Vernetzung von einem Multi-Agenten-System

und

Visualisierung der Kommunikation im Multi-Agenten-System

Bachelorabschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Engineering

Reg.-Nr.: T03/03/SS2006

Technische Fachhochschule Wildau

Fachbereich Ingenieurwesen/Wirtschaftsingenieurwesen

Studiengang Telematik

Tag der Abgabe:	24. August 2006
Betreuer:	Prof. Dr. Michael Syrjakow
Themensteller:	TFH Wildau, Prof. Dr. Michael Syrjakow

Bibliografische Beschreibung und Referat

Dassow, Stefan

Vernetzung von einem Multi-Agenten-System und Visualisierung der Kommunikation im Multi-Agenten-System

Bachelorabschlussarbeit, Technische Fachhochschule Wildau 2006, 47 Seiten, 9 Abbildungen, 3 Anlagen, 1 Beilage

Ziel:

Weiterentwicklung einer verteilten JAVA basierten Anwendung aus dem Bereich „künstliches Leben“ und Einsatz der Anwendung zur Durchführung von Simulationsexperimenten.

Inhalt:

Erarbeitung einer Netzwerkstruktur zur Schaffung gut vermaschter Simulationsnetzwerke.

Aufbau einer Client-Server Architektur zur Steuerung der Simulationen von einer zentralen Stelle.

Modellierung eines Datenbankmodells zur strukturierten Speicherung von Simulationsdaten.

Auswertung von Simulationsexperimenten zur Ermittlung einer Simulationskonfiguration, die einen möglichst lange Simulationsdauer ermöglicht.

Erstellung einer grafischen Oberfläche, die die Kommunikation der Simulationsbewohner visualisiert.

Hiermit versichere ich, dass ich diese Bachelorabschlussarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Hohen Neuendorf, den 23. August 2006

_____

Stefan Dassow

Inhaltsverzeichnis

Stichwortverzeichnis

Kapitel 1 – Einleitung

Die vorliegende Bachelorabschlussarbeit beschäftigt sich mit dem Themenbereich der Multi-Agenten-Systeme (MAS), der aus der heutigen Softwareentwicklung nicht mehr wegzudenken ist. Multi-Agenten-Systeme sind am bekanntesten aus den zahlreichen Computerspielen, in denen jeder Computergegner einen Agenten verkörpert. Durch ihre autonome Arbeitsweise können mit MAS komplexe Architekturen geschaffen werden.

Komplexe Softwaresysteme werden heute auf mehrere Rechnersysteme verteilt. Die Verteilung der Software auf verschiedene Rechner erfordert eine Kommunikation der einzelnen Teile über ein Netzwerk. Im Laufe der Entwicklung der Informatik entwickelten sich verschiedene Techniken zur Kommunikation über ein Netzwerk. Konnte man in der Anfangszeit nur Daten seriell über eine Leitung schicken, ermöglichen moderne Standards die parallele Steuerung von Methoden (Funktionen) auf einem anderen Computer. Ein wichtiges Ziel dieser Arbeit ist es, ein Multi-Agenten-System weiterzuentwickeln, das künstliches Leben simuliert. Die Simulation ist so aufgebaut, dass mehrere Teilsysteme auf unterschiedlichen Computern verteilt sein können. Die Kommunikation zwischen diesen Teilsystemen ist aber nur rudimentär implementiert. Mit dieser Arbeit sollen die Netzwerkeigenschaften so verändert werden, dass von einer zentralen Stelle alle Teile der Simulation parallel gesteuert werden können. Dazu wird nicht nur der Datenaustausch neu organisiert, sondern auch der Aufruf entfernter Funktionen implementiert.

Die Simulation dient durch Einstellmöglichkeiten vieler Parameter dem Experimentieren mit komplexen Systemen. Zum Experimentieren gehört auch eine ausführliche Dokumentation der Experimente. Die Daten, die bei der Simulation anfallen, sind durch die Speicherung in Textdateien schlecht zu verarbeiten. Zur vereinfachten Verarbeitung der Daten wird im Rahmen dieser Arbeit eine strukturierte Datenspeicherung installiert. Die Daten werden am Ende in einer Datenbank gespeichert, damit sie komfortabel abrufbar sind. Mit Simulationsexperimenten wird in dieser Arbeit ermittelt, an welchen Parametern man Einstellungen vornehmen kann, die ein frühes Ende einer Simulation zu verhindern.

Das Multi-Agenten-System zeichnet sich durch die Kommunikation unter den Agenten aus, die in der Grundversion für den Nutzer aber nicht sichtbar ist. Die Visualisierung dieser Kommunikation wird im Zuge dieser Arbeit für den Nutzer ermöglicht. Sie ist für das Experimentieren eine Arbeitserleichterung, da sie die Vorgänge im Hintergrund für den Nutzer verständlich macht. Die Simulation besitzt einen gewissen Unterhaltungswert. Dieser soll mit dieser Arbeit erhöht werden, indem bei der Visualisierung der Kommunikation Texte verwendet werden, die einen gewissen Spass-Faktor für Jugendliche darstellen. Die Texte werden am Ende aber einfach für verschiedene Nutzergruppen anzupassen sein.

Die vorliegende Arbeit beschreibt in Kapitel 2 die Grundlagen für diese Arbeit vorgestellt. Im dritten Kapitel werden die verschiedenen Konzepte zur Bearbeitung der Aufgabenstellung erläutert. Dazu gehört das Konzept zur Erstellung des Client-Server-Systems, die Ideen zur Multimedia-Erweiterung der Simulation und die Ideen für die Datenspeicherung. Weiterhin befasst sich die

Arbeit in Kapitel 4 mit der Überarbeitung der bisherigen Simulation und der Realisierung der einzelnen Erweiterungen. Bei den Erweiterungen wird zunächst der Schwerpunkt auf die Neuentwicklungen im Bereich des Netzwerkes gelegt. Danach folgt die Beschreibung des neu entwickelten Servers und die Visualisierung der Kommunikation. Im fünften Kapitel werden abschließend die Ergebnisse dieser Arbeit zusammengefasst. Sie enthalten unter anderem Hinweise zur Bedienung des Servers und die Erkenntnisse aus den Simulationsexperimenten. Zum Schluss werden Möglichkeiten für die weitere Ausbaufähigkeit der Simulation aufgezeigt.

Kapitel 2 – Grundlagen

2.1. JAVA

JAVA[2] setzt sich zusammen aus der gleichnamigen Programmiersprache, der JAVA Virtual Machine und der JAVA Plattform. Bei der Programmiersprache handelt es sich um die Syntax und Semantik, in der Java-Programme geschrieben werden. Bei der Kompilierung des Quelltextes entsteht sogenannter Bytecode, welcher nicht maschinenlesbar ist. Da kommt die JAVA Virtual Machine (JVM) zum Einsatz. Sie führt den generierten Bytecode aus. Die JVM kann sowohl Hardware als auch Software sein, wobei letzterer Fall häufiger vorkommt. Die JAVA Plattform umfasst eine vordefinierte Menge an Klassen[1] .

Bei der Programmiersprache handelt es sich um eine moderne, objektorientierte Hochsprache, welche eine der Programmiersprache "C" ähnliche Syntax aufweist. Die Sprache konnte sich seit Anfang der neunziger Jahre schnell etablieren und zählt heute zu einer der populärsten Programmiersprachen, wenn es um Anwendungen für das Internet geht. JAVA wird gerne als plattformunabhängig bezeichnet. Dies ist insofern richtig, dass nur für solche Betriebssysteme, für die es eine JAVA Virtual Machine gibt, diese Unabhängigkeit gilt. Alle Klassen der jeweiligen JAVA-Plattform können in jeder JVM ausgeführt werden, die die Spezifikationen der JAVA-Plattform erfüllt. Die Portablität führt zu der Möglichkeit der zentralen Verwaltung der Applikationen in einem Netzwerk.

Durch die Ausführung des Byte-Code in der JAVA Virtual Machine entsteht ein hoher Sicherheitsstandard. Alle Zugriffe auf die Hardware des Zielsystems erfolgt durch die JVM, wodurch die Ausführung von korrupten oder defekten Codes so gut wie unmöglich ist. Bei Applets wird das Zielsystem sogar gegen sämtliche Lese-/Schreibaktionen durch eine JAVA-Applikation geschützt (Sand-Box-Prinzip). Dieser Mechanismus sichert zwar das Zielsystem, schränkt aber die Applikation in ihrem Handlungsspielraum ein.

Implementierungen von JAVA gibt es heute für fast alle erdenklichen Rechnerplattformen. Sun, die Entwickler von JAVA, stellen JVM's z.B. für Desktops, Server und Mobiltelefone bereit. Weiterhin existieren auch noch Implementierungen anderer Unternehmen. Den heute hohen Verbreitungsgrad verdankt JAVA aber nicht zuletzt auch der kostenlosen Bereitstellung des JAVA Development Kit, das sowohl eine Referenzimplementierung der Sprache, als auch die Klassenbibliothek beinhaltet.

Wie oben erwähnt handelt es sich bei der Sprache JAVA um eine objektorientierte Programmiersprache. Sie fasst Attribute und Funktionen von Objekten in Klassen zusammen. In JAVA werden die Funktionen einer Klasse als Methoden bezeichnet und Attribute als Datenfelder benannt. Aus Klassen werden dann die Objekte generiert. Die Objekte sind Klassen, die mit konkreten Werten für die Datenfelder initialisiert wurden. Durch die Methoden ist ein Zugriff auf die Objekte gewährleistet. Die Plattform ist in einer Baumstruktur aufgebaut. Das Wurzelelement ist die

1 Klasse: "Spezifizierung der Gemeinsamkeiten einer Menge von Objekten" (PSE3-14) [9]

„Object“-Klasse. Alle weiteren Klassen sind von genau einer Klasse abgeleitet, welche über einen längeren oder kürzeren Weg aber alle von „Object“ abgeleitet sind. JAVA unterstützt keine Mehrfachvererbung. Neben Klassen spezifiziert die Sprache auch „Interfaces“. Diese Art an Konstrukten definiert Schnittstellen, welche nur die Methodensignaturen definiert, aber selbst keinen Code enthält. Ein Interface kann auch von einer Klasse oder einem Interface abgeleitet sein, wodurch sich über Umwege eine Mehrfachvererbung realisieren ließe, da eine Klasse mehrere Interfaces implementieren kann. Wenn eine Klasse ein Interface implementiert, muss sie die Gesamtheit der Methoden des Interface implementieren. Um den Baum der Plattform und/oder der Applikation zu strukturieren, kann man Interfaces und Klassen in Packages zusammenfassen.

In anderen Hochsprachen, wie zum Beispiel „C“ muss sich der Programmierer selbst um die Speicherverwaltung kümmern. Dies birgt Risiken für Pufferüberläufe oder andere Fehler, welche zu defekten oder korrupten Codes führen können. In JAVA wird die Reservierung von Speicher durch die JAVA Virtual Machine vorgenommen. Wird ein Objekt nicht mehr verwendet, wird durch den Vorgang der „Garbage Collection“ der Speicherbereich wieder freigegeben. Dieser Prozess der Freigabe des Speichers wird nebenläufig durch die JVM organisiert. Ein Objekt wird immer dann von der „Garbage Collection“ zerstört, wenn im ganzen Programm keine Referenz mehr auf dieses Objekt existiert. Die Unterbindung von Bugs durch Speicherfehler wiegt den Performanceverlust, der durch den Prozess entsteht, auf.

Der nebenläufige Ablauf von Programmteilen wird durch die Klasse Thread unterstützt. Man kann also bestimmte Abläufe in verschiedene Threads packen. Monitore gewährleisten den exklusiven Zugriff auf Daten in den Threads.

Persistenz von Objekten wird in JAVA durch das Interface „Serializable“ unterstützt. Die Einbindung dieses Interfaces signalisiert der JVM, dass ein Objekt linearisierbar ist. Das Interface enthält keine Methodensignaturen. Eine andere Art und Weise der persistenten Speicherung von Objekten ist die Speicherung von Daten in eine Datenbank. Frameworks wie „Hibernate“ oder „Torque“ kümmern sich nach gewissen Einstellungen um die Verwaltung der Objekte in der Datenbank.

Die Plattform stellt eine Vielzahl an Paketen/Klassen schon von Hause aus zur Verfügung. Die Klassenbibliothek enthält Containerklassen (z.B. Vektoren, Listen), die sich zur Verwaltung von Daten eignen. Wichtig sind auch die Pakete zur grafischen Darstellung. Sie stellen Zeichnungen, Bilder usw. auf allen Plattformen annähernd gleich dar. Auch findet man in verschiedenen Paketen Unterstützung für die verschiedensten Formen der Netzwerkkommunikation. Zu den einfachsten Möglichkeiten zählt das Versenden mittels Streams (Datenströme) über Sockets. Objektbasierte Middleware lässt sich sowohl durch „Remote Method Invocation“ als auch durch die „Common Object Request Broker Architecture“ realisieren. Dienst-orientierte Architekturen können mittels „JAVA Intelligent Network Infrastructure“ erzeugt werden. Durch die gezielte Ausrichtung von JAVA auf Netzwerke ist die Arbeit mit Netzwerkkomponenten sehr komfortabel und einfach.

2.2. Simulation

Unter einer Simulation[3] versteht man eine Nachbildung, bei der nicht das reale System selbst, sondern ersatzweise das Modell des Systems untersucht wird. Gründe für eine Simulation können sein, dass eine Untersuchung am realen System zu teuer oder zu aufwändig wäre. Auch spielt die Dauer eine entscheidene Rolle. Wie im vorliegenden Beispiel hat man nicht die Zeit, sich einen gesamten Lebenszyklus anzuschauen, um danach dann die Ergebnisse auszuwerten. Weiterhin ist es in einer Simulation einfacher möglich, Modifizierungen vorzunehmen, als an einem realen System.

Die Simulation definiert sich als Reproduktion des dynamischen und/oder statischen Verhaltens eines realen Systems. Die Reproduktion geschieht auf Grundlage eines Abbildes der Realität, welches als Modell bezeichnet wird. Das Modell umfasst alle Eigenschaften und Funktionen des realen Systems, welche für zu erreichende Erkenntnisgewinnung von Interesse sind. Das Modell muss so geschaffen sein, dass aus den Simulationsergebnissen Rückschlüsse auf das reale System getätigt werden können. Das Modell muss validierbar sein, um optimale Ergebnisse in der Simulation zu liefern.

Laut Möller[3] sind Simulationen nicht als Experiment zu verstehen, da sie nicht Gegenstand der Untersuchung und nicht unabhängig vom Messverfahren sind. Andererseits sieht Troitzsch[8] die Simulation als eine Art Experimentieren mit Modellen. Er bezieht sich dabei darauf, dass man aus bestimmten Anfangsbedingungen und Parameterkombinationen Schlussfolgerungen aus dem Modellergebnis für das reale System ziehen kann. Was passiert aber, wenn das Modell von der Realität nicht mehr zu unterscheiden ist? Die Simulation wird zwangsweise ein Experiment.

Simulationen[5] können als Sichtweise sowohl *statisch* als auch *dynamisch* verstanden werden.

Bei der *statischen* Simulation erfolgt die Darstellung zu genau einem Zeitpunkt und zwar dem des Gleichgewichtes des Systems. Man spricht auch davon, dass das System/Modell unabhängig von der Zeit ist.

Bei der *dynamischen* Simulation wird der Verlauf der Veränderungen in Bezug auf die fortschreitende Zeit dargestellt. Die Veränderungen resultieren aus den Aktivitäten des Systems. Unterscheidet man Simulationen nach der Verarbeitung der Ausgangswerte, lassen sich Simulationen in *deterministische* und *stochastische* Simulationen einteilen.

Deterministische Simulationen haben eindeutig bestimmte Systemvariablen. Keine Systemvariable ist ohne Wert. Für jeden Eingangswert existiert ein bestimmter Ausgangwert. Im Gegensatz dazu müssen in einer *stochastischen* Simulation nicht alle Systemvariablen/-werte diesen Anforderungen genügen. Durch Einfluss des Zufalls existieren für jeden Input eine Menge unterschiedlicher Outputs. In sich abgeschlossene Simulationen werden als *endogen* bezeichnet. Eine Beeinflussung von außen findet nicht statt. Davon abweichend ist es bei *exogenen* Simulationen der Fall, dass Variablen und Aktivitäten außerhalb des Systems/Modells stattfinden und die Simulation in ihren Ausführungen beeinflussen. Bei der Differenzierung in *kontinuierliche* und *diskrete* Systeme kommt es auf die Stetigkeit der Zustandsvariablen an. Man spricht von einem *kontinuierlichen* System,

wenn diese Variablen stetig sind. Zur Darstellung der Abhängigkeiten werden Differentialgleichungen verwendet. Bei einer *diskreten* Simulation werden Zustandsänderungen durch Aktivitäten oder Ereignisse hervorgerufen. Damit sind die Zustände im System zählbar und diskret. Auslösende Ereignisse finden zu vorher definierten Zeitpunkten oder in Zeitintervallen, also nicht stetig statt.

Troitzsch warnt: "Wer immer aber Simulationen zur Klärung [...] relevanter Fragen verwendet, sollte [...] vor allem sein Publikum darüber aufklären, dass eine Simulation immer nur eine Lösung eines Ansatzes für eine bestimmte - zu rechtfertigende - Parameterkombination [...] ist."[2]

2.3. Multi-Agenten-Systeme

Ein Agent[4] ist eine Einheit, die die Fähigkeit besitzt vom Benutzer oder von anderen Agenten gestellte Aufgaben selbstständig zu erfüllen. Er ist eine Art von Softwaresystem, welches verschiedene Charakteristika aufweist. Agenten sind zusätzlich zu ihrer Autonomie und Pro- und Reaktivtät sozial und, in gewissen Grenzen, auch lernfähig.

Ein Agent ist immer ein Teil eines größeren Software-Systems. Sind Umgebungsinformationen[7] für einen Agenten schnell und zuverlässig erreichbar, spricht man von einer *zugänglichen* Welt. Umgebungen sind in den komplexeren Systemen nicht deterministisch. Ist eine Umgebung von den eintreffenden Ereignissen unabhängig und können die Ereignisse ohne Vorkenntnisse immer wieder verarbeitet werden, so spricht man von einer *episodischen* Umgebung. Eine nicht an Aktionen und Zuständen begrenzbare Umgebung bezeichnet man als *dynamisch*. In einer *dynamischen* Umgebung entzieht sich das Verhalten dieser der vollständigen Kontrolle durch einen Agenten. Der Agent kennt seine Umwelt, was nicht immer zwingend das ganze System sein muss. Eher das Gegenteil ist der Fall. Er besitzt die Fähigkeit, seine bekannte Umgebung teilweise zu beeinflussen. Andererseits beeinflusst auch die Umgebung das Handeln des Agenten, welches sich in seiner Reaktivität niederschlägt. Durch Sensoren ist ein Agent in der Lage seine Umwelt wahrzunehmen und darauf zu reagieren. Seine Aktivitäten werden aber auch häufig Auswirkungen auf seine Umwelt haben.

Damit ein Agent überhaupt autonom arbeiten kann, benötigt er Wissen, um Probleme lösen zu können. Dieses Wissen liegt zum Beispiel in Form von Regelwerken vor. Eine andere Wissensbasis können neuronale Netze darstellen. Moderne Agenten können aus ihrem Handeln lernen, indem sie bereits aufgetretene Situationen analysieren und ihr Wissen ergänzen.

Agenten sind in der Lage, ihre Aufgaben durch ihr Wissen meist unabhängig vom Auftraggeber durchzuführen. Sie können ohne Einflussnahme von außen ihre mehr oder weniger präzise gestellten Aufgaben zu Ende bringen. Der Agent agiert hierbei proaktiv, da er nach Kenntnis der Vorgaben die Aufgabe bis zum Ergebnis zielstrebig verfolgt. Je allgemeiner die Probleme sind, die ein Agent in der Lage ist zu lösen, desto größer ist sein Handlungsspielraum und sein Verantwortungsgrad.

2 Probleme sozialwissenschaftlicher Computersimulation (S. 180) [8]

2.4. Population und Lebensraum

In ihrem Lebensraum nutzen Tiere oft nur wenige der vorhandenen Möglichkeiten, um sich zu ernähren. Das Gleiche gilt auch für die Wahl der Brut- und Nistplätze oder auch

Verstecke u.Ä.. Damit sind nicht alle Faktoren, die sich in der Umwelt eines Lebewesens befinden, für dessen Existenz von Bedeutung. Die Summe aller Faktoren, die für das Überleben einer Art von Bedeutung sind, nennt man ökologische Nische. Die Simulation beschränkt sich in ihren Ausführungen auf die Faktoren, welche der ökologischen Nische der Schwärmer(Quarries) und Agenten[3] zuzuordnen sind.

In der Simulation pflanzen sich die Schwärmer durch Teilung fort. Da dieser Vorgang periodisch abläuft, würde diese Rasse ohne Einflüsse von außen und bei Unsterblichkeit in ihrer Population exponentiell wachsen.

Beispiel:

Generation	***0***	***1***	***2***	***3***	***4***
Anzahl Schwärmer	$1=2^0$	$2=2^1$	$4=2^2$	$8=2^3$	$16=2^4$
	5	***6***	***7***	***...***	***n***
	$32=2^5$	$64=2^6$	$128=2^7$	...	2^n

Tabelle 1: Nachkommen bei exponentiellem Wachstum

3 Schwärmer, Agenten: Lebewesen in der Simulation siehe Seite 11

Durch die natürliche Sterberate wird das exponentielle Wachstum aber nur geringfügig nach unten korrigiert. Erst die Verknappung der Nahrung bringt die Sterberate auf ungefähr gleiches Niveau wie die Geburtenrate. Dieser Zustand wird Kapazität K des Lebensraumes genannt. Exponentielles Wachstum, wie bei den Schwärmern, führt zu regelmäßigen Schwingungen(siehe Abbildung 1) um

Abbildung 1: Schwingung einer Population um seine Kapazität

die Kapazität des Lebensraumes. Die Population liegt also in begrenzten Bereichen um die Kapazität. Würden die Schwankungen chaotische (unregelmäßige) Formen annehmen, könnte es zum Aussterben der Art durch zu große Amplituden kommen.

Die zweite ökologische Nische dreht sich um die Agenten. Die Dynamik der Population wird hier durch ein Räuber-Beute-System dargestellt. Hierbei bilden die Schwärmer die Beute und die Agenten sind die Räuber. Da die Agenten als Nahrungsgrundlage nur die Quarries haben, ist die Population der Agenten direkt abhängig von der Anzahl der Quarries. Wenn die Anzahl der Beutetiere zunimmt, steigt auch die Anzahl der Räuber. Die Wirkung der Zunahme an Beutetieren tritt aber immer mit einer gewissen Verspätung ein. Die Verspätung steigt, je länger die Generationsdauer bei den Räubern ist. In einem geschlossenen System mindert aber die Zunahme der Räuber die Anzahl der Beutetiere. Man spricht hierbei von einer negativen Rückwirkung. Auch hier ist eine Verzögerung zu erkennen. Ein Regelkreis kann diese Wechselwirkungen beschreiben. Wenn in diesem Regelkreis lange Verzögerungszeiten auftreten, kommt das ganze System ins Schwingen. Dabei sind die Schwingungen der Beutetiere und Räuber um die Verzögerungszeiten phasenverschoben.

Kapitel 3 – Konzept

Die vorliegende Arbeit hat als Ziel eine vorhandene Simulation eines Multi-Agenten-Systems im Bereich seiner Konfiguration und Netzwerkeigenschaften zu erweitern. Die Routen zwischen den einzelnen Welten sollen automatisch erstellt und verwaltet werden. Das Konfigurationswerkzeug soll die Eigenschaften der Agenten einstellen können. Die Steuerung der Simulation soll zentral erfolgen. Um eine verbesserte Auswertung der Gesamtsimulation zu erreichen, müssen die statistischen Daten zentral durch den Server persistent gespeichert werden. Im Folgenden sollen die Ansätze zur Lösung der Aufgabe und die Grundlagen der vorliegenden Simulation erläutert werden.

3.1. Aufbau der bisherigen Simulation

Die Simulation setzt sich aus der grafischen Oberfläche und dem Multi-Agenten-System zusammen.

Das Multi-Agenten-System modelliert eine künstliche Welt. "Die simulierte Welt besteht aus einem Torus[4] und einer Menge von Objekten, die sich auf dessen Oberfläche befinden und als Bewohner der Welt bezeichnet werden."[5] Das Abbild der Welt stellt sich für den Betrachter als Rechteck dar. Aus Sicht der Bewohner schließen sich zwei gegenüber liegende Seiten nahtlos zusammen. Die Position eines Bewohners wird durch das projizierte Rechteck bestimmt. Der Ursprung liegt dabei in

4 Torus: geometrische Form ähnlich einem Fahrradschlauch

5 Zitat: [4] Auszug aus dem Konzept - S.23

der linken oberen Ecke. Alle zu ermittelnden Entfernungen werden auch auf Basis dieses Rechteckes errechnet. Weiterhin werden die Entfernungen zweier Objekte immer zwischen ihren Mittelpunkten berechnet. Die Form eines Objektes ist dabei unerheblich und wird von dem Objekt selbst definiert. Ein Algorithmus verhindert die Mehrfachbelegung einer Position durch mehrere Objekte. Die Objekte sind generell auf der Oberfläche bewegbar. Auch lassen sich Objekte aus der Welt entfernen oder ihr hinzufügen. Die Simulation lässt einen Übertritt von Objekten von einer Welt in eine andere durch sogenannte Portale zu. So lässt sich die DNA der Agenten verschiedener Welten mischen. Die Simulation beherbergt fünf verschiedene Arten von Objekten. Zum einen sind das die schon beschriebenen Portale, Berge als Hindernisse und drei verschiedene Arten an Lebewesen. Sie heißen Nahrung, Schwärmer (Quarries) und Agenten. Die Lebewesen lassen sich innerhalb einer Nahrungskette abbilden.

Am Anfang der Nahrungskette steht die sogenannte Nahrung. Diese Nahrungsmittel bilden die Ernährungsgrundlage für die danach folgenden Schwärmer. Nahrung wächst regelmäßig nach und kann somit nicht aussterben. Nahrungsobjekte sind nicht in der Lage sich zu bewegen. Weiterhin besitzen sie keine räumliche Ausdehnung. Sie haben einen vordefinierten Energiewert, der durch Nahrungsmittelaufnahme eines Schwärmer reduziert wird. Besitzt ein Nahrungsobjekt keine Energie mehr, so wird es aus der Welt entfernt. Das Nachwachsen wird vom Benutzer durch eine festzulegende Wahrscheinlichkeit gesteuert; die Position von nachwachsenden Elemente ist zufällig.

Schwärmer haben eine Ausdehnung von eins und sind in der Lage, sich mit einer bestimmten Geschwindigkeit auf der Oberfläche zu bewegen. Der Schwärmer verbraucht für seine Aktionen Energie, die er durch Nahrungsaufnahme ausgleichen muss. Schwärmer kennen eine Form von Gesundheit. Sie kann durch Angriffe von Agenten und fehlende Nahrungsmittel abschwächen. Fällt sie auf null, so stirbt ein Schwärmer. Andererseits besitzt ein Schwärmer auch die Möglichkeit der Regeneration. Angriffe durch Agenten können Schwärmer auch lähmen, was zu einer Unbeweglichkeit dieser führt. In regelmäßigen Abständen reproduzieren sich Schwärmer durch Teilung. Die Teilung erfolgt aber mit Mutationen, so dass die Kopie nicht die gleichen Eigenschaften aufweist wie das Original.

Am Ende der Nahrungskette stehen die Agenten. Sie bilden die komplexesten Lebewesen innerhalb dieses MAS. Agenten besitzen Erbgut in Form einer DNA. Sie legt Eigenschaften wie z.B. das Geschlecht, die Ausdauer, Sichtradius und andere Eigenschaften fest. Nahrungsmittel für Agenten sind allein die Schwärmer. Zur Nahrungsaufnahme müssen diese erst einmal eingefangen werden. Jeder Agent besitzt dazu Spezialfähigkeiten. Agenten können die Schwärmer angreifen (Gesundheit schwächen, sofern sie sich nicht bewegen), lähmen oder als Leiter einer Agentengruppe agieren. Um eine effektive Jagd auf Schwärmer zu organisieren, müssen die Agenten miteinander kommunizieren. Dazu bedienen sie sich der KQML[11]. Dieses Sprachprotokoll standardisiert die Kommunikation. Die Agenten haben einen vordefinierten Wortschatz, um verständlich miteinander kommunizieren zu können. Der Wortschatz umfasst

Phrasen[4] z.B. zur Anforderung von Umgebungsinformationen, zur Jagd eines bestimmten Zieles oder dem Austritt aus einer Gruppe. Befindet sich ein Agent in der Nähe eines Portals, wechselt er mit einer definierten Wahrscheinlichkeit über das Portal die Welt. Die Wahrscheinlichkeit steigt, je schlechter die Situation eines Agenten ist. Die Situation wird durch den Nahrungspegel, den Jagderfolg und die Zugehörigkeit zu einer Gruppe definiert. Agenten verschiedenen Geschlechts können sich bei Bedarf fortpflanzen.

Die Implementierung des Multi-Agenten-Systems in einer Simulation erfolgte mittels eines Frameworks. Das als „Java Media Tool"[4] bezeichnete Framework ist eine seitenorientierte Ablaufumgebung für Multimedia-Anwendungen. Handler[6] verwalten das Medium, Aussehen und Verhalten von Seitenelementen. Zur Seitenbeschreibung wird die Auszeichnungssprache „Java Media Tool Language" verwendet, welche auf XML basiert. Das `<page>`-Element bildet die Wurzel des Beschreibungsbaumes. Einzelne Objekte werden durch einen Namen identifiziert und ihnen wird ein Layout und Verhalten zugeordnet. Das Verhalten einzelner Objekte wird durch eine eigene Skriptsprache gesteuert. Diese Skriptsprache[4] lehnt sich stark an die Syntax von JAVA und JavaScript an. Der `Binder` enthält das Hauptprogramm und liest die Seitenbeschreibung aus der JMTL-Datei ein. Die Klasse erzeugt daraus die entsprechenden Handler und hängt die sichtbaren Elemente als Container in eine Grafikumgebung ein. Nach Erzeugung der Handler erkennt das Framework Benutzereingaben, verwaltet sie autonom und leitet sie an die entsprechenden Skripte weiter.

3.2. Netzwerkschnittstellen

3.2.1 Portale

In der Grundversion werden die Agenten durch Portale[4] von einer Welt zu einer anderen transportiert. Die Portale werden bei der Initialisierung mit einem Ziel versehen, welches in einer Property-Datei[7] festgelegt ist. Sollte dieses Ziel nicht erreichbar sein, so wird das Portal als nicht funktionsfähig deklariert. Die Konfiguration einer Vielzahl von Portalen bei vielen Clients ist somit sehr arbeitsintensiv. Bei jedem Client sind die Property-Dateien anzupassen. Daher werden innerhalb dieser Arbeit die Portale neu zu entwickeln.

Zur einfacheren Konfiguration sollen die für die neue Version entwickelten Portale ihre Ziele nicht mehr kennen. Die Konfiguration der Routen von einem Portal zu einem anderen soll durch eine externe Instanz erfolgen. Damit die neuen Portale problemlos ohne größeren Aufwand in die Welt bzw. Simulation integriert werden können, benötigen sie die gleiche Signatur wie die Portale der Grundversion. Zusätzlich zu den bisherigen Methoden müssen Methoden bereitgestellt werden, die eine An-/Abmeldung einer Instanz ermöglichen, welche den Transport der Agenten übernimmt. Die Aufgabe der neuen Portale beschränkt sich auf das Einfügen/Entfernen der Agenten in/aus der Welt und die Weiterleitung dieser an die Transportschicht.

6 Handler: JAVA-Objekt

7 Property-Datei: Textdatei mit Schlüssel-Wert-Paaren im Klartext

3.2.2 Client und neuer Server

Ziel dieser Arbeit ist es, eine verteilte Simulation mehrerer Welten einfacher zu ermöglichen. Die Konfiguration und Steuerung der Simulation sollte zentral erfolgen können, dazu aber mehr im nächsten Abschnitt. Hier drängt sich geradezu eine Client-Server-Architektur auf. Die Clients sind dabei die Multi-Agenten-Systeme mit ihrer optionalen grafischen Darstellung.

JAVA stellt die verschiedensten Mechanismen bereit, um über ein Netzwerk miteinander zu kommunizieren. Die einfachste Möglichkeit ist der Aufbau von Sockets, wie bisher geschehen und das Versenden von Objekten oder Textnachrichten. Dies hat aber ein Auslesen der Nachrichten bzw. die Interpretation der Objekte zur Folge. Mit der Unterstützung von CORBA steht in JAVA ein mächtiges Werkzeug für verteilte Anwendungen zur Verfügung. Die Installation und Konfiguration von CORBA ist aber nicht trivial und für diese Anwendung einfach übertrieben. Ein andere Möglichkeit zur Netzwerkkommunikation bietet JAVA mit RMI. Mit RMI lassen sich Methoden einer entfernten Anwendung (außerhalb der eigenen JVM) aufrufen. Dazu erzeugt man „Stubs“ und „Skeletons“ von der JAVA-Klasse, welche über das Netzwerk nutzbar sein sollen. Die Stubs sind clientseitige Objekte, während die „Skeletons“ beim Server liegen. Beide Formen[12] regeln den Netzwerkverkehr auf der jeweiligen Seite. Der Client initialisiert ein „Stub“, welcher bei Aufruf einer Methode in Kontakt mit dem gleichnamigen „Skeleton“ auf der Serverseite tritt. Der Client muss den Ort des Server kennen, bevor er mit ihm in Kontakt treten kann. Nach dem Wissen des Ortes kann der Client über den Naming-Service von JAVA sein „Stub“ initialisieren. Seit JAVA 1.5 lassen sich Stub und Skeleton dynamisch zur Laufzeit erzeugen. Damit man die Technik überhaupt verwenden kann, müssen alle fernnutzbaren Klassen von der Klasse `java.rmi.Remote` abgeleitet sein. Diese Technik eignet sich sehr gut, um die Client-Server-Architektur umzusetzen.

Um das MAS auch ohne Netzwerkverbindung zu nutzen, muss die Simulation um eine Struktur erweitert werden, welche sich um die gesamte Netzwerkarbeit kümmert. Die Struktur muss in beide Richtungen kommunizieren können. Das bedeutet, dass sowohl Agenten an den Server weitergegeben, als auch vom Server angenommen werden können. Die Netzwerkschnittstelle muss auch die Steuerung des Client ermöglichen und Steuerinformationen, welche vom Client ausgehen, an den Server weiterleiten. Nicht zu vergessen ist die Einstellung der Parameter des MAS, welche vom Server durchgeführt werden soll. Weiterhin sollen die Daten, die bei der Statistik anfallen, auch an den Server zur Weiterverarbeitung übertragen werden.

Das MAS wird derzeit über die Skripte des JMT gesteuert und initialisiert. Die Einbindung der hierfür entwickelten neuen Schnittstelle zum Netzwerk kann über diese Skripte erfolgen. Die Netzwerkschnittstelle muss weiterhin absichern, dass der Server auch gefunden wird. Um das Ziel der Arbeit zu verfolgen, sollte der Server automatisch im Netzwerk gefunden, anstatt über eine Konfigurationsdatei vorgegeben werden. Hierzu ist ein Algorithmus zu entwickeln, der eine einfache Suche implementiert. Auch ist eine Eingabe der Netzwerkadresse durch den Benutzer nicht erstrebenswert, da auftretende Fehler aufwändig abgefangen werden müssen. Die Simulation ist für Präsentationszwecke gedacht, so dass man davon ausgehen kann, dass sich alle Clients und der

Server in einem kleineren Netzwerk befinden. Um einen Server ausfindig zu machen, könnte man nun über sämtliche Netzwerkadapter eines Clients eine `whois`-Anfrage an die Broadcast-Adresse[8] absetzen. In vielen Netzwerken werden Broadcast-Pakete aber abgelehnt, so dass der Suchalgorithmus sämtliche Adressen in einer Schleife durchlaufen muss. Für diese Suche empfiehlt sich ein eigenes Protokoll auf Basis von UDP. Das Protokoll soll ein Handshake[9] beinhalten und auch Informationen vom Client an den Server übertragen.

Der Server muss ähnliche Strukturen wie der Client bereitstellen, um im Netzwerk zu kommunizieren. Er muss sich wie der Client beim lokalen Naming-Service anmelden, damit er mittels RMI mit den Clients kommunizieren kann. Weiterhin muss er eine Möglichkeit bieten, auf die whois-Anfragen der Clients zu reagieren und den Handshake zwischen Client und Server durchzuführen. Da der Server als Mittler der Agenten zwischen den einzelnen Clients fungiert, muss er die Möglichkeit bereitstellen Agenten aufzunehmen und sie an die entsprechenden Stellen weiterzuleiten. Weiterhin muss der Server Funktionen bereitstellen, die den Zugang zu allen Clients sicherstellen, um sie zu steuern und Einstellungen vorzunehmen. Zu einer verbesserten Auswertung der anfallenden statistischen Daten muss der Server eine zu entwickelnde Schnittstelle zu einer Datenbank bereitstellen. Um die ohnehin sehr speicherintensive Simulation nicht noch durch Frameworks für Object-Mapping[10] aufzublähen, soll die Abspeicherung mittels JDBC-Treiber auf der einfachsten Ebene erfolgen.

3.2.3 Handshake-Protokoll

Um einen autorisierten Zugriff eines Clients auf den Server zu ermöglichen, müssen sich beide Anwendungen erst einmal miteinander bekannt machen. Dazu soll ein eigenes vereinfachtes Handshake-Verfahren durchgeführt werden. Der Handshake soll mittels Text-Nachrichten erfolgen. Die Nachrichten beginnen mit einem Auftaktwort gefolgt vom Programmnamen. Im Anschluss erfolgt ein Schlüsselwort mit einem optionalen Schlüsselwert. Hinter das Schlüsselwort kann eine Parameterliste gesetzt werden. Beendet wird eine Nachricht von einem Schlusswort.

```
<Nachricht>      ::= START<Seperator><Programmname><Seperator>
                     <Schlüssel>[<Seperator><Parameterliste>]
                     <Seperator>END
<Programmname>   ::= <Zeichenkette>
<Schlüssel>      ::= <Name>[:<Wert>]
<Parameterliste> ::= <Parameter>{<Seperator><Parameter>}
<Parameter>      ::= <Name>:<Wert>
<Name>           ::= <Zeichenkette>
<Wert>           ::= <Zeichenkette>|<Zahl>
<Zeichenkette>   ::= <Zeichen>{<Zeichen>}
<Seperator>      ::= <Zeichen>
<Zahl>           ::= digit{digit}[.|,digit{digit}[f|F]]
<Zeichen>        ::= character
```

8 Broadcast-Adresse: Netzwerkadresse (Bits der Rechneradresse sind alle auf 1 gesetzt), bei der sich alle Rechner eines Netzwerkes angesprochen fühlen

9 Handshake: Authentifizierungsverfahren

10 Object-Mapping: Abbildung von Datenstrukturen in einer Datenbank

Das Verfahren soll in der ersten Nachricht über das Schlüsselwort "NEW" eingeleitet werden. Das Schlüsselwort besitzt keinen Wert und auch keine weiteren Parameter. Der Server antwortet daraufhin mit dem Schlüsselwort „AUTH“ für Authentizität mit einem Sitzungsschlüssel als Wert. Weiterhin erhält der Client als Parameter(„CLNT“) seine eigene IP-Adresse zurückgeschickt. Im letzten Schritt des Verfahren schickt der Client seinen Sitzungsschlüssel mitsamt seinen Parametern, die für den reibungslosen Ablauf benötigt werden, an den Server zurück. Stimmt der Sitzungsschlüssel überein sollen alle Daten an die Schicht der Applikation weitergeleitet werden, die für die Clientverwaltung zuständig ist.

3.3. Konfigurationswerkzeug

Das von mir entwickelte Konfigurationswerkzeug soll die einfache Verwaltung und Steuerung der einzelnen Simulationen ermöglichen. Es entspricht in dem Client-Server-Modell dem Server. Das Konfigurationswerkzeug wird in vier Abschnitte gegliedert. Der eine Abschnitt ist für die Darstellung aktueller Statusinformationen zuständig. Ein anderer Abschnitt dient der Steuerung der Clients. Im dritten Abschnitt soll die Konfiguration vorgenommen und/oder angezeigt werden. Der letzte Abschnitt soll sich um alle Abläufe kümmern, die dem Benutzer verborgen bleiben.

3.3.1 Steuerung des Clients

Die Steuerung soll den Ablauf der Simulation beeinflussen. Es geht hier nicht um die Beeinflussung der Darstellung, sondern um die grundlegenden Funktionen. Zu den grundlegenden Funktionen zählen zum Beispiel das Starten und Stoppen der Simulation. Dazu müssen alle Clients Funktionen bereitstellen, um Remote[11] gesteuert werden zu können. Nach der Eingabe durch den Benutzer müssen alle Clients angewiesen werden, ihre Simulation zu stoppen bzw. zu starten. Die Benutzereingabe soll wie beim bisherigen Client durch entsprechende Button erfolgen.
Um die Funktionalität der Clients nicht zu stark einschränken zu müssen, sollen Steuerungsereignisse des Clients an den Server übermittelt werden. Dieser muss sich dann um die Weiterleitung der Anweisung an alle weiteren Clients kümmern.

3.3.2 Routing

Zentraler Bestandteil des Server ist die Erstellung von Routen zwischen den einzelnen Portalen/Welten und dem Transfer von Agenten entlang dieser Routen.
Die Routen sollen während eines Simulationsexperimentes statisch bleiben. Somit ist eine Nachvollziehung von Simulationsergebnissen gewährleistet. Würden sich die Routen immer wieder ändern, könnte man das Zu-/Abwanderungsverhalten in den einzelnen Welten nicht näher analysieren. Eine Welt sollte aus mindestens zwei Portalen bestehen, damit sich überhaupt ein vernünftiges Netzwerk an Welten aufbauen lässt. Man könnte auch einen Weltenverbund mit einer Transferwelt (n Portale bei n weiteren Welten) und n Welten mit jeweils einem Portal aufbauen, was aber auch in der Natur nicht vorkommt.
Im Folgenden soll der Algorithmus beschrieben werden, der im Zuge dieser Arbeit für die

11 Remote (engl.): ferngesteuert

Routenerstellung entwickelt wurde. Bei der Anmeldung der Clients müssen diese ihre Anzahl an Portalen an den Server übermitteln. Die Anmeldung erfolgt seriell. Die Erstellung der Routen soll in Reihenfolge der Anmeldung geschehen. Mit dem Start der Simulation muss die Möglichkeit der Anmeldung von weiteren Clients unterbunden werden. Die Endpunkte einer Route sind immer die Portale, die im System eindeutig durch die Client-Id und die Portalnummer bezeichnet sind. Für die Initialisierung der Routen sind ein Stapel und ein erweiterbares Feld notwendig. Das dynamische Feld speichert alle erstellten Routen. Auf dem Stapel werden die noch nicht verbundenen Portale abgelegt. Soll nun für ein Portal eine Route erstellt werden, wird geschaut, ob auf dem Stapel ein freies Portal vorhanden ist. Ist kein freies Portal auf dem Stapel, wird das neue Portal auf diesem Stapel abgelegt. Befindet sich ein Portal auf dem Stapel und ist noch keine Route im Feld gespeichert, wird eine Route aus dem neuen Portal und dem Portal aus dem Stapel erstellt. Diese Route wird dann dem Feld hinzugefügt. Sollten schon Routen im Feld vorhanden sein, wird sich zufällig eine Route aus dem Feld herausgesucht, von der der Endpunkt ermittelt wird. Die unidirektionale Betrachtungsweise ist notwendig, um Routen innerhalb nur einer Welt zu verhindern. Der Endpunkt der bestehenden Route wird bei diesem Algorithmus gegen das neue Portal ausgetauscht. Übrig bleiben nun der alte Endpunkt und das Portal auf dem Stapel. Um das Risiko von weltinternen Routen zu minimieren, wird der alte Endpunkt zum Anfangspunkt der neuen Route und das Portal auf dem Stapel bildet den Endpunkt. Nach diesem Algorithmus ist es aber leicht möglich, dass eine Welt nur Routen zugewiesen bekommt, die nur zwischen ihren eigenen Portalen verstrickt sind. Diese Möglichkeit ist besonders wahrscheinlich, wenn in dem Netzwerk zwei Welten mit einer ungeraden Anzahl an Portalen vorhanden sind. Um diese Wahrscheinlichkeit zu senken, muss verhindert werden, dass bei der zufälligen Auswahl der Route ein und dieselbe Route kurz hintereinander ausgewählt wird. Deshalb muss bei der Implementierung des Algorithmus darauf geachtet werden, dass eine Sperranzahl an neuen Routen definiert wird, bevor eine Route nochmals zufällig ausgewählt werden darf.

Pseudocode für Erstellung der Routen:

```
var portalstapel;
var routenliste;
funktion erstelleRoute(vektor portaleliste)
{
     füralle(portal in portalliste)
     {
           wenn (portalstapel == leer)
           {
                 lege portal auf portalstapel;
           }
           sonst wenn (routenliste == leer)
           {
                 neuroute = portalstapel.erstesElement + portal;
           }
           sonst
           {
                 var zufall = bildezufallszahl;
                 wenn (zufall schon einmal benutzt)
                 {
```

```
                    zufall = neuezufallszahl;
                }
                var route = nehmeroute an stelle zufall;
                route.tausche(routenendpunkt mit portal);
                neuroute =  ehemaligerroutenendpunkt +
                            portalstapel.erstesElement;
            }
        }
    }
```

Trotz dieses Algorithmus lässt sich eine optimale Vernetzung der Welten nicht immer realisieren. An einem Beispiel sollen die Probleme dieses Algorithmus aufgezeigt werden. Es melden sich nacheinander `'2n'`-Clients mit nur einem Portal an. Die `'2n'`-Clients erzeugen `'n'` Routen mit `'n'` Paralleluniversen, da ja jeder Client nur mit einem Client verbunden sein kann. Der Austausch mit einer anderen Welt ist für jeden Client weiterhin gesichert. Fügt man nun noch eine Welt (Transitwelt) hinzu, die `'2n'` Portale aufweist, ist es rein theoretisch möglich, dass alle Welten durch den Algorithmus mit dieser Transitwelt verbunden sind. Es würde unter optimalen Bedingungen ein zentralisiertes Netzwerk entstehen. Da der Austausch der Endpunkte bei ein und derselben Route untersagt ist, besteht bei hinreichend kleinem `'n'` keine Sorge von einem oder mehreren Paralleluniversen. Ist `'n'` aber größer als der minimale Abstand für die gleiche Zufallsroute, dann könnten zwei Portale der Transitwelt untereinander verbunden und eines der Paralleluniversen nicht in das Transitsystem eingebunden werden.

Ein ähnliches Problem besteht bei 3 Welten, wobei 2 Welten nur ein Portal besitzen und eine Welt zwei Portale. Meldet sich die Welt mit zwei Portalen nun zwischen den anderen beiden an, kommt es zu folgender Situation: Die erste Route(`w1p1 -> w2p1`) besitzt als Anfangspunkt das einzige Portal von Welt1 (`w1p1`) und als Endpunkt das Portal1 von Welt2 (`w1p1`). Das zweite Portal der zweiten Welt(`w2p2`) wird auf den Stack[12] gepackt und wartet nun so lange, bis die dritte Welt sich anmeldet. Der Endpunkt der ersten Route (`w2p1`) wird nun gegen das Portal der dritten Welt (`w3p1`) ausgetauscht, wobei die Route dann von (`w1p1`) zu (`w3p1`) verläuft. Die neue Route ist dann folglich zwischen (`w2p1`) und (`w2p2`). Damit bleibt die zweite Welt außerhalb des Netzwerkes. Nimmt man anstatt des neuen Portals, das Portal vom Stack zum Austausch, wird diese Situation behoben. Dafür können bei anderen Konstellationen Paralleluniversen entstehen.

3.3.3 Einstellungen

Die Simulation lässt sich durch eine Vielzahl von Parametern in ihrem Verlauf beeinflussen. Jedes Lebewesen besitzt eigene Parameter. Um den Nutzer nicht mit einer Liste von allen Parametern zu überfordern, werden im Rahmen dieser Arbeit die Einstellungen nach Lebewesen beim Server getrennt vorgenommen. Um Fehler durch Nutzereingaben zu vermeiden, sollen dem Nutzer die Eigenschaften als Liste angezeigt werden und die Einstellungen innerhalb von vordefinierten Grenzen vorgenommen werden können. Zur Definition der Grenzen von Werten werden beim Server Scroll-Balken verwendet. Da bei einigen Parametern die Wertebereiche sehr hoch sind, sind Schrittweiten für die Scroll-Balken vorzugeben.

12 Stack (engl.): Stapel

3.4. Multimediaerweiterung

Bei der Erweiterung der multimedialen Fähigkeiten der Simulation soll es in dieser Arbeit um das Sichtbarmachen von versteckten Vorgängen gehen. Die Rasse der Agenten ist in der Lage untereinander zu kommunizieren. Die Kommunikation ist für den Nutzer bisher aber nicht erlebbar gewesen. Ziel dieser Ausarbeitung ist es, die komplizierte Sprache für Nutzer einfach lesbar zu machen. Dazu wird bei dieser Arbeit ein neuer `MessageLogger` entwickelt, sowie ein Übersetzer der Sprache KQML.

Der `MessageLogger` und der Übersetzer sollen letztendlich in einem FrontEnd[13] zusammengeführt werden. Das FrontEnd soll immer nur die Kommunikation eines Agenten darstellen, damit die Übersichtlichkeit nicht verloren geht. Als Einstellungen im FrontEnd soll der Nutzer die Anzeige der Kommunikation auf einen Partner einschränken oder nur eingehende oder ausgehende Nachrichten auswählen können.

Der `MessageLogger` soll alle nativen Sprachkonstrukte die vom Agenten entgegengenommen werden, oder welche er ausspricht, ablegen. Der Logger muss zur Identifikation noch die Richtung der Nachricht und den Gesprächspartner abspeichern. Dazu muss in der Kommunikationsklasse des Agenten bei jeder Kommunikationsaktion der Logger benachrichtigt werden. Der Logger muss entsprechende Methoden zur Rekonstruktion der Kommunikation bereitstellen. Dazu zählt sowohl die Wiedergabe aller Nachrichten, die mitgeschnitten wurden, als auch die Rückgabe nur derjenigen Nachrichten, die einem bestimmten Gesprächspartner zugeordnet sind. Der Logger soll keine persistente Speicherung der Nachrichten vornehmen. Da immer nur die Kommunikation eines Agenten mit dieser Arbeit zur Ansicht gebracht wird, reicht es alle diejenigen Nachrichten zu protokollieren, die dem ausgewählten Agenten zugeschrieben werden können. Der Logger sollte also auch an- bzw. abschaltbar sein, um den Speicherbedarf zu reduzieren.

Der Wortschatz der Agenten ist eher klein, so dass man für jedes Sprachkonstrukt eine vollständige Übersetzung vorformulieren kann. Die Übersetzung der Konstrukte soll in Property-Dateien abgelegt werden. Die Speicherung der verständlichen Texte in einer lesbaren Form, lässt einen einfachen Austausch der Sätze zu, damit man die Texte bei Bedarf an die einzelnen Nutzergruppen anpassen könnte.

3.5. Simulationsexperimente

Im realen Leben ist eine Tierart bestrebt längst möglich zu überleben. Dazu passt sie sich ihrer Umwelt an. In dieser Simulation können sich die Agenten an die Umgebung anpassen. Bei der Fortpflanzung von zwei Agenten wird die DNA für den Nachwuchs aus der DNA beider Elternteile gebildet. Damit die Art der Agenten nicht ausstirbt, muss sichergestellt sein, dass immer genug Beutetiere vorhanden sind. Auch müssen immer beide Geschlechter bei den Agenten vorhanden sein, da eine Fortpflanzung durch Paarung mit dem gleichen Geschlecht nicht möglich ist.

13 FrontEnd (engl.): grafische Ausgabe auf dem Bildschirm

Die größte Wahrscheinlichkeit eines Aussterbens der Agenten liegt beim Fehlen an Beute. Die Kapazität[6] der Beute(Quarries) in der Welt liegt bei rund 90 Beutetieren. Diese Kapazität ergibt sich aus der Nachwuchsrate von deren Futtermitteln. Die Geburtenrate liegt bei den Quarries in einer Generation bei eins. Die Sterberate pegelt sich durch fehlendes Futter auch bei eins ein, wenn die Kapazität von 90 erreicht wird und somit das Populationswachstum auf null sinkt. Die wirkliche Anzahl der Quarries schwankt ohne Einfluss von Agenten um die Kapazität, wie aus Abbildung 1 ersichtlich wird.

Damit die Agenten ihre Nahrungsgrundlage nicht vollständig ausrotten, kann man mehrere Parameter ändern. Im Folgenden soll beschrieben werden, wie in dieser Arbeit eine Parametereinstellung gefunden werden soll, die ein langes Überleben der Rasse der Agenten ermöglicht.

Damit man Rückschlüsse der Experimente auf die Parametereinstellungen treffen kann, darf in einer Simulationsreihe nur eine Eigenschaft einer Rasse geändert werden. Viele Eigenschaften beruhen auf dem Zufall. Um die zufälligen Einflüsse zu reduzieren, wird jede Einstellung in den Experimenten zehnmal durchlaufen. Zur Auswertung werden der Mittelwert und der Median[14] aller Durchläufe herangezogen. Der Median hilft große, aber wenige, Abweichungen aus der Statistik zu eliminieren.

Ein Teil der Eigenschaften der Lebewesen beruht auf Distanzen. So gibt es maximale Sichtweiten der einzelnen Rassen, maximale Jagdweiten oder auch maximale Entfernungen zur Abwehr von Angriffen. Eine genaue Untersuchung all dieser Einstellungen ist zeitlich sehr aufwändig und es gibt keine Garantie, dass sich Änderungen, die in ihren Einzelexperimenten eine positive Wirkung zeigten, beim Zusammenwirken nicht genau das Gegenteil hervorrufen. Um eine Wirkung auf alle Distanzeinstellungen zu erzielen, kann man auch die Größe der Welt einfach verändern. Die Größenänderung hat Auswirkungen auf das Sichtfeld eines jeden Bewohners. Die Beute ist mit zunehmender Größe schlechter ausfindig zu machen, da der sichtbare Anteil der Welt immer kleiner wird und auch die Streuung der einzelnen Bewohner zunimmt.

Damit die Beutetiere eine größere Überlebenschance haben, kann man ihren Frühwarnmechanismus vor Agenten verbessern. Dazu muss der Wert für die Warndistanz nach oben gesetzt werden. Somit hat das Beutetier eher die Chance vor einem Agenten zu flüchten. Um die Beutejagd weiter einzuschränken, kann man die maximale Distanz zum Lähmen der Beute verringern. Da nur gelähmte Tiere verzehrbar sind, wird dadurch der Jagderfolg gesenkt.

Um die Auswirkungen einer zu extensiven Jagd zu lindern, kann man den Eintritt der Folgen verzögern. Dies geschieht durch die Verlängerung der Reproduktionspausen bei den Agenten. Je größer der Abstand zwischen den einzelnen Generationen ist, desto später reagiert die Population auf gute Nahrungsverhältnisse.

14 Median: Wert des Element in einem sortierten Datenfeld, was genau in der Mitte liegt. (Bei neun Elementen ist der Median der Wert des 5. Element)

Kapitel 4 – Realisierung

4.1. Überarbeitung der Klassen

JAVA 5.0[10] bringt zahlreiche Neuerungen mit sich. Zu den größten Erweiterungen zählen dabei wohl die Einführung von generischen Datentypen. Listen mussten bisher immer mittels `Iterator` auf noch folgende Elemente geprüft und dann in einer Schleife ausgewertet werden. In der neuen Version existiert eine for-Schleife, die solche Datentypen innerhalb dieser auswerten kann. Viele Datenstrukturen benötigen Objekte zur Bearbeitung. Primitive Datentypen waren ausgeschlossen oder mussten mittels Wrapper-Klassen in Objekte umgewandelt werden. Mit JAVA 5.0 übernimmt dieses Boxing der Compiler automatisch. Der Einsatz von Annotations in der aktuellen Version erleichtert die Arbeit in den verschiedensten Frameworks, wo zusätzlich zu dem Programm noch andere Steuerinformationen benötigt werden. Bei RMI entfällt das Vorkompilieren der Stub-Klassen. Dies geschieht jetzt dynamisch.

Die Simulation arbeitet mit einem Framework zur Darstellung der Grafiken. Eine Aktualisierung dieser Klassen auf die neueste Version wurde nicht durchgeführt. Viele der in der Simulation vorkommenden Klassen nutzen Datenstrukturen, die mittlerweile generische Datentypen unterstützen. Dies betrifft insbesondere Vektoren. Soweit sich eine Eingrenzung der Datentypen vornehmen ließ, wurden diese auch vorgenommen. Bei Datenstrukturen, die mit dem GUI-Framework in Verbindung stehen, war die Einführung der generischen Datentypen nicht möglich. Dies begründet sich damit, dass dieses Framework mit JAVA 1.4 kompiliert wurde.

Beispiel:

```
public class AgentKomm
{
     public static int cGroups,cGroupsDissolved,cMembers,cMaxMembers;

     public int groupsJoined,groupsLeft,groupsDissolved,cancellations;

     protected Agent leader;
     protected Vector<Agent> partners;

     private Agent thisAgent;

     private Vector<Performative> requests,    // new requests
                  recruitRequests,             // the requests this
                                                  agent send to others
                  joinRequests;                // the ones it got from
others
     private Vector<Agent> potentialPartners,
                  potentialReprodPartners;
     private long lastEnvFromPartners;
[...]
}
```

Die Umstellung auf das automatische Boxing ist nicht erfolgt, da es keine Performancevorteile bringt und die Stabilität der Simulation nicht erhöht. Die Notwendigkeit der Verwendung des neuen Schleifenkonstruktes bestand nicht. Auf Netzwerkebene wurde bisher nur mit Object-Streams gearbeitet, wodurch die Vorteile von RMI in JAVA 1.5 nicht zum Tragen kommen.

Weiterhin wurden Methoden mit nötigen Annotations versehen, wenn zum Beispiel Methoden überschrieben wurden.

Beispiel:

```
public class Agent extends MovingItem implements IAgent,
java.io.Serializable {
[...]
    @Override
    public void itemRemovedFromWorld(Item item) {
        environment.removeElement(item);
        if ((currentState == HUNTING || currentState == EATING)
                && targetToHunt == item) {
            targetToHunt = null;
            currentNeed = NONE;
            currentState = NONE;
        } else if (item instanceof Agent) {
            komm.partnerWithdraw((Agent) item);
            if ( currentState == REPRODUCING2 &&
                 reproductionPartner== item) {
                currentState = NONE;
                currentNeed = NONE;
                reproductionPartner = null;
            }
        }
    }
[...]
}
```

JAVA-Programme sind besonders bei Grafikanwendungen sehr rechenintensiv. Das trifft auch auf die vorliegende Simulation zu. Besonders das Mapping der Welt auf die Oberfläche ist sehr zeitintensiv. Die Welt besteht intern aus einem zweidimensionalen Feld, welches alle Elemente referenziert. Die Darstellung erfolgt auf der Grundlage dieser Feldinformationen. Einen weitaus größeren Rechenaufwand stellt die Erstellung der Statistik dar. Hierzu müssen sämtliche Daten aller Lebewesen in der Welt erfasst werden.

Die Welt verfügt über eine Netzwerkschnittstelle. Über diese Schnittstelle können Agenten von einer Welt zu einer anderen übertreten. In der Ursprungsversion geschieht das mittels Sockets und einem `Object`-Stream. Die Portale über die der Übertritt realisiert wird, haben festgelegte Ports, auf denen sie lauschen und senden. Bei Annahme einer Verbindung wird ein weiterer Port zur eigentlich Kommunikation geöffnet. Werden diese im Anschluss an die Datenübertragung nicht geschlossen, verbrauchen sie weiterhin Systemressourcen[1]. In der Grundversion, wurden keine der Sockets und Streams geschlossen. Die Portale sind von mir so geändert worden, dass die Applikation jetzt flüssiger läuft.

Beispiel:

```
public class Portal extends Item {
[...]
        @Override
        public void run() {
            Socket socket;
            int counter = 0;
            try {
                while (running) {
                    try {
                        socket = listenSocket.accept();
                        counter++;
                        ObjectInputStream in = new ObjectInputStream
                                        (socket.getInputStream());
                        boolean transferring = false;
                        transferring = in.readBoolean();
                        if (transferring) {
                            Agent agent = (Agent) in.readObject();
                            agent.setWorld(world);
                            agent.initAgent(thisPortal);
                            agent.moveTo(x, y);

                            world.addItem(agent);
                        }
                        in.close();
                        socket.close();
[...]
}
```

In der erweiterten Version wird diese Art von Portalen nicht weiterverwendet. Sie dienen nur der Abwärtskompatibilität. Die neuen Portale werden zentral durch einen RMI-Server verwaltet. Dieser Server kümmert sich dann um den Auf- und Abbau der Verbindungen.

4.2. Das neue Portal

Das Portal aus der Grundversion versendet/empfängt Agenten über eine Socket-Verbindung, die bei der Initialisierung aufgebaut wird. Das neu entwickelte zentrale System arbeitet aber nicht mehr mit einfachen Sockets sondern mittels RMI-Technologie. Das neue `CentralizedPortal` verwendet für den Transport die gleichen Methodensignaturen, wie das einfache Portal. Dies dient dazu, dass keine großen Änderungen in den Skripten oder anderen Klassen vorgenommen werden müssen. Das `CentralizedPortal` speichert zusätzlich zur Welt, in der das Portal steht, den `RemoteClient`[siehe S. 25], über den der Agententransport abgewickelt wird. Das neue Portal kennt nun nicht mehr das Ziel, sondern nur die Transportschicht. Um zu prüfen, ob ein Agent auch wirklich transportiert wurde, verlangt das Portal bei Aufruf der entsprechenden Methode im `RemoteClient` einen Wahrheitswert zum Versandstatus. In den weiteren Eigenschaften gleicht das neue `CentralizedPortal` dem Portal der Ausgangsversion.

4.3. RMI

4.3.1 Interface und Remote Server

Zur Kommunikation mittels RMI musste eine Schnittstelle erstellt werden, die von der Klasse Remote abgeleitet ist. Das Interface muss alle Methoden enthalten, welche später durch eine externe Anwendung aufgerufen werden können. Der Server soll ähnliche Aufgaben verrichten, wie die Clients. Es bietet sich also an, ein Interface zu definieren, das für beide Anwendungen verwendbar ist.

```
public interface RemoteSystem extends Remote {

    /** Methode stoppt die Simulation */
    public boolean stop() throws RemoteException;

   /** Methode startet die Simulation */
    public boolean start() throws RemoteException;

    /** Methode beendet die erste Simulation und erzeugt eine neue
     * Welt, die dann gestartet wird. */
    public boolean restart() throws RemoteException;

    /** Methode übernimmt einen Agenten */
    public boolean receiveAgent(int portal, Agent agent, String from)
throws RemoteException;

    /** Setzt die DNA */
    public @Deprecated void setDNA(DNA dna) throws RemoteException;

    /** Verändert die Eigenschaften von Items. */
    public void setAttribut(Class item, String attribute, double
value) throws RemoteException;

    /** Speichert die Daten der Statistik. */
    public void setStatistic(Object[] data) throws RemoteException;
}
```

Alle Methoden, die nicht bei beiden Elementen vorkommen, müssen von der jeweils anderen einfach leer implementiert werden.

Die Methoden zum Setzen von Parametern sind eigentlich nur für die Clients von Bedeutung, da nur der Server die Parametrierung vornehmen soll. Das Setzen der kompletten DNA war vorgesehen, ist aber nicht als sinnvoll erachtet worden und wurde in dieser Arbeit durch die Methode zum Setzen der Attribute ausgetauscht.

Der RemoteServer kennt ein Server-Objekt, welches sich um die Bearbeitung aller Funktionen im Client kümmert. Alle Aufrufe von Methoden des RemoteServer werden an die entsprechenden Instanzen vom Server-Objekt weitergeleitet. Als Beispiel hierfür soll die Methode "receiveAgent()" der Remote-Schnittstelle dienen:

```
/** Nimmt einen Agenten entgegen und versucht ihn zu
 * transportieren. Der Identifikationsstring (<code>from</code>)
 * ist wie folgt anzugeben:<br/>
 * <code>$SenderIP?$SenderID</code></p>
 * <p>Sollte keine Route gefunden werden, oder Netzwerkprobleme
 * auftauchen, wird <code>false</code> zurückgegeben.</p>
 */
public boolean receiveAgent(String from, int portal, Agent agent)
        throws RemoteException {
    log.info("accept agent from portal " + portal +
             " of client " + from);
    return serv.transitAgent(portal, agent, from);
}
```

Die ersten beiden Parameter dienen der Bestimmung der Herkunft des zu transportierenden Agenten. Auf Grundlage dieser beiden Daten wird das Ziel des Agenten bestimmt. Der letzte Parameter ist der Agent, welcher transportiert werden soll.

4.3.2 RemoteClient

Im Gegensatz zum `RemoteServer` werden in der `RemoteClient`-Klasse alle Anweisungen auch ausgeführt und nicht an andere Instanzen weitergeleitet. Um zu verhindern, dass der Client von mehreren Servern gesteuert wird, basiert der `RemoteClient` auf dem Singleton-Pattern. Dieses Pattern sagt aus, dass maximal eine Instanz der Klasse in der gesamten VM vorhanden sein darf. In JAVA wird dazu der Konstruktor der Klasse nicht öffentlich gemacht. Eine statische Methode[15] erstellt eine neue Instanz von der Klasse, sofern noch keine Instanz erzeugt wurde, die dann der Applikation zur Verfügung steht.

15 Methode die auch ohne eine Instanz der Klasse aufrufbar ist

```
/** Initialisiert den Fremdsteuerungsmechanismus.
  * Sollte schon ein RemoteClient in der VM existieren,
  * wird dieser verwendet. Damit der
  * RemoteClient auch funktionsfähig ist, muss einmalig
  * noch eine Generierungsmaschine für die Welt und ein
  *Statistik-Objekt gesetzt werden.
  */
public static RemoteClient getInstance() throws RemoteException
{
     if(rc != null)
        {
            return rc;
        }
        return rc = new RemoteClient();
}
```

Damit der Client auch funktionsfähig wird, muss noch der Generator für die Welt und die Statistik mit eingebunden werden. Dies geschieht durch zwei set-Methoden, die aber nur bei erstmaligem Aufruf die betreffenden Elemente setzen, damit auch immer an den gleichen Daten gearbeitet wird.

Set-Methode für den Generator:

```
/** Setzt den benötigten WorldGenerator, aus dem
  * die Welt bezogen wird, bzw. mit dem Neustarts usw.
  * durchgeführt werden.
  * Der Generator kann nur einmalig gesetzt werden. Danach
  * wird die Aktion einfach ignoriert.
  *
  */
public void setWorldGenerator(WorldGenerator generator)
{
        if(this.generator == null)
        {
            this.generator = generator;
            this.world = generator.getWorld();
            update();
            register(18003);
        }
}
```

Das Anmeldeverfahren beim Server ist über eine `register()`-Methode implementiert. Sie blockiert den Ablauf der Anwendung so lange, bis ein Server auf ihre Anfrage geantwortet hat. Da zu Präsentationszwecken unter Umständen mehrere Clients auf einem Rechner agieren, muss für das Anmeldeverfahren sichergestellt sein, dass der Port für die Suchanfrage auch nicht belegt ist. Deshalb ist der Registrierungsmethode eine untere Portgrenze anzugeben. Sollte dieser Port belegt sein, wird durch eine Schleife der nächst höhere freie Port gesucht und dieser für das Anmeldeverfahren verwendet.

```
/** Registriert den Client beim Server.
  * Der Client baut eine UDP-Verbindung zum
  * Server auf. Der dafür freigschaltene
  * Port ist >= port
  */
private void register(int port)
    {
```

```
        DatagramSocket sock = null;
        while(true)
        {
            try
            {
                sock = createSocket(port);
                break;
            }
            catch(Exception e)
            {
                port++; // search next free port
            }
        }
        try
        {
            sock.setReuseAddress(true);
            // Anfragen an Server verschicken
            broadCast(sock);

            // wait for first response
            DatagramPacket pack =
                  new DatagramPacket( new byte[1024], 1024 );
            sock.receive( pack );
            byte data[]  = pack.getData();
            try
            {
                String adr =
                 "//" + pack.getAddress().getHostAddress() + "/MAS";
                log.log(Level.CONFIG, "server address " + adr);
                server = (RemoteSystem)Naming.lookup(adr);
            }
            catch(Exception exc)
            {
                log.log(Level.SEVERE, "system not bound");
            }
            String received = new String( data, 0, data.length -1 );
            StringTokenizer tokens =
                                    new StringTokenizer(received, "|");
            log.info("msg from server: " + received);
            while(tokens.hasMoreElements())
            {
                String next = tokens.nextToken();
                if(next.startsWith("AUTH:"))
                 auth = (next.split(":"))[1]; // read key
                if(next.startsWith("CLNT:"))
                 // get interface ip
                 url = (next.split(":"))[1].substring(1);
            }
            Registry reg;
            try
            {
                reg = LocateRegistry.createRegistry(1099);
                log.info("registry startet on port 1099");
            }
            catch(Exception exc)
            {
                log.info("registry still startet");
                reg = LocateRegistry.getRegistry();
            }
            try
            {
```

```
                reg.rebind(auth , this);
                log.log(Level.INFO, "client bound at registry");
            }
            catch(Exception exc)
            {
                exc.printStackTrace();
            }

            // send parameter
            String param = "START|MAS|AUTH:" + auth +
                           "|NAMING:1099|URL:" + url;
            param +=   "|DOCK:" +
                       (portals == null ? "0" :
                       Integer.toString(portals.size())) +
                       "|END";
            pack = new DatagramPacket(   param.getBytes(),
                                         param.length()
                                         pack.getAddress(), 18001);
            sock.send(pack);
            log.info("client bound at server");
            sock.close();
        }
        catch(Exception exc)
        {
            exc.printStackTrace();
        }
}
```

Um an alle Adressen zu senden, wird über die Broadcast-Methode ein Array mit Netzwerkadressen verwendet. Dieses ergibt sich aus der Methode `getBroadcastAddress()`. Hier werden für eine beliebige Adresse alle anderen Adressen ermittelt, die in den ersten 3 Bytes der IPv4[16] Adresse übereinstimmen.

```
 private InetAddress[] getBroadcastAddress(InetAddress address) throws
UnknownHostException
{
      byte[] ipBytes = address.getAddress(); //get bytes of address
      byte[] rawBytes = address.getAddress();
      // return only one adress at loopback interface
      if(address.getHostAddress().equalsIgnoreCase("127.0.0.1"))
      {
          InetAddress[] broadcast = new InetAddress[1];
          broadcast[0] = address;
          return broadcast;
      }
      // init array of adresses without own adress and not x.x.x.0 and
not x.x.x.255
      InetAddress[] broadcast = new InetAddress[253];
      int j = 0; // counter
      for(int i = 1; i<255; i++)
      {
         ipBytes[3] = (byte)(i);
         if(ipBytes[3] == rawBytes[3]) { j--; }
         else broadcast[j] = InetAddress.getByAddress(ipBytes);
         j++;
      }
      return broadcast;
}
```

16 IPv4 ist ein Internetprotkoll, wobei die Adressierung über eine 4-Byte-Kennung erfolgt

Der `RemoteClient` dient aber nicht allein der Steuerung von außen, sondern auch der Weiterleitung von Nutzereingaben an den Server. Somit müssen alle Methoden doppelt vorhanden sein, die von außerhalb der VM erreichbar sind, um sie sowohl lokal als auch remote aufrufen zu können. Nach erfolgreicher Anmeldung des Clients steht dem `RemoteClient` das entfernte `RemoteServer`-Objekt zur Verfügung. Um die lokalen Methoden von den entfernten Methoden zu unterscheiden, sind sie entweder mit dem zusätzlichen Wort "Remote"[entfernt:`start()` und lokal:`startRemote()`] versehen oder beschreiben das Gegenteil [entfernt:`receiveAgent()` und lokal:`sendAgent()`]. Die lokalen Methoden rufen die entsprechenden entfernten Methoden beim Server auf. Also ruft die Methode `startRemote()` beim Client die Methode `start()` des Server auf.

4.4. Server

4.4.1 Graphical User Interface (GUI)

Das GUI des hier erstellten Server gliedert sich in drei Bereiche. Auf der linken Seite befinden sich die Steuerelemente der Simulation. Die Steuerelemente sind in einem Zeichenbereich gruppiert und nehmen die verschiedenen Knöpfe zur Bedienung der Simulation auf. Die Knöpfe sind in ihrem Aussehen dem im Client angelehnt. Es werden die gleichen Symbole verwendet, um die Wiedererkennung der Elemente zu erleichtern. Im zentralen und rechten Bereich findet sich eine Fläche mit mehreren Karteireitern (Tab). Die einzelnen Reiter nehmen jeweils eine eigene Funktion wahr.

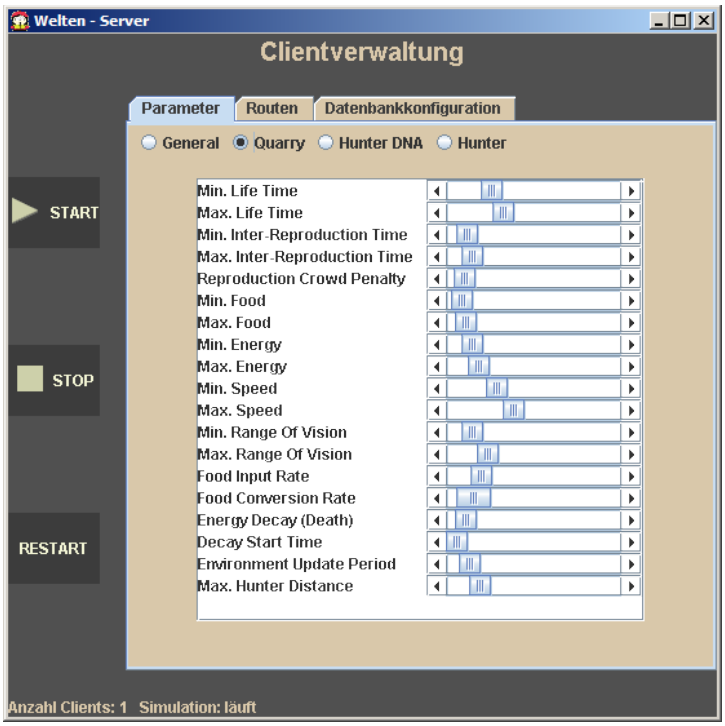

Abbildung 2: Server in Ausgangsansicht

Die erste Kartei befasst sich mit den Eigenschaften der simulierten Welt. Die Kartei ist unterteilt in zwei Bereiche. Im oberen Bereich sind Auswahl Buttons platziert, die die Art der Parameter festlegen, welche angezeigt/bearbeitet werden sollen. Unterhalb dieser Auswahl findet man alle Parameter zu diesem Bereich. Dazu kommen neben jeden Parameter eine Scrollbar, zum Einstellen des Parameter. Somit wird verhindert, dass der Nutzer falsche oder sinnlose Eingaben vornimmt.

Im nächsten Reiter werden alle Routen gelistet, die im Server bestand haben. Die Auflistung geschieht in einer Tabelle. Die Tabelle nimmt die beiden Endpunkte der Route auf. Die Endpunkte werden durch die Client-ID und eine Portalnummer identifiziert. Die Client-ID wird in dieser Tabelle verkürzt dargestellt. Sie beschränkt sich auf die letzten sieben Zeichen der eigentlichen ID.

Abbildung 3: Kartei für den Datenbankzugang

Im letzten Reiter befindet sich eine Möglichkeit zur Definition der Datenbankverbindung. Die Datenbankverbindung dient der statistischen Aufbereitung der Simulationsabläufe. Der Reiter stellt Textfelder zur Eingabe des verwendeten Datenbanktreibers, der Datenbankadresse (Hostname + Datenbankname), dem anzumeldenden Nutzer und dessen Passwort bereit. Das Passwort wird in der Anzeige durch Stern-Zeichen unlesbar gemacht. Die Daten werden bei Simulationsstart automatisch durch den Server ausgelesen.

Die Server-Steuerung erfolgt zweiseitig. Einmal vom Client und zusätzlich durch den Konfigurationsserver. Das bedeutet: wird die Simulation durch einen Client gestoppt oder gestartet, so wird dieser Befehl auch an die anderen Clients weitergeleitet. Der Server stellt dazu verschiedene Methoden bereit.

Methoden zur Steuerung im Server:

```
public class Server implements Observer {

    public Server() throws RemoteException {
    public void update(Observable o, Object arg) {
    public void manageAction(JButton button)
    public boolean transitAgent(int portal, Agent agent, String from)
    private void setAttribute(JScrollBar bar)
}
```

Die Methode `update()` wird immer dann benachrichtigt, wenn etwas in einem registrierten Objekt passiert. Die Methode entscheidet daraufhin, je nachdem woher der Aufruf kommt, was zu tun ist. Alle Aktionen die von der Netzwerkklasse kommen oder von einem Button aus dem GUI, werden an die Methode `manageAction()` weitergeleitet. Nach der Differenzierung um welche Art der Aktion es sich handelt, werden in allen Clients entsprechende Methoden aufgerufen.

```
if (button.getText().equalsIgnoreCase("restart")) {
   // stop all clients
   for (int k = 0; k < clients.size(); k++) {
        client = (RemoteSystem) Naming.lookup("//" +
                 ((ClientDescription)clients.get(i)).
                 getHost().getHostAddress()
                 + "/" +
                 ((ClientDescription) clients.get(i)).getId());
```

```
            client.stop();
        }
        window.setStatusSimulation(MainWindow.SIM_CLEARING);
        try { Thread.sleep(1000); } catch (InterruptedException e) { }
        // register clients at new experiment
        registerExperiment();
        try { Thread.sleep(1000); } catch (InterruptedException e) { }
        window.setStatusSimulation(MainWindow.SIM_INIT);
        // generate new world in all clients
        for (int k = 0; k < clients.size(); k++) {
            client = (RemoteSystem) Naming.lookup("//" +
                    ((ClientDescription)clients.get(i)).
                    getHost().getHostAddress()
                    + "/" +
                    ((ClientDescription) clients.get(i)).getId());
            client.restart();
        }
        // start new simulation
        for (int k = 0; k < clients.size(); k++) {
            client = (RemoteSystem) Naming.lookup("//" +
                    ((ClientDescription)clients.get(i)).
                    getHost().getHostAddress()
                    + "/" +
                    ((ClientDescription) clients.get(i)).getId());
            client.start();
        }
        window.setStatusSimulation(MainWindow.SIM_STARTED);
        return;
    }
```

Sollte ein Client einen Agenten an den Server übergeben, wird die Methode `transitAgent()` aufgerufen. Diese sucht in den installierten Routen nach dem Portal, aus dem der Agent kommt. Ist die richtige Route gefunden, so wird der Agent an den passenden Client weitergesendet.

Suche nach passender Route:

```
log.log(Level.INFO, "searching route");
for(int i = 0; i < routing.size(); i++)
{
    if(routing.get(i).containsEndpunkt(kn))
    {
        log.log(Level.FINE, "found knot");
        Knoten dest = (Knoten)routing.get(i).getOtherEndpunkt(kn);
        String adr =   "/" +
                       dest.getHost().toString() +
                       ":" +
                       dest.getPort() +
                       "/" +
                       dest.getId();
        log.log(Level.FINE, "dest address " + adr);
        RemoteSystem client = (RemoteSystem)Naming.lookup(adr);
        if(client == null) log.warning("client not found");
        return client.receiveAgent(     dest.getKnotenId(),
                                        agent,
                                        "MAS|" + portal);
    }
}
log.warning("source portal could not be find in routing list (no
reasons)");
```

Der letzte Teil der Steuerung ist die Einstellung der Eigenschaften bei den Clients. Hier führt eine Änderung im Client nicht dazu, dass diese Änderungen auch auf alle Clients übertragen werden. So lässt sich mit völlig unterschiedlich vorkonfigurierten Welten arbeiten. Sollten Parametereinstellungen am Server vorgenommen werden, so findet eine Übertragung dieser Einstellung an alle Clients statt. Der Name der Scroll-Bar (Scroll-Balken), welche in ihrem Wert verändert wurde, bestimmt den Parameter, der im Client zu ändern ist. Daraufhin wird der neue Wert des Balken ausgelesen und an die Clients verschickt.

```
private void setAttribute(JScrollBar bar)
    {
        int value       = bar.getValue();     // value of scroll-bar
        String name     = bar.getName();      // name of scroll-bar
        String parent   = bar.getParent().getName();//parent element
                                                        of scroll-bar
        // send data to all clients
        for (int i = 0; i < clients.size(); i++) {
            RemoteSystem client = null;
            try {
                client = (RemoteSystem) Naming.lookup(
                            "//" +
                            ((ClientDescription)clients.get(i)).
                            getHost().getHostAddress() +
                            "/" +
                            ((ClientDescription) clients.get(i)).
                            getId());
                if(parent.equalsIgnoreCase("DNA"))
                {
                    client.setAttribut(DNA.class, name, value);
                }
                else if(parent.equalsIgnoreCase("Agent"))
                {
                    client.setAttribut(Agent.class, name, value);
```

```
                }
                else if(parent.equalsIgnoreCase("Quarry"))
                {
                    client.setAttribut(MovingFood.class, name, value);
                }
                else if(parent.equalsIgnoreCase("General"))
                {
                    client.setAttribut(World.class, name, value);
                }
            } catch (RemoteException re) {    // catch rmi error
                log.warning( "client not reached: "  +
                             clients.get(i).getHost().getHostAddress);
            } catch (NotBoundException nbe) { // client not found
                log.warning( "service not bound on: " +
                             clients.get(i).getHost().getHostAddress);
            } catch (MalformedURLException e) {// wrong client url
                log.warning( "wrong url:" +
                             clients.get(i).getHost().getHostAddress);
            } catch (IllegalArgumentException iae) { // wrong attr
                log.warning( "wrong attribute - item not contains '" +
                             name + "'");
            }
        }
        log.info(      "BarAenderung: Wert/" +
                       value +
                       " - Name/" +
                       name +
                       " - Parent/" +
                       parent);
    }
```

4.4.2 Datenerhebung

Um die anfallenden Daten eines Experiments für eine spätere Weiterverarbeitung zu sichern, kann man die Daten wie bisher in einer Datei speichern, oder in einer Datenbank ablegen. Die bisherige Form hatte den Nachteil, dass für jedes Experiment und jeden Client eine eigene Datei angelegt wurde. Im Server ist eine Schnittstelle zu einer Datenbank implementiert. Diese Schnittstelle kümmert sich um die Speicherung der Daten aller Clients in einer gemeinsamen Datenbank. Die Daten, werden durch die Statistik-Klasse `WorldStat` in einzelnen Skriptobjekten für jedes Lebewesen erfasst. Daraus ergeben sich auch die einzelnen Entitäten[17]. Alle Daten der Lebewesen sind von den allgemeinen Weltdaten abhängig. Somit müssen die Daten für die Agenten und die Schwärmer immer in Zusammenhang mit der Welt gebracht werden. Während eines Experimentes fallen viele Weltdaten an, die auch unterschiedlichen Clients zuzuordnen sind. Zur Identifizierung der Clients in der Datenbank werden ihre ID's verwendet. Da die ID's die IP-Adresse des Clients beinhalten, muss der Name des Clients beim Beenden der Anwendung gelöscht werden.

17 Entität (engl. entity): Objekt aus der realen Welt

Name	Statistik - ERM
Beschreibung	Das ERM soll die Daten aufnehmen, die während eines Simulationsexperimentes anfallen. Zu einem Experiment gehören ein oder mehrere Clients, mit jeweils einer Welt, die altert.
Entitäten	- experiment (Beschreibung des Experimentes) - clientatexperiment (Listung der Clients, die am Experiment teilnehmen) - finalvalue (statische Daten eines Experimentes, die sich nicht ändern) - world (grundsätzliche Weltdaten) - quarry (Daten der Beutetiere) - agent (Daten der Räuber) - group (Daten zu den Gruppen der Räuber)

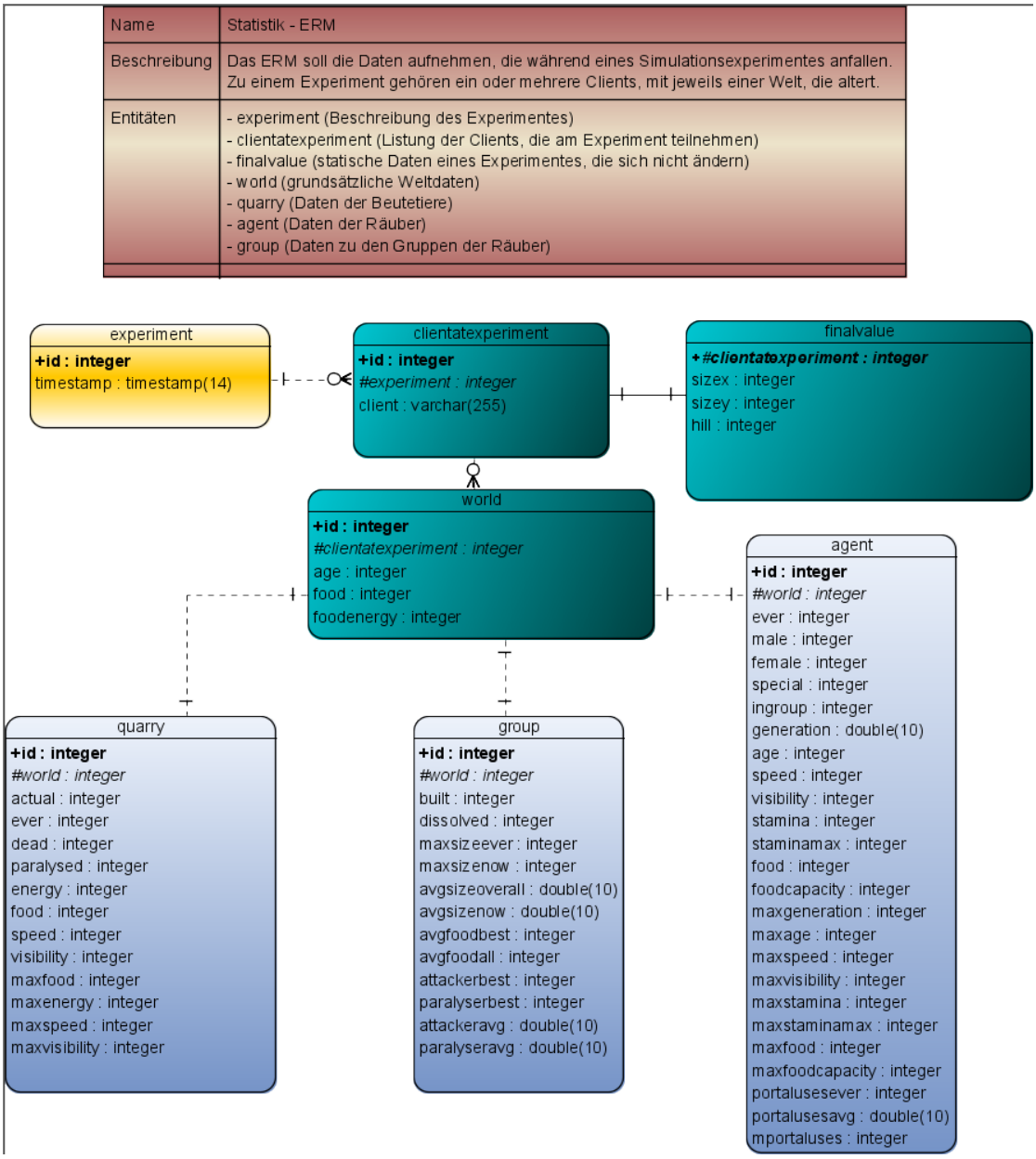

Abbildung 4: Datenbank - ERM

Routine zum Beenden der Simulation mit Löschen der Clientnamen:

```
if (arg instanceof GUI) { // signal for terminating program
            manageAction(new JButton("stop"));
            closeAllConnections();      // close open socket streams
            if(conn != null)            // close db connection
            {
                try
                {
                    Statement state = conn.createStatement();
                    state.executeUpdate(
                       "UPDATE clientatexperiment SET client = '';");
                    state.close();
                    conn.close();
                }
                catch(Exception exc)
                {
                    log.warning("couldn't close database connection");
                }
            }
            System.exit(1);
        }
```

Zur Speicherung der Skriptobjekt-Daten müssen diese erst noch in ein gültiges SQL-Statement umgewandelt werden. Aufgrund der Tatsache, dass die Felder in der Datenbank nicht den Schlüsseln in den Skriptobjekten entsprechen, müssen alle Felder einzeln zugeordnet werden. Die Datenbankfelder sind stellenweise anders bezeichnet worden, weil die bisherigen Namen nicht unbedingt aussagekräftig genug waren oder irritierten.

Erstellung des SQL-Befehls für die Schwärmer:

```
if (script.getName().equalsIgnoreCase("quarry"))
            {
                 query += "INSERT INTO quarry ";
                 query += "(";
                 query += "world, actual, ever, dead, paralysed, " +
                 query += "energy, food, speed, visibility, ";
                 query += "maxfood, maxenergy, maxspeed, " +
                 query += "maxvisibility ";
                 query += ") ";
                 query += "VALUES ";
                 query += "(" + world + ", ";
                 query += script.get("n") + ", ";
                 query += script.get("never") + ", ";
                 query += script.get("ndead") + ", ";
                 query += script.get("nparalysed") + ", ";
                 query += script.get("energy") + ", ";
                 query += script.get("food") + ", ";
                 query += script.get("speed") + ", ";
                 query += script.get("visib") + ", ";
                 query += script.get("maxEnergy") + ", ";
                 query += script.get("maxFood") + ", ";
                 query += script.get("maxSpeed") + ", ";
                 query += script.get("maxVisib");
                 query += ");";
                 // write query into variable for quarry
                 query_quarry = query;
            }
```

4.5. MessageLogger

Der `MessageLogger` allein speichert nur die Inhalte der Kommunikation eines bestimmten Agenten mit allen seinen Gesprächspartnern. Er stellt aber Methoden bereit, diese Nachrichten in den verschiedensten Formen wiederzugeben.

```
public class MessageLogger {
    public synchronized List<Message> getAllNewMessages()
    public synchronized List<Message> getAllOldMessages()
    public synchronized List<Message> getAllMessages()
    public synchronized List<Message> getAllSendMessages()
    public synchronized List<Message> getAllReceivedMessages()
    public synchronized List<Message> getAllMsgBy(  Agent ag)
    public synchronized List<Message> getAllMsgBy(  Agent ag,
                                                    boolean in)
}
```

Über die Methoden kann man eine differenzierte Rückgabe der gewünschten Nachrichten vornehmen. Mit der Methode `getAllMsgBy()` werden zum Beispiel nur die Nachrichten zurückgegeben, die einem bestimmten Agenten zugeordnet werden können. Weiterhin lassen sich die Gesprächspartner über die Methode `getAllComPartner()` ermitteln. Der Logger lässt sich über die Methoden `disable()` und `enable()` de- bzw. aktivieren. Im deaktivierten Zustand werden sämtliche Nachrichten die über eine der `addMessage()` Methoden hinzugefügt werden, nicht gespeichert.

Die Übersetzung einer Nachricht erfolgt mit dem neu entwickelten `ContentTranslator`. Zur Übersetzung ist keine Instanz der Klasse nötig. Die Klasse stellt die Funktion `translate()` bereit, die sich um das Übersetzen einer Nachricht kümmert. Die Nachricht wird in einem ersten Schritt in Kategorien eingeteilt. Für jede Kategorie existiert eine eigene Methode, die sich um die genauere Analyse der Nachricht kümmert. Sind in der Nachricht bestimmte Merkmale vorhanden, wird die passende Übersetzung aus einem Katalog ausgesucht.

Beispiel zur Übersetzung einer Anfrage nach Informationen über etwas:

```
private static String transAskAbout(Message msg, int style)
    {
        String content = (String)msg.getContent().get(":content");
        // what is ask about
        if(content.equalsIgnoreCase("attributes"))
        {
            // ask about attributes of an agent
            return phrases.getProperty("taa1");
        }
        else if(content.equalsIgnoreCase("environment"))
        {
            // ask about items in the environment
            return phrases.getProperty("taa2");
        }
        else return phrases.getProperty("error");
    }
```

Der Katalog ist in einem `Properties`-Objekt abgelegt. Der Katalog wird zuvor aus einer Datei gelesen, so kann man einfach die Texte austauschen. Der Katalog für Übersetzung im Anhang enthält eine Auflistung der Schlüsselwörter und deren Bedeutung.

Die Anzeige der Nachrichten wird über einen eigenen Frame vorgenommen. Zur Aktivierung des Loggers wurde in die Detailansicht des Clients eine Checkbox hinzugefügt. Diese Checkbox bringt die Anzeige des `MessageLogger` in den Vordergrund. Das GUI des Loggers besteht aus zwei großen Bereichen und zwei Auswahlfeldern, für die Richtung der angezeigten Nachrichten.

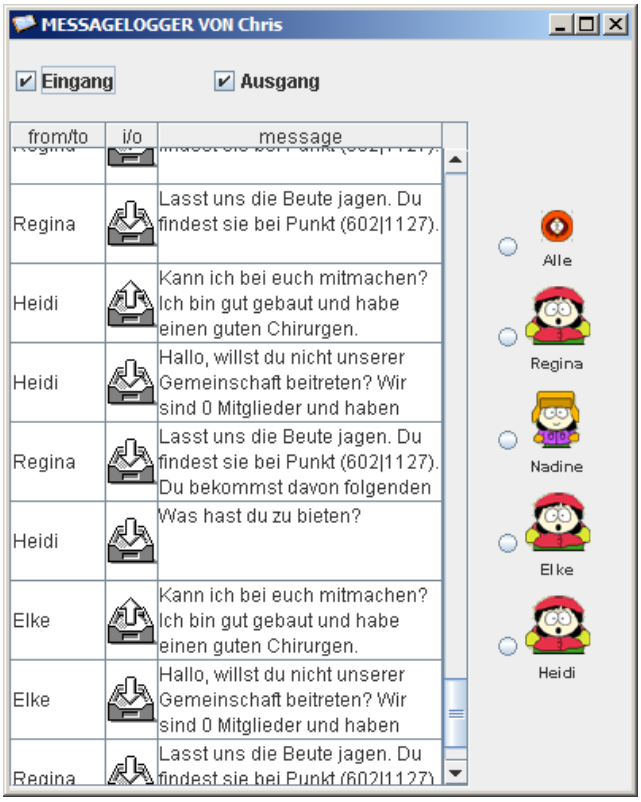

Abbildung 5: Logger der Kommunikation

Im rechten Bereich befindet sich die Liste der Agenten, mit welchen der betreffende Agent bisher protokolliert kommuniziert hat. Jedem Agenten ist ein Symbol vorangestellt, welches seine Aufgabenstellung in der Welt abbildet. Durch Auswahl eines bestimmten Agenten dieser Liste mittels der daneben liegenden `RadioButton`[18] wird die Anzeige auf die Nachrichten des betreffenden Agenten eingeschränkt.

Auf der linken Seite findet man die eigentlichen Nachrichten. Sie sind in einer Tabelle untergebracht. Die Tabelle besteht aus maximal drei Spalten. Die Spalten sind der Name des Gesprächspartners, die Richtung der Nachricht und die eigentliche Nachricht. Wird die Anzeige der Nachrichten entweder in der Richtung oder dem Gesprächspartner eingeschränkt, werden die entsprechenden Spalten nicht angezeigt.

Damit den Agenten auch Namen zugeordnet werden können, musste die Klasse `Agent` erweitert werden. Die Namen werden auch hier in eine separate Datei ausgelagert. Der Mechanismus der Namenvergabe ist nur für die neue Version möglich. Wird die alte Variante der Vernetzung genutzt, werden die Namen auf Nummern beschränkt. Die Festlegung des Namen eines Agenten erfolgt in der Methode `initAgent()`, auf Grundlage seiner Identifikationsnummer.

18 RadioButton: (Auswahl)Knopf, in einer Gruppe von diesen Knöpfen ist meist nur ein wählbar

Namenermittlung:

```
int max = Integer.parseInt(names.getProperty("maxNr", "0"));
int mod = this.agentID % max;
String strNames = names.getProperty(Integer.toString(mod));
if(strNames != null)
{
      if(this.getDNA().male)
      {
            setName(strNames.substring(strNames.indexOf('#') + 1));
      }
      else
      {
            setName(strNames.substring(0, strNames.indexOf('#')));
      }
}
```

Die Aktualisierung der Anzeige wird durch die Aktualisierung der Statistik im Client gesteuert. Mit jeder Statistikerneuerung wird auch die Anzeige des `MessageLogger` aktualisiert.

Kapitel 5 – Ergebnisse

5.1. Hinweise zur Benutzerführung

Die Benutzerführung des Clients ist in [4] ab Seite 82 beschrieben.

5.1.1 MessageLogger

Der `MessageLogger` lässt sich sehr intuitiv bedienen. Die einzigen Einstellmöglichkeiten sind die Festlegung der Richtung einer Nachricht und die Festlegung auf einen oder alle Gesprächspartner.

Bei der Richtung kann man festlegen, ob man Nachrichten sehen möchte, die vom Inhaber des Loggers verschickt oder empfangen wurden. Es ist auch möglich, beide Richtungen zu aktivieren. In diesem Fall erscheint in der Anzeige ein Symbol, für die Richtung der Nachricht.

Standardmäßig zeigt der Logger die Nachrichten aller seiner Gesprächspartner an. In diesem Fall wird vor jede Nachricht der Name geschrieben, an den die Nachricht ging, bzw. von wem sie kam. Beschränkt man sich bei der Anzeige nur auf einen Agenten entfallen die Namen.

5.1.2 Server

Der `Server` muss im Netzwerk zuerst gestartet werden. Die Clients schicken nur eine Anfrage an alle Rechner im Netz. Ist der `Server` zu dem Zeitpunkt nicht gestartet, schlägt die Suche nach ihm fehl. Der `Server` zeichnet sich durch seine einfache Handhabung aus. Aktionen wie das Starten oder Stoppen erfolgen durch die gleichnamigen Buttons auf der rechten Seite.

Die Einstellungen für die Datenbank werden in dem Reiter „Datenbankkonfiguration vorgenommen“. Die Applikation ist derzeit auf die Datenbankserver von Oracle und MySql ausgelegt, vorausgesetzt, dass die entsprechenden Treiber vorhanden sind. Der vorgegebene Treibername ist der Standardtreiber für die Version 5 der MySql Datenbank. Er befindet sich im Classpath der Startskripte. Eine genauere Liste des CD-Inhalt befindet sich im Anhang. Die Angaben können auch über entsprechende Parameter beim Programmaufruf gesetzt werden. Um die Parameter zu ermitteln, muss das Startskript mit dem Parameter `--help` aufgerufen werden.

Im Reiter Parameter können die Eigenschaften der einzelnen Objekte in der Welt verändert werden. Die Übertragung der neuen Einstellungen erfolgt unmittelbar. Sie werden nicht zwischengespeichert und können durch Clients auch nicht abgefragt werden. Somit müssen Einstellungen nach Anmeldung aller Clients und einem ersten Start vorgenommen werden.

5.2. Auswertung der Simulationsexperimente

Die Experimente fanden immer mit abgeschlossenen Einzelsystemen statt. Es konnte somit kein Ungleichgewicht aufgrund von Agenten anderer Systeme entstehen.

Ein wichtiger Faktor bei den Experimenten war die Größe der Welt. Die Experimente mit der Ausgangszahl der Schwärmer und der Warndistanz der Schwärmer haben wahrscheinlich nicht die erwarteten Ergebnisse gebracht, weil die Ausdehnung der Welt zu klein bemessen war. Die Parameter zur Fortpflanzung, als auch die Parameter zur Nahrungsaufnahme wurden sowohl bei den Quarries, als auch bei den Agenten nicht verändert und entsprachen den Vorgaben aus der Erstentwicklung der Simulation.

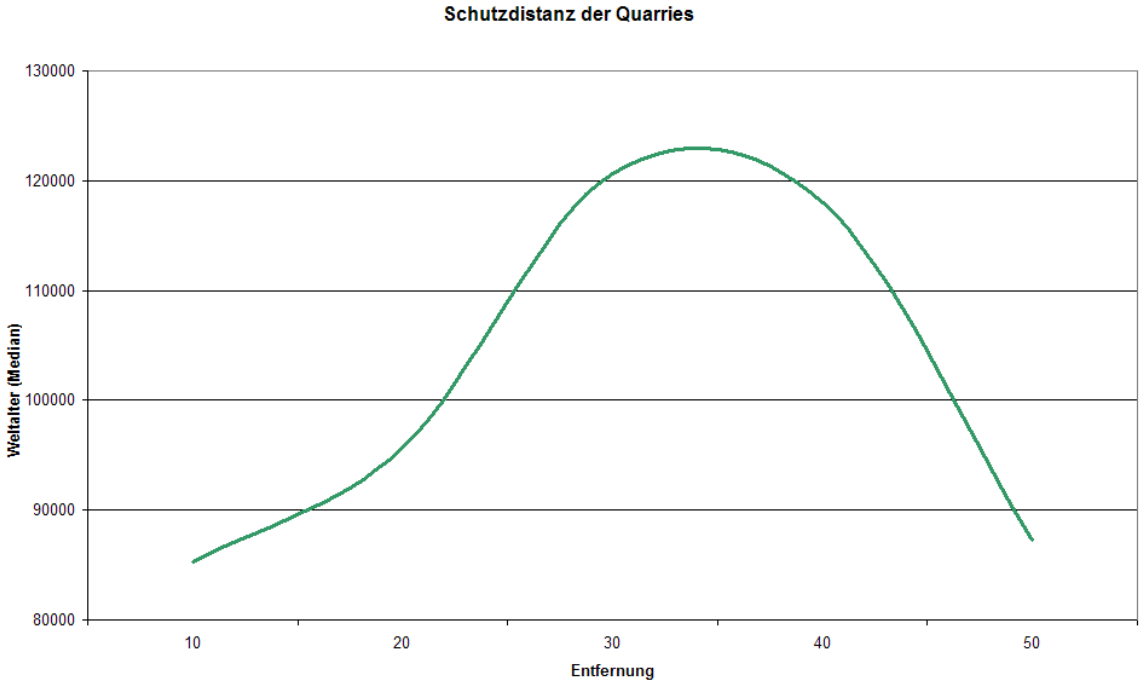

Abbildung 6: Entwicklung des Weltalters bei Änderung des Fluchtverhalten der Schwärmer

In einem ersten Experiment sollte das Aussterben der Schwärmer durch Verbesserung ihrer Flucht vor Agenten verhindert werden. Die Flucht wird durch die Distanz zu einem Agenten in Gang gesetzt. Kommt ein Agent an die Fluchtgrenze, setzt sich der Quarry in Bewegung vom Agenten weg. Durch eine Erhöhung der Distanz zwischen Quarry und Agent sollten die Quarries häufiger erfolgreich vor ihren Jägern fliehen können. Die ersten Werte aus Abbildung 6 entsprechen dieser Theorie. Agenten können nicht nur aussterben, weil sie ihre Beute ausgerottet haben, sondern auch, weil sie keine Beute mehr erlegt bekommen und somit trotz vorhandener Beute verhungern. Die Werte ab einer Distanz von 40 Einheiten spricht für ein Aussterben der Agenten, aufgrund von erfolglosen Jagdversuchen. Dies war aber in den Experimenten nie der Fall. Die Agenten haben, trotz des verbesserten Fluchtverhalten der Schwärmer, es immer geschafft, die Rasse ihrer Beute auszurotten. Dieses Experiment wurde bei einer Weltgröße von 1200x1200 Einheiten und einer maximalen Sichtweite der Agenten von 700 Einheiten durchgeführt. Somit konnten die besten Agenten die ganze Welt erfassen, ohne sich bewegen zu müssen. Der Höhepunkt ist nur damit

erklärbar, dass die Standardabweichung (siehe Anhang C1) im Alter der Welt im Verhältnis zum Durchschnitt eher gering ist. Die Simulation scheint bei Distanzen um 30 Einheiten meist gleich verlaufen zu sein, während bei den anderen Werten die Experimente zu Extremwerten bei dem maximalen Alter neigen.

In einem zweiten Experiment dieser Arbeit wurden die Jagdfähigkeiten der Agenten beobachtet. In der Simulation wurde die Fähigkeit der Agenten zum Lähmen der Quarries in Form der maximalen Aktionsradius verändert. Die restlichen Einstellungen wurden aus dem ersten Experiment unverändert übernommen und die Fluchtdistanz auf 30 Einheiten für die Quarries festgelegt.

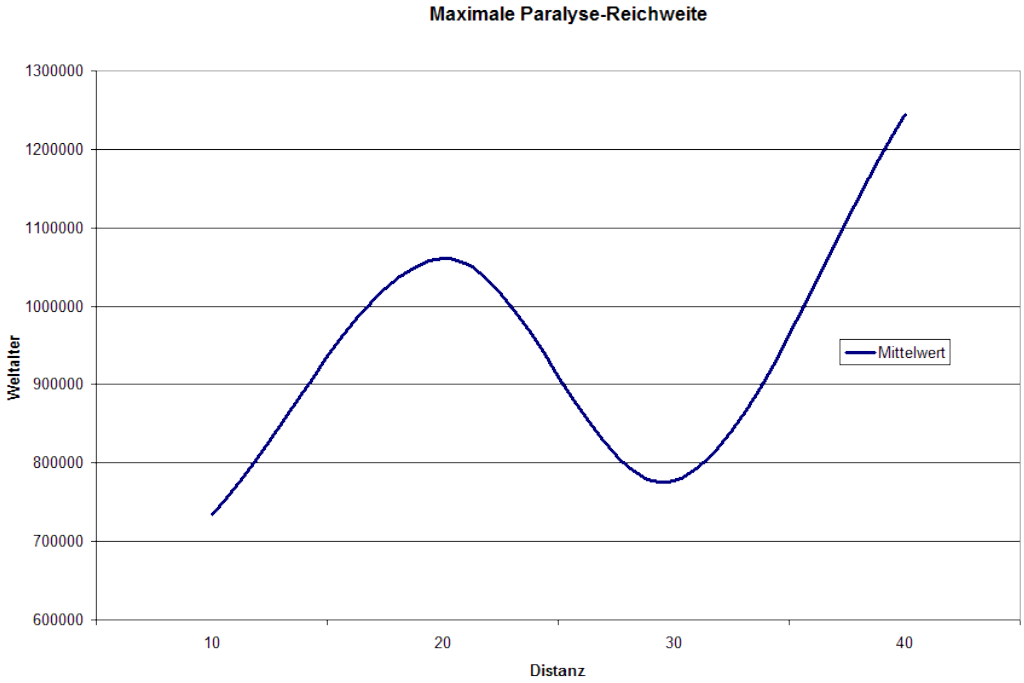

Abbildung 7: Entwicklung des durchschnittlichen Weltalter bei Veränderung der Parylsedistanz der Agenten

Man sollte annehmen, dass mit zunehmenden Fähigkeiten bei den Agenten die Beutetiere schneller ausgerottet werden würden. Das Experiment spiegelt genau das Gegenteil wider. Die Kurve aus Abbildung 7 zeigt auch wieder ein Verhalten, das auf Verhungern der Agenten schließen würde. Tatsächlich waren die Beutetiere aber vorher ausgestorben. Die Kurve lässt sich nur durch die sehr hohen Schwankungen in den einzelnen Durchlaufergebnissen (siehe Anhang C2) erklären. Die Standardabweichung ist in ihren Dimensionen so groß wie der Mittelwert, wodurch eine Bewertung der Ergebnisse bei der relativ geringen Anzahl an Werten je Konfiguration nicht möglich ist.

In einem weiteren Experiment wurde der Einfluss der Schwärmerzahl zu Beginn einer Simulation ermittelt. Bei diesen Ergebnissen haben sich zwei Bereiche herauskristallisiert, bei denen die Welt am längst überlebt. Die Bereiche für eine gute Anzahl an Schwärmern liegt um die Kapazität der Quarries (90) und bei einem Drittel dieses Wertes. Die Abweichungen der Einzelwerte liegen hier aber besonders hoch (siehe Anhang C3). Das bedeutet, dass man bei diesen Werten entweder sehr lange Versuchsreihen hat, oder besonders kurze. Für eine durchschnittlich lange Simulation sind diese Werte also nicht zu gebrauchen.

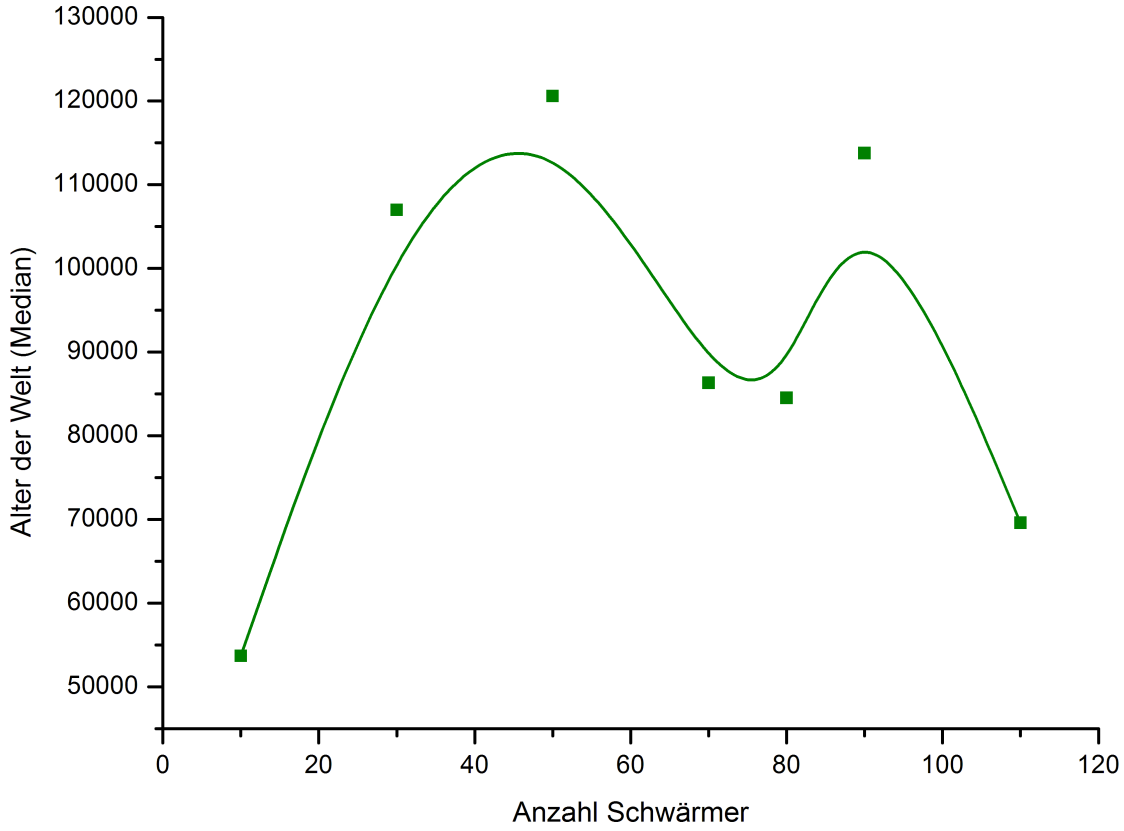

Abbildung 8: Einfluss der Beutepopulation auf die Simulation

Die meiste Aussagekraft hatte das Experiment mit den unterschiedlichen Weltgrößen. Die Veränderung der Größe der Welt hatte Einfluss auf alle Parameter, die mit Entfernungen in Verbindungen standen und alle Lebewesen wurden von den Einstellungen gleichermaßen beeinflusst. Es ist klar erkennbar, dass der Jagderfolg der Agenten mit zunehmender Größe der Welt abnimmt, ohne gleich zum Verhungern der Agenten zu führen. In der Reihe, der für die Arbeit durchgeführten Experimente, mit den verschiedenen Größen, ist keine Welt regelmäßig auf Grund von Verhungern zu Ende gegangen. Es zeigte sich aber, dass mit zunehmender Größe der Welt, diese im Verhalten der Bewohner stabiler wurde (siehe Abbildung 9).

Aus diesen Ergebnissen lässt sich schlussfolgern, dass bei einer Größe von über 1600 Quadrateinheiten, einer Paralyse- sowie Warndistanz von rund 30 Einheiten und einer Anfangszahl von 50 Quarries eine relativ stabile Simulation durchgeführt werden kann, die nicht schon in den ersten Generationen der Agenten zu Ende geht.

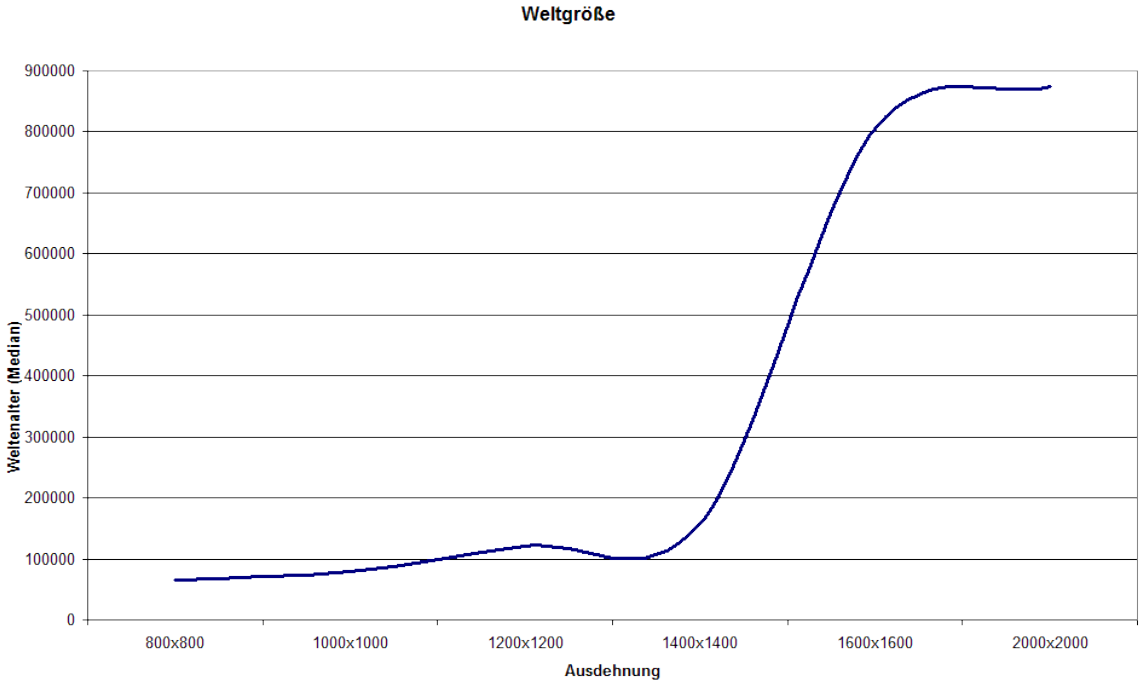

Abbildung 9: Entwicklung des Weltenalters bei unterschiedlichen Weltgrößen

5.3. Fazit

Die im Rahmen dieser Bachelorarbeit entstandenen Erweiterungen der Simulation einer künstlichen Welt ermöglichen eine vereinfachte Konfiguration dieser. Aus dieser Vereinfachung ergeben sich neue Einsatzmöglichkeiten der Anwendung zu Präsentationszwecken.

Die Simulation ist von Wechselmedien aus lauffähig. Man könnte also auf einem Rechner einen Server von CD starten und auf mehreren anderen Computern könnte man die Clients zum Laufen bringen. Die wegfallende Konfiguration der Netzwerkverbindungen ermöglicht Simulationsläufe über Computergrenzen hinweg, ohne dass man sich mit dem Netzwerk auskennen muss.

Die Datenbankerweiterung bringt einen hohen Mehrwert für das Experimentieren mit dieser Simulation. Die strukturierte Speicherung der anfallenden Daten ermöglicht eine schnelle und komfortable Zusammenfassung von benötigten Daten.

Mit der zusätzlichen Visualisierung der Kommunikation steigt der Unterhaltungswert der Applikation. Sie ermöglicht aber auch das bessere Verständnis von Abläufen innerhalb der Simulation. Man kann verstehen, warum Agenten gerade das tun, was sie tun, wenn man in das Kommunikationsprotokoll schaut.

Kapitel 6 – Zusammenfassung und Ausblick

Die hier im Rahmen der vorgestellten Arbeit entwickelten Erweiterungen für eine künstliche Simulation bieten Möglichkeiten zum vereinfachten Experimentieren und zum Präsentieren.

Das zugrunde liegende Multi-Agenten-System besticht durch seine strukturierte Programmierung und ist leicht verständlich. Anders sieht es bei dem JMT aus. Durch die fehlende Dokumentation für das Framework sind einige Lösungen nicht sehr elegant in die Simulation integriert.

Die Verbindung der einzelnen Clients wurde durch die vorliegende Arbeit vollständig automatisiert. Die automatische Suche nach Servern ist derzeit sehr einfach gestaltet. Sie findet in kleineren Netzwerken alle Server ohne Probleme. In größeren Netzwerken schlägt die Suche fehl, da nur ein begrenzter Kreis an Adressen abgesucht wird. Eine erweiterte Suche nach Vorgaben durch den Nutzer sind für die Zukunft denkbar. Bei mehreren vorhandenen Servern kann man sich eine Auswahl vorstellen.

Der Nutzer muss jetzt nur einen Server und einen Client gestartet haben, um die Simulation vollständig nutzen zu können. In der Grundversion musste man erst in den Konfigurationsdateien Einstellungen vornehmen, um die Simulation ohne Beeinträchtigungen nutzen zu können. Die Steuerung funktioniert in stabilen Netzwerken ohne Probleme. Der Wegfall von Clients aus einer Simulation wird durch den Server zuverlässig erkannt und jegliche Transfers von Agenten zu diesem Client abgelehnt. Die Routen werden automatisch vor Simulationsbeginn gebildet und führen zu einem gut ausgebildeten Netz an Routen zwischen den Welten. Bei Ausfall von Portalen oder Clients werden die betreffenden Routen gesperrt. Eine Neubildung von Routen aus dem noch aktiven Teil der Route ist derzeit nicht möglich.

Der `ContentTranslator` übersetzt derzeit fehlerfrei alle Kommunikation der Agenten untereinander. Erweiterungen im Wortschatz der Agenten kann der Übersetzer nicht von allein ausgleichen. Die Speicherung der Übersetzungen in einer Textdatei macht einen leichten Austausch und die Anpassung an das Publikum/den Nutzer sehr einfach.

Das Auffinden von Einstellungen für eine möglichst langlebige Simulation ist nicht einfach. Durch die hohe Anzahl an Parametern ist ein Optimum kaum zu finden. Die Auswertungen zeigten aber, dass wenige Änderungen die Stabilität positiv beeinflussen. Ein System, welches in seinen Populationsschwankungen regelmäßig ist, ist kaum zu realisieren.

Literaturverzeichnis

[1] Ch. ULLENBAUM: Java ist auch eine Insel, GalileoComputing, 2005

[2] D. FLANAGAN: JAVA in a Nutshell, O'Reilly Verlag, 2000

[3] D. MÖLLER: Modellbildung, Simulation und Identifikation dynamischer Systeme, Springer Verlag, 1992

[4] D. PÜTTMANN: Künstliches Leben, Simulation und Visualisierung durch ein verteiltes Multiagenten-System, Universität Karlsruhe, 2000

[5] H. ADELSBERGER: Simulation Begriffe und Modellierung , Universität Essen, http://wip.wi-inf.uni-essen.de/imperia/md/content/veranstaltungen/sim/03simulation_ss03.pdf , 2003 (Abruf: 20.07.2006 10:00)

[6] H. BAYRHUBER: Lindner Biologie, Schroedel Verlag, 2003

[7] H. SCHEPPERLE: Multi-Agenten-Systeme Grundlagen und Begriffsbildung , Universität Karlsruhe, http://www.ipd.uka.de/~oosem/AIA2001/pdf/SchepperleAusarbeitung.pdf , 2001 (Abruf: 20.07.2006 12:45)

[8] K. TROITZSCH: Modellbildung und Simulation in den Sozialwissenschaften, Westdeutscher Verlag, 1990

[9] R. VANDENHOUTEN : Software Engineering 1, TFH Wildau, Vorlesungsskript

[10] SUN: Release Notes JAVA 1.5 , SUN Inc., http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html , 2005 (Abruf: 12.06.2006 15:00)

[11] T. FINNIN : Specification of the KQML Agent-Communication Language, ARPA Knowledge Sharing Initiative,

[12] W. GROOSO: JAVA RMI, O'Reilly & Associates, 2002

Abbildungsverzeichnis

Anlagenverzeichnis

Anhang A: Katalog für Übersetzung

Schlüsselwort	*Bedeutung*
error	Fehlermeldung
taj	Anfrage Mitgliedschaft durch Gruppenleiter
tah1	Aufruf zur Jagd
tah2	Zuweisung an Beuteanteilen
tar1	Partnersuche
targ1-3	Gesundheitliche Fitness in 3 Stufen (Stufe 3: nie krank)
tarsm1-3	Status von männlichen Agenten in der Welt in 3 Stufen (Stufe 3: am Angesehensten)
tarsw1-3	Status von weiblichen Agenten in der Welt in 3 Stufen (Stufe 3: am Angesehensten)
tav1	Anfrage Mitgliedschaft durch gruppenlosen Agenten
taa1	Frage nach den Eigenschaften eines Agenten
taa2	Frage nach Umgebungsinformationen
td	Austrittserklärung
ts1	Ablehnung bei Partnersuche
ti	Zusage zur Mitgliedschaft
tt1	Umgebungsinformationen
tt2	Informationen zum Lebewesen
tu1	Ablehnung zu taj
tu2	Ausschluss aus einer Gruppe

Anhang B: CD-Inhalt

Verzeichnis	*Inhalte*
/	Startskripte, Konfigurationsdateien, Portliste
/bin	Binärdateien
/defaultBehavior	Skripte für Gestaltungsmittel
/doc	JAVA-Dokumentation
/lib	Klassenbibliotheken, Katalog der Namen und Übersetzungen
/media	Medien (Bilder für die Anwendung)
/scriptAL	Skripte zur Simulationssteuerung
/src	Quelltexte der Simulation
/pdf	Bachelorarbeit
/software	JAVA Runtime Environment für Linux und Windows

Anhang C: Simulationsdaten

1. Frühwarnmechanismus

Warndistanz	*10*	*20*	*30*	*40*	*50*
1	116850	86500	435300	102000	94200
2	112800	88000	341850	119850	73500
3	76950	98850	85200	116100	200100
4	93300	159150	117000	495750	75300
5	90000	67650	82000	492150	104250
6	80250	114000	47000	254100	175950
7	275500	92500	162150	78600	75300
8	80550	390750	124200	99900	511200
9	80400	91200	85500	77400	80400
10	79200	1005750	369000	407500	60450
Mittelwert	**108580**	**219435**	**184920**	**224335**	**145065**
Median	**85275**	**95675**	**120600**	**117975**	**87300**
Standardabweichung	**57248**	**277033**	**133964**	**166053**	**129803**

2. Maximale Paralysedistanz

Max. Paralysedistanz	*10*	*20*	*30*	*40*
1	75150	126600	555600	2871000
2	104850	1549050	2805000	276300
3	141150	1753800	682350	1464150
4	109350	828300	134850	103200
5	2509500	96900	1068900	124950
6	1966200	706800	894750	2623800
7	826500	1365900	811500	
8	180300	718500	299250	
9	1265100	254450	164500	
10	164400	919650	360000	
Mittelwert	**734250**	**1060995**	**777670**	**1243900**
Median	**172350**	**873975**	**618975**	**870225**
Standardabweichung	**846401**	**716327**	**738963**	**1161148**

3. Anzahl Schwärmer zu Simulationsbeginn

Anzahl Quarries	10	30	50	70	80	90	110
1	51450	98500	435300	86250	82050	1736100	135750
2	34950	97200	341850	60600	83250	88500	230500
3	78750	88500	85200	80250	96450	103500	78900
4	77400	809850	117000	89700	85800	84150	65100
5	51300	116700	82000	83100	162900	76500	72300
6	55950	115500	47000	86400	76650	243500	68850
7	46500	86250	162150	100500	88200	124050	69600
8	57150	193650	124200	235500	79350	84000	63450
9	57450	298800	85500	91950	72150	580500	39150
10	47850	98250	369000	77700	108900	490950	
Mittelwert	**55875**	**200320**	**184920**	**99195**	**93570**	**361175**	**91511**
Median	**53700**	**107000**	**120600**	**86325**	**84525**	**113775**	**69600**
Standard-abweichung	**12733**	**212695**	**133964**	**46502**	**25151**	**489979**	**54796**

4. Weltengröße

Ausdehnung	800 x 800	1000 x 1000	1200 x 1200	1400 x 1400	1600 x 1600	2000 x 2000
1	80100	80100	435300	117150	379500	634650
2	70000	69000	341850	90150	972000	883800
3	69150	70500	85200	499500	138000	590700
4	56400	91500	117000	117450	870450	890250
5	55950	62250	82000	1479000	112650	2128200
6	39150	84150	47000	191700	54900	568950
7	77250	290850	162150	120900	1281750	102300
8	40350	71550	124200	62400	1080300	1116300
9	65250	120600	85500	772050	737500	866000
10	64800		369000	307800	1124700	888150
Mittelwert	**61840**	**104500**	**184920**	**375810**	**675175**	**866930**
Median	**65025**	**801000**	**120600**	**156300**	**803975**	**874900**
Standardabweichung	**13237**	**67865**	**133964**	**425008**	**440614**	**495780**