinski und Timotin 2007]). Dabei generieren die Pakete mehrere Alternativgeboten pro Route, um ihre Erfolgschance bei der Verhandlung zu erhöhen. Nachdem alle Gebote und alle Angebote gesammelt wurden, beginnt die erste Verhandlungsrunde. Ab dieser Runde dürfen die Trucks ihre Routenrichtung bis zum Abschluss der Verhandlung nicht ändern. Während der Verhandlungsrunde iteriert der Mediator über das Sequenz der Routen und vergleicht in jeder Iteration die Gebote und Angebote miteinander. Der Vergleich (engl. matching) ist ein zweistufiger Prozess, in dem während der ersten Stufe die Gebote für einen Truck ohne Anhänger und in zweiter Stufe für einen Truck mit Anhänger gesucht werden. Die Auswahl erfolgt nach bestimmten Kriterien des Verhandlungsverfahrens (s. Kap. 5.1.3.2 [Ruwinski und Timotin 2007]). Während des Vergleichs stellen die ausgewählte Gebote und Angebote einen Tourenkandidat dar. Im weiteren Verlauf der Arbeit wird die Semantik des Match-Verfahrens beschrieben. (s. Kap.6.2.5)

Abschließend nach dem Vergleich wird eine Reservierung der Lagerfläche für ausgehandelte Paketladung eines Trucks in einem Ziel-HUB durchgeführt. Wenn die Reservierung erfolgreich ist, d. h. der Ziel-HUB genug freie Lagerfläche zur Verfügung hat, werden alle Verträge abgeschlossen und die Trucks beladen. Im Fall, wenn der Ziel-HUB doch keine freie Lagerfläche hat, werden alle Verträge aufgelöst und die aktuelle Route zwischen Mediator und Ziel-HUB für die weiteren Verhandlungsrunden geschlossen. Nach dem Verhandlungsschluss werden alle beladenen Trucks zum Abmelden von der Verhandlungsplattform aufgefordert.

3.6 Datenbank

In der Simulation hat DB zwei Zwecke. Eins davon ist die Speicherung von relevanten Daten, die während der Simulation generiert werden. Diese Daten werden später vom Analysetool aus der DB gelesen, ausgewertet und schließlich visuell dargestellt. Das zweite Ziel ist ein gemeinsames, globales Wissen der Agenten. Dieses Wissen repräsentiert eine zentrale Stelle mit wichtigen für Agenten Daten. Der Zugriff auf globales Wissen wird vom DB-Manager verwaltet.

3.7 Verteilung der Agentenanwendung

MbFSlP ist ein verteiltes Multiagentenbasiertes Framework. Die einzelne Komponente des Frameworks wie Simulation, Analyse Tool und Persistenzschicht lassen sich verteilt ausführen. Die DB ist ein zentraler Punkt in dieser Verteilung. Die Multiagentensimulation und Analysetool besitzen die notwendigen Verbindungsdaten um eine Verbindung zur DB herzustellen.

4 Plattformen

In diesem Kapitel werden zwei Agentenplattformen JADE Plattform der Firma Telekom Italia Lab und LS/TS Plattform der Firma Whitestein dargestellt. Das Kapitel befasst sich auch mit unterschiedlichen Programmiermodellen, die von diesen Firmen zum Implementieren der Agenten zur Verfügung stellen sind.

4.1 Jadex

Zum Modellieren und Implementieren der Agenten des zu portierenden MASs wurden Jadex Programmiermodell und von JADE angebotene Software benutzt.[19] Dieses Kapitel beschafft die notwendige Jadex Grundkenntnisse, die das Reengineering des Agentensystems während des Portierungsprozesses auf Whitestein ermöglichen.

4.1.1 JADE Plattform

Das Java implementierten Framework dient zum Entwickeln der FIPA konformen MAS (vgl. [Sudeikat 2004]). JADE stellt aber nicht nur ein Framework zur Verfügung, sondern auch eine Agentenplattform mit Basisdiensten wie *Agent Directory, Directory Service, Message Transport, Agent Communication Language* (s. Kap 2.2.2).

JADE stellt auch die Software zur Verfügung zum Administrieren der Agentenplattformen und zum Testen der Agenten. (vgl. [Sudeikat 2004])

- *Directory Facilitator Graphical User Interface (DFGUI)* dient zum grafischen Darstellung der DF eines Agenten.

- *Remote Monitoring Agent (RMA)* ist als ein eigenständiger Agent implementiert und stellt ein grafisches Werkzeug zum Administrieren der Agentenplattform. Mit RMA ist es möglich die neuen Agenten zu starten. Es gibt eine Übersicht der lokalvorhandenen Laufzeitumgebungen und Agenten. RMA ermöglicht das Administrieren der entfernten Laufzeitumgebungen.

- Dummy Agent stellt eine grafische Schnittstelle zum Versenden der ACL Nachrichten.

- Sniffer Agent, der die Nachrichtenübermittlung zwischen Agenten beobachtet und grafisch darstellt.

- Socket Proxy Agent wird benutzt während der Kommunikation zwischen entfernten Agenten. Er empfängt die Nachrichten über ein Socket und leitet an den Empfänger weiter.

---

19 Jadex stellt ein Programmiermodell zum Implementieren zielorientierten (BDI) und reaktiven (Mikro) Agenten, standalone Plattform, eine Plattform auf JADE Basis und realisiert das Konzept eines Frameworks zur Entwicklung einer Agentenanwendung.

4.1.2 Jadex Programmiermodell

Jadex ist ein FIPA konformes Agenten System, das als eine Erweiterung von JADE an der Universität Hamburg entwickelt wurde.

> *„Im Projekt JADEX wird daher der Ansatz verfolgt, bestehende agentenorientierte Middleware- Plattformen um die ihnen bisher fehlenden Konzepte rationaler Agenten zu erweitern.“ [Pokahr 2005a]*

Agent Definition File (ADF) ist eine XML Datei, die ein Agentenmodel repräsentiert. Die ADF Datei spezifiziert einen Agenten und stellt eine einheitliche, deklarative Beschreibung eines Jadex Agenten vor. ADF besteht aus folgenden Kernelementen: *-beliefs; -goals; -plans; -events; -expressions; -configurations.*

Zum Kreieren und Starten eines Jadex Agenten beschreibt ADF einen Initialisierungszustand, in dem alle startnotwendige Informationen eines Agenten wie *beliefs*, *goals* und die *plan bibliothek* zur Verfügung stehen.

Das Wissen (engl. belief) einer Jadex Agenten über die Agentenumwelt kann durch ein Java Objekt repräsentiert werden. Das ermöglicht die Werkzeuge wie DB-Schnittstellen und Ontologie-Entwurfswerkzeugen generierte Klassen direkt im Agenten weiterzuverwenden (vgl. [Pokahr 2005a]). Die gespeicherte Werte, nämlich Java Objekte in einem *belief*, sind durch einen *fakt* repräsentiert. Jadex definiert zwei Arten von Speichern der *fakten*: *belief* für einen *fakt* Element und *beliefset* für eine Menge von *fakt* Elementen. Das Wissen eines Jadex-Agenten wird in einem *belief base* persistent gehalten und kann zu jeder Zeit abgefragt oder modifiziert werden. Neben statischen *beliefs* verwaltet Jadex dynamische *beliefs*, die zur Laufzeit keine Werte besitzen sondern einen Ausdrück. Diese Art von Wissen wird bei jeder Abfrage neu ausgewertet.

Jadex unterstützt vier verschiedene Typen von **Zielen** (engl. goal) eines Softwareagenten die deklarativ in ADF beschrieben sind. Alle vier Jadex-Ziele werden zwischen reaktiven und proaktiven Zielen unterschieden (s. Tabelle 4.1).

Tabelle 4.1: Jadex Ziele

Ziele		Beschreibung
Reaktives Ziel	*Perform Goal*	Das Ziel dient zum einmaligen, direkten Ausführen einer Action.
	Query Goal	Es ist ähnlich *Achieve Goal* nur mit einem Unterschied, dass die Beschaffung der anwendungsrelevanten Daten zum weiteren Ausführungsverlauf, Erreichen eines bestimmten Zustandes einer Anwendung notwendig ist.
Proaktives Ziel	*Achieve Goal*	Es ist ein Ziel zum Erreichen eines bestimmten Zustandes. Mit *targetcondition* wird ein gewünschter Zielzustand vordefiniert und mit der *failurecondition* wird die Abbruchbedingung des Ziels definiert.
	Maintain Goal	Das Ziel dient zum Aufrechtbehalten eines bestimmten Zustandes. Das Erreichen dieses Ziels wird, nach dem Verletzen einer definierte in *maintaincondition* Zustand, zum Ausführen gebracht. Beim Erreichen, in *targetcondition* definierten Zielzustand, wird *Maintain Goal* vorübergehen deaktiviert.

Die Ziele in Jadex Programmiermodell werden zwischen *top-level goal* und *subgoal* unterschieden. Die *top-level goal* existieren unabhängig von Plänen und werden direkt nach der

Geburt eines Agenten erzeugt und deaktiviert mit der Terminierung eines Agenten. Sie werden auch nach dem Auslösen eines Triggers oder aus einem Plan erzeugt. Die *subgoal* im Gegensatz zum *top-level goals* werden nur aus einem Plan erzeugt und existieren solange der Plan nicht terminiert oder seine Ausführung nicht abgebrochen wird.

Ein Jadex Plan besteht aus einer deklarative und einer prozedurale Teilen. In der deklarativen XML Teil-Plan Kopf (engl. plan head) wird beschrieben, unter welchen Umständen ein Plan ausgewählt und ausgeführt werden soll. Es werden folgende Elemente definiert:

- Ein Ziel die ein Plan verfolgen muss.
- Ein Trigger der über Zeitpunkt der Instanziierung eines Planes signalisiert. Der Trigger wird beim Eintreffen der internen oder externen Nachrichten, Ziel Ereignissen ausgelöst.
- Vor und Kontextbedingungen die mit Booleschen Ausdrücken spezifiziert sein können.
- Parameter die aus *Plan Body* Teil gelesen und modifiziert sein können.
- Warteschlange. Es wird benutzt um eingehende Ereignisse (engl. event) sequenziell zu bearbeiten.

Im Tag *<body>…</body>* wird zuständiges prozedurales Teil des Planes nämlich Java Klasse bekannt gegeben. In der prozeduralen Java Teil-Plan Körper (engl. plan body) werden Handlungen und Aktivitäten eines Agenten beschrieben. Auf Grund der Unterbrechungsmöglichkeit eines Standarten Planes werden die *atomic blocks* zum Ausführen einer Reihe von Aktionen in einem Stuck zu Verfügung gestellt. Alle Pläne eines Jadex Agenten werden in *plan bibliothek* Datenstruktur registriert. Jedem Plan wird ein Java Thread zugeordnet, der bis zum Eintreffen eines Ereignisses passiv auf die Ausführungsmöglichkeit Wartet. Ein Plan ist in der Lage die Unterziele erzeugen und damit auch die zugehörige Pläne, die diese Unterziele verfolgen sollen. In diesem Fall wird ein Plan als Erfolgreich Abgeschlossen bezeichnet nur wenn alle seine Unterziele erreicht wurden.

Jadex unterstützt asynchrone und synchrone Generierung von Unterzielen. Ebenfalls können auch die Nachrichten asynchron und synchron aus einem Plan versendet werden. Jadex definiert zwei Typen der Ereignisse:

- Interne Ereignisse. Sind in der Lage einzelne oder mehrere Parameter in einem agenteninternen Nachrichtenaustausch zu transportieren. Sie werden asynchron versendet.

Nachrichten Ereignisse (engl. message event). Jadex unterstützt FIPA konforme Nachrichten[20] mit allen ihren Parameter und additional die Parameter *content_start*, *content_class*, *action_class*, mehr dazu [Jadex 2007] Tabelle 9.3). Der Nachrichten Inhalt kann als ein String oder ein Java Objekt Implementiert werden.

Die Ausdrücksprache (engl. expressions) wurde speziell für Jadex entwickelt. Sie besitzt eine Java ähnliche Syntax und *Object Query Language (OQL)* Anweisungen. Die OQL wurde von *Object Data Management Group (ODMG)*[21] standardisiert. Die Ausdrücke sind in zwei Kategorien

20 http://www.fipa.org/specs/fipa00061/SC00061G.html (August 2009)
21 http://www.odbms.org/ODMG/ (August 2009)

statische und dynamische aufgeteilt. Dynamische Ausdrücke werden beim jedem neuen Zugriff neu ausgewertet während die statische nur einmal zum Zeitpunkt der Erzeugung eines Ausdruckes. Sie können zum Spezifizieren des Agentenwissens oder zum Definieren von Bedingungen für die Zielausführung eingesetzt werden.

Die Konfiguration (engl. configuration) repräsentiert einen Initialisierung und/oder Endzustand eines Agenten. Jadex bietet einem Entwickler die Möglichkeit *beliefs, goals, plans* und *events* bei der Kreierung eines Agenten initialisieren und beim Terminieren die Aktionen des Endzustandes zu ausführen. Die Konfiguration erlaubt Start der Agenten mit unterschiedlichem, anfänglichem Wissen und Zielen.

Jadex stellt einen DF-Dienst (s. Kap. 2.2.2) und eine Menge der FIPA-konformen Protokolle der Agenteninteraktion zur Verfügung: Request-, Contract Net-, Iterated Contract Net-, English Auction-und Dutch Auction Interaction Protocol[22].

4.2 Whitestein

An dieser Stelle wird die Agentenplattform vorgestellt, auf die das MAS portiert werden soll. Im Verlauf dieses Kapitels werden drei Plattformeditionen ihre Dienste und ihre charakteristische Merkmale beschrieben. Es wird besonders auf die verschiedene Agenten-Programmiermodelle konzentriert, um sich später für ein am besten geeignetes Modell zu entscheiden.

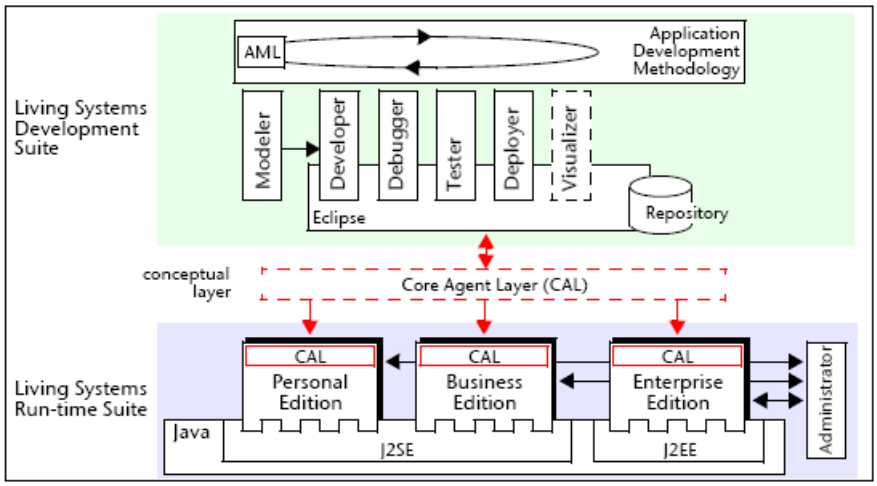
Whitestein war im Jahr 1999 gegründet. Im Jahr 2003 übernimmt Whitestein die Firma Living Systems GmbH, die im Jahr 1996 gegründet war. Seit 1997 entwickelte Living Systems das Produkt *Living Agents Runtime System (LARS)* [AgentLink 2000] der von Whitestein Technology Group AG zum kommerziellen Zwecken weiterentwickelt wurde. Whitestein ist ein Mitglied der JADE Board Organisation, die von Telecom Italia zum Förderung der Weiterentwicklung von JADE gegründet war.

4.2.1 Living Systems Plattform

Living Systems Technology besteht aus drei Komponenten: Run-time Environments & Tools, Development Tools, Methodology. Die Laufzeitumgebung ist mit *Java Plattform Standard Edition (J2SE)*[23] und *Java Plattform Enterprise Edition (J2EE)*[24] Technologien aufgebaut.

Weiter unten werden die einzelnen Architektonischen Komponenten der *Living Systems Technology Suite (LS/TS)* beschrieben (s. u. Abbildung 4.1).

22 http://www.fipa.org/repository/standardspecs.html (August 2009)
23 http://java.sun.com/javase (August 2009)
24 http://java.sun.com/javaee (August 2009)



Quelle [ProdOver 2006]

Abbildung 4.1: Living Systems Technology Suite.

Core Agent Layer (CAL*)* ist eine Schnittstelle zwischen Laufzeitumgebung und Development Tools.

Administration Tool stellt Werkzeuge zur Problemanalyse, und Debugging zur Verfügung.

Development Tools besteht aus einem Modeller, Developer, Debugger, Tester, Deployer und Visualizer Komponenten. Alle Komponente, außer Modeller, sind auf der *Eclipse IDE (Integrated Development Environment)*[25] entwickelt.

Developer stellt verschiedene Entwicklungstools wie: Agenten Explorer, Creation Wizards, Extended search oder Code Tamplates zur Verfügung.

Debugger unterstützt lokale und entfernte Debugging.

Deployer liefert eine Unterstützung beim Konfigurieren der Laufzeitumgebung, Agentenapplikation und automatischen Einbinden der Konfigurationsbibliotheken, Konfigurationsdateien.

Tester ist auf Basis der *Java Unit Test (JUnit)*[26] Framework aufgebaut. Er unterstützt *Mock Support* für *Servant* und *Data Access Object (DAO)* Agenten sowie Testen der Interaktionen zwischen Agenten.

Modeller dient zum Analysieren und Entwerfen der Agentenanwendungen mithilfe *Agenten Modeling Language (AML)*[27] und *Unified Modeling Language (UML)*[28].

Die *LS/TS Runtime Environment (LS/TS RE)* bietet drei Editionen einer Agentenplattform: *Personal-*, *Business-* und *Enterprise-Edition (entsprechend Abk.: LS/TS PE, LS/TS BE, LS/TS EE)* mit unterschiedlichen Eigenschaften, Diensten und Einsatzzielen. Eine der verfolgten Ziele ist die Deckung der unterschiedlichen, kundenspezifischen Anforderungen mit Hilfe von verschiedenen Editionen.

---

25 http://www.eclipse.org (August 2009)
26 http://www.junit.org (August 2009)
27 http://www.whitestein.com/aml (August 2009)
28 http://www.uml.org (August 2009)

LS/TS PE ist die einfachste Edition von allen. Sie ist gut geeignet für die Entwicklung in kleinen Projekten und zum Studieren des LS/TS Produktes. PE ist nicht geeignet für Einsatz als Ausführungsplattform in größeren Projekten auf Grund fehlenden Features, die in weiteren Editionen (s.u.) vorhanden sind. Durch die Integrierung LS/TS PE in die Eclipse IDE und Unterstützung durch LS/TS Development Suite, stellt einem Entwickler ein Entwicklungswerkzeug zur Verfügung. Mit dem ein Entwickler eine Agentenanwendung direkt aus Eclipse IDE oder eigenständig ausführen kann. Diese Version wurde unter Verwendung der J2SE entwickelt.

LS/TS BE wurde unter Verwendung J2SE entwickelt und im Vergleich zu LS/TS PE bietet ein erweitertes Features-Packet an. Weiterhin sind einige Erweiterungen mit einer kurzen Beschreibung aufgelistet:

- Persistent Agent. Der Zustand dieses Agenten wird in der *DB* gespeichert, um im Fall eines Maschinenabsturzes eine weitere Ausführung nach dem Problembeheben gewährleisten.

- Ressourcen Verwaltung (eng. ressource management) verwendbar nur mit nicht persistenten Agenten. Diese Erweiterung kann eingeschaltet werden, wenn große Mengen an Agenten oder Nachrichten im Agentensystem zu erwarten sind.

- Föderation bietet eine Möglichkeit der Agentenkooperation in Form einer Kommunikation zwischen mehreren von einander unabhängigen LS/TS RE-Instanzen.

- System Überwachung gibt die wichtigsten Informationen über Zustand der Laufzeitumgebung und über das Verhalten der Agentenanwendung. So können die Speicherauslastung, Große der Nachrichtenwarteschlange, Anzahl der Agenten im System usw. ermittelt werden.

LS/TS EE ist im Gegensatz zu o.g. Editionen auf der Basis J2EE entwickelt. Ein wichtiges Merkmal dieser Architektur ist der Einsatz von *Enterprise Java Beans (EJB)*[29]. Durch Zusammenführen von Agenten und EJB mit Hilfe s. g. „Glue“ Objekten, die als Konnektor und Transformator zwischen beiden Elementen dient, ist die LS/TS EE in einem Application Server eingebaut (mehr Information dazu in [AinJ2EE 2002]). J2EE Application Server bietet Zuverlässigkeit, Ausfallsicherheit, Performance, Skalierbarkeit, Sicherheit und zusammen mit DB die Transaktionen, Sitzungen (engl. session) und Persistenz an. EE bietet auch den Einsatz von Clustering an. Die Edition ist für große industrielle Projekte einsatzfähig.

Das Ausführen einer Agentenanwendung ist unabhängig von der Edition, so kann eine entwickelte Agentenanwendung in allen drei Editionen ohne Änderungen im Quellcode ausgeführt werden.

DB Unterstützung. LS/TS benutzt ein *Objektrelationales Mapping (ORM)* um Persistenz der Agenten gewährleisten. Als OR Mapper nimmt LS/TS ein Hibernate Framework [30] zu Grunde. LS/TS definiert zwei Typen einer DB:

- System Database wird ausschließlich von der Laufzeitumgebung zum Ausführen von systeminternen Aufgaben benutzt.

- Application Database wird zum Implementieren von DAO (s. Kap. 4.2.2) eingesetzt, die für unterschiedliche Zwecke einer Agentenanwendung benutzt werden.

---

29 http://java.sun.com/products/ejb (August 2009)
30 http://www.hibernate.org (August 2009)

Persistenter Agent. LS/TS RE definiert zwei Typen der Agenten: persistente- und nicht persistente Agenten. Zwischen beiden Typen existiert kein Unterschied im Quellcode. Sie bringen aber unterschiedliche Eigenschaften mit sich. Persistente Agenten sind in der Lage ein Abstürzt der Maschine zu überleben, während ein nicht persistenter Agent es nicht kann.

Föderation. Der Feature Föderation verbindet mehreren von einander unabhängigen Laufzeitumgebungen, die ihre eigene DB besitzen und ihre eigenen autonomen Agenten verwalten, zu einer virtuellen Laufzeitumgebung.

> *"The Federation is most usable in situations applications are naturally split to nearly independent parts, every part solving its own part of problem while communication with other nodes is not typical and not often."[Run Time 2006]*

Föderation ermöglicht einen Nachrichtenaustausch via RMI, Socket, *Hypertext Transfer Protocol (HTTP)*[31] Protokolle zwischen Agenten, die sich in verschiedenen Laufzeitumgebungen befinden. Die Nachrichten werden von einer Agentenplattform (Sender) über virtuelles Netz zu einer anderen Agentenplattform (Empfänger) versendet, dadurch wird die gesamte Agentenanwendung langsamer im Vergleich zu ihrer lokalen Ausführung. Eine Agentenanwendung kann durch aus mehreren Föderationen definieren.

Mehr zum Konfigurieren des *Deployments* und der LS/TS Agentenplattform, Anwendungs- und System DB ist im Anhang A2 Deployment und Konfiguration der LS/TS Plattform zu lesen.

4.2.2 Core Agent Layer (CAL)

CAL repräsentiert eine abstrakte Schnittstelle in Whitestein Architektur und einen Agenten mit grundlegenden Funktionen, die zum Fundament eines LS/TS Softwareagenten gehören. CAL wurde zum Implementieren von unterschiedlichen Agentenarchitekturen wie MARGE, LISA, MDAL eingesetzt und erlaubt einen Einsatz der verschiedenen Laufzeitumgebungseditionen (s. Abbildung 4.1). Er definiert drei unterschiedliche Komponententypen mit verschiedenen Funktionalitäten und Eigenschaften.

Data Access Object (DAO) ist ein reaktiver Agent, der einen Zugriff auf gemeinsam benutzte System- und Agentenspezifischen Daten vermittelt. Er erleichtert die Verwaltung der konsistent- und persistent relevanten Daten, die per synchrone Abfrage über einen Persistenzdienst abgerufen oder modifiziert werden können.

Servant ist ein zustandsloser, reaktiver Agent. In der Regel werden *Servants* eingesetzt, um spezifische, synchrone Aufrufe der Anwendungslogik oder gemeinsam benutzte Agentenfunktionalität zu implementieren.

Beide Agenten DAO und Servant übernehmen eine passive Rolle in einer Agentenanwendung. Dabei kann ein Servant die DAOs und andere Servants in seine Funktionalität hineinziehen, um seine Leistung zu steigern.

Autonomer Agent ist die zentrale Komponente in einer LS/TS Agentenanwendung. Er ist der einziger, der mit anderen Agenten koordiniert und seine eigene Ausführung indirekt kontrolliert.

---

31 http://www.w3.org/Protocols (August 2009)

Die Laufzeitumgebung besitzt volle Kontrolle über seine Ausführung. Ein autonomer Agent besteht aus drei Komponenten:

- Execution Engine besitzt ganze Funktionalität eines Agenten.
- Agent Metainformation besitzt Identität eines Agenten, Informationen über sein Besitzer oder grafische Repräsentation.
- Agent Data Storage ist zum Speichern des Wissens und Informationen über seine Ausführung genutzt.

Jeder autonomer Agent stellt generische Dienste zur Verfugung wie *life-cycle services* zum Kreieren und Terminieren eines Agenten und *communication services* vermitteln der Nachrichten und Meldungen an Agenten. Die Kommunikation mit anderen Agenten erfolgt durch das Versenden der asynchronen Nachrichten. Die Referenzen zum Servant und DAO werden über *agent component lookup service* ausfindig gemacht.

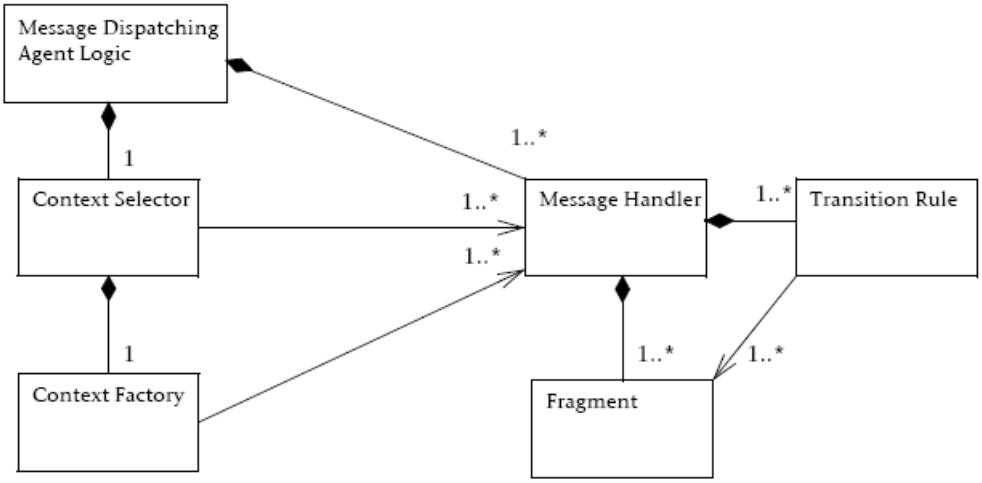
4.2.3 Message Dispatching Agent Logic (MDAL)

> *„MDAL repräsentiert eine taskmodellähnliche Architektur, die jedoch auf eine sehr nachrichtenorientierte Perspektive fokussiert.“ [Braubach 2007]*

MDAL besteht aus fünf folgenden architektonischen Komponenten (s. u. Abbildung 4.2).

Context Selector sendet alle empfangenen Nachrichten an einen Kontext. Ein Kontext in MDAL ist eine Umgebung, wo die eingehenden Nachrichten behandelt werden und kapselt das Wissen zur Verarbeitung der Nachrichten (vgl. [Braubach 2007]). Der Kontext wird durch einen Message Handler repräsentiert, der einer Lebenslinie in AML Modellierung entspricht.

- Context Factory ist zuständig für eingehende Nachrichten. Es wird ein geeignete Message Handler ausgewählt und zu instanziiert wenn eine Nachricht zur keiner bestehende Interaktion zugeordnet werden kann (vgl. [Pokahr 2007]).
- Message Handler sind Aktivitäten eines Agenten und werden gezielt für passende Nachrichten instanziiert und ausgeführt (vgl. [Pokahr 2007]).
- Translation Rule definiert die Ausführungsreihenfolge der Fragmente. Die Transitionen stellen dabei einen Übergang der Anwendung von einem alten zu einem neuen Zustand.
- Fragment. Ein Fragment kapselt die Anwendungslogik einer empfangenen Nachricht ein (vgl. [Braubach 2007]).



Quelle [MDAL 2006]
Abbildung 4.2: MDAL Übersicht

Interne und externe Nachrichten. Ein MDAL Agent benutzt externe Nachrichten und interne Nachrichten zum Kommunizieren. Das Versenden von internen Nachrichten charakterisiert einen MDAL Agent als einen unabhängigen und autonomen Agent. Die Agenten können auch ein oder mehrere Gespräche miteinander führen, die durch Kontexte repräsentiert werden. Ein MDAL Agent kann ein verschachteltes Gespräch führen. Während die ganze Interaktion asynchron abläuft, wird ein verschachteltes Gespräch synchron ausgeführt.

Kontextlose und kontextbehaftete Nachrichten. MDAL unterscheidet zwischen zwei Typen der Nachrichten: kontextlose, kontextbehaftete Nachrichten. Die kontextlosen Nachrichten sind diejenigen, die keine Information über Empfängers Kontext besitzen. Solche Nachrichten werden als Anweisungen, die keine Antwort erwarten, oder als Gesprächsanfang interpretiert. Im Fall einer kontextlosen Nachricht, sendet der Context Selector die eingetroffene Nachricht an Context Factory um einen passenden Message Handler zu finden und zu instanziieren. Die kontextbehafteten Nachrichten sind diejenige, die Informationen über Empfängers Kontext verfügen. In diesem Fall sendet der Context Selector die eingetroffene Nachricht direkt an einen Kontext, der für die Bearbeitung der Nachricht zuständig ist.

Subject. Eine MDAL Nachricht besitzt Information über einen Empfänger/Absender Kontext, Thema (engl. subject), Performative und Inhalt. Das Thema wird genutzt, wenn Agent eine kontextlose Nachricht erhält um einen passenden geeigneten Message Handler auszuwählen. Um festzulegen welcher Message Handler beim Eintritt einer kontextlosen Nachricht instanziieren soll, muss manuell eine Abbildungstabelle angelegt werden, die das Thema einer Nachricht zu einem Message Handler zuordnet (vgl. [Pokahr 2007]).

Transitionsregel. Wie oben erwähnt wurde, können die Agenteninteraktionen einfach oder komplex sein. Aus diesem Grund unterstützt MDAL Simple und komplexe Message Handler. Die einfache Message Handler können nur mit einer Regel (z. B. senden und empfangen einer Nachricht) existieren, während die komplexe Massage Handler über eine eigene Regel-Engine verfügen, die als ein Zustandsautomat implementiert ist. Zum Konzept des Zustandsautomaten werden die *Petrinetze* (s. Kap 2.3.2 und Kap. 7.3) genommen. Ein MDAL Message Handler stellt dabei einen Repräsentant eines Netzes vor, der eine Transition Rule definiert. Die Transition Rule implementiert die Regeln, die einen Übergang von einem Zustand eines Netzes (Stelle oder MDAL Position) zum Anderen beschreiben. Der Übergang erfolgt über ein Fragment, das eine Transition darstellt. Dabei kann das Ausführen der Transition durch eine Vorbedingung oder eine

Nachtbedingung beeinflusst werden. Die Nachricht stellt in MDAL Architektur eine Marke vor, die eine Transition hervorruft. Der Automat verfügt über einen Anfangszustand (POS_INITIAL) und einen Endzustand (POS_TERMINAL). Dabei kann das Ausführen eines Fragmentes im Anfangszustand während der Initialisierung des Handlers mit *initialmessage* hervorgerufen werden und das Fragment im Endzustand kann vor dem Terminieren des Handlers ausgeführt werden. Mehr dazu in [MDAL 2006].

Fragment. Der Zustand einer Agentenanwendung ist durch einen Fragment repräsentiert, in dem die Aktionen eines Agenten implementiert sind und in einzelne Stritte der Methoden *step_1(), ..., step_n()* aufgeteilt. Das Fragment selbst kann als eine Methode, eine innere Java Klasse oder als eine separate Java Klasse implementiert werden.

Die Performative (s. Kap. 2.2.1) einer MDAL Nachricht wird für Transitionsregel benutzt, um ein geeignetes Fragment zu finden bzw. einen zugeordneten Zustand auswählen. Die Zuordnung der Performativen zu den Fragmenten findet in einer Abbildungstabelle, die manuell angelegt werden soll.

4.2.4 Multi-Agent Reasoning based on Goal-oriented Execution (MARGE)

MARGE wurde als Erweiterung einer MDAL-Programmiermodell implementiert und bietet Einsatz der *Beliefe-Desire-Intention (BDI)* Architektur an (s. Kap. 2.1.7). Ein MARGE-Agent besitzt drei Grundsteine der BDI Architektur: Wissen, Ziele und Pläne (s. Anhang A3 LS/TS Programmiermodelle). Das Modell unterstützt sowohl rationale als auch proaktive Agenten, die über eine Wissensbasis Datenstruktur besitzen. Über diese Datenstruktur erfolgt ein Zugriff auf Internes Wissen und aktuell verfolgte Ziele eines Agenten. Das Wissen und die Ziele sind mit *First-Order Predicate Logic (FOPL)* beschrieben. Mehr dazu in [traveling example 2006] und [MARGE 2006].

MARGE BDI Interpreter ist als ein MDAL Massage Handler implementiert und aus diesem Grund muss während der Initialisierung eines Softwareagenten manuell gestartet werden (s. [MDAL 2006]).

4.2.5 Library for Interactions and Structured Actions Enginem (LISA)

LISA stellt ein Task- Model Architektur dar und wurde als eine Alternative zu MDAL entwickelt. Das Programmiermodell in LISA wurde deutlich vereinfacht im Vergleich zu MDAL, weil die Aktivitäten nicht als ein Zustandsautomat spezifiziert werden müssen. Sondern als *if-else Regel* (vgl. [Pokahr 2007]). Die wichtigen Eigenschaften von LISA sind gleich wie bei MDAL:

- LISA manipuliert nicht ihren eigenen Ausführungsthread.
- Sie ist nachrichtenorientiert und wird zum Ausführen nur nach dem Eintritt eines Ereignisses gebracht werden. Der Thread wird nach der Ausführung zurück zum Thread Pool zugeteilt.
- Ein Agent bekommt nur einen Thread zugeteilt und bearbeitet nur einen Ereignis zur gleichen Zeit.
- LISA gewährleistet die parallele Ausführung der Task.

LISA Agent kapselt entsprechend der Architektonischen Richtlinien der LS/TS Produktes einen CAL Agenten ein und wandelt sein Grundfunktionalität zum LISA Agent um. Damit ist es möglich, LISA Agent auf allen drei Plattformeditionen zu nutzen. Die Basiskomponente in LISA Architektur ist ein Task. Der Entwickler erzeugt eine Unterklasse eines Tasks, um eine Agentenlogik zu implementieren und an die CAL Funktionalität ran zu kommen. Das Verhalten eines Agenten wird entweder mit einem oder mit mehreren Task implementiert. Ein LISA Task ist in der Lage einen *Child-, Sub-, Independent Task* oder einen *InputHandler* zu definieren. Jeder Task kapselt einen Kontext ein, der durch aus zum einen oder mehreren Tasks gehören kann. Alle interne Taskkomponenten, die zum Definieren eines Agentenverhaltens gehören, werden *Continuations* genannt (s. Anhang A3).

> *"From an abstract point of view, it can be said that a continuation accept an input or an internal LISA Engine signal and executes certain functionality as a consequence. The action to be executed by a continuation is defined by means of a continuation command." [LISA 2006]*

4.2.6 Semantic Communication (SemCom)

> *"Semantic Communication is the new concept of communication aims at improving multiagent programming by a higher level and more specific interaction model." [ProdOver 2006]*

Semantic Communication (SemCom) ist eine weitere Erweiterung des MDAL Programmiermodells, die eine Vierschichtenarchitektur darstellt:

- Die Interaktionsschicht (engl. interaction tier) stellt reinen Nachrichtenaustausch (engl. message passing) Mechanismus dar. Agenten dieser Schicht übermitteln die Nachrichten interpretieren sie aber nicht. Dieses Mechanismus wird von MDAL gewährleistet.

- Sprachschicht (engl. linguistic tier) definiert eine allgemeine Taxonomie[32] von Nachrichten, die in jedem Anwendungsbereich angewandt werden kann.

- Bereichsschicht (engl. domain tier) definiert Bedingungen und Einschränkungen für die Struktur eines Nachrichteninhaltes.

- Sozialschicht (engl. social tier) definiert einen Nachrichtenverlauf zum Strukturieren eines Agentengesprächs.

Es gilt für oben vorgestelltes Schichtenmodell:

> *"The more layers that two agents share, the deeper they can nteroperate."[SemCom 2006]*

Eine detaillierte Beschreibung der einzelnen Schichten der SemCom Architektur befindet sich im Anhang A3 LS/TS Programmiermodelle.

32Taxonomie (v. griech. táxis „Ordnung", -nómos „Gesetz") - Anordnung [Wahrig 1999]

4.2.7 Process Algebra

MDAL Process Algebra Handler ist eine Alternative zum MDAL Message Handler. Der Handler beschreibt die Anwendungslogik eines Agenten mit prozessalgebraischen Ausdrücken. LS/TS definiert drei Ausdrücktypen: Sequenziell, Parallel und Alternativ. Diese Ausdrücke beinhalten unterschiedliche Handlungen, die Parallel, Sequenziell oder Alternativ ausgeführt sein können: Fragment, verschachteltes Message Handler, Methoden innerhalb der Handler, Goals. Mit Eigenschaften der algebraischen Ausdrucke ist es möglich, die Start-, Stopp-, Wiederholungsbedingungen, Start Trigger so wie die Synchronisationseigenschaften, beim Ausführen mehrerer Actions zu definieren. Im Gegensatz zum direkten Aufruf einer Handlung aus MDAL Transitionen, gibt die Prozess Algebra mehr Flexibilität beim Definieren der Agentenhandlungen. Mehr dazu in [DevGuide 2006].

4.2.8 LS/TS Client

LS/TS bittet Einsatz eines Clients, zum Kommunizieren mit LS/TS Agenten an. Der MDAL Client ist eine separate Softwarekomponente, die unabhängig von der Agentenplattform ausgeführt wird. Sein Ausführen erfolgt aus der Eclipse IDE. Zum kommunizieren mit Agenten stellt LS/TS einen Client Framework zu Verfugung, der die Schnittstellen zum Verwalten von Verbindungsaufbau, Nachrichten- und Befehlsaustausch definiert. Der Verbindungsaufbau erfolgt nur mit der Instanz einer Plattform, das Verbinden mit einer Föderation ist nicht realisiert. Der Client repräsentiert einen Agenten und benutz dasselbe Kommunikationsschnittstelle wie autonome Agenten einer Anwendung. Er benutzt dieselbe *life-cycle services* und *communication services,* um Agenten zu Kreieren, Terminieren oder mit ihnen per Nachrichtenübertragung kommunizieren. Das Kreieren oder Terminieren der Agenten geschieht synchron. Es besteht auch eine Möglichkeit Befehle asynchron oder synchron an die Laufzeitumgebung zu senden. Agenten besitzen keine Rechte eine Verbindung selbst aufzubauen oder trennen aufgrund der Einschränkungen (s. Kap. 5.2.3).

4.2.9 Test-Framework

Das LS/TS Test-Framework benutzt das Konzept der Mock Objekten und vergleicht einen Ist- mit Sollzustand eines Agenten. Zum Realisieren dieses Frameworks wurde *JUnit*[33] Test-Framework und *JMock*[34] Bibliothek eingesetzt. Die Testumgebung dient zum Testen der autonomen Agenten, DAO-Agenten und Servant-Agenten eines MDAL Programmiermodells. Die Nachrichten in MDAL bestimmen und beeinflussen internen Zustand eines Agenten oder seiner Umwelt und aus diesem Grund sie (die Nachrichten) sind Bestandteile des Test-Frameworks. Die richtige Reihenfolge der empfangenen Nachrichten im Test-Framework führt zum Ausführen eines Fragmentes, das einen Agentenzustand oder Umweltzustand verändert und damit einen Sollzustand eines Mocks Objektes definiert. Mehr Information in [LS/TS 2006]

33 http://www.junit.org
34 http://www.jmock.org

4.3 Vergleichsbewertung der Jadex und LS/TS

Nachdem Jadex und LS/TS kennengelernt wurden, werden die beide Produkte mit Hinsicht auf die Portierbarkeit miteinander verglichen:

- Jadex und LS/TS sind FIPA konforme Agentenplattformen, die AMS, DF, MT und ACL Schnittstellen zu Verfügung stellen und FIPA standardisierte Interaktionsprotokolle implementieren.

- Beide Produkten bieten eine Agentenumgebung an, in der autonomen Agenten nebenläufig oder parallel ihre Aktivitäten ausführen und miteinander kommunizieren.

- Jadex und LS/TS unterstützen „Level-2-Agenten“ mit BDI Architektur. Es besteht nur ein Unterschied in der Realisierung des BDI Konzeptes. Jadex verwendet zum Ausführen der Agentenaktivitäten einen BDI Interpreter und LS/TS realisiert dieses Konzept in MARGE Programmiermodell mittels FOPL. Der zielorientierte Agenteneinsatz wird in Jadex von vier Zieltypen realisiert: *Achieve Goal, Perform Goal, Query Goal, Maintain Goal,* und in LS/TS MARGE nur von einem Zieltype *Achieve Goal*. Die Unterziele werden sowohl von Jadex als auch von LS/TS MARGE unterstützt. LS/TS setzt MDAL Programmiermodell auch für „Level-0-Agenten“ und „Level-1-Agenten“ ein. Die Ausführung der Agentenaktivitäten dieser Klassen ist mittels Petrinetz Konzept realisiert.

- Jadex und LS/TS Agenten besitzen ein *beliefbase* und haben eine Möglichkeit auf eine DB zugreifen. Zusammen mit DB ist es möglich ein Blackboardsystem aufzubauen. Es besteht aber ein Unterschied in der Einbettung der DB zu einer Agentenanwendung. So verwendet LS/TS ein Hibernate Framework für DB Verbindungsverwaltung und Speicherung der Anwendungsdaten. Jadex überlässt dies einem Entwickler.

- Eine Agentenaktivität ist in Jadex von einem Plan und in LS/TS von einem MDAL Fragment oder einem MARGE Plan dargestellt. Die Jadex und MARGE Pläne bestehen aus einer Deklarations- und einer Auktionsteil und werden in einem Planbase eines Agenten gehalten. Jadex Plan und MDAL Fragment verfügen über atomare Agentenaktivitäten: Jadex *Atomic Blocks* und *steps*, die während der Ausführung der Agentenaktivitäten nicht unterbrochen werden.

Nach der Betrachtung der allgemeinen Merkmalen beider Produkten ist es feststellbar, dass Jadex und LS/TS in ihrer Architektur mehrere gemeinsame Anhaltspunkte haben. Diese Gemeinsamkeiten, die durch den Einsatz FIPA Standarten entstanden sind, machen die Portierbarkeit einer Agentenanwendung möglich. Die Unterstützung von gleichen Agentenarchitekturen und ähnlichen Anhaltspunkten in den Programmiermodellen macht eine Portierung des MASs von Jadex auf LS/TS realisierbar.

5 Analyse

Im Verlauf dieses Kapitels wird eine Analyse der wichtigen charakteristischen Merkmale des portierenden MASs, Portierung der Agentenanwendung und abschließend praktisches Testen des MASs durchgeführt.

5.1 Analyse des Multiagentensystem-basiertes Frameworks zur Simulation logistischer Prozesse (MbFSlP)

Das MAS wurde mit Jadex Programmiermodell implementiert. Whitestein bietet verschiedene Programmiermodellen zum Auswahl. Darunter sind Modelle für proaktive, zielgerichtete BDI Agenten (MARGE), reaktive Agenten (MDAL, LISA) und Agenten, die mit Hilfe des SemCom Programmiermodells realisiert sind. Um später ein geeignetes Modell zum Portieren des MASs auszuwählen, werden die Agenten und Agentensystem nach ihren verschiedenen Anhaltspunkten analysiert.

Die zwei folgenden Unterkapitel beschreiben einen Analysevorgang im Hinblick auf die Entscheidung zwischen MARGE, MDAL und LISA LS/TS Modellen.

5.1.1 Ansatz der BDI Architektur in der Simulation

Nach der Analyse des MbFSlPs wird ersichtlich, dass MAS über ein internes Wissensmodel, Ziele und Pläne verfügt, deren Anwesenheit einige Eigenschaften eines BDI Systems aufweisen. Allerdings das Prozess *practical reasoning* (s. Kap. 2.1.6) konnte im Framework nicht nachgewiesen. Die ADF Deskriptoren besitzen keine <*delibiration*> Tags, die von Agenten zum Definieren der Zielalternativen mit ihren Ausführungskriterien benutzt werden. Sie besitzen auch keine <*metagoal*> Tags, die zum Auswahl zwischen mehreren Plänen benutzt werden. Jeder Agent hat klar definierte einzelne Ziele, die nur einen einzelnen Plan zum Ausführen der Agentenaktivitäten bzw. Erreichen eines festgelegten Ziels haben. Die Agenten haben keine Zielalternativen und Planalternativen. Das deutet darauf hin, dass das Grundkonzept der zielorientierten Agenten *practical reasoning* im Framework nicht benutzt wurde.

Daraus folgend war die Architektur der BDI Agenten (s. Kap. 2.1.7) in MbFSIP nicht vollständig realisiert.

Trotzt der nicht benutzten Konzepte der BDI, bleibt eine Frage offen: welche Agentenarchitektur, reaktive Agenten mit ihren „Stimulus-Response-Architektur“ Prinzipien oder deliberative Agenten mit ihren proaktiven Eigenschaften (s. Kap. 2.1.6), den Agenten zugrunde liegen?

5.1.2 Charakterisierung der Agenten als Proaktiv oder Reaktiv?

Nach der vorgeführten Definition der Reaktivität, reaktives Verhalten und Proaktivität (s. Kap. 2.1.2) wird die Proaktivität mit dem Begriff Zielgerichtetheit verglichen (s. Kap. 2.1.6). Also mit Hilfe von Zielen eines Agenten lässt sich proaktives Verhalten eines Agenten zu implementieren. Im Betracht auf MbFSlP wird festgestellt, dass MAS über mehrere verschiedene Ziele wie *Achive Goal*, *Maintain Goal*, *Perform Goal*, *Query Goal* (s. Tabelle 4.1) verfügt und damit die

Anforderungen der Definitionen erfüllt. Ist es aber ausreichend für ein proaktives Agentenverhalten nur die Ziele besitzen?

> *„…, den um selbstständig initiativ werden zu können, muss ein Agent über wohldefinierte Ziele oder sogar ein komplexes Zielsystem verfügen. Nur dann macht es für einen Agent Sinn, aktiv auf seine Umgebung einzuwirken um seine eigenen Ziele zu verfolgen. Von Bedeutung ist in diesem Zusammenhang die Tatsache, wie umfangreich und komplex das jeweilige Zielsystem ist. Hat ein Agent beispielsweise nur das nicht näher definierte Ziel, Information über einen bestimmten Bereich zu sammeln, so kann er nicht viel tun, als bestimmte Informationsquellen zu überwachen und auf die Änderungen, in diesem Fall das eintreffen neuer Informationen aus dem für ihm interessanten Bereich, zu reagieren. Ein umfangreiches Zielsystem würde dahingegen nicht nur aus einem generellen Gesamtziel bestehen, sondern sich aus einer Vielzahl von Teilzielen zusammensetzen, mittels derer der Agent seine Aufgaben wesentlich gründlicher erledigen kann. Nur mit derartigen komplexen Zielsystemen ist echtes proaktives Verhalten möglich.“ [Brenner et al 1997]*

Nach dieser Definition es ist durchaus möglich, dass ein Agent trotzt der existierenden Ziele ein reines reaktives Verhalten aufweist. Das MAS soll ein komplexes Zielsystem mit Teilzielen verfügen, um eine echte Proaktivität zu beweisen. Das Agentensystem MbFSlP verfügt über ein komplexes Zielsystem mit unterschiedlichen Zieltypen und Teilzielen *Sub Goal*. Es ist aber wieder nicht ausreichend, ein MAS als proaktiv zu bezeichnen. A. Pokahr steigt eine weitere Ebene tiefer und beschreibt, dass die *Sub Goals* auch in der Lage sind, rein reaktives Verhalten zu besitzen:

> *„Anderes als in Objektorientierung erfolgt die Auswahl der Implementation jedoch nicht aufgrund einer zuvor expliziert getätigten Variablenzuweisung, sondern automatisch und adaptiv, bezogen auf den aktuellen Umweltzustand, da die Pläne für Unterziele kontextsensitiv, z. B. auf Basis von Vorbedingungen, ausgewählt werden. Zu dem ist die PRS[35]- Architektur nicht auf die Ausführung eines Plans für ein Ziel beschränkt, sondern führt für jedes Ziel alle anwendbaren Pläne, bis dies erreicht ist. Im Vergleich zum traditionellen Methodenaufruf wird also nicht eine der verfügbaren Implementationen ausgeführt, sondern im Falle von Fehlern, evtl. gleich mehrere.“ [Pokahr 2007]*

Nach dieser Definition soll der Einsatz von *means-ends reasoning* gewährleistet um automatisch und adaptiv auf die Veränderungen der Umweltumgebung zu reagieren. Das wurde aber in MbFSlP als Teil von *practical reasoning* in der BDI Architektur nicht realisiert.

Nach den oben vorgeführten Definitionen wurde festgestellt, dass die Anwesenheit der Zielen und ihren komplexen Strukturen mit den Teilzielen nicht unbedingt als ein Teil der Proaktivität betrachtet werden soll. Was muss eigentlich in Acht genommen werden während der zielorientierten Implementierung der Proaktivität eines Agenten?

> *„Der Software- Agent hat seine eigenen Ziele, die entsprechend seiner eigenen Präferenzen und Strategien versucht zu erreichen. Er übernimmt die Initiative und verfolgt neue Möglichkeiten, wenn sich dies ergeben sollte. Zielgerichtetes Handeln alleine kann durch einfache Programmierung erreicht werden, indem der Designer*

35 Procedural Reasoning System (PRS)

> *von einem Anfangszustand ausgeht (Vorbedingung) und sowohl Ziel (Nachtbedingung) als auch Verfahren vorgibt. Unter der Bedingung einer statischen Umgebung und eines korrekten Verfahrens kann angenommen werden, dass bei Terminierung der Zielzustand eingetreten ist. Trifft dieser Annahme nicht zu, muss der Agent reagieren und sein Verfahren der Zielverfolgung den veränderten Bedingungen anpassen." [Eymann 2003]*

Daraus folgt, dass ein Ziel über einen Anfangszustand besitzen muss, der in einer Wissensbasis gespeichert wird z. B. b*eliefsbase*. Es muss auch ein Ziel selbst existieren, als eine Nachtbedingung (Zustand, der erreicht werden soll) und ein Verfahren, der beschreibt, wie das Ziel erreicht, kann z. B. Plan. Das alles muss auch von *practical reasoning* unterstützt werden, um die Zielverfolgung den veränderten Bedingungen anzupassen.

Nach der Betrachtung der Ziele und Überlegungen ist es klar, dass die Proaktivität der MbFSlP Agenten durch Einsatz von Jadex Zielen wie *Maintain Goal und Achive Goal* und die Reaktivität der Agenten durch *Perform Goal* und *Query Goal* gewährleistet werden. In der Programmierpraxis hängt den Bedingungen ab, ob *Maintain Goal* und *Achive Goal* proaktives oder nur reaktives Verhalten implementieren. Umgehend kann proaktives Verhalten auch durch Goal-Plan-Sequenzen enstehen.

Im MbFSlP System werden die *Maintain Goal* mit einer Bedingung (engl. target condition) eingesetzt. Das entspricht der Definition der Proaktivität. Der Agent übernimmt die Initiative, wenn die Bedingung verletzt war, und führt ein oder mehrere vordefinierte Verfahren (Planen) aus, um gewünschtes Ziel zu erreichen. Die benutzte Ziele *Achieve Goal* im Gegenteil zu *Maintain Goal* werden ohne Bedingungen in MbFSlP eingesetzt. Die Abwesenheit der Bedingungen <*targetcondition*> in *Achive Goal* lässt sich dieses Ziel mit *Perform Goal* vergleichen. Der einzige Unterschied zwischen beiden Zielen liegt in ihrer Erreichbarkeit. Das Ziel *Achiev Goal* ohne Bedingungen wird nach dem Ausführen des ersten fehlerfreien Plans erreicht. Das Ziel *Perform Goal* wird, in Gegensatz zu *Achieve Goal,* erst dann als erreicht bezeichnet, wenn alle Pläne des Ziels ausgeführt wurden.

In einem Fall wenn *Perform Goal* und *Achieve Goal* nur einen Plan zum Erreichen des jeweiligen Ziels hat und das *Achieve Goal* keine Bedingung über einen gewünschten Agentenzustand hat, sind beide Ziele äquivalent zueinander. Die Zielen sind gleich und erweisen reinen reaktiven Agentenverhalten, weil es nach dem Ausführen des erstens und einzigen Plans entschieden wird, ob das Erreichen des Ziels gescheitert ist oder nicht. Dabei die Abwesenheit der Bedingung macht die Ziele gleich im Hinblick auf die Reaktivität und die Abwesenheit der Planalternativen im Bezug auf die Ausführung der Ziele.

Damit lässt sich die Proaktivität von Agenten in MbFSlP nur auf das Ziel *Maintain Goal* abgrenzen.

5.1.3 Kommunikation (Nachrichtenprotokoll in Multiagentensystem)

Die erworbenen Kenntnisse der Agentenkommunikation werden benutzt, um zu entscheiden, ob LS/TS SemCom Programmiermodell für das Portieren geeignet ist.

In der Agentenkommunikation des Frameworks werden FIPA standardisierte Sprechakten-Performativen *inform, request, failure, confirm, agree,* selbst definierte Performativen wie z. B.

„simulation_start“ , nicht FIPA konforme Nachrichtenparameter wie *content-start* und *content-class,* die von Jadex zur Verfügung gestellt werden, benutzt. Die versendeten Nachrichten beinhalten nicht nur String-Objekte in ihrem Content, sonder auch die anwendungsspezifischen Objekte, die als *Java Bean* Objekte implementiert z. B. *Offer.java*.

Die MbFSlP Agenten versenden und empfangen einzelne asynchrone externe[36] und interne[37] Nachrichten zu bzw. von anderen Agenten und führen Gespräche miteinander. Ein Gespräch zwischen Agenten ist vergleichbar mit dem Versenden der synchronen externen Nachrichten.

In den meisten Fällen realisiert die Kommunikationsschnittstelle in MbFSlP eine *stimulus-response architektur*. Die Nachrichten werden als Anreiz betrachtet, der einen Agenten zur einer oder mehreren Aktivitäten, Reaktionen zwingt. Der Anreiz ist mit einer Marke der Petrinetzen vergleichbar (s. Kap. 2.3.2), die während des Übergangs von einem Zustand zu einem anderen eine Transition hervorruft. Das Versenden der Nachrichten geschieht in MbFSlP aus unterschiedlichen Gründen:

- Eintreffen einer internen oder externen Nachricht.
- Erreichen eines Ziels.
- Erfüllung von bestimmten Bedingungen

Das Nachrichtenprotokoll des Frameworks (Jadex Version) ist im Anhang A4 Laufzeit Eigenschaften zu finden.

5.1.4 Verteilung in dem Framework und in der Multiagentensimulation

Die Agentenanwendung stellt ein verteiltes System dar (s. Kap. 3.7). Die Multiagentensimulation an sich selbst ist aber nicht verteilt. Alle Softwareagenten werden auf einer Maschine bzw. einer Agentenplattform ausgeführt. Die Architektur des MAS enthält die Merkmale eines verteilten Systems z. B. Einsatz eines zentralen Koordinators in der Agentenanwendung (s. Kap. 4.1.1.4 [Ruwinski und Timotin 2007]), andererseits wurden in MbFSlP trotzt der Existenz einer zentralen Konfigurationsdatei und ihres Datenmodells, keine Mechanismen festgestellt, die das Übermitteln der Agentendaten oder Simulationsdaten an eine entfernte Maschine gewährleisten. Die globalen Parameter werden lokal erzeugt und gelesen. Das Versenden der Parameter während des Nachrichtenaustauschs zwischen Agenten wurde auch nicht nachgewiesen.

5.1.5 Anwendungsdatenbank

DB der Agentenanwendung stellt ein Blackboardsystem (s. Kap. 2.2.1) dar. Die Environment-, Truck- und HUB-Agenten veröffentlichen ihre Simulationsdaten für alle Beteiligten des Agentensystems und benutzen die veröffentlichen Daten des Blackboardsystems, um ihre eigenen Aktivitäten, Handlungen auszuführen. Das Schreiben und Lesen der Information erfolgt über eine Persistenzschicht *DBService.java*, die auch das Filtern der relevanten Informationen realisiert.

---

36 Externe Nachricht: Eine Nachricht, die von einem Agenten zu einem anderem Agent übermittelt wird.
37 Interne Nachricht: Eine Nachricht, die innerhalb eines Agenten übermittelt wird.

5.1.6 Parametrisierung der Simulation

Das Multiagentensystem-basiertes Framework (MbFSlP) war entworfen um logistische Prozesse eines Paketdienstunternehmens zu simulieren. Dabei, wie die meiste Simulationen, besitzt diese Agentensimulation ein vereinfachtes Modell der Realität einer logistischen Welt mit Eingabevariablen für das Konfigurieren der Ausgangssituationen und Ausgabevariablen zum Analysieren eines Experimentes. Die Ausgangssituation der Simulation wird in einer *.xml Konfigurationsdatei, DB, Globale Parameter und unmittelbar in ADF Deskriptoren der Agenten definiert. Auf Grund der Entwicklungsphase des Produktes sind die Konfigurationsparameter über die ganze Anwendung verstreut. Die DB ist im Besitz der Initialisierungsparameter für die HUB Agenten. Diese Parameter sind sehr wichtig für den Start der Agentenanwendung. Sie werden sowohl von Agenten als auch von Analysetool und Controltool benutzt. Die Reihenfolge der HUB in DB ist nicht zufällig, sondern besitzt eine Struktur. So sind die ersten sechs HUB in der Tabelle „*ALLHUB“* von Type Metropol-HUB. Die Nächsten drei sind Versand-HUB und die Letzten Standard-HUB, die wieder in zwei Hälften geteilt sind, Süd- und Nord-HUB. Die XML-Datei wird automatisch erzeugt, falls sie noch nicht existiert. Sie wird immer mit standarddefinierten Parameter und ihren Werten erzeugt. Die Parameter können nach der Erstinitialisierung erweitert werden und später in der Simulation angewendet werden. Allerdings muss bekannt gegeben werden, dass die neuen Simulationsparameter nicht gesichert werden und nach der neuen Generierung der XML-Datei mit gleichen Namen verloren gehen.

GlobalParameter.java repräsentieren ein Datenmodell der XML-Konfigurationsdatei. Diese Parameter im Gegenteil zu XML-Parameter besitzen statische und dynamische Eingabevariablen. Die dynamischen Parameter werden im Laufe des Simulationsexperimentes geändert wie z. B. Variable „*truck*“, die Anzahl der generierten Truck-Agenten in der Simulation speichert. Die Truck-Agenten werden sequenziell über alle HUB-Agenten generiert und mit jedem neuen Fahrzeug wird die Variable „*truck*“ inkrementiert, bis Limit der LKWs im HUB-Agenten oder in gesamter Simulation erreicht wird.

Die gesamte Menge der Parameter soll in zwei Kategorien aufgeteilt: Die Erste soll die Parameter beinhalten, die einen Einfluss auf den Verlauf eines Experimentes haben und in der Zweiter sollen die Parameter kategorisiert werden, die einen direkten Einfluss auf die Systemauslastung oder Rechnerauslastung haben (s. Kap. 5.4.2).

5.1.7 Wartbarkeit

Die Wartbarkeit einer Anwendung wird nicht nur nach Merkmalen der Wartung aus Kap. 2.4.1 eingeschätzt, sondern auch aufgrund ihrer Komplexität. Das zu untersuchende Framework besteht aus mehreren Softwarekomponenten wie Controltool, Analysetool, DB, Agentensimulation, deren Verteilungsmöglichkeit einen Grad der Wartbarkeit minimiert. Die mehreren Konfigurationsdateien für DB, Agentensimulation und einzelne Agenten (ADF-Deskriptor) machen die Anwendung noch komplexer.

Der Wartungseinsatz ist durch den Einsatz der Agentensimulation, MAS, Agenten mit ihrer Autonomietät, Proaktivität, Fähigkeit zum Handel, Versenden der asynchronen und synchronen Nachrichten, Versenden der externen und internen Nachrichten und explizierten oder implizierten Einfluss auf die Umgebung oder Handlungen anderer Agenten erschwert.

Die einzelnen architektonischen Komponenten des Frameworks sind qualitativ gut und ausführlich in der Arbeit von Herren Ruwinski und Timotin beschrieben, was die Wartbarkeit der Anwendung erhöht. Aber die knappen oder ganz fehlenden Inlinekommentaren reduzieren diesen Vorteil sehr stark. Die verstreute Initialisierungsparameter, fehlende Nachrichtenprotokoll und Beschreibung der Synchronisationsmechanismen in der Agentensimulation erschweren nicht nur Analyse, sondern auch die Änderungen in der Anwendung.

Die Auswirkung der Änderungen auf die Stabilität der Agentenanwendung ist schwer vorherzusagen, einzuschätzen.

Einsatz der verteilten Architektur und Agentenorientierung macht die Testbarkeit komplizierter und aufwendig. Die Resultate können nur mit Erwartungen oder Realen Daten verglichen werden. Das Verhalten der Simulation kann zum Teil nur aufgrund eigener eingeschätzt werden.

Jadex ist einfach zu lesen und sein Programmiermodel ist leicht nachzuvollziehen. Der Einsatz verschiedenen Zielen minimiert die Wartbarkeit. Die Unterziele, mit ihrer Bereitschaft jeder Zeit zu expandieren, können leicht unterschätzt werden. Während des Reverse Engineering einer Jadex Anwendung entsteht ein großer Bedarf an Dokumentation der Entwickler. Im Verlauf dieses Prozesses taucht immer wieder der Informationsbedarf auf:

- Ziel: Welcher Plan führt das Ziel aus? Welche Aktivitäten werden nach dem Start eines Unterziels zur Ausführung gebracht? Welche globalen Ziele sind im Moment initialisiert oder verfolgt werden?

- Bedingung: Wenn die Bedingung in ADF definiert ist wo werden dann im Quellcode die relevanten Beliefs gesetzt? In welchem Plan wird die Bedingung benutzt? Was ändert die Bedingung? Welcher Plan oder welche Pläne warten auf die Bedingung?

- Belief: Ein Belief kann als eine Variable in einer Bedingung benutzt werden. In welcher? Welches Ziel wartet auf einen bestimmten Wert des Belief? Welche Aktivitäten werden ausgelöst, wenn einem Belief ein Wert zugewiesen wird? Wo war dieses Belief im Quellcode benutzt? Wo war das Belief initialisiert, in einem Plan oder in ADF?

- Plan: Wer oder was hat den Plan aktiviert? Die Parametrisierung eines Planes aus dem Quellcode ist nicht nachvollziehbar. Hat der Plan irgendwelche Parameter? Wird der Plan von einem oder mehreren Zielen benutzt?

- Nachricht: Eine Nachricht kann gleich als Trigger in einem Plan eines ADF benutzt werden und auch im Quellcode als Wartebedingung. Wer oder was wartet auf eine Nachricht? Wer hat die Nachricht geschickt? In welchem Plan wird die Nachricht empfangen?

5.2 Portierungsanalyse

In diesem Abschnitt der Arbeit wird es auf Grund der oben betrachteten Anhaltspunkte der Agenten für ein geeignetes Programmiermodell entschieden. Die Portierungsanalyse beschreibt:

- Auswahl einer geeigneten Verteilungsmöglichkeit der Agentenanwendung sowohl einer Plattformedition als auch eines Programmiermodells.

- Änderungen der Agentenanwendung.
- Synchronisierungsmechanismen des MASs
- Analyse der Programmiertechniken

5.2.1 Verteilungsmöglichkeit der Agentensimulation

Whitestein bietet zwei Lösungen für eine Verteilung der Agentenanwendung: Föderation Modus und Cluster Modus (s. Kap. 4.2.1). Föderation Modus wird von Business und Enterprise Editionen der Laufzeitumgebung unterstützt. Cluster Modus wird nur durch Einsatz der Enterprise Edition möglich. Beide Varianten ermöglichen das Verteilen einer Agentenanwendung auf mehrere Rechner, allerdings mit einigen gravierenden Unterschieden. So vereinigt der Föderation Modus von einander unabhängige Teile einer Anwendung. Jede Instanz der Laufzeit ist im Besitz ihrer eigenen DB und vollständig unabhängig von dem Rest der verteilten Anwendung. Der Cluster Modus wird erst mit dem Einsatz von Applikation Server und Zusammenführen von Agenten mit EJB Technologie möglich. In diesem Modus werden alle Laufzeitumgebungen zu einer virtuellen Einheit zusammengefasst, die eine Anwendung ausführt und eine einzige DB hat.

Die Vorteile eines Applikation Servers sind klar und deutlich: Zuverlässigkeit, Skalierbarkeit, Wartbarkeit und Sicherheit. Im Betracht auf die Aufgabestellung ist die Skalierung eine relevante Eigenschaft. Applikation Server macht eine Skalierung ohne Änderungen in einer Anwendung möglich. Die Verteilung der Agentensimulation mit Föderation Modus erfordert allerdings keine zusätzliche Technologien. Die Agentenanwendung kann jeder Zeit mit Enterprise Edition in Cluster Modus ausgeführt werden und damit die vollständige Abwärtskompatibilität für alle Editionen der Laufzeitumgebungen beibehalten. Der Nachteil der Föderation ist, dass die Synchronisierung zwischen verteilten Laufzeitumgebungen auf den Schultern eines Entwicklers liegt und nicht voll automatisiert ist, wie bei Cluster Modus eines Applikation Servers.

5.2.2 Auswahl der geeigneten LS/TS Produkten

Die Entscheidung für eine Edition der Laufzeitumgebung wird als Erstes getroffen, weil sie die Arbeitsvorgehensweise des Projektes bestimmt.

Die Integration der PE zusammen mit LS/TS Development Suite in die Eclipse IDE macht ihren Einsatz im Entwicklungsprojekt unverzichtbar und gilt als erste Voraussetzung. Allerdings unterstützt diese Edition das Verteilen der Agentenanwendung nicht. Das ist ein Grund zum Einsatz der Business Edition mit Föderation Modus. Dazu wird die Ressourcen Verwaltung der Edition in Betracht genommen, die Einsatz von großen Agentenmengen und einem großen Nachrichten Durchsatz möglich macht. Die Enterprise Edition bleibt bei der Entscheidung nur Wünschenswert aber nicht notwendig um das Projekt zu realisieren. Eine entwickelte Agentenanwendung mit Personal Edition und getestet mit Business Edition kann jeder Zeit mit Enterprise Edition ausgeführt und damit auch im Bezug auf die Skalierung optimiert werden.

Mit dem nächsten Schritt wird für ein Programmiermodell des Whitestein Produktes entschieden. Zum Analysieren stehen vier Modelle zur Verfügung: MDAL, MARGE, LISA und SemCom (s. Kap. 1) und drei verschiedenen Agentenarchitekturen zur Auswahl: reaktive Agenten, proaktive

Agenten und Agenten mit semantischer Kommunikation. Um ein geeignetes Programmiermodell zu wählen, soll zuerst die Architektur der Simulationsagenten bestimmt werden.

Die Analyse zeigt, dass die Agenten trotz der zugrunde genommener BDI Architektur nicht deliberativ sind (s. Kap. 5.1.2). Die Ziele, unabhängig von ihrer Komplexität und unterschiedlichen Zieltypen, zeigen ein reaktives Agentenverhalten. Die Analyse zeigt auch, dass die Untererziele zum Realisieren des reaktiven Agentenverhaltens benutzt wurden und nur zum synchronen Aufruf der Agentenaktivitäten eingesetzt, was mit Aufruf einer OO Methode vergleichbar ist. Die Agenten haben keine Zielalternativen und keine Planalternativen. Jedes Top-Ziel oder Sub-Ziel besitzt nur einen direkten Plan, somit stellt das Programmiermodell des Frameworks eine eins zu eins Abbildungstabelle eines Ziels zu genau einem Plan. Die Proaktivität eines Agenten wird nur durch *Maintain Goal* realisiert. Damit realisieren die Agenten unabhängig von ihrer minimalen Proaktivität eine reaktive Architektur.

Die Kommunikation zwischen Agenten erfolgt mittels Nachrichtenübermittlung. Es werden sowohl direkte Kommunikation mit Unicastnachrichten als auch Multicastnachrichten benutzt. Die vordefinierte Interaktionsprotokolle des Jadex Programmiermodells wurden in der Agentenanwendung nicht benutzt. Die meistens Agentendialogen enden mit einer Antwort auf eine Anfrage oder einer Bestätigung auf einen Befehl. Das Nachrichtenprotokoll basiert sich auf dem *stimulus-response prinzip* (s. Kap. 5.1.3).

Die Agenten sind reaktiv und schließen den MARGE Einsatz aus. LISA ist ein nicht abgeschlossenes Experiment und fällt unabhängig von ihren Verbesserungen auch aus. Das SemCom Programmiermodell kommt auch nicht zum Einsatz, weil die Agentenkommunikation einfach ist und die meisten Eigenschaften beim Einsatz nicht angesprochen bleiben. MDAL ist die geeignetste Auswahl von allen Modellen. Das Modell unterstützt reaktive Agenten. Die Nachrichten des Frameworks, mit ihrem Stimulus Response Prinzip, sind in das Petrinetz Konzept des MDAL Programmiermodells integrierbar. Es ist für nachrichtenorientierte Anwendungen konstruiert und außerdem ist ein Fundament, das mit der Zeit erweitert oder verbessert wurde. Die MDAL Mechanismen sind in jedem Programmiermodell enthalten.

Nach der Entscheidung für MDAL Programmiermodell soll zwischen Petri Netze oder Prozess Algebra entschieden werden. Diese Entscheidung wird aufgrund unvollständiger Unterstützung der Prozess Algebra von LS/TS Test-Framework für Petrinetze getroffen.

5.2.3 Unvermeidliche Änderungen im Bezug auf die LS/TS Restriktion

Aus Whitestein Dokumentation (vgl. [Run Time 2006]):

- Ein Agent darf nicht eine Nachricht lange Zeit bearbeiten oder seine Ausführungszeit einfach für eine unbestimmte Zeit nur für sich behalten. Das kann die Reaktion der restlichen Agenten beeinflussen. Wenn viele Agenten ihre Ausführungszeiten unbegrenzt oder für lange Zeit besitzen, kann das ganze Agentensystem für die Nachrichten unempfänglich sein. Wenn eine Agentenanwendung die persistenten Agenten benutzt oder die Agenten mit DAO zusammenarbeiten, kann es zu *time-outs* oder *deadlocks* führen.

- Ein Agent darf nicht sein eigenes Server Socket öffnen oder sein eigenes RMI Server erzeugen.

Die großen Agenten Mengen erfordern Einsatz von Ressourcen Manager, der die Nachrichten oder auch die Agenten selbst in System DB persistent schreibt. Das ist ein Grund die Jadex Pläne aufzuteilen um ihre Bearbeitungszeit zu reduzieren. Ein weiterer Grund dafür ist das Verteilen der Agentensimulation, das die Integration der Anwendung DB zu LS/TS und dadurch einen Einsatz der DAO-Agenten hervorrufen kann.

Das originale Framework baut eigene Verbindung zu DB auf. Der Socket wird allerdings nicht expliziert in einem Agenten erzeugt und geöffnet sonder in einer separate Klasse. Das kann trotzdem zur Komplikationen führen. Aus dieser Begründung ist es nicht ausgeschlossen, dass die DB in Whitestein integrieren sein soll.

5.2.4 Synchronisationswege zum Realisieren der regelbasierten, verteilten Simulation mittels Jadex Framework, Nachrichtenkommunikation und zielorientierten Techniken

Zum Synchronisieren der Agentensimulation oder zum Realisieren ihres regelbasierten Verhaltens wurden unterschiedliche Synchronisationsmechanismen realisiert. Die Simulation wird mit Hilfe der internen und externen Nachrichten, Bedingungen und Zustandsvariablen in Beliefsbase, Topzielen und Unterzielen, Trigger und Warteschlangen synchronisiert. Dabei können die Mechanismen zwischen zwei Gruppen unterschieden werden: proaktive und reaktive.

Hier sind Beispiele der Synchronisationsmechanismen einer proaktiven Gruppe.

Listing 5.1: Truck.Agent.xml

```
<maintaingoal name=”maintaingoal_login”>

    <maintaincondition>
        !($beliefbase.next_login_in_round == $beliefbase.round)
    </maintaincondition>

</maintaingoal>
```

Listing 5.2: HUB.agent.xml

```
<condition name="create_new_packets">
    <![CDATA[  ($beliefbase.flag_add_new_packets == true) ]]>
 </condition>
```

Listing 5.3: CreateNewPacketsPlan.java

```
while (true) {

    ICondition condition = getCondition("create_new_packets");
    waitForCondition(condition);
…
}
```

Die Beispiele stellen eine Implementierung eines Agentenpoaktivitätes dar, allerdings mit Hilfe von zwei verschiedenen Techniken: agentenorientierte (s. Listing 5.1) und Objektorientierte (s. Listing 5.2 und Listing 5.3). Die beide Mechanismen verfolgen nur ein Ziel: Ergreifen der Initiative, wenn bestimmte Bedingungen verletzt oder erfühlt sind. Damit sind beide Mechanismen im Prinzip äquivalent zu einander.

Die reaktive Gruppe der Synchronisierungsmechanismen wird sowohl mit Hilfe der objektorientierten als auch nachrichtenorientierten und agentenorientierten Techniken realisiert. Während die ersten zwei Techniken ganz trivial sind, wie z. B. if-Abfrage einer Belief Variable oder Aufruf eines Planes zum Ausführen durch eine Nachricht, Versenden einer synchronen Nachricht, stellt die letzte agentenorientierte Variante einen besonderen Fall vor. In diesem Fall wird mit Unterzielen eines Agenten synchronisiert (s. Listing 5.4).

Listing 5.4: LogoutTrucksPlan.java

```
for (Truck truck : trucks.values()) {

    IGoal logout = createGoal("goal_truck_logout");
    logout.getParameter("truck").setValue(truck);
    dispatchSubgoalAndWait(logout);

}
```

Die Proaktivitätsanalyse (s. Kap. 5.1.2) zeigt, dass die Simulationsteilnehmer keine Planalternativen besitzen. Dieses Merkmal macht diese Technik mit einem gewöhnlichen Aufruf einer Methode in Objektorientierung vergleichbar. Die Unterziele werden aber nicht nur zum Synchronisieren von einzelnen Agentenaktivitäten, sondern auch zum Definieren der Regelsequenzen eingesetzt. Der folgende Beispiel (s. Listing 5.5) aus der Klasse *StartTradePlan.java* zeigt, wie mit Unterzielen ein endlicher Automat mit klar definierten Zustanden aufgebaut sein kann:

Listing 5.5: StartTradePlan.java

```
while (true) {

    ICondition condition = getCondition("start_trade");
    waitForCondition(condition);

        for (int i=1; I <= GlobalParameters.TRADE_ROUNDS; i++) {

                getBeliefbase().getBelief("trade_round").setFact(i);
                IGoal make_bids_goal = createGoal("performgoal_make_bids_for_packets");
                dispatchSubgoalAndWait(make_bids_goal);

                IGoal inform_trucks_goal = createGoal("performgoal_inform_truck_round");
                dispatchSubgoalAndWait(inform_trucks_goal);

                IGoal matching_goal = createGoal("achievegoal_matching");
                dispatchSubgoalAndWait(matching_goal);
        }

        IGoal logout_goal = createGoal("performgoal_logout_trucks");
        dispatchSubgoalAndWait(logout_goal);

        IGoal matching_goal = createGoal("performgoal_send_ack");
        dispatchSubgoalAndWait(matching_goal);

}
```

5.2.5 Einfluss der Programmiertechniken auf die Portierung

Einen besonderen Einfluss auf die Portierung haben die eingesetzten Programmiertechniken in der Simulation. In der gesamter Agentensimulation werden die Namen für Belief und Ziele direkt als Zeichenketten benutzt (s. Listing 5.5).

Der Einsatz einer statischen Variable für Belief könnte die Reverse Engineering erleichtern und das Verfolgen der Wertzuweisung für einzelne beliefs im Programmiercode fürs Möglich machen. Es wäre auch ein großer Plus für Belief, die in Bedingungen benutz waren und für die Bedingungen selbst (s. Listing 5.2). Die statische Variable würden einem Entwickler eine Möglichkeit geben, die Stellen in Quellcode schnell zu finden, wo die Bedingungen verändert wurden und die Stellen wo auf diese Veränderung gewartet wird.

Die Zielnamen könnten in zugehörigen Plänen statisch definiert werden (ein Plan der von dem Ziel aufgerufen wird). Das gilt allerdings nur für aktuelle Agentenanwendung mit einer eins zu eins Abbilden der Ziele auf Pläne. So z. B. der Name des A*chieve Goal* „goal_truck_logout“ könnte im Plan „LogoutTruckPlan“ statisch definiert werden und damit jeder Aufruf dieses Ziels in Quellcode verfolgbar machen (s. Listing 5.6).

Listing 5.6: HUB.agent.xml

```
<achievegoal name="goal_truck_logout">
    <parameter name="truck" class="net.portrix.packetworld.common.beans.Truck" />
</achievegoal>

<plan name="logout_truck">

    <parameter name="truck" class="net.portrix.packetworld.common.beans.Truck" >
        <goalmapping ref="goal_truck_logout.truck"/>
    </parameter>

    <body>new LogoutTruckPlan()</body>

    <trigger>
        <goal ref="goal_truck_logout" />
    </trigger>

</plan>
```

Größe Monolithe wie Matching (s. Kap. 6.1.5) Methode mit mehreren in einander geschachtelten for-Schleifen und while-Schleifen, tief geschachtelten if-Abfragen machen das Verstehen dieses Algorithmus sehr schwer und auch das Zerlegen in mehrere kleine Teile sehr aufwendig.

Das Versenden der synchronen Nachrichten aus einer Schleife erfordert besondere Anpassungstechniken bei dem Portieren.

Standardisierung der Synchronisationsmechanismen könnte auf die Implementierung nicht unbedingt einen positiven Einfluss bewirken, aber für Reverse Engineering eine enorme Erleichterung sein.

5.3 Risikoanalyse

Die zentrale Frage ist: ist die Portierung des Jadex basierten Frameworks auf die Whitestein Plattform möglich? Die Antwort lautet: Ja, die Portierung ist möglich. Beide Produkte sind FIPA konform. Sie unterstützen Java und bieten eine Agentenplattform mit Diensten, wie Directory Facilitator, Agent Management System und Kommunikationsdiensten, zum Versenden von asynchronen internen und externen Nachrichten, Realisierung einer verteilten Agentenanwendung. Beide Produkte unterstützen ACL Grundlagen. Die Agenten sind auch in der Lage mittels vordefinierten Interaktionsprotokollen interagieren. Die Wissensbasis der Jadex Agenten und auch der LS/TS Agenten ist ein assoziativer Speicher mit „key-value tupel“ und ein Jadex atomare Block mit einem „step“ des MDAL Fragments äquivalent ist.

Die Unterschiede zwischen beiden Plattformen und ihren Programmiermodellen machen die bevorstehende Portierung aufwendiger und komplizierter aber nicht unmöglich. Der Unterschied zwischen Jadex und LS/TS liegt in Unterstützung verschiedenen Agentenarchitekturen. Jadex ist ein Programmiermodell, das für proaktive Agenten konzipiert ist und LS/TS erlaubt Einsatz von proaktiven und reaktiven Architekturen. Zum Portieren wurde für ein nachrichtenorientiertes MDAL-Programmiermodell entschieden, das zum Implementieren der reaktiven Agenten konzipiert wurde (s. Kap.5.2.2). Die reaktiven Agenten des Frameworks (MbFSlP) wurden mittels zielorientierten Techniken gebaut und besitzen trotz ihrer Architektur ein Teil der Proaktivität, das mit *Maintain Goal* implementiert wurde (s. Kap. 5.1.2). Genau dieses Merkmal soll als einer der Schwerpunkt der Portierung betrachtet werden. Die *Maintain Goal* Eigenschaften sollen in reaktiven Agenten mithilfe OO Techniken realisiert werden.

> *„Um Proaktivität dennoch als Teil des Agentenmodells einsetzen zu können, müssen Datenstrukturen für Ziele und proaktive Verhaltungsweisen vom Programmierer zusätzlich erstellt werden.“ [Pokahr 2007]*

T. Eymann (s. Kap. 5.1.2) beschreibt, wie ein Zielgerichtetes Handeln durch einfache Programmierung ersetzt sein kann. Ein weiterer Unterschied und auch ein Schwerpunkt der Portierung sind die synchron aufgerufenen Jadex Unterziele. In Kap. 5.2.4 wird beschrieben, dass die Unterziele mit einem gewöhnlichen Aufruf einer OO-Methode äquivalent sind. Aus diesem Grund soll ihr Zielgerichtetes Handeln nicht als ein besonderer Fall behandelt werden. Hier spielt die Synchronität des Aufrufes eine entscheidende Rolle. Die Unterziele rufen eine Agentenaktivität hervor, die als einem MDAL Plan in Jadex Programmiermodell repräsentiert. Die Pläne sind als Java Threads implementiert. Bei einem synchronen Aufruf eines Unterziels wird ein aufrufender Plan so lange suspendiert, bis ein aufgerufener Plan ausgeführt ist. Dieses Merkmal wird oft als Synchronisierungspunkt für die ganze Agentensimulation benutzt. Ein Beispiel dafür ist die Regelsequenz aus der Klasse *StartTradePlan.java* (s. Listing 5.5) dargestellt. In dem ausgewählten MDAL Programmiermodell werden die Agentenaktivitäten durch ein Fragment repräsentiert, das in mehrere kleine Teile sogenannte „steps“ aufgeteilt ist. Auf Grund der LS/TS Restriktionen (s. Kap 5.2.3) dürfen die „steps“ nicht sehr groß sein und eine Agentenaktivität in kleine Aktivitätspartikel aufteilen. Diese Aufteilung ermöglicht das Ausführen von anderen Agentenaktivitäten zwischen einzelnen „steps“, was eigentlich im Fall mit Jadex Unterzielen nicht erwünscht ist. Dieser Unterschied erfordert Einsatz von anderen Synchronisationsmechanismen.

Ein weiterer Unterschied zwischen Jadex und Whitestein liegt bei der Fähigkeit eines Jadex Agenten eine synchrone externe Nachricht zu versenden. Diese Eigenschaft soll mithilfe von MDAL Mechanismen nachgebildet werden (s. Abbildung 6.6).

Die LS/TS Restriktionen erfordern das Zerlegen der monolithischen Methoden wie z. B. Matching (s. Kap. 6.1.5). Dieser Prozess hängt vom Grad der Wartbarkeit einer Agentenanwendung und Unterstützungsqualität der Reverse Engineering von Jadex ab. Aus dem Grund, dass die beiden Abhängigkeiten nur zum Teil realisiert sind, wird es als ein nächster Schwerpunkt der Portierung betrachtet.

Zu den weiteren Schwerpunkten der Portierung zählt die Wiederherstellung eines Nachrichtenprotokolls (s. Anhang A4 Laufzeit Eigenschaften) und Einsatz der Petri Netzen (s. Kap. 2.3.2) zum Realisieren der parallelen Prozesse und ihrer Synchronisierung. Im Bezug auf die Optimierung der Agentensimulation, nämlich ihre Verteilung, kann die Notwendigkeit an der DB-Integration in Whitestein entstehen (s. Kap. 7.4). Das kann als ein möglicher Schwerpunkt der Portierung betrachtet werden.

5.4 Praktisches Testen

Die Erprobungsphase oder ein praktisches Testen der Agentenanwendung soll ihre Stärke und ihre Schwäche zeigen. Vor allem sollte ein klarer Überblick über Stabilität, Skalierung, Funktionalität der Agentenanwendung und den Verlauf der Simulation geschafft werden. Der praktische Test ist im Anhang A4 Laufzeit Eigenschaften dargestellt. Hauptaufgabe dieses Tests war die Untersuchung der wachsenden Zahl der Pakete im System. Deshalb mussten beschriebene im Anhang Recherchen eingesetzt werden.

5.4.1 Laufzeit Eigenschaften des Multiagentensystems (MbFSlP)

Das Framework verfügt über eine Schnittstelle, die eine vordefinierte Konfigurationsdatei mit Initialisierungsparametern für die Simulation und ihre Teilnehmer generiert. Die ersten Testversuche waren mit diesen standardisierten Parametern ausgeführt. Die vordefinierten Parameter können nicht sofort zum Simulieren eingesetzt werden, ohne sie zu konfigurieren, angepasst zu haben. Im Verlauf der weiteren Erprobungstests wurde der Simulationsverlauf in meisten Fällen vom System mit einer Fehlermeldung *„OutOfMemoryError“* abgebrochen. Bei dem Simulationsverlauf auf einer Maschine mit *Core 2 Duo Intel Pentium Prozessor* und auf einer Maschine mit *AMD Athlon (tm) 64* (s. Fußnote mit einer technischen Beschreibung des Rechners)[38] wurden manchmal einige Truck-Agenten verloren und die Simulation wurde nach dem Auftreten der Anwendungsmeldung *„still awaiting acks from folows HUB“* in passives Warten versetzt. Ein nächstes Problem ist ein permanentes Wachstum der Paketobjekten in der Simulation, der die Agentenanwendung überlastet und in einer absehbaren Zeit nicht ansprechbar macht.

38 Prozessor Modell : Intel(R) Core(TM)2 CPU T5500 @ 1.66GHz Prozessor Typ: Mobile, Dual-Core, Front Side Bus Geschwindigkeit : 4x 166MHz (664MHz), Gesamtspeicher : 1.5GB DDR2 SO-DIMM, Speicherbusgeschwindigkeit : 2x 266MHz (532MHz), Betriebssystem: Windowssystem : Microsoft Windows XP Professional 5.01.2600 (Service Pack 3), Platform Compliance : x86.

Prozessor Modell : AMD Athlon(tm) 64 Processor 2800+, Geschwindigkeit: 1.8GHz, Front Side Bus Geschwindigkeit : 2x 800MHz (1.6GHz), Gesamtspeicher : 2GB DDR, Speicherbusgeschwindigkeit : 2x 164MHz (328MHz), Betriebssystem, Windowssystem : Microsoft Windows XP Professional 5.01.2600 (Service Pack 3), Platform Compliance : x86.

Der Laufzeiteigenschaften zur Folge wird angenommen, dass Systemfehlermeldung *„OutOfMemoryError“* mit dem permanenten Wachstum der Paketobjekten in der Simulation zusammenhängt und die Anwendungsmeldung *„still awaiting acks from folows HUBs“* ist nur ein Synchronisationsproblem des MASs. Im folgenden Arbeitsabschnitt werden die Problemstellen der Agentenanwendung lokalisiert, die ihr Laufzeitverhalten beeinflussen können.

5.4.2 Kritische Bereiche der Agentensimulation (MbFSlP)

Die kritischen Bereiche der Simulation werden in drei Kategorien aufgeteilt: semantische, funktionale und parametrische Simulationsbereiche.

In der Simulationssemantik können die Pakete über mehrere Spielrunden im System verbleiben, bis sie ans Ziel ankommen. Das kann ein Grund des Paketwachstums sein. Es gibt dafür unterschiedliche Gründe:

- Eine erfolglose Teilnahme an einer Verhandlung.
- Nicht genügender Transportmittel in einem HUB, um einen Paket zum gewünschten Zielort abzutransportieren.
- Die Lagerkapazität eines HUBs, die mit dem Verlauf der Simulation ausgeschöpft werden kann. Dabei kann die Knappheit an Transportmittel durch ein unkontrolliertes Handel der Trucks und Bestimmung ihres nächsten Ziel entstehen. Die Ausschöpfung der Lagerkapazität blockiert die Paket Reservierung. Das kann die Lieferfristen und die gesamte Stabilität der Simulation beeinflussen.
- Die Lernfähigkeit der Pakete. Die Pakete sind in der Lage eine Historie mit unterschiedlichen Informationen zu führen, die später bei der Auswahl einer Route oder Generieren eines Gebotes benutz werden können.

Zu funktionalen kritischen Bereichen gehört *MatchingPlan.java*, der eine zentrale Rolle in den Verhandlungen hat. Der Plan unterscheidet sich vom Rest der Anwendung mit seiner Rechenintensivität. Zu diesem Bereich gehört auch *MakePacketsBidsPlan.java*, der für jedes Paket ein Gebot generiert. Die Besonderheit an diesem Plan ist der Parameter „ALTERNATIVE“, der für jedes Paket im System mehreren alternativen Geboten erzeugt.

Die Simulationsparameter unterscheiden sich im Bezug auf die Skalierung der Simulation durch zwei Eigenschaften: rechen- und speicherintensiv.

- ALTERNATIVES. Der Parameter gibt die Anzahl der alternativen Geboten eines Paketes. Eine maximale Anzahl der Alternativen ist von der Anzahl der HUB abhängig. Um die Gesamtpaketmenge im System zur Verhandlungszeitpunkt auszurechnen, wird aktuelle Menge der Pakete im System mit Alternative multipliziert. Dieser Parameter ist sehr speicherintensiv.
- packets_in_system ist sowohl speicher als auch rechenintensiver Parameter. Die Menge der generierten Pakete im System hat einen direkten Einfluss auf die Ausführungszeiten des Match-Verfahrens und das Generieren der Gebote mit ihren Alternativen.

- trucks_number_limit beeinflusst die Speicherauslastung durch die Anzahl der Truck-Agenten im System und die Ausführungszeiten des Match-Verfahrens, weil jeder Truck an den Verhandlungen teilnimmt.

- K_PAKETS und K_TRUCK vermindern die Auswirkungen der Parameter packets_in_system und trucks_nummber_limit auf das System, haben aber gleiche Merkmale. Diese „K_Parameter“ definieren Paket- und Truckeinheiten (eng. units). So z. B. 100 Pakete pro 1 Einheit. Mit diesen Parametern wird die Grobkörnigkeit der Agentensimulation gesteuert.

- trade_rounds ist ein besonderer Parameter, der einen Einfluss auf die Qualität der Verhandlungen und auf die Ausführung des Match-Verfahrens hat. Dieser Parameter wird direkt beim Generieren der Gebote und Angebote benutzt. Je länger die Verhandlungsdauer ist, desto sparsamer handeln ihre Teilnehmer. Andererseits werden mit jeder weiteren Verhandlungsrunde die Erfolgschancen eines Trucks Agenten erhöht. Schließlich kann es sein, dass ein Agent kein passendes Gebot in der ersten Runde findet und erst in der nächsten Runde mit geringerer Konkurrenz erfolgreich in der Verhandlung wird.

- max_delivered_duration_exp und max_delivered_duration_std bestimmen die Lieferfristen eines Paketen. Je länger ein Paket im System verbleibt, desto größer ist die Gesamtpaketmenge.

5.4.3 Testauswertung

Während der Erprobungsphase wurde die Skalierung und Stabilität der Agentenanwendung getestet. Es wurden kritische Bereiche der Agentensimulation festgestellt und einige Parameter auf ihre Auswirkungskraft untersucht. Das Problem der wachsenden Pakete im System konnte leider nicht eliminiert werden. Ein Grund dafür sind wahrscheinlich die vernachlässigten Strategien der Paketobjekten und Truck-Agenten, die zusammen mit Konfigurationsparameter einen gewünschten Effekt zeigen können. Es ist unbestritten, dass die Lernfähigkeit der Paketobjekte einen großen Einfluss auf die Speicherbelastung hat. Die letzten zwei Testen zeigten, dass Jadex Plattform unabhängig von der Objektmenge und Durchsatz der Nachrichten im System zuverlässig und stabil ist. Der Einfluss von *Garbage Collector (GC)*[39] auf die Ausführung der Simulation kann leider durch manuellen Systemabbruch nicht festgestellt werden. Der Zuwachs der Objekte, der in Abbildung A4 1zu sehen ist, kann durch das Generieren der Pakete, Gebote oder Angebote erklärt werden, kann aber auch durch andere Faktoren entstehen. Die beiden Tests sollen eventuell wiederholt werden.

Nach dem Studieren des Whitestein Produktes und der Eigenschaften der Laufzeitumgebung stehen folgende Anforderungen an der Agentenanwendung:

- Um Skalierbarkeit der Anwendung zu erhöhen, soll die Agentensimulation verteilt verlaufen. Damit wird die Performance der gesamten Anwendung verbessert. Allein die Tatsache der Verteilung ermöglicht einen manuellen oder automatischen Lastverteilung der Simulation auf mehreren Rechner (Agenten Plattformen), sei es die Menge der Agenten oder Objekten.

- Die Ressourcen Verwaltung für nicht persistente Agenten erlaubt einen Einsatz von großen Agentenmengen.

---

39 http://msdn.microsoft.com/de-de/library/0xy59wtx.aspx (August 2009)

- Durch eine sparsame Threads Vergabe ist eine stabile Anwendung zu erwarten, die jede Zeit einem Anwender zur Verfügung steht.

- Die entfernte Kontrolle der Agentensimulation wird durch LS/TS Clients ermöglicht.

- Die Laufzeit der Agentensimulation (reale Zeit sowie die Spiel Runden) soll die Laufzeiten der original Jadex Version deutlich überschreiten. Der Begriff „unendliches Spiel“ soll in der Simulation nicht buchstäblich realisiert werden, sondern um ein paar Schritten nähe gebracht.

Die vorher festgelegten Anforderungen aus der Arbeit [Ruwinski und Timotin 2007] Kapitel 3.3.3 sollen ebenfalls erfühl bleiben und angehalten werden.

6 Design

Das Kapitel beschreibt das Design der portierten Agentenanwendung. Hier werden unterschiedliche Sichten auf die Architektur der Agentenanwendung, -system und -kommunikation dargestellt.

6.1 Redesign

Durch das Portieren des MbFSlP Frameworks auf LS/TS Plattform ist das Ändern der Softwarearchitektur nicht zu verhindern. Es verlangt einige Redesign-Maßnahmen durchzuführen. Das Verteilen der Agentensimulation erfordert ein einfaches Parametrisieren der Anwendung, Implementierung einer zentralen Stelle zum Kontrollieren, Steuern von Simulationsverläufen und Auswerten ihrer Simulationsdaten. MbFSlP verfügt über eine separate Softwarekomponente „Analysetool“ zum Auswerten der Informationen. Eine Schnittstelle zum entfernten Steuern und Kontrollieren der Simulation ist aber nicht vorhanden. Die bevorstehende Verteilung erfordert die Änderungen in der Parametrisierungsmechanismen, die für eine lokale Agentenanwendung entwickelt wurden. Die Parameter sollen nach Möglichkeit zusammengefasst und von einer zentralen Stelle für alle verteilten Simulationseinheiten zur Verfügung gestellt werden. Bezogen auf LS/TS Restriktionen (s. Kap 5.2.3) soll ein Notplan zur Hand liegen, wenn das Integrieren der Agentenanwendung DB zur Whitestein Plattform unvermeidlich wird. Das Zerlegen der großen Jadex Pläne in kleinere endliche Automaten, nach Restriktionen, wird eine Änderung der Synchronisationsmechanismen hervorrufen. Das Implementieren der Proaktivitäten eines Agenten und synchronen Nachrichten setzt ebenfalls Redesign-Maßnahmen voraus.

6.1.1 Verteilung der Agentenanwendung und ihrer Teilnehmer

Die Agentenanwendung besteht nach wie vor aus einer Agentensimulation und Hilfssoftware (s. Abbildung 6.1).

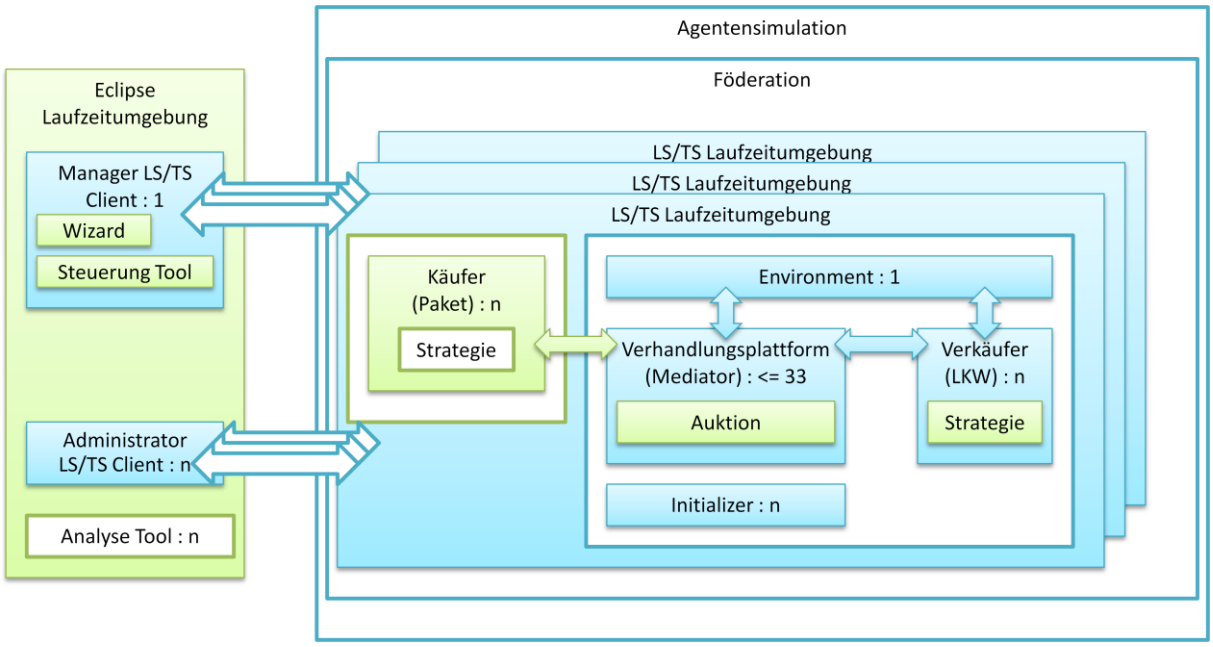


Abbildung 6.1: Simulationsteilnehmersicht der Agentenanwendung

Sie dienen zum Generieren der Konfigurationsparameter für die Agentensimulation, zum Unterstützen der Anwender beim Steuern, Kontrollieren und Analysieren. Die zuvor lokal ausführbare Agentensimulation verläuft verteilt. Die Tools (Analyse Tool, Administrator Client und Manager Client) sind aus der Eclipse IDE Entwicklungsumgebung ausführbar und stellen eine externe Komponente der Agentenanwendung dar. Das Analysetool wurde vollständig ohne Veränderungen übernommen. Der Administrator Client ist von Whitestein zur Verfügung gestellt und gibt eine Übersicht über alle angemeldeten Laufzeitumgebungen mit ihren Agenten und Auslastung der Agentenplattform. Der Manager Client ist genauso wie Administrator Client ein LS/TS Agent. Vorher stellte der Manager einen lokal getriebenen Agenten dar. Zu seinem Aufgabenfeld gehörten das Parametrisieren, Initialisieren und Starten der Agentensimulation. Die aktuelle Version erweitert die Aufgaben dieses Managers.

Der Manager ist separat von der Simulation ausführbar und stellt einen Agenten mit Client-Architektur dar. Er stellt eine Netzwerkverbindung mit allen Beteiligten an der Simulation her, startet, hält an und beendet eine verteilte Agentensimulation. Zu den weiteren Manageraufgaben gehört das Generieren neuer XML Simulationsparameter, ihr Modifizieren mithilfe eines Steuerungstools und einschließlich Versenden der Parameter an alle entfernten Laufzeitumgebungen. Die visuelle Steuerungssoftware kann im Gegenteil zu Jadex Version abgeschaltet werden, um die Performance des Systems zu verbessern. Das Steuerungsprogramm dient ausschließlich zum Verändern der existierenden Simulationsparameter.

Die Abbildung 6.1 zeigt alle Teilnehmer der Agentenanwendung mit ihren Kardinalitäten. Es darf nur eine Managerinstanz existieren, um einen Konflikt in Zuständen „sim start“, „sim end“ und „sim_run“ zu vermeiden. Ein mögliches Konfliktszenario: ein Manager beendet die Simulationsausführung, während ein anderer Client immer noch ihr Steuern für möglich macht.

Die Agentensimulation besteht aus einer Agentenföderation, die mehrere Agentenplattformen auf physikalisch entfernten Rechner in eine virtuelle Einheit zusammenfasst. Die Simulation kann beliebig viele Paketobjekte oder Truck-Agenten haben. Es dürfen allerdings nur 33 Verhandlungsplattformen (HUB) über alle Agentenplattformen der Föderation verteilt werden. In jeder Plattform existiert nur ein Initializer-Agent. Seine Existenz ist zwingend in der Simulation, weil er das Initialisieren der Parameter in der verteilten Agentenanwendung übernimmt. Ebenso darf es nur einen Environment-Agenten in der gesamten Agentensimulation geben, weil er für zentrales Synchronisieren der Simulation zuständig ist.

Intern verfügt der Manager über ein Wizard-Programm, das einen Anwender bei der Steuerung der Agentensimulation unterstützt. Die Abbildung 6.2 s. u. stellt einen internen Aufbau dieses Programms dar.

Der Wizard ist nicht in der Lage sein Zustand persistent zu schreiben. Aus diesem Grund beschafft er beim Start die Informationen über Verlaufszustand der Simulation. Ist die Simulation schon gestartet? Soll jetzt die Verbindung mit der Simulation[40] aufgebaut werden, um sie anzuhalten oder weiterlaufen zu lassen? Während der Erstinitialisierung der Simulation wird das Generieren der Standardparameter, Modifizierung der Parameter mithilfe eines Steuerungstools, Versenden der Parameter an Simulationsteilnehmer und einschließlich Starten der Simulation vorgeschlagen. Nach der Initialisierung und Starten der Simulation verfügt ein Anwender über die Möglichkeit sie anzuhalten, wieder zu starten, den Manager zu verlassen oder die Simulation zu beenden. Solange

40 Die Verbindung wird nur mit der Laufzeitumgebungen der Agentensimulation aufgebaut.

die Simulation nicht gestartet ist, wird es erlaubt fehlerhafte Aktivitäten rückgängig zu machen. Das Verlassen des Clients hat keinen Einfluss auf den globalen Zustand der Simulation und ihr Verlauf. Der Manager Client ist der einzige Agent, der über mehrere Parametrisierungsdateien die Simulation initialisieren darf.

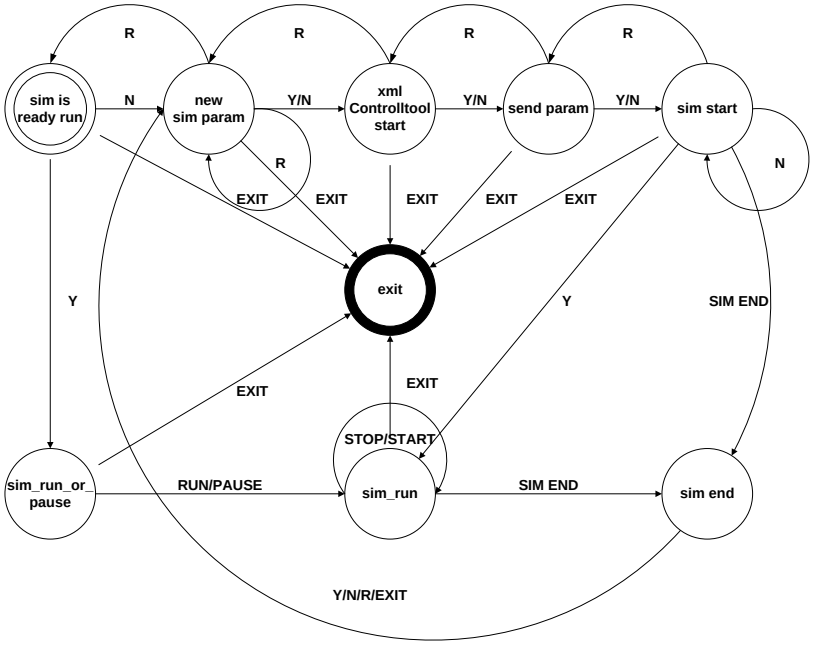


Abbildung 6.2: Wizard endliche Maschine

6.1.2 Integration der Datenbank in Whitestein

Die Agentenanwendung verfügt über zwei DB. Die Erste ist System DB, die von Whitestein zur Verfugung gestellt ist (s. Kap. 4.2.1). Die Plattform lagert ihre Agenten über Hibernate Schnittstelle in die DB, um die Persistenz der Agenten zu gewährleisten oder fürs LS/TS Ressourcen Manager Einsatz. Die Zweite ist eine Anwendung DB, die von Agenten, zum Speichern ihren simulationsrelevanten Daten benutzt wird (s. Kap. 4.1.5 [Ruwinski und Timotin 2007]).

Im weiteren Verlauf werden die betroffenen Redesign-Maßnahmen bei dem Integrieren der Anwendung DB in Whitestein beschrieben.

Der Framework Hibernate erlaubt einen Einsatz einer MySQL DB[41] und damit das Portieren der relationalen Tabellen in ein anderes *Datenbank Management System (DBMS)* fürs unnötig macht. Die implementierte Persistenzschicht (s. Kap. 5.1.4 [Ruwinski und Timotin 2007]).) der Jadex Version wird nur von Analyse Tool und Manager Client benutzt. Das Analyse Tool bleibt damit unverändert und der Manager vermeidet einen unnötigen Einsatz von höheren Technologien wie Hibernate. Das Verwenden der Anwendung DB aus der Logik eines autonomen Agenten erfordert das Implementieren von neuen reaktiven Agenten wie Servant und DAO (s. Kap. 4.2.2). Die Persistenzschicht aus der früheren Version wird durch Hibernate ersetzt.

41 www.mysql.de

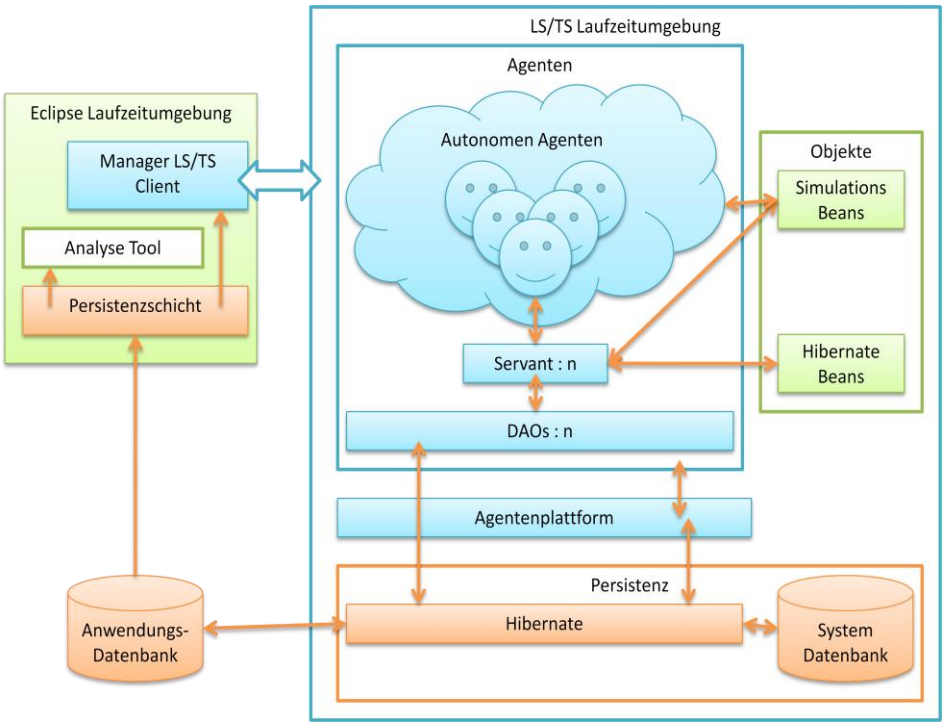


Abbildung 6.3: Persistenzsicht in der Agentenanwendung

Bezogen auf die Abbildung 6.3 speichern die autonomen Agenten ihre Daten in Simulation-Beans wie z. B. „net.portrix.packetworld.common.beans.Paket“. Sie werden zum Speichern an einen Servant Agent gesendet. Dieser Agent vorbereitet die Simulation-Beans zum Speichern und konvertiert sie schließlich zur Hibernate-Beans. Die konvertierte Beans werden an DAO, einen weiteren Mitglied der Persistenzkette gesendet um mit Hilfe der *Hibernate Query Language (HQL)*[42] die Beans über Hibernate Schnittstelle in die DB persistent zu schreiben. Damit teilt sich die Persistenzkette in mehreren Komponenten auf:

- Servant-Agent, der DB-Abfragen eines autonomen Agenten definiert und die relevanten Daten zum Speichern in die DB oder zum Konsumieren für einen Agenten vorbereitet.
- einen DAO Agent, der eine Sprache zum Abfragen der Daten und ihr Speichern bereit stellt.
- Hibernate, der Herstellen der Verbindung mit DB und das Übermitteln der Daten übernimmt.

Alle neue reaktive Agenten-Servant und DAO-Agenten müssen mindestens einmal in jeder Agentenplattform einer verteilten Agentensimulation vorhanden sein und werden automatisch, unmittelbar nach dem Start des MASs erzeugt.

6.1.3 Konfiguration und Parametrisierung der verteilten Agentensimulation

Die frühere Version der Agentenanwendung verfügte über eine Konfigurationsdatei für eine DB und eine Datei mit Parameter der Simulation. Die beiden Dateien wurden als wohlgeformte XML-Dateien implementiert. Sie besitzen allerdings keine XML Schema[43]. Das Generieren der Simulationsparameter wird von einem selbstdefinierten Parser übernommen. Der Manager

42 http://www.hibernate.org/hib_docs/nhibernate/html/queryhql.html (August 2009)
43 http://www.w3.org/XML/Schema (August 2009)

übernimmt das Starten eines Steuerungstools, das Erzeugen der Datei und eine implizite Initialisierung der globalen Parameter. Die neue Version der Simulation hat drei XML-Dateien (s. Abbildung 6.4): Verteilungsparameter, Simulationsparameter und DB Verbindungsparameter.

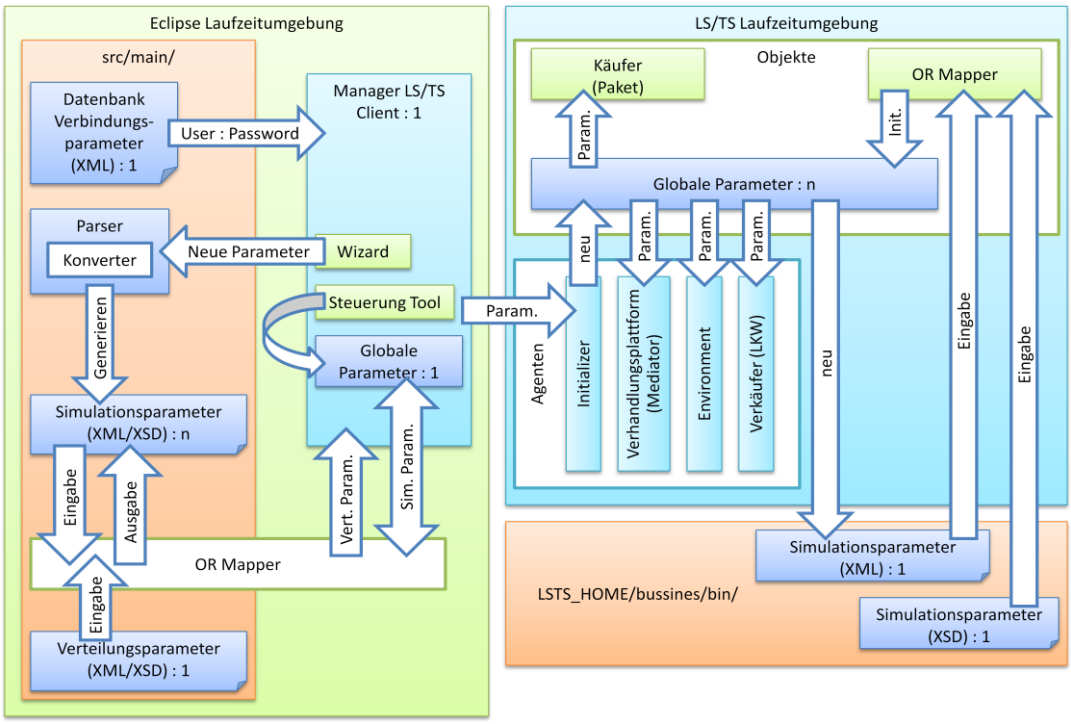


Abbildung 6.4: Parametersicht der Agentenanwendung

Die ersten zwei besitzen eine XML Schema. Zum Lesen der Datei, zum Schreiben in die Dateien oder zum Initialisieren der globalen Parameter wird ein OR-Mapper eingesetzt. Die Datei Verteilungsparameter verfügt über eine Agentenliste und Adressen der entfernten Maschinen. Das Verteilen der Agenten geschieht manuell oder automatisch. Das automatische Verteilen der Agenten wird mit Hilfe von Strategien implementiert.

Die Simulationsparameter werden mit einem Parser aus Jadex Version erzeugt und mit einem Konverter neuen Standarten angepasst. Die Parameter der Simulation sind zusammengefasst und neu gruppiert, um Lesbarkeit der Parameter zu verbessern (s. Kap. 7.6). Nachdem die Parameterdateien erzeugt sind, können sie jederzeit mit Steuerungstool modifiziert werden. Das Verändern der Parameter erfordert Initialisierung der globalen Parameter, die über vordefinierte Schnittstellen auf die XML Parameter zugreift, um Parameter zu lesen und zu schreiben. Das Initialisieren der Agentensimulation mit Parametern wird über Manager Client erledigt, der mit gesamter Agentensimulation verbunden ist. Er sendet die Parameter an jeder Laufzeitumgebung, wo ein Initializer-Agent die Parameter entgegennimmt. Der Initializer-Agent initialisiert eine Instanz der Klasse *GlobalParameter.java*, die im Verzeichnis der LS/TS Laufzeitumgebung eine XML-Datei mit Simulationsparametern erzeugt. Zum Lesen der Parameter wird ein OR-Mapper benutzt. Die XML Simulationsparameter und ihre XML Schema befinden sich im gleichen Verzeichnis LSTS_HOME/business/bin/. Die Simulationsparameter werden nur einmal gelesen, um die globalen Parameter zu initialisieren, die nachher jedem Agenten der Simulation zur Verfügung stehen.

6.1.4 Nachrichtenprotokoll und Ablaufprotokoll der Agentensimulation

Zum Darstellen der Nachrichten und Simulationsaktivitäten werden UML Sequenzdiagrammen eingesetzt. Aufgrund der Anwendungskomplexität und ihrer Umfang werden einige Einschränkungen während der Darstellung getroffen. Diese Einschränkungen verallgemeinern ein Diagramm, verzerren aber auf keinem Fall die Anwendungslogik. Während der Abbildung der Agentenanwendung auf Diagramme werden die Simulationsaktivitäten zusammengefasst und in mehreren einzelnen Diagrammen (Sichten oder UML Regionen) aufgeteilt. Jade Sicht stellt damit eine abstrakte, vertiefte Ebene dar, die eine bessere Übersicht der Anwendungsaktivitäten liefert. Eine Lebenslinie wird durch einen MDAL Message Handler repräsentiert. Die MDAL Fragmente und ihre „steps“ werden nicht berücksichtigt genauso wie Inhalt einer Nachricht. Die Nachrichtenaktivität wird von einer Sicht gezeigt, in der die Agentenanwendung nur auf einem Rechner ausgeführt wird. In der Agentenkommunikation werden die Gespräche entsprechend der Abbildung 6.5 dargestellt und der Austausch von synchronen Nachrichten zwischen Agenten entsprechend Abbildung 6.6. Die einzelnen Sequenzdiagramme mit ihren detaillierten Beschreibungen sind im Anhang A6 LS/TS Agenten Nachrichtenprotokoll dargestellt.

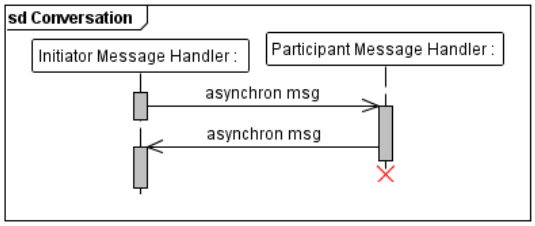


Abbildung 6.5: Dialog

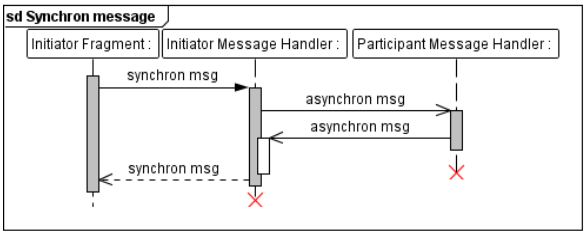


Abbildung 6.6: Synchrone Nachricht

6.1.5. Matching

Matching ist ein Verhandlungsmechanismus, der in der Klasse *MatchingPlan.java* einer Jadex Version des Frameworks implementiert ist. Ein MDAL-Fragment *MatchingFrag.java* stellt einen Zustandsautomat dar. Entsprechend der LS/TS Architektur besteht ein Fragment aus mehreren einzelnen MDAL „steps“, deren Ausführung nach LS/TS Einschränkungen so kurz wie möglich gehalten soll. Sie repräsentieren ein atomares architektonischen Teil, das während seiner Ausführung nicht unterbrochen wird. Bei der Unterbrechung des Fragmentes (Zeit zwischen einzelnen „steps“) sichert LS/TS Plattform nur den Zustand eines MDAL-Fragmentes aber nicht den Zustand der Anwendung. Aus diesem Grund soll der Anwendungszustand manuell im internen Speicher eines Agenten gesichert werden. Das Match-Verfahren (s. Abbildung 6.7) besteht aus drei in einander verschachtelten Schleifen. Die äußere Schleife („step 2-11“) verläuft über die Liste mit Routen eines HUBs. Die innere Schleife („step 4-6,10“) läuft über eine Liste mit Angeboten eines Trucks (jeder Truck generiert ein Angebot für eine Route). Die innere Schleife besitzt intern noch eine weitere Schleife, die über eine Liste mit Geboten eines Paketes läuft. Sie wird vollständig in einem „step“ abgearbeitet und deswegen in der Abbildung nicht dargestellt ist. Die innere Schleife besteht aus drei Teilaktivitäten: Matching, Reservierung einer Lagerfläche in einem HUB und Beladen eines Trucks mit Paketen.

Hier sind die einzelnen Schritte des Mach-Verfahrens:

- „step 1“. In diesem Zustand des Verfahrens werden die Initialisierungsmaßnahmen durchgeführt.

- „step 2“. Startvorgang einer Iteration über eine Sequenz der Routen (äußere Schleife).

- „step 4“. Starten einer Iteration über eine Sequenz der Angeboten/Offers (innere Schleife).

- „step 5“. Sequenzieller Vergleich der Paketgebote mit einem Truck-Angebot für einen Truck ohne Anhänger. Im Fall, wenn Gebot und Angebot nach bestimmten Kriterien der Verhandlung zu einander passen, wird das Gebot in eine separate Liste eingeführt und als Kandidat für eine Tour betrachtet. Zu diesem Zeitpunkt der Verhandlung stellt das Gebot einen nicht abgeschlossenen Vertrag zwischen einem Paket und einem Truck dar. Das Verfahren läuft so lange, bis alle Gebote abgearbeitet sind oder der Truck keine freie Ladekapazität hat.
  Das Ziel jedes Trucks ist eine Paketmenge zu finden, die ihm einen Profit erbringt. Daraus folgend darf ein Truck keine Pakete abtransportieren, deren Gesamtkosten unter einer minimalen Kostengrenze liegen. Wenn diese Kostengrenze nicht erreicht wird, erfolgt ein Übergang zum „step 6“, sonst zum „step 7“.

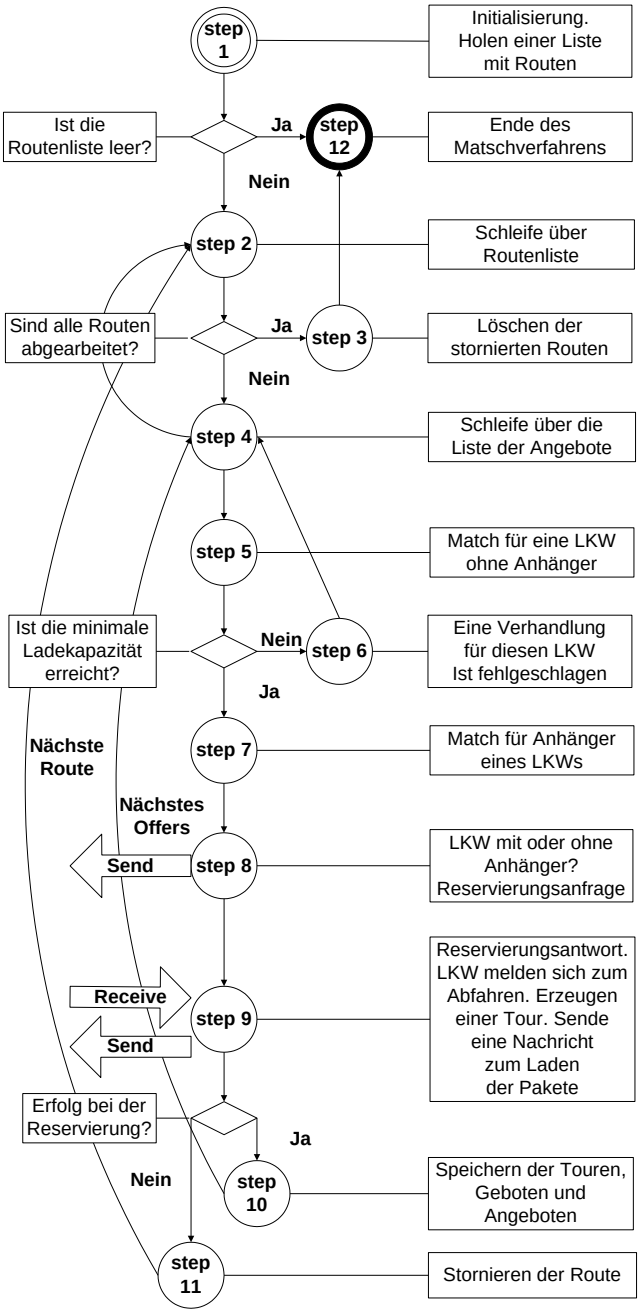


Abbildung 6.7: Matching

- „step 6“. Mit dem Ausführen dieses „steps“ wird das Matching für diesen Truck beendet und die innere Schleife verlassen. Die Tour-Kandidaten werden in diesem Fall aus der Kandidatenliste entfernt (der Vertrag wird aufgelöst).

- „step 7“. Wenn nach dem „step 5“ die Gesamtkosten der Paketmenge doch über die Kostengrenze liegen, wird ein zweites Teil des Match-Verfahrens zum Ausführen gebracht. Es werden Gebote für einen Truck mit Anhänger nach dem gleichen Prinzip wie im „step 5“ gesucht.

- „step 8“. Hier wird geprüft, ob die Menge der Pakete für einen Anhänger ausreichend ist. Zum Schluss wird eine synchrone Reservierung gestartet (s. Abbildung A6 5). Während dieser Reservierung beauftragt der Verhandlungskoordinator (HUB-Agenten) einen Truck, um eine Lagerfläche bei einem Ziel-HUB für seine ausgehandelte Truckladung zu reservieren. Das Reservieren erfolgt für einen Truck mit oder ohne Anhänger.

- „step 9“. Wenn die Reservierung erfolgreich war, wird eine Tour generiert; die Verträge werden abgeschlossen; die Trucks werden abschließend mit Paketen beladen. Die Beladung erfolgt synchron (s. Abbildung A6 6). Nachher werden die Gebote, Angebote und die Touren im „step 10“ persistent geschrieben.

- Im Fall wenn die Reservierung fehlschlägt (ein HUB hat keine freie Lagerfläche), wird die aktuelle Route im „step 11“ für weitere Verhandlungen storniert.

- „step 10“. Speichern der Gebote, Angeboten und Touren in DB.

- „step 11“. Die Route wird storniert, weil der Ziel-HUB keine weitere Paketmenge aufnehmen kann. Die äußere Schleife wird weiter zu nächster Route iteriert.

- „step 3“. Am Ende des Match-Verfahrens werden die stornierten Routen aus der Simulation für die Zeit der aktuellen Spielrunde entfernt.

- „step 12“. Das Ende des Verfahrens erfolgt mit der Aufräumarbeit. Hier werden DS Variablen freigegeben, die zum Speichern des Zustandes zwischen einzelnen „steps“ benutzt wurden.

7 Realisierung

Das Kapitel beschreibt einen Realisierungsprozess der Portierung eines MbFSlP Multiagentensystem auf LS/TS Plattform. Das Portieren erfordert:

- Kenntnisse der intelligenten Agenten und Kenntnisse der Agentenanwendung, um zu verstehen, wer portiert wird;
- Kenntnisse der Jadex Programmiermodell, um die Agentenanwendung verstehen zu können;
- Kenntnisse der LS/TS Plattform um zu wissen, wohin die Agentenanwendung portiert werden soll;
- Kenntnisse des LS/TS MDAL Programmiermodells um zu wissen, womit die portierende Agentenanwendung für die LS/TS Plattform angepasst werden soll.

Die Portierung erfordert einen allgemeinen Plan, nach dem das Portieren durchgeführt werden soll und eine Vereinbarung der allgemeinen Regeln, die während der Portierung verfolgt werden sollen.

7.1 Implementierungsvorgehensweise

Die Portierung wurde in zwei Projektphasen aufgeteilt. Jede diese Phase soll mit der Implementierung eines Prototyps abgeschlossen werden.

In der ersten Phase soll die Agentensimulation mit LS/TS PE (s. Kap.4.2.1) auf eine lokale Maschine portiert werden. Diese Phase schließt die allgemeinen Phasen des Reengineering ein (s. Kap. 2.4.1). Während des Reengineering werden erforderliche Redesign-Maßnahmen bezüglich der LS/TS Restriktionen (s. Kap. 5.2.3) durchgeführt. Die Redesign-Maßnahmen werden auch durch das Portieren eines MASs von Jadex (s. Kap.4.2.1) auf LS/TS MDAL (s. Kap. 4.2.3) Programmiermodel hervorgerufen. Dieser Phase erfordert Architekturkenntnisse der Agentenanwendung (s. Kap. 3.1), Semantik der Agentensimulation (s. Kap. 3.2) und technische Kenntnisse in der Realisierung des MASs (s. Anhang A1 Technische Realisierung der Jadex Agenten), um das Reengineering Prozess erfolgreich auszuführen. Während dieser Phase werden Portierungsregeln festgelegt (s. u. Kap. 7.2) und die benötigten Mechanismen z. B. zum Abbilden der Ziele auf die Techniken der LS/TS MDAL Programmiermodell entwickelt werden. Die monolithischen Teile der Anwendung (zeit- und speicheraufwendige Jadex Pläne) werden in kleine Teile zerlegt.

Der Schwerpunk dieser Phase liegt jedoch in der Rekonstruktion des Nachrichtenprotokolls und Synchronisationsmechanismen von mehreren parallel (s. Kap. 2.3) ausführbaren Plänen des Frameworks. Der erste Prototyp soll zum Schluss dieser Phase auf einer Maschine getestet werden.

In der zweiten Phase soll die Agentensimulation unter Einsatz der LS/TS BE verteilt werden. Diese Phase der Portierung erfordert eine Agentensimulation, die stabil auf einer LS/TS Plattform verläuft. Diese Projektphase hervorruft eine Redesign Maßnahme während dem Verteilen der Agentensimulation und zieht mit sich die Veränderungen der Synchronisationsmechanismen in der Agentenanwendung hinein. Der Schwerpunkt dieser Phase liegt in der Parametrisierung einer verteilter Anwendung, ihrer Steuerung, Synchronisierung und Integrierung der DB in Whitestein

Produkt. Am Ende dieser Phase soll die Agentensimulation verteilt über mehrere Maschinen getestet.

Die vorgenommenen Änderungen sind im Anhang A5 kurz zusammengefasst.

Die folgende Liste gibt eine detaillierte Ablaufübersicht der Portierung:

1. Einsatz der LS/TS PE
   - 1.1. Parameter zusammenfassen.
   - 1.2. XML Binder JAXB 2.0 Einsatz.
   - 1.3. Portierung der Agentensimulation
     - 1.3.1. Agentenimplementierung (Manager, Environment, HUB, Truck)
     - 1.3.2. Beliefsbase Integrierung.
     - 1.3.3. Sequenzielles Reverse Engineering der Simulationsaktivitäten und ihr abbilden auf die LS/TS Mechanismen. Realisierung einer Redesign Maßnahme bezüglich der Whitestein Restriktionen.
   - 1.4. Test einer lokalen Agentensimulation.
   - 1.5. Implementierung eines LS/TS Client
     - 1.5.1. Redesign eines Manageragenten zum LS/TS Client
   - 1.6. Test einer lokalen Agentensimulation mit einem Client.
2. Einsatz der LS/TS BE
   - 2.1. Redesign der Agentenanwendung im Bezug auf ihre Verteilung
     - 2.1.1. Redesign eines Manageragenten
       - 2.1.1.1. Verbindung mit einer verteilten Agentensimulation.
       - 2.1.1.2. Implementierung einer Konfigurationsdatei zum Verteilen der Agentensimulation.
       - 2.1.1.3. Parametrisierung einer verteilten Agentensimulation.
   - 2.2. Test einer verteilten Agentensimulation.
   - 2.3. Redesign der Synchronisationsmechanismen in der Agentensimulation.
   - 2.4. Test einer verteilten Agentensimulation.
   - 2.5. Einsatz der Hibernate-Framewoks zum Integrieren der Anwendung DB in die Agentensimulation.
     - 2.5.1. Konvertierung der Anwendung DB zur Hibernate Bean-Objekte.
     - 2.5.2. Implementierung der Servants- und DAO-Agenten.
   - 2.6. Test einer verteilten Agentensimulation.
   - 2.7. Implementierung eines Wizards in einem Managerclient.
3. Test.

7.2 Das Festlegen der Portierungsregeln

In diesem Abschnitt werden die allgemeinen Konventionen der Portierung beschlossen.

Die J Jadex Pläne sind generell durch LS/TS Fragmenten ersetzbar. Die Jadex Version eines Frameworks hat zwei Arten von Jadex Plänen. Eine Art wird durch das Eintreffen einer Nachricht oder durch ein Jadex Top Ziel zum asynchronen Ausführen gebracht. Diese Pläne werden als externe LS/TS Fragmente portiert. Eine andere Art der Pläne wird mit Unterzielen synchron aufgerufen. Diese Pläne werden als LS/TS Message Handler mit internen Fragmenten portiert. Die Namen der Pläne sollen erhalten bleiben.

Die einzelne Belief aus der ADF eines Agenten (Environment.agent.xml, HUB.agent.xml und Truck.agent.xml) sollen in DS eines LS/TS Agenten integriert werden. Die Namen sollen ebenfalls unverändert bleiben.

Die Veränderungen in der Agentenanwendung, die durch das Portieren entstehen können, dürfen keine Änderungen in der Semantik der Agentensimulation hervorrufen.

Die Jadex Ziele werden als Nachrichten portiert (s. u. Kap. 7.3).

Nach dem, die allgemeine Portierungsregeln festgelegt wurden, sollen auch die Mechanismen entwickelt werden, um die reaktive und proaktive Jadex Ziele (s. Tabelle 4.1) mittels MDAL Techniken nachbilden zu können.

7.3 Entwicklung der geeigneten Techniken zum Portieren der Jadex oder Framework (MbFSlP) spezifischen Mechanismen auf LS/TS Plattform.

Die Jadex-Ziele *Perform Goal* und *Achive Goal* wurden im Framework zum Realisieren eines reaktiven Agentenverhaltens benutzt (s. Kap. 5.1.2). Jedes Ziel wird während der Portierung durch eine Nachricht ersetzt, die immer nur eine Agentenaktivität zum Ausführen bringt und damit gleiches Verhalten, wie Jadex-Ziele im Framework, widerspiegelt (s. Kap. 5.1.1). Die Eigenschaften der Jadex Zielen werden mit internen und externen Nachrichten, Mechanismen zum ihren synchronen und asynchronen Versenden der Nachrichten abgedeckt. Nur *Maintain Goal* soll besonders behandelt werden, weile dieses Ziel für proaktives Verhalten der Agenten zuständig ist (s. Kap. 5.1.2). Ihre Aufgabe ist die Aufrechterhaltung eines bestimmtes Agenten- oder Umweltzustandes. Das Beispiel (s. Listing 7.1) zeiget eine Zieldefinierung aus *Environment.agent.xml* ADF. Das Ziel wird ausgeführt, wenn interner Agentenzustand eine bestimmte Kondition erreicht und wenn alle HUB-Agenten eine Bestätigung gesendet haben.

Listing 7.1: Environment.agent.xml

```
<maintaingoal name="maintaingoal_all_acks_from_HUBs_received">

   <maintaincondition> <![CDATA[
        ! (($beliefbase.state == 2) && ($beliefbase.unreceived_acks_from_HUBs_size==0))]]>
   </maintaincondition>

</maintaingoal>
```

Als Nächstes wird ein Beispiel dargestellt, wie ein oben vorgestelltes *Maintain Goal* aus Listing 7.1 mit MDAL Programmiermodell implementiert werden kann.

Der Codeausschnitt (s. u. Listing 7.2) stellt ein Beispiel einer MDAL-Transitionsregel aus *InformHUBSimulationStartMH.java*, die Zustandsübergangsregeln eines Petrinetzes definiert (s. Kap. 4.2.3). Der Automat repräsentiert eine *Postkiste*, in der die Nachrichten von HUB-Agenten gesammelt werden. Er wird nach dem Start der Agentensimulation gestartet und versendet nachher eine *Multicast* Nachricht (s. Kap. 2.2.1) an alle HUB-Agenten aus dem „POS_INITIAL“ Zustand. Nach dem wechselt er in Zustand „POS_RECEIVE“ und wartet auf das Empfang der Nachrichten. Bei jedem Eintreffen einer Nachricht prüft der Handler eine Liste „DS_UNRECEIVED_ACKS_FROM_HUBS“, ob alle Agenten eine Antwort gesendet haben. Wenn alle Nachrichten empfangen wurden, terminiert der Automat. Das Prüfen der Liste mit

Agenten, deren Antwort erwartet wird, erfolgt mit einer Vorbedingung eines „POS_TERMINAL“ Zustandes.

Der Mechanismus kann sowohl asynchron als auch synchron ausgeführt werden.

Listing 7.2: InformHUBSimulationStartMH.java

```
protected BasicTransitionRule[] defineRules() throws AgentLogicException {

return new BasicTransitionRule[] {

        new TransitionRule(POS_INITIAL, PERF_INFORM_SIMULATION_START,
                                new String[] { FRAG_SEND_MSG_TO_HUBS },
    POS_RECEIVE),
        new TransitionRule(POS_RECEIVE, PERF_INFORM_HUB_ACK,
                                new String[] { FRAG_RESEIVED_ACKS_FROM_HUBS }),
        new TransitionRule(POS_RECEIVE, POS_TERMINAL) {
                                protected boolean condition(IReceivedMessage msg) {
                                        return((ArrayList) getFromDataStorage(
EnvironmentAutonomousAgent.DS_UNRECEIVED_ACKS_FROM_HUBS)).isEmpty();
}}};
}
```

An diesem Beispiel soll auch das realisiertes in LS/TS MDAL das Konzept des Petrinetzes erläutert werden. Die Transitionsregel des Message Handlers *InformHUBSimulationStartMH.java* repräsentiert das Netz mit seinen Transitionen (s. Kap. 2.3.2). Die statischen Modellelemente des Netzes, wie Zustand, werden durch eine Position (POS_) abgebildet und die Prä- und Postkanten sind die Übergänge von einer Position zu Anderen. Die Marke wird mit einer Nachricht dargestellt, die von oben nach unten der Transitionsregel fließt. Das MDAL-Fragment stellt hier eine Transition dar, die eine Bedingung an einer Prästelle der Transition hat (s. Listing 7.2). Das dynamische Verhalten der Marke wird in MDAL mit „Synchronisation“ Standardsituation realisiert (s. **Fehler! Verweisquelle konnte nicht gefunden werden.**). In dieser Situation erfolgt nach einem Empfang einer Nachricht ein Positionswechsel über ein Fragment und das Netz verbleibt in der nächster Position, bis eine neue Nachricht empfangen wird.

7.4 Datenbankintegration

Während der Portierung entstand der Not einer DB Integration zu LS/TS Plattform. Die vorgegebene DB Schnittstelle wurde entwickelt mit der Sicht auf eine lokal ausführbare Agentensimulation und nur einer Instanz dieser Schnittstelle. Der Simulationstest mit einer verteilten Agentensimulation und der ursprünglicher DB war gescheitert.

Zum Lösen dieses Problems standen zwei Möglichkeiten:

1. Suche einer Lösung für den Einsatz der ursprünglichen DB Schnittstelle.

2. Integration der DB in LS/TS mit dem Einsatz der Servanten und DAOs Agenten.

Bei der Entscheidung spielten die Faktoren wie Zeit, Aufwand und Effizienz eine zentrale Rolle. So zum Realisieren der ersten Variante ist ein Reverse Engineering der Schnittstelle und eventuell eine Entwicklung neuer Synchronisationsmechanismen erforderlich. Die Zeit und Aufwand sind dabei unklar und nicht vorhersehbar sind. Die zweite Losung ist nicht nur Zeit- und Aufwand

messbar durch gut beschriebene Vorgehensweise in der LS/TS Dokumentation, sondern auch effizienter. Mit der Integrierung der DB werden einzelne Verbindungen durch Agentenplattform gesteuert und verwaltet. Der Zugriff auf die DB erfolgt nicht mehr separat und unabhängig von der Agentenplattform. Es entsteht kein Konflikt mit LS/TS Restriktionen im Bezug auf den Verbindungsaufbau mit DB (s. Kap. 5.2.3).

LS/TS setzt einen *Hibernate* Framework zum Speichern der Daten in einer DB ein. Dieser Einsatz erfordert das Implementieren für jede einzelne relationale DB Tabelle eines Bean-Objektes und eines DAO-Agenten. Die Bean-Objekte widerspiegeln internes Modell der DB-Tabellen und die DAO-Agenten definieren die HQL-Abfragen.

Zum Implementieren der Bean-Objekte und Definieren der HQL-Abfragen wurde während der Portierung ein Hilfsprogramm *Hibernate Tools*[44] benutzt. Das Tool ermöglicht die Durchführung eines Reverse Engineering der DB Tabellen und Modellieren, Definieren, Testen der DB Abfragen.

Ein weiteres neues Mitglied der Agentensimulation ist ein Servant-Agent *DBServiceServant.java*, der eine Abfrage eines autonomen Agenten (HUB, Truck, Environment) entgegennimmt, bearbeitet sie und leitet an einen oder mehrere DAO-Agenten weiter. Er definiert die gleichen Abfragen, wie die Klasse *DBService.java* (s. Kap. 5.1.4 [Ruwinski und Timotin 2007]). Zum Speichern der Informationen benutzt diese Persistenzschicht bereits existierende Bean-Objekte, die interne Modelle der Tabellen nur zum Teil widerspiegeln. Aus diesem Grund konvertiert der Servant-Agent vor dem DB-Zugriff die empfangenen Bean-Objekte in ein „DAO-Format“ und macht es rückgängig, wenn ein Rückgabewert erforderlich ist.

7.5 Das Testen

Zum Testen der Agenten Anwendung wurden unterschiedliche Methoden, Einsätze benutzt. So z. B. zum Kennenlernen der Funktionalitäten und Möglichkeiten der MDAL Programmiermodell wurde eine Testumgebung in einem separaten Projekt aufgebaut. Zum Testen einer verteilten Agentenanwendung wurde während des Simulationsverlaufs die textuelle Ausgabe mittels „*system out printline“* benutzt, die zum Feststellen der Synchronisationsfehler oder semantischen Abweichungen der Simulation als ein unverzichtbares Werkzeug gewesen. LS/TS stellt auch ein Test-Framework zur Verfügung, das zum Testen einer lokalen Anwendung eingesetzt werden kann (s. Kap. 4.2.9). Während der Portierung charakterisierte sich das Test-Framework durch aufwendige Prozesse, die nur am Anfangsstadium zum Einsatz kamen. Zu diesen Prozessen gehört das Vorbereiten eines Agentenzustandes, das in MDAL-Programmiermodell mit Transitionsregel eines Message Handlers definiert wird. Ein MDAL-Agent besteht im normalen Fall aus mehreren Handler, die durch Mock-Objekte repräsentiert sind und einzeln initialisiert werden sollen. Ihre separate Testbehandlung erfordert bei einem verschachtelten, synchronen Aufruf eines Message Handlers ein manuelles Kopieren der DS eines Agenten von dem aufrufenden Handler zum aufgerufenen Handler, um Konsistenz des Agentenzustandes beibehalten. Das Testverfahren ist ohne Einsatz der Dummy Objekten nicht vorstellbar.

Ständiges Umbauen der Agentenanwendung erfordert eine permanente Anpassung der Testfällen und ihren Testmethoden.

---

44 www.hibernate.org

Ein lokaler Einsatz des Test-Frameworks machte ihn nur in erster Phase der Portierung zum Nutzen.

7.6 XML-Konfigurationsparameter

Um die Agentenanwendung auf LS/TS Plattform zu portieren, wurde nach einem Einstiegspunkt in das System gesucht. Die einzige Möglichkeit, den Revers Engineering Prozess zu starten, war über die Simulationsparameter des Frameworks, die vor dem Anwendungsstart initialisiert werden (s. Kap. 5.1.5.1 [Ruwinski und Timotin 2007]). Das System wird mit diesen Parametern instanziiert und sie beschreiben den Start-, Ausgangszustand der Agentensimulation. Aufgrund der Verteilung der Parameter wurden sie nach Möglichkeit an einer Stelle, nämlich XML Datei zusammengefasst. Um die Lesbarkeit den Simulationsparametern zu verbessern, wurden sie nach ihrer Type gruppiert: Trucks-, Pakets-, HUBs- und generische Parameter (s. Listing 7.3).

Listing 7.3: XML-Struktur der Simulationsparameter

```
<simulation_parameter>
    <truck_parameter > </truck_parameter>
    <packet_parameter> </packet_parameter>
    <HUB_parameter> </HUB_parameter>
    <generic_parameters> </generic_parameters>
</simulation_parameter>
```

Während die ersten drei Parametergruppen für sich selbst sprechen, fördert die Letzte eine Erklärung. In dieser Gruppe sind so genannte „K Parameter“ (s. Kap. 5.4.2) und allgemeine simulationsrelevante Parameter zusammengefasst, die keiner Typisierung vorliegen.

Für das automatische Generieren der Konfigurationsdatei ist ein Parser aus der Klasse *ParametersParser.java* zuständig. Der Parser wurde aus der Jadex Version übernommen und im Bezug auf die Strukturveränderung der Konfigurationsdatei modifiziert. Zum Schreiben und Lesen der Konfigurationsparameter wurde ein XML Binder (JAXB 2.0) eingesetzt. Um den Einsatz des Binders gewährleisten zu können, wurde XML Schema entwickelt, das die Simulationsparameter in zwei Typelementen teilt. Zu einer Type gehören sogenannte „min_max“ Elemente, die Attributen „min“ und „max“ haben (s. Listing 7.4).

Listing 7.4: "min-max" Simulationsparameter

```
<param name="avg_speed" min="0" max="130">120</param>
```

Zu anderer (s. Listing 7.5) Type gehören sogenannte „k_element“ Elemente, die nur einen nummerischen Wert haben:

Listing 7.5: "K-Parameter" Simulationsparameter

```
<k_param name="K_PACKETS">100</k_param>
```

Die Konfigurationsparameter für die Trucks, Paketen und HUBs sind von der „min_max“ Type. Die generische Parameter können sowohl „min_max“ als auch „k_element“ Typen besitzen. Die Abbildung 7.1 s.u. stellt die gesamte Aufbaustruktur der Konfigurationsparameter dar.

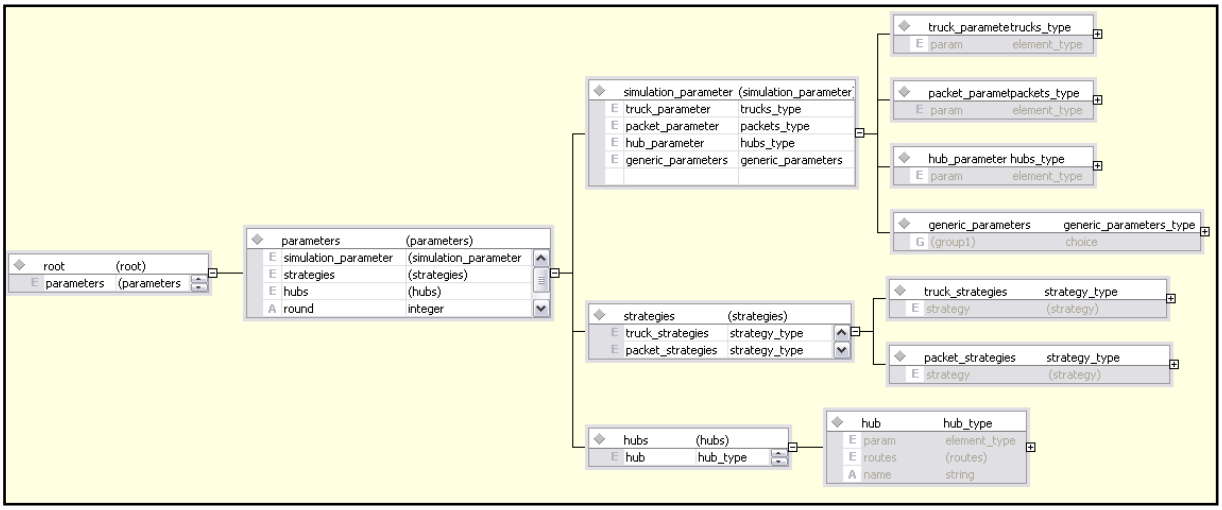


Abbildung 7.1: XML Schema der Konfigurationsparameter

Zum Verteilen der Agentenanwendung wurde eine separate XML Konfigurationsdatei entwickelt Sie wird nur zum Lesen benutzt und besitzt eine Liste der Agenten, die erzeugt werden sollen, eine Liste mit entfernten Rechnern (s. Listing 7.6). Die Konfiguration erlaubt zwei Moden der Verteilung: ein manuelles und ein automatisches Verteilen der Agenten. Die manuelle Verteilung gibt einem Anwender eine Möglichkeit die Agentenmenge „per Hand“ zu adressieren. Die automatische Verteilung setzt voraus, dass eine Verteilungsstrategie bestimmt ist. Das portiertes Framework unterstützt eine „ESY“ Verteilung. Diese Strategie läuft sequenziell über eine Liste der Agenten und eine Liste der entfernten Rechner. In jeder Iteration der Sequenz wird ein Paar aus einem HUB-Agenten und einer Adresse der Agentenplattform (entfernten Rechners) erzeugt.

Listing 7.6: XML Verteilungsparameter

```
<root>
    <distribution type="AUTO">
        <automatic_distribution art_of_distribution="ESY">
            <agent> ENVIRONMENT KOL MAN …</agent>
            <remoteAdress ip="192.168.2.12" port="1099" username="user"
    password="lsts"/>
            <remoteAdress ip="192.168.2.13" port="1099" username="user"
    password="lsts"/>
        </automatic_distribution>
    </distribution>
</root>
```

Die folgende Abbildung 7.2 zegt eine volständige Aufbaustruktur dieser Konfigurationsdatei.

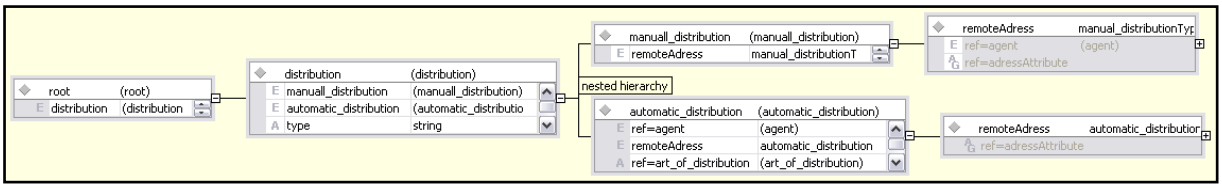


Abbildung 7.2: XML Schema der Verteilungsparameter

Die XML Schema der Verteilungsparameter definiert eine Liste der Agenten, die zum Einsatz in der Simulation kommen dürfen und eine Liste mit Strategien, um fehlerhafte Eingabe zu vermeiden. Die IP Adresse eines entfernten Rechners ist auf einen Numerischen Wert begrenzt.

7.7 Agenten

In diesem Abschnitt der Arbeit wird technische Realisierung der einzelnen Agenten kurz gefasst. Alle implementierte Agenten widerspiegel ihre ursprüngliche Anwendungslogik, die im Anhang A1 Technische Realisierung der Jadex Agentendargestellt ist. Eine detaillierte Beschreibung der portierten Agentenaktivitäten ist im Anhang A7 Technische Realisierung der LS/TS Agentenbeschrieben, in dem jeder MDAL Message Handler eines Agenten geschildert ist.

Manager Client. Die Funktionen eines Manager-Agenten des MbFSlP Frameworks wurden beibehalten (s. Anhang A1 Technische Realisierung der Jadex Agenten) und modifiziert (s. Anhang).

Der Manager besteht aus zwei Klassen: *SimulationsManeger.java* und *ManagerClient.java* (s. Anhang A7 Technische Realisierung der LS/TS Agenten). Er stellt eine Verbindung mit der verteilten Agentensimulation her, initialisiert die Simulationsparameter, startet die Simulation und steuert ihr Verlauf (s. Kap. 6.1.1). Alle Aktivitäten dieses Managers sind sowohl lokal als auch entfernt ausführbar. Die Ausführung des Managers erfolg aus der *Eclipse IDE*. Zum Verbindungsaufbau mit der Anwendung DB wird eine Konfigurationsdatei „settings.xml“ benutzt, so wie die übernommene Persistenzschicht (s. Abbildung 6.3 und Kap. 5.1.4 [Ruwinski und Timotin 2007]). Die zentralen Klassen in dieser Schicht sind *DBService.java* und *ConnectionPool.java.*

- *DBService.java* definiert *Structured Query Language (SQL)* Abfragen für die Anwendung DB.
- *ConnectionPool.java* baut eine DB Verbindung und steuert die Vergabe der geöffneten Verbindungen.

Zum Verbinden mit einzelnen verteilten LS/TS Laufzeitumgebungen der Agentensimulation werden die Konfigurationsdateien „clients.properties“ und „agent_distribution.xml“ benutzt.

Initializer-Agent. Zu der Aufgabe dieses Agenten gehört der Empfang der Simulationsparameter, Bestätigung des Empfangs und Initialisierung der *GlobalParamters.java* (s. Kap. 6.1.4). Die Bestätigung erfolgt, nachdem die Parameter initialisiert, wurden und enthält die IP Adresse des Rechners, auf dem die Agentenplattform ausgeführt wird. Auf Grund geringerer Aufgabenmenge besitzt der Agent keine Data Storage und hat nur eine Aktivität - Initialisierung der Simulationsparameter.

Environment-Agenten. Hauptaufgaben eines Environment-Agenten sind das Koordinieren der verteilten Agentensimulation, Vorbereitung der Simulationsstart, Synchronisierung einzelnen Ausführungsschritten (s. Anhang A1 Technische Realisierung der Jadex Agenten). Es darf nur eine Instanz dieses Agenten in der Simulation existieren.

EnvironmentAutonomousAgent.java definiert eine DS eines Agenten (s. Anhang A7 Technische Realisierung der LS/TS Agenten).

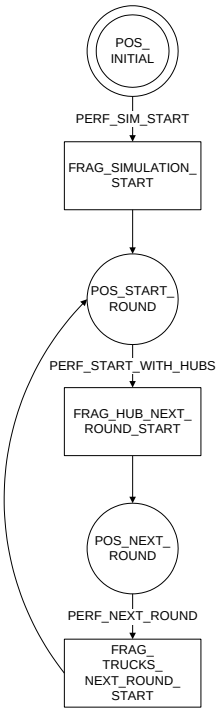


Abbildung 7.3: EnvironmentMH

EnvironmentMH.java stellt eine äußere Iteration (Schleife) der Agentensimulation, die eine Simulationsspielrunde repräsentiert (s. Abbildung 7.3). Diese

Iterationsschleife besteht aus einer Initialisierungsphase und der Schleife selbst. Das Initialisieren des Agenten und der Simulation findet in der Klasse *SimulationStartFrag.java* statt. Das Fragment initialisiert DS eines Environments und informiert alle HUB-Agenten über den Start der Simulation. Die Benachrichtigung der Agenten erfolgt mit einer synchronen Nachricht. Wenn alle Agenten eine Bestätigung zurück gesendet haben, startet das Environment eine neue Spielrunde.

HUB-Agenten Der HUB-Agent (s. Anhang A1 Technische Realisierung der Jadex Agenten) hat in der Agentensimulation drei Rollen (s. Kap. 3.4).

Hub MH.java ist eine zentrale Stelle eines HUBs (s. Abbildung 7.4). Er steuert die Vorbereitungen eines HUB-Agenten zu der Verhandlung, die erst nach seiner Terminierung gestartet wird. Die ersten Agentenaktivitäten dieses Handlers werden in dem Fragment *NextRoundStartedFrag*.java ausgeführt, in dem die Verbindungen zwischen HUBs (Routen) erzeugt werden. Dabei entsteht eine m:n Verbindung.

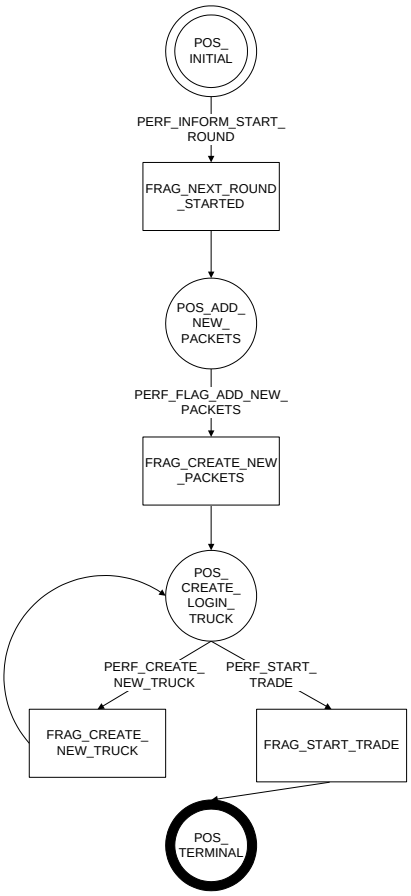


Abbildung 7.4: HubMH

Der nächste Schritt ist das Laden der neuen Pakete aus DB, die ein Environment am Anfang der Spielrunde neu generiert hat. Es findet in *CreateNewPacketsFrag.java* statt (s. Abbildung 7.5).

In der ersten Spielrunde werden die Truck-Agenten erzeugt. In den weiteren Spielrunden wird die Verhandlung gestartet, wenn:

1. Kein Truck hat in den vergangenen Spielrunden die Lagerfläche reserviert.

2. Die Lagerfläche wurde reserviert. Hier ist aus Synchronisationsgründen ein Timer eingebaut. Nachdem alle erwarteten Trucks sich gemeldet haben, wird eine Nachricht „READY“ an

WaitToTradingStartMH.java gesendet. Es wird dem HUB-Agenten mitgeteilt, dass die Trucks zum Verhandeln bereit sind und nur auf den HUB warten. Wenn das der Fall ist, sendet der HUB auch seine Bestätigung. Im Fall, wenn die Truck-Agenten noch nicht bereit zum Verhandeln sind, startet der Timer. Nach dem Ablauf der Zeit erfolgt ein Verhandlungsstart ohne vermisste Trucks.

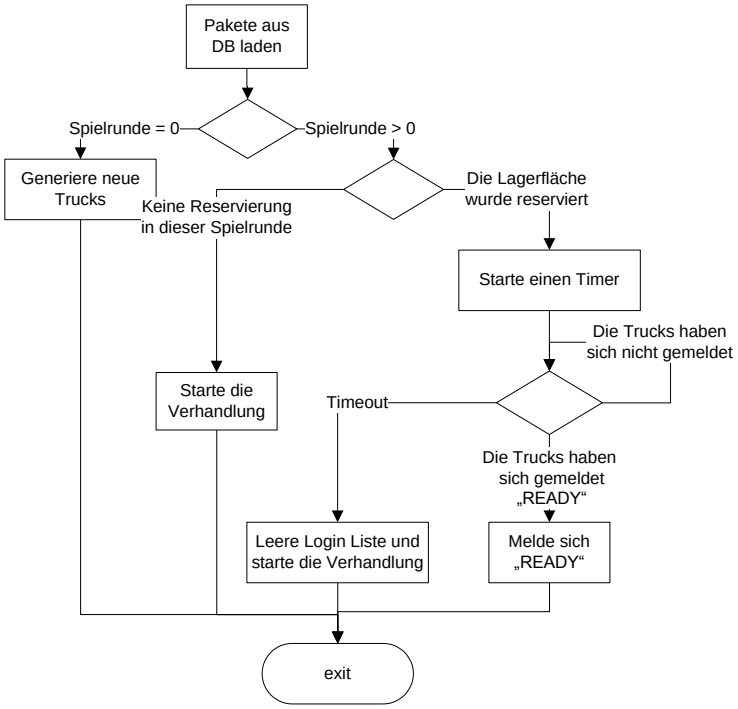


Abbildung 7.5: Verhandlungsstart

CreateNewTruckFrag.java dient zum synchronen Erzeugen der Truck-Agenten in *CreateNewTruckMH.java*. Im *StartTradeFrag.java* werden die Verhandlungsrunden durchgeführt, wo jede einzelne Verhandlung synchron aufgerufen wird. Nach dem Verhandlungsende werden die Trucks, die erfolgreich verhandelt haben, von dem Fuhrpark getrennt und aufgefordert, sich abzumelden. Wenn die Trucks den HUB verlassen haben, wird der Environment-Agenten über die Verhandlungsende informiert.

HUBAutonomousAgent.java definiert eine Agenten DS (s. Anhang A7 Technische Realisierung der LS/TS Agenten).

Truck-Agenten. Der Truck-Agent repräsentiert einen Transportmittel, der die Pakete von einem HUB zu einem Anderen transportiert. Während einer Spielrunde unternimmt der Agent zahlreiche Aktivitäten wie Anmeldung, Beladung, Entladung, Reservierung usw. (s. Anhang A1 Technische Realisierung der Jadex Agenten).

TruckAutonomousAgent.java definiert eine Agenten DS (s. Anhang A7 Technische Realisierung der LS/TS Agenten).

TruckMH.java repräsentiert einen Lebenszyklus eines Truck-Agenten. Der Agent ist aktiv, wenn er bei einem HUB-Agenten angemeldet ist. Der Rest der Zeit ist der Truck unterwegs. Während seiner Fahrt unternimmt der Truck keine Aktivitäten. Aus dem *LoginFrag.java* sendet der Truck seine Paketladung an einen HUB um sich anzumelden und entladen. Im *LoginAgreeFrag.java* wird die Bestätigung der erfolgreichen Anmeldung und Entladung empfangen, nach dem zählt der Truck als entladen.

8 Bewertung und Test

In diesem Kapitel soll auf Basis der durchgeführten Portierungsmaßnahmen eine Bewertung der aktuellen Version eines Multiagentensystem-basierten Frameworks zur Simulation logistischer Prozesse gegeben werden.

Die Portierung dieses Produktes wurde erfolgreich abgeschlossen. Die Ausführung der Agentenanwendung erfolgt auf der LS/TS Agentenplattform mit einer verteilten Agentensimulation. Durch das Portieren wurde die Laufzeit der Simulation deutlich verlängert und die Skalierung der Agentenanwendung bereitgestellt.

Die semantische Ebene der Agentensimulation wurde trotz der technischen Veränderungen nicht geändert. Die einzelnen Agenten haben gleiche Ziele und Mittel zum ihren Verfolgen wie in der früherer Jadex Version.

Zum Testen der Agentenanwendung wurden unterschiedliche Verfahren benutzt. Der einzige und unverzichtbare Test einer verteilten Anwendung ist das Validieren der Simulation während ihres Verlaufs gewesen also wurden während der Portierung zahlreiche Simulationstests mit unterschiedlichen Zielen durchgeführt:

- Stabilität der Simulation.
- Skalierung der Agentenanwendung.
- Lokale und verteilte Ausführbarkeit der Agentensimulation.
- Validierung der einzelnen Agentenaktivitäten während lokalen und verteilten Verlaufs der Simulation.

In Bezug auf die Implementierungsvorgehensweise, die in Kapitel 7.1 beschrieben ist, wurden zwei Tests für beide Implementierungsphasen des Projektes ausgeführt.

Ein Test in der ersten Phase sollte der Test zeigen, ob die portierte Agentenanwendung auf einer Rechenmaschine ausführbar ist. Während dessen sollte die Semantik der Simulation und die Stabilität der Agentenanwendung geprüft werden. Der Simulationstest lief ohne Laufzeitfehler eine Nacht über und wurde nach erreichten 349 Spielrunden manuell abgebrochen (s. u. Abbildung 8.1). Die Paketobjekte und die Truck-Agenten konnten unter Aufsicht eines HUB-Agenten nicht nur verhandeln, sondern sogar einen Gewinn erbringen. Aus der semantischen Sicht der Simulation wurden keine Abweichungen vom Simulationsszenario beobachtet (s. Kap. 3.2). Der Testerfolg war ein Übergangsindikator zu der nächsten Portierungsphase.

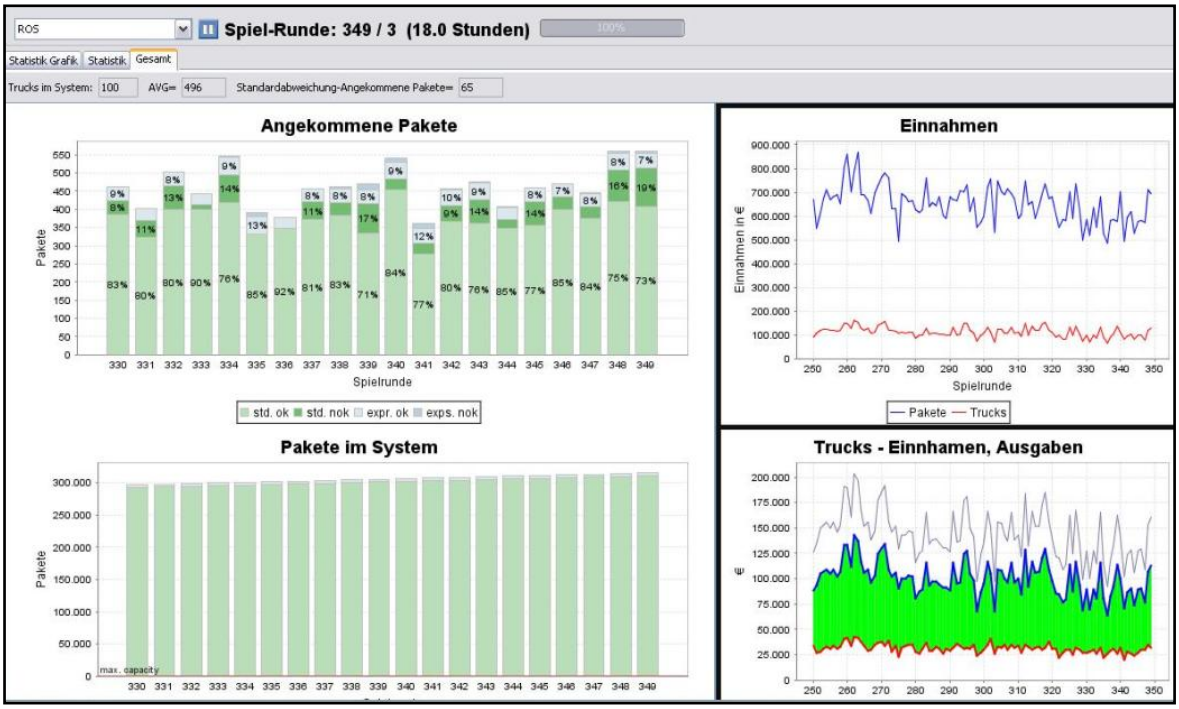


Abbildung 8.1: Lokaler Test

Am Ende der zweiten Phase wurde der Test auf fünf Rechner mit 300 Truck-Agenten und 3 Mio. Paketen verteilt ausgeführt (s. Abbildung 8.2). Die Simulation lief drei Tage lang und erreichte 669 Spielrunden. Sie wurde durch einen manuellen Abbruch beendet. Der Erfolg dieser Tests war ein Indiz für eine gelungene technische Verteilung der Agentensimulation.

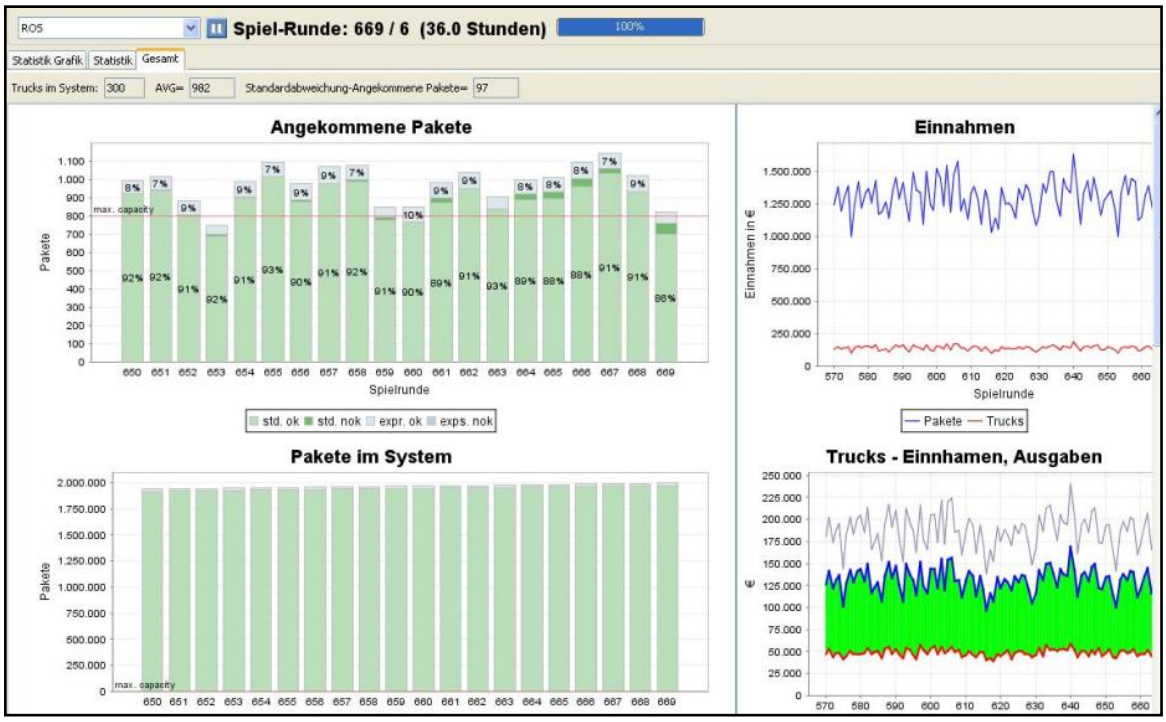


Abbildung 8.2: Verteilter Test

Mit dem weiteren Test wurde das Nutzen der Verteilung gemessen. Das erfolgte durch systematisches Ausführen der Simulation mit gleichen Simulationsparametern und einer dynamische Anzahl der Rechenmaschinen in der verteilten Agentensimulation. In Listing 8.1 sind die Simulationsparameter dargestellt, die in einer Standardkonfigurationsdatei geändert wurden.

Listing 8.1: Simulationsparametern für einen Test mit N Rechnern/Trucks

```
<param name="trucks_number_limit" min="0" max="100">300</param>
<param name="packets_in_system" min="0" max="5000000">3000000</param>
<k_param name="K_TRUCKS">1</k_param>
<k_param name="ALTERNATIVES">1</k_param>
```

Die Abbildung 8.3 stellt dar, dass die Kurve der Simulationsausführungszeit (für 10 Spielrunden) bis zum Einfügen der sechsten Recheneinheit einen fallenden Verlauf aufweißt. Die kurzen Ausführungszeiten der Simulationsprozesse zeigen die Skalierung und Lastverteilung des Systems. Mit der siebten Recheneinheit ist eine Steigerung der Simulationslaufzeit zu betrachten. Es wird auch vermutet, dass mit jeder weiteren eingefügten Recheneinheit die Steigung der Laufzeit wachsen wird. Diese Vermutung deutet auf die Existenz eines Engpasses in der Agentenanwendung, das wahrscheinlich durch den Einsatz einer DB und Synchronisationsmechanismen entstehen kann.

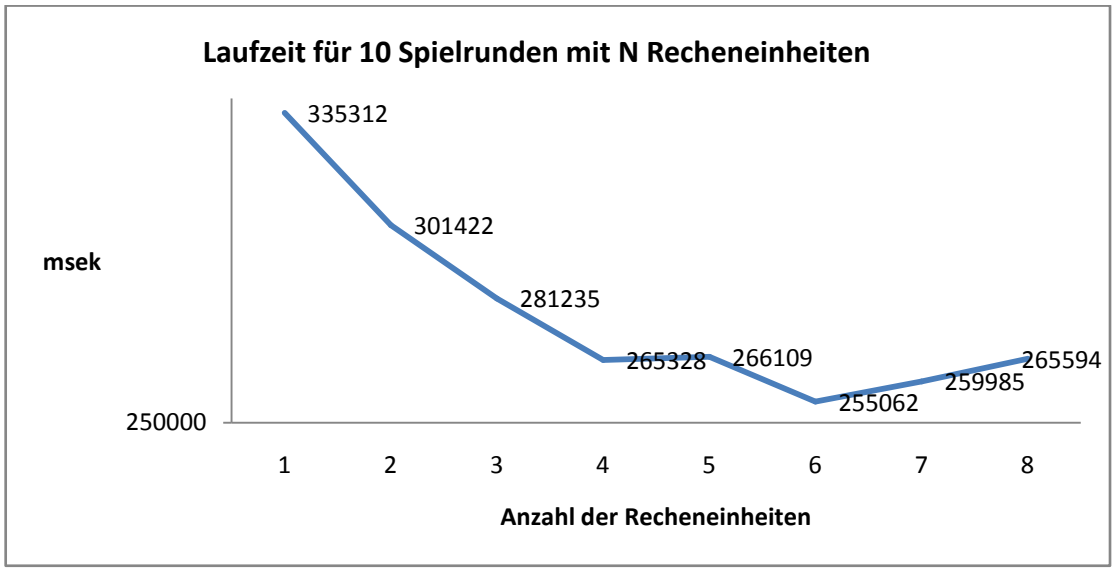


Abbildung 8.3: Test mit N Rechner

Mit dem folgenden Test wurde die Skalierung der Agentenanwendung geprüft. Es wurden gleiche, wie im oben beschriebenem Test, Simulationsparameter genommen und gleiche Vorgehensweise angewendet (s. Listing 8.1). Allerdings wird an der Stelle der Rechenmaschinen eine Menge der Truck-Agenten als dynamische Größe benutzt (s. u. Abbildung 8.4).

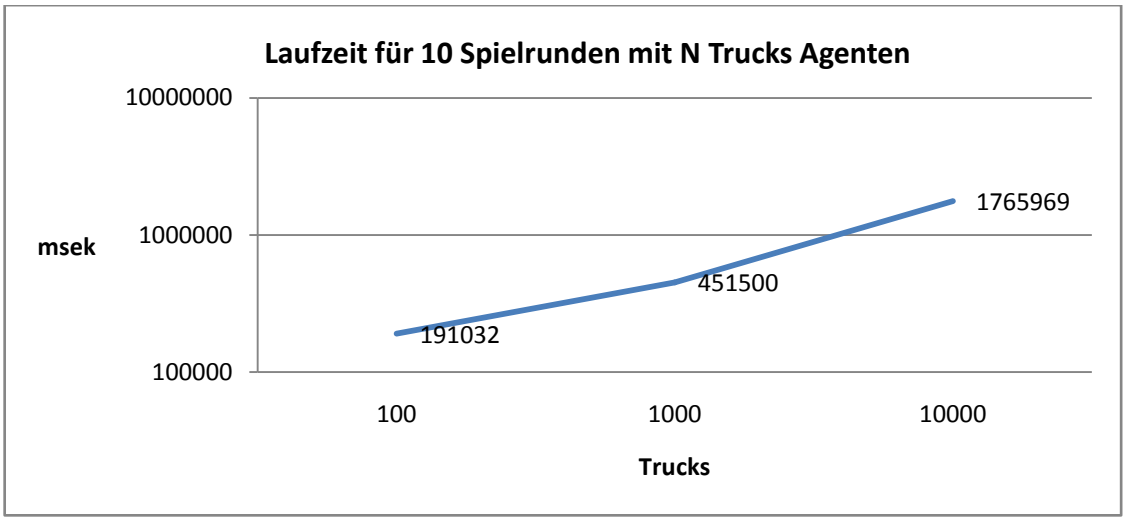


Abbildung 8.4: Test für N Trucks

Während des Testverlaufs mit einer 10-fach wachsenden Agentenmenge zeigte das MAS, abgesehen von der steigenden Ausführungszeit der Simulationsprozesse, ihre stabile und fehlerfreie Ausführung. Die vorgenommen Grenze der Truck-Menge mit 100000 Agenten konnte leider nicht erreicht werden. Ein vermutlicher Grund dafür sind lange Ausführungszeiten der Agentenaktivitäten im Bezug auf die LS/TS Restriktionen (s. Kap. 5.2.3) oder nicht ausreichende Synchronisierungsmaßnahmen.

Mit dem letzen Test wurde ein Versuch unternommen, ein Wachstum der Pakete im System zu beeinflussen. Dabei spielten die Strategien eines Trucks und eines Pakets eine zentrale Rolle. In Listing 8.2 sind die Simulationsparameter und eine Konstellation der Strategien dargestellt, die ein bestimmtes Verhalten der Truck-Agenten und Paketobjekten hervorrufen soll. Das Verhalten der Paketagenten soll zeigen, dass sie erst zu den leistungsvollen HUB-Agenten (Metropol-HUB oder Versand-HUB) abtransportiert werden und erst danach werden sie diese „Sammelpunkte“ in der Richtung Zielort verlassen.

Listing 8.2: Simulationsparametern für einen Test mit strategischen Einsatz

```
<param name="trucks_number_limit" min="0" max="100">-500</param>
…
<param name="packets_in_system" min="0" max="5000000">3000000</param>
…
<k_param name="K_PACKETS">100</k_param>
<k_param name="K_TRUCKS">1</k_param>
<k_param name="ALTERNATIVES">1</k_param>
<k_param name="TRADE_ROUNDS">10</k_param>
…
<strategy frequency="0.0">...LessSupplyMostDemandRouteTruckStrategy</strategy>
<strategy frequency="0.25">...truck.ShortestRouteTruckStrategy</strategy>
<strategy frequency="0.3"> truck.HUBsWithMaxInCapacityTruckStrategy</strategy>
<strategy frequency="0.45"> truck.HUBsWithMaxOutCapacityTruckStrategy</strategy>
<strategy frequency="0.0"> truck.ProfitableRouteTruckStrategy</strategy>
<strategy frequency="0.0"> truck.MostDemandRouteTruckStrategy</strategy>
…
<strategy frequency="0.0">packet.NearestRoutePacketStrategy</strategy>
<strategy frequency="1.0"> packet.DirectRoutePacketStrategy</strategy>
<strategy frequency="0.0"> packet.MaxOutCapacityPacketStrategy</strategy>
<strategy frequency="0.0"> packet.XStrategy</strategy>
```

Die Abbildungen unten stellen den Testverlauf dar. Bei der Betrachtung der Abbildung 8.5 wird ersichtlich, dass die Paketmenge im System während des Simulationsverlaufs zwischen Runden 8 und 14 eine konstante Menge bildet. Im weiteren Verlauf zeigt die Simulation eine kontinuierlich wachsende Menge der Pakete im System, derer Wachstum bis zum Ende des Tests unverändert bleibt (s. Abbildung 8.6).

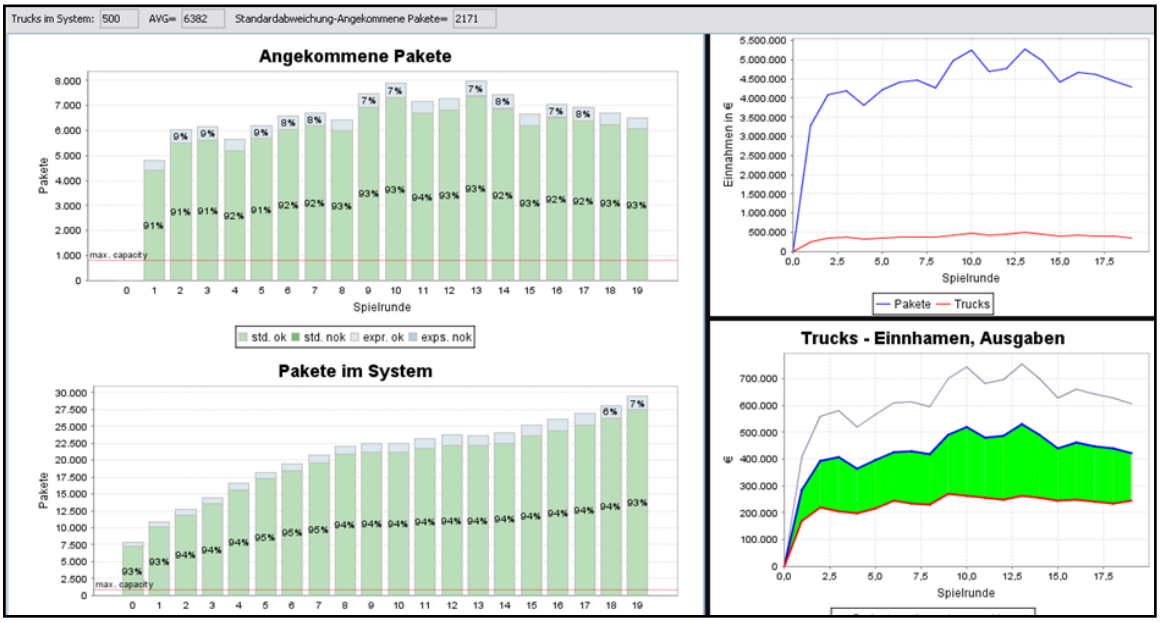


Abbildung 8.5: Paketmenge im System Spielrunde 0 bis 19

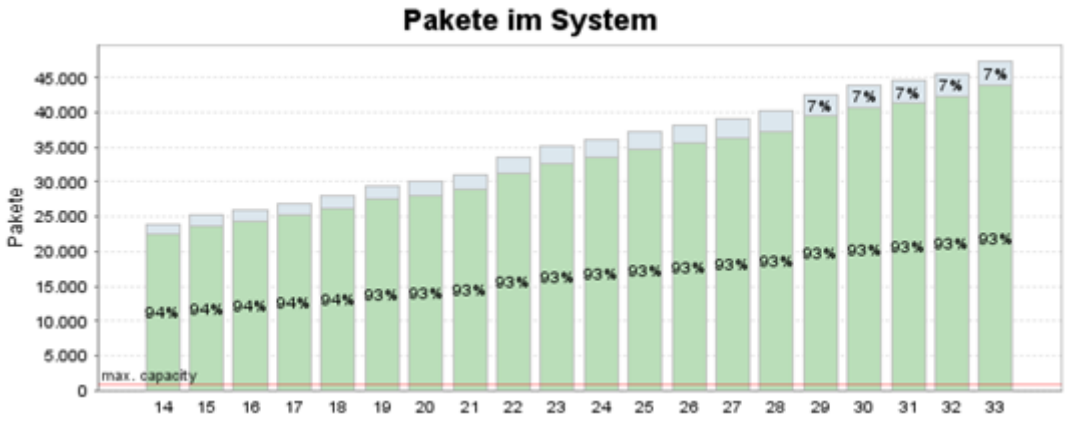


Abbildung 8.6: Paketmenge im System Spielrunde 14 bis 33

Ein Versuch, das Wachstum der Pakete im System zu analysieren, ist gescheitert. Der Grund dafür ist nicht ausreichende Information über die Prozessabläufe der Mikroebene der Agentensimulation. Es kann kein Aussage über das Verhalten eines Paketen, Trucks oder HUBs getroffen werden, obwohl genau in dem Verhalten der Schlüssel fürs Wachstum der Pakete im System versteckt sein kann.

Das in der Arbeit von Herren W. Ruwinski und D. Timotin beschriebene Testergebnis konnte leider mit 3 Mio. Paketen wegen nicht ausreichender Testbeschreibungen nicht wiederholt werden.

9 Zusammenfassung

Vor dem Beginn der Arbeit wurden klare Ziele definiert. Zum Ersten sollte das entwickelte Jadex Multiagentensystem-basiertes Framework zur Simulation logistischer Prozesse auf eine kommerzielle LS/TS Agentenplattform der Firma Whitestein Technology Group AG portiert werden und anschließend die implementierte Paketobjekte in Agenten umgewandelt werden. Zum Zweiten sollen die Konfigurationsparameter der Agentensimulation mit Hinblick auf die Pfadfindung der Simulationsteilnehmer: Truck-Agenten und Paket Objekten, und ständigen Wachstum der Paket Objekten im MAS während des Simulationsverlaufs untersucht werden.

Am Anfang der Arbeit wurde das zu portierende Produkt studiert und die theoretischen Grundlagen der Agentenparadigma beschaffen, um eine Übersicht über die Architektur des Frameworks, über Prozessabläufe in der Agentensimulation und über Mittel, die zum Realisieren der Agentenanwendung benutzt wurden, zu erwerben. Im Weiteren wurden die Grundlagen der Jadex Plattform, ihres Programmiermodells und des Whitestein Produktes mit seinen verschiedenen LS/TS Plattformeditionen, Programmiermodellen studiert. Hier wurden Entscheidungen zur Auswahl einer Edition der Plattform und einer geeigneten Programmiermodell getroffen. Whitestein bietet unterschiedliche Programmiermodelle an, die bei der Agentenrealisierung einen Einsatz der verschiedenen Agentenarchitekturen ermöglicht. Aus diesem Grund lag der Schwerpunkt dieses Abschnittes bei der Entscheidung für ein Modell, mit dem das Agentensystem auf die neue Plattform portiert werden sollte. Um diese Entscheidung zu treffen, wurde die Architektur der portierten Agenten analysiert und die Unterschiede bzw. Gleichheiten der Programmiermodelle festgestellt. Abschließend wurden die Eigenschaften der Laufzeit dieser Anwendung untersucht.

Als nächster Arbeitsschritt wurde das Design der portierenden Agentenanwendung entworfen. In dieser Phase der Arbeit spielten die LS/TS Restriktionen und die Unterschiede zwischen Jadex und LS/TS Programmiermodellen einen großen Einfluss auf das Design der Anwendung. Diese Restriktionen und Unterschiede haben das Reengineering des Systems nötig gemacht.

Im vorletzten Abschnitt, Phase der Realisierung, wurde das Jadex Agentensystem auf LS/TS Plattform portiert. Hier wurden erworbene theoretische und praktische Kenntnisse bei der Portierung umgesetzt. Der Schwerpunkt dieses Abschnittes lag in der Wiederherstellung des Nachrichtenprotokolls des portierten Systems und ihr Reverse Engineering. Das Portieren wurde mit einem besonderen kritischen Blick auf die Anwendung DB durchgeführt, da sie in Whitestein integriert werden sollte, um das Verteilen der Agentensimulation zu gewährleisten. Während des Verteilens der Simulation wurden neue Synchronisationsmechanismen entwickelt. Ein besonderer Abschnitt dieser Arbeit ist das Modellieren eines Petrinetzes für portiertes Agentensystem und die Übertragung der Jadex Zielen und zielorientierten Synchronisationsmechanismen auf das LS/TS MDAL Programmiermodell.

Zum Schluss der Arbeit wurde das Agentensystem mit Hinblick auf die Skalierung und Stabilität getestet.

Die portierte Agentenanwendung ist immer noch nicht bereit zum industriellen Einsatz. Durch die Portierung wurden einige Vorteile gewonnen, z. B. größere Agentenmengen im Agentensystem oder verteilter Ablauf der Agentenaktivitäten in der Simulation, deren Lastverteilung durch DB Optimieren verbessert werden kann. Genau wie der Vorgänger (Jadex Version) enthalt die Agentenanwendung gleiche Merkmale in Bezug auf das Modellieren der logistischen Prozessen

einer realen Welt. Dafür sind die investierte Zeit und der für die Portierung entstandener Aufwand in diesem Projekt aus der wirtschaftlichen Sicht nicht unangemessen.

Bezüglich der Portierbarkeit von anderen Multiagentensystemen von der Open Source Plattform Jadex auf das kommerzielle Plattform Whitestein LS/TS können folgende Folgerungen gezogen werden:

Agentenarchitektur. Vor der Portierung soll eine ausführliche Analyse des zu portierenden Agentensystems durchgeführt werden. Es ist wichtig zu verstehen, welche Agentenarchitektur in MAS angewendet wird, weil damit der Arbeitaufwand der bevorstehenden Portierung eingeschätzt werden kann. Das Portieren von proaktiven Agenten erfordert mehr Entwicklungsaufwand als von reaktiven Agenten.

Ziele. Eine Analyse der Zielorientierung der Agenten soll als ein gesonderter Teil des Projektes erfolgen, denn genau in den Zielen liegt der Portierungsschwerpunkt des MAS.

Die Agentenziele und ihre Zieltypen sollen festgestellt werden. Es soll ermittelt werden, in welchem Kontext die Ziele benutzt werden, ob proaktive Ziele für die Realisierung des reaktiven Verhaltens oder reaktive Ziele für Realisierung des proaktiven Verhaltens eines Agenten eingesetzt werden. Es soll auch die Deliberation der Agenten untersucht werden, ob die Agenten Zielalternativen bzw. Planalternativen haben. Die Abwesenheit des *practical reasoning* verweist auf reaktives Verhalten der Agenten.

Mittels Jadex-Unterzielen und ihrem synchronen Aufruf werden Synchronisationsmechanismen, die bei der Portierung auf Whitestein Plattform berücksichtigt müssen, konstruiert.

Programmiermodell. Das MDAL Programmiermodell wird in den meisten Erweiterungen und Optimierungen des Whitestein Produktes eingesetzt. Bei dem Einsatz dieses Modells soll die Ziel-Plan-Hierarchie mittels Petrinetz-Konzept realisiert werden. Die Auswahl zwischen Jadex Ziel- und Planalternativen (*practical reasoning*) soll in MDAL Transitionsregeln ebenfalls mittels Petrinetz Konzept implementiert werden.

Ein Jadex Plan kann als ein MDAL Fragment portiert werden. Es gibt aber einen Unterschied in ihrer Ausführung. Der Jadex Plan ist ein Java Thread während das MDAL Fragment eine Sammlung von atomaren Agentenaktivitäten (steps) ist, die sequenziell ausgeführt werden.

Nachrichten. Jadex und Whitestein Programmiermodelle unterscheiden sich bei der Realisierung der synchronen Nachrichten, die zur Synchronisierung und Koordinierung der Agentenaktivitäten benutzt werden. Das erfordert besondere Modellierungsmaßnahmen, denn diese Art von Nachrichten wird in LS/TS MDAL nicht mit OO Techniken sondern mit Petrinetz Methoden realisiert.

Synchronisierung. Die synchronen Nachrichten und synchron aufgerufene Unterziele erfordern vom Entwickler ebenso eine besondere Behandlung.
Aktivität eines Jadex Agenten beginnt an der Stelle, wo sie unterbrochen wurde (sendMessageAndWait() bzw. dispatchSubgoalAndWait()) und Aktivitäten des MDAL Agenten finden ihren Wiedereinstieg in einem Zustand (ein atomares „step“ eines Fragmentes) entsprechend dem Petrinetz Konzept. Beim Aufruf eines synchronen Ziels oder beim Versenden einer synchronen Nachricht aus tief geschachtelten Schleifen wird es kritisch, weil jede Schleife in mehrere Arbeitsschritte (steps) aufgeteilt werden muss, um Aktivitäten der synchronen Nachrichten oder Unterzielen ordnungsgemäß durchzuführen.

Verteilung. Im Hinblick auf die Verteilung eines MAS soll zwischen Clustering und Föderation entschieden werden. So kann sich ein Entwickler bei dem Einsatz eines Cluster auf die automatische Lastverteilung der Agentenanwendung verlassen. Die Föderation erfordert nicht nur eine manuelle Verteilung, sondern auch die Entwicklung der Synchronisationsmechanismen mit expliziter Berücksichtigung einer Verteilung mit parallel ausgeführten Agenten des MAS.

Literaturverzeichnis

[AgentLink 2000] AgentLink. News Nr.5-2000, S.6-9. www.agentlink.org

[AgentLink 2002] AgentLink. News Nr.9-2002, S.7-10. www.agentlink.org

[AinJ2EE 2002] Whitestein-Technolgies_Whitepaper_Agents-in-a-J2EE-World (1). *Agents in a J2EE World*, Brantschen, Stefan; Haas; Thomas. v1.4 2002-03-13

[Braubach 2007] Braubach, Lars: *Architekturen und Methoden zur Entwicklung verteilter, agentenorientierter Softwaresysteme.* 2007, ISBN 978-3-00-023107-0

[Brenner et al 1997] Brenner, Walter; Zarnekow, Rüdiger; Witting, Hartmut: *Intelligente Softwareagenten: Grundlagen und Anwendungen.* Springer Verlag ,Berlin 1998, ISBN 3-540-63431-2

[Brössler und Siedersleben 2000] Brössler, Peter; Siedersleben, Johannes (Hrsg.): *„Softwaretechnik“*, München Wien 2000. ISBN 3-446-21168-3

[Buchholz et al 1998] Buchholz,J; Clausen,U; Vastag,A.: *Handbuch der Verkehrslogistik. Logistik in Industrie, Handel und Dienstleistungen.* Springer Verlag 1998, ISBN 3-540-64517-9

[CAL 2006] lsts.doc.cal, *Core Agent Layer,* v 1.3 2006-02-15

[Depke 2004] Depke, Ralph: Dissertation„*Visuelle Modellierung agentenbasierter Systeme“,* Universität Paderborn 2004

[DevGuide 2006] lsts.doc.dev-guide, *Developer Guide,* v1.4, 2006-02-15

[Dumke 2000] Dumke, Reiner: *Software Engineering: Eine Einführung für Informatiker und Ingenieure: Systeme, Erfahrungen, Methoden, Tools.* Friedr. Vieweg & Sohn Verlagsgesellschaft mbH. ISBN: 3-528-153555-5

[Eymann 2000] Eymann, Torsten: *Dissertation „AVALANCHE - Ein agentenbasierter dezentraler Koordinationsmechanismus für elektronische Märkte.* 2000

[Eymann 2003] Eymann, Torsten: *Digitale Geschäftsagenten: Softwareagenten im Einsatz.* Springer Verlag, Berlin 2003. ISBN 3-540-44019-4

[Gildhoff 2007] Gildhoff, Hinnerk: Diplomarbeit „*Eine Simulationsunterstützung für Agentenplattformen“,* Universität Hamburg 2007. www.informatik.uni-hamburg.de/publications/index.php/stud/2007

[Görz et al 2003] Görz, Günter; Rollinger, Claus-Rainer; Schneeberger, Josef: *Handbuch der künstlichen Intelligenz.* 4. Auflage. Oldenburg 2003, ISBN 3-486-27212-8

[Hill et al 1994] Hill, W.;Fehlbaum, R.; Ulrich, P.: *Organisationslehre 1.*Bern/Stuttgart/Wien 1984.

[Jadex 2007] Pokahr, Alexander; Braubach, Lars: *Jadex User Guide.* 2007, userguide.pdf

[Linstaedt 2006] Linstaedt, Sven: Diplomarbeit „*Integration von Agentenplattformen in Middleware – am Beispiel von Jadex und Java EE“,* Universität Hamburg 2006. www.informatik.uni-hamburg.de/getDoc.php/thesis/425/linstaedt_da.pdf

[LISA 2006] lsts_doc_lisa *Library for Interactions and Structured Actions Engine -LISA Concept* v.1.0 2006-02-27

[LS/TS 2006] lsts.doc.tester *LS/TS Tester,* v 1.3 2006-02-15

[Lunze 2006] Prof. Dr.-Ing. Lunze, Jan: *Ereignisdiskrete Systeme: Modellierung und Analyse dynamischer Systeme mit Automaten, Petrinetzen und Markovketten.* Gebundene Ausgabe. Oldenbourg 2006. ISBN-13: 978-3486580716

[MARGE 2006] lsts.doc.merge, *Multi-Agent Reasoning based on Goal-oriented Execution-MARGE,* v1.0, 2006-02-15

[MDAL 2006] lsts.doc.mdal, *Message Dispatching Agent Logic- MDAL* , v1.3, 2006-02-15

[Muscholl 2001] Muscholl, Klaus Matthias: Doktorarbeit „*Interaktion und Koordination in Multiagentensystemen",* Universität Stuttgart 2001.

[Oechsle 2007] Oechsle, Rainer: *Parallele und verteilte Anwendungen in Java*. Hanser Fachbuch, 2007. ISBN 978-3446407145

[Pokahr et al 2005a] Pokahr, Alexander; Braubach, Lars; Lamersdorf, Winfried: *Dezentrale Steuerung verteilter Anwendungen mit rationalen Agenten.* Universität Hamburg 2005.

[Pokahr et al 2005b] Pokahr, Alexander; Braubach, Lars; Lamersdorf, Winfried: *Multi-Agent Programming. Languages, Platforms and Applications*. Springer Verlag, 2005. ISBN 978-0-387-26350-2

[Pokahr 2007] Pokahr, Alexander: *Programmiersprachen und Werkzeuge zur Entwicklung verteilter agentenorientierter Softwaresysteme.* 2007, ISBN 978-3-00-023105-6

[ProdOver 2006] lsts_doc_overveiw *Product Overview,* v1.4 2006-02-15

[Raffel 2005] Raffel, Wolf-Ulrich: *Dissertation „Agentenbasierte Simulation als Verfeinerung der Diskreten-Ereignis-Simulation unter besonderer Berücksichtigung des Beispiels Fahrerloser Transportsysteme"*, 03.03.2005.

[Rosenstengel und Winand 1991] Rosenstengel, Bernd; Winand, Udo: *Petrinetze: eine anwendungsorientierte Einführung*. 4. Auflage, Vieweg Verlag, Braunschweig 1991. ISBN: 3-528-33-582-3.

[Run Time 2006] lsts_doc_runtime, *Run-Time Environment,* v1.3 2006-02-15.

[Russell und Norvig 1995] Russell, Stuart; Norvig, Peter.: *Artificial Intelligence: A Modern Approach.*2.Edition. Prentice-Hall, 1995.

[Ruwinski und Timotin 2007] Ruwinski, Wladimir; Timotin, Dennis: Bachelorarbeit *„Entwicklung eines MAS-basierten Frameworks zur Simulation logischer Prozesse"*, 22.06.2007

[SemCom 2006] lsts_doc_semcom, *Semantic Communication,* v1.1 2006-02-15

[Sudeikat 2004] Sudeikat, Jan: Diplomarbeit *„Betrachtung und Auswahl der Methoden zur Entwicklung von Agentensystemen"*, HAW Hamburg 2004. www.infotmatik.uni-hamburg.de/publications/viewThesis.php/189/sudeikat_da.pdf

[Tanenbaum und Stehen 2007] Tanenbaum, Andrew S., Stehen, Maarten van: *Verteilte Systeme: Prinzipien und Paradigmen.* Gebundene Ausgabe. Verlag Pearson Studium, 2. Auflage, 2007. ISBN 978-3-8273-7293-2

[traveling example 2006] lsts_doc_traveling_example, *Goal Oriented Agents Example application Public transport system*, 2006-02-15

[Unland et al 2005] Unland, Reiner; Klusch, Matthias; Calisti, Monigue: *Software Agent-Based Applications, Platforms and Development Kits.* Birkhäuser 2005, ISBN 3-7643-7347-4

[Wahrig 1999] Hrgb. Dr. Wahrig-Burfeind, Renate: *Wahrig Fremdwörterlexikon*. Bertelsmann Lexikon Verlag GmbH Gütersloh, München 1999, Nr.053884

[Weiß 2001] Weiß, Gerhard: *Agentenorientiertes Software Ingineering.* Zeizschrift „Informatik Spektrum" Nr.2-04.2001

[Weiß und Jakob 2005] Weiß, Gerhard; Jakob, Ralf*: Agentenorientierte Softwareentwicklung*. Springer, 2005, ISBN 3-540-00062-3

[Wooldridge 1999] Wooldridge, M.J, Gerhard Weiss (Editor): *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press 1999, ISBN 978-0262232036

[Wooldridge 2001] Wooldridge, M.J.: *An introduction to multiagent systems*. John Wiley & Sons Ltd. England 2001. ISBN 0 471 49691 X

Anhang

A1 Technische Realisierung der Jadex Agenten

Die Simulationsparameter definieren ein Ausgangmodell der Agentensimulation, derer Parametrisieren in XML Deskriptor, DB und einer Klasse *GlobalParameter.java* erfolgt.

Die XML Konfigurationsdatei enthält die meisten Parameter und wird als eine zentrale Parametrisierungsstelle der Simulation betrachtet. In der Datei werden relevante Parameter für das ganze System, für einzelne Simulationsteilnehmer z. B. Truck und für Komponenten des Simulationsmodells z. B. Route definiert. In der DB sind die Initialisierungsparameter eines HUBs (geografische Position, Anzahl der Eingängen und Ausgängen) untergebracht. Die Klasse *GlobalParameters.java* widerspiegelt die Parameter der XML-Konfigurationsdatei und Parameter aus DB. In der Instanz dieser Klasse wird die verschachtelte Struktur des XML Deskriptors durch einen objektorientierten Einsatz wiedergegeben. Die Simulationssteuerung verfügt über ein visuelles Tool zum Einstellen der Parameter, die im XML Deskriptor deklariert sind.

Manageragent. Der Manager startet die Simulation und erzeugt nachher einen Environment und HUB-Agenten. Zum Kreieren den HUB greift der Manager auf DB, um die Parameter zum Initialisieren dieser Agenten zu holen. Der Manager verfügt über keinen internen Speicher.

Environment-Agenten. Ein Environment übernimmt die Aufgabe einer zentralen Stelle für globale Koordinierung der Simulation (s. Kap. 4.2.1 [Ruwinski und Timotin 2007]). Der Agent gibt Takt für alle Beteiligten der Simulation außer Manager. Er kontrolliert das Starten einer Spielrunde und ihr Beenden. Die Simulationskoordinierung beansprucht die Kenntnisse über globalen Zustand des Simulationsmodells, deswegen übernimmt der Agent auch das Speichern des Modellzustandes. In seiner Zuständigkeit fällt auch das Generieren der erforderlichen Paketmengen ein. Der Environment-Agenten greift auf DB aus der Klasse *GenerateNewPacketsPlan.java* um neu generierte Pakete in DB persistent zu schreiben, und aus der Klasse *SimulationStartPlan.java* um die Konfigurationsparameter für alle HUB-Agenten, die in dem Simulationsmodell aktiv sind, zu laden.

Zum Starten einer Spielrunde benötigt das Environment eine Bestätigung von allen Trucks. Dasselbe Verfahren gilt auch für das Ende der Spielrunde nur mit einem Unterschied, dass die Bestätigungen von HUB erwartet werden. Die Koordination hat eine einfache Aufbaustruktur. Vor dem Start oder Ende einer Spielrunde werden alle Agentennamen, von denen eine Bestätigung erwartet ist, in einer Liste *unreceived_acks_from_HUBs* oder *unreceived_acks_from_trucks* eingetragen und nach dem Ankommen der Bestätigung wird der zugehörige Agentennamen aus der Liste entfernt. Eine leere Liste repräsentiert Start oder Ende der Spielrunde.

HUB-Agenten. Nachdem das ganze System initialisiertest ist und der Environment-Agenten eine Benachrichtigung über das Starten einer neuer Spielrunde versendet hat, startet der HUB-Agenten eine Vorbereitung zu der Verhandlungen. Während dieser Vorbereitung lädt der HUB-Agenten neu generierte Pakete, sammelt alle Gebote von den Paketen und Angebote von den Trucks. Erst nach dem alle Gebote und Angebote generiert und erhalten wurden, startet der HUB-Agent eine Verhandlung. Zur Zeit der Verhandlung koordiniert der Agent das Starten und Beenden einer Verhandlungsrunde.

Die erteilten Agentenaufgaben erfordern ein intensives Nutzen der DB:

LoginTruckPlan.java Nach dem Ausladen eines Trucks werden alle Pakete die ihre Transportreise im aktuellen HUB beendet haben (gelieferte Pakete), in DB persistent geschrieben.

CreateNewPacketsPlan.java In dieser Klasse werden generierte Pakete von einem Environment-Agenten für aktuellen HUB und aktuelle Spielrunde aus der DB geholt und zum Lagerbestand zugefügt.

InformTruckTradeRoundPlan.java In dieser Klasse werden die Angebote persistent geschrieben.

MakePacketsBidsPlan.java Klasse dient zum persistenten schreiben der Gebote.

MatchingPlan.java In dieser Klasse werden die Touren, Gebote und Angebote persistent geschrieben.

Als Verhandlungsplattform koordiniert der Agent eine Verhandlung zwischen Paketen und Tracks. In der Klasse *StartTradePlan.java* wurde gesamter Verhandlungsablauf als eine Klassensequenz definiert. Jede Iteration dieser Sequenz ist eine Verhandlungsrunde. Folgend ist die Übersicht der einzelnen Klassen:

- *StartTradePlan.java* Starten einer neuen Verhandlungsrunde.
- *InformTruckTradeRoundPlan.java* Sammeln von Geboten der Trucks.
- *MakePacketsBidsPlan.java* Sammeln von Angeboten der Paketen.
- *MatchingPlan.java* Ausführen eines Auktionsalgorithmus.

Eine exakte Beschreibung des Koordinierungsmechanismus ist im Kapitel 4.2.2 Abb. 4.13 der Arbeit [Ruwinski und Timotin 2007]) gegeben.

Truck-Agenten. Dieser Agent meldet sich unverzüglich nach dem Anfang einer neuen Spielrunde an einem Umschlagspunkt. Er lädt seine Lieferung aus und wartet auf die Verhandlungen. Nach einer erfolgreichen Verhandlung wird eine Tour generiert. Dies gilt als ein Vertrag zwischen einem Truck und einem Paket. Der Vertrag verpflichtet diesen Truck einen Paket abtransportieren. Nach dem die Verhandlungen beendet wurden, meldet sich der beladene Truck von einer Verhandlungsplattform ab. Ab diesen Moment beginnt der Transportweg eines Truck-Agenten. Der Transportweg wird in Spielrunden gemessen, der abhängig von einer geografischen Entfernung zwischen HUB-Agenten über mehrere Spielrunden dauern kann. Auf die DB greift dieser Agent nur einmal, um seine Instanz persistent zu schreiben. Die Aktivität wird aus der Klasse *TruckCreatedPlan.java* ausgeführt.

Paket. Neben der agentenorientierten Welt existiert im Simulationsmodell auch eine objektorientierte Welt. In dieser virtuellen Umgebung werden Instanzen, die kein proaktives Verhalten verfügen müssen oder können und diejenige Instanzen, die nicht als Agenten implementiert werden könnten, hereingebracht.

Die Objekte *Bid.java*, *HUB.java*, *Offer.java*, *Paket.java*, *Tour.java*, *Truck.java* sind als Java Bean implementiert und kapseln die Informationen, die DB für eine spätere Analyse persistent geschrieben werden. Die Objekte *Location.java*, *Route.java*, *RouteExtended.java*, *TruckExtended.java* sind einfache Hilfsobjekten, die von allen Simulationsteilnehmern benutzt werden.

A2 Deployment und Konfiguration der LS/TS Plattform

Anwendung Deployment. Um eine Agentenanwendung zu starten, braucht die Laufzeitumgebung Startinformation über Agenten: Agenten Name, Typen (persistent oder nicht persistent), DB Konfigurationsdaten, Servants und DAOs Information. Alle diese Meta-Daten Informationen sind in verschiedenen XML Dateien gespeichert. LS/TS benutzt zwei Arten der Deskriptoren für Meta-Daten:

- Agent Mapping. Der Deskriptor beschreibt, welche Klasse implementiert einen autonomen Agenten und sein Type. Für Servants und DAOs werden nur Implementierungsklassen und ihre registrierten Namen bei *Lookup Dienst* beschrieben.

- Persistente Provider Konfiguration. Diese Datei beinhaltet die datenbankrelevanten Konfigurationsdaten. Zum Inhalt gehören Edition Type der Laufzeitumgebung, DB Treiber, Anwender Name, Passwort usw.

Das Starten der Agentenanwendung erfolgt mithilfe eines LS/TS Client (s. u.) oder mit einer Konfigurationsdatei *applicationStartup.xml*, die Name und Type eines Agenten oder Sicherheitsinformationen, z. B. Zugehörigkeit zu einer Gruppe beschreibt. Additional ist es möglich die Initialisierungsparameter an einen Agent zu senden.

DB Unterstützung. Die DAOs sind in einem **.hbm.xml* Deskriptor beschrieben, der DAO Java-Klasse mit einer relationale Tabelle eines DBs verknüpft. Die *Hibernate* Konfigurationsparameter befinden sich in *hib-persistence.xml* Datei. Sie beinhaltet allgemeine Information über die Position der DB Treiber, Anwender Name, Passwort, Große des Connection Pool und Aufzählung der **.hbm.xml* Dateien mit definierten DAOs.

Konfiguration der Laufzeitumgebung. Einem Anwender steht eine Möglichkeit die Laufzeitumgebung zu konfigurieren/modifizieren. LS/TS benutzt *Open Source Framework Spring*[45] um die Konfiguration zu lesen und schließlich eine Instanz der RE erzeugen.

lsts- personal.xml, lsts- business xml, lsts- enterprise.xml Konfigurationsdateien beinhalten Plattformkonfigurationen und haben unterschiedliche, abhängig von der RE Edition Positionierungsorte. *lsts.config* enthält Konfigurationsparameter der Dienstleistungen einer Laufzeitumgebung. Hier werden die Anwendungsspezifischen Dienste verwaltet. *lsts-login.config* enthält Name und Password eines Administrators und eines Anwenders. Die Datei ist nur für PE notwendig. *log4j.properties* Konfigurationsdatei für *Log4j Framework*[46].

Eine Agentenanwendung wird direkt aus dem *Eclipse IDE* oder durch Ausführen der Dateien *run.bat* (für Windows) oder *run.sh* (für Linux) gestartet, welche eine Möglichkeit die RMI Port und Java-Heap Große zu konfigurieren bieten.

Personal Edition. Die *lsts.config* bietet nur eine Konfigurationsmöglichkeit, nämlich Konfigurierung eines persistenten Frameworks für die DB. Die Konfigurationsdatei *persistence.xml* wird manuell im selben Verzeichnis angelegt, falls eine DB benutzt werden soll.

45 http://www.springsource.org
46 http://logging.apache.org/log4j

Business Edition. Die Ressourcen Verwaltung in der Konfigurationsdatei *lsts.config* für nicht persistente Agenten ist standardmäßig deaktiviert. Sie besteht aus zwei Teilen:

- Agent Storages Ressource Management. Damit wird ein nummerischen Grenzwert der Agenten im System zu einem Zeitpunkt kontrolliert. Wenn die Grenze überschritten ist, werden inaktive Agenten in DB temporär, persistent geschrieben.

- Message Queue Ressource Management. Damit werden drei nummerische Parameter *highWatermark, lowWatermark* und *blockSize* kontrolliert. Wenn die Große der Nachrichtenwarteschlange (eng. Message Queue) im Agentensystem den Grenzwert *highWatermark* erreicht, werden die Nachrichten in einen temporären Speicher gelagert. Beim Erreichen der *lowWatermark* Grenze, werden die Nachrichten aus dem Speicher zurück geholt bis die *highWatermark* wieder erreicht wird, oder bis der Speicher leer ist. Der Parameter *blockSize* definiert Anzahl der Nachrichten pro Speicherauktion.

Föderationen. Eine Agentenanwendung kann durch aus mehreren Föderationen definieren. Der Konfigurationsparameter *domaneName* in *lsts.config* identifiziert ein virtuelles Netz.

A3 LS/TS Programmiermodelle

MARGE

Plan hat eine interne Struktur, die aus einer Deklarationsteil und einer Auktionsteil bestehet. Die Deklarationsteil enthält *relevance condition* und *post condition*. Alle Pläne werden manuell zum Startzeitpunkt eines Agenten in einer Planbibliothek (eng. plan repository) registriert, die als eine Abbildungstabelle der Pläne mit ihren Zielen implementiert ist. In der Bibliothek wird jeder Plan mit einem Ziel verknüpft, das erreicht werden soll. Es können auch mehrere Pläne mit einem Ziel verknüpft werden. Abhängig von der Logik eines Agenten sind drei Typen eines Planes einsetzbar:

- MDAL Plan [MDAL (2006)]
- SemCom Plan [SemCom (2006)]
- Process Algebra Plan [DevGuide (2006)]

Goal wird In MERGE BDI Welt durch FOPL Formeln dargestellt, die aus Prädikaten und Variablen besteht. Die Ziele werden in der *initBdi(…)* eines Planes in der Planbibliothek registriert. LS/TS definiert nur *Achieve Goal* und kann auch ein Unterziel zu erzeugen, das zum synchronen Aufruf eines Plans führt. Das Kreieren einer Unterziel folgt immer aus einem Plan, deren Ausführung so lange geblockt bleibt bis das Ziel erreicht wird. Wenn das Erreichen der festgelegten Unterziel gescheitert ist, scheitert auch die gesamte Planausführung. Die Unterziel ist von Type *MDALGoal.java*.

Intention ist das logische Bindeglied zwischen Wissen, Zielen und Plänen eines Agenten. Sie sind als Datenstrukturen definiert, die Informationen über das Ziel mit ihren gescheiterten, suspendierten und aktuellen Plänen besitzen.

Environment & Knowledge Die Darstellung der MARGE Umwelt wird mit Ontologie beschrieben, die die Prädikaten und Funktionen definieren. Der Zugriff auf die Ontologie mit ihrer definierten Welt erfolgt über internes Wissensbasis. Die unterschiedlichen Unifikationsmethoden dienen zum Herausfinden eines geeigneten Plans oder Planmenge zum Erreichen eines Ziels oder zum Abfragen der Wissensbasis. Mehr dazu in [MARGE 2006].

LISA

Es existieren folgende *Continuations* Typen:

- *input hadler* ist eine Komponente, die nach dem Eintritt eines Ereignisses darauf reagiert, es konsumiert und eine Aktivität als Resultat ausführt. Der *input handler* besitzt spezifische Befehle und Filtern. Wenn eine Filterbedingung positiv ausgewertet wird, werden die entsprechenden Befehle ausgeführt.
- *child task* kann einen oder mehreren *child task* besitzen, die im gleichen Kontext wie die *parent task* ausgeführt werden. Bei Eintritt einer Nachricht zum *input handler* besitzt der *parent task* einen höheren Priorität als *child task.*

- Subtask kann einen oder mehrere *sub task* besitzen und kreieren.s*Sub task* verfügt über sein eigenes Kontext. Damit steht der p*arent task* in voller Unbewusstheit über eingetroffene Nachrichten an Kontext seines Kindes.

- Trigger löst das Ausführen der *Continuations Command.*

- Notifacation werden zu bestimmten Zeiten ausgeführt. Die Ausführung kann periodisch ablaufen.

Die Nachrichten eines LISA Agenten werden in einem *root task* verwaltet und behandelt. LISA b*esitz über* einen *input inbox* zum Speichern der eingetroffenen Nachrichten. Dabei wird unterschieden zwischen kontextlosen und kontextbehafteten *input*. Es sind drei Szenarios beim Empfang einer Nachricht zu unterscheiden:

- Beim Eintreffen einer kontextlosen Nachricht wird LISA Agent einen neuen Kontext und einen neuen *root task* erzeugen. Wenn der *root task* die eingetroffene Nachricht nicht behandelt, wird sie verworfen.

- In dem Fall, wenn eine kontextbehaftete Nachricht zu einem nicht existierenden Kontext eintritt, wird die Nachricht von speziellem *processor task* behandelt, der für verlorene Nachrichten, Ereignisse zuständig ist.

- Im letzten Fall, wenn Nachricht zu einem existierenden Kontext adressiert, wird die Nachricht an *task* dieses Kontextes zum Bearbeiten geschickt.

Zum Behandeln einer kontextlosen Nachricht kann der *root task* einen *child task,* einen *sub task* oder einen *Independent Task* erzeugen und ihn beauftragen sie zu bearbeiten.

SemCom

Linguistic Tier realisiert *FIPA ACL* Standarten definiert vier Performative:

- *request* beauftragt einen Empfänger eine bestimmte Handlung ausführen.

- *notify* signalisiert über Eintreffen eines Ereignisses

- *ask-if* ist eine Anfrage die nur war oder falsch sein kann.

- *ask-which*. In dieser Anfrage verwändet SemCom analog der Prolog Programmiersprache, logische Variable, die in einer unvollständigen Anfrage geschickt werden. Der Empfänger bindet fehlende Information an die Variable und sendet die vollständige Information in einer Nachricht zurück.

Ein weiteres wichtiges Element dieser Schicht ist die ACL Ontologie[47]:

> *"The set of entities that are required to appear in the message content as the consequence of the constraints set by the chosen performatives are called the ACL Ontology". [SemCom 2006]*

47 Ontologie (v. griech.) - Lehre von Sein und seinen Prinzipien [Wahrig 1999]

ACL Ontologie definiert Drei Objekte der Kommunikationsmodell: eine Handlung (eng. Auktion), ein Ereignis und eine Objektbeschreibung (engl. object description). *Object decription* ist eine Entität der Agentenwelt, die ihr Eigenschaften (engl. property), *actions* und *evets* hat. Die Ontologie beschreibt eine Menge speziellen Sätzen (engl. Clause): *isTrue, isFalse, isAnswerFor* und eine Menge der Actionoperatoren: *Done, WillDo, WillNotDo, HaveFaled*. Die Sätze *isTrue* und *isFalse* behaupten, dass der Argument war oder falsch ist. Der Satz *isAnswerFor* besteht aus einer Liste der logischen Variablen mit ihrem gebundenen Werten und einem Satz. Mit dem Actionoperatoren *Done* wird über eine ausgeführte *action* signalisiert. Die *WillDo* und *WillNotDo* Actionoperatoren drücken die Bereitschaft eines Agenten zum Ausführen einer *action*. Mit *HaveFailed* Actionoperatoren wird benachrichtigt, dass Ausführen einer *action* fehlgeschlagen ist.

Domain Tier In dieser Schicht werden die Gesprächsthemen definiert.

> *„A structured conversation topic is called Domain Ontology, simply Ontology.”[SemCom 2006].*

Die Ontologie ist mit *Ontology Web Language (OWL)*[48] definiert. SemCom implementiert drei Arten dieses Domain als Untertype der ACL Ontologie: Domain Action, Domain Event und Domain Object Description. Die ACL Ontologie ist selbst in OWL ausgedrückt, um richtige Verweise auf Elemente der Domainontologie durchzuführen.

Social Tier verbindet mehrere Nachrichten zusammen, um einen Gespräch auf zu bauen. In dieser Schicht definiert LS/TS mehrere Interaktionsprotokolle. Zum Implementieren der Protokolle wird MDAL Message Handler benutzt. Die Agenten bekommen, abhängig von einem Protokolls und ihrer Rolle im Gespräch, eine Menge von Fragmenten, die analog MDAL in einzelne Methoden *step_1(), …, step_n()* aufgeteilt sind. Einem Programmierer ist eine Möglichkeit gegeben, die beide architektonische Elemente SemCom und MDAL zum Definieren seiner eigenen Interaktionsprotokolle benutzen. Mehr zu LS/TS Interaktionsprotokollen in [SemCom (2006)].

LS/TS bietet automatisches Generieren von OWL Objekten zu Java Instanzen um das zu realisieren wird *Simple Ontology Framework API (SOFA)*[49] benutzt.

48 http://www.w3.org/TR/owl-ref
49 http://sofa.projects.semwebcentral.org

A4 Laufzeit Eigenschaften des Multiagentensystems (MbFSlP)

Zum Anfang der Erprobungsphase betrug die Anzahl der Spielrunden in der Agentensimulation etwa 25 Runden und von der Laufzeit etwa 2 Stunden (s. Kap. 5.4.1). Aufgrund ständig wachsender Paketmenge im System, war zu dieser Zeit die Stabilität der Simulation und Ausführungszeit mit der Anzahl der Spielrunden als Indikatoren des Paketwachstums gesetzt. Als erstes Ziel wurde es festgelegt, die Einflussfaktoren (Konfigurationsparameter) zu finden um die wachsenden Pakete stabilisieren zu können. Leider zeigten die zahlreiche und vor allem zeitaufwendige Testen, dass die Parameteränderungen keinen oder geringen gewünschten Effekt bringen. Der Einsatz der vertikalen Skalierung (engl. scaling up)[50] erhöhte die Ausführungszeit nur bis zu 40 Spielrunden. Ein Teil der Testversuche ist in der Tabelle A4. 1 dargestellt. Zu den ersten Tests der Erprobungsphase gehören die Testversuche 1 bis 8, 12 und 13.

Das Ändern der Konfigurationsparameter brachte keine positiven Ergebnissen, der Paketwachturm blieb konstant. Das erforderte neue Einsätze beim Testen. Um die Ursache des Paketwachstums festzustellen, wurde ein Versuch vorgenommen die Simulationszeiten zu verlängern. In diesem Versuch werden die Pakete, die mit einer Spielrunde nicht zum Zielort abtransportiert sind, aus der Simulation gelöscht. Die neu generierten Pakete im System sollen dabei eine stabile und konstante Menge bilden, die einen Simulationsverlauf nah zum unendlich machen.

Es wurden zwei Testversuche ausgeführt. Bei dem ersten Testversuch wurden die Pakete sofort nach erster Runde gelöscht (s. Test 9). Bei dem zweiten Testversuch wurden die Pakete erst nach der dritten Runde aus dem System entfernt (s. Test 10). Wie es zu erwarten war, zeigten die Versuchsergebnisse die drastisch gestiegen Ausführungszeiten der Simulation. So schaffte die Simulation bei dem ersten Test 179 Spielrunden. Aus der Beschreibung des Tests 9:

Ablauf: Alle Pakete, die in einem Schritt zum Ziel nicht kommen, werden gelöscht und die neuen Pakete an ihre Stelle neu generiert werden.

Resultat*:* Am Anfang der Simulation beanspruchte sie 816 MB. Speicher und am Ende 1,45 GB. Das zeigt, dass der Speicherverbrauch der Simulation wächst sogar ohne nicht angekommenen oder verspäteten Paketen.

Beide Tests zeigten, dass die Simulation trotz der abgeschalteten Pakete immer schwerer wird und ihre Kapazitätsgrenze erreicht. Was bringt dann das System in diesen Zustand? Es gibt drei weitere Varianten zum Untersuchen:

- Lernfähigkeit der Pakete, weil sie im Verlauf der Simulation einige Informationen in Gedächtnis der Agenten speichern. Diese Information wird eventuell für die ganze Ausführungszeit der Simulation beibehalten und damit eine mögliche Auswirkung auf den Speicherverbrauch haben.

- GC soll unter die Luppe genommen werden. Die Agentensimulation arbeitet mit mehreren Millionen Objekten und es kann sein, dass nicht alle Objektreferenzen freigegeben werden.

50 http://msdn.microsoft.com/en-us/library/aa292203.aspx

- Jadex Plattform. Die Agentenanwendung ist sehr Nachrichtenorientiert und es kann vorkommen, dass die Menge der gesendeten Nachrichten für das ständige Steigen des Speicherverbrauchs verantwortlich ist.

Um die Frage zu beantworten, war ein weiterer Testversuch durchgeführt, in dem die Lernfähigkeit der Agenten abgeschaltet war. Aus der Beschreibung des Tests 14:

Ablauf: Der Test 9 soll erweitert werden, es wird versucht alle Elemente der Lernfähigkeit zu eliminieren. Unter Elementen der Lernfähigkeit ist die Historie der Truck-Agenten und der Paketobjekten gemeint. Die Trucks besitzen zwei Historie Listen „know HUB“ und „route_history“, die in der Simulation nicht benutzt werden. Die Pakete haben "HUBs_history" und "history" Listen. Die Paketlernfähigkeit kommt in der Klasse *MatchingPlan.java* zum Einsatz. Sie wird nur für eine Ausgabe eines Loggers in der Klasse *LoginTruckPlan.java* benutzt.

Resultat: Die Simulation ist mit der Anwendungsmeldung *"still awaiting acks from folows HUBs: ..."* hängen geblieben (s. Test 14). Der weitere Test 15 soll diesen Testversuch wiederholen. Das vorläufige Ergebnis des Tests ist, dass Agentensystem immer noch ein Wachstum an den Objekten aufweist.

Der Test 15 gezeigte, dass die Agentensimulation sogar mit geringerer Paketmenge im System und abgeschalteter Lernfähigkeit immer noch schwerer und speicherintensiver durch ständig wachsende Objekte wird. Allerdings, hat sich die Ausführungszeit der Simulation bei diesem Test im Vergleich zum Test 9 verdoppelt. Daraus folgend wächst die Speicherbelastung immer noch weiter nur über mehrere Spielrunden verteilt. Der Test war durch manuellen Abbruch beendet (s. Test 15).

Zum Testen zwei weiteren Varianten (GC und Jadex) wurde ein Analysesoftware JProfiler[51] benutzt. Test Vorbereitung:

- Die Lernfähigkeit der Agenten wird abgeschaltet

- Die Existenz der Pakete wird unabhängig von ihrer Type (Express, Standard) und Lieferungsfrist auf eine 1 Spielrunde begrenzt.

Resultat: Der Simulationsverlauf dauerte etwa 18 Stunden und war manuell abgebrochen. Die Abbildung A4 1 zeigt die Ausführungszeit der Simulation und einen leichten Zuwachs der Objekte. Jadex dagegen zeigte sich während der Simulation als eine solide und stabile Plattform. Das System zeigte einen stabilen Charakter (s. Abbildung A4 2).

51 http://www.ej-technologies.com/ (August 2009)

Session: New session
Time of export: Saturday, March 28, 2009 11:28:36 AM CET
JVM time: 1105:21

Sorted by: Instance count
Aggregation level: Classes

Name	Instance count	Difference	Size
java.lang.Object[ ]	472.725	+43.116 (+10 %)	19 MB
java.util.ArrayList	341.574	+28.180 (+9 %)	7 MB
java.util.HashMap$Entry	192.038	+65.900 (+52 %)	4 MB
char[ ]	166.835	-108.099 (-39 %)	10 MB
java.lang.String	134.234	-96.292 (-42 %)	3 MB

Abbildung A4 1: Testlaufzeit.

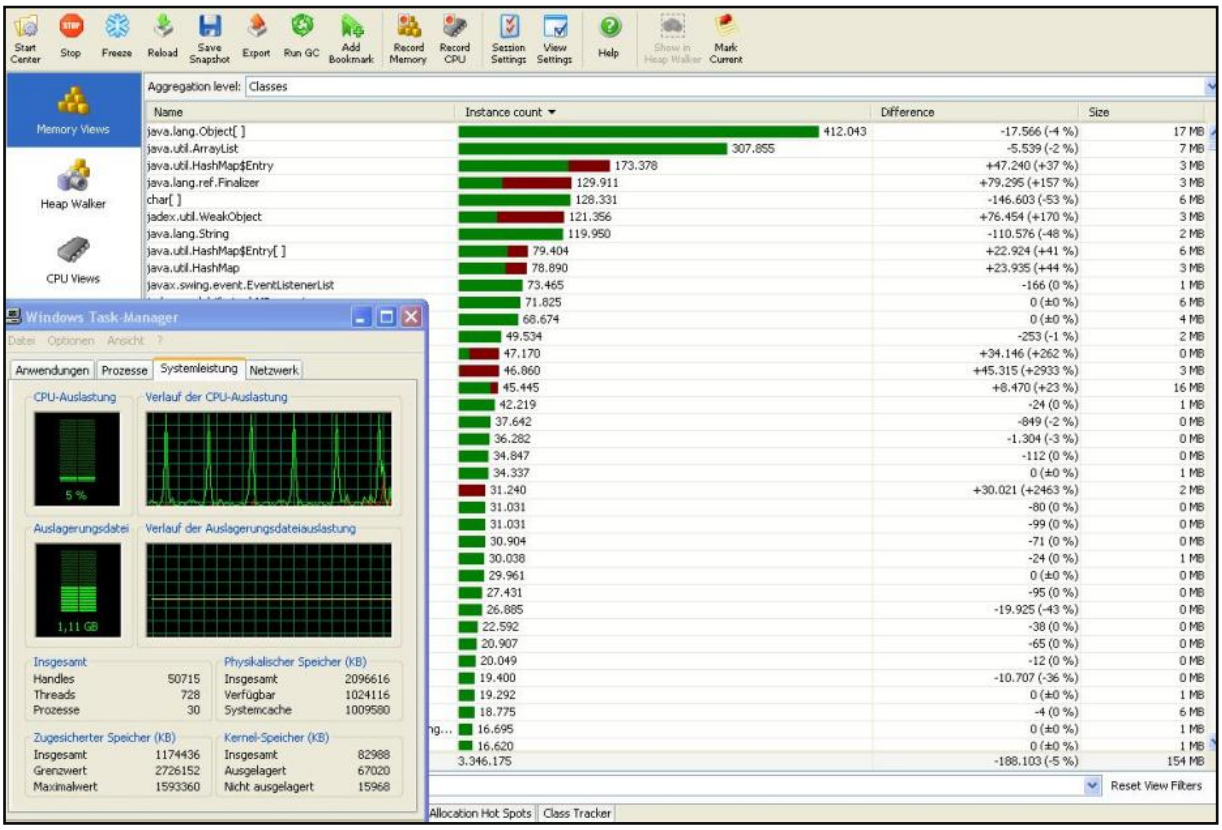


Abbildung A4 2: Agentenanwendung Test mit JProfiler.

Bei der Durchführung eines weiteren Tests wurden die gleichen Konfigurationsparameter benutzt aber ohne JProfiler.

Resultat: Die Simulation dauerte etwa 27 Stunden und hat 835 Spielrunden geschafft. Die Simulation wurde allerdings durch eine Systemfehlermeldung *„OutOfMemoryError“* abgebrochen. Der letzte Test hat keine neuen Erkenntnisse gebracht.

Tabelle A4. 1: Tests Übersicht

Testversuch	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Paket im System	3Mil		1Mil									2Mil	2Mil			
Trucks Limit	80	150				200	200	300	200	200	200	100	100	200	200	200
Alternative	5						33	33	33	33				33	33	33
K_Pakets	200															
K_Trucks	10						15	15								
Truck Strategie								DRP Strategy*								
Packet Strategie								PRT Strategy*								
Preis standard Paket	7			3	3											
price_per_km_without_trailer min	1												2			
price_per_km_with_trailer min	2												4			
Preis expess Paket	15			6	6											
Deadline standard Paket	90				25											
Deadline express Paket	96				50											
Budget Standard Paket	70									100	100			100	100	100
Budget Express Paket	70									100	100			100	100	100
max_delivery_duration_exp min	4								4	4	4			4	4	4
max_delivery_duration_std min	12								8	8	8			8	8	8
Runde		40					35		179	52	37	34	34		303	264
Start um		9:28	12:28	10:06	13:06	9:26	11:10	15:30	16:13	18:47				11:11	12:50	12:32
Ende um		11:26	14:22	11:10	13:45	11:15	14:16	18:20	18:25	20:32				12:41	17:04	15:15

DRP Strtegy*- DirectRoutePacketStrategy.java

PRT Strategy*- ProfitableRouteTruckStrategy.java

A5 Jadex Agenten Nachrichtenprotokoll

A5 1: Jadex zu Environment. First Initial Message.

Jadex Initiator		Environment Participant	
Send	Message		Receive
	Jadex	*request_start_simulation*	
	Performative		
	Request		
	Content		
	„start“		
Plan	Jadex	*start_simulation*	Plan
Body	Jadex	*SimulationStartPlan.java*	Body

Legende: Die Nachricht *request_start_simulation* ist der Grundstein in der gesamten Multiagentensimulation. Genau nach dem Eingang der Nachricht mit Inhalt *„start“* als Parameter wird der Plan *start_simulation* (*SimulationStartPlan.java*) durch einen Trigger ausgelöst und damit den gesamten Simulationsablauf zum Ausführen anstoßen. Die Nachricht wird manuell von Jadex Agenten geschickt.

A5 2: Environment zu HUB. Simulationsstart.

Environment Initiator		HUB Participant	
Send	Message		Receive
	inform_simulation_start	*Inform_simulation_start*	
	Performative		
	“simulation_start”		
	Content		
	MsgEnvHUBLis OfHUBs.class		
Receive	Reply		Send
	inform_ack	*Inform_ack*	
	Performative		
	Inform		
	Content		
	“ok_from ” + Name des HUB Agenten		
Plan	*start_simulation*	*simulation_start*	Plan
Body	*SimulationstartPlan.java*	*SimulationStartedPlan.java*	Body

Legende: Zum Starten der Simulation versendet der Agent an alle HUB-Agenten die Nachricht *Inform_simulation_start* mit einer Benachrichtigung über Startvorgang der Simulation und einer Liste mit allen Beteiligten HUB-Agenten. Auf diese Nachricht wird eine Antwort mit dem Namen des Absenders verlangt.

A5 3: Environment zu HUB. Spielrundenstart.

Environment Initiator		HUB Participant	
Send	Message		Receive
	inform_start_round	*request_round_started*	
	Performative		
	Inform		
	Content		
	„round_started “ + Nummer der Spielrunde		
Plan	*all_acks_from_trucks_received*	*start_new_round*	Plan
Body	*HUBsNextRoundStartPlan.java*	*NextRoundStartedPlan.java*	Body

Legende: Die Nachricht *inform_start_round* wird vor dem Start jeder neuen Spielrunde an alle HUB-Agenten mit der aktuellen Spielrundennummer versendet.

A5 4: Environment zu Truck. Tracks Parameter Veränderung.

Environment Initiator		Truck Participant	
Send	Message		Receive
	Inform_parameter_changed	*parameter_changed*	
	Performative		
	Inform		
	Content		
	"new_parameters “ + Paramterliste		
Plan	*generate_new_packets*	*change_parameters*	Plan
Body	*GenerateNewPacketsPlan.java*	*ChangeParametersPlan.java*	Body

Legende: Die Nachricht *Inform_parameter_changed* wird an alle Truck-Agenten versendet, wenn irgend-welche Änderungen an Truck-Parameter vorgenommen wurden. Die Nachricht beinhaltet aktuelle Parameterdaten, die für einen Truck-Agenten relevant sind.

A5 5: Environment zu Truck. Spielrundestart.

Environment Initiator		Truck Participant	
Send	Message		Receive
	inform_start_round	*request_round_started*	
	Performative		
	Inform		
	Content		
	“round_started”		
Plan	*start_round*	*round_started*	Plan
Body	*TrucksNextRoundStartPlan.java*	*NextRoundStartedPlan.java*	Body

Legende: Die Nachricht *inform_start_round* wird an alle Truck-Agenten gesendet. Sie teilt über einen Anfang einer neuer Spielrunde mit. Die Nachricht wird verschickt nur dann, wenn sich alle Beteiligten an der Simulation LKWs beim Environment-Agenten gemeldet haben.

A5 6: HUB zu Environment. Positive Bestätigung über das Ende der Verhandlungen.

HUB-Initiator		Environment Participant	
Send	Message		Receive
	inform_ack	*inform_HUB_ack*	
	Performative		
	Inform		
	Content		
	„HUB_ack “ + Name des HUB-Agenten		
Plan	*sent_ack*	*inform_HUB_ack*	Plan
Body	*SendAckPlan.java*	*HUBAckReceivedPlan.java*	Body

Legende: Die Nachricht *inform_ack* wird nach dem Beenden der Verhandlungen im HUB-Agenten zwischen LKWs und Paketen an Environment-Agenten geschickt. Sie signalisiert dem Environment-Agenten über die Bereitschaft des HUB-Agenten zur neuen Spielrunde.

A5 7: HUB zu Truck. Anfrage zu einem Angebot.

HUB-Initiator		Truck Participant	
Send	Message		Receive
	inform_truck_trade_round	*inform_make_offer*	
	Performative		
	Request		
	Content		
	MsgHUBTruckPacketsDistribution.java		
Receive	Reply		Send
	inform_offer_truck	*inform*	
	Performative		
	Inform		
	Content		
	Offer.java		
Plan	*inform_truck_round*	*make_offer*	Plan
Body	*InformTruckTradeRoundPlan.java*	*MakeOfferPlan.java*	Body

Legende: Die Nachricht *inform_truck_trade_round* wird an einen Truck-Agenten verschickt um einen Angebot *Offer.java* vom Verkäufer zu erhalten.

A5 8: HUB zu Truck. Pakete laden.

HUB-Initiator		Truck Participant	
Send	Message		Receive
	inform_load_packets	*load_packets*	
	Performative		
	Inform		
	Content		
	MsgHUBTruckListOfPackets.java		
Plan	*load_packets*	*load_packets*	Plan
Body	*LoadPacketsPlan.java*	*LoadPacketsPlan.java*	Body

Legende: Die Nachricht *inform_load_packets* signalisiert einem Truck-Agenten, dass er die ausgehandelte Transportgut-Ladung beladen muss.

A5 9: HUB zu Truck. negative Bestätigung über erfolgloses Handeln.

HUB-Initiator		Truck Participant	
Send	Message		Receive
	trade_nack	*trade_nack*	
	Performative		
	Failure		
	Content		
	"trade_nack"		
Plan	*matching*	*trade_failed*	Plan
Body	*MatchingPlan.java*	*TradeFailurePlan.java*	Body

Legende: Die Nachricht *trade_nack* wird nur im Fall erfolglosen Verhandlungen an einen Truck-Agenten versendet.

A5 10: HUB zu Tuck. Start des Reservierungsvorganges.

HUB-Initiator		Truck Participant	
Send	Message		Receive
	trade_ack	*trade_ack*	
	Performative		
	Inform		
	Content		
	"trade_ack" + Anzahl Geboten von Paketen.		
Receive	Reply		Send
	reservation_result	*reservation_result*	
	Performative		
	Inform		
	Content		
	“reservation_result ”		
Plan	*Matching*	*trade_ack*	Plan
Body	*MatchingPlan.java*	Bo *TradeAckPlan.java*	Body

Legende: Die Nachricht *trade_ack* wird an einen Truck-Agenten gesendet, der an den Verhandlungen teilgenommen hat. Diese Nachricht signalisiert einem Truck-Agenten über erfolgreich abgeschlossene Verhandlung zwischen Paketen und LKW.

Genau diese Nachricht ist wie eine Anstoßkugel in der Aktivitätsreihe zum Reservieren der Lagerfläche in einem HUB-Umschlagspunkt (es wird ein Ziel-HUB gemeint, zu dem der LKW nach der Verhandlung seine Transportgut-Ladung abtransportieren soll). Der nummerische Wert *Anzahl Geboten von Paketen* repräsentiert eine Menge der Pakete, die in einem LKW von aktuellen HUB (Verhandlungsort) zum Ziel-HUB abtransportiert muss. Als Antwort wird eine positive oder eine negative Bestätigung des Reservierungsvorgangs erwartet. Die erfolgreiche Reservierung einer Lagerfläche führt zum Erzeugen einer Tour.

A5 11: HUB zu Truck. Abmeldung eines LKWs von einem HUB.

HUB-Initiator		Truck Participant	
Send	Message		Receive
	confirm_logout	*logout*	
	Performative		
	Confirm		
	Content		
	“logout”		
Receive	Reply		Send
	confirm_logout_ack	*confirm_logout*	
	Performative		
	Confirm		
	Content		
	"ok"		
Plan	*logout_truck*	*logout_plan*	Plan
Body	*LogoutTruckPlan.java*	*LogoutPlan.java*	Body

Legende: Die Nachricht *confirm_logout* wird an einen Truck-Agenten gesendet, um ihn zum Abmelden von HUB-Plattform aufzufordern. Auf diese Nachricht wird eine Bestätigung vom korrespondierenden Truck erwartet.

A5 12: HUB Interne Nachricht. LKW ist abgemeldet.

HUB			
Send	Message		Receive
	truck_logined		
Plan	*login_truck*	*create_new_truck*	Plan
Body	*LoginTruckPlan.java*	*CreateNewTruckPlan.java*	Body

Legende: Die interne Nachricht *truck_logined* wird versendet, wenn sich ein frisch erzeugtes Fahrzeug erfolgreich an der HUB-Plattform angemeldet hat. Erst nach dem Anmelden des LKWs wird es dem HUB-Agenten gestattet, weitere Fahrzeuge zu erzeugen.

A5 13: Truck zu Environment. Positive Bereitschaft eines LKW zur nächsten Spielrunde.

Truck Initiator		Environment Participant	
Send	Message		Receive
	inform_ack	*Inform_truck_ack*	
	Performative		
	Inform		
	Content		
	„truck_ack “ + Name der Truck-Agenten		
Plan	*send_ack*	*inform_truck_ack*	Plan
Body	*SendAckPlan.java*	*TruckAckReceivedPlan.java*	Body

Legende: Die Nachricht *inform_ack* wird an Environment-Agenten mit dem Namen des LKWs Agenten verschickt. Sie signalisiert über die Bereitschaft eines LKWs Agenten zur einer neuen Spielrunde.

A5 14: Truck zu HUB. Anmeldung eines LKW in einem HUB.

Truck Initiator		HUB-Participant	
Send	Message		Receive
	inform_login_truck	*inform_login_truck*	
	Performative		
	"login_truck"		
	Content		
	MsgTruckHUBListOfPackets.java		
Receive	Reply		Send
	agree	*agree*	
	Performative		
	Agree		
	Content		
Plan	*Login*	*login_truck*	Plan
Body	*LoginPlan.java*	*LoginTruckPlan.java*	Body

Legende: Die Nachricht *inform_login_truck* wird vor dem Registrieren eines LKWs in einem HUB-Umschlagspunkt mit der Information über seine Transportgut-Ladung und aktuelle Spielrunde gesendet. Auf diese Nachricht wird eine Bestätigung von der HUB-Plattform erwartet. Nach der Anmeldung des Truck-Agenten an der HUB-Plattform folgt es: das Aufnehmen des Fahrzeuges in die Liste der registrierten LKWs; Erhöhung der gesamten Transportladekapazität des HUBs, um die Ladegröße des LKWs, Aufnahme der entladenen Pakete in HUBs in der Bestandsliste zu registrieren.

A5 15: Truck zu HUB. LKW Anfrage um freien Lagerplatz.

Truck Initiator		HUB-Participant	
Send	Message		Receive
	request_free_space	*reservation_request*	
	Performative		
	Request		
	Content		
	"reservation_request" + *offer.getGame_round()* + *offer.getDuration()* +		
Receive	Reply		Send
	reservation_request	*reservation_answer*	
	Performative		
	Inform		
	Content		
	"reservation_answer"		
Plan	*plan_request_free_space*	*reservation_request_plan* Pln	Plan
Body	*RequestFreeSpacePlan.java*	*ReservationPlan.java* Body	Body

Legende: Die Nachricht *request_free_space* wird an einen HUB-Agenten versendet, der als nächstes eventuell mögliches Ziel auf dem Transportweg des LKWs steht. Mit dieser Nachricht wird nachgefragt, ob die HUB-Plattform genügend freie Lagerfläche für transportierende Paketmenge zur Verfügung hat. Abhängig davon wird eine Zusage oder eine Absage an Truck-Agenten mit *"reservation_answer"* + *"reservation_answer ok"* bwz. *"reservation_answer nok"* geschickt.

A6 LS/TS Agenten Nachrichtenprotokoll

Die Abbildung A6 1 s. u. stellt das Starten der Agentensimulation und Aktivitäten eines Manager Clients dar:

1. Der Client versendet einen Befehl zum Erzeugen der Agenten. Die Reihenfolge der Agentenkreierung ist festgelegt. Der HUB-Agent bekommt dabei seine Lagerkapazität als Initialisierungsparameter.

2. Die Simulationsparameter werden unverschlüsselt an Initializer-Agenten versendet. Der Agent legt eine neue XML-Datei mit Konfigurationsparameter der Simulation an und versendet eine Bestätigung zurück.

3. „PERF_SIM_START“- Nach der Bestätigung startet der Manager die Agentensimulation.

4. „PERF_SIM_STOP oder PERF_SIM_START“- Im weiteren Verlauf der Simulation kann ein Manager die Simulation zu jeder Zeit anhalten oder wieder starten.

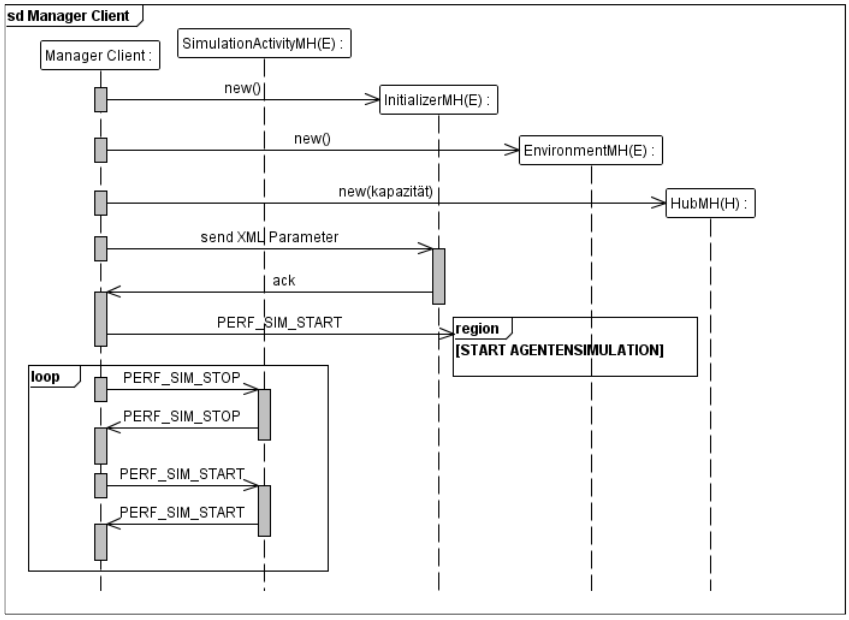


Abbildung A6 1: Start der Agentensimulation

Absteigend eine Abstraktionsebene tiefer stellt das Sequenzdiagramm „Start Agentensimulation“ (s. u. Abbildung A6 2) ein Ablaufszenario aus der Sicht eines Environment-Agenten dar:

1. „PERF_INFORM_SIMULATION_START“- Das Environment informiert alle HUBs über Start der Simulation und wartet bis eine Bestätigung für diese Nachricht von allen HUBs erhalten wird.

2. „PERF_START_WITH_HUBS“- Erst wenn alle HUBs geantwortet haben, wird die erste Spielrunde gestartet. Mit dem Start der ersten Runde gerät der Agent in eine unendliche Schleife, die nur durch einen Manager-Client unterbrochen werden kann.

3. „PERF_CREATE_NEW_PACKETS“- Am Anfang jeder Spielrunde werden immer neue Pakete generiert. Der Timer dient zum Synchronisieren der DB und Agentensimulation. Er

wird nur dann zum Ausführen gebracht, wenn die Pakete noch nicht oder nur zum Teil persistent geschrieben sind.

4. „PERF_GENERATE_TRUCK_DISTRIBUTION“- Als nächster Schritt folgt das Generieren der Verteilung der Truck-Agenten über die HUBs. Das wird nur in der ersten Spielrunde erledigt.

5. „PERF_INFORM_START_ROUND“- Der Agent informiert alle HUB-Agenten über Start einer neuen Spielrunde. Die Nachricht dient auch als eine Erlaubnis die Verhandlung zwischen Paketen und LKW zu starten.

6. „PERF_OK_FROM_HUB“- Alle HUBs senden an Environment eine Bestätigung über das Ende der Verhandlungen und ihrer Bereitschaft zur nächster Spielrunde. Erst nach dem Eintreffen der Bestätigungen von allen HUBs, werden alle Simulationsteilnehmer (HUB-Agenten und Truck-Agenten) mit einer neuen Spielrunde synchronisiert.

7. „PERF_START_WITH_HUBS“- Die Simulation tritt in eine neue Spielrunde.

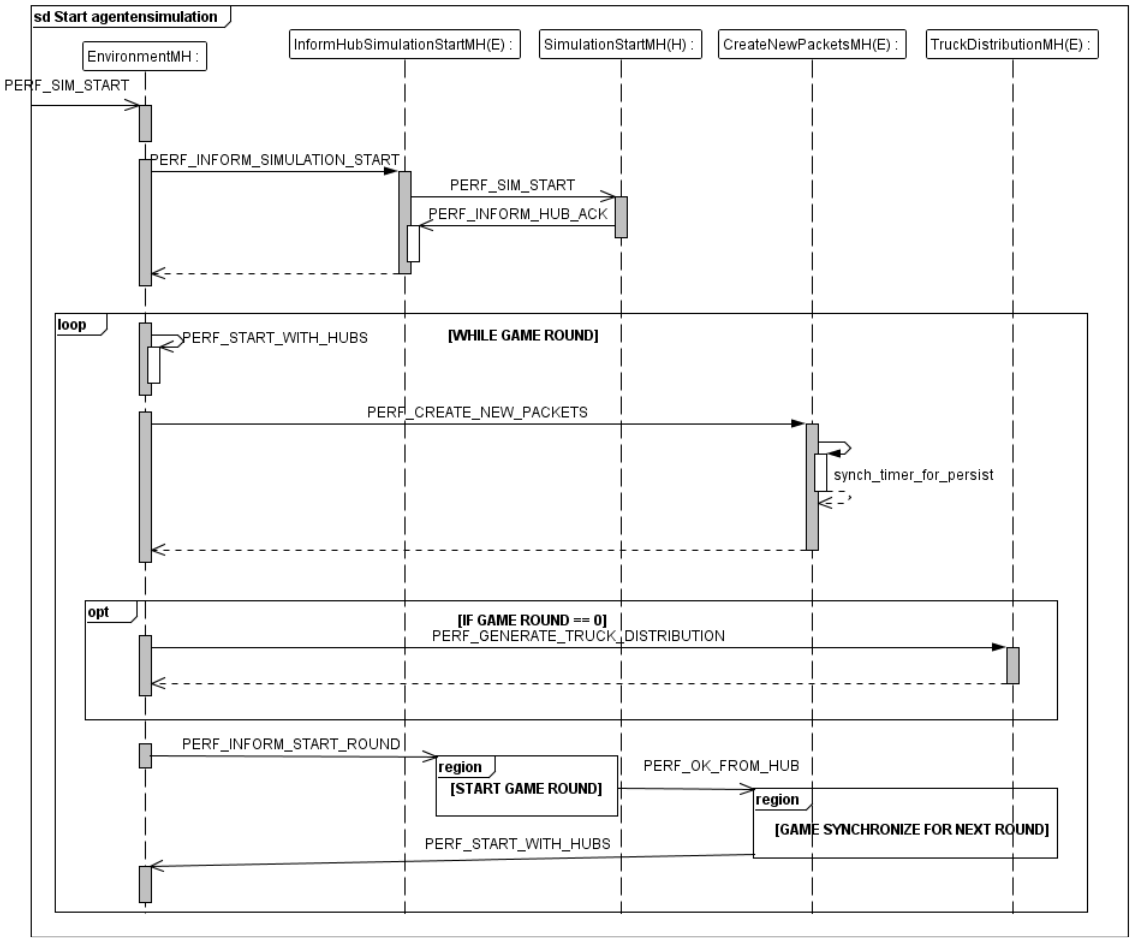


Abbildung A6 2: Verlauf der Agentensimulation

Das Diagramm „Start game round“ (s. u. Abbildung A6 3) zeigt wie ein HUB am Anfang jeder neuen Spielrunde sich zum Start der Verhandlungen zwischen Paketen und LKWs vorbereitet:

1. „PERF_FLAG_ADD_NEW_PACKETS“- Nach dem Eintreffen einer Nachricht von einem Environment holt der HUB neue Pakete von der DB.

2. „PERF_CREATE_NEW_TRUCK“- Im nächsten Schritt, wenn die Agentensimulation sich in der ersten Spielrunde befindet, werden die Truck-Agenten erzeugt. Zum Kreieren der Truck-Agenten für den aktuellen HUB-Agenten wird eine Liste mit Truck-IDs von Environment geholt. Nachdem die Liste da ist, werden die Truck-Agenten sequenziell generiert.

3. „PERF_TRUCK_LOGIN“- Der Truck-Agenten meldet sich in einem HUB.

4. „PERF_AGREE“- Nach dem Anmelden eines Trucks beim HUB-Agenten wird eine Bestätigung zurückgesendet.

5. „PERF_TRUCK_2_DISTRIBUTION“- HUB sendet eine Nachricht, um sequenziellen Verlauf der Truck-Generierung fortzusetzen.

6. „PERF_START_TRADE“- Wenn alle Truck-Agenten in diesem HUB erzeugt sind, wird diese Aktivität verlassen und die Verhandlungen werden gestartet.

7. Alternativ, wenn die Agentensimulation sich in einer weiteren Spielrunde befindet, wird geprüft, ob die Truck-Agenten in dieser Spielrunde zum Anmelden erwartet werden.

8. „PERF_START_TRADE“- Im Fall, wenn kein Truck erwartet wird, startet der HUB die Verhandlungen für die Trucks, die in letzter Runde den HUB nicht verlasen haben.

9. „PERF_START_TRADE“- Wenn jedoch einige Trucks erwartet werden, die aber noch nicht am HUB angemeldet sind, startet ein Timer. Der HUB wartet eine bestimmte Zeit auf die fehlenden Trucks. Nach dem Ablauf der Zeit, wenn die Trucks immer noch nicht angemeldet sind, werden die fehlenden Trucks aus der Anmeldungsliste gestrichen und die Verhandlung wird gestartet.

10. „PERF_READY“- Die Verhandlung wird gestartet, wenn alle Truck-Agenten sich am HUB angemeldet haben und der HUB selbst bereit ist zum Starten der Verhandlung.

11. „PERF_START_TRADE“- Verhandlung

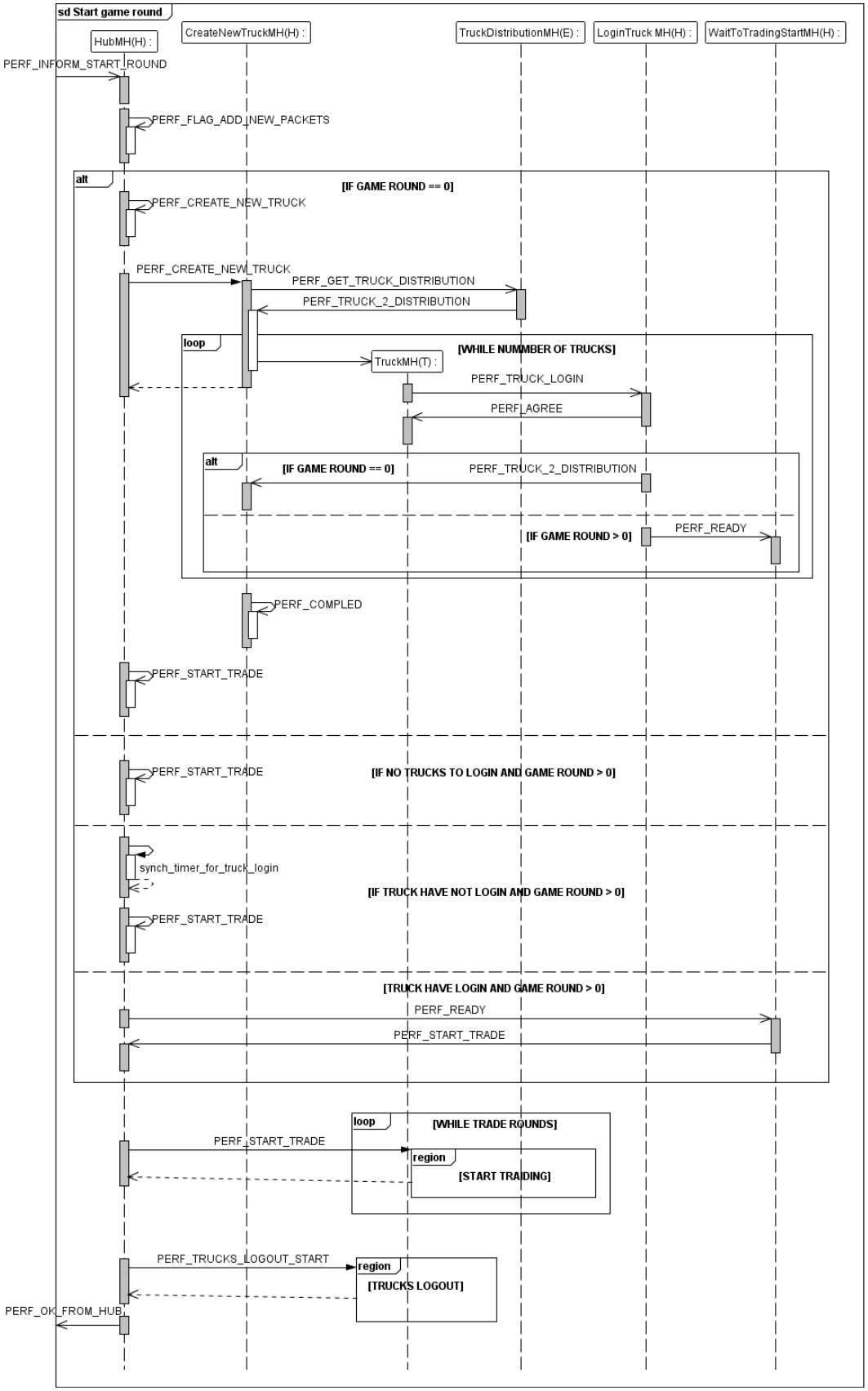


Abbildung A6 3: Start der Verhandlung

12. „PERF_TRUCKS_LOGOUT_START“- Nach der Verhandlung werden die beladenen Trucks abgemeldet (s. Abbildung A6 4).

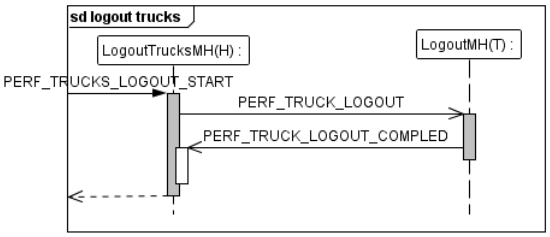


Abbildung A6 4: Abmelden des Trucks

Das Diagramm „Start traiding“ (s. u. Abbildung A6 5) stellt eine Verhandlung zwischen Agenten dar:

1. Zum Anfang der Verhandlung generiert ein HUB die Gebote für jedes Paket in System (aufgrund der getroffenen Einschränkungen wird diese Aktivität im Diagramm nicht dargestellt, weil sie in einem MDAL-Fragment generiert wird).

2. „PERF_INFORM_TRUCK_TRADE_ROUND“- Der HUB informiert alle angemeldeten Truck-Agenten über den Start einer neuer Verhandlungsrunde. Danach versetzt sich der Agent in einen Wartezustand, bis die Trucks ihre Angebote erzeugen und an HUB-Agenten senden.

3. „MATCH“- Nachdem alle Gebote und Angebote erzeugt wurden, startet der HUB-Agenten das Matching.

   Während des Matching wird eine synchrone Reservierungsanfrage gesendet.

4. „PERF_REQUEST_FREE_SPACE“- Der HUB informiert einen Truck, dass er einen Platz für seine eventuell mögliche Ladung reservieren muss.

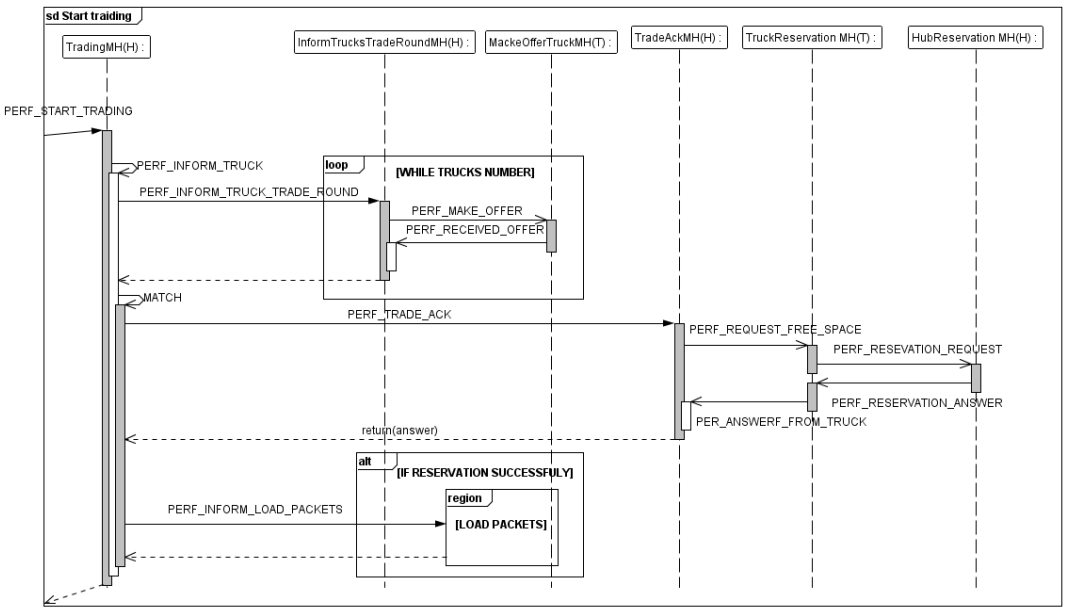


Abbildung A6 5: Verhandlung

5. „PERF_RESEVATION_REQUEST“- Der Truck sendet die Nachricht im Auftrag des HUBs an einen Ziel-HUB und wartet auf eine Bestätigung oder Ablehnung seiner Reservierung.

6. „PERF_RESERVATION_ANSWER“- Der Ziel-HUB prüft seine Ladekapazität und sendet einschließlich eine Antwort zum Truck.

7. „PERF_ANSWERF_FROM_TRUCK“- Der Truck leitet die Antwort weiter an HUB zum Matching.

8. „PERF_INFORM_LOAD_PACKETS“- Wenn die Antwort positiv ist und der HUB die Ladung aufnehmen kann, werden die Pakete in Truck geladen (s. Abbildung A6 6).

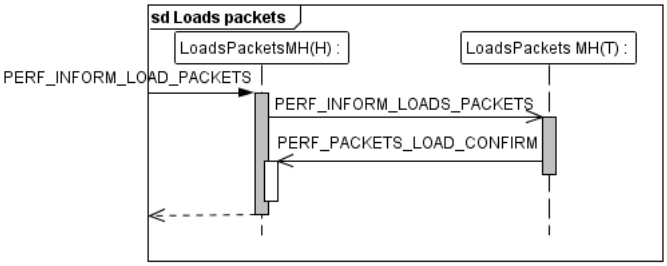


Abbildung A6 6: Beladen des Trucks

Das Diagramm „Game synchronize for next round“ (s. u. Abbildung A6 7) tellt gleich zwei Aktivitäten der Agentensimulation dar: Synchronisierung der Spielrunde und Anhalten der Simulation. Nachdem die Verhandlung abgelaufen ist und alle Truck-Agenten sich von dem HUB-Agenten abgemeldet haben, startet das Mechanismus zum Synchronisieren der Spielrunde in der gesamten Agentensimulation. Die Nachricht „PERF_OK_FROM_HUB“ ist die letzte in einer Spielrunde. Am Ende jeder Runde wird geprüft, ob die Simulation von einem Manager Client angehalten wurde.

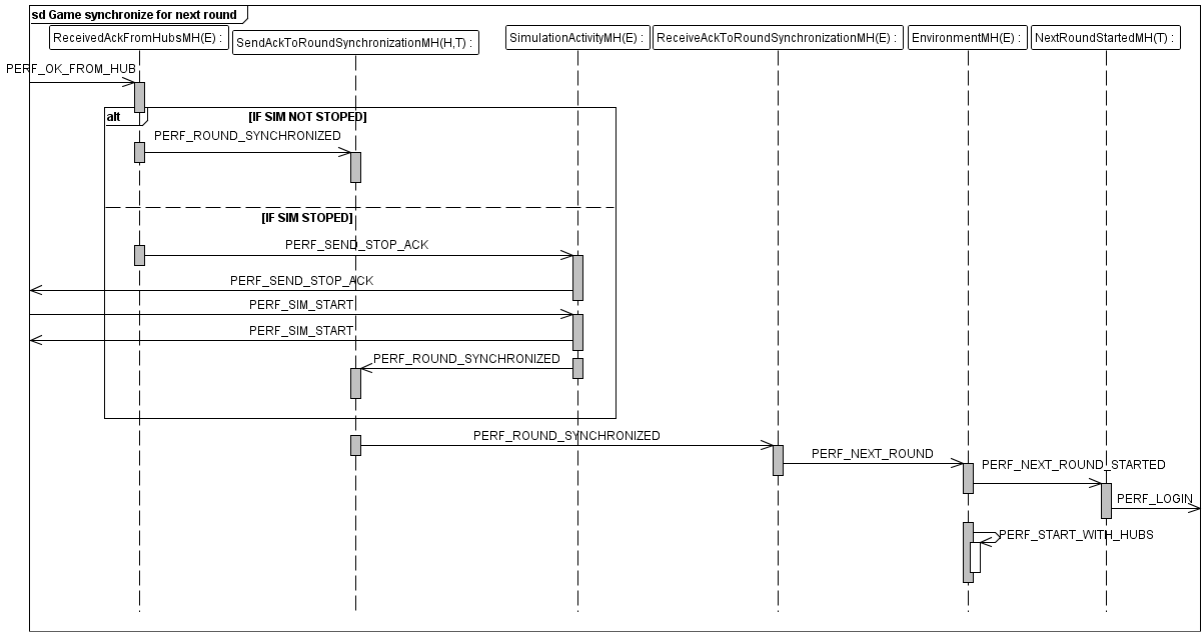


Abbildung A6 7: Spielrundensynchronisierung

1. „PERF_SEND_STOP_ACK“- Wenn das der Fall ist und die Simulation angehalten werden soll, unterbricht Environment die Simulation und sendet eine Nachricht zu einer Schnittstelle, die fürs Aktivität der Simulation verantwortlich ist (SimulationsActivityMH). Von dieser Schnittstelle sendet Environment eine Nachricht um einen Manager zu informieren, dass die Simulation unterbrochen wurde und befindet sich zur dieser Zeit im passiven Warten.

2. „PERF_SIM_START“- Nach dem Eintreten der Nachricht von einem Manager, sendet Environment eine Bestätigung zurück und informiert die Truck-Agenten und HUB-Agenten über Start einer neuen Spielrunde.

3. „PERF_ROUND_SYNCHRONIZED“- Wenn die Simulation nicht angehalten werden soll, vermeidet der Agent die Aktivitätsschnittstelle und sendet eine direkte Nachricht an die Truck-Agenten und HUB-Agenten.

4. Die HUBs und die Trucks bestätigen das Synchronisieren einer Spielrunde mit einer versendeten Nachricht an Environment.

5. „PERF_NEXT_ROUND“- Alle Agenten haben die Synchronisierung der Spielrunde erfolgreich abgeschlossen.

6. „PERF_NEXT_ROUND_STARTED“- Environment informiert die Truck-Agenten über den Start einer neuen Spielrunde und schlägt allen Truck-Agenten vor, sich bei den HUB-Agenten anzumelden.

A7 Technische Realisierung der LS/TS Agenten

In diesem Abschnitt der Arbeit wird das technische Realisieren der Agenten beschrieben.

Manager Client.

SimulationsManager.java Die Hauptaufgabe dieser Klasse ist die Eingaben eines Anwenders zu verarbeiten. Die Ausführung des Managers beginnt mit dem Start eines Wizards, der die Anweisungen entgegennimmt und ausführt (s. Abbildung 6.2). Im folgenden Verlauf der Arbeit sind die einzelnen Schritte des Wizards verfolgt.

Das Generieren der Datei mit Simulationsparameter erfolgt durch Ausführen der Methode *generateDefaultXML(String xml_param*). Die Methode erzeugt mit Hilfe eines *ParametersParser.java* eine Simulationsdatei, die nachher von einem *XMLParameterConverter.java* an die neue XML Struktur angepasst wird. Der Parser an seiner Stelle benötigt zum Generieren der Parameter eine Liste mit HUBs Namen, die bei der Initialisierung eines Clients aus der DB geholt wird.

Das Modifizieren der Parameter mit der Unterstützung eines Visualisierungstools geschieht durch das Ausführen der Methode *startControllTool(String fileName, List HUBs)*. Das Tool erfordert nicht nur eine Liste mit Namen aller HUB-Agenten, sondern auch das Instanziieren der *GlobalParameters.java*. Es wird mit Parametern aus dieser Instanz initialisiert und die modifizierten Parameter werden ebenfalls zurück in *GlobalParameters.java* gespeichert. Das Steuerung Tool wird in einem separaten Thread ausgeführt.

Zum Parametrisieren der Agentensimulation werden zuerst die LS/TS Clients (s.u. *ManagerClient.java*) mit *createClients()* erzeugt. Zum Instanziieren jedes einzelnen Klienten werden die Konfigurationsdateien „clients.properties“ und „agent_distribution.xml“ benutzt sowie die Referenz eines Managers und Liste mit Agentennamen. Nach dem Verbindungsaufbau werden die Agenten mit *createAgent()* erzeugt.

Aus der Sicht eines LS/TS Client bedeutet das Kreieren eines Agenten das Versenden eines „*Create Command*“ Befehls an eine entfernte Agentenplattform. Dann wird an der ersten Stelle in jeder LS/TS Laufzeitumgebung ein Initializer-Agent erzeugt (s. u. Kap. 0). Dieser Agent nimmt gesendeten Simulationsparameter entgegen (s. Abbildung 6.4). Erst dann werden der Environment-Agent und die HUB-Agenten kreiert. Die HUB-Agenten werden mit Initialisierungsparametern instanziiert. Für jeden HUB-Agenten wird mit „*Create Command*“ eine Lagerkapazität der HUB versendet. Schließlich werden die Simulationsparameter mit *sendXMLToAgents(String fileName)* als ein String an jede entfernte Agentenplattform gesendet. Danach versetzt sich der Manager in Zustand *„Passives Warten“.* Er verbleibt in diesem Zustand so lange bis alle Initializer-Agenten das Erhalten der Parameter bestätigen.

Das Starten der Agentensimulation wird mit *simulationStart()* begonnen. Dafür wird eine asynchrone Nachricht an einen Environment-Agent gesendet.

Das Steuern des Simulationsverlaufes (Anhalten und Wiederstarten ihrer Ausführung) ist mit einem *Synchronize Block* implementiert. Nach jedem Nachrichtenversand wartet der Manager auf eine Bestätigung. Das Anhalten der Nachricht wird allerdings doppelt bestätigt zuerst als Erhalten der Anweisung und dann als Ausführen der Anweisung.

Die Kommunikation zwischen Manager und einer verteilten Simulation geschieht über die Schnittstelle *onMessage(IReceiveMessage message)* die von dem Interface *MessageListner.java* zur Verfugung gestellt wird. Jeder einzelnen LS/TS Client erhält die Nachrichten über dieselbe Schnittstelle und leiten die angekommenen Nachrichten an Manager über seine Referenz, die bei der Instanziierung mitgeteilt wurde.

ManagerClient.java ist eine Instanz, die eine Verbindung mit einer Agentenplattform aufbaut, die Verbindungsparameter initialisiert, Anweisungen (Befehle und Nachrichten) versendet, Kapazität eines HUBs mit *getHUBCapacity(String HUBName)* berechnet.

Environment-Agenten

Tabelle A7 1: Environment Data Storage

Data Storage	Beschreibung
DS_ROUND	Aktuelle Spielrunde in der Agentensimulation
DS_STATUS_QUO	Boolesche Variable, um die Parameter über Web Schnittstelle zu erhalten.
DS_DS_UNRECEIVED_ACKS_FROM_HUBS	Eine Liste mit Namen der HUB-Agenten, die noch keine Bestätigung an Environment gesendet haben. Sie wird zum Synchronisationszwecken benutzt.
DS_ DS_SYNCHONIZATION_AGENT_LIST	Eine Liste mit Namen von allen Agenten (HUB-Agenten und Truck-Agenten), die noch keine Bestätigung an Environment gesendet haben. Sie wird zum Synchronisationszwecken benutzt.
DS_ DS_HUBS_ID_LIST	Eine Liste mit ID's aller HUB-Agenten der Simulation.
DS_ DS_TRUCKS_ID_LIST	Eine Liste mit ID's aller Truck-Agenten der Simulation.
DS_ DS_TRUCKS_4_DISTRIBUTION	Eine „Key – Value" Liste. Sie enthält einen HUB als Key und eine Menge der Trucks als Value; wird zum Verteilen der Trucks über HUBs benutzt.
DS_ DS_HUBS	Eine Liste mit *HUB.java* Objekten. Sie wird an jeden HUB-Agenten gesendet.
DS_ DS_PACKETS_VOL	Eine Liste wird als eine temporäre Variable beim generieren der Paketen benutzt.
DS_SIM_STOP	Eine Liste repräsentiert einen Zustand des Simulationsverlaufs.

HUBsNextRoundStartFrag.java repräsentiert einen Anfang der Spielrunde. Das Fragment generiert in jeder Runde neue Pakete. In der ersten Spielrunde wird eine Verteilung der Truck-Agenten über die HUB-Agenten generiert. Und zum Schluss werden alle HUB-Agenten über den Start einer neuer Spielrunde informiert. Danach verbleibt der Agent im passiven Zustand, bis er benachrichtigt wird, dass alle HUB-Agenten ihre Verhandlungen abgeschlossen haben und alle Trucks den Verhandlungsort verlassen haben.

TrucksNextRoundStartFrag.java stellt eine Übergangsphase aus einer abgeschlossenen Spielrunde zu einer neuen Spielrunde dar. Aus diesem Fragment werden alle Truck-Agenten über Start einer neuer Spielrunde informiert und zum Anmelden am Zielort (Ziel-HUB) aufgefordert.

InformHUBSimulationStartMH.java stellt in seinem Ganzem ein Mechanismus zum Versenden der synchronen Nachrichten mittels Petrinetzen und asynchronen Nachrichten dar (s. Abb.6.5). Das aufrufende Fragment (in diesem Fall *SimulationStartFrag.java*) wird nach dem versenden einer internen Nachricht „*msg*“ (s. Listing A7 1), die *InformHUBSimulationStartMH.java* zum Ausführen bringt, in Zustand „*passives Warten*“ versetzt und erst nach dem Terminieren des Handlers *InformHUBSimulationStartMH.java* aus diesem Zustand raus genommen.

Listing A7 1: SimulationStartFrag.java

```
IInternalMessageToSend msg = createInternalStatement(
        EnvironmentCF.SUBJ_INFORM_HUB_START_SIMULATION,
        InformHUBSimulationStartMH.PERF_INFORM_SIMULATION_START);
startNested(msg);
```

Während das aufrufende Fragment unterbrochen wird, sendet der aufgerufene Handler eine asynchrone Nachricht aus *SendMsgToHUBsFragment.java* an alle HUB-Agenten. Mit der Nachricht wird einer Liste mit allen HUB-Agenten der Simulation versendet. Die Liste enthält *HUB.java* Objekte, die aus der DB geholt werden. Der Zweck dieser Aktion ist, jedem HUB seinen Nachbarn bekannt zu machen, als eine Art Nachbildung eines Directory Facilitators. Nach dem Versenden der Nachricht bleibt Handler passiv, bis er alle Bestätigungen im *ReceivedAcksFromHUBsFragment.java* erhalten hat.

***CreateNewPacketsMH*.java** wird zum synchronen Generieren der Pakete im System benutzt. Er ruft *GenerateNewPacketsFrag.java* auf, das die Pakete für die nächste Spielrunde generiert und in die DB persistent schreibt. Aus Synchronisationsgründen verfügt das Fragment über einen Timer, weil die DB nicht immer in der Lage ist, die Ausführungsgeschwindigkeit der Agentenanwendung mit zu halten.

SimulationActivityMH.java wird durch das Eintreffen der Nachricht eines Manager Clients aktiv. Er verfolgt die Anweisungen eines Managers. Seine Aufgabe ist eine Zustandsänderung der Simulation (run/stop) registrieren und über jede Veränderung dieses Zustandes dem Manager melden. In *SimActivityFragment.java* werden die Nachrichten empfangen und der Variable *DS_SIM_STOP* einen neuen Wert zugewiesen. Nachdem sendet der Handler eine Bestätigung zurück, dass die Zustandsänderung registriert wurde. *SendStopActivityMsgFragment.java* informiert einen Manager, dass die Simulation angehalten wurde.

ReceivedAcksFromHUBsMH.java stellt ein Synchronisierungsmechanismus dar. Seine Aufgabe ist zu signalisieren, wann die Spielrunde beendet werden darf. Er empfängt die Nachrichten von den HUB-Agenten, die dem Environment über das Ende ihrer Verhandlungen mitteilen. Im *ReceivedAcksFromHUBsFragment.java* wird geprüft, ob alle HUB-Agenten die Verhandlung abgeschlossen haben. Ist es der Fall, so wird entweder das Synchronisieren der Spielrunde gestartet oder die Simulation wird angehalten, wenn es von dem Manager Client aufgefordert wurde. **Damit erfolgt das Anhalten Agentensimulation erst nach dem Abschluss aller Verhandlungen.** Die Spielrunde Synchronisierung erfolgt durch Versenden einer Nachricht an alle Simulationsteilnehmer mit einer neuen Spielrunde. Während dieser Synchronisierung bleibt der Environment-Agenten im Zustand „passives Warten“.

ReceiveAckToRoundSynchronizationMH.java empfängt von Truck- und HUB-Agenten die Bestätigungen einer Rundensynchronisirung. Wenn alle Agenten eine Antwort gesendet haben, sendet er aus ReceivedGameRoundSynchronizationAcksFragment.java eine interne Nachricht, um eine neue Spielrunde zu starten.

TruckDistributionMH.java generiert synchron im *GenerateTruckDistributionFragment.java* eine Verteilung der Truck-Agenten über die HUB-Agenten (s. Abbildung A7 1). Das Verteilen erfolgt iterativ über eine Sequenz der HUB-Agenten. Der Simulationsparameter "trucks_number_limit" bestimmt die Truckmenge im System. Zu unterscheiden sind drei nummerischen Werten:

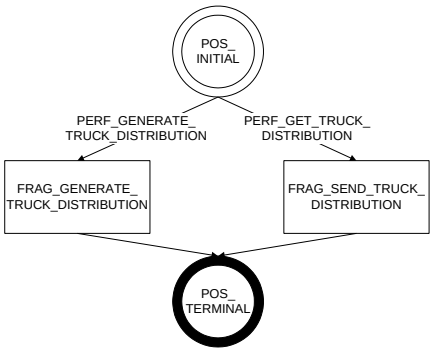


Abbildung A7 1: TruckDistributionMH

1. „*trucks_number_limit"* < 0. Es wird genau vorgegebene im Parameter Menge der Trucks generiert.
2. „trucks_number_limit" = 0. Die Menge der Trucks hängt von der Menge der Paketen in einem HUB (Gesamtladekapazität der Trucks pro HUB = Paketmenge pro HUB).
3. „trucks_number_limit" > 0. Die Menge der Trucks überschreitet nicht das Limit im System und die Paketmenge eines HUB-Agenten.

Nachdem die Verteilung generiert ist, wird der Handler terminieren. Er wird allerdings wieder zum Ausführen gebracht, wenn die HUB-Agenten eine Anfrage senden, um ihnen zugeteilte Trucks ID's zu bekommen. Das passiert in *SendTruckDistributionFragment.java.*

HUB-Agenten

CreateHUBWithInitialMsgMH.java wird während der Instanziierung eines HUB-Agenten aufgerufen, um Agenten DS zu initialisieren.

SimulationStartMH.java initialisiert im Fragment ReceivedSimStartFragment.java eine Liste mit HUB-Agenten, die von dem Environment gesendet wurde.

CreateNewTruckMH.java holt aus *GetTruckDistributionFragment.java* von dem Environment-Agenten eine Liste mit Trucks ID's. Danach erzeugt der HUB-Agent die Truck-Agenten. Die Trucks werden iterativ aus *CreateNewTruckFragment.java* kreiert. Nach der Erzeugung eines Trucks wird er in einer Liste DS_TRUCKS2LOGIN angemeldet. Diese Liste repräsentiert eine Reservierung eines Platzes im Fuhrpark. Erst nach der Agentenkreierung und seiner Anmeldung an einem HUB, wird der nächster Truck erzeugt, bis die Liste mit Trucks ID's leer ist.

LoginTruckMH.java registriert die Truck-Agenten. Das Registrieren wird im *LoginTruckFragment.java* durch das Eintragen einen Truck in die Liste DS_TRUCKS und Entfernen aus der Liste DS_TRUCKS2LOGIN implementiert. Während der Registrierung wird der Truck entladen und die entladenen Pakete werden aussortiert. Pakete, die ihren Zielort erreicht haben werden in die DB persistent geschrieben und aus dem System entfernt. Die Anderen (Transit Pakete), die nur auf der Durchreise sind, werden zwischengelagert. Nach der erfolgreichen Registrierung erhält der Truck eine Bestätigung und wenn alle erwarteten Trucks registriert sind, sendet der Handler ein „READY" an *WaitToTradingStartMH.java* und informiert den HUB, dass die Trucks zum Verhandeln bereit sind. Im Fall, wenn der Truck erst erzeugt wurde, wird eine Bestätigung an *CreateNewTruckMH.java* gesendet.

Tabelle A7 2: HUB Data Storage

Data Storage	Beschreibung
DS_UNRECEIVED_TRUCK_LIST_FOR_OFFER	Eine Liste mit Trucks ID's, die ein Angebot an HUB-Agenten senden sollen.
DS_PACKETS	Hier wird die aktuelle Paketmenge eines HUB-Agenten gespeichert. Die Liste repräsentiert eine Lagerfläche.
DS_TRUCKS	In dieser Liste werden die angemeldeten Truck-Agenten gespeichert. Sie stellt einen Fuhrpark dar.
DS_RESERVATION	Eine „Key – Value" Liste. In dieser Liste wird die reservierte Lagerfläche gespeichert.
DS_ROUND	Aktuelle Spielrunde.
DS_TRADE_ROUND	Aktuelle Verhandlungsrunde.
DS_HUBS	Eine Liste mit allen HUB-Agenten, die an der Simulation teilnehmen.
DS_ROUTES	Eine Routenliste.
DS_MY_HUB	Hier werden eigene Eigenschaften eines HUB-Agenten wie Name, geografische Position, maximal Input Kapazität … gespeichert.
DS_SUM_TRUCK_CAPACITY	Gesamte Ladekapazität.
DS_CAPACITY	Lagerkapazität eines HUB-Agenten.
DS_TRUCKS2LOGOUT	Eine Liste mit Truck-Agenten, die am Ende der Verhandlung den aktuellen Standort (HUB) verlassen sollen.
DS_TRUCKS2LOGIN	Eine Liste mit Truck-Agenten, die am Anfang der Spielrunde erwartet werden.
DS_FIRST_PACKET_I	Temporäre Variable. Sie speichert vorläufigen Index der bearbeiteten Pakete während der Gebotsgenerierung.
DS_INTERVAL_PACKET_N	Paketmenge, die während Gebotsgenerierung eines Paketes in einem „step" verarbeitet darf.
DS_TIME_INTERVAL	Der Parameter bestimmt ein Zeitintervall zum Prüfen, ob der Truck sich gemeldet hat. Der Timer wird auf 10000 ms gestellt.
DS_TIME_2_WAIT	Der Parameter bestimmt die Zeit, wie lange es auf einen verlorenen Truck-Agenten gewartet soll. Der Wert wird auf 60000 ms = 1 min gestellt.
DS_TIME	Temporäre Variable speichert die vergangene Zeit während des Wartens.

LogoutTrucksMH.java sendet aus *SendMsgToTrucksFragment.java* an einen Truck eine Anweisung zum Abmelden von dem HUB und erhält eine Bestätigung im *ReceivedAcksFromTruckFragment.java* zurück. Der Handler blockiert die Ausführung eines HUB Agenten bis alle Trucks aus der Liste *DS_TRUCKS2LOGOUT* abgemeldet sind.

TradingMH.java steuert die Agentenaktivitäten in einer Verhandlungsrunde. Zum seinen Aufgaben gehört das Generieren der Gebote eines Paketes, das Informieren eines Trucks über das Start der Verhandlung und das Verhandeln. Die Gebote werden im *MakePacketsBidsFrag.java*

generiert. Das Generieren der Gebote für die aktuell vorhandenen Pakete im HUB wurde aufgrund der LS/TS Restriktion aufgeteilt. Die Angebote werden immer in kleinen Mengen erzeugt. Fürs Generieren wird ein rekursiver Einsatz benutzt. Bei dem Kreieren der Gebote wird ein Simulationsparameter „ALTERNATIVES“ eingesetzt. Der Parameter regelt die Anzahl der besten Routen, die ein aktives Potenzial von „ALTERNATIVES“ ersten Routen mit maximalem Nutzen bilden. Nach dem die Gebote erzeugt sind, werden die Trucks über Start der Verhandlung informiert und zum Generieren der Angebote aufgefordert. Aus *InformTruckTradeRoundFrag.java* wird der Handler *InformTruckTradeRoundMH.java* synchron aufgerufen. Nach dem Empfang der Angebote startet der Handler eine Verhandlung (*MatchingFrag.java*).

TradeAckMH.java wird synchron aus *MatchingFrag.java* aufgerufen, um eine Reservierungsanfrage über *SendTradeAckToTruckFragment.java* an einen Truck zu senden und über *ReceivedAcksFromTruckFragment.java* die Antwort zu empfangen.

LoadsPacketsMH.java wird synchron aus *MatchingFrag.java* aufgerufen. Aus dem Fragment *LoadPacketsFragment.java* sendet der Handler einem Truck seine Paketladung.

InformTruckTradeRoundMH.java sendet aus *SendMsgToTruckFrag.java* an alle Truck-Agenten eine Liste mit Routen eines HUBs und empfängt über *ReceivedOfferFromTruckFrag.java* ein Angebot, der einschließlich persistent geschrieben wird. Der Handler terminiert nur dann, wenn alle Trucks ihre Angebote gesendet haben.

HUBReservationMH.java empfängt eine Reservierungsanfrage für eine Paketmenge von einem Truck-Agenten. Im *ReservationFragment.java* wird geprüft, ob der HUB genug freie Lagerkapazität für diese Menge zur Verfügung hat. Wenn eine Reservierung möglich ist, wird der Truck in die Liste DS_TRUCKS2LOGIN eingetragen und der Lagerplatz wird für diese Paketladung reserviert. Die Reservierung scheitert, wenn ein HUB keine feie Lagerfläche hat. In beiden Fällen sendet der HUB eine Nachricht an einen Truck zurück mit einer positiven oder einer negativen Antwort.

WaitToTradingStartMH.java wird zum Synchronisationszwecken zwischen HUBs und Trucks eingesetzt. Er sammelt die „READY“ Nachrichten von einem HUB-Agenten und allen erwarteten Truck-Agenten und startet die Verhandlungen aus *AllReadyFragmen.java*, wenn HUBs und Trucks zum Verhandeln bereit sind.

Truck-Agenten

CreateTruckWithInitialMsgMH.java wird zum DS-Initialisierung aufgerufen. Nach dem Ausführen dieses Handlers erfolgt eine Registrierung (Einloggen) in einem HUB.

LoadPacketsMH.java Der Truck wird vor dem abfahrt mit Paketen beladen.

MackeOfferMH.java generiert ein Angebot, der mit Hilfe eines strategischen Einsatzes aus einer Routenliste erzeugt wird. Die Routen werden von einem HUB an Truck-Agenten gesendet.

LogoutMH.java beendet das Lebenszyklus eines Truck-Agenten. Er speichert in DS Informationen, die semantisch bedeuten, dass der Truck über DS_TEMP_ROUTE zu DS_DEST_LOCATION, beladen mit den Paketen, fährt und seine Fahrt bis Spielrunde DS_NEXT_LOGIN_IN_ROUND dauert.

Tabelle A7 3: Truck Data Storage

Data Storage	Beschreibung
DS_CREATED	Der Parameter speichert eine Spielrunde, in der der Agent kreiert wurde.
DS_AGE	Alter der Truck-Agenten.
DS_MAX_TRUCK_CAPACITY	Maximale Ladekapazität eines Trucks ohne Anhänger.
DS_MAX_TRAILER_CAPACITY	Maximale Ladekapazität eines Anhängers.
DS_COST_PER_KM_WITHOUT_TRAILER	Fixkosten eines Trucks ohne Anhänger.
DS_COST_PER_KM_WITH_TRAILER	Fixkosten eines Anhängers.
DS_AVG_SPEED	Die Geschwindigkeit eines Trucks.
DS_FIXED_COST_PER_HOUR	Fixkosten eines Trucks pro Stunde.
DS_ROUND	Aktuelle Spielrunde.
DS_TRADE_ROUND	Aktuelle Verhandlungsrunde.
DS_BASE_LOCATION	Ursprungsort eines Trucks (HUB in dem der Truck erzeugt wurde).
DS_ORIG_LOCATION	HUB, in dem der Truck zu dieser Zeit angemeldet ist.
DS_DEST_LOCATION	Zielort (HUB) eines Trucks.
DS_TEMP_ROUTE	Die Variable speichert aktuelle Route, die von dem Truck befahren wird. Der Variablenwert wird in jeder Spielrunde geändert.
DS_TEMP_OFFER	Die Variable speichert einen Angebot für die Länge eier Spielrunde.
DS_ALLOWED_UNPROFIT_ROUNDS	Anzahl der Spielrunden, die ein Truck ohne Gewinn von HUB zu HUB fahren darf.
DS_ROUTE_HISTORY	Liste der besuchten HUBs.
DS_SUCCESSFUL_OFFERS	Speichert Angebote, für die eine Lagerfläche in einem HUB reserviert wurde.
DS_PACKETS	Die Variable repräsentiert ein Laderaum eines Truck-Agenten.
DS_NEXT_LOGIN_IN_ROUND	Hier wird eine Spielrunde speichert, in der ein Truck sich anmelden soll.
DS_STRATEGY	Die Variable speichert eine Strategie eines Trucks.
DS_TRUCK	Die Variable speichert einen Bean-Objekt *Truck.java*, das internen Zustand eines Agenten widerspiegelt.

NextRoundStartedMH.java wir eine Nachricht von einem Environment empfangen. Diese Nachricht informiert einem Truck-Agenten über Start einer neuer Spielrunde. Wenn die Fahrt zu Ende ist (die aktuelle Spielrunde gleich DS_NEXT_LOGIN_IN_ROUND) meldet der Truck an seinem Zielort an.

TruckReservationMH.java empfängt im *ReqeustFreeSpaceFragment.java* von einem HUB eine Nachricht, die einen Truck-Agenten das Reservieren einer Ladefläche in einem anderen HUB beauftragt. Der Agent sendet im Auftrag dieses HUB-Agenten eine Reservierungsanfrage an einen HUB, der die Pakete eventuell annehmen kann. Unabhängig von dem Ausgang der Reservierung sendet der Truck aus *AnswerFragment.java* eine Reservierungsantwort an den Verhandlungsmediator.

A8 Änderungen in der Agentenanwendung

Der Manager Agent wurde als ein LS/TS Client realisiert, der eine Simulation startet, erzeugt die Agenten, verteilt die XML Parameter an alle Simulationsteilnehmer, erzeugt die XML Datei mit Simulationsparametern und hält die gesamte Simulation an oder setzt ihre Ausführung weiter vor. Der Client erlaubt das Starten der Simulation mit und ohne Unterstützung eines Steuerungstools, mit und ohne das Versenden der Simulationsparameter, mit und ohne Erzeugung der Agenten.

Die Dateien *agent_distirbution.xml* und die *xml_configuration.xml* werden mit Hilfe eines XML Binder JAXB generiert. Beide XML wohlgeformte Dateien besitzen ein Schema. Im XML Deskriptor *agent_distirbution.xml* kann eine manuelle oder eine automatische Verteilung der Agenten eingestellt.

Die Simulationsparameter werden an alle LS/TS Plattformen einer Föderation gesendet. Auf der Seite der Agentenplattform werden die Parameter von einem Initializer Agenten entgegengenommen und in die XML Datei *xml_configuration.xml*, die vorher manuell in das Verzeichnis LSTS_HOME/re-business/bin/ angelegt werden soll, gespeichert.

Die Simulationsparameter werden automatisch erzeugt, manuell per Hand und können manuell oder mit einem Steuerungstool modifiziert werden. Die Struktur der XML Datei wurde verändert. Zum generieren einer XML Datei wird der original Quellcode ein Parsers benutzt mit einer späterer Konvertierung des Inhaltes der Datei.

Die Schnittstelle *DBService.java* wurde von der Agentensimulation abgelöst. Diese Schnittstelle wird von dem Manager Client und Analyse Tool benutz. Die Simulation bekam Hibernet Schnittstelle zwischen der Anwendung DB und dem MAS. Die DB wurde mit Hilfe eines Hibernet Konvertierungstool an LS/TS angepasst (die benötigte JavaBean wurden generiert). Es wurde das Reverse Engineering eingesetzt um DB nicht ändern. Fürs Anfrage wird HQL benutz. In einem Servant Agent erfolgt eine Trennung zwischen Agenten Instruktionen und DB Anfragen.

Die Agentensimulation verläuft sowohl lokal als auch verteilt auf mehreren Rechnern. Der Client ist eben so unabhängig von dem System, wie die DB und Analyse Tool. Das Konfiguration Tool des Steuerungssystems ist auf Manager Client angewiesen und soll auf demselben Rechner ausgeführt werden, wo der Client gestartet wird.

Dynamisches update der Agenten Simulationsparameter und Web- Service wurden nicht realisiert.

Simulationsteilnehmer: Manager Client, Initialisier-, Environment-, Truck-, HUB-, DAO-, Servant Agent und Paket-Objekt. Das interne Modell der Simulation wurde an Whitestein angepasst:

- Das Generieren der Truck-Agenten verläuft nicht synchron und nicht auf der Seite des HUB-Agenten zu Laufzeit, sondern auf der Seite eines Environment-Agenten vor dem Start der Simulation. Das Verfahren benutzt aber die gleich iterative Ressourcenverteilung wie der Original.

- Das Kreieren der Gebote wird in *N* Schritten aufgeteilt um das System zu entlasten.

- Das Matchverfahren wurde zu einem Zustandsautomat auf Grund der LS/TS Konventionen überarbeitet.

- Synchronisierungsregeln der Simulation wurden neu entwickelt.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw. §24(4) bzw. §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt haben.

Hamburg, 31. August 2009

Ort, Datum

Unterschrift (Vitalij Freese)