Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Christian Thiel

Konzeptioneller Vergleich der Multiagenten-BDI-Systeme Jason und Jadex am Beispiel eines Massively Multiplayer Online Games

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Thiel

Konzeptioneller Vergleich der Multiagenten-BDI-Systeme Jason und Jadex am Beispiel eines Massively Multiplayer Online Games

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

Betreuender Prüfer : Prof. Dr. Michael Neitzke
Zweitgutachter : Prof. Dr. Thomas Thiel-Clemen

Abgegeben am 10. August 2010

Christian Thiel

Thema der Bachelorarbeit

Konzeptioneller Vergleich der Multi-Agenten-BDI-Systeme Jason und Jadex am Beispiel eines Massively Multiplayer Online Games

Stichworte

Multiagentensystem, Künstliche Intelligenz, BDI, Jadex, Jason

Kurzzusammenfassung

Diese Arbeit versucht die beiden Multiagenten-BDI-Frameworks Jadex und Jason sowohl konzeptionell als auch praktisch an einem Beispiel miteinander zu vergleichen, um Unterschiede herauszuarbeiten und zu zeigen, welches Framework sich für welchen Anwendungsfall besser eignet.

Für den Vergleich wird hauptsächlich untersucht, wie getreu die Frameworks die theoretischen Grundlagen von BDI umsetzen, jedoch wird ebenso auf praktische Eigenschaften wie Sicherheit, Performance und Entwicklungsgeschwindigkeit wert gelegt.

Als praktisches Beispiel dient eine bereits für Jadex implementierte Simulation eines MMOGs, welches in dieser Arbeit auf Jason portiert wird.

Christian Thiel

Title of the paper

Conceptual Comparison of Multi-Agent-BDI-Systems Jason and Jadex using MMOG as an Example

Keywords

Multiagentsystem, Artificial Intelligence, BDI, Jadex, Jason

Abstract

This paper will compare the two Multiagent-BDI-Frameworks Jadex and Jason both in a conceptual and a practical way, using an example to demonstrate the differences between the two frameworks and to show in which case which framework should be applied.

The main aspects of the comparison will be how strict the frameworks implement the theoretical basics of BDI, but also more practical aspects like security, performance and development speed will be considered.

A simulation of a MMOG which was already implemented in Jadex will be ported to Jason and thus serve as a practical example.

Inhaltsverzeichnis

1. Einführung

In vielen Bereichen der künstlichen Intelligenz trifft man häufig auf Situationen, in denen verschiedene autonom agierende Objekte auf äußere Einflüsse reagieren müssen. Sei es ein Staubsaug-Roboter, welcher auf den Ladestand seiner Batterie achten, oder eine Spiele-KI, welche auf die Bewegungen des menschlichen Mitspielers reagieren muss.

Diese Arbeit beschäftigt sich nun damit, mit der BDI-Architektur einen der vielen Modellierungsansätze für künstlich-intelligente autonome Objekte zu untersuchen. BDI steht konkret für *Belief-Desire-Intention* und stellt eine Modellierung des Wissens über die Umwelt (Belief), der Ziele, die es gilt zu erreichen (Desire) und der Absichten bzw. Aktionen, mit denen diese Ziele erreicht werden können (Intention), dar. Mit Jason und Jadex stehen für diese BDI-Architektur zwei grundverschiedene Plattformen zur Verfügung, welche in dieser Arbeit daraufhin untersucht werden, wie sie die BDI-Architekturoder Teile dieser umgesetzt haben und welche Plattform für welches Anwendungsgebiet besser geeignet ist.

Für diesen Vergleich sind konkret Punkte wie Implementation der Wissensbasis, der Ziele und Pläne sowie Parallelität und Nebenläufigkeit, Dynamik und Fehlerbehandlung, Kommunikation, Sicherheit, Performance als auch die Einarbeitungszeit, Entwicklungsgeschwindigkeit und mitgelieferte Entwicklungswerkzeuge wichtig. Speziell soll auch betrachtet werden, inwieweit sich die beiden Frameworks für MMOGs[1] eignen könnten, wofür vor allem die schnelle Reaktionsfähigkeit des Agenten und eine starke Verteilbarkeit des gesamten Systems benötigt wird.

1.1. Hunters-and-Deers

Als Referenzanwendung, um die beiden BDI-Systeme Jadex und Jason real miteinander vergleichen zu können, wird auf eine frühere Bachelorarbeit von Carsten Canow [7] zurück gegriffen. In dieser Arbeit hat Herr Canow bereits auf Jadex zurückgegriffen, um in einer einfachen Simulation das Konzept der Potentialfelder aufzuzeigen. In diesem *Hunters-and-Deers* genannten Programm untersuchte er die Nutzung von Potentialfeldern zur Bewegung

[1] Massively Multiplayer Online Games

von Figuren. Die Simulation dreht sich hierbei um eine Menge von Rehen und Jägern, welche sich in einem Wald bewegen. Die Rehe versuchen dabei, den Jägern aus dem Weg zu gehen und sich nebenbei von vorhandenen Grassoden zu ernähren. Analog dazu versuchen die Jäger zur Ernährung die Rehe zu erlegen und zu verzehren. Die Bewegung dabei steuern die erwähnten Potentialfelder, indem Objekte positive und negative Felder ausstrahlen, vergleichbar mit verschieden gepolten Magnetfeldern, welche sich gegenseitig überlagern. Die Objekte bewegen sich dabei immer zur positiven Feldstärke hin.
Die einzelnen bewegbaren Objekte werden dabei durch einzelne Agenten realisiert, welche über den Gameserver *Darkstar* [15] miteinander kommunizieren, wahrnehmen und agieren. Der Gameserver stellt dabei einen *Ground Truth* dar, sodass zu jeder Zeit ein valider Gesamtzustand der Simulation besteht.

Im Rahmen dieser Bachelorarbeit werden die bereits vorhandenen Jadex-Agenten nun auf Jason portiert, damit an einem konkreten Beispiel mit festgelegten Aufgaben und Rahmenbedingungen ein Vergleich zwischen Jadex und Jason veranschaulicht werden kann.

1.2. Aufbau der Arbeit

Im Kapitel 2 wird zunächst beschrieben, was Multiagentensysteme sind, wofür sie genutzt werden können und welche Anforderungen sie an eine Architektur stellen.
Das dritte Kapitel beschäftigt sich dann im Detail mit der bereits erwähnten BDI-Architektur, beschreibt den Grundgedanken und die Funktion ihrer einzelnen Komponenten.
Kapitel 4 stellt mit dem eigentlichen Vergleich der Frameworks den Hauptteil der Arbeit dar. Hier werden sämtliche Komponenten detailliert begutachtet und in verschiedenen Punkten sowohl miteinander als auch mit den theoretischen Grundlagen des BDI-Konzeptes verglichen.

Im Laufe der Arbeit werden wir auf immer mehr Fachbegriffe im Zusammenhang mit BDI stoßen. Da sowohl die theoretische Grundlagenliteratur als auch die programmtechnischen Umsetzungen allesamt in englisch verfasst sind und sich die Begriffe meist nur schwer übersetzen lassen, ohne komplett den Bezug zum Original zu verlieren, werden diese Fachbegriffe in dieser Arbeit in der englischen Fassung verwendet.

2. Multiagentensysteme

Die Erforschung komplexerer Probleme der künstlichen Intelligenz sowie Themen wie das Verhalten einzelner Individuen in größeren Gruppen stellt die Informatik vor neue Probleme wie die Einhaltung psychologischer und sozialer Prinzipien und Kommunikationsmuster innerhalb dieser Gruppen. Sobald die Größe der Gruppen bzw. die Komplexität der Simulation gewisse Grenzen überschreitet, sind solche Probleme nur schwer in zentralisierten, monolithischen Systemen zu lösen.

Der verbreitetste Ansatz, dieses Performanceproblem zu lösen, ist die Verwendung eines Multiagentensystems (MAS). In einem solchen MAS stellen die einzelnen Agenten spezialisierte Individuen einer Gruppe dar, welche gemeinsam versuchen, ein Problem zu lösen.

2.1. Eigenschaften eines MAS

Ein Multiagentensystem muss bestimmte Eigenschaften erfüllen, um auf der einen Seite die psychologischen und soziologischen Eigenschaften einer Gruppe modellieren zu können und auf der anderen Seite die Skalierbarkeit zu erreichen, welche für solch komplexe Systeme erforderlich ist.

2.1.1. Begrenzte Wahrnehmung

Die wichtigste Eigenschaft von Softwareagenten ist die Unabhängigkeit von anderen Agenten. Sowohl Handlung als auch Wissensstand eines Agenten muss sich also vollkommen unabhängig von anderen Agenten entwickeln. Ähnlich wie ein Mensch nur Entscheidungen auf seinem eigenen Wissensstand trifft, weil er nicht einfach in die Köpfe anderer Menschen hineinschauen kann, sind Softwareagenten auch auf das angewiesen, was sie selbst wahrnehmen und auf die Rückschlüsse, die sie daraus ziehen.

Um auf äußere Einflüsse zu reagieren, benötigt ein Agent Schnittstellen zur Außenwelt. Im technischen Umfeld sind dies meist Sensoren wie Kameras, Mikrofone, Abstandssensoren oder ähnliches, im Softwarebereich häufig schlicht die Netzwerkschnittstelle, wo Nachrichten von anderen Agenten empfangen werden.

Das Wissen eines Agenten beschränkt sich also ausschließlich auf das, was er durch seine Sensoren wahrnimmt. Das Wahrgenommene muss nicht der Realität entsprechen und kann auch vom Wahrgenommenen anderer beteiligter Agenten abweichen. Ein beliebtes Beispiel dafür ist eine Ameisenkolonie. Dort ist der Hauptsensor einer nach Nahrung suchenden Arbeiterameise das Fühlerpaar, welches eine Duftspur wahrnimmt. Die Ameise weiß also, dass sie gerade einer Duftspur folgt, sie weiß aber weder, wo ihre Kolleginnen sind, noch, ob am Ende der Duftspur überhaupt noch Nahrung zu finden ist.

2.1.2. Keine globale Kontrolle

In einem Multiagentensystem sind alle Agenten per Definition gleichberechtigt zueinander, d.h. es gibt keine übergeordnete kontrollierende Instanz, welche zu jedem Zeitpunkt verfolgen kann, was welcher Agent weiß und was er gerade tut.

Am Beispiel der Ameisenkolonie kann die Königin ihre Arbeiterameisen nicht fernsteuern, sie weiß nicht einmal, wo ihre Ameisen gerade sind und was diese machen. Die Ameisenkönigin ist in ihrer Kolonie auch nur ein Agent mit begrenzter Wahrnehmung.

In vielen konkreten Multiagentensystemen existiert ein Manager-Agent, der andere Agenten koordiniert. Diese Koordination ist jedoch dann reine Modellierungssache und nicht fest in der technischen Schicht des MAS verankert.

2.1.3. Dezentrales Wissen

So wie eine Gruppe von Individuen in der Realität auch kein kollektives Wissen hat, ist der Wissensbestand eines Multiagentensystems ebenfalls dezentral. Jeder Agent hat seine eigenen Sensoren, mit denen er seine Umwelt wahrnimmt und seine Wissensbasis damit erweitert. Aufgrund der individuellen und so auch unterschiedlichen Wahrnehmung kann es so auch zu Situationen kommen, wo das Wissen des MAS nicht nur dezentral sondern auch inkonsistent ist.

Als Beispiel hierfür dient wieder unsere bekannte Ameisenkolonie. Dort findet eine Ameise nun eine Nahrungsquelle, speichert ab, dass an einer bestimmten Stelle Nahrung zu finden ist und legt eine Duftspur für ihre Kolleginnen. Diese folgen der Duftspur und ernten die Nahrungsquelle ab. Nun hat die erste Ameise intern an dieser Position noch immer eine gefundene Nahrungsquelle abgespeichert, obwohl andere Ameisen wissen, dass dieser Ort bereits abgeerntet ist.

2.1.4. Asynchronität

Wie in jedem verteilten System üblich, erfolgt die Lösung des Problems auf vielen Systemen gleichzeitig. Zusätzlich hat die Berechnung und Kommunikation in einem MAS asynchron zu erfolgen. Sendet ein Agent eine Nachricht an einen anderen bzw. per Broadcast an alle Agenten, wartet er nicht auf die Reaktion des Empfängers, sondern macht direkt mit seiner eigenen Arbeit weiter. Wäre es Agenten möglich, einen blockierenden Zustand zu erreichen, könnten sie nicht mehr nebenbei auf andere Ereignisse reagieren und eine korrekte Simulation wäre nicht mehr möglich.

2.2. Arten von Agenten

In der Softwarewelt gibt es bisher keine eindeutige Definition eines Softwareagenten, oder in welche Arten bzw. Klassen man die verschiedenen Agenten unterteilen kann. Im Allgemeinen kann man jedoch zwischen zwei großen Gruppen von Agenten unterscheiden (vgl. [19] S. 123):

2.2.1. Reaktive Agenten

Reaktive bzw reflexive Agenten besitzen kein spezielles Gedächtnis sondern reagieren mit Aktionen direkt auf äußere Einflüsse.

- **reaktiver Agent** Ein reaktiver Agent besitzt keine Wissensbasis und reagiert nur auf Basis seiner Sensoren mit Aktionen auf seine Umwelt.
- **modellbasierter Agent** Ein modellbasierter Agent ist ein Agent welcher zusätzlich um eine Art Zustandsmodell erweitert wird, welches ein primitives Gedächtnis darstellt.

Solche reaktiven Agenten werden meist mit einer einfachen Tabelle realisiert, welche für jedes Eingangssignal eine entsprechende Aktion referenziert. Bei modellbasierten Agenten ist diese zusätzlich um einen Zustand oder eine Folge von vergangenen Wahrnehmungen erweitert, um das Gedächtnis zu integrieren.

2.2.2. Kognitive Agenten

Kognitive Agenten verwalten in ihrer Wissensbasis ein Modell ihrer Umwelt, wodurch durch die längerfristige Planung von Aktionen ein zielgerichtetes Handeln ermöglicht wird. Typische Vertreter dieser Art von Softwareagenten sind BDI-Agenten, welche mit Hilfe von Plänen vordefinierte Ziele verfolgen.

- **Zielbasierter Agent** Der Agent besitzt ein vorher definiertes Ziel und versucht dieses zu erreichen. Dafür besitzt er Informationen über die Folgen seiner möglichen Pläne und Aktionen und kann so zielgerichtet Pläne auswählen. Da Ziele oft nicht direkt mit einer einzigen Aktion erreicht werden können, kann der Agent planen und verschiedene Aktionen miteinander verknüpfen.
- **nutzenbasierter Agent** Der nutzenbasierte Agent besitzt ebenso wie der zielbasierte Agent eine Zielvorgabe, kann zusätzlich jedoch verschiedene Schritte im voraus schätzen und berechnen, welche der möglichen Pläne den größten Nutzen zu Erreichung seiner Ziele bringen. Dies ist vor allem wichtig, wenn der Agent mehrere Ziele gleichzeitig zu verfolgen hat und entscheiden muss, welches Ziel die höchste Priorität besitzt.

2.3. Agentenumgebungen

In Kapitel 2.1.1 sprachen wir davon, dass Agenten eine begrenzte Wahrnehmung ihrer Umwelt besitzen. Diese Wahrnehmung bzw. eher die Umgebungen, in der sich ein Agent befinden kann, können nach [14, S 40f.] noch weiter klassifiziert werden.

- **Beobachtbarkeit** Eine Agentenumgebung gilt als *vollständig beobachtbar*, wenn der Agent mit seinen Sensoren zu jeder Zeit vollen Zugriff auf die komplette Umwelt hat, wie es z. B. bei einem Schach-Agenten der Fall ist. Dort kennen beide Agenten zu jeder Zeit die Position jeder einzelnen Figur auf dem Spielfeld und müssen sich diese so nicht in einem eigenen Gedächtnis merken. Ist eine Umgebung nicht vollständig beobachtbar, gilt sie als *teilweise beobachtbar*. Dies ist der Fall, wenn die Sensoren des Agenten z.B. unregelmäßigen Störungen unterliegen oder schlicht nicht der gesamte Zustand der Umwelt durch die Sensoren erfasst werden kann, wie es beispielsweise bei Agenten der Fall ist, die durch Kameras in ihrem Sichtbereich begrenzt sind. Solche Agenten müssen eine Art von Gedächtnis führen, um trotzdem einen möglichst genauen Zustand der Umwelt zu haben. Je länger eine Wahrnehmung jedoch her ist, desto unwahrscheinlicher ist sie immer noch aktuell.

- **Determinismus** Wenn sich der Folgezustand einer Umgebung nur aus dem aktuellen Zustand und der vom Agenten ausgeführten Aktion ergibt, spricht man von einer *deterministischen* Umgebung. Agenten in einer deterministischen Welt können mehrere Schritte im Voraus planen, ohne ein Risiko der Fehlplanung einzugehen, da sich die Umwelt nicht unvorhersehbar ändert. Ein Beispiel ist ein Navigationssystem, welches versucht, eine Strecke von Hamburg nach München zu finden. Einen zu Programmstart aktuellen Streckenplan vorausgesetzt, ist es sehr unwahrscheinlich, dass plötzlich Autobahnen verschwinden oder hinzukommen, welche die errechnete Route während der Fahrt fehlschlagen lassen. Ist eine Welt nicht deterministisch, so gilt sie als *stochastisch*. Hier kann der Agent nicht sicher sein, wie der Folgezustand aussieht, weil er nicht alle ändernden Faktoren kennt und muss sich auf Schätzungen und Wahrscheinlichkeiten verlassen.
- **Episodisch vs. sequenziell** Lässt sich der Lebenszyklus eines Agenten in mehrere kleine, atomare Zeitabschnitte teilen, welche untereinander unabhängig sind, spricht man von einer *episodischen* Umgebung, wie es z.B. bei einem Druckagenten der Fall ist, wo die Druckaufträge unabhängig voneinander ablaufen. Im Gegenzug ist eine Umgebung *sequenziell*, wenn sich die einzelnen Aktionen eines Agenten wie bei einem Schachspiel auch auf die fernere Zukunft auswirken können und es wichtig ist, in welcher Reihenfolge die Aktionen ausgeführt werden.
- **Statisch vs. dynamisch** Eine *dynamische* Umgebung, kann ihren Zustand erneut ändern, während der Agent noch versucht auf den letzten Zustand zu reagieren. So ist ein Fußballspiel eine dynamische Umgebung, weil sich die anderen Spieler und vor allem der Ball schon wieder weiter bewegt haben können, während man selbst überlegt, wo man den Ball hinschießen möchte.
- **Diskret vs. stetig** Ist eine Umgebung *diskret*, bedeutet dies, dass die Umgebung nur eine endliche Anzahl verschiedener Zustände und Aktionen kennt, wie es zum Beispiel bei einem Schachspiel der Fall ist. Im Gegenzug ist eine Umgebung *stetig*, wenn ihre Zustandswerte (z.B. die Position eines Fußballspielers), einen stetigen Bereich durchlaufen, ohne sprunghafte Änderungen zuzulassen. Die Position des Balles und der 22 Spieler ist zwar zu jeder Zeit definiert, jedoch ist die Anzahl der möglichen Gesamtzustände der Umgebung praktisch unendlich.

2.4. Kommunikation im MAS

Ein Hauptbestandteil der als verteiltes System agierenden Multiagentensysteme ist die ständige Kommunikation der beteiligten Agenten untereinander. Aus diesem Grund ist ein leistungsfähiges Kommunikationsnetzwerk unabdingbar. Eine gute und effiziente Kommunika-

tionsstruktur ist ebenso erforderlich, da das Wissen der Agenten dezentral organisiert ist und sich die einzelnen Agenten zur Entscheidungsfindung mit anderen Agenten abstimmen müssen oder sich vorher ein ausreichendes Bild der aktuellen Gesamtsituation zu machen haben.

Grundsätzlich kann man zwischen 3 Arten der Kommunikation unterscheiden (vgl. [19] S. 127):

- Die einzelnen Agenten müssen zu jeder Zeit über die verfügbaren Dienste der ausführenden MAS-Plattform informiert sein. Dafür müssen Agent und Plattform Informationen austauschen, wozu auch die Behandlung von Ausnahmen und Fehlern gehört.
- Die in einem Netzwerk kooperierenden Agenten müssen direkt oder indirekt Informationen austauschen, was lokal in einem System oder über das Netzwerk erfolgen kann.
- Für die Mobilität und Plattformunabhängigkeit ist auch die Kommunikation zwischen den einzelnen MAS-Plattformen erforderlich. Damit können auch Agenten verschiedener Plattformen kommunizieren und Agenten von einem System auf ein anderes migriert werden, was die Robustheit des Systems steigert.

Für die Kommunikation der Agenten untereinander gibt es 3 verschiedene Arten:

- **Punkt-zu-Punkt** Die direkte Kommunikation zweier Agenten miteinander wird meist nur benutzt, um auf die Wissensbasis des anderen Agenten zuzugreifen, ist daher in den meisten Fällen eine blockierende Kommunikation.
- **Broadcast** Der Broadcast ist die häufigste Art der Agentenkommunikation. Hierbei wird eine Nachricht, meist eine Mitteilung über eine eigene Statusänderung, an alle beteiligten Agenten geschickt und jeder Agent kann selbst entscheiden, ob die Nachricht von belang ist oder nicht.
- **Signal** Das Signal ist das Gegenstück zur Punkt-zu-Punkt-Kommunikation. Auch hier kommunizieren nur zwei Agenten direkt miteinander, jedoch ist die Kommunikation asynchron und damit nicht blockierend. Signale werden meist benutzt, wenn relevante Agenten zuvor eindeutig bekannt sind und eine Broadcastnachricht zu teuer ist.

3. Die BDI-Architektur

Die meisten Ansätze, ein intelligentes Agentensystem zu modellieren, versuchen sich an der Natur der menschlichen Denkweise zu orientieren. Das Handeln des Menschen ist im Allgemeinen von seinen Zielen bestimmt. Sei es, dass ein Student, einen möglichst guten Abschluss erreichen möchte, ein Projektleiter sein Projekt zum Erfolg bringen möchte oder jemand beliebiges einfach nur seinen Hunger stillen möchte. Um diese Ziele zu erreichen, müssen diese Menschen in irgend einer Art und Weise handeln. Die Schwierigkeit für den Menschen ist nun, welche Aktionen er ausführt, um seine individuellen Ziele zu erreichen. Der Informatikstudent *weiß*, dass er im ersten Semester studiert und er *weiß*, dass die Vorlesung *Grundlagen der Informatik* zum ersten Semester gehört und dass er diese besuchen muss, um den Abschluss zu bekommen. D.h. der Mensch hat neben seinen Zielen auch ein Wissen über seine Umwelt, welches er nutzt, um durch sein gezieltes Handeln seine Ziele zu erreichen oder ihnen zumindest näher zu kommen.

Dieses Konzept spiegelt sich in der BDI-Architektur wider. BDI steht kurz für *Belief* (Glaube/Wissen), *Desire* (Ziele) und *Intention* (Absichten/Pläne).

3.1. Grundlagen von BDI

3.1.1. Das Weltwissen

Jeder BDI-Agent unterhält eine Wissensbasis über alles, was er glaubt, von der Welt zu wissen. Wir haben jedoch bereits festgestellt, dass die Sensoren eines Agenten nicht immer die Wahrheit widerspiegeln, weshalb der Begriff *Beliefbase* eher korrekt mit *Glaubensbasis* übersetzt werden sollte.

In der Wissensbasis muss nicht zwingend nur die Umwelt des Agenten abgebildet und die Signale und Nachrichten seiner Sensoren abgespeichert werden. Intelligente Agenten sind oft auch fähig, aus zwei bekannten Fakten auf einen dritten, neuen Fakt zu schließen, den sie aufgrund einer teilweise beobachtbaren Welt nicht durch die Sensoren wahrnehmen können (z.B. ein Objekt verschwindet aus dem Sichtbereich und der Agent schätzt seine Position). Natürlich sind diese selbst gezogenen Schlüsse und Fakten deutlich vager als das, was die Sensoren wahrnehmen.

3.1.2. Ziele

Die Ziele eines Software-Agenten sind seine Lebensaufgabe, auf welche er all seine Aktionen versucht auszurichten. Ein Ziel kann es sein, sich selbst in einen bestimmten Zustand zu bringen (z.B. bei einer Navigationssoftware, das Ziel zu erreichen), seine Umwelt in einen bestimmten Zustand zu bringen (z.B. in einem Spiel den Gegner zu vernichten) oder einen festgelegten, bereits erreichten Zustand zu erhalten (z.B. bei einem Roboter den Ladezustand des Akkus nicht unter ein bestimmtes Niveau fallen zu lassen). Eine weitere Art von Zielen können auch Endlos-Ziele sein. Ein Service-Agent kann das Ziel haben, Anfragen von anderen Agenten zu beantworten (z.B. ein Webserver oder Fileserver). Der Service-Agent wird seinem Ziel nie näher kommen, geschweige denn es jemals erreichen, denn dann braucht der Service ja nicht mehr angeboten werden.

Ziele müssen nicht immer die gesamte Lebensaufgabe eines Agenten an sich darstellen. Ein Ziel kann auch ein kleiner Schritt (*Subgoal*) auf dem Weg zum großen Ziel sein. Diese Subgoals werden meist zur Laufzeit erzeugt und bearbeitet, während die großen Ziele eines Agenten bereits zu Programmstart bekannt und einkodiert sind und bei Erfüllen der Agent seine Aufgabe erfüllt hat (z.B. “Das Spiel gewinnen”).

3.1.3. Pläne

Die Menge aller Aktionen, die ein Agent seiner Umwelt oder sich selbst gegenüber ausüben kann, bezeichnet man als Pläne. Diese haben den Zweck, den Agenten seinen Zielen näher zu bringen oder diese sogar zu erfüllen. Pläne haben in den meisten Fällen nur einen begrenzten Einsatzbereich oder besitzen einige Vorbedingungen, welche erfüllt sein müssen, um den Plan auszuführen. Innerhalb eines Planes können wiederum Unterpläne ausgeführt werden, sodass hierarchische Strukturen möglich sind, oder neue Ziele definiert werden (so kann ein Plan *Satt werden* das Ziel *Essen kochen* hinzufügen, welches dann wieder weitere Pläne aufruft).

3.1.4. Ereignisse

Ein weiterer Teil jedes Agentensystems sind Ereignisse. Agenten handeln in den meisten Fällen nicht von sich heraus, sondern warten auf Signale ihren Sensoren, auf die sie dann versuchen intelligent zu reagieren. Hier wird zwischen zwei großen Arten von Ereignissen unterschieden. Die erste Art der Ereignisse sind Änderungsereignisse, welche auftreten, wenn ein Sensor eine Veränderung in der Umwelt des Agenten wahrnimmt (z.B. eine Positionsänderung eines bekannten Objektes oder das Auftauchen eines neuen Objektes). Die zweite Gruppe von Ereignissen entstehen durch die Kommunikation mit anderen Agenten.

Es kann passieren, dass einem ein anderer Agent seine Zustandsänderung mitteilt, oder seine Wahrnehmung oder sein Wissen mit dem eigenen Agenten teilen möchte, sodass beide gemeinsam ein umfassenderes Wissen über die Umwelt besitzen. In den meisten Fällen resultiert aus einem Kommunikationsereignis auch ein Änderungsereignis, da sich durch das Kommunizierte eine eigene Zustandsänderung ergibt, auf welche eventuell reagiert werden muss.

3.2. Practical Reasoning

Nehmen wir nun einmal an, ein Agent hat aktuell mehrere Ziele, welche er erreichen möchte. Einmal ist der Agent ein Student und möchte seinen Abschluss erreichen. Dafür hat er bereits herausgefunden, dass er Vorlesungen besuchen muss. Weiterhin muss der Agent noch ein Buch in der Stadtbibliothek abgeben und zu allem Überfluss hat er auch noch Hunger. Der Agent kennt eine Menge von Plänen und Aktionen, welche ihm helfen, alle dieser Ziele zu erreichen. Die interessante Frage ist nun, was der Agent als erstes macht, denn er hat ja vier verschiedene Ziele, die er erreichen will, welche größtenteils in ihrer Erfüllung Konflikte mit anderen Zielen verursachen würden. Ein Agent, der vor dem Herd steht, kann schlecht zur Universität gehen, um die Vorlesung zu besuchen und da sich die Stadtbibliothek am anderen Ende der Stadt befindet kann dieses Ziel ebenfalls nicht parallel erreicht werden: Der Agent muss sich entscheiden, was er zuerst versucht zu erledigen.

3.2.1. Goal Deliberation

Dieser Vorgang, bei dem der Agent auswählt, welche Ziele er aktuell verfolgen möchte, nennt sich *Goal Deliberation*. Hierbei sucht der Agent sich zuerst alle Ziele heraus, welche seiner Meinung nach aus seinem aktuellen Zustand aus erreichbar bzw. erfüllbar sind. Stößt der Agent dabei auf Ziele, welche unmöglich zu erreichen sind, werden diese nicht in die Liste aufgenommen. Ist das Ziel nur im Moment nicht aus eigener Kraft zu erreichen, weil die Erfüllung des Zieles von der Reaktion anderen Agenten abhängt, wird das Ziel nur zurückgestellt. Ist das Ziel jedoch dauerhaft unmöglich zu erreichen oder ist es schlicht nicht mehr sinnvoll, dieses Ziel zu verfolgen, wird das Ziel komplett aus der Liste der zu verfolgenden Ziele gestrichen.

Hat der Agent nun eine Menge an erreichbaren, sinnvollen Zielen, sucht er nach Konflikten und Abhängigkeiten, welche verhindern, dass einige der Ziele parallel erreicht werden können, weil die nötigen Aktionen Ressourcen blockieren (z.B. muss der Agent sich beim Autofahren auf die Straße konzentrieren und kann nebenbei kein Buch lesen) oder die Aktionen

den Agenten in komplementäre Richtungen führen, z.B. wenn der Agent vor hat, gleichzeitig zur Universität und zur am anderen Ende der Stadt gelegenen Bibliothek zu gelangen.

Ein wichtiger Punkt bei der Modellierung von Zielen und Plänen in einem Agentensystem ist es, darauf zu achten, dass möglichst viele Ziele parallel verfolgt werden können, denn nur durch die hohe Parallelität können Agenten ihre Leistung richtig ausspielen. Hierfür ist es oft nötig, die großen Ziele wie “Uniabschluss erreichen” in viele kleine Ziele zu zerteilen, wie “Vorlesung besuchen” oder noch granularer wie “zur Universität fahren”, “Vorlesung besuchen” und “nach Hause fahren”, weil der Agent dann feststellen kann, dass er während der Busfahrt zur Universität noch das Ziel “Buch lesen” verfolgen oder das Ziel “Einkaufen” auf dem Heimweg erreichen kann. Je mehr Ziele der Agent parallel verfolgen kann, ohne dass sich die Pläne, diese Ziele zu erreichen, gegenseitig behindern oder blockieren, desto schneller kann der Agent seine Ziele erreichen und desto dynamischer kann er auf verschiedene äußere Einflüsse reagieren.

Schlussendlich hat der Agent nun eine Menge an Zielen, welche er sich entschieden hat für den Moment zu verfolgen und auch festgestellt, welche dieser Ziele im Konflikt zueinander stehen und welche unabhängig voneinander parallel verfolgt werden können. Diese Ziele werden nun in eine Art Ausführungsplan gebracht, welcher bestimmt, welche Konstellation von Zielen als erstes verfolgt werden soll.

3.2.2. Planen

Hat der Agent entschieden, welchen Zielen er sich zuwenden möchte, folgt mit der Plansuche nun der nächste Schritt. Einem Agenten stehen eine große Menge an Aktionen zur Verfügung, um den eigenen Zustand oder den seiner Umwelt so zu beeinflussen, dass dieser einem Zielzustand entspricht. In den wenigsten Fällen ist dieser Zielzustand jedoch mit nur einer Aktion erreichbar, sodass der Agent sich eine längere Folge von Aktionen überlegen muss, um sein Ziel zu erreichen.

Diese Aktionenfolge, also der Plan, mit dem das Ziel erreicht werden kann, wird von einem *Planer* ermittelt. Dieser Planer hat für die Erstellung des Planes Zugriff auf alle Ziele, die der Agent aktuell versucht, zu erreichen, auf die aktuelle Wissensbasis des Agenten und er kennt alle Aktionen, die der Agent ausführen kann, um die Umwelt zu verändern. Aus dem aktuellen Zustand des Agenten und der Menge aller Aktionen versucht der Planer nun einen Suchgraphen zu erstellen, die Knoten dieses Graphen sind die möglichen Zustände des Agenten, die Kanten stellen die Aktionen dar, die der Agent ausüben kann. Nun kann man sich vorstellen, dass ein Agent nicht nur zwei oder drei Aktionen zur Verfügung hat, sondern durchaus auch 20, 50 oder 100 verschiedene Aktionen, welche alle zu anderen Zuständen führen. Bei solchen Graphen sind uninformierte, primitive Suchverfahren wie die Tiefen- oder

Breitensuche bei weitem zu langsam und teilweise auch zu speicherintensiv. Eine Möglichkeit, die Zahl der möglichen Aktionen zu reduzieren, ist die Definition von Vorbedingungen. Jede Aktion besitzt eine Menge an Bedingungen, welche der aktuelle Zustand erfüllen muss, damit die Aktion möglich ist, z.B. ist es unsinnig zu versuchen das Gaspedal betätigen zu wollen, wenn man selbst noch gar nicht im Auto sitzt.

Durch diese Definition von Vorbedingungen lassen sich die möglichen Aktionen in den meisten Fällen stark einschränken, doch es lässt sich auch dann nicht verhindern, dass der Planer im Suchgraph eine zu große Tiefe erreicht und durch den exponentiellen Anstieg der Knoten je Stufe die vorhandene Hardware an ihre Grenzen treibt. Abhilfe können hier informierte Suchverfahren schaffen. Diese Verfahren besitzen für jeden Knoten eine heuristische Nutzenfunktion, welche abschätzen kann, ob es nicht nur möglich, sondern ob es überhaupt sinnvoll ist, die konkrete Aktion auszuführen und ob diese Aktion den Planer bzw. im Endeffekt den Agenten seinem Zielzustand näher bringt. Solche Suchverfahren, wie zum Beispiel der *AStar-Algorithmus*, untersuchen bei ihrer Suche nur Knoten, welche eine heuristisch geringe Entfernung zum Zielknoten besitzen und lassen damit viele Knoten bzw. Aktionen außen vor, da sie vom Algorithmus für nicht sinnvoll angesehen werden und beschleunigen damit die Suche nach einem Plan um ein Vielfaches. Für weitere Informationen zu Suchverfahren und Planen siehe [14, Kap. 3, 4 und 11]

Die informierten Suchverfahren sind jedoch auch spätestens dann machtlos, wenn die Aktionen einen nicht diskreten Parameterraum besitzen (z.B. bei der Aktion *links(float winkel)* sind durch den Parameter *winkel* praktisch unendlich viele verschiedene Aktionen möglich), denn in einem Suchbaum mit unendlich vielen Kindknoten kann eine Suche auch unendlich lange dauern und eine Modellierung, welche ohne diese nicht-diskreten Aktionen auskommt, ist oft zu komplex und das Ergebnis noch immer zu zeitaufwändig.

Aus diesem Grund vereinfacht BDI das Problem des Planens radikal, indem einfach der größte Teil der Planungsarbeit beim Entwickler belassen wird. Der Softwareentwickler, welcher den Agenten programmiert, definiert von Beginn an eine Menge an fertigen Plänen, welche eine Folge verschiedener Aktionen darstellen. Es ist Aufgabe des Entwicklers, dafür zu sorgen, dass ein Plan in sich konsistent ist und keine unmöglichen Aktionen enthält.
Die Aufgabe des Planers innerhalb des Agenten oder der Agentenumgebung ist es nur noch, aus einer begrenzten Anzahl von Plänen den besten auszuwählen und an den Agenten zurückzugeben. Bei dieser Entscheidung helfen dem Agenten u.a. Vorbedingungen, welche für den kompletten Plan gelten müssen, damit dieser ausführbar wird.

```
1: B ← B0; // B0 ist das Anfangswissen
2: I ← I0; // I0 sind die Anfangsziele
3: while true do
4:    // hole nächste Wahrnehmung ρ via sensors
5:    B ← brf(B, ρ);
6:    D ← options(B, I);
7:    I ← filter(B, D, I);
8:    π ← plan(B, I, Ac);
9:    while not (empty(π) or succeeded(I, B) or impossible(I, B)) do
10:      α ← first element of π;
11:      execute(α);
12:      π ← tailof(π);
13:      // observe environment to get next percept ρ;
14:      B ← brf(B, ρ);
15:      if reconsider(I, B) then
16:         D ← options(B, I);
17:         I ← filter(B, D, I);
18:      end if
19:      if not sound(π, I, B) then
20:         π ← plan(B, I, Ac);
21:      end if
22:   end while
23: end while
```

Abbildung 3.1.: Kontrollschleife eines BDI-Agenten [4, S. 21]

3.3. Beispiel eines BDI-Modells

Wollen wir nun einmal einen Agenten auf Basis von *practical reasoning* implementieren, werden wir ziemlich schnell darauf kommen, dass unser Agent in vier Phasen arbeitet:

- Umwelt beobachten und Wissensbasis aktualisieren
- Überlegen, welche Ziele verfolgt werden sollen
- Einen passenden Plan finden
- Den Plan ausführen

Betrachten wir diese vier jedoch genauer, fällt uns auf, dass dieses Modell sehr starr ist. Einmal auf einen Plan festgelegt, wird dieser auf Teufel komm raus ausgeführt, bis er erfüllt ist

oder komplett fehlschlägt. Bis der Plan komplettiert wurde, werden keine weiteren Wahrnehmungen analysiert. Abhängig davon, wie aufwändig der gewählte Plan ist, kann es für den Agenten sehr von Nachteil sein, da sich die Umwelt eines Agenten jederzeit ändern kann. Es kann sein, dass sich die Umwelt so verändert, dass es unmöglich ist, den Plan zu vollenden oder dass er sich bereits von selbst erfüllt hat. Es können während der Ausführung Ereignisse auftreten, welche es nötig machen, dass der Agent seine Ziele und Absichten komplett überdenkt.

Aus diesem Grund ist das Modell in Abbildung 3.1 um eine weitere innere Schleife erweitert worden. Die Hauptkontrollschleife des Agenten(Zeile 3-23) kennen wir bereits aus unseren vorigen Überlegungen. Zuerst wird die Umwelt untersucht und eine Wahrnehmung empfangen (4). Aus dieser Wahrnehmung und der letzten Wissensbasis wird die aktuelle Wissensbasis bestimmt (5). Aus dieser Wissensbasis werden alle Ziele gewonnen (6), die der Agent verfolgen möchte (wir erinnern uns, einige Ziele werden erst durch Wahrnehmungen von außen getriggert). Diese Menge aller Ziele wird als nächstes auf die Menge aller möglichen und sinnvollen Ziele reduziert (7). Zu guter Letzt wird nun aus den möglichen Zielen ein Plan gebaut, welcher diese Ziele erfüllen möchte (8).

Dieser Planauswahl folgt nun ein neuer, verbesserter Teil (9-23). In den Zeilen (10-12) wird die erste Aktion des aktuellen Planes geholt, ausgeführt und aus dem Plan entfernt. Ist die Anweisung ausgeführt, beobachtet der Agent nun erst einmal die Umgebung um eventuelle Änderungen wahrzunehmen. In Folge dessen wird die Wissensbasis entsprechend aktualisiert (14), geprüft, ob es nötig ist die aktuellen Ziele und Pläne zu überdenken (15) und dies gegebenenfalls getan (16-17). Schlussendlich prüft der Agent noch, ob der aktuelle Plan überhaupt noch sinnvoll ist oder ob sich die Umwelt schon entsprechend verändert hat, dass das Ziel bereits erreicht ist, der Plan nicht mehr zum Ziel führt oder im Plan folgende Aktionen zu einem Fehlschlag führen würden. Ist der Agent mit seinem Plan nicht mehr zufrieden, erstellt er sich einen neuen, ansonsten beginnt die Planausführungsschleife von neuem.

3.4. Kommunikation

So wie die Entscheidungsfindung im BDI-Modell der menschlichen Philosophie angenähert ist, beruht auch die Kommunikation zwischen BDI-Agenten auf Beobachtungen der realen Welt. Kommuniziert ein Agent a mit einem anderen Agenten b, versucht Agent a damit etwas zu erreichen, sei es eine reine Information, ein Befehl oder ähnliches. Die Kommunikation in BDI-Systemen unterscheidet sich kaum von den generellen Grundsätzen der Kommunikation in Multiagentensystemen im allgemeinen. An dieser Stelle soll aufgezeigt werden, welche Arten von Kommunikation es gibt und welcher Motivation diese entspringen.

John Searle betrachtet dies unter dem Begriff *Speech-Act* (Sprechakttheorie) und gliedert dabei den Sprechakt in verschiedene sogenannte illokutionäre Sprechakte [20, S. 166]:

- **Repräsentativa** (auch Assertiva) sind informierende Sprechakte, welche den Hörern vermitteln sollen, dass der Sprecher von der Wahrheit der getätigten Aussage überzeugt ist
- **Direktiva** sind Sprechakte, bei denen der Sprecher sich eine Handlungsreaktion der Hörer verspricht.
- **Kommissiva** sind Sprechakte, welche den Sprecher zu zukünftigen Handlungen verpflichten
- **Expressiva** sind Sprechakte, mit denen der Sprecher seinen psychischen Zustand ausdrücken möchte und sich dabei gesellschaftlicher “Aufrichtigkeitsregeln” bedient (z.B. danken, gratulieren, entschuldigen)
- **Deklarativa** sind Sprechakte, welche die Realität verändern (z.B. jemanden schuldig sprechen, ein Ehepaar trauen)

Wo in der natürlichen Sprache diese fünf Klassen implizit in der Sprache versteckt und für uns Menschen auch nicht schwer zu erkennen sind, haben Programme damit bis heute starke Probleme, weshalb man sich in der Agenten-Kommunikation darauf geeinigt hat, dass eine Nachricht bzw ein Sprechakt aus zwei Teilen besteht:

- Das **Aktionsverb**, welches die Klasse des Sprechaktes beschreibt, z.B. *inform*, *achieve* oder *request*
- Den **ausgesagten Inhalt**, also die eigentliche Nachricht, welche übermittelt werden soll.

Die beiden in dieser Arbeit untersuchten BDI-Systeme Jadex und Jason können im Unterbau mit verschiedenen Agenten-Kommunikations-Sprachen zusammenarbeiten, zwei verbreitete Vertreter sind hier der *FIPA-Standard für Agentenkommunikation* und die *Knowledge Query and Manipulation Language* (KQML), welche beide in späteren Kapiteln noch behandelt werden.

4. Jason und Jadex im Vergleich

In diesem Kapitel wollen wir nun die beiden Frameworks an den im vorigen Kapitel aufgezeigten Punkten vergleichen und analysieren, welches Framework das BDI-Konzept besser umgesetzt hat.
Zuerst wollen wir uns kurz einen Überblick über die grobe Struktur und die Arbeitsweise der Frameworks machen, bevor wir in den folgenden Sektionen ins Detail gehen.

Jadex

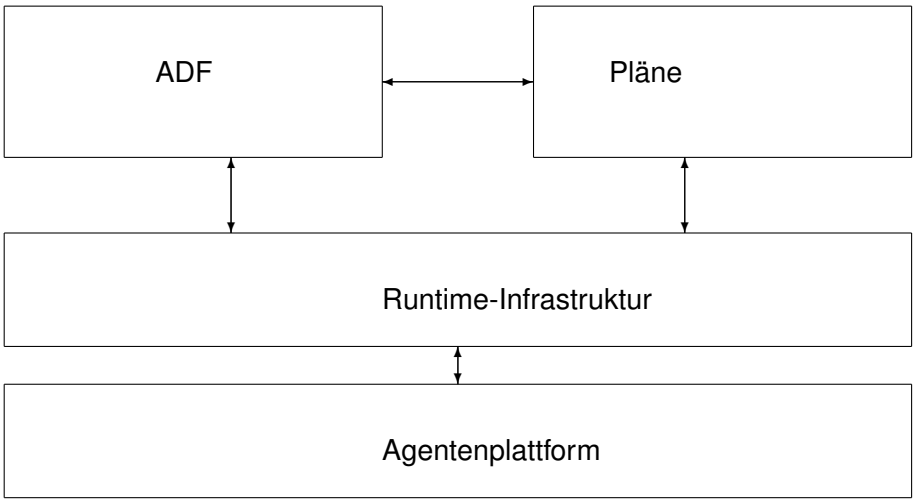


Abbildung 4.1.: Komponenten von Jadex

Jadex ([5]) ist ein von der Universität Hamburg entwickeltes BDI-Framework, welches sich hauptsächlich der Programmiersprache Java bedient. Ein Jadex-Agent besteht aus zwei Komponenten: Zum einen aus der Agent Definition File (ADF), welche im Detail Definitionen sämtlicher Beliefs, Ziele, Pläne, Ereignisse und anderen Elementen enthält. Im Gegensatz zum Rest des Agenten ist die ADF in XML definiert. Die andere Komponente des Agenten besteht aus den in Java formulierten Plänen, welche die eigentlichen Aktionen des Agenten beinhalten. Genauso wie die Pläne sich beliebig in verschiedenste Klassen aufteilen können,

ist es ebenso möglich, die Elemente einer ADF in verschiedene Komponenten aufzuteilen. Der Entwickler hat die Möglichkeit, mit sogenannten *Capabilities*, welche ebenfalls ADFs sind, so gut wie alle Elemente der ADF zu abstrahieren. Diese Capabilities können dann beliebig in verschiedenen wirklichen Agenten oder wieder anderen Capabilities importiert und referenziert werden.

Unter der ADF und der Planbibliothek befindet sich die Infrastruktur des Agenten. Diese ist dafür zuständig, die in der ADF definierten Elemente korrekt zu interpretieren und zu verwalten, das Reasoning durchzuführen und letztlich die Pläne auszuführen.
Die unterste Schicht des Jadex-Frameworks ist die Agentenplattform. Die Hauptaufgaben dieser Komponente sind einmal die Verwaltung aller Agenten des Systems (starten, pausieren, beenden. . .) und das Bereitstellen von Systemschnittstellen für die Agenten. Hauptsächlich ist damit die Kommunikationskomponente gemeint, über die zwei Agenten miteinander in Kontakt treten können. Während die oberen Schichten fest verdrahtet sind, ist die Plattform austauschbar. In der Standardausführung stehen dem Entwickler sowohl eine Standalone-Plattform als auch die deutlich mächtigere JADE-Plattform[1] zur Verfügung.

Jason

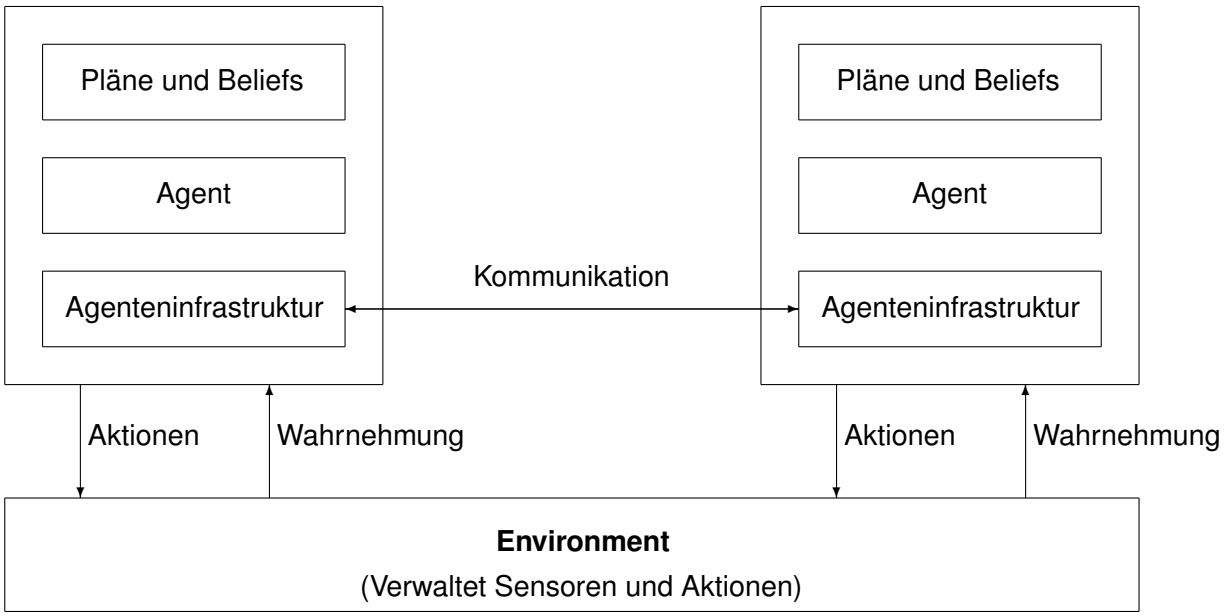


Abbildung 4.2.: Komponenten von Jason

[1] Java Agent DEvelopment Framework [16]

Jason ([4, 3]) ist ein BDI-Framework welches hauptsächlich von Jomi Hübner und Rafael Bordini an der Staatlichen Universität von Santa Catarina entwickelt wurde und verwendet einen gänzlich anderen Ansatz als Jadex. Jason basiert intern zwar ebenfalls auf Java, benutzt für die Programmierung seiner Pläne und Beliefs eine deklarative Sprache namens *Agentspeak*. Durch diese deklarative Programmierung werden Programme deutlich übersichtlicher und lesbarer, wenngleich die Programmierung für viele Entwickler sicherlich ungewöhnlich ist.

Während die Kernkomponenten des Agenten zu Jadex recht ähnlich sind (Beliefs und Pläne, Agent und Kommunikationsinfrastruktur), zeigt sich durch die Environment-Komponente ein klarer Unterschied. Während Jadex keinen konkreten Weg für Agenten zur Interaktion mit der Außenwelt vorgibt und dies dem Entwickler selbst überlässt, definiert Jason ein Environment-Objekt, welches für jegliche Wahrnehmung des Agenten zuständig ist und auch die Aktionen ausführt, welche die Umwelt verändern. Es stellt quasi eine kontrollierende Instanz dar, was jeder einzelne Agent darf und was er nicht darf.

Eine praktische Eigenschaft von Jason ist, dass so gut wie jede Komponente austauschbar ist. Sei es die Implementation der Beliefbase, Auswahl von Plänen, die Kommunikationsinfrastruktur oder das Environment. Im Jason-Paket sind für viele dieser Komponenten mehrere Beispielvarianten enthalten, welche beliebig gegeneinander ausgetauscht werden können, um deren Funktionen an den Agenten anzupassen.

Hunters and Deers

Als Beispielanwendung, an der die einzelnen Unterschiede aufgezeigt werden sollen, dient uns das Eingangs erwähnte Hunters-And-Deers-Szenario von Carsten Canow. Im Rahmen seiner Bachelorarbeit entwickelte er mit Hilfe des DarkStar-Gameservers eine Simulation, welche mit Hilfe von Potentialfeldern die Bewegungen von Rehen und Jägern simuliert, wobei Rehe verständlicherweise vom Jäger fliehen und Jäger die Rehe erlegen wollen (vgl. dazu [7]). Die eigentlichen Agenten implementierte er dort in Jadex, sodass diese Agenten im Rahmen dieser Arbeit lediglich auf Jason portiert wurden. Es wurden keinerlei neue Funktionen hinzugefügt.
Das Hunters-and-Deers-Beispiel wird in den meisten Fällen nur die grundlegenden Komponenten der Frameworks vergleichen können, da eine Anwendung, welche sämtliche Funktionen der Systeme beispielhaft verdeutlicht, den Rahmen dieser Bachelorarbeit sprengen würde.

Vergleich

Nach diesem kurzen Einstieg in die beiden Frameworks beginnt nun der Vergleich in den einzelnen Komponenten der Systeme. Dabei werden wir darauf eingehen, wie strikt sich das Framework an den traditionellen BDI-Entwurf gehalten hat und wie gelungen die Umsetzung technisch ist. Im Detail bezieht sich der Vergleich auf folgende Komponenten:

- Beliefbase
- Ziele
- Pläne
- Reasoning-Cycle
- Kommunikation
- Sicherheit
- Entwicklung
- Performance

4.1. BeliefBase

Die Beliefbase ist neben den Zieldefinitionen und der Planbibliothek einer der drei großen Grundpfeiler eines BDI-Agenten. In ihr speichert der Agent alles, was er glaubt, über seine Umwelt zu wissen. Dieses Wissen ist dabei sein exklusives Wissen. Es muss nicht der Wahrheit entsprechen und andere Agenten können durchaus für den gleichen Sachverhalt ein anderes Belief gespeichert haben.

Die Daten, welche in seiner Beliefbase stehen, sollte der Agent selbst bestimmen. Diese Daten fassen zusammen, was der Agent an Informationen von seinen Sensoren bekommt und auch von anderen Agenten empfängt. Zusätzlich kann der Agent jedoch auch selbst logische Schlüsse ziehen und aus wahrgenommenen Beliefs neue Beliefs erzeugen und in der Beliefbase ablegen. Hierbei müssen Agenten jedoch vorsichtig sein, denn es kann jederzeit vorkommen, dass sich bisher als wahr geglaubte Informationen plötzlich als falsch herausstellen. Dafür sollten BDI-Agenten einen Mechanismus bereit stellen, welcher es erlaubt, bei neuen Wahrnehmungen die Beliefbase auf Inkonsistenzen zu untersuchen. Stellt der Agent zuerst den Fakt $color(box1, blue)$ fest, danach aber den Fakt $color(box1, red)$, sollte der erste Fakt revidiert werden. Diesen Mechanismus nennt man in der BDI-Welt auch *Belief Revision*.

In diesem Zusammenhang spielt auch die Verlässlichkeit der Daten eine Rolle, die Abbildung des aktuellen Zustandes der Welt im Agenten heißt nicht umsonst *Glaubensbasis*. Wo man noch im wissenschaftlichen Umfeld davon ausgehen kann, dass alle Agenten zusammen arbeiten und keiner dem anderen etwas Böses will, kann es im Produktiveinsatz schon eher dazu kommen, dass fremden Agenten nicht getraut werden kann. Vor allem im Spielebereich, wo die einzelnen Agenten von verschiedenen Parteien programmiert werden, können Agenten ihren Kontrahenten absichtlich Fehlinformationen zustecken, um diese aufs Glatteis zu führen. Generell ist die Kontrolle der Glaubwürdigkeit von empfangenen Daten auch aus technischen Gründen ein wichtiger Aspekt. Ist die Wahrnehmung eines Roboter-Agenten auf externe Sensoren angewiesen, muss in der Modellierung des Agenteninnenlebens stets damit gerechnet werden, dass die Sensoren fehlerhaft oder ungenau sind und z.B. auf Redundanzsysteme zurückgegriffen werden sollte. Im Agentensystem müssen also Mechanismen existieren, welche die Glaubwürdigkeit von Fakten bewerten können, sei es auf Basis der Herkunft der Informationen oder auch mit Hilfe der oben genannten Belief Revision.

Da die Beliefbase eines BDI-Agenten aufgrund der bereits gestellten Anforderungen ein sehr komplexes Konstrukt darstellt, ist es unabdingbar, dass diese bereits vom BDI-Framework als Komponente zur Verfügung gestellt und verwaltet wird. Damit der Agent in seinen Zieldefinitionen und Plänen auf dieses Wissen zugreifen kann, werden effektive Zugriffsmechanismen benötigt. Dem Agenten muss es möglich sein, aus der Beliefbase Mengen von Fakten herauszusuchen, welche bestimmten Kriterien entsprechen (z.B. „Gib mir alle Informationen zur Person Paul“). Auch muss eine Beliefbase nach dem Wahrheitsgehalt von Informationen gefragt werden (z.B sinngemäß „Hat Paul braune Augen?“).
Ein weiterer technischer Aspekt ist die Beobachtung von Änderungen. Die einzelnen Komponenten in Form von Zielen oder Zielkategorien sollten dazu in der Lage sein, unabhängig voneinander die Beliefbase im Hinblick auf bestimmte Fakten zu beobachten und bei einer Veränderung des betreffenden Informationsstandes informiert zu werden.

Im Folgenden wollen wir also unser Augenmerk auf folgende Punkte richten.

- Allgemeine Struktur der Beliefbase (deklarativ, imperativ)
- Woher kommt der Inhalt der Beliefbase
- Zugriffsmechanismen
- Beobachtung von Veränderungen innerhalb der Beliefbase
- Mechanismen, um die Verlässlichkeit eines Datums zu bewerten
- Belief Revision

4.1.1. Struktur

Jadex

Die Struktur der Beliefbase ist vor allem wichtig für die Modellierung der Fakten, den Zugriff auf diese und hier haben wir bereits den ersten großen Unterschied zwischen den beiden Frameworks:
Die Definition der Beliefbase in Jadex erfolgt in der Agent Definition File. Dort wird zu Anfang definiert, welche Beliefs es gibt, wie sie heißen und welche Java-Klasse sie darstellen. Konkret gibt es zwei verschiedene Typen von Beliefs:

- **Belief:** Das Belief ist die einfachste Form der Datenspeicherung und stellt eine einzelne Variable dar. Der Typ dieser Variable kann alles sein, was auch in Java als Variablentyp dienen kann, mit der Ausnahme, dass Generics nicht unterstützt werden[2]
- **Beliefset:** Ein Beliefset ist eine Menge von Beliefs des gleichen Typs. Auch hier kann der Typ jede beliebige Java-Klasse sein. Die interne Implementation dieser Menge ist eine Liste, sodass ein und das selbe Element (Referenzgleichheit) auch mehrfach in der Menge vorkommen kann.
- **Beliefreferenz** Sowohl für Beliefs als auf für Beliefsets gibt es die Möglichkeit, Beliefs anderer Capabilities in den eigenen Sichtbereich zu importieren, um auf diese zuzugreifen, sofern sie dort als *exported* markiert wurde.

```
1   <beliefs>
2     <beliefref name="me" class="Agent" exported="true">
3       <abstract />
4     </beliefref>
5     <beliefset name="enemies" class="WorldObject" exported="true" />
6     <beliefset name="friends" class="WorldObject" exported="true" />
7     <beliefset name="prey" class="WorldObject" exported="true" />
8     <beliefset name="food" class="WorldObject" exported="true" />
9     <beliefset name="markers" class="Marker" exported="true" />
10    <belief name="vision" class="Vision" exported="true">
11      <fact>new Vision()</fact>
12    </belief>
13    <belief name="gui" class="boolean" exported="true">
14      <fact>false</fact>
15    </belief>
16  </beliefs>
```

Abbildung 4.3.: Beliefbase der abstrakten *Object*-Capability in Jadex

[2] Bis zur Java-Version 1.6 wird die Benutzung von Generics in der Reflection-API nicht ausreichend unterstützt und kann von Jadex daher auch nicht angewendet werden

Wie in Abbildung 4.3 zu sehen ist, können Beliefs von Capabilities auch abstrakt sein, um sie erst im Agenten selbst zu initialisieren. Auch können Beliefs bei jedem Zugriff dynamisch erneut mit Daten gefüllt werden, sodass es möglich ist, z.B. ein Belief *time* zu erstellen, welches bei jedem Aufruf erneut mit *System.timeMillis()* evaluiert wird.
Da Jadex als Beliefbase Java-Objekte benutzt, kann man durch das Singleton-Pattern verschiedenen Agenten ein und die selben Belief-Objekte zuweisen, sofern diese in der gleichen JVM ausgeführt werden. Das kann in einigen Fällen vielleicht performanter sein, weil Speicher gespart wird, es verleitet jedoch dazu, die Kommunikationsschicht von Jadex zu umgehen und widerspricht auch dem BDI-Modell, in welchem jeder Agent seine exklusiv eigenen Beliefs haben muss.
Die Tatsache, dass Jadex die Beliefbase mit Java Objekten realisiert, hat einen weiteren Konflikt mit dem BDI-Modell zur Folge: Laut diesem ist die Beliefbase eine reine Sammlung von Fakten, die der Agent über seine Umwelt annimmt. Da es in Jadex jedoch kein explizites, separates Konzept gibt, wie der Agent mit seiner Umwelt agieren kann, werden die Schnittstellen mit der Umwelt ebenfalls als Beliefs modelliert, sodass der Agent quasi über seine Beliefbase mit der Umwelt agiert.

Jason

```
1  myID(100).
2
3  worldObject(100, "deer1", deer, idle, position(1, 2, 3)).
4  worldObject(101, "deer2", deer, idle, position(10.5, 2.2, 6)).
5
6  loggedIn.
7
8  // belief rule
9  me(ID, Name, Class, State, Position) :- myID(ID) & worldObject(ID, Name, Class,
10 State, Position).
```

Abbildung 4.4.: Beliefbase in Jason

Jason geht in der Strukturierung seiner Fakten einen deutlich anderen Weg. Während Jadex sich komplett auf Java stützt, betrachtet Jason seine Beliefs als logische Fakten und Schlüsse. Generell gibt es in Jason zwei verschiedene Arten von Beliefs. Einmal sind das einfache Fakten, modelliert als Literale und First-Order-Logic und zum zweiten sind es die Belief-Rules, welche als logische Regeln versuchen, aus den Fakten eine passende Kombination zu finden. Die von Jason benutzte Agentensprache *Agentspeak* ist generell eine typenlose Sprache. Es gibt zwar verschiedene Sprachkonstrukte wie Atome, Zahlen, Strukturen und Listen (welche wieder rekursiv aus anderen Strukturen, Listen, Atomen und Zahlen bestehen können), jedoch können definierte Variablen alle diese Typen annehmen. Es gibt

auch keine klare Struktur in der Menge aller Beliefs. Bei der Suche von Beliefs werden generell erst einmal alle vorhandenen Beliefs in Betracht gezogen und nach Übereinstimmungen gesucht (Name der Struktur und Anzahl ihrer Elemente).

Einer der wichtigen Unterschiede zwischen den Beliefbases von Jadex und Jason ist, dass die Beliefbase von Jadex durch Java einen imperativen Aufbau hat, Jason eine deklarative Darstellung verwendet, basierend auf logischen Konstrukten. Diese Konstrukte erlauben es nicht nur, auf einfache Weise logische Schlussfolgerungen aus bekannten Fakten zu ziehen und sich so neues Wissen zu erschließen, die Logik von Agentspeak ist außerdem eine dreiwertige Logik. Unter Agentspeak ist es nicht nur möglich, einen Belief $color(box1, blue)$ zu definieren, sondern auch einen gegenteiligen Belief $\sim color(box1, blue)$,welcher ausdrückt, dass der Agent zwar nicht weiß, welche Farbe die Box hat, aber dass sie auf jeden Fall nicht blau ist. Durch diese explizite Unterstützung der *Open-World-Assumption*, in der es neben *true* und *false* auch noch den logischen Zustand *unknown* gibt, können Zusammenhänge deutlich einfacher dargestellt werden, als in Jadex, wo eine solche Funktionalität zuerst recht umständlich in Java modelliert werden muss.

Die eigentliche Implementierung der Beliefbase in Jason ist in Java realisiert. Hierbei ist schon die erste Stärke von Jason zu sehen. Der Agent erwartet für die Verwaltung seiner Beliefs einzig ein Interface zu einer Menge, in welche er Fakten hinzufügen und entnehmen kann und diese Menge durchsuchen kann. Wie diese Literale abgelegt werden, bleibt der Implementation überlassen, welche zum Start eines Agenten frei definiert werden kann. Die Standardimplementation der Beliefbase ist zu Beginn leer und speichert hinzugefügte Fakten in optimierten Listen im Hauptspeicher. Weitere schon mitgelieferte Implementationen lesen zu Beginn einen Satz Beliefs aus einer Datei und sichern neue Beliefs regelmässig in diese Datei. Eine andere Beliefbase ist über JDBC an eine beliebige Datenbank angebunden und ließt ihre Beliefs dort aus und speichert sie dort hinein.
Man könnte sich z.B. auch vorstellen, dass bestimmte Literale einen Primärschlüssel besitzen. Die Standardimplementation würde nun Objekte mit dem gleichen Primärschlüssel mehrfach speichern, wenn sich diese in anderen Attributen unterscheiden. Mit einer entsprechenden Modifikation der Beliefbase kann dies aber abgefangen werden indem bestimmte Literale gesondert behandelt werden.

Die deklarative Darstellung der Beliefs kann jedoch in einigen Fällen auch einen Nachteil darstellen. Sind die Jason-Agenten Teil eines größeren Softwareprojektes, kann es sein, dass Teile der Logik außerhalb des Agenten modelliert werden müssen (z.B. bei komplexen externen Wahrnehmungssystemen). Da das Backend von Jason in Java programmiert wird, bedeutet dies, dass dafür auch Teile des Datenmodells als Java-Objekte realisiert werden müssen, um vom Backend behandelt werden zu können, da sich die Literal-Objekte von Jason zum einen nur sehr kompliziert in Java nutzen lassen und zum anderen mit vielen Frameworks von Drittanbietern, z.B. Hibernate als Datenbank-Zugriffs-Layer, nicht kompati-

bel sind. Diese doppelte Modellierung ist zwar problemlos möglich, jedoch stellt sie auch ein großes Fehlerpotenzial und einen hohen Wartungsaufwand dar.

4.1.2. Zugriff

```
<expressions>
  <expression name="find_person" class="Person">
    select one Person $person
    from $person in $beliefbase.persons
    where $person.getSurname().equals($surname)
    <parameter name="$surname" class="String"/>
  </expression>
  ...
</expressions>
```

Abbildung 4.5.: Beispielausdruck für die OQL von Jadex. [6](S. 62)

Der Zugriff auf die Beliefbase gliedert sich in Jadex in zwei Sektionen. Einmal kann der Zugriff aus der Agent-Definition-File erfolgen, wenn z.B. Bedingungen für die Zielverfolgung definiert werden. Weiterhin kann auch aus Java-Plänen heraus auf die Beliefbase zurück gegriffen werden.
In der ADF benutzt Jadex eine etwas abgewandelte Form der OQL[3], um die Beliefbase zu durchsuchen. Diese erlaubt es dem Entwickler, sowohl einfache Konstrukte wie *$beliefbase.food.length* zu formulieren, um die Anzahl der bekannten Futterstellen zu ermitteln, als auch kompliziertere Ausdrücke mit verschachtelten Bedingungen zu formulieren.

```
public class ConsumeFoodPlan extends Plan {
  private Agent me;

  public void body() {
    me = (Agent) getBeliefbase().getBelief("me").getFact();

    WorldObject[] foodFacts = (WorldObject[]) me.getAgent().getBeliefbase().getBeliefSet("
        food").getFacts();
    ...
  }
}
```

Abbildung 4.6.: Zugriff auf die Beliefbase innerhalb eines Plans

Der Zugriff auf Beliefs innerhalb eines Planes gestaltet sich eher unspektakulär mit dem Aufruf entsprechender Methoden und der Angabe des Beliefbezeichners. Etwas ungewöhnlich

[3]Object Query Language: Erweiterung von SQL, um auf objektrelationalen Datenbanken Abfragen zu definieren

für den Java-Entwickler ist, dass das zurückgegebene Objekt noch einmal zum passenden Typ gecastet werden muss. Da Jadex die Beliefbase erst zur Laufzeit erzeugt, kann nicht auf die interne Typsicherheit von Java zurück gegriffen werden, sodass Jadex diese selbst implementieren muss, was sowohl Zeit kostet, als auch für den Entwickler in umständlichen Casts endet oder gar zu unschönen Ergebnissen zur Laufzeit führen kann, wenn der Typ des Beliefs nicht mit dem Zieltyp des Casts kompatibel ist. Auf gleichem Wege werden der Beliefbase auch neue Daten hinzu gefügt. Die initiale Faktenmenge der Beliefbase wird direkt in der ADF angegeben, weitere Fakten können dann nur noch über die üblichen Java-Beans- oder Collection-Methoden innerhalb von Plänen hinzugefügt werden, eine Modifikation der Daten von außerhalb, z.B. vom Kommunikationsframework findet nicht statt.

Der Zugriff auf die Beliefbase in Jason geschieht hauptsächlich durch Unifikation. Hierbei wird ein Literal erstellt, welches in der Signatur (Name und Anzahl der Parameter) Literalen gleicht, welche bereits in der Beliefbase vorhanden sind, jedoch werden einige Parameter einfach offen gelassen bzw. mit Variablen besetzt. Der Interpreter durchsucht nun die Beliefbase nach passenden Literalen ab und ersetzt dabei die Variablen mit den entsprechend gefundenen Werten. Wird eine Beliefbase mit den Beliefs $a(20)$, $a(40)$, $b(1,p)$, $b(2,q)$ mit dem Literal $a(X)$ durchsucht, hat X danach den Wert 20, da standardmäßig immer der erste gefundene Wert zurückgeliefert wird. Das Literal $b(1,X)$ unifiziert X mit p, die Suche nach dem Literal $a(30)$ dagegen schlägt komplett fehl, da kein passender Belief in der Beliefbase gefunden wurde.

Das Hinzufügen neuer Informationen zur Beliefbase findet sich an drei Stellen wieder. Einmal werden Wahrnehmungen, welche das Environment an den Agenten weiterleitet, von der Agenten-Infrasturktur direkt in die Beliefbase geschrieben. Gleiches passiert ebenso mit Informationen, welche von anderen Agenten empfangen werden. Im dritten Fall kann die Beliefbase eines Agenten auch von innerhalb seiner Pläne modifiziert werden, indem neue Fakten hinzugefügt oder vorhandene gelöscht werden. Die Änderung vorhandener Fakten ist durch die deklarative Struktur hingegen nicht möglich.

```
1  .findall(worldObject(A, B, grass, D, Pos1), (worldObject(A, B, grass, D, Pos1) &
       distanceSquared(Position, Pos1, Distance) & Distance <= Range*Range), Food)
```

Abbildung 4.7.: Beispielaufruf der Durchsuchungsfunktion von Jason

Ein besonderes Augenmerk ist in Jason auf die interne Aktion $.findall()$ zu richten. Mit dieser Funktion kann die gesamte Beliefbase mit einer logischen Formel nach passenden Fakten durchsucht werden, Dabei kann das Ergebnisliteral auch aus Daten verschiedener gesuchter Literale zusammengesetzt werden.

4.1.3. Beobachtung von Veränderungen

Da Agenten in einem Multiagentensystem meist reaktive Agenten sind, welche auf Veränderungen in der Umwelt reagieren müssen und so ihre Ziele setzen, oder wenn sie proaktive Agenten sind, müssen sie ihre Ziele und Pläne auf Veränderungen in ihrer Umwelt anpassen. Darum ist es wichtig zu analysieren, wie Veränderungen in der Umwelt an den Agenten weitergereicht werden und wie diese darauf reagieren können.

Sowohl in Jadex als auch in Jason werden Veränderungen in der Umwelt zuerst in die Beliefbase eingetragen. In Jadex gibt es nun die Möglichkeit, Beliefs als Observable zu definieren, was bedeutet, dass sie Teile des PropertyChangeSupport-Interfaces implementieren und dass das Beliefobjekt diese Methoden auch entsprechend nutzt. Findet der Agent diese Methoden bei einem Belief-Objekt, registriert er einen Observer und wird so über alle Veränderungen des Beliefs unterrichtet. Das häufigste Anwendungsgebiet dieser Technik sind die Aufrufbedingungen in Zieldefinitionen, dort werden Ziele abhängig von vorher definierten Bedingungen erstellt, pausiert oder abgebrochen.

In Jason resultieren Veränderungen an der Beliefbase in automatisch generierten Ereignissen: Einmal das Ereignis, dass ein Belief hinzugefügt und einmal, dass eines entfernt wurde. Bei jedem dieser Ereignisse wird nun nach einem Plan gesucht, welcher dieses Ereignis als Auslöser hat und im aktuellen Kontext ausführbar ist. Ist dieser gefunden, wird er zur Liste der Intentionen hinzugefügt und bearbeitet.

Der wichtige Unterschied beider Vorgehensweisen liegt im Detail: Bei einem Belief-Objekt in Jadex können nach dem Observer-Pattern beliebig viele Observer registriert werden, sprich wenn das Belief in mehreren Ziel-Bedingungen als Auslöser definiert ist, kann eine einzige Beliefänderung auch mehrere neue Ziele zur Folge haben, welche jedoch komplett unabhängig voneinander definiert sind und nichts miteinander zu tun haben müssen.
In Jason hat jede Beliefänderung genau einen Plan zur Folge (außer es wurde keiner gefunden). Sollte ein Agent an mehreren Stellen im Programmcode auf eine Änderung in der Beliefbase warten wollen, ist dies nur mit einem Zwischenschritt möglich, in welchem ein Plan das Belief beobachtet und dann explizit andere Programmteile aufruft, welche auf die Beliefänderung reagieren wollen. Dieser Zwischenplan führt jedoch zu einer starken Kopplung der einzelnen Komponenten eines Agenten, da der Zwischenplan alle betreffenden Komponenten kennen muss und dies ist in der Softwareentwicklung zu vermeiden.

4.1.4. Modellierung von Vertrauenswürdigkeit

Wie schon an anderen Stellen betont, kann die Vertrauenswürdigkeit der externen Informationen eine große Rolle spielen. Sei es, dass andere Agenten in entsprechenden Multiagentensystemen (z.B. Spielen) ihre Nachbarn schlicht anlügen, um von ihnen Fehler zu

provozieren oder dass die eigenen Sensoren fehlerhafte oder ungenaue Daten liefern. In beiden Fällen muss das Agentensystem in der Lage sein, die Glaubwürdigkeit einer Information festzustellen und diese dann entsprechend zu behandeln bzw. zu kennzeichnen.

```
1 color(box1, red)[source(ag1)].
2 color(box2, red)[source(ag2)].
3 color(box2, blue)[source(ag3)].
4 ~color(box4, green)[source(ag3)].
5 color(box4, yellow)[source(percept)].
6
7 liar(ag3).
```

Abbildung 4.8.: Beispiel für die Nutzung von Annotations

Während Jadex diesen Aspekt des BDI-Modelles ignoriert und dem Entwickler die Modellierung entsprechender Konstrukte überlässt, bietet Agentspeak die Möglichkeit, jedem Belief Annotations bzw. Anmerkungen hinzuzufügen. Diese Anmerkungen haben für den eigentlichen Interpreter erst einmal keine gesonderte Bedeutung, jedoch können diese Anmerkungen in allen Abfragen an die Beliefbase genutzt werden. Jedes Belief hat zum Beispiel eine source-Annotation, welche aussagt, woher dieses Belief kommt. Entweder hat der Agent es wahrgenommen (*percept*), er hat es sich selbst erschlossen (*self*) oder von einem fremden Agenten über die Kommunikationsschnittstelle erhalten. Bekommt man über ein und den selben Sachverhalt, wie in Abbildung 4.8(2-3) dargestellt, nun mehrere Informationen (einmal ist box2 rot, einmal blau), kann im Nachhinein analysiert werden, welchem Fakt eher Glauben geschenkt werden sollte.
Was für Annotationen verwendet werden, ist gänzlich dem Entwickler überlassen, jedoch gibt es einige vordefinierte Annotations wie die source-Annotation, die dann vom Interpreter gesondert behandelt werden (wir werden noch weitere kennen lernen). Die Anwendungsbereiche sind vielfältig; Beliefs können mit einem Datum versehen werden, wann sie erstellt wurden, mit einem Wert, wie vertrauenswürdig dieser Fakt ist oder Pläne mit Prioritäten, in welcher Reihenfolge sie abgearbeitet werden sollen.

In Zusammenhang mit der Validierung von Beliefs spielt der Mechanismus der so genannten *Belief Revision* eine große Rolle. Belief Revision ist ein Vorgang, welcher bei neuen Wahrnehmungen diese daraufhin untersucht, ob durch diese neuen Kenntnisse bisherige Fakten in Frage gestellt werden sollten (Vgl. [4, S. 22]). Das einfachste Beispiel ist, dass der Agent einen Fakt $b(1)$ kennt, nun aber merkt, dass $b(1)$ gilt, also die Negation von $b(1)$. Eine Beliefbase, welche beide Fakten speichert, wäre logischerweise inkonsistent. Während Jadex dieses recht komplexe Gebiet komplett dem Entwickler überlässt, greift Jason ihm zumindest ein wenig unter die Arme. Dort wird jeder neue Fakt, welcher wahrgenommen wird (oder auch alte Fakt, welcher nun nicht mehr wahrgenommen wird), einer Belief-Revision-Function übergeben. In der Standardimplementation macht diese Funktion nichts, jedoch ist geplant, sie in künftigen Versionen mit einer sinnvollen Implementation zu versehen. Bis

es soweit ist, muss der Entwickler den Algorithmus selbst implementieren, Beispielkonzepte findet man z.B. in [1].

4.1.5. Zusammenfassung

	Jadex	Jason
Struktur	imperativ in Java	deklarativ in First Order Logic
Zugriff	OQL, Java-Beans	logische Formeln, Unifikation
Informationsquellen	Modifizierung nur beim Start oder aus Plänen heraus	Modifizierung vom Environment, der Kommunikationsschnittstelle und aus Plänen heraus
Beobachtung von Veränderungen	Observer-Pattern	ereignisgesteuert, nur ein Observer pro Ereignis möglich
Modellierung von Vertrauenswürdigkeit	Eigenbau nötig	Annotations (Speziell *source-Anntoation*)
Belief Revision	Eigenbau nötig	Grundgerüst vorhanden, Implementation fehlt

Tabelle 4.1.: Zusammenfassung: Vergleich der Beliefbases

Wir haben gesehen, dass die Beliefbases von Jadex und Jason in ihrem Aufbau nicht verschiedener sein könnten. Während Jadex auf eine imperative, in Java realisierte Wissensbasis baut, ist die Beliefbase von Jason in Agentspeak durchweg deklarativ aufgebaut, wenn auch sie in einem Java-Unterbau realisiert ist.
Die deklarative Beschreibung von Fakten lässt die Beliefbase sehr übersichtlich erscheinen, was die Wartbarkeit erhöht und Abfragen vereinfacht. In der Beliefbase in Jadex lässt sich durch die objektorientierte Programmierung von Java aus eine komplexe Umwelt gut und einfach modellieren, jedoch weist sie hier zwei Konflikte mit dem BDI-Konpezt auf (nicht-exklusive Wissensbasis durch das Singleton Pattern und Trennung von Beliefs und Aktionslogik).

Historisch stammt die Sprache Agentspeak aus dem wissenschaftlichen Sektor, wo Planungsalgorithmen wie STRIPS[4] oder GOAP[5] mit logischen Ausdrücken beschrieben werden. Diese Literale können ohne große Übersetzungen einfach aus der Theorie in ein

[4]Stanford Research Institute Problem Solver: Planungsalgorithmus entwickelt von der Stanford Universität [14, S. 377]
[5]Goal Oriented Action Planning [13]

Agentspeak-Programm übernommen werden, wo in Jadex mit Java eine komplett neue Modellierung erforderlich ist. Agentspeak wurde konkret als Sprache auf Agentensysteme, speziell BDI, entwickelt, sodass direkt praktische Mechanismen wie Belief Revision oder Annotations eingebaut werden konnten, welche in Java zwar auch ohne weiteres nachgebaut werden können, dort aber die Komplexität deutlich steigen lassen.

Nachholbedarf gibt es bei Jason definitiv bei der Beobachtung von Veränderungen in der Beliefbase. Während Jadex dabei das Observer-Pattern anwendet, ist es bei Jason zwar möglich, auf eine Änderung zu reagieren, dies jedoch nur mit einem einzigen Observer, was die Kopplung von Komponenten stark erhöht und das Programm schlecht wartbar macht.

4.2. Ziele

Der zweite große Grundpfeiler von BDI ist die Deklarierung von Zielen. Sie modellieren, was der Agent in seinem Lebenzyklus erreichen möchte. Da die Ziele eines Agenten sehr vielfältig sein können, benötigen BDI-Systeme ebenso vielfältige Definitionsmöglichkeiten. Die zwei wichtigsten Typen von Zielen, auf die kaum Agenten verzichten können sind einmal die *Achieve Goals* und die *Maintain-Goals* (vgl. [20, S. 41]).

Verfolgt der Agent ein Achieve Goal, versucht er, einen im Achieve Goal definierten Zustand zu erreichen und geht davon aus, dass dieser Zustand auch erreicht ist, sobald der gewählte Plan fertig ausgeführt wurde. Unter Umständen kann es vorkommen, dass das Ziel eines Achievegoals bereits vor Beendigung des Planes erfüllt ist, weil äußere Einwirkungen den Agenten bereits in den Zielzustand versetzen. Dann muss es eine Möglichkeit geben, das Ziel vorfristig abzubrechen, wobei darauf geachtet werden muss, dass abgebrochene Pläne keinen inkonsistenten Zustand hinterlassen.

Der Agent kann von Beginn an einige Achievegoals verfolgen oder aber auch neue Achievegoals bekommen, wenn sich die äußeren Einflüsse des Agenten ändern (z.B der Agent neue Wahrnehmungen oder Nachrichten von anderen Agenten erhalten hat). Entsprechend müssen für den zweiten Fall Mechanismen vorhanden sein, welche es erlauben, nach Wahrnehmungszyklen auf eventuelle neue Ziele zu prüfen, welche verfolgt werden sollten.

Das Maintain Goal auf der anderen Seite ist ein eher passives Ziel. In ihm ist eine Bedingung definiert, welche der Agent versucht einzuhalten. Stellt der Agent während seiner Wahrnehmungszyklen fest, dass die Maintain-Bedingung nicht mehr eingehalten wird, wird das Ziel aktiv und der Agent versucht Pläne zu finden, damit die Bedingung wieder erfüllt werden kann. Wie auch beim Achievegoal kann das Ziel vorfristig erfüllt werden, sodass Pläne, welche die Maintain-Bedingung versuchen wiederherzustellen, abgebrochen werden sollten, ohne Inkonsistenzen zu hinterlassen.

In einigen Fällen kann es sein, dass der Agent eine Aktion ausführen soll, ohne das genaue Ergebnis dieser Aktion zu kennen, also ohne dass der Agent eine Zielbedingung angeben kann, welche erfüllt werden muss (z.B. „Patroulliere zwischen A und B“). Für diesen Fall benötigen Agenten eine Art *Performgoal*, bei welchen der Agent einfach einen passenden Plan aussucht und ausführt, ohne auf Bedingungen zu achten. Im gleichen Zug kann es vorkommen, dass Agenten zyklisch Ziele erreichen wollen, z.B. alle 10 Sekunden einen Schritt weiter gehen oder einen anderen Agenten nach Informationen fragen. Für diese Funktionalität muss eine Art Scheduling möglich sein, welches wiederkehrende Ziele planen kann.

Ein wichtiger Punkt bei Zielen ist die Konsistenz dieser. Hierbei geht es darum, dass Agenten im Normalfall mehrere Ziele parallel verfolgen und der Agent dabei verhindern muss, dass sich zwischen diesen Zielen Konflikte ergeben. Entdeckt ein Saugroboter z.B. sowohl links als auch rechts neben sich Dreck, wäre es recht ungünstig, gleichzeitig den linken als auch den rechten Fleck säubern zu wollen. Genauso sollte das Ziel *Batterie aufladen* immer eine höhere Priorität haben als das Ziel *Saugen*. In den BDI-Systemen muss es also eine Umsetzung von *Goal Deliberation* geben, welche eine konsistente Zielauswahl ermöglicht.

Unsere Vergleichspunkte für Ziele in BDI-Systemen sind also folgende:

- Achievegoal
- Maintaingoal
- Performgoal
- Ziele aus Ereignissen heraus verfolgen
- Zeitplanung für Ziele
- Goal Deliberation

4.2.1. Definition von Zielen

Jadex

Jadex ist bei der Definition verschiedener Ziele XML-typisch äußerst spendabel und unterstützt vier Haupttypen von Zielen:

- **Performgoal** Mit dem Performgoal versucht der Agent eine Aktion auszuführen, welche jedoch nicht direkt an einen Zielzustand gebunden ist, weil dieser evtl. unbekannt ist (z.B. wenn ein Roboter wiederholt zwischen zwei Punkten patroullieren soll).

- **Achievegoal** Das Achievegoal versucht nun, eine bestimmte Zielbedingung zu erfüllen, welche sich zumeist auf Informationen in der Beliefbase bezieht. Dafür werden, abhängig von der Konfiguration alle in Frage kommenden Ziele der Reihe nach aufgerufen, bis die Zielbedingung erreicht ist, oder das Ziel abgebrochen wird, was unter Umständen auch während einer Planausführung passieren kann.
- **Querygoal** Das Querygoal ist ein sehr technisches Ziel, welches in dieser Form in der Theorie nicht vorkommt. Es dient zur Informationsfindung, sollte ein benötigter Fakt nicht in der Beliefbase existieren. Dieses Ziel ist vergleichbar mit dem Achievegoal („Erreiche, dass die Information verfügbar ist“).
- **Maintaingoal** Das Maintaingoal ist eine Art erweitertes Achievegoal. Das wichtigste Merkmal des Maintaingoals ist, dass es von Beginn an aktiv ist und prüft, ob die zu erhaltende Bedingung eingehalten wurde. Stellt das Ziel fest, dass dem nicht mehr so ist, wird versucht, darauf zu reagieren und Pläne zu finden, die eine spezifizierte Zielbedingung wieder herstellen, z.B. die Batterie des Agenten wieder voll aufladen.

Zusätzlich zu diesen vier hauptsächlich verwendeten Zieltypen gibt es mit dem *Metagoal* noch einen weiteren Zieltyp, welcher für das Reasoning eine wichtige Rolle spielt. Das Metagoal ist eine Spezialform des Querygoal, welches aufgerufen werden kann, wenn der Agent zu einem Ziel mehrere mögliche Pläne findet. In diesem Fall kann ein Metagoal aufgerufen werden, welches aus allen möglichen Plänen einen Plan auswählt und dem Agenten zurück gibt.

Jeder dieser Zieltypen kann mit einigen weiteren Attributen genauer spezifiziert werden.

- **retry** (**true**|false) Schlägt ein gewählter Plan für dieses Ziel fehl, werden andere Pläne gesucht und ausgeführt.
- **retrydelay** (long value) Gibt an, in welchen Abständen das Ziel erneut versucht werden soll, erreicht zu werden.
- **recur** (**true**|false) Gibt an, ob das Ziel automatisch erneut angegangen werden soll. So kann man z.B. ein *walk*-Ziel definieren, welches den Agenten alle 1000ms ein Feld bewegen soll.
- **recurdelay** (long value) bestimmt, in welchen Abständen das Ziel erneut erstellt werden soll.

Bei der Erstellung dieser Ziele können ebenfalls vorher in der Agent Definition File festgelegte Parameter übergeben werden. Durch die Möglichkeit, Parameter als *in* oder *out* zu kennzeichnen, sind auch Rückgabewerte möglich, wenn Ziele in anderen Plänen als Unterziele definiert werden.

```
1   <goals>
2     <achievegoal name="achieve_consumeFood" exclude="never"
3       exported="true">
4       <unique />
5       <creationcondition>
6         $beliefbase.food.length &gt; 0
7       </creationcondition>
8       <contextcondition>
9         $beliefbase.me.isAlive()
10      </contextcondition>
11      <targetcondition>
12        $beliefbase.food.length == 0
13      </targetcondition>
14    </achievegoal>
15    <performgoal name="perform_walk" retry="true" exclude="never"
16      retrydelay="1000" exported="true">
17      <unique />
18      <contextcondition>
19        $beliefbase.me.isAlive()
20      </contextcondition>
21    </performgoal>
22  </goals>
```

Abbildung 4.9.: Zieldefinitionen in Jadex

Allen Zielen können verschiedene Bedingungen annektiert werden, welche die Ausführung beeinflussen:

- Ist die **Createcondition** eines Zieles erfüllt, wird eine neue Instanz dieses Zieles erstellt und aktiv verfolgt.
- Die **Contextcondition** prüft während der Aktivzeit eines Zieles durchgehend, ob dieses Ziel aktuell verfolgt werden sollte. Schlägt sie fehl, wird das Ziel pausiert, bis die Bedingung wieder erfüllt ist.
- Mit der **Dropcondition** kann ein Ziel vorzeitig abgebrochen werden, wenn es bereits erfüllt oder nicht mehr sinnvoll ist.

Alle diese Bedingungen werden als Observer in den entsprechenden Beliefs eingetragen und automatisch getriggert, sobald sich diese ändern, sodass kein Polling nötig ist und das Ergebnis der Bedingungen trotzdem immer auf dem aktuellsten Stand ist.

Mit diesen Definitionen können sich so gut wie alle benötigten Ziele definieren lassen. Verwirrend dabei ist, dass die oben genannten Attribute für sämtliche Zieltypen verfügbar sind. So ist unverständlich, warum ein Maintaingoal, welches von sich aus bereits ein dauerhaftes Ziel ist, das Attribut *recur* unterstützt, gleiches gilt auch für die Typen Query- und Metagoal, welche vom Konzept her aus gerade ausgeführten Plänen heraus aufgerufen werden und diesen Daten zurückgeben sollten.

Die Verwendung von *retry* ist ebenfalls mit Vorsicht zu genießen, da Pläne nicht atomar konsistent ausführt, was dazu führen kann, dass ein Plan, welcher kurz vor dem Ende fehlgeschlagen ist, bereits die Umwelt in einer Weise verändert hat, dass weitere Pläne für dieses Ziel nicht mehr anwendbar sind. Die Konsistenz ist auch bei Zielabbrüchen ein wichtiger Punkt. Hier muss der Entwickler sicherstellen, dass Pläne, welche mitten in ihrer Ausführung abgebrochen werden, keinen „Restmüll“ hinterlassen oder die Beliefbase in einen inkonsistenten Zustand versetzen.

Jason

Die Definition oder Erstellung von Zielen in Jason gestaltet sich verglichen mit der Erstellung der Ziele in Jadex erheblich anders. Genau genommen gibt es in der Agentensprache Agentspeak keine Ziele im eigentlichen Sinne. Dort geschieht die Zielauswahl ausschließlich durch Ereignisse. Von diesen gibt es in Agentspeak zwei grundlegende Typen:

- *interne Ereignisse* werden vom Agenten selbst ausgelöst, während er versucht einen Plan auszuführen. Das können z.B. explizite Aufrufe von Subzielen sein oder Modifikationen in der Beliefbase, welche ebenfalls eine Änderungsereignis zur Folge haben. Interne Ereignisse blockieren den auslösenden Plan solange, bis das Ereignis vollständig abgearbeitet wurde, sodass sich bei verschachtelten Ereignissen eine Art Stack bildet, welcher von oben nach unten abgearbeitet wird.
- *externe Ereignisse* werden von außerhalb des Agenten ausgelöst, wofür es generell nur drei potenzielle Quellen gibt: Zum einen können andere Agenten dem Agenten Nachrichten schicken, welche ihn dazu veranlassen, neue Ziele verfolgen zu wollen oder welche ihm neue Informationen mitteilen, welche eine Nachbearbeitung benötigen. Weiterhin kann der Agent neue Wahrnehmungen von der Umwelt empfangen oder alte verlieren, was ebenso in einer Änderung der Beliefbase resultiert und ein Ereignis zur Folge hat. Der letzte und besondere Fall ist ein internes Ereignis, welches explizit als externes Ereignis wahrgenommen werden soll. Externe Ereignisse haben für den Interpreter keinen konkreten „Verursacher“, welchen er unterbrechen und zum Warten zwingen könnte, wodurch diese Ereignisse parallel zu anderen Ereignisse bearbeitet werden und etwaige sonstige Ereignisse in ihrer Abarbeitung nur bedingt beeinflussen.

Eine weitere Klassifizierung für Ereignisse, welche der Interpreter vornimmt basiert nicht auf der Herkunft eines Ereignisses sondern auf seinem Zweck:

- **+!g** (Aufruf: $!g$) bezeichnet das Hinzufügen eines neuen *Achievegoals* g, welches der Agent erreichen möchte. Als Besonderheit kann dieses Ziel innerhalb von Plänen auch

```
1  /* initial goals */
2  !login.
3  !walk.
4
5  +!login: true
6    <- login(deer).
7
8  +!walk : me(ID)
9      & worldObject(ID, Name, Type, State, Position)
10   <-
11     !calcTargetPosition(Position, TargetPosition);
12     ...
```

Abbildung 4.10.: Ziele und Pläne in Jason

mit dem Ausdruck $!!g$ aufgerufen werden, welches zur Folge hat, dass es als externes Ereignis behandelt wird

- **-!g** wird ausgelöst, wenn ein Plan, welcher ein Achievementgoal bearbeitet, fehlschlägt oder das Ziel von außerhalb manuell entfernt wird
- **+g** beschreibt das Hinzufügen eines neuen Fakts g zur Beliefbase des Agenten
- **-g** bezeichnet das Entfernen bereits vorhandener Fakten aus der Beliefbase
- **+?g** (Aufruf: $?g$) beschreibt das Ziel, die Beliefbase nach dem Fakt g
- **-?g** wird ausgelöst, wenn das Ziel, die Beliefbase zu durchsuchen, fehlschlägt oder von außen beendet wird

Gemeinhin wird bei einem Ereignis $!g$ von einem Achievegoal und bei einem Ereignis $?g$ von einem Testgoal gesprochen.

Generell geht beim Verständnis von Zielen in Jason/Agentspeak die Theorie und Praxis weit auseinander. Michael Wooldridge beschreibt in der Theorie von Agentspeak zwei verschiedene Arten von Achievegoals (vgl. [4, Seite 40/41]): Die einfache Form ist das prozeduale Ziel, welches, ähnlich wie eine Funktion in C eine simple Folge von Anweisungen ist, die ausgeführt werden sollen, also das Komplement zum Performgoal in Jadex. Der laut ihm bei weitem wichtigere Typ ist das deklarative Ziel, welches das eigentliche Achievegoal darstellt. Man nehme an, ein Agent habe das Ziel $!open(door)$ erhalten. Daraus folgt, dass der Agent die Tür öffnen soll und es wird auch vorausgesetzt, dass die Tür nach dem Ausführen eines passenden Planes tatsächlich offen ist. Programmtechnisch sichergestellt wird diese Nachbedingung allerdings erst durch ein Testgoal $?open(door)$, welches am Ende des Planes ausgeführt wird und nur erfolgreich abschließt, wenn ein Belief $open(door)$ in der Beliefbase existiert.
In Agentspeak selbst wird nirgendwo zwischen diesen zwei verschiedenen Zielen unterschieden. Hat der Entwickler nicht selbstständig das Testgoal an den Plan angefügt, könnte nicht

```
1  { begin dg(open(Door)) }
2  +!open(Door) : true <- openTheDoorAction(Door).
3  { end }
4
5  /* becomes: */
6
7  //do nothing if Door is already open
8  +!open(Door): open(Door).
9  //do something to open the door an check if it succeeded
10 +!open(Door) <- openTheDoorAction(Door); ?open(Door).
11 // stop trying to open the door if it became open by other actions
12 +open(Door) <- .succeed_goal(open(Door)).
```

Abbildung 4.11.: Die Declarative Goal Direktive in Jason

garantiert werden, dass die Tür tatsächlich offen ist.
Ein Achievegoal ist für den Interpreter generell nur ein Funktionsaufruf, bei dem in der Funktionsbibliothek nach einem passenden Kandidaten gesucht wird, vergleichbar mit Programmiersprachen, in denen Funktionen nicht nur durch ihren Namen, sondern auch durch die Typen ihrer Parameter definiert sind. Es ist weder möglich, vor oder während der Ausführung von Plänen auf andere Ereignisse zu warten oder Beliefs zu beobachten. Von Haus aus gibt es keine Art von Maintaingoal, welches dauerhaft passiv ist und nur aktiv wird, wenn sich in der Beliefbase entsprechende Fakten ändern.
Für viele der genannten Schwachstellen stellt die Jason-Umgebung sogenannte Direktiven zur Verfügung (vgl. Abbildung 4.11), vergleichbar mit C-Makros, welche ausgewählte Pläne automatisch so mit weiteren Anweisungen modifizieren oder umwandeln, sodass implizit eine weitere Unterteilung in Perform-, Achieve- und Maintaingoal erfolgen kann, jedoch wirkt dieser Mechanismus aufgesetzt und nicht sofort ersichtlich, was zu Verwirrung führen kann.

4.2.2. Zeitplanung

Eine Art Zeitplanung gibt es in Jadex an zwei Stellen. Einmal können in sämtlichen Zieldefinitionen die Attribute *recur* und *recurdelay* angegeben werden, welche ein Ziel nach einer definierten Zeit erneut auftreten lassen. Bei der Zielerstellung angegebene Parameter bleiben dabei gleich. Die zweite Stelle, an der eine Art Zeitverwaltung stattfindet, ist innerhalb von Plänen, wo mit Hilfe der Funktion $waitFor(...)$ auf verschiedenste Ereignisse gewartet werden kann. Das können Zielerfüllungen sein oder auch Beliefänderungen oder einfach eine bestimmte Zeitspanne.

Zeitplanung in Jason wird mangels konkreter Zieldefinitionen auf die Pläne ausgelagert. Hier stehen dem Entwickler die zwei internen Funktionen $.wait(Timeout)$ und

$.at(When, What)$ zur Verfügung. Die Funktion $.at$ erzeugt zum gegebenen Zeitpunkt das übergebene Ereignis.

4.2.3. Goaldeliberation

```
1  <maintaingoal name="maintainbatteryloaded">
2    <!-- Omitted conditions for brevity. -->
3    <deliberation>
4      <inhibits ref="performlookforwaste" inhibit="when_in_process"/>
5      <inhibits ref="achievecleanup" inhibit="when_in_process"/>
6      <inhibits ref="performpatrol" inhibit="when_in_process"/>
7    </deliberation>
8  </maintaingoal>
9  <achievegoal name="achievecleanup" retry="true" exclude="when_failed">
10   <parameter name="waste" class="Waste" />
11   <!-- Omitted conditions for brevity. -->
12   <deliberation cardinality="1">
13   <inhibits ref="performlookforwaste"/>
14   <inhibits ref="achievecleanup">
15     $beliefbase.my_location.getDistance($goal.waste.getLocation())
16     &amp;lt; $beliefbase.my_location.getDistance($ref.waste.getLocation())
17   </inhibits>
18   </deliberation>
19 </achievegoal>
```

Abbildung 4.12.: Goal Deliberation in Jadex

Der Mechanismus, mit dem Jadex Deliberation beschreibt, nennt sich *Easy Deliberation* und wird direkt in der AgentDefinition File festgelegt. Dort kann im Deliberation-Tag festgelegt werden, wie viele Ziele dieses Typs gleichzeitig verfolgt werden sollen (z.B. soll ein Saugroboter nur ein einziges Feld gleichzeitig versuchen zu säubern) und es kann auch festgelegt werden, welche anderen Ziele nicht verfolgt werden dürfen, solange dieses Ziel aktiv ist. Für jedes andere Ziel, welches während der Verfolgung des aktuellen Zieles pausieren soll, kann ebenso eine Bedingung festgelegt werden, unter der das Ziel pausieren soll.

Goal Deliberation gibt es in Jason nur auf den zweiten Blick, da mangels konkreter Zieldefinitionen andere Wege gefunden werden müssen. In Jason hat der Entwickler die Möglichkeit, während eines Planes bestimmte Ziele bzw. Ereignisse zu blockieren. So wird beim Aufruf von $.suspend(clean)$ die gesamte Menge an Intentionen durchsucht, die der Agent aktuell besitzt, ob der Auslöser bzw. das Ereignis für einen der Pläne, die auf dem Stack dieser Intention liegen, ein $+!clean$ war und die gesamte Intention an der Stelle, wo sie gerade war, unterbrochen. Interessanterweise können auf diese Weise nur Intentionen und Pläne unterbrochen werden, welche bereits aufgetreten sind. Tritt nach dem $.suspend(clean)$ erneut das Ereignis $+!clean$ auf, wird dieses nicht pausiert, sondern es muss in den Contextconditions der Pläne geprüft werden, ob das Ziel aktuell verfolgt werden soll. Für diesen Fall gibt

```
1
2  !walk(1, 2).
3
4  +!walk(X, Y) <-
5    .suspend(clean);
6    // do walking
7    .resume(clean).
8
9  @cleanPlan[atomic]
10 +!clean <-
11   // do cleaning
```

Abbildung 4.13.: Die Declarative Goal Direktive in Jason

es mit *.suspend* (ohne Parameter) die Möglichkeit, sich selbst zu pausieren, was aber immer mit Vorsicht zu genießen ist, da der Agent sich auf diese Weise schnell in eine Art toten Zustand versetzen kann, wenn Pläne sich selbst pausieren, aber keiner sie mehr aufweckt. Im Gegensatz zu Jadex gibt es bei Jason nicht die Möglichkeit, die parallelen Instanzen eines Zieltyps zu begrenzen (z.B. nur ein *walk*-Ziel gleichzeitig), jedoch kann in Jason ein Plan mit der *atomic*-Annotation dafür gesorgt werden, dass diesen während seiner Ausführung kein Ereignis oder Plan unterbricht.

4.2.4. Zusammenfassung

Bei den Zieldefinitionen sieht man ebenso klare Unterschiede zwischen den beiden Frameworks, wie schon zuvor bei der Beliefbase. Während Jadex in der Agent Definition File möglichst viele Typen, Attribute und Parameter definiert, um so gut wie jede Eventualität abzudecken, setzt Jasons Agentspeak auf extreme Einfachheit. Den vier umfangreichen Typen in Jadex treten in Jason nur zwei sehr simpel gehaltene Typen (Achieve- und Testgoal) entgegen, wobei das Achievegoal faktisch nichts weiter als ein Funktionsaufruf ist und alle weiteren Funktionalitäten vom Entwickler nachimplementiert werden müssen. Diesem wird durch einige verfügbare Direktiven zwar unter die Arme gegriffen, jedoch wirken diese sehr improvisiert und nicht durchdacht. Vor allem das Fehlen des Maintaingoals mit der Maintain-Bedingung fehlt hier dem Jason-Entwickler, da die Maintenance-Direktive nur geringe Abhilfe schafft, sie prüft nur auf das Vorhandensein eines Faktes, logische Formeln mit Grenzwerten sind hingegen nicht möglich.

Auch wenn ereignisorientiere Softwaresysteme oft viele Vorteile bei der Skalierung und Entkopplung von Komponenten haben, wird dieses Konzept in Jason nur mangelhaft umgesetzt. Wo es bei Achievegoals noch sinnvoll und sogar notwendig ist, dass nur ein Plan aufgerufen wird, ist es bei den Beliefänderungsereignissen ein großer Nachteil, dass der Entwickler

	Jadex	Jason
Achievegoal	als konkreter Typ modelliert	nicht direkt vorhanden, kann mit Direktiven aus Performgoal nachgebaut werden
Maintaingoal	als konkreter Typ modelliert	nicht direkt vorhanden, kann mit Direktiven aus Performgoal nachgebaut werden, Umsetzung jedoch lückenhaft
Performgoal	als konkreter Typ modelliert	als konkreter Typ modelliert
Ziele aus Ereignissen heraus verfolgen	Conditions mit Observer-Pattern	Ziele sind Ereignisse und haben direkt Pläne zur Folge
Zeitplanung für Ziele	umfangreiche Timer vorhanden	nicht in Zieldefinitionen, nur in Plänen (.wait, .at)
Goal Deliberation	Umfangreiche Definition von Abhängigkeiten und Limitierungen in Zielen	nicht in Zielen, nur in Plänen (.suspend, .resume)

Tabelle 4.2.: Zusammenfassung: Vergleich der Ziele

nicht an beliebigen Stellen im Programm auf beliebige Ereignisse warten und darauf reagieren kann, sondern dass jedes Ereignis nur einen einzigen Plan zu Folge hat.
Jadex löst dieses mit dem klassischen Observer-Pattern deutlich geschickter, wobei dann aber hier darauf geachtet werden muss, dass der Agent nicht aus Versehen zu viel macht. Auch die Zeitsteuerung von Zielen und die Möglichkeit einen Plan bis zum Auftreten eines bestimmten Ereignisses pausieren zu lassen, ermöglichen in Jadex mehr Modellierungsmöglichkeiten, wo hingegen die Zeitsteuerung in Jason nur sehr primitiv ausgelegt ist, für die meisten Fälle jedoch ausreichen mag.

Goal Deliberation ist in Jadex Aufgabe der Ziele oder im Spezialfall Aufgabe des Planes eines Metagoals und kann auch dort in der ADF sehr genau geregelt werden. Jason leitet die Aufgabe der Goal Deliberation an die Pläne weiter und stellt dort teilweise unzureichende Mittel zur Verfügung. Dass die suspend-Funktion dazu führt, dass nur bereits aufgetretene Ereignisse pausieren, nicht aber nach dem Suspend auftretende Ereignisse bis zum Resume zurück gehalten werden, erfordert vom Entwickler die Implementierung weiterer Mechanismen, um dort die Pläne an der Ausführung zu hindern.

4.3. Pläne

Pläne sind in BDI die eigentlichen Arbeitstiere. Nachdem sich der Agent durch Abgleich seiner Wissensbasis und mit Hilfe verschiedener Zielauswahlen ein Ziel gesucht hat, ist die Planbibliothek an der Reihe und hat die Aufgabe, den passenden Plan zu suchen und auszuführen. Der Plan selber führt dann verschiedenste Aktionen aus, um den Agenten dem Ziel näher zu bringen oder das Ziel im Idealfall zu erfüllen.

Die erste Aufgabe der Planungskomponente ist es also erst einmal, einen passenden Plan zu finden. Dafür gibt es praktisch nur zwei Möglichkeiten der Realisierung: Entweder ein Ziel kennt direkt eine Menge der Pläne, welche es erfüllen oder das Ziel formuliert nur das Problem und die Pläne wissen, ob sie das Problem lösen können. Idealerweise gibt es für beide Varianten die Möglichkeit, dass ein Ziel von verschiedenen Plänen erfüllt werden kann, welche sich jedoch vom benötigten Aufwand und dem vorausgesetzten Kontext unterscheiden, damit der Agent in jeder Situation eine Handlungsmöglichkeit besitzt.

Da in einem Plan die gesamte Geschäftslogik eines Agenten stattfindet, wird für dessen Beschreibung möglichst eine turingvollständige Sprache benötigt, um alle vorhandenen Probleme lösen und das Ziel erfüllen zu können. Zusätzlich zu diesen benötigten syntaktischen und semantischen Sprachkonstrukten können die elementaren Aktionen eines Agenten in folgende aufgeteilt werden:

- Zugriff auf die Beliefbase (lesend und schreibend)
- Erstellung weiterer Unterziele und warten auf diese
- Senden von Nachrichten an anderen Agenten
- Ausführen von umweltverändernden Aktionen
- Warten auf Ereignisse, z.B. die Antwort eines anderen Agenten
- Beeinflussung der Nebenläufigkeit von Plänen/Zielen innerhalb des Agenten

In Kapitel 4.2 haben wir gelernt, dass Agenten mehrere Ziele gleichzeitig verfolgen müssen und Jadex und Jason dies auch tun. Wie in jedem Programm, welches mehrere Threads parallel ausführt, muss das BDI-System dafür sorgen, dass sich die Pläne bzw. Threads nicht untereinander ungewollt beeinflussen, da solche Nebenläufigkeitseffekte sehr schwer nachzuvollziehen und zu beheben sind. Der Agent muss also die Möglichkeit haben, einige externe Ressourcen exklusiv für sich sperren zu können.

Ein Problem in jedem Programm sind Fehler. Während der Ausführung eines Planes kann immer etwas Unvorhergesehenes passieren oder es können kleinere Programmierfehler zu Planabstürzen führen. In solchen Fällen darf der Agent aber nicht plötzlich handlungsunfähig

werden, sondern muss diese Fehler gesondert behandeln und andere Wege suchen können, z.B. einen Ausweichplan ausführen.

Unsere Analyse wird sich also auf folgende Punkte beziehen:

- Definition und Struktur
- Planauswahl
- Planabbruch
- Programmiersprache der Pläne
- Aktionsumfang innerhalb der Pläne
- Parallelität und Nebenläufigkeit

4.3.1. Definition und Struktur

Jadex

```
1   <plans>
2     <plan name="consumeFood" exported="true">
3       <body class="ConsumeFoodPlan" />
4       <trigger>
5         <goal ref="achieve_consumeFood" />
6       </trigger>
7     </plan>
8     <plan name="walk" exported="true">
9       <body class="WalkingPlan" />
10      <trigger>
11        <goal ref="perform_walk" />
12      </trigger>
13    </plan>
14  </plans>
```

Abbildung 4.14.: Pläne in Jadex

In Jadex bestehen Pläne aus zwei Teilen: Einmal wird in der ADF der Kopf des Planes definiert und zum anderen besitzt jeder Plan eine eigene Java-Klasse, welche die eigentliche ausführbare Implementierung darstellt.

Im Kopf des Planes wird definiert, wann ein Plan ausgeführt werden soll, wofür es mehrere Typen von Triggern gibt:

- Zieltrigger, welche aktiviert werden, wenn ein Ziel aktiviert wird
- Bedingungen, welche den Plan starten, sobald die Bedingung erfüllt ist

- Beliefänderung (generell Änderung oder nur Hinzufügen oder Entfernen von Fakten), welche den Plan aktivieren, sobald sich das beobachtete Belief entsprechend ändert
- Internal-Event-Trigger, welcher auf interne Ereignisse reagiert
- Message-Event-Trigger, welcher auf Nachrichten von außerhalb reagiert

Diese Trigger können beliebig und in verschiedener Anzahl miteinander kombiniert werden, sodass es auch möglich ist, dass ein einziger Plan auf verschiedenste Ereignisse reagieren kann.

Jeder Plankopf besitzt zudem noch zwei Bedingungen: Die *Vorbedingung* eines Planes muss erfüllt sein, damit der Plan überhaupt gestartet werden kann. So können Pläne, welche nicht in den aktuellen Kontext des Agenten passen, von vornherein ausgeschlossen werden. Die zweite Bedingung, die *Kontextbedingung*, kontrolliert während der Ausführung des Planes, ob dieser noch ausgeführt werden sollte. Sobald die Kontextbedingung nicht mehr erfüllt ist, wird der Plan automatisch abgebrochen.

```
1  public class WalkingPlan extends Plan {
2    ...
3    public WalkingPlan() {
4      super();
5    }
6
7    public void body() {
8      // get me & my current position
9      me = (Agent) getBeliefbase().getBelief("me").getFact();
10     try {
11       myPos = Environment.getInstance().getObject(me.getID()).getPos();
12     } catch (NullPointerException npe) {
13       fail();
14       return;
15     }
16     // get vision
17     vision = (Vision) getBeliefbase().getBelief("vision").getFact();
18
19     nextMovePos = null;
20     currentVision = vision.getExpectedVision(myPos);
21     // [...] calculate next position to move to and do the move
22     waitFor(100);
23   }
24 }
```

Abbildung 4.15.: Planimplementation in Jadex

Der zweite Teil eines Planes ist nun die eigentliche Implementierung in Java. Hierfür wird eine Unterklasse der Jadex-Klasse *Plan* erstellt, welche vier wichtige Methoden zum Überschreiben anbietet.

- **body:** Die eigentliche Ausführungsmethode des Planes. Sie wird aufgerufen, wenn der Plan gestartet werden soll

- **passed:** Diese Methode wird aufgerufen, sobald der Plan erfolgreich abgeschlossen wurde
- **failed:** Die Fehlermethode wird aufgerufen, wenn ein Plan fertig ausgeführt wurde, aber das Ziel trotzdem nicht erfüllt ist.
- **aborted:** Wird ein Plan von außerhalb abgebrochen, weil z.B. die Kontextbedingung nicht mehr erfüllt ist, wird diese Methode aufgerufen

Die letzten drei Methoden dienen generell dazu, Planresourcen wieder aufzuräumen und im Fehlerfall Inkonsistenzen zu beseitigen (vor allem bei der failed- und der aborted-Methode). Die Fehlerbehandlung innerhalb eines Planes (Exceptions, Errors) kann mit den Java-eigenen Mitteln wie try-catch-Blöcken erreicht werden.

Zusätzlich zum normalen Plan gibt es in Jadex noch eine zweite Plan-Klasse: Den Mobile-Plan. Der Vorteil dieses Plantyps ist, dass er es Agenten erlaubt, auch während der Ausführung des Planes auf andere Systeme zu migrieren. Möglich wird dies dadurch, dass der Plan keine body-Methode besitzt, welche beim Aufruf der diversen waitFor-Methoden (Unterziele, Ereignisse) blockiert, sondern eine Methode $action(IEventevt)$, welche bei jedem den Plan betreffenden Ereignis aufgerufen wird. Dadurch dass die waitFor-Methoden nie blockieren und die action-Methode direkt an dieser Stelle zurück gibt, taucht der Plan zwischen den einzelnen Schritten nie auf einem Stack auf und kann so auch zwischen verschiedenen Systemen verschickt werden. Der Nachteil bei dieser Variante der Pläne ist, dass sie durch die Asynchronität deutlich schwieriger zu programmieren sind, da immer der aktuelle interne Zustand des Planes festgehalten werden muss.

```
1  +!walk :
2        me(ID)
3     & worldObject(ID, Name, Type, State, Position)
4    <-
5     !calcTargetPosition(Position, TargetPosition);
6     de.lunaticsoft.agent.jason.actions.time_millis(Now);
7     +marker(Position)[expireTime(Now+10000)];
8     sendMoveMsg(Position, TargetPosition);
9     .at("now +1000 milliseconds", "+!walk").
10
11 -!walk :
12    true
13   <-
14    .at("now +1000 milliseconds", "+!walk").
```

Abbildung 4.16.: Pläne in Jason

Pläne in Jason bestehen aus drei Teilen.
Der erste Teil eines Planes ist seine Signatur, welche den Ereignistyp ($+$, $-$, $+!$, $-!$, $+?$, $-?$), den Ereignisnamen, eine Liste der Parameter und auch die Annotationen des Ereignisses umfasst. Bei den letzten drei Teilen wird mit Unifikation gearbeitet, d.h. ist eine Variable der

Parameterliste bereits an einen Wert gebunden, muss der übergebene Wert dem vorgegebenen Wert entsprechen. Ein Plan $+!attack(Enemy, knife)$ wäre also nur benutzbar, wenn das zweite Argument des Ereignisses $knife$ ist.

Der zweite Teil eines Planes ist die Kontextbedingung. Diese ist eine logische Formel, welche vom Interpreter versucht wird, zu erfüllen. Meist wird hierbei auf die Beliefbase des Agenten zurück gegriffen und geprüft, ob bestimmte Beliefs exisitieren, welche für die Ausführung des Planes nötig sind. Kann die Kontextbedingung nicht erfüllt werden, weil eine der versuchten Aktionen *false* zurückgibt, wird der Plan verworfen.

Der dritte und wichtigste Teil ist der Plankörper. Dieser enthält die eigentlichen Aktionen des Planes, wo sich auch der größte Unterschied zu Plänen in Jadex befindet. Pläne werden in Agentspeak ebenfalls deklarativ dargestellt und programmiert, was zur Folge hat, dass der gesamte Plankörper ebenfalls genau genommen eine einfache logische Formel ist. In Agentspeak besitzt jede Regel, Aktion oder Ziel einen booleschen Rückgabewert. Beliefbase-Regeln geben true zurück, wenn ein entsprechendes Belief gefunden wurde, ansonsten false, interne Aktionen sowie Environment-Aktionen geben ebenfalls im Erfolgsfall true zurück. Ziele geben bei vollständiger Beendigung true zurück, ansonsten ebenfalls false. Im Plankörper muss jede Anweisung erfolgreich sein (also true zurück geben), damit der Plan fortgesetzt wird. Sobald die erste Anweisung false ergibt, schlägt der Plan fehl und es wird ein Fehler-Ereignis generiert ($-g$ zu $+g$, $-!g$ zu $+!g$, $-?g$ zu $+?g$), welches wiederum durch Pläne behandelt werden kann. Plankörper bestehen in Agentspeak also nur als Anweisungen, was zur Folge hat, dass es keinerlei Kontrollstrukturen gibt. Sämtliche If-then-else-, while-do- oder foreach-Blöcke werden deklarativ durch Unifikation und Rekursion gelöst, was dazu führt, dass in Jason deutlich mehr Ziele und Pläne existieren als in Jadex, da jede Iteration und jede Verzweigung in neuen Plänen oder Regeln dargestellt werden muss.

4.3.2. Planauswahl

Im ersten Teil dieses Kapitels sprachen wir davon, dass ein Ziel durchaus von mehreren Plänen erfüllt werden kann und sich diese oft vom aktuellen Kontext und von ihren Kosten unterscheiden, sodass es nötig wird, eine Planauswahl einzuführen.

Damit in Jadex ein Plan ausgeführt wird, sucht der Agent nicht aktiv nach einem Plan, welcher das Ziel erfüllen kann, sondern „ruft“ quasi in den Raum hinein, dass er nun folgendes Ziel verfolge und die Pläne reagieren mit ihren Triggern darauf, sodass die Planauswahl nicht durch direkte Auswahl sondern eher indirekt durch passende Trigger erfolgt. Hierbei kann jeder Plan eine Start- und eine Kontextbedingung haben, welche beide erfüllt sein müssen, damit der Plan in Frage kommen kann. Durch diese Bedingungen kann der Entwickler die

Pläne so konstruieren, dass möglichst bei jeder Situation nur ein Plan alle seine Bedingungen erfüllt und ausgeführt werden kann.
Etwas schwieriger wird es, wenn mehrere Pläne durch ein Ereignis getriggert werden und bei mehr als einem die Startbedingung erfüllt ist (also dass es für ein Ereignis und den aktuellen Kontext zwei passende Pläne gibt). Dann kann der Agent entweder in eher zufälliger Reihenfolge alle Pläne ausprobieren, bis der erste erfolgreich ist oder er fragt in einem Zwischenschritt ein Metagoal. Das Metagoal bekommt alle passenden Pläne in speziellen Parametern übergeben und wählt einen davon aus, welcher dann ausgeführt wird.

Bei Jason sieht dieses Problem deutlich einfacher aus. Durch die Signatur der Pläne können sehr schnell alle passenden Pläne gefunden werden und durch die Kontextbedingung werden alle Pläne aussortiert, welche aktuell nicht passen. Auch hier bleibt im Optimalfall nur ein einziger Plan übrig, welcher ausgeführt wird. Sind es doch mehrere Pläne, welche in Signatur und Kontextbedingung passen, kommt die Agenteninfrastruktur in Java zum Einsatz, welche nun selbst einen Plan auswählt. Die Standardimplementation wählt einfach den Plan aus, der zuerst im Quellcode steht, jedoch ist diese austauschbar. Es wäre z.B. denkbar, dass Pläne mit einer $priority(X)$ Annotation versehen werden, wo der Agent dann den Plan mit der höchsten Priorität auswählt.

4.3.3. Funktionsumfang

Damit der Agent innerhalb seiner Pläne genügend Handlungsfreiraum hat, benötigt er einen genügend großen Satz an Systemfunktionen, welche hier tabellarisch miteinander verglichen werden sollen:

Jadex	**Jason**
Zugriff auf Beliefbase	
Java-Bean-Methoden, OQL	hinzufügen, entfernen, suchen mit logischen Formeln
Erstellung weiterer Unterziele	
möglich, warten optional	möglich, warten optional
Senden von Nachrichten an andere Agenten	
Nachrichten nach FIPA-Protokoll	Nachrichten nach KQML-Protokoll, automatisch es Abfragen anderer Beliefbases und Planbibliotheken, kann Pläne verschicken; Kommunikation ist synchron
Ausführen von umweltverändernden Aktionen	
kein explizites Environmentobjekt, muss in Java nachgebaut werden, da beliebige Funktionen	Literale an Environmentobjekt senden, dort in Java beliebige Aktionen möglich

Warten auf Ereignisse	
diverse waitFor-Methoden im Plan, warten auf Ereignisse, Pläne, Ziele und Bedingungen möglich	explizites Warten nur bei Erstellung von Unterzielen, sonst nicht möglich
Beeinflussung von Nebenläufigkeit und Planausführung	
mit waitFor warten auf Ereigmisse möglich, atomare Blöcke	atomare Pläne, diverse Funktionen, um aktuelle Ziele und Pläne zu beeinflussen
Erweiterbarkeit	
Erlaubt ist alles, was Java kann	Einbau eigener interner Aktionen durch Interface, Implementation in Java; dort Zugriff auf komplette Agentenstruktur (BB, Pläne, Ziele etc.)

4.3.4. Nebenläufigkeit

Dem Thema Nebenläufigkeit begegnen Jadex und Jason auf ähnliche Weise. Beide Systeme lassen es zwar zu, dass Agenten mehrere Ziele und auch die entsprechenden Pläne gleichzeitig haben kann, allerdings wird per Definition nur ein einziges Ziel gleichzeitig ausgeführt. Dafür implementieren beide eine Art Scheduler, welcher die gerade aktiv ausgeführten Pläne abwechselnd unterbricht und ausführt.
Da Jadex-Pläne in Java verfasst sind, ist es für den Jadex-Scheduler schwierig, diese einfach während der Ausführung zu unterbrechen[6], sodass dort ein Trick angewendet werden muss: Sobald der Plan eine Methode des Jadex-Frameworks ausführt (z.B. waitFor(), Beliefänderung, Unterziele), gibt der Plan automatisch die Kontrolle an den Scheduler ab, welcher dann erneut entscheidet, welcher Plan als nächstes an die Reihe kommt. Dabei kann es sein, dass Pläne in einen blockierten Zustand kommen, wenn sie z.B. auf Ereignisse warten (z.B. ein Unterziel), sodass sie vorerst vom Scheduler ausgeschlossen werden und erst durch das Ereignis wieder in den *ready*-Zustand gelangen. Damit das ständige Wechseln zwischen Scheduler und Plänen nicht in einem Stackoverflow enden (während der Blockade verlässt der Plan seine body-Methode nicht), befindet sich jeder Plan in einem eigenen Thread, welcher vom Scheduler über einen Monitor benachrichtigt wird, wenn er an der Reihe ist und den Monitor bei jedem Unterbrechungspunkt wieder freigibt.
Jasons Agentspeak als Interpretersprache hat da einige Vorteile. Da der Agentspeak-Plan direkt während der Ausführung erst interpretiert wird, kann der Interpreter die Ausführung direkt beeinflussen. So sind auch in Jason die Pläne in einzelne Anweisungen unterteilt, aber dadurch, dass die einzelnen Anweisungen in Echtzeit interpretiert werden, kann die

[6] Java-Threads können nicht innerhalb der JVM von einem anderen Thread unterbrochen werden, sondern müssen ihre Kontrolle von sich aus abgeben

Agenteninfraktruktur nach jeder Anweisung den Plan wechseln oder andere Aktionen ausführen. In der Standardimplementation wechselt der Agent nach jeder Anweisung den Plan im Round-Robin-Modus, jedoch kann diese leicht verändert werden. So ist es z.B. möglich, Pläne mit einer *Priotity*-Annotation zu versehen, welche der Eigenbau-Scheduler benutzt, um zwischen Plänen zu wechseln.

In beiden Frameworks ist es möglich, Sperren zu errichten. Diese beziehen sich dabei jedoch nicht auf einzelne Variablen oder Komponenten sondern immer auf den gesamten Agenten. So können mit Hilfe der Methode $startAtomic()$ und $endAtomic()$ atomare Blöcke in Jadex erzeugt werden, sodass z.B. eine Reihe von Beliefs verändert werden können, ohne dass der Plan durch PropertyChangeEvents unterbrochen wird. In diesen Blöcken ist es logischerweise nicht möglich, mit den waitFor-Methoden auf andere Ereignisse zu warten. In Jason können nur ganze Pläne mit der $atomic$-Annotation versehen werden, wodurch sie ununterbrechbar werden. Die Sperre bezieht sich hierbei auf den gesamten Stack, den ein Plan erzeugt. Erzeugt ein atomarer Plan während der Ausführung Ereignisse (Unterziele, Beliefänderungen), werden diese Ereignisse direkt auf den aktuellen Planstack gelegt und ebenfalls atomar ausgeführt.

Auch wenn es sowohl in Jadex als auch in Jason theoretisch möglich wäre, die durch Java gegebenen Möglichkeiten betreffend Nebenläufigkeiten zu nutzen (Monitore, Semaphoren), wäre der Agent bei deren Benutzung nicht mehr funktionsfähig, da dies den Aufruf des Schedulers verhindert. In Jadex müssen die einzelnen Plan-Threads ihre Kontrolle selbst abgeben, was durch die Blockade nicht mehr möglich wäre. In Jason hat der Agent sowieso nur einen Thread, welcher bei Benutzung von Monitoren oder Semaphoren in einer endlosen Blockade landen würde.

Dadurch, dass Jason Goal-Deliberation komplett auf die Pläne auslagert, gibt es auch noch die in Abbildung 4.13 gezeigten Methoden, um innerhalb von Plänen andere Pläne bzw. deren Startereignisse zu pausieren oder sie mit weiteren Funktionen komplett zu entfernen.

4.3.5. Zusammenfassung

Die Plankomponenten in Jadex und Jason sind beide recht ähnlich in ihrer Funktionsweise und im Funktionsumfang. Der wahrscheinlich herausragendste Unterschied ist die Programmiersprache der Plankörper, welche bei Jadex pures Java ist, in Jason hauptsächlich das deklarative Agentspeak zum Zuge kommt, welches jedoch durch Java-Methoden erweitert werden kann. Dort liegt auch eine kleine Design-Unzulänglichkeit. Theoretisch ist es möglich die kompletten Plankörper in Java zu schreiben und im Agentspeak-Plan nur die implementierte Java-Funktion aufzurufen, was jedoch von den Entwicklern von Jason sicherlich nicht so gewollt ist. Es ist hier dem Entwickler selbst überlassen, welche Teile er in Agentspeak und welche er in Java realisiert.

	Jadex	Jason
Definition & Struktur	Kopf in XML, Körper in Java	komplett in Agentspeak
Planauswahl	durch Start- und Kontextbedingung oder Metagoal	durch Kontextbedingung
Planfehler	Exceptions in Java abfangen, diverse „Aufräummethoden“	Fail-Ereignis + entsprechende Pläne
Programmiersprache der Pläne	Java, imperativ	Agentspeak, deklarativ, aber Funktionsbibliothek in Java erweiterbar
Aktionsumfang	siehe Kapitel 4.3.3	siehe Kapitel 4.3.3
Parallelität und Nebenläufigkeit	keine echte Paralellität, atomare Blöcke	keine echte Paralellität, atomare Blöcke

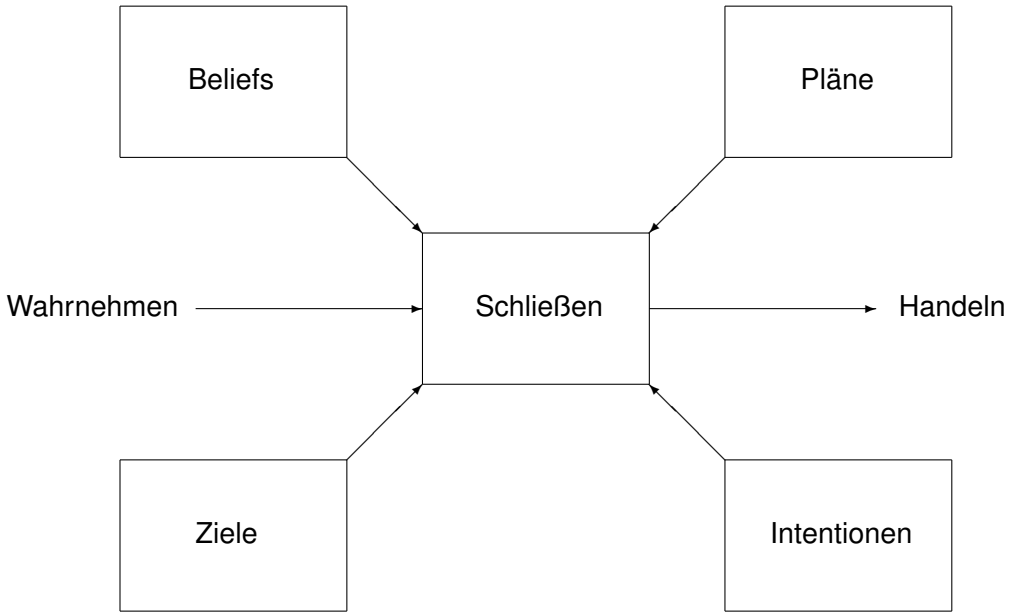
Tabelle 4.4.: Zusammenfassung: Vergleich der Pläne

Auf der anderen Seite findet sich ein ähnlicher Punkt in Jadex. Während in Jason die Planauswahl ausschliesslich auf die Pläne selbst ausgelagert ist und Ziele ebenfalls nur explizit erstellt werden, haben in Jadex sowohl Ziele als auch Pläne Start- und Kontextbedingungen, die beide den Zweck der Planauswahl erfüllen können. Programmiertechnisch ist es immer schlecht, einen Sachverhalt an zwei verschiedenen Stellen regeln zu können, da dies gefährliche Fehlerquellen sein können.

4.4. Reasoning Cycle

Jeder Agent in einem Agentensystem besitzt eine Art von Reasoning Cycle, eine Programm-Schleife in welcher der Agent aus seinen Wahrnehmungen schließt, welche Aktion er als nächstes ausführen möchte, was einfach dargestellt wie folgt aussieht.

Bei einfachen Agenten, wie reaktiven oder modellbasierten Agenten, sind die Reasoning-Funktionen sehr simpel aufgebaut, sodass sie meist in einer Tabelle einen Eingangswert auf einen Ausgangswert abbilden. Bei kognitiven Agenten sieht das dagegen etwas komplizierter aus, weil diese intelligenter sind und weitere Komponenten eine Rolle spielen:



Bei jeder Wahrnehmung, die ein BDI-Agent bekommt, muss er diese zuerst mit seinen Beliefs und Zielen abgleichen, Pläne finden und diese wiederum mit dem Intention Stack abgleichen, bis er sich einen Plan aussucht, von welchem er sich die nächste Aktion heraussucht und ausführt. Die Schritte, die ein Agent ausführen muss, sehen im Detail so aus:

1. Wahrnehmung abfragen und daraus Ereignisse generieren
2. Nach Plänen suchen, die auf das neue Ereignis passen -> relevante Pläne
3. Aus diesen Plänen diejenigen auswählen, welche zum aktuellen Kontext passen -> aktive Pläne
4. aus den aktiven Plänen einen auswählen, welcher zu einer wirklichen Intention wird
5. den gewählten Plan in die Liste der Intentionen einfügen
6. eine Intention auswählen und die nächste Aktion dieser Intention ausführen

Wie wir schon in Kapitel 4.3.4 festgestellt haben, arbeiten die Agenten also per Design niemals multi-threaded sondern generell sequenziell[7]. Das ist notwendig, weil jedes Ereignis, welches der Agent wahrnimmt, die aktuellen Ziele und deren Pläne verändern kann. Werden Aktionen ausgeführt, während der Agent gerade wahrnimmt, kann es passieren, dass Aktionen auf inkonsistenten Wahrnehmungen basieren oder sich Pläne, Ziele und Intentionen ebenfalls in inkonsistenten Zuständen befinden.

[7] Für Informationen zu multi-threaded BDI-Systemen siehe [21]

4.4.1. Jadex

Aufgrund der Tatsache, dass Jadex keine explizite Umwelt hat, in der der Agent Wahrnehmungen erfahren kann, ist der Reasoning Cycle in Jadex etwas anders aufgebaut bzw. er existiert gar nicht.

Die Ausführung eines Jadex-Agenten basiert nicht auf der festen Struktur *Wahrnehmen->Schließen->Handeln*, sondern auf einer sogenannten Agenda. In dieser Agenda, welche, einfach gesagt, eine schlichte Menge von Codeblöcken ist, ist die gesamte Logik des Agenten abgebildet. In Jadex ist sämtliche Logik des Agenten in viele kleine Stückchen aufgeteilt, die in diese Agenda eingereiht und nacheinander ausgeführt werden. Diese kleinen Codestückchen oder *AgendaActions*, wie sie konkret heißen, können Pläne sein, die bis zum nächsten Unterbrechungspunkt ausgeführt werden, die Behandlung einer eingehenden Nachricht, die Erstellung oder auch Löschung eines Zieles oder ähnliches (aktuell gibt es über 40 solcher Agenda-Aktionen). Die komplette BDI-Logik findet dementsprechend auch nur innerhalb dieser Aktionen statt und ist so über eine Vielzahl von Klassen verteilt und wird ungeordnet und nicht in einer festen Reihenfolge ausgeführt.
Da oft eine Vielzahl von Aktionen in der Agenda auf die Ausführung warten, kann es passieren, dass sich der Zustand des Agenten insofern geändert hat, dass die Aktion keinen Sinn mehr macht, weil sie auf veralteten Daten beruht. Damit die Aktion in solchen Fällen nicht ausgeführt wird, besitzt jede Agenda-Aktion eine Validierungsfunktion, welche vor ihrer endgültigen Ausführung aufgerufen wird und die Aktion verwirft, sollte sie nicht mehr zutreffend sein.

Problematisch dabei ist nun, dass der erste Schritt im klassischen Reasoning-Cycle, das Wahrnehmen, komplett unter den Tisch fällt. Jadex besitzt kein zentrales Konzept, wie die Sensoren eines Agenten darzustellen sind und wie sie sich in das Reasoning einfügen. Scheinbar gewollt ist, dass die einzige Wahrnehmungsschnittstelle des Agenten die Kommunikation ist, denn diese ist durch Agenda-Aktionen abgedeckt. Werden jedoch andere Sensoren benötigt, welche nicht auf die Kommunikationsschnittstelle zurückgreifen, sind unsaubere Methoden gefragt, welche die Beliefs von außerhalb des BDI-Agenten ändern, sodass dieser darauf reagiert.
Dieser Fall tritt konkret bei unserem *Hunters and Deers*-Beispiel auf. Dort stellt ein spezieller externer Darkstar-Game-Server die Umwelt dar und verwaltet alle existierenden Objekte mit ihren Eigenschaften. Die Agenten bauen selbst eine Verbindung zu diesem Server auf und werden darüber über Veränderungen in der Umwelt informiert. Diese Veränderungen werden dann asynchron und komplett unabhängig zur aktuellen Agenda in die Beliefbase eingetragen.

Jadex folgt mit dieser Agenda nicht dem klassischen BDI-Reasoning sondern verlagert die einzelnen Stufen in verschiedenste einzelne Komponenten, um die Ausführung des Agenten dynamischer gestalten und die Kopplung der Komponenten gering halten zu können. Die

einzige, noch dem BDI-Konzept entsprechende Aktion ist die *ExecutePlanStepAction*, welche in einem Plan die nächste Aktion ausführt. Diese Agenda-Aktion wird sowohl zu Beginn eines Planes in die Agenda eingefügt als auch jedes Mal, wenn ein noch nicht beendeter Plan seine Kontrolle an den Scheduler übergibt. Dadurch, dass die Standardimplementation der Agenda immer die erste valide Aktion ausführt, ergibt sich für die Pläne eine Art FILO-Warteschlange[8], sodass die Pläne im Round-Robin-Modus ausgeführt werden.

4.4.2. Jason

Jason auf der anderen Seite orientiert sich bei so gut wie allen Belangen an den theoretischen Grundlagen des BDI-Systemes, so auch beim Reasoning Cycle.

Der Reasoning-Cycle von Jason teilt sich jedoch nicht in sechs sondern in zehn einzelne Schritte auf (vgl. [4, S. 66f]).

1. **Die Umwelt wahrnehmen** Im ersten Schritt befragt der Agent das Environment-Objekt nach neuen Wahrnehmungen. Die Quellen oder Sensoren des Environments können sehr vielfältig sein. In den meisten Fällen existiert dort eine Art Weltmodellierung, welche einen Gesamtweltzustand darstellt, also eine Art *Ground Truth*. In diesem Fall erstellt das Environment die Wahrnehmungen des Agenten intern, jedoch kann es, da es in Java geschrieben ist beliebige Systemschnittstellen als Sensoren benutzen, um die Wahrnehmung des Agenten zu ermitteln.
2. **Beliefbase aktualisieren** Hat der Agent seine neuen Wahrnehmungen erhalten, vergleicht er sie mit dem, was er im letzten Zyklus wahrgenommen hat und aktualisiert seine Beliefs dementsprechend, er fügt also neue Wahrnehmungen hinzu und entfernt alte Wahrnehmungen, welche vom Environment nun nicht mehr gemeldet werden. Dabei wird bei jeder Änderung ein externes Ereignis generiert und in die Liste der Ereignisse eingefügt. Jede Wahrnehmungsänderung ruft vor ihrem tatsächlichen Eintreten noch einmal die Belief-Revision-Funktion auf, welche, sofern der Agent dort Logik implementiert hat, weitere Änderungen zur Folge hat.
3. **Nachrichten von anderen Agenten empfangen** Im nächsten Schritt wird die *Mailbox* des Agenten nach neuen Nachrichten geprüft, wobei in jedem Zyklus nur eine einzige Nachricht abgeholt wird. Die Auswahlfunktion, welche die Nachricht auswählt, kann vom Entwickler überschrieben werden, sodass auch hier die Vergabe von Prioritäten möglich ist.
4. **Nachricht prüfen** Bevor eine Nachricht von einem anderen Agent tatsächlich behandelt wird, wird sie zuvor durch eine Art Spamfilter geschickt, welcher prüft, ob die

[8] First-In-Last-Out

Nachricht überhaupt angenommen werden soll. Auch hier kann der Entwickler die Standardimplementation überschreiben, um seine eigene Logik zu implementieren. Hat die Nachricht diesen Filter passiert, wird aus ihr ein externes Ereignis generiert.

5. **Ereignis auswählen** Nun wird aus allen Ereignissen, welche in diesem Zyklus neu wahrgenommen wurden oder noch aus früheren Zyklen stammen, eines ausgewählt. Wie an vielen anderen Stellen des Reasoning-Cycles kann auch diese Auswahlfunktion vom Entwickler überschrieben werden.

6. **Mögliche Pläne suchen** Der Agent durchsucht als nächstes, welche Pläne aus seiner Planbibliothek auf das generierte Ereignis passen könnten. Dabei vergleicht der Agent die Signatur des Ereignisses (Typ, Name, Anzahl der Parameter, Annotationen) mit der Signatur aller Pläne und erhält so die Menge aller möglichen Pläne.

7. **Relevante Pläne suchen** Hat der Agent die Menge aller möglichen Pläne ermittelt, untersucht er die Kontextbedingung der Pläne, ob der Plan zum aktuellen Agentenkontext passt. Alle Pläne, die diese Prüfung bestanden haben, nennt man relevante Pläne.

8. **Plan auswählen** Im nächsten Schritt wählt der Agent aus dieser Menge an relevanten Plänen einen aus, welcher vom Agenten zur Ausführung in Betracht gezogen werden soll. Auch hier kann die Auswahlfunktion wieder beliebig angepasst werden. Der gewählte Plan wird im Falle eines internen Ereignisses auf den Stack des Auslösers gepackt oder in einen neuen Stack eingefügt.

9. **Intention Stack auswählen** Kurz vor dem Ziel des Zyklusses muss der Agent nun eine Intention auswählen, von welcher er die nächste Aktion des obersten Planes ausführen möchte. Es braucht nicht mehr wirklich erwähnt zu werden, dass auch hier die Auswahlfunktion beliebig anpassbar ist.

10. **Ausführung** Schlussendlich wird nun die nächste Aktion des obersten Planes der gewählten Intention ausgeführt und der Zyklus danach beendet.

Der Reasoning-Cycle von Jason setzt also sehr exakt die Theorie um und ist durch diese Strukturierung auch sehr leicht nachzuvollziehen. Während in Jadex das Reasoning auf zig Klassen verteilt ist, konzentriert es sich in Jason auf gerade einmal zwei Klassen (Die Klasse *TransitionSystem* mit dem eigentlichen Reasoning-Cycle und die Klasse *Agent* mit den überschreibbaren Auswahlfunktionen).
Ein leichter Nachteil könnte bei einigen Agenten dadurch entstehen, dass während eines Durchlaufs des Reasoning nur eine einzige Anweisung ausgeführt wird. Dies führt bei aufwändigen Plänen, welche inkl. Unterplänen 100 Anweisungen oder mehr haben, dazu, dass bei jeder Anweisung ein sehr großer Overhead entsteht. Vor allem der Abgleich der neuen

Wahrnehmung mit der bisherigen verbraucht, abhängig von der Menge der Wahrnehmungsliterale, erheblich Rechenzeit, was sich bei vielen Anweisungen eines Planes deutlich bemerkbar macht.
Konkret führte dieses Verhalten bei unseren Hunters-and-Deers-Agenten dazu, dass die Bewegungsberechnung eines Agenten teilweise so lange dauerte, dass sich die Position umgebender Objekte schon längst wieder geändert hat, diese aber dem bereits gestarteten Plan nicht bekannt war, was in der Bewegung teilweise zu starken Rucklern führte.

Da Jason während des Reasonings auf eine zentrale Wahrnehmung zurückgreifen kann, können so auch externe Sensoren wie der Darkstar-Server des Hunters-and-Deers-Programmes sehr einfach eingegliedert werden. Auf der technischen Seite hält das Environment permanent eine Liste aller Wahrnehmungen für jeden Agenten vor, welche auch asynchron zum Reasoning-Cycle verändert werden kann und vom Agenten nur zyklisch abgerufen wird.

4.4.3. Zusammenfassung

Der Reasoning-Cycle ist zumindest aus technischer Sicht bei beiden Systemen sehr unterschiedlich. Während der Zyklus bei Jason, basierend auf den theoretischen Grundlagen, in klare Schritte gegliedert ist, wird die Funktionalität des Reasoning in einzelne Aktionen ausgelagert, welche der Ausführungszyklus nur sehr abstrakt wahrnimmt. Diese Abstrahierung diverser Aktionen gibt Jadex die Möglichkeit seine Ausführung deutlich dynamischer zu bestimmen, jedoch auf Kosten des Environment-Objektes, welches bei Jason eine Art Kontroll-Objekt darstellt, welche vorgibt, was der Agent darf und was nicht. Dies muss in Jadex selbst implementiert und in die Agentenstruktur eingegliedert werden.

Der wirklich signifikante Vorteil liegt jedoch bei Jason mit seiner Fähigkeit, sämtliche Selektions-Funktionen, welche in den einzelnen Schritten aufgerufen werden, selbst zu implementieren und so das Reasoning aktiv im Innersten zu beeinflussen. Das Reasoning von Jadex ist in dieser Hinsicht starr und kann nur durch die Vorgaben aus der ADF beeinflusst werden, wobei sich jedoch im Quellcode von Jadex an verschiedenen Stellen Anmerkungen finden lassen, welche darauf schließen, dass in späteren Versionen unterschiedliche Reasoning-Strategien verfolgt werden können.

Bauen aufwändige Jason-Pläne auf Beliefs auf, die sich sehr häufig ändern, kann es dazu kommen, dass die Pläne auf Basis zu alter Daten handeln. Es muss bei der Entwicklung also entweder auf solch lange Pläne verzichtet werden oder die Pläne müssen während der Ausführung immer wieder die entsprechenden Beliefs prüfen ung ggf. die Ausführung ändern.

4.5. Kommunikation

Auch wenn die einzige offizielle Informationsquelle eines Agenten seine Sensoren sind, wird in der Praxis meist zwischen zwei verschiedenen Quellen unterschieden: Dort bekommt der Agent einmal Informationen über die Umwelt selbst oder wirkliche Sensoren, wie Kameras oder Mikrophone, einen anderen Teil seines Wissens bekommt der Agent jedoch bereits fertig aufbereitet von anderen Agenten. Diese Kommunikation der Agenten ist ein elementarer Bestandteil jedes Multiagentensystemes, welches als verteiltes System zwingend auf Kommunikation angewiesen ist, um zu funktionieren.

Die Ambitionen, wenn ein Agent mit einem anderen Agenten kommunizieren möchte, kann man grob in drei Klassen einteilen:

- Agent A versucht Agent B über neue Sachverhalte zu informieren
- Agent A möchte Agent B nach Informationen fragen bzw. sie abfragen
- Agent A möchte ein Ziel an Agent B delegieren

Kommunikation in Agentensystemen dient meist dazu, sich in kollektiven Prozessen abzustimmen, z.B. Aufgaben oder Resourcen aufzuteilen oder zu reservieren. So könnten z.B. zwei befreundete Agenten ihren jeweiligen Sichtbereich mit dem anderen Agenten teilen oder, z.B. im Falle von Staubsaugerrobotern, sich abstimmen, wer wo nach Dreck sucht.

Eine Problematik bei jeder Art von Kommunikation in der IT, nicht nur bei Agenten, ist die Sicherheit. Nicht alle Agenten in der Welt sind *gute Agenten*. In Kapitel 4.1.4 sprachen wir bereits davon, dass Agenten absichtlich falsche Informationen verteilen können. In der Kommunikation ist jedoch nicht nur das Verteilen falscher Informationen ein Thema, sondern auch sicherheitstechnische Belange wie Zugriffsrechte und Sicherheitslücken. Man stelle sich vor, ein Agent A delegiert ein Ziel an Agent B und dieser nimmt dieses Ziel ohne weitere Prüfung als gegeben hin und versucht es zu erreichen. In einem anderen Fall könnte eine Request-Nachricht von Agent A im schlimmsten Fall automatisch Informationen aus der Beliefbase von Agent B liefern. Das Kommunikationskonzept eines BDI-Agenten muss ein Sicherheitskonzept vorlegen, welches unauthorisierte Zugriffe verhindern kann.

4.5.1. Jadex

Die Kommunikationsschicht in Jadex baut auf dem verbreiteten FIPA-Standard[9] auf. Da Jadex die Kommunikationsschicht weitestgehend abstrahiert, werden wir nicht im Detail auf den Aufbau des FIPA-Protokolls eingehen, uns jedoch die Grundzüge einmal anschauen. Jede

[9] Foundation for Intelligent Physical Agents [10]

```
<events>
  <!-- A query-ref message with content "ping" -->
  <messageevent name="query_ping" type="fipa" direction="receive">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.QUERY_REF</value>
    </parameter>
    <parameter name="content" class="String" direction="fixed">
      <value>"ping"</value>
    </parameter>
  </messageevent>
  <!-- An inform message where content contains the word "hello" -->
  <messageevent name="inform_hello" type="fipa" direction="receive">
    <parameter name="performative" class="String" direction="fixed">
      <value>SFipa.INFORM</value>
    </parameter>
    <match>((String)$content).indexOf("hello")!=-1</match>
  </messageevent>
</events>
```

Abbildung 4.17.: Beispieldefinition von Nachrichten in Jadex. [5, S. 51]

FIPA-Nachricht besteht simpel gesagt aus einer Liste von Parametern, welche der Sender setzen muss bzw. kann, um dem Empfänger Daten und Informationen zu übermitteln. Jede Nachricht muss dabei die vier folgenden Parameter besitzen:

- **performative** Jede Nachricht besitzt einen Typ, welcher dem Agenten schon vor dem Betrachten des eigentlichen Inhalten Hinweise darauf gibt, wie dieser zu interpretieren ist. Gängige Performatives sind z.B. *inform* (Wissen mitteilen), *request* (Ziel hinzufügen) oder *refuse* (Antwort auf *request*, Ziel abgewiesen)
- **sender** Der Agent-Identifier[10] des Senders
- **receiver** Die Agent-Identifier des Empfängers
- **content** Der eigentliche Inhalt der Nachricht, welcher vom Agenten selbst interpretiert wird

Der FIPA-Standard umfasst noch viele weitere Parameter wie *language*, *ontology* oder *conversation-id* (Für längere Dialoge zwischen Agenten), welche beliebig für die Belange des Agenten genutzt werden können. Eine vollständige Definition des FIPA-Protokolls befindet sich unter `http://www.fipa.org/specs/fipa00061/SC00061G.html`. Zusätzlich zu den Standard-Parametertypen erweitert Jadex diesen Standard um einige weitere Parameter. So kann der Sender z.B. mit dem Parameter *content-class* Hinweise geben, ob der Inhalt der Nachricht ein serialisiertes Java-Objekt ist, welches dann deserialisiert wird und als Objekt dem Empfänger zur Verfügung steht.

[10] Der lokale Name des Agenten ggf. erweitert durch die Adresse der Agentenplattform

Möchte man Nachrichten in Jadex-Agenten verwenden, so müssen diese, wie alles andere auch, zuvor in der ADF definiert werden (vgl. Abbildung 4.17). Wie im Beispiel zu sehen, generiert jede eingehende Nachricht ein Ereignis, auf welches beliebige Trigger (z.B. Ziele oder Pläne) angewendet werden können. Die jeweiligen Messageevents werden nur ausgelöst, wenn alle definierten Parameter den entsprechenden Wert besitzen und das ggf. definierte Match-Element erfüllt ist. Eingehende Nachrichten, die keinem Messageevent entsprechen, werden ignoriert.

Wie man sieht, ist das einzige, was bei eingehenden Nachrichten passiert, das Auslösen eines Triggers, sofern die Nachricht auf ein Messageevent passt und diesem ein Trigger lauscht. Damit wird sämtliche Logik, welche auf eine eingehende Nachricht folgen kann, vom Entwickler als Plan formuliert und steht unter seiner Kontrolle. Die Kommunikationsschnittstelle hat keinen Zugriff auf die Beliefbase oder die Planbibliothek, sodass dort unbefugte Zugriffe ausgeschlossen sind. Mit dieser Whitelist an akzeptierten Nachrichtenmustern können potentielle Hackerangriffe pauschal vorgefiltert werden, wobei auf die üblichen Prüfungen auf Sinn der Nachricht innerhalb der Pläne nicht verzichtet werden sollte.

Das Versenden von Nachrichten geschieht auf ähnlichem Wege. Hierbei dienen die in der ADF definierten Nachrichten als Templates, welche innerhalb von Plänen nur noch mit den individuellen Daten ergänzt werden müssen (meist muss der Empfänger bestimmt werden). Beim Versenden der Nachricht gibt es zusätzlich die Möglichkeit mit $IMessageEvent sendMessageAndWait(IMessageEvent)$ direkt auf eine Antwort zu warten (z.B. bei Request-Nachrichten auf die angeforderten Daten).

Ein Aspekt, der zwar nur bedingt zum Thema Kommunikation passt, aber trotzdem nicht unerwähnt bleiben sollte, ist die Möglichkeit, einen Agenten von einer Plattform auf eine andere Plattform, also einen anderen Computer zu migrieren. Dabei werden entsprechende Funktionalitäten der JADE-Plattform genutzt. Eine Migration eines Agenten zwischen zwei Plattformen setzt voraus, dass die gesamte Beliefbase des Agenten serialisierbar ist und der Agent zum Zeitpunkt der Migration ausschließlich *MobilePlans* ausführt (siehe Kapitel 4.3.1).

4.5.2. Jason

Das Kommunikationskonzept von Jason baut nicht auf FIPA auf, sondern auf KQML[11], wobei KQML und und die FIPA Communication Language vom Aufbau her sehr ähnlich sind. Auch KQML definiert eine Reihe von *performatives*, welche dem Agenten Interpretationshinweise zu Nachrichten geben (vgl. [4, S. 118/119]):

[11] Knowledge Query and Manupulation Language [20, S. 170], [18]

- **tell**/**untell** Den Empfänger über neue bzw. nicht mehr geltende Sachverhalte informieren.
- **tellHow**/**untellHow** Dem Empfänger neue Pläne übermitteln bzw. Pläne löschen
- **askOne**/**askAll** Den Empfänger nach einem passenden bzw. allen passenden Fakten in seiner Beliefbase fragen (per Unifikation)
- **askHow** Den Empfänger nach einem passenden Plan für das gegebene Ziel fragen
- **achieve**/**unachieve** Dem Empfänger ein neues Ziel hinzufügen, ein altes Ziel entfernen

Weitere Performatives sind nicht möglich, die Kommunikationsschnittstelle ist auch bei den Parametern festgelegt.

Empfangene Nachrichten werden, im Gegensatz zur Jadex-Variante, direkt von der Agenten-Infrastruktur interpretiert. So werden bei tell- und untell-Nachrichten direkt entsprechende Änderungen in der Beliefbase vorgenommen, woraus natürlich entsprechende Ereignisse resultieren, auf die mit Plänen reagiert werden kann. Bei tellHow- und untellHow-Nachrichten wird ebenso direkt die Planbibiothek modifiziert.
Die drei ask-Nachrichtentypen veranlassen ebenso ein direktes Durchsuchen von Beliefbase bzw. Planbibliothek, sowie auch direkt das Versenden der Antwort, ohne dass der Agent auf der Planebene etwas davon mitbekommt.

Die Tatsache, dass Nachrichten direkt interpretiert werden und die Verarbeitung nicht dem Agenten selbst überlassen wird, stellt ein Sicherheitsloch von der Größe eines Scheunentors dar. Anderen Agenten die Möglichkeit zu geben, die eigene Beliefbase ohne Nachfrage zu verändern und nur nachträglich auf Änderungen reagieren zu können, eröffnen Angreifern so gut wie alle Wege, Böses anzustellen. Auf der einen Seite ist es ein starker Vorteil von Jason, dass Agenten Pläne untereinander austauschen können, was sicherlich für lernende Umgebungen eine Rolle spielen mag, auf der anderen Seite sind aber auch gefährliche Schadcode-Injektionen möglich. So kann Agent A zuerst mit einer tellHow-Nachricht einen Plan mit Schadcode an Agent B senden, gefolgt von einer achieve-Nachricht, welche den Plan aufruft.

Die scheinbar gewollte Lösung dieses Sicherheitslecks befindet sich in der „Nachricht prüfen“-Funktion des Reasoning Cycles (vgl. Kapitel 4). Diese Funktion muss jedoch statisch als Java-Code in die Infrastruktur des Agenten implementiert werden und ist somit nur schwer dynamisch zu gestalten, was das Führen von dynamischen Black- oder Whitelists erschwert. Zudem ist die Verlagerung der Sicherheitsschicht aus der eigentlichen Modellierung des Agenten hin zur Agenteninfrastruktur unglücklich, weil so quer durch mehrere Komponenten hindurch programmiert werden muss.

```
1  .send(reveiver, tell, open(door)).
2  .send(reveiver, askOne, open(Door), Reply).
3  .send(reveiver, tellHow, "[...]" /*Plan als Quellcode*/).
4
5  .broadcast(tell, open(door)
```

Abbildung 4.18.: Nachrichtenversand in Jason (vgl. [4, S. 120f.])

Bis auf die ask-Nachrichtentypen, sind alle Nachrichten beim Versenden nicht blockierend, der sendende Agent setzt seine Ausführung direkt nach dem Versenden der Nachricht fort. Eventuelle Antworten werden dann als neue Ereignisse interpretiert. Ein nützliches Feature der Kommunikationskomponente von Jason ist der Broadcast. Da das Environment als zentrale Komponente des gesamten Agentensystems alle angemeldeten Agenten kennt, ist es möglich, mit Hilfe der $.broadcast(...)$-Funktion Nachrichten an alle Agenten zu schicken, anstatt mit der $.send(...)$-Funktion nur an einen einzigen bzw. eine ausgewählte Liste von Empfängern.

Die unterliegende Kommunikationsschicht der Agenten, welche das tatsächliche Versenden der Nachrichten übernimmt, ist wie fast alles andere ebenfalls austauschbar. Die Standardimplementation der Multiagenten-Infrastruktur ist eine zentralisierte Struktur, welche alle Agenten in einer einzigen JVM laufen lässt, sodass keine Netzwerkkommunikation erforderlich ist. Weiterin sind mit einer JADE-Infrastruktur [16] und einer SACI-Infrastruktur [17] zwei Möglichkeitern mitgeliefert, die Agenten auf verschiedene Systeme zu verteilen, um eine bessere Skalierung zu erreichen. Widersinnigerweise werden im Falle der JADE-Infrastruktur die KQML-Nachrichten für den Transport in FIPA-konforme Nachrichten umgewandelt, wobei jedoch ein eigener Serialisierungsalgorithmus verwendet wird, sodass Jason-Agenten nicht ohne weiteres mit Agenten kommunizieren können, die eine FIPA-Kommunikation benutzen.

4.5.3. Zusammenfassung

Bei der Kommunikation finden sich vor allem im Bereich der Sicherheit gravierende Unterschiede zwischen den beidem BDI-Systemen. Wo die Kommunikationsprotokolle mit FIPA und KQML noch recht ähnlich sind und in der Funktionalität auch keine Wünsche offen lassen, offenbaren sich in der Verarbeitung der Nachrichten zwei verschiedene Wege. Während Jadex seine Nachrichten einfach als Ereignisse abbildet und Trigger aufruft, welche wiederrum Pläne zur Folge haben können, die dann die Nachricht selbst bearbeiten, übernimmt in Jason eine fest implementiere und ausnahmsweise nicht modifizierbare Komponente die Interpretation der Nachrichten. Wo Jadex seine Nachrichten mittels einer Art Unifikation bei den in der ADF definierten Messageevents filtert, bleibt dem Jason-Entwickler nur Filterung

mit Hilfe der *checkMail*-Funktion, was sich als sehr aufwändig und nicht zweckmäßig herausstellt.

In sicheren Forschungsumgebungen, wo sich alle Agenten gegenseitig vertrauen, ist Jason mit seiner extrem einfachen Handhabung des Versendens von Nachrichten deutlich im Vorteil, wohingegen in produktiven, unsicheren Umgebungen Angreifern Tür und Tor geöffnet sind und sich diese in komplexeren Systemen nur schwer zu schliessen sind.

Ein Punkt in dem Jason Jadex voraus ist, ist die Möglichkeit, Broadcast-Nachrichten zu versenden, da in einem Jason-MAS generell alle Agenten bekannt sind, wohingegen in Jadex durch verteile JADE-Plattformen nie alle Agenten sicher bekannt sein können. Weiterhin gibt es in Jason die Möglichkeit, andere Agenten nach Plänen zu fragen und dementsprechend auch Pläne an andere Agenten zu verschicken, was das Multiagentensystem deutlich dynamischer gestaltbar macht.

4.6. Technik

Nachdem wir schon die Kernpunkte des BDI-Konzeptes analysiert haben, wenden wir uns einigen eher allgemeineren Themen zu, welche bei jedem Programmierframework nicht weniger wichtig sind und für eine Bewertung von Jason und Jadex unumgänglich sind. Als erstes widmen wir uns dem Thema *Sicherheit*.

4.6.1. Sicherheit

Unter dem Stichwort *Sicherheit* wird in der Informatik jede Art von Sicherheitsmangel verstanden, welche von außenstehenden Parteien genutzt werden kann, um den Agenten anzugreifen und für die eigenen Zwecke zu missbrauchen oder zu sabotieren. Jeder Softwareentwickler weiß, dass es fast unmöglich ist, ein Programm zu schreiben, welches absolut frei von Bugs und den daraus resultierenden Sicherheitslücken ist. Da solche Programmierfehler allerdings in den meisten Fällen sehr schnell nach Bekanntwerden behoben werden, sollen diese für diesen Vergleich keine Rolle spielen, wir konzentrieren uns ausschließlich auf konzeptionelle Sicherheitsschwächen der Systeme.

Wie schon in Kapitel 4.5 aufgezeigt, offenbart Jason in Punkto Sicherheit erhebliche Mängel, welche Angreifern große Möglichkeiten geben, fremde Agenten zu beeinflussen oder zu kompromitieren. Fremde Beliefs sind beliebig modifizierbar und durchsuchbar, selbst Planbibliotheken und Ziele können ohne vorige Abfrage gelöscht oder hinzugefügt werden. Zwar können Nachrichten über die *socAcc*()-Funktion während des Reasoning-Cycles in der

Agenten-Klasse gefiltert werden, jedoch ist der Aufbau einer Whitelist, wie sie in Jadex benutzt wird, umständlich, da sie immer wieder an die Pläne des Agenten angepasst werden muss, was eine große Fehlerquelle darstellt.

Die weitaus größere Sicherheitslücke, welche man als Entwickler nicht schließen kann, befindet sich in der Agenten-Kontrolle. Sowohl unter Jadex als auch unter Jason ist es möglich, zur Laufzeit auf den lokalen Plattformen neue Agenten zu starten. Zusätzlich dazu können mit der gleichen API jedoch auch bereits laufende Agenten angehalten oder gar komplett terminiert werden, ohne dass diese Agenten sich dagegen wehren können (vergleichbar mit der Prozessterminierung in Betriebssystemen). Jadex bietet, anders als Jason, jedoch die Möglichkeit, das AMS[12] der Agentenplattform ohne einen entsprechenden Agenten zu starten, sodass das AMS nur von innerhalb des Agenten erreichbar ist, nicht aber von anderen Plattformen aus, wodurch das Zerstören von Agenten von außerhalb unterbunden werden kann.
Da bei Jason diese Funktion auf keine Weise abschaltbar ist, eignet sich Jason denkbar schlecht für Anwendungen, bei denen nicht alle Agenten unter einheitlicher oder vertrauenswürdiger Kontrolle sind, z.B. Spiele mit mehreren unabhängigen Teilnehmern, welche ihre Agenten selbst verwalten.

4.6.2. Performance

Ein wichtiges Thema bei jedem IT-Projekt ist die Performance des Programmes. Speziell zu beobachten sind hierbei die beiden Punkte Prozessorlast und Speicherverbrauch. Beide Kriterien beeinflussen die Anzahl und Leistungsfähigkeit der Agenten, die auf einem Computer laufen können. Um die Frameworks in diesen Punkten vergleichen zu können, werden wir mit beiden Systemen die gleiche Anwendung simulieren, nämlich unsere bereits bekannte *Hunters and Deers*-Simulation. Hier die Eckpunkte des Benchmarks:

- **Prozessor** Intel Core2Duo E6400 2x2.13GHz
- **RAM** 6GB DDR2 @ 533MHz (2x2GB, 2x1GB)
- **Betriebssystem** Kubuntu 10.04 AMD64
- **Java-Version** OpenJDK 6 (Version 6b18-1.8-0ubuntu1)
- **Jadex-Version** Jadex 0.96
- **Jason-Version** Jason 1.4
- **Anzahl der Agenten** 5 Rehe und 5 Jäger

[12] Agent Management System

Für verlässliche Daten werden beide Plattformen jeweils 3 mal für 5 Minuten gestartet und dabei sekündlich CPU-Last und Speicherverbrauch protokolliert.

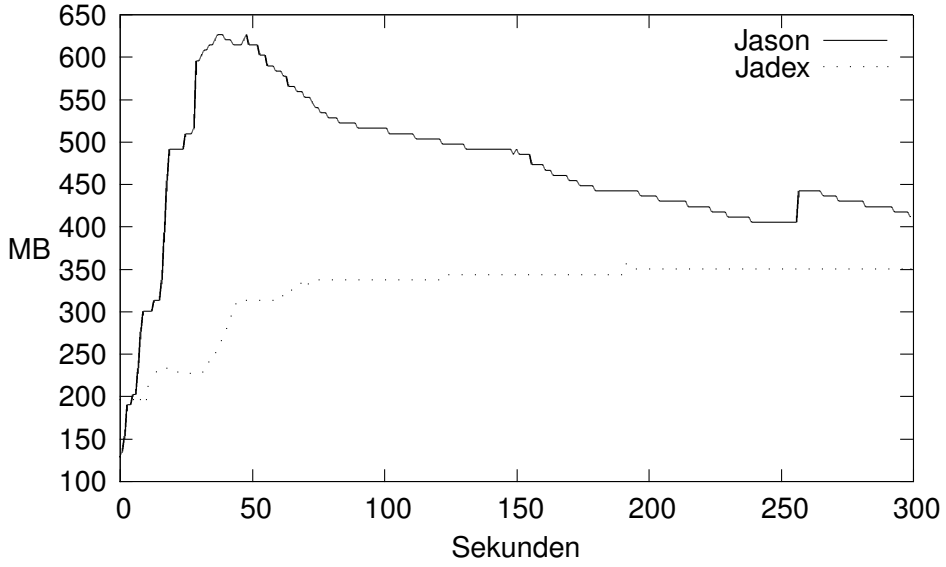


Abbildung 4.19.: Vergleich des Speicherverbrauchs der Agenten

Wie in den Abbildungen 4.19 und 4.20 zu sehen ist, besitzt Jason erhebliche Schwächen im Performancebereich. Der Speicherverbrauch von Jadex und Jason ist, wie bei Java-Anwendungen typisch, recht hoch, liegt jedoch bei beiden noch im akzeptablen Bereich. Der Spitzenwert der Jason-Simulation lässt sich mit der anfänglichen Informationsflut erklären, wenn alle Agenten den kompletten Weltzustand vom Server erfragen. Ist diese beseitigt, verringert sich der Speicherverbrauch langsam, bleibt aber über dem konstanten Verbrauch von Jadex.
Der CPU-Verbrauch offenbart ein weitaus größeres Problem. Während die 10 Jadex-Agenten meist kaum über 10 Prozent CPU-Last heraus kommt, liegt die Jason-Simulation konstant bei 90 Prozent Last, was zusammen mit vom System benutzten Resourcen bedeutet, dass die 10 Jason-Agenten das System voll auslasten. Bei solch einer desaströsen CPU-Performance lohnt es jedoch, sich die Ursachen genauer zu analysieren. Die zwei größten CPU-Fresser in Jason sind der Reasoning-Cycle und die Listenrekursion. In fast[13] jedem Reasoning-Cycle ruft der Agent beim Environment-Objekt seine komplette Wahrnehmungsliste in Form von Literalen ab und vergleicht diese mit der im vorigen Durchlauf erhaltenen Liste. Da diese Liste im Hunters-and-Deers-Programm mehrere hundert Literale umfasst, stellt dieser Abgleich einen erheblichen Overhead dar, wenn man den Aufwand mit

[13] Das Environment kann dem Agenten signalisieren, dass seit dem letzten Durchlauf keine Änderung aufgetreten ist. Da im Hunters-and-Deers Programm die Position der Objekte ständig neu interpoliert und publiziert wird, ist dies jedoch selten der Fall

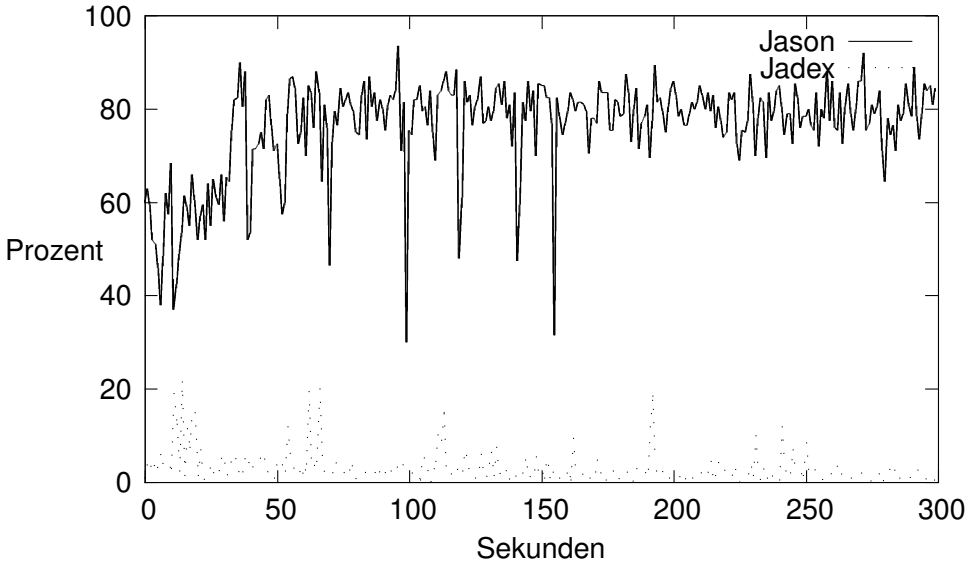


Abbildung 4.20.: Vergleich der CPU-Auslastung beider Agenten

der Laufzeit der eigentlich ausgeführten Aktion vergleicht. Selbst die aufwändigste Aktion des Agenten, das Potential eines Feldes zu berechnen, dauerte in einem Test nur wenige Mikrosekunden und steht damit in keinem Verhältnis zur Dauerlast, die durch die Wahrnehmung erzeugt wird.

Die Rekursion in Listen ist die zweite CPU-belastende Komponente. Da Listen nicht iterativ, sondern nur rekursiv durchlaufen werden können und dabei nicht auf die Java-typischen Referenzen gesetzt wird, sondern die Tail-Listen kopiert werden, um Veränderungen zu verhindern, erzeugt dies bei großen Listen durch ständiges Kopieren viel CPU-Last und auch einen hohen Speicherverbrauch. Im schlimmsten Fall kann die Rekursion zu einem StackOverflowError führen, da bei jedem Agentspeak-Rekursionsschritt der Java-Stack um drei Java-Methodenaufrufe größer wird.

Während sich die Listenrekursion notfalls in Java-Funktionen auslagern lässt, wie es bei Hunters-and-Deers geschehen ist, lässt sich der Overhead beim Wahrnehmungsabgleich nur bedingt verhindern. Entweder es werden durch die Modellierung deutlich weniger Wahrnehmungen erzeugt, dass die zu vergleichende Menge kleiner ist, oder die Wahrnehmung wird deutlich seltener verändert, was einerseits zwar den Overhead verringert, jedoch zur Folge hat, dass die Wahrnehmung ggf. ungenau wird.

Weiterhin bietet sich bei BDI-Systemen ein Vergleich der Reasoning-Performance an, konkret die Anzahl der Durchläufe des Reasoning-Cycles der beiden Frameworks. Da Jadex in der untersuchten Version leider nicht mehr auf dem klassischen Reasoning-Cycle aufbaut und die bereits beschriebene Agenda-Struktur vorzieht, ist ein Vergleich inhaltlich sinnlos.

Alles in allem hat Jadex auch in Punkte Performance die Nase vorn. Die Implementierung einer deklarativen Programmiersprache im objektorientieren Java ist deutlich langsamer als die Implementierung des BDI-Konzeptes in purem Java. Sämtliche Op-Code Optimierungen, welche die Java VM intern vornehmen kann, können in Jason durch den zwischengeschalteten Interpreter nicht angewendet werden, was Jadex einen deutlichen Vorteil gibt. Vor allem die Listenrekursion von Jason macht der Ausführung zu schaffen, da der Speicherverbrauch in der Rekursion durch das ständige Kopieren quadratisch zur Menge der Elemente ansteigt.

4.6.3. Entwicklung

Ein weiterer wichtiger Punkt bei der Auswahl des passenden Frameworks ist die Entwicklungsgeschwindigkeit. Speziell, wie schnell sich ein Entwickler von Grund auf in das neue Framework einarbeiten kann und welche IDEs und andere Tools es gibt und welche Features diese zur Verfügung stellen, um eine schnelle und unkomplizierte Entwicklung zu ermöglichen.

Einarbeitungszeit

Angenommen, ein Entwickler kennt sich sowohl mit Java und XML aus und weiß ebenso mit deklarativer Programmierung etwas anzufangen, ist die Einarbeitungszeit in Jason allein aufgrund des Umfanges des Frameworks deutlich geringer als bei Jadex. Dies wird deutlich, wenn man sich alleine den Klassenumfang anschaut. Während das komplette Jason-Framework gerade einmal 245 Klassen umfasst, kommt Jadex in diesem Punkt auf annähernd eintausend Klassen. Zudem perfektioniert Jadex geradezu die für die objektorientierte Programmierung übliche Delegation und Abstraktion von Funktionen, sodass ein tieferes Verständnis der Arbeit von Jadex sehr zeitaufwändig ist. Die Strukturierung von Jason in die Komponenten *Agentspeak*, *Agent* und *Infrastruktur* macht die Einarbeitung deutlich einfacher. Vor allem ist es aber die Eigenschaft von Jason, mit möglichst einfachen Mitteln (z.B. wenige Zieltypen) möglichst viel Funktionalität zu schaffen. Leider führt diese extreme Vereinfachung an einigen Stellen zu Funkionsmängeln, welche in Jadex nicht auftreten.

Jadex Control Center

Die Hauptentwicklungsplattform für Jadex ist das bereits mitgelieferte Jadex Control Center. Die Funktionen des JCC beziehen sich allerdings weniger auf das Entwickeln der Agenten an sich, sondern vorwiegend auf die Ausführung und das Testen der Agenten zur Laufzeit.

Neben den grundlegenden Funktionen zum Start von Agenten umfasst das JCC noch eine Reihe weiterer Tools:

- **Conversation Center** Agenten selbstgebaute Nachrichten schicken
- **Introspector** Der Debugger des JCC erlaubt die Beobachtung sämtlicher Beliefs, Pläne und Ziele sowie die schrittweise Ausführung von Agenda-Aktionen
- **BDI Tracer** Nützliches Tool, um zu sehen, wie die einzelnen Beliefs, Ziele, Pläne und Ereignisse zur Laufzeit zusammenhängen, visualisiert als Graph
- **Test Center** Ausführungstool für JUnit-ähnliche Tests auf Agenten mit ausführichen Reports. Schreiben von Tests deutlich aufwändiger als unter JUnit
- **Jadexdoc Tool** Tool zur Erstellung von Dokumentationen der Agenten

Bei der Arbeit mit dem Tool fiel auf, dass es sich gelegentlich recht eigenwillig verhält, was das Laden von jar-Archiven betrifft, in denen sich auszuführende Agenten befinden, sodass diese Agenten nicht gefunden und gestartet werden können. Ebenso werden in mehreren Konfigurationsdateien absolute Pfade verwendet, was eine Weitergabe von Projekten oder eine verteilte Entwicklung erschwert.

Alles in allem bietet das JCC sehr umfangreiche Funktionen, um Agenten Laufzeittests zu unterziehen, auch wenn einige noch nicht ganz fehlerfrei funktionieren. Für die eigentliche Programmierung des Quellcodes sollten aber andere IDEs bevorzugt werden, da das JCC in dieser Hinsicht keinerlei Unterstützung bietet.

Jason IDE

Die Jason-IDE ist wiederum das genaue Gegenteil zum JCC. Genau genommen ist die Jason-IDE nur eine Erweiterung des auf Java basierten Texteditors jEdit [2] und bietet somit viele Standardfunktionen, welche jeder bessere Editor auch bietet, mit Ausnahme der Syntaxhervorhebung von Agentspeak und Java. Schmerzlich vermisst man die automatische Codevervollständigung und Code-Templates, wie sie in vielen IDEs üblich sind, auch fehlt eine Syntaxanalyse sowohl beim Speichern als auch zur Laufzeit. Die Laufzeitumgebung von Jason bietet deutlich weniger Features als das JCC. Die Standardplattform von Jason bietet eine GUI, in der Agenten verwaltet werden können und gewährt auch einen Einblick in die internen Zustände der Agenten. Im Debug-Modus ist das schrittweise Ausführen des Reasoning Cycles möglich, welche im Zusammenhang mit der Breakpoint-Annotation von Plänen nützlich ist. Agenten, die auf der JADE-Plattform ausgeführt werden bieten grundlegende Möglichkeiten, Agenten zu verwalten. Auch wenn die JADE-Plattform einen Dialog zur Verfügung stellt, Nachrichten an Agenten zu schicken, ist dies bei Jason-Agenten sehr kompliziert, da der JADE-Adapter für Jason die Nachrichten in ein eigenes Format umwandelt.

Plugins für bekannte IDEs

Sowohl für Jadex als auch für Jason sind Plugins für die bekannte IDE Eclipse [9] erhältlich. Da Jadex ausschließlich auf Java und XML aufbaut, bietet sich Eclipse, welches ebenfalls hauptsächlich dafür genutzt wird, perfekt für die Entwicklung von Jadex an. Folglich mangelt es dort kaum an offenen Wünschen für die reine Programmierung der Agenten. Die Runtimeumgebung ist jedoch recht spartanisch gehalten. Das Plugin *EJade* [8] stellt eine JADE-Laufzeitumgebung zur Verfügung in der die Jadexagenten gestartet werden. Die JADE-Umgebung startet mit einer GUI, in welcher die Agenten verwaltet werden können (starten, pausieren, beenden, migrieren usw.). Praktisch ist, dass die normalen Debugging-Mechanismen auf die Agenten angewendet werden können. Ein Nachteil des Plugins für Eclipse ist, dass beim Start eines Agenten nicht der Projekt-Classpath benutzt wird, sondern ein eigener, nicht konfigurierbarer Classpath, was es unmöglich macht, Agenten innerhalb von Eclipse auszuführen, sobald diese Bibliotheken von Drittanbietern benutzen.
Das Jason-Plugin *Jason-IDE*[11] für Eclipse kann aufgrund von Agentspeak nicht auf bereits vorhandene Funktionen zurückgreifen. Während für die Java-Teile noch die üblichen Features von Eclipse genutzt werden können, ist der Agentspeak-Editor sehr einfach gehalten und bietet außer Syntax-Hervorhebung keine nennenswerten Features. Eine Syntaxprüfung findet nur beim Speichern der Datei statt und dort sind die Fehlermeldungen teils wenig aussagekräftig, manchmal erinnert die Meldung „Fehler beim Speichern“ sehr an die Suche der Nadel im Heuhaufen. Als Laufzeitumgebung baut das Plugin auf die Jason-eigenen Mittel, da dort bereits brauchbare Agentenverwaltungs- und Debugging-Mechanismen existieren. Java-Anteile der Jason-Agenten können wie beim Jadex-Plugin auch mit Eclipse debuggt werden.

Das große Konkurrenzprojekt zu Eclipse *NetBeans* [12] bietet für keine der Frameworks passende Plugins.

5. Zusammenfassung

Nachdem wir im 4. Kapitel Jadex und Jason Stück für Stück miteinander vergleichen haben, fassen wir unsere Ergebnisse noch einmal zusammen und ziehen ein Fazit.

5.1. Zusammenfassung

Beliefbase Jadex	**Beliefbase Jason**
- imperativ als Java-Objekte und Collections - Zugriff und Überwachung über OQL, JavaBeans und Observer-Pattern	- deklarativ mit logischen Literalen - Zugriff über Unifikation und logische Formeln - Annotations - Belief-Revision vorbereitet

Auch wenn die Beliefbases im Aufbau sehr verschieden sind, gleichen sich letztlich die Funktionalitäten der Implementierungen aus. Den Malus bei der Änderungsüberwachung gleicht Jason mit der Verfügbarkeit von Annotationen und der vorbereiteten Belief-Revision wieder aus. Letztlich ist die Entscheidung zwischen dem objektorientieren Java und dem deklarativen Agentspeak eher Geschmackssache.

Ziele Jadex	**Ziele Jason**
- 4 Zieltypen (plus Metagoal) mit diversen Konfigurationsmöglichkeiten - Bedingungen prüfen über Observer-Pattern Korrektheit der Ziele - umfangreiche Goaldeliberation	- nur zwei Ziele (Achieve- und Testgoal) - keine Konfigurationen, Ziele sind nur Ereignisse - Goaldeliberation und Zeitplanung nur in Plänen

Die Ziele sind in Jason der extremen Einfachheit zum Opfer gefallen. Während Jadex mehrere Zieltypen mit üppigen Konfigurationen zur Verfügung stellt, sind Ziele in Jason nichts

weiter als erweiterte Funktionsaufrufe. Mechanismen wie Goaldeliberation oder Zeitsteuerung müssen aus Plänen heraus realisiert werden.

Pläne Jadex	**Pläne Jason**
- Definition und Konfiguration in ADF, Körper in Java - Planauswahl durch Trigger und Start- und Kontextbedingungen - Funktionen zur Fehlerbehandlung in Java	- komplett in Agentspeak geschrieben - Planauswahl durch Signatur und Kontextbedingung - Fehlerbehandung durch Failgoal

Die Pläne weisen in Jadex und Jason viele Ähnlichkeiten auf. Beide Framworks bieten ausreichend Funktionen zur Planauswahl, wobei Jadex mit seiner Kontextbedingung, die zur Laufzeit des Planes den Plan nachträglich auf seine Richtigkeit prüft. Im Umfang der möglichen Aktionen sind keine nennenswerten Unterschiede festzustellen, da im Notfall auch Jason auf Java zurückgreifen kann, um fehlende Funktionen nachzubauen.

Reasoning-Cycle Jadex	**Reasoning-Cycle Jason**
- kein traditioneller Reasoning-Cycle mit einzelnen Schritten, sondern agendaorientierte Liste von Aktionen - kein Environment mit expliziterWahrnehmung - Modifikation nur durch Konfigurationen in ADF	- traditioneller zehnstufiger Reasoning-Cycle - explizite Wahrnehmung über Environment-Objekt - sämtliche Auswahlfunktionen in Java selbst implementierbar

Technisch sind die Reasoning-Cycles der Frameworks sehr verschieden. Während Jadex gar keinen Cycle im eigentlichen Sinne aufweist und sämtliche innere Logik auf einzelne Aktionen abbildet, die nacheinander ausgeführt werden, baut Jason auf einen übersichtlichen, klar strukturierten Reasoning-Cycle, welcher an allen wichtigen Stellen den eigenen Bedürfnissen anpassbar ist.
Schmerzhaft fehlt in Jadex eine zentrale Wahrnehmungssteuerung wie in Jason. Basiert die Wahrnehmung nicht ausschließlich auf der Kommunikationsschnittstelle, müssen die Sensoren unsauber von außen die Beliefs verändern oder es müssen Pläne eingerichtet werden, welche zyklisches Polling an den Sensoren betreiben.

Kommunikation Jadex	**Kommunikation Jason**
- FIPA-Protokoll mit diversen Performatives und Steuerungsattributen - Filterung von Nachrichten mit *Message-events* in der ADF - Messageevents triggern Pläne und Ziele	- KQML-Protokoll mit mehreren Performatives (u.a. tellHow und askHow für Pläne) - Interpretation der Nachrichten durch Infrastruktur -> direkte Modifizierung und Abfrage der Beliefbase ohne Agentenkontrolle - Filterung ausschließlich über $socAcc()$-Funktion während des Reasoning-Cycles

Auch wenn die Kommunikationsprotokolle sehr ähnlich sind, was Aufbau und Funktionsweise betrifft, offenbart die Behandlung der Nachrichten innerhalb des Agenten erste Sicherheitslücken in Jason. Nachrichten werden in Jason außerhalb der Kontrolle des Agenten selbst interpretiert und bearbeitet und der Agent kann nur auf daraus resultierende Ereignisse reagieren. Die $socAcc()$-Funktion bietet zwar die Möglichkeit, Nachrichten oder Performatives im Vorfeld zu blocken, jedoch wird dadurch die Kommunikationskontrolle quer durch den Agenten verteilt, was die Programmierung erschwert. Jadex hingegen betreibt eine Whitelist-Filterung in der ADF, welche direkt Trigger auslöst.

Sicherheit Jadex	**Sicherheit Jason**
- Kommunikationsfilterung mit Whitelist - Agent-Management-System kann Agenten starten und stoppen, der Aufruf von außen kann jedoch geblockt werden	- komplizierte Filterung von schädlichen Nachrichten, direkte Interpretation durch Infrastruktur - jeder Agent kann jeden Agenten abschießen und das MAS anhalten

In Punkto Sicherheit ist Jason der klare Verlierer. Wo die Sicherheitslücke bei der Kommunikation noch regulär aber aufwändig gestossen werden kann, ist ein System, welches es Agenten erlaubt, beliebige andere Agenten im gesamten MAS einfach zu beenden, im produktiven Umfeld einfach nicht vertretbar.

Performance Jadex	**Performance Jason**
- sämtliche Vorteile von Java (Referenzen, Garbage-Collection etc.)	- direkter Interpreter für deklarative Sprache in Java aufwändig, keinerlei Optimierungen der JVM anwendbar - sehr langsame Listenrekursion mit häufigen Kopiervorgängen - Rekursion führt zu großen Stacks - Wahrnehmung im Reasoning-Cycle erzeugt sehr großen Overhead

Die Performance ist neben der Sicherheit Jasons zweite große Schwachstelle. Die unoptimierte Interpretation der deklarativen Sprache in Java führt vor allem bei aufwändigeren Listenrekursionen zu starken Leistungseinbußen, sodass diese mit Zusatzfunktionen in Java nachgebaut werden müssen. Ebenfalls das Wahrnehmen im Reasoning-Cycle stellt den Entwickler vor große Probleme. Auf dem Testsystem konnten problemlos deutlich mehr Jadex-Agenten als Jason-Agenten laufen, ohne dass nennenswerte Einbußen erkennbar waren.

Entwicklung Jadex	**Entwicklung Jason**
- hoher Einarbeitungsaufwand durch komplexe Klassenstruktur und viele Konfigurationsmöglichkeiten - mit JCC sehr umfangreiches Testcenter für Laufzeittests und Untersuchungen - rudimentäres Eclipse-Plugin zur Programmierung und zum Starten von Agenten	- geringer Einarbeitungsaufwand durch gute Strukturierung und geringe Komplexität - mitgelieferte IDE sehr rudimentär, einfache Debuggingmöglichkeiten - einfaches Eclipse-Plugin mit Debuggingmöglichkeiten der Jason-IDE

Den sehr hohen Einarbeitungsaufwand versucht Jadex mit einer Vielzahl von Entwicklertools auszugleichen, mit denen Agenten umfangreich untersucht und getestet werden können. Für die eigentliche Programmierung werden externe Editoren benötigt (z.B. Eclipse). Die Jason-IDE ist nur ein erweiterter Texteditor, welcher in seinen Features problemlos durch das Jason-Eclipse-Plugin übertroffen wird, wobei auch dieses noch nicht ausgereift ist (nichtssagende Fehlermeldungen, keine Codevervollständigung).

5.2. Fazit

Betrachtet man alle Punkte in Summe, ist Jadex bei produktiven Anwendungen alleine wegen der eklatanten Sicherheitsmängel von Jason die eindeutig bessere Wahl. Da MMOGs, welche auf Multiagentensystemen basieren, sicherlich meist darauf aufbauen, dass die einzelnen Charaktere des Spieles von verschiedenen Agenten gesteuert werden, welche direkt von den konkurrierenden Parteien geschrieben werden, ist Jadex eindeutig zu bevorzugen, da hier die Agenten wirklich autonom betrieben werden können und vor unerlaubten Zugriffen seitens der Konkurrenz geschützt sind.

In wissenschaftlichen Bereichen allerdings, wo vorwiegend Forschung mit Simulationen betrieben wird und Hackerangriffe nicht vorkommen, sieht die Lage jedoch anders aus. In Summe gleichen sich viele Vor- und Nachteile der Frameworks gegenseitig weitgehend aus, sodass die Wahl wohl meist aufgrund der konkreten Anforderungen der Simulation getroffen werden muss. Die Komponenten *Pläne* und *Kommunikation* sind im Funktionsumfang quasi ebenbürtig, die fehlenden Funktionen in den Zieldefinitionen von Jason werden größtenteils nur auf andere Komponenten verlagert und sind damit auch vorhanden. Streitpunkt wird die Modellierung der Wahrnehmung der Agenten. Läuft die gesamte Wahrnehmung ausschließlich über die Kommunikationsschnittstelle, sodass die Environment-Wahrnehmung nicht benötigt wird, ist Jadex auch in Punkto Reasoning-Cycle annähernd auf gleicher Höhe wie Jason. Bei zu vielen Sensoren, welche nicht die Kommunikationsschnittstelle benutzen, wird es in Jadex aufwändig und unsauber, diese einzugliedern, wohingegen es bei Jason eine zentrale Anlaufstelle dafür gibt.
Ein signifikanter Nachteil von Jason für größere und komplexere Simulationen ist sicher die schlechtere Performance des Agentspeak-Interpreters, für die immer wieder Java-Auswege gesucht werden müssen. Echtzeitsysteme mit vielen, sich häufig ändernden Wahrnehmungen, welche schnell bearbeitet werden müssen, sind in Jason praktisch nicht zu realisieren, da der erzeugte Overhead der Wahrnehmung nicht zu verkraften ist.
Im Punkt Verteilbarkeit hat Jadex gerade durch die Abwesenheit eines Environment-Objektes die Nase vorne, da so kein Flaschenhals existiert, welcher das System verlangsamen könnte. Vom Design her können tausende Agenten in einem Jadex-System existieren, welche untereinander nur wenige andere Agenten kennen, was die Menge der kommunizierten Daten deutlich verringert. Die maximale Anzahl der möglichen Jason-Agenten hängt dagegen stets von der Leistung des Systems, auf dem das Environment-Objekt betrieben wird, und von der Menge der kommunizierten Wahrnehmungsdaten ab. Durch diese Begrenzung ist Jason nur eingeschränkt für den Betrieb von MMOGs geeignet.

Letzter Entscheidungspunkt zwischen beiden Frameworks ist wie so oft die Fähigkeiten der Entwickler, da die Frameworks auf zwei grundverschiedene Programmierparadigmen setzen, welche sich schon bei der Weltmodellierung bemerkbar machen. Erfahrene Entwickler aus dem Java-Lager werden sich sicher mit deklarativer Programmierung schwer tun und viel

Einarbeitungszeit brauchen. Da für Jason sowohl Erfahrung in deklarativer als auch in objektorientierter Programmierung gebraucht wird, müssen die Entwickler dort generell erfahrener oder zumindest flexibler sein, auch wenn die Einarbeitungszeit in das Jason-Framework an sich recht kurz ist. Das Jadex-Framework fordert jedoch auch von Java-Programmierern aufgrund der unglaublich hohen Vielfalt von Klassen, Typen und Konfigurationen viel Einarbeitungszeit, um komplexere Aufgaben zu lösen. Letztlich entscheidet in diesem Punkt wohl häufig der persönliche Geschmack der Entwickler.

Ein Hauptziel der Arbeit war es, Jadex und Jason konzeptuell zu vergleichen und auch zu untersuchen, welches Framework das BDI-Konzept korrekter bzw. besser umgesetzt hat. Jadex hat sich an einigen Stellen deutlich vom ursprünglichen Konzept von BDI entfernt und hat mehr Wert auf die praxisnäheren Aspekte gelegt. Für zukünfige Versionen ist Jadex sicher deutlich flexibler erweiterbar, wohingegen sich Jason fast überall fast exakt an das BDI-Konzept gehalten hat, jedoch damit auch die ein oder andere Schwäche übernommen hat. Allgemein werden in Jason Weiterentwicklungen sicher schwerer umsetzbar sein.
Wie bereits gezeigt, haben Jadex und Jason grundlegend verschiedene Zielgebiete, was die Anwendung betrifft. Jadex wurde mehr für praxisorientierte, produktive Anwendungsfälle entwickelt, Jason dagegen verzichtet auf diverse Sicherheitsstrukturen, wodurch viele Dinge deutlich einfacher werden, sodass Jasons Stärken in Bereichen liegt, wo Sicherheit nicht benötigt wird.

A. CD-Anlage

Der Bachelorarbeit liegt eine CD mit folgendem Inhalt bei:

- **bin/lib** Die einzelnen ausführbaren Programme
  - **HadObserver** Ein grafischer Client zum Beobachten der Spielwelt
  - **HadServer** Der HaD-Darkstar-Server für die Verwaltung der Spielwelt
  - **jadex** Der Jadex-Agent
  - **jason** Der Jason-Agent
- **bin/unix** Startscripte der Programme für Unix-basierte Systeme
- **bin/win32** Startscripte der Programme für Windows-Systeme
- **src** Der Quellcode zu den Programmen
  - **HuntersAndDeersCommon** Das Common-Projekt beinhaltet allgemeine Funktionen, welche von allen anderen Projekten gemeinsam genutzt werden
  - **HuntersAndDeersJadex** Der Quellcode zum Jadex-Agenten
  - **HuntersAndDeersJason** Der Jason-Quellcode
  - **HuntersAndDeersObserver** Der Quellcode zum Beobachter-Client
  - **HuntersAndDeersServer** Quellcode des Had-Darkstar-Servers
- **BA_christian_thiel.pdf** Die digitale Version dieser Bachelorarbeit im PDF-Format
- **readme.txt** Eine Kurzanleitung zur Benutzung der einzelnen Komponenten

Wie an anderen Stellen bereits betont, wurde im Rahmen dieser Bachelorarbeit ausschließlich der Jason-Agent entwickelt. Alle anderen Projekte (der Had-Server, der Jadex-Agent, der Observer und das Common-Projekt) wurden im Rahmen der Bachelorarbeit von Carsten Canow [7] entwickelt.

B. Der Jason-Agent

Dieses Kapitel beschreibt in kurzen Zügen den Aufbau des Jason-Agenten.

B.1. Hunters-and-Deers-Server

Der Hunters-and-Deers-Server stellt die zentrale Komponente in diesem Multiagentensystem dar und verwaltet alle angemeldeten Rehe und Jäger sowie selbst erstelle Baum- und Gras-Objekte. Der Server verwaltet ebenso die Bewegungen der Agenten. Dabei sendet jeder Client eine Bewegungsanfrage mit einer Zielangabe an den Server und dieser sendet diese nach einer Prüfung an alle Agenten weiter. Die Agenten bekommen also nicht periodisch die aktuelle Position eines Agenten zugeschickt, sondern müssen diese anhand des erhaltenen Bewegungsvektors und der Geschwindigkeit des jeweiligen Agenten selbst interpolieren.

B.2. Communication

Die Kommunikationskomponente ist das clientseitige Gegenstück zum Server und verwaltet die bidirektionale Kommunikation zu diesem. Die Klasse *Client* übernimmt dabei die eigentliche Kommunikation und die Verwaltung der Sockets und Channels. Die Klasse *ClientMessageProcessor* bekommt die empfangenen Nachrichten weitergeleitet und interpretiert sie entsprechend, wobei die meiste Funktionalität an das Environment ausgelagert wurde und die Nachricht schlicht delegiert wird.

B.3. Environment

Das Environment besteht aus zwei verschiedenen Komponenten.
Die Klasse *HADEnvironment* ist abgeleitet von der Environment-Klasse von Jason und verwaltet somit sämtliche Wahrnehmungen und auch die meisten Nachrichten, welche vom

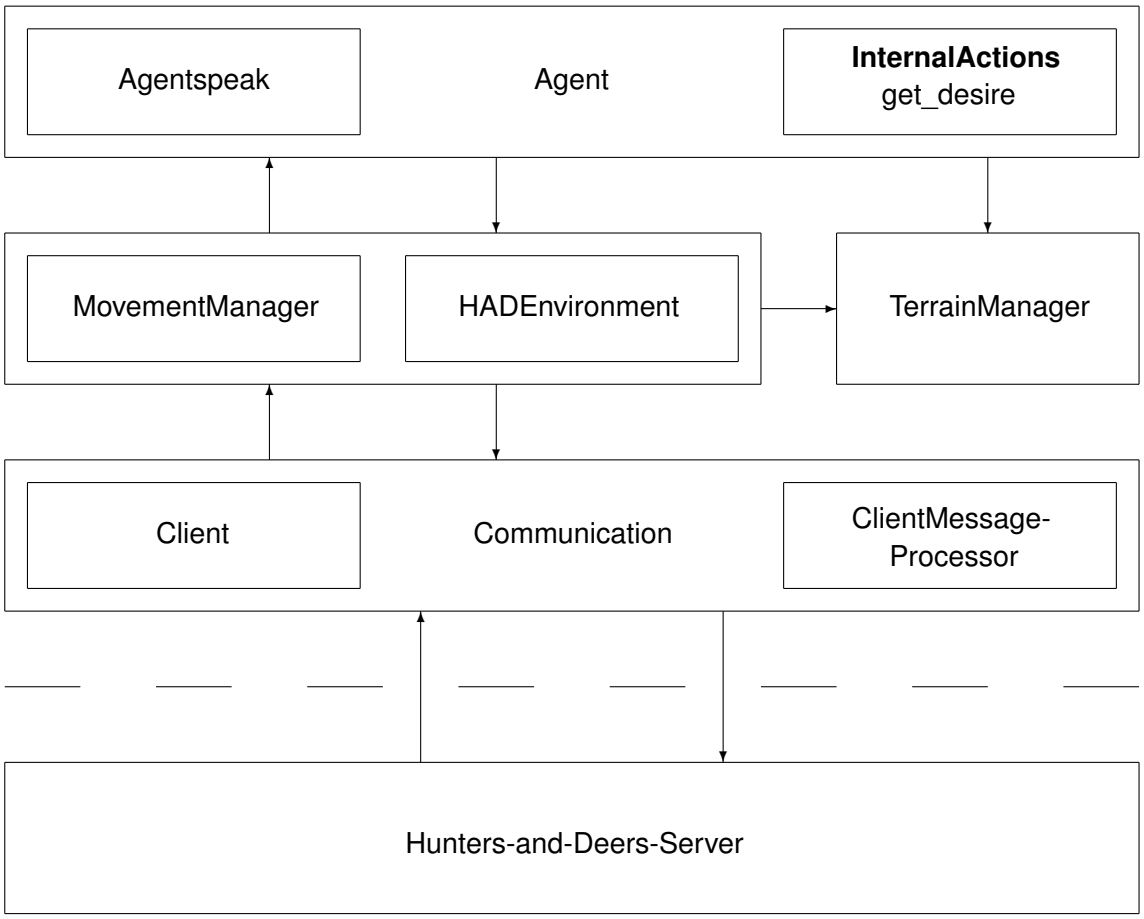


Abbildung B.1.: Komponenten des Jason-HaD-Agenten

Server empfangen werden. Innerhalb der Environment-Komponente werden alle Aktionen, mit welchen Agenten ihre Umwelt verändern können, vom HADEnvironment in eigene, vom *IEnvironmentAction*-Interface abgeleitete Klassen ausgelagert, um die Übersichtlichkeit zu wahren. Die einzelnen *EnvironmentActions* senden über die Kommunikationskomponente Nachrichten an den Server, um dort Aktionen wie Bewegung, Angriff oder Konsumieren von Nahrung auszuführen.
Die zweite wichtige Komponente des Environment ist der *MovementManager*. Dieser bekommt vom *HADEnviromnent* sämtliche Bewegungen mit Start- und Zielposition sowie Geschwindigkeit weitergeleitet und interpoliert aus diesen Daten in regelmäßigen Abständen in einem eigenen Thread die neue Position des Objektes und leitet sie an das *HADEnvironment* weiter, welches die neue Position dann publiziert.

Am Rande der Environment-Komponente befindet sich der *TerrainManager*, welcher aus dem Jadex-Agenten übernommen wurde und das Terrain der Welt für den lokalen Agenten

verwaltet. Sowohl das Environment als auch die Agenten selbst nutzen den *TerrainManager* zur Ermittlung der korrekten Höhe einer xy-Koordinate in der Spielwelt, welche der Manager aus Heightmaps errechnet.

B.4. Agent

Der Agent teilt seine Arbeit in zwei Teile auf. Der Hauptteil der Arbeit fällt auf die in Agentspeak geschriebenen Pläne. Die Sourcecode-Datei *agent.asl* bietet Pläne, welche die Bewegung des Agenten organisieren und auf allgemeine Wahrnehmungen wie neue Objekte, deren weitere Eigenschaften vom Server erfragt werden müssen, reagiert. Die Dateien *deer.asl* und *hunter.asl* stellen spezifische Pläne zur Verfügung, welche die individuellen Aktionen der Rehe und Jäger ausführen, sobald sich entsprechende Zielobjekte nähern. Der zweite Teil des Agenten sind seine internen Aktionen. Die wichtige Aktion ist hierbei *get_desire*, welche für den Bewegungsplan für eine bestimmte Position den entsprechenden Potentialwert berechnen kann. Diese Funktion wurde ursprünglich ebenfalls in Agentspeak modelliert, was sich aufgrund der zu langen Laufzeit durch lange Listenrekursionen jedoch als untauglich erwiesen hat.

Literaturverzeichnis

[1] Natasha Alechina, Rafael H. Bordini, Jomi Fred Hübner, Mark Jago, and Brian Logan. Belief revision for agentspeak agents. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1288–1290, New York, NY, USA, 2006. ACM.

[2] Dale Anson and Alan Ezust. jEdit. `http://www.jedit.org/`.

[3] Rafael H. Bordini, Jomi F. Hübner, and Michael Wooldridge. Jason. `http://jason.sourceforge.net/Jason/Jason.html`.

[4] Rafael H. Bordini, Jomi F. Hübner, and Michael Wooldridge. *Programming multi-agent systems in Agentspeak using Jason*. Wiley, 2007.

[5] Lars Braubach and Alexander Pokahr. Jadex. `http://jadex.informatik.uni-hamburg.de/`.

[6] Lars Braubach and Alexander Pokahr. Jadex Userguide. `http://freefr.dl.sourceforge.net/project/jadex/jadex/0.96/userguide-0.96.pdf`.

[7] Carsten Canow. Simulation menschlicher Spieler in einem Massively Multiplayer Online Game durch Methoden der Künstlichen Intelligenz. Master's thesis, HAW Hamburg, 2010.

[8] Cu Duy Nguyen. EJADE. `http://disi.unitn.it/~dnguyen/ejade/`.

[9] Eclipse Foundation. Eclipse. `http://www.eclipse.org/`.

[10] Foundation for Intelligent Physical Agents. FIPA - Foundation for Intelligent Physical Agents. `http://www.fipa.org/`.

[11] Germano Fronza, Jomi F. Hübner, and Rafael H. Bordini. Jason Plugin for Eclipse. `http://jasonplugin.wikidot.com/equipe-de-desenvolvimento`.

[12] Oracle Corporation. NetBeans. `http://netbeans.org/`.

[13] Jeff Orkin. Goal-Oriented Action Planning. `http://web.media.mit.edu/~jorkin/goap.html`.

[14] Stuart Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Pearson Education International, 2003.

[15] Sun Labs. Project darkstar. `http://www.projectdarkstar.com/`.

[16] Telecom Italia S.p.A. DE - Java Agent DEvelopment Framework. `http://jade.tilab.com/`.

[17] Universidade de São Paulo. SACI - Simple Agent Communication Infrastructure. `http://www.lti.pcs.usp.br/saci/`.

[18] University of Maryland Baltimore County. KQML - Knowledge Query Manipulation Language. `http://www.cs.umbc.edu/research/kqml/`.

[19] Frank Gillert Wolf R. Hansen. *RFID für die Optimierung von Geschäftsprozessen*. Hanser Fachbuchverlag, 2006.

[20] Michael Wooldridge. *An Introduction to Multi Agent Systems*. Wiley, 2006.

[21] Huiliang Zhang and Shell Ying Huang. A general framework for parallel bdi agents. In *IAT '06: Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.

Abbildungsverzeichnis

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 10. August 2010	
Ort, Datum	Unterschrift