Projektgruppe 491:
Wissen in Multiagentensystemen

Endbericht

Adib Dado
<adib.dado@web.de>

Mohamed Dalil
<mmdalil@gmx.de>

Igor Drobiazko
<igor.drobiazko@udo.edu>

Jens Eckstein
<jens.eckstein@udo.edu>

Ulrich Engler
<ulrich-engler@web.de>

Daniela Kölling
<daniela.koelling@udo.edu>

Patrick Krümpelmann
<patrick.kruempelmann@udo.edu>

Dennis Rüther
<dennis_ruether@web.de>

Özlem Sentürk
<ozlem.s@web.de>

Matthias Thimm
<matthias.thimm@udo.edu>

Stefan Tittel
<stefan@tittel.net>

Max Vorderstemann
<max.vorderstemann@udo.edu>

Universität Dortmund, Fachbereich Informatik
Lehrstuhl 6 – Information Engineering

24. Mai 2007

Vorwort

Dies ist der Endbericht der Projektgruppe 491 des Fachbereichs Informatik der Universität Dortmund. Die Projektgruppe fand im Sommersemester 2006 und im Wintersemester 2006/2007 statt und wurde von Fr. Prof. Dr. Kern-Isberner und Dipl.-Inform. Manuela Mark vom Lehrstuhl 6, Arbeitsgruppe *Information Engineering* betreut. Das Thema der Projektgruppe war die Konzeption und Realisierung von Wissen in Multiagentensystemen.

Der vorliegende Endbericht ist ein Resultat dieser einjährigen Forschungsarbeit in Wissensrepräsentation, -verarbeitung und -darstellung, sowie Interaktionen von Agenten in verteilten Systemen. Das andere Resultat ist das Programm KIMAS, welches die in diesem Endbericht vorgestellten Konzepte von Wissen in Multiagentensystem implementiert. Neben dem konzeptuellen Teil dieses Berichts, wird ausserdem auch dieses Programm vorgestellt.

Inhaltsverzeichnis

1 Einleitung **7**

2 Konzeptualisierung **10**
2.1 Überblick . . . 10
2.2 Logikbasierte Fragen und Antworten . . . 11
2.2.1 Motivation . . . 11
2.2.2 Logische Fragen . . . 11
2.2.3 Logische Antworten . . . 14
2.2.4 Weitere Logische Informationen . . . 16
2.3 Wissenszustand . . . 17
2.4 Wissensoperatoren . . . 17
2.5 Erkennung und Behandlung von Lügen . . . 18
2.5.1 Problemstellung . . . 18
2.5.2 Der Ansatz des (erweiterten) *SimpleOperator* . . . 19
2.5.3 Der Ansatz des UpdateOperators . . . 21
2.6 Vertrauenswürdigkeit von Agenten . . . 23
2.6.1 Überblick . . . 23
2.6.2 Reliabilities . . . 23
2.7 Agentenmodell . . . 24
2.7.1 Externe Sicht . . . 24
2.7.2 Interne Sicht . . . 25
2.7.3 Formale Darstellung . . . 25
2.8 Know How . . . 29
2.8.1 Theoretische Grundlagen . . . 29
2.8.2 Das erweiterte Modell . . . 30
2.8.3 Ziele und Aktionen . . . 30
2.9 Ablauf von Szenen im Multiagentensystem . . . 32
2.9.1 Überprüfung der Szenenziele . . . 33

3 Wissensdarstellung **35**
3.1 Wissensoperationen . . . 35
3.1.1 Einführung . . . 35
3.1.2 Der SimpleOperator . . . 35
3.1.3 Der Update-Program-Operator . . . 36
3.1.4 Minimale Antwortmengen . . . 40
3.1.5 Strikt Minimale Antwortmengen . . . 42

3.1.6 Der UpdateOperator . . . 43
3.2 Vertrauenswürdigkeiten . . . 45
3.3 Lügen . . . 46
3.3.1 Verbergen von vorhandenem Wissen . . . 46
3.3.2 Lügen und Generierung alternativer Antworten . . . 47

4 Wissensmodellierung 51
4.1 Roman . . . 51
4.1.1 Das fehlende Glied in der Kette . . . 51
4.1.2 Änderungen gegenüber der Romanvorlage . . . 52
4.2 Inferenzbaum . . . 54
4.2.1 Aufbau des Inferenzbaums . . . 56
4.3 Szenenbeschreibung und Einteilung der Szenen . . . 61
4.3.1 Hintergrund und Herausforderung . . . 61
4.3.2 Handlungsstränge . . . 62
4.3.3 Szeneneinteilung . . . 65
4.4 Modellierung in DLV . . . 71
4.4.1 Prädikate . . . 71
4.4.2 Initiales Wissen . . . 79
4.4.3 Aktionen . . . 82
4.4.4 Fragen . . . 84
4.4.5 Lügengenerierung . . . 85
4.4.6 Erste Szene: Tatortuntersuchung . . . 87
4.4.7 Letzte Szene: Überführung des Mörders . . . 89

5 Realisierung 90
5.1 Überblick . . . 90
5.2 Agentenarchitektur . . . 91
5.2.1 Logikbasiertes Wissen . . . 92
5.2.2 Know-How und Intentionen . . . 96
5.2.3 Antwortoperator . . . 99
5.3 Agenteninitialisierung . . . 99
5.3.1 Startordnung, constraint rules und Starten der Agenten . . . 100
5.3.2 Initialisierung der Agenten . . . 100
5.4 Nachrichtentransfer . . . 101
5.4.1 SendMessagePlan . . . 102
5.4.2 HandleMessagePlan . . . 102
5.4.3 ReceiveMessagePlan . . . 103
5.5 Inferenz . . . 103
5.5.1 Abstrahierung von Operatoren . . . 104
5.5.2 Die Implementierung des *SimpleOperator* . . . 104
5.5.3 Die Implementierung des *Update-Operator* . . . 105
5.5.4 Die Implementierung der *Update-Program-Operators* . . . 106

5.5.5 Die Implementierung des strikten Minimalitätsfilters für Update Programme . . . 106
5.6 Modellierung der Umwelt . . . 106
5.6.1 Nachrichtentransfer . . . 107
5.6.2 Szenenverwaltung . . . 107
5.6.3 Modellierung von Orten und Dingen . . . 108
5.6.4 Interaktion . . . 109
5.7 Testen . . . 110
5.7.1 Was ist unit testing? . . . 110
5.7.2 Richtlinien für unit testing . . . 111
5.7.3 Ausreden, keine unit tests zu schreiben . . . 111
5.7.4 JUnit . . . 113
5.7.5 Was ist code coverage? . . . 117
5.7.6 Arbeitsweise der *coverage tools* . . . 119
5.7.7 Cobertura . . . 119
5.8 Editor . . . 122
5.8.1 Einführung . . . 122
5.8.2 Modelle, Metamodelle und Meta-Metamodelle . . . 122
5.8.3 *MOF: meta-object facility* . . . 123
5.8.4 *EMF: Eclipse Modelling Framework* . . . 123
5.8.5 Motivation für den Editor . . . 124
5.8.6 Editor-Entwicklung . . . 124
5.9 GUI . . . 128
5.9.1 Schnittstellen zur anderen KiMAS-Klassen . . . 129
5.9.2 Verwaltung der Daten . . . 131
5.9.3 Abhängigkeiten der Komponenten . . . 132
5.9.4 Formatierung der Ausgaben . . . 136

6 Benutzerhandbuch 139
6.1 Installation . . . 139
6.1.1 Voraussetzungen . . . 139
6.1.2 Kimas starten . . . 139
6.2 Quickstart . . . 139
6.3 Entwicklung eigener Szenarien . . . 144
6.3.1 Erstellen des CDFs . . . 145
6.3.2 Erstellen des EDFs . . . 149
6.3.3 Erstellen des SDFs . . . 152
6.4 Bedienung der graphischen Benutzeroberfläche . . . 154
6.4.1 Starten der GUI . . . 155
6.4.2 Überblick über die GUI . . . 155
6.4.3 Laden und Starten eines Scenarios . . . 159
6.4.4 Darstellung des Agenteninnenlebens . . . 159
6.4.5 Darstellung von Kommunikation und Systemnachrichten . . . 159
6.4.6 Sonstige Ausgaben . . . 160

7 Erfahrungsbericht **162**
7.1 Seminarphase . . . 162
7.2 Sommersemester 2006 . . . 163
7.2.1 Entwicklungsmodell und Konventionen . . . 163
7.2.2 Erste Gruppeneinteilung . . . 163
7.2.3 Einarbeitungsphase . . . 164
7.2.4 Konzeptualisierungsphase . . . 164
7.3 Wintersemester 2006/2007 . . . 164
7.3.1 Organisatorische Änderungen . . . 164
7.3.2 Implementierung . . . 165
7.3.3 Vorträge während des Semester . . . 165

8 Fazit und Ausblick **166**

Glossar **170**

Literaturverzeichnis **171**

Stichwortverzeichnis **173**

Danksagung **175**

1 Einleitung

Dennis Rüther

Die vorliegende Arbeit ist der Endbericht der PROJEKTGRUPPE 491. Die Projektgruppe beschäftigt sich mit der Erstellung eines *Multiagentensystems*, in dem Wissen erworben und verarbeitet wird. Die Agenten sollen in einer modellierten Umwelt agieren, kommunizieren und ihr Wissen verändern können. Die Vorlage der zu modellierenden Umwelt ist Agatha Christies *"The Mysterious Affair At Styles"*. Einfach gesagt, sollen einige Agenten den Mörder oder die Mörder (in diesem Fall sind es zwei Mörder) finden. Verschiedene Komponenten sind dabei zu beachten:

- Agentenarchitektur
- Die Modellierung des Wissens und der Umwelt der Agenten
- Realisierung wichtiger Operatoren unter der Beachtung eines Lügenkonzepts
- Oberfläche und Funktionalität

Die praktische Umsetzung des geforderten Ziels unter Beachtung verschiedener für die aktuelle Forschung relevanter Aspekte, wird zudem ein weiterer Themenschwerpunkt sein. Es müssen Architekturen für Agenten entwickelt werden. Unter einem Agenten wird dabei ein autonomes, proaktives, reaktives, soziales sowie lern- und anpassungsfähiges Programm verstanden. Die von der Projektgruppe entwickelten Agenten sind nach dem BDI-Modell konzipiert. Ein Agent besitzt eine logikbasierte Wissensbasis, generiert eigene Ziele und verknüpft Ziele und Handlungen durch sein *Know-How*. Die grundsätzlichen Konzepte für die Agentenarchitektur werden im Abschnitt 2.7 beschrieben. Verschiedene Operatoren müssen für die Agenten realisiert werden. Ein Wissensoperator dient zur Integration neuen Wissens in die Wissensbasis der Agenten. Ein Kommunikationsoperator regelt das Generieren von Fragen und Antworten. Die Kommunikation der Agenten wird weiterhin durch gegenseitige Vertrauenswürdigkeit der Agenten untereinander beeinflußt. Dabei ist zu beachten, dass zum Konzept der beschriebenen Agenten, die Möglichkeit des Lügens gehört. Agenten sollen die Option besitzen, Aussagen auf Fragen, die die Erfüllung eigener Ziele verhindern, falsch zu tätigen oder zu verweigern. Die relevanten Konzepte zum Lügen finden sich im Abschnitt 2.5. Es sei an dieser Stelle darauf hingewiesen, dass die Generierung von Lügen kein triviales Problem darstellt. Zudem ist das Generieren von Lügen eng mit den Operatoren für Wissen und Kommunikation verbunden.

Ein weiterer Schwerpunkt der Arbeit der Projektgruppe ist die Modellierung. Modelliert wird dabei nicht nur das Wissen der Agenten als logische Regeln und Fakten, sondern auch die Umwelt, in der die Agenten sich bewegen. So wurden auf der Basis des

Kriminalromans Szenarien entwickelt, in denen die wichtigen Handlungsstränge des Romans modelliert sind. Der Roman bietet in diesem Zusammenhang den Vorteil, dass die meisten Aktionen der Protagonisten aus interaktiver, verbaler Kommunikation bestehen und sich nicht auf die Umwelt auswirken. Szenen finden an einem Ort statt und Orte bestehen aus Gegenständen. Die Modellierung eines Szenarios umfasst unter anderem Gegenstände mit *Attributen* und *Contains-Beziehungen*. Die Kapitel 3 und 4 beschreiben die Modellierung, sowohl der Umwelt in XML, als auch die Modellierung der Agenten in XML und DLV. So stehen in Kapitel 4 eher der *Inferenzbaum* und die Modellierung in DLV im Mittelpunkt, während in Kapitel 3 Wissensoperatoren und Zustände der Agenten und deren Umgang mit Unwahrheit thematisiert wird. Der *Inferenzbaum* ist die bildliche Darstellung sämtlicher Inferenzmöglichkeiten, und stellt somit die Abfolge möglicher Schlußfolgerungsergebnisse anschaulich dar.

Neben den oben beschriebenen Schwerpunkten, wie die Konzipierung einer Agentenarchitektur, ist die technische und praktische Umsetzung ein weiterer Schwerpunkt. Da es noch sehr wenig praktische Arbeiten gibt, die sich mit der Wissensmodellierung- und Verarbeitung beschäftigen, ist die technische Umsetzung Gegenstand der Ausarbeitung. In den Kapiteln 2 und 5 werden die Konzepte und ihre Umsetzung beschrieben. Die Realisation der genannten Ziele und Vorstellungen ist KIMAS (*Knowledge in Multiagentsystems*) . KIMAS verbindet die Multiagentenplattform JADEX mit DLV (siehe[EFLP00]). JADEX wurde in JAVA implementiert und dient als Werkzeug, um Multiagentensysteme komfortabel entwickeln zu können. Standardtechnologien, wie JAVA und XML, und die Anlehnung an das BDI-Modell bieten eine Menge Vorteile. DLV ist eine von der TU Wien entwickelten Inferenzmaschine, die aus entsprechend logisch modelliertem Wissen Schlussfolgerungen ziehen kann. Somit bietet es notwendige Funktionen, wie zum Beispiel die Unterstützung von Logik und Inferenzen an. Das logische Format ist die ANTWORTMENGEN-PROGRAMMIERUNG (ASP). Die Vorteile der ANTWORTMENGEN-PROGRAMMIERUNG sind strikte Negation und *Default* Negation. Insbesondere *Default* Negation ermöglicht nicht-monotone Inferenz und (un)sicheres Wissen. Auf den Roman bezogen bedeutet nicht-monotone Inferenz, dass ein Agent die Menge seines Wissens zwar vergrößert, aber die Menge der daraus resultierenden Schlussfolgerungen verkleinert. So kann zum Beispiel das falsche Alibi eines vermutlichen Verdächtigen die Wissensmenge vergrößern, aber die möglichen Inferenzen, wie zum Beispiel Schlussfolgerungen zum Tathergang, ausschliessen.

Die Agenten und Szenarien werden durch *XML*-Dateien spezifiziert. Weiterhin ist das Zusamenspiel zwischen DLV, JADEX-*Standalone*, *GUI* und Kernimplementation so organisiert, daß die Entwicklung eigener Szenarien, also der Modellierung eigener Umfelder, durch Benutzer so einfach wie möglich bleibt.

Die gesamte praktische Realisierung des Projekts wurde unter genauen theoretischen Vorgaben seitens der PG-Leitung gebunden. So ist das Ziel des Projektes der PROJEKTGRUPPE 491 die Entwicklung eines Multiagentensystems (MAS), in dem Wissen behandelt wird. Dieses System soll in einem Szenario die Verarbeitung und den Austausch von Wissen und Information leisten, so dass alle Agenten in der Lage sind, ihre Informationen mit eigenen, untereinander getrennten Wissenszuständen zu aktualisieren. In diesem Zusammenhang muss ein präziser Umgang von subjektivem (eigens aus den

Szenarien inferiertem) und objektivem (aus Fakten und Beobachtungen entstandenem) Wissen, geschaffen werden. Die Verarbeitung und Aktualisierung des durch Informationen erlangten neuen Wissens in die *Wissenszustände* durch *Wissensoperationen* soll durch *Wissensupdate- und Revision* realisiert werden. Die notwendige *Nichtmonotonie* von Operationen auf diesem Wissen, die komplexe Wissenverarbeitung erst ermöglicht, geht mit *Wissensupdate- und Revision* einher (beide Theorien sind eng miteinander verknüpft, siehe dazu [Gä94]). Dieser Bereich der Operationen der Wissensänderungen, wird und wurde von vielen Autoren untersucht (siehe [KM91a] und [KM91b]). Prinzipiell arbeitet die Revision auf dem abgeleiteten Wissen und nicht dem Basiswissen, so dass aus diesem Fakt resultierend, das Basiswissen nicht verändert wird. Das Update ist weitreichender und zieht Veränderungen auch im Basiswissen mit sich. Zudem ist die Realisierung solcher Operationen ein weiteres elementares Ziel. Vorgegeben ist weiterhin das BDI-Modell für die Agentenarchitektur und die Fähigkeit der Agenten unsicheres Wissen, und damit auch Falschinformationen, zu erfassen. Die daraus resultierende Notwendigkeit, Glaubwürdigkeiten einzelner Agenten neu zu beurteilen, also die Konzipierung der Wissensdynamik unter Berücksichtigung der Unsicherheit von Wissen und Information, ist somit auch vorgegeben.

Die letzte Komponente ist die Oberfläche und Funktionalität. So können eigene oder von der PROJEKTGRUPPE 491 entwickelte Szenarien in KIMAS angeschaut werden. Es ist möglich, alle internen Wissenszustände der Agenten, die Kommunikation der Agenten und auch systeminterne Nachrichten zu erfassen. Weiterhin ist die Kommunikation der Agenten untereinander grafisch dargestellt. Die grafische Darstellung und die Formatierung der Kommunikation sollen dem menschlichen Bedürfnis vermeindliche intelligente Muster, als menschlich und auch intelligent zu betrachten, entgegenkommen. Dem Betrachter von KIMAS soll der Eindruck vermittelt werden, in den Agenten mehr als nur eine Menge von disjunkten Softwareeinheiten zu sehen, die in einem abgeschlossenen Universum logische Programme ausführen. Dieses Phänomen ist durchaus bekannt, siehe dazu [J.77]. Es schafft im Umgang mit MA-Systemen, vielleicht ähnlich wie bei der Entwicklung von Robotern, eine ambivalent geprägte Wahrnemung. Es treffen allgemein psychologische und linguistische Fragen von Kognition und Kommunikation auf theoretische Fragen der Mathematik und Informatik. So ist zum Beispiel die Frage, wie es um das Lügen des Menschen und das Lügen eines Agenten bestellt sei, durchaus schwierig und nicht allgemeingültig zu beantworten. Da Agenten zwar auf den Wahrheitsgehalt empfangender Nachrichten entsprechend reagieren müssen, aber nicht wie Menschen bewußt und manipulativ lügen können, ist es nicht möglich, Begrifflichkeiten verschiedener Disziplinen ohne weiteres zusammenzulegen. Es ist vereinbartes Ziel, bei der Modellierung der Szenarien möglichst nah an der Romanvorlage von Agatha Christies *"The Mysterious Affair At Styles"* [Chr99] zu bleiben. Die programmierten Agenten vollführen interaktive Kommunikation, besitzen eigenes Wissen und Ziele und planen ihre Vorgehensweise semiautonom. Auch die Darstellung durch die graphische Oberfläche, soll beim Rezipienten das Gefühl von erlebter Kommunikation, im Sinne von nicht maschineller Kommunikation, erzeugen.

2 Konzeptualisierung

2.1 Überblick

Ulrich Engler

Um ein wie in Kapitel 1 beschriebenes Multiagentensystem zu erstellen bedarf es offenkundig geeigneter theoretischer Konzepte und der entsprechenden Umsetzung mittels der uns zur Verfügung stehenden Komponenten, wie `DLV` `Jadex` `XML` oder `JAVA`. Diese Konzepte werden in diesem Kapitel vorgestellt und näher beschrieben.

So benötigt man ein Konzept zum Nachrichtenaustausch zwischen den einzelnen Agenten, um ihr Wissen auszutauschen, wobei hier zum einen der technische Aspekt ein Rolle spielt hinsichtlich der praktischen Kommunikationsmöglichkeiten und zum anderen der Inhalt der Nachrichten. So soll es den Agenten auf Basis ihres Wissens möglich sein entscheiden zu können welche Informationen sie preisgegen wollen oder ob sie gar falsche Informationen schicken, also lügen sollen. Zudem muss auch mittels der Logik auf zu stellende Fragen geschlossen werden können.

Um erlangte Informationen bzw. Wissen auch wie gefordert verarbeiten und bewerten zu können, bedarf es *Wissensoperatoren*, die anhand von Glaubwürdigkeiten und dem Alter der Informationen diese mit dem eigenen Wissen des Agenten vereinen müssen, da es durchaus vorkommen kann, dass sich Informationen, die von verschiedenen Quellen stammen sich widersprechen, wenn etwa ein Agent gelogen hat. Hier muss das Wissen also *revidiert* werden. Allerdings ist es ebenfalls denkbar, dass ein *Update* durchgeführt werden muss, wenn alte Informationen durch neue ersetzt werden müssen, da sich der Zustand der Umwelt in der Zwischenzeit geändert hat. Ziel ist es mit Hilfe des *Wissenoperators* das logische Programm so abzuändern, dass es `DLV` möglich ist den aktuellen korrekten Glaubenszustand zu inferieren.

Um mögliche *Lügen* überhaupt als solche entlarven zu können, muss zum einen ein Widerspruch zu einer Information durch eine weitere Quelle bestehen und zum anderen entschieden werden, welche der beiden sich widersprechenden Informationen als falsch und damit als gelogen angesehen werden kann oder sogar muss. Als Basis eines solchen Vergleichs dienen bei uns *Vertrauenswürdigkeiten* der Quellen. Anhand dieser Werte wird nun entschieden welche Informationen geglaubt werden sollen und ob die Quelle der falschen Informationen als Lügner bezeichnet und gegebenenfalls bestraft werden soll. Eine solche Bestrafung könnte in der Abstufung der *Vertrauenswürdigkeit* enden. Eine weitere Herausforderung ist in diesem Zusammenhang natürlich auch die notwendige Fähigkeit der Agenten überhaupt lügen zu können. Daher bedarf es auch hier eines geeigneten Mechanismusses um festzustellen, wann gelogen oder nur geschwiegen werden soll und durch was die richtigen Informationen gegebenenfalls ersetzt werden müssen.

Innerhalb dieses Kapitels wird zudem das Modell des Agenten selbst thematisiert und das Konzept des *Know-Hows* vorgestellt. Mit Hilfe des *Know-Hows* werden den Agenten Möglichkeiten des Handelns und Vorgehens in bestimmten Situationen zur Verfügung gestellt, so dass er seine Ziele erfolgreich abschließen kann. Außerdem wird noch die bei uns zur besseren Übersichtlichkeit und zur Strukturierung der Modellierung verwendete Einteilung in Szenen beschrieben.

2.2 Logikbasierte Fragen und Antworten

Matthias Thimm

2.2.1 Motivation

Die wichtigsten Aktionen der zu betrachtenden Agenten, sind Kommunikationsaktionen und somit im wesentlichen Fragen, Antworten und allgemeine Mitteilungen. Jede dieser Aktionen überträgt eine Menge von logischen Informationen von einem Agenten zu einem anderen und es ist nötig diese Informationen entsprechend darzustellen, damit ein Agent zwischen eigenem Wissen und Wissen, das er durch Informationsaustausch mit anderen Agenten erhalten hat, unterscheiden kann. Weiterhin ist es möglich, dass manche Agenten lügen (siehe Abschnitt 2.5), und somit ist eine eventuell umfangreiche *Deliberation* über die erhaltenen Informationen erforderlich, bevor diese in die Wissensmenge eines Agenten gelangen können.
Aus diesen Gründen ist es nötig, logische Kommunikationsprogramme zu formalisieren und zu strukturieren. Wie schon eingangs erwähnt, unterscheiden wir drei Grundtypen von logischen Kommunikationsprogrammen:

- **Frage**: Eine Frage zielt darauf ab, bestimmte Informationen von einem bestimmten Agenten zu erhalten. Jeder Agent soll in der Lage sein, Fragen aus seiner Wissensbasis zu generieren, um sein Wissen zu vervollständigen.
- **Antwort**: Eine Antwort wird genau einer Frage zugeordnet und enthält die angefragten Informationen. Diese Informationen müssen nicht der Wahrheit entsprechen und leiten sich von der Wissensbasis und der Motivation des Befrageten ab.
- **Allgemeine Mitteilung**: Eine allgemeine Mitteilung ist ein logisches Kommunikationsprogramm, welches Informationen enthält, die nicht explizit erfragt wurden.

Diese Grundtypen werden im Folgenden genauer spezifiziert und an Beispielen erklärt.

2.2.2 Logische Fragen

Fragetypen Wir unterscheiden folgende Typen von Fragen:

- **Instanziierungsfrage**: Bei diesem Fragetyp wird dem Befragten ein Prädikatbezeichner übergeben und dazu aufgefordert, dieses Prädikat mithilfe seiner Wissensbasis zu instanziieren.

> **Beispiel** Angenommen Agent a möchte von Agent b wissen, wer alles Krankenschwester ist. Zu diesem Zweck kann er b eine Instanziierungsfrage über das Prädikat Nurse/1 stellen. Falls Agent b über entsprechendes Wissen verfügt, könnte er mit Nurse(evie) antworten.

- **Beschreibungsfragen**: Bei diesem Fragetyp wird kein Prädikatbezeichner, sondern ein Objektbezeichner übergeben und der Befragte dazu aufgefordert, instanziierte Prädikate zurückzuliefern, die das angeforderte Element enthalten.

  > **Beispiel** Nachdem Agent a erfahren hat, das Agent evie von Beruf Krankenschwester ist, möchte er mehr über sie erfahren. Er stellt somit Agent b eine Beschreibungsfrage zu evie. Agent b könnte nun als eine adequate Antwort Attractive(evie) zurückliefern.

- **Ja-Nein-Frage**: Falls ein Agent ein bestimmtes Faktum auf seinen Wahrheitsgehalt prüfen möchte, so kann er eine Ja-Nein-Frage stellen. Dabei wird dem Befragten eine konkrete Prädikatinstanziierung übergeben, der dieser zustimmen oder widersprechen soll.

  > **Beispiel** Agent a glaubt nun, dass Agent evie die Frau seines Lebens ist, möchte aber um einer Entäuschung aus dem Weg zu gehen, Agent b zunächst befragen ob die Dame auch ledig ist. Er stellt dazu Agent b eine Ja-Nein-Frage und übergibt die Prädikateninstanziierung Single(evie). Agent b muss Agent a nun enttäuschen und gibt aufgrund der Informationen in seiner Wissensbasis die Antwort no.

Aufbau von logischen Fragen Fragen werden als eine Menge von Fakten in einem erweiterten logischen Programm dargestellt. Die Tatsache, dass Fragen als Fakten dargestellt werden, bedeutet jedoch nicht, dass die darin enthaltenen Informationen als sicher gelten. Vielmehr bedeutet die Darstellung als eine Menge von Fakten, dass die Fragestellung an sich ein Faktum ist – die Frage wurde sicher gestellt. In späteren Inferenzen kann aus den Informationen aus Fragen und Antworten in Abhängigkeit von der Vertrauenswürdigkeit des Befragten Wissen abgeleitet werden.
Eine logische Frage besteht zunächst aus Fakten von Prädikaten Query/1, Query_Type/2,Query_Sender/2, Query_Arity/2 und Query_Content/3. Fragen werden mithilfe von Reifikation dargestellt und somit besitzt jede Frage einen eindeutigen Bezeichner.

- Query(x): Dies ist die initiale Fragendefinition. Ein Bezeichner x wird hiermit als Fragebezeichner definiert, um in weiteren Fakten genauer spezifiziert zu werden.

- Query_Type(x,q_type): Wie schon oben erwähnt betrachten wir drei verschiedene Grundtypen von Fragen: Instanziierungsfragen, Beschreibungsfragen und Ja-Nein-Fragen. Der Grundtyp einer Frage x wird mit Query_Type(x,q_type) festgelegt, wobei für q_type die drei möglichen Ausprägungen q_instantiate (für Instanziierungsfragen), q_element_info (für Beschreibungsfragen) und q_yes_no (für Ja-Nein-Fragen) zur Verfügung stehen.

- Query_Sender(x,q_sender): Jede Frage wird einem eindeutigen Fragensteller zugeordnet, der in diesem Faktum durch q_sender festgelegt ist.

- Query_Arity(x, n) und Query_Content(x, i, c): In diesen beiden Prädikaten wird der eigentliche Inhalt der Frage dargestellt. Manche Fragen können mehr als einen Parameter besitzen – so muss beispielsweise für eine Ja-Nein-Frage ein instanziiertes Prädikat übergeben werden, wobei sowohl der Prädikatbezeichner, als auch jedes Argument als ein eigener Parameter in der Frage spezifiziert werden muss. Aus diesem Grund wird mit Query_Arity(x, n) zunächst die Anzahl der Query_Content-Fakten angegeben, dabei entspricht x wieder dem Fragebezeichner und n ist eine natürliche Zahl, die die Anzahl der übergebenen Query_Content-Fakten entspricht. Für jedes i $\in \{1, ..., n\}$ existiert dann im Frageprogramm ein Faktum Query_Content(x,i,e) wobei auch hier x wieder für den Fragebezeichner steht, i den Index des Inhaltselementes angibt und e der i-te Parameter der Frage ist. Für die einzelnen Fragegrundtypen haben die Inhaltselemente ein andere Semantik, welche im Folgenden genauer beschrieben wird.

Beschreibungs- und Instanziierungsfragen enthalten nur genau einen Parameter. Beschreibungsfragen fordern den Befragten dazu auf, Wissen über einen bestimmten Objektbezeichner zu liefern und somit muss auch nur dieser Bezeichner übergeben werden. Analog dazu fordern Instanziierungsfragen zur Instanziierung eines Prädikats und somit muss auch nur der Prädikatenbezeichner übergeben werden. Im Falle von Instanziierungsfragen ist allerdings eine Anpassung des Parameters an die Syntax der erweiterten logischen Programmierung nötig, da Prädikate nicht geschachtelt werden dürfen. Jedes Prädikat erhält dafür einen eindeutigen Objektbezeichner, der in logischer Kommunikation stellvertretend für das eigentliche Prädikat steht. Nach Konvention werden Prädikatenbezeichner durch ein vorangestelltes p_ an ihren in Kleinbuchstaben geschriebenen Namen dargestellt. Ein Prädikat Nurse hat somit den Prädikatenbezeichner p_nurse. Da in beiden Fällen stets nur ein Parameter geliefert wird, enthält das Frageprogramm auch ein Faktum Query_Arity(x,1), wenn x der Bezeichner der Frage ist.
Ja-Nein-Fragen enthalten im Allgemeinen jedoch mehrere Parameter. Soll beispielsweise als Parameter einer Ja-Nein-Frage das Faktum Chasing(tom, jerry) übergeben werden, so ist eine Kodierung der einzelnen Parameter Chasing, tom und jerry nötig. Wie im Fall der Instanziierungsfrage, muss auch hier das Prädikat durch Verwendung der oben beschriebenen Prädikatenbezeichner speziell kodiert werden, die Argumente des Faktums können jedoch direkt in die Argumentliste der Query_Content-Fakten geschrieben. Bei der Kodierung der Parameter von Fragen ist die Reihenfolge der Argumente strikt einzuhalten, der erste Parameter ist stets der Prädikatenbezeichner, gefolgt von den übrigen Argumenten in der natürlichen Reihenfolge. Die obige Frage würde beispielsweise die Fakten Query_Arity(x, 3), Query_Content(x,1,p_chasing), Query_Content(x,2,tom) und Query_Content(x,3,jerry) enthalten.

> **Beispiel 1**. Agent tom möchte eine logische Instanziierungsfrage nach dem Prädikat Mouse/1 stellen. Die Kodierung dieser Frage in einem erweiterten logischen Programm ist

```
Query(q1).
Query_Type(q1, q_instantiate).
Query_Sender(q1, tom).
Query_Arity(q1, 1).
Query_Content(q1, 1, p_mouse).
```

Beispiel 2. Agent tom möchte eine logische Beschreibungsfrage über den Agenten jerry stellen. Die Kodierung dieser Frage in einem erweiterten logischen Programm ist

```
Query(q2).
Query_Type(q2, q_element_info).
Query_Sender(q2, tom).
Query_Arity(q2, 1).
Query_Content(q2, 1, jerry).
```

Beispiel 3. Agent tom möchte eine logische Ja-Nein-Frage über das Faktum Location(jerry, mousehole) stellen. Die Kodierung dieser Frage in einem erweiterten logischen Programm ist

```
Query(q3).
Query_Type(q3, q_yes_no).
Query_Sender(q3, tom).
Query_Arity(q3, 3).
Query_Content(q3, 1, p_location).
Query_Content(q3, 2, jerry).
Query_Content(q3, 3, mousehole).
```

Es dürfte auffallen, dass in der Kodierung einer Frage der Adressat der Frage nicht spezifiert wird. Wie gehen zunächst vereinfachend davon aus, dass es für die Erkenntnis, dass eine Frage gestellt wurde, nur wichtig ist, wer die Frage gestellt hat und nicht wem. Falls eine Frage jedoch beantwortet wird, ist es von höchstem Interesse, wer die Frage **beantwortet** hat. Um die Problematik der logischen Antworten geht es im nächsten Abschnitt.

2.2.3 Logische Antworten

Antworttypen Genau wie Fragen, haben auch Antworten einen gewissen Grundtyp. Da Antworten sich jedoch immer direkt auf eine Frage beziehen, erben diese den Typ der Frage. Eine Antwort auf eine Ja-Nein-Frage soll somit stets *Ja* oder *Nein* sein und niemals tweety. Aus diesem Grund wird bei der Definition einer Antwort als erweitertes logisches Programm der Bezeichner der Frage als Parameter übergeben, um den Bezug der Antwort zu der Frage sicherzustellen. Antworten sollen somit stets die mit ihnen verbundene Frage syntaktisch korrekt beantworten. Der Wahrheitsgehalt der in der Antwort enthaltenen Information muss dabei jedoch nicht immer korrekt sein.

Aufbau von logischen Antworten Genau wie Fragen werden auch Antworten als eine Menge von Fakten eines erweiterten logischen Programmes dargestellt. Die Antworten enthalten stets nur die Informationen, die zur Beantwortung der Frage nötig sind und somit beispielsweise keine Informationen aus der Frage selbst. Beispielsweise übergibt eine logische Beschreibungsfrage einen Objektbezeichner, welcher jedoch in einer korrekten Antwort nicht wiederholt wird. Dadurch ist eine realistischere Darstellung von Kommunikation möglich, da auch bei einer Frage wie etwa „Was kannst Du mir über Evie sagen?“ sinnvollerweise als Antwort „*Sie* ist Krankenschwester“ und im Allgemeinen nicht „Evie ist Krankenschwester“ übergeben wird. Aus demselben Grund enthalten Fragen und Antworten auch keine Information über den jeweiligen Adressaten. Logische Fragen und Antworten stellen nur das Wissen der jeweiligen Aussage dar und bieten keine extra-logischen Informationen. Stellt beispielsweise in einem geschlossenen Raum ein Agent A einem Agenten B eine Frage, so darf es i. A. einem Agenten C ausserhalb des Raumes nicht möglich sein, allein aus der Frage des A Information über die Identität des B zu erhalten.

Genau wie Fragen, werden auch Antworten durch Reifikation dargestellt und erhalten einen eindeutigen Bezeichner. Logische Antworten werden durch folgende Prädikate beschrieben:

- Answer(a,q,t): Hiermit wird festgelegt, dass der Bezeichner a eine Antwort auf die Frage mit dem Bezeichner q darstellt. Der Parameter t gibt dabei den Zeitindex der Antwort an, welcher bei der Analyse der Antwort hilfreich sein kann.
- Answer_Sender(a, s): Jede Antwort a wird einem eindeutigen Absender s zugeordnet.
- Answer_Arity(a, n) und Answer_Content(a,i,c): Analog wie bei Fragen, kann eine Antwort mehrere Inhaltselemente besitzen.

Bei Beschreibungs- und Instanziierungsfragen wird in der zugehörigen logischen Antwort stets ein instanziiertes Prädikat übergeben. Die Darstellung mit Answer_Arity- und Answer_Content-Prädikaten entspricht dabei genau der Darstellung von Query_Arity- und Query_Content-Prädikaten bei Ja-Nein-Fragen. Falls ein Agent zu einer Frage mehr als ein instanziiertes Prädikat als Antwort zurückliefern möchte, so ist für jedes Prädikat eine eigene Menge von Answer, Answer_Sender und $Answer_Content$ Fakten nötig.

Die Antwort zu einer Ja-Nein-Frage besteht stets aus nur einem Parameter a_yes oder a_no, je nachdem ob die Antwort „Ja“ oder „Nein“ lautet.

Bezugnehmend auf die Beispielfragen des vorherigen Abschnitts, könnten folgende Antworten gegeben werden:

> **Beispiel 1**. Die Frage von Agent tom „Wer ist eine Maus?“, könnte durch Agent jerry folgendermaßen beantwortet werden:
>
> ```
> Answer(a1, q1, 43).
> Answer_Sender(a1, jerry).
> Answer_Arity(a1, 1).
> Answer_Content(a1, 1, speedy_gonzalez).
> ```

Beispiel 2. Die Frage von Agent tom „Was weißt Du über Jerry?“, könnte durch Agent spike folgendermaßen beantwortet werden:

```
Answer(a2, q2, 44).
Answer_Sender(a2, spike).
Answer_Arity(a2, 3).
Answer_Content(a2, 1, p_likes).
Answer_Content(a2, 2, jerry).
Answer_Content(a2, 3, goldy).
```

Beispiel 3. Die Frage von Agent tom „Ist Jerry in seinem Mauseloch?“, könnte durch Agent spike folgendermaßen beantwortet werden:

```
Answer(a3, q3, 45).
Answer_Sender(a3, spike).
Answer_Arity(a3, 1).
Answer_Content(a3, 1, a_no).
```

2.2.4 Weitere Logische Informationen

Das *Inform*-Prädikat Nicht immer werden Informationen in Reaktion auf eine Frage mitgeteilt. Genauso wie Fragen proaktiv gestellt werden können, können auch allgemeine Mitteilungen jederzeit von einem Agenten zu einem anderen geschickt werden. Da sich diese Informationen nicht unbedingt auf eine andere Information beziehen, müssen diese stets eine vollständige Instanziierung eines Prädikats beeinhalten. Insofern gleicht die Struktur von allgemeinen Mitteilungen, die mit Hilfe von Inform-Prädikaten dargestellt werden, der von Antworten zu Instanziierungs- und Beschreibungsfragen. Es entsprechen somit die Verwendung der Prädikate Inform_Arity/2, Inform_Sender/2 und Inform_Content/2 entsprechen somit der vorhergehenden Erklärung zu den entsprechenden Answer-Prädikaten. Wie Fragen und Antworten werden also auch allgemeine Mitteilungen durch Reifikation dargestellt und mittels des Prädikats Inform(x,t) wird ein Bezeichner x als allgemeine Mitteilung klassifiziert und einem Zeitpunkt t zugeordnet.

Beispiel. Agent tom will Agent jerry aus seinem Mauseloch herauslocken und schickt ihm dazu folgende Mitteilung:

```
Inform(i1, 46).
Inform_Sender(i1, tom).
Inform_Arity(i1, 3).
Inform_Content(i1, 1, p_location).
Inform_Content(i1, 2, cheese).
Inform_Content(i1, 3, kitchen).
```

2.3 Wissenszustand

Patrick Krümpelmann

Das Wissen eines Agenten teilt sich auf in initiales Wissen, über das er zu Beginn seiner Aktivierung verfügt, und über Wissen, das er erst während der Laufzeit durch die logikbasierte Kommunikation, wie im vorangegangenen Abschnitt vorgestellt wurde, akkumuliert. Während andere Ansätze von Wissensrevision und -update neue Informationen bei Erhalt in die Wissensbasis integrieren, werden bei unserem Ansatz neue Informationen zunächst in einem Historie-Objekt gespeichert.

In dieser Informations-Historie wird jede von einem anderen Agenten erhaltene Information gespeichert. Die Information wird in Form eines logischen Programms erhalten und zusätzlich zu diesem wird die Quelle dieser Information, d.h. der Agent von dem die Nachricht erhalten wurde, gespeichert und den Zeitpunkt des Erhaltes der Information.

Teil des initialen Wissen eines Agenten ist eine Tabelle von Glaubwürdigkeiten welche jedem dem Agenten bekannten Agenten eine initiale Glaubwürdigkeit zuordnet. Diese Tabelle muss weder vollständig noch statisch sein.

Durch diese Aufteilung des Wissens eines Agenten geht keine Information verloren und es kann stets nachvollzogen werden, wie der Wissenszustand des Agenten zu einem bestimmten Zeitpunkt ausgesehen hat. Ist eine Betrachtung des aktuellen Wissenszustandes nötig, so wird mit Hilfe eines Wissensoperators aus der initialen Wissensbasis und der protokollierten Informationen aus dem Historie-Objekt der derzeitige Wissenszustand generiert. Wir haben uns für diese Art der Protokollierung der Informations-Historie entschieden, um eine größtmögliche Flexibilität bei den Wissensoperatoren zu erhalten.

Wir definieren nun Begriffe und die verschiedenen Wissensformen zu charakterisieren. Die gerade angesprochene Informations-Historie bildet zusammen mit dem initialen Wissen die *Belief-Base* eines Agenten. Durch Anwendung eines Wissensoperators wird aus der *Belief-Base* ein einziges logisches Programm konstruiert. Dieses logische Programm stellt den *Belief-State* eines Agenten dar. Der *Belief-State* wird durch einen *ASP-Reasoner* ausgewertet und eine Antwortmenge erzeugt. Diese Antwortmenge stellt nun die aktuellen *Beliefs* des Agenten dar.

2.4 Wissensoperatoren

Patrick Krümpelmann

Die Darstellung von logikbasierter Kommunikation, wie sie vorgestellt wurde, ist noch nicht ausreichend, um die in den logischen Kommunikationsprogrammen enthaltenen Informationen auch zu verarbeiten. Hat ein Agent einen Dialog mit Fragen und Antworten protokolliert, so kann er das erhaltene Wissen noch nicht vergleichen und neues Wissen ableiten. Es sind also noch spezielle Operationen erforderlich, die die Informationen in den Kommunikationsprogrammen aufbereiten, auf Konsistenz prüfen und nach bestimmten Kriterien dem *Belief-State* hinzufügen. Diese Operationen leisten Wissensoperatoren, welche auf unterschiedliche Weise die *Belief-Base* in den *Belief-State* überführen. Ein weiterer Vorteil von Wissensoperatoren ist, dass eine Wissensoperation nicht generisch für

alle Agenten des Systems sein muss. Verschiedene Wissensoperatoren bieten die Möglichkeit, jeden Agenten mit einem individuellen Inferenzmechanismus auszustatten, der seinem Typ entspricht. Die Austauschbarkeit von Wissensoperatoren gewährleistet dabei die nötige Flexibilität.

2.5 Erkennung und Behandlung von Lügen

Jens Eckstein

2.5.1 Problemstellung

Da das Lügen und Täuschen bei den Menschen zu finden sind, müssen die Agenten, die das Verhalten der Menschen simulieren oder nachahmen, die Fähigkeit haben, lügen oder täuschen zu können. Ein Grund für das Lügen eines Agenten ist die Manipulation eines anderen Agenten. Ein Agent kann falsche Informationen über sein Wissen oder seine Ziele liefern, mit dem Zweck das Verhalten eines anderen Agenten zu beeinflussen. Ein zweiter Grund für das Lügen ist das Verheimlichen von Informationen gegenüber den anderen Agenten. Wenn ein Agent sein Wissen oder seine richtigen Ziele nicht an andere Agenten verraten will, kann er falsche Informationen an diesen Agenten weitergeben. In diesen Fällen kooperieren die Agenten nicht. Sie sind in einer Umgebung, wo jeder Agent seine Interssen verfolgt. Auf der anderen Seite könnte ein Agent ohne Absicht lügen, wenn er selbst aus irgend einem Grund (Kommunikation mit anderen Agenten, falsche Beobachtungen, etc.) falsche Informationen hat. Dies kann der von uns benutzte *Wissensoperator* aber nicht unterscheiden, deshalb haben wir in der Modellierung darauf geachtet, dass ein solcher Fall nicht eintritt.

Wenn die Agenten in einer Umgebung lügen können, kommt es vor, dass ein Agent wiedersprüchliche Informationen bekommt. Welche Informationen soll er glauben? Als Antwort auf diese Frage wurden verschiedene Theorien entwickelt. Alle diese Theorien basieren auf der Vertrauenswürdigkeit eines Agenten einem anderen gegenüber, deswegen spielt das Vertrauen zwischen den Agenten eine zentrale Rolle im Multiagentensystem. Ein Beispiel für den Vertrauensaufbau zwischen den Agenten ist das reputation management (Ruf-Management) Prinzip. Das Reputation-Management-Prinzip wird bei vielen Internetseiten verwendet, wie zum Beispiel bei e-bay, die Partner in einer Geschäftsabwicklung können sich gegenseitig bewerten. Die anderen Benutzer des Systems können diese Bewertung lesen und einen Überblick für sich schaffen, ob sie einem Nutzer vertrauen oder nicht vertrauen.

Der von uns gewählte Ansatz basiert aber darauf, dass Agenten zunächst einmal untereinander eine hohe Vertrauenswürdigkeit besitzen, welche dann, wie im Folgenden beschrieben, über die Zeit abnehmen kann, wenn sich die Informationen eines Agenten als unzuverlässig erweisen.

Im folgenden werden zwei Ansätze für das Lügen behandelt. Anschließend wird auf die Vertrauenswürdigkeit zwischen den Agenten eingegangen.

2.5.2 Der Ansatz des (erweiterten) *SimpleOperator*

Der *SimpleOperator*, als der einfachere der beiden von uns entwickelten Wissenoperatoren, ist nicht in der Lage, individuelle Lügen zu erkennen, da er stets Agenten als nur vertrauenswürdig oder nicht vertrauenswürdig einstuft und somit die Akzeptanz von Aussagen nur nach diesen beiden Klassen entscheidet. Falls ein vertrauenswürdiger Agent lügt, wird dies trotzdem geglaubt. Wir betrachten im Folgenden eine kleine Erweiterung des *SimpleOperator*, der eine genauere Einteilung von Vertrauenswürdigkeitsklassen erlaubt. Dazu wird das Prädikat Reliable/1 durch das Prädikat Reliable/2 mit der Bedeutung

Reliable(A,Z): Agent A hat einen Vertrauenswürdigkeitswert von Z

ersetzt. Mithilfe einer Menge von Fakten dieses Prädikats können alle bekannten Agenten in beliebig viele Vertrauenswürdigkeitsklassen eingeteilt werden. Um weiterhin zwischen widersprüchlichen Aussagen verschiedener Agenten zu entscheiden, müssen die Hold-Prädikate[1] erweitert werden, dass sie auch einen Verweis auf den jeweiligen Sender enthalten. Für jede Arität T von Prädikaten, gibt es somit ein Prädikat HoldT(P,X1,..,XT,S), wobei das zusätzliche Argument S den Sender der Information bezeichnet. Sind von logischen Dialogen diese Hold-Fakten extrahiert worden und widersprüchlich, so kann durch einen Vergleich der Vertrauenswürdigkeitswerte, die plausiblere Aussage geglaubt und die andere Aussage als Lüge betrachtet werden.

Beispiel 1. Angenommen Agent jerry möchte Agent tom der Lüge überführen und stellt sowohl Agent tom als auch Agent goldy die Frage, ob in der Küche Käse ist:

```
Query(q1).
Query_Type(q1, q_yes_no).
Query_Arity(q1, 3).
Query_Content(q1, 1, p_location).
Query_Content(q1, 2, cheese).
Query_Content(q1, 3, kitchen).
```

Weiterhin enthalte die Wissensbasis von Agent jerry eine Reihe von Vertrauenswürdigkeitsfakten, unter anderem beispielsweise

```
Reliable(tom, 55).
Reliable(goldy, 90).
```

Auf die obige Frage erhält Agent jerry folgende Antwort von Agent tom

```
Answer(a1, q1, 76).
Answer_Sender(a1, tom).
Answer_Arity(a1, 1).
Answer_Content(a1, 1, a_yes).
```

und folgende Antwort von Agent goldy:

[1] Hold-Prädikate werden benötigt, um die logikbasierten Antwortprogramme (vgl. Kapitel 2.2) wieder in normale logische Fakten zurück zu transformieren. Siehe auch Kapitel 3.1.2. Das erweiterte Hold-Prädikat ist auf Seite 20 angegeben.

```
Answer(a2, q1, 77).
Answer_Sender(a2, goldy).
Answer_Arity(a2, 1).
Answer_Content(a2, 1, a_no).
```

Diese beiden Antworten enthalten offensichtlich widersprüchliche Informationen: Agent tom behauptet, dass in der Küche Käse sei, während Agent goldy das Gegenteil behauptet. Um nun aus diesen beiden Aussagen die plausibelste zu akzeptieren, werden zunächst entsprechende Hold-Fakten erzeugt

```
 Hold2(P, X1, X2, S) :- Query(Q),
                        Query_Type(Q, q_yes_no),
                        Query_Arity(Q, 3),
                        Query_Content(Q, 1, P),
                        Query_Content(Q, 2, X1),
                        Query_Content(Q, 3, X2),
                        Answer(A, Q, T),
                        Answer_Content(A, a_yes),
                        Answer_Sender(A, S).

-Hold2(P, X1, X2, S) :- Query(Q),
                        Query_Type(Q, q_yes_no),
                        Query_Arity(Q, 3),
                        Query_Content(Q, 1, P),
                        Query_Content(Q, 2, X1),
                        Query_Content(Q, 3, X2),
                        Answer(A, Q, T),
                        Answer_Content(A, a_no),
                        Answer_Sender(A, S).
```

Mithilfe dieser beiden Regeln werden die beiden Hold-Fakten

```
Hold2(p_location, cheese, kitchen, tom)
-Hold2(p_location, cheese, kitchen, goldy)
```

erzeugt. Um nun eine Lüge zu entdecken, müssen die Vertrauenswürdigkeitswerte der Agenten verglichen werden:

```
Liar(S1, P) :- Hold2(P, X1, X2, S1), Reliable(S1, T1),
               -Hold2(P, X1, X2, S2), Reliable(S2, T2),
               T2 > T1.

Liar(S1, P) :- -Hold2(P, X1, X2, S1), Reliable(S1, T1),
               Hold2(P, X1, X2, S2), Reliable(S2, T2),
               T2 > T1.
```

Mit anderen Worten: Falls Agent S1 sagt, dass P instanziiert mit X1 und X2 gilt und der vertrauenswürdigere Agent S2 sagt, dass dies nicht gilt, so lügt Agent S1 in Bezug

auf P. Die zweite Regel betrachtet den Fall der umgekehrten Negation.
Durch Anwendung der ersten Regel und der obigen Fakten wird das Faktum Liar(tom, p_location) generiert, womit zunächst feststeht, dass Agent tom in Bezug auf das Prädikat Location gelogen hat. Um nun noch die korrekte Antwort auf die Frage q1 in das aktuelle Wissen von Agent jerry zu übernehmen, sind folgende Regeln nötig:

```
 Location(X1, X2) :- Hold2(p_location, X1, X2, S),
                     not Liar(S, p_location).

-Location(X1, X2) :- -Hold2(p_location, X1, X2, S),
                     not Liar(S, p_location).
```

Es wird eine Aussage somit stets geglaubt, wenn es nicht bewiesen werden kann, dass der jeweilige Sender der Aussage ein Lügner in Bezug auf das enthaltene Prädikat ist.

2.5.3 Der Ansatz des UpdateOperators

Der *UpdateOperator* benutzt ein Verfahren, um eine Sequenz von erweiterten logischen Programmen zu einem einzigen, konsistenten logischen Programm zu vereinigen. Die Verarbeitung der Sequenz folgt dabei der Ordnung jener. Das heißt es wird in der Ordnung übergeordnetes Wissen als stärker verwurzelt angenommen. Bei dem Auftreten von Konflikten wird also höher geordnetes Wissen priorisiert.

In unserem Fall besteht die Sequenz aus dem initialen Wissen und den Programmen der Informations-Historie. In genau dieser Historie sind neben den eigentlichen logischen Programmen Zusatzinformationen gespeichert. Dies sind Informationen über den Sender der Information und den Zeitpunkt des Erhalts der Information. Diese Informationen lassen sich mit dem besprochenen Update-Mechanismus kombinieren (vgl. Kapitel 3.1) . Dabei werden die zu den Agenten gehörenden Glaubwürdigkeiten (siehe auch Kapitel 2.6) und die zeitliche Reihenfolge benutzt, um die Ordnung der Sequenz festzulegen.

```
reliabilities(
  <Mary, 40>
  <Dorcas, 80>
)

programs(
  program
    source: Mary
    timestamp
    content: {Clothes(mary, green)}
  program
    source: Dorcas
    timestamp
    content: {-Clothes(mary, green)}
)
```

Listing 2.1: Beispiel zur Speicherung von Zusatzinformationen

Es werden also die einzelnen Informationen, sprich die einzelnen Programme anhand der assoziierten Glaubwürdigkeiten und Zeitinformationen geordnet und Konflikte dadurch gelöst, dass übergeordnetes Wissen bevorzugt wird. Hierbei wird das initiale Wissen mit der höchsten Glaubwürdigkeit versehen und entsprechend maximales Element.

Ausgehend von der so geordneten Sequenz wird das den aktuellen Wissenszustand repräsentierende Programm erstellt. Hierbei wird das Alphabet erweitert, es werden Reject Prädikate eingefügt, welche Regeln blockieren, die im Konflikt mit höher geordnerten Programmen stehen. Für das resultierende Update Programm werden nun die Antwortmengen bestimmt. Taucht in einer Antwortmenge eines der Reject Prädikate auf, so kann daraus geschlossen werden, dass ein Konflikt vorlag. Anhand des Index des Reject Prädikates kann auch bestimmt werden, welche Regel aus welchem Programm blockiert wurde. Nun kann der zu dem Programm gehörende Sender bestimmt werden, dessen Information im Konflikt zu anderen Informationen stand und dessen Information aufgrund einer geringeren Glaubwürdigkeit zurückgewiesen wurde. Eine detailierte Beschreibung der Funktionsweise des *UpdateOperators* ist in Kapitel 3.1.3 nachzulesen.

Beispiel 2. In dem Beispiel aus Listing 2.1 gibt es zwei Programme, die Informationen von zwei verschiedenen Agenten mit unterschiedlicher Glaubwürdigkeit enthalten. Dabei ist *Dorcas* mit einer Glaubwürdigkeit von 80 der vertraunswürdigere Agent, weswegen das von ihr stammende Programm in der Ordnung der Programme über dem von *Mary* angeordnet wird. Daher würden wir folgendes Update Programm erhalten.

```
-Clothes_1(mary,green) :- not Rej(r1).
Clothes_2(mary,green).
Rej(r1) :- Clothes_2(mary,green).
Clothes_1(mary,green) :- Clothes_2(mary,green).
Clothes(mary,green) :- Clothes_1(mary,green).
```

DLV wird dazu folgende Update Antwortmenge zurückliefern:

```
{Clothes_2(mary,green), Rej(r1),
Clothes_1(mary,green), Clothes(mary,green)}
```

Das bedeutet, dass der Agent somit der Aussage von *Dorcas* glaubt und die Information von *Mary* nicht. Das macht Mary aber noch nicht automatisch zum Lügner. Um einen Agenten als Lügner zu bezichtigen, sind zwei Dinge erforderlich. Zum einen muss eine von ihm stammende Information in dem Update-Programm abgelehnt werden, zum anderen muss aber die gegensätzliche Information, die zur Ablehnung der ersten führte, von einem Agenten stammen, dessen Vertrauenwürdigkeit über einem global festgelegten Schwellenwert, dem *Liar-Threshold*, liegt. Diese zweite Bedingung haben wir hauptsächlich deswegen aufgenommen, um dem oben bereits angesprochenen Problem beim Unterscheiden von beabsichtigten oder unbeabsichtigten Lügen Rechnung zu tragen. Es ist so zu verstehen, dass wir eine Falschinformation nur dann als Lüge bezeichenen, wenn sie einer Information eines als sehr glaubwürdig[2] geltenden Agenten widerspricht. Wäre also z. B. der Liar−Threshold = 75, so würde *Mary* auch noch zusätzlich als Lügner entlarvt. Diese neu erworbene Information kann nun, z. B. in Form eines *Liar(X,Y)* Prädikates, in

[2] je nach Wert des Liar-Threshold

das Wissen integriert werden. Die indizierten Prädikate und die Reject Prädikate werden nach der Auswertung entfernt und die resultierende Antwortmenge formt das aktuelle Wissen.

Nach Projektion auf das Ursprüngliche Alphabet bleibt

```
{Clothes(mary,green)}
```

Allerdings können wir die Informationen aus der Update Antwortmenge nutzen um anhand von `Rej(r1)` zu inferieren, dass Mary in Bezug auf das Prädikat *Clothes* gelogen hat und der Antwortmenge ein entsprechendes Prädikat hinzufügen:

```
{Clothes(mary,green), Liar(mary,p_clothes)}
```

Eine „Bestrafung“ der Lüge wäre insofern möglich, dass jedes mal, wenn ein Agent einer neuen Lüge überführt wird, seine Vertrauenswürdigkeit prozentual oder um einen geringen festen Betrag abgesenkt wird. Dies führt dazu, dass einem Agenten immer weniger geglaubt wird, je häufiger er lügt, was wohl ein gutes Abbild eines realen sozialen Systems wäre.

2.6 Vertrauenswürdigkeit von Agenten

Jens Eckstein

2.6.1 Überblick

Da es in dem, dieser Projektarbeit zugrundeliegenden Agentensystem den Agenten gestattet bzw. möglich ist zu lügen, bedarf es ebenfalls geeigneter Möglichkeiten für die Agenten, den Wahrheitsgehalt von Informationen anderer Agenten bewerten zu können. Ein Bestandteil dabei ist die allgemeine Vertrauenswürdigkeit der Agenten. Da sich die einzelnen Agenten gegenüber den jeweils anderen Agenten auch unterschiedlich verhalten können, müssen alle Agenten eine eigene Liste der Werte über die Vertrauenswürdigkeit besitzen. Zusätzlich ist es in unserem Agentensystem auch möglich, dass sich die Werte im Laufe der Zeit verändern können bzw. verändert werden müssen, da vielleicht falsche Informationen diesem Wert zu Grunde lagen (siehe Kapitel 2.5.3).

2.6.2 Reliabilities

Die *Reliability* bezeichnet einen ganzzahligen Wert auf einer Skala von 0 bis 100, der die Vertrauenswürdigkeit, die ein spezifischer Agent für einen anderen Agenten empfindet, bezeichnet. Dabei ist 0 die geringste und 100 die höchste Vertrauenswürdigkeit.

Da die Liste der Vertrauenswürdigkeiten für jeden Agenten einzeln vorgehalten wird und diese auch unterschiedliche Anfangswerte haben können, werden die Vertrauenswürdigkeiten anfänglich unter `<reliabilities>` im `CDF` spezifiziert. Hier ein Beispiel:

```
<reliabilities>
  <reliability value="100" agent="Environment"/>
```

```
  <reliability value="80" agent="hastings"/>
  <reliability value="60" agent="alfred"/>
  ...
</reliabilities>
```

Für jeden im Szenario vorkommenden Agenten muss dort eine anfängliche Vertrauenswürdigkeit[3] festgelegt werden. Die Reliabilities sind *Meta-Informtionen* und somit nicht Bestandteil des *logischen Faktenwissens* der Agenten. Das zeigt auch schon eine Sache auf, die Reliabilities nicht zu leisten im Stande sind, nämlich in irgendeiner Weise in den *logischen Regeln* der Agenten verwendet zu werden und somit die Schlussfolgerungen des Agenten zu beeinflussen.

Sie werden stattdessen gebraucht, um bei widersprüchlichen Informationen diejenige auszuwählen, die der Agent glaubt (und die andere entsprechend verwirft bzw. zur Lügenerkennung verwendet). Neues Wissen wird mit dem *Wissensoperator* verarbeitet. Informationen können dabei um die Vertrauenswürdigkeit ihres Senders ergänzt werden und somit kann dann die glaubwürdigere Information bestimmt werden. Wie das im Einzelnen funktioniert hängt natürlich maßgeblich vom verwendeten Wissensoperator (vgl. Kaptiel 3.1.2 und 3.1.3) ab.

Wie bereits oben erwähnt sind diese Reliabilities aber nicht statisch, sondern können im Verlauf geändert werden. Eine von uns vorgesehene Funktion ist das Absenken der Vertrauenswürdigkeit, wenn ein Agent nachweislich gelogen hat (Kapitel 2.5.3). Ebenso wäre es möglich, eine geringe Vertrauenswürdigkeit aufzuwerten, wenn ein Agent häufig die Wahrheit sagt. Dies ist in unserem System aber nicht berücksichtigt.

Bei den von uns entwickelten Wissensoperatoren werden die Vertrauenswürdigkeiten wie gesagt nur zu Rate gezogen, wenn der Agent eine widersprüchliche Information erhält. Es wird also alles (auch Informationen von unglaubwürdigen Agenten) geglaubt, so lange es keinen Grund gibt, daran zu zweifeln. Man könnte aber genauso einen Operator konstruieren, der nur Informationen akzeptiert, die von glaubwürdigen[4] Agenten stammen.

2.7 Agentenmodell

Mohamed Dalil

2.7.1 Externe Sicht

Ein KIMAS-Agent ist zur Aussenwelt eine Blackbox, die mit anderen Agenten Nachrichten austauscht *(Interaktion)*. Er kann seine Umgebung wahrnehmen und diese wieder durch seine Aktionen beeinflussen, indem er Nachrichten empfängt und Nachrichten sendet. Bei den Nachrichten handelt es sich um Fragen, Antworten oder Mitteilungen.

[3] Der Umweltagent bekommt immer die Vertrauenswürdigkeit 100. Angaben von der Umwelt entsprechen den Informationen, die ein Agent direkt durch Interaktion oder Beobachtung erhält. Diese sollten gegenüber den Informationen durch andere Agenten immer bevorzugt werden.

[4] Agenten, deren Reiliability über einem Schwellenwert liegt

Von externer Sicht besitzt der Kimas-Agent die folgenden Fähigkeiten:

- Fragen an andere Agenten stellen
- Antworten auf Fragen von anderen Agenten generieren
- Mitteilungen empfangen und senden

2.7.2 Interne Sicht

Intern besteht ein KIMAS-Agent aus Komponenten. Diese Komponenten sind entweder Zustandskomponenten oder Kontrollkomponenten[Woo99]. Die Zustandskomponenten, wie *beliefs*, *desire*, etc., sind Datenmodelle. In diesen Modellen stehen die Daten, die der Agent benötigt, damit er seinen aktuellen Zustand bestimmen kann. Die Kontrollkomponenten (Funktionen) sind Interpreter zur Verwaltung der Daten. Die Funktionen operieren (verändern, löschen, oder neue Daten hinzufügen) auf den Daten der Zustandskomponenten. Eine Operation führt einen Zustand des Agenten in den nächsten Zustand über. Das Verhalten eines Agenten wird durch die beiden Komponententypen festgelegt, das heißt durch den aktuellen Zustand (aktuellen Wissensstand, aktuell verfolgtes Ziel, etc.) und die aktuell möglichen Operationen (Wissen aktualisieren, Ziel entfernen, Ziel generieren, etc.) darauf.

Die Zerlegung der Komponenten innerhalb des KIMAS-Agentenmodells basiert auf dem BDI Modell. Die grundlegende Struktur eines BDI-Agenten sind Weltwissen *(beliefs)*, Wünsche bzw. Ziele *(desires)* und aktuelle Absichten bzw. Verpflichtungen *(intentions)*. Ein KIMAS-Agent erweitert diese Struktur um *Know-How*, *skills* und *attitudes*, siehe Abbildung 2.1.

Die interne Sicht des Agentenmodells wird formal wie folgt dargestellt.

2.7.3 Formale Darstellung

Zunächst definieren wir die folgenden Abkürzungen:

- Bel: die Menge aller logischen Beliefbase,
- Des: die Menge aller Ziele,
- Int: die Menge aller Intentionen,
- KH: die Menge aller Know-How,
- SK: die Menge aller Skills und
- ATT: die Menge aller Attitudes

Ein KIMAS-Agent KA wird durch ein Tupel $KA = (K, P, F)$ repräsentieret. In K sind die $(B, D, Int, K_H, Ski, Att)$ gekapselt, wobei

- $B \in Bel$ das logische Wissen des Agenten,

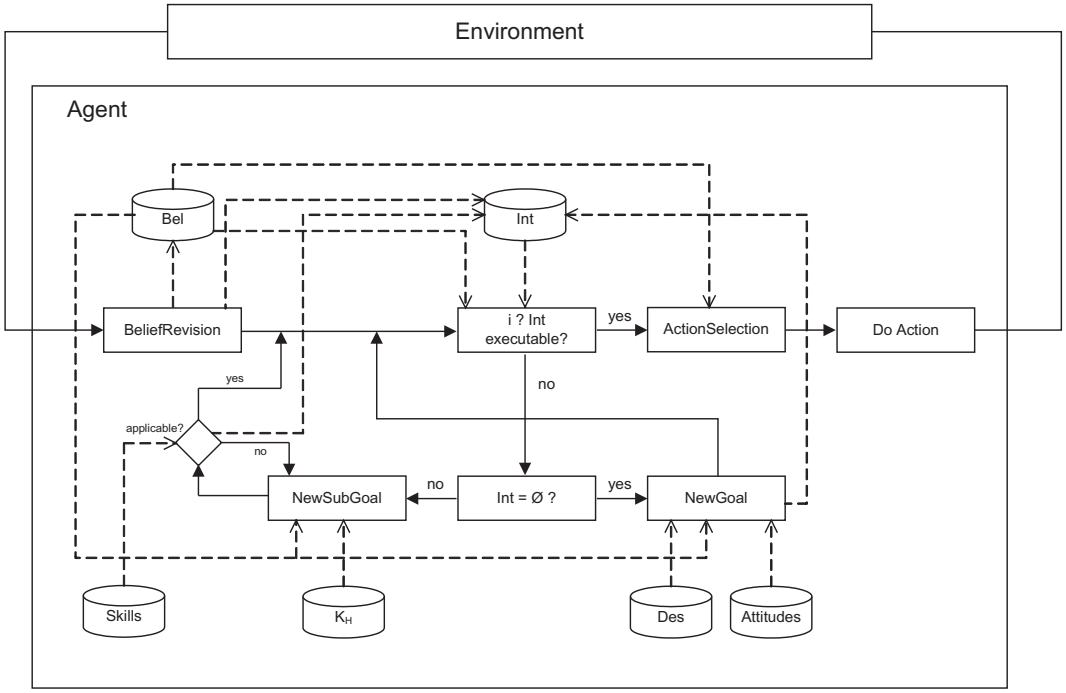

Abbildung 2.1: Interne Sicht

- $D \in Des$ die Ziele des Agenten,
- $I \in Int$ die gerade verfolgten Absichten des Agenten,
- $K_H \in KH$ das extra–logische Wissen zur Aufgabenbewältigung des Agenten,
- $Ski \in SKI$ die physischen Einschränkungen bzw. die Fähigkeiten, die der Agent hat und
- $Att \in ATT$ die Motivationen, mit denen der Agent ausgestattet ist.

Die einzelnen Komponenten werden unter 2.7.3.1 dargestellt. P enthält eine Menge von atomaren Aktionen, die der Agent zur Verfügung hat. F wird in 2.7.3.2 vorgestellt. Der Zustand eines Agenten zu einem Zeitpunkt wird durch ein Tupel (B, D, I) repräsentiert, wobei

- $B \in Bel$ das logische Wissen des Agenten,
- $D \in Des$ die Ziele des Agenten und
- $I \in Int$ die gerade verfolgten Absichten des Agenten

beschreiben.

2.7.3.1 Zustandskomponenten

Beliefs stellen das logische Wissen des Agenten über seine Umwelt und sich selbst dar. Aufgrund dieses Weltwissens werden seine Entscheidungen gefällt. Die Wissensbasis wird kontinuierlich aktualisiert, beeinflusst durch externe Änderungen, die vom Agenten wahrgenommen werden, und interne Schlussfolgerungen.

Desires sind übergeordnete Wünsche oder Ziele des Agenten, welche er realisieren will und anhand derer er sein Verhalten ausrichtet. Die Ziele werden ausführlich unter 2.8 erlütert.

Intentions sind Absichten des Agenten, es sind die *desires* zu deren Verfolgung er sich entschlossen hat. Meist hat ein Agent zuviele *desires*, als dass es sinnvoll oder auch möglich wäre alle davon zu verfolgen oder gar zu verwirklichen, daher ist eine solche Untermenge von tatsächlich verfolgten Zielen erforderlich. Die *intentions* dürfen sich nicht gegenseitig behindern oder ausschliessen.

Attitudes Da die Motivation im ursprünglichen BDI-Modell nicht vorhanden ist, wurde ein Konzept zur Integration der Motivation im KIMAS-Agentenmodell entwickelt. Das BDI-Modell wurde um eine neue Zustandskomponente *attitudes* erweitert. Mit Hilfe von *attitudes* wird die Motivation in dem Kimas-Agentenmodell integriert. Die *attitudes* besteht aus einer Menge von *attitude*. Aus *Attitude* werden Ziele generiert. Das heißt, diese Ziele können nur aus Motivation erwachsen, und anhand von Motivation bekommen die Ziele unterschiedliche Wertigkeiten. Das Ziel mit der höchsten Wertigkeit wird ausgeführt. Die Auswahl der Ziele, die der Agent demnächst verfolgen soll, wird durch die Motivation beeinflusst. Auf diese Weise wird das Verhalten des Agenten durch die Motivtion kontrolliert.

Beispiel: Ein Agent, der mit der Motivation *Ehrlichkeit* ausgestattet ist, wird ein Ziel generieren (Fall lösen), wenn er erfährt, dass jemand ermordet wurde. Ferner wird dieser Agent eine Sache, die er auf die Strasse gefunden hat, zum Fundbüro bringen. Dieses Beispiel wird in Tabelle 2.1 illustriert.

attitude	*triggers*	*condition*
respectability	*solve_case*	*someone_was_murdered*
	get_wallet_to_lost-property-office	*wallet_on_street*

Tabelle 2.1: Attitudes

Know-How ist das extra-logische Wissen zur Aufgabenbewältigung des Agenten und wird in 2.8 ausführlich erläutert.

Skills Darin wird die physichen Fähigkeiten eines Agenten charakterisiert, das heißt, welche Ziele und Aktionen kann ein Agent aufgrund physischer Einschränkung nicht ausführen.

Beispiel:

Angenommen hat ein Agent das folgende Ziel generiert (Transportiere eines Autos von Punkt A nach Punkt B). Aber in sein Skills steht, dass er ein Auto nicht tragen kann, deswegen ist dieser Agent nicht in der Lage, dieses Ziel zu verfolgen.

2.7.3.2 Funktionen

In F sind die Funktionen ($BeliefRevision, NewGoal, NewSubGoal, ActionSelection$) gekapselt. Durch diese Funktionen wird der Entscheidungsprozess eines Agenten repräsentiert. Der Entscheidungsprozess eines KIMAS-Agenten wird aus zwei Teilen zusammengesetzt:

1. deliberation: welche Ziele sollen erreicht werden?
2. means-end reasoning: wie werden die Ziele erreicht?

Zunächst werden die einzelnen Funktionen erläutert. Anschließend wird auf die Aufgabe jeder Funktion im Entscheidungsprozess eingegangen.

- BeliefRevision: Durch diese Funktion wird der Wissensstand eines Agenten aktualisiert. Die Aktualisierung hängt vom Wissensstand sowie von der aktuellen Wahrnehmung (empfangene Nachricht) des Agenten ab. Formal wird diese Funktion durch die Abbildung $Bel \times Inf \rightarrow Bel$ repräsentiert. Inf ist die Menge aller möglischen neuen Informtionen. Auf die BeliefRevision wird in anderen Abschnitten eingegangen.
- NewGoal: Mit Hilfe dieser Funktion werden neue Ziele generiert. Die Generierung eines neuen Ziels hängt von dem aktuellen Wissensstand und den Wünschen des Agenten ab. Die Abbildung $Bel \times Des \times ATT \rightarrow Int$ stellt die *NewGoal* Funktion formal dar. Die durch diese Funktion generierten Ziele werden in kleinere Unterziele zerlegt. Ein Ziel wird erst erreicht, wenn alle seine Unterziele erreicht worden sind.
- NewSubGoal: Damit der Agent seinen Intentionn nachgehen kann, erzeugt er intern Zwischenziele, mit deren Hilfe er Teilaufgaben überwinden kann. Die Generierung der Zwischenziele wird von dem aktuellen Wissensstand und dem Know-How des Agenten festgelegt. $Bel \times KH \rightarrow Int$
- ActionSelection: Aus mehreren möglichen Aktionen wählt diese Funktion eine Aktion zur Ausführung aus, um ein Ziel näher zu bestimmen. Die Auswahl wird durch den aktuellen Wissensstand und der Menge der im Moment zur Verfügung stehenden Aktionen bestimmt[5]. Die ActionSelection Funktion wird durch die Abbildung $Bel \times \wp(P) \rightarrow P$ dargestellt.

[5] Die Auswahl einer Aktion aus mehreren möglischen Aktionen kann durch verschiedene Ansätze realisiert werden, wie zum Beispiel zufällige Auswahl oder die Auswahl bevorzugter Aktionen

Während die Festlegung eines neuen Ziels (*Deliberation*) in unserem Agentenmodell durch die beiden Funktionen *NewGoal* und *NewSubGoal* charakterisiert wird, wird die Auswahl von Plänen zur Erreichung des Ziels (Means-ends-reasoning) durch die ActionSelection Funktion charakterisiert.

2.8 Know How

Matthias Thimm

2.8.1 Theoretische Grundlagen

Die Ziele eines Agenten haben eine unmittelbare Auswirkung auf die auszuführenden Aktionen. Schließlich agieren die Agenten, um ihre Ziele zu erfüllen. Jedoch genügt es für einen Agenten nicht alleine, dass er weiß, was er erreichen will. Es ist weiterhin wichtig, dass er auch weiß, *wie* er es erreichen kann. Zu diesem Zweck wird nun Know-How vorgestellt und informal definiert [Sin99].

Wir sagen, dass ein Agent x weiß, wie er ein Ziel p erreichen kann, wenn er in der Lage ist, p durch eine Folge von Aktionen wahr werden zu lassen. Wir definieren nun dieses Wissen induktiv über die Anzahl der Aktionen und die Anzahl ihrer Anwendungsreihenfolgen. Sei dazu Υ die Menge der Aktionsbäume über eine Menge $\mathcal{B}$ von Aktionen definiert durch:

- $\emptyset \in \Upsilon$ ($\emptyset$ sei der leere Baum).
- Für jede Aktion $a \in \mathcal{B}$ ist $a \in \Upsilon$.
- Seien $\tau_1, ..., \tau_m \in \Upsilon$ mit paarweise verschiedenen Wurzeln und $a \in \mathcal{B}$. Dann ist $\langle a; \tau_1, ..., \tau_m \rangle \in \Upsilon$.

Ein Aktionsbaum besteht also aus einer Aktion als Wurzel und eine Menge von nachfolgenden Teilbäumen. Ein Agent, der einen Baum $\tau \in \Upsilon$ abarbeitet, führt zunächst die Aktion der Wurzel aus, um dann je nach entstehendem Weltzustand einen geeigneten Teilbaum für die weitere Handlungsweise auszuwählen.

Know-How wird folgendermaßen rekursiv definiert

- Ein Agent weiß, dass er ein Ziel p erreichen kann, wenn er keine Aktion ausführt, g.d.w. er p bereits weiß. In diesem Fall ist somit keine Aktion nötig und das verlangte Ziel erfüllt.
- Ein Agent weiß, dass er ein Ziel p durch Ausführen einer einzigen Aktion a erreichen kann, g.d.w. wenn er a ausführen kann und er weiß, dass unabhängig von Aktionen anderer Agenten im System, p danach erfüllt ist.
- Ein Agent weiß, dass er ein Ziel p durch Ausführung eines Aktionsbaums τ erreichen kann, g.d.w. wenn er zunächst die Aktion an der Wurzel ausführt und für alle Teilbäume von τ weiß, dass durch sie p erreicht werden kann.

Für eine formale Definition von Know-How nach Singh siehe [Sin99], [SRG99].

2.8.2 Das erweiterte Modell

In einer sehr dynamischen Umgebung ist es nicht immer effizient, für ein komplexes Ziel alle möglichen Wege, dieses Ziel zu erreichen, im Know-How eines Agenten zu integrieren. Vielmehr ist es sinnvoll zu einem Ziel eine Menge von Teilzielen anzugegeben, nach deren Erfüllung auch automatisch das Elternziel erfüllt sein wird. Da jedoch in einem Multiagentensystem äußere Umstände nicht unberücksichtigt werden dürfen, erhält man einen gewissen Grad von Dynamik durch Angabe mehrerer alternativer Mengen von Teilzielen. Ein Ziel p kann somit einerseits durch Abarbeitung einer Teilzielmenge $\{q_1, \ldots, q_n\}$ oder einer Teilzielmenge $\{r_1, \ldots, r_m\}$ erreicht werden. Hier ist eine Parallele zum Ansatz von Singh erkennbar, wobei $q_1, \ldots, q_n, r_1, \ldots, r_m$ nicht nur Aktionen sind (das dürfen sie jedoch auch), sondern auch wieder weitere Ziele, mit weiteren alternativen Teilzielmengen. Will ein Agent ein Ziel p erreichen, so sucht er sich zunächst eine der Teilzielmengen von p aus, um sie zu erreichen. Jedes Teilziel kann weiter aufgespalten werden und neue Alternativen erzeugen. Mißlingt dem Agenten die Erfüllung eines Teilziels, so wird die Abarbeitung der aktuellen Teilzielmenge abgebrochen und versucht, das Elternziel mit der Abarbeitung der nächsten Teilzielmenge zu erfüllen. Ist das Know-How eines Agenten ausreichend genau spezifiziert, so gibt es auf jeden Fall eine Teilzielmenge, die in der aktuellen Situation vollständig ausgeführt werden kann und das Ziel erfüllt.

2.8.3 Ziele und Aktionen

Das Know-How eines Agenten ist ein Paar (G, A) mit einer Menge von Zielen G und einer Menge von Aktionen A. Ziele und Aktionen haben zunächst ein grundsätzlich ähnliches strukturelles Aussehen und werden deswegen hier als Spezialisierungen eines Objekts *Know-How-Element* definiert.

Definition 1 *Ein* Know-How-Element *ist ein Tupel* (n, T, C)*, wobei*

- n *der Name des Know-How-Elements ist,*
- T *eine Menge von Triggern dieses Elements ist und*
- C *eine Menge von conditions dieses Elements ist.*

Jedes Know-How-Element in $G \cup A$ sollte einen eindeutigen Namen n haben, da es in den Teilzielmengen anderer Goals eindeutig referenziert werden muss. *Trigger* und *conditions* sind logische Formeln der unterlegten Logik des Multiagentensystems und beziehen sich stets auf die aktuelle Wissensmenge des Agenten. Sobald ein $t \in T$ in der aktuellen Wissensmenge des Agenten liegt, soll dieses Know-How-Element direkt ausgeführt (im Falle von Aktionen) oder als Ziel gesetzt werden (im Falle von Zielen). Die logischen Formeln in C geben Bedingungen an, die in der aktuellen Wissensmenge erfüllt sein müssen, damit dieses Element ausgeführt bzw. als Ziel gesetzt werden darf.

Definition 2 *Ein* Ziel *ist ein Tupel* (n, T, C, S)*, wobei*

- (n, T, C) *ein Know-How-Element und*

- *S eine Menge von Teilzielmengen mit $S \subseteq \mathfrak{P}(G \cup A)$ ist.*

Jede Teilzielmenge eines Ziels kann also sowohl weitere Ziele, als auch atomare Aktionen enthalten.

Definition 3 *Eine* Aktion *ist ein Tupel (n, T, C, A, D), wobei*

- *(n, T, C) ein Know-How-Element und*
- *A bzw. D eine Menge von logischen Formeln ist.*

Dabei ist A eine Menge von Formeln, die nach Ausführung der Aktion in die Wissensbasis integriert werden sollen und D eine Menge von Formeln, die nach Ausführung der Aktion aus der Wissensbasis entfernt werden sollen, ähnlich der A- und D-Listen in STRIPS *[FN71].*

Die Repräsentationen von Zielen und Aktionen in der Know-How Struktur sind nur Projektionen der eigentlichen Strukturen für Ziele und Aktionen des Agenten. Ziele und Aktionen sind zunächst unabhängig von ihrer Darstellung im Know-How. Zur Realisierung von Know-How müssen sie allerdings geeignet repräsentiert und ihre wechselseitigen Beziehungen untereinander definiert werden, wie dies oben geschehen ist.

Beispiel 1 *Seien im Folgenden $\alpha, \beta, \gamma, \delta$ Formeln in der der Spezifikation des Systems unterlegten Sprache und (G, A) das Know-How eines Agenten mit*

$$G = \{X, Y\}$$
$$A = \{a_1, a_2, a_3\}$$

Dabei sei

$$X = (zielX, \{\alpha, \beta\}, \{\gamma\}, \{(zielY, aktionA1), (aktionA2)\})$$
$$Y = (zielY, \emptyset, \emptyset, \{(aktionA2)\})$$
$$a_1 = (aktionA1, \emptyset, \{\delta\}, \{\iota\}, \{\delta\})$$
$$a_2 = (aktionA2, \{\beta\}, \emptyset, \{\delta\}, \emptyset)$$
$$a_3 = (aktionA3, \{\iota, \beta\}, \{\gamma\}, \{\kappa\}, \{\gamma\})$$

Angenommen es ist $\Delta = \{\gamma\}$ die Wissensmenge eines Agenten und zielA sein initiales Ziel. Die condition für die Verfolgung des Ziels zielA ist $\{\gamma\}$ und wegen $\{\gamma\} \subseteq \Delta$ darf der Agent mit der Verfolgung dieses Ziels fortfahren. zielA enthält zwei alternative Vorgehensweisen für seine Erfüllung: Zum einen kann der Agent nacheinander zunächst das Ziel zielB erfüllen und dann die Aktion aktionA1 ausführen, um zielA zu erreichen. Zum anderen könnte er zielA aber auch durch die Aktion aktionA2 erreichen. Der Agent wähle nun die erste Teilzielmenge $(zielB, aktionA1)$ für die nächste Verfolgung und somit zunächst das Ziel zielB. zielB hat keine conditions und darf somit verfolgt werden. zielB hat als einzige Teilzielmenge $(aktionA2)$, also das Ausführen der Aktion

aktion$A2$. aktion$A2$ hat keine conditions und kann somit direkt ausgeführt werden, d.h. es werden die Formeln aus $\{\delta\}$ der Wissensmenge Δ hinzugefügt und die Formeln aus $\emptyset$ der Wissensmenge Δ entnommen, soweit dies möglich ist. Die neue Wissensmenge des Agenten ist nun also $\Delta' = \{\gamma, \delta\}$ und das Ziel zielB wurde erfüllt. Als nächstes Teilziel von zielA soll die Aktion aktion$A1$ ausgeführt werden. Aktion aktion$A1$ hat als conditions die Menge $\{\delta\}$ und wegen $\{\delta\} \subseteq \Delta'$ darf aktion$A1$ ausgeführt werden, d.h. es werden die Formeln aus $\{\iota\}$ der Wissensmenge Δ' hinzugefügt und die Formeln aus $\{\delta\}$ der Wissensmenge Δ' entnommen, soweit dies möglich ist. Die neue Wissensmenge des Agenten ist nun also $\Delta'' = \{\iota, \gamma\}$ und das Ziel zielA wurde erfüllt. Da nun $\iota \in \Delta''$ wird die Aktion aktion$A3$ getriggert und da die conditions wegen $\{\gamma\} \subseteq \Delta''$ erfüllt sind direkt ausgeführt, d.h. es werden die Formeln aus $\{\kappa\}$ der Wissensmenge Δ'' hinzugefügt und die Formeln aus $\{\gamma\}$ der Wissensmenge Δ'' entnommen, soweit dies möglich ist. Die neue Wissensmenge des Agenten ist nun also $\Delta''' = \{\iota, \kappa\}$ und es sind keine weiteren Ziele zu verfolgen.

2.9 Ablauf von Szenen im Multiagentensystem

Max Vorderstemann

Das zugrundeliegende Szenario *„The Mysterious Affair at Styles“* wird bei KIMAS in einzelne Szenen unterteilt. Eine Szene ist hierbei ein Handlungsabschnitt des Gesamtszenarios, der an einem festen Ort und mit einer festen Anzahl von teilnehmenden Agenten stattfindet. Innerhalb jeder Szene ist für jeden der teilnehmenden Agenten genau definiert, welche Ziele er zum Ende dieser Szene erreicht haben soll, um die Szene erfolgreich zu beenden und bevor er mit der nächsten Szene fortfahren kann. Auch wenn aus konzeptioneller Sicht, durch die Einteilung des Szenarios in einzelne Szenen und der Vorgabe von Zielen, den Agenten in unserem Multiagentensystem ein Teil ihrer Autonomie genommen wird, so wird auf der anderen Seite die Modellierung und die Spezifikation des Szenarios erheblich erleichtert. Da die Modellierung eines größeren Szenarios, welches durch unsere Romanvorlage ohne Zweifel gegeben ist, eine immense Komplexität der logischen Modellierung sowie der Agenten- und Umweltspezifikationen mit sich bringt, wäre es bei so umfangreichen Informationen, wie z.B. die *belief states*, schier unmöglich die gesammte Szenariospezifikation sinnvoll kontrollieren und überprüfen zu können. Somit machen wir durch unsere Szeneneinteilung das Szenario unseres Multiagentensystems anschaulicher und vor dem Hintergrund der verwendeten Konzepte auch einfacher nachvollziehbar.

Für die genaue Beschreibung und Einteilung der Szenen des in KIMAS verwendeten Szenarios siehe Kapitel 4.3. Dort werden die vorgenommene Einteilung der Szenen und die dafür zugrundeliegenden Handlungsstränge im Detail beschrieben, sowie jede Szene samt Ort, teilnehmender Agenten und ihre entsprechenden Ziele spezifiziert. Desweiteren wird die Szeneneinteilung und der allgemeine Ablauf einer Szene im Abschnitt 5.6.2 thematisiert.

2.9.1 Überprüfung der Szenenziele

Durch die Vorgabe von Teilzielen in einer Szene soll der Agent keine expliziten Informationen über seine zu erledigenden Aufgaben erhalten. Das bedeutet, dass wir dem Agent nicht explizit seine Ziele vorgeben, die er in der entsprechenden Szene verfolgen soll, sondern lediglich überprüfen wollen, ob er die zur Beendigung der Szene nötigigen Ziele auch erreicht hat. Der Agent erreicht seine *goals* nur mit Hilfe sein eigenes Fakten- und Regelwissens und durch sein *Know How*. Zu Begin einer Szene werden die Agenten und die Umwelt entsprechend initialisiert und mit den nötigen Informationen versorgt[6]. Neben allen regulären *goals*, die entweder direkt aus der Logik oder mittels des Know-Hows des Agenten erzeugt werden, besitzt jeder Agent für die Szenen an denen er beteiligt ist ein *goal* mit dem Namen *SceneXSatisfaction*. Dabei ist X in diesem Fall ein Platzhalter für die entsprechende Nummer der jeweiligen Szene. Damit dieses Ziel auch erfüllt wird, müssen bestimmte Bedingungen erfüllt sein, damit der Agent die Szene beenden kann. Ist dies der Fall wird durch das Ziel die Aktion *FinishScene* ausgelöst, welche den Agenten dazu veranlasst der Umwelt durch eine entsprechende Steuernachricht mitzuteilen, dass er die Szeneziele erreicht hat. Dann nimmt einen reaktiven Zustand an in dem er nur noch auf Anfragen anderer Agenten antwortet, aber keine eigenen Anfragen mehr stellt. Hat der Umweltagent von allen an der Szene teilnehmenden Agenten eine solche Nachricht erhalten, wird die nächste Szene gestartet. Dabei kann das Szenenziel eines Agenten auch trivial erfüllbar sein, wenn z.B. der Agent in einer Szene keine eigenen Informationen erlangt oder Aktionen ausführt, sondern lediglich auf Anfragen anderer Agenten reagiert. Als Beispiel hier der entsprechende Auszug aus Johns CDF:

```
<goal name="Scene3Satisfaction" satisfaction="3">
 <triggers>
  <trigger>
   <condition>CurrentScene(3)</condition>
   <condition>SceneRunning(3)</condition>
  </trigger>
 </triggers>
 <subelementsets>
  <subelementset>
   <doaction name="FinishScene"/>
  </subelementset>
 </subelementsets>
</goal>
```

In diesem Fall erreicht John sein Ziel in Szene drei damit, dass die Bedingungen *CurruntScene(3)* und *SceneRunning(3)* trivialerweise in Szene drei immer erfüllt sind. Der Agent John würde, wie bereits erwähnt, lediglich auf Anfragen anderer Agenten reagieren, da er sein Ziel bereits erreicht hat.

Der wirkliche Nutzen der Szeneneinteilung offenbart sich allerdings erst bei der Betrachtung eines aktiven Agenten. Wenn man nun einmal den Agenten Poirot betrachtet,

[6] siehe auch 5.3 und 5.6.2

dessen Aufgabe es ist den Mörder zu finden, dann wird die immense Menge an Informationen die er durch Befragungen und Schlussfolgerungen gewinnt, wie bereits angesprochen, sehr unübersichtlich. Am Ende des gesamten Szenarios wäre die Schlussfolgerungskette nur sehr aufwändig zu rekonstruieren, um im Falle eines falschen Ermittlungsergebnisses die genaue Ursache für den Fehlschluss zu finden. Damit Poirot nicht schon in der ersten Szene eine entscheidene Information verpasst, hier sein entsprechendes *SceneXSatisfaction*-Ziel mit den dazugehörigen Bedingungen für die erste Szene:

```
<goal name="Scene1Satisfaction" satisfaction="1">
 <triggers>
  <trigger>
   <condition>Ordered(vases,1)</condition>
   <condition>LocationTimeless(testament,emilysRoom)</condition>
   <condition>EnvThing(emptyPharmacyBox,1)</condition>
   <condition>SceneRunning(1)</condition>
   <condition>CurrentScene(1)</condition>
  </trigger>
 </triggers>
 <subelementsets>
  <subelementset>
   <doaction name="FinishScene"/>
  </subelementset>
 </subelementsets>
</goal>
```

Somit wird dafür gesorgt, dass Poirot die Szene erst beenden kann, wenn er die Vasen auf dem Kamin zurechtgerückt hat (was für die Aufklärung des Mordfalls von essentieller Bedeutung ist), sowie das verbrannte Testament und die leere Medizinschachtel endeckt. Durch die Bedingungen wird sichergestellt, dass dem jeweilige Agent keine bedeutsamen Informationen, sei es durch Befragung anderer Agenten, Untersuchung des Orts oder wichtige Inferenzen, entgehen. Der Agent kann aber nicht aufgrund der Bedingungen, welche er natürlich implizit weiß, versuchen diese zu erfüllen, vielmehr wäre dann die Aufgabe die Spezifikation und Modellierung des Szenarios so anzupassen, dass kein Agent in einer Szene steckenbleibt.

3 Wissensdarstellung

3.1 Wissensoperationen

Patrick Krümpelmann

3.1.1 Einführung

Wie bereits in Kapitel 2.4 beschrieben, werden Wissensoperatoren genutzt, um die Informations-Historie eines Agenten zu verarbeiten und einen konsistenten Wissenszustand zu erreichen. Wir haben die im folgenden beschriebenen Operatoren entwickelt.

3.1.2 Der SimpleOperator

Der *SimpleOperator* berechnet auf einfache Weise anhand der initialen Wissensbasis und der Informations-Historie einen aktuellen Wissenszustand.

Modellberechnung Sei also ein Agent mit einer initialen Wissensbasis und einer Menge von protokollierten Informationen (Fragen, Antworten und allgemeine Mitteilungen) gegeben. Jede dieser protokollierten Informationen enthält ein instanziiertes Prädikat, welches allerdings nicht direkt in den Wissenszustand übernommen werden soll, sondern zunächst daraufhin überprüft wird, ob der jeweilige Agent vertrauenswürdig genug ist. Dies wird beim *SimpleOperator* durch Angabe von Fakten Reliable/1 erreicht, die bestimmte Agenten als vertrauenswürdig einstufen. Ist der Absender einer bestimmten Information vertrauenswürdig, so wird das enthaltene Faktum zunächst in ein Hold-Faktum umgewandelt, das in einem weiteren Inferenzschritt dann in ein Faktum des Orginalprädikats überführt wird. Für jede Arität von Prädikaten ist somit ein bestimmtes HoldX-Prädikat erforderlich, wobei X die Arität des enthaltenen Prädikats angibt.

> **Beispiel.** Betrachte nocheinmal die Frage aus Beispiel 1 und die zugehörige Antwort:

```
Query(q1).
Query_Type(q1, q_instantiate).
Query_Sender(q1, tom).
Query_Arity(q1, 1).
Query_Content(q1, 1, p_mouse).

Answer(a1, q1, 43).
Answer_Sender(a1, jerry).
```

```
Answer_Arity(a1, 1).
Answer_Content(a1, 1, speedy_gonzalez).
```

Das in dieser Konversation enthaltene Faktum ist Mouse(speedy_gonzalez) und der für diese Information verantwortliche Agent ist jerry. Angenommen Agent tom möchte diese Konversation nun analysieren. Dazu enthaltene seine Wissensbasis zunächst das Faktum Reliable(jerry) – tom hält jerry also für einen vetrauenswürdigen Agenten. Um aus einer q_instantiate Frage die nötigen Informationen zu extrahieren, benötigt tom noch weitere Regeln in seiner Wissensbasis (Für jede Arität von bekannten Prädikaten ist eine solche Regel nötig – hier wird nur die Regel für einstellige Prädikate aufgeführt):

```
Hold1(P, X1) :- Query(Q),
                Query_Type(Q, q_instantiate),
                Query_Content(Q, 1, P),
                Answer(A, Q, T),
                Answer_Content(A, 1, X1),
                Answer_Sender(A, Y), Reliable(Y).
```

Mit anderen Worten: Gibt es eine Frage Q, die vom Typ q_instantiate ist und nach dem Prädikat P fragt und weiterhin eine dazu passende Antwort A, die P mit X1 instanziiert und vom vertrauenswürdigen Agenten Y stammt, so glaube, dass P mit X1 instanziiert gültig ist.
Um aus dem Hold-Faktum noch das Faktum des Originalprädikats zu erhalten, ist für jedes Prädikat folgende Regel nötig (hier für das Prädikat Mouse):

```
Mouse(X1) :- Hold1(p_mouse, X1).
```

Die im Beispiel genannten Regeln müssen für jeden Typ von Konversation (Fragen vom Typ q_instantiate, q_element_info und q_yes_no, sowie Inform-Mitteilungen) und für jede Arität von Prädikaten vorhanden sein. Wir verzichten zunächst darauf, hier alle erforderlichen Regeln aufzulisten, da sie sehr der Regel aus dem Beispiel ähneln.

3.1.3 Der Update-Program-Operator

Der *Update-Program-Operator* berechnet anhand der initialen Wissensbasis und der Informations-Historie einen aktuellen Wissenszustand. Dabei wird ein Ansatz zum Wissensupdate mit erweiterten logischen Programmen nach Eiter, Fink, Sabbatini, Tompits [EFST02] verwendet. In diesem Ansatz wird eine Sequenz von erweiterten logischen Programmen zu einem einzigen Update-Programm kompiliert. Dies entspricht unserem Ansatz der Informations-Historie.

Update-Programme basieren auf dem Prinzip der begründeten Ablehnung. Eine Regel r wird nur abgelehnt, wenn dafür ein Grund besteht. Ein Grund zur Ablehnung ist eine aktuellere, nicht abgelehnte Regel r' welche im Konflikt zu r steht. Hierbei wird neuerem Wissen also eine höhere Relevanz zugeteilt. Im Folgenden wird dieser Ansatz detailiert beschreiben.

3.1.3.1 Syntax und Konstruktion

Gegeben sei eine Sequenz von erweiterten logischen Programmen $\mathbf{P} = (P_1, ..., P_n)$ über dem Alphabet $\mathcal{A}$. Diese stellen sukzessive Update-Informationen der vorherigen Programme dar. Nun ist das Ziel ein einziges erweitertes logisches Programm $P_\lhd$ aus der Sequenz zu erzeugen.

Die Sequenz der Programme wird also zu einem einzigen Programm zusammengesetzt. Dazu ist die Erweiterung des Alphabets notwendig. Das Alphabet $\mathcal{A}$ wird erweitert zu $\mathcal{A}^*$. Zum einen durch die Prädikate $rej(r)$, welche benutzt werden, um die Anwendung der Regel r zu kontrollieren und somit Konflikte zu beseitigen, also die Regel abzulehnen (reject). Des weiteren wird jedes Vorkommen eines Atoms A entsprechend der Programmnummer seines Vorkommens indiziert und zu A_i umbenannt. Eine injektive Namensfunktion $N(,)$ wird eingeführt, um Regeln unterschiedlicher Programme unterscheidbar zu machen. Schließlich werden die Literale L entsprechend der Nummerierung der Atome zu L_i umbenannt. Aus den Programmen P_i der Sequenz wird in vier Schritten das resultierende Update-Programm konstruiert. Diese sollen nun beschrieben werden.

Definiere ein Update-Programm $P_\lhd = P_1 \lhd ... \lhd P_n$ über $\mathcal{A}^*$ bestehend aus:

1. für jede Regel $r \in P_i, 1 \leq i \leq n$:

$$L_i \leftarrow B(r), not\ rej(r) \quad H(r) = L$$

2. für jede Regel $r \in P_i, 1 \leq i < n$:

$$rej(r) \leftarrow B(r), \neg L_{i+1} \quad H(r) = L$$

3. für jedes Literal L in $\mathbf{P}$ $(1 \leq i < n)$:

$$L_i \leftarrow L_{i+1}, \quad L \leftarrow L_1$$

4. allen Constraints aus $P_i, 1 \leq i \leq n$

Im ersten Schritt werden die Regeln aus allen Programmen übernommen wobei sie um den Zusatz $not\ rej(r)$ erweitert werden. Dieser sorgt dafür, dass sie nicht zur Anwendung kommen, falls sie abgelehnt wurden. Wann eine Regel abgelehnt wird, ist durch den zweiten Schritt definiert. Sie wird abgelehnt, wenn der Rumpf der Regel erfüllt ist, das Kopfliteral jedoch mit dem einer aktuelleren Regel im Konflikt steht. Im dritten Schritt werden die indizierten Literale verkettet, wodurch eine geschichtete Erfüllung der Literale realisiert wird. Die Literale des neuesten Programmes P_n stehen dabei an oberster Stelle, die des ältesten P_1 an Unterster. Ein erfülltes Literal mit dem Index i impliziert sukzessive alle Literale mit Index $j < i$. Auf unterster Ebene wird das Literal global erfüllt. Im vierten Schritt werden alle *constraints* aus allen Programmen gesammelt und in das neue Programm übernommen.

3.1.3.2 Semantik

Im folgenden beschreiben wir die Antwortmengensemantik von Update-Programmen. Eine Antwortmenge S' des kompilierten Programms $P_{\lhd}$ ist über dem erweiteren Alphabet $\mathcal{A}^*$ definiert und muss auf das ursprüngliche Alphabet $\mathcal{A}$ projiziert werden.

Theorem 1 *S ist eine Update-Antwortmenge der Update-Sequenz $\boldsymbol{P} = (P_1, ...P_n)$ **gdw.** $S = S' \cap \mathcal{A}$ für eine Antwortmenge $\mathcal{S}'$ von $\mathcal{P}_{\lhd}$*

$\mathcal{U}(P)$ bezeichne die Menge aller Update-Antwortmengen und $Bel(P) = Bel(\mathcal{U}(P))$ die Wissensmenge zu P.

3.1.3.3 Beispiel

Im folgenden wird das vorgestellte Prinzip anhand eines vollständigen Beispieles verdeutlicht. Gegeben seien die beiden erweiterten Logischen Programme P_1 und P_2.

$$
\begin{aligned}
P_1 &= \{r_1 : sleep \leftarrow not\ tv\_on; r_2 : night \leftarrow; r_3 : tv\_on \leftarrow; r_4 : watch\_tv \leftarrow tv\_on\} \\
P_2 &= \{r_5 : \neg tv\_on \leftarrow power\_failure; r_6 : power\_failure \leftarrow\}
\end{aligned}
$$

Nun konstruieren wir das dazu gehörige Update-Programm $P_{\lhd}$ gemäß den oben genannten Konstruktionsregeln.

$$
\begin{aligned}
P_{\lhd} = \{ & \\
& r_1 : sleep_1 \leftarrow not\ tv\_on, not\ rej(r_1) \\
& r_2 : night_1 \leftarrow not\ rej(r_2) \\
& r_3 : tv\_on_1 \leftarrow not\ rej(r_3) \\
& r_4 : watch\_tv_1 \leftarrow tv\_on, not\ rej(r_4) \\
& r_5 : \neg tv\_on_2 \leftarrow power\_failure, not\ rej(r_5) \\
& r_6 : power\_failure_1 \leftarrow not\ rej(r_6) \\
& r_7 : rej(r_1) \leftarrow not\ tv\_on, \neg sleep_2 \\
& r_8 : rej(r_2) \leftarrow \neg night_2 \\
& r_9 : rej(r_3) \leftarrow \neg tv\_on_2 \\
& r_{10} : rej(r_4) \leftarrow tv\_on, \neg watch_t v_2 \\
& r_{11} : sleep_2 \leftarrow sleep_1 \\
& r_{12} : sleep \leftarrow sleep_1 \\
& r_{13} : night_2 \leftarrow night_1 \\
& r_{14} : night \leftarrow night_1 \\
& r_{15} : tv\_on_2 \leftarrow tv\_on_1 \\
& r_{16} : tv\_on \leftarrow tv\_on_1 \\
& r_{17} : power\_failure \leftarrow power\_failure_1 \\
& \}
\end{aligned}
$$

Wie klar sichtbar ist wurden die Regeln r_1 bis r_6 durch den ersten Konstruktionsschritt erzeugt, die Regeln r_7 bis r_{10} durch den Zweiten und die restlichen durch den Dritten. *Constraints* entfallen in diesem Beispiel.

Dieses Update-Programm besitzt eine einzige Update-Antwortmenge.

$$\begin{aligned} S' = \{&power\_failure_2, power\_failure_1, power\_failure, \neg tv\_on_2, \\ &\neg tv\_on_1, \neg tv\_on, rej(r_3), sleep_1, sleep, night_1, night\} \end{aligned}$$

Die Antwortmenge zu $\mathbf{P}$ ergibt sich folglich durch Projektion auf das ursprüngliche Alphabet zu $(S' \cap \mathcal{A})$:

$$S = \{power\_failure, \neg tv\_on, sleep, night\}$$

3.1.3.4 Minimalität

Update-Programme sind in der bisherigen Definition nicht minimal in der Änderung. Wenn ein neues Programm P_2, also eine neue Menge von Regeln, in ein bestehendes Programm P_1 eingearbeitet werden soll, ist es wünschenswert bei der Einarbeitung so wenig wie möglich zu ändern. Um eine Minimierung der Änderung zu erreichen müssen wir zunächst bestimmen wie Änderung und Ähnlichkeit definiert ist. Ähnlichkeit kann auf der Modell-Ebene, als auch auf einer syntaktischen Ebene definiert werden. Dem syntaktischem Ansatz der Update-Programme gerecht werdend, definieren wir die Änderung eines Programmes P_1 durch ein Update P_2, anhand der abgelehnten Regeln.

Dazu führen wir Rejection-Sets $Rej(S, \mathbf{P})$ ein welche die Menge der Regeln enthält die abgelehnt wurden. Ein Rejection-Set ist definiert durch $Rej(S, \mathbf{P}) = \bigcup_{i=1}^{n} Rej_i(S, \mathbf{P})$ wobei $Rej_n(S, \mathbf{P}) = \emptyset$ und für $n > i \geq 1$,

$$\begin{aligned} Rej_i(S, \mathbf{P}) = \{&r \in P_i | \exists r' \in P_j \setminus Rej_k(S, \mathbf{P}) \text{ für ein } k \in \{i+1, \ldots, n\}, \\ &\text{so dass } r, r' \text{ im Konflikt stehen und } S \models B(r) \cup B(r')\}. \end{aligned}$$

Wir ziehen eine Antwortmenge S_1 von $\mathbf{P} = (P_1, P_2)$ einer Antwortmenge S_2 vor, wenn S_1 eine größere Menge Regeln von P_1 erfüllt als S_2. Mit den Rejection-Sets führt das zu folgender Definition:

Definition 4 *Eine Antwortmenge $S \in \mathcal{U}(\boldsymbol{P})$ ist minimal **gdw.** kein $S' \in \mathcal{U}(\boldsymbol{P}$ existiert, so dass $Rej(S', \boldsymbol{P}) \subset Rej(S, \boldsymbol{P})$.*

Somit ist eine Antwortmenge minimal, wenn sie die geringste Anzahl zurückgewiesener Regeln enthält und somit am meisten gültige Regeln besitzt. Hierbei kann es vorkommen, dass eine Antwortmenge vorgezogen wird welche Regeln aktuellerer Programme ablehnt, um Regeln in älteren Programmen intakt zu lassen. Dies verletzt jedoch die temporale Ordnung auf den Updates welche gerade durch das Verfahren angestrebt wird.

3.1.3.5 Strikte Minimalität

Um den gerade beschriebenen Nachteil der Definition Minimaler Antwortmengen zu beseitigen, werden strikt minimale Antwortmengen definiert.

Definition 5 *Eine Antwortmenge S ist strikt minimal wenn keine andere Antwortmenge $S' \in \mathcal{U}(\boldsymbol{P})$ bevorzugt wird.*

Hierbei ist die Bevorzugung definiert durch:

Definition 6 *S wird S' vorgezogen **gdw.** es ein i gibt, so dass:*

$$i) Rej_i(S,P) \subset Rej_i(S',P)$$
$$ii) Rej_k(S,P) = Rej_k(S',P) \text{ für alle } k \in \{i+1,...,n\}$$

Durch die so definierte strikte Minimalität wird die temporale Ordnung aufrecht erhalten.

3.1.3.6 Antwortmengenfilter

Minimalität und strikte Minimalität können durch im folgenden vorgestellte Filterprogramme erreicht werden.

Um minimale beziehungsweise strikt minimale Antwortmengen zu erhalten, gehen wir in zwei Schritten vor. Zunächsten werden alle Update-Antwortmengen des Update-Programmes $P_\lhd$ erzeugt. Anschliessend werden die so erhaltenen Kandidaten auf Minimalität bzw. strikte Minimalität getestet. Zum Testen eines Kandidaten S auf Minimalität bzw. strikte Minimalität wird ein Testprogramm P_S^{min} bzw. P_S^{strict} aus den Regeln des Update-Programmes $P_\lhd$ und zusätzlichen Regeln erzeugt. Durch die zusätzlichen Regeln gegenüber $P_\lhd$ werden die möglichen Antwortmengen der Testprogramme P_S^{min} bzw. P_S^{strict} so beschränkt, dass sie weniger Regeln ablehnen als die Antwortmenge S ablehnt bzw. so, dass die Antwortmengen der Testprogramme bevorzugt werden gegenüber der Antwortmenge S. Also ist eine Kandidaten-Antwortmenge minimal bzw. strikt minimal, wenn das ensprechende Testrogramm P_S^{min} bzw. P_S^{strict} keine Antwortmenge besitzt.

3.1.4 Minimale Antwortmengen

Im folgenden wird dieser Prozess etwas formaler dargestellt. Sei $P_\lhd = P_1 \lhd ... \lhd P_n$ ein Update-Programm und S eine Antwortmenge von $P_\lhd$. Wir führen nun neue Prädikate ein. Zum einen das nullstellige Prädikat oder Atom ok, dann Prädikate s_r für jedes Prädikat rej_r, das in $P_\lhd$ vorkommt. Das Prädikat s_r habe wie rej_r Stelligkeit 1. Das Testprogramm für Minimalität P_S^{min} wird gebildet aus allen Regeln und *constraints* aus $P_\lhd$ und folgenden Erweiterungen:

1. für jedes Prädikaten Symbol rej_r in $P_\lhd$:

$$\leftarrow rej_r(X), not\ s_r(X).$$

2. für jede instanzierte Formel $rej_r(t) \in S$:

$$ok \leftarrow not\ rej_r(t).$$

$$s_r(t) \leftarrow .$$

3. und ein Constraint:

$$\leftarrow not\ ok.$$

Hier wird klar, dass lediglich der zweite Schritt von der Kandidaten-Antwortmenge S abhängig ist. Die *constraints* im ersten Schritt sorgen dafür, dass alle Antwortmengen ausgeschlossen werden, deren *Rejection Sets* keine Teilmenge von $Rej(S, \mathbf{P})$ sein können. Das heisst, dass die Kandidaten-Antwortmenge mindestens eine Regel ablehnt, welche in S nicht abgelehnt wird. Falls Antwortmengen übrig sind, sind diese entweder *ok*, d.h. dass *ok* in ihnen wahr ist, oder *ok* ist falsch und damit werden sie über die *constraints* in Schritt 3. eliminiert.

Folgendes Theorem untermauert dieses Verfahren:

Theorem 2 *Gegeben sei ein Update-Programm $P_\triangleleft$. Sei S eine Antwortmenge von $P_\triangleleft$. S ist eine minimale Antwortmenge von $P_\triangleleft$* ***gdw.*** *P_S^{min} keine Antwortmenge besitzt.*

Der Beweis zu diesem Theorem ist zu finden in [EFST02].

Dieses Ergebnis ermöglicht es uns, alle minimalen Antwortmengen eines Update-Programms $P_\triangleleft$ mit einem einfachen Algorithmus bestimmen zu können. Dieser Algorithmus sieht im Pseudocode wie folgt aus:

```
Algorithm Compute_Minimal_Models(P)
Input: A sequence of ELPs P = (P_1, ..., P_n).
Output: All minimal answer sets of P.

var Cands: SetOfAnserSets;
var MinModels: SetOfAnserSets;
Cands := Compute_Answer_Sets(P_◁)
for all S ∈ Cands do
        var Counter: SetOfAnserSets;
        Counter . = Compute_Answer_Sets(P_S^min);
        if (Counter = ∅ then
                MinModels := MinModels ∪{S};
        fi
rof
return MinModels;
```

Dieser Algorithmus berechnet zunächst alle Antwortmengen von $P_\triangleleft$ und überprüft anschließend für jede Antwortmenge S, ob das dazugehörige Testprogramm P_S^{min} mindestens eine Antwortmenge besitzt. Falls nicht wird S zu den minimalen Antwortmengen von $P_\triangleleft$ hinzugefügt.

3.1.5 Strikt Minimale Antwortmengen

Zur Bestimmung strikt minimaler Antwortmengen verwenden wir folgendes Verfahren. Sei $P_\lhd = P_1 \lhd ... \lhd P_n$ ein update Programm und S eine Antwortmenge von $P_\lhd$. Wir führen folgende neue Prädikate ein: ok, ok_i $(1 \leq i \leq n)$ und eq_i $(1 \leq i \leq n+1)$ seien nullstellige Prädikate. Desweiteren Prädikate s_r für jedes Prädikat rej_r das in $P_\lhd$ vorkommt. s_r habe die selbe Stelligkeit wie rej_r. Das Testprogramm für strikte Minimalität P_S^{strict} wird nun gebildet aus allen Regeln und Constraints aus $P_\lhd$ und folgenden Erweiterungen:

1. für jedes Prädikaten Symbol rej_r in $P_\lhd$, welches zu $r \in P_i$ gehört:

$$\leftarrow rej_r(X), not\ s_r(X), eq_{i+1}.$$

2. für jeden instanzierten Term $rej_r(t) \in S$, welcher zu $r \in P_i$ gehört:

$$ok_i \leftarrow not\ rej_r(t), eq_{i+1}.$$

$$s_r(t) \leftarrow .$$

3. für $1 \leq i \leq n$:

$$eq_i \leftarrow eq_{i+1}, not\ ok_i.$$

$$ok \leftarrow ok_i.$$

4. ein Constraint:

$$\leftarrow not\ ok.$$

5. und ein Faktum:

$$eq_{n+1} \leftarrow .$$

Auch hier hängt P_S^{strict} lediglich durch den zweiten Schritt von der Kandidaten Antwortmenge S ab. Mittels der *constraints* im ersten Schritt werden alle Antwortmengen S' aussortiert, welche nicht S vorgezogen werden können, weil sie für ein i eine Regel ablehnen, welche nicht durch S abgelehnt wird und $Rej(S, \mathbf{P}) = Rej(S', \mathbf{P})$ gilt für alle $j = i+1, ..., n$. Letzteres wird durch die Atome eq_{i+1} ausgedrückt. Für die eventuell verbleibenden Antwortmengen gilt wiederum, dass ok entweder wahr oder falsch ist. Ist ok wahr in S', dann gilt auch ok_i in S' für ein i. Das heißt S' lehnt eine Regel aus P_i nicht ab, welche in S abgelehnt wird und es gilt $Rej(S, \mathbf{P}) = Rej(S', \mathbf{P})$ für alle $j = i+1, ..., n$. Also wird S' bevorzugt über S. Ist ok falsch in S', dann gilt $Rej(S, \mathbf{P}) = Rej(S', \mathbf{P})$ für alle $i = 1, ..., n$ und S' wird durch den vierten Schritt aussortiert.

Analog zu dem Theorem für die Minimalitäts-Testprogramme lässt sich nun auch folgendes Theorem für die Testprogramme für strikte Minimalität P_S^{strict} aufstellen:

Theorem 3 *Gegeben sei eine Update-Sequenz* $\mathbf{P}$. *Sei* S *eine Antwortmenge von* $P_\lhd$. S *ist eine stikt minimale Antwortmenge von* $P_\lhd$ ***gdw.*** P_S^{strict} *keine Antwortmenge besitzt.*

Auch der Beweis zu diesem Theorem ist zu finden in [EFST02]. Mit diesem Theorem lässt sich der oben angegebene Algorithmus für minimale Antwortmengen nun leicht für strikt minimale Antwortmengen benutzen.

3.1.5.1 Komplexität

Die Komplexität von Update-Programmen besitzt als natürliche untere Schranke die Komplexität von normalen logischen Programmen welche *NP-schwer* sind. Da die Konstruktion des Update-Programmes aus einer Sequenz von Updates klar in polynomieller Zeit möglich ist ergibt sich folgendes: Gegeben sei eine Update-Sequenz $\mathbf{P} = (P_1, ..., P_n)$ über dem Alphabet $\mathcal{A}$:

- das Entscheidungsproblem ob $\mathbf{P}$ eine Antwortmenge besitzt, ist NP-vollständig,
- das Entscheidungsproblem ob ein Literal $L \in Bel(\mathbf{P})$ ist co-NP-vollständig.

Für die Eigenschaft der Minimalität und der strikten Minimalität muss ein noch höherer Preis gezahlt werden. Für minimale und strikt minimale Updates gilt:

- das Entscheidungsproblem ob $r \in Bel_{min}(\mathbf{P})$ und
- das Entscheidungsproblem ob $r \in Bel_{str}(\mathbf{P})$

sind $\prod_2^P$ −vollständig.

3.1.5.2 Anpassungen an KiMAS

In KiMAS ist eine streng temporale Ordnung der Update Sequenz nicht sinnvoll, da wir Informationen von verschiedenen Quellen mit unterschiederlichen Glaubwürdigkeiten erhalten. Um diesem gerecht zu werden, haben wir den Ansatz der Update-Programme dahin gehend modifiziert. Hierzu haben wir die Ordnung der Update-Sequenz geändert, an dem eigentlichen Verfahren allerdings nichts. Als Ordnung benutzen wir hierbei eine lexikographische Ordnung in den Glaubwürdigkeiten der Quelle und den Zeitpunkten des Erhaltes der Informationen.

3.1.6 Der UpdateOperator

Der *UpdateOperator* benutz die Informations-Historie eines Agenten, welche Metainformationen über die Quelle und Zeit der Informationen enthält, um einen konsistenten Wissenszustand zu erreichen. Dabei wird eine Menge erweiterter logischer Programme zu einem einzigem, konsistenten erweitertem logischen Programm kompiliert. Zusätzlich zu der Informations-Historie besitzt der Agent Werte über die Glaubwürdigkeit ihm bekannter Agenten. Mit Hilfe der Metainformationen über die Quelle jedes erhaltenen Programmes lässt sich jedem Programm eine Priorität zuordnen. Anhand dieser Prioritäten und der Zeitinformation lässt sich nun eine lexikographische Ordnung auf den Programmen definieren.

Es stellte sich als ungünstig heraus Regeln eines Programmes im Konfliktfall pauschal durch Regeln aus höher geordneten Programmen zu überschreiben. Wie folgendes Beispiel aufzeigt kann dies zu unintuitiven Ergebnissen führen da die Rumpfliterale und deren Glaubwürdigkeiten in keiner Weise Einfluss auf die Priorität der Regel hat.

Index	Glaubwürdigkeit	Programm
P_1	3	$B.$
P_2	8	$\neg A.$
P_3	10	$A \longleftarrow B.$

Hier würde die Regel in P_2 abgelehnt durch die in P_3 obwohl letztere nur auf der weniger Glaubwürdigen Information aus P_1 beruht.

Daher erachten wir es als notwendig die jeweilige Instanziierung einer Regel zu betrachten und diese mit in die Priorisierung einzubeziehen. Die eingangs erwähnte Priorisierung der Programme ist damit lediglich als Initialisierung zu verstehen. Des weiteren betrachten wir nicht Prioritäten für ganze Regeln, sondern Priorisieren einzelne Literale. Die tatsächlichen Prioritäten von Literalen ergeben sich dynamisch aus dem gesamten Programm.

In unserem Ansatz werden Prioritäten durch Regeln an ihre Kopfliterale vererbt. Die Priorität eines Kopfliterals einer Regel wird durch die Priorität der Regel selbst, als auch den Prioritäten der instanziierten Rumpfliterale bestimmt. Die Priorität einer Regel ergibt sich direkt aus der Priorität des zugehörigen Programms. Fakten sind somit ein Spezialfall dieser Vererbung und erhalten die Priorität des Programmes. Durch diesen Prozess wird eine Schicht von priorisierten Literalen gebildet, welche über der Schicht der unpriorisierten Literalen liegt und mit dieser gekoppelt ist. Auf dieser priorisierten Schicht arbeitend findet unsere Konfliktbehandlung statt. Es werden Konflikte detektiert und entsprechend der geerbten Prioritäten aufgelöst, indem der niedriger priorisierte Konfliktteilnehmer blockiert wird.

3.1.6.1 Konstruktion

Gegeben sei eine Sequenz von erweiterten logischen Programmen mit Metainformationen Quelle q und Zeit t $\mathbf{P} = (\langle P_1, q_1, t_1\rangle, ..., \langle P_n, q_n, t_n\rangle)$ über dem Alphabet $\mathcal{A}$. Diese stellen eine Menge von Updates dar. Nun ist das Ziel ein einziges erweitertes logisches Programm $P_\diamond$ aus diese Menge zu erzeugen.

Dazu ist die Erweiterung des Alphabets notwendig. Das Alphabet $\mathcal{A}$ wird erweitert zu $\mathcal{A}^*$ durch eine Reihe neuer Prädikate und Atome, welche im folgenden eingeführt werden. Die Menge der Attribute eines Literals L sei mit $\mathcal{A}_L$ bezeichnet. Zu jedem Literal $L(\mathcal{A}_L)$, das in $\mathbf{P}$ vorkommt wird eine priorisierte Version $L(\mathcal{A}_L, R)$ erstellt. Hierbei repräsentiert die Variable R die Priorität der betrachteten Ausprägung des Literals L. $\mathcal{H}(\mathbf{R})$ bezeichne im folgenden die Menge der in Regelköpfen auftretenden Literalen.

Im ersten Konstruktionsschritt werden wird die Menge $\mathbf{P}$ lexikografisch in q und t geordnet und entsprechend dieser Ordnung linear mit Prioritäten versehen. Nun können wir die Konstruktion des Programms $P_\diamond = P_1 \diamond ... \diamond P_n$ beschreiben durch die Erzeugung folgender Regeln:

1. für jede Regel $r \in P_i, 1 \leq i \leq n$ mit Kopf $H(\mathcal{A}_H, R)$:

$$H(\mathcal{A}_H, R) \leftarrow \mathcal{B}(r), min\{\{max_{R'}\{B(\mathcal{A}_B, R')\} | B(\mathcal{A}_B, R') \in \mathcal{B}^+(r)\}, i\} = R.$$

2. für jedes Literal $L \in \mathcal{H}(\mathbf{P})$:

$$L(\mathcal{A}_L) \leftarrow L(\mathcal{A}_L, R), not\ rej(L, R, R').$$

3. für jedes Literal $L \in \mathcal{H}(\mathbf{P})$:

$$rej(L, R, R') \leftarrow L(\mathcal{A}_L, R), \neg L(\mathcal{A}_L, R'), R < R'.$$

4. allen Constraints aus $P_i, 1 \leq i \leq n$

Im ersten Konstruktionsschritt werden die original Regeln aus $\mathbf{P}$ so erweitert, dass sie die Vererbung von Prioritäten unterstützen. Zur Bestimmung der Priorität wird das Minimum über die Maxima der Prioritäten der Rumpfliterale und der Priorität der Regel gebildet. Im zweiten Schritt findet die Koppelung der Prioritätsschicht an die unpriorisierte Schicht statt. Ein Literal gilt in der unpriorisierten Schicht, wenn es in der priorisierten Schicht gilt und in der priorisierten nicht abgelehnt wurde. Der dritte Schritt erzeugt die *rej* Prädikate, welche für die Ablehnung priorisierter Literale sorgen und somit eine Weiterreichung an die unpriorisierte Schicht verhindern. Ein Literal wird abgelehnt, wenn es im Konflikt mit einem entsprechenden negierten Literal höherer Priorität im Konflikt steht.

3.1.6.2 Semantik

Im folgenden wird die die Antwortmengensemantik der so erstellten Programmen beschrieben. Eine Antwortmenge S' des kompilierten Programms $P_\diamond$ ist über dem erweiteren Alphabet $\mathcal{A}^*$ definiert und muss auf das ursprüngliche Alphabet $\mathcal{A}$ projiziert werden.

Theorem 4 *S ist eine update Antwortmenge der update Sequenz $\boldsymbol{P} = (P_1, ...P_n)$ **gdw.** $S = S' \cap \mathcal{A}$ für eine Antwortmenge $\mathcal{S}'$ von $\mathcal{P}_\diamond$*

3.2 Vertrauenswürdigkeiten

Ulrich Engler

Im Gegensatz zur Umsetzung, die dem Zwischenbericht als Grundlage diente, werden nunmehr *Vertrauenswürdigkeiten* nicht mehr über `DLV` inferiert, sondern anfangs in den `CDFs` abgelegt. Die ursprüngliche Idee dabei war eigentlich, dass man bei der Verwendung von Logik zur Gewinnung der Werte eben auch die Modellierung des Szenarios mit hätte einbeziehen können, um sie so durch Regeln zu verändern oder sie selbst als Basis von Regeln, also im Rumpf zu verwenden. Da sich der Umfang und die Komplexität der *Vertrauenswürdigkeiten* jedoch soweit reduziert hat, auch hinsichtlich der Bedeutung der *Vertrauenswürdigkeiten*, wäre eine Umsetzung innerhalb der Logik unnötig umständlich gewesen. Die Werte dienen jetzt lediglich noch als Vergleichswert für den *Update*- bzw. *Revisionsprozess*, um so zu entscheiden, welche Informationen geglaubt und welche

Agenten gegebenenfalls als Lügner bezeichnet werden sollen (Kapitel 3.1). Zudem sind sie relativ statisch geworden, da sie derzeit nur bei einer *Lüge* herabgesetzt werden können (Kapitel 2.5). Insofern sind sie damit kein direkter Teil der Wissensdarstellung mehr, sondern eher ein Bestandteil des Wissensoperators (Kapitel 2.4). Nähere Informationen zum Thema Vertrauenswürdigkeiten finden sie in Kapitel 2.6.

3.3 Lügen

Max Vorderstemann

Unser Szenario erfordert es, dass die Agenten in unserem Multiagentensystem die Fähigkeit besitzen zu lügen. Da in einem Kriminalfall der oder die Täter im Normalfall nicht des Verbrechens überführt werden wollen, das sie begangen haben, werden sie bestimmte Informationen nicht preisgeben oder auch falsche Informationen verbreiten. Aber auch andere Agenten können Gründe haben gewisse Informationen zurückzuhalten oder zu verfälschen. Um diesem Umstand Rechnung zu tragen gibt es in unserem Multiagentensystem zwei mögliche Ausnahmefälle in denen gelogen werden kann, welche im Folgenden beispielhaft erklärt werden.

Voraussetzung ist es, dass alle Agenten defaultmäßig immer die Wahrheit sagen. Damit soll sichergestellt werden, dass die Agenten nicht nach Lust und Laune auf Fragen antworten, sondern bis auf die folgenden Ausnahmefälle immer wahrheitsgetreu antworten. Außerdem müsste sonst zusätzlich noch zwischen absichtlichen und unabsichtlichen Lügen unterschieden werden. Hat der Agent ein Fakt in seinem aktuellen Wissen, über das er lügen möchte, kann er die entsprechende Information entweder geheimhalten oder stattdessen falsche Informationen preisgeben. Für uns stellt damit sowohl das Verschweigen von Wissen als auch die Ausgabe von falschen Wissen eine Lüge da. Der Fakt bzw. das Prädikat, über das der Agent lügen will, muss also in jedem Fall in seinem aktuellen Wissen enthalten sein, da der Agent über Sachverhalten die er nicht kennt keine falschen Auskünfte geben soll und kann (im Fall von Unwissenheit wird keine Antwort gegeben). Zusätzlich benötigt der Agent einen entsprechenden Hinweis in seinem aktuellen Wissen über welche Aussagen er lügen möchte. Liegt dieser Hinweis nicht vor, antwortet der Agent wahrheitsgemäß.

3.3.1 Verbergen von vorhandenem Wissen

Der Agent kann nun vorhandenes Wissen verbergen und keine Antwort auf eine entsprechende Anfrage geben, falls es in seinem aktuellen Wissen ein Prädikat der Form *KeepSecret2(p_ Pred,a,b)* gibt. Dabei bibt die Zahl hinter *KeepSecret* die Stelligkeit des Prädikates an, in diesem Fall also das zweistellige Prädikat *Pred(a,b)*.

Betrachten wir einmal folgendes Beispiel: Der Agent John hat sich am Tag vor dem Mord heftig mit seiner Stiefmutter, dem späteren Mordopfer, gestritten. Bei der Zeugenbefragung durch den Agenten Poirot will John diesen Streit nun lieber verschweigen, da er befürchtet das dieser Streit in den Augen des Detektivs ein mögliches Motiv für den Mord an seiner Stiefmutter darstellt. Wenn in seinem Wissen die Fakten

```
Disputing(emily,john,1039).
KeepSecret3(p_Disputing,emily,john,1039).
```

sind verbirgt der Agent das Prädikat *Disputing*, falls eine entsprechende Anfrage an ihn gestellt wird. Das entsprechende *KeepSecretX*-Faktum kann auch regelbasiert erzeugt werden. So kann Agent John die Regel

```
KeepSecret3(p_Disputing,X,Y,T) :-
        Victim(X),
        MyName(Y),
        TimeOfDisputeWithEmily(T).
```

haben, die besagt, dass wenn er sich zu einem bestimmten Zeitpunkt mit dem Mordpfer gestritten hat, dieses Prädikat geheimgehalten werden soll.

3.3.2 Lügen und Generierung alternativer Antworten

Falls der Agent in der Art über vorhandes Wissen lügen will, dass er auf eine konkrete Anfrage falsches Wissen, d.h. Wissen das er nicht glaubt und somit auch nicht in seinem aktuellen Wissen enthalten ist, preisgibt, muss er zusätzlich auch die Fähigkeit besitzen alternative Antworten zu erzeugen. Wie auch schon beim Verschweigen von Wissen muss ein entsprechendes Prädikat der Form *LieAbout3(p_Pred,a,b,c)* im aktuellen Wissen des Agenten liegen, damit er wie in diesem Fall über das dreistellige Prädikat *Pred(a,b,c)* lügen kann. Die Zahl hinter *LieAbout* gibt nach wie vor die Stelligkeit des Prädikates an. Desweiteren benötigt der Agent auch noch die nötige Information, was er als Antwort auf eine Anfrage ausgeben will. Dies sollte aber nicht beliebig geschehen, um seltsame Antworten wie *Murderer(johnsRoom)* oder *HasProfession(lawrence,coffee)* zu vermeiden. Dies geschieht indem zusätzlich zu den *LieAboutX*-Fakt in seinem aktuellen Wissen Fakten der Form *ArgumentSemantic(p_Pred,X,p_Characteristic)* liegen. Der Fakt sagt aus, dass wenn ich über das Prädikat *Pred* lügen will, dann soll für das Argument an Stelle X dieses Prädikates eine zwingende Eigenschaft gelten. Die zwingende Eigenschaft wird durch das Prädikat *Characteristic* festgelegt, d.h. das jeweilige Argument an Stelle X muss also immer das entsprechende *Characteristic*-Prädikat erfüllen, wobei diese Eigenschafts-Prädikate einstellig sein müssen. Liegen im Wissen des Agenten alle erforderlichen Fakten, kann er so eine alternative Antwort bzw. eine Lüge erzeugen.

Als Beispiel betrachten wir folgende Situation: Der Agent Lawrence glaubt, dass Cynthia für den Tod von Emily verantwortlich ist. Da Lawrence Cynthia liebt, möchte er allerdings nicht, dass sie des Mordes überführt wird und will deswegen über Wissen, dass zu ihrer Ergreifung dienen könnte, lügen. Hier ein kleiner Auszug seines dafür relevanten Wissens.

```
Workplace(Cynthia,hospital)
Knowledge(Name1,Poison) :-
        Workplace(Name1,hospital),
        Poison(Toxin).
```

Durch Cynthias Arbeit in der Krankenhausapotheke weiß Lawrence, dass Cynthia sich dementsprechend gut mit Medizin und auch Gift auskennt.

```
WasOpened(doorEmilyCynthia).
WasInsideRoomOfCrime(Name1) :-
        WasOpened(Door),
        DoorFromTo(Door,Place1,Place2),
        RoomOfCrime(Place1),
        BelongsTo(Place2,Name1).
```

Lawrence hat in der Mordnacht beobachtet, dass die Tür zwischen Cynthias und Emilys Zimmer, dem Tatort, offen war. Aufgrund seiner obigen Regel glaubt er, dass Cynthia also am Tatort gewesen ist.

```
CauseOfDeath(strychnine).
```

Als studierter Mediziner diagnostiziert Lawrence anhand von Emilys Symptomen, dass es sich in diesem Fall bei der Todesursache wohl um eine Strychninvergiftung handelt. Zusammen mit seinem Grundwissen über Cynthias Arbeitstelle und ihrer damit verbundenen Kenntnisse über Gift kommt er zu dem Schluss, dass durch die Todesursache Gift Cynthia unter Tatverdacht stehen würde.

```
Suspect(Name1) :- Poison(X),
        Knowledge(Name1,X),
        WasInsideRoomOfCrime(Name1),
        CauseOfDeath(X).
```

Lawrence weiß durch sein Medizinstudium zwar, dass die Todesursache Strychnin ist, allerdings will er bedingt durch seine Liebe zu Cynthia diesen Fakt nicht nur verschweigen, sondern Poirot bewußt eine andere Todesursache nennen, um so Cynthia nicht zu belasten. Zu diesem Zweck hat er eine Regel, die ihm das passende *LieAboutX*-Fakt erzeugt.

```
LieAbout1(p_CauseOfDeath,X) :-
        Sympathy(Z,Name1,10),
        MyName(Z),
        Suspect(Name1).
```

Da Lawrence keine Todesursache nennen will, die Cynthia belasten kann, braucht er das Wissen über mögliche Todesursachen, die in keinem Fall mit einem Verbechen in Verbindung gebracht werden. Eine nicht durch eine Straftat bedingte Todesursache ist ein Unfall.

```
NonCriminalDeath(accident).
ArgumentSemantic(p_CauseOfDeath,1,p_NonCriminalDeath).
```

Zusammen mit dem Hinweis durch den *ArgumentSemantic*-Fakt gibt Lawrence auf Anfragen, welche die Todesursache betreffen, also die folgende Antwort bzw. die Lüge aus.

```
CauseOfDeath(accident).
```

Ein weiteres kurzes Beispiel soll zeigen, dass es auch eine Reihe von zwingenden Eigenschaften geben kann, die bei der Erzeugung einer alternativen Antwort bzw. Lüge berücksichtigt werden müssen. Betrachen wir noch einmal die Situation aus dem ersten Beispiel, in dem der Agent John seinen Streit mit dem späteren Mordopfer verschweigt. In diesem Fall weiß auch Johns Ehefrau Mary von dem Streit zwischen John und Emily.

```
Disputing(emily,john,1039).
```

Auch Mary glaubt, dass ein Streit mit dem Opfer in den Augen der Ermittler ein mögliches Motiv darstellen könnte und so ihr Mann John als Verdächtiger gelten würde.

```
Suspect(X) :-
        Disputing(Name1,X _),
        Person(X),
        Victim(Name1).
```

Aus diesem Grund will Mary den Streit nicht nur geheimhalten, sondern auf entsprechende Anfragen eine Lüge entgegnen. Die folgende Regel lässt sie über einen Streit genau dann lügen, wenn eine der am Streit beteiligten Personen ein Mordopfer ist und die andere Person mit ihr verheiratet ist und unter Verdacht steht. Dies trifft auf ihren Mann John zu.

```
LieAbout3(p_Disputing,Y,X,T) :-
        Victim(Y),
        Suspect(X),
        Married(X,Z),
        MyName(Z),
        TimeOfDisputeWithEmily(T).
```

Da Mary bezweckt den Verdacht von ihrem Mann zu nehmem, will sie den Streit einer anderen Person anhängen. Deshalb entscheidet sie sich, den Hauptverdächtigen Alfred als Beteiligten des Streits mit Emily zu nennen.

```
ArgumentSemantic(p_Disputing,1,p_Person).
ArgumentSemantic(p_Disputing,1,p_Victim).
ArgumentSemantic(p_Disputing,2,p_Person).
ArgumentSemantic(p_Disputing,2,p_MainSuspect).
ArgumentSemantic(p_Disputing,3,p_TimOfDisputeWithEmily).
```

Diese *ArgumentSemantic*-Fakten stellen sicher, dass Mary die Argumente des *Disputing*-Prädikats richtig wählt. Das erste Argument muss die Eigenschaften haben eine Person zu sein, die auch gleichzit das Opfer ist. Das zweite Argument muss ebenfalls eine Person, sowie der Hauptverdächtige, sein. Der Zeitpunkt des Streits wird durch das dritte Argument festgelegt. Zusammen mit Marys erforderlichem Wissen

```
Person(emily).
Victim(emily).
Person(alfred).
MainSuspect(alfred).
TimOfDisputeWithEmily(1039).
```

und dem entsprechenden *LieAboutX*-Fakt gibt Mary auf Anfragen, welche die Todesursache betreffen, also die folgende Antwort bzw. die Lüge aus.

```
Disputing(emily,alfred,1039).
```

4 Wissensmodellierung

4.1 Roman

Jens Eckstein

Als Grundlage für die Modellierung des Gesamtszenarios dient Agatha Christies Kriminalroman *„Das fehlende Glied in der Kette“* (OT: „The Mysterious Affair at Styles“, 1920). Für den Fall, dass dem Leser die Lektüre unbekannt ist oder es eventuell ein wenig Auffrischung bedarf, folgt eine kurze Inhaltsangabe, um die Rahmenhandlung des Romans darzustellen.

4.1.1 Das fehlende Glied in der Kette

Während des Fronturlaubs hält sich Hastings als Gast seines Jugendfreundes John Cavendish auf dem Anwesen Styles auf. Die Bewohner des Landsitzes sind John und seine Frau Mary, Johns Bruder Lawrence und ihre Stiefmutter Emily Inglethorp. Diese erbte damals anstatt ihrer Stiefsöhne das komplette Vermögen der Familie, welches sie zum Teil für karitative Zwecke aufwendet, wie zum Beispiel für die Unterbringung der Waisin Cynthia Murdoch auf Styles. Sowohl John als auch Lawrence sind bedingt durch einige eher mäßig erfolgreiche Ausflüge in die berufliche Selbstständigkeit wieder nach Hause zurückgekehrt. Um die Stimmung auf dem Herrensitz ist es seit der kürzlichen Heirat von Emily mit dem bedeutend jüngeren Alfred Inglethorp nicht besonders gut bestimmt. Alle Familienmitglieder hegen eine mehr oder weniger starke Antipathie gegenüber dem neuen Bewohner von Styles, wobei dies vor allem wohl durch den Verdacht der Erbschleicherei bedingt ist. Blind vor Liebe reagiert Emily natürlich wenig erfreut über solche Anschuldigungen seitens der Familie, was dazu führt, dass es zu einigen Auseinandersetzungen zwischen den Hausbewohnern kommt und das Dienstmädchen Evelyn Howard, welche sich mit ihren Verbalattacken gegen den neuen Ehemann nicht zurückhalten kann, sich mit ihr überwirft und abreist. Die angespannte Lage gipfelt schließlich darin, dass Emily eines Nachts an den Folgen einer Strychninvergiftung stirbt. Nun schaltet Hastings seinen alten Freund Hercule Poirot, einen pensionierten Polizisten und Detektiv, ein und er bekommt den Auftrag sich mit dem Fall zu befassen. Dem ersten Anschein nach ist die Beweis- und Faktenlage klar und deutet auf den allseits unbeliebten Alfred Inglethorp hin. Doch genauere Untersuchungen von Poirots Seite zeigen, dass es sich eben nicht ganz so verhält wie zuerst angenommen und es im Bezug auf den Tathergang viele offene Fragen gibt. So gehen Poirot und Hastings den zahlreichen Spuren und Indizien nach und versuchen so das „Puzzle“ zusammenzusetzen, um herauszufinden wie der Mord genau begangen wurde und wer der Täter ist. Es wird offenbar, dass auch andere Bewohner des Hauses ein Motiv für den Mord besitzen und neben ihrem teils merkwürdigen Verhalten

gibt es verschiedene Indizien, dass eben auch sie als Täter in Frage kommen könnten. Am meisten erhärtet sich der Verdacht gegen John und schon bald gerät Hastings Freund unter akuten Tatverdacht, während Alfred ein scheinbar lückenloses Alibi vorlegen kann. Doch Poirot erkennt mit seiner logischen Herangehensweise die falschen Fährten und hat bald ein genaues Bild des Mordhergangs im Kopf, doch leider kann er dem Täter aufgrund mangelnder Beweise noch nicht belangen. Erst eine zufällig gemachte Bemerkung von Hastings bringt ihn auf das fehlende Glied in der (Schlussfolgerungs-)Kette, den endgültigen und belastenden Beweis. Poirot kann so letztendlich Alfred Inglethorp und Evelyn Howard als die wahren Täter präsentieren. Die beiden haben gemeinsam den Plan geschmiedet Emily Inglethorp zu ermorden, um sich mit der Erbschaft, die deren Ehemann Alfred zugefallen wäre, aus dem Staub zu machen. Zu diesem Zweck haben sie Mrs. Inglethorp's strychninhaltiger Medizin eine Bromlösung zugesetzt, was zur Folge hatte, daß sich das Strychnin am Boden und damit in der letzten Dosis in tödlicher Konzentration gesammelt hat. Zusätzlich haben sie auch ein paar Beweise fingiert, um so die Spur zu jemand anderem führen zu lassen, während sie selber für die Mordnacht ein Alibi besaßen.

4.1.2 Änderungen gegenüber der Romanvorlage

Der Roman eignet sich aus mehreren Gründen ganz gut für die Modellierung von Agenten und ihrer Umgebung, da die Welt des Buches größtenteils auf einen Ausschnitt reduziert ist, der von der realen Welt isoliert ist – hier das Landhaus Styles. Aus diesem Grund gestaltet sich die zu modellierende Umwelt relativ überschaubar und kompakt. Zum anderen werden die Ermittlungen nur durch Befragung von Verdächtigen und Zeugen, sowie Untersuchungen des Tatorts und der Umgebung vorangetrieben.

Dennoch haben wir einige Sachverhalte ändern bzw. weglassen müssen, da diese den Rahmen und Umfang dieser Projetktgruppenarbeit überschritten hätten. So beginnt das von uns ausgewählte und im folgenden beschriebene Szenario erst an dem Zeitpunkt, als *Poirot* am Morgen nach dem Mord in Styles ankommt. Alle wesentlichen vorherigen Geschehnisse und Informationen, die neben den üblichen einführenden Charkterbeschreibungen natürlich hauptsächlich die Ereignisse der Tage vor und während des Mordes betreffen, sind den jeweiligen Agenten als *Startwissen* mitgegeben. Dieses beinhaltet auch schon umfassendes Wissen über den Tathergang, welches der Roman verständlicherweise erst zum Schluss offenbart. Ein weiterer entfallener Teil des Romans betrifft die Festnahme und Gerichtsverhandlung von *John Cavendish*. Zum einen ist diese für die Überführung des Mörders von zweitrangiger Bedeutung und dient – nach Poirots eigener Aussage – mehr dazu, die Ehe der Cavendishs zu retten. Zum anderen hätten solche Ausflüge jenseits der überschaubaren Welt von Styles, das Szenario unnötig aufgebläht und eine sehr viel umfangreichere Modellierung der Umgebung erforderlich gemacht.

Kommen wir im folgenden zu einigen Änderungen im Detail, die wir gegenüber der Romanvorlage vorgenommen haben.

4.1.2.1 Agenten

Obwohl in diesem Kriminalroman nur vergleichsweise wenige Personen mitspielen – es beschränkt sich im Wesentlichen auf die Angehörigen, Polizei, Detektive und einige Zeugen – ist die Anzahl zu hoch, um sie ordentlich handhaben zu können. Daher haben wir zunächst die kleinen *Nebenrollen*, die häufig nur an einer oder wenigen Stellen im Roman auftauchen, entfernt und ihre Funktion entweder durch vorhandene Charaktere ersetzt (z. B. erhält Poirot die Informationen, die laut Vorlage der Apotheker liefert, nun vom Hausarzt) oder, da sie überflüssig waren, ersatzlos gestrichen. An einigen Stellen erhält Poirot von anderen Personen Fachwissen (z. B. über Pharmazeutik), welches er für die weiteren Untersuchungen benötigt. Dieses haben wir ihm dann in den meisten Fällen bereits als Startwissen mitgegeben. Der gesamte Handlungsstrang um die geheime Identität von *Dr. Bauerstein* ist weggefallen. Darüber hinaus haben wir die beiden Doktoren zu einem einzigen mit dem Namen *Wilkinstein* zusammengefaßt. Desgleichen haben wir alle Bediensteten im Landhaus Styles zu einem *Personal*-Agenten vereinigt. Neben den Agenten, gibt es auch noch Personen in unserer Modellierung, über welche die Agenten zwar Informationen besitzen können (z. B. das Mordopfer *Emily*), mit denen sie aber nicht interagieren können.

4.1.2.2 Orte

Wie bereits oben erwähnt, ist die Anzahl der Schauplätze sehr überschaubar. In der Tat hat sich aber heraus gestellt, dass es möglich war, diese noch deutlich zu verkleinern ohne dass dies eine wesentliche Einschränkung darstellt. Neben dem Wegfallen der meisten abseits von Styles spielenden Handlungsstränge sind ohnehin die beteiligten Agenten von größerer Bedeutung, als der Ort, an dem die Begegnung stattfindet. Es gibt natürlich Ausnahmen wie den Tatort. Daher beschränken wir uns auf die vier Räumlichkeiten *Emilys Zimmer*, *Styles*, *Krankenhaus* und *Apotheke*. Eine weitere Differenzierung der räumlichen Gegebenheiten des Landsitzes, wie sie zwischenzeitlich von uns in Erwägung gezogen wurde, hat sich als unnötig herausgestellt.

4.1.2.3 Informationsursprung

Unter anderem aus dem Zusammenfassen einiger Personen zu einem einzigen Agenten ergibt sich die Notwendigkeit, dass Poirot nun manche Informationen, abweichend vom Roman, durch anderen Agenten erhält. Dies ist natürlich dann der Fall, wenn der ensprechende Charakter gestrichen wurde, aber z. B. auch dann, wenn zwei nun zusammengefasste Personen ursprünglich widersprüchliche Informationen gegeben haben. Einige Änderungen sind auch ein Zugeständnis an unsere Konzepte, da bei der Erkennung von Lügen (siehe Kapitel 2.5) die vom Agenten als subjektiv wahr angenommene Information von einem glaubwürdigeren Agenten kommen muss als die andere Information. Deshalb ist es stellenweise nötig gewesen, bestimmte Fakten noch einmal durch sehr glaubwürdige Agenten zu bestätigen.

4.1.2.4 zeitliche Abfolge

Auch die zeitliche Abfolge der Ereignisse konnte nicht bis ins kleinste Detail erhalten werden, da es dann in manchen Abschnitten des Szenarios unangenehm viele Sprünge gegeben hätte. Daher haben wir versucht, Interaktionen, die in einen gemeinsamen zeitlichen Kontext passen, auch zeitnah durchzuführen. So werden zum Beispiel die Zeugen anstatt nacheinander – soweit dies unproblematisch ist – alle zusammen befragt. Auch findet jetzt nur noch ein Gespräch mit dem Arzt statt, was zum Beispiel dazu führt, dass Poirot die Informationen über die Todesursache erst etwas später erhält, als das im Roman geschieht. *Poirot* untersucht manche Gegenstände (z. B. den Kakao) erst beim zweiten Betreten des Tatorts, da er zu Beginn eigentlich noch keinen Grund hat, diese mit dem Verbrechen in Verbindung zu bringen. Dies geschieht erst nach der Zeugenbefragung. Es kommt auch vor, dass wir bestimmte Agenten erst später auf *Poirot* treffen lassen, damit er entscheidende Informationen nicht immer sofort erhält und auch mal „in die falsche Richtung“ ermittelt.

4.1.2.5 Vereinfachungen

Einige Handlungsstränge oder Ausschmückungen haben wir überdies ausgelassen, damit die Anzahl der Szenen (siehe Kapitel 4.3) überschaubar bleibt. Hier einige Beispiele dazu:

- Die *Schriftanalyse* der Apothekenquittung, die letztlich *Evie* belastet, entfällt, da das dort erworbene Gift letztlich nichts mit dem Mord zu tun hat.
- Anstatt dass *John* und *Lawrence* durch die Anschrift auf dem Pakets vom Kostümversand belastet werden, geschieht dies nun einfach dadurch, dass die Kiste mit den Kostümen in ihrem Besitz ist.
- Der Test von *Poirot* und *Hastings*, ob ein umkippender Tisch auch noch im anderen Gebäudeflügel gehört werden kann, entfällt. Dies wird ohne Überprüfung angenommen.
- Während im Roman *Hastings* den entscheidenden Hinweis gibt, um den *Bekennerbrief* zu finden, endeckt *Poirot* in unserem Szenario bei einer zweiten Tatortbesichtigung die erneut verrückten Vasen eigenständig und findet so den darin versteckten Brief.
- Ebenfalls haben wir auf die Rekonstruktion der Ereignisse vor dem Mord weitestgehend verzichtet, da wir vor allem darin Schwierigkeiten gesehen haben, die dafür relevanten Fakten aus der gesamten Wissensmenge des Agenten zu identifizieren.

4.2 Inferenzbaum

Nachdem man sich für ein Szenario entschieden hat, in dem das Multiagentensystem ablaufen soll, muss als nächster Schritt eben dieses Szenario mittels logischer Regeln formalisiert werden. Da für eine direkte Modellierung der Regeln aus dem Roman heraus

das Szenario zu komplex und somit für eine korrekte und vollständige Modellierung zu unüberschaubar ist, soll schematisch ein guter Überblick des ganzen relevanten Wissens gegeben werden. Ebenso soll die Schlussfolgerungskette, die zur Überführung des Mörders führt, einigermaßen klar nachvollziehbar sein. Dieses Inferenzschema wird durch den *Inferenzbaum* dargestellt.

Der Name *Inferenzbaum* ist eigentlich ein wenig irreführend, da das Aussehen des Inferenzschemas keine echte Baumstruktur aufweist (zumindest keine, die der genauen Definition eines Baums im Sinne der Informatik entspricht). Im weitesten Sinn könnte man die vorliegende Datenstruktur noch als einen Wald bezeichnen, d.h. eine Ansammlung von Bäumen, die durch verschiedene Zusammenhangskomponenten miteinander verbunden sind. Da der Name allerdings bereits vor der Realisierung des Inferenzschemas gewählt wurde und mittlerweile die geläufige Bezeichnung für eben dieses ist, wird er auch hier weiterhin verwendet.

Die Idee hinter dem *Inferenzbaum* ist, wie bereits erwähnt, die Formalisierung und Modellierung logischer Regeln des zugrunde liegenden Romans – „The mysterious affair at styles“ – zu erleichtern. Mit Hilfe des *Inferenzbaums* soll sich für die darauf folgenden Modellierung der Regeln mittels DLV ein möglichst guter überblick über die Gesamtheit des Szenarios, sowie vor allem über die zentralen Handlungsstränge und Inferenzen des Romans ergeben. Da der *Inferenzbaum* sozusagen einen Zwischenschritt bei der Modellierung darstellt, werden sämtliche Informationen und Inferenzen noch auf natürlichsprachliche Weise repräsentiert, was vor allem der Lesbarkeit des *Inferenzbaums* zugute kommt, da die Inhalte der einzelnen Knoten selbsterklärend sind und sich seine Semantik somit von selbst erschließt. Manche Inferenzen kommen einer echten logischen Regel schon relativ nahe, während manch andere Sachverhalte bei der Ableitung einer entsprechenden logischen Regel ein wenig Raum zur Interpretation und bei der Realisierung lassen. Nicht alle Informationen und Inferenzen des *Inferenzbaum* werden später auch notwndigerweise modelliert, eventuelle Kürzungen und Vereinfachungen sind nicht ausgeschlossen.

Das der *Inferenzbaum* kein Baum im klassischen Sinn ist bzw. es keine durchgängige und stringente Schlussfolgerungskette gibt hat zweierlei Gründe. Zum einen lässt sich der Roman in kein simples Prinzip wie „aus A und B folgt C, und zusammen mit D folgt daraus dann F“ reduzieren. Damit ist gemeint, dass die Herangehensweisen und Ermittlungen und somit auch die damit verbundenen Schlussfolgerungen nicht immer vollständig durch strikte logische Regeln zu fassen sind. Diese Schwierigkeit wird erst im nächsten Schritt, der konkreten Modellierung, versucht zu lösen. Zum anderen ergibt sich bisweilen das Problem, dass im Roman nicht jeder Gedankengang und jede Inferenz einer Person vollständig preisgegeben werden. Dies ist durch die Erzählstruktur des Romans bedingt, da man nicht in die einzelnen Figuren, vor allem in die in unseren Augen wichtigste Figur des Detektivs, hineinsehen kann (von der Figur des Erzählers mal abgesehen), um so alle entsprechenden Gedanken und Schlussfolgerungen zu erfahren und nachzuvollziehen. Man sieht sich des öfteren mit der Situation konfrontiert, dass man zwar das Wissen, welches als Ausgangsbasis für bestimmte Inferenzen dient, besitzt und auch das ungefähre Ergebnis bzw. Resultat dieser Inferenz, aber über den genauen „Lösungsweg“ einer Schlussfolgerung und über die Frage wie genau und warum eine bestimmte Schlussfolgerung gezogen wurde bleibt man im Unklaren. Das Problem er-

wächst vor allem aus dem Anspruch eine Modellierung mit möglichst allgemeingültigen Regeln zu haben, d.h. die Regeln dürfen zwar durchaus romanspezifisch sein, sollen aber allgemein anwenbar sein und nicht für alle entsprechenden Fälle fest kodiert werden. Zusätzlich zu den genannten Schwierigkeiten kommt die Tatsache, dass es im Roman zwar zahlreiche Indizien gegen fast jede Person gibt und somit fast jeder als Täter in Frage kommt, allerdings sind auch viele dieser Indizien und Hinweise von der Autorin bewußt platziert, um den Leser, dessen detektivische Fähigkeiten sich meist nicht mit denen des fiktiven Dedektivs Poirot messen können, auf eine falsche Spur zu führen. Dies hat zur Folge, dass man einigen Fährten mitunter mehr Bedeutung beimisst als es der ermittelnde Detektiv Poirot tut, da man, wie bereits erwähnt, oft über seinen wirklichen Wissenszustand im Dunkeln gelassen wird.

An dieser Stelle sei darauf hingewiesen, dass der *Inferenzbaum* nur das Wissen und die Inferenzen von Poirot darstellt, da dies für die Aufklärung des Verbrechens die zentralen und auch die umfangreichsten Informationen sind. Die anderen Agenten haben weder ein so umfangreiches (Regel-)Wissen noch alle Fakten des *Inferenzbaums*. Da ihre Handlungsmöglichkeiten und Inferenzen eingeschränkt sind und auch in unserem Multiagentensystem nur eine untergeordnete Rolle spielen, können sie auch ohne ein eigenes Inferenzschema modelliert werden. Ebenfalls gibt der *Inferenzbaum* keine Auskünfte über die zeitliche Abfolge in der das Wissen zu Verfügung steht und in welcher die Schlussfolgerungen gezogen werden.

4.2.1 Aufbau des Inferenzbaums

Im *Inferenzbaum* werden die Informationen, wie Indizien, Beweise oder Aussagen, als Blätter dargestellt. Die baumähnliche Struktur und deren Verzweigungen entstehen durch die vielen Schlüsse, welche auf den gegebenen Informationen basierend, gezogen werden. Von diesen Blättern gehen nun Pfeile aus, die auf (innere) Knoten zeigen, deren Inhalt wieder neue Informationen bzw. Wissen ist. In dem „Wurzelknoten“ des *Inferenzbaums* ist dann das Ergebnis der finalen Inferenz in unserer Schlussfolgerungskette abzulesen. Dabei stehen eben diese Pfeile für eine Inferenz. Im gesamten *Inferenzbaum* wird nun das relevante Wissen, ob Grundwissen oder gefolgertes Wissen, entsprechend miteinander kombiniert und verbunden, um so die Schlussfolgerungskette bis zur finalen Inferenz, welche den Täter liefert, fortzusetzen. Ein Blick auf den gesamten *Inferenzbaum* zeigt diese Schlussfolgerungskette.

Der *Inferenzbaum* deckt die zentralen Themengebiete des Szenarios ab und lässt sich in weitere Teilbäume aufteilen. Der erste und auch größte Teil des gesamten Inferenzschemas beschäftigt sich mit allen verdächtigen Personen und ihrer Rolle im Mord, wobei jeder Verdächtige seinen eigenen Teilbaum besitzt. Das zweite Themengebiet des *Inferenzbaums* beinhaltet die Frage nach dem Mordinstrument, welches sich in unserem speziellen Fall als die Frage darstellt, wie und auf welchem Wege das Opfer vergiftet wurde. Der nächste Teil geht der verworrenen Testamentsituation etwas genauer auf den Grund, um so die entscheidenen Abläufe vor dem Mord zu rekonstruieren. Das letzte Themengebiet nimmt die Auffindung des fehlenden Beweisstücks unter die Lupe, anhand wessen der Fall letztendlich erst aufgeklärt wird. Zusätzlich gibt es noch die zwei kleinen

untergeordneten Themen: „wer hat das Mordinstrument gekauft“ bzw. „wer hat sich verkleidet“ und „wer war in der Mordnacht am Tatort“, welche sich in den Themenkomplex der verdächtigen Personen eingefügen. Zusammen betrachtet ergibt sich aus all diesen Teilen das vollständige Inferenzschema anhand dessen der Kriminalfall gelöst wird. Im Folgenden werden die Themengebiete bzw. die entsprechenden Teile des *Inferenzbaums* kurz ein wenig genauer beschrieben. Die vollständige und genaue Auflistung der Indizien und Inferenzen ist dem *Inferenzbaum* zu entnehmen.

4.2.1.1 Die Verdächtigen

Den meisten Platz im *Inferenzbaum* nimmt die Untersuchung der tatverdächtigen Personen ein. Das sind in unserem Fall: Alfred, Evie, Lawrence, John, Mary und Cynthia (in der Reihenfolge ihres Auftretens im *Inferenzbaum* von links nach rechts). Jede dieser Personen besitzt einen eigenen Teilbaum, welcher bei allen die gleiche Struktur aufweist. Wie bereits in der Einleitung erwähnt, existieren bei jeder dieser Personen mehr oder weniger starke Verdachtsmomente für und/oder gegen sie, welche durch Beweise, Indizien oder Aussagen und entsprechende Inferenzen hervorgerufen werden. Dem Umstand, dass die einzelnen Inferenzen, die aus dieser Vielzahl von Indizien hervorgehen und welche immer neue Indizien erzeugen, sowie deren Zusammenspiel untereinander und deren Gewichtung, meist nicht vollständig erfasst werden können, wird auf folgende Weise Rechnung getragen: Für jeden der Verdächtigen existiert jeweils ein mit seinem Namen ausgezeichneter *Indizien pro-* und eine *Indizien contra*-Knoten in welche dann die entsprechenden Informationen einfließen, die die jeweilige Person als verdächtig bzw. als nicht verdächtig gelten lassen. Das heißt alle „Kinderknoten“ eines *Indizien pro*-Knotens sind alle belastenden Indizien, Beweise und Aussagen, während alle „Kinderknoten“ eines *Indizien contra*-Knotens dementsprechend alle entlastenden Fakten sind. Auf diese Weise kann man die entsprechenden Inferenzen getrennt voneinander betrachten. Aus den jeweiligen *Indizien pro-* und *Indizien contra*-Knoten der verdächtigen Personen ergibt sich dann die entsprechende Schlussfolgerung, welche besagt, ob eine Person potentiell schuldig oder unschuldig ist. In unserem Szenario wird nur ausgesagt ob eine Person unschuldig ist oder ob sie unter dringendem Tatverdacht steht. Dies liegt an der Tatsache, dass zum finalen Schuldspruch und der zugehörigen Inferenz noch der entscheidende Beweis gefunden werden muss, welcher gesondert betrachtet wird. Wie genau die Inferenzen in diese „Urteilsfindung“ einfließen, wie ihr Zusammenspiel ist und wie die einzelnen Indizien gewichtet sind, geht aus der Darstellung im *Inferenzbaum* nicht hervor. Vielmehr dient der *Inferenzbaum* dazu die Vielzahl von Indizien aufzuzeigen, während deren genaue Abwägung und Beurteilung ihrer Bedeutung eine Aufgabe der Modellierung konkreter logischer Regeln ist. Es folgt eine kurze Beschreibung einiger wichtiger Indizien und deren Zusammenhang mit den einzelnen Verdächtigen.

Alfred Für Alfred als Täter spricht vor allem die Tatsache, dass er von Emilys Tod am meisten profitiert. Dies wird durch das Testament belegt, welches ihn als ihren Mann zum Haupterben erklärt. Da Alfred bedeutend jünger als Emily ist, liegt der Verdacht der Erbschleicherei ebenfalls nahe. Zusammengenommen liefert dies ein mögliches Motiv

für den Mord. Am belastendsten scheint aber ein mit Alfreds Unterschrift quittierter Kauf von Strychnin wenige Tage vor dem Mord für den es einen Zeugen gibt. Unter eigenem Namen Gift kaufen und nach einem Streit seine eigene Frau zu vergiften, wenn man wissen müsste, dass man der Hauptverdächtige ist, scheint allerdings bei einen durchaus schlauen Menschen wie Alfred sehr merkwürdig. Dies alles wird hinfällig, wenn das Ergebnis einer Schriftanalyse zeigt, dass die besagte Unterschrift nicht von Alfred stammt und der Apotheker Alfred nur flüchtig kannte und somit auch jemand anderes das Strychnin gekauft haben könnte. Gegen Alfred als Mörder spricht der Tatsache, dass er in der Mordnacht abwesend war und dafür auch ein Alibi samt Zeugen aufweisen kann. Sehr suspekt erscheint allerdings Alfred Verhalten während der gesamten Ermittlungen, da er durch sein langes Schweigen zu den Anschuldigungen den Anschein erweckt als wolle er seine Verhaftung provozieren. Nimmt man alle vorhandenen Indizien des *Inferenzbaums* für und gegen Alfred zusammen und berücksichtigt den Umstand, dass alle anderen Verdächtigen (außer Evie) ausgeschlossen werden können, kommt man zu dem Schluss, dass Alfred unter dringendem Tatverdacht steht.

Evie Für Evie als Täterin scheint zu Begin der Ermittlungen nichts zu sprechen. Da sie schon lange Zeit auf dem Anwesen Styles arbeitet und als treue Seele bei allen Bewohnern des Hauses beliebt ist, wird sie nicht verdächtigt. Entscheident ist vor allem der Umstand, dass Evie als einfache Angestellte des Hauses, die nichts erbt, in keinster Weise von dem Mord profitiert und außerdem ein Alibi besitzt, da sie in der Mordnacht ebenfalls nicht anwesend war. Im Laufe der Ermittlungen offenbaren sich allerdings einige Anhaltspunkte, die ein anderes Licht auf Evies Rolle im Mordfall werfen. Es scheint als würde Evie bewusst Beweise manipulieren. Wenn die lange offene Frage nach dem Mordinstrument, d.h. der Weg auf dem Emily vergiftet wurde, geklärt ist, erscheinen sowohl Evies Kenntnisse über Gift und ihre Zugangsmöglichkeiten, welche sie durch ihre Tätigkeit als Krankenschwester besitzt, als auch ihre Zuständigkeit für Emilys Medizin sehr verdächtig. Unterm Strich gilt auch Evie als dringend tatverdächtig.

Lawrence Gegen Lawrence Täterschaft spricht zu Beginn außer dem Umstand, dass er Emilys Sohn ist und beim Erbe erst nach Alfred und seinem Bruder John berücksichtigt wird nichts. Vielmehr liefert die neue Erbschaftsverteilung durch Emilys Hochzeit mit Alfred und die Tatsache, dass Emily nur seine Stiefmutter ist, die anstelle von ihm und John das komplette Vermögen der Familie erbte und kontrolliert, ein mögliches Motiv. Seine berufliche Erfolgslosigkeit und die finanzielle Abhängigkeit von der Mutter fügen sich da scheinbar gut ins Bild, ebenso wie seine Kenntnisse über Gift und seine Fingerabdrücke auf einer Strychninflasche. Außerdem gerät Lawrence in den Verdacht sich als Alfred ausgegeben und Stychnin gekauft zu haben, um ihm das Verbrechen anzuhängen. Eine genauere Untersuchungen bringt eine an Lawrence adressierte Bestellung bei einem Theaterkostümverleih, sowie diverse andere Verkleidungsutensilien in einer Kiste, die Lawrence und John gehören, ans Tageslicht. Lawrence war aber zur Zeit der Bestellung außer Landes. Zusätzlich legt Lawrence in der Mordnacht und vor Gericht ein sehr merkwürdiges Verhalten an den Tag und bestreitet die Todesursache Gift vehement,

obwohl er es als studierter Mediziner besser wissen sollte. Es stellt sich jedoch heraus, dass Lawrence dies nur tut, da er Cynthia für verdächtig hält und sie aus Liebe schützen will. Da Lawrence auf Poirots Befehl auch wichtige entlastende Beweise liefert ergibt sich zusammengenommen der Schluss, dass Lawrence nicht schuldig ist.

John Auch gegen Johns Täterschaft spricht zu Begin scheinbar viel. Er besitzt ein ähnliches Motiv, wie sein Bruder Lawrence, da auch er beruflich erfolglos und mit der neuen Erbfolge sehr unzufrieden ist. Im Gegensatz zu seinem Bruder erbt er allerdings den Landsitz der Familie, was ein weiterer Punkt gegen ihn zu sein scheint. Außerdem hat er sich am Tag des Mordes mit seiner Stiefmutter heftig gestritten. Desweiteren steht er im Verdacht als Alfred verkleidet das Strychnin gekauft zu haben, da die angesprochene Verkleidungkiste auch ihm gehört und das Packpapier des Pakets vom Theaterkostümverleih auf seinem Schrank gefunden wurde. Auch sein Alibi für die Zeit des Giftkaufs wirkt unglaubwürdig, da er keine Zeugen vorweisen kann, sondern nur einen anonymen Brief, der ihn zu besagter Zeit an einen entlegenen und menschenleeren Ort bestellt hat. Dazu kommt, dass die Handschrift auf der Quittung für das Strychnin der Handschrift des Briefes verblüffend ähnelt, und so die Vermutung nahe liegt, dass das Alibi selbstkonstruiert ist. Der Fund, der in der Apotheke gekauften, Strychninflasche in Johns Zimmer erhärtet den Verdacht immer mehr. Allerdings scheint es doch ein wenig merkwürdig, dass ein relativ schlauer Mensch wie John (er ist immerhin Anwalt) so einen belastenden Beweis nicht verschwinden lassen hat, vor allem da er mehr als genug Zeit dazu gehabt hätte. Es bleibt also die Möglichkeit, dass eine andere Person die Beweise platziert hat. Am Ende ist John nicht schuldig.

Verkleidung/Strychninkauf Als Unterthema der Verdächtigen gibt es einen kleinen Teil im *Inferenzbaum*, der der Frage nach dem Strychninkauf bzw. der Verkleidung nachgeht. Alle in den vorherigen Abschnitten bereits erwähnten Informationen kommen hier zusammen und führen mit weiteren Indizien gegen Evie, wie z.B. ihr und Alfreds ähnliches Erscheinungsbild, zu dem Schluss, dass Evie sich als Alfred ausgegeben und unter seinem Namen das Gift gekauft hat. Desweiteren hat sie Beweise fingiert und alle Spuren so manipuliert, dass der Verdacht auf John und Lawrence fällt.

Mary Mary profitiert von Emilys Tod nur indirekt über ihren Mann John. Sie wird nur verdächtig, da sich herausstellt, dass sie in der Mordnacht in Emilys Zimmer, dem Tatort, gewesen ist. Allerdings führt der ware Grund warum Mary im Zimmer war zu dem Schluss, dass auch sie unschuldig ist

Wer war am Tatort Ein eigener Teilbaum widmet sich der Fage wer in der Nacht am Tatort war und geht sämtlichen Spuren, wie dem Wachsfleck auf dem Teppich und dem grünen Stofffetzen an der Tür zu Cynthias Zimmer, nach, die einen letztendlich auf Mary und auch ihr Motiv hierfür führen.

Cynthia Für Cynthias Täterschaft sprechen lediglich ihr Wissen über Gift und ihre Zugangsmöglichkeiten, da sie in der Krankenhausapotheke arbeitet. Ihre Fingerabdrücke auf einer Strychninflasche im Krankenhaus sind also mit ihrem täglichen Umgang mit Medizin zu erklären. Allerdings profitiert Cynthia in keinster Weise von dem Mord, da sie als Waisin von Emily nach Styles geholt wurde und auch vom Testament nicht begünstigt wird. Im Laufe der Untersuchung der Frage „Wer war am Tatort?“ stellt sich heraus, dass Cynthia in der Mordnacht unter Schlafmitteleinfluss stand.

4.2.1.2 Das Mordinstrument

Die Klärung des Mordinstrumentes gestaltet ähnlich wie die Untersuchung der Verdächtigen. Auch hier gibt es ein paar Kandidaten die als Mordinstrument in Frage kommen und auf die es mehr oder weniger gute Indizien und Beweise gibt. Die Autopsie der Toten liefert den Befund, welcher auch schon vermutet wurde, nämlich dass die Todesursche Strychnin ist. Die entscheidene Frage, der in diesem Teil des *Inferenzbaums* nachgegengen wird, ist: *Auf welchem Weg* und *zu welcher Zeit* hat Emily das Gift verabreicht bekommen? Die beiden „Hauptverdächtigen“ sind zunächst der Kaffee und der Kakao.

Für den Kaffee spricht vor allem die Tatsache, dass er durch seinen bitteren Geschmack den ebenfalls bitteren Geschmack des Gifts gut überdecken kann. Außerdem findet sich am Tatort eine zerbrochene Kaffetasse, welche allerdings nicht mehr untersucht werden kann, was auf eine eventuelle Spurenverwischung des Täters hinweist. Gegen den Kaffee als Trägersubstanz des Gifts spricht, dass der Zeitpunkt der Symptome der Vergiftung und der anschließende Tod bei Berücksichtigung des Wirkungsgrads von Strychnin nicht zu dem Zeitpunkt passt an dem der Kaffee serviert und getrunken wurde. Als Poirot bei der Rekonstruktion der Vorgänge herausfindet, dass Emily den Kaffee gar nicht getrunken hat, scheidet der Kaffee als Mordinstrument natürlich aus.

Auf den Kakao deutet, dass auf dem Tablett auf dem der Kakao serviert wird eine strychninähnliche Substanz gesehen wurde. Außerdem hatte Emily die Angewohnheit den in der Küche zubereiteten Kakao zu später Stunde in ihrem Zimmer zu erwärmen und zu trinken, was mit der Wirkungsdauer des Gifts übereinstimmen würde. Das Kakao den bitteren Geschmack des Strychnins nicht überdecken kann spricht ebenso gegen ihn als Mordinstrument, wie die entscheidende Analyse der Kakaorückstände in der Tasse, welche nur Schlafmittel nachweist.

Durch den Ausschluss dieser Kandidaten kommt ein anderer Verdächtiger ins Spiel: die Medizin. Obwohl Emilys Medizin, die sie jeden abend nimmt, bereits einen Anteil Strychnin enthält, wird sie zunächst als Mordinstrument ausgeschlossen, da die Konzentration zu schwach ist um tödlich zu sein. Als Poirot herausfindet, dass wenn man einer strychninhaltigen Medizin eine Bromlösung zusetzt, sich das Strychnin am Boden der Medizinflasche kristallisiert, ändert sich die Sachlage. Denn dies hat zur Folge, dass sich das Gift am Boden, und damit in der letzten Dosis, in tödlicher Konzentration sammelt. Da nur Evie für die Medizin von Emily zuständig ist und sie sich, bedingt durch ihren Beruf als Krankenschwester, auch mit diesem Sachverhalt auskennt, können alle anderen Personen als Täter ausgeschlossen werden. Auch Evies Alibi für die Mordnacht ist nun hinfällig, da sie die Medizin schon weit vorher präpariert haben kann.

4.2.1.3 Das Testament

Die Frage nachdem Testament hilft vor allem bei der Rekonstruktion der Vorgänge auf dem Anwesen und liefert einen kleinen Hinweis auf den fehlenden letzten Beweis. Es finden sich Reste eines Dokuments, welches als Testament identifiziert wird, in Emilys Kamin. Dies ist ein wichtiger Hinweis, um auf einen Tatverdächtigen zu schließen. Falls nämlich das Erbe das Tatmotiv ist, so hätte der Täter unter Umständen ein Interesse ein gegebenenfalls geändertes Testament zu verbrennen. Jedoch ist zu diesem Zeitpunkt noch nicht klar, ob es sich um ein neu erstelltes Testament handelt, wodurch der Mörder unter Umständen hätte enterbt werden können. Durch die Zeugenaussagen des Personals, dass es ein neues Testament notariell beglaubigt hat, weißPoirot, dass Emily am Tag vor ihrem Tod ein neues Testament erstellt hat. Wer auch immer dies verbrannt hat ist somit tatverdächtigt. Da es angeblich einen Streit zwischen Emily und Alfred gegeben haben soll, wird Alfred zunächst verdächtigt, das Testament verbrannt zu haben. Es stellt sich aber heraus, dass sich nicht Emily und Alfred, sondern Emily und John gestritten haben, wodurch John belastet wird. Aufgrund der Tatsache, dass es am Tag der Testamentverbrennung 30 Grad warm war und der Information, dass Emily selbst den Kamin angezündet hat, ist ersichtlich, dass Emily das Testament verbrannt hat, womit auch John entlastet wird. Wenn Poirot den Fall zum Schluss rekonstruiert, wird klar warum erst ein Tetsmment geschrieben und anschließend wieder verbrannt wurde.

4.2.1.4 Der entscheidene Beweis

Der letzte Teil des *Inferenzbaums* beinhaltet das Erlangen des finalen Beweises mit dessen Hilfe Alfred und Evie erst dingfest gemacht werden können. Erst eine zufällig gemachte Bemerkung von Hastings bringt Poirot auf das fehlende Glied in der Schlussfolgerungskette, den endgültigen und belastenden Beweis. In den vorangegangenen Ermittlungen wird offenbar, dass während der Zeugenbefragung sich jemand Zutritt zum verschlossenen Tatort verschafft hat und dort etwas aus Emilys Aktenkoffer entwendet hat, den Poirot dort aufgebrochen vorfindet. Von seiner Ordnungsliebe bzw. Ordnungswahn getrieben rückt Poirot die Vasen auf dem Kaminsims gerade. Erst als Hastings Poirot dies mit einer zufälligen Bemerkung wieder in Erinnerung ruft, bemerkt Poirot, dass er diese Vasen schon bei der Tatortbegehung gerade gerückt hat und so also jemand sich an diesen Vasen zu schaffen gemacht hat. Bei der darauf folgenden Untersuchung der Vasen findet Poirot den Brief, der Alfred und Evie als Täter entgültig entlarvt.

4.3 Szenenbeschreibung und Einteilung der Szenen

Ulrich Engler

4.3.1 Hintergrund und Herausforderung

Damit das durch unser Multiagentensystem verarbeitete Szenario sowohl anschaulicher als auch einfacher hinsichtlich der verwendeten Konzepte nachzuvollziehen ist, haben

wir uns dazu entschlossen das Gesamtszenario des Romans in Szenen zu unterteilen. Dadurch wird zugleich die Modellierung strukturierter, da die einzelnen Szenen gewisse Abschlussbedingungen besitzen müssen, wie das Inferieren bestimmter Fakten oder eine erfolgreiche Revision. So haben auch einzelne Szenen ausschließlich als Ziel das bestehende Wissen durch eine neue Information zu revidieren.

Um das Gesamtszenario auch sinnvoll aufteilen zu können haben wir ersteinmal verschiedene Handlungsstränge definiert, die mitentscheidend sind, den Mörder zu finden. Die Handlungsstränge selber werden nun wiederum in mehrere Zwischenziele unterteilt. Die Zwischenziele selber können sich sowohl widersprechen, wenn das bestehende Wissen zum Ende einer Szene revidiert werden muss, als auch nur zusätzliche konsistente Informationen beinhalten.

Diese Handlungsstränge werden daraufhin auf einzelne Szenen verteilt. Dabei müssen die Szenen einige Voraussetzungen erfüllen:

- Jede Szene darf nur an einem Ort stattfinden, womit eine Ortsänderung auch einen Szenenwechsel verlangt.
- Innerhalb einer Szenen bleiben die handelnden Agenten gleich. Somit gilt auch hier, dass bei Veränderung der teilnehmenden Agenten ein Szenenwechsel notwendig ist.
- Es dürfen zwar innerhalb einer Szene mehrere Handlungsstränge thematisiert werden, jedoch darf pro Szene und Handlungsstrang nur ein Zwischenziel erreichbar sein.

Bei diesen Restriktionen und der Wahl der Szenen, sowie der Aufteilung der Handlungsstränge muss auch gewährleistet werden, dass sich die handelnden Agenten nur innerhalb des so gesteckten Rahmens bewegen können. Da das *Know-How* und das generische Wissen der einzelnen Agenten bei allen Szenen gleich bleibt besteht nur so die Möglichkeiten das Handeln der Agenten zu beschränken.

4.3.2 Handlungsstränge

Das Finden des Mörders in unserem zu modellierenden Roman läßt sich in unterschiedliche Teilgebiete unterteilen, die entweder einen Verdächtigen oder ein Indiz als Inhalt haben. Die Teilgebiete bilden die Vorlage für unsere Handlungsstränge, wobei hier schon die von uns reduzierte und modifizierte Version des Romans als Basis dient (Kapitel 4.1). Insgesamt gibt es nach der Strukturierung sechs Handlungsstränge die dann auf die Szenen verteilt werden müssen. Folgende Handlungsstränge wurden ausgewählt:

1. **Mordinstrument: Was war das Mordinstrument und wer war dafür verantwortlich?**

   Dieser Handlungsstrang thematisiert das verwendete Mordinstrument, was letztendlich die mit Brom angereicherte Medizin war, wodurch sich die Strychninkonzentration erhöht hat. Nachdem Strychnin als Todesursache festeht gibt es für Poirot nur eine begrenzte Anzahl an möglichen Mordinstrumenten, nämlich Emilies

Kaffee, Kakao oder eben Medizin. Ziel für Poirot muss also lauten entweder das richtige Mordinstrument zu identifizieren oder die beiden anderen auszuschließen. Innerhalb des Romans entstehen so zwei Zwischenziele für Poirot um die Medizin als Mordinstrument zu identifizieren.

a) Der Kaffee wird ausgeschlossen, da ihn Emily nicht gedrunken haben kann, weil er auf einem kaputten Tisch abgestellt worden ist, wodurch die Tasse zerbrach.

b) Der Kakao wird durch eine Analyse als Mordinstrument ausgeschlossen, da sich in ihm kein Strychnin nachweisen lässt.

2. **Strychnin: Wer hat das Gift besorgt?**

   Nachdem Poirot herausfindet, dass Strychnin die Todesursache ist, ist es ebenfalls entscheidend herauszufinden, wer Kontakt zu Strychnin hat. Dieser Fakt würde die entsprechende Person demnach stark belasten.

   a) Zuerst wird Alfred eine mutmaßliche Verbindung zu Strychnin nachgewiesen, da er anfangs als einziger auf die Beschreibung des Apothekers, bei dem Strychnin zuvor gekauft worden ist, passt.

   b) Alfred wird durch ein Alibi entlastet. Zudem wird eine Verkleidung gefunden, die darauf schließen lässt, dass sich John oder Lawrence verkleidet haben könnten, um das Strychnin in der Apotheke zu kaufen.

   c) Im Krankenhaus werden auf einer Strychninflasche die Fingerabdrücke von Lawrence entdeckt, wodurch er zusätzlich belastet wird.

   d) Das Personal bezeugt, dass Evie die Verkleidung in die Kiste gelegt hat, womit John und Lawrence diesbezüglich entlastet werden.

   e) Nachdem Poirot herausfindet, dass die Medizin das Mordinstrument ist und so das Strychnin verabreicht worden ist, werden sämtliche bisher mit Strychnin in Verbindung gebrachten Personen entlastet, da feststeht, dass das in der Apotheke gekaufte und auch das im Krankenhaus befindliche Gift nichts mit dem Mord zu tun haben.

3. **Testament: Wer hat das Testament verbrannt?**

   Ebenfalls ein wichtiger Hinweis, um auf einen Tatverdächtigen zu schließen, ist das verbrannte Testament. Falls nämlich das Erbe das Tatmotiv ist, so hätte der Täter unter Umständen ein Interesse ein gegebenenfalls geändertes Testament zu verbrennen.

   a) Es wird ein verbrannter Zettel gefunden, bei dem es sich um Testament handelt. Jedoch ist zu diesem Zeitpunkt noch nicht klar, dass es sich um ein neu erstelltes Testament handelt, wodurch der Mörder unter Umständen hätte enterbt werden können.

b) Durch die Information des Personals, dass es ein neues Testament notariell beglaubigt hat, weiß Poirot, dass Emily am Tag vor ihrem Tod ein neues Testament erstellt hat. Wer auch immer dies verbrannt hat ist somit tatverdächtigt. Da es angeblich einen Streit zwischen Emily und Alfred gegeben haben soll, wird Alfred verdächtigt, das Testament verbrannt zu haben.

c) Es stellt sich heraus, dass sich nicht Emily und Alfred, sondern Emily und John gestritten haben, wodurch John belastet wird.

d) Aufgrund der Tatsache, dass es am Tag der Testamentverbrennung 30°C warm war und der Information, dass Emily selbst den Kamin angezündet hat, ist ersichtlich, dass Emily das Testament verbrannt hat, womit auch John entlastet wird.

4. **Wer war im Zimmer?**

Hier wird die Suche nach der Person thematisiert, die sich neben Emily ebenfalls in der Mordnacht in Emilies Zimmer befunden haben muss.

a) Durch ein gefundenes Stück Kleidung im Türrahmen wird ersichtlich, dass sich eine weitere Person in der Mordnacht in Emilies Zimmer befunden haben muss.

b) Das gefundene grüne Kleidungsstück lässt sich nur mit Mary in Verbindung bringen, da sie die einzige ist, die grüne Kleidung besitzt.

5. **Wo ist die fehlende Kaffeetasse?**

Durch die Tatsache, dass ein Getränk das Mordinstrument sein muss, sind sämtliche Kaffeetassen von Interesse und wer Kaffee am Mordabend getrunken hat.

a) Durch eine Befragung wird festgestellt, wer alles am Mordabend Kaffee getrunken hat. So wird festgestellt, dass eine Kaffeetasse fehlen muss.

b) Cynthia berichtet, dass sie ihren Kaffee ohne Zucker trinkt. Die fehlende Tasse gehört demnach ihr.

c) Um Cynthia zu entlasten findet Lawrence die fehlende Tasse, da er sich in Styles besser auskennt und so mögliche Verstecke untersuchen kann.

6. **Der entscheidende Brief**

Durch den gefundenen Brief von Alfred an Evie wird der Täter und die Mittäterin letzendlich überführt.

a) Alfred sucht den Brief in Emilies Zimmer und versteckt ihn in den Vasen.

b) Durch die erneut ungeordneten Vasen wird Poirot auf die Vasen aufmerksam, wodurch er sie untersucht und so den entscheidenden Brief findet.

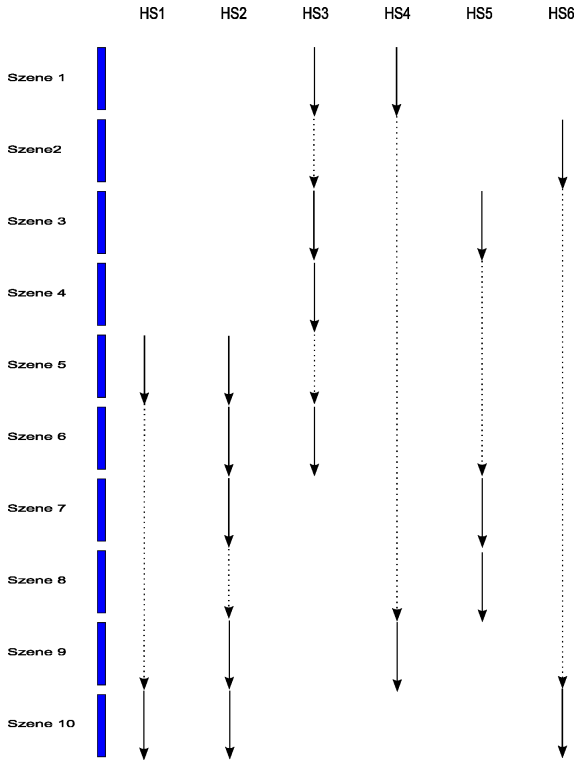

Abbildung 4.1: Szeneneinteilung

4.3.3 Szeneneinteilung

Die oben beschriebenen Handlungsstränge mussten nun auf Szenen verteilt werden. Ziel war hier eine Einteilung zu finden, bei denen die geeigneten Personen am passenden Ort zusammenkamen, um die Zwischenziele der einzelnen Handlungsstränge erzielen zu können. Insgesamt entstanden so zehn Szenen, wobei einige relativ umfangreich sein können und andere auch sehr kurz, wenn etwa nur eine Revision oder eine zusätzliche Information notwendig ist. Die bestimmenden Charakteristika der einzelnen Szenen sind dabei also die handelnden Agenten, der Ort, die betreffenden Handlungsstränge und auch die wichtigsten Szenenendbedingungen. Die Szeneneinteilung ist in der Abbildung 4.1 zusammengefasst und sieht schlussendlich folgendermaßen aus:

1. **Tatortuntersuchung**

   Poirot und Hastings befinden sich am Tatort und untersuchen etwaige Hinweise bzw. verdächtige Gegenstände, wie das an der Tür gefundene grüne Kleidungsstück. Zudem wird ein verbrannter Zettel gefunden, bei dem es sich um ein Testament handelt, da sich zum einen der Begriff „ment“ auf dem Papier befindet und es zum anderen aus dickem Papier besteht, was für Testamente verwendet wird. Viele Hinweise haben zu diesem Zeitpunkt jedoch noch keine Bedeutung, da Poirot noch nicht weiß, dass Strychnin die Todesursache ist. So werden sämtliche Getränke

noch nicht weiter untersucht. Außerdem wird der im Koffer befindliche Zettel nicht gefunden, da Poirot solche Gegenstände nicht ohne die Anwesenheit eines Anwalts öffnet. Ebenfalls entscheidend in dieser Szene ist die Tatsache, dass Poirot die ungeordneten Vasen ordnet.

- **Ort:** Emilies Raum
- **Agenten:**
  - Poirot
  - Hastings
- **Handlungsstränge:**
  - 3a
  - 4a
- **Szenenendbedingungen:**
  - verbrannter Zettel wird gefunden
    → bei dem verbrannten Zettel handelt es sich um ein Testament
  - Teil eines grünen Kleidungsstücks wird gefunden
    → eine weitere Person befand sich im Raum
  - die Vasen sind geordnet

2. **Alfred versteckt den Brief**

   In dieser Szene wird Alfred, da er sich allein am Tatort befindet dazu veranlasst, nach Beweisen gegen ihn zu suchen. Dabei sucht er überall dort, wo er meint, dass auch Poirot nachschauen würde. So findet er den Brief, den er daraufhin an einem geeigneten Platz versteckt. In diesem Raum kommen nur die Vasen als Versteck in Frage. Damit sorgt er jedoch dafür, dass die Vasen erneut ungeordnet sind.

   - **Ort:** Emilies Raum
   - **Agenten:**
     - Alfred
   - **Handlungsstränge:**
     - 6a
   - **Szenenendbedingungen:**
     - Alfred findet und versteckt den Brief
     - die Vasen sind ungeordnet

3. **Was wissen die Agenten über Emilies Tod?**

   Dies ist die Szene in der Poirot zum ersten Mal auf viele mögliche Verdächtige stößt. Das veranlasst ihn dazu bestimmte wichtige Fragen bezüglich der Mordnacht, etwaiger Motive und der Todesursache zu stellen. Außerdem nutzt er die Gelegenheit zu

erfragen, ob jemand das Testament notariell beglaubigt hat, so dass das gefundene Testament als neu angesehen werden kann, woraufhin er sofort Alfred verdächtigt. Poirot hat nämlich auch durch die Frage nach den Motiven erfahren, dass sich Alfred und Emily gestritten haben. Diese Information beruht jedoch auf einer Lüge durch Mary, da sie ihren Mann, der in Wirklichkeit statt Alfred in dem Streit involviert war, schützen will. Zudem lügt Lawrence in Bezug auf die Todesursache, da er persönlich Cynthia verdächtigt, die er daraufhin schützen will.

- **Ort:** Styles
- **Agenten:**
  - Poirot
  - Hastings
  - Personal
  - Mary
  - Cynthia
  - John
  - Lawrence
- **Handlungsstränge:**
  - 3b
  - 5a
- **Szenenendbedingungen:**
  - Mary berichtet von einem Streit zwischen Alfred und Emily
    → Alfred wird verdächtigt, da er das Testament verbrannt hat
  - eine Kaffeetasse fehlt
  - Lawrence äußert Vermutung des Unfalls über Todesursache

4. **Neue Informationen zum Streit**

   In dieser Szene wird Poirot lediglich darüber informiert, dass sich John und Emily gestritten haben und nicht Alfred und Emily. Poirot glaubt eher dieser Information, da sie vom Peronal, was er als vertrauenswürdiger einstuft, stammt. Es ist sogar so vertrauensürdig, dass ab diesem Zeitpunkt Mary als Lügnerin für ihn feststeht. In Folge dessen wird jetzt John der Testamentverbrennung verdächtigt.

   - **Ort:** Styles
   - **Agenten:**
     - Poirot
     - Hastings
     - Personal
   - **Handlungsstränge:**
     - 3c

- **Szenenendbedingungen:**
  - Personal berichtet von einem Streit zwischen Emily und John
    → Alfred wird entlastet
    → John wird belastet, da er das Testament verbrannt hat
    → Mary wird der Lüge überführt

5. **Aussagen des Apothekers**

   In dieser Szene erfährt Poirot die wirkliche Todesursache, nämlich Strychnin. Damit wird auch Lawrence der Lüge überführt. Wilkinstein liefert zudem die Beschreibung einer Person, die Strychnin erst vor kurzem gekauft hat. Diese Beschreibung passt zu diesem Zeitpunkt ausschließlich auf Alfred.

   - **Ort:** Apotheke
   - **Agenten:**
     - Poirot
     - Hastings
     - Wilkinstein
   - **Handlungsstränge:**
     - 1a
     - 2a
   - **Szenenendbedingungen:**
     - Wilkinstein liefert die wahre Todesursache: Strychnin
       → Lawrence wird der Lüge überführt
       → Verdacht auf Lawrence, da er bezüglich der Todesursache gelogen hat, obwohl er selber Arzt ist
     - Wilkinstein liefert eine Beschreibung des Strychninkäufers
       → Verdacht auf Alfred, da er auf die Beschreibung passt

6. **Alfred hat das Strychnin nicht gekauft**

   In dieser Szene berichtet das Personal, dass Alfred für den vermuteten Strychninkauf ein Alibi besitzt. Somit wird er als Käufer ausgeschlossen. Da Poirot aber auch eine Verkleidung als Möglichkeit betrachtet, wird so die Verkleidungskiste, die John und Lawrence gehört, verdächtig und untersucht. Er findet auch eine entsprechende Verkleidung, womit nun John und Lawrence mit dem Strychninkauf in Verbindung gebracht werden. Außerdem klärt diese Szene die Testamentverbrennung auf, da das Personal die Information liefert, dass Emily selbst den Kamin angezündet hat. In Zusammenhang mit den 30°C am Tag vor dem Mord, steht nun fest, dass Emily selbst das Testament verbrannt hat. Aufgrund der Tatsache, dass Strychnin die Todesursache ist, fragt Poirot nach den Getränken. So stellt er fest, dass eine Kaffeetasse fehlt.

- **Ort:** Styles
- **Agenten:**
  - Poirot
  - Hastings
  - Personal
- **Handlungsstränge:**
  - 2b
  - 3d
- **Szenenendbedingungen:**
  - Personal berichtet, dass Emily selbst den Kamin angezündet hat
    → Emily hat selbst das Testament verbrannt
  - Personal liefert Alibi für Alfred bezüglich des Strychninkaufs
    → Poirot findet Verkleidung
    → John oder Lawrence könnten sich verkleidet haben

7. **Cynthia im Krankenhaus**

   Diese Szene ist wichtig, um auch Lawrence mit Strychnin in Verbindung zu bringen, da auf einer Strychninflasche im Krankenhaus seine Fingerabdrücke zu finden sind. Zudem liefert Cynthia die Information, dass sie ihren Kaffee ohne Zucker trinkt, woraufhin feststeht, dass die fehlende Kaffeetasse ihr gehört.

   - **Ort:** Krankenhaus
   - **Agenten:**
     - Poirot
     - Hastings
     - Cynthia
   - **Handlungsstränge:**
     - 2c
     - 5b
   - **Szenenendbedingungen:**
     - Fingerabdrücke von Lawrence auf der Strychninflasche
       → Verdacht auf Lawrence
     - die fehlende Tasse gehört Cynthia

8. **Lawrence findet die fehlende Tasse**

   Da für Poirot feststeht, dass Cynthias Kaffeetasse fehlt und er Lawrence darüber informiert, wird dieser dazu veranlasst diese Tasse zu suchen. Da er mögliche Verstecke der Kaffeetasse kennt, findet er sie, im Gegensatz zu Poirot. Durch das darin gefundene Schlafmittel wird Cynthia entlastet.

- **Ort:** Styles
- **Agenten:**
  - Poirot
  - Hastings
  - Lawrence
- **Handlungsstränge:**
  - 5c
- **Szenenendbedingungen:**
  - Lawrence findet Cynthias Kaffeetasse
    → in Cynthias Kaffee befindet sich Schlafmittel
    → Cynthia wird als mögliche Täterin ausgeschlossen

9. **Mary war im Zimmer und Evie wird verdächtigt**

   Durch Informationen des Personals, dass Evie die Verkleidung in John und Lawrence Kiste versteckt hat, werden zum einen John und Lawrence entlastet und zum anderen Evie verdächtigt. Darüber hinaus klärt sich auch, dass Mary in der Mordnacht im Zimmer war.

   - **Ort:** Styles
   - **Agenten:**
     - Poirot
     - Hastings
     - Personal
     - Mary
   - **Handlungsstränge:**
     - 2d
     - 4b
   - **Szenenendbedingungen:**
     - Personal bezeugt, dass Evie die Verkleidung versteckt hat
       → John und Lawrence werden entlastet
       → Evie wird verdächtigt
     - Personal liefert den Hinweis, dass Mary doch grüne Kleidung besitzt
       → Mary war im Zimmer

10. **Alfred und Evie werden überführt**

    In dieser entscheidenden Szene, in der sich Poirot zum ersten Mal seit der ersten Szene wieder am Tatort befindet, wird der Mord letztendlich aufgeklärt. Dies liegt an zwei Tatsachen. Poirot besitzt jetzt das Wissen über die Todesursache und untersucht somit auch den Kakao, in dem kein Strychnin nachzuweisen ist. Damit

steht die Medizin als Mordinstrument fest. Die Medizin wird mit Evie in Verbindung gebracht. Zudem fällt Poirot auf, dass die Vasen erneut ungeordnet sind. Damit werden die Vasen verdächtig und so untersucht. Poirot findet daraufhin den belastenden Brief und entlarvt so Alfred und Evie als Mörder Emilies.

- **Ort:** Emilies Raum
- **Agenten:**
  - Poirot
  - Hastings
- **Handlungsstränge:**
  - 1b
  - 2e
  - 6b
- **Szenenendbedingungen:**
  - Im Kakao befindet sich kein Strychnin
    → Kakao wird als Mordinstrument ausgeschlossen
    → Die Medizin ist das Mordinstrument
  - Poirot findet den Brief
    → Alfred und Evie sind die Mörder

4.4 Modellierung in DLV

Daniela Kölling

4.4.1 Prädikate

Im Zusammenhang mit der Umsetzung des Romans wurden verschiedene Prädikate verwendet. Dabei handelt es sich beispielsweise um Prädikate, welche Eigenschaften von Personen oder von Objekten beschreiben, Prädikate, die Relationen zwischen Personen oder Objekten beschreiben, Prädikate, welche die Umwelt näher beschreiben, oder Prädikate, die Aktionen widerspiegeln. Die verwendeten Prädikate sind von Agent zu Agent und von Szene zu Szene unterschiedlich. Die folgende Auflistung stellt eine Beschreibung der im Szenario verwendeten Prädikate dar. Zur Verwendung der Prädikate folgen in den nächsten Kapiteln einige Beispiele.

Es wurde versucht, die Prädikate aussagekräftig zu benennen. Zusammengesetzte Namen (z. B. *access to*) werden ohne Trennzeichen zusammen geschrieben, wobei jedes neue Wort großgeschrieben wird. Prädikate, die mit *Do* beginnen, bezeichnen die Ausführung einer Aktion. Prädikate, die mit *Env* beginnen, bezeichnen Prädikate, die der Agent von der Umwelt mitgeteilt bekommt.

+(X, Y, Z) Beschreibt die Addition zweier Zahlen X und Y mit dem Ergebnis Z. Entspricht der Formel $X + Y = Z$.

!=(X, Y) X soll ungleich Y sein. Dabei können X und Y sowohl Personen als auch Zahlenwerte sein.

 # int(X) Dieses Fakt ist wahr, wenn es sich bei X um einen Integerwert handelt.

<(X, Y) Der Wert X ist kleiner als der Wert Y. Entspricht der Formel $X < Y$.

Action(A) Bei A handelt es sich um eine Tätigkeit. Diese Aktion kann durch andere Prädikate näher beschrieben werden.

ActionPerson(Activity, Name) Dieses Prädikat beschreibt die Tätigkeit *Activity* näher und gibt eine an der Tätigkeit beteiligte Person *Name* an.

ActionPlace(Activity,Place) Dieses Prädikat beschreibt die Tätigkeit *Activity* näher, welche am Ort *Place* stattfindet.

ActionTime(Activity, Time) Dieses Prädikat beschreibt die Tätigkeit *Activity* näher und gibt den Zeitpunkt *Time* an, an dem die Tätigkeit stattfindet.

Age(Name, X) Die Person *Name* ist X Jahre alt.

Analysis(X,Y) Beschreibt das Ergebnis eines Labortests in dem Substanz Y in Substanz X nachgewiesen wurde.

Appearance(Name, Attribute) Die Person *Name* hat das Aussehen *Attribute*. Dabei kann es sich bei *Attribute* um einen Bart, eine Brille, etc. handeln.

ArgumentSemantic(X, Y, Z) Hierbei handelt es sich um ein Hilfsprädikat für die Generierung von Lügen. X steht für das Prädikat über das gelogen wird. Z ist eine nähere Beschreibung der Stelle Y des Prädikates, so dass der Agent eine semantische korrekte Antwort geben kann.

Behaviour(X) Bei X handelt es sich um ein Verhalten, das ein Agent zeigen kann.

BelongsTo(Item, X) Der Gegenstand *Item* gehört zu X. Dabei kann es sich bei X um Personen, aber auch Orte oder Gegenstände handeln.

Bottle(Item) Der Gegenstand *Item* ist eine Flasche.

Briefcase(Item, Scene) Bei *Item* handelt es sich um eine Aktentasche, die in der Szene *Scene* vorkommt.

Broken(Item, Scene) Der Gegenstand *Item* ist in der Szene *Scene* kaputt.

Brother(Name,Name2) Die Person *Name* ist der Bruder der Person *Name2*.

Burning(Name, Item, Time) Der Gegenstand *Item* wurde von der Person *Name* zum Zeitpunkt *Time* verbrannt.

BurningTestament(Name) Die Person *Name* hat ein Testament verbrannt.

Burnt(Item, Name) Der Gegenstand X wurde von der Person *Name* verbrannt.

BurntIn(Item, Place) Der Gegenstand *Item* wurde am Ort *Place* verbrannt.

Cacao(Item, Scene) Bei *Item* handelt es sich um einen Kakao, der in der Szene *Scene* vorkommt.

CacaoCup(Item, Scene) Bei *Item* handelt es sich um eine Kakaotasse, die in der Szene *Scene* vorkommt.

Caring(Name1, Name2) Die Person *Name1* kümmert sich um die Person *Name2*.

CauseOfBurningTestament(Reason) *Reason* ist ein Grund, um ein Testament zu verbrennen.

CauseOfDeath(Reason) Bei *Reason* handelt es sich um die Todesursache.

CauseOfDeathQueryPredicate(X) Bei X handelt es sich um ein Prädikat, für das eine Todesursache erfragt werden kann.

CauseOfMakingNewTestament(Reason) *Reason* ist ein Grund dafür, ein neues Testament anzufertigen.

Clock(Item, Scene) Der Gegenstand *Item* ist eine Uhr, die in der Szene *Scene* vorkommt.

Closed(X) Der Gegenstand bzw. die Tür X ist verschlossen.

Coffee(Item, Scene) Bei *Item* handelt es sich um Kaffee, der in der Szene *Scene* vorkommt.

CoffeeCup(Item, Scene) Der Gegenstand *Item* ist eine Kaffeetasse, die in der Szene *Scene* vorkommt.

Color(X) X ist eine Farbe.

ColorOf(Item,X) *Item* hat die Farbe X.

Container(Item, Scene) Bei *Item* handelt es sich um einen Behälter, einen Gegenstand, der andere Gegenstände aufnehmen kann, welcher in der Szene *Scene* vorkommt.

Contains(A,X) A beeinhaltet X. Dabei kann A ein Gegenstand oder ein Ort sein.

Cup(Item) Der Gegenstand *Item* ist eine Tasse.

CurrentLocation(Place) Bei *Place* handelt es sich um den aktuellen Aufenthaltsort.

CurrentScene(Scene) Bei *Scene* handelt es sich um die aktuelle Szene.

Dirt(X, Scene) X ist Schmutz, der in der Szene *Scene* vorkommt.

Disinheriting(Name1, Name2) Die Person *Name1* enterbt die Person *Name2*.

Disputing(Name, Name2, Time) Die Personen *Name1* und *Name2* haben sich zum Zeitpunkt *Time* gestritten.

DoHide(OldPlace, NewPlace , Item) Beschreibt die Ausführung einer Aktion, einen Gegenstand *Item* von Ort *OldPlace* zu nehmen und an Ort *NewPlace* zu verstecken.

DoInvestigate(X) Beschreibt die Ausführung einer Aktion, X zu untersuchen. Dabei kann X ein Gegenstand oder eine Tür sein.

DoOpen(X) Beschreibt die Ausführung einer Aktion, einen Gegenstand oder eine Tür X zu öffnen.

Door(X, Scene) X ist eine Tür, die in Szene *Scene* vorkommt.

DoorFromTo(X,Room,Room1) Bei X handelt es sich um eine Tür zwischen den Räumen *Room* und *Room1*.

DoQueryElementInfo(X, Name) Ein Agent *Name* soll nach der Instanz X gefragt werden.

DoQueryInstantiate(X, Name) Ein Agent *Name* soll nach dem Prädikat X gefragt werden.

DoTidyUp(Item) Beschreibt die Ausführung einer Aktion, einen Gegenstand X zu ordnen bzw. aufzuräumen.

Drink(Item) Bei dem Gegenstand *Item* handelt es sich um ein Getränk.

Eating(Name) Die Person *Name* hat etwas gegessen.

EnvAttribOfThing(X, Y, Scene) X ist in der Szene *Scene* eine Eigenschaft von Y bzw. etwas, was Y näher beschreibt.

EnvChange(Item, Y) Der Gegenstand *Item* wird in den Zustand Y versetzt.

EnvChangedBy(Item, X, Name, Scene) Der Zustand des Gegenstandes *Item* wurde von der Person *Name* in der Szene *Scene* in den Zustand X geändert.

EnvCharacter(Name, Scene) Die Person *Name* kommt in der Szene *Scene* vor.

EnvInvestigate(Item) Der Gegenstand *Item* wird untersucht.

EnvInvestigated(Item, Scene) Der Gegenstand *Item* wurde in der Szene *Scene* untersucht.

EnvLocation(Place, Scene) Der Ort *Place* ist der Handlungsort der Szene *Scene*.

EnvMadeOf(Item, Y) Der Gegenstand *Item* ist aus dem Material Y.

EnvMove(Item, OldPlace, NewPlace) Der Gegenstand *Item* wird von Ort *OldPlace* zu Ort *NewPlace* bewegt.

EnvScene(Scene) Bei *Scene* handelt es sich um die aktuelle Szene.

EnvStateTransition(Item, Old, New, Scene) Der Gegenstand *Item* wurde in der Szene *Scene* vom Zustand *Old* in den Zustand *New* geändert.

EnvThing(Item, Scene) *Item* ist ein Gegenstand, der in Szene *Scene* vorkommt.

EnvThinAddable(Container, Item, Scene) *Item* kann dem Gegenstand *Container* in der Szene *Scene* hinzugefügt werden.

EnvThingContained(Container, Item, Scene) In Szene *Scene* enthält der Gegenstand *Container* den Gegenstand *Item*.

Evidence(X) X ist ein Beweisstück.

Exculpatory(Name) Die Person *Name* ist entlastet.

Fact(X) X ist ein Fakt.

FailedCareer(Name, Profession) Die Person *Name* hat eine gescheiterte Karriere als *Profession*.

FinancialDependent(Name1,Name2) Die Person *Name1* ist finanziell abhängig von der Person *Name2*.

FinancialDifficulties(Name) Die Person *Name* steckt in finanziellen Schwierigkeiten.

FinishedScene(Scene) Die Szene *Scene* ist beendet.

Furniture(Item) Bei *Item* handelt es sich um ein Möbelstück.

Generous(Name) Die Person *Name* ist grosszügig.

HasProfession(Name,P) Die Person *Name* hat den Beruf *P*.

Hidden(Item) Der Gegenstand *Item* ist versteckt.

HidingPlace(Place, Item, Scene) Der Ort *Place* ist in Szene *Scene* ein Versteck für den Gegenstand *Item*.

HomeAlone(Name) Die Person *Name* ist alleine zu Hause bzw. alleine im Raum.

Hour(Time) Zeitzähler im Stundenformat. Bei der Zahl *Time* handelt es sich um eine Stunde.

Inflaming(Name, Item, Time) Die Person *Name* hat den Gegenstand *Item* zum Zeitpunkt *Time* angezündet.

Ingredient(X, Scene) X ist in der aktuellen Szene ein Inhaltsstoff.

Inheriting(Name1, Name2) Die Person *Name1* vererbt einen Teil ihres Vermögens an die Person *Name2*.

Investigated(Item, Scene) Der Gegenstand *Item* wurde in der Szene *Scene* untersucht.

KeepSecret3(W, X, Y, Z) Der Agent verschweigt sein Wissen über das Prädikat W, welches aus den Variablen X, Y und Z besteht.

KnowingWhoBurntThing(Item) Der Agent weiß auch, wer den Gegenstand *Item* verbrannt hat.

Knowledge(Name, Y) Die Person *Name* hat Wissen über Y.

LegacyHunter(Name) Die Person *Name* ist ein Erbschleicher.

LieAbout1(X, Y) Der Agent lügt über das einstellige Prädikat X mit der Variable Y.

LieAbout3(W, X, Y, Z) Der Agent lügt über das dreistellige Prädikat W, welches aus den Variablen X, Y und Z besteht.

Location(X,Place, Day, Hour) Die Person oder der Gegenstand X ist am Tag *Day* zur Stunde *Hour* am Ort *Place*.

LocationDay(X, Place, Day) Die Person oder der Gegenstand X ist am Tag *Day* am Ort *Place*.

LocationRelative(X, Place , Time) Die Person oder der Gegenstand X ist zum Zeitpunkt *Time* am Ort *Place*.

LocationTimeless(X, Place) Die Person oder der Gegenstand X befindet sich zu einem nicht näher bestimmten Zeitpunkt am Ort *Place*.

MadeOf(Item, Y) Der Gegenstand *Item* ist aus dem Material Y.

MainSuspect(Name) Die Person *Name* ist der Hauptverdächtige.

MakingNewTestament(Name) Die Person *Name* hat ein neues Testament gemacht.

MakingNewTestamentAt(Name, Time) Die Person *Name* hat zum Zeitpunkt *Time* ein neues Testament gemacht.

Married(Name1, Name2) Die Personen *Name1* und *Name2* sind verheiratet.

Medicine(Item, Scene) Der Gegenstand *Item* ist in der aktuellen Szene eine Medizin.

MedicineBottle(Item, Scene) Bei *Item* handelt es sich um eine Medizinflasche.

Motive(X) X ist ein (Mord-) Motiv.

MotiveQueryPredicate(X) Mit diesem Prädikat kann der Agent nach einem Motiv für das Prädikat X fragen.

Murderer(Name) Die Person *Name* ist der Mörder.

MyName(Name) *Name* ist der Name des Agenten.

MyRoom(Place) *Place* ist das Zimmer, das dem Agenten gehört.

NonCriminalDeath(Reason) *Reason* ist eine nicht kriminelle Todesursache, z.B. ein Unfall.

Oiled(Item, Scene) Der Gegenstand *Item* wurde geölt.

On(Item, Scene) Der Gegenstand *Item* war an.

Opened(X, Scene) Der Gegenstand oder die Tür X ist offen.

Ordered(Item, Scene) Der Gegenstand *Item* befindet sich in einem ordentlichen Zustand.

Paper(Item, Scene) Der Gegenstand *Item* ist ein Papier.

PartOf(X, Y) X ist ein Teil von Y.

PartOfFurniture(X, Scene) X ist ein Möbelteil.

Person(Name) *Name* ist eine im Szenario vorkommende natürliche Person.

Place(P) P ist ein Ort.

Poison(Item) Bei *Item* handelt es sich um ein Gift.

PoisonBottle(Item, Scene) Bei dem Gegenstand *Item* handelt es sich um eine Giftflasche.

Profession(P) P ist ein Beruf (z.B. Krankenschwester oder Anwalt).

Proof(X) Bei X handelt es sich um einen Beweis.

QueryDone(X, Y, Name, Time) Dem Agenten *Name* wurde die Frage X nach der Instanz oder dem Prädikat Y zum Zeitpunkt *Time* gestellt.

QueryDoneAnytime(X, Y, Name) Dem Agenten *Name* wurde die Frage X nach Y gestellt.

RecentlyOpened(X) Der Gegenstand bzw. die Tür X wurde vor kurzem geöffnet.

Relatives(Name1, Name2) Die Personen *Name1* und *Name2* sind miteinander verwandt.

Responsible(Name, X) Person *Name* ist für X verantwortlich. Dabei kann X z. B. ein Vorgang oder ein Gegenstand sein.

RoomOfCrime(Place) *Place* ist der Tatort. Im Gegensatz zu *SiteOfCrime(Place)* handelt es sich hierbei um den Raum, in dem das Opfer ums Leben kam.

SceneRunning(Scene) *Scene* ist die Szene die aktuell läuft.

SimiliarAppearance(Name1, Name2) Die Personen *Name1* und *Name2* haben ein ähnliches Aussehen.

Stepmother(Name1, Name2) Die Person *Name1* ist die Stiefmutter der Person *Name2*.

Suspect(Name) Die Person *Name* ist verdächtig.

Sympathy(Name1, Name2, Z) Die Person *Name1* empfindet der Person *Name2* gegenüber eine Sympathie von Z. Wobei der Wert 10 Liebe und der Wert 1 starke Abneigung ausdrückt.

SympathyValue(X) Bei der Zahl X handelt es sich um einen Sympathiewert.

TemperatureDay(X, Day) Die Temperatur am Tag *Day* betrug X Grad.

TemperatureTime(X, Time) Die Temperatur zum Zeitpunkt *Time* betrug X Grad.

TestamentRequiredElement(X) X wird für ein gültiges Testament benötigt.

Textile(Item, Scene) Bei dem Gegenstand *Item* handelt es sich um ein Textil.

Thing(Item) *Item* ist ein Gegenstand (Tasse, Testament, ...).

TimeOfDeath(Time) Der Zeitpunkt *Time* ist der Todeszeitpunkt.

TimeOfDisputeWithEmily(Time) Zum Zeitpunkt *Time* fand ein Streit mit Emily statt.

TimeOfEffect(X, Y, Z) Die Wirkung des Mittels X auf Z tritt nach Y Stunden auf.

Timestamp(X, Day, Time) Das Ereignis X fand am Tag *Day* zum Zeitpunkt *Time* statt.

TimestampDay(E,Day) Das Ereignis E fand am Tag *Day* statt.

TimestampRelative(E,Time) Das Ereignis E fand zum Zeitpunkt X statt.

ToHide(Item, Scene) Der Gegenstand *Item* muß versteckt werden.

Unordered(Item, Scene) Der Gegenstand *Item* ist unordentlich.

Victim(Name) Die Person *Name* ist das Mordopfer.

WasInsideRoomOfCrime(Name) Die Person *Name* war am Tatort.

WasOpened(Item) Der Gegenstand *Item* wurde geöffnet.

Wax(Item, Scene) Bei *Item* handelt es sich um Wachs.

Wealthy(Name) Die Person *Name* ist wohlhabend.

WingAffiliation(R,W) Der Raum *R* gehört zum Flügel *W*.

Workplace(Name, Place) Die Person *Name* arbeitet am Ort *Place*.

4.4.2 Initiales Wissen

Das Wissen der Agenten kann in zwei Bereiche unterteilt werden. Das gemeinsame Wissen, welches die Agenten teilen und das spezielle Wissen, welches für jeden Agenten unterschiedlich ist. Dabei unterscheidet sich Poirot von den anderen Agenten besonders durch seine Menge an Regeln. Poirot besitzt durch seine Rolle als Hauptcharakter, welcher den Mordfall lösen soll, mehr Regeln, aber auch die Möglichkeit mehr Aktionen auszuführen, als die anderen Agenten.

Durch das Ausführen von Aktionen und die Kommunikation mit anderen Agenten können die Agenten im Laufe des Szenarios zu weiteren Informationen gelangen und so ihr Wissen erweitern. Hierauf wird in den späteren Kapiteln eingegangen. Anschliessend werden als Beispiel die erste und die letzte Szene näher betrachtet.

Im folgenden sind auszugsweise einige Regeln und Fakten aufgelistet, die zum initialen Wissen der einzelnen Agenten gehören. Es handelt sich nicht um eine vollständige Aufzählung des gesamten Wissens der einzelnen Agenten. Die drei Agenten Poirot, Hastings und Alfred stehen stellvertretend für alle im Szenario vorkommenden Agenten, da diese drei Agenten, die ersten im Szenario vorkommenden Agenten sind.

4.4.2.1 Gemeinsames Wissen

Hierbei handelt es sich um generische Regeln und Fakten, die allen beteiligten Agenten zur Verfügung stehen. Beispiele dafür sind die Fakten, dass Alfred einen Bart und eine Brille trägt, oder Lawrence von Beruf Arzt ist.

Die Regeln beschränken sich größtenteils auf Symmetrieregeln oder Hilfsregeln. Komplexere Regeln sind fast immer agentenspezifisch. Die nachfolgenden Regeln und Fakten beschreiben auszugsweise das gemeinsame Wissen der Agenten.

```
Relatives(A,B) :-
        Relatives(B,A).

CurrentLocation(Place) :-
        CurrentScene(Scene),
        EnvLocation(Place,Scene).

Appearance(alfred,beard).
```

```
Appearance(alfred,glasses).
Age(alfred,50).
BelongsTo(johnsRoom,john).
HasProfession(lawrence,doctor).
```

4.4.2.2 Poirot

Da Poirot der Hauptakteur des Szenarios ist, besitzt er eine größere Wissensbasis als die anderen Agenten. Dabei handelt es sich um reine Fakten, Regeln um Aktionen durchführen zu können, Wissen über Gift, Verdächtige und vieles mehr.

```
MadeOf(testament,thickPaper).
Poison(strychnine).

DoInvestigate(X) :-
        Evidence(X),
        EnvThing(X,Scene),
        CurrentScene(Scene),
         not Closed(X,Scene),
         not EnvInvestigated(X,Scene),
         not Closed(X,Scene).

Suspect(X) :-
        BurningTestament(X),
        Inheriting(X,V),
        Victim(V).

Suspect(X) :-
        Motive(X),
        Person(X).

Motive(X) :-
        FinancialDifficulties(X),
        Inheriting(V,X),
        Wealthy(V),
        Victim(V),
        Person(X).

TimeOfEffect(strychnine,2,V) :-
        not PoisonDelay(V),
        Victim(V).

TimeOfEffect(strychnine,10,V) :-
```

```
        PoisonDelay(V),
        Victim(V).
```

Poirot weiß, dass ein Testament aus dickerem Papier besteht (*MadeOf(testament, thickPaper)*) und Strychnin ein Gift ist (*Poison(strychnine)*), dass jemand verdächtig ist (*Suspect(X)*), wenn er ein Testament verbrennt und erbt, oder ein Motiv hat (*Motive(X)*). Ein Mordmotiv liegt beispielsweise vor, wenn man in finanziellen Schwierigkeiten steckt und von dem reichen Mordopfer erbt.

Des Weiteren weiß Poirot, dass Strychnine ein schnell wirkendes Gift ist und der Tod innerhalb von zwei Stunden eintritt (*TimeOfEffect(strychnine,2,V)*). Dieses kann sich aber verzögern, beispielsweise wenn das Opfer Schlafmittel genommen hat (*PoisonDelay(V)*). Dann tritt die Wirkung des Giftes erst nach zehn Stunden ein (*TimeOfEffect(strychnine,10,V)*).

4.4.2.3 Hastings

Hastings ist kein besonders geübter Detektiv, daher entgehen ihm viele Details und seine Wissensbasis ist zu Beginn nicht besonders groß. So weiß er nur, dass jemand verdächtig ist, wenn er sich am Todestag am Tatort befand, kann allerdings mit den am Tatort gefundenen Hinweisen nichts anfgangen.

```
Suspect(X) :-
        LocationDay(X,P,D),
        RoomOfCrime(P),
        TimestampDay(death,D),
        Person(X).
```

4.4.2.4 Alfred

Alfred versucht eindeutige Beweise an seiner Beteiligung am Mord zu vertuschen. Da er einen Brief hinterlassen hat, mit dem er seine Tat eingesteht, muß er diesen verschwinden lassen. Damit er dabei nicht erwischt wird, muß er allerdings sicherstellen, dass er zu dem Zeitpunkt alleine im Zimmer ist. Dann kann er den Gegenstand, den er verschwinden lassen muß, verstecken.

```
-HomeAlone(Name1) :-
        EnvCaracter(Name1,Scene),
        EnvCaracter(Name2,Scene),
        CurrentScene(Scene),
        !=(Name1,Name2).

DoHide(From,To,Item) :-
        HidingPlace(To,Item,Scene),
        CurrentScene(Scene),
```

```
        ToHide(Item, Scene),
        CurrentPlace(Place),
        RoomOfCrime(Place),
        EnvThingContained(From, Item, Scene),
        not -HomeAlone(alfred).
```

4.4.3 Aktionen

Im Laufe des Szenarios führen die Agenten verschiedene Aktionen aus. Dadurch können sie neue Informationen über ihre Umwelt oder andere Agenten sammeln. Hat sich ein Agent zur Ausführung einer Aktion entschieden, so sendet er einen Aktionsbefehl an die Umwelt, welche diesen umsetzt und dem Agenten eine entsprechende Antwort liefert. Eine Aktion kann beispielsweise das Öffnen eines geschlossenen Objektes, oder das Verstecken einen Gegenstandes sein. Nicht jeder Agent kann alle Aktionen ausführen, so kann beispielsweise Poirot Gegenstände näher untersuchen, aber keine Gegenstände verstecken. Diese Möglichkeit hat nur der Agent Alfred. Die einzelnen Agenten unterscheiden sich daher nicht nur in ihrem Wissen, sondern auch die Möglichkeit, Aktionen auszuführen. Es folgt ein Beispiel, über die Aktionen, die Poirot zu Beginn des Szenarios ausführen kann.

Beispiel Poirot Poirot hat zu Beginn des Szenarios die Möglichkeit, verschiedene Gegenstände zu untersuchen. Diese Aktion soll beispielsweise ausgeführt werden, wenn Poirot etwas als ein Indiz identifiziert.

```
DoInvestigate(X) :-
        Evidence(X),
        EnvThing(X, Scene),
        CurrentScene(Scene),
        not EnvInvestigated(X, Scene),
        not Closed(X, Scene).
```

Wenn es sich bei dem Gegenstand X um ein Indiz handelt (*Evidence(X)*), welches in der aktuellen Szene vorkommt (*EnvThing(X, Scene), CurrentScene(Scene)*) und weder verschlossen ist (*not Closed(X, Scene)*), noch bereits untersucht wurde (*not EnvInvestigated(X, Scene)*), wird das Fakt *DoInvestigate(X)* geschlossen. Dieses wiederum führt im KnowHow zur Ausführung der Aktion *Investigate*.

Die Aktion wird ausgeführt, indem eine Nachricht an die Umwelt gesendet wird. Diese sendet daraufhin eine Antwort, über Erfolg oder Misserfolg der Aktion.

```
<goal name="Investigate">
  <triggers>
  <trigger>
   <condition>DoInvestigate(X)</condition>
```

```
  </trigger>
 </triggers>
 <subelementsets>
  <subelementset>
   <doaction name="EnvRequest">
    <parameter name="addressee">
     <value>Environment</value>
    </parameter>
    <parameter name="content" constant="false">
     <for variable="X">DoInvestigate(X)</for>
     <value>EnvInvestigate(X)</value>
    </parameter>
   </doaction>
  </subelementset>
 </subelementsets>
</goal>
```

Auf die gleiche Weise verläuft das Öffnen eines Gegenstandes, beispielsweise einer Tür.

```
DoOpen(X) :-
        not EnvChangedBy(X, opened, Person, Scene),
        MyName(Person),
        Closed(X, Scene),
        not Opened(X, Scene),
        EnvThing(X, Scene),
        EnvLocation(Place, Scene),
        CurrentScene(Scene),
        not Briefcase(X, Scene).
```

Der Gegenstand X wird geöffnet, wenn er in dieser Szene nicht bereits von dem Agenten geöffnet wurde (*not EnvChangedBy(X, opened, Person, Scene), MyName(Person), CurrentScene(Scene)*), der Gegenstand verschlossen ist (*Closed(X, Scene)*) und sich in dieser Szene am selben Ort befindet, wie der Agent (*EnvLocation(Place, Scene)*). Dabei hat der Agent Poirot die Beschränkung, dass er keine Aktenkoffer öffnen darf (*not Briefcase(X, Scene)*), so lange kein Anwalt dabei ist. Wird *DoOpen(X)* geschlossen, so wird die Aktion *Open* ausgeführt. Dabei wird eine Nachricht an die Umwelt gesendet, die den ausgewählten Gegenstand öffnet.

```
<goal name="Open">
 <triggers>
  <trigger>
   <condition>DoOpen(X)</condition>
  </trigger>
 </triggers>
```

```
 <subelementsets>
  <subelementset>
   <doaction name="EnvRequest">
    <parameter name="addressee">
     <value>Environment</value>
    </parameter>
    <parameter name="content" constant="false">
     <for variable="X">DoOpen(X)</for>
     <value>EnvChange(X,open)</value>
    </parameter>
   </doaction>
  </subelementset>
 </subelementsets>
</goal>
```

Szenenbeispiel In der ersten Szene entdeckt Poirot einen Kamin im Zimmer des Mordopfers. Dieser Kamin kommt ihm verdächtig vor, daher will er ihn näher untersuchen. Er sendet eine Nachricht an die Umwelt mit dem Inhalt *EnvInvestigate(chimney)*. Dadurch erhält er die Information, dass im Kamin ein Stück Papier liegt, auf dem das Wort *MENT* steht.

4.4.4 Fragen

Die Kommunikation zwischen den Agenten beschränkt sich größtenteils auf die Fragen, die Poirot anderen beteiligten Agenten stellt. Diese werden dazu verwendet, das Wissen der Agenten über die Umwelt und über andere Agenten zu erweitern, um neue Schlussfolgerungen zu ermöglichen. Im folgenden wird anhand von zwei Beispielen gezeigt, wie Poirot Fragen stellt.

Nachdem Poirot in der ersten Szene das Zimmer von Emily untersucht hat, beginnt er mit seinen Nachforschungen. Dabei erscheint es ihm sinnvoll erst einmal alle Agenten, die eine Ahnung über die Todesursache haben könnten, zu fragen, was sie ihm über die Todesursache sagen können:

```
CauseOfDeathQueryPredicate(p_CauseOfDeath).

DoQueryInstantiate(Pred,Adressee) :-
        EnvCharacter(Adressee,Scene),
        CurrentScene(Scene),
        HasProfession(Adressee,doctor),
        not QueryDoneAnytime(p_instantiate,Pred,Adressee),
        CauseOfDeathQueryPredicate(Pred).
```

Poirot weiß, dass das Prädikat *CauseOfDeath(X)*, dass Prädikat ist, nach dem gefragt werden muß, wenn man die Todesursache wissen will (*CauseOfDeathQueryPredica-*

te(p_ CauseOfDeath)). Prädikate, nach denen gefragt werden soll, werden als Instanzen mit *p_* beginnend dargestellt. Er weiß auch, dass ein Arzt etwas über die Todesursache wissen könnte (*HasProfession(Adressee,doctor)*). Daher fragt er einen Agenten nach der Todesursache, wenn dieser ein Arzt ist und sich zum selben Zeitpunkt am selben Ort, wie Poirot befindet. Dieses ist der Fall, wenn beide in der gleichen Szene sind (*EnvCharacter(Adressee,Scene), CurrentScene(Scene)*). Außerdem wird sichergestellt, das Poirot dem gleichen Agenten dieselbe Frage nicht mehrmals stellt (*not QueryDoneAnytime(p_ instantiate,Pred,Adressee)*).

Als weitere Kommunikationsaktion befragt Poirot jeden, über die Personen, die nach Poirots Meinung mögliche Täter sind.

```
DoQueryElementInfo(Element, Adressee):-
        EnvCharacter(Adressee, Scene),
        CurrentScene(Scene),
        Person(Element),
        Suspect(Element),
        not QueryDoneAnytime(p_element_info, Element, Adressee),
        !=(Adressee, Element),
        !=(Adressee, Me),
        MyName(Me).
```

Auch hier wird wieder sichergestellt, dass Poirot und der befragte Agent sich in derselben Szene befinden und Poirot den Agenten nicht schon zum selben Thema befragt hat. Des Weiteren stellt Poirot sicher, dass er weder sich selbst befragt (*!=(Adressee,Me), MyName(Me)*), noch dass ein verdächtiger Agent über sich selbst befragt wird (*!=(Adressee,Element), Person(Element), Suspect(Element)*).

Beide Fragen unterscheiden sich in der Art der Frage. Bei der ersten Frage handelt es sich um eine *Instantiate*-Frage. Dabei wird der Agent aufgefordert, die Variable eines Prädikates zu ersetzen. So soll auf die Frage nach *CauseOfDeath(X)* die Variable X näher bestimmt werden. Bei der zweiten Frage handelt es sich um eine *ElementInfo*-Frage. Hier wird der Agent dazu aufgefordert, alles zu erzählen, was er über die Instanz weiß, also alle Prädikate zurückgeben, die das gefragte Element enthalten. Dadurch kann Mary auf die Frage *DoQueryElementInfo(Cynthia, Mary)* beispielsweise *HasProfession(cynthia, pharmacist), Workplace(cynthia, hospital)*, usw. antworten.

Durch die Antworten auf seine Fragen kann der Agent neue Schlussfolgerungen ziehen, aber nicht immer erhält er eine Antwort auf eine Frage. Es kann vorkommen, dass der gefragte Agent kein Wissen zu der gefragten Instanz hat, oder ihm ein Prädikat unbekannt ist. Es kann aber auch vorkommen, dass ein Agent bewusst nicht die Wahrheit sagen will. Darauf wird im folgenden Abschnitt eingegangen.

4.4.5 Lügengenerierung

Im Laufe des Szenarios kommt es vor, dass die Agenten lügen, oder die Wahrheit verschweigen. Dieses geschieht mittels einfacher Regeln.

4.4.5.1 Lügen

Das Verfahren der Lügengenerierung wird an einem Beispiel beschrieben:

Mary hat einen Streit zwischen ihrem Mann und dem Mordopfer gehört. Da sie sich sicher ist, dass jemand, der sich mit der Toten gestritten hat, verdächtigt wird (*Suspect(X)*), versucht sie ihren Mann zu schützen. Die beste Lösung, die ihr dabei einfällt, ist die Beschuldigung des Hauptverdächtigen.

```
Suspect(X) :-
        Disputing(X,Y,_),
        Person(Y),
        Victim(X).

LieAbout3(p_Disputing,X,Y,V) :-
        Victim(X),
        Married(Y,Z),
        MyName(Z),
        TimeOfDisputeWithEmily(V).

ArgumentSemantic(p_Disputing,1,p_Person).
ArgumentSemantic(p_Disputing,1,p_Victim).
ArgumentSemantic(p_Disputing,2,p_Person).
ArgumentSemantic(p_Disputing,2,p_MainSuspect).
ArgumentSemantic(p_Disputing,3,p_TimOfDisputeWithEmily).

MainSuspect(alfred).
```

Bei dem Prädikat, über das gelogen werden soll, handelt es sich um *Disputing(emily, john, 1039)*. Dieses bedeutet, dass das Mordopfer Emily und die Person John sich zum Zeitpunkt 1039 (der Zeitpunkt 1054 bezeichnet den Tod von Emily) gestritten haben. Über dieses Prädikat wird nur gelogen, wenn der Agent, der das Wissen über den Streit besitzt (Mary) mit dem am Streit beteiligten Agenten verheiratet ist (*Married(Y,Z), MyName(Z)*).

Damit der Agent keine unsinnigen Antworten gibt, so dass eine Lüge direkt durchschaubar ist, muß sicher gestellt werden, dass die Antworten im richtigen Kontext gegeben werden. Es wäre beispielsweise nicht sonderlich hilfreich, wenn Mary auf die Frage, wer sich mit Emily gestritten hat, *emily* oder *commode* zur Antwort gibt. Dieses wird durch die Prädikate *ArgumentSemantic(X, Y, Z)* sichergestellt. Dabei werden Prädikate, über die gelogen werden soll, als Instanzen mit *p_* beginnend dargestellt, d.h. die erste und die dritte Instanz von *ArgumentSemantic*, also X bzw. Z, stehen jeweils stellvertretend für ein Prädikat. Die zweite Instanz (Y) bezeichnet die Stelle des Prädikates X, die mit Hilfe des Prädikates Z genauer definiert wird.

Bei dem Prädikat *Disputing(X, Y, V)* muß es sich bei der ersten und zweiten Stelle auf jeden Fall um eine Person handeln. Bei der Stelle, über die gelogen werden muß,

muß sichergestellt werden, dass nicht aus Versehen die richtige Antwort gegeben wird. Daher muß die in Frage kommende Antwort durch das Prädikat *ArgumentSemantic(X, Y, Z)* näher bestimmt werden. In diesem Fall ist es sinnvoll, den Hauptverdächtigen als Streitteilnehmer zu beschuldigen (*ArgumentSemantic(p_Disputing,2,p_MainSuspect)*).

4.4.5.2 Verschweigen der Wahrheit

In manchen Fällen kann es vorkommen, dass ein Agent der Meinung ist, dass ihn die Antwort auf eine Frage in Schwierigkeiten bringt oder jemandem schadet, den er schützen möchte. Es kann vorkommen, dass es dem Agenten sinnvoller scheint, Unwissen vorzutäuschen, statt eine falsche Antwort zu geben. John ist beispielsweise im Gegensatz zu Mary der Meinung, dass er über seinen Streit mit Emily lieber schweigt, anstatt zu behaupten, jemand anderes hätte sich mit dem Mordopfer gestritten.

```
KeepSecret3(p_Disputing,X,Y,Z) :-
        Victim(X),
        MyName(Y),
        TimeOfDisputeWithEmily(Z).
```

Die hier aufgeführte Regel beschreibt das Verschweigen der Wahrheit, wenn sich der Agent mit dem Mordopfer gestrittet hat. Dabei antwortet der Agent auf die Frage nach *Disputing(X,Y,Z)*, dass er nichts dazu sagen kann.

4.4.6 Erste Szene: Tatortuntersuchung

Zu Beginn der ersten Szene hat Poirot viele grundlgegenden Fakten über den Mord schon mitgeteilt bekommen. Er weiß beispielsweise, wer die beteiligten Personen sind, was für Berufe sie ausüben, usw. Sein erster Schritt besteht darin, gemeinsam mit Hastings das Zimmer zu untersuchen, in dem das Opfer starb. Dabei entdecken sie neue Fakten, die ihrer Wissensbasis hinzugefügt werden.

Es gibt zwei Arten von Information, die die Agenten erlangen können. Einmal Informationen, die jeder Agent mitgeteilt bekommt, wenn er an einen Ort kommt. Andererseits gibt es Informationen, die nur durch genaueres Nachfragen bzw. Untersuchen entdeckt werden können.

4.4.6.1 Offensichtliche Informationen aus der Umwelt

Die Umwelt liefert verschiedene Informationen. Sie gibt an, in welcher Szene und an welchem Ort die Agenten sich gerade befinden. Des Weiteren gibt sie an, welche Gegenstände und Personen sich im Raum befinden. Dabei handelt es sich in dieser Szene z. B. um einen Kamin, einen Tisch und Vasen und die Personen Poirot und Hastings. Es folgt ein kleiner Auszug aus den Informationen, die die Umwelt liefert:

```
EnvScene(1).
```

```
EnvLocation(emilysRoom,1).

EnvThing(chimney,1).
Furniture(chimney,1).
EnvAttribOfThing(on, chimney,1).
On(chimney,1).

EnvThing(waxStain,1).
Wax(waxStain,1).

EnvThing(vases,1).
Container(vases,1).
EnvAttribOfThing(unordered, vases,1).
Unordered(vases,1).

EnvThing(commode,1).
Container(commode,1).
EnvAttribOfThing(closed, commode,1).
Closed(commode,1).

EnvCharacter(poirot,1).
EnvCharacter(hastings,1).
```

4.4.6.2 Untersuchungsergebnisse

Durch unterschiedliche Aktionen kann Poirot weiter Informationen erlangen. Er kann beispielsweise die Kommode öffnen und den Kamin untersuchen. Dabei findet er z. B. bei der Untersuchung des Kamins ein Stück Papier oder bei näherer Betrachtung der Türen einen Stofffetzen.

```
EnvThing(burntPaper,1).
Paper(burntPaper,1).
EnvAttribOfThing(burnt, burntPaper,1).
EnvAttribOfThing(ment, burntPaper,1).
EnvMadeOf(burntPaper, thickPaper,1).
EnvThing(emptyPharmacyBox,1).
Container(emptyPharmacyBox,1).
EnvThing(pieceOfCloth,1).
Clothes(pieceOfCloth,1).
EnvAttribOfThing(green,pieceOfCloth,1).
```

Andere Agenten, wie z. B. Hastings können diese Gegenstände nicht entdecken, da ihnen die Ermittlungsfähigkeiten eines Detektives fehlen.

4.4.7 Letzte Szene: Überführung des Mörders

Nachdem Poirot verschiedene Indizien gesammelt hat, einige Agenten entlastet hat und andere unter starkem Verdacht hat, fehlt ihm noch der letzte Beweis.

Zu Beginn der Szene erhalten die Agenten die offensichtlichen Informationen aus der Umgebung, welche sich nicht von den Informationen unterscheiden, die Poirot und Hastings in der ersten Szene erhalten haben.

Dabei fällt Poirot auf, dass die Vasen wieder unordentlich sind, obwohl er sie bereits in der ersten Szene ordentlich hingestellt hat.

```
Evidence(Item):-
        EnvAttribOfThing(Item,CurrentAttribute,Now),
        CurrentScene(Now),
        EnvStateTransition(_,OldAttribute,CurrentAttribute,_),
        EnvAttribOfThing(Item,OldAttribute,Before),
        <(Before,Now),
        EnvLocation(Place,Scene),
        RoomOfCrime(Place).

DoInvestigate(X) :-
        Evidence(X),
        EnvThing(X,Scene),
        CurrentScene(Scene),
        not EnvInvestigated(X,Scene),
        not Closed(X, Scene).
```

Poirot weiß, dass die Vasen in der ersten Szene ordentlich waren (*EnvAttribOfThing(Item,OldAttribute,Before)*) und jetzt in der letzten Szene wieder unordentlich sind (*(EnvAttribOfThing(Item,CurrentAttribute,Now)*). Es halt also eine *EnvStateTransition(_,OldAttribute,CurrentAttribute,_)* stattgefunden. Da es sich um den Tatort handelt (*EnvLocation(Place,Scene), RoomOfCrime(Place)*), findet Poirot diese Tatsache verdächtig und er sieht die Vasen als mögliches Beweisstück an (*Evidence(Item)*). Dadurch schlussfolgert er (*DoInvestigate(X)*), dass er sich die Vasen näher ansehen muß. Dabei findet er einen Brief von Alfred an Evie, mit dem die beiden den Mord eingestehen.

```
Contains(vases,confessionLetter).
```

Poirot hat einen Beweis gefunden, der seinen Verdacht bestätigt. Damit hat er sein Ziel erreicht und den Mörder, bzw. in diesem Fall die Mörder, gefunden. Mit der Schlussfolgerung *Murderer(evie)* und *Murderer(alfred)* wird das Szenario beendet.

5 Realisierung

5.1 Überblick

Adib Dado

KiMAS ist ein Multiagentensystem. Es unterstützt die Darstellung und Verarbeitung von Wissen. Dazu ist eine Multiagentenplattform und eine Inferenzmaschine notwendig. Beide Komponenten sind aus PG-Anforderungsgründen nicht selbst implementiert worden. Wir haben JADEX als Agentenframework ausgewählt. JADEX wurde in JAVA geschrieben und realisiert die zum Betrieb der Agenten notwendige Dienste. Als Inferenzmaschine wird DLV verwendet (siehe 4.4). Die beiden Komponenten sind in KiMAS integriert.

Jeder KiMAS-Agent verfügt über eine Wissensbasis und Ziele. Die Wissensbasis stellt das Wissen eines Agenten dar. Damit ein KiMAS-Agent seine Ziele erreichen und verwirklichen kann, muss er eine oder mehrere Aktionen durchzuführen. Darüberhinaus soll ein KiMAS-Agent die Fähigkeit haben Fragen stellen und Antworten generieren können. Wie ein KiMAS-Agent definiert und wie seine Architektur realisiert ist, wird im Abschnitt 5.2 beschrieben.

Die einzelnen KiMAS-Agenten und Szenen werden im XML-Format beschrieben. Für die Erfassung der Files wird einen Editor entwickelt, um die manuelle Erfassung zu erleichtern. Siehe Abschnitt 5.3 und 5.8.

Im KiMAS-System können Agenten Nachrichten mit anderen Agenten austauschen. Ein Agent kann zu einem neuen Wissen gelangen, wenn er neue Informationen von anderen Agenten erhält. Diese Kommunikation geschieht nicht direkt, sondern über den Umweltagent. Ein Umweltagent stellt den Rahmen, in dem der Agenten sich befinden dar. Darüberhinaus teilt der Umweltagent Informationen über die Objekte, die sich in der Umwelt befinden mit. Im Abschnitt 5.6 wird die Modellierung der Umwelt genau beschrieben. Im Kapitel 5.4 wird der Nachrichtentransfer zwischen den Agenten näher erläutert. Das neue Wissen, das ein Agent durch die Nachrichtenaustausch mit anderen Agenten gewonnen hat, wird nicht direkt in das vorhandenen Wissen aufgenommen. Das neue Wissen muss noch auf Konsistenz mit dem vorhandenen Wissen überprüft werden. Das Überprüfen und die Aufnahme geschehen mit Hilfe der Operatoren und des DLV-Systems.

KiMAS wurde mittels JUNIT getestet. Die genaue Testvorgänge und Testergebnisse werden im Abschnitt 5.7 erläutert. Für die KiMAS-Bedienung wird eine Grafische Benutzungsschnittstelle entwickelt, um das KiMAS leicht zu bedienen, Kommunikation und Wissensänderung der Agenten während des Nachrichtenaustauschs darzustellen. Für die Entwicklung wird JAVA-Swing verwendet.

In der Abbildung 5.1 wird der Zusammenhang zwischen den einzelnen Komponenten

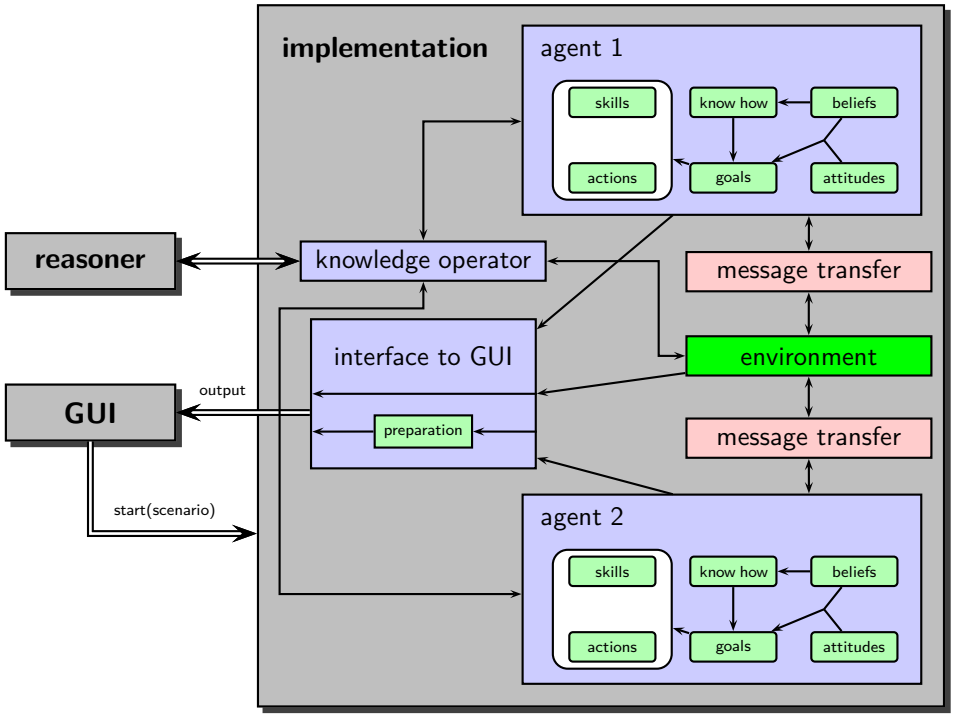

Abbildung 5.1: Architekturmodell

dargestellt. Die Kommunikation zwischen den Agenten geschieht, wie bereits erwähnt wurde, über den Umweltagenten „Environment“. Das neue Wissen wird mit Hilfe von Wissensoperatoren und der Inferenzmaschine von den Agenten aufgenommen. Die Graphische Benutzungsschnittstelle hat keinen direkten Zugriff auf die Fachkonzept-Klassen. Der Zugriff geschieht über eine Schnittstelle „Interface to GUI“.

In folgenden Abschnitten werden die verwendeten Architekturen und Techniken der KIMAS-Realisierung erläutert.

5.2 Agentenarchitektur

Igor Drobiazko, Patrick Krümpelmann, Matthias Thimm

Dieser Abschnitt beschäftigt sich mit der Laufzeit-Architektur eines KIMAS-Agenten. Die Initialisierung eines KIMAS-Agenten und die zur Interaktion mit anderen Agenten nötige Umwelt wird in den Abschnitten 5.3 bzw. 5.6 genauer beschrieben. Das der Architektur zugrundeliegende Konzept eines Agentenmodells wurde bereits in Abschnitt 2.7 beschrieben. Hier folgt zunächst eine kurze Konkretisierung der abstrakten Konzepte für ein Laufzeit-System und anschließend werden die einzelnen Komponenten genauer beschrieben.

Ein KIMAS-Agent ist grundsätzlich nach dem BDI-Prinzip aufgebaut. Er besitzt eine

(logikbasierte) Wissenbasis, die sein Wissen repräsentiert, und generiert aufgrund seiner Ziele eine Reihe von Intentionen, die die Erfüllung seiner Ziele verwirklichen sollen. Da die Projektgruppe aufgrund des begrenzten Zeitrahmens ein zielgerichtetes Planen von Aktionen zur Verwirklichung von Zielen nicht verfolgen kann, verfügt ein KIMAS-Agent über eine Know-How Struktur, wie sie in Abschnitt 2.8 beschrieben wurde, um in einfacher Weise Ziele und Aktionen miteinander zu verknüpfen. Aktionen führen Veränderungen in der Umwelt oder in anderen Agenten hervor und Aktionen anderer Agenten können das eigene Wissen verändern. Um Wissensoperationen wie Revision und Update konkret zu realisieren, besitzt ein KIMAS-Agent einen Wissensoperator, der als konkretes Objekt für den Vorgang der Wissensintegration zuständig ist. Um die Generierung von Antworten zu Anfragen anderer Agenten zu realisieren, besitzt ein KIMAS-Agent einen Antwortoperator, der entsprechend seiner Ziele und seines Wissens die Antworten erstellt. Zu dem Wissen eines Agenten gehören neben dem rein logischen Wissen über sich und seine Umwelt auch Metainformationen über andere Agenten. Da sowohl Wissensoperationen als auch Kommunikation in KIMAS eine zentrale Rolle spielen, ist es wichtig eine Verknüpfung dieser beiden Bereiche mithilfe von Vertrauenswürdigkeiten zu erzeugen. Jeder Agent in KIMAS kann einem anderen Agenten mit einem bestimmten Grad des Glaubens vertrauen. Diese Vertrauenswürdigkeiten sind wichtig, damit Informationen anderer Agenten korrekt in die eigene logikbasierte Wissensbasis integriert werden können.

Die zentralen Komponenten eines KIMAS-Agenten sind also:

1. eine logikbasierte Wissensbasis,
2. das Know-How, welches u.a. die Ziele des Agenten beeinhaltet,
3. eine Menge von Intentionen, d.h. die aktuell verfolgten Ziele und die geplanten Aktionen,
4. einen Antwortoperator zur Generierung von Antworten zu Anfragen,
5. einen Wissensoperator zur Integration neuer Informationen und
6. eine Menge von Vertrauenswürdigkeiten (*Reliabilities*).

In den nächsten Unterabschnitten werden die einzelnen Komponenten eines KIMAS-Agenten genauer beschrieben.

5.2.1 Logikbasiertes Wissen

Die zentrale Komponente eines KIMAS-Agenten ist die logikbasierte Wissensbasis. Hier wird das für ein Szenario relevante Wissen des Agenten repräsentiert, wobei zunächst zwischen initialem Wissen und innerhalb eines Szenarios gewonnenem Wissen unterschieden wird. Das initiale Wissen eines Agenten wird im CDF spezifiziert, während neues Wissen durch Interaktion mit der Umwelt und anderen Agenten von dem Agenten selber aufgenommen wird. Das Wissen wird dabei in einer der Antwortmengenprogrammierung

ähnlichen Weise logisch strukturiert durch Fakten und Regeln dargestellt. Um aus einer in dieser Art logisch repräsentierten Wissenbasis eine Antwortmenge zu berechnen wird DLV benutzt [EFLP00].

Das Package `edu.udo.cs.ie.kimas.logic` enthält alle Klassen für die Erzeugung der disjunktiven logischen Programme und hat folgende Struktur:

- Interfaces
  - `ILiteral`
- Klassen
  - `Atom`
  - `Neg`
  - `Not`
  - `Predicate`
  - `Program`
  - `Rule`

Im Nachfolgenden beschreiben wir die einzelnen Klassen:

ILiteral Dieses Interface modelliert ein atomares Element der aussagenlogischen oder prädikatenlogischen Umgebung. Es ist das Basisinterface für alle Literale und spezifiziert die Methoden, die jedes Literal implementieren muss. Zurzeit wird nur die Methode `getName()` spezifiziert, welche den Namen des Literals zurückgibt.

Atom Die einfachste Realisierung des Interfaces `ILiteral` für die Aussagenlogik. Ausser der Implementierung der spezifizierten Methode `getName()` hat diese Klasse keine weitere Logik.

Not Diese Implementierung des Interfaces `ILiteral` modelliert eine *negation-as-failure* eines Literal. Die Benutzung dieser Klasse ist ähnlich wie im Fall der Klasse `Neg`: Um ein Objekt, welches ein Literal darstellt, mittels *not* zu negieren, wird dieses Objekt von der Klasse `Not` umhüllt.

```
Predicate p = new Predicate("p");
Not not = new Not(p.instantiate("X"));
```

Neg Diese Implementierung des Interfaces `ILiteral` modelliert eine explizite Negation der Literale, wie sie von Gelfond and Lifschitz definiert wurde. Ein explizit negiertes Literal kann implizit negiert werden, wenn es einer *negation-as-failure* folgt. Umgekehrt ist es nicht möglich. Beispielsweise ist $\neg a, not \neg b, \neg p(x)$ eine gültige Syntax. Um eine explizite Negation eines Literals zu erzeugen, muss dieses Literal von der Klasse `Neg` umhüllt werden. Betrachte dazu ein Beispiel: Wir wollen das Prädikat $\neg p(x)$ durch ein Java-Objekt repräsentieren.

```
Predicate p = new Predicate("p");
Neg neg = new Neg(p.instantiate("X"));
```

Beachte, dass folgendes Beispiel gültig ist:

```
Predicate p = new Predicate("p");
Neg neg = new Neg(p.instantiate("X"));
Not not = new Not(neg);
```

Dagegen ist folgender Code nicht erlaubt:

```
Predicate p = new Predicate("p");
Not not = new Not(p.instantiate("X"));
Neg neg = new Neg(not);
```

Predicate Eine weitere Realisierung des Interfaces `ILiteral` für die Prädikatenlogik, welche eine Erweiterung der Klasse `Atom` ist. Die aussagenlogische Implementierung wird unter anderem um folgende Methoden erweitert:

- `getArity()`: Gibt die Stelligkeit des Prädikats zurück.
- `getInstanceName()`: Gibt den Instanznamen des Prädikats zurück. z.B. das Prädikat `Nurse` würde `p_nurse` zurückgeben. Solche Instanznamen werden in DLV benutzt, um die Schachtelung von Prädikaten zu vermeiden.
- `getValues()`: Gibt eine Liste der Werte des Prädikats zurück.
- *Eine Menge von Methoden zur Instantiierung der Prädikate:*Es existieren eine Reihe von Methoden um aus einem uninstantiierten Prädikat Instanzen abzuleiten. Wir betrachten nur eine von diesen Methoden und instantiieren ein zweistelliges Prädikat $friend(x, y)$ aus einem String.

  ```
  Predicate p = new Predicate("friend", 2);
  p.instantiate("Peter Paul");
  p.instantiate("Paul Mary");
  ```

  Diesem Beispiel nach ist Peter ein Freund von Paul und Paul ein Freund von Mary.

Rule Diese Klasse modelliert eine disjunktive Datalogregel. Grundsätzlich haben die Regeln die folgende Form:

$$h_1 \vee ... \vee h_n :- b_1, ..., b_m$$

Dabei repräsentieren $h_1 \vee ... \vee h_n$ den Kopf und $b_1, ..., b_m$ den Körper der Regel. Analog zu der logischen Repräsentation hat auch die Klasse `Rule` eine Liste von Literalen, die zum Kopf gehören, und eine Liste von Literalen, die zum Körper gehören. Wir betrachten als Beispiel die folgende Regel:

```
jail(X) :- murder(X).
```

Diese Regel kann erzeugt werden, indem zwei *Predicate* - Objekte dem Objekt der Klasse *Regel* hinzugefügt werden:

```
Predicate murder = new Predicate("murder");
Predicate jail = new Predicate("jail");

Rule rule = new Rule();
rule.addBody(murder.instantiate("X"));
rule.addHead(jail.instantiate("X"));
```

Program Diese Klasse modelliert ein disjunktes logisches Programm. Ein Programm besteht aus einer Anzahl der Fakten, Regeln und Constraints. Die Fakten können durch instantiierte Prädikate oder Regeln ohne Body repräsentiert werden. Die Constraints sind Regeln ohne Kopf. Wir betrachten als Beispiel ein Programm, das aus einem Fakt und einer Regel besteht:

```
jail(X) :- murder(X).
murder(mary).
```

Die Repräsentation von diesem Programm mittels Java-Objekten ist:

```
Predicate murder = new Predicate("murder");
Predicate jail = new Predicate("jail");

Rule rule = new Rule();
rule.addBody(murder.instantiate("X"));
rule.addHead(jail.instantiate("X"));

Program program = new Program(rule);
program.add(murder.instantiate("mary"));
```

Unter anderem enthält diese Klasse folgende Methoden:

- `add(ILiteral)`: Fügt dem Programm ein neues Literal hinzu.
- `add(Rule)`: Fügt dem Program eine neue Regel hinzu.
- `getFactsByName(String)`: Gibt eine Liste der Fakten mit einem bestimmten Namen zurück.
- `getMissingEvidences(ILiteral)`: Gibt eine Liste der fehlenden Evidenzen, die nötig sind, um ein Literal zu inferieren. Betrachten wir als Beispiel folgendes Programm:

```
r1:    r(X) v s(X) :- p(X), q(X).
r2:    r(X) v s(X) :- w(X), v(X).
       p(Tweety).
```

Um den Fakt $r(Tweety)$ zu inferieren, muss man die Fakten $p(Tweety)$ und $q(Tweety)$ wissen, um die Regel $r1$ zu benutzen. Unser Programm enthält bereits $p(Tweety)$, so dass wir nur noch $q(Tweety)$ brauchen. Ausserdem könnte man die Regel $r2$ benutzen, wenn man die Fakten $w(Tweety)$ und $v(Tweety)$ weiß. Für dieses Beispiel würde die Methode also die folgende Liste zurückgeben:

$$\{\{\mathrm{q(X)}\},\ \{\mathrm{w(X)},\ \mathrm{v(X)}\}\}$$

- `getModels()`: Gibt eine Liste der Modelle des Programms zurück. Intern macht diese Methode einen Aufruf von DLV.
- `merge(Program)`: Vereinigt zwei Programme auf einfachste Weise, indem eine Vereinigung gebildet wird. Diese Methode nimmt an, dass die beiden Programme konsistent sind.

Die JAVA-Struktur zur Kapselung von logischem Wissen ist die Klasse `ProgramSet`. Jeder KIMAS-Agent verfügt über ein Objekt dieser Klasse, welche sein logisches Wissen beeinhaltet. In ihm wird das initiale Wissen des Agenten in Form eines logischen Programms, sowie eine Menge von Dialogen, an denen im Laufe eines Szenarios der Agent teilgenommen hat, gespeichert. Durch Anwendung von Wissensoperatoren (siehe Abschnitt 2.4) kann aus diesen Informationen die aktuelle Wissensmenge des Agenten berechnet werden.

5.2.2 Know-How und Intentionen

Ein KIMAS-Agent kann Aktionen ausführen, um neues Wissen zu erlangen. Diese Aktionen können in Aktionsfolgen geordnet werden, um bestimmte Ziele des Agenten zu erreichen. Ein KIMAS-Agent verfügt zu jedem Zeitpunkt über eine Menge von Zielen, die es zu erreichen gilt. Um von einem Ziel auf eine Folge von Aktionen zu schliessen, ist Strukturwissen über die Art des Ziels nötig. Dieses Strukturwissen bietet das *Know-How* des Agenten. Jeder KIMAS-Agent verfügt über eine Instanz der Klasse `KnowHow`, welche eine Menge von Zielen und eine Menge von Aktionen enthält. Ziele können andere Ziele als Unterziele haben, die die Ausführung bestimmter Aktionen implizieren. Zur Interpretation des Know-Hows wird für jedes Ziel des Agenten eine Instanz des JADEX-Plans `MetaPlan` ausgeführt, welcher das Know-How des Agenten nach dem gerade verfolgten Ziel durchsucht. Wird das Ziel gefunden, so wird das erste darin genannte Unterziel als nächstes zu erreichende Ziel ausgewählt. Kann dieses Ziel direkt durch eine Aktion des Agenten erfüllt werden, so führt der Agent die entsprechende Aktion aus. Die möglichen Aktionen eines KIMAS-Agenten sind derzeit:

- Schicken einer Nachricht an einen anderen Agenten,
- Manipulieren oder Beobachten der Umwelt, und
- Schicken einer Nachricht an den Benutzer.

Mithilfe dieser atomaren Aktionen können durch das Know-How beliebig komplexe Ziele modelliert werden.

Die beiden Pakete `edu.udo.cs.ie.kimas.knowhow` und `edu.udo.cs.ie.kimas.plans` enthalten alle Klassen zur Darstellung von Know-How und seiner Interpretation. Die Klassen sind im einzelnen:

- `edu.udo.cs.ie.kimas.knowhow`
  - `Action`
  - `Element`
  - `ElementParameter`
  - `ElementParameterFor`
  - `Goal`
  - `KnowHow`
  - `SubElementSet`
  - `Trigger`
- `edu.udo.cs.ie.kimas.plans`
  - `InformUserPlan`
  - `MetaPlan`
  - `SendEnvRequestPlan`
  - `SendMessagePlan`

Es folgt nun eine Beschreibung der einzelnen Klassen.

`Action` Diese Klasse modelliert eine atomare Aktion als einen Spezialfall eines Know-How-`Element`. Neben den Eigenschaften von `Element` können Aktionen auch über Seiteneffekte in Form logischer Fakten verfügen, welche nach der Ausführung der Aktion dem Wissen des Agenten hinzugefügt werden.

`Element` Die Klasse `Element` ist die Superklasse der Klassen `Action` und `Goal` und fasst die gemeinsamen Eigenschaften dieser Elemente zusammen. Dazu gehört der Bezeichner des Elements, *trigger*, Parameter und Bedingungen.

`ElementParameter` Die Klasse `ElementParameter` modelliert einen Parameter eines Know-How-`Element`. Dies sind bestimmte Variablendeklarationen, die bei einer Instantiierung dieses Elements definiert werden müssen.

`ElementParameterFor` Die Parameter eines Know-How-`Element` können zur Laufzeit dynamisch aus dem logischen Wissen des Agenten generiert werden. Eine Instanz der Klasse `ElementParameter` kann dazu über Instanzen der Klasse `ElementParameterFor` verfügen, welche variable Prädikate enthält, aus denen die Parametervariablen gesetzt werden können. Beispielsweise würde die folgende Aktion InformUser für jede gültige Instanz des Prädikats Murderer einmal instantiiert werden:

```
<doaction name="InformUser">
        <parameter name="message" constant="false">
        <for variable="X">Murderer(X)</for>
        <value>I've determined the murderer: it was X.</value>
        </parameter>
</doaction>
```

`Goal` Diese Klasse modelliert ein Ziel als ein Spezialfall eines Know-How-`Element`. Ziele können verschiedene alternative Unterziele enthalten.

`KnowHow` Die Klasse `KnowHow` kapselt alle möglichen Ziele eines Agenten und beinhaltet eine Liste aller möglichen Aktionen.

`SubElementSet` Ein `SubElementSet` fasst eine Menge von Know-How-`Element` zu einer Sequenz zusammen, welche als ein alternativer Lösungsweg für die Erfüllung eines Zieles dienen kann.

`Trigger` Ein `Trigger` kapselt ein logisches Fakt und gibt somit Bedingungen für das direkte Auslösen eines Know-How-`Element` an.

`InformUserPlan` Der `InformUserPlan` wird von einem KIMAS-Agenten benutzt, wenn er dem Benutzer des Systems eine Nachricht zukommenlassen will.

`MetaPlan` Der `MetaPlan` dient der Interpretation von Know-How. Jedes Ziel wird von `MetaPlan` in Unterziele geteilt, deren Parameter er entsprechend den Vorgaben der zugehörigen Instanzen von `ElementParameterFor` instantiiert.

`SendEnvRequestPlan` Der `SendEnvRequestPlan` dient der Kommunikation mit der Umwelt, siehe dazu Abschnitt 5.6.

`SendMessagePlan` Mithilfe des `SendMessagePlan` können Agenten anderen Agenten des Systems Nachrichten schicken.

5.2.3 Antwortoperator

Agenten müssen in der Lage sein, auf Anfragen anderer Agenten passende Antworten zu geben. Die automatische Generierung von logischen Antworten nach Abschnitt 2.2 auf logische Anfragen leistet die Klasse `AnswerOperator`, die von einer Instanz des JADEX-Plans `ReceiveMessagePlan` für Anfragen anderen Agenten aufgerufen wird. Die Klasse `AnswerOperator` implementiert die in Abschnitt 2.5 vorgestellte Methodik zur Generierung plausibler Antworten.

5.3 Agenteninitialisierung

Stefan Tittel

Hinsichtlich des Startens und Initialisierens von Agenten reichen die Fähigkeiten des von KIMAS verwendeten Agentenframeworks JADEX in zwei Punkten nicht aus:

1. Es wird keine Möglichkeit geboten, mehrere Agenten automatisiert zu starten und dabei sicherzustellen, dass die gestarteten Agenten zum richtigen Zeitpunkt aktiv werden.

2. Es kann zwar Agentenfunktionalität in sogenannte *capabilities* ausgelagert werden, da JADEX jedoch von sich aus keine logische Modellierung unterstützt und eine *belief base* bei JADEX nur aus einer generischen Menge von JAVA-Objekten besteht, würde die Spezifikation jedes Agenten im JADEX-Spezifikationsformat unverhältnismäßig aufwändig.

KIMAS kapselt daher die JADEX-Mechanismen zum Starten und Spezifizieren von Agenten vollständig, so dass der Nutzer nur mit KIMAS-Komponenten in Berührung kommt.

Agenten und Szenarien werden durch *character definition files* (CDF) und *scenario definition files* (SDF) spezifiziert (Details werden in 6.3 beschrieben). Die relative Startordnung der Agenten wird dabei im SDF angegeben. Um nun das Szenario auszuführen, wird JADEX so instantiiert, dass nach der Instantiierung der sogenannte Starteragent ausgeführt wird. Dieser Agent hat folgende Funktionen:

1. Evaluation der absoluten Startordnung

2. Generierung von *constraint rules*

3. Starten der Agenten

4. Übertragung der notwendigen Informationen, damit sich die Agenten initialisieren können

5. Aktivierung der Agenten in der richtigen Reihenfolge, nachdem alle Agenten initialisiert sind

5.3.1 Startordnung, constraint rules und Starten der Agenten

Im SDF ist die Startordnung durch eine Menge von Vorher-Nachher-Tupeln gegeben, die eine Halbordnung beschreiben. Mittels topologischer Sortierung berechnet der Starteragent hieraus zunächst eine vollständige Startordnung. Ferner generiert er ein logisches Programm, welches in Form von *constraints* festlegt, welche Eigenschaften von Gegenständen der Umwelt sich ausschließen. Gibt es in der Umwelt (siehe 5.6) bspw. ein Radio mit drei Stationstasten, so würde dieses Programm *constraints* beinhalten, die besagen, dass immer nur eine Taste gleichzeitig gedrückt sein kann. Nachdem die *constraints* generiert wurden, werden die Agenten gestartet.

5.3.2 Initialisierung der Agenten

Grundsätzlich wird in KiMAS zwischen drei Agententypen unterschieden:

1. Starteragenten
2. Charakteragenten
3. Umweltagenten

Die Aufgaben des Starteragenten sind im vorhergehenden Abschnitt beschrieben. Unter einem Charakteragenten wird ein Agent verstanden, der eine handelnde Person (im Styles-Szenario, z. B. Hercule oder Alfred) repräsentiert. Nach dem Start sind Charakteragenten aus JADEX-Sicht bis auf den Namen völlig identisch, d. h. sie verfügen über die gleichen *beliefs*, *goals* and *plans*. Erst nach dem Start erhält ein Charakteragent vom Starteragenten die folgenden Informationen:

- Ort des CDF, das die agentenspezifische Konfiguration enthält
- Ort des SDF
- den `AgentIdentifier`, der die Umwelt identifiziert
- das logische Programm, welches die *constraints* enthält

Der Charakteragent instantiiert nun Fabrikklassen unter Angabe des Ortes von CDF und SDF. Die für das SDF zuständige Fabrikklasse stellt nun den Teil der Agentenspezifikation bereit, den alle Charakteragenten des Szenarios gemein haben, während die für das CDF zuständige Fabrik gleiches für den individuellen Teil der Spezifikation des Charakteragenten tut. Die von den Fabriken gelieferten JAVA-Objekte halten sich dabei recht eng an das Format von CDF bzw. SDF. Deshalb muss der Initialisierungsplan des Charakteragenten zunächst die erhaltenen Objekte umwandeln, um JADEX-kompatible *beliefs*, *goals* und *plans* zu erhalten. Danach werden diese initialisiert.

Ferner wird der Umweltagent gestartet. Dieser hat die Aufgabe, die Umwelt zu repräsentieren und den Szenarioablauf zu verwalten (siehe 5.6). Der Umweltagent bekommt nach dem Start vom Starteragenten mitgeteilt, welche Charakteragenten es gibt, deren

Startordnung sowie den Ort des SDF. Bevor das Szenario nun beginnt, teilt der Umweltagent allen an der ersten Szene[1] beteiligten Agenten Informationen über die dort herrschende Umwelt mit. Nachdem nun alle Charakteragenten und der Umweltagent initialisiert sind, stellt der Umweltagent über entsprechende Steuernachrichten sicher, dass die Charakteragenten in der vorgesehenen Startreihenfolge aktiviert werden. Das Szenario ist nun gestartet.

5.4 Nachrichtentransfer

Adib Dado

Die Kommunikation zwischen den Agenten spielt in KIMAS eine zentrale Rolle. Die wichtigsten Aktionen der KIMAS-Agenten sind die Kommunikationsaktionen. Das bedeutet, dass der Agent die Fähigkeit hat, Fragen stellen, Antworten generieren und allgemeine Mitteilungen verarbeiten. Der Kommunikationsdienst wird von JADEX bereitgestellt. Da JADEX ein eventbasiertes System ist, wird der Nachrichtentransfer zwischen den Agenten durch den Message Event realisiert.Wie die Fragen und Antworten strukturiert sind, werden im Abschnitt 2.2 näher erläutert.

Der Umweltagent spielt eine wichtige Rolle für den Nachrichtenaustausch. Ohne den Umweltagent wird keine Kommunikation stattfinden. Wenn ein Agent mit einem anderen Agent kommunizieren möchte, schickt er die Nachricht zuerst dem Umweltagent. Dann leitet der Umweltagent die Nachricht den eigentlichen Empfänger weiter. Das Konzept des Umweltagenten ist notwendig, weil Interaktionen zwischen Charakteragenten und die Umwelt stattfindet und durch den Umweltagent wird das Mithören realisiert.

Eine Nachricht ist ein Objekt der Klasse `KimasMessage` und besteht aus 3 Komponenten:

- Inhalt einer Nachricht(logisches Program)
- Absender (String)
- Liste der Empfänger (String)

Eine Nachricht kann gleichzeitig an mehreren Agenten gesendet werden, daher ist der Empfänger als eine Liste in der Klasse `KimasMessage` implementiert worden. Die Liste enthält die Empfängernamen. Die Nachricht wird zunächst vom Absender-Agent dem Umweltagent geschickt. Der Umweltagent erhält die Nachricht und extrahiert die Empfänger aus der Nachricht (Objekt von `KimasMessage`) und leitet die Nachricht dem tatsächlichen Empfänger weiter.

Als Nächstes wird erläutert, wie das Senden, Weiterleiten und Empfangen von Nachrichten, geschieht. Am Nachrichtentransfern beteiligen sich verschiedene Klassen. Wir werden die folgenden Klassen näher betrachten:

- `SendMessagePlan`

[1] zu Szenen siehe 5.6.2

- `HandleMessagePlan`
- `ReceiveMessagePlan`

5.4.1 SendMessagePlan

`SendMessagePlan` ist ein Plan von Charakteragenten und wird von `jadex.runtime.Plan` abgeleitet und ist für das Senden von Mitteilungen und Anfragen zuständig. Im Kapitel 2.2 werden die Nachrichtenstruktur und Nachrichtentypen näher erläutert . Der *Goal*-Event wird durch die Methode `getInitialEvent()` ermittelt und der Typ der Mitteilung wird bestimmt.

Im `SendMessagePlan` werden die Mitteilungen vom Typ `PQuery` und `PInform` behandelt. `PQuery` wird aus der Klasse *Program* abgeleitet und ist für die Fragen-Generierung zuständig. `PInform` ist eine allgemeine Mitteilung und wird ebenfalls aus der Klasse *Program* abgeleitet. Es wird anhand der Mitteilung eine Message erstellt, die als Fakt mit Hilfe der Beliefoperatoren in die *Beliefbase* des Agenten aufgenommen.

```
IGoalEvent trigger = (IGoalEvent) getInitialEvent();
message = new PQuery(type, getAgentName());
message = new PInform(getAgentName(),new TimeGen().timeAction())
    ;
KimasMessage kimasmessage = new KimasMessage(getAgentName(),
 addressee, message);
IMessageEvent sendMe = createMessageEvent("general_request");
sendMe.getParameterSet(jadex.adapter.fipa.SFipa.RECEIVERS).
addValue(environment);
sendMe.setContent(kimasmessage);
```

Ein Objekt der Klasse `KimasMessage` „kimasmessage“ wird erstellt und in ein Objekt der Klasse `IMessageEvent` „sendMe“ eingefügt und an den Umweltagenten geschickt. Der Parameter `jadex.adapter.fipa.SFipa.RECEIVERS` ist eine Klasse vom Jadexplatteform und wird benötigt, um eine Nachricht an den Empfäger zu senden. Die Mitteilungen vom Typ `answer-messages` werden in `ReceiveMessagePlan` behandelt.

5.4.2 HandleMessagePlan

Beim `HandleMessagePlan` handelt es sich um einen Plan des Umweltagenten. Der Event wird durch die Methode `getInitialEvent()` ermittelt. Der Inhalt des Events wird mit Hilfe des *Casting-Operators* in die `KimasMessage` gewandelt.

```
IMessageEvent me = (IMessageEvent)getInitialEvent();
KimasMessage message = (KimasMessage)me.getContent();
```

Der *Beliefbase* vom Umweltagent enthält eine JAVA-Klasse *Map* namens „agentNamesToIdentifiers“ , in der die Identität des Agenten als AgentIdentifier gespeichert wird. Wenn der Umweltagent eine Nachricht erhält, wird eine Liste aus AgentIdentifier für die Empfänger der Nachricht aus der Map erstellt, ein neues Objekt der Klasse `MessageEvent`

erzeugt, der Inhalt der Nachricht im Event gesetzt und der `IMeassageEvent` an die Empfängerliste geschickt.

```
Map agentMap=(Map)getBeliefbase().getBelief("
    agentNamesToIdentifiers").
getFact();
currentReceivers=(AgentIdentifier)agentMap.get(message.
getReceivers());
IMessageEvent mevent = createMessageEvent("general_request");
mevent.getParameterSet(jadex.adapter.fipa.SFipa.RECEIVERS).
addValue();
mevent.setContent(message);
sendMessage(mevent);
```

5.4.3 ReceiveMessagePlan

`ReceiveMessagePlan` ist ein Plan von Charakteragenten und wird von der Klasse `jadex.runtime.Plan` abgeleitet. Dieser Plan ist zuständig für die Verarbeitung der ankommenden Mitteilungen und Beantworten von Fragen.

Der `IMessageEvent` wird durch die Methode `getInitialEvent()` ermittelt. Aus dem Event wird die `KimasMessage` durch die Methode `getContent()` extrahiert.

```
IMessageEvent messageevent = (IMessageEvent)getInitialEvent();
KimasMessage message = (KimasMessage)messageevent.getContent();
```

Der Mitteilungstyp aus der Message wird überprüft. Folgende Typen können vorkommen:

- PQuery
- PAnswer
- PInform
- Logisches Program vom Umweltagent

Ist eine Mitteilung eine Anfrage (`PQuery`), so wird eine Antwort mit Hilfe des *AnswerOperators* erstellt. Die Antwort wird in einem Objekt der `KimasMessage` gekapselt. Das Objekt wird mit Hilfe des `IMessageEvents` an den Umweltagent geschickt. Wie die Logikbasierte Fragen und Antworten gebaut sind, werden im Abschnitt2.2 genau erläutert.

Alle Mitteilungen werden mit Hilfe der BeliefOperatoren in die *Beliefbase* des Agenten eingefügt. Neue Ziele können durch das Einfügen von neuen Informationen in die *Beliefbase* generiert werden.

5.5 Inferenz

Patrick Krümpelmann

5.5.1 Abstrahierung von Operatoren

Wie schon zuvor erwähnt, sollen verschiedenste Operationen auf den Wissensbasen der Agenten realisiert werden. Um eine möglichst hohe Modularität in der Verwendung von Wissensoperatoren zu gewährleisten, erben alle Wissensoperatoren von der abstrakten Klasse `BeliefOperator`, welche derzeit nur eine Implementierung der Methode `getModel` für die erbenden Klassen vorschreibt. Diese Methode generiert anhand des initialen Wissens und der protokollierten Informationen im Szenarioverlauf mit Hilfe bestimmter Wissensoperationen den aktuellen Wissenszustand.
Die Verwendung von Wissensoperatoren in den Agentenplänen gestaltet sich durch die Verwendung der abstrakten Klasse `BeliefOperator` recht einfach. Für jeden Agenten kann in der CDF ein Wissensoperator angegeben werden. Ein entsprechender Aufruf im CDF für den *Update-Program-Operator* würde dementsprechend folgendermaßen aussehen:

```
<beliefoperator name="UpdateProgramOperator">
  new edu.udo.cs.ie.kimas.logic.operators.UpdateProgramOperator
      ()
</beliefoperator>
```

Der im CDF genannte Operator kann dann in den Agentenplänen benutzt werden:

```
BeliefOperator myBeliefOperator =
  (BeliefOperator)getBeliefbase()
  .getBelief("beliefoperator").getFact();
Program model = myBeliefOperator.getModel(programSet);
```

5.5.2 Die Implementierung des *SimpleOperator*

Es folgt eine gekürzte Fassung des Quellcodes des *SimpleOperator*

```
package edu.udo.cs.ie.kimas.logic.operators;

/*
 import statements
 */

public class SimpleOperator extends BeliefOperator{

  /*
   auxiliary methods
   */

 public Program getModel(ProgramSet programSet){
   Program logic =
     generateStandardPredicateRules(
```

```
        programSet.getPredicates())
          .merge(programSet.getMergedProgram() );          (*1)

    logic.merge(standardAnalysis(
      programSet.getPredicates()));                        (*2)

    return (Program) logic.getModels().get(0);             (*3)
  }
}
```

Der *SimpleOperator* besteht im wesentlichen aus der Methode `Program getModel()`: Diese Methode berechnet aus der Wissensbasis des Agenten und den protokollierten Dialogen den aktuellen Wissenszustand.

Es wird nun die Funktionsweise der einzelnen Methoden schematisch anhand der im Quellcode angegebenen Zeilennummern dargestellt:

- Für die Methode Program getModel:

  (*1) Mithilfe der Methode `generateStandardPredicateRules()` werden zunächst für alle dem Agenten bekannten Prädikate, Regeln der Form

  Pred(X1,..,XN) :- HoldN(p_pred,X1,..,XN)}

  erzeugt, welche für die Dialoganalyse benötigt werden (siehe dazu auch Abschnitt 3.1.2). Diese Menge von Regeln wird dann vereinigt mit der Wissensbasis und der Menge der protokollierten Dialoge (welche sich beide im Objekt programSet befinden).

  (*2) Als nächstes wird diesem logischen Programm eine Menge von Regeln zur Analyse der Dialoge hinzugefügt, die daraus eine Menge von Hold-Fakten erzeugen.

  (*3) Schließlich wird zu diesem gesamten logischen Programm mithilfe von DLV ein Modell berechnet, welches den aktuellen Wissenszustand des Agenten darstellt.

5.5.3 Die Implementierung des *Update-Operator*

«««< .mine Die Methode zum Erzeugen der Antwortmengen zum aktuellen Wissenszustand ist die durch das Operator Interface vorgegebene `getModel(ProgramSet programSet, Map reliabilities)`. ======= Die Methode zum Erzeugen der Antwortmengen zum aktuellen Wissenszustand ist die durch das Operator Interface vorgegebene `getModel(ProgramSet programSet, Map reliabilities)`. »»»> .r1498 In der Klasse `programSet` werden die Dialoge des Agenten gespeichert, das heißt die ausgetauschten logischen Programme mit jeweiligem Sender und der Zeit des Austausches. In der `HashMap reliabilities` sind die dem aktuellen Agenten bekannten Agenten mit deren Glaubwürdigkeit gespeichert. Mit diesen beiden Übergabewerten sind alle für die Konstruktion der Update-Programme benötigten Informationen gegeben.

Aus den beiden Übergabewerten wir eine Liste von `ProgramPlus` erstellt, welche die in Kapitel 3.1.6 vorgestellte Menge $\mathbf{P} = (\langle P_1, q_1, t_1\rangle, ..., \langle P_n, q_n, t_n\rangle)$ repräsentiert. Diese wird durch den `Comparator` der `ProgramPlus` in die richtige Ordnung gebracht. Auf die so geordnete Liste wird nun der Algorithmus zur Erstellung des Programms angewandt. Das generierte Programm wird anschließend zur Modellbildung an DLV übergeben. Die zurückgelieferten Modelle werden nun auf das ursprüngliche Alphabet reduziert und an die aufrufende Klasse zurückgegeben.

5.5.4 Die Implementierung der *Update-Program-Operators*

Der *Update-Program-Operator* wurde entsprechend der Beschreibung aus Kapitel 3.1.3 implementiert. Hierfür werden zunächst die Dialoge des Agenten zu einer Liste von Programmen zusammengestellt. Diese werden in Reihenfolge der Glaubwürdigkeiten ihrer Quellen geordnet. Dies geschieht im Prinzip so wie bei dem *Update-Operator*. Gleiches gilt für das Aufrufen von DLV zur Modellfindung des generierten UpdateProgramms und für die anschließende Filterung. Optional wird noch der Minimalitätsfilter angewendet.

5.5.5 Die Implementierung des strikten Minimalitätsfilters für Update Programme

Zur Herstellung der Minimalität haben wir den in Kapitel 3.1.5 vorgestellten Algorithmus implementiert. Dieser berechnet zunächst wie oben beschrieben alle Update-Antwortmengen und testet anschließend jede einzelne auf strikte Minimalität und gibt nur die reduzierten strikt minimalen Antwortmengen zurück.

5.6 Modellierung der Umwelt

Stefan Tittel

Durch die von Jadex bereitgestellten Mechanismen sind Agenten in der Lage, untereinander zu kommunizieren. Dies ist jedoch zur Modellierung des Szenariohintergrundes nicht ausreichend. Wie in 5.4 erläutert, ist es notwendig, Nachrichten nicht direkt zwischen Agenten zu verschicken, sondern dies indirekt über den Umweltagenten zu tun. Der Umweltagent ist dabei ein Agent, der im Gegensatz zu den Charakteragenten nicht aktiv am Geschehen teilnimmt, sondern die Umgebung modelliert, in der die Agenten sich bewegen (im Falle des „The Mysterious Affair at Styles“-Szenarios also die Styles-Villa). Konkret erfüllt er die folgenden Aufgaben:

- Nachrichtentransfer zwischen Agenten
- Verwaltung von Szenen
- Modellierung von Orten und den dort vorzufindenden Gegenständen
- Interaktion von Agenten mit der Umwelt

- Untersuchen von Dingen
- Ändern von Attributen
- Verschieben von Dingen

Dieser Abschnitt behandelt die Beschreibung des Umweltagenten auf abstrakter Ebene. Die Spezifikation eigener Umweltdefinitionsdateien wird in 6.3 behandelt.

5.6.1 Nachrichtentransfer

Konzeptionell gesehen modelliert der Umweltagent die Luft, die zur Schallübertragung und somit Kommunikation zwischen Agenten notwendig ist. Möchte ein Agent mit einem anderen Agenten kommunizieren, schickt er die Nachricht daher zunächst an den Umweltagenten, welcher die Nachricht dann an den tatsächlichen Empfänger weiterleitet. Von konzeptionellen Aspekten abgesehen, ist diese vermeintlich etwas umständliche Realisierung (im Vergleich dazu, Agenten einfach direkt miteinander kommunizieren zu lassen) aus pragmatischen Gründen sehr sinnvoll. Näheres dazu erläutert 5.4.

5.6.2 Szenenverwaltung

Personen im realen Leben unterteilen dieses überlicherweise nicht in artifiziell gewählte Blöcke und auch die durch Agenten umgesetzten Charaktere des Szenariohintergrundes wissen natürlich nicht, welche Szenen Agatha Christie oder diese Projektgruppe für sie vorgesehen haben. Dennoch unterteilt sich ein Szenario bei KIMAS in *Szenen*. Eine Szene ist dabei ein Handlungsabschnitt, der an einem festen Ort mit einer festen Menge von teilnehmenden Agenten stattfindet und für den definiert ist, welche Agenten welche Ziele an dessen Ende zu erreichen haben. Auch wenn aus konzeptioneller Sicht ein freifließendes Szenario, bei dem die Agenten selbst ihren Aufenthaltsort mitbestimmen können, sicherlich seine Vorzüge hätte, so erscheint die Einführung von Szenen dennoch sinnvoll:

Ein Hauptproblem bei der Modellierung eines größeren Szenarios besteht darin, die immense Komplexität der logischen Modellierung, Agentenspezifikationen und Umweltspezifikation zu überblicken. Würde auf Szenen verzichtet, so gäbe es nur zwei Möglichkeiten für den Entwickler eines Szenarios festzustellen, ob seine Szenariospezifikation sich auch tatsächlich so verhält, wie von ihm vorgesehen:

1. manuelle Betrachtung der Agenteninnenleben zu beliebigen Zeitpunkten mittels der GUI
2. Überprüfung, ob alle Agenten ihre zu erreichenden Ziele am Ende des Szenarios erreichen

In Anbetracht des möglichen Umfangs eines Szenarios und in Anbetracht sehr umfangreicher *belief states* erschien dies als nicht ausreichend, um die sinnvolle Kontrolle bei der Entwicklung größerer Szenarien zur gewährleisten. Ferner wird durch den Verzicht auf dynamische Ortswechsel die Komplexität der Implementierung erheblich eingeschränkt, ohne dass dies zu Problemen führen würde, den Szenariohintergrund umzusetzen.

KIMAS implementiert Szenen derart, dass für jedes *goal* eines Agenten angegeben werden kann, dass dieses am Ende einer Szene zu erreichen ist. Eine Szene terminiert nur dann, wenn alle Agenten alle *goals* erreicht haben, die sie in dieser Szene erreichen sollten. Statt nun wie beim Verzicht auf Szenen nur am Szenarioende feststellen zu können, ob ein Agent sich wie vorgesehen verhält, ist dies somit granular am Ende jeder Szene möglich. Insbesondere kann so gewährleistet werden, dass zu Beginn der nächsten Szene die Agenten sich in einem (zumindest größtenteils) definierten Zustand befinden, also z. B. in Szene n das notwendige Wissen erlangt haben, um ihre Ziele in Szene $n + 1$ zu erreichen.

Jede Szene verfügt über einen Namen, eine Ordnungsnummer (deren relative Ordnung festlegt, wann die Szene ausgeführt wird), einen Ort und eine Menge schlafender Agenten. Alle Agenten des Szenarios, die für eine Szene nicht als schlafend markiert sind, nehmen an ihr teil. Zu Beginn einer Szene teilt der Umweltagent allen an der Szene teilnehmenden Agenten die offensichtlichen Umweltinformationen mit (also all das, was die Agenten beim Betreten eines Ortes ohne nähere Untersuchung sofort sehen können). Die in einer Szene von ihm zu erreichenden Ziele kennt der Agent selbst. Erreicht ein Agent alle seine Szenenziele, teilt er dies dem Umweltagenten in Form einer entsprechenden Steuernachricht mit und nimmt einen reaktiven Zustand an (d. h. er antwortet nur noch auf Anfragen anderer Agenten, stellt aber keine eigenen Anfragen mehr). Hat der Umweltagent von allen an der Szene teilnehmenden Agenten eine solche Steuernachricht erhalten, wird die nächste Szene durch das Versenden von Umweltinformationen an alle an der neuen Szene beteiligten Agenten gestartet.

5.6.3 Modellierung von Orten und Dingen

Wie im vorhergehenden Abschnitt erläutert, gehört zu jeder Szene genau ein *Ort*. Ein Ort in KIMAS ist dabei eine Menge von Dingen (*things*). Jedes Ding kann beliebig viele Attribute besitzen (z. B. „eingeschaltet“, „Radiostation 5“ im Falle eines Radios). Des Weiteren kann jedes Ding über eine Funktion der Form *Aktion* $\times$ *alter Zustand* $\rightarrow$ *neuer Zustand* verfügen. Darüberhinaus kann ein Ding andere Dinge enthalten (z. B. ein Glas, das auf dem Tisch steht). Damit den Agenten Interaktion mit Dingen möglich ist, kann ferner spezifiziert werden, welche Dinge ein Ding potenziell enthalten kann (also *addable* sind) und welche potenziell enthaltenen Dinge wieder entfernt werden können (also *removable* sind).

Dinge und Attribute sind entweder offensichtlich (*obvious*) oder nicht offensichtlich. Ist ein Ding offensichtlich, so bekommen alle an einer Szene beteiligten Agenten beim Szenenstart mitgeteilt, dass dieses Ding am Ort der Szene existiert. Ist ein Ding nicht offensichtlich, so geschieht dies nicht und das Ding kann nur dann gefunden werden, wenn es in einem anderen Ding enthalten ist und ein Agent dieses andere Ding untersucht. Beispiele für nicht offensichtliche Dinge sind ein Ding „kleiner, unscheinbarer Fleck“, welches der Teppich enthält, oder die explizite Modellierung des offensichtlichen Dings „Raum“, welches dann z. B. ein nicht offensichtliches Ding enthalten könnte, welches ein Agent bei der Untersuchung des Raumes finden soll. Auch Attribute eines Dings können nicht offensichtlich sein. Sie werden in diesem Falle nur dann gefunden, wenn ein Agent das

Prädikat	Beschreibung
EnvScene(S)	Nummer der aktuellen Szene
EnvLocation(*Ort*, S)	Name des Ortes in Szene S
EnvThing(*Ding*, S)	Ding in Szene S
Material(*Ding*, S)	Ding in Szene S besteht aus *Material*
EnvMadeOf(*Ding*, *Material*)	Ding besteht aus *Material*
EnvAttribOfThing(*Zustand*, *Ding*, S)	Ding hat Attribut mit Zustand *Zustand* in Szene S
Zustand(*Ding*, S)	Ding hat Attribut mit Zustand *Zustand* in Szene S
EnvStateTransition(*Aktion*, *Zustand1*, *Zustand2*, S)	Tripel der Attributszustandsübergangsfunktion
EnvThingContained(*Ding1*, *Ding2*, S)	*Ding1* beinhaltet *Ding2* in Szene S
EnvThingAddable(*Ding1*, *Ding2*, S)	*Ding2* ist *Ding1* in Szene S hinzufügbar
EnvThingRemovable(*Ding1*, *Ding2*, S)	*Ding2* ist von *Ding1* in Szene S entfernbar
EnvCharacter(*Agent*, S)	*Agent* ist in Szene S anwesend
EnvInvestigate(*Ding*)	untersuche *Ding*
EnvInvestigated(*Ding*, S)	*Ding* wurde in Szene S vom Empfänger untersucht
EnvChange(*Ding*, *Aktion*)	führe *Aktion* auf *Ding* aus
EnvChangedBy(*Ding*, *Zustand*, *Agent*, S)	*Ding* wurde in Szene S von *Agent* in Zustand *Zustand* überführt
EnvMove(*Ding1*, *Ding2*, *Ding3*)	verschiebe *Ding1* von *Ding2* nach *Ding3*
EnvMovedBy(*Ding1*, *Ding2*, *Agent*, S)	*Ding1* wurde von *Agent* nach *Ding2* verschoben

Tabelle 5.1: Umweltprädikate und ihre Bedeutung

betreffende Ding näher untersucht. Anderenfalls werden sie den Agenten vom Umweltagenten zum Start der Szene (im Falle eines offensichtlichen Dings) oder nach Entdecken des Dinges (im Falle eines nicht offensichtlichen Dings) mitgeteilt.

5.6.4 Interaktion

Agenten können mit der Umwelt interagieren. Dies geschieht, indem Agenten logische Programme mit entsprechenden Prädikaten an die Umwelt senden (siehe Tabelle 5.1). Resultiert eine Interaktion in einer Änderung, bekommt der die Interaktion auslösende Agent den neuen Zustand des geänderten Dings bzw. Attributes mitgeteilt. Alle anderen Agenten erhalten diesen neuen Zustand genau dann, wenn das geänderte Ding bzw. Attribut offensichtlich ist, ansonsten nicht. Es gibt drei Interaktionsmöglichkeiten.

5.6.4.1 Untersuchung

Wie in 5.6.3 dargestellt, kann es Dinge oder Attribute geben, die nicht offensichtlich sind. Um nun die nicht offensichtlichen Dinge zu entdecken und nicht offensichtliche Attribute zu erfahren, ist es notwendig, dass ein Agent Dinge untersuchen (*investigate*) kann. Wird ein Ding von einem Agenten untersucht, so erhält dieser vom Umweltagenten alle nicht offensichtlichen Attribute und alle in diesem Ding enthaltenen nicht offensichtlichen Dinge. Eventuell nicht offensichtliche Attribute der nicht offensichtlichen Dinge hingegen werden erst geliefert, wenn auch das entsprechende nicht offensichtliche Ding untersucht wurde.

5.6.4.2 Ändern von Attribute

Ist ein Attribut dynamisch (d. h. es kann grundsätzlich geändert werden) und teilt ein Agent der Umwelt mit, dass er eine bestimmte Aktion für das zugehörige Ding ausführen möchte und ist das Tupel aus Aktion und aktuellem Attributszustand Element des Definitionsbereiches der Attributszustandsübergangsfunktion, so wird der Funktionswert zum neuen Attributszustand. Attributsänderungen bleiben szenenübergreifend persistent.

5.6.4.3 Verschieben von Dingen

Ist ein Ding in einem anderen Ding enthalten und dort entfernbar (*removable*), kann der Agent durch Senden einer entsprechenden Verschiebeanweisung das enthaltene Ding in ein anderes Ding verschieben, sofern dieses dort hinzufügbar (*addable*) ist. Änderungen an Enthaltenseinsbeziehungen bleiben szenenübergreifend persistent.

5.7 Testen

Igor Drobiazko

5.7.1 Was ist unit testing?

Ein *unit test* ist eine Sequenz eines Programmcodes, die einen kleinen, spezifischen Teil eines Programmablaufs durchläuft und durchtestet. Üblicherweise überprüft ein *unit test* eine Methode in einem besonderen Kontext. Beispielsweise könnte man ein neues Objekt in eine Liste hinzufügen und anschließend überprüfen, ob dieses Objekt tatsächlich eingefügt wurde. Ein *unit test* wird ausgeführt, um zu belegen, dass ein Abschnitt eines Quellcodes genau das tut, was ein Entwickler von ihm erwartet. Idealerweise werden die *unit tests* parallel zum zu testenden Quellcode geschrieben. In der agilen Softwareentwicklung werden die *unit tests* sogar vor den zu testenden Komponenten entwickelt. In diesem Fall spezifizieren die Tests das Verhalten einer Softwarekomponente und sind zunächst nicht erfolgreich. Das Ziel der Entwicklung besteht darin, die Tests erfolgreich zu machen, indem man den Quellcode entwickelt. Im Folgenden wird nur die parallele Entwicklung der Tests und der jeweiligen zu testenden Systeme betrachtet.

Die Aufgabe der *unit tests* besteht nicht darin, eine formale Verifikation, Validität und Korrektheit eines Softwaremodels durchzuführen. Auch Effizienz ist zu diesem Zeitpunkt uninteressant. Das primäre Ziel ist die Überprüfung der kleinen, isolierten Teile der Funktionalität. Die Größe und die Isolierung sind essentiell, da man bei möglichen Änderungen der Software möglichst wenig Tests anpassen möchte. Die Anzahl der Tests, die man anpassen muss, kann als ein Maß für die Qualität der *unit tests* und des zu testenden Quellcodes angesehen werden.

5.7.2 Richtlinien für unit testing

Um die Effektivität der *unit tests* zu steigern und den damit verbundenen Aufwand zu verringern, sollte man beim Testen einige Richtlinien verfolgen. Als Erstes muss man sich Gedanken darüber machen, wie man die jeweilige Methode testet, bevor man mit der eigentlichen Implementierung anfängt. Wird der Aspekt der Testbarkeit während der Implementierung nicht beachtet, ist es in der Regel schwieriger den testenden Code zu erzeugen. In Extremfällen ist es sogar unmöglich ohne die zu testende Komponente umzuschreiben. Ein gutes Beispiel ist die Entscheidung, ob eine Methode auf eine Klassenvariable zugreift oder ob die benötigte Variable als Eingabeparameter an die Methode übergeben wird. Falls die Berechnung des Wertes der Variable aufwendig ist, ist es sinvoller einen Parameter in die Signatur der Methode hinzuzufügen. Außerdem wird die Methode dadurch zustandslos und damit leichter testbar. Damit sieht man, dass ein paralleles Schreiben des Testcodes zum Quellcode von enormer Bedeutung ist.

Desweiteren sollten die Entwickler sich nicht damit begnügen, ihre eigenen Tests oder nur die neuen Tests durchzuführen. In großen Systemen ist die Gefahr, dass man beim Schreiben einer neuen Funktionalität Fehler in die anderen Teile der Software einfügt, sehr groß. Aus diesem Grunde ist es von großer Bedeutung, dass zumindest am Ende des Entwicklungsprozesses alle vorhandenen Tests ausgeführt werden.

Eine Visualisierung der Testergebnisse ist ein weiteres wichtiges Kriterium für die erfolgreiche Entwicklung der Tests. Die jahrelange Geschichte der Softwareentwicklung hat gezeigt, dass die Testergebnisse immer präsent oder zumindest leicht abrufbar sein müssen. Falls Nachschlagen der Ergebnisse mit einem Aufwand verbunden ist, ist es nur eine Frage der Zeit bis die Entwickler aufhören, sich über den Erfolg zu informieren.

5.7.3 Ausreden, keine unit tests zu schreiben

Die Mehrheit der Entwickler ist einverstanden, dass *unit testing* ein fester Bestandteil des Entwicklungsprozesses einer Software sein sollte. Die Vorteile, die ein automatischer Test bietet, sind in der Regel klar. Jedoch gibt es immer noch sehr viele Projekte, die manuell getestet werden. Würde man eine Befragung durchführen, dann würde man schnell erkennen, dass die meisten Entwickler fast immer die gleichen Argumente bringen, warum sie *unit testing* vernachlässigen. Die häufigsten Ausreden, keine *Unit*-Tests zu schreiben, sind:

Es dauert viel zu lange Dieses Argument wird bevorzugt von den Neulingen des *unit testings* gebracht. Die meisten Etwickler betrachten das Testen als eine der letzten

Phasen in dem Prozess der Software - Entwicklung. Beim WASSERFALLMODELL wird beispielsweise die Testphase erst vor Auslieferung und Einsatz der Software gestartet. Wie schon erwähnt, ist es kostengünstiger die Tests während der Implementierung zu schreiben. Zum einen werden damit die Fehler früher erkannt, zum anderen wird der Aufwand minimiert.

In vielen Fällen ist der Zeitdruck so groß, dass für das automatische Testen überhaupt keine Zeit bleibt. Falls man dieser Meinung ist, sollte man sich folgende Fragen stellen:

1. Wie viel Zeit vebringt man beim Debuggen des Codes, um diesen zu verstehen?
2. Wie viel Zeit benötigt man, um den Code umzuschreiben, bei dem man fälschlicherweise geglaubt hat, dass es richtig funktioniert?
3. Wie viel Zeit verbringt man beim Isolieren und Identifizieren eines fehlerhaften Programmabschnitts?

Unit testing ermöglicht, dass man bei den erwähnten Punkten Zeit spart. Die gesparte Zeit kann investiert werden, um die *unit tests* zu schreiben. Auf die Dauer gesehen kann man sogar von einer Zeitersparnis reden.

Testen ist nicht meine Aufgabe Dieses Argument ist nur teilweise gültig. In der Tat gibt es bei vielen Softwareherstellern Testgruppen, die sich ausschließlich auf das Testen konzentrieren. Es gehört jedoch zu den Aufgaben eines Softwareentwicklers eine fehlerfreie Software zu schreiben. Eine Testgruppe hat in der Regel eine Menge von Testbeschreibungen, die durchgeführt werden müssen. Falls bei einem manuellen Test ein Fehler gefunden wird, muss dieser korrigiert werden. Während der Korrekturzeit überprüft die Testgruppe weitere Tests. Sobald die gefundenen Fehler korrigiert werden, werden diejenigen Tests wiederholt, bei denen die Fehler entdeckt wurden. Es ist wirtschaftlich überhaupt nicht machbar, dass man nach jeder Fehlerkorrektur alle Tests wiederholt. Im allgemeinen baut man beim Korrigieren der Fehler neue Fehler ein, die eventuell nicht mehr entdeckt werden. Aus diesem Grunde liegt die Verantwortung, eine fehlerfreie Software zu schreiben, bei dem jeweiligen Autor. Im Gegensatz zu menschlichen Testern können die automatische Tests jeden Tag, ja jede Stunde, eingesetzt werden.

Ausführung der Tests dauert viel zu lange Wenn dies der Fall ist, dann sind die Tests nicht richtig programmiert. Die meisten Test sind kleine Codesegmente, die kleine und isolierte Teile der Software testen. Falls dies nicht der Fall ist, sollte man sich überlegen, seine Tests umzuschreiben. Beispielsweise ist ein Test, der eine Datenbankverbindung benötigt, ein schlechter Test. Die Aufgabe der *unit tests* besteht nicht darin, zu überprüfen, ob eine SQL-Query das richtige Objekt in der Datenbank findet. An dieser Stelle sollte man MOCK OBJECTS einführen. Außerdem ist ein *unit test* nicht geeignet, zu überprüfen, ob alle Komponenten einer Software richtig zusammen funktionieren. Dies ist die Aufgabe der *integration tests*.

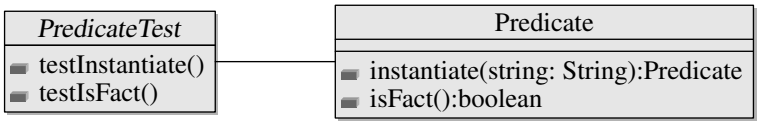

Abbildung 5.2: Klassendiagramm `JUnit`

5.7.4 JUnit

JUnit ist ein *open source unit testing framework* für Java entwickelt von Kent Beck and Erich Gamma. Innerhalb kürzester Zeit ist JUnit zum meistverwendeten *unit testing framework* geworden.

5.7.4.1 Namenskonventionen bei JUnit

Bei der Benutzung von JUnit muss man sich an einige Namenskonventionen halten. Im Bild 5.2 sieht man ein Beispiel für den Test der Klasse *edu.udo.cs.ie.kimas.logic.Predicate*. Für jede zu testende Java-Klasse wird eine Testklasse geschrieben, deren Name sich aus dem Präfix *Test* und den Namen der zu testenden Klasse zusammensetzt. Im Beispiel heißt *TestPredicate* die Testklasse von *Predicate*. *TestPredicate* wird von der Klasse *junit.framework.TestCase* abgeleitet und erbt damit eine große Menge an Funktionalität. Jede Methode einer Testklasse, die mit dem Präfix *test* anfängt und in ihrer Signatur keinen Eingabe-Parameter enthält, wird als eine Test-Methode angesehen und beim Test ausgeführt.

5.7.4.2 Vorgehen beim Testen mit JUnit

Beim Entwickeln einer Testklasse werden in der Regeln folgende Schritte durchgeführt:

1. Setup aller benötigten Zustände, die für das Testen nötig sind. Beispielsweise Instantiierung der benötigten Objekte oder Allokation der benötigten Ressourcen.
2. Aufrufen der zu testenden Methode
3. Überprüfung des erwarteten Verhaltens
4. Aufräumen der Testumgebung

Für das Erzeugen und das Aufräumen der Testumgebung liefert die Klasse *TestCase* aus dem Packet *junit.framework* die benötigte Funktionalität: Methoden *setUp()* und *tearDown()*. Im Listing 5.1 ist ein Beispiel für einen einfachen JUnit-Test zu sehen.

Jeder Test wird unabhängig von allen andere Tests gestartet. Um dies zu gewährleisten werden die beiden Methoden *setUp()* und *tearDown()* für jede einzelne Test-Methode ausgeführt. Für das Beispiel aus dem Listing 5.1 ergibt sich dann die folgende Ausführungssequenz, wobei die Reihenfolge der Test-Methoden unabhängig von dem Auftreten im Code ist:

1. setUp()

```
public class MathTest extends TestCase {
        private double x;
        private double x;

        protected void setUp() {
                x = 2.0;
                y = 3.0;
        }
        protected void tearDown() {
                ...
        }

        public void testAdd(){
                double result = x + y;
                assertTrue(result == 5.0);
        }

        public void testMultiply(){
                double result = x * y;
                assertTrue(result == 6.0);
        }
}
```

Listing 5.1: Einfacher Test

2. testAdd()
3. tearDown()
4. setUp()
5. testMultiply()
6. tearDown()

5.7.4.3 JUnit Asserts

Wie schon erwähnt, stellt die Klasse *junit.framework.TestCase* eine breite Palette an Funktionalität zur Verfügung. Darunter sind die so genannten *assert methods*. Mithilfe dieser Methoden ist es sehr einfach zu überprüfen, ob bestimmte Bedingungen erfüllt sind. Beispielsweise überprüft man, ob zwei Objekte identisch sind oder ein boolescher Wert wahr ist. Im Folgenden werden einige dieser Methoden erläutert:

assertEquals(Object expected, Object actual) ist das am häufigsten angewendete *assert* zur Überprüfung von zwei Objekten auf Gleichheit. *Expected* ist das erwartete Ergebnis und wird oft hart kodiert. *Actual* wird von dem, unter Test stehenden, Code produziert. Dabei werden entsprechende *equals()*-Methoden der zu vergleichenden Objekte benutzt. Es gibt eine weitere Methode mit vergleichbarer Signatur: *assertEquals(String message, Object expected, Object actual)*. Im Falle eines nicht erfolgreichen Vergleichs bekommt man eine vom Entwickler definierte Nachricht zu sehen. Desweiteren gibt es eine Reihe vergleichbarer Methoden für alle primitiven Typen in JAVA: *assertEquals(boolean expected, boolean actual)*, *assertEquals(int expected, int actual)*, *assertEquals(short expected, short actual)* usw.

assertNotEquals(Object expected, Object actual) ist das Gegenstück zu *assertEquals*.

assertSame(Object expected, Object actual) bestätigt, dass die unter Vergleich stehenden Objekte Referenzen auf das gleiche Objekt sind. Ansonsten wird der Test fehlschlagen.

assertNotSame(Object expected, Object actual) ist das Gegenstück zu *assertSame*.

assertNull(Object object) überprüft, ob ein Objekt *null* ist. Falls dies nicht der Fall ist, schlägt der Test fehl.

assertNotNull(Object expected, Object actual) ist ähnlich wie *assertNull*. Es überprüft, ob ein Objekt nicht *null* ist.

assertTrue(boolean condition) bestätigt, dass der gegebene boolesche Ausdruck *true* ist. Ansonsten schlägt der Test fehl.

assertFalse(boolean condition) bestätigt, dass der gegebene boolesche Ausdruck *false* ist. Ansonsten schlägt der Test fehl.

5.7.4.4 TestSuite

Mehrere Test-Klassen können zu einer *TestSuite* gebündelt werden. Dies kann man durch Benutzung der Klasse *junit.framework.TestSuite* erreichen. Im Listing 5.2 ist ein Beispiel der *TestSuite* für das Paket *edu.udo.cs.ie.kimas.logic* zu sehen. Üblicherweise erzeugt man pro Paket eine *TestSuite*, die in der Klasse *AllTests* enthalten ist. Die *TestSuites* können ihrerseits zu weiteren *TestSuites* zusammengefasst werden, so dass man durch Ausführen einer *TestSuite* auf der obersten Ebene alle möglichen Tests ausführen kann. Im Listing 5.3 ist die *TestSuites* des Projekts KIMAS zu sehen. Für weitere Informationen über *unit testing* mit JUNIT siehe [OM04] und [AH03].

```
public class AllTests {

        public static Test suite() {
                TestSuite suite = new TestSuite();
                suite.addTestSuite(NegTest.class);
                suite.addTestSuite(NotTest.class);
                suite.addTestSuite(PredicateTest.class);
                suite.addTestSuite(ProgramTest.class);
                suite.addTestSuite(RuleTest.class);
                return suite;
        }

}
```

Listing 5.2: *TestSuite* des Pakets *edu.udo.cs.ie.kimas.logic*

```
public class AllTests {

        public static Test suite() {
                TestSuite suite = new TestSuite();
                suite.addTest(edu...dlv.AllTests.suite());
                suite.addTest(edu...factory.AllTests.suite());
                suite.addTest(edu...logic.AllTests.suite());
                suite.addTest(edu...programs.AllTests.suite());
                suite.addTest(edu...plans.AllTests.suite());
                return suite;
        }

}
```

Listing 5.3: *TestSuite* des Projekts

```
public void doSomething(){
        ....
        if(flag){
                System.out.println("Hello world");
        }
        ....
}
```

Listing 5.4: Line coverage

```
try{
        ....
}catch(Exception e){
        //exception handling
}
```

Listing 5.5: Vergessenes catch-block

5.7.5 Was ist code coverage?

Es gibt viele Arten, wie man eine *code coverage* messen könnte. Wir betrachten nur zwei davon, da das von unserem Projekt eingesetztes *coverage tool* diese beiden zur Verfügung stellt.

5.7.5.1 Line coverage

Die einfachste Form der *code coverage* ist die Überprüfung, ob alle Zeilen des Codes bei einem Test durchgelaufen wurden. Bei einem linearen Verlauf wird die Überdeckung des Codes ziemlich hoch sein, da die Zeilen von oben nach unten durchlaufen werden. Lücken können entstehen, sobald Bedingungen erfüllt sein müssen, damit gewisse Teile des Codes ausgeführt werden. Betrachten wir ein Beispiel im Listing 5.4. Der String „Hello world“ wird nur dann ausgegeben, wenn vorher im Code der Wert der Variable *flag* auf *true* gesetzt wurde. Man kann sich vorstellen, dass bei komplexen Anwendungen mehrere Hunderte Zeilen des Codes in solche *if-statements* eingeschlossen sind. Dadurch kann es zur erheblichen Lücken beim Testen kommen, was sich auf die Qualität des Codes auswirkt.

Einen der häufigsten Fälle, der beim Testen vergessenen Codeabschnitte, sieht man im Listing 5.5. Viele Entwickler schreiben Tests nur für den Code innerhalb des *try-blocks*. Besonders Neulinge begnügen sich damit, die Software nur auf das normale Verhalten hin zu testen. Es ist aber von enormer Bedeutung über das Verhalten in Ausnahmefällen Bescheid zu wissen.

```
if (a && b){
        ...
}
```

Listing 5.6: Durch UND verknüptftes if-statement

Die meisten *coverage tools* visualisieren ihr Ergebnis auf der Code-Ebene. Meistens gibt es eine farbliche Unterscheidung der ausgeführten und nicht ausgeführten Zeilen. Dies ist besonders hilfreich, um die vergessenen Code-Passagen, wie den Inhalt der *catch-blocks*, zu identifizieren.

5.7.5.2 Branch coverage

Die *line coverage* verschafft einen ersten Eindruck darüber, wie gut eine *test suite* ein Softwareprodukt testet. Bei näheren Betrachtung stellt sich jedoch heraus, dass *line coverage* allein nicht ausreichend ist. Auch wenn ein Test für die Methode aus dem Listing 5.4 eine *coverage* von 100% erzeugt, kann man nicht sicher sein, dass man alle möglichen Fälle abgedeckt hat. Mit den Mitteln der *line coverage* kann man nur eine Aussage darüber machen, dass alle möglichen Abzweigungen erwischt wurden. Im Falle eines *if-statements* kann man aber nicht unterscheiden, ob der Fall, bei dem die if-Bedingung nicht erfüllt war, auch behandelt wurde. Im Listing 5.6 wird ersichtlich, wo die Schwächen der *line coverage* sind. Falls ein Test mit den Werten *a=true* und *b=true* ausgeführt wurde, berichtet die *line coverage*, dass das *if-statement* vom Test erfasst wurde. Dies erzeugt aber einen falschen Eindruck, da es noch drei weitere Fälle gibt, die betrachtet werden sollten. In der Tabelle 5.2 sind alle möglichen Werte aufgelistet.

a	b
false	false
false	true
true	false
true	true

Tabelle 5.2: Alle möglichen Werte für zwei binäre Variablen

Die *branch coverage* liefert in diesem Beispiel erst dann 100%, wenn alle möglichen Fälle getestet wurden.

5.7.6 Arbeitsweise der *coverage tools*

Die *coverage tools* berechnen das Maß der Überdeckung mittels *Instrumentierung*. Dabei werden im zu testenden Code Proben abgelegt, die markiert werden, sobald sie ausgefürt wurden. Es gibt unterschiedliche Arten der *Instrumentierung*:

source-level instrumentation Einige Tools analysieren den *source code* und fügen zusätzliche Code-Passagen hinzu, die die Aussage machen, ob und wie oft die jeweiligen Zeilen durchlaufen wurden. Dabei wird der Aufruf des Compilers abgefangen und der exisitierende *source code* wird verändert. Somit entsteht *byte code*, welcher die zur Überdeckung benötigten Zeilen enthält. Der Nachteil dieser Technik ist, dass sie sprachenspezifisch ist. Außerdem muss dabei der *build process* der Software angepasst werden.

bytecode-level instrumentation Bei dieser Technik geht es um die Modifikation des *byte codes*. Dabei wird der eigentliche *source code* kompiliert und das Ergebnis der Kompilierung wird instrumentiert. Auch in diesem Fall muss der *build process* angepasst werden.

runtime instrumentation Diese Art der Instrumentierung ist vergleichbar mit *bytecode-level instrumentation*, wobei der *byte code* erst während der Laufzeit modifiziert wird.

5.7.7 Cobertura

COBERTURA ist ein *open source tool* zur Messung der Überdeckung für JAVA-Programme. Es berechnet den Prozentsatz des Codes, welcher von den Tests aufgerufen wurde. COBERTURA wird unmittelbar vor dem Testen gestartet. Dabei werden die zu testenden Klassen mittels *bytecode-level instrumentation* modifiziert und die Tests an den instrumentierten Klassen durchgeführt. Nach den Tests werden Reports generiert, die zur Analyse und Interpretation der Tests angewendet werden können. Die Reports werden im XML- und HTML-Format generiert und enthalten Informationen über *line coverage* und *branch coverage*. Weitere Informationen über COBERTURA findet man unter [Dol06].

5.7.7.1 Benutzung von Cobertura

COBERTURA kann sowohl von der Kommandozeile als auch als ANT-Tasks gestertet werden. Im Folgenden werden nur die ANT-Task betrachtet. Als Erstes werden dem ANT-Task die COBERTURA-Parameter bekannt gemacht. Im Listing 5.7 wird der Ort von COBERTURA eingestellt und *classpath* für die Instrumentierung erzeugt. Diese Zeilen werden üblicherweise am Anfang des *build files* build.xml platziert. Als Nächstes werden die zu testenden Klassen kompiliert und von COBERTURA instrumentiert. Dies ist in den Listings 5.8 und 5.9 zu sehen.

```
<property name="cobertura.dir" value="C:/cobertura" />

<path id="cobertura.classpath">
    <fileset dir="${cobertura.dir}">
        <include name="cobertura.jar" />
        <include name="lib/**/*.jar" />
    </fileset>
</path>

<taskdef classpathref="cobertura.classpath"
         resource="tasks.properties" />
```

Listing 5.7: COBERTURA-Parameter

Anschließend werden die JUNIT-Tests gestartet, wobei die instrumentierten Klassen vor den originalen Klassen im *classpath* auftauchen sollten. Außerdem muss COBERTURA und ihre Abhängigkeiten im *classpath* erscheinen. Ein Beispiel kann im Listing 5.10 betrachtet werden. Falls die Tests erfolgreich waren, wird ein Report erzeugt. Siehe dazu Listing 5.11. In diesem Report kann man jede einzelne Klasse anschauen und das Ergebnis des Tests der Klasse interpretieren. Dabei werden die ausgeführten Zeilen grün und die nicht ausgeführten Zeilen rot gefärbt. Auf diese Weise können Testlücken identifiziert und behoben werden. Für jede Klasse gibt es eine Angabe des Prozentsatzes der *line coverage* und der *branch coverage*. Neben der Klassenansicht gibt es eine Paketansicht, für die ein gewichtetes Mittel aus der Überdeckung der Klassen des Pakets und der Unterpakete berechnet wird. Die Berechnung des Mittels basiert auf der Komplexität der einzelnen Klassen. Somit tragen die längeren Klassen oder Klassen mit mehrfach geschachtelten *if-statements* verstärkt zur gemeinsamen Überdeckung des Codes bei. Für die Pakete ist außerdem die Anzahl der enthaltenen Klassen entscheidend. Auch für die Paketansicht gibt es die Angabe der *line coverage* und der *branch coverage*. Die Pakete und Klassen können nach unterschiedlichen Kriterien sortiert werden: Name, Maß der Überdeckung und Komplexität. Bei der Berechnung der Überdeckung können Klassen oder Pakete ausgeschlossen werden. Dies ist der Fall bei diesem Projekt. Das Paket *edu.udo.cs.ie.kimas.gui* und alle Klassen, die auf die JADEX-API zugreifen, wurden bei der Berechnung der *coverage* nicht berücksichtigt, da ein Testen der erwähnten Klassen mit JUNIT nicht ohne Weiteres möglich ist. Im Bild 5.3 ist ein Ausschnitt aus dem *coverage* - Report für das Projekt KIMAS zu sehen.

ackage	Line Coverage		Branch Coverage	
	59%	2899/4953	65%	520/804
kimas	21%	49/231	18%	5/28
kimas.characters	0%	0/419	0%	0/87
kimas.dlv	90%	86/96	100%	15/15
kimas.environment	48%	193/404	62%	44/71
kimas.factory	85%	1217/1440	97%	207/214
kimas.knowhow	37%	116/314	17%	7/42
kimas.logic	80%	506/629	90%	117/130
kimas.logic.operators	62%	417/674	63%	71/112
kimas.logic.programs	68%	128/188	89%	16/18
kimas.outputcatcher	92%	102/111	100%	20/20
kimas.plans	19%	85/447	27%	18/67

Abbildung 5.3: *coverage*-Report

```
<target name="compile" depends="clean,init">
        <javac destdir="${classes.dir}" debug="yes">
                <src path="${src.dir}"/>
                <src path="${src.test.dir}"/>
                <classpath refid="cobertura.classpath" />
        </javac>
</target>
```

Listing 5.8: Kompilieren des Codes

```
<cobertura-instrument todir="${instrumented.dir}">
    <ignore regex="org.apache.log4j.*" />
    <fileset dir="${classes.dir}">
        <include name="**/*.class" />
        <exclude name="**/*Test.class" />
    </fileset>
</cobertura-instrument>
```

Listing 5.9: Instrumentieren des Codes

```
<junit fork="yes" dir="${basedir}" printsummary="yes">
        <classpath location="${instrumented.dir}" />
        <classpath location="${classes.dir}" />
        <classpath refid="cobertura.classpath" />
</junit>
```

Listing 5.10: Starten der Tests

```
<cobertura-report format="html"
                  destdir="${coveragereport.dir}" >
        <fileset dir="${src.dir}">
                <include name="**/*.java" />
        </fileset>
</cobertura-report>
```

Listing 5.11: Erzeugen eines Reports

5.8 Editor

Igor Drobiazko

5.8.1 Einführung

Wie im Kapitel 6.3 beschrieben ist, werden Szenarien im XML-Format beschrieben. Desweiteren werden alle Charaktere der Szenarien und die Umwelt mittels XML beschrieben. Im Falle komplexer Szenarien ist die manuelle Spezifikation und Wartbarkeit der XML-Dokumente nicht mehr effizient und sehr aufwendig. Aus diesem Grund wurde entschieden, einen graphischen Editor zu entwicklen, der die manuelle Arbeit erleichtern sollte. Das XML-Format zur Definition der Szenarien, Charaktere und der Umwelt wurde spezifiziert mittels XML Schemas. Diese Schemas werden im Folgenden wie folgt abgekürzt:

- *CDF (character definition file)*: kimas-cdf.xsd
- *SDF (scenario definition file)*: kimas-sdf.xsd
- *EDF (environment definition file)*: kimas-edf.xsd

Zur Erzeugung des Editors wurde (Eclipse Modelling Framework (EMF) ausgewält, welches auf der MOF (meta-object facility)-Spezifikation basiert. Im Folgenden wird MOF, EMF und Realisation des Editors vorgestellt.

5.8.2 Modelle, Metamodelle und Meta-Metamodelle

Unter *meta modelling* versteht man die Konstruktion einer Menge von Konzepten innerhalb einer Domain. Dabei spielt der Begriff von Metadaten eine wesentliche Rolle. Metadaten sind Informationen, die zur Beschreibung der anderen Daten angewendet werden. Beispielsweise könnte man ein Buch durch folgende Metadaten beschreiben: Autor, ISBN, Erscheinungsjahr und Verlag. Metadaten von Modellen bezeichnet man als Metamodelle, da sie diese Modelle beschreiben. Die Modelle der Metamodelle werden dementsprechend Meta-Metamodelle bezeichnet. Vereinfachend kann man immer von Modellen sprechen, die benutzt werden, um Sachverhalte zu modellieren. Ein Modell ist also ein Metamodell, wenn es benutzt wird, um Modelle zu definieren. Dagegen ist ein Modell ein Meta-Metamodell, wenn es benutzt wird, um Metamodelle zu definieren.

5.8.3 *MOF: meta-object facility*

Die Spezifikation von MOF (META-OBJECT FACILITY) definiert eine abstrakte Sprache und ein Framework für Spezifikation, Konstruktion und Management der technologieneutralen Metamodelle. Ein Metamodell in MOF definiert eine abstrakte Sprache zur Beschreibung von Metadaten beliebiger Art. MOF wird in vielen Bereichen zur Beschreibung von Metadaten angewendet: *data warehousing* oder *model driven development*. In der Tabelle 5.3 ist die Architektur von MOF als eine 4-Schichten-Architektur dargestellt. Die M3-Schicht stellt Meta-Metamodelle bereit, die auch als M3-Modelle bezeichnet werden. Diese M3-Modelle definieren eine Sprache, die von MOF benutzt wird, um Metamodelle zu beschreiben. Solche Metamodelle befinden sich auf der M2-Schicht. Das prominenteste Beispiel eines M2-Modells ist das UML-Metamodell. Dieses wird benutzt, um UML zu definieren. Ein weiteres Beispiel für ein M2-Modell ist das XML SCHEMA-Metamodell, das XML SCHEMA beschreibt. Die Metamodelle auf der M2-Schicht beschreiben Modelle der M1-Schicht. M1-Modelle, sind Modelle, die beispielsweise mit UML beschrieben sind. Es sind also entweder UML-Diagramme oder JAVA-Klassen, die von diesen Diagrammen beschrieben werden. Auf der M0-Schicht befinden sich die instantiierten JAVA-Objekte zur Laufzeit. Für weitere Details über die MOF - Spezifikation siehe [Gro06].

M3-Layer (Meta-Metamodelle)	MOF-Modell
M2-Layer (Metamodelle)	UML-Metamodell, XML SCHEMA-Metamodell
M2-Layer (Modelle)	UML-Modelle, XML-Modelle
M0-Layer	Beschreibung der realen Welt

Tabelle 5.3: MOF-Architektur

5.8.4 *EMF: Eclipse Modelling Framework*

MOF ist nur ein abstraktes architektonisches Konzept, das einen Autausch von Metadaten quer duch alle Plattformen und Programmiersprachen erlaubt. In der JAVA-Welt ist MOF im *Eclipse Tools Project* realisiert. *Eclipse Tools Project* hat mehrere Unterprojekte. Drei von diesen Projekten beschäftigen sich mit der Modellierung:

Eclipse Modelling Framework (EMF) ist ein Framework zur Modellierung und Codegenerierung und eine Realisation von MOF.

Graphical Editing Framework (GEF) ist Framework zur Visualisierung von Modellen.

Graphical Modelling Framework (GMF) ist eine Integration von EMF und GEF.

EMF besteht aus 3 fundamentalen Bestandteilen:

EMF Core enthält das EMF-Metamodell *Ecore*, das zur Beschreibung der EMF - Modelle dient. Weitere Features sind: *modell change notifications*, Serialisierung/Deserialisierung von Modellen mittels XMI und eine *Reflection-API* zur Modifikation von EMF-Objekten.

EMF.Edit enthält wiederverwendbare Klassen zur Erzeugung der Editoren für Modelle.

EMF.Codegen ist ein Mechanismus zur Generierung der Bausteine zum Erzeugen der Editoren.

5.8.4.1 *Ecore*

Ecore ist ein ein Dialekt von UML, der von *Eclipse*-Entwicklern für EMF entwickelt wurde. Damit ist *Ecore* ein Modell. Ob es ein Metamodell oder Meta-Metamodell ist, hängt davon ab, in welchem Kontext es gerade benutzt wird. Wenn *Ecore* benutzt wird, um irgendein System zu modellieren, dann ist *Ecore* ein Metamodell. *EMF Genmodel* ist ein Metamodell, das Modelle für Codegenerierung definiert. *Ecore* seinerseits definiert *EMF Genmodel*. Damit ist *Ecore* ein Meta-Metamodell. *Ecore* kann aus verschiedenen Formaten erzeugt werden. Unter anderem sind folgende Formate möglich:

- Rational Rose Model
- Annotierte JAVA-Interfaces
- XML SCHEMA
- EMF-API

5.8.4.2 *Genmodel*

EMF Genmodel ist ein Metamodell, das Modelle für Codegenerierung definiert. Es beobachtet sein Modell (*Ecore*) und wird über jede Veränderung notifiziert. Damit werden basierend auf den Metamodellen aus den Code-Templates JAVA-Klassen generiert. Diese Klassen entsprechen den Modellen auf der M1-Schicht der MOF-Architektur. Die generierten Klassen können bearbeitet werden. Bei der erneuten Codegenerierung werden diese Modifikationen beibehalten. Für weitere Details siehe [dt06].

5.8.5 Motivation für den Editor

Wie im Kapitel 6.3 beschrieben ist, werden Szenarien im XML-Format beschrieben. Das XML-Format zur Definition der Szenarien, Charaktere und der Umwelt wurde spezifiziert mittels XML SCHEMAS. Da die XML SCHEMAS die Modelle der M1-Schicht und die entsprechenden XML-Dokumente, die bzgl. dieser Schemas validiert werden, die Instanzen der M0-Schicht darstellen, wurde entschieden, EMF als Framework zur Erzeugung des Editors zu benutzen. Ein weiterer Vorteil von EMF ist die Integration in *Eclipse*, das von den KIMAS-Mitgliedern als IDE benutzt wurde.

5.8.6 Editor-Entwicklung

Für die Entwicklung des Editors wurden sechs *Eclipse*-Projekte eingeführt.

edu.udo.cs.ie.kimas.editor ist das Hauptprojekt für den Editor. Es enthält das *Ecore*-Modell, *Genmodel*, ein Mapping zwischen XML SCHEMA und *Ecore*. Desweiteren sind in diesem Projekt die grundlegenden Klassen enthalten, die den Typen und Elementen aus den *CDF*, *SDF* und *EDF* entsprechen.

edu.udo.cs.ie.kimas.editor.edit enthält generierte Adaptor-Klassen, die von dem Plugin intern benutzt werden.

edu.udo.cs.ie.kimas.editor.editor enthält die Implementierung des Editor-Plugins für das EMF - Modell.

edu.udo.cs.ie.kimas.editor.feature repräsentiert ein Paket von Plugins. Diese Plugins werden mittels einer Update-Site zur Verfügung gestellt.

edu.udo.cs.ie.kimas.editor.tests enthält generierte *test cases* zum Testen der generierten Modell - Klassen mit JUNIT.

edu.udo.cs.ie.kimas.editor.update repräsentiert eine Update-Site, die benutzt werden kann, um den Editor zum Download bereitzustellen.

Da die Beschreibung aller Projekte den Rahmen dieses Endberichts sprengen würde, beschränken wir uns auf die Beschreibung der wichtigsten Projekte.

5.8.6.1 Projekt *edu.udo.cs.ie.kimas.editor*

Im ersten Schritt in der Entwicklung eines EMF-Editors muss ein Modell konstruiert werden. Wie schon erwähnt, wird das Modell von drei XML SCHEMAS repräsentiert: *CDF*, *SDF* und *EDF*. Ausgehend aus diesem Modell wurde ein *Eclipse EMF Project* konstruiert. Dieses Projekt wird durch folgende Schritte erzeugt:

1. Wähle im File Menu: New $\mapsto$ Project
2. In dem auftauchenden Popup Wizard wähle Ecore Model Framework $\mapsto$ EMF Project
3. Im nächsten Schritt vergebe dem Projekt einen Namen und klicke auf *Next*
4. Bei der Auswahl des Modells wähle *Ecore model* aus
5. Klicke auf *Browse workspace* und wähle die XML SCHEMAS aus, die das Modell repräsentieren. Dabei klicke auf die Checkbox *Create XML Schema to Ecore Map* und vergebe dem GENMODEL einen Namen im Feld *Generator model file name*. Dieser Name muss eine Endung *.genmodel* haben.
6. Klicke auf *Next* und im nächsten Schritt auf *Finish*.

Es wird ein *Eclipse EMF Project* erzeugt, das im Verzeichniss *model* drei Dateien enthält.

- *kimas.ecore* ist das *Ecore* Model

```
<?xml version="1.0" encoding="UTF-8"?>
<feature id="edu.udo.cs.ie.kimas.editor.feature"
         label="edu.udo.cs.ie.kimas.editor.feature"
         version="1.0.0"
         provider-name="kimas.sourceforge.net">

   <plugin id="edu.udo.cs.ie.kimas.editor"
           download-size="0" install-size="0"
           version="1.0.0" unpack="false"/>

   <plugin id="edu.udo.cs.ie.kimas.editor.edit"
           download-size="0" install-size="0"
           version="1.0.0" unpack="false"/>

   <plugin id="edu.udo.cs.ie.kimas.editor.editor"
           download-size="0" install-size="0"
           version="1.0.0" unpack="false"/>

</feature>
```

Listing 5.12: Datei feature.xml

- *kimas.genmodel* ist das Model zur Generierung der JAVA - Klassen
- *kimas.xsd2ecore* definiert das Mapping zwischen XML SCHEMA und *Ecore* Modell.

Diese Verzeichnis-Struktur kann man im Bild 5.4 betrachten. Um die Generierung der Projekte abzuschließen, führe folgende Schritte durch:

1. Doppelklick auf der Datei *kimas.genmodel*
2. in dem geöffneten *Genmodel* klicke auf den Wurzelknoten mit der rechten Maustaste und wähle *Generate All* aus. Siehe das Bild 5.5
3. Konfiguriere die generierten Projekte entsprechend den Bedürfnissen

5.8.6.2 Projekt *edu.udo.cs.ie.kimas.feature*

Diese Projekt dient zum Zusammenstellen eines *Features* aus mehreren *Plugins*. Der wichtigste Bestandteil dieses Projekts ist die Datei *feature.xml*, die im Listing 5.12 erläutert wird. Das *Feature* hat einen eindeutigen Namen und definiert in der Version 1.0.0 drei *Plugins*. Bei der Installation des *Features* kann das Attribut *provider-name* benutzt werden, um die erfolgreiche Installation feststellen zu können.

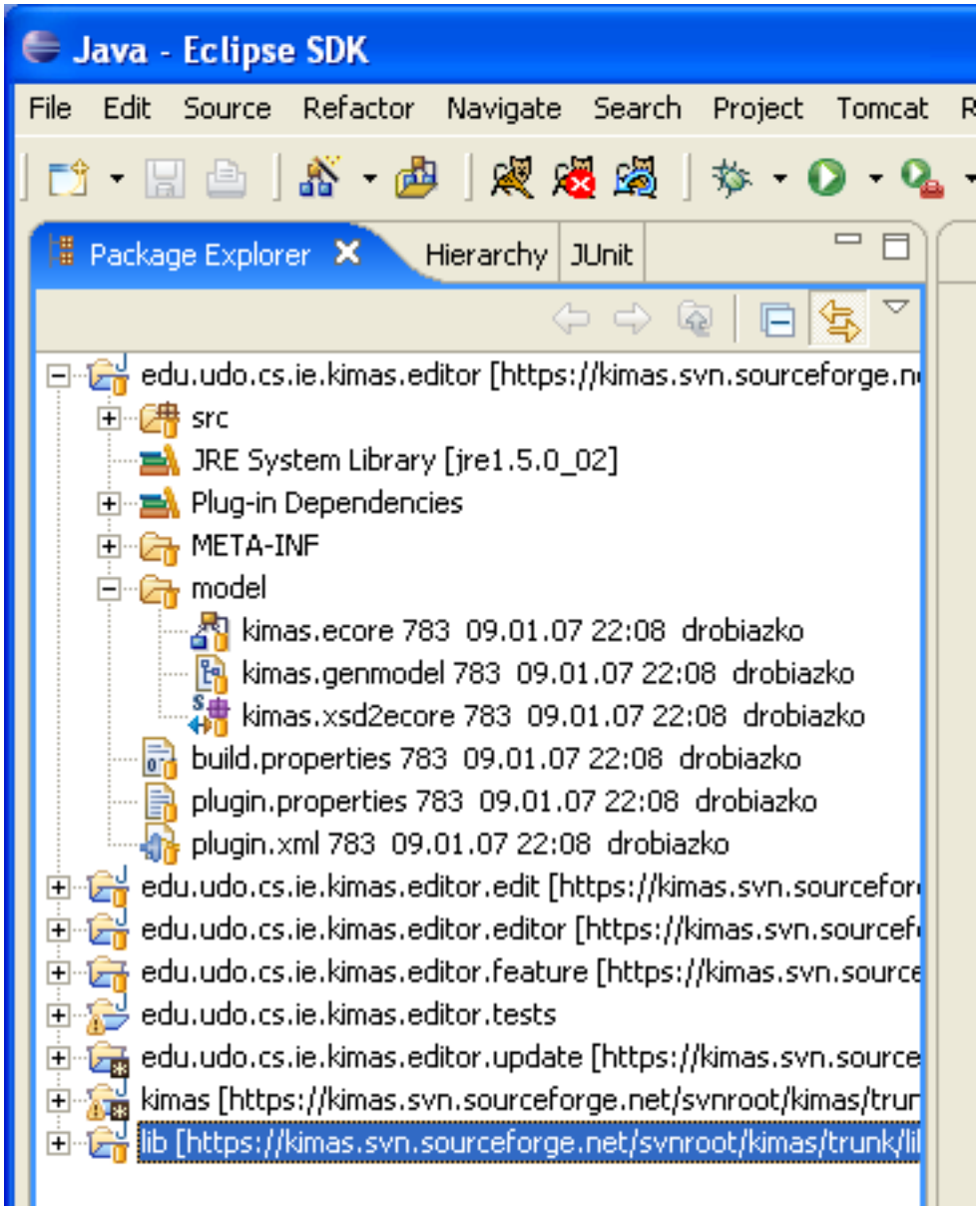

Abbildung 5.4: Projektübersicht

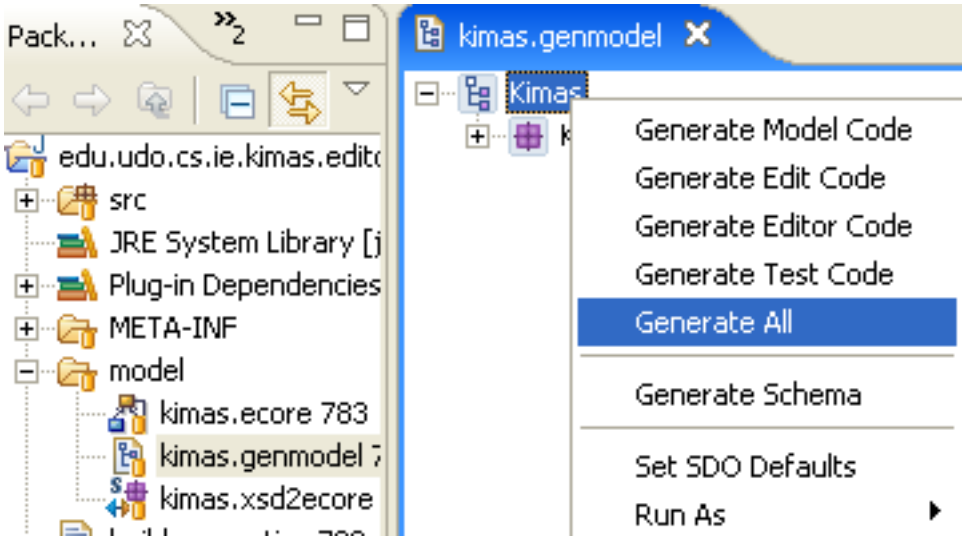

Abbildung 5.5: Generieren der Klassen

5.8.6.3 Projekt *edu.udo.cs.ie.kimas.update*

Dieses Projekt dient zur Generierung einer Web-Seite, von der das *Feature* mit den *Plugins* runtergeladen werden kann. Der wichtigste Bestandteil dieses Projektes ist die Datei *site.xml*, in der die *Plugins* aufgelistet werden. Diese Datei wird benutzt, um den Inhalt der Update - Seite zu generieren. Dieser besteht aus folgenden Bestandteilen:

- Datei *index.html* als Root der Seite
- Verzeichnisse *features* und *plugins*, die die jeweiligen Archive enthalten
- Verzeichniss *web* mit einem *CSS* und XSLT - Template, das auf *index.html* angewendet wird.

Alle diese Bestandteile müssen auf einen Server hochgeladen werden, damit man den Editor mittels *Eclipse Download Manager* installieren kann. Im Listing 5.13 ist ein Beispiel für eine *site.xml* zu sehen.

5.9 GUI

Mohamed Dalil

Die GUI (*G*raphical User Interface) ist die Grafische Benutzungsschnittstelle, die die Aufgabe hat, Anwendungssoftware auf einem Rechner mittels grafischer Elemente zu bedienen. Zur graphischen Interaktion zwischen unserer KIMAS-Anwendung und dem Benutzer haben wir eine GUI entwickelt. Dabei haben wir die Swing-Komponenten der

```
<?xml version="1.0" encoding="UTF-8"?>
<site>
   <feature
      url="edu.udo.cs.ie.kimas.editor.feature_1.0.0.jar"
      id="edu.udo.cs.ie.kimas.editor.feature"
      version="1.0.0">
       <category name="Editor"/>
   </feature>
   <category-def name="Editor" label="Editor">
       <description>
           Plugin for editing CDF, SDF and ADF.
       </description>
   </category-def>
</site>
```

Listing 5.13: Datei site.xml

Programmiersprache JAVA verwendet. Die Swing Bibliothek bietet mächtige Werkzeuge, die viele Bereiche der GUI Entwicklung abdecken. Von diesen Werkzeuge haben wir im Wesentlichen die folgenden Klassen verwendet:

1. Layout-Manager, ist dafür verantwortlich, Elemente eines Containers nach einem bestimmten Verfahren anzuordnen

2. Klassen, die grafische Komponenten bieten, wie Fenster, Schaltflächen, Textfelder, Menüs und Container.

3. Klassen, die ein Model zur Behandlung von Ereignissen definieren.

Für eine detaillierte Beschreibung der GUI Entwickelung mit Hilfe von Swing Komponenten wird auf [Ull06] verwiesen. Zunächst gehen wir auf die Daten, die in unsere GUI dargestellt werden ein. Ferner stellen wir die Abhängigkeit der GUI-Komponenten vor. Anschließend gehen wir auf die Formatierung der Ausgaben ein.

5.9.1 Schnittstellen zur anderen KiMAS-Klassen

In der GUI werden hauptsächlich das Innenleben der Agenten und die Kommunikation zwischen den Agenten in einem Szenario dargestellt. Das Innenleben eines Agenten wird durch sein aktuelles Wissen, seine aktuell verfolgten Ziele, sein Know-How und sein aktuelles Vertrauen zu den anderen Agenten charakterisiert. Die Kommunikation zwischen den Agenten wird durch drei Nachrichtentypen (*info, debug* und *message*) dargestellt. Die für die Darstellung in der GUI benötigten Daten werden durch zwei Schnittstellen zur Verfügung gestellt:

Die erste Schnittstelle zwischen der GUI und den anderen KIMAS-Klassen ist die Klasse `Outputcatcher`. Diese Klasse stellt alle dynamischen Ausgaben zur Verfügung,

die durch die GUI dargestellt werden sollen. Die dynamischen Ausgaben sind die zeitabhängigen Daten, die sich während des Szenarios ändern. Zu den zeitabhängigen Daten zählen: die Kommunikation zwischen den Agenten, das Wissen des Agenten, seine verfolgten Ziele und sein Vertrauen zu den anderen Agenten. Das aktuelle Wissen eines Agenten wird in der GUI durch drei Wissenstypen *ASP-Belief*, *Beliefbase* und *Beliefstate* dargestellt. Auf den Unterschied zwischen den drei Wissenstypen der Agenten wird in den anderen Abschnitten eingegangen. Während der JADEX-Ausführung wird in der `Outputcatcher` Klasse eine Liste mit `Output`-Objekten gefüllt. Dabei charakterisieren die `Output`-Objekten die Änderungen, die bei den dynamischen Daten stattfinden. Ein `Output`-Objekt besteht aus vier Attributen:

- *agent*: Name des Agenten, zu dem das `Output` Objekt gehört.
- *time*: Die Zeit, bei der das `Output` Objekt erzeugt wurde.
- *logtype* und *content* der `Output`-Objekte: Um zu unterscheiden, welche Datenänderungen das `Output`-Objekt darstellt, hat jedes `Output`-Objekt einen Logtype. Der Inhalt des `Output`-Objektes variiert abhängig vom *Logtype* des `Output` Objektes. In der Tabelle 5.4 sind alle möglichen Logtypen und die dazugehörigen Inhalte eines `Output` Objektes aufgelistet.

logtype	Inhalt	Type
info	allgemeine Information	*String*
debug	*Debug* Nachrichten	*String*
ASP-Belief	Antwortmenge des aktuellen Wissens	*Program*
beliefState	aktuelles Wissen nach der Anwendung eines Wissenoperators	*Program*
beliefBase	aktuelles logisches Wissen eines Agenten	*ProgramSet*
message	Nachrichten zwischen den Agenten	*KimasMessage*
reliabilities	Glaubwürdigkeit der anderen Agenten	*Map*
userInform	Informationen von einem Agent an den Nutzer	*String*
adoptGoal	aufgenommenes Ziel	*Tuple*
removeGoal	entferntes Ziel	*Tuple*

Tabelle 5.4: Logtype und Inhalt der Output Objekte

Die zweite Schnittstelle zu den KIMAS-Klassen sind die (CDF, SDF, EDF) *Factories*. Bei diesen *Factories* handelt es sich allerdings nicht nur um eine Schnittstelle zur GUI, sondern sie haben auch andere Aufgaben, die in anderen Abschnitten beschrieben werden. Aus diesen *Factories* (CDF, SDF, EDF) werden die statischen Ausgaben geholt, die in der GUI repräsentiert werden sollen. Die statischen Ausgaben sind zeitunabhängige Initialdaten, die in zwei Kategorien unterteilt werden können:

1. die initialen Daten der Agenten, wie ihr Initialwissen, ihre Initialziele und ihr Know-How

2. die initialen Daten des Szenarios, wie zum Beispiel, Anzahl bzw. Namen der Szenen in dem Szenario etc.

Folgende statische Daten werden von den *Factories* geholt:

- Die Namen der Agenten, die in dem Szenario vorhanden sind. Diese Namen werden von der SDF *Factory* mit der Methode `getAllAgents()` geholt.

- Die Anzahl der Szenen werden von der EDF *Factory* mit der Methode `getAllScenes()` geholt.

- Die *Icons* der Agenten, die im `Graphicpanel` angezeigt werden, werden aus der CDF *Factory* geholt.

- *Initialbelief*: Das allgemeine *Initialbelief*, das jeder Agent in dem Szenario haben soll, wird von der SDF *Factory* geholt. Das spezifische *Initialbelief*, dass das Wissen eines Agenten spezifiziert, wird von der CDF *Factory* geholt. Die beiden Typen vom *Initialbelief* werden miteinander gemischt und dem entsprechenden Agent zugeordnet.

- *Know-How*: Da das *Know-How* eine agentenspezifische Fähigkeit ist, wird es von der CDF *Factory* geholt. Allerdings gibt es auch allgemeines *Know-How*, das jeder Agent in dem Szenario besitzen soll. Dieses allgemeine *Know-How* wird von der SDF *Factory* geholt. Die beiden *Know-How* werden gemischt und zu dem entsprechenden Agenten zugeordnet.

5.9.2 Verwaltung der Daten

Da die in der GUI dargestellten Daten aus verschiedenen Datenquellen kommen, haben wir eine neue Hilfklasse mit dem Name `GuiOutput` eingeführt, die diese Daten aus den entsprechenden Schnittstellen holt. Ferner werden in dieser Klasse einige Ausgaben vor der Darstellung formatiert und vorbereitet. Das heißt, dass diese Klasse einen Zwischenschritt zur Verwaltung der Daten darstellt, die in der GUI angezeigt werden sollen. Die Verwaltung der Daten beinhaltet folgende Schritte:

- Holen der statischen Daten aus den *Factories*: Die Initialdaten der Agenten werden mit der Methode `initializeAgentStates`() aus den entsprechenden *Factories* geholt. Diese Daten werden in zwei *Maps* gespeichert. Die erste *Map* ist für die Speicherung von *Initialbeliefs* der Agenten, während die zweite *Map* für die Speicherung des *Know-Hows* der Agenten zuständig ist.

- Holen der Daten aus dem `Outputcatcher`: Nachdem JADEX zu Ende ausgeführt wird, wird im `Guioutput` eine Liste mit den erzeugten `Output` Objekten gefüllt.

- Darstellung einiger dynamischer Ausgaben, die vom `Outputcatcher` nicht geliefert werden, wie zum Beispiel: *Beliefchange*, *Goalchange* und *GoalState*.

  Darstellung von *Beliefchange*: Wenn ein Agent sein Wissen zu einem Zeitpunkt t ändert, wird ein Output Objekt mit dem folgenden Inhalt erzeugt:

  > (Agentenname, Zeitpunkt=t, Logtype = aspBelief, content= das neue aktuelle Wissen)

  Um die Änderung zum vorherigen Wissen in der GUI darstellen zu können, gehen wir wie folgt vor: Wir laufen die Liste der `Output` Objekte zurück, bis wir ein Output Objekt mit dem folgenden Inhalt finden:

  > (gleicher Agentenname, beliebiger voriger Zeitpunkt, Logtype = aspBelief, content= Wissen)

  Der content der beiden Output Objekte wird miteinander verglichen und die Differenz unter dem Beliefchange des Agenten dargestellt. Zur Darstellung von *Goalchange* und *GoalState* verwenden wir das gleiche Verfahren.

- Formatierung einiger Ausgaben: auf die Formatierung wird in Abschnitt 5.9.4 eingegangen

5.9.3 Abhängigkeiten der Komponenten

Zuerst stellen wir die in diesem Abschnitt verwendeten Namen kurz vor:

- `Dialogdisplay`: ist eine GUI-Komponente zur Anzeige der Kommunikation zwischen den Agenten. Zusätzlich werden in `Dialogdisplay` die Informationen von den Agenten an den Nutzer dargestellt.
- `Agentdisplay`: ist eine GUI-Komponente zur Anzeige vom Agenteninnenleben.
- `Timedisplay`: ist eine GUI-Komponente zur Anzeige der aktuellen Zeit
- `SceneCombobox`: ist eine GUI-Komponente zur Anzeige vom Name der aktuellen Szene. Ausserdem kann durch diese Komponente eine Szene ausgewählt werden
- `Graphicpanel`: ist eine GUI-Komponente zur Anzeige der *Icons* der Agenten. Diese Komponente hat allerdings auch andere Aufgaben, die unter 5.9.3.3 erläutert werden.
- `Agentencombobox`: ist eine GUI-Komponente, die die Namen der in der Szenario vorhandenen Agenten enthält.
- `Agentstatecombobox`: ist eine GUI-Komponente, die die Namen der Ausgaben enthält, wie zum Beispiel, *aspBelief*, *Beliefstate*, *Initialbelief* und *Knowhow* etc.

5.9.3.1 Anforderungen

Durch die GUI soll der Nutzer ein Szenario auswählen und verfolgen. Um eine bessere Übersicht der ausgegebenen Daten zu erhalten, kann der Nutzer den zeitlichen Abstand zwischen den Ausgaben variieren. Des Weiteren kann er auswählen, was angezeigt wird. Daher werden die Ausgaben in verschiedene Komponenten aufgeteilt dargestellt.

Bei der Darstellung der Daten in der GUI spielen die folgenden Faktoren eine Rolle:

- Zeit: jede Ausgabe hat einen Zeitindex, durch den die Reihenfolge der Darstellung der `Output` Objekte festgelegt wird. Ein `Output` Objekte, welches früher stattfindet, erhält einen niedrigeren Zeitindex.
- *Logtype*: Durch den *Logtype* werden die `Output`-Objekte in den vorgeschriebenen Komponenten der GUI dargestellt, d. h. sie werden thematisch unterteilt dargestellt.
- Name eines Agenten: der Name eines Agenten wird in einigen Komponenten verwendet, um die zum Agenten gehörenden Ausgaben anzuzeigen.

Damit die GUI diese Anforderungen erfüllen kann, muss sie einen Zeitgeber haben. Dieser Zeitgeber ändert die abstrakte Zeit ständig. Die Komponenten der GUI werden bei jedem Zeitwechsel über diese Zustandsänderung informiert. Die Komponenten aktualisieren sich entsprechend, in dem sie die richtigen Ausgaben in der richtigen Reihenfolge anzeigen (im Fall der *Textareas*) oder sich richtig anpassen (im Fall des `Graphicpanel` und der `SceneCombobox`).

Zusätzlich müssen bei der Anzeige der `Output` Objekte die Logtypen und evtl. die Namen der Agenten berücksichtigt werden, um den Inhalt der `Output` Objekt in der richtigen GUI-Komponente anzuzeigen.

Um diese Zusammenhänge zwischen den GUI Komponenten gerecht zu werden, verwenden wir das `Observer` *Pattern*. Das `Observer` *Pattern* ist ein *Design Pattern*, das eine bewährte Vorlage für die Lösung bestimmte Entwurfsprobleme ist. Zunächst wird dieses *Patterns* kurz erläutert. Anschließend wird die Realisierung dieses Konzeptes in unsere GUI vorgestellt.

5.9.3.2 Konzept

Das Pattern besteht aus zwei Klassen, dem `Observer` (Beobachter) und dem `Observable` Objekt. Der `Observer` beobachtet das Objekt, dieses benachrichtigt bei Änderungen seines Zustandes alle `Observer` die an ihm interessiert sind. Deswegen bietet das beobachtete Objekt einen Mechanismus, um Beobachter an- und abzumelden und diese über Änderungen zu informieren. Es kennt alle seine Beobachter nur über die Schnittstelle Beobachter. Es meldet jede Änderung an jeden angemeldeten Beobachter. Die Beobachter implementieren ihrerseits eine (spezifische) Methode, um auf die Änderung zu reagieren. Das `Observer` *Pattern* findet bei JAVA *Swing* weite Verwendung, deswegen stehen in JAVA die *Interfaces* `java.util.Observable` und `java.util.Observer` zur Verfügung, die das `Observer` *Pattern* implementieren. Das beobachtete Objekt hat zwei Methoden um interressierte `Observer` hinzuzufügen `addObserver()` und wegzunehmen `removeObserver()`.[dev03]

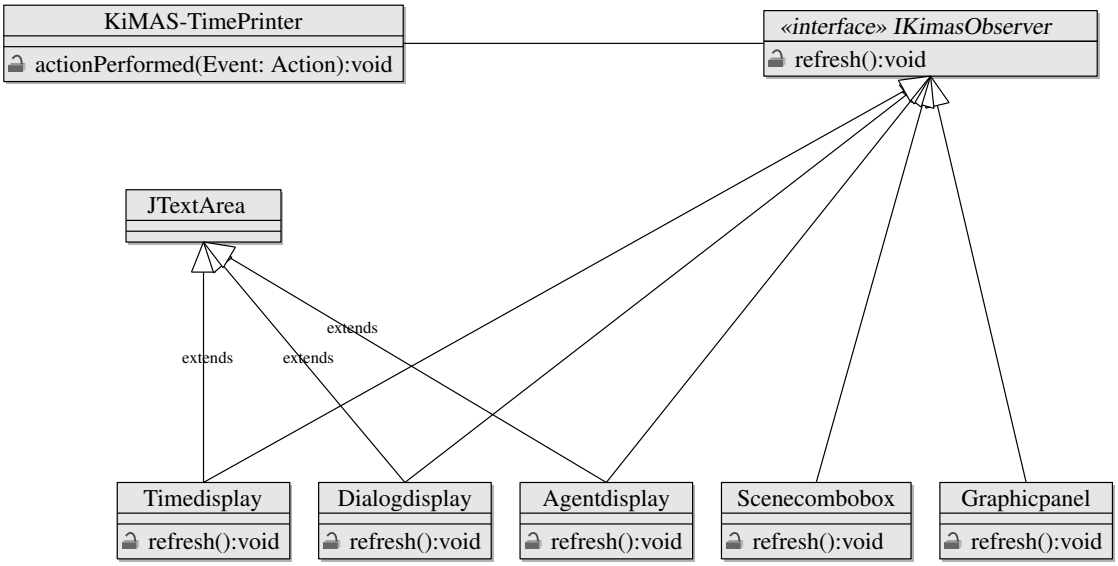

Abbildung 5.6: Klassendiagramm

```
public interface Observable {
        public void addObserver( Observer o );
        public void removeObserver( Observer o );
}
```

Listing 5.14: Observable

Ändert sich der Zustand eines beobachteten Objektes, so wird in allen registrierten *Observern* die Methode `update(...)` aufgerufen.

```
public interface Observer {
        public void update( Observable o );
}
```

Listing 5.15: Observer

Eine ähnliche Struktur verwenden wir für die Realisierung unserer GUI, siehe Abbildung 5.6

5.9.3.3 Realisierung

Die Einstellung der Geschwindigkeit funktioniert mittels eines Schiebereglers, wodurch die Ausgabe beschleunigt oder verlangsamt werden kann. Der Zeitgeber wird durch ein Objekt der Klasse `javax.swing.Timer` realisiert, welches in Zeitintervallen ein *Action-Event* erzeugt. Die eingestellte Geschwindigkeit wird im ersten Parameter des Konstruktors des Timer Objektes übergeben. Durch den zweiten Parameter wird ein *Action-Listener* (`KimasTimePrinter` Objekt) zum `Timer`-Objekt hinzugefügt. Dieses Objekt reagiert auf die Ereignisse, die durch das `Timer`-Objekt ausgelöst werden, in dem es eine

`actionPerformed` Methode ausführt. In dieser Methode wird zuerst der Zeitschritt um eins erhöht. Anschließend werden so genannte Beobachter mittels der Methode `refreshAll` über diese Zeitänderung informiert. Bei den Beobachtern handelt es sich um `Dialog-display`, `Timedisplay`, `Agentdisplay`, `SceneCombobox` und `Graphicpanel`. Im Folgenden werden die `refresh` Methoden jedes Beobachters kurz vorgestellt.

- `Dialogdisplay`: in der `refresh` Methode wird abgefragt, ob ein `Output` Objekt vom Logtype *(Info, debug, message, userinform)* zu dieser Zeit existiert. Der Inhalt dieses `Output` Objektes wird dann in `Dialogdisplay` dargestellt.
- `Timedisplay`: Die `refresh` Methode zeigt die Zeitänderung in der `Timedisplay`-komponente an.
- `Agentdisplay`: Durch die `refresh` Methode werden `Output` Objekte vom *logtype* *aspBelief, beliefState, beliefBase, adoptGoal* oder `removeGoal` sowie *Initialbelief* und *Knowhow* angezeigt. Die Anzeige dieser `Output` Objekte hängt von dem ausgewählten Agent in der `Agentencombobox` und dem ausgewählten *Item* in der `Agentstatecombobox` ab.
- `Scenecombobox`: Die `refresh` Methode zeigt den Namen der Szene zu diesem Zeitpunkt an.
- `Graphicpanel`: Die `refresh` Methode im `Graphicpanel` hat die folgenden Aufgaben:

  1. Die *Icons* der aktiven Agenten in einer Szene in `Graphicpanel` anzeigen
  2. Gespräche zwischen den Agenten erkennen und das `Graphicpanel` entsprechend aktualisieren. Die Erkennung eines Gespräches zwischen den Agenten wird durch die *ConversationManager* Klasse realisiert. Wenn die `refresh` Methode mit einem Zeitpunkt t aufgerufen wird, wird mit Hilfe der Methode *getConversationsWithTime(t)* abgefragt, ob zu diesem Zeitpunkt ein Gespräch beginnt. Falls ein Gespräch zu diesem Zeitpunkt zwischen den Agenten stattfindet, wird das *Icon* des Agenten, der eine Frage stellt, an die Position des Senders gebracht. Dieses geschieht mit Hilfe der Methode *putOnSenderPosition(Conversation con)*. Die *Icons* der Agenten, die diese Nachricht empfangen, werden mit der Methode *putOnReceiverPosition(Conversation con)* an die Position der Empfänger verschoben werden. Eine Sonderrolle hierbei spielen Gespräche mit dem Umweltagenten. Findet ein Gespräch zwischen einem Agent und dem Umgebungsagenten statt, wird das *Icon* des Agenten, der eine Nachricht an die Umgebung sendet, mit der Methode *putOnMiddlePosition(Conversation con)* in die Mitte des `Graphicpanels` verschoben. Nach Ende des Gespräches werden die *Icons* der Agenten an ihre ursprünglichen Positionen zurückgesetzt.
  3. Unterscheidung zwischen den Fällen, dass ein Agent eine Frage stellt oder eine Antwort sendet. Dieses geschieht mittels der Anzeige eines Fragezeichens bzw.

eines Ausrufezeichens. Zusätzlich wird durch Mundbewegungen signalisiert, dass ein Agent zu diesem Zeitpunkt spricht.

5.9.4 Formatierung der Ausgaben

5.9.4.1 Message

Kommen aus dem *Outputcatcher*. Die *Messages* werden je nach Frage oder Antworttyp als ganzer Satz (Name Agent + sagt: "Gib mir alle Informationen über xy ") und Variablen realisiert.

5.9.4.2 Fragen

In Abschnitt 2.2 wurden die verschiedenen Fragetypen und ihre Kodierung ausführlich erläutert. In diesem Abschnitt werden die Ausgaben zu den logisch kodierten Fragen angegeben, das heißt, wie werden diese Fragen in der GUI angezeigt. Deswegen werden die Beispiele, die in 2.2 dargestellt sind, aufgegriffen und zu jedem Fragetyp die entsprechende Ausgabe (Textform) genannt. In den folgenden Beispielen steht A für die Agenten, die eine Nachricht empfangen.

> **Beispiel 1**. Agent tom möchte eine logische Instanziierungsfrage nach dem Prädikat Mouse/1 stellen. Die Kodierung dieser Frage in einem erweiterten logischen Programm ist
>
> ```
> Query(q1).
> Query_Type(q1, q_instantiate).
> Query_Sender(q1, tom).
> Query_Arity(q1, 1).
> Query_Content(q1, 1, p_mouse).
> ```

Die oben benannte Frage Query(q1) wird in der folgenden Form ausgegeben:

> **Tom asks A, whats assiociated with:** mouse

Die Ausgabe setzt sich zusammen aus:
2. Argument aus $Query\_Sender(q, x)$ + asks + Name der Emfänger + Ausgabe für Instantiierungsfragen + 3. Argument aus $Query\_Content(q, 1, c)$

Fragetyp Elementinfo

> **Beispiel 2**. Agent tom möchte eine logische Beschreibungsfrage über den Agenten jerry stellen. Die Kodierung dieser Frage in einem erweiterten logischen Programm ist

```
Query(q2).
Query_Type(q2, q_element_info).
Query_Sender(q2, tom).
Query_Arity(q2, 1).
Query_Content(q2, 1, jerry).
```

Die oben benannte Frage Query(q2) wird in der folgenden Form ausgegeben:

> **Tom says to A, tell me everything about:** jerry

Die Ausgabe setzt sich zusammen aus:
2. Argument aus $Query\_Sender(q, x)$ + says to + Namen der Empfänger+ Ausgabe für Elementinfo + 3. Argument aus $Query\_Content(q, 1, c)$

Fragetyp Ja-Nein

> **Beispiel 3**. Agent tom möchte eine logische Ja-Nein-Frage über das Faktum Location(jerry, mousehole) stellen. Die Kodierung dieser Frage in einem erweiterten logischen Programm ist

```
Query(q3).
Query_Type(q3, q_yes_no).
Query_Sender(q3, tom).
Query_Arity(q3, 3).
Query_Content(q3, 1, p_location).
Query_Content(q3, 2, jerry).
Query_Content(q3, 3, mousehole).
```

Die oben benannte Frage Query(q3) wird in der folgenden Form ausgegeben:

> **Tom asks A, answer with yes or no: Is there a correct Relation?** location jerry mousehole

Die Ausgabe setzt sich zusammen aus:
2. Argument aus $Query\_Sender(q, x)$ + asks + Namen der Empfänger+ Ausgabe für Ja-Nein-Frage + 3. Argument aus $Query\_Content(q, 1, c)$

5.9.4.3 Antworten

Die logische Kodierung der Antworten wurde auch in Abschnitt 2.2 ausführlich erläutert. In diesem Abschnitt wird angegeben, wie diese Antworten in der GUI dargestellt werden. Die Ausgabe einer Antwort setzt sich zusammen aus:
Name des Senders der Antwort: + Textausgabe für die Antwort + Bezeichner der Frage q + is + content c der Antwort
Bezugnehmend auf die Beispielfragen des vorherigen Abschnitts, könnten folgende Antworten gegeben:

Beispiel 1. Die Frage von Agent tom „Wer ist eine Maus?“, könnte durch Agent jerry folgendermaßen beantwortet werden:

```
Answer(a1, q1, 43).
Answer_Sender(a1, jerry).
Answer_Arity(a1, 1).
Answer_Content(a1, 1, speedy_gonzalez).
```

Die Antwort auf die Frage Query(q1) wird in der folgenden Form ausgegeben:

Jerry: My answer to q1 is speedy_gonzalez

Beispiel 2. Die Frage von Agent tom „Was weißt Du über Jerry?“, könnte durch Agent spike folgendermaßen beantwortet werden:

```
Answer(a2, q2, 44).
Answer_Sender(a2, spike).
Answer_Arity(a2, 3).
Answer_Content(a2, 1, p_likes).
Answer_Content(a2, 2, jerry).
Answer_Content(a2, 3, goldy).
```

Die oben benannte Antwort auf die Frage Query(q2) wird in der folgenden Form ausgegeben:

Spike: My answer to q2 is likes, jerry, goldy

Beispiel 3. Die Frage von Agent tom „Ist Jerry in seinem Mauseloch?“, könnte durch Agent spike folgendermaßen beantwortet werden:

```
Answer(a3, q3, 45).
Answer_Sender(a3, spike).
Answer_Arity(a3, 1).
Answer_Content(a3, 1, a_no).
```

Die Antwort auf die Frage Query(q3) wird in der folgenden Form ausgegeben werden:

Spike: My answer to q3 is no

6 Benutzerhandbuch

6.1 Installation

Adib Dado

In diesem Kapitel werden die Installation und die Konfiguration von KiMAS beschrieben. Die folgenden Voraussetzungen müssen erfüllt werden, damit man KiMAS ausführen kann.

6.1.1 Voraussetzungen

- KiMAS wurde in Java implementiert und ist somit Plattformunabhängig. Für die Ausführung der KiMAS ist die Java Laufzeitumgebung ab Version 1.4 erförderlich.
- Download der InferenzmaschineDLV.exe
- Download Kimas.zip

6.1.2 Kimas starten

- Entpacken die Datei Kimas.zip
- Anpassung der Klassenpfad-Variabel. Der KlassenPfad muss alle Jar-Dateien im Verzeichnis `/kimas/lib` enthalten
- Entpacken der Datei `kimas_0.1.jar` im Verzeichnis `kimas/lib`
- Im Verzeichnis `kimas_0.1` wird die *GUI* mit dem folgendem Aufruf gestartet: `java edu.udo.cs.ie.kimas.gui.GUILauncher`
- Nach dem Start von KiMAS muss der Pfad zur Inferenzmaschine und zum Szenario angegeben werden. Nach Auswahl des Menüpunktes *Preferences* im Menü *File*, muss der Pfad zur dlv.exe bzw. zur Kimas.Scenrios.xml im entsprechenden Textfeld angegeben werden.

6.2 Quickstart

Özlem Sentürk

Nach dem Start des Programms muss zunächst ein Szenario, welches in Form der XML-Datei im System vorliegt und die Eigenschaften des Agenten, die in dem Szenario enthalten sind, spezifiziert, geladen werden. Klicken Sie dazu in der Menüleiste auf *File*, dann

wählen Sie den Untermenüpunkt *Open* aus (siehe Abbildung 6.1). Folgen Sie den zu einem *Szenario Definition File* (SDF) führenden Pfad (siehe Abbildung(6.2)). Laden Sie ein *Szenario Definition File* (SDF), welches in dem XML-Dateiformat als `name.scenario.xml` im Verzeichnis vorliegt (siehe Abbildung 6.3).

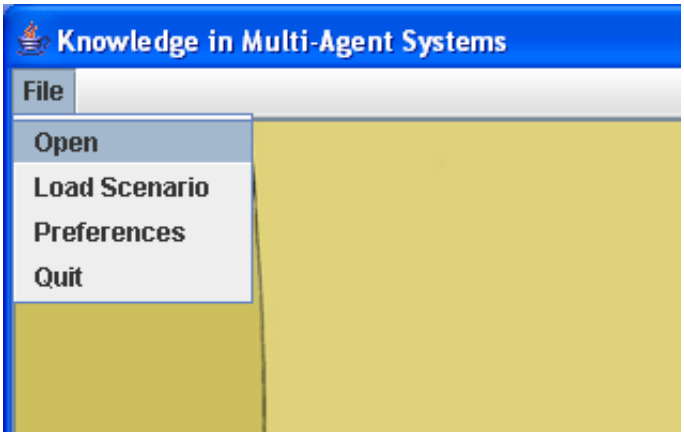

Abbildung 6.1: Open

Nachdem das Szenario geladen wurde, werden die *Szenencombobox* mit den einzelnen Szenen, aus denen das geladene Szenario besteht, in der festgelegten Reihenfolge, die *Agentencomboboxen* mit den Agenten, die in dem geladenen Szenario enthalten sind, und die *Agentstatecomboboxen* mit den Ausgabetypen, wie *InitialBelief*, *Beliefstate*, *Knowhow* etc., gefüllt.

Sie können eine beliebige Szene, die in die Szenencombobox *Scenes* angezeigt sind, auswählen und die gewählte Szene ablaufen lassen, indem Sie den *Play-Button* betätigen.

Durch Anklicken der Checkboxen, welche mit den Logtypen *message*, *info* und *debug* dargestellt sind, werden beim Ablauf der Szene die Ausgaben von dem ausgewählten Logtyp in dem Textfeld links unten angezeigt (siehe Abbildung 6.4).

Sie können eine Szene auch von einem frei wählbaren Zeitpunkt aus starten. Klicken Sie dazu auf den *Stop-* bzw. *Pause-Button*. Der Szenarioablauf wird dadurch angehalten und das Textfeld, in dem der aktuelle Zeitpunkt zu sehen ist, wird dadurch editierbar. Sie können einen beliebigen Zeitpunkt eingeben. Durch erneutes Klicken des *Play-Buttons* wird die Szene von diesem Zeitpunkt aus gestartet (bzw. fortgesetzt).

In den Agentencomboboxen *Agent* werden die Agenten, die im geladenen Szenario spezifiziert sind, angezeigt. Sie können Agenten aus den Agentencomboboxen *Agent* (siehe Abbildung 6.5) und Ausgabetypen aus den Agentstatecomboboxen *Agentstate* (siehe Abbildung 6.6) auswählen. Darüber hinaus können Sie während des Ablaufs einer Szene die Wissensänderung des ausgewählten Agenten in jedem der vier Textfeldern unten rechts beobachten.

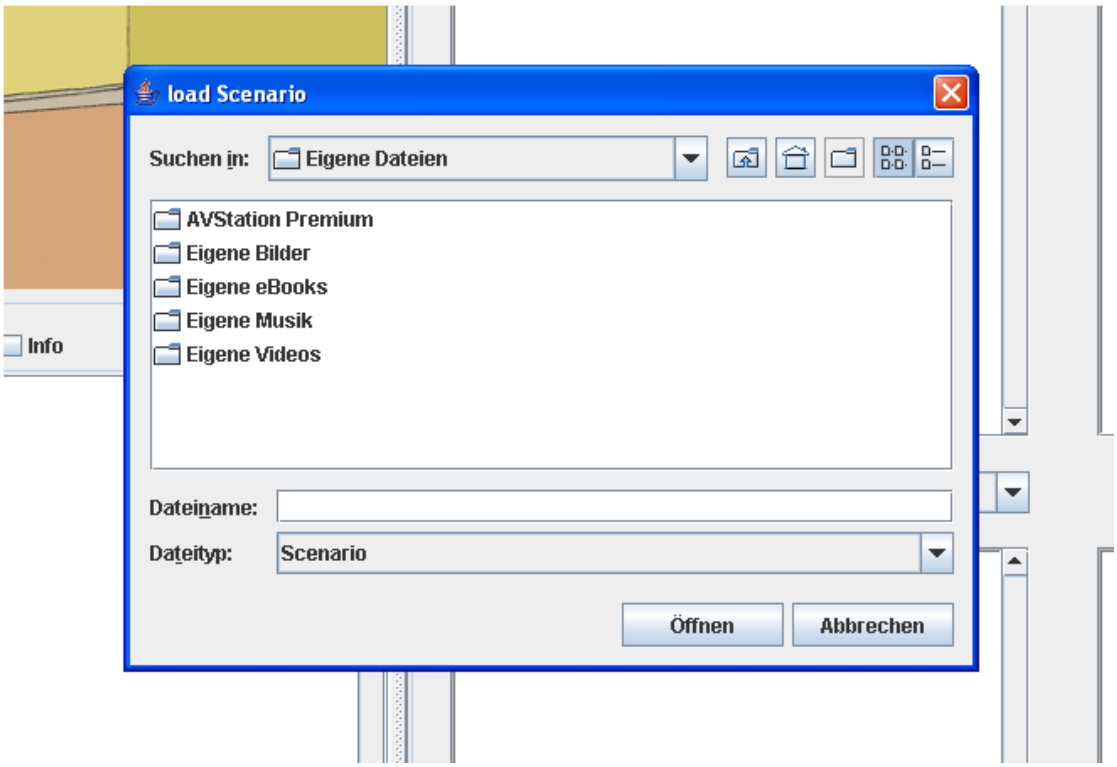

Abbildung 6.2: Pfad zur SDF-Datei

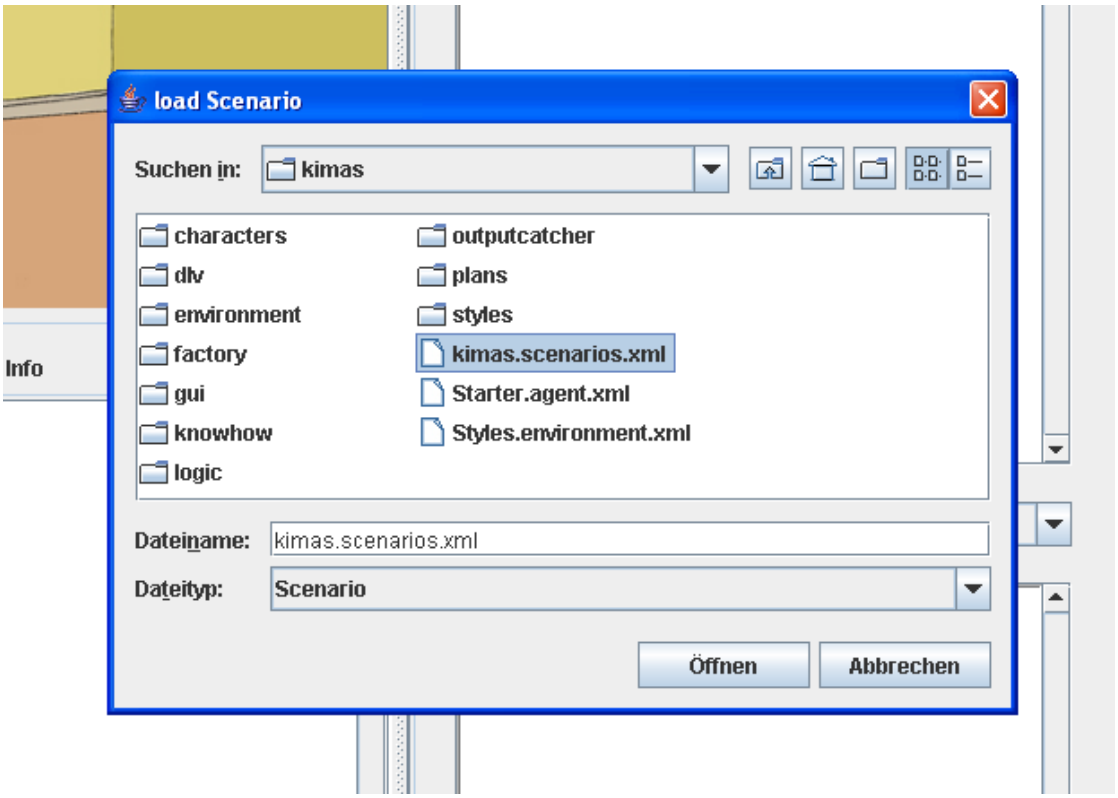

Abbildung 6.3: Laden der SDF-Datei

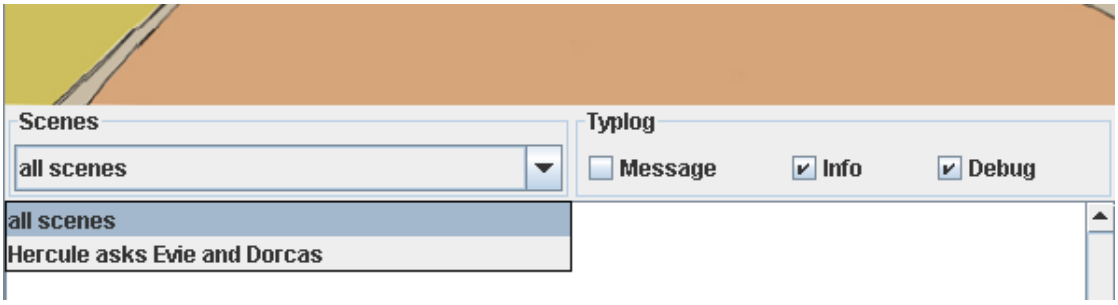

Abbildung 6.4: Wählen einer Szene und des Logtyps

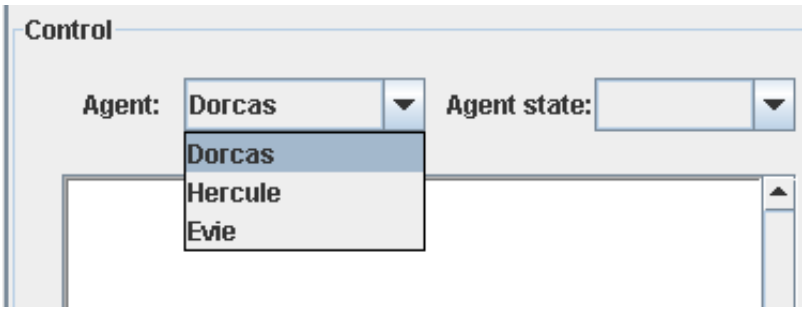

Abbildung 6.5: Wählen eines Agent

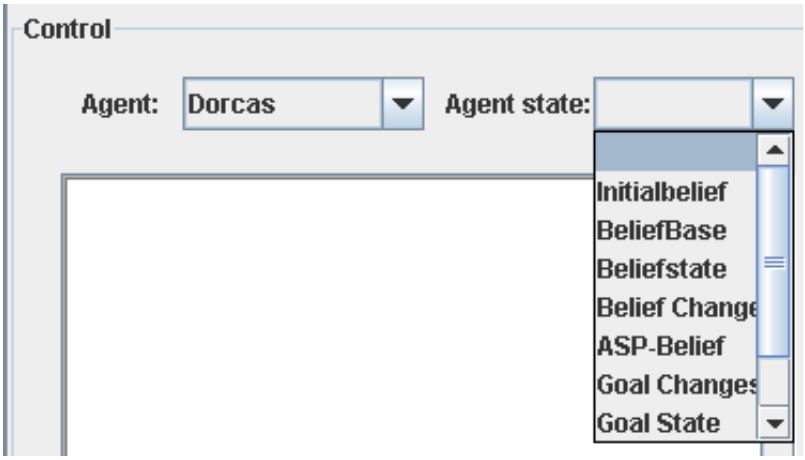

Abbildung 6.6: Wählen eines Ausgabetypes

6.3 Entwicklung eigener Szenarien

Özlem Sentürk

In diesem Kapitel werden die Erstellungen des SDFs, EDFs und CDFs sowie die Installation des *KiMAS-Editor*s detaillierter beschrieben. Es gibt zwei Möglichkeiten, um SDFs, EDFs und CDFs zu erstellen. Entweder Sie öffnen jeweils eine XML-Datei mit einem geeigneten Texteditor und erstellen die Dateien von Hand oder Sie benutzen den *Kimas–Editor*.

Kimas–Editor ist ein Plugin für die Entwicklungsumgebung *Eclipse*, welches in dieser Projektarbeit entwickelt wurde, um damit eigene Szenarien, Umwelt- und Charakteragenten erzeugen zu können. Bevor Sie das Plugin *Kimas–Editor* installieren, stellen Sie sich sicher, dass die Plugins *Eclipse Modeling Framework* (EMF) und *Graphical Editing Framework* (GEF) in Eclipse installiert sind. In dem Fall, dass die Plugins *EMF* und *GEF* nicht installiert sind, installieren Sie diese, bevor Sie *Kimas–Editor* installieren.

Führen Sie die folgende Schritte aus, um das Plugin *Kimas–Editor* in Eclipse zu installieren.

1. Klicken Sie in der Menüleiste auf *Help* in Eclipse. Wählen Sie *Software Updates* im geöffneten Fenster. Folgen Sie dem Pfeil und klicken Sie *Find and Install*. In dem geöffneten Fenster haben Sie die Möglichkeit, neue Plugins zu installieren und die Plugins, die schon installiert sind, zu aktualisieren. Da Sie ein Plugin neu installieren, klicken Sie *Search for new features to install* an und dann auf *Next* um fortzufahren.

2. Wählen Sie *New Remote Site* in dem neu geöffneten Fenster.

3. Darüber hinaus wird ein Fenster mit den zwei Feldern geöffnet. Füllen Sie die Felder *Name* (z. B. Kimas Update Site) und *URL* aus. Die URL lautet `http://kimas.sourceforge.net/editor/updates/`. Klicken sie nach der Eingabe auf *OK* und bestätigen Sie nochmals mit *Finish*.

4. Wählen Sie eine *update site* auf dem geöffneten Fenster aus.

5. Wählen Sie *Kimas Update Site* $\mapsto$ *Editor* aus. Danach klicken Sie auf *Next*, um mit der Installation fortzufahren.

6. Es werden Features gefunden. Um die Features zu installieren, müssen Sie die Lizenz akzeptieren. Lesen Sie den Lizenzvertrag aufmerksam durch. Wenn Sie den Lizenzvereinbarungen zustimmen, klicken Sie auf *I accept the terms in the license aggreements*.

7. Bei der Verifikation klicken Sie auf *Install all*.

8. Starten Sie Eclipse neu, damit alle neue Einstellungen wirksam werden.

9. Vergewissern Sie sich, dass die Installation erfolgreich durchgeführt wurde. Wählen Sie *Help* aus. Klicken Sie auf *About Eclipse SDK*. Um alle Plugins, die in Eclipse

installiert sind, zu sehen, klicken Sie auf *Plugin Details*. Wenn die Installation erfolgreich durchgeführt wurde, müssen nun drei Einträge mit *kimas.sourceforge.net* in der Plugintabelle vorhanden sein.

Sie können jetzt mit dem *Kimas–Editor* arbeiten. Klicken Sie auf *File*, dann *New*. Wählen Sie *Other* aus. In dem geöffneten Fenster wählen Sie *Kimas Model* unter *Kimas*. Geben Sie einen Namen für die erzeugte Datei ein. Beachten Sie, dass die Endung für die Datei *.kimas sein soll. Klicken Sie auf *Next*, um fortzufahren. Wählen Sie ein *Model Object* und einen Zeichensatz, mit welchem das Dokument kodiert werden soll, aus. Der Zeichensatz UTF-8 ist als Standardzeichensatz gesetzt. Bestätigen Sie mit *Finish*.

In den folgenden Abschnitten wird detailliert erklärt, wie Sie CDF, EDF und SDF einzeln erstellen können.

6.3.1 Erstellen des CDFs

Bei *Charakter Definition File* (CDF) handelt es sich um eine XML-Datei, welche in System mit `name.charakter.xml` bezeichnet wird. In einem *Charakter Definition File* wird ein Agent, welcher aktiv an einer oder mehreren Szenen teilnimmt, spezifiziert. Die Eigenschaften des Agents, die ihn auszeichnen, werden sowie dessen beliefs, goals, plans und realibilities in CDF abgelegt.

Ein Agent besitzt zwei wichtige Eigenschaften, einen *Namen* und ein grafisches *Icon*. Ein Agent kann ein oder mehrere Icons besitzen. Innerhalb der `<icons>`-Sektion werden durch das Attribut *location* die Pfade angegeben, die zu den Icons gelangen. Die Abfolge der Icons, die in der Implementierungen der Animationen eine wichtige Rolle spielen, werden dort durch das Attribut *seq* gesetzt.

```
<character name="poirot">
  .........
  <icons>
    <icon location="pfad/zu/poirot1.gif" seq="0"/>
    <icon location="pfad/zu/poirot2.gif" seq="1"/>
  </icons>
```

Jeder Agent verfügt über seine eigene *beliefs*, *goals* und *plans*. Um das Wissen des Agenten über seine Umwelt und sich selbst darzustellen, werden *beliefs* verwendet. Die `<beliefs>`-Sektion besteht aus vier Untersektionen *logic*, *singlebeliefs*, *multibeliefs* und *knowhow*. Die `<logic>`-Sektion besteht aus der *predicates*, *facts* und *rules*, welche das individuelle Fakten- und Regelwissen des Agenten repräsentieren. Dieses Fakten- und Regelwissen des Agenten bildet den Initialbelief des Agenten.

Innerhalb der `<predicates>`-Sektion werden die einzelne Prädikate durch das Element `<predicate>` definiert, die dem Agenten bekannt sind. Die Prädikate sind mit zwei weiteren Attributen *name* und *arity* spezifiziert. Mithilfe des Attributes *arity* wird die Stelligkeit des Prädikates angegeben. Alle Prädikate, die innerhalb von Fakt und Regel vorkommen, müssen innerhalb der `<predicates>`-Sektion definiert werden. Wenn ein

nicht definiertes Prädikat in irgendeinem Fakt oder irgendeiner Regel vorkommt, wird es in der Logik nicht berücksichtigt.

Beispiel: Die folgende Prädikate werden in XML-Datei folgendermaßen kodiert und für den Agent bereitgestellt.

- **RoomOfCrime(L)** das Prädikat *RoomOfCrime* hat einen Parameter *L*. *L* ist der Tatort. Es handelt sich um den Raum, in dem das Opfer ums Leben kam.
- **Sympathy(A,B,V)** das Prädikat *Sympathy* hat drei Parameter. *V* ist der Wert der Sympathie, welche die Person *A* gegenüber Person *B* empfindet. *V* liegt dabei zwischen *0* (tiefster Hass) und *10* (innige Liebe).

```
<beliefs>
  <logic>
    <predicates>
      <predicate arity = "1" name = "RoomOfCrime"/>
      <predicate arity = "3" name = "Sympathy"/>
    </predicates>
    .......
```

Die in der `<facts>`-Sektion spezifizierten Prädikate werden in der `<facts>`-Sektion benutzt, um Faktenwissen des Agenten darzustellen. Das `<facts>`-Element enthält die Elemente vom Typ `<instance>` mit den zwei möglichen Attributen *name* und *value*. Das Attribut *name* bezeichnet den Name des Prädikates und das Attribut *value* den initialen Wert.

```
<facts>
  <instance name = "RoomOfCrime" value = "emilysRoom"/>
  <instance name = "Sympathy" value = "poirot alfred 5"/>
  <instance name = "Sympathy" value = "poirot emily 5"/>
  <instance name = "Sympathy" value = "poirot evie 10"/>
</facts>
```

Die `<rules>`-Sektion besteht aus einer Menge des Elements `<rule>`. Das Element `<rule>` beinhaltet die Elemente `<head>` und `<body>`. Der Kopf der Regel werden innerhalb von `<head>` und der Rumpf der Regel innerhalb von `<body>` angegeben, indem die `<instance>`-Elemente benutzt wird. Die logischen Regeln des Agenten werden hier aufgebaut.

Beispiel: Die logische Regel `ToHide(Item,Scene)` wird durch die unten definierte Prädikate `Proof(item)`, `CurrentScene(Scene)` und `RoomOfCrime(Place)` erzeugt und in XML-Datei folgendermaßen kodiert:

```
ToHide(Item,Scene) :- Proof(Item), CurrentScene(Scene),
    RoomOfCrime(Place)
```

ToHide(Item, Scene) Der Gegenstand *Item* muß versteckt werden.

Proof(X) Bei X handelt es sich um einen Beweis.

CurrentScene(Scene) Bei *Scene* handelt es sich um die aktuelle Szene.

RoomOfCrime(Place) *Place* ist der Tatort. Es handelt es sich hierbei um den Raum, in dem das Opfer ums Leben kam.

```
<rules>
  <rule>
    <head>
      <instance name = "ToHide" value = "Item␣Scene"/>
    </head>
    <body>
      <instance name = "Proof" value = "Item"/>
      <instance name = "CurrentScene" value = "Scene"/>
      <instance name = "RoomOfCrime" value = "Place"/>
    </body>
  </rule>
 </rules>
```

Die `<singlebeliefs>`-Sektion besteht aus Elementen vom Typ `<singlebelief>`. Ein solcher `<singlebelief>` ist der *beliefOperator*. Durch das Attribut *class* wird die JAVA-Klasse festgelegt, welche verantwortlich ist, neues Wissen des Agents zu verarbeiten.

```
<singlebeliefs>
  <singlebelief class="paketname.BeliefOperator"
                      name="beliefOperator">
    <fact> new edu.udo.cs.ie.kimas.
                logic.operators.UpdateOperator2()
    </fact>
  </singlebelief>
</singlebeliefs>
```

Die *goals* and *actions* des Agenten werden innerhalb der `<knowhow>`-Sektion spezifiert. Diese bestimmen, welche Ziele der Agent hat und welche Aktionen der Agent durchführen kann. Goals können entweder am Beginn der Szene ausgeführt oder durch logische Fakten getriggert werden. Die logischen Fakten werden als Bedingung interpretiert. Erst wenn die Bedingung erfüllt ist, wird das *goal* getriggert. Darüber hinaus wird die zugehörige Aktion aufgerufen.

Die Bedingung wird durch das Element `<condition>` innerhalb der `<goals>`-Sektion festgelegt. Innerhalb der `<subelementsets>`-Sektion werden Abfolgen von Element *doaction* gesetzt, welche für das angegebene *goal* vordefiniert ist.

```
<goals>
  <goal name="HideProof">
    <triggers>
      <trigger>
        <condition>DoHide(From To Item)</condition>
      </trigger>
    </triggers>
    <subelementsets>
      <subelementset>
        <doaction name="EnvRequest">
          <parameter name="addressee">
            <value>Environment</value>
          </parameter>
          <parameter name="content" constant="false">
            <for variable="Item">DoHide(From To Item)</for>
            <for variable="From">DoHide(From To Item)</for>
            <for variable="To">DoHide(From To Item)</for>
            <value>EnvMove(Item,From,To)</value>
          </parameter>
        </doaction>
      </subelementset>
    </subelementsets>
  </goal>
</goals>
```

Jede *action* ist mit einem *goal* verknüpft. Diese *actions* werden ausgeführt, um das zugehörige goal zu erfüllen. Für jede action existiert ein JADEX-Plan. Jedes mal wenn eine action durchgeführt wird, wird die zugehörige JADEX-Plan aufgerufen.

```
<actions>
  <action name="EnvRequest">
    <parameters>
      <parameter variable="W" name="addressee"></parameter>
      <parameter variable="Z" name="content"></parameter>
    </parameters>
    <operation>new SendEnvRequestPlan()</operation>
  </action>
</actions>
```

Ein Agent hat die Fähigkeit, neues Wissen zu verarbeiten. Dafür benutzt der Agent einen *BeliefOperator*, der eine JAVA-Klasse ist. Manche BeliefOperators benötigen dafür die Vertrauenswürdigkeiten der anderen Agenten. Diese können in der `<reliabilities>`-Sektion festgelegt werden.

```
<reliabilities>
  <reliability value="100" agent="Environment"/>
  <reliability value="70" agent="hastings"/>
  <reliability value="70" agent="poirot"/>
</reliabilities>
```

6.3.2 Erstellen des EDFs

Ein Umweltagent, der die relevanten Informationen über die Szenen liefert, wird in einem *Environment Definition File* (EDF), welche in Form einer XML-Datei ist, repräsentiert und mit `name.environment.xml` als *EDF* kenntlich gemacht. Seine Aufgabe besteht darin, die Eigenschaften der Umwelt zu repräsentieren. Alle Gegenstände, die in einem Szenario vorkommen, werden mit deren Eigenschaften im EDF spezifiziert. Außerdem werden die einzelne Szenen, aus denen ein spezielles Szenario besteht, im EDF durch die Angabe der bestimmten Attributen zugeordnet und deren Eigenschaften dargestellt.

Im allgemeinen setzt sich ein EDF aus zwei möglichen Sektionen `<locations>` und `<scenes>` zusammen.

In der `<locations>`-Sektion werden die Gegenstände, deren Eigenschaften und Beziehungen untereinander durch das Element `<thing>` und die noch weitere Elemente und deren Attribute spezifiziert. Der Name des Ortes, wo sich die Gegenstände befinden, soll zunächst festgelegt werden. Das geschieht durch das Attribut *name* des Elementes `<location>`. Der Name der in dem Ort vorhandenen Gegenstände werden durch die Angabe des Attributes *name* und deren Typen mittels des Attributes *type* in EDF angegeben. In einigen Fällen ist es notwendig, dass manche Gegenstände nicht für alle Agenten ersichtlich sein sollen. Die in dieser eingestuften Gegenstände müssen mit dem Attribut *obvious* in EDF versehen werden. Die Gegenstände, für die *true* gesetzt wurde, werden zum Szenebeginn allen am laufenden Szene beteiligten Agenten mitgeteilt.

Die weitere Eigenschaften, die das Gegenstand auch auszeichnen, werden innerhalb der `<attributes>`-Sektion und `<contains>`-Sektion bekanntgemacht.

In der `<attributes>`-Sektion werden die weitere Eigenschaften eines Gegenstandes mithilfe des Elements `<static>` in EDF angegeben. Durch das Attribut *obvious* des Elements `<static>` werden die *sichtbaren* und *unsichtbaren* Eigenschaften eines Gegenstandes festgelegt.

Durch die Verwendung des Element `<madeof>` wird für den Gegenstand festgelegt, woraus er hergestellt wurde.

```
<locations>
  <location name="emilysRoom">
    <thing name="clodOfEarth" type="dirt">
    </thing>
    <thing name="burntPaper" obvious="false" type="paper">
      <attributes>
        <static obvious="true">burnt</static>
```

```
        <static obvious="false">ment</static>
      </attributes>
      <madeof>thickPaper</madeof>
    </thing>
  </location>
</locations>
```

Ein Gegenstand kann weitere Gegenstände beinhalten. In diesem Fall wird in der `<contains>`-Sektion durch das `<initial>`-Element spezifiziert, welches Objekt sich in diesem befindet.

```
<location name="styles">
  <thing name="coffee" type="coffee" obvious="false">
    <contains>
      <initial>sugar</initial>
    </contains>
  </thing>
</location>
```

Es besteht die Möglichkeit, die Eigenschaften eines Gegenstandes zu manipulieren. Der Initialzustand eines Gegenstandes, der mithilfe des Attributes *initial* des Elementes `<changeable>` angegeben wird, wird durch die Ausführung der Aktion, welche innerhalb von `<action>` vordefiniert liegt, verändert. Es wird auch vorgegeben werden, welchen Zustand der Gegenstand nach der durchgeführten Aktion aufnehmen soll. Es könnte jedoch nicht ausreichen, nur den Initialzustand eines Gegenstandes und dessen Zustand nach der Aktion zu betrachten, da zwei zusammenhängende Aktionen existieren können. Darüberhin muss der Zustand eines Gegenstandes vor der Aktion auch festgestellt werden.

Wenn der Gegenstand irgendwelchen Inhalt, welcher in der `<contains>`-Sektion durch das Element `<initial>` gesetzt wird, besitzt, ist es möglich, den Inhalt des Gegenstandes zu entfernen.

```
<locations>
  <location name="emilysRoom">
    <thing name="emilysBriefcase" type="briefcase">
      <attributes>
        <static>emily</static>
        <changeable initial="closed" obvious="true">
          <changedef>
            <action>open</action>
            <before>closed</before>
            <after>opened</after>
          </changedef>
```

```
            <changedef>
              <action>close</action>
              <before>opened</before>
              <after>closed</after>
            </changedef>
          </changeable>
        </attributes>
        <initial>confessionLetter</initial>
          <removable>confessionLetter</removable>
        </contains>
      </thing>
    </location>
</locations>
```

Es ist auch realisierbar, ein beliebiges Objekt zu einem Gegenstand durch das Element `<addable>` hinzuzufügen.

```
<location name="emilysRoom">
  <thing name="vases" type="container">
    <contains>
      <addable>confessionLetter</addable>
    </contains>
  </thing>
</location>
```

Die andere grundlegende Sektion des EDFs ist wie schon erwähnt die `<scenes>`-Sektion. Innerhalb der `<scenes>`-Sektion werden die einzelne Szenen, aus denen das Szenario besteht, aufgelistet und deren Startordnung durch das Attribut *order*, deren Name durch das Attribut *name* und deren Ort, wo der Szene stattfindet, durch das Attibut *location* spezifiziert. Die an der Szene *nicht* beteiligten Agenten werden durch das Element `<sleepy>` angegeben.

```
<scenes>
  <scene order="1" name="CSIStyles" location="emilysRoom">
    <sleepy>Alfred</sleepy>
    <sleepy>Poirot</sleepy>
  </scene>
  <scene order="2" name="Alfred␣hides␣proofs"
                                    location="emilysRoom">
    <sleepy>Poirot</sleepy>
    <sleepy>Hastings</sleepy>
  </scene>
</scenes>
```

6.3.3 Erstellen des SDFs

Bei *Scenario Definition File* (SDF) handelt sich um eine spezielle XML-Datei, welche im System mit einem Dateinamen der Art `name.scenario.xml` vorliegen muss. Ein Szenario, welches aus einer oder mehreren Szenen besteht, wird in einem *Scenario Definition File* (SDF) repräsentiert. Im SDF werden alle generellen Informationen der Agenten, die im Szenario enthalten sind, abgelegt.

Dort wird zusätzlich auch spezifiziert, was mit schon gestarteten Agenten zum Startzeitpunkt des Szenarios geschehen soll. Durch die Angabe des Elements `<kill_ and_ restart value="true"/>` werden festgelegt, dass die beim Start des Szenarios schon laufende Agenten getötet und neu gestartet werden sollen. Wenn es für das Attribut *false* gesetzt wird, werden die bereits laufende Agenten nicht berücksichtigt.

Ein *Environment Definition File* (EDF), welches die Form einer XML-Datei hat und die szenariospezifische Informationen enthält, werden durch das Element `<edflocation>` angegeben.

Innerhalb der `<agents>`-Sektion wird angegeben, wo sich die generischen *Agent Definition File* (ADF) der Charakter- und des Umweltagentes befinden. Die Agenten werden durch das Element `<agent>` mit dessen Attibut *name* und *cdf* in SDF angegeben. Durch dessen Attribut *cdf* werden die Datei, die die Eigenschaften des Agenten spezifiziert, und durch dessen Attribut *name* einen Name für den Agent gesetzt.

```
<kill_ and_ restart value="true"/>
<edflocation> testscenario.environment.xml</edflocation>
<agents>
  <character>paketname.Character</character>
  <environment>paketname.Environment</environment>
  <agent cdf="Dorcas.character.xml" name="Dorcas"/>
  <agent cdf="Evie.character.xml" name="Evie"/>
  <agent cdf="Hercule.character.xml" name="Hercule"/>
</agents>
```

In der `<agentgroups>`-Sektion können Agenten durch das Element `<agentgroup>` in den beliebig bestimmten Gruppen zugeordnet werden. Die Agenten, die in derselben Gruppe zugeordnet sind, werden durch das Element `<agentref>` mit dessen Attribut *name* in SDF referenziert.

```
<agentgroups>
  <agentgroup name="suspects">
    <agentref name="Dorcas"/>
    <agentref name="Evie"/>
  </agentgroup>
</agentgroups>
```

Der Name des Szenarios und die Agenten, die an diesem Szenario beteiligen, mit deren Startreihenfolge werden innerhalb der `<scenario>`-Sektion angegeben. Die zu einer Gruppe zusammengefasste Agenten werden nach deren Startreihenfolge durch das Element `<before>` und `<after>` in SDF referenziert. Wie in dem folgenden Beispiel zu sehen ist, wird erst die Agenten, die als *suspects* referenziert sind, gestartet, dann der Agent "Hercule".

```
<scenario name="TheMysteriousffairAtStyles">
  <order>
    <before>
      <agentgroupref name="suspects"/>
    </before>
    <after>
      <agentref name="Hercule"/>
    </after>
  </order>
</scenario>
```

Wenn es nicht relevant ist, zu welchem Zeitpunkt die Agenten gestartet werden, werden die Agenten durch das Element `<agentref>` mit dem Attribut *name* innerhalb von `<startanytime>` referenziert.

```
<scenario name="TheMysteriousffairAtStyles">
  <startanytime>
    <agentref name="Poirot"/>
    <agentref name="Alfred"/>
    <agentref name="Hastings"/>
  </startanytime>
</scenario>
```

Die Fakten- und Regelwissen, die am Szenario beteiligten Agenten besitzen, werden in der `<beliefs>`-Sektion des SDFs spezifiziert. Die Prädikate, die jedem Agenten bekannt sein sollen, werden durch das Element `<predicate>` mit dessen Attribut *arity* und *name* innerhalb der `<predicates>`-Sektion definiert.

```
<predicates>
  <predicate arity="1" name="Action"/>
  <predicate arity="2" name="ActionPlace"/>
</predicates>
```

Die Regeln, die in diesem Szenario enthaltene Agenten besitzen, werden in der `<rules>`-Sektion aufgebaut. Ein Regel wird durch *head* und *body* zusammengesetzt. Innerhalb von

`<head>` werden der Kopf und innerhalb von `<body>` der Rumpf der Regel angegeben, indem die `<instance>`-Elemente benutzt wird.

Beispiel: Bei der Entwicklung der folgenden logischen Regel `SceneRunning(Scene) :- Action(A), ActionPlace(Activity, Place)` werden zwei unten definierte Prädikate zusammengesetzt und in XML-Datei folgendermaßen kodiert.

SceneRunning(Scene) *Scene* ist die Szene die aktuell läuft.

Action(A) Bei *A* handelt es sich um eine Tätigkeit. Diese Aktion kann durch andere Prädikate näher beschrieben werden.

ActionPlace(Activity,Place) Dieses Prädikat beschreibt die Tätigkeit *Activity* näher, welche am Ort *Place* stattfindet.

```
<rules>
  <rule>
    <head>
      <instance name="SceneRunning" value="X"/>
    </head>
    <body>
     <instance name="Action" value="subscribe"/>
     <instance name="ActionPlace"
                    value="doNewBedOfFlowers_garden"/>
    </body>
  </rule>
</rules>
```

Wenn die Prädikate in Regeln bzw. Fakten vorkommen, die in der `<predicates>`-Sektion nicht definiert sind, werden diese Prädikate in der Logik ignoriert. Die Fakten werden gelöscht und Regeln entsprechend gekürzt.

6.4 Bedienung der graphischen Benutzeroberfläche

Adib Dado, Dennis Rüther

Die GUI von KIMAS besteht aus einer ganzen Reihe von miteinander verknüpften Komponenten. Die Ergonomie dieser Benutzeroberfläche ist bewußt an die Bedienstrukturen gängiger *Media-Player* angelehnt. Um ein Szenario verfolgen zu können, müssen drei Schritte sequentiell befolgt werden. Als erstes muss ein Szenario, ein eigenes oder ein von der PG491 entwickeltes, geladen werden. Schließlich beginnt die eigentliche Bedienung der Software. Es ist eine Tatsache, dass das Verständnis für das inhalliche Geschehen der GUI in den Kapiteln 2 bis 5 erworben werden muss und nicht in dem Zusammenhang der Bedienung der GUI erläutert wird. Da es sich um eine wissenschaftliches Projekt einer Informatikfakultät handelt, müssen Konzepte wie zum Beispiel *Inferenz*, *Wissensbasis*

oder *Anwortmengenprogrammierung* vorrausgesetzt werden, um ein adequates Arbeiten zu ermöglichen. Laien dieses Gebietes sei KIMAS als spielerische Möglichkeit an die Hand gegeben, logische Wissensmodellierung und Verarbeitung in praktischer Anwendung zu beobachten.

6.4.1 Starten der GUI

Der *Quickstart* der GUI wird im Abschnitt 6.2 behandelt.

6.4.2 Überblick über die GUI

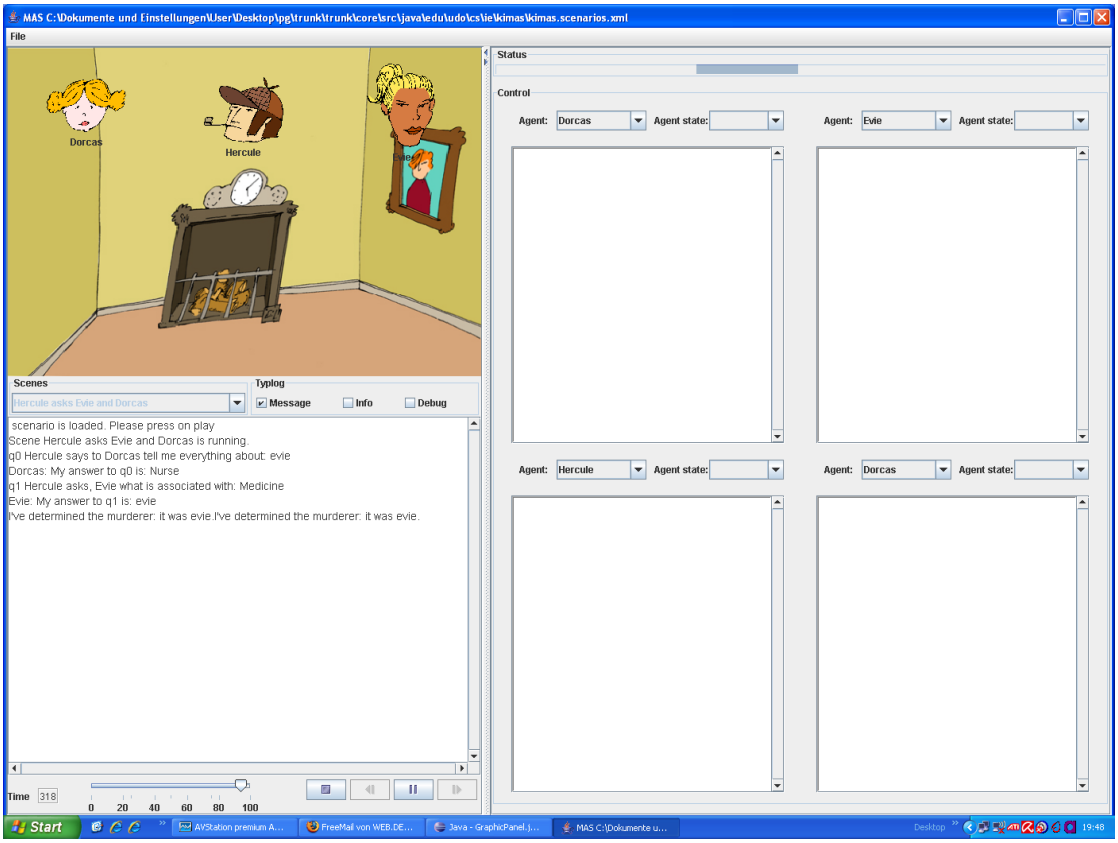

Abbildung 6.7: KIMAS im Überblick

Die GUI von KIMAS besteht aus einem Fenster. Lediglich das Laden von Dateien und die Angabe von Voreinstellungen, durch den Menüpunkt "*Preferences*", öffnen weitere Fenster. Allerdings sind diese Fenster nur während des Aufrufs aktuell und werden danach sofort geschlossen. Somit bleibt ein Fenster in dem alle Aktivitäten und Veränderungen eines Multiagentensystems zu beobachten sind. Man kann die Oberfläche in zwei Hälften unterteilen, wobei die linke Seite eher zur Darstellung von Kommunikation der Agenten dient, während die rechte Seite das *Agenteninnenleben* zeigt.

Die linke Seite der Abbildung zeigt, dass das Design dieser Oberfläche bewußt einem *Mediaplayer* nachempfunden ist. So existieren die gängigen Schaltknöpfe wie "*Forward*", "*Backward*", "*Play*", "*Pause*" und "*Stop*", die links unten in der Bedienleiste positioniert sind. Direkt neben den Schaltknöpfen befindet sich ein Schieberegler, mit dem das Tempo der Ausgabe reguliert werden kann. Da Wissenszustände von Agenten in diskreten Schritten dargestellt wird, befindet sich wiederum links vom Schieberegler der Zähler, der den aktuellen Zeitzustand i als ganze Zahl angibt. Initialisiert wird der Zähler mit null. Die Zeitzustände können über die Schaltknöpfe manipuliert werden. Über den Schaltknöpfen befindet sich ein *Panel*, namens *Dialogdisplay*, mit zwei weiteren Bedienelementen. Das linke Element dient zur Auswahl der Szenen, innerhalb eines Gesamtszenarios, und ist eine *Combobox*. Das von der PROJEKTGRUPPE 491 entwickelte Szenario ist ein Krimi und ist in mehrere Szenen unterteilt. Es ist zum Verständnis der Bedienung wichtig, dass alle Änderungen in diesem Element, Änderungen in fast allen Komponenten von KIMAS nach sich ziehen. Die rechte Komponente steuert die Ausgabe des unter ihm angelegten *Dialogdisplays*, durch Auswahl der *Checkboxitems*. Noch oberhalb des *Dialogdisplays* liegt das so genannte *Grafikdisplay*, in dem die Kommunikation der Agenten untereinander grafisch simuliert wird. Die rechte Seite der Oberfläche wird durch vier Felder dominiert, in denen das Agenteninnenleben angezeigt wird. Zu jedem Ausgabefeld gehören zwei *Comboboxen*, die eine Auswahl der Agenten und des jeweiligen Wissenstyps ermöglichen. Oberhalb dieser vier Komponenten befindet sich ein Fortschrittsbalken der den Fortschritt der einzelnen Szenen nach dem Starten eines Gesamtszenarios anzeigt.

Viele Funktionen der Oberfläche hängen vom Zustand des Programms ab. Dabei existieren nach dem Öffnen der Oberfläche drei Zustände:

- Die Oberfläche ist geladen, aber kein Szenario. Dementsprechend ist keine der Komponenten außerhalb des Menüs aktiv.
- Ein Szenario ist geladen. Die Inferenz ist gestartet, und läuft im Hintergrund. Die Anzeige des Szenarios durch "Start" ist noch nicht begonnen worden.
- Das Szenario ist gestartet. Wissenveränderung findet statt und die Agenten kommunizieren. Alle Komponenten sind aktiv bis auf "*Play*".

6.4.2.1 Menü

Über das Menü können übergeordnete Funktionen, wie das Einladen von Szenarien, Einstellungen bearbeiten oder das Schließen der Anwendung vorgenommen werden. Sollte das Ziel bestehen eigene Szenarien zu produzieren, ist zu beachten, dass der Umgang mit eigenen Szenarien ein gewisses Vorwissen über die Zusammenarbeit von JADEX , DLV und eigenen Szenarien bedarf, siehe Kapitel 6.3.

Der Menüpunkt "*Open*" öffnet einen *Filechooser* über den ein Pfad zu einem Kimasszenario geöffnet werden kann. Der Pfad wird in die Datei `kimas.Properties` eingeschrieben und JADEX wird auf dem gewählten Szenario gestartet.

Der Menüpunkt "*LoadScenario*" lädt das zuletzt in die Datei `kimas.Properties` geschriebene Szenario. Dieses wird geöffnet bzw. JADEX wird auf ihm gestartet.

Grundsätzliche Einstellungen wie DLV-Pfade und das Einstellen von Defaultszenarien in `kimas.Properties` können über den Menüpunkt *Preferences* vorgenommen werden.

Die komplette Anwendung kann über *Quit* geschlossen werden. Allerdings nicht bei laufendem Szenario.

6.4.2.2 Fortschrittsbalken

Der Fortschrittsbalken wird erst nach dem Einladen eines beliebigen Szenarios über "*LoadScenario*" oder "*Open*" und dem Start durch Drücken des Startknopfes aktiv. Dann zeigt er den Fortschritt von einzelnen Szenarien auf dem gewählten Gesamtszenario an. Steht er bei 100% ist das Gesamtszenario durchgelaufen. Nach Durchlauf eines von drei Szenarien, wäre die Anzeige bei 33%.

6.4.2.3 Grafikdisplay

Das Grafikdisplay wird erstmals nach dem Einladen eines beliebigen Szenarios über "*LoadScenario*" oder "*Open*" aktiv. Dann zeigt es die aktiven Agenten des gewählten Szenarios an, indem es sie am oberen Rand des Grafikdisplays anordnet. Nach dem Starten durch "Start" wird die Kommunikation der Agenten untereinander gestartet. Sprechende Agenten werden in die Mitte des Fensters gerückt und ihr Fragen und Antworten werden symbolisch durch Sprechblasen dargestellt.

6.4.2.4 Dialogdisplay

Das Dialogdisplay gibt, je nach Wahl des Benutzers, verschiedene Kombinationen von Ausgaben aus. Dabei können sowohl alle Punkte der *Checkbox* aktiviert werden, als auch Einzelkombinationen. Alle diese Ausgaben haben mit der Komunikation der Agenten zu tun:

Message Es werden alle Dialoge der Agenten in Frage und Antwort Stil formatiert ausgegeben.

Debug Informationen über das Starten und die Kommunikation der Laufzeitagenten.

Info Informationen über das Starten und die Kommunikation der Laufzeitagenten.

6.4.2.5 Szenarioanzeige

Die Szenarioanzeige wird erst nach dem Einladen eines beliebigen Szenarios über "*LoadScenario*" oder "*Open*" aktiv. Dann zeigt die Szenarioanzeige in einer *Combobox* die Auswahl der Teilszenarien in einem Geamtszenario. Auf diese Art legt der Benutzer die Szene oder das Gesamtszenario fest, dass er anschauen will. Natürlich kann man auch

auf Gesamtszenario gehen und somit das ganze Szenario nach Drücken von "Start" anschauen. Wichtig hierbei ist, dass jedes Unterszenario 1 bis n auch durch den Zähler erfasst wird. Das heißt zum Beipiel, dass wenn eine Szene bis zum Zeitpunkt i geht, die nachfolgende Szene zum Zeitpunkt $i + 1$ anfängt.

6.4.2.6 Zähler und Bedienleiste

Die unter dem Dialogdisplay angeordnete Leiste mit von links nach rechts angeordneten Elementen, wie Zähler, Schieberegler, Rückwärtsknopf, Start/Stop, Pause und Vorwärtsknopf dient der Steuerung der Wiedergabe aller Elemente, die Informationen wiedergeben, die nach dem Starten von JADEX erzeugt werden. Das heißt, die grafische Simulation, alle Darstellungen des Dialogdisplays und die meisten Ausgaben des Agenteninnenlebens sind davon betroffen. Jedes Szenario ist in eine diskrete, endliche und linear geordnete Menge von Zeitpunkten strukturiert, indem sämtliche Ereignisse wie Kommunikation, Wissenserwerb oder Wissensverarbeitung der beteiligten Agenten eingebettet ist. Folgende Elemente sind vorhanden:

Zähler Hier wird der aktuelle Zeitpunkt des Szenaros angegeben. Das Tempo des Zählers ist nicht statisch, sondern durch den Schieberegler beeinflußbar.

Schieberegler Der Schieberegler reguliert das Tempo, in dem das Szenario dargestellt wird. Stellt man den Schieberegler auf 0 bleibt die Darstellung stehen. Diese Zustand ist mit dem Drücken des Pauseknopfes gleichzusetzen.

Rückwärts Hier kann die Szenariozeit, nach dem Drücken von "*Pause*" um einen Zeitzustand nach hinten versetzt werden.

Pause Der Pausenknopf lässt den Durchlauf des Szenarios pausieren, so dass alle Vorgänge, Darstellungen usw. eingefroren werden. Der Zähler bleibt auf seinem aktuellen Zustand.

Start Der Startknopf startet das geladene Szenario. Sämtliche Ausgaben sind von diesem Zeitpunkt aktuell. Es können nur geladene Szenarien im Zeitzustand gleich null gestartet werden.

Stop Stop lässt, ähnlich wie Pause, alle Vorgänge einfrieren, setzt aber zudem den Zeitzustand auf null und setzt somit das Szenario zurück. Es kann nun durch Start wieder gestartet werden.

Vorwärts Hier kann die Szenariozeit, nach dem Drücken von "*Pause*" oder bei Zählerzustand 0, um einen Zeitzustand nach vorne versetzt werden.

6.4.2.7 Agenteninnenleben

Das Agenteninnenleben kann für vier Agenten angezeigt werden. Welche Agenten angezeigt werden, ist dabei egal. Es werden aber nur im Szenario beteiligte Agenten dargestellt. Zu jedem Agenten kann man wählen, welcher Wissenstyp angezeigt wird.

6.4.3 Laden und Starten eines Scenarios

Es gibt prinzipiell zwei Wege um ein beliebiges Szenario zu öffnen. Der im Abschnitt 6.4.2 beschriebene Menüpunkt "*Open*" innerhalb des Menüs öffnet ein Verzeichnisfenster. Hier kann ein Szenario geladen werden. Bei selbst entwickelten Szenarien bitte das Verzeichnis mit der zum Starten ausgewählten XML-Datei auswählen. Der Pfad des gewählten Kimasszeanrios wird in die dazu vorgesehene Datei `kimas.Properties` eingeschrieben und JADEX wird auf diesem Szenario gestartet. Die zweite Möglichkeit, ein Szenario zu laden, funktioniert über den Menüunterpunkt *"LoadScenario"*. Hier wird das Szenario geladen, welches bereits in `kimas.Properties` eingeschrieben steht. JADEX wird dann automatisch für dieses Szenario gestartet. Dieser Menüpunkt funktioniert also nicht beim Erststart von KIMAS .

Folgende Reaktionen sind zu sehen:

1. Die Agentenboxen werden mit dem Namen der Agenten belegt. Die ersten vier Agenten werden von links Oben nach rechts Unten in Agentenboxen angezeigt.
2. Der Name des Szenarios wird in dem Textfeld, oberhalb des Dialogdisplays, eingeschrieben. Auch hier steht zu Anfang der Name des Gesamtszenarios.
3. Im Grafikdisplay sehen sie die agierenden Agenten des gewählten Szenarios.
4. Der Startknopf wird auf "Aktiv" gesetzt.

Das gewählte Szenario kann nun mit dem Startknopf gestartet werden. Es wird bis zu seinem Ende durchlaufen, wenn es nicht durch das Drücken von "Pause" oder "Stop" unterbrochen wird. Man kann das Starten am besten durch das Weiterzählen des Zählers kontrollieren. Zudem kann man innerhalb aller Fenster wie GRAFIKDISPLAY, DIALOGDISPLAY, AGENTINNENLEBENFENSTER und ZÄHLER eine Vielzahl von Veränderungen beobachten. Die Fenster können konfiguriert werden (siehe dazu 6.4.4 und 6.4.5).

6.4.4 Darstellung des Agenteninnenlebens

Es gibt zur Darstellung des Agenteninnenlebens vier seperate Fenster in denen beliebige im Szenario vorhandene Agenten dargestellt werden können. So bleibt es dem Anwender überlassen, wie er die vier Fenster konfiguriert. Er kann sich dabei vier gleiche Agenten mit verschiedenen Wissenstypen oder aber auch vier gleiche Wissenstypen für vier verschiedene Agenten anzeigen lassen. Die Fenster sind von einander unabhängig. Zur Konfiguration eines Fensters gibt es zwei *Comboboxen*. Die linke *Combobox* dient zur Auswahl des Agenten und zeigt beim Anklicken die aktiven Agenten. Die rechte der *Comboboxen* zeigt die vorhandenen Wissenstypen. Zur Auswahl stehen *ASP-Belief*, *InitialBelief*, *BeliefChange*, *BeliefSet*, *GoalAdopted*, *GoalChange*.

6.4.5 Darstellung von Kommunikation und Systemnachrichten

Die Kommunikation der Agenten und ihre Systemnachrichten werden im DIALOGDISPLAY dargestellt. Die Kommunikation der Agenten wird zu dem Zeitpunkt ausgegeben in der

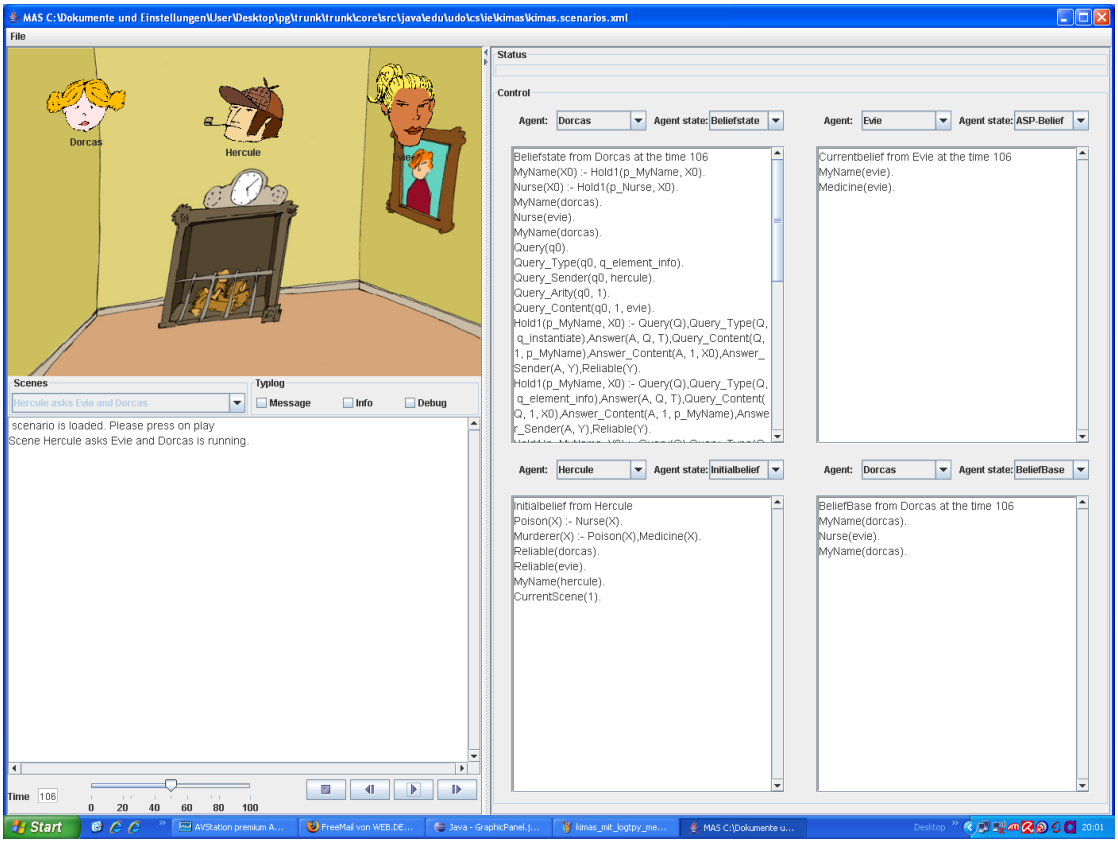

Abbildung 6.8: Das Agenteninnenleben

sie stattfindet. Es gibt drei Typen von Kommunikation, die *Messages*, *Debug* und *Info*. Diese sogenannten *Logtypes* können über die *Checkbox* am oberen Rand des DIALOGDISPLAY angesteuert werden. Auch hier sind beliebige Kombinationen möglich. Werden alle *Boxen* in der *Checkbox* angewählt, werden alle *Logtypes* zu einem Zeitpunkt i ausgegeben. Den *Logtypes* liegen verschiedene Ausgaben bzw. inhaltliche Konzepte zu Grunde. Während der Typ *Message* die formatierte Kommunikation der Agenten darstellt, sind die beiden Typen *Info* und *Debug* für systeminterne Ausgaben, wie die Startzeitpunkte der Agenten oder Adressierungen von Dateien zuständig. Man kann an dieser Stelle sagen, daß ersterer Typ für den Inhalt, während die beiden weiteren Typen für die Arbeit am System unumgänglich sind.

6.4.6 Sonstige Ausgaben

Über das DIALOGDISPLAY werden auch die *ExeptionEvents* ausgegeben. Die meisten Ausgaben sind dabei selbsterklärend.

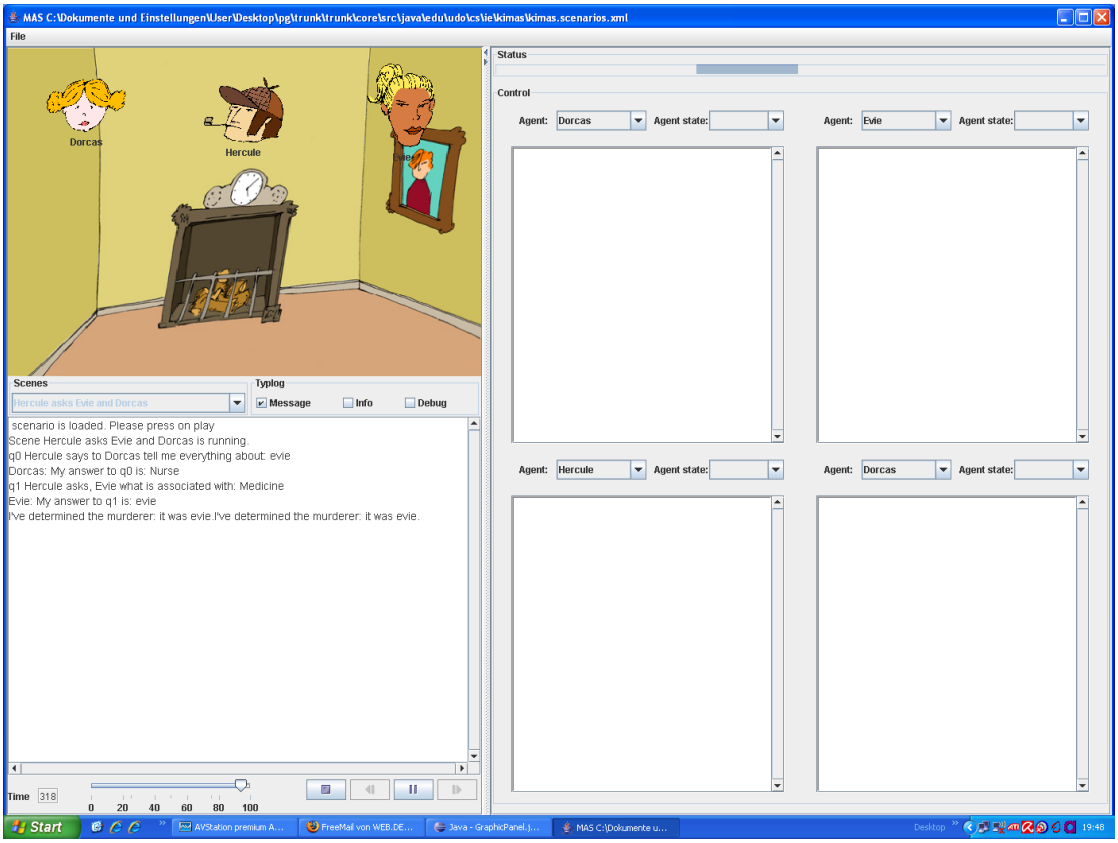

Abbildung 6.9: Die Kommunikation der Agenten

7 Erfahrungsbericht

Stefan Tittel

7.1 Seminarphase

Um das zum Erreichen der Projektgruppenzielsetzung notwendige Wissen zu erlangen, begann die Projektgruppe mit einem Seminar. Dabei oblag es jedem Teilnehmer, sich in ein für die Projektgruppe relevantes Teilgebiet einzuarbeiten, dieses aufzubereiten und den restlichen Teilnehmern durch einen Vortrag zugänglich zu machen. Das Seminar fand am 27. und 28. März 2006 in der Heimvolkshochschule Gottfried Könzgen in Haltern am See statt. Es wurden die folgenden Themen behandelt:

1. Kommunikation und Interaktion in Multiagentensystemen
2. verteilte Problemlösungsstrategien und Planen
3. verteiltes rationales Entscheiden
4. Lernen in Multiagentensystemen
5. formale Methoden in einem Multiagentensystemn zur logikbasierten Wissenrepräsentation und zum Schlussfolgern
6. der Motivationsbegriff im BDI-Modell
7. eine Einführung in das von der Projektgruppe verwendete JADEX-Multiagentensystem
8. Prozessmodelle in Software-Projekten
9. Revision von Wissensmengen und Wissensbasen
10. das SATEN-System zur Wissensrevision
11. Unterschiede zwischen Wissensrevision und -update
12. Wissensupdate mit der von uns verwendenten Inferenzmaschine DLV
13. Wissensrevision in Multiagentensystemen

Nach jedem Vortrag wurde dessen Relevanz für die Projektgruppenarbeit im Gruppengespräch evaluiert. Am Abend des ersten Seminartages fand zudem ein Kegeln zwecks besseren Kennenlernens statt. Ferner wurden Termine für die zwei Mal wöchentlich stattfindenden Projektgruppensitzungen festgelegt. Für diese Sitzungen wurde vereinbart, dass jeweils reihum Sitzungsleitung und Protokollführung von Projektgruppenmitgliedern durchgeführt werden sollten.

7.2 Sommersemester 2006

7.2.1 Entwicklungsmodell und Konventionen

In der ersten regulären Sitzung der Projektgruppe wurde kontrovers diskutiert, welches Prozessmodell bei der Softwareentwicklung Verwendung finden soll. Es wurde entschieden, *extreme programming* mit leichten Modifikationen umsetzen. Im Einzelnen:

- *pair programming* auf freiwilliger Basis
- feste Wochenstundenanzahl von etwa 20 Stunden pro Projektgruppenmitglied unter Vermeidung von Überstunden
- Quellcode-Änderungen jedem erlaubt, ohne dessen Autor vorher um Erlaubnis bitten zu müssen
- fortlaufende Integration der Komponenten
- Schreiben von Testtreibern vor oder nach der Implementierung
- Treffen grundsätzlicher Designentscheidungen in der Gruppe, Treffen kleinerer Designentscheidungen durch den Entwickler

Darüberhinaus wurde festgelegt, mit welcher JAVA-Version zu arbeiten ist, wie Dateinamen benannt werden sollen, wie der Quellcode zu formatieren ist etc.

7.2.2 Erste Gruppeneinteilung

Es wurde recht früh offensichtlich, dass eine Unterteilung der Projektgruppe in mehrere Untergruppen sinnvoll ist. Dazu wurden die zu bearbeitenden Aufgaben in drei Bereiche unterteilt:

1. Formalisierung des Szenariohintergrundes „The Mysterious Affair at Styles“ (bearbeitet durch die *Drehbuchgruppe*)
2. Integration von DLV (bearbeitet durch die *DLV-Gruppe*)
3. Implementierung des Multiagentensystems (bearbeitet durch die JADEX-*Gruppe*)

7.2.3 Einarbeitungsphase

Jede Gruppe benötigte zunächst einige Zeit, um sich in die von ihr zu bearbeitenden Aufgaben einzuarbeiten. Die dabei erzielten Fortschritte wurden regelmäßig den anderen Projektgruppenteilnehmern im Rahmen der Projektgruppensitzungen vorgetragen. Das vorrangige Ziel der JADEX-Gruppe bestand zunächst darin, ein minimales Szenario im Multiagentensystem lauffähig zu bekommen. Die DLV-Gruppe beschäftigte sich mit der Erstellung eines DLV-Wrappers und die Drehbuchgruppe befasste sich mit grundlegenden Methoden zur Formalisierung des Szenariohintergrunds.

7.2.4 Konzeptualisierungsphase

Nach den Vorarbeiten der Einarbeitungsphase wurde mit der Konzeptualisierung begonnen. Für die JADEX-Gruppe bedeutete dies die Erstellung eines Strukturmodells sowie die Erstellung von Klassen- und Sequenzdiagrammen zu ausgewählten Teilen der Implementierung. Die Drehbuchgruppe beschäftigte sich in dieser Zeit mit der Erstellung einer graphischen Inferenzstruktur, um die Zusammenhänge im logischen Ablauf des Szenariohintergrundbuches übersichtlich darzustellen. Da die DLV-Gruppe bereits zu einem sehr frühen Zeitpunkt die Integration von DLV in das Multiagentensystem fertigstellte, gingen ihre Teilnehmer in den anderen beiden Gruppen auf.

7.3 Wintersemester 2006/2007

7.3.1 Organisatorische Änderungen

Zu Beginn des Wintersemesters wurden verschiedene organistorische Änderungen verabschiedet. So sollte die Koordination der Projektgruppe vom Veranstalter teilweise auf zwei gewählte Studenten übertragen und die Gruppeneinteilung neu gebildet werden.

Der Wunsch zur Gruppenneubildung resultiert vor allem daher, dass die Notwendigkeit bestand, eine graphische Benutzeroberfäche zu entwickeln. Im vorhergehenden Semester befasste sich die Projektgruppe ausschließlich mit logischen Konzepten und deren Umsetzung sowie mit der Implementierung des Multiagentensystems. Die Erstellung einer GUI war daher eine neue Aufgabe, die erst der Zuweisung an Projektgruppenmitglieder bedurfte. Die neue Gruppeneinteilung ergab sich wie folgt:

- GUI-Gruppe: Erstellung der graphischen Benutzerschnittstelle
- Wissensrepräsentationsgruppe: Darstellung und Verarbeitung logischen Wissens
- Implementierungsgruppe: Implementieren des Multiagentensystems

Jede Gruppe wählte erstmalig einen Gruppenleiter, der sich für die Arbeitsorganisation innerhalb der Gruppe verantwortlich zeigte und als Ansprechpartner nach außen diente. Ferner wurde eine rechnergestützte Aufgabenverwaltung eingeführt, um einen besseren Überblick über den Fortschritt einzelner Aufgaben zu erhalten. Dies erwies sich insbesondere deshalb als vorteilhaft, weil zwischen vielen Aufgaben Interdependenzen bestanden.

Eine weitere Neuerung bestand darin, dass in der jeweils darauffolgenden Sitzung eine Gruppendiskussion über die Qualität der letzten Sitzungsleitung stattfinden sollte. Dies geschah in Reaktion darauf, dass einige eher stille Projektgruppenmitglieder im vorhergehenden Semester nicht in der Lage waren, die Sitzung mit dem notwendigen Nachdruck zu strukturieren. Ebenso wurde entschieden, dass der Sitzungsleiter künftig vor Kopf des Raumes sitzt bzw. steht, um ihm eine bessere Präsenz zu verleihen. Beide Maßnahmen trugen Früchte; die Qualität der Sitzungsleitungen stieg im zweiten Projektgruppensemester erheblich.

Ferner wurden Zeitpläne erstellt. Zunächst ein grober Zeitplan, welcher die wichtigsten Zwischenziele der Projektgruppenarbeit aufführte. Darüberhinaus ein spezifischerer Zeitplan von jeder Gruppe, der die Ziele der jeweiligen Gruppe mit Terminen versah. Dies war insbesondere deshalb notwendig, weil einige Aufgaben der einen Gruppe erst bearbeitet werden konnten, nachdem andere Aufgaben einer anderen Gruppe fertiggestellt worden waren. Gegen Ende der Projektgruppe erwies sich die Einhaltung dieser Zeitpläne jedoch als schwierig.

7.3.2 Implementierung

Schon bald zeigte sich, dass es zwischen der Wissensrepräsentationsgruppe und der Implementierungsgruppe Überlappungen gab, so z. B. bei der Entwicklung von *Update*-Operatoren oder der Erstellung des Umweltagenten. Dies ist mit der GUI-Gruppe nicht der Fall gewesen. Die dortige Entwicklung lief nach Fertigstellung der GUI-Schnittstelle größtenteils unabhängig von der Implementierungsgruppe. Die Überlappungen führten dazu, dass hinsichtlich einiger Aufgaben die Gruppengrenzen verwischten und Dinge gruppenübergreifend gemeinsam bearbeitet wurden.

Ferner stellte sich heraus, dass einige Konzepte (insbesondere im Bereich der Wissensdarstellung und -verarbeitung) nicht wie vorgesehen umgesetzt werden konnten. In diesen Bereichen wurden teils sehr umfangreiche Anpassungen vorgenommen. Durch die dadurch bedingte Zeitnot wurde es notwendig, auch in der vorlesungsfreien Zeit nach dem Wintersemester zwecks formloser Besprechungen und gemeinsamer Lösungssuche regelmäßig zusammenzukommen.

7.3.3 Vorträge während des Semester

Im Verlaufe der Projektgruppenarbeit wurde deutlich, dass nicht jedes Projektgruppenmitglied über alle notwendigen Kompetenzen im Bereich der Erstellung von Textdokumenten und der Programmierung in JAVA verfügte. Ebenso bedurfte es der Auffrischung und Vertiefung einiger theoretischer Grundlagen zu Themen wie z. B. Lügenbehandlung und es sollten neu erarbeitete Konzepte der Projektgruppe vorgestellt werden. Sobald ein solcher Anlass bestand, wurden die entsprechend kundigen Projektgruppenmitglieder mit der Vorbereitung von Vorträgen beauftragt, durch welche den anderen Projektgruppenmitgliedern der notwendige Kenntnisstand vermittelt wurde.

8 Fazit und Ausblick

Dennis Rüther

Die Planung und Realisierung eines Multiagentensystems war das Ziel der Projektgruppe 491. Die Agenten sollen, in einem von der Projektgruppe 491 entwickeltem Szenario, Wissen erwerben und verarbeiten können. Das Szenario wurde nach der Romanvorlage von Agatha Christies *"The Mysterious Affair At Styles"* entwickelt. Die Aufgabe der Agenten besteht darin, den Mörder zu finden oder besser gesagt inferieren zu können. Somit war geplant ein Multiagentensystem zu realisieren, dass in einer modellierten Umwelt agiert. Zudem sollten das logische Wissen, Fakten und Regeln, mit denen Softwareagenten schlussfolgern, für die Vorgabe des Romans auf die Agenten zugeschnitten werden. Sämtliche Operatoren, wie Schlussfolgrungs- und Kommunikationsoperatoren, sollten die Inferenz und Kommunikation der Agenten untereinander gewährleisten. In diesem Zusammenhang mußte ein Konzept für die geforderte Fähigkeit der Agenten zu Lügen gefunden werden, da in diesem Kontext Kommunikation, Inferenz und Glaubwürdigkeit stark verknüpft sind. Als wissenschaftliches Projekt einer Infomatikfakultät, wurden weitere Aspekte, durch die Projektleitung vermittelt und vorgegeben:

Die Agenten sollten reaktiv, proaktiv und sozial agieren. Ferner wurde einer hoher Maßstab an die Autonomie in der Planung, im Handeln und in der Kommunikation gefordert. Außerdem sollten die Agenten die Möglichkeit besitzen, in der Kommunikation mit Unwahrheiten umzugehen. Einige Agenten sind in der Lage zu lügen. Alle Agenten besitzen zu anderen Agenten Vertrauenswürdigkeiten, durch die sie den Wahrheitswert einer Aussage abschätzen können. Die Agenten mußten mit eigenen Wissenszuständen versehen werden. Klassische Probleme der Wissensmodellierung- und Verarbeitung wurden in den Fokus der Untersuchung gerückt, wie der Umgang mit subjektivem und objektivem Wissen, nichtmonotoner Inferenz und insbesondere die schwierige Trennung von Revision und Update. Als grundsätzliches Konzept für eine Agentenarchitektur war das BDI-Modell vorgesehen und wurde auch umgesetzt.

Diese Menge an Anforderungen verteilte sich in der praktischen Arbeit auf drei Bereiche:

- Die Modellierung
  - des Wissens und logischer Regeln der Agenten
  - der Umwelt
- die Implementierung
- die Realisierung notwendiger logischer Operatoren

Die Modellierung des Wissens wurde in mehreren Schritten gelöst. Zunächst mußte die Romanvorlage von Agatha Christie inhaltlich und im weiteren Sinne logisch excerpiert werden. Handlungsstränge und Personen mußten aus dem Roman herausgearbeitet und von für das Projekt unwichtigem Wissen getrennt werden. Die so erstellten Informationen wurden dann kapitelweise aufgeschrieben und die wichtigen Informationen herausgefiltert. Im zweiten Schritt wurde ein grundsätzlicher Inferenzbaum erstellt. Schlussendlich wurden Szenarien, Agenten und Umwelt in Abhängigkeit und enger Absprache, insbesondere mit den Entwicklern des JAVACODES und den Entwicklern der Operatoren, in XML-Dateien geschrieben. So gestaltete sich die Erstellung von Konzepten zur Behandlung von Vertrauenswürdigkeit als anspruchsvolles Problem. Trotzdem ist es gelungen die Protagonisten des Romans und ihre Umwelt in logisches Wissen so zu überführen, dass anhand der im Roman verwendeteten Beweisketten eine Inferenz zum Mörder (in dem Roman sind es zwei Mörder) führt. Das entwickelte Szenario ist in drei Unterszenen aufgeteilt, in denen Agenten die Möglichkeit haben ihre Umwelt zu untersuchen oder ihre Ziele durch interaktive Kommunikation zu erreichen. Die Anzahl der Aktionen, in denen Agenten ihre Umwelt verändern, ist allein schon durch die Romanvorlage beschränkt. Vereinbartes Ziel war es, umfassende Planung von Aktionen durch Agenten, die die Umwelt verändern, in der Modellierung der Szenen nicht stattfinden soll. Ein solches Konzept hätte den Umfang und die Komplexität des Gesamtkonzepts, und nicht nur hinsichtlich des Inferenzbaumes, extrem erhöht. Allerdings sollten solche Erweiterungen des Konzeptes für Agentenarchitekturen durchaus Potential für weiterführende Projekte haben.

Die Agentenarchitektur wurde grundsätzlich nach dem BDI-Modell konzipiert und auch umgesetzt. Jeder Agent besitzt eine logikbasierte Wissensbasis. Die Wissenbasen der einzelnen Agenten sind voneinander getrennt. Die Agenten generieren eigene Ziele. Da eine vollständige Autonomie eines Agenten nicht realisierbar ist, sollte möglichst große Autonomie erreicht werden. Diese Semiautonomie wurde über eine KNOW-HOW Struktur erreicht, so dass Ziele und Aktionen der Agenten miteinander verknüpft sind. Ein wirklich zielgerichtetes Planen von Aktionen zur Verwirklichung von Zielen, konnte aufgrund der zeitlichen Begrenzung dieser PG nicht verfolgt werden. Allerdings bietet die vorliegende Agentenarchitektur die Basis für weiterführende Projekte. Desweiteren konnten sowohl die Kommunikation, als auch die notwendige Fähigkeit des Schlussfolgerns, durch eigens dafür entwickelte Operatoren, realisiert werden. Die Entwicklung und Umsetzung eines Lügenkonzepts war eine starke Erweiterung des Gesamtkonzepts, obwohl analog zur Autonomie von Agenten, ein sehr komplexes Lügen, wie im zwischenmenschlichen Diskurs getätigt, kaum erreichbar scheint. Das Lügenkonzept basiert auf der Möglichkeit des Agenten, bei seiner Antwort, die potentielle Verhinderung eigener Ziele, miteinzubeziehen. Alle Agenten haben bestimmte Vorstellungen der Vertrauenswürdigkeiten anderer Agenten. Die Vertrauenswürdigkeit ändert sich, in dem Fall, dass ein Agent der Lüge überführt wird. Das Hauptproblem war es, dass die Aussagen eines Agenten, dessen Vertrauenswürdigkeit zu stark sinkt, automatisch als Lügen bewertet werden. Außerdem mußte gewährleistet werden, dass durch das Lügen Prioritäten von Fakten und Regeln nicht durcheinander geraten.

Das Lügenkonzept konnte je nach Antworttyp verschieden gut umgesetzt werden. So

ist zum Beipiel auf eine Ja-Nein Frage eine Lüge leicht zu generieren, während die Frage nach einem bestimmten Prädikat keine sinnvolle Lüge ohne weiteres zulässt. So ist das Schweigen eines Agenten, als Möglichkeit zu Lügen, eingeführt worden. Grundsätzlich soll der Agent defaultmäßig nicht lügen. Zu Wissen, zu dem er lügen soll, muss in seiner Wissensbasis ein Vermerk stehen.

Die wohl zentrale Komponente war das Konzept und die Realisierung der Wissenverarbeitung. So mussten initiales Wissen, alle Dialoge und die Glaubwürdigkeiten (*Reliabilities*) im logischen Sinn verknüpft und miteinander berechnet werden. Das "Agenteninnenleben"verfügt über eine Wissensbasis. Eine *beliefbase* umfasst *base-belief* (initiales Wissen), alle Dialoge und *Reliabilities*. Eine *beliefbase* ist ein logisches Programm, wobei die Regeln einen Verwurzelungsgrad besitzen. Die Umsetzung dieser Operatoren, Update und Revision, konnte somit umgesetzt werden. Auch die Trennung von subjektivem und objektivem Wissen hat sich nicht nur als funktionierendes, sondern auch als sinniges Konzept erwiesen. Es muss an dieser Stelle nicht darauf hingewiesen werden, welches Potential weiterführende Untersuchungen haben. Die Anzahl möglicher Konzepte Wissen zu inferieren ist sehr hoch. Dementsprechend sollte an dieser Stelle durchaus der wertende Aspekt mit einfließen, dass die Wissensverarbeitung eine Technik ist, die eines hohen Maßes an praktischer (und natürlich auch theoretischer) Erfahrung bedarf.

Neben den oben beschriebenen Schwerpunkten, wie Agentenarchitektur, ist die technische und praktische Umsetzung ein weiterer Schwerpunkt gewesen. KIMAS das fertige Produkt, besteht aus existierenden Komponenten und Standardtechnologien, wie zum Beispiel JAVA. So war im Vornherein klar, dass nicht alles selbst programmiert werden kann. Da eine Menge von zum Teil sehr guten Softwarelösungen zu Verfügung stehen, mußten die verschiedenen möglichen Komponenten auf ihre Vor- und Nachteile, insbesondere im gegenseitigen Zusammenspiel überprüft werden. KIMAS verbindet die Multiagentenplatform JADEX mit DLV [EFLP00]. JADEX wurde in JAVA implementiert und dient als Werkzeug, um Multiagentensysteme komfortabel entwickeln zu können. Standardtechnologien, wie JAVA und XML, und die Anlehnung an das BDI-Modell boten die idealen Vorraussetzungen, um effizient arbeiten zu können. Allerdings könnten eigens entwickelte Softwarelösungen auch in diesem Bereich Vorteile bieten. DLV ist eine von der TU Wien entwickelten Inferenzmaschine, die aus entsprechend logisch modelliertem Wissen, Schlussfolgerungen ziehen kann. Somit bietet es notwendige Funktionen, wie zum Beispiel die Unterstützung von Logik und Inferenz an. Das logische Format ist die *Antwortmengen-Programmierung* (ASP). Die Vorteile der ANTWORTMENGEN-PROGRAMMIERUNG, strikte Negation und *Default* Negation, gaben den Anstoß, sie ähnlichen Sprachen vorzuziehen. Insbesondere *Default* Negation war ein grosser Vorteil. KIMAS erfüllt die gesetzten Ansprüche auch in implementatorischer Hinsicht. Es ist ein Werkzeug, um auf hohem Niveau Wissenverarbeitung in einem Multiagentensystem zu entwickeln und sichtbar zu machen. Ferner ist es gelungen, die Programmierung so zu gestalten, dass fortführende Forschungsprojekte dieser Art auf unserer Arbeit aufbauen können. Das ehrgeizige Ziel KIMAS mit einem Editor zu versehen, in dem eigene Szenarien entwickelt werden können, sollte unbedingt weiterverfolgt werden.

Zusammenfassend kann man die Umsetzung der Anforderungen als gelungen betrachten. Die oben beschriebenen Anforderungen sind, soweit möglich, erfüllt. Weitreichende

Erfahrung im Umgang der Wissensverarbeitung konnten gemacht und praktisch umgesetzt werden. KiMAS ist eine in der Praxis anwendbare Umsetzungen, in seinem Gebiet, und bietet Möglichkeiten als Instrument für weiterführende Studien an.

Glossar

KiMas	Jadex	Fachliteratur
atomerer (Jadex) Plan mit Parametern	Plan	Aktion
Beliefbase (Teil der Jadex Beliefbase)		Beliefbase
Metaplan	Nicht Metalevel-Plan	Planungs-Algo.
	Metalevel-Plan	Actionselection
Jadex Beliefbase	Beliefbase	Planungs-Algo.
Skills		Menge möglicher Aktionen eines Agenten
Attitudes		Motivation
	Capabilities	
	Teil der Goalbase	Intentions
	Teil der Goalbase	Desire
Belief Set		Antwortmenge
Know-How		Know-How
Goal	Goal	Ziel

Literaturverzeichnis

[AH03] David Thomas Andrew Hunt. *Pragmatic Unit Testing in Java with JUnit.* The Pragmatic Programmers, 2003.

[Chr99] Agatha Christie. *The Mysterious Affair At Styles.* Scherz Verlag, Bern, 1999.

[dev03] Java developers. `http://www.javangelist.de`, 2003.

[Dol06] Mark Doliner. `http://cobertura.sourceforge.net/`, 2005–2006.

[dt06] Eclipse development team. `http://www.eclipse.org/emf`, 2006.

[EFLP00] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the dlv system. *Logic-based artificial intelligence*, pages 79–103, 2000.

[EFST02] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. On properties of update sequences based on causal rejection. *Theory Pract. Log. Program.*, 2(6):711–767, 2002.

[FN71] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[Gro06] The Object Management Group. Meta object facility core specification version 2.0. `http://www.omg.org`, 2006.

[Gä94] H. Gärdenfors, P. und Rott. *Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford University Press, 1994.

[J.77] Weizenbaum J. *Die Macht der Computer und die Ohnmacht der Vernunft.* Frankfurt am Main:Suhrkamp, 1977.

[KM91a] H. Katsuno and A. Mendelzon. Propositional knowledge base revision and minimal change. *Artificial Intelligence*, 52:263–294, 1991.

[KM91b] Hirofumi Katsuno and Alberto Mendelzon. On the difference between updating a knowledge base and revising it. In Richard Fikes James F. Allen and Erik Sandewall, editors, *Principles of Knowledge Representation and Reasoning*, pages 387–394. Morgan Kaufmann, San Mateo, California, 1991.

[OM04] Incorporated Object Mentor. `http://www.junit.org`, 2001-04.

[Sin99] M. P. Singh. Know-how. In Anand S. Rao and M. J. Wooldridge, editors, *Foundations of Rational Agency, Applied Logic Series*, pages 105–132. Kluwer, 1999.

[SRG99] M. P. Singh, A. S. Rao, and M. P. Georgeff. Formal methods in DAI: Logic-based representation and reasoning. In G. Weiss, editor, *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*, pages 331–376. The MIT Press, Cambridge, Massachusetts, 1999.

[Ull06] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Press, 2006.

[Woo99] Michael Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, Massachusetts, 1999.

Stichwortverzeichnis

ADF, 152
Agentenarchitektur, 91
Agentencombobox, 140
Agenteninnenleben, 129
Agentenmodell, 24, 91
Agentstatecombobox, 140
Aktionen, 82
Aktionsauswahl, 28
Aktionsbaum, 29
Ant, 119
Antwortmenge, 22
Antwortoperator, 92, 99
assert method, 114
Attitudes, 27
Aufgabenverwaltung, 164

BDI, 91
    Beliefs, *siehe* Beliefs
    Modell, 25
Belief-Base, 17
Belief-State, 17
Beliefchange, 132
Beliefrevision, 28
Beliefs, 17, 27
build file, 119
build process, 119
byte code, 119

CDF, *siehe* character definition file, 145
character definition file, 99, 145
Charakteragent, 100, 106
classpath, 119
Cobertura, 119
code coverage, 117
    branch, 118, 119
    line, 117–119
constraint rules, 99, 100
coverage tool, 117–119

Datalogregel, 94
debug, 140
Deliberation, 28
Dialogdisplay, 157
Ding, 108
    ändern von, 110
    verschieben von, 110
DLV, 93

Eclipse, 124, 144
    Download Manager, 128
    Feature, 126, 128
    Plugin, 126, 128
Ecore, 123–125
EDF, 149
EMF, 122–124, 144
    Genmodel, 124, 125
environment definition file, 149
extreme programming, 163

Fortschrittsbalken, 157

GEF, 144
Grafikdisplay, 157
Gruppeneinteilung, 163, 164
GUI, 128

Handlungsstrang, 62
    Einteilung, 62
Hold-Fakten, 19, 35

info, 140
Informations-Historie, 17, 35
Initialisierung, 100
instrumentation

bytecode-level, 119
runtime, 119
source-level, 119
integration test, 112
Intention, 27, 92
Interaktion
Umwelt, 109

JUnit, 113, 125

Kimas Editor, 144
Know-How, 29, 92, 96
Kommunikation, 129
Fragen, 84

Lügen
Lügengenerierung, 85
Verschweigen, 87
Layout-Manager, 129
Logische Antworten, 14
Logische Fragen, 11

Means-ends-reasoning, 28
message, 140
meta modelling, 122
Meta-Metamodell, 122
meta-object facility, 123
Metadaten, 122
Metamodell, 122
Metaplan, 98
Minimalitat, 106
Mock Objects, 112
MOF, *siehe* meta-object facility

Nachrichtentransfer, 107
negation-as-failure, 93

Observable, 133
Observer, 133
Observer-Pattern, 133
obvious, *siehe* offensichtlich
offensichtlich, 108
Organisation, 164
Ort, 108

Prädikate, 71
Programm
logisches, 93
Update, 22, 37
Projektphase, 163–165
Prozessmodell, 163

Rejection Set, 41

scenario definition file, 99, 140, 152
SDF, *siehe* scenario definition file
Seminar, 162
SimpleOperator, *siehe* Wissensoperator, 105
Starteragent, 100
Startordnung, 99, 100
Swing, 129
Szene, 62, 107
Einteilung, 65
Szenencombobox, 140
Szenenverwaltung, 107–108

TestSuite, 115

UML, 123, 124
Umweltagent, 100, 106–110
unit test, 110–113, 115
Unterziele, 28
Update, 36
UpdateOperator, 106

Vertrauenswürdigkeit, 18
Vertrauenswurdigkeit, 92
Vertrauenswürdigkeit, 35, 45
Vortrag, 162, 165

Wasserfallmodell, 112
Wissen, 130
gemeinsames, 79, 131
initiales, 79, 131
Wissensbasis, 92
Wissensoperator, 17, 92, 104

XML Schema, 122–126
XSLT, 128

Zeitplan, 165
Zielgenerierung, 28

Danksagung

Unser Dank gebürt in erster Linie den Veranstalter der Projektgruppe 491 für ihre Geduld und fachliche Unterstützung, namentlich Prof. Dr. Gabriele Kern-Isberner und Dipl.-Inform. Manuela Mark. Desweiteren bedanken wir uns bei folgenden *open source*- und *free ware*-Projekten in alphabetischer Reihenfolge: Ant, ASM, BeanShell, Cobertura, DLV, Dom4j, Eclipse, Jadex, Apache ORO, JUnit, Log4J, SourceForge.net, Subversion, Xalan, Xerces.