FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Triebgesteuerte Fauna für Artificial Life Simulation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Rene Formanek, BSc
Matrikelnummer 0527696

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.Prof. Dipl.-Ing. Dr. Dietmar Dietrich
Mitwirkung: Dipl.-Ing. (FH) Clemens Muchitsch

Wien, TT.MM.JJJJ

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Rene Formanek,
Michael-Dietmann-Gasse 6/20, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____ _____

(Ort, Datum) (Unterschrift)

Kurzfassung

Bei der triebgesteuerten Fauna für Artificial Life Simulation handelt es sich um Software-Agenten, welche die Simulationsumgebung des Artificial Recognition System (ARS) Projekts des Instituts für Computertechnologie der Technischen Universität in Wien erweitern. Im ARS-Projekt wird eine kognitive Architektur entwickelt, deren Entscheidungsfindung an die des menschlichen mentalen Apparats angelehnt ist. Hierfür werden psychoanalytische Theorien als Basis für die Architektur verwendet. Das Verhalten der Agenten, die diese Kontrollarchitektur verwenden, soll dabei menschenähnlich sein. Da Architekturen für künstliche Intelligenz - insbesondere jene, deren Agenten menschenähnliches Verhalten zeigen sollen - nicht ausreichend durch klassische Software Engineering-Techniken getestet werden können, müssen menschliche Beobachter das Verhalten der Agenten bewerten. Diesen Beobachtern muss zur Beurteilung der Fähigkeiten und Verhaltensweisen eine Simulationsumgebung mit ausreichend vielfältigen und komplexen Möglichkeiten für Testszenarien zur Verfügung gestellt werden. In der vorliegenden Arbeit wurde eine triebgesteuerte Fauna entwickelt, deren Agenten mit einer künstlichen Intelligenz ausgestattet sind, die auf die grundlegenden körperlichen Bedürfnisse von Tieren reagiert. Diese Agenten wurden in einem Belief-Desire-Intention-Framework entwickelt und werden über Properties-Dateien angepasst. In der Simulationsumgebung wird das Verhalten der Agenten visualisiert, um Änderungen der Konfiguration nachzuvollziehen. Da sie den gleichen Simulator für die Simulationsumgebung verwenden, können die Agenten in die Artificial Life Simulation des ARS-Projekts integriert werden. Dadurch werden die Möglichkeiten, die Testern im Simulator geboten werden, erweitert und die Erstellung von zusätzlichen Testszenarien wird vereinfacht.

Abstract

The desire-driven fauna for artificial life simulation are software agents that expand the simulation environment of the Artificial Recognition System (ARS) project of the Institute of Computer Technology of the Vienna University of Technology. In the ARS project, a cognitive architecture is being developed which takes the human mental apparatus as archetype for designing decision-making. Psychoanalytical theories are being used as a basis for this architecture. The developed agents, which use this control architecture, are meant to show human-like behaviour. Architectures for artificial intelligence, especially those with agents that behave like humans, cannot be tested with classical software engineering techniques. Therefore, human observers have to evaluate the behaviour of the agents. To make this evaluation possible there has to be a simulation environment with adequate diverse and complex possibilities for test scenarios. In this thesis a desire-driven fauna was developed, which consists of agents with artificial intelligence that reacts to fundamental physical needs. For the design of these agents a belief-desire-intention framework was used and their properties are managed in separate files. In the simulation environment the behaviour of the agents can be shown and changes to their configuration made. As the agents of the desire-driven fauna and those of the ARS project use the same simulation framework, they can be easily combined in one simulation environment. In this way there are more possibilities for testers to create new test scenarios.

Danksagung

Ich möchte mich an dieser Stelle beim Institut für Computertechnologie der Technischen Universität Wien für die Möglichkeit der Durchführung dieser Diplomarbeit bedanken. Mein Dank ergeht insbesondere an o.Univ.Prof. Dipl.-Ing. Dr. Dietmar Dietrich und Dipl.-Ing. (FH) Clemens Muchitsch für die Betreuung dieses Themas.

Ich möchte aber auch allen anderen danken, die mich bei der Anfertigung meiner Diplomarbeit unterstützt haben. Besonderer Dank gebührt meiner Frau für ihre Geduld und ihr Verständnis, wenn die Arbeit an diesem Dokument mitunter sehr viel Zeit in Anspruch genommen hat.

Inhaltsverzeichnis

Abkürzungen

2D	zweidimensional
3D	dreidimensional
ACC	Agent Communication Channel
ACL	Agent Communication Language
ADF	Agent Definition File
AGI	Artificial General Intelligence
AI	Artificial Intelligence
AID	Agent Identifier
AL	Artificial Life
AMS	Agent Management System
AP	Agent Platform
API	Application Programming Interface
ARS	Artificial Recognition System
ARSIN	ein Agent des Artificial Recognition System-Projekts
BDI	Belief-Desire-Intention
DF	Directory Facilitator
EBNF	Erweiterte Backus-Naur-Form
ECLab	Evolutionary Computation Laboratory
EEG	Elektroenzephalografie
ERL	Evolutionary Reinforcement Learning
FIPA	Foundation for Intelligent Physical Agents
FPS	First-Person Shooter
GUI	Graphical User Interface
IDE	Integrated Development Environments
IIOP	Internet Inter-Object Request Broker Protocol
JADE	Java Agent Development Framework
JADEX	Java Agent Development Framework eXtension
JVM	Java Virtual Machine
KI	Künstliche Intelligenz
MASON	Multi-Agent Simulator Of Neighbourhoods... or Networks... or something...
MTS	Message Transport System
NPCs	Non-Player Characters
OQL	Object Query Language

PEAS	Performance, Environment, Actuators, Sensors
RMI	Remote Method Invocation
RPG	Role-Playing Game
SQL	Structured Query Language
XML	Extensible Markup Language
ZAMIN	Zoological Agents for Modification and Improvement of Neocreatures

1. Einleitung

Der Begriff künstliche Intelligenz (KI) umfasst ein breites Themengebiet, da alleine der Begriff unterschiedlich definiert wird. Die folgende Arbeit befasst sich vorwiegend mit jenem Bereich der KI, der sich der Erstellung von *Artificial Life* (AL), also künstlichem Leben, widmet.

Da das Verhalten der Agenten in einem AL nicht durch klassische Testmethoden des Software Engineerings überprüft werden kann, bleibt es den menschlichen Betrachtern der Agenten vorbehalten deren Verhalten zu beurteilen. Insbesondere, wenn Agenten zu beurteilen sind, die menschliches Verhalten imitieren, können letztlich nur Menschen beurteilen, ob das Verhalten des Agenten dem eines Menschen ähnelt. Je unterschiedlicher und komplexer die Testsituationen sind, die den beurteilenden Menschen präsentiert werden, desto mehr seiner Fertigkeiten kann ein Agent unter Beweis stellen. Denn je höher die Komplexität der Umgebung eines Agenten ist, desto eher kann er hochentwickeltes Verhalten demonstrieren.

Kontext der Arbeit

Am Institut für Computertechnologie der Technischen Universität Wien [1] wurde von o.Univ.Prof. Dipl.-Ing. Dr. Dietmar Dietrich ein Projekt initiiert, das sich mit der Erarbeitung eines simulationsfähigen Modells eines menschlichen Denkapparates auf mentaler Ebene beschäftigt: das *Artificial Recognition System* (ARS) [2]. Erkenntnisse der modernen Neuro-Psychoanalyse sowie die psychoanalytischen Theorien von Sigmund Freud liefern hierfür die Grundlage. Insbesondere findet das Strukturmodell der Psyche von Freud [Fre75] Anwendung. Darin werden drei Instanzen der menschlichen Psyche definiert: das Es, das Ich und das Über-Ich. Im ARS-Projekt wird die Funktionalität der Kontrollarchitektur anhand dieser drei Instanzen aufgeteilt und miteinander verknüpft. Im Zusammenspiel mit Sensoren und Aktuatoren wird das ARS-Modell für Agenten in Simulationen verwendet.

Das ARS-Projekt beinhaltet die Implementierung des Modells in der sogenannten ARSIN-*World* - der Simulationsumgebung, die mit dem Simulator MASON (Multi-Agent Simulator Of Neighborhoods… or Networks… or something…) entwickelt wurde. In dieser Simulationsumgebung sind Agenten mit diesem Modell ausgestattet und werden vor Herausforderungen gestellt, um deren Verhalten zu überwachen und zu überprüfen. Zusätzlich zu den Agenten können Hindernisse und Objekte in die Simulationsumgebung hinzugefügt werden.

Fragestellung der Diplomarbeit

Ziel dieser Arbeit ist es, eine Simulationsumgebung mit JADEX (*Java Agent Development Framework Extension*) zu erstellen, die Individuen einer triebgesteuerten Fauna enthält und dadurch eine

Erweiterung der ARSIN-*World* ermöglicht. Diese wird in der vorliegenden Arbeit *Belief-Desire-Intention*-Fauna, kurz BDI-Fauna, genannt.

Durch die Erweiterung der bestehenden ARSIN-*World* kann die *decision unit* des ARS-Projekts vor zusätzliche Herausforderungen gestellt werden. So besteht die Möglichkeit das Verhalten der Agenten in weiteren Situationen zu beobachten. Durch die umfangreicheren Testmöglichkeiten können die Verhaltensweisen der Agenten in der ARSIN-*World* verbessert werden. Zusätzliche Herausforderungen sind zum Beispiel, wenn sie bedroht werden und sich wehren, oder fliehen müssen. Oder wenn sie bei der Jagd kooperieren müssen, um ihre Beute zu fangen. Zu diesem Zweck werden, repräsentativ für Jäger und Beute, Wölfe und Hasen in die Simulationsumgebung gesetzt. Das Jagd- beziehungsweise Fluchtverhalten der beiden Arten entspricht hierbei jenem der Tierwelt, wobei die jeweilige tatsächliche Ausprägung der Eigenschaften eines Individuums oder einer Art für einen Anwendungsfall der Simulation anpassbar ist.

Im Gegensatz zu den Agenten im ARS-Projekt beruhen die triebgesteuerten Lebewesen der vorliegenden Arbeit nicht auf psychoanalytischen Modellen. Ihr Wesen ist auf die grundlegenden Bedürfnisse von Tieren beschränkt, wobei jedes Individuum hauptsächlich danach strebt selbst zu überleben. Es ist also nicht vorgesehen, dass die Tiere kooperativ handeln, weder innerhalb der eigenen Art, noch mit anderen Lebewesen der Simulation. Außerdem ist keine Kommunikation zwischen den Tieren notwendig, da ein solches Verhalten über grundlegende Bedürfnisse hinausgeht und allenfalls für zukünftige Erweiterungen des AL ein Thema ist. Ebenfalls für diese Arbeit nicht vorgesehen sind lernende Agenten. Die Agenten lernen also nicht aus ihrem Verhalten und verhalten sich daher während der gesamten Simulation entsprechend ihrer anfänglichen Programmierung.

Durch diese Einschränkungen werden Agenten für die BDI-Fauna entwickelt, die sich auf die grundlegenden Verhaltensweisen von Tieren beschränken. Da die Tiere der BDI-Fauna nicht auf das umfangreiche Modell des ARS-Projekts [3] zurückgreifen, sind sie flexibler anpassbar und vereinfachen dadurch die Durchführung von Tests für die ARSIN-Agenten.

Methode und Verfahrensweise

Bei der Entwicklung der Agenten für die BDI-Fauna wird auf JADEX zurückgegriffen, welche eine von vielen Plattformen ist, die *agent-oriented software development* unterstützen und zusätzlich einige *Tools* dafür anbietet. JADEX basiert auf JADE (Java Agent DEvelopment Framework), das wiederum den FIPA-Standard (Foundation for Intelligent Physical Agents) für die Entwicklung von Agentenplattformen implementiert.

Um die ARSIN-Agenten und die Agenten der BDI-Fauna in einer Simulation kombinieren zu können, wird zur Ablaufsteuerung und Visualisierung der Agenten MASON, das bereits im ARS-Projekt eingesetzt wird, verwendet. Zur Illustration der Simulation wird die Umgebung in einer zweidimensionalen Welt dargestellt. Für die Simulation in MASON werden die Objekte, die simuliert werden (in diesem Fall die Agenten), in einem Ablaufplan registriert und unabhängig von der Ausprägung des Objekts in jedem Planschritt aufgerufen. Daher können unterschiedliche Agenten in einer Simulation zusammengefasst und gleichzeitig simuliert werden.

Die Kombination aus JADEX und MASON ermöglicht den Einsatz des BDI-Schemas für Agenten und die Visualisierung der Simulationsumgebung in der die Agenten agieren.

Die Überprüfung des Verhaltens der Tiere erfolgt durch das Kontrollieren der Visualisierung der Simulation durch einen menschlichen Beobachter. Um einen besseren Einblick in den Zustand eines Tieres zu erlangen, werden - abgesehen von der Position der Tiere - weitere Grafikelemente angezeigt, die Aufschluss darüber liefern, in welchem Zustand sich die Agenten befinden.

Ergebnisse

Die BDI-Fauna liefert dem ARS-Projekt eine Möglichkeit die Umgebung der ARSIN-Agenten zu erweitern und dadurch zusätzliche Komplexität für die Agenten zu erzeugen. Die zusätzlichen Agenten in der Simulationsumgebung, die Tiere repräsentieren, ermöglichen das Erstellen weiterer Testfälle, um das Verhalten der ARSIN-Agenten zu überprüfen, gegebenenfalls zu verbessern oder zu erweitern.

Da die Agenten mit der BDI-Architektur im JADEX-*Framework* implementiert werden, wird die Anpassung des Verhaltens beziehungsweise des Wissens über sich selbst und die Umgebung erleichtert, da die Agentendefinition in einer strukturierten XML-Datei (*Extensible Markup Language*-Datei) vorgenommen wird. Mit Hilfe der *Properties*-Dateien können die Tiere ohne Veränderungen an der Agentendefinition in vielen Aspekten verändert werden, wodurch Testszenarien schnell angepasst werden können und viele unterschiedliche Testausprägungen möglich sind.

Aufbau der Arbeit

Nach der Einleitung behandelt Kapitel 2 den für die vorliegende Arbeit relevanten *State of the Art*. Hierbei befasst sich das erste Unterkapitel mit der Thematik der künstlichen Intelligenz und wie Systeme mit künstlicher Intelligenz kategorisiert werden können. Zur Verkörperung der künstlichen Intelligenz werden Software-Agenten eingesetzt. Grundlagen zu diesem Thema folgen in Kapitel 2.2. Anschließend werden zwei Beispiele - Turing-Test und *Chinese Room* - vorgestellt, die dazu beitragen die Problematik der Beurteilung von intelligentem Verhalten von künstlicher Intelligenz besser zu verstehen. Eine spezielle Ausprägung von Software-Agenten sind *Believable Agents*, die daran anschließend erläutert werden, da deren Bewertung unter anderem durch adaptierte Turing-Tests erfolgen kann. Beim *Artificial Recognition System* (ARS) in Kapitel 2.5 werden *Believable Agents* mit einem neuen Ansatz entwickelt und in einer Simulationsumgebung getestet. Diese Simulationsumgebung wird in der vorliegenden Arbeit erweitert. Die Erweiterung wird im Kapitel *Artificial Life Simulation* mit bereits existierenden Simulationen von künstlichem Leben verglichen. Die Implementierung der Erweiterung der Simulationsumgebung erfolgt mit einer *Belief-Desire-Intention*-Architektur, die im abschließenden Kapitel des *State of the Art* vorgestellt wird. Das Kapitel über den *State of the Art* zeigt schrittweise den Weg von einem System mit künstlicher Intelligenz über Agenten und Artificial Life Simulationen zu einer Belief-Desire-Intention-Architektur. Das verwendete Framework für die Implementierung der triebgesteuerten Fauna verwendet eine solche Architektur um eine Artificial Life Simulation mit Agenten zu erstellen, wodurch ein System mit künstlicher Intelligenz entsteht.

Kapitel 3 bildet die Grundlagen für das darauf folgende Kapitel 4. Diese beiden Kapitel bilden trotz der Trennung eine Einheit, da sie zusammen die Implementierung der triebgesteuerten Fauna vollständig beschreiben.

Kapitel 3 beschreibt zu Beginn jenen Standard, der beim verwendeten Framework Verwendung findet. Anschließend wird ein kurzer Überblick über das *Java Agent Development Framework* gegeben. Das Kernstück dieses Kapitels beschäftigt sich mit der verwendeten Funktionalität der Erweiterung dieses Frameworks (JADEX) in Kapitel 3.3. Dabei werden die Erkenntnisse der Arbeit mit JADEX und die wichtigsten Komponenten davon beschrieben. Dieses Kapitel liefert Informationen, auf die in den folgenden Kapiteln Bezug genommen wird und deren Verständnis bei der Lektüre von Kapitel 4 hilfreich ist. Außerdem enthält es die Beschreibung von Komponenten, die in der Implementierung der triebgesteuerten Fauna Verwendung finden. Während der Implementierung von Agenten unterstützen vor allem jene Tools, die in Kapitel 3.4 vorgestellt werden, den Entwickler. Das abschließende Kapitel befasst sich mit MASON, das zur Simulationssteuerung und Visualisierung verwendet wird. Da MASON, im Gegensatz zu JADEX, bereits im *Artificial Recognition System*-Projekt verwendet wird, ist dieses Kapitel bewusst kürzer gefasst.

In Kapitel 4 wird die entwickelte Fauna und deren Implementierung beschrieben. Zu Beginn wird die Analyse der Fauna erläutert. Anschließend erfolgt eine Auflistung der Eigenschaften der Tiere und Nahrungsquellen und eine Beschreibung von deren Verwaltung. Einen wichtigen Abschnitt stellt Kapitel 4.3 dar, da darin die Struktur des Programms erläutert wird. Hierbei wird - wie auch im darauf folgenden Kapitel - auf die Komponenten aus Kapitel 3 Bezug genommen. Eine Komponente des Programms stellen die verwendeten Agenten dar, deren Repräsentation in Kapitel 4.4 ersichtlich ist. Abschließend wird der Aufbau der Simulationsumgebung näher gebracht, wobei die Struktur und die Darstellung der Elemente erklärt werden.

Die Ergebnisse sind in Kapitel 5 zusammengefasst, wobei das Hauptaugenmerk auf den Ergebnissen der Implementierung liegt. Es werden Situationen in Simulationen gezeigt, die das Verhalten der Tiere veranschaulichen.

Den Abschluss bildet eine kurze Zusammenfassung der vorliegenden Arbeit und ein Ausblick auf Weiterentwicklungs- und Adaptionsmöglichkeiten.

2. State of the Art

In den folgenden Abschnitten wird der Stand der Forschung, der für die vorliegende Arbeit relevant ist, erläutert. Die Publikationen reichen hierbei von 1950 bis 2012, da ältere Publikationen, wie zum Beispiel der Turing-Test in einer aktuellen Thematik, wie zum Beispiel den *Believable Agents*, neu aufgegriffen werden.

Zu Beginn dieses Kapitels wird die Thematik der KI behandelt und in Teilgebiete gegliedert. Da im Zusammenhang mit KI und der Implementierung der vorliegenden Arbeit Software-Agenten zum Einsatz kommen, werden die unterschiedlichen Arten und Eigenschaften dieser Agenten in Kapitel 2.2 erläutert. Anschließend erfolgt ein Auszug über die Möglichkeiten Intelligenz zu messen (Kapitel 2.3) und glaubwürdige Agenten zu erstellen beziehungsweise zu testen (Kapitel 2.4). Das Umfeld der vorliegenden Arbeit ist durch das ARS-Projekt gegeben, das in Kapitel 2.5 vorgestellt wird. Ebenso wird ein Einblick in dessen Simulationsumgebung gegeben, die durch die triebgesteuerte Fauna erweitert wird. Beispiele für andere AL-Simulationen werden in Kapitel 2.6 vorgestellt und mit der in der vorliegenden Arbeit entwickelten Simulation verglichen. Die Agenten der triebgesteuerten Fauna folgen der BDI-Architektur, die im abschließenden Kapitel 2.7 behandelt wird.

2.1 Künstliche Intelligenz

Künstliche Intelligenz (KI; auch: AI) ist ein Schlagwort für eine sehr breit gestreute Thematik. Eine ausführliche Literaturrecherche zeigt, dass in Abhängigkeit unterschiedlicher Autoren verschiedene Bedeutungen dieses Begriffs vorliegen. Laut [PG07, S. 1] wird zumeist eine „narrow AI“ behandelt, also eine KI, die in speziellen Gebieten ihre Intelligenz unter Beweis stellt. Dabei ist sie zum Teil sehr erfolgreich in dem Bereich, für den sie entwickelt wurde. Allerdings ist das ursprüngliche Ziel der KI-Forschung eine *artificial general intelligence* (AGI). Dies ist eine Intelligenz, die unterschiedlichste komplexe Probleme in verschiedensten Umgebungen lösen kann. Hierbei soll sie sich selbst autonom, mit eigenen Gedanken, Ängsten, Gefühlen, Stärken, Schwächen und Neigungen kontrollieren. In [PG07, S. 1] wird das Erstellen einer AGI als schwieriger erachtet als die Entwicklung einer KI in einem speziellen eingeschränkten Anwendungsgebiet. Die Thematik der AGI ist die aktuellste Forschungsrichtung für die Entwicklung von Architekturen für KI und kommt im ARS-Projekt zum Einsatz. Bei der AGI wird eine sehr generelle Intelligenz angestrebt, was auch für die Kontrollarchitektur des ARS-Projekts gilt. Dieses Ziel wird jedoch in der vorliegenden Arbeit nicht

verfolgt, da die triebgesteuerte Fauna sehr spezielle Anforderungen erfüllt und einen vordefinierten Einsatzzweck hat.

Russel und Norvig unterscheiden in [RN04, S. 17-22] bei der Kategorisierung von KI vier Gruppen, die Systeme mit KI aus unterschiedlichen Winkeln betrachten. Diese werden im Folgenden vorgestellt.

2.1.1 Systeme mit menschenähnlichen Denkprozessen

Eine Art künstliche Intelligenz zu beschreiben, befasst sich mit der Nachbildung des menschlichen Denkens. Bei der Entwicklung von Systemen, mit menschenähnlichen Denkprozessen, stellt sich zuerst die Herausforderung, festzustellen, wie Menschen denken. Um diese Herausforderung zu meistern und damit eine technisch anwendbare Theorie des Verstandes zu entwickeln, werden unterschiedliche Methoden angewendet.

Mit Hilfe der Elektroenzephalografie (EEG) können Gehirnaktivitäten aufgezeichnet und analysiert werden. Solche Auswertungen ermöglichen die Zuordnung von Gehirnregionen zu bestimmten Aktivitäten des Menschen, insbesondere aber auch zu unterschiedlichsten intelligenten Handlungen wie zum Beispiel der Sprachverarbeitung. Da jedoch nur elektrische Spannungsschwankungen an der Kopfoberfläche aufgezeichnet werden, kann nicht beurteilt werden, wie eine Person denkt geschweige denn, was sie denkt.

Eine weitere Möglichkeit den menschlichen Gedanken auf die Spur zu kommen, bietet die Introspektion [SS97, S. 47]. Darunter versteht man in der Psychologie die Selbstwahrnehmung durch nach innen gerichtete Beobachtung. Es wird also versucht durch die Betrachtung, Beschreibung und Analyse des eigenen Erlebens und Verhaltens die eigenen Gedanken aufzufangen, während sie entstehen. Ein Problem bei der Analyse des menschlichen Denkens durch Introspektion ist die Möglichkeit der Selbsttäuschung während der Selbstbeobachtung [Pet84, S. 513]. Doch auch, wenn man von möglichen Problemen absieht, kann nur jener Teil der menschlichen Intelligenz durch Introspektion erfasst werden, der tatsächlich für eine Person selbst wahrnehmbar ist. Zum Beispiel ist die Anwendung der Introspektion eine Leistung des menschlichen Verstands, die so nicht analysiert wird.

Psychologische Experimente bilden eine weitere Möglichkeit das Verhalten von Menschen zu analysieren um dadurch Rückschlüsse auf die Denkvorgänge zu ziehen und so besser zu verstehen wie Menschen denken. Hierbei werden Versuchspersonen in zwei Gruppen geteilt, wobei nur bei einer Gruppe eine Aktion gesetzt wird. Anschließend kann die Auswirkung dieser Aktion ermittelt werden, indem die beiden Gruppen verglichen werden. Besonders wichtig ist, dass - abgesehen von der zu messenden Auswirkung - alle anderen Parameter konstant gehalten werden.

Sobald die Theorie des menschlichen Verstandes ausreichend genau ist, kann der Versuch unternommen werden, diese Theorie als Modell auszudrücken. Es ist also naheliegend, dass die auf diese Weise entwickelten Computerprogramme umso besser beziehungsweise menschenähnlicher sind, je weiter das Verständnis des menschlichen Denkens vorangeschritten ist.

2.1.2 Systeme, inspiriert vom menschlichen Agieren

Beim Versuch Systeme zu entwickeln, die vom menschlichen Agieren inspiriert sind, müssen gemäß Russel und Norvig [RN04, S. 19] zumindest die folgenden Eigenschaften erfüllt werden.

- Verarbeitung natürlicher Sprache:
  Die Verarbeitung einer natürlichen Sprache ermöglicht dem System Sprache zu erfassen und sinnvolle Antworten zu liefern. Ohne die Verarbeitung natürlicher Sprache kann keine intuitive Kommunikation mit dem System erfolgen.
- Wissensrepräsentation:
  Die Wissensrepräsentation ermöglicht dem System das Speichern der Informationen, über die es bereits verfügt und auch jener, die es im Zuge der Kommunikation aufnimmt.
- Automatisches logisches Schließen:
  Mit Hilfe des automatischen logischen Schließens können die gespeicherten Informationen verarbeitet und dadurch während der Kommunikation genutzt werden. Zusätzlich kann das System dadurch Fragen beantworten und auch neue Schlüsse ziehen.
- Maschinenlernen:
  Erst durch das Maschinenlernen kann das System auf neue Umstände reagieren und sich daran anpassen. Zusätzlich kann es Muster erkennen und extrapolieren.

Joseph Weizenbaum hat 1966 in [Wei66] ein System beschrieben, welches das menschliche Handeln imitiert. Er nannte sein Computerprogramm ELIZA. Dieses Programm simulierte anhand von festgelegten Regeln die Kommunikation mit einem Psychotherapeuten. Weizenbaum wählte das Gebiet der Psychotherapie, weil dadurch das Programm über kein Wissen über die Welt verfügen muss, aber trotzdem dem Anwender des Programms gegenüber glaubwürdig erscheinen kann. Die Regeln, die ELIZA für die Kommunikation anwendet, basieren auf der Analyse der Aussagen des Gesprächspartners. In erster Linie wird die Aussage des Anwenders als Frage umformuliert und dann als Antwort zurückgegeben. Zusätzlich wird die Aussage auf Schlüsselwörter untersucht, die dann zur Simulation von Verständnis verwendet werden können. Ein Beispiel hierfür ist die Interpretation der Wörter Vater und Mutter. Kommt eines dieser Wörter in einer Aussage vor, geht ELIZA davon aus, das eine Antwort mit Bezug auf die Familie angebracht sein könnte. Ein Beispiel aus [Wei66, S. 37] hierfür sind die folgenden Sätze.

Anwender: „My mother takes care of me."

ELIZA: „Who else in your family takes care of you?"

Einer der Hauptgründe warum Menschen von ELIZA getäuscht werden können, liegt laut [Wei66, S. 42] darin begründet, dass die Anwender psychoanalytische Gründe hinter den Fragen von ELIZA vermuten können. Die Kommunikationssituation, dass der Anwender sich in ein psychoanalytisches Gespräch versetzt fühlt und dadurch von sich aus sehr aktiv erzählt, erleichtert zusätzlich die Täuschung, dass der Gesprächspartner ein menschlicher Psychiater sei.

2.1.3 Systeme, die rationalem Denken nachempfunden sind

Systeme, die rationalem Denken nachempfunden sind, sollen Schlussfolgerungen beherrschen. Die natürliche Sprache muss dabei in eine Notation übersetzt werden, die vom System verstanden werden kann. Diese Schlussfolgerungen werden durch logisches Schließen erreicht. Ihr Denken basiert also auf den Regeln der Logik, welche die Entwickler solcher Computersysteme aufgrund ihrer Notation vor zumindest zwei Probleme stellen. Einerseits gibt es in der realen Welt Wissen, dessen Repräsentation mit Hilfe von Formeln nicht einfach zu bewerkstelligen ist. Zusätzlich erschwert Wissen, das nicht sicher ist oder dessen Gültigkeitsdauer nur kurz ist, die formale Beschreibung. Andererseits sind die Hardwareressourcen von Computersystemen begrenzt und ermöglichen dadurch nicht, dass jedes Problem, das theoretisch gelöst werden kann, auch bei der praktischen Bearbeitung eine Lösung hat.

Die Systeme verarbeiten eine Menge an gültigen Aussagen und probieren durch Anwendung der vorher festgelegten Regeln zu einer ebenfalls, nachweislich, gültigen Aussage zu gelangen. Der Vorgang des Probierens kann allerdings auch zu einem nie endenden Versuch werden, falls es zu einer Problemstellung keine Lösung gibt.

2.1.4 Systeme, die rational agieren

Ein System, das rational agiert, soll immer das „Richtige“ tun. Es entscheidet jeweils auf Basis des optimalen Ergebnisses, oder zumindest derart, dass das beste Ergebnis erwartet wird, falls es Unsicherheiten gibt. Da bei unsicheren Ereignissen keine Möglichkeit besteht optimal zu handeln, kann ein solches System nur versuchen eine möglichst gute Entscheidung zu treffen. Daher werden zum Beispiel mit Hilfe von Erwartungswerten und Schätzfunktionen die Möglichkeiten verglichen und die, im Sinne des Systems, beste ausgewählt.

Abgesehen davon gibt es Situationen ohne Zufälligkeit. In diesen Fällen entscheidet das System anhand von logischen Schlussfolgerungen welche Aktionen zum Ziel führen und führt diese Aktionen anschließend durch. Die Regeln, die zu logischen Schlussfolgerungen führen, sind klar definiert und allgemein gültig. Allerdings ist - ähnlich den Systemen, die rationalem Denken nachempfunden sind - die formale Beschreibung von Wissen um logische Schlüsse daraus zu ziehen nicht immer möglich. Zusätzlich kann die Komplexität der Systemumgebung das Ziel einer perfekten, oder zumindest möglichst guten, Entscheidung erschweren oder in der gegebenen Zeit sogar unmöglich machen, da die Hardwareleistung von Systemen begrenzt ist.

Die wichtigste Art eines Systems mit künstlicher Intelligenz ist in der vorliegenden Arbeit ein System mit menschenähnlichen Denkprozessen, da die künstliche Intelligenz im ARS-Projekt (siehe Kapitel 2.5) vom menschlichen mentalen Apparat inspiriert ist. Dieses Kapitel zeigt, dass der Begriff „künstliche Intelligenz” sehr starken Bezug zur menschlichen Intelligenz oder zu Rationalität hat. Rationalität wiederum hat ebenso einen Bezug zu menschlichen Entscheidungen, da Menschen bei Systemen mit künstlicher Intelligenz definieren, welche Entscheidungen rational sind. Die triebge-

steuerte Fauna fällt in keine der vier genannten Kategorien für Systeme mit künstlicher Intelligenz, weil das Verhalten der Tiere auf grundlegenden Trieben basiert. Allerdings kann man diese Fauna als System beschreiben, das vom tierischen Agieren inspiriert ist. Um künstlicher Intelligenz einen Körper zu bieten, werden Software-Agenten eingesetzt. Diese werden im folgenden Kapitel behandelt.

2.2 Software-Agenten

Ebenso wie bei der Definition von künstlicher Intelligenz, gibt es auch für die Definition von Agenten, unterschiedliche Sichtweisen auf die Thematik und umso mehr unterschiedliche Begriffsbestimmungen. Russel und Stuart definieren Agenten wie folgt: „Ein Agent ist alles, was seine Umgebung über Sensoren wahrnehmen kann und in dieser Umgebung durch Aktuatoren handelt.“ [RN04, S. 55].

In der vorliegenden Arbeit liegt der Fokus auf Software-Agenten. Für diese gilt im Speziellen, dass ihre Sensoren unter anderem Tastencodes, Dateiinhalte oder Netzwerkpakete sind und sie zum Beispiel einen Monitor zum Anzeigen ihrer Aktionen als Aktuator benutzen. Software-Agenten können auch durch das Drucken von Dateien oder den Versand von Netzwerkpaketen in ihrer Umgebung agieren.

Für Michael Wooldridge sind Software-Agenten Computersysteme, die autonom Aktionen in einer bestimmten Umgebung durchführen können um ihre Ziele zu erreichen [Wei99, S. 32]. Sie verwenden typischerweise Sensoren um Informationen über ihre Umwelt wahrzunehmen und verfügen über ein Repertoire an Aktionen um die Umgebung zu verändern, wobei die Umgebung gegebenenfalls nicht deterministisch auf diese Aktionen reagiert.

Görz, Rollinger und Schneeberger [GRS00, S. 949] definieren Software-Agenten ohne die explizite Erwähnung von Sensoren oder Aktuatoren:

„Ein Software-Agent ist ein längerfristig arbeitendes Programm, dessen Arbeit als eigenständiges Erledigen von Aufträgen oder Verfolgen von Zielen in Interaktion mit einer Umwelt beschrieben werden kann.” [GRS00, S. 949].

Diese Definition beinhaltet die Autonomie, die Orientierung auf Ziele und Aufträge und die Interaktion mit einer Umwelt. Bei der Beschreibung eines Agenten spielen aber auch andere Eigenschaften eine Rolle und helfen bei der Unterscheidung von Agentenarten. Görz, Rollinger und Schneeberger haben in [GRS00, S. 949-950] die folgenden Merkmale zusammengetragen, mit denen Agenten charakterisiert werden können.

- Andauernde Verfügbarkeit/Aktivität:
  Agenten, denen diese Eigenschaft zugesprochen wird, sind über einen längeren Zeitraum aktiv und reagieren auf Nutzer beziehungsweise andere Agenten.

- Interaktion mit einer Umwelt:
  Durch die Aufnahme von Informationen aus der Umwelt und entsprechende Reaktionen mit Auswirkung auf die Umwelt, können Agenten die Umwelt ihren Vorhaben entsprechend beeinflussen.
- Situiertheit:
  In Bezug auf die Einbettung der Agenten in eine Umwelt wird betont, dass komplexes Agentenverhalten auch als Ergebnis von direkten Reaktionen auf Umwelteinflüsse erzeugbar ist, sogenanntes „emergentes Verhalten“. Dies kann zu einfacheren Architekturen führen.
- Eigenständigkeit:
  Agenten handeln selbständig und werden nicht durch einen Nutzer gesteuert, ihre Handlungen können nach sehr komplexen Mustern ablaufen.
- Reaktivität:
  Hierunter ist das unmittelbare Reagieren eines Agenten auf die Einflüsse aus seiner Umwelt und die Interaktion mit dieser zu verstehen.
- Zielgerichtetheit:
  Agenten, die Ziele beziehungsweise Aufträge verfolgen, müssen sich unter Umständen über einen längeren Zeitraum hin an äußere Umstände anpassen.
- Pro-Aktivität:
  Ein Agent handelt eigeninitiativ und verfolgt seine Absichten ähnlich dem Verhalten zielgerichteter Agenten.
- Deliberatives Verhalten:
  Die explizit modellierte Auswahl von Zielen bzw. Absichten wird als Deliberation bezeichnet. Der Begriff wird als Gegensatz zu reaktivem Verhalten verwendet.
- Intelligenz:
  Hierunter werden Handlungen von Agenten verstanden, die von Menschen als intelligent bewertet werden würden.
- Rationalität:
  Sinnvoll erscheinende Entscheidungen von Agenten, welche in angemessener Zeit getroffen werden und die Verfügbarkeit eigener Ressourcen einbeziehen, werden als rational bezeichnet.
- Lernfähigkeit:
  Agenten sind in der Lage, Informationen zu speichern und können sie später für eigene Interaktionen nutzen. Hierfür müssen sie Fähigkeiten oder Entscheidungsprozesse gewinnbringend einsetzen können.
- Mobilität:
  Mobilität von Agenten findet hauptsächlich bei Middleware Verwendung und bezeichnet das selbständige Migrieren, inklusive der aktuellen Daten und dem Zugriff auf die dortigen lokalen Ressourcen, auf andere Plattformen.
- Kooperation:
  Hierunter wird das Zusammenwirken von Agenten und Menschen und / oder anderen Agenten verstanden.

- Wohlwollen:
  Agenten können in der Lage sein, Handlungen zu setzen, die selbstlos auf andere ausgerichtet sind und Wohlwollen beim Empfänger auslösen.
- Soziales Verhalten:
  Vorgegebene Normen und Regeln bestimmen das Verhalten der Agenten, sie agieren beziehungsweise reagieren abhängig von anderen Individuen in der Umwelt.
- Emotionales Verhalten:
  Bei Agenten mit emotionalem Verhalten beeinflussen - mitunter zusätzlich zu Zielen und Absichten - Emotionen das Verhalten von Individuen. Ihr Verhalten untereinander verbessert sich dadurch und ebenso können sie die Interaktion zwischen Agent und Mensch verbessern.
- Glaubwürdigkeit:
  Anhand des Aussehens ist erkennbar, welche Aktionen der Agent gerade durchführt und umgekehrt. Das heißt, ein Agent wird als Individuum erkannt, welches Ziele, Bedürfnisse und Emotionen aufweist.

Einige dieser Eigenschaften bezeichnen jeweilige Gegensätze anderer Eigenschaften, wodurch ein Agent nie alle Eigenschaften gleichzeitig haben kann. Anhand der Beschreibung eines Agenten mit Hilfe dieser Eigenschaften kann auch die Komplexität des Agenten abgeschätzt werden, da manche Eigenschaften leichter erfüllt werden können als andere. Nichtsdestotrotz liefert eine Agentenbeschreibung anhand dieser Kriterien nur eine Tendenz, da die Eigenschaften nicht nur erfüllt oder nicht erfüllt sein können, sondern jeweils sehr unterschiedlich stark ausgeprägt sein können.

Viele Aspekte eines Software-Agenten treffen auch auf Objekte einer objektorientierten Programmiersprache zu. So verfügen beide über einen inneren Zustand, können Aktionen durchführen und können über Nachrichten kommunizieren.

Zur Unterscheidung, wann es sich um einen Agenten handelt und wann ein Objekt einer objektorientierten Programmiersprache die gleichen Anforderungen erfüllen kann, hat Michael Wooldridge in [Wei99, S. 34-36] Unterscheidungsmerkmale ausgearbeitet. Ein entscheidender Aspekt ist der Grad der Autonomie. Für Objekte gilt, dass sie Variablen speichern, deren Zustand nur sie alleine verändern können, weil sie für andere Objekte nicht sichtbar sind. Um dennoch eine Interaktion mit anderen Objekten zu ermöglichen, können Objekte Methoden anbieten, die von anderen Objekten verwendet werden können. Da die Variablen, abgesehen von global definierten, in den Zustand des Objekts eingekapselt sind, verfügt das Objekt über die Autonomie über den eigenen Zustand. Allerdings verfügt es nicht über die Kontrolle des eigenen Verhaltens. Die Methoden, die anderen Objekten zur Verfügung gestellt werden, können von diesen aufgerufen werden, wann immer sie wollen. Das Objekt, das diese Methoden zur Verfügung stellt, hat dabei keine Möglichkeit zu entscheiden, ob es die Methode auch tatsächlich ausführen will oder nicht. Bei Agenten ist die Selbstbestimmung, ob eine Methode ausgeführt wird oder nicht, vorhanden. Methoden eines Agenten werden nicht von anderen Agenten aufgerufen, sondern deren Ausführung wird angefragt. Der Agent kann dann selbst entscheiden, ob er der Anfrage zusagt und die entsprechende Methode ausführt. Im Fall eines Objekts in einer objektorientierten Programmiersprache liegt die Entscheidung darüber, ob eine Methode ausgeführt wird, bei dem aufrufenden Objekt. Im Fall von Agenten liegt die Entscheidung beim

Agenten, der diese Methode bereitstellt, selbst. Die Unterscheidung von Agenten und Objekten wird von Wooldridge mit dem folgenden Spruch zusammengefasst: „*Objects do it for free; agents do it for money*.“ [Wei99, S. 35].

Agenten können flexibles Verhalten an den Tag legen, das sowohl reaktives und proaktives als auch soziales Verhalten sein kann. Diese Art von unterschiedlichem Verhalten ist bei Objekten prinzipiell nicht vorgesehen, kann aber selbstverständlich durch Erweiterung der Methoden eines Objekts erreicht werden, indem zusätzliche Funktionalität integriert wird, die flexibles Verhalten simuliert.

Ein weiterer Unterschied besteht in der Anzahl der Threads, welche die Aktionen bearbeiten. Obwohl es auch bei objektorientierten Ansätzen Möglichkeiten für multi-threaded programming gibt, ist es nicht vorgesehen, dass jedes Objekt einen eigenen Thread hat. Bei Agenten gilt, dass jeder Agent zumindest einen Thread für sich hat. Multi-Agenten-Systeme haben also grundsätzlich mehrere Threads.

Obwohl es klare Unterschiede zwischen Agenten und Objekten gibt, können Agenten auch mit Objekten simuliert werden, wenn die Eigenschaften von klassischen Objekten erweitert werden und zusätzliche Funktionalität ergänzt wird. So ist es möglich, dass Objekte zu Agenten werden. Dies liefert jedoch einen Widerspruch, da sie dann keine Objekte im klassischen Sinn sind.

2.2.1 Ablaufschritte eines Agenten

Grundsätzlich arbeiten Agenten laut [RN04, S. 55-61] zyklisch die folgenden drei Schritte ab, um ihre Aufgaben zu erfüllen. Manche Agenten können diese drei Schritte auch parallel ausführen.

- Informationen aufnehmen:
  Die Aufnahme von Informationen beinhaltet die Analyse der Umwelt, empfangener Nachrichten und auch das Beobachten von Auswirkungen des eigenen Handelns. Um Informationen aufnehmen zu können, werden Agenten mit Sensoren ausgestattet. Diese Sensoren können sehr unterschiedlich sein, da sie einerseits von den zu messenden Umgebungsfaktoren abhängen und andererseits von der Beschaffenheit des Sensors selbst. Zum Beispiel verfügen Roboteragenten über Hardware-Sensoren, Software-Agenten im Gegensatz dazu, über Software-Sensoren.
- Wissen verarbeiten und entscheiden:
  Anhand des jeweils aktuellen Informationsstands und dem Faktenwissen des Agenten wird das Wissen mit Hilfe von Schlussfolgerungen unter Umständen erweitert und die aktuelle Situation bewertet. Die daraus resultierende Wissensbasis nutzt der Agent um Entscheidungen zu fällen. Diese Entscheidungen können abgesehen von der unmittelbar nächsten durchzuführenden Aktion auch Pläne für zukünftige Handlungen betreffen.
- Aktionen ausführen:
  Aufgrund von zuvor getroffenen Entscheidungen führt der Agent Aktionen aus, die ihm dem Erreichen seiner Ziele näherbringen. Solche Aktionen können bei hardwarenahen Agenten im Ansteuern von Motoren oder anderen Aktoren resultieren, beziehungsweise bei rein softwarebezogenen Agenten zum Beispiel zu einer aktualisierten Darstellung von Ergebnis-

sen, Berechnungen oder Zuständen führen. Eine weitere nennenswerte Aktion ist die Kommunikation mit anderen Agenten. Dies kann zum Beispiel über den Versand von Nachrichten erfolgen.

Im speziellen für rationale Agenten gilt, dass sie sich in ihrer Umgebung so gut wie möglich verhalten. Sie streben danach das „Richtige" zu tun und handeln immer dann richtig, wenn ihr Verhalten dazu führt, dass sie so erfolgreich wie möglich sind. Sie versuchen die erwartete Leistung zu maximieren.

Rationale Agenten werden von Russel und Norvig wie folgt definiert: „Ein rationaler Agent soll für jede mögliche Wahrnehmungsfolge eine Aktion auswählen, von der erwartet werden kann, dass sie seine Leistungsbewertung maximiert, wenn man seine Wahrnehmungsfolge sowie vorhandenes Wissen, über das er verfügt, in Betracht zieht." [RN04, S. 59].

Um eine möglichst gute Leistung zu erbringen, müssen sie aus ihrer Wahrnehmung lernen und ihr Vorwissen über ihrer Umwelt durch ihre Erfahrungen erweitern.

Agenten können laut Russel und Norvig in jeder denkbaren Umgebung eingesetzt werden, allerdings ist das Abschneiden eines Agenten beim Versuch sich so gut wie möglich zu verhalten, durch die Art der Umgebung beeinträchtigt. Zur Beurteilung dieses Abschneidens wird eine Leistungsbewertung benötigt. Gemeinsam mit der Beschreibung der Sensoren und Aktuatoren beziehungsweise der Umgebung eines Agenten liefert die Leistungsbewertung eine vollständige Beschreibung des Agenten und dessen Aufgabe.

2.2.2 Umgebungen eines Agenten

Umgebungen sind laut Russel und Norvig [RN04, S. 62] Probleme, deren Lösungen die rationalen Agenten darstellen. Durch das Spezifizieren der Umgebung kann im Anschluss daran ein adäquater Agententyp gewählt werden. Je vollständiger die Umgebung spezifiziert werden kann, desto leichter wird die Entwicklung eines Agenten. Die Spezifizierung des Problems kann mit Hilfe der PEAS-Beschreibung (Performance, Environment, Actuators, Sensors) erfolgen. Dadurch ist definiert, welche Kriterien zur Leistungsbewertung herangezogen werden, in welcher Umgebung der Agent handelt, welche Aktuatoren ihm dafür zur Verfügung stehen und welche Sensoren ihm Informationen verschaffen. Die Leistungsbewertung ermöglicht einerseits die Bewertung des Abschneidens eines Agenten für andere und kann andererseits auch dem Agenten selbst dabei helfen bessere Entscheidungen zu treffen, wenn er die Auswirkungen seiner Aktionen schon vorab bewerten kann.

Russel und Norvig identifizieren in [RN04, S. 66-70] die folgenden Dimensionen um Umgebungen zu charakterisieren.

- Vollständig beobachtbar / teilweise beobachtbar:
  Ein Agent befindet sich dann in einer vollständig beobachtbaren Umgebung, wenn er mit seinen Sensoren zu jeder Zeit sämtliche Informationen der Umgebung wahrnehmen kann, die er für die Leistungsbewertung und Aktionsauswahl benötigt. Aspekte, die für die Tätigkeit des Agenten keine Rolle spielen, müssen also für die Charakterisierung als vollständig

beobachtbare Umgebung vom Agenten nicht beobachtbar sein. Aufgrund der ständigen Verfügbarkeit sämtlicher relevanter Daten über die Umgebung, muss der Agent keine internen Zustände verwalten.
Fehlen dem Agenten Informationen über die Umgebung, wird sie als teilweise beobachtbar bezeichnet.

- Deterministisch / stochastisch:
  Wenn der Agent aufgrund der aktuellen Informationen über die Umgebung und dem Wissen über die Auswirkungen seiner Aktionen den nächsten Zustand der Umgebung bestimmen kann, handelt es sich um eine deterministische Umgebung. Die Zustandsänderungen der Umgebung sind also durch vorbestimmte Regeln definiert.
  Verhält sich andererseits die Umgebung, oder Teile davon, zufällig, wird sie als stochastisch bezeichnet. Die Planung von zukünftigen Aktionen wird dadurch erheblich erschwert, da der Agent Unsicherheiten berücksichtigen muss.
  Falls der Agent keine vollständige Information über seine Umgebung hat, obwohl sie deterministisch ist, kann sie auf den Agenten wie eine stochastische Umgebung erscheinen.
- Episodisch / sequenziell:
  In episodischen Umgebungen muss der Agent jeweils nur den aktuellen Zyklus aus Bestandsaufnahme, Entscheidungsfindung und Aktionsdurchführung berücksichtigen und bewerten. Vergangene und zukünftige Episoden spielen keine Rolle für die aktuelle Entscheidung, wodurch die Entscheidungsfindung des Agenten vereinfacht wird.
  Bei sequenziellen Umgebung hingegen haben Kurzzeitaktionen Langzeitwirkungen. Der Agent muss vorausplanen und kann vergangene Aktionen und veraltete Informationen über die Umgebung für eine bessere Entscheidungsfindung verwenden.
- Statisch / dynamisch:
  Eine statische Umgebung ändert sich nicht, während ein Agent eine Entscheidung trifft. Die Leistungsbewertung des Agenten bezüglich seiner Aktionen bleibt also konstant.
  Im Gegensatz dazu ändert sich eine dynamische Umgebung ständig. Der Agent muss also jederzeit die Umgebung beobachten, während er Entscheidungen trifft, da sich aufgrund der geänderten Umgebung andere Aktionen als die ursprünglich getroffene als besser erweisen können.
- Diskret / stetig:
  Die Unterscheidung ob die Umgebung diskret oder stetig ist, kann auf mehreren Ebenen getroffen werden. Hierzu zählen der Zustand der Umgebung, die Behandlung der Zeit und die Wahrnehmungen beziehungsweise die Aktionen des Agenten. Bei diskreten Zuständen hat eine Umgebung eine endliche Anzahl an Zuständen und konkrete Zustandsübergänge. Im Gegensatz dazu gibt es für stetige Zustände unendlich viele Zustände mit fließenden Übergängen. Eine ebensolche Unterscheidung gilt für die anderen drei oben angeführten Ebenen.
- Einzel-Agent / Multi-Agent:
  Eine Umgebung kann einerseits nur von einem einzelnen Agenten beeinflusst werden und andererseits gleich von mehreren gleichzeitig. Die Unterscheidung, ob Einheiten im System als Agenten bezeichnet werden müssen, hängt von ihrer Fähigkeit rationell zu handeln ab. So lange ein Agent sich in einer Umgebung befindet, die nur stochastische Objekte beinhaltet, handelt er in einer Einzel-Agent-Umgebung. Befinden sich jedoch zumindest zwei selb-

ständig handelnde Agenten in der Umgebung, handelt es sich um eine Multi-Agenten-Umgebung. In einer solchen Umgebung können die Agenten sowohl konkurrierend, als auch kooperatives Verhalten an den Tag legen.

Für die Entscheidungsfindung eines Agenten, der rational handelt, sind Umgebungen, die teilweise beobachtbar, stochastisch, sequenziell, dynamisch und stetig sind und von mehreren Agenten beeinflusst werden, schwieriger zu behandeln als vollständig beobachtbare, deterministische, episodische, statische und diskrete Umgebungen, die nur von einem Agenten beeinflusst werden.

2.2.3 Struktur von Agenten

Die folgenden Abschnitte beschreiben grundlegende Arten von Agentenprogrammen, die über die Verwaltung und Verarbeitung einer Tabelle zur Entscheidungsfindung hinausgehen und daher in den meisten intelligenten Systemen Verwendung finden. Die beschriebenen Agententypen werden dabei von den einfachen Reflex-Agenten und den modellbasierten Reflex-Agenten über die zielbasierten Agenten und die nutzenbasierten Agenten bis zu den lernenden Agenten jeweils komplexer.

Abgesehen von einem Agentenprogramm, das die Agentenfunktion implementiert, welche wiederum die Wahrnehmungen des Agenten auf Aktionen abbildet, besteht ein Agent auch aus einer Architektur. Für Software-Agenten kann, laut Russel und Norvig [RN04, S. 70], ein Computer mit zugehörigen Sensoren und Aktuatoren als Architektur angesehen werden. Agentenprogramme müssen jeweils für die Architektur geeignet sein, da Aktionen des Programms sonst nicht ausgeführt oder erforderliche Informationen der Sensoren nicht empfangen werden können. Agentenprogramme wiederum nehmen nur die aktuelle Wahrnehmung der Sensoren auf und müssen die Verwaltung der Wahrnehmungsfolge selbst übernehmen, falls die Aktionen des Agenten davon abhängen. Hierfür kann der Agent eine Tabelle verwalten, die anhand der aktuellen Wahrnehmungsfolge eine entsprechende Aktion bereithält. Problematisch hierbei kann allerdings die benötigte Größe und das Befüllen der Tabelle werden, da für sämtliche Zustände entsprechende Aktionen eingetragen sein müssen.

Einfache Reflex-Agenten

Einfache Reflex-Agenten sind laut [RN04, S. 72] die einfachste Ausprägung von Agenten, da sie Aktionen nur aufgrund ihrer aktuellen Wahrnehmung auswählen und den bisherigen Verlauf der wahrgenommenen Umgebungszustände ignorieren. Sie handeln stets ihrem Regelwerk entsprechend, das aus *if-then*-Regeln besteht. Der Agent prüft also jeweils die Bedingungen einer Aktion und führt jene Aktion aus, deren sämtliche Bedingungen erfüllt sind. Falls keine Aktion durchgeführt werden kann, weil keine Regel für die aktuelle Situation zu finden ist, könnte der Agent in einer Endlosschleife verharren. Ein Ausweg aus dieser Sackgasse ist, den Agenten in solchen Situationen zufällige Aktionen durchführen zu lassen. Außerdem können Endlosschleifen vermieden werden, wenn die Umgebung vollständig beobachtbar ist und dem Agenten für alle möglichen Situationen Regeln zur Verfügung stehen.

Modellbasierte Reflex-Agenten

Modellbasierte Reflex-Agenten verwenden ein Modell ihrer Umgebung um mit nur teilweiser Beobachtbarkeit zu Recht zu kommen. Dieses Modell repräsentiert eine Beschreibung wie die Welt, in der der Agent sich befindet, funktioniert. Der Agent kann mit Hilfe des Modells den Teil der Umgebung, den er nicht sehen kann, anhand des Modells selbst verwalten und dadurch sehr effektiv die Unwissenheit über diesen Teil begrenzen, oder sogar eliminieren. Er muss gemäß [RN04, S. 75-76] zwei Arten von Wissen parallel speichern und verarbeiten können:

1. das Wissen, wie sich die Welt unabhängig vom Agenten entwickelt
2. das Wissen, wie sich die Aktionen des Agenten auf die Welt auswirken

So kann der Agent jederzeit seinen Wissensstand dieser beiden Arten aktualisieren und bei seinen Entscheidungen berücksichtigen.

Zielbasierte Agenten

Zielbasierte Agenten entscheiden - abgesehen vom aktuellen Zustand der Umgebung - anhand konkreter Zielvorgaben. Diese Zielvorgaben informieren den Agenten über Situationen, die er erreichen soll. Aufgrund der Informationen über die Auswirkung von Aktionen, kann der Agent sich für jene Aktionen entscheiden, die ihn näher an den gewünschten Zielzustand heranbringen. Das kann sich einfach gestalten, wenn eine Aktion alleine zum Ziel führt, kann aber mitunter sehr komplex werden, wenn eine Abfolge von vielen Aktionen zum Erreichen des Ziels notwendig ist. Um diese Abfolge von mehreren Aktionen zu identifizieren, verwenden Agenten laut [RN04, S. 77-78] Such- und Planalgorithmen.

Nutzenbasierte Agenten

Für einen nutzenbasierten Agenten bieten Zielzustände, die angestrebt werden, nicht immer ausreichend Entscheidungsgrundlage um die - für den Agenten richtigen - Entscheidungen treffen zu können. Das kann einerseits sein, wenn zwei Ziele zueinander in Konflikt stehen und dadurch nur jeweils teilweise erreicht werden können, und andererseits, wenn der Agent mehrere Ziele verfolgen kann, aber keines der beiden mit Sicherheit erreicht werden kann. Diese Probleme werden beim nutzenbasierten Agenten laut [RN04, S. 78-79] gelöst, indem der Agent als zusätzliche Information die Zustände der Umgebung, die aufgrund von zukünftigen Aktionen entstehen können, mit einer Nutzenfunktion bewertet. Diese Nutzenfunktion liefert dem Agenten eine reelle Zahl für jeden bewerteten Zustand, wodurch der Nutzen von Aktionen und daraus resultierenden Zuständen direkt verglichen werden kann. Diese Bewertung kann auch eine Zustandsfolge abbilden und verbessert dadurch die Planung zukünftiger Aktionen des Agenten. Der Agent wählt jeweils jene Aktion, die für ihn den größten Nutzen hat.

Lernende Agenten

Lernende Agenten erweitern ihren Wissensstand über ihre Umgebung, während sie darin agieren. Sie können selbst in einer anfänglich unbekannten Umgebung handeln und verbessern ihr Ausgangswissen stetig. Durch das immer ausgereiftere Wissen über die Umgebung können lernende Agenten im Laufe der Zeit kompetentere Aktionen durchführen.

Lernende Agenten bestehen bei [RN04, S. 79-82] im wesentlichen aus den folgenden vier konzeptuellen Komponenten, deren Zusammenspiel in Abbildung 1 veranschaulicht wird.

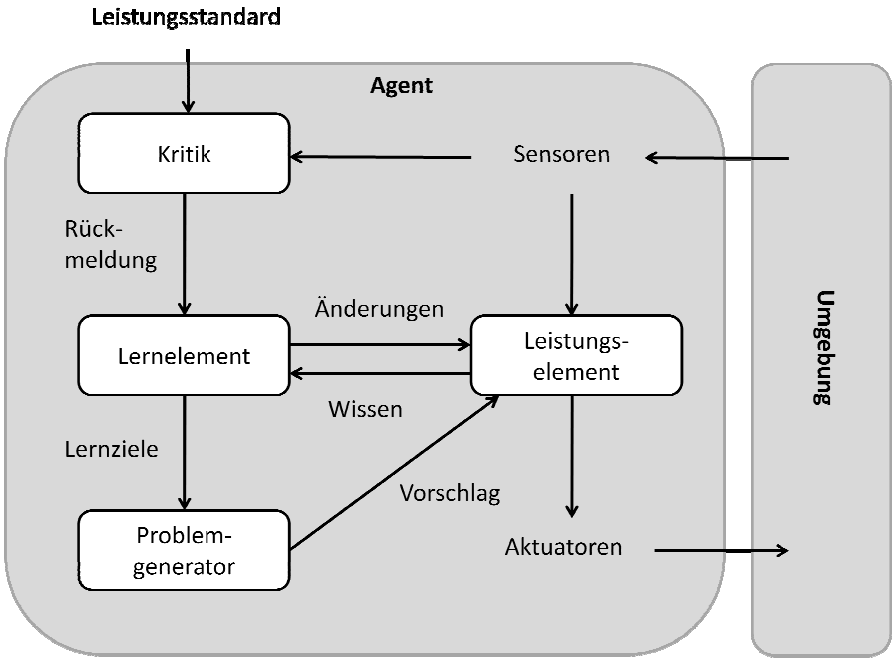

Abbildung 1 - Struktur eines lernenden Agenten [RN03, S. 53]

- Lernelement:
  Das Lernelement ist für die Verbesserung der getätigten Aktionen zuständig. Es verarbeitet das Feedback der Kritik und gibt Änderungen an das Leistungselement weiter um eine Verhaltensänderung zu bewirken. Außerdem leitet das Lernelement neue Lernziele an den Problemgenerator weiter.
- Leistungselement:
  Das Leistungselement ist für die Auswahl der jeweils durchzuführenden Aktionen verantwortlich. Es entscheidet anhand der Informationen, welche die Sensoren liefern, und den Verhaltensänderungen des Lernelements, welche Aktion in der jeweiligen Situation die beste ist. Diese Entscheidung wird vom Problemgenerator beeinflusst, der zusätzliche Aktionen vorschlägt.
- Kritik:
  Die Kritik bewertet das Verhalten des Agenten anhand der externen Leistungsstandards. Diese Standards ermöglichen dem Agenten zu beurteilen, ob die wahrgenommene Umgebungssituation einen Erfolg darstellt. Die Sensoren liefern dem Agenten die Informationen über die derzeitige Umgebungssituation und nach der Beurteilung dieser Situation wird Feedback an das Lernelement weitergegeben.

- Problemgenerator:
  Der Problemgenerator hilft dem Agenten neue Aktionen auszuprobieren. Er schlägt dem Leistungselement anhand der Lernziele, die er vom Lernelement erhält, neue Aktionen vor, die auf längere Sicht zu noch besseren Ergebnissen führen können, selbst wenn der kurzfristige Erfolg geringer ausfällt. Der Agent könnte nämlich neue Möglichkeiten ignorieren, wenn jeweils nur das Leistungselement über die durchzuführenden Aktionen anhand des aktuellen Wissensstandes entscheiden würde. Das Leistungselement würde dann jeweils nur die derzeit besten Aktionen durchführen und schlechteren Möglichkeiten keine Beachtung schenken.

Die Ausprägung der einzelnen Komponenten kann sehr unterschiedlich sein, wodurch sich viele unterschiedliche Lernmethoden ergeben können. Allen gemeinsam ist das Ziel der Verbesserung der Leistung des Agenten, welche durch die stetige Anpassung der einzelnen Komponenten anhand der verfügbaren Feedbackinformationen erreicht wird.

In der vorliegenden Arbeit wird auf Lernmethoden verzichtet, da die Agenten während einer Simulation stets das gleiche Verhalten zeigen sollen. Die Agenten der triebgesteuerten Fauna sind zielbasierte Agenten. Ihre Implementierung folgt den in Kapitel 2.2.1 vorgestellten Ablaufschritten eines Agenten. Die Agenten nehmen also Informationen auf, verarbeiten das Wissen, entscheiden daraufhin und führen Aktionen aus. Die Bewertung, wie intelligent die Aktionen von Agenten sind, und ob intelligentes Verhalten von Computern oder Computerprogrammen überhaupt möglich ist, wird in den folgenden zwei Kapiteln behandelt.

2.3 Messen von Intelligenz

Bei der Frage was künstliche Intelligenz ist, gilt es vorher zu definieren, wodurch sich Intelligenz kennzeichnet. Erst anschließend kann entschieden werden, ob ein System künstliche Intelligenz hat. Die folgenden zwei Kapitel widmen sich dem Messen von Intelligenz. In Kapitel 2.4 wird dieses Thema in Bezug auf *believable agents* weiter behandelt.

2.3.1 Turing-Test

Alan Turing hat sich 1950 in [Tur50, S. 433-437] mit dem Thema beschäftigt, ob Maschinen denken können. Allerdings müsste hierfür klar definiert sein, was unter den Begriffen „Maschine“ und „denken“ verstanden wird. Um die Mehrdeutigkeiten auszuschließen, formulierte er seine Frage „Can machines think?“ um. Er beschrieb die Situation zur neuen Fragestellung mit Hilfe von drei Personen, die ein Spiel spielen sollen - das sogenannte *Imitation Game*. Die Akteure des Spiels sind ein Mann, eine Frau und ein Fragesteller, dessen Geschlecht keine Rolle spielt. Der Fragesteller soll

herausfinden, wer von den beiden anderen der Mann und wer die Frau ist. Allerdings kennt er die beiden nur als Person X und Person Y. Um das jeweilige Geschlecht herauszufinden, darf der Fragesteller beiden Personen Fragen stellen. Zum Beispiel könnte er eine der Personen nach der Länge seines Haares fragen. Falls die Frage an den Mann gerichtet wurde, wird dieser versuchen, den Fragesteller in die Irre zu führen, da seine Aufgabe darin besteht, den Fragesteller zu täuschen. Für den Fall, dass die Frage an die Frau gerichtet wurde, wird diese alles daran setzen den Fragesteller zu überzeugen, dass sie die Frau ist, da ihre Aufgabe darin besteht, dem Fragesteller zu helfen. Allerdings könnten falsch gemeinte Hilfestellungen auch genauso gut vom Mann kommen.

Um die Entscheidung über das Geschlecht nicht durch die Tonlage der Stimme der Personen zu vereinfachen, könnte ein Mittelsmann eingesetzt werden oder die Kommunikation über einen Teleprompter stattfinden. Die neue Fragestellung von Alan Turing ergibt sich nun aus folgenden zwei Fragen:

> „Was passiert, wenn eine Maschine die Aufgabe des Mannes übernimmt?“
>
> „Wird sich der Fragesteller ebenso oft falsch entscheiden, wenn eine Maschine die Aufgabe des Mannes übernimmt, als wenn ein Mann und eine Frau dem Fragesteller antworten?“

Im Speziellen präzisiert Alan Turing, dass die Menge der Maschinen auf Digitalcomputer beschränkt werden soll. Es ist nicht erforderlich, dass alle Digitalcomputer, die es gibt oder geben könnte, dieses Spiel gleich gut wie Menschen beherrschen können. Gleiches gilt für einen speziellen Digitalcomputer, den es bereits gibt. Einzig relevant für Turing ist die Frage, ob es vorstellbar ist, dass es zukünftig zumindest einen Digitalcomputer gibt, der einem Menschen in diesem Spiel ebenbürtig ist.

Die von Turing aufgeworfenen Fragen und das von ihm beschriebene *Imitation Game* können als Turing-Test zusammengefasst werden. Mit Hilfe dieses Tests soll die Frage beantwortet werden können, ob ein Computer als intelligent gilt. In der Testsituation tritt der Tester über eine Tastatur und einen Monitor in den Dialog mit einem Computer. Fällt es dem Menschen hierbei schwer zu beurteilen, ob es sich um einen menschlichen Gesprächspartner oder um ein Computerprogramm handelt, kann der Computer als intelligent angesehen werden.

„Wenn ein Computer einen menschlichen Fragesteller dazu bringen kann, zu denken, sein Gesprächspartner sei ebenfalls eine Person, dann müsse der Computer definitionsgemäß intelligent sein.“ [Haw06, S. 23]

2.3.2 Chinese Room

John Searle entwickelte 1980 ein Gedankenexperiment um zu beweisen, dass Computer nicht intelligent seien und es auch niemals werden könnten [Haw06, S. 28-29].

In diesem Gedankenexperiment beschrieb er ein sogenanntes Chinesisches Zimmer. Hierbei handelt es sich um einen Raum mit einem Schlitz in einer Wand, in dem eine englischsprachige Person an einem Schreibtisch sitzt. Diese Person hat einen unbegrenzten Papiervorrat und ebenso unbegrenzt viele Stifte zur Verfügung. Außerdem hat sie ein dickes Buch mit Regeln, die beschreiben, wie chinesische Zeichen handzuhaben sind. Anhand der Regeln, die in Englisch verfasst sind, ist der Person

klar, wie die chinesischen Schriftzeichen kopiert, getilgt, neu angeordnet oder umgeschrieben werden müssen. Die Anwendung einer Regel ergibt dadurch mitunter die Notwendigkeit eine weitere Regel zu befolgen, die wiederum weitere Regeln implizieren kann, und so weiter, bis schließlich alle Regeln abgearbeitet sind.

Der Ablauf des Gedankenexperiments war wie folgt: Durch den Schlitz in einer Wand des Zimmers wird von einer Person außerhalb des Zimmers ein Zettel durchgeschoben. Auf dem Blatt steht eine Geschichte und dazugehörige Fragen, wobei alles in chinesischen Zeichen verfasst ist. Die englischsprachige Person im Chinesischen Zimmer versteht weder die chinesischen Zeichen noch die chinesische Sprache. Nichtsdestotrotz bearbeitet sie mit Hilfe des dicken Regelbuchs den Zettel, der in das Zimmer geschoben wurde. Die Person wendet also die Regeln des Buchs der Reihe nach an und hört erst damit auf, als in dem Buch steht, dass sie fertig ist. Als Resultat des kopieren, löschen, neu anordnen und umschreiben der chinesischen Schriftzeichen hat die Person ein weiteres Blatt mit chinesischen Schriftzeichen angefertigt, das die Antworten auf die Fragen zur Geschichte enthält. Die Person weiß aber nicht, was auf dem Papier steht, da sie kein chinesisch versteht, sondern einzig die Regeln des Buches angewandt hat. Der letzten Anweisung des Buches folgend, schiebt die Person das neu angefertigte Blatt durch den Schlitz nach draußen. Außerhalb des Zimmers nimmt eine Person, die des Chinesischen mächtig ist, den Zettel entgegen und überprüft die Antworten. Die Antworten stimmen alle und sind sogar sehr einfühlsam. Also würde diese Person davon ausgehen, dass eine intelligente Person, welche die Geschichte verstanden hat, die Fragen beantwortet hat.

Doch woher sollte diese Intelligenz kommen? Die Person im Zimmer, die kein chinesisch versteht und strikt nach den Anweisungen des Buches gehandelt hat, hat sicherlich nicht intelligent gehandelt, da sie ohne Buch keine einzige Tätigkeit ausgeführt hätte. Ebenso wenig kommt die Intelligenz von dem Buch, da es ja lediglich ein dickes Buch auf einem Schreibtisch ist. Die Papierzettel und der Schreibtisch können auch nicht als Quelle der Intelligenz angesehen werden. Für sie gilt die gleiche Argumentation, dass sie nur gewöhnliche Dinge in dem Zimmer sind.

John Searle behauptet, dass in diesem Prozess nirgendwo Intelligenz war. Es handelt sich bloß um geistloses Seitenumblättern und Papierbeschreiben.

Anhand dieses Gedankenexperiments will Searle zeigen, dass auch Computer nicht intelligent sein können, selbst wenn sie noch so gut darauf programmiert sind, intelligentes Verhalten zu simulieren. Die Person im Chinesischen Zimmer entspricht dem Prozessor eines Computers, das Buch dem Softwareprogramm und das Papier repräsentiert den Speicher. Dieser Computer kann also das gleiche Verhalten wie ein Mensch an den Tag legen und ist trotzdem nicht intelligent, da er nicht versteht, was er tut [Haw06, S. 28-29].

Das Gedankenexperiment des *Chinese Room* zeigt, dass intelligentes Verhalten von Programmen gezeigt werden kann, selbst wenn es nicht möglich ist, dass die Programme Intelligenz besitzen. Die Beurteilung, wie intelligent eine KI ist oder zumindest wie intelligent ihr gezeigtes Verhalten ist, bleibt Menschen überlassen. Bei der Bewertung von KI helfen Testverfahren wie das *Imitation Game* des Turing-Tests. Daran angelehnte Tests werden für *believable agents* (siehe Kapitel 2.4) verwendet. Die Ergebnisse der vorliegenden Arbeit liefern Entwicklern zusätzliche Möglichkeiten um ihre KI zu testen. Dadurch kann diese KI bei Messverfahren zur Ermittlung der Intelligenz - wie zum

Beispiel dem Turing-Test nachempfundene Tests - besser abschneiden als sie es ohne weitere Testmöglichkeiten würde.

2.4 Believable Agents

Bei *believable agents* handelt es sich um eine spezielle Form von Agenten. Sie haben das Ziel möglichst glaubhaft zu agieren. Die Glaubhaftigkeit wird dabei auf einen Vergleich mit einem Menschen bezogen. Ein Agent ist also dann glaubhaft, wenn er sehr ähnlich einem Menschen agiert. Ein Beispiel für eine Veranstaltung, bei der das Verhalten von Agenten mit jenem von Menschen verglichen wird, ist der *BotPrize*. Dieser wird im zweiten Teil dieses Kapitels vorgestellt.

Bei Software für Unterhaltung, Schulung und Trainings werden laut [NW10] virtuelle interaktive Charaktere eingesetzt um dem Anwender Interaktionsmöglichkeiten zu bieten. Diese Charaktere können einerseits von Menschen und andererseits von Software-Agenten gesteuert werden. Wenn Menschen die Steuerung übernehmen, sind damit im Vergleich zu computergesteuerten Charakteren, höhere Kosten verbunden. Die Interaktionsmöglichkeiten sind dann allerdings nicht eingeschränkt und von hoher Qualität. Bei Simulationen für Schulungen und Trainings ist ein hohes Maß an Glaubwürdigkeit von den Agenten gefordert, damit die Simulation realitätsnah ist.

Speziell bei vielen Computerspielen werden [MU11] zufolge *non-player-characters* (NPCs) - virtuelle Agenten, die von einer künstlichen Intelligenz gesteuert werden - eingesetzt. Diese dienen als neutrale Charaktere, Gegner oder Mitspieler. Hierbei wird der Erfolg der *believable agents* nicht daran gemessen, wie erfolgreich sie eigene Ziele im Spiel erreichen, sondern wie sehr der Agent realistisch - im Sinne von menschenähnlich - handelt und wie viel Spaß eine Interaktion mit diesem Agenten einem menschlichen Spieler macht. Allerdings ist es laut [UMR12, S. 42] für einen größeren Spielspaß ebenso erforderlich, dass ein Agent effektives Verhalten beim Verfolgen der eigenen Ziele zeigt und dadurch mit einem menschlichen Spieler in einen Wettstreit treten kann. Das Ziel bei der Entwicklung eines NPCs ist dementsprechend eine Mischung aus glaubwürdigem Verhalten und hoher Spielfertigkeit. Einerseits kann ein sehr glaubwürdiger Agent ein sehr schwacher Gegner für einen menschlichen Spieler sein. Andererseits ist ein Agent mit ausgezeichneten Spielfertigkeiten meistens nicht sehr glaubwürdig.

Glende beschreibt die folgenden sechs Aspekte, die beim Design eines realistischen Agenten Beachtung finden sollen, da sie für menschliche Spieler charakteristisch sind [Gle04].

1. Unberechenbarkeit:
   Obwohl menschliche Spieler manchmal, wie zum Beispiel beim Verwenden von favorisierten Routen in einem *first-person shooter* (FPS), vorhersehbar handeln, tendieren sie zu unberechenbarem Verhalten. Dieses Verhalten kann auch Teil einer Taktik sein um vom eigentlichen Vorhaben abzulenken.

2. Kreativität beim Lösen von Problemen:
   Menschliche Spieler haben eine enormes Repertoire an Problemlösungsstrategien und schaffen es meist leicht neue, kreative Strategien zu entwickeln um sich den Anforderungen einer Situation anzupassen.
3. Persönlichkeit:
   Um die Illusion von menschlichem Verhalten zu gewährleisten, soll jeder Charakter mit einer Persönlichkeit ausgestattet sein, die sein Verhalten bestimmt. Zum Beispiel würden besonnene Charaktere in einem *role-playing game* (RPG) Kämpfen eher aus dem Weg gehen als aggressive.
4. Autonomes Handeln:
   Da menschliche Spieler nicht nur auf Wahrnehmungen reagieren, sondern ein bestimmtes Ziel verfolgen, sollen auch die Aktionen von Agenten abhängig von Bedürfnissen, Wünschen und dem Wissensstand erfolgen. Die Agenten sollen also auch unabhängig von menschlichen Spielern agieren.
5. Improvisieren und planen:
   Beim Planen einer Strategie können menschliche Spieler zumeist schnell auf Veränderungen reagieren und ohne großen Aufwand durch Improvisation auf die neue Situation reagieren. Die Pläne werden dabei bei unvorhersehbaren Situationen ohne Mühe adaptiert, was bei Echtzeitspielen für Agenten eine große Herausforderung ist.
6. Lernen:
   Die Fähigkeit aus vorangegangenen Aktionen oder ganzen Spielen zu lernen, ist eine essentielle Fähigkeit von menschlichen Spielern. Aktionen, die nicht im gewünschten Ergebnis resultierten, werden zukünftig eher vermieden. Außerdem versuchen menschliche Spieler die Strategie der Gegenspieler aus bisherigen Erfahrungen vorherzusagen.

Um die Ähnlichkeit des Verhaltens eines Agenten im Vergleich zu einem Menschen beurteilen zu können, kann zum Beispiel eine dafür adaptierte Variante des Turing-Tests (wie in [Hin09]) verwendet werden. Ein Beispiel hierfür ist der *BotPrize*.

BotPrize

Das Ziel der Veranstaltung *BotPrize* ist das Erstellen und Testen von *Bots* (Agenten) für Computerspiele, die im besten Fall nicht von menschlichen Spielern unterschieden werden können. Für Computerspiele oder Simulationen werden solche *Bots* benötigt, da nicht alle Charaktere durch menschliche Akteure abgedeckt werden können. Für ein realistisches Erlebnis ist es erforderlich, dass das Verhalten der *Bots* dem menschlichen Verhalten nachempfunden ist.

Das Spielerlebnis ist laut einer Studie in [WWH+08] besser, wenn ein Spieler glaubt gegen einen menschlichen Spieler zu spielen. In dieser Studie spielten zwei Gruppen von Spielern ein Online-Rollenspiel gegen computergesteuerte Spieler. Der einen Gruppe wurde versichert, dass die Gegner von menschlichen Spielern gesteuert wurden. Der anderen Gruppe wurden die Gegner als computergesteuerte Spieler beschrieben. Jene Gruppe, die annahm, dass sie gegen menschliche Spieler spielte, empfand mehr Vergnügen beim Spielen und ein besseres Spielerlebnis. In [LR10] wurde außerdem festgestellt, dass die körperliche Erregung, bei ansonsten gleichen Interaktionen, beim Spielen gegen angeblich menschliche Spieler, im Vergleich zu computergesteuerten Spielern, höher war.

Um ein menschenähnliches Verhalten bei *Bots* zu erreichen, werden die Fähigkeiten der computergesteuerten Spieler laut [Hin10] bewusst verschlechtert. Sie könnten zum Beispiel bei einem *First-Person Shooter* (FPS) schneller schießen und bei jedem Schuss exakt treffen. Dieses Verhalten würde bei menschlichen Spielern allerdings sehr schnell zu der Annahme führen, dass sie einen *Bot* als Gegner haben. Daher wird die Reaktionszeit verlangsamt und die Zielgenauigkeit herabgesetzt.

Im Jahr 2008 wurde der erste *BotPrize* organisiert. Programmierer waren hierbei angehalten einen *Bot* für den FPS *Unreal Tournament 2004* zu erstellen. Dieser sollte eine Jury überzeugen können, dass es sich bei dem *Bot* um einen menschlichen Spieler handelt. Die Jury spielte zu Beurteilung der *Bots* jeweils eine Runde mit einem *Bot* und einem menschlichen Spieler und musste anschließend bestimmen welcher der beiden der *Bot* ist. Dieser Wettkampf wird von Hingston in [Hin09] beschrieben. Es handelt sich bei der Beurteilung der *Bots* in diesem Wettkampf um einen von Hingston für intelligente Agenten adaptierten Turing-Test (siehe Kapitel 2.3.1).

Manche *Bots* konnten Teilerfolge verbuchen indem sie einen oder zwei der fünf Jurymitglieder täuschten. Allerdings konnten die Jurymitglieder insgesamt sehr deutlich zwischen den *Bots* und den menschlichen Spielern unterscheiden. Die wichtigsten Charakteristika, die bei der Identifizierung der *Bots* verwendet wurden, waren laut [Hin10]:

- offensichtliches Fehlen von Planung
- kein konsistentes Handeln - Gegenspieler wurden vergessen
- steckenbleiben in einer Aktion
- statische Bewegungsabläufe
- sehr hohe Schussgenauigkeit
- zu wenig Achtsamkeit
- Unnachgiebigkeit
- Angriffe trotz Patt-Situation

Solche Verhaltensweisen ließen die Jurymitglieder darauf schließen, dass es sich bei einem Spieler um einen *Bot* handelte. Obwohl damit keine Sicherheit für die Beurteilung gegeben ist und auch menschliche Spieler als *Bot* deklariert wurden, haben diese Eigenschaften bei der Einschätzung eines Spielers geholfen.

Durch zusätzliche Testmöglichkeiten - wie sie die in dieser Arbeit entwickelte BDI Fauna für die ARS-Agenten ermöglicht - können manche der oben angeführten Charakteristika ausgiebiger getestet werden. Insbesondere die Punkte „kein konsistentes Handeln“, „zu wenig Achtsamkeit“ und „offensichtliches Fehlen von Planung“ können bei der Konfrontation von Agenten mit Tieren auftreten. Bei der Jagd auf ein Tier kann „kein konsistentes Handeln“ etwa dann auftreten, wenn kein Tier bis zum Erlegen verfolgt wird, sondern immer wieder ein anderes Tier als Ziel bestimmt wird und dadurch keines tatsächlich getötet wird. Im umgekehrten Fall kann „zu wenig Achtsamkeit“ schnell zum Tod des Agenten führen, wenn der Gefahr eines angreifenden Tieres zu wenig Beachtung geschenkt wird. Ein Beispiel für eine Situation, die den Punkt „offensichtliches Fehlen von Planung“ aufzeigen könnte, wäre eine Jagd von mehreren Agenten auf Tiere, wobei eine vorherige Planung des Angriffsverhaltens zielführender ist.

Im BotPrize des Jahres 2012 [6] konnten zwei Bots die Jurymitglieder soweit überzeugen, dass sie im Durchschnitt mit 52,2% und 51,9% bewertet wurden. Diese Zahlen bedeuten, dass die Jurymitglieder in jeweils ungefähr 52% der Situationen im Spielgeschehen diese Bots für menschliche Spieler gehalten haben. Umso bemerkenswerter ist die Tatsache, dass die menschlichen Spieler im Durchschnitt nur zu 41,4% Prozent als solche bewertet wurden. Die beiden Bots erreichten damit als erste Bots der bisherigen fünf BotPrize-Veranstaltungen, dass sie auf die Jurymitglieder menschlicher wirkten als die menschlichen Spieler.

Auch die Agenten des ARS-Projekts, das im folgenden Kapitel vorgestellt wird, sollen glaubhaft agieren. Die Beurteilung der Glaubhaftigkeit erfolgt dabei ebenfalls durch Menschen, da das Abschneiden der Agenten nur für spezifische Probleme und Einsatzgebiete anhand festgelegter Kriterien bewertet werden kann.

2.5 Artificial Recognition System

Im ARS-Projekt (*Artificial Recognition System*-Projekt) [2] des Instituts für Computertechnik der Technischen Universität Wien [1] wird an der Erstellung einer Kontrollarchitektur gearbeitet, die komplexe Situationen verarbeiten kann. Dieses wird in den folgenden Absätzen vorgestellt. Anschließend erfolgt ein Überblick über die ARSIN-World, die eine Simulationsumgebung für die künstlichen Agenten des ARS-Projekts darstellt.

Eine Herausforderung beim ARS-Projekt ist, große Datenmengen in kurzer Zeit zu verarbeiten und die wichtigsten Informationen daraus abzuleiten. Ein Modell der Natur, das diesen Anforderungen gewachsen ist, ist der menschliche mentale Apparat. Im ARS-Projekt wird versucht ein System zu entwickeln, dessen Informationsverarbeitung der des menschlichen mentalen Apparats nachempfunden ist. In [Sch12, S. 5] wird betont, dass hierbei insbesondere die unbewussten Prozesse in der menschlichen Informationsverarbeitung nachgebildet werden sollen, da diese in konventionellen AI-Projekten vernachlässigt werden, obwohl ein Großteil der menschlichen Fähigkeiten auf unbewussten Prozessen beruhen. Laut [Deu11a, S. 4] ist Psychoanalyse der beste Ansatz um ein ganzheitliches, funktionelles *top-down*-Modell für das ARS-Projekt zu entwickeln. Eine Beschreibung wie eine Kombination aus Aspekten der Neurobiologie, Psychologie und Psychoanalyse für diesen Zweck verwendet werden können, wird in [BDK+04] gegeben.

Ein Kernaspekt der Nachbildung des menschlichen Denkens wird aus Sigmund Freuds Strukturmodell der Psyche [Fre75] abgeleitet. Darin wird die Psyche des Menschen in drei Instanzen unterteilt: das Es, das Ich und das Über-Ich. Gemäß [Deu11a, S. 12] ist das Zusammenspiel aller drei Instanzen der Psyche für das ARS-Modell erforderlich. Diese werden in den folgenden Absätzen erläutert.

- Es:
  Das Es bezeichnet jenen Bereich der Psyche, der die Triebe, Bedürfnisse und Affekte enthält. Dieser Bereich ist bereits ab dem Zeitpunkt der Geburt vorhanden und unterscheidet

sich dadurch vom Ich und dem Über-Ich. Das Es strebt nach unmittelbarer Befriedigung der Bedürfnisse. Es handelt nach dem Lustprinzip und versucht Schmerz oder Unlust zu vermeiden. Dieses Handeln ist unbewusst und unterliegt keiner Bewertung ob es gut oder schlecht ist.

- Ich:
  Das Ich ist unter anderem für die Bewusstseinsleistungen verantwortlich. Dazu zählt das Wahrnehmen, das Denken und das Gedächtnis. Das Handeln des Ichs erfolgt nach dem Realitätsprinzip. Es versucht die Bedürfnisse des Es mit der Realität in Einklang zu bringen. Außerdem vermittelt das Ich zwischen dem Über-Ich und dem Es.
- Über-Ich:
  Im Über-Ich sind die sozialen Normen, also Gebote und Verbote, repräsentiert. Zusätzlich umfasst es Wertvorstellungen, Gehorsam, Moral und das Gewissen. Im Gegensatz zum Es, das von innen bestimmt wird und von Geburt an vorhanden ist, wird das Über-Ich von außen durch die Erziehung geformt. Durch die fortschreitende Formung des Über-Ichs werden die Triebregungen des Es kontrolliert und dadurch sozialgerechtes Verhalten ermöglicht.

Das Funktionale Modell des ARS, das in Abbildung 2 dargestellt wird, zeigt, dass die drei Instanzen der Psyche als Teile der *decision unit* implementiert werden.

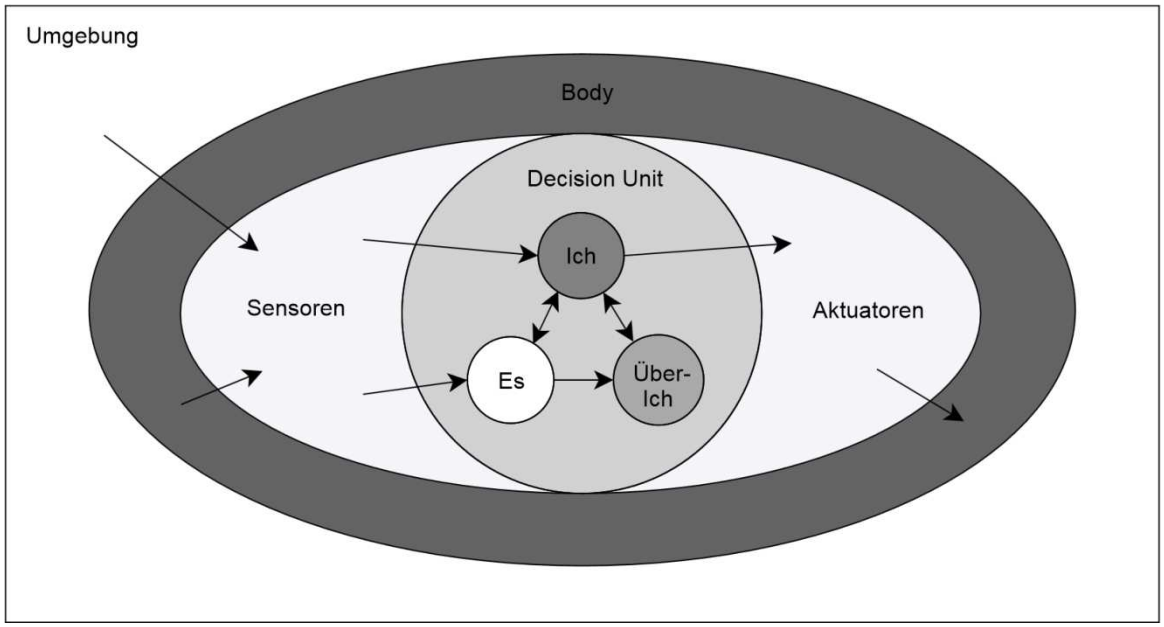

Abbildung 2 - Funktionales Modell des Artificial Recognition System [Zei10, S. 62]

Die Pfeile stellen den Informationsfluss zwischen den Komponenten dar. Innerhalb der *decision unit* ist zum Beispiel erkennbar, dass das Ich die Koordinierung der Ansprüche des Es und des Über-Ichs übernimmt. Das Es erlangt, ebenso wie das Ich, Daten der Sensoren. Diese Daten können einerseits aus der Umgebung stammen, andererseits aus der Körperhülle des Agenten (*body*). Das Ich ent-

scheidet abhängig von den Sensordaten, dem Es und dem Über-Ich, welche Reaktion an die Aktuatoren weitergeleitet wird. Diese wiederum beeinflusst den *body* des Agenten.

In Abbildung 3 werden die Haupt-Informationspfade des ARS-Modells [3] dargestellt. Die Interfaces verdeutlichen hierbei, wie die drei Instanzen der Psyche im ARS-Projekt in Verbindung stehen. Außerdem ist ein Überblick über die Funktionalität, die den einzelnen Bereichen zugeordnet wird, ersichtlich. Die Verarbeitung der Sensordaten erfolgt je nach Sensor im Es (Id) und im Ich (Ego), bevor die Funktionalität des Über-Ichs (Super-Ego) zum Einsatz kommt. Nach der Koordinierung des Ichs zwischen den Instanzen der Psyche und dem Übergang vom Lustprinzip zum Realitätsprinzip, werden schließlich die Entscheidungen an die Aktuatoren weitergegeben. Eine ausführliche Beschreibung des funktionellen Modells des ARS-Projekts geht über den Umfang der vorliegenden Arbeit hinaus. Weitere Informationen können bei [3], [Deu11a, S. 67-107] und [Sch12, S. 22-26] gefunden werden.

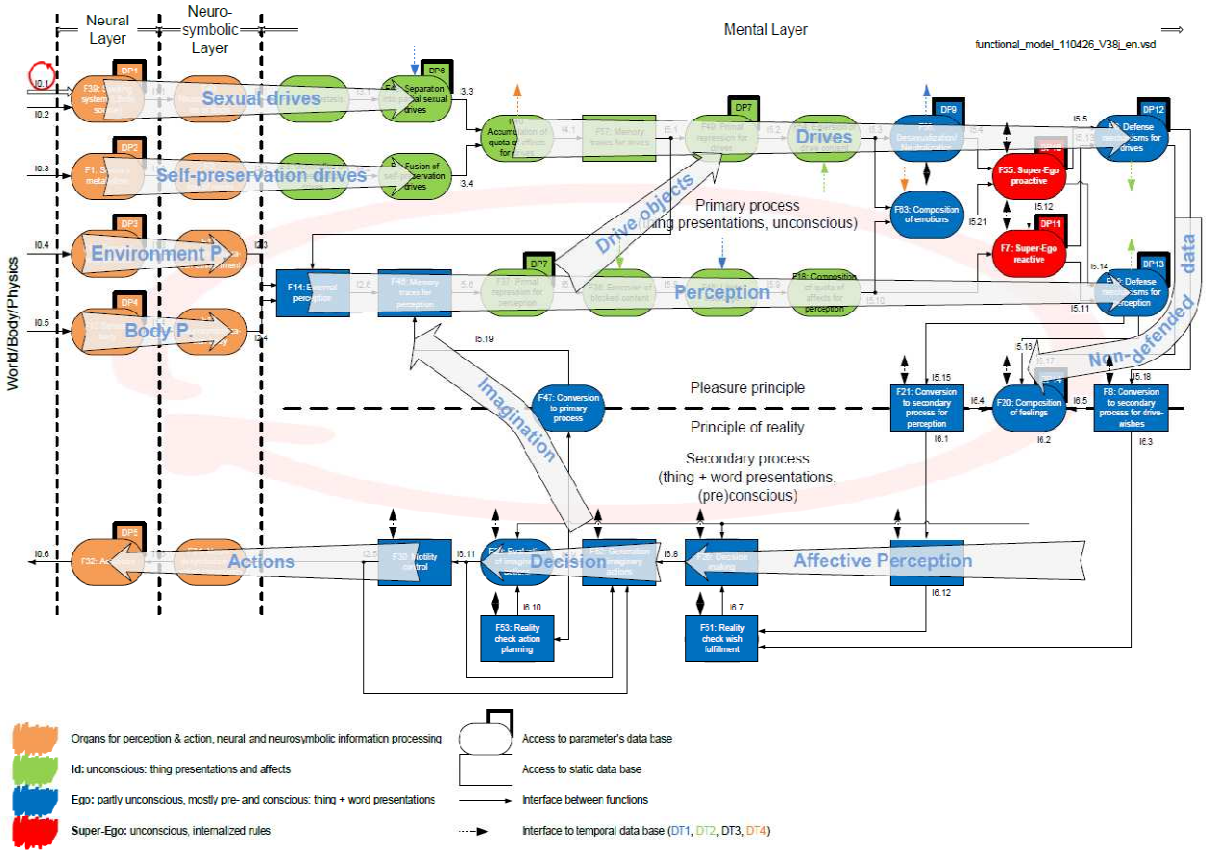

Abbildung 3 - Details des funktionellen Modells des Artificial Recognition System [Sch12, S. 26]

Das Psychoanalytische Modell des menschlichen Denkapparates für das ARS-Projekt, das in [DBMT09, S. 5-7] beschrieben wird, behandelt Motivation, Wunschgenerierung, Entscheidungsfindung, Planung und Ausführung. All das wird durch die Vermittlung zwischen unterschiedlichen

Begehren ermöglicht. Diese Vermittlung erfolgt zwischen Ich, Es und Über-Ich. Die Begehren werden in [DBMT09, S. 5] in die drei folgenden sehr unterschiedlichen Typen gruppiert.

- Begehren des Triebs:
  Damit werden jene Begehren, die auf physiologischen Anforderungen des Körpers basieren, zusammengefasst.
- Begehren der Realität:
  Diese Begehren bestehen aus internem Wissen über Fakten der Umgebungsrealität. Darunter fallen die Möglichkeiten und Einschränkungen in der Umgebung und subjektive Konsequenzen, die aus der Wahrnehmung der Umgebung resultieren.
- Begehren des Über-Ichs:
  Soziale und kulturelle Regeln stellen die Begehren des Über-Ichs dar. Zusätzlich aber auch die Bedrohungen durch Konsequenzen, die beim Bruch der Regeln drohen.

Da die Tiere der BDI-Fauna (siehe Kapitel 4) triebgesteuert sind und daher hauptsächlich die Begehren des Triebs in einer adaptierten Variante für Tiere Verwendung finden, beinhaltet die vorliegende Arbeit ausschließlich eine nähere Betrachtung der Begehren des Triebs. Im Gegensatz zur BDI-Fauna werden beim ARS-Projekt Triebe des Menschen behandelt. Daher sei an dieser Stelle erwähnt, dass der Begriff Trieb in Verbindung mit der BDI-Fauna lediglich Ursachen für Aktionen beschreibt, wie zum Beispiel Hunger oder das Schlafbedürfnis. Im Gegensatz dazu wird im ARS-Projekt ein Trieb umfassender definiert. In [Deu11a, S. 79-82] wird das Konzept der Triebe als Zusammenspiel von vier Komponenten beschrieben. Diese vier Komponenten bestehen aus der Quelle, dem Bestreben und einem Objekt des Triebs und dem Affekt.

- Die Quelle beschreibt die körperliche Ursache des Triebs, wie zum Beispiel ein Organ. Dessen Unausgewogenheit ist die treibende Kraft bei dem Trieb.
- Das Bestreben des Triebs ist es, jene Stimulation zu entfernen, die für die Aktivität des Triebs ursächlich ist.
- Durch das Objekt des Triebs kann die Unausgewogenheit entfernt werden. Es ist die variabelste Komponente eines Triebs.
- Der Affekt beschreibt, wie dringend die Befriedigung des Triebs ist und ist daher die treibende Kraft hinter einem Trieb.

ARSIN-World

Die Simulationsumgebung für die künstlichen Agenten des ARS-Projekts wird ARSIN-*World* genannt, da ein Agent als ARSIN bezeichnet wird. Die ARSIN World entstand aus der Notwendigkeit heraus, dass das Verhalten der Agenten getestet werden musste. Hierfür wurde eine Simulationsumgebung erstellt, die Objekte und Körperhüllen für die Agenten (oder auch Tiere) bereitstellt. In diese Hüllen wird die Entscheidungseinheit des ARS-Projekts eingebettet. Dadurch sind die Agenten ein Teil der Simulationsumgebung und können mit Hilfe von Sensoren ihre Umwelt wahrnehmen. Als Reaktion auf die Umwelt und seinen inneren Zustand trifft der Agent Entscheidungen, die nach Möglichkeit von Aktuatoren umgesetzt werden.

Die Implementierung der ARSIN-*World* erfolgte auf Basis von MASON (siehe Kapitel 3.5). Sie dient als Testumgebung für unterschiedliche Kontrollarchitekturen, wobei der Fokus laut Deutsch [Deu11a, S. 108] auf *Artificial General Intelligence* (AGI) Projekten liegt. Im Gegensatz zu spezialisierter AI wird bei der Entwicklung von AGI laut [WG06, S. 1] versucht die generelle Natur von menschlicher Intelligenz abzubilden. Da die ganzheitliche Kontrollarchitektur des ARS-Projekts dem AGI-Ansatz folgt, ist sie laut [Deu11b, S. 1] schwerer zu testen als spezialisierte Problemlösungen. Sie ist für dynamische, komplexe, reelle Umgebungen und Probleme geeignet und nicht nur für einen kleinen isolierten Anwendungsfall. Dadurch ist sie auch nicht mit anderen Kontrollarchitekturen direkt vergleichbar. Um AGI-Architekturen (insbesondere die psychoanalytische ARS-Architektur) zu testen, schlägt [Deu11b, S. 1] vor, dass eine virtuelle Welt simuliert wird, in der autonome Agenten mit Körperhüllen ausgestattet werden. Die Welt, in der die Agenten getestet werden, soll hierbei laut [Deu11b, S. 3] ebenso komplex sein wie das zu testende Modell.

Bei der Durchführung eines Tests wird eine Situation vorgegeben, in der ein Agent ein Verhalten zeigen soll, ohne zu wissen, welches Verhalten von ihm erwartet wird. Die Situation wird unter anderem durch die vorhandenen Ressourcen, die beteiligten Agenten und die Beschaffenheit der Umgebung selbst definiert. Die Evaluierung des Tests erfolgt durch einen menschlichen Beobachter, der überprüft, ob ein Agent das zu testende Verhalten zeigt.

In [Koh08, S. 40-57] werden *use cases* beschrieben, die es einem Tester ermöglichen, emergentes Verhalten der Agenten zu beobachten. Hierfür wird eine Situation textuell beschrieben, wobei die Akteure, die Vorbedingungen, der Ablauf, alternative Abläufe und erwartete Ergebnisse beschrieben werden. In [Koh08, S. 41] wird darauf hingewiesen, dass diese *use case*-Beschreibungen die wahrscheinlichsten Abläufe darstellen, aber auch komplett anderes Verhalten auftreten kann. Laut [Deu11b, S. 5] können *use cases* auch dann erfolgreich abgeschlossen werden, wenn der Agent keinen der beschriebenen Abläufe durchläuft. Das Gleiche gilt für nicht erreichte erwartete Ergebnisse. Da das Verhalten der Agenten menschlich sein soll, ist es nicht erforderlich, dass vordefiniertes Verhalten exakt vorhersehbar auftritt. Auch unerwartete Verhaltensweisen können als menschenähnliches Verhalten bewertet werden. Umso naheliegender ist daher, dass menschliche Beobachter für die Evaluierung des Verhaltens der Agenten benötigt werden.

Die ARSIN-*World* dient als *Artificial Life Simulation* der Beobachtung des Verhaltens der Agenten. Durch diese Beobachtung können Rückschlüsse darauf gezogen werden, ob die erwarteten Verhaltensweisen tatsächlich von den Agenten gezeigt werden. Im folgenden Kapitel werden weitere *Artificial Life Simulationen* vorgestellt und anschließend mit der triebgesteuerten Fauna der vorliegenden Arbeit verglichen.

2.6 Artificial Life Simulation

Im Gebiet des *Artificial Life* (AL) wird künstliches Leben auf der Basis von natürlichen Lebewesen kreiert. Dieses wird anschließend in einer Simulation getestet. Langton definiert AL wie folgt:

> *„Artificial Life is a field of study devoted to understanding life by attempting to abstract the fundamental dynamical principles underlying biological phenomena, and recreating these dynamics in other physical media - such as computers - making them accessible to new kinds of experimental manipulation and testing.“* [Lan92, S. xiv].

Laut Deutsch [Deu11a, S. 116] werden AL-Simulationen ebenso für agentenbasierte Modelle eingesetzt und zur Evaluierung von Kontrollarchitekturen verwendet, obwohl diese Definition ihren Fokus auf biologischen Phänomenen und Emergenz hat.

In den folgenden beiden Kapiteln werden drei Beispiele für künstliches Leben näher erläutert. Hierbei handelt es sich um den sehr bekannten zellularen Automaten von John Horton Conway, der *Game of Life* genannt wird. Außerdem wird der *Fungus Eater* von Masanao Toda beschrieben, der ein Beispiel für eine Agentensimulation darstellt. Das dritte Beispiel handelt von *Zamin*, einer zoologischen Agentensimulation. In Kapitel 2.6.4 werden diese Beispiele in Bezug zur vorliegenden Arbeit gesetzt.

2.6.1 Conway's Game of Life

Der Mathematiker John Horton Conway hat 1970 eine Umsetzung zur Automatentheorie von Stanisław Marcin Ulam entwickelt, die als Spiel des Lebens bezeichnet wird und erstmals in [Gar70] beschrieben wurde. Es handelt sich hierbei nicht um ein Spiel im klassischen Sinn, da es keine Spieler gibt und somit auch keine Gewinner oder Verlierer geben kann. Die einzige Aktion, die das Verhalten des Spiels beeinflusst, ist die Wahl einer Ausgangssituation. Anschließend bleibt nur noch die Möglichkeit die Entwicklung zu beobachten. Dabei entstehen mitunter komplexe Strukturen, die jedoch auf einfachen Regeln beruhen. Auf Makroebene kann der Eindruck entstehen, dass die Strukturen auf ihre Umgebung reagieren und sich reproduzieren können.

Das Spielfeld besteht aus einem zweidimensionalen Raster, der im Idealfall unendlich groß ist. Im Raster liegen quadratische Zellen nebeneinander, die nur zwei Zustände annehmen können: lebendig oder tot. Jede Zelle hat acht Nachbarn, die das eigene Verhalten der Zelle beeinflussen. Die acht Nachbarn ergeben sich aus zwei horizontalen und vertikalen Nachbarn beziehungsweise den vier diagonal benachbarten Zellen. Bei der Ermittlung des Zustands einer Zelle gelten folgende Regeln:

1. eine tote Zelle mit exakt drei Nachbarn wird zu einer lebendigen Zelle
2. eine lebende Zelle mit zwei oder drei Nachbarn bleibt lebendig
3. in allen anderen Fällen wird eine lebende Zelle zu einer toten Zelle

Die Anwendung dieser Regeln erfolgt in Schritten für alle Zellen gleichzeitig. Es muss also jeweils jene Situation für eine Zelle berücksichtigt werden, die gegeben ist, bevor die Regeln auf die erste

Zelle angewendet wurden. Anders gesagt: jene Situation, die aus dem Anwenden der Regeln auf alle Zellen im Schritt davor entstanden ist.

Die folgenden Abbildungen bieten Beispiele für die Anwendung der Regeln, wobei jeweils nur der Folgezustand der zentralen Zelle betrachtet wird.

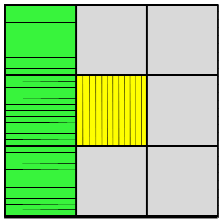

Abbildung 4 - Entstehung einer lebenden Zelle

Regel 1: In Abbildung 4 symbolisiert die gelbe Zelle (vertikale Linien) in der Mitte eine tote Zelle, die in der folgenden Generation zu einer lebenden Zelle wird, da genau drei ihrer acht möglichen Nachbarn lebende Zellen sind. Die grünen Zellen (horizontale Linien) stellen bereits lebende Zellen dar, die bezüglich ihrer Folgegeneration nicht weiter untersucht werden. Einzig ihre Auswirkung auf die zentrale Zelle ist für diese Betrachtung relevant.

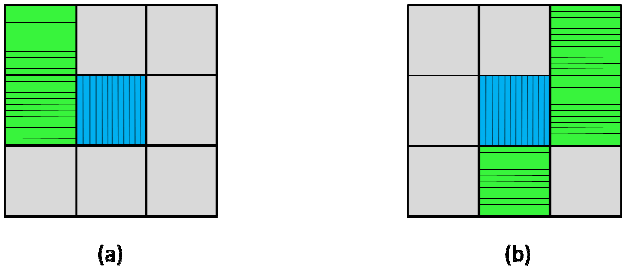

Abbildung 5 - Überleben von lebenden Zellen

Regel 2: Wie bereits zuvor, sind in Abbildung 5 die lebenden Zellen, deren Folgegeneration nicht weiter betrachtet wird, grün (horizontale Linien) eingefärbt. Die Abbildung veranschaulicht das Überleben einer lebenden Zelle, die genau zwei (a) oder drei (b) lebende Nachbarn hat, wobei nur der Folgezustand der bereits lebenden zentralen Zelle untersucht wird, die blau (vertikale Linien) gefärbt ist.

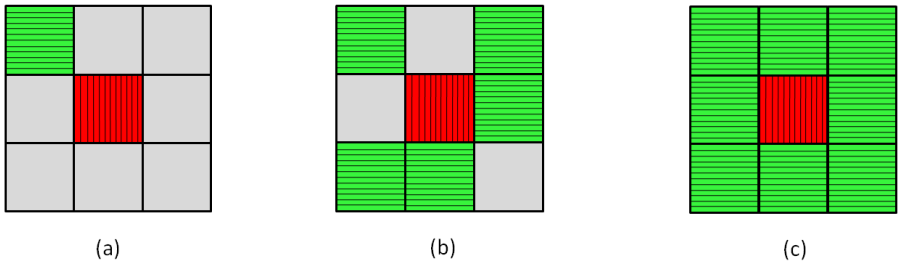

Abbildung 6 - Sterben von lebenden Zellen

Regel 3: Die Abbildung 6 zeigt Zellen, die wie folgt gekennzeichnet sind. Die rote Zelle (vertikale Linien), die sich jeweils in der Mitte befindet, repräsentiert eine Zelle, die lebt, aber in der Folgegeneration entweder aus Vereinsamung (a) oder aus Überbevölkerung (b, c) stirbt. Die grünen Zellen (horizontale Linien) stellen wiederum lebende Zellen dar, deren Entwicklung nicht untersucht wird, die aber die Entwicklung der zentralen Zelle beeinflussen.

Durch Anwendung der drei Regeln können verschiedenste Muster entstehen.

Manche Konstellationen ändern ihre Form nicht mehr, da die betroffenen Zellen einen stabilen Zustand erreicht haben, der nur noch durch die Einwirkung von einer nicht beteiligten Zelle verändert werden kann. Diese lebenden Zellen entsprechen auch nach Anwendung der Regeln ihren Zustand nicht, wodurch die derzeitige Generation auch der Folgegeneration entspricht. Die vier laut [4] häufigsten Beispiele für statische Objekte werden in Abbildung 7 veranschaulicht.

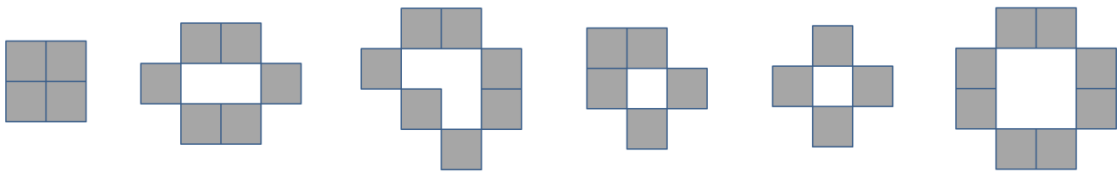

Abbildung 7 - Statische Objekte [4]

Oszillierende Objekte verfolgen ein Schema, das sich nach einer endlichen Anzahl an Schritten wiederholt. Sie bleiben, ebenso wie statische Objekte, in ihrer Form unverändert bis sie von nicht beteiligten Zellen beeinflusst werden. Die drei laut [5] häufigsten Beispiele für oszillierende Objekte mit zwei Zuständen, werden in Abbildung 8 veranschaulicht. Abgesehen davon gibt es auch oszillierende Objekte, die innerhalb von mehr als zwei Zuständen oszillieren. Ein Beispiel hierfür ist der sogenannte Fünfzehnkämpfer, der jeweils nach fünfzehn Zuständen wieder den Anfangszustand erreicht.

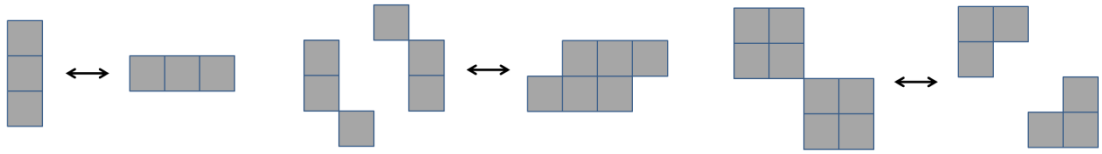

Abbildung 8 - Oszillierende Objekte [5]

2.6.2 Todas Fungus-Eater

Bei Todas *Fungus-Eater* handelt es sich um die Beschreibung einer Experimentalsituation von Masanao Toda aus dem Jahr 1982 [Tod82, S. 89-99]. Bei psychologischen Experimenten wird gemäß Toda [Tod82, S. 94] oftmals nur ein spezifisches Verhalten in einer vorgegebenen Laborsituation untersucht. Im Gegensatz dazu soll die Experimentalsituation die effiziente Aufnahme und Verarbeitung von Informationen durch Menschen zeigen. Es erfordert bei den Teilnehmern des Experiments Wahrnehmung, Lernen, Denken, Verhalten und die effektive Organisation dieser Aktivitäten gleichzeitig.

Als Umgebung für sein Experiment erfand Toda eine Science-Fiction-Welt [Tod82, S. 95]. Jeder Versuchsteilnehmer steuert einen Roboter, der *„Fungus-Eater“* heißt. Der Name des Roboters leitet sich von der Energiequelle für seinen biochemischen Motor ab, da dieser wilde Pilze für den Betrieb benötigt. Diese wilden Pilze sind überall auf der Oberfläche des Planeten Taros, auf den der Roboter entsandt wurde, verstreut. Ebenso auf der Oberfläche verstreut ist Urangestein, dessen Aufsammeln die eigentliche Aufgabe des Roboters ist. Schwarze und weiße Steine behindern ihn dabei, indem sie das Vorankommen erschweren.

Die Versuchsteilnehmer können die Bewegungen des Roboters steuern, aber auch die Empfindlichkeit seiner Sensoren verändern. Mit Hilfe der Sensoren können Pilze und Urangestein aufgespürt werden, wobei je mehr Energie verbraucht wird, desto sensibler die Sensoren arbeiten.

Der Auftrag besteht einzig darin viel Urangestein zu sammeln. Es ist dabei unerheblich wie viele Pilze gesammelt werden, da es dafür keine Belohnung gibt. Allerdings kann der Roboter nur dann Urangestein sammeln, wenn er ausreichend Energie zur Verfügung hat. Jede Aktivität verbraucht Energie, die den Pilzvorrat reduziert.

Der Versuch endet, wenn der Roboter keine Energie mehr hat, da er dann auch keine Aktion mehr durchführen kann um weitere Pilze aufzunehmen.

Die Teilnehmer werden schon zu Beginn des Szenarios in den Konflikt gebracht, ob Uran - das als einziges Leistungskriterium herangezogen wird - oder Pilze, die als Energiequelle dienen und für das Sammeln von Uran daher notwendig sind, gesucht und dann gesammelt werden soll. Die Entscheidung ist hierbei allerdings noch recht trivial, da bei ausreichend Energie das Augenmerk auf das Uran gerichtet sein kann. Bei zu wenig Energie ist es wiederum naheliegend die Pilze zu fokussieren. Schwieriger wird die Entscheidungssituation, wenn Hindernisse das Sammeln erschweren und

dadurch längerfristige Pläne erstellt werden müssen. Die dabei benötigte Gehirnaktivität verbraucht allerdings Energie, wodurch weniger Uran gesammelt werden kann.
Umso schwieriger wird die Experimentalsituation für jeden Teilnehmer, wenn sie sich mit anderen Teilnehmern gleichzeitig auf dem gleichen Planeten befinden und dadurch um die vorhandenen Ressourcen konkurrieren müssen. So ergeben sich nämlich unvorhersehbare Ereignisse durch die Aktionen der anderen *Fungus-Eater* und Pläne müssen entsprechend geändert werden. Zum Beispiel könnten die Pilze, die von einem *Fungus-Eater* zur Energiezuführung ausgewählt wurden, bis zum Erreichen der entsprechenden Stelle bereits von einem anderen *Fungus-Eater* aufgenommen worden sein. Ausgiebigere Planungen sind also erforderlich. Dadurch wird aber wiederum mehr Energie verbraucht, wodurch letztendlich weniger Uran gesammelt werden kann. Die Optimierung zwischen Energieverbrauch durch Einsatz des Gehirns, Einsammeln von Uran und der Aufrechterhaltung des Energiespeichers ist dadurch keineswegs trivial. Obwohl das Experiment auf wenigen Regeln und Aktionsmöglichkeiten basiert, entstehen spätestens durch die Multi-Agent-Situation - wenn mehrere *Fungus-Eater* gleichzeitig agieren - komplexe Entscheidungs- und Planungsprozesse, die aus der ursprünglichen Beschreibung nicht hervorgehen.

2.6.3 Zoological Agents for Modification and Improvement of Neocreatures

Zamin bedeutet einerseits „Erde“ auf persisch und andererseits ist es die Abkürzung für *„Zoological Agents for Modification and Improvement of Neocreatures“*. Halavati und Shouraki haben sich bei der Entwicklung von *Zamin* [HS02] zum Ziel gesetzt eine performante *high-level artificial life* Plattform zu kreieren, die für die Simulation von Evolutionsvorgängen von Organismen, die real existierenden ähnlich sind, geeignet ist. Hierfür wurde die bereits existierende Plattform ERL (Evolutionary Reinforcement Learning) [AL92, S. 487–507] als Basis verwendet. Die Plattform von Zamin wird im folgenden Abschnitt vorgestellt. Daran anschließend wird eine Umgebung für diese Plattform präsentiert, die für Simulationen Verwendung findet. Den Abschluss dieses Kapitels bildet eine adaptierte Variante von *Zamin*, die vollständig auf Agenten beruht.

Plattform von Zamin

Die Agenten in *Zamin* verwenden externe Sensoren, die die Daten bereits vorab auswerten, bevor sie an die *Processing Unit* weitergegeben werden. Ihre internen Sensoren messen das jeweils aktuelle Energielevel des Agenten und verfügen außerdem über Informationen bezüglich der verbrauchten Energie bei der letzten Aktion.

Um Pläne, die über eine Aktion hinausgehen, durchführen zu können, verwenden die Agenten Gene als Speicher. Weitere Gene individualisieren die Nahrungsvorlieben, den Energieverbrauch und die Bewegungsgeschwindigkeit der einzelnen Kreaturen. Ein Mutations-Controller, der selbst auch von Zeit zu Zeit einer Mutation unterliegt, verändert Gruppen von Genen anhand eines Faktors, der die Wahrscheinlichkeit der Mutation dieser Gengruppe beschreibt. Entscheidend für die Geschwindigkeit, in der neue Muster gelernt und erinnert werden können, ist die *Processing Unit*, die bei *Zamin* auf problembasiertes Lernen aufbaut, wobei aus vergangenen Erfahrungen Schlüsse für gegenwärtige Probleme gezogen werden.

Umgebung von Zamin

Auf Basis dieser Plattform haben Zadeh, Halavati und Shouraki in [ZSH04] eine Umgebung für *Zamin* beschrieben, die für Simulationen verwendet werden kann. Die Welt von *Zamin* besteht aus einem sphärischen Gitter, wodurch ein Lebewesen, das immer der gleichen Richtung folgt, wieder an ihren Startpunkt zurückgelangt. Jede Zelle in diesem Gitter kann nur von einem lebenden Lebewesen gleichzeitig belegt sein. Die drei unterschiedlichen Typen von Lebewesen in der Welt von *Zamin* werden in [HS03, S. 602-605] und [ZSH04, S. 1674-1676] detailliert beschrieben. Namentlich sind das *Aryos*, *Plants* und *Sentinels* und werden in den folgenden Punkten vorgestellt.

- Aryos:
  Aryos sind die wichtigsten Lebewesen in der Welt, da sie es sind, die beobachtet und dadurch erforscht werden sollen. Es handelt sich bei ihnen um mobile, lernende Organismen, die über ein Energielevel verfügen. Dieses Energielevel darf nie unter einen Mindestwert fallen, da sonst der Tod des Individuums eintreten würde. Das Energielevel kann erhöht werden indem ein Aryo Nahrung zu sich nimmt, die einerseits von Pflanzen, oder andererseits von toten Kreaturen stammen kann. Wird ein Aryo angegriffen, verliert es Energie. Ebenso muss es einen Teil seiner Energie abgeben, wenn es sich reproduziert. Für die Reproduktion wird ein Teil des Genmaterials des Aryos mutiert und anschließend an das Kind weitergegeben.
  Abgesehen von dem Energielevel verfügt ein *Aryo* mit Hilfe seiner internen Sensoren auch über Daten bezüglich der letzten Veränderung des Energielevels und sein Alter. Durch seine externen Sensoren erhält es Informationen über das nächste Objekt in seiner Umgebung. Handelt es sich hierbei um eine Pflanze, beschränken sich die Informationen auf die Entfernung, den relativen Winkel und die Energie der Pflanze. Ist ein anderer *Aryo* oder ein *Sentinel* das nächste Objekt, liefern die Sensoren zusätzliche Eigenschaften der Kreatur.
  Zu jedem Schritt in der Simulation kann sich ein *Aryo* für eine Aktion entscheiden. Mögliche Aktionen sind: in die derzeitige Richtung weitergehen, nach links oder rechts drehen, eine Reproduktion durchführen, eine andere Kreatur angreifen, eine Pflanze oder ein totes Lebewesen fressen oder einfach nur rasten. Bei jeder Aktion ergibt sich ein sogenannter *Pleasure*[1]-Wert, der beschreibt wie zufrieden ein *Aryo* mit der derzeitigen Situation ist. Für zukünftige Entscheidungen kann er dann vergangene *Pleasure*-Werte heranziehen um sich zwischen mehreren Aktionen zu entscheiden. Die Wahl wird hierbei mit größerer Wahrscheinlichkeit auf eine Aktion fallen, die in der Vergangenheit in einer ähnlichen Situation den *Pleasure*-Wert positiv beeinflusst hat, als auf eine Aktion, die einen negativen Einfluss auf den *Pleasure*-Wert nach sich gezogen hat. Durch die ständige Bewertung der Situation lernt ein *Aryo* sein Verhalten zu verbessern.

[1] Im ARS-Projekt wird der Begriff *pleasure* wie folgt definiert: „Pleasure is a measurable quantity that reinforces certain reactions and behaviors of a creature and constitutes an attractive purpose of actions the creature may plan and undertake.“ [Deu11a, S. 46]

- Plants:
  Pflanzen sind immobil und erfüllen nur den Zweck Nahrungsquelle für Aryos zu sein. Ihr einziges Attribut beschreibt die Energie, die sie einem Aryo liefert, wenn sie gefressen wird. Sie wachsen mit einem konstanten Faktor, haben aber eine limitierte Gesamtanzahl.
- Sentinels:
  Sentinels sind ebenfalls mobile Lebewesen und haben auch sonst viele Eigenschaften und Fähigkeiten der Aryos. Allerdings ist ihr Verhalten vorab festgelegt. Sie lernen im Laufe ihres Lebens nicht dazu und verfolgen jeweils nur ein Ziel: Aryos zu finden und anschließend zu töten. Durch Angriffe auf Aryos erlangen sie Energie um ihren Energielevel zu erhöhen und so dem Tod durch Energiemangel zu entgehen. Dies ist die einzige Form der Energiegewinnung für Sentinels, da sie nicht fressen.

Durch die evolutionären Möglichkeiten der *Aryos* entsteht laut [ZSH04] ein breites Anwendungsgebiet für die *Zamin artificial world*. Insbesondere für Studien, die sich mit der Evolution von Gattungen mit einem hohen Level mentaler und verhaltensbezogener Fähigkeiten befassen, kann *Zamin* als Testumgebung überzeugen.

Adaptierte Implementierung von Zamin

In [HSZ+04] wurde eine adaptierte Implementierung von *Zamin* vorgestellt, die vollständig auf Agenten beruht. Die Welt speichert hierbei nur die Positionen der Kreaturen und ihre Nachrichten. Sechs Agenten regeln das Erstellen und Entfernen von Kreaturen, verwalten die Zeit, überwachen die Positionen und Ressourcen, statten Kreaturen mit Informationen aus und führen ihre Wünsche aus. Für zusätzlich Tiere in der Welt müssen die Agenten nicht verändert werden, es muss lediglich eine Klasse implementiert werden, welche die Nachrichten der Agenten versteht. Durch die Anpassung von *Zamin* konnte die Menge des Codes um 30-50% verringert werden, wobei die Funktionalität vollständig beibehalten wurde [HSZ+04, S. 164]. Zusätzlich wurde die Modularität erheblich verbessert. Änderungen am Modell können dadurch mit weniger Aufwand durchgeführt werden.

2.6.4 Vergleich mit der triebgesteuerten Fauna für Artificial Life Simulation

Die triebgesteuerte Fauna für *Artificial Life Simulation* wird in der vorliegenden Arbeit als BDI-Fauna (siehe Kapitel 4) bezeichnet, da sie auf einer *Belief-Desire-Intention*-Architektur (BDI-Architektur, siehe Kapitel 2.7) aufbaut. Die Beispiele für künstliches Leben nähern sich vom zellulären Automaten (wie beim *Game of Life* in Kapitel 2.6.1) über den *Fungus Eater* (von Kapitel 2.6.2) bis zu *Zamin* (2.6.3) an die Anforderungen der BDI-Fauna (siehe Kapitel 1, Fragestellung der Diplomarbeit) an.

Im *Game of Life* wird anhand von Regeln entschieden wann die einzelnen Teile der Simulation überleben und wann nicht. Diese Entscheidung bildet die Basis für die BDI-Fauna, da hier jeder Agent primär danach strebt zu überleben. Von diesem Bestreben werden die Entscheidungen beeinflusst.

Beim *Fungus Eater* wird ein weiterer Aspekt der BDI-Fauna behandelt: Das Sammeln von Nahrung. Um das eigene Überleben zu gewährleisten, müssen die Agenten Nahrung zu sich nehmen. Abgese-

hen von der Notwendigkeit der Nahrungsaufnahme wird der Agent vor zusätzliche Probleme gestellt: er muss sich entscheiden, wann er Nahrung sammelt und wann der Fokus auf Uran liegt. Hierbei ist er allerdings auf sich gestellt und muss nicht in Betracht ziehen von anderen Agenten getötet zu werden.

Der Aspekt der Gefahr durch andere Agenten wird in *Zamin* behandelt. Diese zoologische Simulation weist große Ähnlichkeiten mit der BDI-Fauna auf. Die offensichtlichste Gemeinsamkeit ist, dass bei beiden Simulationen Agenten, die Tiere repräsentieren, die Protagonisten sind. In beiden Simulationen geben Regeln die Rahmenbedingungen der Simulation vor. Ebenso wird das Sammeln von Nahrung für das eigene Überleben, mit der Gefahr von anderen Agenten getötet zu werden, vorausgesetzt.

Die Gestaltung der BDI-Fauna mit einer ausführlichen Beschreibung der Simulationsumgebung wird in Kapitel 4 behandelt. Das folgende Kapitel erläutert die *Belief-Desire-Intention*-Architektur im Allgemeinen.

2.7 Belief-Desire-Intention-Architektur

Eine *Belief-Desire-Intention*-Architektur (BDI-Architektur) geht auf Überlegungen zu einem Modell des *human practical reasoning* von Michael Bratman [Bra87] zurück und leitet sich daher grundlegend von Modellen des menschlichen Schlussfolgerns ab. *Belief* steht für die Annahmen eines Agenten und repräsentiert das Umweltmodell. Die Gesamtheit der *beliefs* eines Agenten enthält sein Wissen über die Welt in der er sich befindet. *Desire* bildet die Wünsche beziehungsweise Ziele eines Agenten ab, wobei nicht jeder Wunsch in einer Absicht resultiert. Diese Absichten werden *intentions* genannt. Das sind die Verpflichtungen, für die sich der Agent entschieden hat. Sie enthalten die Aktionen, die der Agent tätigt um seine Ziele zu erreichen. Außerdem verfügen BDI-Architekturen über eine *event queue*, welche die auftretenden Ereignisse aufnimmt. Diese Ereignisse können entweder von externen Faktoren ausgelöst werden, wie zum Beispiel Sensoren, oder aus internen Abläufen resultieren.

In [Wei99, S. 57-58] wird eine generische BDI-Architektur vorgestellt, die aus sieben Hauptkomponenten besteht und in Abbildung 9 veranschaulicht ist:

1. Eine *belief revision function*, die aufgrund der aktuellen Wahrnehmung und den *beliefs*, auf Basis dieser, neue *beliefs* erstellt.
2. Die aktuellen *beliefs*, die den Wissensstand des Agenten über seine Umwelt enthalten.
3. Eine *option generation function*, die anhand der *beliefs* und *intentions* die verfügbaren *options* (*desires*) des Agenten ermittelt.
4. Die Liste der aktuell möglichen *desires* des Agenten.
5. Eine *filter function*, die den *deliberation process* repräsentiert. Damit ermittelt der Agent abhängig von den aktuellen *beliefs*, *desires* und *intentions* die aktuellen *intentions*.
6. Die Liste der aktuellen *intentions*, die den aktuellen Fokus des Agenten beschreiben.

7. Eine *action selection function*, bei der aufgrund der aktuellen *intentions* eine Aktion ausgewählt wird, die durchgeführt wird.

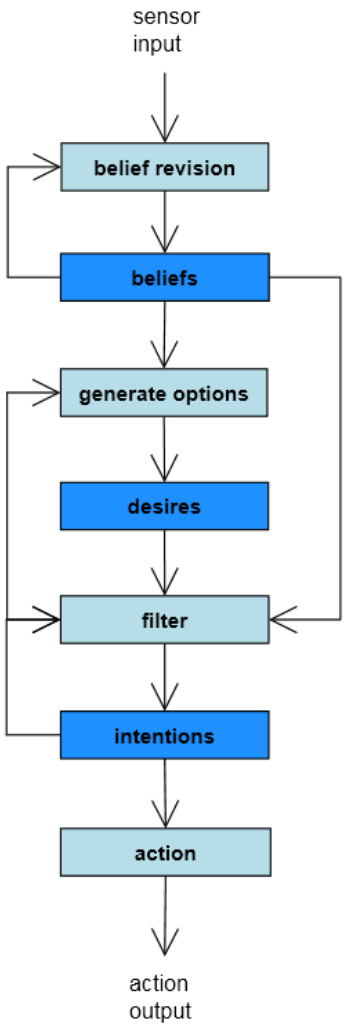

Abbildung 9 - Generische BDI-Architektur [Wei99, S. 58]

Die konkretere Implementierung einer BDI-Architektur wird in [MDA05, S. 9] mit dem typischen *execution cycle* einer BDI-Architekturen durch die folgenden Schritte angegeben:

1. Beobachten des aktuellen Zustands der Welt und des inneren Zustands des Agenten. Als Konsequenz der Beobachtung wird die *event queue* aktualisiert.
2. Das Erzeugen von neuen Plan-Instanzen, deren *trigger event* mit einem Event in der *event queue* übereinstimmt - diese werden relevant *plan instances* genannt. Anschließend die Einschränkung dieser Plan-Instanzen auf jene, deren Vorbedingung erfüllt ist - diese werden *applicable plan instances* genannt.
3. Auswahl einer Instanz der *applicable plan instances* für die Ausführung

4. Weiterleitung der ausgewählten Instanz auf einen bestehenden oder einen neuen *intention stack*.
5. Abarbeiten eines *intention stacks*, indem der nächste Planschritt einer Plan-Instanz durchgeführt wird.

Auf die Umsetzung der BDI-Architektur in JADEX, das für die vorliegende Arbeit verwendet wurde, wird in Kapitel 3.3 eingegangen. Eine BDI-Architektur bietet eine Möglichkeit den Entscheidungsprozess von Agenten strukturiert und schlank zu gestalten. Dies bewirkt bei der Entwicklung der triebgesteuerten Fauna, dass Mehraufwand für nicht benötigte Elemente vermieden wird beziehungsweise die Agentendefinition übersichtlich strukturiert ist. Ein Beispiel für Mechanismen, die von BDI-Architekturen nicht unterstützt werden, aber in der vorliegenden Arbeit auch nicht benötigt werden, sind Lernmechanismen. Die Anforderungen an die künstliche Intelligenz der triebgesteuerten Fauna beschränken sich auf grundlegende Verhaltensweisen von Tieren. Im Vergleich zu den Agenten des ARS-Projekts, deren Entscheidungsfindung vom menschlichen Denkapparat inspiriert ist, benötigen die Agenten der triebgesteuerten Fauna eine weitaus weniger komplexe künstliche Intelligenz. Die Kapitel des *State of the Art* zeigen, dass künstliche Intelligenz sehr stark von der menschlichen Intelligenz geprägt ist. Die Aussage, ob beziehungsweise in welchem Ausmaß Tiere intelligent sind, ist kein Thema der vorliegenden Arbeit. Hier wird das Verhalten der Tiere als triebgesteuert bezeichnet, wobei der Begriff „Trieb" als Ursache für Aktionen im Zusammenhang mit körperlichen Bedürfnissen verwendet wird. Für die Agenten des ARS-Projekts ist jedoch die Leistungsfähigkeit ihrer künstlichen Intelligenz von Bedeutung. Da Intelligenz für Computerprogramme, also auch Software-Agenten, nicht für allgemeine Probleme und Einsatzgebiete bewertet werden kann, erfolgt die Beurteilung der künstlichen Intelligenz durch Menschen. Diese Vorgangsweise wird bei den Agenten des ARS-Projekts aufgrund ihrer vielfältig einsetzbaren künstlichen Intelligenz ebenfalls angewandt. Hierfür wird eine *Artificial Life Simulation* verwendet, die durch die vorliegende Arbeit erweitert wird. Die Erweiterung erfolgt im Gegensatz zur Architektur des ARS-Projekts durch eine *Belief-Desire-Intention-Architektur*. Diese Architektur wird von JADEX unterstützt, dessen Einsatz für die vorliegende Arbeit vorgegeben ist. Im folgenden Kapitel werden neben JADEX weitere Komponenten vorgestellt, die bei der Entwicklung der triebgesteuerten Fauna verwendet wurden.

3. Framework

Das BDI-*Framework* JADEX [PBL03] wurde zur Realisierung der agentenbasierten Simulation eingesetzt. Die folgenden Kapitel bieten einen Überblick über jene Bestandteile der JADEX-Distribution, die in der vorliegenden Arbeit verwendet wurden, und beschreiben Prinzipien und Konzepte des *Frameworks* für die Implementierung der vorliegenden Arbeit. Die Implementierung besteht aus mehreren Komponenten, wobei der wichtigsten Komponente, der BDI-Fauna, ein separates Kapitel (Kapitel 4) gewidmet ist. Für die Beschreibung der BDI-Fauna wird auf Aspekte und Bestandteile der anderen Komponenten der Implementierung Bezug genommen. Durch die Trennung von Kapitel 3 und 4 kann auf wiederkehrende Elemente übersichtlicher verwiesen werden. Diese Elemente werden in den nun folgenden Kapiteln behandelt. Ein wesentlicher Bestandteil des JADEX-*Frameworks* ist JADE, dem ein separates Kapitel (Kapitel 0) gewidmet wurde. Da sowohl JADE als auch JADEX die Spezifikation der *Foundation for Intelligent Physical Agents* (FIPA) erfüllen, wird diese zu Beginn (3.1) vorgestellt. Für die Implementierung von Agenten mit JADEX wurden *Tools* des *Frameworks* verwendet, die in Kapitel 3.4 vorgestellt werden. Für die Visualisierung der Agenten wurde der Simulator MASON (siehe Kapitel 3.5) verwendet, da er auch im ARS-Projekt Verwendung findet.

3.1 Foundation for Intelligent Physical Agents

Die *Foundation for Intelligent Physical Agents* (FIPA) [7] ist eine Organisation, die Spezifikationen für intelligente Agenten erstellt. Diese Spezifikationen dienen der besseren Zusammenarbeit von Agenten oder agentenbasierten Anwendungen zwischen unterschiedlichen Systemen.
Ein Standard der FIPA ist die FIPA *Agent Management Specification* [8], welche offene Standard-Interfaces für den Zugriff auf *Agent Management Services* beschreibt. Ein Teil der Spezifikation ist das *Agent Management*-Referenzmodell, das in Abbildung 10 ersichtlich ist. Es beschreibt logische Strukturen, die für die Erstellung, Registrierung, Lokalisierung, Kommunikation, Migration und Löschung von Agenten verwendet werden sollen. Die tatsächliche Ausgestaltung der *services* liegt hierbei beim Entwickler der *Agent Platform* (AP). Es besteht aus den folgenden logischen Komponenten.

- Agent:
  Ein Agent ist der autonome Teil in einer AP. Er kommuniziert mit Hilfe der *Agent Communication Language* (ACL) mit anderen Agenten über das *Message Transport System* (MTS). Jeder Agent muss über einen *Agent Identifier* (AID) verfügen, der ihn innerhalb des Agentenuniversums eindeutig identifiziert.
- *Directory Facilitator* (DF):
  Der DF ist eine optionale Komponente. Wird sie eingesetzt, bietet sie ein Verzeichnis der *services*, die von den Agenten angeboten werden. Die Agenten können ihre *services* im DF registrieren um sie anderen Agenten zur Verfügung zu stellen oder selbst das Verzeichnis nach *services* durchsuchen.
- *Agent Management System* (AMS):
  Das AMS ist eine zentrale Komponente einer AP, da es die AIDs der Agenten einer AP verwaltet. Jeder Agent muss sich beim AMS registrieren um einen gültigen AID zu erhalten.
- *Message Transport System* (MTS):
  Das MTS wird zur Kommunikation der Agenten verwendet. Bei der Kommunikation zwischen mehreren APs erfolgt der Nachrichtenaustausch über das jeweilige MTS.
- *Agent Platform* (AP):
  Die AP fasst die oben angeführten Komponenten zu einer Einheit zusammen und stellt somit die Infrastruktur zur Verfügung in der Agenten verwendet werden können. Sie besteht aus dem physischen Gerät, dem Betriebssystem, der *Agent Support Software*, den FIPA *Agent Management Components* (DF, AMS, MTS) und den Agenten selbst.

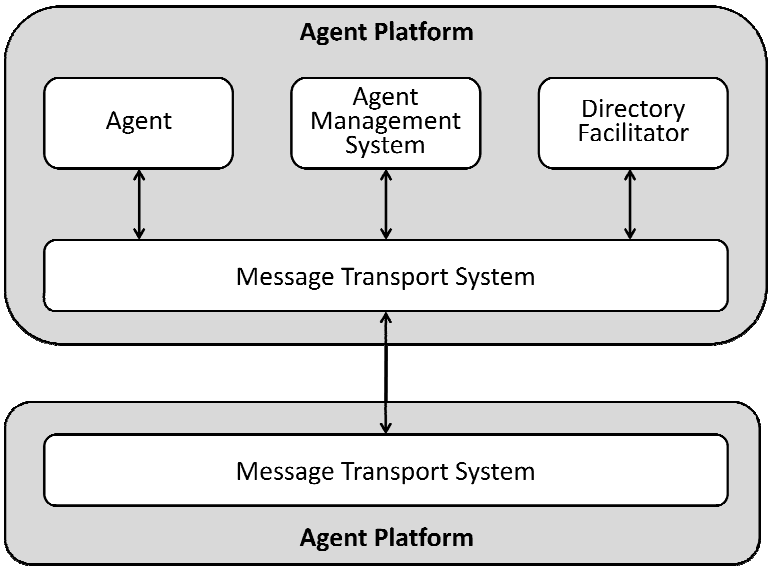

Abbildung 10 - FIPA Agent Management Referenzmodell [8]

Die FIPA *Agent Management Specification* [8] beschreibt detailliert, welche Funktionalität diese Komponenten aufweisen sollen. Ebenso wird der Aufbau und die Funktionsweise der AIDs spezifi-

ziert. Diese sind bei der Adressierung von Nachrichten essentiell, da dadurch der Sender und Empfänger der Nachricht identifiziert werden kann. Der Nachrichtentransfer selbst wird in der FIPA *Agent Message Transport Service Specification* [9] spezifiziert, wobei der ACL [10] eine entscheidende Rolle zukommt, da sie den Nachrichteninhalt beschreibt.
Die FIPA *Agent Communication Language Message Structure Specification* [10] spezifiziert den Aufbau einer Nachricht zur Kommunikation der Agenten untereinander. Eine Nachricht besteht aus den folgenden fünf Teilbereichen, wobei nicht alle für eine gültige Nachricht erforderlich sind.

- *type of communicative acts*:
  Hierbei wird die Art der Nachricht festgelegt. Die vollständige Spezifikation der Verfügbaren Typen kann in der FIPA *Communicative Act Library Specification* [11] gefunden werden.
- *participants in communication*:
  Durch die Festlegung der beteiligten Agenten wird bestimmt, welcher Agent die Nachricht versendet, welcher sie empfängt und an wen eine Antwort gerichtet wird.
- *content of message*:
  Der Inhalt der Nachricht enthält die tatsächliche Information, die von einem Agenten einem anderen mitgeteilt wird. Bei manchen Nachrichtentypen ist der Nachrichteninhalt überflüssig, da die Art der Nachricht schon alleine ausreichend Aussagekraft hat.
- *description of content*:
  Um dem Empfänger zu beschreiben, wie der Nachrichteninhalt aufgebaut ist und wie er zu lesen ist, können die Parameter dieses Bereichs genutzt werden.
- *control of conversation*:
  Bei Nachrichten, die Teil einer Konversation sind, können zusätzliche Informationen angegeben werden, die es ermöglichen einzelne Nachrichten mit anderen in Verbindung zu bringen.

In Tabelle 1 werden die Parameter, die zur Beschreibung einer Nachricht zur Verfügung stehen aufgelistet. Die wichtigsten Parameter einer Nachricht sind *performative*, *receiver* und *content*. Durch diese drei Parameter ist geklärt, welchen Typ die Nachricht hat, an wen sie verschickt wird und welchen Inhalt sie enthält.

Tabelle 1 - FIPA ACL Message Parameters [10]

Parameter	**Kategorie des Parameters**	**Optional**
performative	Type of communicative acts	nein
sender	Participant in communication	ja
receiver	Participant in communication	in Spezialfällen

reply-to	Participant in communication	ja
content	Content of message	in Spezialfällen
language	Description of content	ja
encoding	Description of content	ja
ontology	Description of content	ja
protocol	Control of conversation	ja
conversation-id	Control of conversation	ja
reply-with	Control of conversation	ja
in-reply-to	Control of conversation	ja
reply-by	Control of conversation	ja

Die FIPA-Standards bieten klare Richtlinien für agentenbasierte Anwendungen. Sowohl das *Java Agent Development Framework* als auch dessen Erweiterung JADEX sind FIPA-konforme Frameworks, die in den beiden folgenden Kapiteln erläutert werden.

3.2 Java Agent Development Framework

Das *Java Agent DEvelopment Framework* (JADE) [BPR99] ist ein *Framework* für Multi-Agenten-Systeme. Es wurde entworfen um die Entwicklung von agentenbasierten Anwendungen zu erleichtern. Eine zusätzliche Anforderung war, dass das *Framework* die FIPA-Spezifikationen (siehe Kapitel 3.1) für plattformunabhängige, intelligente Multi-Agenten-Systeme erfüllt. Zu diesem Zweck bietet es eine AP, die aus einem AMS, einem DF und einem ACC besteht. Weiters kann diese Plattform auf mehrere Hosts aufgeteilt werden, wobei nur jeweils eine *Java Virtual Machine* (JVM) auf jedem Host benötigt wird. Zur Kommunikation der Agenten untereinander wird mes*sage passing* verwendet. Jede Nachricht wird dabei innerhalb einer AP als Java-Objekt verschickt und zur Übermittlung von einer Plattform zu einer anderen als FIPA-konformer String versendet. Die verwendete Sprache für die Nachrichten ist FIPA ACL, die in Kapitel 3.1 vorgestellt wurde.

In Abbildung 11 wird die Software-Architektur einer einzelnen JADE-AP vorgestellt. Die angeführte Plattform besteht aus drei JVMs. Jede Plattform benötigt einen *container*, der nach außen die Plattform repräsentiert. Im Beispiel von Abbildung 11 übernimmt das AP-*Front-End* diese Funktion. Für das Ausführen von Agenten im Web-Browser wird ein spezieller *container* zur Verfügung ge-

stellt. Innerhalb der Plattform werden die Nachrichten vom *Message Dispatcher* mit Hilfe von *Java Remote Method Invocation* (Java RMI) transferiert. Für die Kommunikation mit anderen Plattformen wird das *Internet Inter-Object Request Broker Protocol* (IIOP) verwendet. Dadurch wird gewährleistet, dass Agenten in heterogenen Umgebungen kommunizieren können.

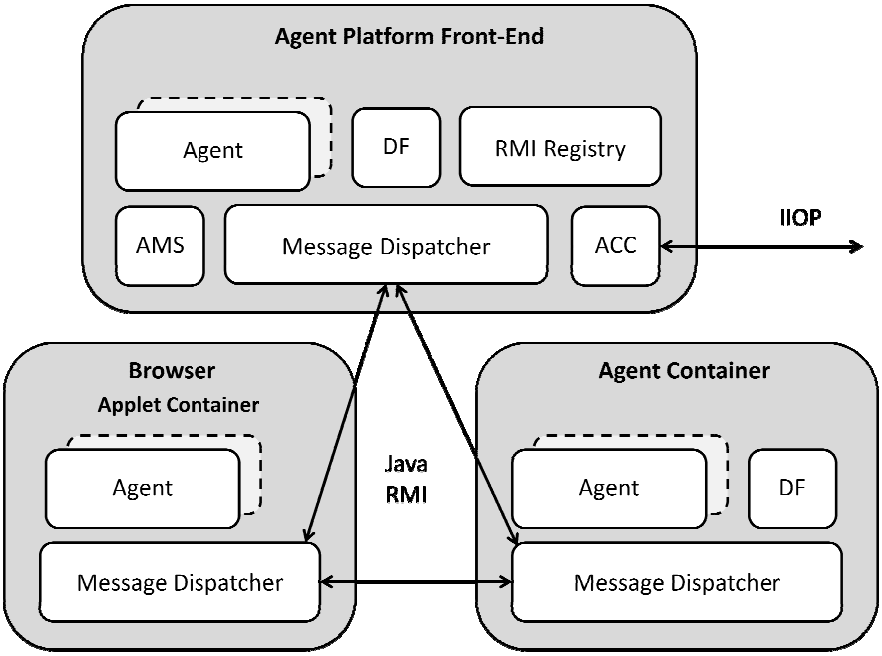

Abbildung 11 - Software Architektur einer JADE-Agenten-Plattform [BPR99, S. 100]

JADE bietet eine Agenten-Plattform, die FIPA-konform implementiert ist und plattformunabhängig als *middleware* genutzt werden kann. JADEX, das im folgenden Kapitel behandelt wird, nutzt diese Plattform als Basis und erweitert sie um das BDI-Modell.

3.3 JADEX

Das JADEX (JADE eXtension)-*Framework* [PBL03] baut auf JADE auf, welches im vorherigen Kapitel vorgestellt wurde. Es stellt eine Erweiterung dar, die dem BDI-Modell (siehe Kapitel 2.7) folgt. Die Dokumentation des *Frameworks* ist unter [12] zu finden. Jene Bereiche, die für die vorliegende Arbeit von Relevanz sind, werden auf Basis dieser Dokumentation und entsprechend der Verwendung in der Implementierung der triebgesteuerten Fauna in diesem Kapitel erläutert. Zu Beginn des Kapitels werden Vorteile von JADEX angeführt, bevor daran anschließend ein Überblick über die Architektur gegeben wird. Die darauf folgenden Kapitel beschäftigen sich mit den Details der Komponenten und wie sie verwendet werden.

3.3.1 Vorteile von JADEX

JADEX bietet laut [PBL05a, S. 14] im Vergleich zu anderen BDI-*Frameworks* einige Vorteile, welche die Verwendung und das Erlernen der bereitgestellten Funktionalität erleichtern. Darunter fällt die Plattformunabhängigkeit von JADEX und die lose Kopplung zur Middleware. Außerdem wird für JADEX keine neue Programmiersprache verwendet, sondern auf Java aufgebaut. Hierdurch können gängige objektorientierte *Integrated Development Environments* (IDEs) verwendet werden. Desweiteren wird durch das Konzept der *capabilities* die Wiederverwendbarkeit von Programmcode ermöglicht. Durch die starke Typisierung der Elemente in der Definition der Agenten können Konsistenzüberprüfungen durchgeführt und dadurch Fehler frühzeitig erkannt werden. Außerdem beseitigt JADEX laut [PBL05a, S. 1] eine Einschränkung von BDI-Architekturen durch die explizite Repräsentation von *goals* (siehe Kapitel 3.3.6).

Abgesehen von den Vorteilen gibt es auch Kritikpunkte an JADEX. Christian Thiel beschreibt in [Thi10, S. 23, 69], dass hoher Einarbeitungsaufwand durch komplexe Klassenstruktur und viele Konfigurationsmöglichkeiten nötig ist. Außerdem widerspricht die Repräsentation der *beliefbase* von JADEX als Sammlung von Java-Objekten der Vorgabe einer BDI-Architektur, dass jeder Agent eigene *beliefs* haben muss. Die Java-Objekte können nämlich zum Beispiel unter Verwendung des *Singleton Patterns* für alle Agenten identisch sein. Des Weiteren bietet JADEX kein explizites Konzept für die Kommunikation eines Agenten mit der Umwelt. Dadurch erfolgt die Interaktion über die *beliefbase*.

3.3.2 JADEX-Architektur

Innerhalb der AP, die für die Implementierung von JADEX-Agenten verwendet wird, können mehrere Agenten definiert werden. Die Definition eines Agenten erfolgt durch zwei Teilbereiche: dem *agent definition file* (ADF) und dem prozeduralen *plan*-Code. Die beiden Komponenten werden in Abbildung 12 veranschaulicht. Das ADF wird in *Extensible Markup Language* (XML) definiert und beschreibt einen Agenten - bis auf den auszuführenden *plan*-Code - vollständig. Der *plan*-Code wird in Java erstellt und interagiert mit der Agentendefinition über das *application programming interface* (API).

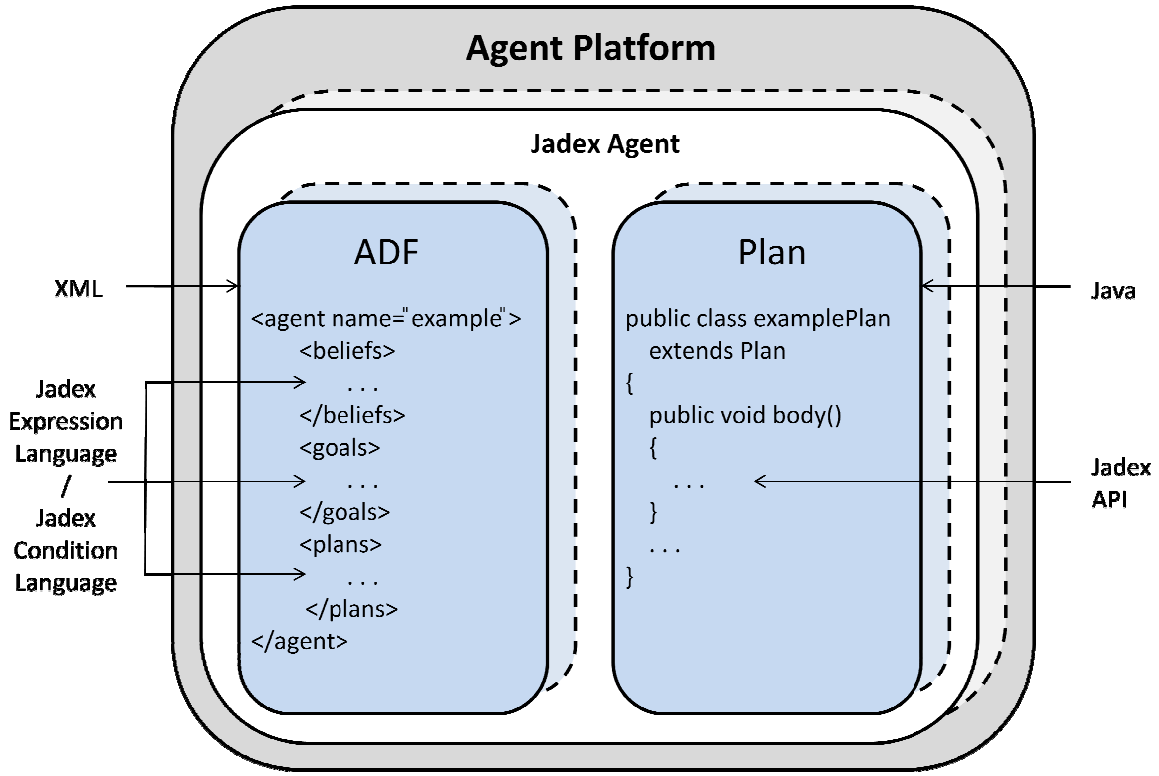

Abbildung 12 - JADEX BDI agent components [13]

Im ADF, das in Kapitel 3.3.3 behandelt wird, erfolgt die Festlegung der Struktur des Agenten durch die drei Kernbereiche des ADF: *beliefs*, *goals* und *plans*. Die Struktur und Syntax des ADF muss hierbei dem XML-Schema von JADEX entsprechen. Zur detaillierteren Ausgestaltung der Bereiche einer ADF kann die JADEX-*Expression Language* und die JADEX-*Condition Language* verwendet werden. Für JADEX wurde keine neue Programmiersprache entwickelt. Beide Sprachen basieren auf Java, enthalten jedoch Erweiterungen, die in Kapitel 3.3.9 erläutert werden.

In einer *plan*-Klasse wird definiert, welche Aktionen der Agent durchführt, wenn ein *plan* aus dem ADF ausgewählt wird. Beim Aufruf eines *plans* wird die Methode *body* aufgerufen. Innerhalb einer *plan*-Klasse kann über die JADEX-API auf die Definition des Agenten im ADF zugegriffen werden. Dadurch kann der Zustand des Agenten während der Laufzeit verändert und das Verhalten gesteuert werden.

JADEX bietet die Möglichkeit über die JADEX-API das Model eines Agenten während der Laufzeit zu verändern. Dadurch könnten zum Beispiel Verfahren für lernende Agenten angewendet werden. Sämtliche Teilbereiche eines Agenten, wie zum Beispiel die *beliefbase*, die *goal library* oder auch die *plan library*, können zur Laufzeit verändert und erweitert werden. Das ist ein wichtiger Aspekt für die Flexibilität von JADEX bei der Entwicklung von Agenten. Allerdings wird in der vorliegenden Arbeit auf ein statisches Modell zurückgegriffen.

In Abbildung 13 wird die abstrakte JADEX-Architektur dargestellt. Von außen betrachtet agiert der Agent als *black box*, die Nachrichten empfängt und verschickt. Die eingehenden Nachrichten dienen

als *input* für den *reaction*- und *deliberation*-Mechanismus, der die zentrale Schaltstelle eines Agenten ist. Des Weiteren können auch interne *events* und *goals* beziehungsweise *sub goals* den Mechanismus beeinflussen. Als Folge der *deliberation* werden neue *plans* aus der *plan library* ausgewählt und instanziiert oder bereits aktive *plans* erhalten die Informationen des *events* weitergereicht. Die aktiven *plans* können die *beliefbase* manipulieren und dadurch interne *events* auslösen, oder direkt neue *events* auslösen. Außerdem können Nachrichten an andere Agenten in aktiven *plans* erstellt und verschickt werden.

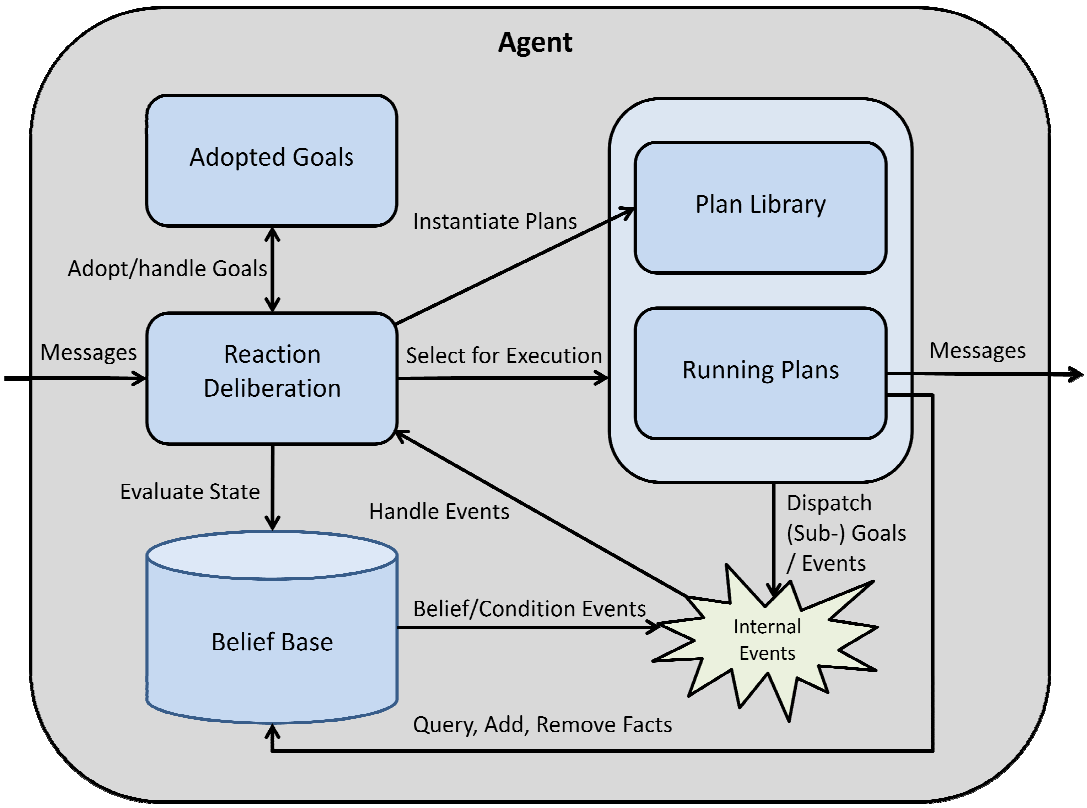

Abbildung 13 - JADEX-Architektur [BPL04, S. 199]

3.3.3 Agent Definition Files

Ein *Agent Definition File* (ADF) wird benötigt um einen JADEX-Agenten zu erstellen. Beim Starten eines Agenten wird zuerst das ADF geladen und anschließend werden die *beliefs*, *goals* und *plans* entsprechend der Definition im ADF initialisiert. Nur eine sowohl wohlgeformte XML-Datei als auch entsprechend der JADEX-XML-Schema-Datei gültige Definition des Agenten kann gestartet werden. Die Struktur eines ADF wird in Abbildung 14 dargestellt. Sie veranschaulicht, welche Elemente ein ADF ausmachen, wobei die *beliefs*, *goals* und *plans* die Kernelemente sind.

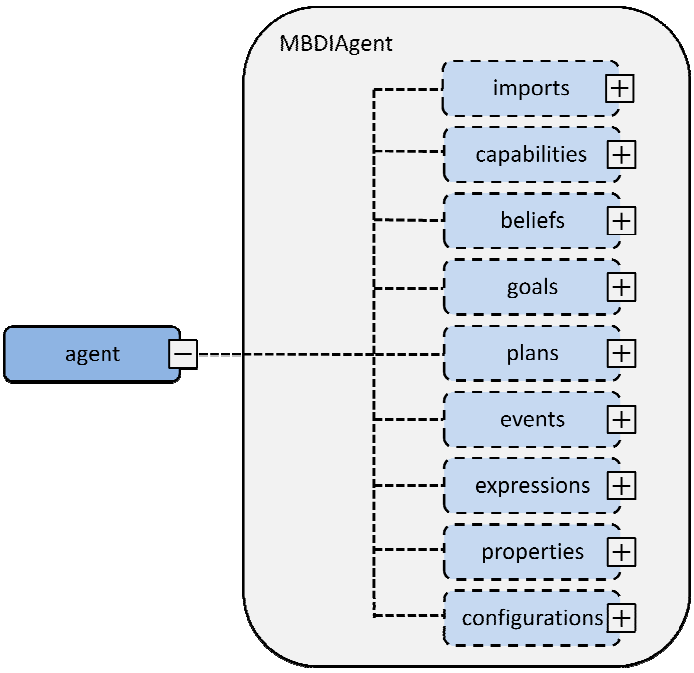

Abbildung 14 - ADF-Struktur [12, S. 12]

Für weitere Java-Klassen des Agenten, die sich nicht im Verzeichnis des ADF befinden, wird der *imports*-Bereich verwendet. Bei der Integration von zusätzlicher Funktionalität in Form von Agenten-Modulen, wird auf *capabilities* zurückgegriffen.

Eine wesentliche *capability*, die bei der BDI-Fauna zum Einsatz kommt, ist das *Agent Management System* (AMS). Dieses wird in Kapitel 3.3.4 erläutert. Die Bereiche des ADF, die sich den *beliefs*, *goals*, *plans*, *events* und *expressions* widmen, werden in den darauf folgenden Kapiteln behandelt.

Properties bieten eine Möglichkeit, das Verhalten der Agenten individuell anzupassen. Unterschiedliche Konfigurationen werden im *configurations*-Bereich erstellt, um den Initial- und den Endzustand des Agenten zu definieren.

3.3.4 Capabilities

Capabilities bieten zusätzliche Funktionalität in Form von *beliefs*, *goals* und *plans*, die in Modulen zusammengefasst werden. Sie ermöglichen dadurch das Wiederverwenden von Agentenbestandteilen. Da in *capabilities* auch *subcapabilities* enthalten sein können, ist ein hierarchischer Aufbau von Modulen möglich. Das JADEX-*Framework* stellt einige vordefinierte *capabilities* zur Verfügung, die grundlegende Aufgaben erfüllen. Darunter fällt auch das AMS, das als einzige der vordefinierten *capabilities* in der BDI-Fauna verwendet wird.

Das *Agent Management System* (AMS) wird in der gleichnamigen *capability* zur Verfügung gestellt und bietet Funktionalität um Agenten zu verwalten. Agenten können mit Hilfe des AMS erstellt, gestartet, zerstört, unterbrochen und fortgesetzt werden. Außerdem kann nach Agenten gesucht und eine AP heruntergefahren werden. Für eine bestimmte Funktionalität muss jeweils nur das entsprechende *goal* im ADF referenziert oder zur Laufzeit hinzugefügt werden. Die Werte der Parameter der *goals* bestimmen dann die Details der Ausführung.

3.3.5 Beliefs

Jeder Agent besitzt eine *beliefbase*, die es ihm ermöglicht, die Fakten seines Wissens in Form von *beliefs* zu speichern. Die *beliefbase* wird im ADF definiert und üblicherweise in *plans* abgerufen oder geändert. Diese *beliefs* können sowohl einfache Werte als auch geordnete Mengen von Fakten beinhalten. Die Mengen von Fakten werden als *belief sets* bezeichnet. Sowohl *beliefs* als auch *belief sets* müssen immer benannt und typisiert werden. Der Name dient zur Zuordnung zu den Fakten in einem *belief*. Die Typisierung wird auch während der Laufzeit überprüft, sodass immer nur Objekte mit dem passenden Datentyp gespeichert werden können. Als Datentyp können beliebige Java-Objekte verwendet werden. Die *beliefs* und *belief sets* können also als *container* für Java-Objekte betrachtet werden.

Die *beliefbase* erlaubt es mit Hilfe einer *query*-Sprache Untermengen von *beliefs* abzufragen oder Ausdrücke anhand des Zustands der *beliefbase* auszuwerten. Ähnliche Auswertungen können durch *conditions* durchgeführt werden. *conditions* beschreiben einen statischen Zustand der *beliefbase*, der mehrere *beliefs* umfassen kann. Wird die *condition* erfüllt, wird automatisch ein interner *event* generiert, der zum Beispiel das Ausführen von bestimmten *plans* auslöst. Zusätzlich können *beliefs* als *expressions* gespeichert werden, die dynamisch bei Bedarf ausgewertet werden.

Abbildung 15 gibt einen Überblick über das XML Schema der JADEX-*beliefs*. Der Aufbau eines *beliefs* erfolgt durch die Definition eines *fact tags*. Dieses kann auch ausbleiben, wenn der *belief* bei der Definition noch keine Fakten enthält. Bei *belief sets* ist es zudem möglich eine Liste von Fakten zu definieren, bei der die genaue Anzahl der Elemente noch nicht bestimmt werden kann, weil die Daten dafür zum Beispiel aus einer Datenbank geladen werden. Für Referenzen auf bestehende *beliefs* und *belief sets* in *capabilities* können die *beliefref*- und *beliefsetref tags* verwendet werden.

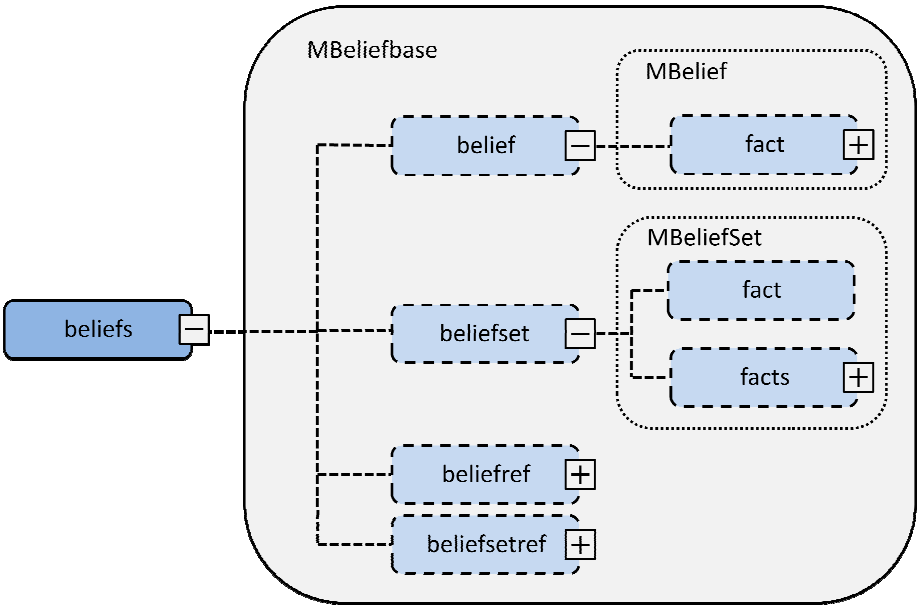

Abbildung 15 - JADEX beliefs XML-Schema [12, S. 21]

3.3.6 Goals

Die *goals* eines Agenten können einen der folgenden vier verschiedenen Typen annehmen.

- *achieve*:
  Bei einem *achieve goal* handelt es sich um einen Wunschzustand, der erreicht werden soll. Dieser Typ ist der häufigste und zugleich auch einfachste *goal*-Typ.
- *query*:
  Das *query goal* wird verwendet um Informationen abzufragen. Es wird erst dann als erfüllt angesehen, wenn die Information, die gesucht wird, verfügbar ist. Das kann einerseits sofort der Fall sein, wenn die Information dem Agenten bereits bekannt ist. Andererseits müssen *plans* ausgeführt werden um an die gewünschten Informationen zu gelangen.
- *maintain*:
  Ein Agent, der ein *maintain goal* verfolgt, überwacht den Zustand, der aufrecht erhalten bleiben soll und unternimmt kontinuierlich Schritte um den Zielzustand wiederherzustellen, sollte das notwendig sein.
- *perform*:
  Beim *perform goal*, steht - im Gegensatz zu den vorangegangenen *goal*-Arten - nicht ein Zustand im Vordergrund, sondern eine Aktion, die ausgeführt werden soll. Es spielt dabei keine Rolle, ob die durchzuführende Aktion erfolgreich ist oder nicht. Jedem *perform goal* ist ein *plan*, der ausgeführt werden soll, zugewiesen.

Zusätzlich gibt es die Möglichkeit *meta goals* zu definieren. Ihre Aufgabe ist es bei *events* oder *goals*, bei denen mehrere *plans* zur Anwendung kommen könnten, zu entscheiden welcher zur Ausführung gelangt.

Zusätzlich besteht die Möglichkeit mit *performgoalref*, *achievegoalref*, *querygoalref*, *maintaingoalref* und *metagoalref* auf *goals* in *capabilities* zu verweisen. Die Details zu den Parametern jedes *goal*-Typs und den allgemeinen Parametern von *goals*, sind in Abbildung 16 abgebildet.

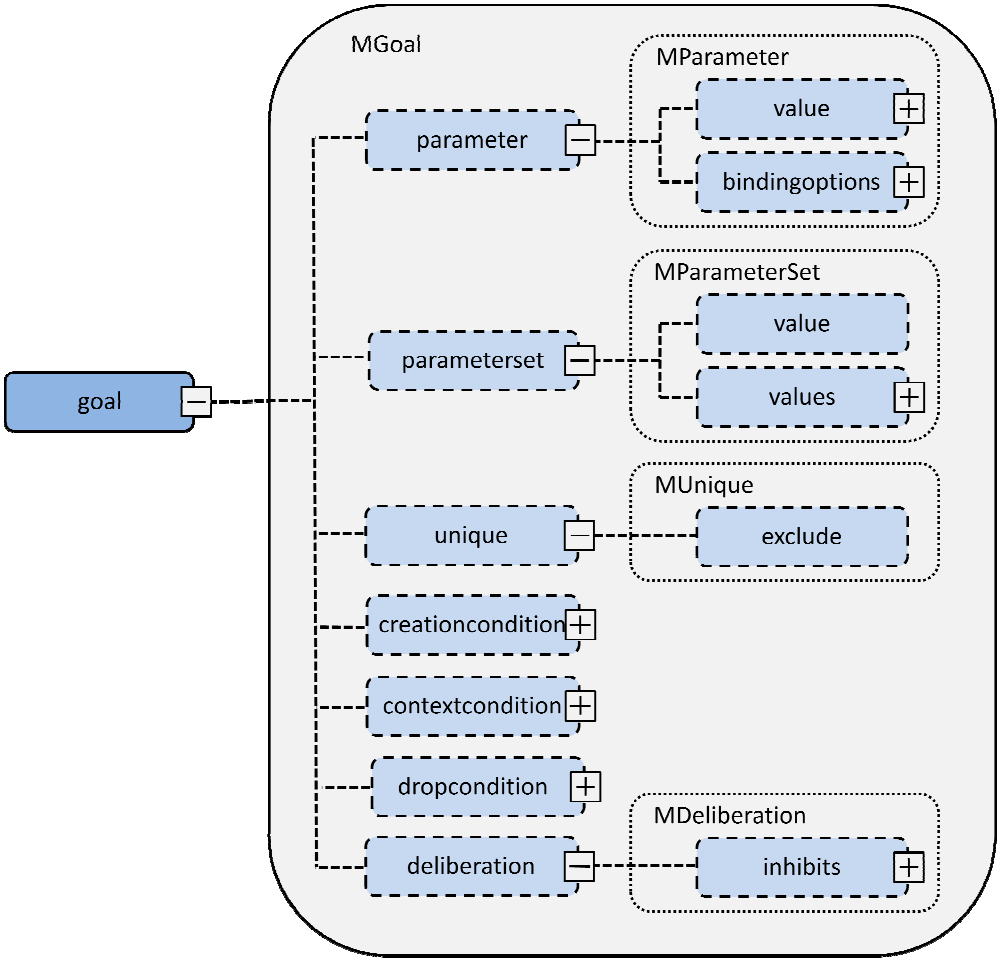

Abbildung 16 - JADEX common goal features [12, S. 27]

Ein *goal* kann mit Parametern versehen werden, die sowohl einzelne Werte als auch eine Menge an Werten enthalten können. Für jeden Parameter kann eine Richtung angegeben werden, die darüber Aufschluss gibt, wann der Parameter gelesen werden kann. Es werden hierbei drei Richtungen unterschieden: *in* (eingehende), *out* (ausgehende) und *inout* (Kombination aus beiden Richtungen). Bei eingehenden Parametern wird der Wert festgelegt, bevor das *goal* verfolgt wird. Im Gegensatz dazu wird bei ausgehenden Parametern der Wert während der Verarbeitung durch den *plan* befüllt und

kann am Ende der *goal*-Verfolgung ausgelesen werden. Für Parameter deren Richtung *inout* ist, gelten sowohl die Eigenschaften für eingehende als auch für ausgehende Parameter.

Die *unique*-Option ermöglicht es mehrere Instanzen des gleichen *goals* zu unterbinden, wobei mit Hilfe von *exclude*-Angaben festgelegt werden kann, welche Kriterien zur Feststellung ob ein *goal* gleich ist, angewendet werden.

Conditions ermöglichen eine Einschränkung unter welchen Voraussetzungen ein *goal* erstellt wird, was gelten muss um ein *goal* beizubehalten und wann ein *goal* verworfen wird.

Die *deliberation* ermöglicht die Steuerung der Relation zwischen unterschiedlichen *goals*. Da Agenten mehrere *goals* gleichzeitig verfolgen können, bietet JADEX mit der *easy deliberation* eine Möglichkeit *goals* in Beziehung zu einander zu setzen. Dadurch weiß der Agent, welche *goals* verfolgt werden sollen und welche nicht. Der *deliberation*-Mechanismus entscheidet, welche *goals* aus der Menge der möglichen *goals* als *option* für die spätere Verarbeitung behalten werden sollen und welche sofort aktiviert werden. Bei der easy *deliberation* kann den *goals* eine Kardinalität zugewiesen werden, welche die Anzahl der gleichzeitig aktiven *goals* eines Typs einschränkt. Zusätzlich können für jedes *goal* über *inhibit*-Regeln andere *goals* angegeben werden, die nicht gleichzeitig verfolgt werden dürfen. Dadurch ergeben sich Abhängigkeiten zwischen den *goals*, die für das gewünschte Verhalten des Agenten notwendig sein können. Ein Beispiel für eine *inhibit*-Regel ist: Wenn ein Tier schläft, kann es nicht gleichzeitig fressen.

Der *lifecycle* von *goals* in JADEX ist in Abbildung 17 dargestellt. Jedes *goal* kann einen der drei Zustände *new*, *adopted* und *finished* annehmen. Der Initialzustand eines *goals* ist *new*, wodurch das *goal* zwar existiert, aber vom *deliberation*-Mechanismus noch nicht beachtet wird. Erst wenn sich das *goal* im Zustand *adopted* befindet, kann es als neues Ziel verfolgt werden bis es fallen gelassen wird oder beendet wird. Zustandsübergänge können entweder aufgrund von Aktionen in *plans* erfolgen, oder aufgrund von *conditions*. Hierbei gibt es zwei Arten von *conditions*: *conditions*, die einen Zustandsübergang überwachen und *conditions*, die Zustandsübergänge auslösen. Die beiden Arten sind in der Legende in Abbildung 17 angeführt. Der Zustand *adopted* besteht aus mehreren Unterzuständen, die den Status eines *goals*, das im *deliberation*-Mechanismus Beachtung findet, näher beschreiben. Hierbei handelt es sich um die Zustände *option*, *suspended* und *active*. Ein *goal* kann jeweils dann als *option* von einem Agenten in Betracht gezogen werden, wenn die *context condition* erfüllt ist. Der *deliberation*-Mechanismus kann ein *goal* dann aktivieren und so die Ausführung des *goals* starten. Ebenso kann er es wieder deaktivieren, so dass das *goal* nur noch eine *option* ist. Abhängig von der *context condition* wird ein *goal* in den Zustand *suspended* versetzt. Ist die *context condition* anschließend wieder erfüllt, wechselt das *goal* in den Zustand *option*. Erst dann kann das *goal* vom *deliberation*-Mechanismus wieder aktiviert werden.

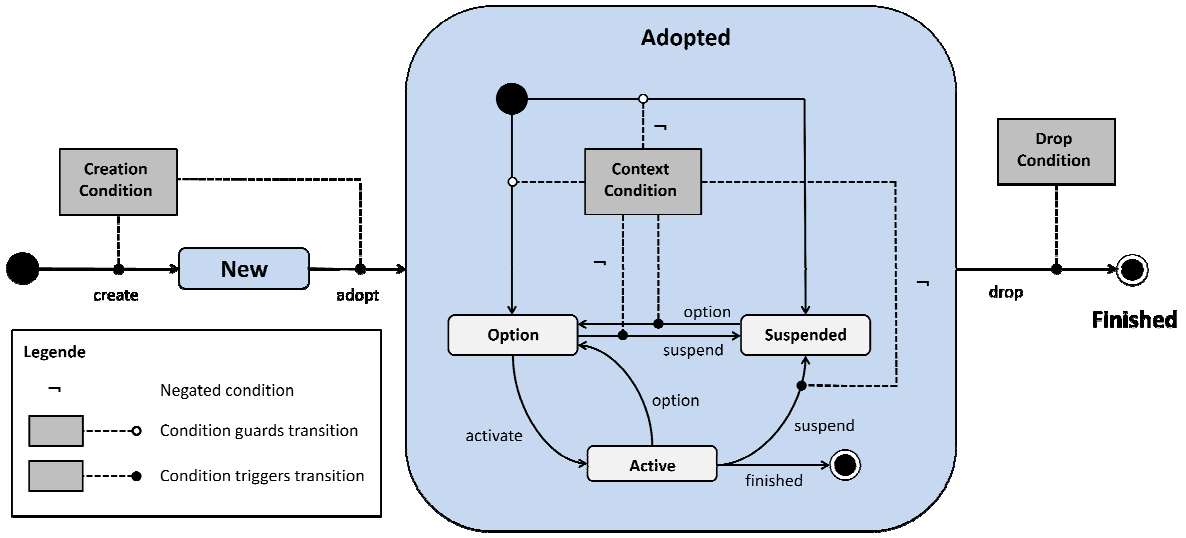

Abbildung 17 - Goal lifecycle [BPM+04, S. 50]

3.3.7 Plans

Aktionen eines Agenten werden in *plans* repräsentiert. Sie beinhalten die Schritte, die ein Agent durchführen kann um bestimmte *goals* zu erreichen. *Plans* werden automatisch je nach aktueller Situation, in der sich der Agent befindet, ausgewählt.

Die Funktionalität der *plans* wird in Java-Klassen definiert. Daher können hier sämtliche Möglichkeiten von Java genutzt werden. Der Kontext der *plans* wird im ADF definiert. Dadurch ist festgelegt, unter welchen Umständen ein *plan* initialisiert und ausgeführt wird.

Die Definition im ADF folgt den Regeln des XML-Schemas von JADEX. In Abbildung 18 ist die Struktur der *plans* abgebildet. Der *plans*-Bereich im ADF kann beliebig viele *plans* beinhalten, die jeweils durch ihren Namen unterschieden werden. Zur Priorisierung der *plans* kann das Attribut *priority* bei jedem *plan* festgelegt werden. Ein *plan* mit einer hohen Priorität wird vor einem mit niedrigerer Priorität ausgewählt, wenn es für ein *event* mehrere Kandidaten bei der Auswahl eines passenden *plans* gibt. Wird keine Priorität vergeben, wird entweder ein zufälliger *plan* gewählt oder derjenige, der im ADF früher definiert wurde. Welche dieser beiden Möglichkeiten Anwendung findet, kann über BDI-*flags* definiert werden.

Ebenso wie *goals* (siehe Kapitel 3.3.6), können *plans* mit Parametern versehen werden. Die Funktionsweise ist hierbei die gleiche wie jene der Parameter der *goals*. Ebenso verhält es sich mit den *conditions* der *plans*. *Preconditions* müssen erfüllt werden, damit ein *plan* aktiviert werden kann und nur solange *context conditions* erfüllt bleiben, kann ein *plan* aktiv sein.

Der *body*-Bereich der *plan*-Definition enthält eine Java *expression*, die einen *plan body* erstellt. Diese expression ist normalerweise ein Aufruf eines Konstruktors einer Java-Klasse, welche die Klasse *jadex.runtime.Plan* erweitert und eine *body*-Methode bereitstellt, in der die Aktionen des jeweiligen

plans enthalten sind. Bei der Abarbeitung der Anweisungen in der *body*-Methode kann es erforderlich sein, dass mehrere Aktionen notwendigerweise gemeinsam ausgeführt werden müssen. Ein Beispiel hierfür ist das aufeinanderfolgende Ändern von zwei *beliefs*. Für dieses Vorhaben können *atomic blocks* verwendet werden. Alle Aktionen, die in einem *atomic block* definiert sind, werden in einem einzelnen *plan step* durchgeführt. Bei Aktionen, welche die *beliefbase* verändern, könnte ohne einen *atomic block* unter Umständen eine *condition* erfüllt werden und dadurch der *plan step* enden, bevor die zweite Änderung erfolgt ist. Die nachfolgenden Aktionen des *plans* werden dadurch nicht zwangsläufig ausgeführt, was zu unerwünschtem Verhalten führen kann.

Trigger werden verwendet um die Ereignisse zu definieren, für die der jeweilige *plan* verwendet werden kann. Hierbei können sämtliche Arten von *events* (siehe Kapitel 3.3.8) als *trigger* definiert werden. Wird ein *trigger* ausgelöst, kann es sein, dass für den *plan* notwendige Ereignisse noch ausgeführt werden müssen, bevor er aktiviert werden kann. Um solche Ereignisse abzuwarten, kann die *waitqueue* verwendet werden.

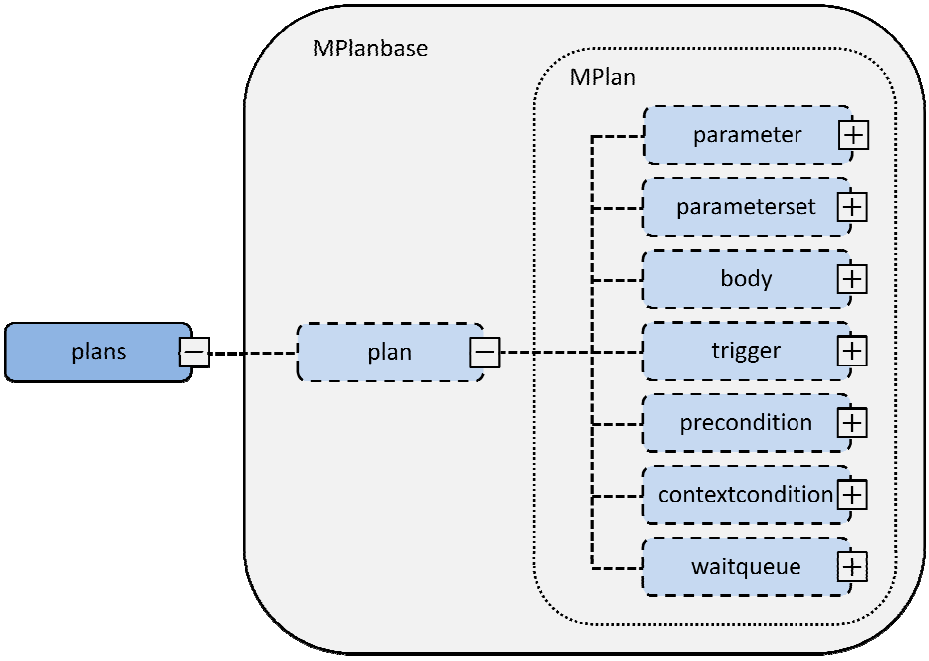

Abbildung 18 - JADEX plans XML-Schema [12, S. 37]

3.3.8 Events

Für Ereignisse, die in einem Agenten auftreten, werden *events* ausgelöst. Diese können einerseits *message events* sein, die beim Empfangen von ACL-Nachrichten ausgelöst werden. Andererseits können interne *events* auftreten, die von einem *goal*, einer Änderung eines *beliefs*, einem *timeout* oder einer erfüllten *condition* ausgelöst werden.

Die Abarbeitung der *message events* erfolgt anhand des *execution model*, das in Abbildung 19 ersichtlich ist. Für jede ankommende Nachricht wird ein entsprechender Eintrag in die *message queue* vorgenommen. Anschließend wird eine *capability* gesucht, welche die *message* verarbeiten kann. Hierfür werden die *eventbases* des Agenten überprüft und die *event templates* nach dem am besten geeigneten durchsucht. Das passendste *event* wird anschließend erzeugt und in die *event list* aufgenommen.

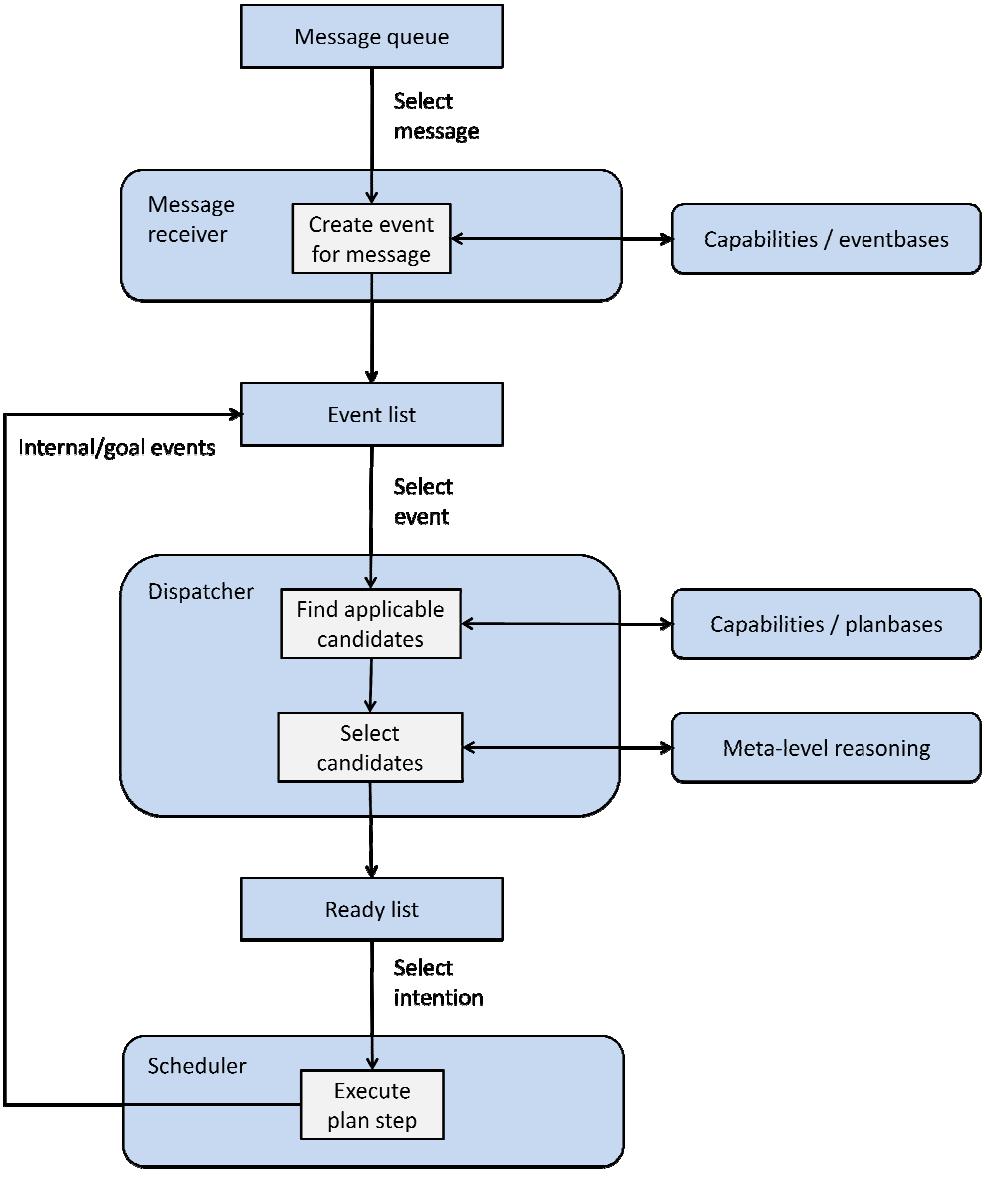

Abbildung 19 - JADEX execution model [PBL05a, S. 7]

Der *dispatcher* ist dafür verantwortlich anwendbare *plans* für jedes *event* der *event list* auszuwählen. Die dafür zur Verfügung stehenden *plans* befinden sich in den *capabilities* des Agenten und deren

planbases. Welche *plans* schließlich zur Abarbeitung des *events* verwendet werden, wird im *meta-level reasoning* entschieden. Diese Entscheidung kann im einfachsten Fall dadurch getroffen werden, dass der erste passende *plan* verwendet wird. Im kompliziertesten Fall werden *meta-plans* zur Entscheidungsfindung ausgeführt.

Ausgewählte *plans* werden in die *ready list* übernommen und verbleiben dort bis zu ihrer Ausführung. Die Ausführung wird vom *scheduler* geplant, wobei ein *plan* jeweils für einen *plan step* ausgeführt wird. Die Dauer eines *steps* hängt vom Kontext ab und nicht vom *plan* selbst. Der *plan* wird *step* für *step* so lange ausgeführt bis er explizit auf ein Ereignis wartet oder den Zustand des Agenten ändert. Die Zustandsänderungen können einerseits direkt oder andererseits durch Seiteneffekte auftreten. Eine Zustandsänderung ist zum Beispiel wenn ein *goal* erstellt oder fallengelassen wird. Ebenso ändert eine Änderung eines *beliefs* den Zustand des Agenten, wenn dadurch eine *creation condition* eines *goals* erfüllt wird.

3.3.9 Expressions

Expressions sind Ausdrücke, die jederzeit während der Laufzeit ausgewertet werden können. Sie dienen der Überprüfung des aktuellen Zustands des Agenten und werden zum Beispiel zur Definition von *conditions* benötigt. Um *expressions* in einer XML-Datei wie dem ADF zu ermöglichen, wurde eine eigene eingebettete *expression language* entwickelt. *Expressions* werden sehr oft systemintern ausgewertet und dürfen daher keine Seiteneffekte aufweisen. Die *expression language* basiert auf den Java *expressions* und wurde mit einer Teilmenge der *object query language* (OQL) [BEJ+00, S. 89-132] ergänzt. Die Syntax der Teilmenge der OQL wird in Abbildung 20 in erweiterter Backus-Naur-Form (EBNF) [14] dargestellt.

```
01: select_expression ::= "SELECT" ("ALL" | "ANY" | "IOTA")?
02:   (
03:     (expression "FROM" ("$" identifier "IN" expression) ("," "$" identifier "IN" expression)* )
04:     | ("$" identifier "FROM" expression)
05:   )
06:   ("WHERE" expression)?
07:   ("ORDER" "BY" expression ("ASC" | "DESC")? )?
```

Abbildung 20 - Syntax der OQL-Erweiterung [PBL05a, S. 10]

Abgesehen von der bekannten *select-from-where*-Form, die zum Beispiel im SQL-92-Standard (Structured Query Language) [15] Verwendung findet, kommen die Schlüsselwörter „ALL“, „ANY“ und „IOTA“ zum Einsatz. „IOTA“ steht für exakt eines, „ANY“ für das erste passende und „ALL“ für alle passenden Resultate. Ein Beispiel für diese Syntax ist:

SELECT $creature FROM $beliefbase.creatures WHERE $creature.isInSight()

Dieses Beispiel könnte verwendet werden um alle Kreaturen auszuwählen, die in Sichtweite des Agenten sind.

Für *expressions* können vordefinierte Schlüsselwörter, wie $beliefbase in obigem Beispiel, verwendet werden um auf den Agenten beziehungsweise Komponenten davon zuzugreifen. In Tabelle 2 werden diese Schlüsselwörter angeführt und jeder Variablen ihre Klasse beziehungsweise ihre Verwendungsbereiche zugeordnet. Die Zugriffsart über Variablen verkürzt den Code, der notwendig ist um auf die gewünschten Objekte zu referenzieren. Die längere Alternative zu den Schlüsselwörtern wäre ein Zugriff über die jeweiligen Interfaces mit den dafür vorgesehenen Methoden.

Tabelle 2 - Reservierte Expression Variablen [12, S. 60-61]

Name	**Class**	**Accessibility**
$agent	RBDIAgent	In any agent expression
$scope	RCapability	In any expression
$beliefbase	RBeliefbase	In any expression
$planbase	RPlanbase	In any expression
$goalbase	RGoalbase	In any expression
$eventbase	REventbase	In any expression
$expressionbase	RExpressionbase	In any expression
$propertybase	RPropertybase	In any expression
$goal	RGoal	In any goal expression (except creation condition and binding options)
$plan	RPlan	In any plan expression (except trigger and pre condition and binding options)
$event	REvent	In any event expression (except binding options)
$ref	RGoal	In any inhibition expression
$messagemap	Map	In match expressions of message events

3.3.10 External Interactions

External interactions sind notwendig, wenn eine andere Systemkomponente mit einem JADEX-Agenten interagieren will. Für diese Interaktionen gibt es bei JADEX zwei Möglichkeiten. Einerseits die *external processes* und andererseits die *agent listeners*.

External Processes

External processes werden verwendet, wenn von einem anderen Prozess auf den Agenten zugegriffen werden soll. Beispielsweise wenn ein Anwender über ein GUI (*Graphical User Interface*) mit den Agenten interagieren will. In diesem Fall würde der *thread* des *event handlers* des GUI auf die Eingabe des Anwenders reagieren und versuchen dem Agenten die Auswirkungen der Aktion weiterzugeben. Da hierfür zwei *threads* asynchron miteinander kommunizieren würden, können *deadlocks* entstehen oder unerwünschtes Verhalten auftreten. Um solche Fälle auszuschließen, lässt JADEX direkte Aufrufe von Agentenfunktionalität nicht zu, beziehungsweise wirft eine *runtime exception*. Ein JADEX-Agent führt immer nur einen Planschritt gleichzeitig aus, wodurch seine Aktionen synchronisiert werden können. Bei Zugriffen von externen Prozessen muss aber auch die Synchronisation der Agenten-*threads* mit dem externen Zugriff gewährleistet werden. Hierfür stellt die *AbstractPlan*-Klasse die Methode *getExternalAccess()* zur Verfügung. Das Objekt, das durch diese Methode zur Verfügung gestellt wird, regelt die Synchronisation mit dem Agenten automatisch. Es bietet Zugriff auf die komplette Funktionalität des *ICapability*-Interfaces, wodurch zum Beispiel auf die *beliefbase*, die *goals* und die *plans* zugegriffen werden kann.

Agent Listeners

Agent listeners werden verwendet, wenn eine Aktion durchgeführt werden soll, sobald sich der Zustand des Agenten ändert. Bei der Verwendung eines GUI können *agent listeners* zum Einsatz kommen. Die Darstellung des Agenten in dem GUI könnte dann abhängig vom *agent listener* verändert werden. *Agent listener* können sowohl im ADF definiert werden, als auch zur Laufzeit hinzugefügt werden. Um das Verhalten des Agenten zu überwachen, sind *listener* für jene JADEX-Elemente verfügbar, die in Tabelle 3 aufgelistet werden. Damit können sowohl Änderungen und Ereignisse des Zustands des Agenten und seiner Wissensbasis als auch seiner Aktionen und *events* beaufsichtigt werden.

Tabelle 3 - Verfügbare Agent Listeners [12, S. 90]

Listener	Element	Listener Methods
IAgentListener	ICapability	agentTerminating()
IBeliefListener	IBelief	beliefChanged()

IBeliefSetListener	IBeliefSet	factAdded(), factRemoved(), beliefSetChanged()
IConditionListener	ICondition, IExpressionbase	conditionTriggered()
IGoalListener	IGoal, IGoalbase	goalAdded(), goalFinished()
IInternalEventListener	IEventbase	internalEventOccured()
IMessageEventListener	IMessageEvent, IEventbase	messageEventReceived(), messageEventSent()
IPlanListener	IPlan, IPlanbase	planAdded(), planFinished()

3.4 JADEX Tools

JADEX bietet *tools*, welche die Entwicklung und Wartung von Agentensystemen vereinfachen beziehungsweise während der Laufzeit Analysen ermöglichen. Die folgenden Kapitel sind einer Auswahl an *tools* gewidmet, die bei der Implementierung der BDI-Fauna Verwendung fanden.

JADEX Control Center

Das JADEX *Control Center* ist die zentrale Anlaufstelle für alle *tools*, die während der Laufzeit zur Verfügung stehen. Es bietet einen Überblick über die weiteren *tools* und ermöglicht die Verwaltung von Konfigurationen für diese. Zusätzlich dient es der Projektverwaltung. Die Funktionalität des *Control Centers* kann mit Hilfe von *plugins* erweitert werden. Jedes eingebettete *tool* erhält einen separaten Bereich, der über den jeweiligen *button* geöffnet werden kann. Folgende *tools* werden derzeit mit der *standard distribution* von JADEX mitgeliefert: *Starter, Directory Facilitator, Browser, Conversation Center, Introspector, Tracer, Test Center* und *Jadexdoc*.

JADEX Starter

Der JADEX *Starter* ermöglicht die Verwaltung aller Agenten. Die geladenen *classpath*-Verzeichnisse werden hierbei automatisch aktualisiert, analysiert und die Integrität der gefundenen Agenten überprüft. Alle gefundenen, fehlerfreien Agenten können gestartet, gestoppt, suspendiert und wieder aktiviert werden. Zusätzlich werden Informationen über die Agenten angezeigt. Solche Informationen sind zum Beispiel der Dateiname, der Agentenname, die Beschreibung des Agenten oder auch die Argumente beim Start des Agenten.

Conversation Center

Das *Conversation Center* hilft Entwicklern beim Erstellen und Senden von Nachrichten an Agenten. Außerdem unterstützt es beim Überprüfen der empfangenen Nachrichten. Hierfür bietet es eine Übersicht über die zuletzt gesendeten und empfangenen Nachrichten. Die Eigenschaften und Parameter einer Nachricht werden in einem separaten Bereich des *Conversation Centers* angezeigt und können dort verändert werden. Das neuerliche Senden einer bereits zuvor gesendeten Nachricht, oder einer geringen Abwandlung davon, wird erleichtert, indem nach der Auswahl einer Nachricht, deren Informationen im Eigenschaftenbereich der Nachricht angezeigt werden.

Das Nachrichtenformat entspricht der FIPA ACL *Message Structure Specification* [10]. Jede Nachricht benötigt einen Sender und einen oder mehrere Empfänger, wobei alle beteiligten Agenten anhand ihrer *identifier* ausgewählt werden. Zusätzlich gibt es die Möglichkeit einen *reply-to*-Agenten auszuwählen, der im Fall einer Antwort auf die Nachricht, diese statt dem Sender bekommt. Ein weiterer notwendiger Parameter einer Nachricht ist die Art des Protokolls. Hierfür muss ein sogenanntes *performative* gewählt werden. Die möglichen Arten sind ebenfalls von der FIPA spezifiziert [11].

Introspector

Der *Introspector* ermöglicht es, den Zustand von Agenten zu beobachten und gegebenenfalls zu manipulieren. Die Informationen über die Agenten, die im *Introspector* angezeigt werden, werden - wenn es Änderungen gibt - automatisch angepasst und in den entsprechenden Teilbereichen visualisiert. Hierbei wird zwischen der *beliefbase*, der *goalbase*, der *planbase* und dem *debugger* unterschieden. Die *beliefbase* veranschaulicht die Fakten der Agenten. Die Werte können verändert werden, wobei Java-Ausdrücke interpretiert werden und sämtliche *import*-Statements des ADF berücksichtigt werden. Außerdem können Fakten natürlich auch gelöscht werden.

Im Bereich der *goalbase* und der *planbase* können Informationen über die *goals* und *plans* abgelesen werden. Der Typ und der Zustand eines *goals* beziehungsweise eines *plans* wird anhand eines entsprechenden Symbols repräsentiert. Die Hierarchie und Zusammenhänge der *plans* und *goals* werden in einer Baumstruktur dargestellt, wodurch die Abhängigkeiten ersichtlich sind. Auch nachdem Ziele verworfen, oder *plans* beendet wurden, sind die Daten darüber verfügbar und der Endzustand kann analysiert werden.

Die *debug*-Ansicht ermöglicht dem Anwender einen Einblick in die Aktionen eines Agenten, indem die Abläufe Schritt für Schritt aufgelistet werden und jederzeit angehalten werden können. Außerdem können zusätzliche Informationen zu jeder Aktion aufgerufen werden und in einer Detailansicht betrachtet werden.

3.5 MASON

MASON (*Multi-Agent Simulator Of Neighborhoods… or Networks… or something…*) [16] wurde vom ECLab[2] (*Evolutionary Computation Laboratory*) und dem *Center for Social Complexity*[3] entwickelt und ist eine Java-Programmbibliothek, welche die Erstellung von Multiagentensimulationen erleichtern soll. Hierbei spielt es keine Rolle, ob MASON nur die Grundlage für große Simulationsumgebungen liefert oder bereits als vollständig einsatzbereite Bibliothek Simulationen erzeugt. Es beinhaltet einerseits eine Modellbibliothek und andererseits Klassen für 2D und 3D Visualisierungen. Die Simulation einer Multi-Agenten-Umgebung kann allerdings auch komplett unabhängig von Visualisierungen durchgeführt werden. MASON-Modelle können fixiert werden und dynamisch auf anderen Plattformen wiederhergestellt werden, wobei die Resultate der Simulationen auf unterschiedlichen Plattformen identisch sind. Zusätzlich bietet MASON die Möglichkeit Momentaufnahmen abzuspeichern beziehungsweise Videos, Diagramme, Graphen und *Streams* zu erstellen.

In Abbildung 21 wird das Basiskonzept anhand der grundlegenden Elemente dargestellt.

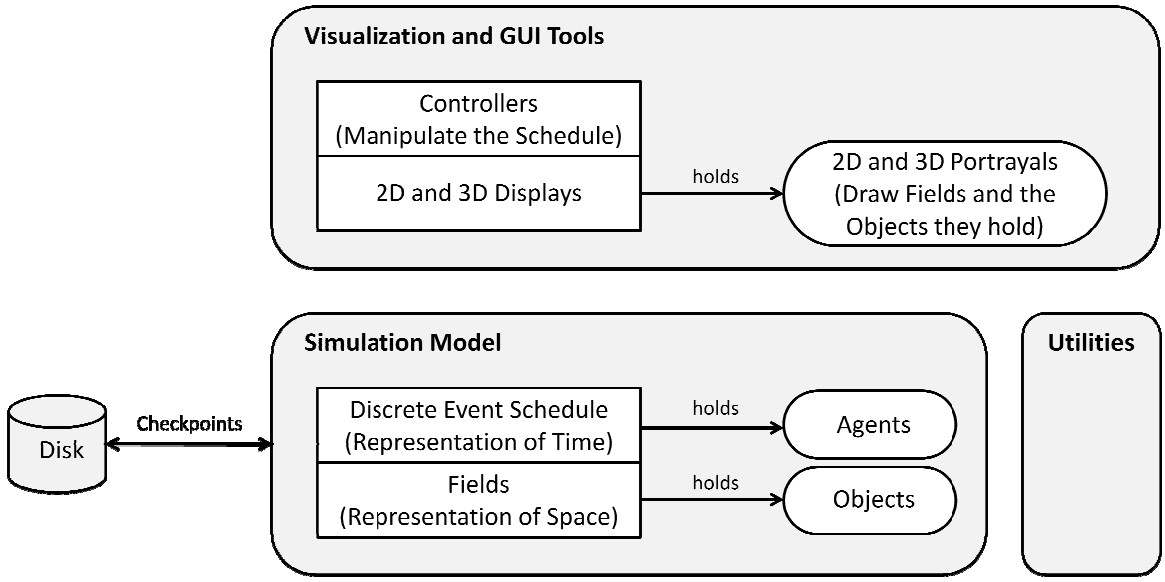

Abbildung 21 - Basiskonzept von MASON [LCPS04, S. 2]

[2]Das *Evolutionary Computation Laboratory* der George Mason University untersucht und entwickelt evolutionäre Modelle und Programme, die unter anderem bei Designproblemen, Optimierungsproblemen und maschinellem Lernen erfolgreich angewendet wurden. [17] bietet weitere Informationen über das EClab.

[3] Das *Center for Social Complexity* der George Mason University untersucht soziale Phänomene mit interdisziplinären Ansätzen, wobei alle Ebenen der sozialen Komplexität in Betracht gezogen werden [18] enthält weitere Informationen darüber.

Simulationszustände (*checkpoints*) können anhand des Zustand des Modells während der Simulation auf ein Speichermedium abgespeichert werden. Das Simulationsmodell enthält einerseits einen *schedule* und andererseits *fields*, die wiederum Agenten und Objekte beinhalten. Der *schedule* umfasst die Informationen über den zeitlichen Zustand einer Simulation und verwaltet den Aufruf der Agenten in jedem Simulationsschritt. In den *fields* wird den Agenten und Objekten, die damit verknüpft sind, eine Position in der Simulationsumgebung zugewiesen. Die Visualisierung erfolgt unabhängig von der Simulation und kann für bessere *performance* bei der Simulation entfallen. Über *controller* kann die Simulation von einem Anwender gesteuert werden. Die *utilities*, beispielsweise Komponenten für statistische Auswertungen, sind bewusst außerhalb der beiden Hauptelemente positioniert, damit sie modular und optional für Simulationen zur Anwendung kommen können.

Da sowohl für die BDI-Fauna (siehe Kapitel 4) als auch im ARS-Projekt (siehe Kapitel 2.5) MASON zur Visualisierung der Simulation verwendet wird, können die Agenten mühelos in gemeinsame Simulationen zusammengeführt werden. Jene Komponenten von MASON, die für die Visualisierung der BDI-Fauna verwendet wurden, beziehungsweise wie die Verbindungen der Komponenten der BDI-Fauna mit MASON umgesetzt wurden, werden in Kapitel 4.3, 4.4 und 4.5 erläutert.

4. BDI-Fauna

Die triebgesteuerte Fauna für *Artificial Life Simulation* wird in der vorliegenden Arbeit als BDI-Fauna bezeichnet, da sie auf einer *Belief-Desire-Intention*-Architektur (BDI-Architektur, siehe Kapitel 2.7) aufbaut. Die BDI-Fauna ist jener Teil des Simulationsprogramms, der die Logik, Attribute und Fertigkeiten der Tiere enthält und eine Verbindung zwischen den Komponenten des Programms herstellt. Die Komponenten des Frameworks, die von der BDI-Fauna verwendet werden, wurden in Kapitel 3 erläutert. Die folgenden Kapitel befassen sich mit der Analyse der Fauna, also der Ermittlung der Eigenschaften und der konkreten Anforderungen an die Tierwelt. Anschließend wird erläutert, wie diese Eigenschaften verwaltet werden. Die Strukturierung der BDI-Fauna und eine detaillierte Beschreibung der Agenten folgt in den weiteren Kapiteln. Abschließend wird die Visualisierung der Simulation vorgestellt.

4.1 Analyse der Fauna

Bei der Analyse der Fauna erfolgt die Festlegung auf zwei Tierarten. Außerdem wird entschieden welche Eigenschaften, Fähigkeiten, Bedürfnisse und Ziele die Tiere haben.

Die Tierarten sollen einen Jäger und einen Gejagten repräsentieren. Mit diesen beiden Typen können die Agenten des ARS-Projekts sowohl in Gefahrensituationen versetzt werden als auch in Situationen mit Kooperationsbedarf erprobt werden. Für den Jäger ist ein fleischfressendes Tier naheliegend und für den Gejagten ein pflanzenfressendes Tier. Für die Pflanzenfresser muss es daher pflanzliche Nahrungsquellen in der Simulationsumgebung geben.

Als Resultat dieser Anforderungen wurden Wölfe als Jäger und Hasen als Gejagte festgelegt. Für die Hasen dienen Karotten als Nahrungsquelle.

Die Wölfe jagen also Hasen und die Hasen wiederum versuchen zu fliehen und ernähren sich nur von statischen Nahrungsquellen - den Karotten - der Simulationsumgebung. Gelingt es den Wölfen nicht einen Hasen zu fangen bevor sie verhungern, sterben sie. Ebenso sterben Hasen, wenn sie ihrerseits zu lange keine Nahrung finden.

Grundlegende Aktionen der Tiere wurden bereits erwähnt. Dabei handelt es sich um die Möglichkeit sich zu bewegen und zu fressen. Die Tiere müssen fressen um zu überleben. Da die Nahrung im

Normalfall nicht zu ihnen kommt, müssen sie danach suchen und sich daher bewegen. Da sie sich nicht ohne Müdigkeitserscheinungen unentwegt bewegen können, müssen sie auch schlafen. Dadurch ergeben sich die folgenden Verhaltensweisen für die Tiere.

1. erkunden: Erkunden dient dem Bewegen in der Simulationsumgebung.
2. schlafen: Schlafen ist notwendig um die Tatkraft wiederherzustellen.
3. fressen: Hierbei wird Nahrung aufgenommen um die Energie wieder aufzufüllen.
4. jagen: Ein anderes Tier wird verfolgt um es zu töten und schließlich zu fressen.
5. töten: Ein anderes Tier wurde eingeholt und wird getötet um es zu fressen.
6. flüchten: Flüchten bezeichnet den Versuch einem anderen Tier zu entkommen.

Diese Aktionen werden in Abbildung 22 in Relation zueinander gesetzt.

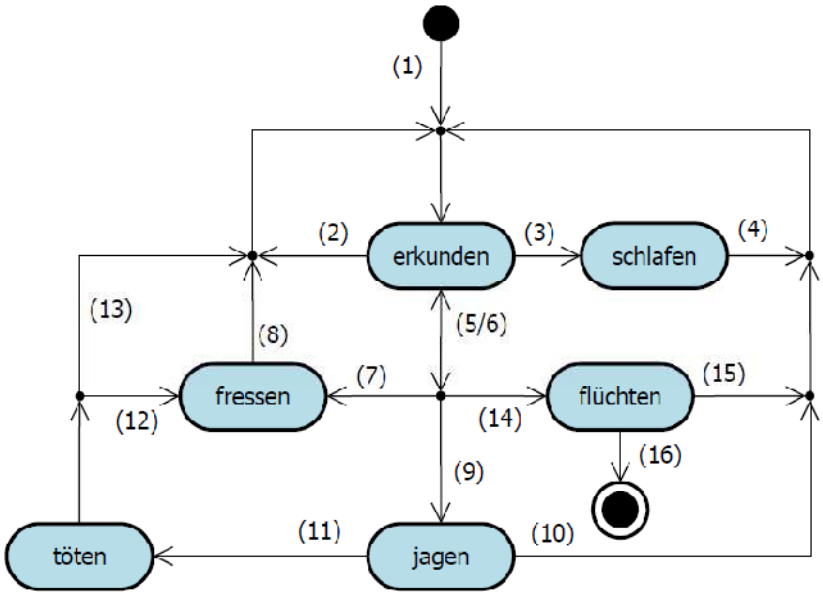

Abbildung 22 - Aktionen der Tiere

Wird ein Tier der Simulationsumgebung hinzugefügt, wurde davor noch keine Aktion durchgeführt (1). Das Tier startet mit der Erkundung der Umgebung, wobei bereits andere Objekte im Sichtbereich sein können. So lange keine anderen Objekte entdeckt werden, wird die Erkundung fortgesetzt (2). Wenn die Tatkraft ausgeschöpft ist, beendet das Tier das Erkunden und schläft (3). Nach dem Schlafen ist es ausgeruht und erkundet wieder (4).

Bei der Erkundung können andere Objekte im Sichtbereich des Tieres entdeckt werden, wodurch sich mehrere Entscheidungsmöglichkeiten ergeben (5): Ist das entdeckte Objekt eine statische Nahrungsquelle, kann das Tier sich entscheiden sich zu ihr zu bewegen und zu fressen (7) oder die Nahrungsquelle wird ignoriert, weil das Tier keinen Hunger hat oder sie für das Tier als Nahrung nicht

in Frage kommt (6). Beim Fressen wird die Energie wieder aufgefüllt, wobei ein Tier nicht unbegrenzt Nahrung aufnehmen kann. Es hört also auf zu fressen, wenn es satt ist (8). Fleischfressende Tiere, die ein anderes Tier entdecken, können sich dazu entschließen es zu jagen (9). Diese Jagd kann einerseits erfolglos verlaufen, wenn das gejagte Tier entkommt, wodurch der Jäger wieder die Umgebung erkundet (10). Andererseits kann sie von Erfolg gekrönt sein, wenn das gejagte Tier eingeholt und anschließend getötet wird (11). In diesem Fall ist es vom Hunger des Jägers abhängig, ob er frisst (12) oder sofort die Umgebung erkundet um vielleicht weitere Beute zu entdecken (13). Ein Tier, das ein anderes Tier als Gefahr wahrnimmt, wird danach trachten zu flüchten (14). Die Flucht kann entweder vorüber sein, weil die Gefahrenquelle nicht mehr in Sichtweite ist (15), oder weil das Tier getötet wurde (16).

Für die Umsetzung der Tiere sind einige Fragen für jede Tierart zu klären, die in den folgenden Absätzen behandelt werden.

Zur Gewährleistung des eigenen Überlebens müssen Tiere regelmäßig fressen. Hierfür sind unter anderem folgende Fragen zur Festlegung der Eigenschaften eines Tieres relevant:

- Wie oft muss das Tier fressen?
- Wie viel kann das Tier auf einmal aufnehmen?
- Welche Nahrung bevorzugt das Tier?
- Wie schnell kann es Nahrung aufnehmen?
- Wie schnell verbraucht das Tier bei verschiedenen Aktionen Energie?

Zur Erweiterung der Simulation können die gleichen Fragen auch für den Flüssigkeitshaushalt eines Tieres gestellt werden.

Außerdem gibt es noch Eigenschaften, die das Äußere eines Tieres beschreiben:

- Welche Maße hat das Tier?
- Wie wird es dargestellt?

Da die Tiere auch Aktionen durchführen sollen, muss geklärt werden, welche das sein können und wodurch sie abgesehen von der Nahrungsaufnahme ausgelöst und beeinflusst werden:

- Kann das Tier laufen/fliegen/schwimmen und wie schnell ist es dabei?
- In welchen Abständen muss das Tier schlafen und wie lange muss es ruhen um wieder ausgeruht zu sein?
- Über welche Sinne verfügt das Tier?
- Wie ausgeprägt sind die einzelnen Sinne?

Einerseits haben Tiere innere Bedürfnisse, die ihr Handeln beeinflussen, und andererseits reagieren sie auf äußere Reize, die sie durch ihre Sinnesorgane wahrnehmen. Sämtliche Aktionen der Tiere dienen dem eigenen Überleben beziehungsweise dem Überleben der Gattung im übertragenen Sinn.

Eine für das Überleben entscheidende Komponente ist die physische Stärke eines Tieres. Sie entscheidet darüber, ob ein Tier im Fall einer Konfrontation mit einem feindlich gesinnten Tier flieht

oder angreift. Endet eine derartige Konfrontation mit dem Tod eines Tieres, muss festgelegt werden, welchen Nahrungswert das tote Tier für andere Tiere darstellt.

Das Verhalten der Tiere soll nachvollziehbar sein und nicht nur zufälligen Bewegungsmustern entsprechen oder unrealistisch sein. Dies macht eine Abstimmung der Eigenschaften aufeinander notwendig. Damit ist zum Beispiel das Verhältnis der Bewegungsgeschwindigkeiten von unterschiedlichen Tieren gemeint, aber ebenso die Geschwindigkeit der Nahrungsaufnahme im Vergleich zum Energieverbrauch eines einzelnen Tieres.

4.2 Verwaltung der Eigenschaften

Die Verwaltung der Eigenschaften erfolgt über Java-*Properties*-Dateien [19]. Die Repräsentation der Eigenschaften erfolgt über Schlüssel-Wert-Paare, die jeweils beide vom Typ String sind und durch ein Gleichheitszeichen getrennt werden. Für jede Eigenschaft ist eine Zeile in der Datei vorgesehen. Die beiden folgenden Kapitel beschreiben die Eigenschaften der Tiere und jene der Nahrungsquellen, die gleichzeitig für tote Tiere verwendet werden.

4.2.1 Eigenschaften der Tiere

Die Eigenschaften der Tiere, die sich aus der Analyse in Kapitel 4.1 ergeben, müssen für die Initialisierung der Agenten gespeichert werden können. Manche Eigenschaften des Agenten beeinflussen die Art der Darstellung in der Simulation, andere bestimmen die Fähigkeiten und wieder andere beschreiben die physischen Gegebenheiten der unterschiedlichen Tiere.

Für jede Tierart werden jeweils zwei *Properties*-Dateien benötigt um zwischen den Eigenschaften eines lebenden Tieres und eines toten Tieres unterscheiden zu können. Diese Dateien beinhalten die Eigenschaften und dienen der Repräsentation von Tierarten oder einzelnen Individuen. Die Eigenschaften von lebenden Tieren beeinflussen das Verhalten der jeweiligen Agenten, jene von toten Tieren die von Objekten, die anderen Tieren als Nahrungsquelle dienen können. Die Eigenschaften von Nahrungsquellen werden im Kapitel 4.2.2 näher behandelt. Nahezu das gesamte Verhalten und die Visualisierung der Tiere kann über die Eigenschaften in den *Properties*-Dateien gesteuert werden. Tabelle 4 liefert einen Überblick über die Eigenschaften von Tieren, die über die *Properties*-Dateien verwaltet werden. Da tote Tiere als Nahrungsquelle betrachtet werden, sind deren Eigenschaften auch über *Properties*-Dateien definiert, deren Attribute denen von Nahrungsquellen gleichen. Die Eigenschaften von toten Tieren werden also auch in Tabelle 5 in Kapitel 4.2.2 erläutert.

Tabelle 4 - *Properties* von Tieren

Eigenschaft	Werte	Beschreibung
speed_running	0 - ∞	Legt die Bewegungsgeschwindigkeit eines Tieres fest. Die Werte werden als Pixelentfernung interpretiert. Der Wert null bedeutet, dass keine Bewegung möglich ist. Unendlich ist nur ein theoretischer Wert, da die Bewegungsmöglichkeiten durch die Größe der Simulationsfläche eingeschränkt sind.
vision_range	0- ∞	Beschreibt die Sichtweite eines Tieres. Die Werte werden als Pixelentfernung interpretiert. Die Sicht eines Tieres entspricht einem Kreis, wobei *vision_range* den Radius definiert. Der Wert null bedeutet, dass das Tier blind ist. Unendlich ist nur ein theoretischer Wert, da die Sichtweite durch die Größe der Simulationsfläche eingeschränkt ist.
width / height	0 - ∞	Stellen die Maße eines Tieres in Pixel dar. Ein Bild zur Visualisierung des Tieres wird entsprechend groß angezeigt.
image	*/*.png	Das Bild, welches das Tier in der Simulationsumgebung repräsentiert, wird aus dem entsprechenden Pfad geladen.
food_preference_flesh	0, 1	Hält fest, ob das Tier Fleisch frisst. Der Wert wird als *Boolean*-Wert behandelt.
food_preference_plant	0, 1	Hält fest, ob das Tier Pflanzen frisst. Der Wert wird als *Boolean*-Wert behandelt.
stomach_size	0 - ∞	Beschreibt die maximale Aufnahmefähigkeit von Nahrung. Je größer der Wert ist, desto mehr Nahrung kann ein Tier aufnehmen.
speed_eating	0 - ∞	Legt die Fressgeschwindigkeit eines Tieres fest. Null bedeutet, dass das Tier zwar frisst, aber keine Nahrung aufnimmt. Eine unendlich hohe Geschwindigkeit bewirkt das sofortige Auffüllen des Magens des Tieres bis zur höchstmöglichen Aufnahmefähigkeit (siehe *stomach_size*). Die Fressgeschwindigkeit sollte höher sein als der Nahrungs- beziehungsweise Energieverbrauch während der Untätigkeit (siehe *food_req_idle*), da sonst auch nach der Nahrungsaufnahme dem Tier weniger Energie zur Verfügung steht als vorher.

food_req_idle	0 - ∞	Bezeichnet den Nahrungs- beziehungsweise Energieverbrauch eines Tieres unabhängig von etwaigen Aktionen. Je höher der Wert ist, desto mehr Energie wird verbraucht. Übersteigt der Wert die maximale Aufnahmefähigkeit von Nahrung (siehe *stomach_size*), stirbt das Tier sofort, da es mehr Nahrung verbraucht als es maximal zur Verfügung haben kann. Wenn, abgesehen von dieser Eigenschaft auch der Nahrungsverbrauch während der Bewegung (siehe *food_req_running*) null ist, muss das Tier nie Nahrung zu sich nehmen.
food_req_running	0 - ∞	Bezeichnet den Nahrungsverbrauch, während sich das Tier in der Simulationsumgebung bewegt. Je höher der Wert, desto höher der Energieverbrauch.
need_sleep	1 - ∞	Beschreibt den Grenzwert, bei dem ein Tier einen etwaigen Schlaf beendet, weil es ausgeschlafen hat. In Verbindung mit der Erholungsrate (siehe *need_sleep_recovery*) und der Müdigkeitszunahme (siehe *need_sleep_consumption*) regelt dieser Grenzwert die Häufigkeit und Dauer des Schlafes.
need_sleep_recovery	0 - ∞	Legt die Erholungsrate während des Schlafens fest. Je höher der Wert ist, desto schneller regeneriert sich das Tier. Bei einem Wert von null, würde das Tier einen begonnenen Schlaf - ohne Gefahr im Verzug oder dem eigenen Tod - nicht mehr beenden.
need_sleep_consumption	0 - ∞	Bezeichnet die Müdigkeitszunahme, wobei ein höherer Wert mehr Schlafpausen hervorruft. Der Wert null bewirkt, dass das Tier nie schlafen muss.
attack_power	0 - ∞	Repräsentiert die Angriffsstärke eines Tieres, wobei ein Tier mit einer Angriffsstärke von null niemals angreifen wird, und ein Wert von unendlich ein unbesiegbares Tier darstellen würde.
elem_dead	#name	Der gewählte Name bezeichnet eine *Properties*-Datei, welche die Informationen beinhaltet, die für den Kadaver des Tieres benötigt werden.

4.2.2 Eigenschaften der Nahrungsquellen

Die Eigenschaften der Nahrungsquellen beinhalten hauptsächlich Eigenschaften für die Art ihrer Darstellung in der Simulation. Außerdem ist der Typ der Nahrungsquelle und die verfügbare Nahrungsmenge von Bedeutung. In Tabelle 5 werden die Eigenschaften von Nahrungsquellen, die über die *Properties*-Dateien verwaltet werden, angeführt. Diese Eigenschaften gelten ebenfalls für tote Tiere, da sie in der Simulation als Nahrungsquelle für andere Tiere behandelt werden.

Tabelle 5 - Properties von Nahrungsquellen

Eigenschaft	Werte	Beschreibung
width / height	0 - ∞	Stellen die Maße einer Nahrungsquelle in Pixel dar. Ein Bild zur Visualisierung der Nahrungsquelle wird entsprechend groß angezeigt.
image	*/*.png	Das Bild, das die Nahrungsquelle in der Simulationsumgebung repräsentiert, wird aus dem entsprechenden Pfad geladen.
food_type_flesh	0, 1	Gibt an, ob es sich um fleischliche Nahrung handelt. Der Wert wird als *Boolean*-Wert behandelt. Auch Mischtypen, also sowohl pflanzliche als auch fleischliche Nahrung in einer Nahrungsquelle, sind möglich (siehe *food_type_plant*).
food_type_plant	0, 1	Gibt an, ob es sich um pflanzliche Nahrung handelt. Der Wert wird als *Boolean*-Wert behandelt. Auch Mischtypen, also sowohl pflanzliche als auch fleischliche Nahrung in einer Nahrungsquelle, sind möglich (siehe *food_type_flesh*).
food_value	1 - ∞	Legt den Initialwert für die vorhandene Nahrungsmenge fest. Ein unendlich großer Wert repräsentiert eine unerschöpfliche Nahrungsquelle.

4.3 Struktur des Programms

Das Programm verwendet drei Komponenten: JADEX, MASON und die neu erstellte BDI-Fauna. JADEX bietet ein *Framework* zur Erstellung und Verwendung von Agenten. MASON ermöglicht die Erstellung und Visualisierung von Simulationen. Die Simulation wird in mehrere Simulationsschritte unterteilt, wobei alle dafür registrierten Elemente in einem Simulationsschritt aufgerufen werden. Diese Funktionalität wird verwendet um die Agenten zu simulieren. Die BDI-Fauna stellt eine Verbindung zwischen den beiden *Frameworks* her und enthält die Definition und Funktionalität der Fauna, die simuliert wird.

In Abbildung 23 werden die Komponenten des Programms und wichtige Teile davon unter Berücksichtigung ihrer Relation zueinander dargestellt. Hierbei werden zur besseren Übersichtlichkeit manche Relationen bewusst nicht dargestellt und die Klassenstruktur wird auf wesentliche Teile reduziert oder zusammengefasst um einen groben Überblick der Struktur zu veranschaulichen.

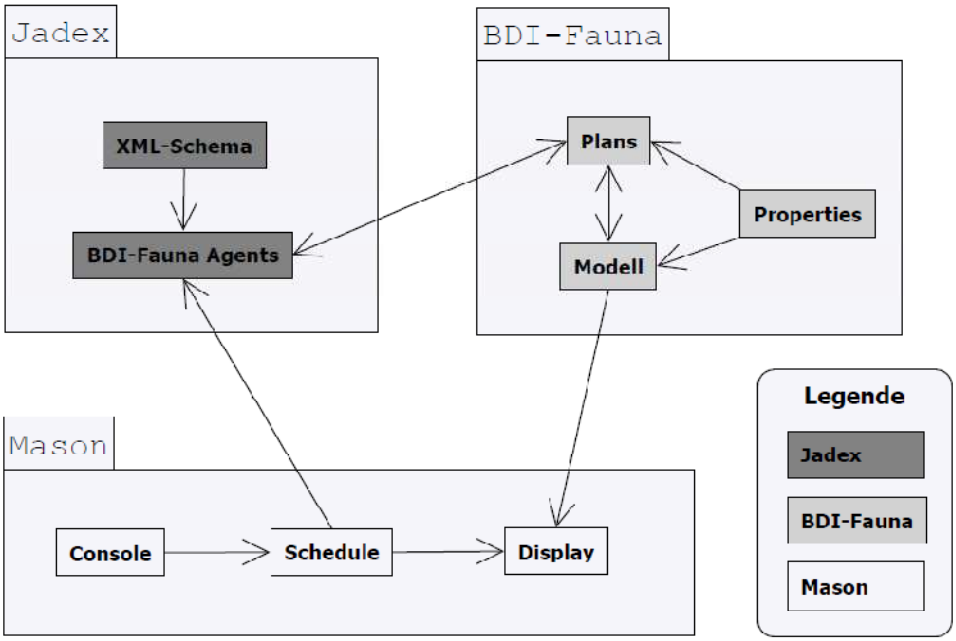

Abbildung 23 - Komponenten des Programms

Wie bereits angesprochen, besteht das Programm aus den drei großen Teilen JADEX, MASON und der BDI-Fauna. Das Programm wird über einen Agenten aus dem JADEX *Control Center* (siehe JADEX *Control Center* in Kapitel 3.4) gestartet. Hierbei wird eine MASON-*Console* erzeugt, die zur Simulationssteuerung einen Schedule beinhaltet. Im Schedule werden die Agenten von JADEX geplant um eine Simulation für alle Agenten zu erstellen. Bei der JADEX-Komponente spielen die BDI-Fauna Agenten eine wesentliche Rolle bei der Verknüpfung mit den anderen beiden Kompo-

nenten. Die Agenten sind von der XML-Schema-Definition des JADEX-*Frameworks* abhängig. Sie werden anhand dieser Definition erstellt und sind in ihren Möglichkeiten auf die Funktionen begrenzt, die im XML-Schema definiert sind. Einzig die *plans* der Agenten, die aus Java-Klassen aufgebaut werden, sind - abgesehen von der Einbettung in das ADF (siehe Kapitel 3.3.3 und Kapitel 3.3.7) - von dem XML-Schema unabhängig. Die *plans* stellen somit die Verbindung zum Modell der BDI-Fauna dar. Sowohl die *plans* als auch das Modell werden durch die *Properties*-Dateien (siehe Kapitel 4.2) der jeweiligen Simulationskonfiguration beeinflusst. Der aktuelle Zustand der BDI-Fauna wird im *Display* der MASON-Simulation angezeigt.

In Abbildung 24 wird auf die Struktur detaillierter eingegangen, wobei auch hier zur besseren Übersichtlichkeit nicht alle Relationen angeführt werden.

Die einzelnen Klassen der drei Komponenten des Programms sind farblich hervorgehoben um die Unterscheidung zu vereinfachen. Die Kernkomponenten von JADEX werden in einem Paket zusammengefasst. Damit wird verdeutlicht, dass die Agenten und die Klasse *clsCreature* mit allen angeführten Teilen von JADEX interagieren.

Die beiden Agenten - *Creature Agent* und *Manager Agent* - haben abgesehen von der dargestellten Relation eine Beziehung über das Kommunikationsprotokoll. Die Agenten senden sich untereinander Nachrichten um Informationen über getätigte Aktionen oder den aktuellen Zustand der Simulationsumgebung auszutauschen. Die abgebildete Relation repräsentiert die Aufgabe des *Manager Agents* die *Creature Agents* zu verwalten. Er ist hierfür mit einer *capability* des JADEX-*Frameworks* (siehe Kapitel 3.3.4) ausgestattet, die es ihm ermöglicht Agenten zu starten und zu beenden.

Die Java-Klassen der BDI-Fauna interagieren mit den Agenten über das *IExternalAccess*-Interface (siehe Kapitel 3.3.10), das von JADEX für den Zugriff externer Prozesse auf Agenten zur Verfügung gestellt wird um die *Thread*-Synchronisation zu gewährleisten.

Die Schnittstelle zwischen dem Modell und der MASON-Simulation erfolgt über das Interface *Steppable*. Dieses Interface wird unter anderem verwendet um zu definieren, welche Aktionen bei einem Simulationsschritt durchgeführt werden. Jedes Element, das von MASON simuliert werden soll, muss dieses Interface implementieren. Die Simulationsschritte bieten eine Möglichkeit die Agenten zu synchronisieren und dadurch Fairness zu gewährleisten. Jeder Agent kann hierdurch pro Simulationsschritt nur jeweils eine Aktion durchführen, bevor er auf den nächsten Simulationsschritt warten muss.

Eine weitere Schnittstelle zwischen der BDI-Fauna und MASON wird über die Klasse *clsTestFauna* hergestellt. Diese Klasse erweitert die MASON-Klasse *SimState* und repräsentiert damit einen Simulationszustand. In diesem Zustand werden sämtliche Daten verwaltet, die eine Simulation zu einem bestimmten Zeitpunkt beschreiben. Hierzu zählt auch der aktuelle Wert der bereits vergangenen Simulationsschritte. Der *schedule* einer Simulation und damit auch die für die Simulation registrierten *Steppable*-Elemente sind ein Teil des durch *SimState* beschriebenen Zustands.

Die Klasse *clsTestFaunaWithUI* bringt den Simulationsstand von *clsTestFauna* in das GUI. Sie repräsentiert einen GUI-*State* und bildet den Einstiegspunkt für eine Simulation.

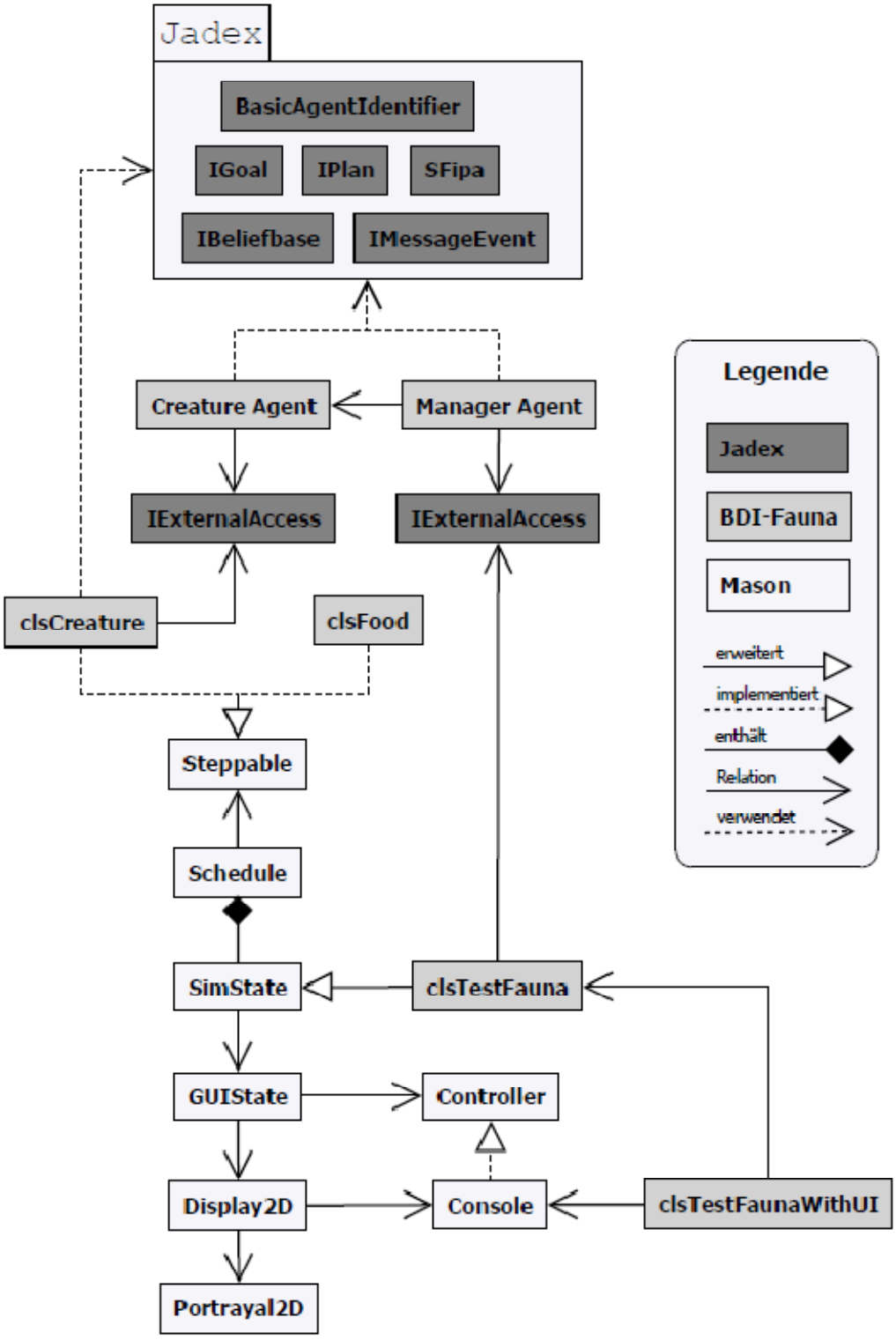

Abbildung 24 - Zusammenspiel der Komponenten

4.4 Repräsentation der Agenten

Die Agenten werden bei JADEX in XML-Dateien beschrieben und für die Simulation geladen. Diese Dateien werden *Agent Definition Files* genannt. Alle Agenten haben aufgrund des XML-Schemas von JADEX einen vorgegebenen Aufbau (siehe Kapitel 3.3.3).

Das erste Unterkapitel ist der Erläuterung des *Manager-Agents* gewidmet. Eine ebenso strukturierte Beschreibung folgt danach für den *Creature-Agent*. Die Kommunikation dieser beiden Agententypen wird im abschließenden Teil über den Nachrichtentransfer behandelt.

4.4.1 Manager-Agent

Der *Manager-Agent* erfüllt die Verwaltung der Simulationsumgebung. Er beinhaltet die Startroutinen, die erforderlich sind um das MASON-*Framework* zu starten und die Visualisierung einzurichten. Im Abschnitt über das ADF des *Manager-Agents* werden die Elemente der Agentenbeschreibung behandelt. Im darauf folgenden Abschnitt über die *plan*-Klassen werden die Java-Klassen zur Umsetzung der Aktionen des Agenten beschrieben.

Die anderen Agenten der Simulation werden mit Hilfe des *Manager-Agents* gestartet. Hierfür wird die *Agent Management System Capability*, eine vordefinierte *capability* aus dem JADEX-*Framework* verwendet. Sämtliche Informationsanfragen, die von anderen Agenten in Bezug auf die Simulationsumgebung geschickt werden, werden vom *Manager-Agent* entgegengenommen und verarbeitet. Hierzu gehören zum Beispiel Anfragen zur Umgebung eines Agenten oder die Anfrage ob noch Nahrung von einer Nahrungsquelle übrig ist. Der *Manager-Agent* liefert den anderen Agenten auf Anfrage jene Informationen über die Umgebung, die von den Sensoren der Tiere wahrgenommen werden können.

Außerdem bietet der *Manager-Agent* Funktionalität und Anknüpfungspunkte für ein *User*-Interface zur Verwaltung der vorhandenen Tiere und Objekte in der Simulation durch einen Anwender.

Agent Definition File

Im *Agent Definition File* (ADF) des *Manager-Agents* werden Tiere festgelegt, die beim Erstellen der Simulationsumgebung bereits Teil davon sind und sich schon darin bewegen, wenn die Simulation startet. Das gleiche gilt für Nahrungsquellen, die bereits von Beginn an vorhanden sind. Diese beiden Einstellungen werden über *beliefsets* getroffen, die jeweils ein Fakt pro Tier beziehungsweise pro Nahrungsquelle beinhalten.

Da der *Manager-Agent* für die Verwaltung der Simulation beziehungsweise der Simulationsumgebung verantwortlich ist, wird sowohl das GUI als auch die 2D-Ebene in jeweils einem *belief* gespeichert.

Die *goals* des Agenten sind beide vom Typ *achieve goal* (siehe Kapitel 3.3.6). Sie beinhalten die Funktionalität aus der *Agent Management System Capability* um Agenten zu erstellen beziehungsweise zu löschen. Außerdem verfügt er über ein *goal* für das Hinzufügen von Nahrungsquellen zur

Simulationsumgebung. Eine ähnliche Vorgangsweise kann hier verwendet werden um Agenten während einer laufenden Simulation zur Umgebung hinzuzufügen. Auf diese und andere Erweiterungen wird in Kapitel 6 näher eingegangen.

Die *plans* des *Manager-Agents* beschränken sich auf die Verarbeitung von Anfragen zur Nahrungsaufnahme und die Abarbeitung des *goals* für das Hinzufügen von Nahrungsquellen. Die *trigger* für diese *plans* sind daher ein *message event* für ersteren und das Verfolgen eines *goals* für letzteren.

Um die Interaktion mit den Tieren zu ermöglichen, verarbeitet der Agent zwei *message events*. Einerseits bidirektional die Anfragen zur Nahrungsaufnahme und andererseits eine Nachricht an die Tiere, die von jedem Tier separat ausgelöst wird und die Elemente in dessen Umfeld enthält. Die Nachricht zur Beschreibung der Elemente im Sichtbereich eines Tieres entspricht der Abfrage eines Sensors. Diese Abfrage wird von jedem Tier bei jedem Simulationsschritt durchgeführt. Weitere Details zum Nachrichtentransfer werden in Kapitel 4.4.3 behandelt.

Die *default*-Konfiguration des *Manager-Agents* enthält den *belief* für das GUI. Das GUI wird mit diesem *belief* instanziiert. Beim Initialisieren des GUI wird ein *GUIState*-Objekt des MASON-*Frameworks* erzeugt, das die Simulationsoberfläche erstellt. Damit das GUI auf die Funktionalität des *Manager-Agents* zugreifen kann, erhält der *Constructor* ein *RCapability*-Objekt von der *getExternalAccess*-Methode des Agenten. Die Funktionsweise des externen Zugriffs auf Agenten wurde in Kapitel 3.3.10 behandelt.

Plan-Klassen

Die *plan*-Klassen des *Manager-Agents* beinhalten die Aktionsmöglichkeiten des Agenten um auf Ereignisse zu reagieren oder *goals* zu erreichen. Abgesehen von der Initialisierungsabläufen werden zwei *plans* vom Agenten verwendet. Einerseits ein *plan* um neue Nahrungsquellen der Simulationsumgebung hinzuzufügen und andererseits ein *plan*, der auf das *message event* zur Nahrungsaufnahme reagiert.

Um die Nahrungsaufnahme zu koordinieren - damit beim gleichzeitigen Fressen einer Nahrungsquelle von mehreren Tieren nicht fälschlicherweise mehr Nahrung von den Tieren aufgenommen werden kann als insgesamt zur Verfügung steht - wird der *Manager-Agent* in diesen Prozess eingebunden. Er reagiert auf die Anfragen der Tiere und verweigert die Nahrungsaufnahme eines Tieres, wenn keine Nahrung mehr bei der jeweiligen Nahrungsquelle zur Verfügung steht.

Eine Alternative zu dieser Vorgangsweise wäre die Koordinierung über die Nahrungsquelle selbst. Diese Möglichkeit würde allerdings bedeuten, dass die Tiere entweder ein Java-Objekt direkt aufrufen, wodurch eine starke Kopplung zwischen den beiden Elementen besteht, oder mit einer Nahrungsquelle kommunizieren müssen. Die Kommunikation mit einem Objekt hätte zur Folge, dass einem statischen Objekt in der Simulationsumgebung Interaktionsmöglichkeiten gegeben werden müssen. Die Kommunikationsmöglichkeiten würden dann - abgesehen von der Kommunikation von einem Agenten zu einem anderen Agenten - auch einen Nachrichtenaustausch zwischen Agenten und Objekten umfassen. Zur Nachrichtenübermittlung mit der ACL der FIPA, die für den Nachrichtenaustausch der Agenten in der BDI-Fauna verwendet wird, müssten die Objekte allerdings zu Agenten werden. Dies würde einen zusätzlichen Implementierungsaufwand notwendig machen, der die Komplexität der Nahrungsquellen erhöhen würde.

4.4.2 Creature-Agent

Der *Creature-Agent* definiert ein beliebiges Tier, wobei grundlegende Verhaltensweisen durch die vorhandenen *beliefs*, *goals*, *plans* und *events* vorgegeben sind. Davon abgesehen werden die Tierarten aufgrund der unterschiedlichen *Properties*-Dateien (siehe Kapitel 4.2.1) individualisiert. Die folgenden beiden Abschnitte beschreiben den Aufbau des ADF des *Creature-Agents* und erläutern dessen *plan*-Klassen.

Agent Definition File

Das ADF beschreibt die Tiere der BDI-Fauna. Die *beliefs* enthalten Informationen über das jeweilige Tier, die zur Laufzeit verändert werden, ebenso wie die Eigenschaften aus den *Properties*-Dateien. Hierzu zählt auch ein *belief*, der eine Referenz auf das Java-Objekt des Modells des Tieres enthält. Zusätzlich werden Daten der Simulationsumgebung, wie zum Beispiel das jeweils aktuelle Sichtfeld, verwaltet. Der *belief*, der die Simulationsschritte repräsentiert, ist ein wesentlicher Aspekt um die Agenten mit der Simulationsumgebung von MASON zu synchronisieren. Über die Aktualisierungsmethode, die bei jedem Simulationsschritt aufgerufen wird, und den *external access* (siehe Kapitel 3.3.10) des Agenten kann der *belief* aktualisiert werden. Dadurch weiß der Agent, wann die Simulation einen weiteren Schritt vorwärts macht. Die Aktionen des Agenten hängen zum Teil von dieser Information ab, da sie nur dann durchgeführt werden, wenn die Simulation um einen Simulationsschritt voranschreitet.

Ein Tier hat *goals* für sämtliche Verhaltensweisen, die in Kapitel 4.1 angeführt worden sind. Da für diese Verhaltensweisen nicht immer nur ein *goal* alle Aspekte des Verhaltens abdecken kann, werden für manche davon mehrere *goals* definiert. Insbesondere die Auswahl eines Ziels für die Bewegung des Tieres wird in mehreren *goals* behandelt, wobei aufgrund der *deliberation* (siehe Kapitel 3.3.6) und der *expressions* (siehe Kapitel 3.3.9) in den *conditions* nur jeweils ein *goal* gleichzeitig verfolgt wird. Diese *goals* umfassen die Auswahl eines Ziels, wenn sich die folgenden Elemente im Sichtbereich befinden:

- kein Element
- eine Nahrungsquelle
- ein totes Tier
- ein schwächeres Tier
- ein stärkeres Tier

Diese Situationen werden abhängig vom aktuellen Zustand des Tieres in unterschiedlichen *goals* behandelt. Außerdem beeinflussen ein weiteres oder auch mehrere weitere Elemente im Sichtbereich die Auswahl des zu verfolgenden *goals*.

Sämtliche *goals*, die für die Zielbestimmung verwendet werden, sind vom Typ *achieve goal*. Für die Aufrechterhaltung der Tatkraft wurde ein *maintain goal* vorgesehen. Dadurch wird gewährleistet, dass ein Tier immer danach strebt zu schlafen, wenn es müde wird. Zusätzlich werden zwei *perform goals* verwendet um Initialisierungsvorgänge des Agenten abzubilden.

Für die *goals* gibt es entsprechende *plans*, die mit *triggern* versehen sind um *goals* oder *message events* zu behandeln. Ebenso wie es mehrere *goals* gibt, die bei der Auswahl eines Ziels für die Bewegung des Tieres relevant sind, gibt es auch mehrere *plans*. Diese werden je nach Situation ausgewählt. Um das ausgewählte Ziel erreichen zu können, wird ein *plan* für die tatsächliche Bewegung des Tieres definiert. Ebenso für das Schlafen und das Fressen.

Für das Töten eines anderen Tieres wird ein *plan* verwendet, der im Erfolgsfall eine Nachricht an das getötete Tier schickt. Dieses kann anschließend vom *Manager-Agent* aus der Simulation entfernt werden und durch eine entsprechende Nahrungsquelle für das tote Tier ersetzt werden.

Die *message events* des *Creature-Agents* befassen sich mit den gleichen Nachrichten, die der *Manager-Agent* behandelt. Die FIPA-Nachrichten werden bei beiden Agenten definiert, wobei die Richtung der Nachrichtenübermittlung entsprechend angepasst werden muss. Eine detailliertere Beschreibung des Nachrichtentransfers ist in Kapitel 4.4.3 zu finden.

Bei der Initialisierung des Agenten werden die Eigenschaften des Tieres aus der jeweiligen *Properties*-Datei geladen. Zusätzlich werden *beliefs* mit Werten gefüllt. Einer dieser *beliefs* ist die *default-Tolerance*, die festlegt bis zu welchem Abstand zwischen den Positionen von zwei Objekten der Simulationsumgebung für diesen Agenten die Positionen als gleich betrachtet werden.

Plan-Klassen

Mit Hilfe der *plan*-Klassen können die Aktionen der Tiere durchgeführt werden. Sie werden je nach Situation für die *goals* ausgewählt und werden in Java-Klassen definiert. In Tabelle 6 werden die *plan*-Klassen, die vom *Creature-Agent* verwendet werden, aufgelistet. Die einzige *plan*-Klasse, die bei jedem Simulationsschritt verwendet wird, ist die Klasse *analyseVisionPlan*. Diese Klasse wertet das Objekt, das den Sichtbereich des Tieres beschreibt, aus und aktualisiert *beliefs* des Agenten.

Tabelle 6 - Plan-Klassen des Creature-Agents

Plan-Klasse	Beschreibung
loadPropertiesPlan	Diese Klasse wird für Initialisierungsaktionen verwendet. Nach der Ausführung dieses *plans* hat der Agent die Eigenschaften aus der *Properties*-Datei in der Wissensbasis abgespeichert.
sleepPlan	Da das Tier von Zeit zu Zeit schlafen muss, wird in dieser *plan*-Klasse beschrieben, welche Aktionen hierbei durchgeführt werden. Die Tatkraft des Tieres wird in jedem Simulationsschritt um einen festgelegten Wert, der aus der *Properties*-Datei bestimmt wird, erhöht.

movePlan	Hierbei wird die tatsächliche Bewegung des Tieres durchgeführt. Außerdem wird die Tatkraft reduziert und der Energieverbrauch abgespeichert. Die Bewegungsrichtung hängt von der aktuellen Zielposition ab. In diese Richtung bewegt sich das Tier mit der Bewegungsgeschwindigkeit.
analyseVisionPlan	Der *analyseVisionPlan* verarbeitet das *clsVision*-Objekt, das den aktuellen Sichtbereich des Tieres repräsentiert, wenn ein neuer Simulationsschritt durchgeführt wird und dadurch eine entsprechende Nachricht vom *Manager-Agent* geschickt wurde.
randomTargetPlan	In dieser *plan*-Klasse wird ein zufälliger Punkt innerhalb des Sichtbereichs des Tieres als neue Zielposition für die Bewegung festgelegt. Diese zufällige Position bewirkt ein zufälliges Erkunden der Umgebung.
foodPlan	Hierbei wird die Zielposition für die Bewegung des Tieres auf jene Position gesetzt, auf der sich eine statische Nahrungsquelle befindet.
eatPlan	Die Nahrungsaufnahme erfolgt bei der Abarbeitung dieser *plan*-Klasse. Solange der *plan* aktiv bleibt, wird bei jedem Simulationsschritt Nahrung aufgenommen bis entweder die maximale Aufnahmemenge erreicht ist oder die Nahrungsquelle verbraucht ist.
huntPlan	Zur Verfolgung eines anderen Tieres wird die *huntPlan*-Klasse verwendet. Sie bestimmt die Position des zu jagenden Tieres und speichert die entsprechende Zielposition für die eigene Bewegung ab. Das Jagen eines anderen Tieres wird nur dann ausgeführt, wenn es sich bei dem zu jagenden Tier um ein schwächeres Tier handelt.
fleeTargetPlan	Um einem anderen Tier zu entkommen werden die Aktionen im *fleeTargetPlan* ausgeführt. Hierfür wird die Position des gegnerischen Tieres ermittelt und die Bewegungsrichtung in die entgegengesetzte Richtung festgelegt.
killPlan	Diese Klasse löst ein *message event* aus, das den Tod eines anderen Tieres bedeutet.
gotKilledPlan	Das *message event* der *killPlan*-Klasse wird in dieser Klasse verarbeitet. Das Ereignis bedeutet für das Tier, das es soeben getötet wurde.

Manche der oben angeführten *plan*-Klassen weisen in der Implementierung große Ähnlichkeiten auf, wurden aber aufgrund der dadurch verbesserten Wartbarkeit und größerer Übersichtlichkeit in separaten Klassen behandelt. Eine Alternative um die Fallunterscheidung zwischen ähnlicher Funktionalität trotz einer gemeinsamen *plan*-Klasse zu gewährleisten, wäre die Verwendung von Parametern für die *plan*-Klassen. Weiters wäre es möglich speziell für die Unterscheidung vorgesehene *beliefs* zu verwenden, die innerhalb der *plan*-Klassen die Funktionalität vorgeben.

4.4.3 Nachrichtentransfer

Die Agenten verwenden FIPA ACL *messages* (siehe Kapitel 3.1) zur Kommunikation untereinander. Die Nachrichtenarten, die von den beiden Agententypen (*Manager-Agent* und *Creature-Agent*) verwendet werden, sind in Abbildung 25 dargestellt. Die Richtung des Pfeils entspricht der Senderichtung, wobei ein Pfeil in beide Richtungen bedeutet, dass beide Agenten den Nachrichtentyp sowohl senden als auch empfangen. Die *killMessage*, die von einem *Creature-Agent* an einen anderen geschickt wird, beinhaltet die Mitteilung, dass ein Tier von einem anderen Tier getötet wurde. *Request_eating* bezeichnet den Nachrichtentransfer zur Koordinierung der Nahrungsaufnahme. Der *Manager-Agent* weist hierbei eine Anfrage zurück, wenn die Nahrungsquelle verbraucht ist. Bei *inform_vision* handelt es sich um die Information des *Manager-Agents* an einen *Creature-Agent* über die Objekte, die sich im Sichtbereich des Tieres befinden. Dieser Informationsfluss wird bei jedem Simulationsschritt vom *Creature*-Objekt initiiert und endet in der Übermittlung eines Java-Objekts - das den Sichtbereich beschreibt - an den *Creature-Agent* durch die *inform_vision*-Nachricht. Dieses Java-Objekt wird durch die Klasse *clsVision* definiert und enthält Listen von Tieren, Nahrungsquellen oder allgemeinen Elementen und zu jedem Listeneintrag die Entfernung zur Position des Tieres. Die Struktur der Elemente wird in Kapitel 4.5.1 behandelt.

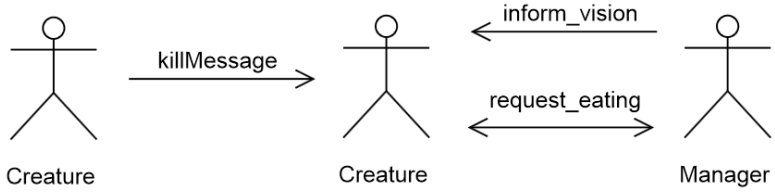

Abbildung 25 - Nachrichtentransfer der Agenten

Eine Erweiterung der Kommunikation zwischen den Agenten erfordert lediglich das Hinzufügen von weiteren *message events* in den ADFs und *plans*, die auf die *events* reagieren beziehungsweise diese auslösen.

In der BDI-Fauna kommen INFORM und REQUEST als *performative* (siehe [11] für eine vollständige Spezifikation des Parameters) zum Einsatz. Bei *inform_vision* und *killMessage* handelt es sich

um Nachrichten vom Typ INFORM und bei *request_eating* um einen REQUEST. Die Konvertierung des Nachrichteninhalts erfolgt über vordefinierte Java-Bean-Konverter, die XML Strings erzeugen. Abgesehen von *inform_vision* wird der Nachrichteninhalt als String übermittelt. Bei *inform_vision* wird zusätzlich definiert, dass es sich bei der Nachricht um ein Objekt der Klasse *clsVision* handelt, wodurch bei der Verarbeitung der Nachricht beim Empfänger die komplette Funktionalität dieser Klasse genutzt werden kann um den Inhalt zu analysieren.

4.5 Aufbau der Simulationsumgebung

Die Simulationsumgebung des MASON-*Frameworks* wird in einem *JFrame* eingebettet und besteht aus einem *ContinuousPortrayal2D*, dessen *field* ein *Continuous2D*-Objekt ist. Die Simulationsfläche ist also eine zweidimensionale Ebene. Auf dieser Ebene werden die Simulationselemente abgebildet. Die Struktur und die Art der Darstellung der Elemente werden in den folgenden beiden Kapiteln behandelt.

4.5.1 Struktur der Elemente

Abbildung 26 zeigt jenen Ausschnitt des Klassendiagramms, der die Struktur der Simulationselemente abbildet und Klassen, die mit dieser Struktur in Verbindung stehen.

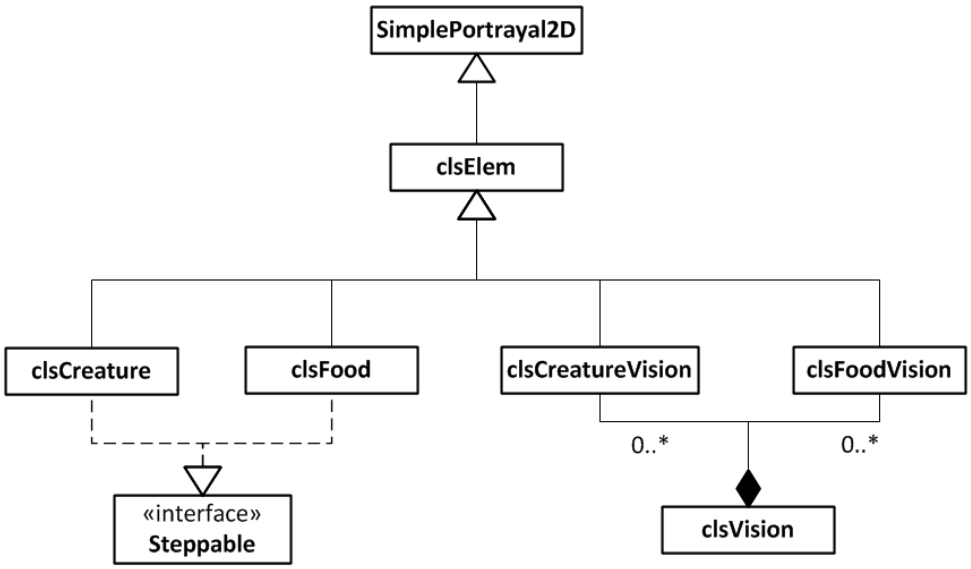

Abbildung 26 - Klassendiagramm der Simulationselemente

Die Simulationselemente werden als *SimplePortrayal2D* in die Simulationsfläche integriert. Dieser *Portrayal*-Typ ist der Basistyp, von dem Spezialisierungen wie zum Beispiel *OvalPortrayal2D*, *RectanglePortrayal2D* oder *HexagonalPortrayal2D* existieren. Er bietet einen *container* um mehrere zusammengehörende Elemente gemeinsam anzuzeigen. Die Darstellung der Simulationselemente wird in Kapitel 4.5.2 näher behandelt.

Als Basisklasse für die Simulationselemente dient die Klasse *clsElem*, die von *SimplePortrayal2D* erbt. Sie dient der Strukturierung der Elemente und enthält lediglich den Identifikator als gemeinsame Grundlage aller Simulationselemente. Es existieren zwei Spezialisierungen von *clsElem*, nämlich *clsCreature* und *clsFood*, die Tiere und Nahrungsquellen beschreiben. Diese beiden Klassen implementieren das Interface *Steppable* um vom MASON-*Framework* als Agent behandelt zu werden. Die Methode *step* des Interfaces beschreibt die Aktionen, die bei jedem Simulationsschritt durchgeführt werden.

Außerdem gibt es noch zwei Spezialisierungen, die Objekte im Sichtbereich eines Tieres beschreiben. Diese Objekte können einerseits Tiere sein (*clsCreatureVision*) oder andererseits Nahrungsquellen (*clsFoodVision*). Die Objekte, die für den Sichtbereich verwendet werden, enthalten lediglich grundlegende Informationen über ein Tier oder eine Nahrungsquelle. Dadurch wird die zu übermittelnde Datenmenge zwischen dem *Manager-Agent* und den *Creature-Agents* klein gehalten. Außerdem enthalten diese Objekte nur jene Informationen, die von den Tieren wahrgenommen werden können beziehungsweise für die Kommunikation notwendig sind. Zum Beispiel die Position des Objekts, um welches Tier oder welche Nahrungsquelle es sich handelt, oder die Agentenidentifikation, die notwendig ist, wenn ein Tier von einem anderen Tier identifiziert werden muss.

Ein *clsVision*-Objekt enthält beliebig viele Tier- oder Nahrungs-Objekte und enthält dadurch alle Simulationselemente, die von einem Tier zu einem bestimmten Zeitpunkt im Sichtbereich wahrgenommen werden können.

4.5.2 Darstellung der Elemente

Die Elemente werden für den Anwender der Simulation auf einer Simulationsfläche dargestellt. So wird das Verfolgen des Simulationsablaufs einfacher und das Verhalten der Tiere kann besser überprüft werden. Die Darstellung der Elemente erfolgt durch Bilder, die entsprechend der jeweils aktuellen Position der Agenten beziehungsweise der Nahrungsquelle auf der Simulationsfläche platziert werden. Es wurden Bilder gewählt, die das darzustellende Element möglichst gut repräsentieren, wobei allerdings auf Detailgenauigkeit oder Realitätsnähe nur bedingt Rücksicht genommen werden konnte. Das Hauptaugenmerk dieser Arbeit liegt in der Implementierung des Verhaltens. Die Visualisierung vereinfacht einzig die Überprüfbarkeit des Verhaltens.

In Abbildung 27 werden die verwendeten Bilder gezeigt. Bild (a) repräsentiert einen Wolf und (b) einen Hasen. Für die Verwendung von weiteren Tieren bedarf es nur weiteren entsprechenden Bildern und zusätzlichen *Properties*-Dateien. Das gleiche gilt für die statischen Nahrungsquellen, wobei die in dieser Arbeit verwendete Karotte in Abbildung 27 Bild (c) abgebildet ist. Für die Darstellung von toten Tieren wurde eine um 90 Grad gedrehte Variante der Bilder der Tiere verwendet.

Diese Bilder sind in Bild (d), das einen toten Wolf darstellt, und (e), das einen toten Hasen zeigt, ersichtlich.

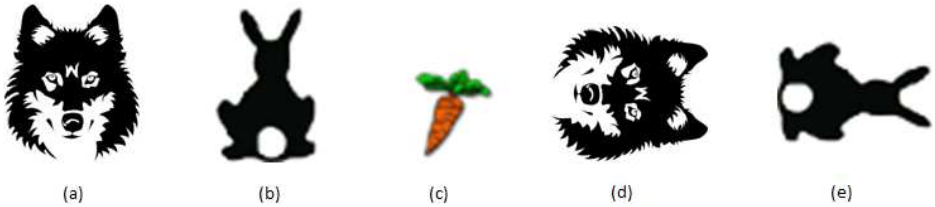

Abbildung 27 - Elemente der Simulation

Abgesehen von der aktuellen Position eines Tieres wird durch Balken links und rechts des Tier-Bildes, seine aktuelle Energie und Tatkraft dargestellt. In Abbildung 28 ist der Energiebalken eines Wolfs mit unterschiedlicher Höhe und Farbe auf der linken Seite des Tieres dargestellt. Der Balken repräsentiert einen prozentuellen Wert der maximal möglichen Energie. Die aktuelle Energie eines Tieres gibt Aufschluss darüber, wie lange das Tier ohne weitere Nahrungsaufnahme noch überleben wird, beziehungsweise wie lange es noch Nahrung zu sich nehmen kann, bevor es komplett satt ist. Der maximal mögliche Wert wird aus der *Properties*-Datei des jeweiligen Tieres übernommen und entspricht 100 Prozent. Dieser Wert, der in Abbildung 28 in Bild (a) ersichtlich ist, wird durch den größtmöglichen Balken in hellgrün dargestellt. In Bild (b) kann man erkennen, dass die verfügbare Energie des Tieres abgenommen hat, da der Balken nicht mehr die volle Höhe hat. Außerdem wechselt die Farbe des Balkens je nach Höhe von hellgrün zu rot. Ein niedriger Balken in roter Farbe ist in Bild (c) dargestellt. Dies bedeutet, dass der Wolf nur noch sehr wenig Energie hat und ohne die Aufnahme zusätzlicher Nahrung bald sterben wird.

Abbildung 28 - Anzeige der Energie

Das Schema der Darstellung der Energie eines Tieres wird auch für die Anzeige der Tatkraft verwendet. Die aktuelle Tatkraft bestimmt, wie lange das Tier noch Aktionen tätigen kann, bevor es schlafen muss beziehungsweise wie lange es noch schlafen muss bis es vollständig ausgeruht ist. In Abbildung 29 werden drei unterschiedliche Werte der Tatkraft eines Wolfs angezeigt. Bild (a) zeigt

einen Wolf, der vollständig ausgeruht ist. Dementsprechend hat der Balken die größtmögliche Höhe und ist blau. In Bild (b) bleibt nur noch etwa die Hälfte der Tatkraft übrig und in Bild (c) steht eine Ruhepause kurz bevor. Der Balken ist sehr klein und die Farbe des Balken hat sich von blau in rot verändert.

Abbildung 29 - Anzeige der Tatkraft

Die zusätzlichen Informationen über ein Tier, die über die seitlichen Balken gewonnen werden können, bringen einen Einblick in den aktuellen Zustand eines Tieres.

Um den Betrachter der Simulation auch die Aktionsmöglichkeiten und das aktuell verfolgte Ziel eines Tieres zu veranschaulichen, wird der Sichtbereich eines Tieres als Kreis um die aktuelle Position dargestellt und das aktuelle Ziel mit einem Kreuz markiert. Diese Visualisierung ist in Abbildung 30 am Beispiel eines Hasen mit seinem aktuellen Ziel, das etwas links und unterhalb seiner aktuellen Position liegt, abgebildet.

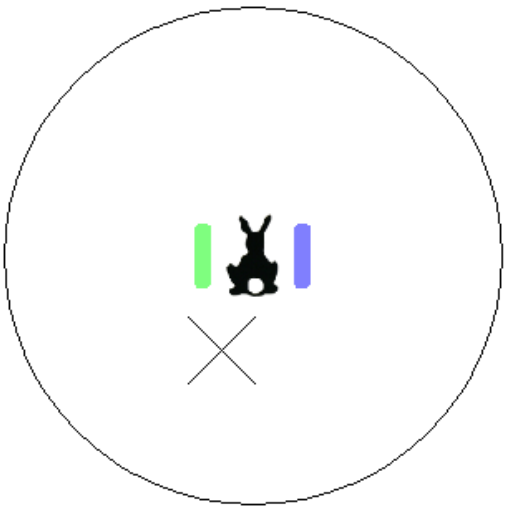

Abbildung 30 - Sichtbereich und aktuelles Ziel eines Tieres

5. Ergebnisse

Das Ergebnis der vorliegenden Arbeit ist eine *Artificial Life Fauna*, BDI-Fauna genannt, deren Agenten auf grundlegende Bedürfnisse der Tiere reagieren. Diese Fauna wurde mit Hilfe des BDI-*Frameworks* JADEX implementiert und mit dem Simulator MASON verknüpft. Dadurch bietet die BDI-Fauna eine Simulationsumgebung, die mit der ARSIN-*World* kombiniert werden kann.

Wenn die ARS-Agenten mit den Tieren der BDI-Fauna konfrontiert werden, können weitere Verhaltensweisen bei den ARS-Agenten beobachtet werden, da sie dann sowohl in die Rolle eines Jägers als auch in die der Beute versetzt werden können. Für die Anpassung der Eigenschaften der Agenten der BDI-Fauna stehen *Properties*-Dateien zur Verfügung um die Testmöglichkeiten variabler zu gestalten. Außerdem können durch Vervielfältigung und Anpassung der *Properties*-Dateien neue Tierarten erstellt werden. Das Verhalten der Tiere kann in strukturierten XML-Dateien (*Extensible Markup Language*) adaptiert und gegebenenfalls erweitert werden.

Die Simulationsergebnisse in diesem Kapitel basieren auf einer Testkonfiguration, die sich nach einigen Simulationen zur Abstimmung ergeben hat. Das Ziel dieses Abstimmungsprozesses war es, ein ausgewogenes Kräfteverhältnis zwischen Hasen und Wölfen herzustellen. Ein Wolf soll weder zu leicht einen Hasen fangen noch soll ein Hase zu mühelos vor einem Wolf fliehen können.

Damit das Verhalten der Tiere nachvollziehbar ist und deren Bewegungsmuster nicht nur zufällig oder unrealistisch erscheinen, ist auch eine Abstimmung der Eigenschaften aufeinander notwendig. Damit ist zum Beispiel das Verhältnis der Bewegungsgeschwindigkeiten von unterschiedlichen Tieren gemeint, aber ebenso die Geschwindigkeit der Nahrungsaufnahme im Vergleich zum Energieverbrauch eines einzelnen Tieres. Zusätzliche Eigenschaften wie zum Beispiel die Nahrungsvorlieben unterscheiden sich natürlich auch und müssen festgelegt werden.

Hierzu mussten aus den in Kapitel 4.2 angeführten Eigenschaften die in Tabelle 7 ersichtlichen Werte bei beiden Tierarten angepasst und variiert werden. Die Vergabe der Werte erfolgte hierbei nicht auf Basis von naturwissenschaftlichen Erkenntnissen, sondern aufgrund von persönlichen Einschätzungen. Diese Einschätzungen dienen für weitere Testsimulationen als Ausgangsbasis und ermöglichen dadurch einen schnelleren Einstieg für die Adaptierung der Eigenschaften.

In Tabelle 7 ist ersichtlich, dass sich die Wölfe schneller fortbewegen können als die Hasen. Dabei verbrauchen sie allerdings auch mehr Nahrung. Zusätzlich verfügen sie über ein größeres Sichtfeld und können dadurch Hasen bereits jagen, wenn diese sie noch nicht wahrnehmen können. Die Nahrungsvorlieben verdeutlichen, dass Hasen Pflanzenfresser sind und die Wölfe Fleisch fressen. In dieser Testkonfiguration erhalten die Wölfe einen größeren Magen als die Hasen und können da-

durch mehr Nahrung auf einmal aufnehmen. Bei der Nahrungsaufnahme sind die Wölfe außerdem schneller und bieten als totes Tier anderen Tieren mehr Nahrung als die Hasen. Die Schlafpausen treten bei Hasen öfter auf, dauern aber aufgrund der schnelleren Erholungsrate kürzer als jene der Wölfe. Bei der Angriffsstärke handelt es sich in dieser Situation einzig um einen Indikator, dass Wölfe stärker sind als Hasen, die bei dieser Wertbelegung keine Angriffe tätigen.

Eine detaillierte Beschreibung der Bedeutung der einzelnen Eigenschaften ist in Kapitel 4.2 ersichtlich.

Tabelle 7 - Werte der Testkonfiguration

Eigenschaft	Werte der Wölfe	Werte der Hasen
speed_running	2	1,5
vision_range	200	150
food_preference_flesh	1	0
food_preference_plant	0	1
stomach_size	300	200
speed_eating	1,35	1,2
food_req_idle	0,0025	0,0025
food_req_running	0,25	0,2
need_sleep	200	200
need_sleep_recovery	0,8	1
need_sleep_consumption	0,25	0,35
food_value (dead)	200	100
attack_power	10	0

Der Start einer Testsimulation erfolgt über das JADEX *Control Center* (siehe Kapitel 3.4). Die Testkonfiguration und die Art und Anzahl der Tiere, die zu Beginn der Simulation in der Simulations-

umgebung existieren, werden geladen indem der *Manager-Agent* gestartet wird. Dieser übernimmt bei der Initialisierung den Aufbau der Simulationsumgebung und startet MASON.

Der Ablauf der Simulation wird mit der Konsole des MASON-*Frameworks* gesteuert. Die Oberfläche dieser Konsole ist in Abbildung 31 dargestellt.

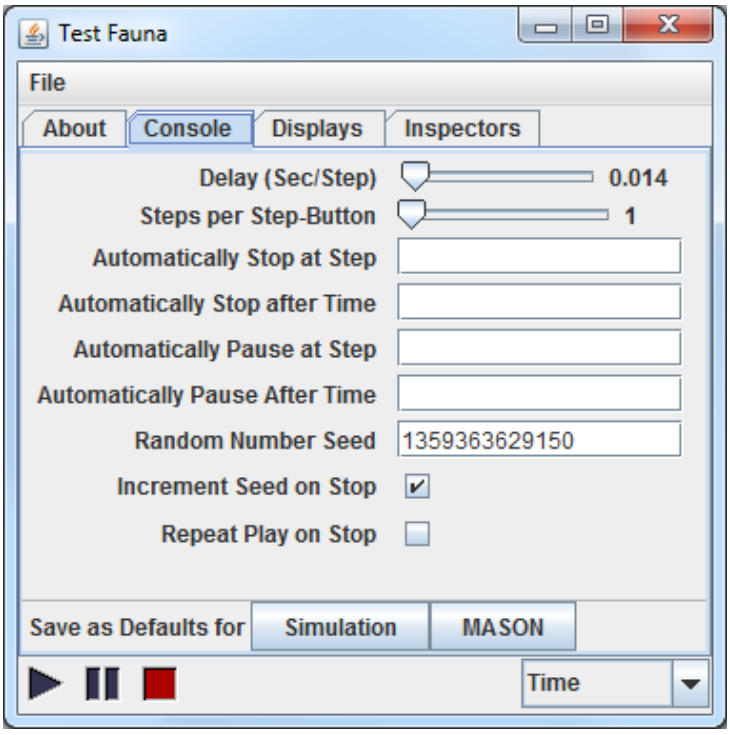

Abbildung 31 - MASON Simulationskonsole

Die wichtigsten Elemente der Konsole sind die drei *Buttons* im linken unteren Eck. Diese dienen zum Starten, Pausieren und Stoppen der Simulation. Abgesehen davon bietet die Konsole die Möglichkeit eine Pause zwischen zwei Simulationsschritten einzufügen und dadurch die Simulationsgeschwindigkeit zu regulieren. Eine weitere Möglichkeit diese Geschwindigkeit zu beeinflussen, ist der Regler für die *Steps* per *Step-Button*. Hierbei kann die Anzahl der Simulationsschritte bestimmt werden, die bei jedem Druck auf den *Step-Button* ausgeführt werden. Der *Step-Button* kommt dann zum Einsatz, wenn die Simulation pausiert ist und statt dem *Play*-Button im linken unteren Eck der Konsole der *Button* für Einzelschritte eingeblendet wird. Damit kann die Simulation um die gewählt Anzahl an Simulationsschritten mit einem Druck auf den *Button* weiterberechnet werden. Zusätzlich ist es möglich die Simulation nach einer bestimmten Simulationszeit oder einer bestimmten Anzahl an Simulationsschritten zu pausieren oder zu stoppen.

Nachdem die Simulation gestartet wurde, werden die Tiere und Nahrungsquellen in der Simulationsumgebung platziert und die Agenten werden gestartet. In Abbildung 32 ist eine Simulationsoberfläche ersichtlich, die zwei Hasen, einen Wolf und zwei Karotten enthält. Die Anzahl und der Typ

der Kreaturen und Nahrungsquellen wird aus dem *Manager-Agent*-ADF ausgelesen (siehe Kapitel 4.4.1). Die Eigenschaften der Tiere und Nahrungsquellen werden aus den jeweiligen *Properties*-Dateien entnommen (siehe Kapitel 4.2).

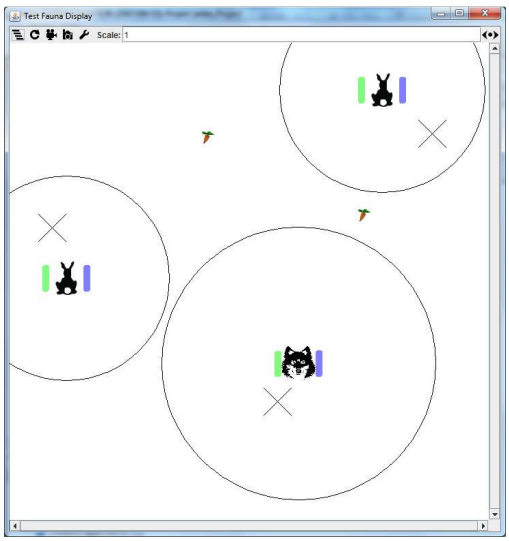

Abbildung 32 - Zufälliges Erkunden bei allen Tieren

So lange ein Tier kein Ziel im Sichtbereich hat, wählt es ein zufälliges Ziel um die Umgebung zu erkunden. Eine Situation, in der alle drei Tiere ihre Umgebung erkunden ist in Abbildung 32 zu sehen. Der Kreis um jedes Tier veranschaulicht den Sichtbereich und das Kreuz markiert die Position ihres aktuellen Ziels.

Neben den Bildern, welche die Tiere darstellen, informieren zwei vertikale Balken über den aktuellen Zustand des Tieres. Der linke Balken stellt die aktuelle Energie des Tieres dar. Verringert sich die Energie, wird der Balken kleiner, bis er gänzlich verschwindet, sobald die Energie komplett verbraucht ist. Außerdem verändert sich die Farbe des Balkens von hellgrün zu rot, wenn das Energielevel von hoch nach niedrig sinkt. Im umgekehrten Fall, also wenn das Tier Nahrung zu sich nimmt, wächst der Balken und die Farbe wechselt von rot zu hellgrün.

Der rechte Balken neben einem Tier gibt Aufschluss darüber, wie groß die aktuelle Müdigkeit des Tieres ist. Hierbei gilt: je größer der Balken ist, desto länger kann das Tier noch ohne Schlaf auskommen. Wenn der Balken kleiner wird, wechselt die Farbe des Balkens von blau zu rot. Im umgekehrten Fall, wechselt die Farbe von rot zu blau. Das ist dann der Fall, wenn das Tier schläft und sich erholt.

In Abbildung 33 kann man erkennen, dass der Wolf einen Hasen entdeckt hat, da das Kreuz, das sein aktuelles Ziel markiert, direkt auf der Position des Hasen platziert ist. Der Wolf hat eine größere Sichtweite als der Hase. Er sieht ihn also schon, der Hase dagegen hat noch nicht bemerkt, dass ein Wolf in der Nähe ist, da sich dieser noch außerhalb seines Sichtbereichs befindet. Diese Tatsache kann über die Eigenschaften in den *Properties*-Dateien der Hasen und Wölfe geändert werden, wo-

durch auch die umgekehrte Situation in einer anderen Simulation möglich wäre. Nämlich, dass der Hase den Wolf bereits sieht, bevor dieser den Hasen sieht.

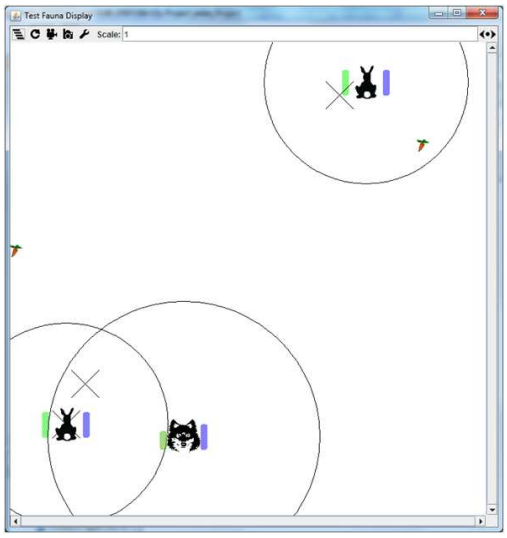

Abbildung 33 - Ein Wolf jagt einen Hasen

Nachdem ein Wolf einen Hasen entdeckt hat, jagt er ihn bis er ihn gefangen hat oder ein anderes Ziel während der Jagd wichtiger wird. Für die Jagd ist die Bewegungsgeschwindigkeit der Tier mitentscheidend für den Ausgang der Jagd. In der Simulation in Abbildung 34 ist der Wolf schneller als der Hase und wird ihn daher bald einholen. Der Hase hat den Wolf bemerkt und versucht in jene Richtung zu fliehen, die den größtmöglichen Abstand zum Wolf herstellt. Das Ziel des Hasen, das durch das Kreuz markiert ist, liegt dementsprechend in der entgegengesetzten Richtung zum Wolf.

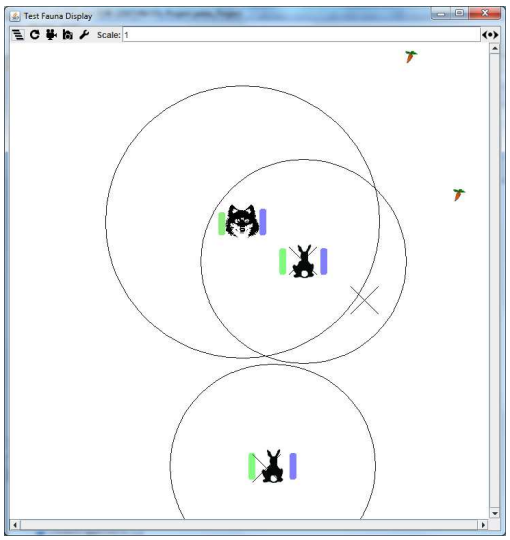

Abbildung 34 - Der Wolf jagt den Hasen, der Hase flieht

In Abbildung 35 hat der Wolf den Hasen bereits erreicht und ihn getötet. Da er aber nicht sehr hungrig war, hat er den Hasen nicht vollständig gefressen. Beim Vergleich des grünen Balkens des Wolfs in Abbildung 34 und Abbildung 35 ist nur ein geringer Unterschied erkennbar.

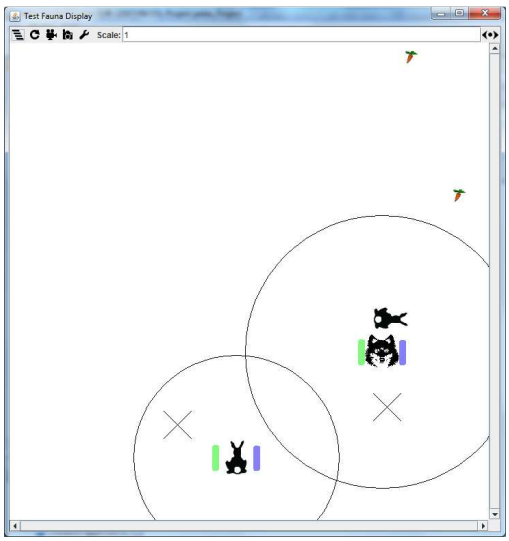

Abbildung 35 - Die Jagd ist zu Ende

Der Wolf hat also den Hasen gejagt und getötet, obwohl er nicht sehr hungrig war und hat dementsprechend auch nicht allzu viel gefressen. Über die c*reation conditions* des *Creature-Agents* (siehe Kapitel 4.4.2) kann dieses Verhalten beeinflusst werden. In der Konfiguration dieser Simulation ist es einem Tier wichtiger ein mögliches Opfer gleich zu töten, als darauf zu warten bis der Hunger groß wird und das Opfer vielleicht nicht mehr in Reichweite ist.

Für die Aufnahme von Nahrung aus statischen Nahrungsquellen wie zum Beispiel Karotten, gilt dieses Verhalten nicht. Es kann also passieren, dass ein Hase eine Karotte sieht und trotzdem nicht frisst, wenn er derzeit nicht ausreichend hungrig ist. Eine solche Situation ist in Abbildung 36 abgebildet. Der Hase sieht eine Karotte in seinem Sichtbereich und entscheidet sich trotzdem nicht dazu sich dorthin zu bewegen und die Karotte zu fressen. Das Kreuz in seinem Sichtbereich macht deutlich, dass er in diesem Moment seine Umgebung erkundet ohne ein spezielles Ziel zu verfolgen.

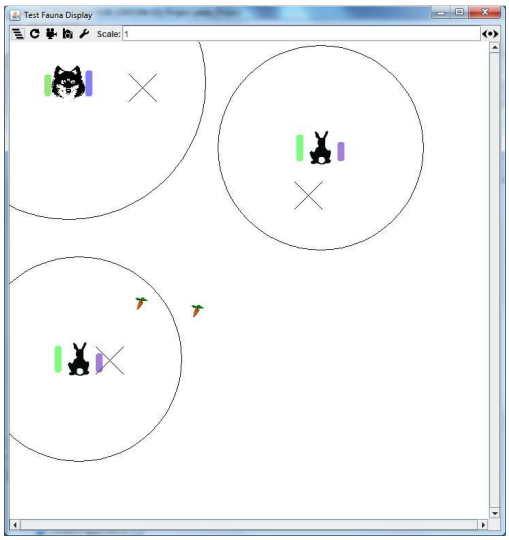

Abbildung 36 - Ein Hase ohne Hunger

In Abbildung 37 zeichnet sich ab, dass der noch lebende Hase bald schlafen muss. Während die Tiere schlafen, erholen sie sich, sind aber trotzdem aufmerksam, ob ihnen eine Gefahr droht. Falls sie sich in Gefahr befinden, unterbrechen sie den Schlaf und versuchen zu fliehen. Das geht soweit, dass sie auch dann noch fliehen können, wenn sie eigentlich zu müde sind um sich zu bewegen. Die bevorstehende Gefahr überdeckt also die Müdigkeit.

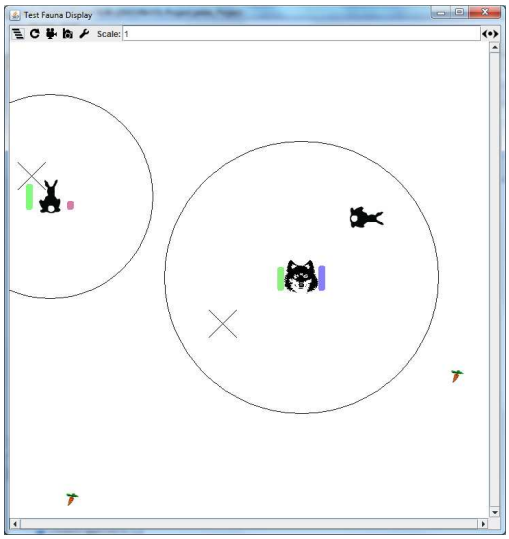

Abbildung 37 - Ein Hase kurz vor einer Ruhephase

Durch zu große Müdigkeit können die Tiere allerdings nicht sterben. Nur wenn sie von einem anderen Tier getötet werden oder wenn sie zu lange keine Nahrung zu sich nehmen, sterben sie. Diese

beiden Fälle sind in Abbildung 38 ersichtlich. Der Hase wurde bereits vom Wolf getötet und noch nicht komplett aufgefressen. Für den Wolf wäre es überlebenswichtig einen Tierkadaver in Reichweite zu haben, da er fast keine Energie mehr hat. Das ist anhand des sehr kleinen roten Balkens links vom Bild des Wolfs erkennbar.

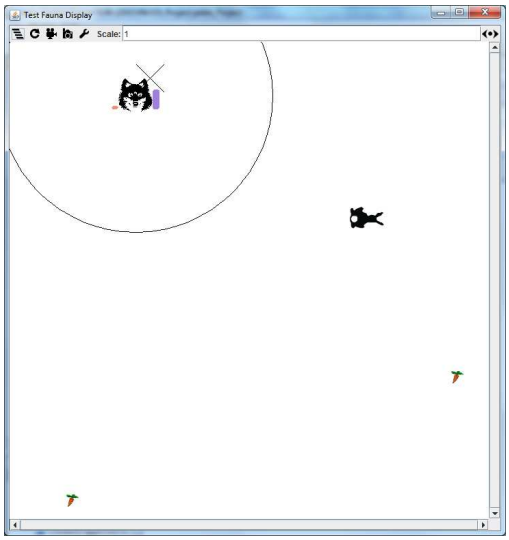

Abbildung 38 - Ein toter Hase und ein sterbender Wolf

Je nach Konfiguration der *Properties*-Dateien der Tiere verbrauchen die Tiere unterschiedlich viel Energie, während sie Aktionen durchführen. Bei einem niedrigen Wert für den Energieverbrauch könnte der Wolf noch etliche Male die gesamte Simulationsumgebung absuchen um Nahrung zu finden. Der Wert muss dafür nur entsprechend klein gewählt werden. Ebenso spielt es eine große Rolle, welchen Wert tote Tiere zur Nahrungsaufnahme haben. Je kleiner ihr Nahrungswert, desto mehr Tiere müsste ein Wolf finden und töten um selbst zu überleben. Das gleiche gilt für die Nahrungsquelle der Hasen. Nur wenn ausreichend viele oder ausreichend nahrhafte Karotten in der Simulationsumgebung vorhanden sind, können die Hasen überleben.

Durch die Eigenschaften in den *Properties*-Dateien der Tiere (siehe Kapitel 4.2) können etliche Parameter der Simulation verändert werden, die entscheidend zum Ablauf der Simulation beitragen und deren Ausgang bestimmen. Ebenso kann das Verhalten der Tiere im ADF des *Creature-Agents* (siehe Kapitel 4.4.2) je nach Wunsch angepasst werden und dadurch eine größere Bandbreite an möglichen Situationen geschaffen werden. Außerdem kann die Startkonfiguration der Simulation im ADF des *Manager-Agents* verändert werden, wodurch zum Beispiel die Anzahl der Tiere und deren Art gewählt werden kann.

6. Zusammenfassung

Die folgenden Absätze liefern eine Zusammenfassung der vorliegenden Arbeit. Daran anschließend wird ein Ausblick auf Weiterentwicklungs- und Adaptionsmöglichkeiten gegeben.

Um eine *Artificial Life* (AL) Simulation zu testen, können Testmethoden der Softwareentwicklung verwendet werden. Diese beschränken sich aber im Allgemeinen auf den verwendeten Programmcode um die AL-Simulation zu implementieren. Für Tests, die einen Einblick darüber geben, wie glaubwürdig ein Agent in einer Simulation agiert, sollten Menschen dessen Verhalten beurteilen. Bei *believable agents* wird diese Beurteilung von Agenten behandelt. Hierbei ist ein Agent umso besser, je menschenähnlicher er sich verhält.

Im ARS-Projekt (*Artificial Recognition System*-Projekt) werden Agenten entwickelt, deren Informationsverarbeitung der des menschlichen mentalen Apparats nachempfunden ist. Dafür wird das Strukturmodell der Psyche von Freud verwendet. Dieses beinhaltet drei Instanzen der Psyche (Ich, Es, Über-Ich), die jeweils unterschiedliche Aufgaben haben. Durch den psychoanalytischen Ansatz werden menschenähnliche Verhaltensweisen für die Agenten entwickelt. Um dieses Verhalten zu überprüfen wurde die ARSIN-*World* entworfen, die eine Simulationsumgebung für die Agenten in einem MASON-Simulator bereitstellt. In dieser Simulationsumgebung werden die Agenten getestet, indem ihr Verhalten von menschlichen Beobachtern überprüft wird. Um die ARSIN-*World* so komplex zu gestalten, dass möglichst viele Aspekte der ARS-Architektur getestet werden können, wurden Hindernisse, Objekte und andere Agenten in die Umgebung integriert.

Zur Erweiterung der ARSIN-*World* wurde in der vorliegenden Arbeit eine künstliche Fauna (BDI-Fauna) aus Agenten, die jeweils ein Tier repräsentieren, mit Hilfe des BDI-*Frameworks* JADEX entwickelt. Diese Tiere agieren in einer Simulationsumgebung, die den Simulator MASON verwendet, und können dank der gemeinsamen Simulationsplattform für zusätzliche Testszenarien mit den ARS-Agenten verwendet werden. In der BDI-Fauna wurden Wölfe und Hasen als Repräsentanten der Fauna ausgewählt, da sie einen Jäger und eine Beute darstellen. Durch die Kombination von Jägern und Beute können viele Testszenarien entworfen werden. Die Fähigkeiten der Tiere beschränken sich auf grundlegende Verhaltensweisen: erkunden, schlafen, fressen, jagen, töten, flüchten.

Um eine variable Simulationsumgebung für Testszenarien zur Verfügung zu haben, wurde bei der Entwicklung der BDI-Fauna darauf geachtet die Eigenschaften der Tiere schnell und einfach ändern zu können. Die Verwendung der BDI-Architektur ermöglicht außerdem eine übersichtliche Bearbeitung des Verhaltens der Tiere.

Anhand der erweiterten Möglichkeiten Tests mit Hilfe der BDI-Fauna zu erstellen kann auch komplexeres Verhalten der *believable agents* überprüft und gegebenenfalls angepasst werden. In der BDI-Architektur werden für jeden Agenten Informationen über die Umwelt, Bedürfnisse und Absichten verwaltet. In der ARS-Architektur gibt es eine ähnliche Herangehensweise, allerdings wird auf die Einhaltung der psychoanalytischen Erkenntnisse geachtet und ein detaillierteres Modell verwendet. Die BDI-Fauna weist aufgrund der BDI-Architektur und den eingeschränkten funktionellen Anforderungen eine geringere Komplexität als das ARS-Modell auf. Dadurch können Änderungen und Erweiterungen leichter durchgeführt beziehungsweise Testszenarien einfacher geändert werden. Die Verwendung von Agenten der BDI-Fauna statt ARS-Agenten, deren Funktionalität eingeschränkt werden kann um sie als Tiere in der Simulationsumgebung zu verwenden, erleichtert somit die Überprüfung des Verhaltens der *believable agents*.

In einem vorangegangenen Projekt innerhalb des ARS-Projekts wurde eine Fauna mit simplen Verhaltensweisen und einfacher Visualisierung implementiert, wobei allerdings keine Agenten verwendet wurden. Die Implementierung erfolgte statt mit einer BDI-Architektur mit *if-then*-Regeln. Diese Regeln bestimmten in jeder Situation der Simulation, welche Aktion jedes Tier durchführt. Dabei wurde nur der aktuelle Status des Tieres berücksichtigt. Die Regeln definierten jeweils eine Aktion zu einer Menge an Situationen, also zum Beispiel bestimmte Variablenbelegungen, die durchgeführt wird. Die Implementierung erfolgte durch Umsetzung der *if-then*-Beziehungen, die vorab festgelegt wurden. Für eine Adaptierung der Regeln oder eine Erweiterung der Fauna ist diese Implementierung allerdings ungeeigneter als die BDI-Architektur, da die Regeln sehr unstrukturiert direkt im Java-Code abgebildet sind. Ein weiterer Vorteil der BDI-Architektur liegt in der Verwendung der Agenten, da dadurch jedes Tier loser mit dem restlichen Programm - sowohl bei der Implementierung als auch zur Laufzeit - verbunden ist. Diese lose Kopplung unterstützt die Vermeidung von unerwünschten Seiteneffekten, die bei der Integration des Codes für ein Tier in das Hauptprogramm mit größerer Wahrscheinlichkeit auftreten.

Die BDI-Fauna der vorliegenden Arbeit ist eine Weiterentwicklung dieser *if-then*-Implementierung und bietet wiederum klare Anknüpfungspunkte um selbst weiterentwickelt oder mit wenig Aufwand adaptiert zu werden.

Ausblick

Die BDI-Fauna kann beispielsweise über die *Properties*-Dateien einfach adaptiert werden, da dafür nur Werte angepasst werden müssen. Allerdings bietet sie auch einige Erweiterungsmöglichkeiten um weitere Testsituationen kreieren zu können.

- Die Eigenschaften der Tiere könnten insofern erweitert werden, dass sie zusätzliche Bewegungsarten ermöglichen. Dadurch könnten zum Beispiel fliegende oder schwimmende Tiere implementiert werden.
- Abgesehen von der visuellen Aufnahme der Umgebung, könnten die Sinne der Tiere erweitert werden um auf zusätzliche Informationen der Umgebung reagieren zu können.
- Die Tiere könnten untereinander interagieren wodurch weitere Verhaltensweisen gegeben wären.

- Tiere, die miteinander interagieren, könnten kooperieren und dadurch Ziele erreichen, wozu sie alleine nicht in der Lage sind.
- Für eine längere Simulation könnten die Tiere aus ihrem Handeln lernen um zukünftig bessere Aktionen durchzuführen.
- Das Verhalten der Tiere könnte erweitert werden um deren Reproduktion zu ermöglichen. Dadurch würde die Simulationen mehr Dynamik bezüglich der Anzahl der darin vorhandenen Tiere erhalten.
- Unabhängig von Reproduktion könnte die Größe der Simulationsumgebung erweitert werden um eine größere Anzahl an Tieren zu simulieren. Diese Überlegung führt zur Simulation von Tierverbänden, die abgesehen vom Individualverhalten auch eine Gruppendynamik zeigen.
- Für die Steuerung und Beeinflussung der Simulation könnte das GUI erweitert werden. Zum Beispiel könnte ein Anwender über das GUI während der Laufzeit weitere Agenten oder statische Objekte, wie zum Beispiel Nahrungsquellen oder Hindernisse, hinzufügen und platzieren. Ebenso könnte der Anwender Möglichkeiten erhalten die *beliefbase* der Agenten und die Eigenschaften der Objekte zur Laufzeit zu verändern.

Abgesehen von den Erweiterungen der BDI-Fauna, kann diese Simulation auch insofern adaptiert werden, dass sie für weitere Einsatzgebiete neben dem ARS-Projekt Verwendung findet. Die Anknüpfungspunkte hierfür sind einerseits JADEX und andererseits MASON. Die Agenten der BDI-Fauna können mit anderen JADEX-Agenten in gemeinsamen Simulationen verwendet werden. Außerdem können die Agenten in MASON-Simulationen mit anderen Agenten, unabhängig von deren Implementierung, simuliert werden.

Literatur

[AL92] D.H. Ackley, M. Littman; *Interactions Between Learning and Evolution*; In C. G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen, editors, Artificial Life II, Volume X of SFI Studies in the Sciences of Complexity, Addison-Wesley, Redwood City, CA, 1992.

[BDK+04] E Brainin, D. Dietrich, W. Kastner, P. Palensky, and C. Rösener; *Neuro-bionic architecture of automation systems: Obstacles and challenges*; Proceedings of 2004 IEEE AFRICON, 7th Africon conference in Africa, Technology Innovation, 2; S. 1219-1222, 2004.

[BEJ+00] M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez; *The Object Data Standard: ODMG 3.0*; Morgan Kaufmann Publishers Inc., 2000.

[BPL04] L. Braubach, A. Pokahr, and W. Lamersdorf; *Jadex: A Short Overview*; In Net.ObjectDays 2004: AgentExpo, S. 195-207, 2004.

[BPM+04] L. Braubach, A. Pokahr, D. Moldt, und W. Lamersdorf; *Goal Representation for BDI Agent Systems*; R.H. Bordini et al. (Eds.): PROMAS 2004, LNAI 3346, Springer-Verlag Berlin Heidelberg, 2005.

[BPR99] F. Bellifemine, A. Poggi, G. Rimassa; *JADE - A FIPA-compliant agent framework*; Telecom Italia internal technical report. Part of this report has been also published in Proceedings of PAAM'99, London, S. 97-108, 1999.

[Bra87] M. Bratman; *Intention, Plans, and Practical Reason*; Harvard University Press, 1987.

[DBMT09] D. Dietrich, D. Bruckner, B. Müller, A. Tmej; *Psychoanalytical model for automation and robotics*; Proceedings of the 9th IEEE AFRICON, S. 5-7, 2009.

[Deu11a] T. Deutsch; *Human Bionically Inspired Autonomous Agents — The Framework Implementation ARSi11 of the Psychoanalytical Entity Id Applied to Embodied Agents*; PhD thesis, Vienna University of Technology, 2011.

[Deu11b] T. Deutsch, S. Kohlhauser, A. Perner, C. Muchitsch; *Use-Cases For Performance Evaluation of Human-Mind Inspired Control Units*, In Proc. 10th IEEE Region 8 AFRICON, S. 1–6, 2011.

[Fre75] S. Freud; *Das Ich und das Es*; in: Studienausgabe, Bd. III: Psychologie des Unbewußten, Frankfurt am Main: Fischer, 1975.

[Gar70] M. Gardner; *Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life"*; In: Scientific American 223, S. 120-123, 1970.

[Gle04] A. Glende; *Agent design to pass computer games*; Proc. of the 42nd annual ACM Southeast regional conference, S. 414-415, 2004.

[GRS00] G. Görz, C.R. Rollinger, J. Schneeberger; *Handbuch der Künstlichen Intelligenz*; Oldenburg Verlag München Wien, 2000.

[Haw06] J. Hawkins; *Die Zukunft der Intelligenz*, rororoscience, 2006.

[Hin09] P. Hingston; *A Turing Test for Computer Game Bots;* IEEE Transactions on Computational Intelligence and AI In Games, Vol. 1, No. 3, S. 169-186, 2009.

[Hin10] P. Hingston; *A New Design for a Turing Test for Bots;* Computational Intelligence and AI in Games, IEEE Transactions on, S. 345-350, 2010.

[HS02] R. Halavati, S. B. Shouraki; *Zamin: An Artificial Ecosystem*; EurAsia-ICT 2002: Information and Communication Technology; Lecture Notes in Computer Science Volume 2510, S. 1008-1016, 2002.

[HS03] R. Halavati, S. B. Shouraki; *A fuzzy artificial world: Zamin ii*. International Conference on Computational Science, S. 601-609, 2003.

[HSZ+04] R. Halavati, S. B. Shouraki, S. H. Zadeh, P. Ziaie, C. Lucas; *Zamin, an agent based artificial life model*; Proceedings of the Fourth International Conference on Hybrid Intelligent Systems (HIS, 04), S. 160-165, 2004.

[Koh08] S. Kohlhauser; *Requirement analysis for a psychoanalytically inspired agent based social system;* Master's thesis, Technische Universität Wien, Institut für Computertechnik, 2008.

[Lan92] C. G. Langton; *Preface*; In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, Artificial life II, Addison-Wesley, 1992.

[LCPS04] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan; *Mason: A new multi-agent simulation toolkit*; In Proceedings of the 2004 Swarmfest Workshop, S. 2, 2004.

[LR10] S. Lim, B. Reeves; *Computer agents versus avatars: Responses to interactive game characters controlled by a computer or other player*; International Journal of Human-Computer Studies, 68(1-2), S. 57-68, 2010.

[MDA05] V. Mascardi, D. Demergasso, D. Ancona; *Languages for programming BDI-style agents: an overview*; In Proc. of WOA'05. Pitagora Editrice, S. 9-15, 2005.

[MU11] M. Mozgovoy, I. Umarov; *Behavior Capture: Building Believable and Effective AI Agents for Video Games*; International Journal of Arts and Sciences, vol. 4(20), S. 1, 2011.

[NW10] J. Niehaus, P. Weyhrauch; *Towards an Architecture for Collaborative Human/AI Control of Interactive Characters*; In Proceedings of AGS, S. 67-75, 2010.

[PBL03] A. Pokahr, L. Braubach, W. Lamersdorf; *Jadex: Implementing a BDI-Infrastructure for JADE Agents*; EXP – in search of innovation, 3(3), S. 76–85, 2003.

[PBL05a] A. Pokahr, L. Braubach, W. Lamersdorf; *Jadex: A BDI Reasoning Engine*; Chapter of Multi-Agent Programming, Kluwer Book, 2005.

[Pet84] U. H. Peters; *Wörterbuch der Psychiatrie und medizinischen Psychologie;* Urban & Schwarzenberg, München, 1984.

[PG07] C. Pennachin, B. Goertzel; *Contemporary approaches to artificial general intelligence*; In B. Goertzel and C. Pennachin, editors, Artificial General Intelligence, Springer, New York, 2007.

[RN03] S. Russell, P. Norvig; *Artificial Intelligence - A Modern Approach*; Second Edition, Pearson Education International, 2003.

[RN04] S. Russell, P. Norvig; *Künstliche Intelligenz: Ein moderner Ansatz*; Pearson Studium, 2. Auflage, 2004.

[Sch12] S. Schaat;. *Integrated Drive Object Categorization in Cognitive Agents*; Master's thesis, Fakultät für Informatik der Technischen Universität Wien, 2012.

[SS97] P. Schuster, M. Springer-Kremser; *Bausteine der Psychoanalyse; Eine Einführung in die Tiefenpsychologie*; 4., überarb. Auflage; WUV, 1997.

[Thi10] Ch. Thiel; *Konzeptioneller Vergleich der Multiagenten-BDI-Systeme Jason und Jadex am Beispiel eines Massively Multiplayer Online Games;* Bachelor thesis, Fakultät Technik und Informatik, Hochschule für Angewandte Wissenschaften Hamburg, 2010.

[Tod82] M. Toda; *Man, Robot, and Society*; Martinus Nijhoff Publishing, 1982.

[Tur50] A. M. Turing; *Computing machinery and intelligence*. Mind, 59, S. 433-460, 1950.

[UMR12] I. Umarov, M. Mozgovoy, and P. C. Rogers; *Believable and effective AI agents in virtual worlds: Current state and future perspectives*; International Journal of Gaming and Computer-Mediated Simulations, vol. 4, no. 2, S. 37–59, 2012.

[Wei66] J. Weizenbaum; *ELIZA - A Computer Program For the Study of Natural Language Communication Between Man And Machine*; in: Communications of the ACM. New York 9.1966,1., S. 36-45, 1966.

[Wei99] G. Weiss; *Multiagent Systems, A Modern Approach to Distributed Artificial Intelligence*; The MIT Press, 1999.

[WG06] P. Wang, B. Goertzel; *Introduction: Aspects of artificial general intelligence*; Proceeding of the 2007 conference on Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms: Proceedings of the AGI Workshop 2006, S. 1-16, 2006.

[WWH+08] D. Weibel, B. Wissmath, S. Habegger, Y. Steiner, R. Groner; *Playing online games against computer- vs. humancontrolled opponents: Effects on presence, flow, and enjoyment*; Computers in Human Behavior, 24(5), S. 2274 - 2291, 2008.

[Zei10] H. Zeilinger; *Bionically Inspired Information Representation for Embodied Software Agents*; PhD thesis, Vienna University of Technology, 2010.

[ZSH04] S. H. Zadeh, S. B. Shouraki, R. Halavati; *A survey on zamin artificial ecosystem*; WSEAS Transactions on Systems, 3(4): S. 1674-1681, 2004.

Internetreferenzen

[1] Institut für Computertechnologie. Homepage; http://www.ict.tuwien.ac.at; abgerufen im Jänner 2013.

[2] Artificial Recognition System. Homepage; http://ars.ict.tuwien.ac.at; abgerufen im Jänner 2013.

[3] Artificial Recognition System. Model; http://ars.ict.tuwien.ac.at/ars-model/; abgerufen im März 2013.

[4] Game of Life, Statische Objekte; http://www.conwaylife.com/wiki/List_of_common_still_lifes; abgerufen im April 2013.

[5] Game of Life, Oszillierende Objekte; http://www.conwaylife.com/wiki/List_of_common_oscillators; abgerufen im April 2013.

[6] BotPrize, Resultate 2012; http://botprize.org/result.html, abgerufen im Februar 2013.

[7] Foundation for Intelligent Physical Agents. Homepage; http://www.fipa.org; abgerufen im Februar 2013.

[8] Foundation for Intelligent Physical Agents. FIPA Agent Management Specification; http://www.fipa.org/specs/fipa00023/SC00023K.html; abgerufen im Februar 2013.

[9] Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Service Specification; http://www.fipa.org/specs/fipa00067/SC00067F.html; abgerufen im Februar 2013.

[10] Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification; http://www.fipa.org/specs/fipa00061/SC00061G.html; abgerufen im Dezember 2012.

[11] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification; http://www.fipa.org/specs/fipa00037/SC00037J.html; abgerufen im Dezember 2012.

[12] Jadex User Guide, Version 0.96 (2007); http://heanet.dl.sourceforge.net/project/jadex/jadex/0.96/userguide-0.96.pdf; abgerufen im Jänner 2013.

[13] Jadex BDI agent components. Abbildung; http://www.activecomponents.org/bin/download/BDI+User+Guide/03+Agent+Specification/jadexagent.png; abgerufen im Februar 2013.

[14] ISO/IEC 14977: 1996(E) − ISO Standard zu EBNF; http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf; abgerufen im Jänner 2013.

[15] ISO/IEC 9075:1992, Database Language SQL; http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt; abgerufen im Jänner 2013.

[16] MASON Library, George Mason University; http://cs.gmu.edu/~eclab/projects/mason/; abgerufen im Dezember 2012.

[17] Evolutionary Computation Laboratory, George Mason University. Homepage; http://cs.gmu.edu/~eclab/; abgerufen im Jänner 2013.

[18] Center for Social Complexity, George Mason University. Homepage; http://socialcomplexity.gmu.edu/; abgerufen im Jänner 2013.

[19] Java API Documentation: java.util.Properties; http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html; abgerufen im Februar 2013.