Endbericht
PG 545 - Intelligent Cowbots

Autoren:
F. Bienek, E. Böhmer,
S. Broszeit, D. Hölzgen,
B. Jablkowski, M. Kruse,
A. Löwen, T. Vengels

1. Dezember 2010

Inhaltsverzeichnis

1 Einleitung **7**

2 BDI-Modell **11**
2.1 Klassisches BDI-Modell 11
2.2 Cowbot BDI-Modell 12
2.3 Belief Revision Function 16
2.3.1 Schnittstellen zu anderen Komponenten 16
2.3.2 Einführung in die Revision 16
2.3.3 ELP - Motivation und Begriffserläuterung 20
2.3.4 Aufbau der Belief Revision Function im Cowbot-BDI-Modell 23
2.3.5 Datenkomponenten der BRF 23
2.3.6 Funktionale Komponenten der BRF 24
2.3.7 Beispiel: Revision in einer Kachelwelt 33
2.4 Desire Generation 36
2.4.1 Einordnung 36
2.4.2 Beschreibung 36
2.4.3 Erster Ansatz nach Meneguzzi und Luck 36
2.4.4 Kritik am Ansatz von Meneguzzi & Luck 38
2.4.5 Umsetzung der Kritik 39
2.4.6 Alternative Ansätze 42
2.5 Deliberation 43
2.6 Planner 45
2.6.1 DLV^K 50
2.7 Action Selection 55
2.8 Vergleich der beiden Modelle 56

3 Multi-Agenten-System **59**
3.1 Das Phasenmodell 60
3.1.1 Einordnung des Phasenmodells 60
3.1.2 Benutzte Begriffe 60
3.1.3 Zwei Arten von Phasenmodellen 60
3.1.4 Entwicklung des Phasenmodells 61
3.1.5 Vier-Phasen-Modell 61
3.1.6 Realisierung 63

3.1.7 Konventionen für das Phasenmodell . . . 63
3.1.8 Verteiltes Planen . . . 64
3.2 Die MAS-Architektur . . . 65
3.3 Szenariobeschreibung . . . 65
3.4 Umwelt . . . 65
3.5 Konzeption . . . 66
3.6 Die Gaia-Methode . . . 66
3.6.1 Vorbemerkungen . . . 66
3.6.2 Analyse . . . 66
3.6.3 Entwurf . . . 72

4 Kommunikation **79**
4.1 Kommunikation . . . 79
4.1.1 Motivation . . . 79
4.1.2 Sprechakttheorie . . . 79
4.1.3 Szenariobezug . . . 80
4.1.4 Kommunikationsverlauf . . . 80
4.1.5 Interpretation der Kommunikation . . . 80
4.1.6 Konsequenzen unseres Kommunikationsmodells . . . 81

5 Wissensdarstellung **83**
5.1 Wissensrepräsentation im Cowbot-BDI-Modell . . . 83
5.2 Agentenprogramme . . . 85

6 Strategien **89**
6.1 Strategien im Szenario . . . 89
6.1.1 Kühe treiben . . . 89
6.1.2 Scout . . . 101
6.1.3 Strategieentwicklung *dooropener* . . . 106

7 Implementierung **109**
7.1 Jason . . . 109
7.1.1 Aufbau und Funktion des Interpreters . . . 109
7.1.2 Komponentenmodell von Jason . . . 111
7.2 MASSim . . . 112
7.2.1 Umwelt . . . 112
7.2.2 Verbindungsprotokoll . . . 115
7.2.3 SVG Ausgabe . . . 115
7.2.4 Spieleclient . . . 116
7.2.5 MapTool . . . 117
7.3 Architektur . . . 117
7.3.1 Überblick . . . 117
7.3.2 Logik Bibliothek . . . 118
7.3.3 Wissensrevision . . . 123
7.3.4 Desire-Generierung . . . 126
7.3.5 Deliberation . . . 130

7.3.6 Dynamisches Planen 131
7.3.7 Action Selection 134
7.3.8 Cowbot Agenten Architektur 135
7.3.9 Cowbot Agenten-Klasse 136
7.3.10 Graphische Oberfläche 138
7.4 Agentenimplementierung 139
7.4.1 Scout-Implementierung 139
7.4.2 Driver-Implementierung 146
7.4.3 Door Opener-Implementierung 149

8 Bedienungsanleitung **153**
8.1 Vorwort 153
8.2 Benötigte Komponenten 153
8.2.1 Java 153
8.2.2 Massim 153
8.2.3 DLV 153
8.2.4 Jason 154
8.2.5 Cowbot Multi Agent System 154
8.3 Installation und Konfiguration 154
8.3.1 mas2j 154
8.3.2 agent.xml 154
8.3.3 local_config.xml 156
8.3.4 brf_cfg.xml 156
8.3.5 Motives 157
8.3.6 logging.properties 157
8.4 Starten des Systems 157

9 Anpassungen für den Contest **159**
9.1 Änderungen am MAS 159
9.2 Änderungen am Framework 160
9.3 Gesammelte Erfahrungen 161

10 Erfahrungsbericht **163**
10.1 Erfahrungsbericht 163
10.1.1 Seminarphase 163
10.1.2 Wintersemester 164
10.1.3 Sommersemester 165
10.2 Organisation der PG 166
10.2.1 Projektleiter 166
10.2.2 Gemeinsame Sitzungen 166
10.2.3 Wiki 167
10.2.4 SVN 167
10.2.5 Tasktime 167

11 Fazit und Ausblick **169**

12 Danksagung **171**

Kapitel 1

Einleitung

M. Kruse

Das folgende Dokument ist der Endbericht der Projektgruppe 545 die im Sommersemester 2009 begann und im Wintersemester 2010 endete. Das Ziel der Projektgruppe war die Konzipierung eines Multiagentensystems, die es den Agenten ermöglicht in einer spezifischen Umwelt intelligent zu agieren. Dabei sollten sich die Agenten durch Kommunikation absprechen können, damit kollaboratives Verhalten ermöglicht wird.

Die Umwelt ist als

- diskret[1],
- dynamisch[2],
- unzugänglich[3] und
- nicht-deterministisch

vorgegeben. Ein Hauptaugenmerk wurde auf die Repräsentation des Wissenszustandes5.1 gelegt, welchem durch Revision und Update neue Informationen hinzufügt werden kann.

Für die praktische Erprobung des *[Multiagentensystems* diente ein Wettstreit der TU Clausthal3.3, der Multi-Agent-Contest. Hierbei handelt es sich aktuell um das *cowbot*-Szenario, indem zwei Teams versuchen mit mehreren Agenten virtuelle Kühe in einen Pferch zu treiben. Dabei bewegen sie sich in einem Labyrinth von Wänden und Gattern, wobei die Gatter von einem Agenten geöffnet

[1] Diskret bedeutet, dass die Umwelt aus einer begrenzten Anzahl von Zuständen besteht.

[2] Eine dynamische Umwelt steht für eine sich verändernde Welt, während der Agent seine Planungen und handlungen durchführt.

[3] Unsere Weltsicht ist unzulänglich, somit kann ein Agent nicht die gesamte Welt in seine Betrachtungen mit einbeziehen. Ein Agent hat nur eine, vorher definierte, Sichtweite. Seine Eindrücke von der Welt setzen sich nur aus seinem bisherigen Wissen, seiner aktuellen Sichtweite und der durch Kommunikation mit anderen Agenten, ausgetauschten Wissen, zusammen.

werden können. Die Kühe ihrerseits versuchen sich von den einzelnen Cowbot-Agenten zu entfernen ohne dabei auf ein statisches Muster zurückzugreifen, da sie sich mit gewissen Wahrscheinlichkeiten wegbewegen. Das Treiben der Kühe stellte eine besondere Herausforderung dar, da eine Neuplanung in den meisten Fällen notwendig wurde. Die Umwelt wird durch einen Server der Universität simuliert, der den o.g. Vorgaben entspricht.

Die Komplexität des Szenarios ist für die Evaluierung des zu implementierenden Systems gut geeignet, da alle Fähigkeiten der Agenten auf die Probe gestellt werden ohne zu komplex zu sein. Darunter zählen:

- Wissensrepräsentation5.1
- Schlussfolgern[4]
- Kommunikation4.1[5]
- Kooperation[6]
- dynamisches Planen2.6[7]
- strategisches Handeln6.1[8]
- Know-How[9]

Als Grundlage diente das klassiche BDI-Modell2.1, das um eine weitere Komponente, der Motivation, erweitert wurde. Die Konzeptionierung und Entwicklung des Agentenmodells schloss die Evaluierung mit Gaia ein und führte so zu einem rollenbasierten Modell.

Ein Ansatz für die verteilte Planung war das von uns angpasste Phasenmodell3.1 von Weiss [24]. Dieses findet Anwendung bei der Absprache von Plänen um Ziele zu erreichen die nur in einer Gruppe zu realsieren sind. Um zum Beispiel durch ein Gatter zu gehen, benötigt ein Agent Hilfe eines anderen Agenten, der ihm das Gatter öffnet. Nach dem Durchschreiten kann der helfende Agent seine eigenen Ziele weiterverfolgen. Das Phasenmodell ermöglich bei solchen Abläufen die Einteilung in verschiedene Phasen. Außerdem wurde das 6-Phasen-Modell von uns auf ein 4-Phasen-Modell übertragen, sodass auch Einzelaufgaben eines Agenten in Phasen eingeteilt werden können.

[4] Aus vorhandenem Wissen weiteres Wissen folgern.

[5] Jeder Agent nimmt einen Teil der Umwelt war. Durch Kommunikation der Agenten untereinander können Aufgaben erledigt werden, die ein Agent allein nicht bewältigen kann. Auf der anderen Seite können auch nur Umweltinformationen übertragen werden, die für andere Agenten interessant sein könnten.

[6] Einige Aufgaben können von Agenten nicht allen bewältigt werden. Daher müssen die Agenten kooperationsbereit sein und sich zur Lösung einer Aufgabe absprechen

[7] Da die Umwelt zu komplex ist um statische Pläne für alle Gegebenheiten zu implementieren, muss es den Agenten möglich sein situationsabhängige Pläne zu entwickeln.

[8] Alle statischen und dynamischen Pläne müssen sich einer Strategie unterwerfen, die im vorhinein festgelegt wurde.

[9] als wiederverwendbar erkannte dynamische Pläne werden dem Know-How eines Agenten hinzugefügt.

Die Programmierung fand in der Java-Implementierung *Jason*7.1 statt, die auf der Spezifikationssprache *AgentSpeak* beruht. Jason unterstützt die Entwicklung von Multiagentensystemen gemäß dem klassischen BDI-Modell. Die zusätzliche Integration von DLV-K bietet die Möglichkeit dynamische Pläne in das System zu integrieren.

Der Enbericht nimmt zunächst einen Vergleich zwichem dem klassichen BDI-Modell und dem Cowbot-BDI-Modell vor, wobei die einzelnen Komponenten genauer beschrieben werden. Darauf folgend wird das Multiagentensystem mit seinem zugrundeliegenden Phasenmodell und seiner Architektur beschrieben. Nachdem die Kommunikation und das Wissensmodell dargestellt werden, wird darauf beruhend die Strategie unserer Agenten und deren rollen erläutert. Die Implementierung und die Contest-Teilnahme, sowie notwendige Änderungen an der Implementierung für den Contest, wird am Ende des Berichts beschrieben.

Kapitel 2

BDI-Modell

S.Broszeit

2.1 Klassisches BDI-Modell

Die BDI-Architektur [24] ist eine Agentenarchitektur, die im Gegensatz zum reaktionären Agentenmodell [24] weitgehend autonome Agenten ermöglicht. Wie in jeder Agentenarchitektur operiert der Agent auf Perzeptionen über seine Umwelt und führt aufgrund seiner internen Prozesse letztendlich eine Aktion, die aus den Perzeptionen gefolgert ist, in der Umwelt aus.

In dieser Architektur besteht ein Agent aus den funktionalen Komponenten *belief revision function*, *desire generation*, *deliberation* und *action selection* sowie den Datenkomponenten *beliefs*, *desires* und *intentions*. Die *beliefs* bilden das Wissen des Agenten über seine Umwelt ab, das die *belief revision function* aus den Perzeptionen folgern kann, die *desires* umfassen, die von der *desire generation* aus den *beliefs* gefolgerte Optionsmenge an Zielen, die er in dieser Umwelt anstreben könne, und die *intentions* entsprechen konkreten Entschlüssen, eine solche Option zu verfolgen, die von der *deliberation* gefasst werden. Letztendlich wählt die *action selection* entsprechend der aktuellen *intention* eine Aktion aus, die in der Umwelt ausgeführt wird.

Die Datenkomponenten stehen in ständiger Wechselwirkung2.1 zueinander. Zum Einen generiert die *desire generation desires* aus *beliefs* und die *deliberation* wählt ein *desire* aus, zu dem eine entsprechende *intention* gebildet wird. Zum anderen treten Seitenbedingungen der Komponenten aufeinander auf: So fließen die *beliefs*, die der Agent bereits besitzt, in die Folgerung neuer *beliefs* über die aktuelle Umwelt ein, die *desire generation* nimmt beim Bilden der *desires* außerdem Rücksicht darauf, dass diese nicht bestehenden Intentionen widersprechen und die *deliberation* gleicht die bestehenden *intentions* mit den aktuellen *beliefs* ab, um zu überprüfen, ob ihre Durchführung noch möglich ist.

Der Zustand des Agenten lässt sich als Tripel (B, D, I) beschreiben, dessen Komponenten jeweils Teilmengen der Mengen Bel, Des, Int sind, die die

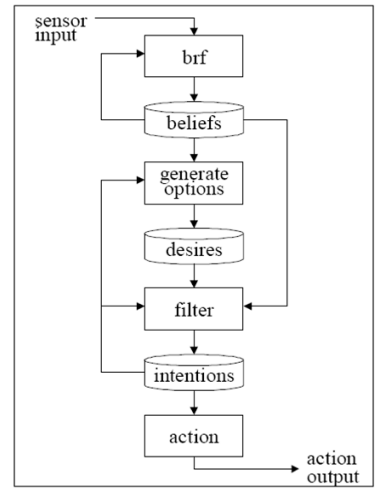

Abbildung 2.1: Der BDI-Prozess [25]

Gesamtmengen über alle möglichen *beliefs*, *desires* sowie *intentions* sind.

Die funktionalen Komponenten des Agenten erfüllen folgende Vorschriften:

- belief revision function: $\mathcal{P}(Bel) \times P \rightarrow \mathcal{P}(Bel)$, wobei P die Menge aller möglichen Perzeptionen ist.
- desire generation: $\mathcal{P}(Bel) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Des)$
- deliberation: $\mathcal{P}(Bel) \times \mathcal{P}(Des) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Int)$
- action selection: $\mathcal{P}(Int) \rightarrow Act$, wobei Act die Menge aller möglichen Aktionen ist

2.2 Cowbot BDI-Modell

Einleitung

Das im Folgenden beschriebene Konzept wurde innerhalb des ersten Semesters unser Projektgruppenarbeit erarbeitet. Ziel der PG ist es ein Agentenkonzept für kooperativ, autonom und intelligent agierende Agenten in einer diskreten, dynamischen, unzugänglichen und nicht-deterministischen Welt aufzustellen und umzusetzten. Ausgangspunkt des Konzeptes bildet neben den in der Seminarphase der PG entstandenen Ausarbeitungen das allgemeine BDI Modell. Dieses wird verwendet, um das Agenten Konzept zu veranschaulichen und zu beschreiben.

Überblick

Das in 2.2 beschriebene BDI-Modell wird von uns in die Bereiche *belief revision function*, *desire generation function*, *deliberation* und *action selection* unterteilt. Dabei kennzeichnen wir die Wissensmengen *beliefs*, *motives*, *desires*, *goals*, *intentions* und *actions*. Die *belief revision function* folgert aufgrund des Wissens des Agenten (*beliefs*) und der Wahrnehmungen des Agenten (*perceptions*), einen an die wahrgenommene Situation angepassten Wissenszustand. Aufgrund dieses neu erworbenen oder gefolgerten Wissens (*beliefs*) stellt die *desire generation function* Wünsche (*desires*) anhand von vordefinierten Motiven (*motives*) für alle vom Agenten durchführbaren Optionen auf. Die *desires* werden von der *deliberation* ausgewertet und mit Hilfe des *planers* mit konkreten Vorgehensweisen untermauert. Die so entstandenen Pläne werden mit dem Kontext, in dem sie verwendet werden sollen, als Intentionen (*intentions*) abgelegt. Letztendlich wählt die *action selection* aus der aktuellen *intention* den nächsten Planschritt aus und gibt die dazu passende Aktion (*action*) an die Umwelt weiter. Diese Aufteilung entspricht - mit Ausnahme der *motives* - der geläufigen BDI-Unterteilung.

Datenkomponenten

Im folgenden Abschnitt wollen wir die Datenkomponenten vorstellen, die vom erweiterten Cowbot-BDI-Modell benutzt werden.

Beliefs Eine der wichtigsten und notwendigsten Komponenten ist das Wissen (*beliefs*). Damit die Bedingung der Rationalität erfüllt wird, müssen Agenten mit Wissen arbeiten können. Dies ist allerdings erst möglich, wenn eine Speicherung der Daten gewährleistet ist. Diese Komponente wird im folgenden auch als *epistemic state* bezeichnet.

Desires Jede menschliche, rationale Handlung ist motiviert. Die Entscheidungen, welche Menschen treffen, um ihre Wünsche (*desires*) zu erfüllen, basieren immer auf konkreten Motiven (*motives*). Die generierten Wünsche verwaltet der Agent in den *desires*.

Intentions Intentionen (*intentions*) repräsentieren die konkreten Vorhaben eines Agenten, seine Ziele (*goals*) zu verfolgen. Zu jedem ausgewählten Ziel wird eine geordnete Liste von Intentionen verwaltet.

Plan Library Diese Datenkomponente beinhaltet die statischen Pläne eines Agenten.

Actions Diese Komponente enthält alle möglichen Aktionen (*actions*) des Agenten. Eine Aktion ist eine einfachste, nicht mehr differenzierbare Handlungen.

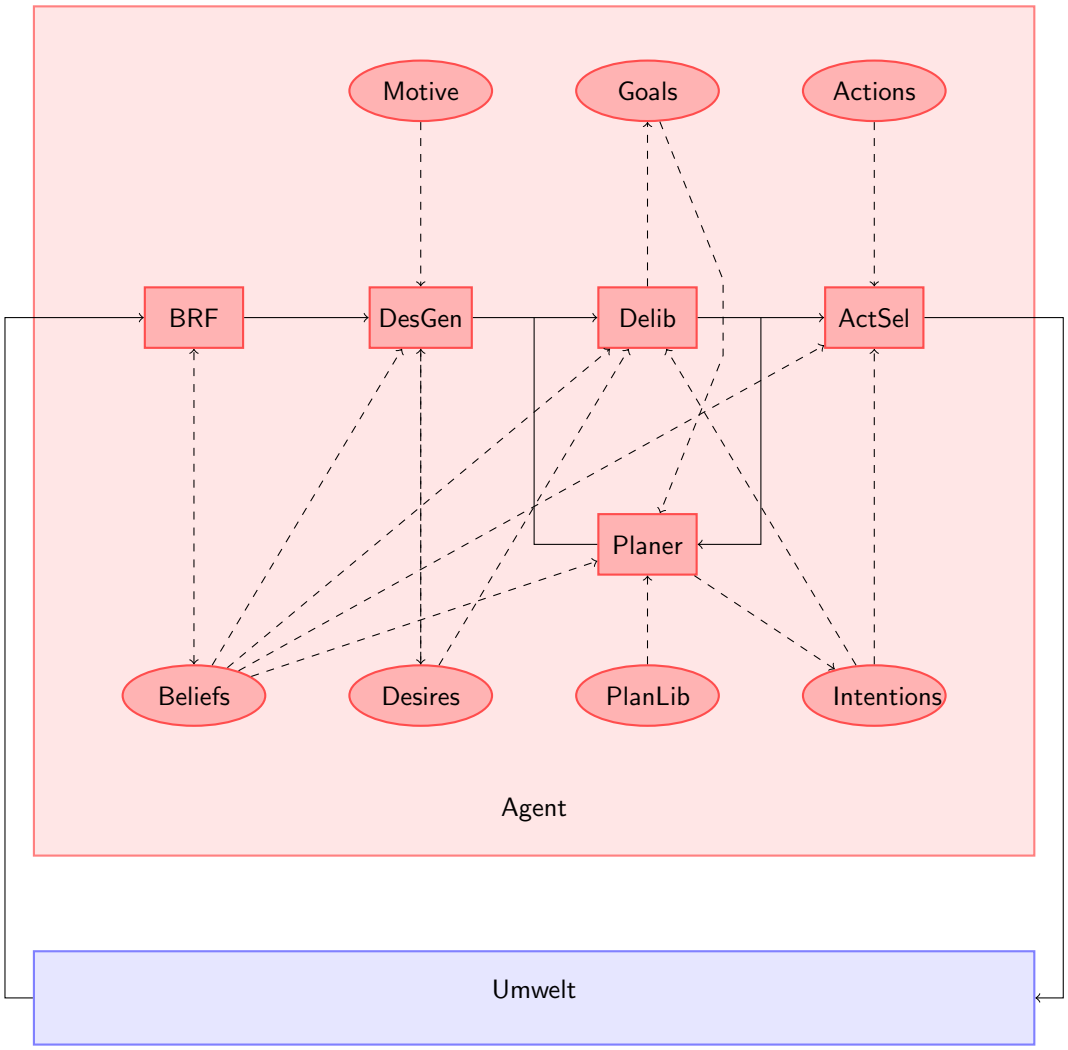

Abbildung 2.2: Überblick über das Cowbot BDI Modell

Funktionale Komponenten

Die funktionalen Komponenten aus 2.2 sind wie folgt definiert.

BRF: Pflegt neues Wissen in das alte Wissen des Agenten ein. Außerdem muss diese Komponente gewährleisten, dass das Wissen des Agenten konsistent gehalten wird.

Definition 2.2.1 (BRF)**.** *Sei Bel die Menge aller* epistemic states*, sei Per die Menge aller Wahrnehmungen (*perceptions*), dann lautet die Funktionsvor-*

schrift: $Bel \times Per \rightarrow Bel$.

DesGen: Generiert Wünsche aus den Motiven mit Einbezug des Wissens.

Definition 2.2.2 (DesGen)**.** *Sei Bel die Menge aller* epistemic states*, sei Mot die Menge aller Motive, dann lautet die Funktionsvorschrift:* $Bel \times Mot \rightarrow Des$. *Des entspricht der Menge aller* desire states.

Delib: Revidiert das aktuell verfolgte Ziel des Agenten. Dazu analysiert sie den Zustand des Agenten durch das Einbeziehen seines Wissens, seiner Wünsche, seines aktuell angestrebten Ziels und seiner Intentionen.

Definition 2.2.3 (DesGen)**.** *Sei Bel die Menge aller* epistemic states*, sei Goa die Menge aller* goal states*, sei Des die Menge aller* desire states *und Int die Menge aller* intention states*, dann lautet die Funktionsvorschrift:* $Des \times Goa \times Bel \times Int \rightarrow Goa$

Planner: Gibt zu einem gegebenen Kontext und Ziel einen passenden Plan aus, den der Agent verfolgen soll.

Definition 2.2.4 (Planner)**.** *Sei Goa die Menge aller* goal states*, sei Bel die Menge aller* epistemic states*, sei Pla die* plan library*.* Int *entspricht der Menge aller* intention states*. Die Funktionsvorschrift lautet:* $Goa \times Bel \times Pla \rightarrow Int$

ActSel: Wählt für eine atomare Intention eine Aktion aus, die diese Intention erfüllen soll.

Definition 2.2.5 (ActSel)**.** *Sei Bel die Menge aller* epistemic states*, sei Int aller* intention states *und sei Act die Menge aller* aktion states*, dann lautet die Funktionsvorschrifft:* $Bel \times Int \times Act \rightarrow A$.

Cowbot-BDI-Durchlauf Einen Cowbot-BDI- Durchlauf nennt man einen kompletten, funktionalen Ablauf des Cowbot-BDI-Modells. Ein Cowbot-BDI-Durchlauf beginnt mit eingehenden Informationen aus der Umwelt (als Umwelt werden auch andere Agenten verstanden). Diese Informationen werden in der *belief revision* mithilfe von entsprechenden Wissensoperatoren bearbeitet und die *beliefs* des Agenten wird aktualisiert. Der BRF-Komponente folgt die *desire generation*. Diese Komponente erzeugt bzw. motiviert *desires* des Agenten anhand von neuem Wissen. Die aktuellen *desires* werden an die *deliberation* weitergereicht. Diese wägt ab, ob das aktuell verfolgte *goal* verworfen werden und ein neues *desire* zum *goal* erhoben werden soll oder dessen Abarbeitung weiter geführt werden soll. Für das aktuell angestrebte *goal* muss ein Plan ausgewählt bzw. erstellt werden. Diese Aufgabe übernimmt der *planner*. Ein *goal* mit einem zuhehörigen Plan heißt *intention*. Das Cowbot-BDI-Durchlauf findet seinen Abschluss in der *action selection*. Diese Komponente realisiert jeweils eine *atomic intention* mit einer *atomic action* welche züruck an die Umwelt geschickt wird.

2.3 Belief Revision Function

F. Bienek, B. Jablkowski, T. Vengels

Die *belief revision function*(BRF) bildet die erste funktionale Komponente des Cowbot-BDI-Modells. Mit ihr beginnt der in dem Begriff des Cowbot-BDI-Durchlaufs definierte Prozessfluss. Im Allgemeinen realisiert diese Komponente die Schnittstelle des Agenten zur seiner Umwelt. Für alle eingehende Informationen, sowohl die perzeptuellen, die wir im folgenden als *percepts* bezeichnen werden, als auch diejenigen die mithilfe der Kommunikation von anderen Agenten propagiert wurden, bildet die BRF den Einstiegspunkt in das vorgestellte Agentenmodell. Die erhaltenen Informationen werden in eine für den Agenten gewählte Wissensrepräsentation überführt, in unserem Fall sind es die Erweiterten Logischen Programme(ELP's) und mithilfe entsprechender Wissensoperatoren in den *epistemic state* des Agenten eingepflegt und weiter verarbeitet. Hier sei anzumerken, dass der *epistemic state* kein Teil der BRF ist, vielmehr ist es eine ausgesonderte Komponente die den aktuellen Wissenszustand des Agenten widerspiegelt. Aus Anforderungsgründen genießt die BRF privilegierte Rechte bezüglich des *epistemic states*.

2.3.1 Schnittstellen zu anderen Komponenten

Die *belief revision function* bildet nicht nur eine externe Schnittstelle zur Außenwelt und ist für das Einpflegen und Verarbeiten vom Wissen zuständig. Sie berechnet zu einem *epistemic state* stehts genau ein *belief set*, so dass andere im Cowbot-BDI-Modell definierten Komponenten mit den *beliefs* des Agenten arbeiten können. Zudem bietet sie die Möglichkeit, spezielle Anfragen an den *epistemic state* zu stellen. Dies wird im späteren Teil genauer erläutert.

2.3.2 Einführung in die Revision

Bevor die eigentliche Funktionsweise der BRF Komponente vorgestellt werden kann, muss zunächst der Begriff „Revision“ konkretisiert werden. Dies erfolgt in drei Schritten: zunächst wird ein Ansatz beschrieben, mit dem sich Revision mathematisch erfassen lässt. Hierzu wird der Revisionsbegriff anhand der AGM Revisionstheorie vorgestellt. Darauf aufbauend wird untersucht, in welchem Kontext ein Agent sein Wissen revidiert. Abschließend wird mit erweiterter logischer Programmierung ein Ansatz vorgestellt, mit der Wissensoperationen praktisch umgesetzt werden können.

Motivation und Revisionsbegriff

Revision von Agentenwissen beschreibt allgemein Operationen, bei der ein Agent neues Wissen in sein vorhandenes Wissen aufnimmt. Um eine solche Operation umzusetzen, müssen sowohl grundlegende Probleme, als auch wünschenswerte Anforderungen an eine Revision erörtert und formuliert werden. Grundlegende Probleme sind hierbei: Wie wird das Agentenwissen repräsentiert (und welche

Auswirkungen hat dies auf die Revisionsoperation)? Wie vermeidet man Inkonsistenzen beim Einplegen neuen Wissens? Wie wird im Falle eines Konfliktes entschieden, welches Wissen verworfen werden soll?

AGM Revisionstheorie Ein formale Beschreibung einer Revisionsoperation liefert die AGM-Revisionstheorie [1]. In dieser Theorie werden drei Wissensoperationen und ihr Verhalten durch Postulate definiert. Um die Wissensoperationen zu konkretisieren, betrachten wir die Mengen K und A einer propositionalen Sprache L. O.B.d.A repräsentiere K das aktuelle Wissen eines Agenten, A eine zu integrierende Information, und $\neg A$ sei eine zu A widersprüchliche Information. In der AGM-Theorie werden drei Operationen für Expansion, Revision und Kontraktion auf Wissensmengen (abgeschlossene Mengen unter der Konsequenzrelation von L) anhand von Postulaten definiert. Der grundlegende Gedanke ist hierbei, dass Informationen unter Berücksichtigung des Prinzips der minimalen Änderungen (*minimal change*) eingepflegt oder entfernt werden. Es soll also so viel Wissen wie möglich erhalten bleiben, insbesonders soll nicht unbegründet Wissen entfernt werden.

Expansion $+$ Neues Wissen wird per Vereinigung zum Wissen eines Agenten hinzugefügt. Dies setzt voraus, dass das neue Wissen nicht inkonsistent zum vorhandenen Wissen ist. Die Expansion von K um A entspricht formal der Vereinigung auf Mengen unter der Konsequenzrelation: $K + A = Cn(K \cup A)$ [14].

Revision $*$ Die Revision fügt neues Wissen A in die Wissensbasis K eines Agenten und garantiert dabei die Konsistenz von K nach dieser Operation. Praktisch bedeutet dies, dass widersprüchliches Wissen zu A entfernt werden muss. Für die Revisionsfunktion $\star$ gelten folgende Postulate:

$(AGM*1)$ $K * A$ ist eine Wissensmenge

$(AGM*2)$ $A \in K * A$

$(AGM*3)$ $K * A \subseteq K \cup A$

$(AGM*4)$ wenn $\neg A \notin K$ dann $K + A \subseteq K * A$

$(AGM*5)$ $K * A$ ist widersprüchlich, wenn A inkonsistent ist

$(AGM*6)$ wenn $A \equiv B$ dann $K * A = K * B$

Hierbei formuliert $(AGM*1)$ die Anforderung, dass sich die Wissensdarstellung nicht ändert und das Ergebnis einer Revision tatsächlich wieder ein *belief set* ist. Postulat $(AGM*2)$ fordert, dass eingehende Informationen gegenüber vorhandenen Wissen priorisiert werden. Mit $(AGM*3)$ wird gefordert, dass eine Revision nicht mehr Wissen erzeugt, als durch eine Expansion erzeugt würde. Postulat $(AGM*4)$ fordert, dass eine Revision mittels Expansion erfolgen soll, falls A nicht mit K in Widerspruch steht. Postulat $(AGM*5)$ untermauert, dass das Ergebnis einer Revisionsfunktion konsistent ist. Allerdings nur, solange A selbst konsistent ist

und eine Revision überhaupt möglich wäre. Postulat $(AGM * 6)$ stellt sicher, dass das Ergebnis einer Revision von der Semantik einer Information abhängt, nicht aber von der syntaktischen Darstellung [14].

Kontraktion $\div$ Die Kontraktion entfernt Wissen aus der Wissensbasis eines Agenten. Hierbei soll A aus K entfernt werden. Die folgenden Postulate spezifizieren die Kontraktion:

$(AGM \div 1)$ $K \div A$ ist eine Wissensmenge

$(AGM \div 2)$ $K \div A \subseteq K$

$(AGM \div 3)$ wenn $A \notin K$ dann $K \div A = K$

$(AGM \div 4)$ ist A keine Tautologie, dann $A \notin K \div A$

$(AGM \div 5)$ $K \subseteq (K \div A) + A$

$(AGM \div 6)$ wenn $A \equiv B$ dann $K \div A = K \div B$

Postulat $(AGM \div 1)$ fordert, dass das Ergebnis der Kontraktion wieder eine Wissensmenge ist. Mit Postulat $(AGM \div 2)$ wird sichergestellt, dass bei der Kontraktion kein neues Wissen gewonnen wird. Postulat $(AGM \div 3)$ fordert, dass sich eine Wissensmenge nicht ändert, wenn die zurückzunehmende Information in ihr nicht vorhanden ist. Postulat $(AGM \div 4)$ fordert, dass zurückgenommenes Wissen nicht mehr Teil der Wissensmenge sein soll (nicht mehr logisch gefolgert werden kann). Eine Ausnahme bilden Tautologien, die immer gelten und somit nicht zurückgenommen werden können. In Postulat $(AGM \div 5)$ werden Kontraktion und Expansion in gewisser Weise invers zueinander betrachtet. Entfernt man zunächst A, und fügt es dann wieder hinzu, soll K mindestens dem Zustand vor der Kontraktion entsprechen. Postulat $(AGM \div 6)$ verlangt, dass das Ergebnis einer Kontraktion nicht von der Syntax abhängig ist [14].

Abschliessend sei erwähnt, dass die AGM-Revisionstheorie Wissensoperationen auf eine sich nicht ändernde Welt oder statische Situation beschreiben. Ein entsprechender Wissensoperator fügt neues Wissen A also unter der Berücksichtigung ein, das sowohl K als auch A sich auf die gleiche Situation beziehen. Beschreibt man Wissensoperationen über eine sich ändernde Welt, wo sich A unfd K auf verschiedene Zeitpunkte (Gegenwart und Vergangenheit) beziehen, spricht man von Update [13].

Revisionsansätze

Im Folgenden werden Revisionsansätze im Kontext von Multiagentensystemen erörtert. Hierzu wird kein Formalismus (wie die AGM Revisionstheorie) gesucht. Stattdessen werden Rahmenbedingungen im erörtert, unter denen ein Agent sein Wissen im Kontext von Multiagentensystemen revidiert. Der Schwerpunkt liegt also darin, herauszufinden wie vorhandene Revisionsparadigmen als Werkzeug zur Revision von Agentenwissen eingesetzt werden können oder erweitert werden müssen. In Multiagentensystemen können verschiedene Stufen oder Konzepte von Wissensrevision abgegrenzt werden [16]:

Single Belief Revision, SBR Die *single belief revision* (kurz SBR) betrachtet Revision in Szenarien mit lediglich einem Agenten. Zudem seien alle wahrgenommenen Informationen wichtiger als sein momentanes Wissen. Will ein Agent stehts neue Informationen als Wissen akzeptieren, kann er dies durch einen Wissensoperator tun, welcher die AGM-Theorie erfüllt.

Multiple Source Belief Revision Die *multiple source belief revision* (kurz MSBR) beschreibt Revisionsoperationen, bei der ein Agent neue Informationen von unterschiedlich glaubwürdigen Quellen empfängt. Nach Dragoni et al. [6] beeinflusst die Zuverlässigkeit einer Informationsquelle unmittelbar die Glaubwürdigkeit einer Information. Anhand von Bewertungen, einer Rangordnung über alle Informationsquellen (einschliesslich den revidierenden Agenten selbst), kann ein Agent entscheiden neues Wissen zu akzeptieren oder zu verwerfen. Für eine Revisionsfunktion bedeutet dies konkret, dass eine neue Information I nicht höchste Priorität geniesst. Stattdessen arbeitet eine solche Revisionsfunktion auf Tupeln der Form $Informationsquelle \times Information$ und weiteren Strukturen, die nötig sind um Informationsquellen (und das damit assoziierte Wissen) bewerten zu können.

Multi Agent Belief Revision Die *multi agent belief revision* (kurz: MABR) erweitert das Konzept der *multiple source belief revision* auf Agentengesellschaften. Gegenstand der Revision ist nun nicht nur das Wissen eines individuellen Agenten, sondern auchvon mehreren Agenten gemeinsam akzeptiertes Wissen, oder Gruppenwissen. Nach dem Ansatz von Kfir-Dahav und Tennenholtz [15] wird das Wissen eines Agenten in zwei Partitionen unterteilt, die *private domain* und *shared domain*. Jeder Agent i besitzt eine *private domain* PD_i, diese repräsentiert sein privates Wissen. Die *shared domain* SD repräsentiert das gemeinsame Wissen aller Agenten und sei als Schnitt über die *private domains* definiert, also $SD = \bigcap PD_i$. Bei dieser Definition von gemeinsamen und privaten Wissen kann es jedoch zu Problemen kommen. So müssen Agenten gleiches privates Wissen immer als gemeinsames Wissen instanziieren, ansonsten wäre die Definition der *shared domain* verletzt. Für die technische Umsetzung bedeutet dies zugleich, dass ein Revisionsoperator globalen Zugriff auf das private Wissen aller Agenten haben muss, was unweigerlich zu einem Verlust an Privatssphäre für jeden einzelnen Agenten führt.

Bezug zur PG In der Projektgruppe wurde ein Agentenmodell für ein kooperatives Agentensystem konzipiert und implementiert. In rein kooperativen Systemen arbeiten Agenten per se miteinander zusammen, insbesonders verfolgen sie keine Eigeninteressen, die zum Nachteil anderer befreundeter Agenten oder der Agentengesellschaft dienen. Sind alle Agenten gleichberechtigt, entfällt die Bewertung von Informationen nach Informationsquellen (in Szenarien mit Zeit kann diese stattdessen für die Aktualität oder Glaubwürdigkeit einer Information herangezogen werden). Schwerpunkt ist stattdessen die Einigung auf gemeinsame Ziele, welche die Agentengesellschaft verfolgen will. Die Ideen der SBR

können direkt auf die unmittelbare Wahrnehmung der Umwelt eines Agenten übertragen werden. In Szenarien ohne Wahrnehmungsfehler ist eine neue Information immer einer alten Wahrnehmung vorzuziehen (ansonsten sind Strategien zur Fehlerkorrektur nötig).

2.3.3 ELP - Motivation und Begriffserläuterung

Nach der allgemeinen kurzen Beschreibung von Wissensrevision in Agentensystemen, folgt die Vorstellung einer Sprache, welche es erlaubt Agentenwissen und Operationen auf Agentenwissen deklarativ darzustellen. Hierzu entschied sich die Projektgruppe für erweiterte logische Programme (Kurzform: ELP). Dieser Formalismus erlaubt die deklarative Darstellung von Agentenwissen. Das Wissen kann aus Fakten und Regeln, mittels derer ein Agent neue Fakten folgern kann, modelliert werden. Das eigentliche Wissen (*belief set*) wird mit Hilfe von *ASP solvern* berechnet, im Falle der Projektgruppe mit dem Programm DLV [22].

Syntax

Wir betrachten disjunktive erweiterte logische Programme. Dies sind Programme, deren Regeln allgemein folgender Form entsprechen:

$$L_{h_1} or \ldots or\, L_{h_m} \leftarrow L_{b_1}, \ldots, L_{b_n}, not\, L_{b_{n+1}}, \ldots, not\, L_{b_n+k} \tag{2.1}$$

Die Elemente von Regeln sind hierbei Literale. Ein Literal ist ein positives oder klassisch negiertes Atom. Die Literale L_{h_i} bilden den Kopf $H(r)$, die Literale L_{b_j} den Körper $B(r)$ einer Regel r. Im Kopf einer Regel können Literale disjunktiv verknüpft sein, in diesem Fall spricht man von disjunktiven erweiterten logischen Programmen. Die Literale im Körper einer Regel können positiv oder *default*-negiert sein, angezeigt durch *not*, dies erlaubt die Auswertung von Literalen mittels *negation as failure*. Je nach Mächtigkeit von $H(r)$ und $B(r)$ werden Regeln unterschiedlich interpretiert:

Fakten Fakten werden durch Regeln der Form

$$L_{h_1} or \ldots or\, L_{h_m} \leftarrow .$$

angegeben. Für die Körperliterale gilt folglich $B(r) = \varnothing$. In logischen Programmen sind Fakten stehts Teil der Antwortmenge, bei disjunktiven Faktenregeln werden entsprechend der Mächtigkeit von L entsprechend viele Antwortmengen generiert.

Regel Bei allgemeinen (respektive disjunktiven) Regeln sind sowohl $H(r)$ als auch $B(r)$ nichtleere Menge:

$$L_{h_1} or \ldots or\, L_{h_m} \leftarrow L_{b_1}, \ldots, L_{b_n}, not\, L_{b_{n+1}}, \ldots, not\, L_{b_n+k}$$

mittels dieser Regeln kann neues Wissen inferiert werden. So werden die Kopfliterale genau dann zu einer Antwortmenge eines logischen Programms, wenn alle Körperliterale erfüllt werden.

Constraint Eine Regel der Form

$$\leftarrow L_{b_1}, \ldots, L_{b_n}, not\, L_{b_{n+1}}, \ldots, not\, L_{b_n+k}$$

ist ein Constraint (genauer: *integrity constraint*). Constraints verbieten Wissen, ein Anwortmengenkandidat wird verworfen falls ein Constraint erfüllt wird. Insbesondere in Verbindung mit disjunktiven logischen Programmen können so mögliche Lösungen eingegrenzt werden. Als logische Programme formulierte Probleme können mittels Regeln und Constraints dann mittels *guess-check* Verfahren gelöst werden (Regeln produzieren hierbei möglicherweise ungültige Lösungen, Constraints verwerfen als ungültige Lösungen betrachtete Antwortmengenkandidaten noch vor der Propagierung zur Antwortmenge).

ELPs und Wissensrepräsentation

Nach der Vorstellung der Syntax folgen Beispiele, wie ELPs zur Darstellung und Bearbeitung von Agentenwissen genutzt werden können.

Agentenwissen Das Agentenwissen lässt sich deklarativ in Form von Fakten und Regeln ausdrücken. Fakten legen das Grundwissen fest, Regeln erlauben Schlussfolgerungen auf der Faktenmenge. Grundwissen einer Agentin, wie ihr Name ist und in welcher Gemeinschaft sie angehört, lässt sich wie folgt darstellen:

$$r_1 : iam(alice).$$
$$r_2 : agent(alice).$$
$$r_3 : team(alice, argonauts)$$

Darüber weiss Agentin *alice*, dass alle Agenten des Teams *argonauts* ihre Freunde sind:

$$r_4\ friend(X, alice) \leftarrow agent(X),\ team(X, argonauts).$$

Nichtmonotones Folgern Mittels der *default*-Negation ist nicht-monotones Schlussfolgern möglich. Dies bedeutet, dass in Gegenwart neuer Informationen die Wissensmenge eines Agenten nicht wachsen muss (oder kleiner wird, Abhängigkeit von den Regeln eines Programms). Ermöglicht wird dies durch den Operator *not*. Beispiel: wir modellieren die mentale Haltung eines Agenten, entweder Kühe treiben zu wollen, wenn er eine Kuh sieht (*herd*), oder nach ihnen zu suchen (*scout*), wenn er keine Kuh sieht.

$$r_1 : herd \leftarrow see(cow).$$
$$r_2 : scout \leftarrow not\ see(cow).$$

Angenommen, ein Agent sieht zunächst keine Kuh, so folgert er als Wissen $\{scout\}$. Nimmt er eine Kuh wahr, ändert sich sein Wissen zu $\{herd.$

$see(cow)\}$. In klassischen, nichtmonotonen Logiken hingegen könnte das Wissen nur wachsen, und er würde $\{scout.\ herd.\ see(cow)\}$ glauben. Das Programm kann auch mittels echter *closed world assumption* modelliert werden, dann wird r_2 substituiert durch:

$$r_{2_1} : \neg see(cow) \leftarrow not\ see(cow).$$
$$r_{2_2} : scout \leftarrow \neg see(cow).$$

Entscheidungsprobleme Entscheidungs (oder Suchprobleme) können mit Hilfe disjunktiver Programme formuliert und ausgewertet werden. Im folgenden Beispiel wollen drei Agenten Alice, Bob und Charlie eine Gruppe bilden. Alle fühlen sich gleichwertig, wollen daher ohne Einschränkung Anführer oder Gehilfe der Gruppe werden. Allerdings kommt nur eine Gruppe zu Stande, wenn genau ein Anführer werden kann. In ELPs kann dieses Problem wie folgt formuliert werden:

$$r_1 : form\_group(alice).$$
$$r_2 : form\_group(bob).$$
$$r_3 : form\_group(charlie).$$
$$r_4 : helper(X) \vee leader(X) \leftarrow form\_group(X).$$
$$r_5 : \leftarrow not\ \#count\{X : leader(X)\} = 1.$$

Die Regeln r_1 bis r_3 beschreiben, dass die drei Agenten eine Gruppe bilden möchten. Mit der Regel r_4 werden alle möglichen Konstellationen aus Helfern und Gruppenleitern gebildet. Die Regel r_5 grenzt die Antwortmengenkandidaten auf jene Kombinationen ein, in genau einen Gruppenleiter haben. In dem Beispiel werden bei mehr als einen Agenten mehrere Antwortmengen gebildet, da keine weitere Bewertung dieser Antwortmengen erfolgt kann in diesem Beispiel eine beliebige Antwortmenge als Ergebnis der Gruppenbildung genommen werden. Das im Körper von r_5 auftauchende Literal mit Präfix # stellt eine Aggregatfunktion dar (siehe [5]).

Auswertung logischer Programme

Im Zuge der PG benutzen wir ELPs unter Verwendung der Antwortmengensemantik. Die Berechnung der Antwortmengen geschieht mittels DLV [22], einem *solver* für disjunktive logische Programme. Eine Antwortmenge ist eine Menge von Fakten, die ein Modell eines Programms ist. Ein logisches Programm kann mehrere Antwortmengen als gültige Modelle haben (in dem Fall hat man disjunktive Programme und/oder Programme mit $default$-Negation), genau eine Antwortmenge oder keine. Die Konstruktion einer Antwortmenge S zu einem Programm P basiert hierbei auf dem Gelfond-Lifschitz Redukt P^S für logsiche Programme mit klassischer Negation und Disjunktion [10].

2.3.4 Aufbau der Belief Revision Function im Cowbot-BDI-Modell

Im Folgenden soll die Struktur der *belief revision function* im Cowbot-BDI-Modell näher beschrieben werden. Abbildung 2.3 gibt einen Überblick über die internen Komponenten der *belief revision function* (oben) und der *beliefs* (unten).

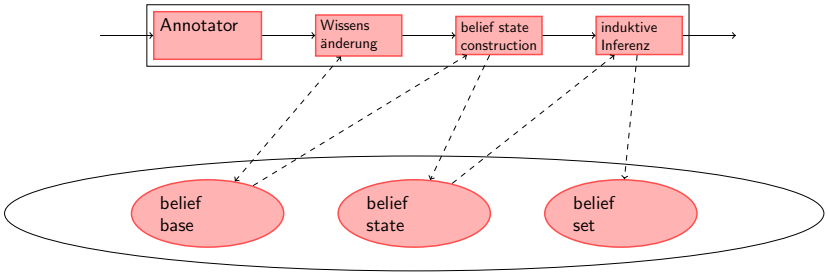

Abbildung 2.3: interner Prozessfluss der BRF- und *beliefs*-Komponenten

2.3.5 Datenkomponenten der BRF

Die Aufgabe der BRF ist es, die *beliefs* eines Agenten zu verwalten und konsistent zu halten. Der *epistemic state* eines Agenten (synonym für *beliefs*) wird also häufig von der BRF benötigt und geändert. Er besteht aus drei Teilen: *belief base*, *belief state* und *belief set*.

Belief base

Die *belief base* eines Agenten spiegelt das Wissen eines Agenten wider, welches im Laufe seines Lebenszyklus angesammelt wurde. Insbesondere können sich in der *belief base* auch Informationen befinden, die Widersprüche enthalten. Um diese Informationen strukturiert zu halten, definieren wir eine Container-Struktur:

Definition 2.3.1 (Information Object)**.** *Ein* information object I *ist ein Tupel* $(P, Meta)$*.* P *ist ein ELP, Meta eine Menge Meta Informationen.*

Ein *information object* ist also ein Konstrukt, welches eingehende Informationen (in Form eines ELPs) und zugehörige Meta-Informationen (etwa die Zeit oder die Quelle der Nachricht) miteinander verknüpft. Die so entstehende Menge von *information objects* wird dann in der *belief base* gespeichert.

Definition 2.3.2 (Belief Base BB)**.** *Die* belief base *ist eine Menge* information objects I*,* $BB = \{I_0, \ldots, I_n\}, n \in \mathbb{N}$*.*

Die erste Komponente des *epistemic states* wird also durch eine Menge von Is gebildet. In ihr sammelt der Agent Informationen, selbst wenn sie veraltet

oder inkonsistent sein sollten. Ein Merkmal dieser Struktur soll hier noch näher erläutert werden.

Granularität Unter der Granularität der *beliefs* verstehen wir, wieviele Informationen in einem ELP kodiert sind. Für ein $I = (P, META)$ sind alle Fakten, Regeln und Constraints in P mit denselben Annotationen $META$ verknüpft. Deshalb ist es sinnvoll, nach Regeln und Fakten zu trennen. Wir halten einen Fakt oder eine Regel pro *information object* für sinnvoll.

Belief State

Der *belief state* eines Agenten ist die Komponente der *beliefs*, in welcher alle Informationen der *belief base* vereinigt werden. Er wird gebildet, indem ein Operator aufgerufen wird, der Inkonsistenten in den einzelnen ELPs der *belief base* behebt und das Ergebnis in einem einzelnen erweiterten logischen Programm sammelt.

Definition 2.3.3 (Belief State $P^\star$)**.** *Der* belief state *ist ein konsistentes erweitertes logisches Programm.*

Der *belief state* entsteht also durch eine Zusammenführung alles Wissen, welches einem Agenten zur Verfügung steht. Dies können Fakten und Regeln aus den konkreten *information objects* sein, aber auch Regeln, die für die Behebung von Inkonsistenzen von einem Operator generiert wurden. Näheres zu Form und Struktur eines *belief states* wird in der Sektion 2.3.6 erläutert.

Belief Set

Das *belief set* ist der letzte Teil der Datenkomponenten. Es wird durch Anwendung eines *answer set solvers* (zB. DLV, [22]) auf den *belief state* gebildet.

Definition 2.3.4 (Belief Set BS)**.** *Das* belief set *ist ist eine konsistente Menge Fakten, es repräsentiert das geglaubte Wissen eines Agenten.*

Das *belief set* repräsentiert also das Wissen, dass ein Agent als in seinem momentanen Zustand als wahr annimmt (Im Gegensatz zur *belief base*, welche, wie erwähnt, auch inkonsistente Informationen in verschiedenen *information objects* speichern kann). Diese Fakten bilden im Ablauf eines Cowbot-BDI-Durchlaufs die Schnittstelle zu den nachfolgenden Komponenten. Genaueres zur Inferenz mit DLV wird unter Sektion 2.3.6 beschrieben.

2.3.6 Funktionale Komponenten der BRF

Nach der Beschreibung der Datenkomponente der *belief revision function* wollen wir nun die funktionalen Komponenten genauer vorstellen. Den Einstiegspunkt für die zu bearbeitenden Informationen bildet der *annotator*. Nach ihm folgt der *knowledge operator*, dann die *belief state construction* und schließlich die *inference*. Die folgende Abbildung soll den Ablauf des Prozessflusses in der BRF grafisch veranschaulichen.

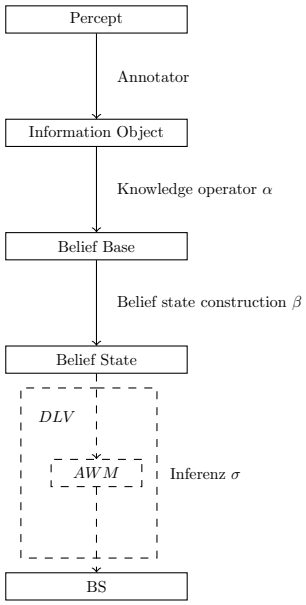

Abbildung 2.4: Schematischer Ablauf der BRF Funktions-Komponenten

Annotator

Im Allgemeinen bildet der *annotator* eine eingehende Information auf ein *information object* ab. Der *annotator* beginnt, indem er Daten in die gewählte Wissensrepräsentation überführt. Dabei wird aus den Eingabedaten ein erweitertes logisches Programm erstellt. Weiterhin werden auch bestimmte *meta informations*, wie Zeit oder Quelle der Information, an das ELP angehängt. Die Funktionsvorschrift für den *annotator* sieht wie folgt aus:

Definition 2.3.5 (Annotator)**.** *Der annotator ist eine Funktion* $f_I : Per \times 2^{Meta} \rightarrow 2^I$.

Knowledge Operator

Die durch den *annotator* erstellten *informaton objects* sollen nun in die *belief base* des Agenten integriert werden. Dies übernimmt die folgende Komponente der *belief revision function*: der *knowledge operator*. Er ist dafür zuständig die vom *annotator* erhaltenen *information objects* in die *belief base* einzupflegen. Dabei besteht die Möglichkeit dies mithilfe verschiedener Vefahren durchzuführen. Die dafür im Cowbot-BDI-Modell vorgesehenen Operatoren sind *expansion* und *update*. Anzumerken sei an dieser Stelle, dass der *knowledge operator* sich nicht um die Konsistenz der Daten in der *belief base* kümmert. Die Anforderung der Konsistenz bezieht sich erst auf den *belief state*. Die Funktionsvorschrift für den *knowledge operator* sieht wie folgt aus:

Definition 2.3.6 (Knowledge operator α)**.** *Ein* knowledge operator *ist eine Funktion der Form* $\alpha : 2^{BB} \times 2^{I} \to 2^{BB}$

Expansion Operator Im Fall, dass für den *knowledge operator* die *expansion* ausgewählt wird, arbeitet er wie folgt:

Beispiel 1. Das folgende *information object* soll in die *belief base* $BB_{old} = \{I_0, \ldots, I_i\}$ eingepflegt werden. Dann ist das Ergebnis der *expansion* die neue *belief base* $BB_{new} = \{I_0, \ldots, I_i, I_j\}$ für $i, j \in \mathbb{N}, i < j$.

$$expansion : BB_{old} \cup \{I_j\} = BB_{new}$$

Die *belief base* wird also um das neue *informaton object* lediglich erweitert ohne jegliche Überprüfung auf Konsistenz.

Update Operator Der *update operator* verfolgt einen etwas anderen Ansatz. Dieser Operator überprüft für jede neue, einzupflegende Information die *belief base* auf das Vorkommen dieser Information. Im Falle einer Übereinstimmung wird nach einem früher bestimmten Kriterium, meist einer oder mehrerer *meta informations*, aktualisiert. Im Falle, dass die *belief base* diese Information nicht enthält, wird das *information object*, wie bei einer *expansion* hinzugefügt. Das resultiert in einer kleineren *belief base*, hat aber den Nachteil, dass man nicht wie im Falle einer *expansion*, die komplette Geschichte des Wissens zur Verfügung hat. Es folgen ein Beispiel für die Funktionsweise des *update operator*. Als Aktualisierungskriterium wird die *meta information* Zeit(t_n) verwendet. Das Beispiel deckt den ersten, oben beschriebenen Fall ab:

Beispiel 1. Das folgende *information object* $I_j = \{ison(id, x, y), (source, t_n)\}$ soll mithilfe des *update operators* in die *belief base* $BB_{old} = \{I_0, \ldots, I_i\}$ einpflegt werden. Das Ergebnis ist die *belief base* $BB_{new} = \{I_0, \ldots, I_i\}$ für $i, j, k \in \mathbb{N}, j = i - k \geq 0$, falls es ein *information object* I_l in BB_{old} gibt mit $I_l = \{ison(id, x, y), (source, t_m)\}$ für $l, m, n \in \mathbb{N}, l = j,\ t_m < t_n$

$$update : BB_{old} + I_j = BB_{new}$$

Belief State Construction

Die Idee hinter dem *belief state constructor* ist es aus allen Informationen, die sich in einer *belief base* befinden, ein konsistentes erweitertes logisches Programm zu konstruieren. Solch ein ELP haben wir als den *belief state* definiert. Dieser Schritt ist erst die Einleitung zu der eigentlich Inferenz. Der Operator soll Konflikte innerhalb der *belief base* erkennen und diese aufheben. Wie diese Konflikte erkannt und gehandhabt werden, hängt stark von dem Ansatz ab für welchen man sich entschieden hat. Die allgemeine Funktionsvorschrift für den *belief state constructor* sieht wie folgt aus:

Definition 2.3.7 (Belief State Constructor)**.** *Der* belief state constructor $\beta : 2^{BB} \rightarrow P^*$ *erzeugt aus der Belief Base den Belief State.*

Ein Verfahren, welches sich für die Implementierung der Funktionsvorschrift des *belief state constructors* eignet, wurde von Eiter, Fink, Sabbatini, Tompits in [9] vorgestellt. Dieser Ansatz ist unter dem Begriff *causal rejection* bekannt. Der *causal rejection operator* erstellt aus einer Sequenz von erweiterten logischen Programmen ein einziges erweitertes logisches Programm. Die Idee hinter dem Prinzip der begründeten Ablehnung ist es nur solche Regeln r in das neue Programm zu übernehmen, welche nicht im Konflikt mit einer aktuelleren, nicht abgelehnten Regel r' stehen. Wie *causal rejection* genau in unser Cowbot-Framework umgesetzt wurde, wie wir Konflikte definieren und handhaben wird später in Sektion 4 beschrieben. Wir wollen nun das Verfahren im Allgemeinen vorstellen.

Causal Rejection Sei P eine Folge $(P_1, ..., P_n)$ von erweiterten logischen Programmen über dem Alphabet A. Über der Folge P= $(P_1, ..., P_n)$ gibt es eine Ordnung mit $j < i$; $i, j \in \mathbb{N}$. Das Ziel des Operators ist es ein einzelnes Programm $P_\lhd$ aus der Sequenz P zu erzeugen. Damit man die abgelehnten Regeln markieren bzw. darstellen kann, muss das Basisalphabet A um das Prädikat $rej(r)$ erweitert werden, wobei r für die abgelehnte Regel steht. Desweiteren ist es nötig die Atome A aus den einzelnen Programm der Sequenz P mit A_i zu indizieren, was die Anpassung der Literale L zu L_i impliziert. Das Programm $P_\lhd$ kann nun in vier Schritten aus der Sequenz P konstruiert werden, dies geschieht wie folgt:

1 alle Constraints aus $P_i, 1 \leq i \leq n$

2 $\forall r \in P_i, 1 \leq i \leq n$

$$L_i \leftarrow body(r), \text{not}\, rej(r) \quad \text{falls}\, head(r) = L$$

3 $\forall r \in P_i, 1 \leq i < n$

$$rej(r) \leftarrow body(r), \neg L_{i+1} \quad \text{falls}\, head(r) = L$$

4 $\forall$ Literal $L \in P, 1 \leq i < n$

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

Im ersten Schritt werden alle *constraints* aus allen P_i in das neue Programm $P_\lhd$ übernommen. Im zweiten Schritt wird sichergestellt, dass nur Regeln übernommen werden, welche nicht abgelehnt wurden, dies geschieht mithilfe von *not* $rej(r)$. Im Schritt drei wird definiert, wann eine Regel abgelehnt werden soll. Nämlich gerade dann, wenn ihr Rumpf erfüllt ist und sie im Konflikt mit einer neueren, nicht abgelehnter Regel steht. Im letzten Schritt werden noch die Literalen verkettet. Das Ganze soll nun anhand eines Beispiels verdeutlicht werden.

Beispiel 3. Gegeben seien die beiden erweiterten logischen Programme P_1 und P_2.

$P_1 = \{r_1 : sleep \leftarrow not\ tv\_on;\ r_2 : night \leftarrow;\ r3 : tv\_on \leftarrow;\ r4 : watch\_tv \leftarrow tv\_on\}$
$P_2 = \{r_5 : \neg tv\_on \leftarrow power\_failure;\ r_6 : power\_failure \leftarrow\}$

Nun konstruieren wir das dazu gehörige Programm $P_\triangleleft$ gemäß den oben genannten Konstruktionsregeln.

$P_\triangleleft = \{$

$r_1 : sleep_1 \leftarrow not\ tv\_on, not\ rej(r_1)$
$r_2 : night_1 \leftarrow not\ rej(r_2)$
$r_3 : tv\_on_1 \leftarrow not\ rej(r_3)$
$r_4 : watch\_tv_1 \leftarrow tv\_on, not\ rej(r_4)$
$r_5 : \neg tv\_on_2 \leftarrow power\_failure, not\ rej(r_5)$
$r_6 : power\_failure_1 \leftarrow not\ rej(r_6)$
$r_7 : rej(r_1) \leftarrow not\ tv\_on, \neg sleep_2$
$r_8 : rej(r_2) \leftarrow \neg night_2$
$r_9 : rej(r_3) \leftarrow \neg tv\_on_2$
$r_{10} : rej(r_4) \leftarrow tv\_on, \neg watchtv_2$
$r_{11} : sleep_2 \leftarrow sleep_1$
$r_{12} : sleep \leftarrow sleep_1$
$r_{13} : night_2 \leftarrow night_1$
$r_{14} : night \leftarrow night_1$
$r_{15} : tv\_on_2 \leftarrow tv\_on_1$
$r_{16} : tv\_on \leftarrow tv\_on_1$
$r_{17} : power\_failure \leftarrow power\_failure_1$

$\}$

Die Regeln r_1 bis r_6 wurden aus der zweiten Konstruktionsvorschrift erzeugt. Die Regeln r_7 bis r_{10} aus der dritten Vorschrift und die restlichen aus der vierten. Da es im Beispiel keine *constraints* gibt findet der erste Konstruktionsschritt keine Anwendung. Die zum diesen Programm gehörige Antwortmenge sieht wie folgt aus:

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

Wir erinnern uns, dass das Alphabet A zu $A*$ erweitert wurde um *casual rejection* zu ermöglichen. Deswegen muss die Antwortmenge S' noch auf das Ursprungsalphabet projeziert werden.

Definition 2.3.8 (Antwortmenge)**.** *S ist eine Antwortmenge der Sequenz $P = (P_1, ..., P_n)$ gdw. S=S'∩A für eine Antwortmenge S' von $P_\triangleleft$.*

Aus dieser Definition ergibt sich die Antwortmenge S zu P wie folgt aussieht:

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

Antwortmengenwahl Bei dem Verfahren der *causal rejection* können mehrere Antwortmengen auftreten, da prinzipiell verschiedene Regeln aus $P_{\lhd}$ verwerfbar sind um Konflikte aufzulösen. Update-Programme in der bisher vorgestellten Form ignorieren zudem das Prinzip der minimalen Änderung. Bei einem Update eines Programms P_1 durch P_2 scheint es aber sinnvoll, jene Antwortmengen zu wählen, bei denen möglichst wenig Regeln aus P_1 durch P_2 verworfen werden.. Eiter et. al haben in [9] hierzu minimale und strikt minimale Antwortmengen definiert. Bevor diese erläutert werden können, führen wir zuerst die Idee des *rejection set* ein:

rejection set $Rej(P,S)$ Zu einer Update-Sequenz $P = (P_1, \ldots, P_n)$ über A, und $S \subseteq Lit_A$ sei $Rej(S,P)$ wie folgt definiert:

$$\begin{aligned} Rej(S,P) &= \bigcup_{i=1}^{n} Rej_i(S,P) \\ Rej_n(S,P) &= \varnothing \\ Rej_i(S,P) &= \{r_i \in P_i | \exists r' \setminus P_j, j > i, r, r' \text{ stehen in Konflikt}, \\ &\qquad S \vDash B(r) \cup B(r')\} \end{aligned}$$

Offensichtlich enthält $Rej(S,P)$ genau jene Regeln, die in P verworfen werden. Darauf aufbauend kann eine Antwortmenge S_1 einer anderen Antwortmenge S_2 vorgezogen werden, wenn $Rej(S_1,P)$ weniger Regeln als $Rej(S_2,P)$ enthält. Dies kann zur Definition von Minimalität genutzt werden:

Definition 2.3.9. *Eine Antwortmenge S mit $Rej(S,P)$ ist genau dann minimal, wenn es keine weitere Antwortmenge S' mit $Rej(S',P) \subset Rej(S,P)$ exisiert.*

Eine Auswahl gemäss Minimalität ist jedoch nicht stark genug, den Charakter von Update-Sequenzen wiederzugeben. Sind die *rejection sets* disjunkt, ist kein Vergleich dieser möglich. Um die Ordnung von Update-Sequenzen zu erhalten, sollte die Priorität der verworfenen Regeln geprüft werden. Berücksichtigt man unter minimalen Antwortmengen lediglich solche, welche neueste Regeln erfüllen, führt dies zu *strikter Minimalität*.

Definition 2.3.10. *Eine Antwortmenge S ist strikt minimal und wird S' vorgezogen, wenn es ein i gibt mit gilt:*

$$\begin{aligned} Rej(S,P) &\subset Rej(S',P) \\ Rej_j(S,P) &= Rej_k(S',P) \forall k \in \{i+1, \ldots, n\} \end{aligned}$$

Gegenbeispiel zu *causal rejection* Der zuvor vorgestellte Algorithmus eignet sich nicht für alle Szenarien. Ein Nachteil ist, dass er keine Constraints berücksichtigt. Dies soll an folgendem Beispiel verdeutlicht werden: Wir beschreiben die Welt mittels $ison(Object, X, Y)$, es gilt $Object \in \{empty, agent\}$

und $X, Y \in \{1, 2\}$. Ein Agent sammelt in diskreten Zeittakten Informationen über die Beispielwelt:

$$P_{t_1} = \{ison(empty, 1, 1).ison(agent, 1, 2).\} \tag{2.2}$$

$$P_{t_2} = \{ison(agent, 1, 1).ison(empty, 1, 2).\} \tag{2.3}$$

Ein einfacher *causal rejection* erzeugt aus der Update-Sequenz folgende Antwortmenge (projiziert):

AWM = {ison(empty,1,1), cell(empty,1,2), cell(agent,1,2), cell(agent,1,1)}

Diese ist syntaktisch widerspruchsfrei, insoweit arbeitet die *causal rejection* korrekt. Allerdings treten nun einige Probleme auf. Es liegen nun mehrere Informationen über eine Zelle vor. Dies ist insbesonders für Planungsprobleme kritisch, zum Beispiel ist ein *path finding* Algorithmus wie A^* darauf angewiesen, freie und belegte Zellen eindeutig zu kennen. Explizite Aufnehmen von Zeitinformationen in die Prädikate ist ebenfalls keine saubere Lösung, da ein eigentliches Ziel der Revision, aktuelle Weltsicht vermitteln, dann auf BDI Komponenten verschoben wird, die mit dem Agentenwissen arbeiten sollen. Fügt man der obigen Programmsequenz ein Constraint der Form

$$P_{t_3} = \{\leftarrow ison(empty, X, Y), ison(agent, X, Y).\} \tag{2.4}$$

hinzu, wird keine Antwortmenge mehr generiert. Daraus folgt die Notwendigkeit, einen Revisionsalgorithmus zu entwickeln, der Constraints behandelt. Dies ist besonders bei Programmen nötig, die überwiegend aus Fakten bestehen. In diesem Fall sind die unter Konstruktionsschritt zwei und drei Regelteile $body(r)$ leer, die *causal rejection* berücksichtigt dann lediglich das Vorzeichen pro Literal. Im Endeffekt wird für jedes Literal lediglich geprüft, ob es positiv ist oder klassisch negiert ist. Insofern wird die Konfliktauflösung zwischen Regeln auf die Auflösung syntaktischer Konflikte zwischen Paaren inkonsistenter Literale reduziert.

Constraint-Behandlung Im Zuge der Erweiterung der *causal rejection* um explizite Constraint-Behandlung wurde zunächst evaluiert, wie der Algorithmus modifiziert werden kann. Ziel war es, mittels Constraints explizit Fakten zu verbieten, bezogen auf den Algorithmus der *causal rejection* kann dies durch die gezielte Erzeugung von *reject*-Prädikaten für die zu eliminierenden Literale geschehen. Hierzu entschieden wir uns für eine Umformung der Constraints in *reject*-Regeln, die auf Ideen der *revision programming* beruht. Dieser von Marek und Truszczynski [17] entwickelte Formalismus, von Pivkinna et al. [20] im Kontext der Revision von Agentenwissen erweitert, beschreibt die Revision von Agentenwissen auf Basis von Constraints. Das Agentenwissen sei hierbei in Form von Mengen von Atomen definiert, und Constraints werden in Form sogenannter Revisionsregeln deklariert. Elemente von Revisionsregeln sind die Revisionsliterale **in** und **out**, welche die Anwesenheit, respektive Abwesenheit,

eines Atoms im Wissen des Agenten fordern. Revisionsprogramme bestehen folglich aus Revisionsregeln der Form

$$\mathbf{in}(a) \leftarrow \mathbf{in}(a_1), \ldots, \mathbf{in}(a_i), \mathbf{out}(b_1), \ldots, \mathbf{out}(b_j)$$
$$\mathbf{out}(a) \leftarrow \mathbf{in}(a_1), \ldots, \mathbf{in}(a_i), \mathbf{out}(b_1), \ldots, \mathbf{out}(b_j)$$

Revisionsprogramme beschreiben kurz gesagt die Transformation von Faktenmengen unter Revisionsprogrammen. Wir vertiefen diesen Ansatz nicht weiter, greifen jedoch die Idee der **out**-Regel auf. Diese definiert genau, welches Element im Fall eines Konfliktes entfernt werden muss, nämlich der Kopf einer verletzten Revisionsregel. Dies führt zu einer sehr pragmatischen Lösung, wie wir Constraints in erweiterten logischen Programmen, auf Basis der Regeltransformationen der *causal rejection*, umformen können. Wir reinterpretieren die Constraints als explizite Unterdrückung eines Literals, Ziel ist hierbei immer das erste Literal:

$$\leftarrow B_0, B_1, \ldots, B_n$$
$$rej_{B_0} \leftarrow B_1, \ldots, B_n$$

Zu Beachten ist, dass nun nicht direkt Regeln, sondern Fakten mittels eines *reject*-Prädikates unterdrückt werden sollen. Insofern muss die Programmkonstruktion leicht abgeändert werden, um *reject*-Prädikate sowohl aus Regelverletzungen als auch Constraint-Verletzungen zu erkennen und zu akzeptieren. Dies führt zu kleineren Modifikationen des Propagations- und Verkettungsmechanis von Literalen und Hilfsliteralen. Das Ergebnis dieser Umformung nennen wir *Super Rejection*, und stellen nun die Umsetzung mittels erweiterter logischer Programmierung vor.

Super Rejection Gegeben sei eine Programmfolge $P = P_1, \ldots, P_n$. Zu jedem Literal L sei $L(i)$ seine indizierte Version. Der Index (i) steht dabei für die Position von L in der Programmfolge P, dieser Index dient zugleich als Priorisierung eines Literals. Mit rej_X werden zusätzliche Literale eingeführt, die Konflikte zwischen Regeln markieren. Es wird folgendes Verfahren angewendet um ein konsistentes ELP P^* zu bauen:

1. Für alle Regeln r_i mit Kopfliteral $h(r_i)$ und Körper B_i aus P:

$$h(r_i, i) \leftarrow B_i. \tag{2.5}$$

2. Für alle Kopfliterale $h(r_i)$ aus P:

$$h(r_i) \leftarrow h(r_i, i), not\, rej_{h(r_i)}. \tag{2.6}$$

3. Für alle Kopfliterale $h(r_i)$ aus P:

$$rej_{h(r_i)} \leftarrow max\{k|h(r_i,k)\}, max\{u|\neg h(r_i,u)\}, k < u, not\, rej_{\neg h(r_i)} \tag{2.7}$$

4. Für alle Constraints c_i der Form $\leftarrow b_0(c_i), B(c_i), P(c_i)$:

$$rej_{b_0(c_i)} \leftarrow b_0(c_i, \_), B(c_i, i), not\, rej_{B(c_i,i)}, P(c_i). \tag{2.8}$$

Schritt eins sagt aus, das jedes Kopfliteral H einer allgemeinen Regel oder eines Fakts in seine indizierte Version $H(i)$ geführt wird. Im nächsten Schritt wird sichergestellt, dass H nur in einer Antwortmenge auftaucht, falls es nicht verworfen wird. In Schritt drei werden die eigentlichen *reject*-Regeln erzeugt. Ein Literal wird dann verworfen, wenn sein Inverses mit einer höheren Priorität vorliegt und selber nicht verworfen wird. In Schritt vier werden Constraints umgeformt, das erste Literal in einem Contraint soll dann verworfen werden wenn die restlichen Contraintliterale erfüllt werden. $P(c_i)$ beschreibt hierbei Prädikate, die zusätzlich bei der Auswertung eines Constraints hinzugezogen werden, ohne selbst Teil des Regelumformungsprozesses zu sein. Fehlen diese Prädikate im ursprünglichen Programm bei Constraints mit mindestens zwei Körperliteralen, wird in der Implementierung automatisch ein Prioritätsvergleich anhand der Indizes von $b(c)_0$ und $b(c)_1$ durchgeführt und somit entschieden, ob $b(c)_0$ verworfen wird.

Antwortmengen der Super Rejection Durch die explizite Umformung von Constraints wird dessen eigentlicher Mechanismus, Antwortmengenkandidaten zu verwerfen, ausser Kraft gesetzt. Dies kann bei disjunktiven Programmen zu unerwünschten Seiteneffekten in Form von leeren Antwortmengen führen. Die Constraint-Behandlung führt zu einem weitereren Effekt. Es ist nicht nachvollziehbar, ob eine Regel aufgrund eines Konfliktes mit einer anderen Regeln, oder gewollt durch ein Constraint verworfen wird. Die Constraint-Behandlung erfolgt global auf Programmebene, und ist nicht an eine Ordnung einer Programmsequenz gebunden. Auf Grund dessen ist die Analyse von möglichen *rejection sets* nicht aussagekräftig genug. Andererseits erlaubt der Constraint-Mechanismus, bei entsprechend sorgfältig geschriebenen Programmen, für eine ausreichende Filterung, so dass jede Antwortmenge als gleichwertig angesehen werden kann.

Inferenz mit DLV

Die letzte Stufe der BRF-Komponente befasst sich mit der Generierung des *belief sets* aus dem *belief state*. Dies geschieht in den folgenden Schritten:

1. Aufruf von DLV. Als Argument wird das Programm P^* übergeben. DLV berechnet zu diesem Programm dann eine (oder mehrere) Antwortmenge(n).

2. Auswerten der Antwortmenge. Im Zuge der *belief state construction* können zusätzliche Regeln und Fakten erzeugt worden sein, um zu garantieren dass P^* konsistent ist. Diese Fakten sind kein Teil des Wissens eines Agenten, folglich sollten sie nicht im *belief set* auftreten. Dieser Schritt entspricht der am Beispiel der *causal rejection* vorgestellten Projektion einer Antwortmenge von einem erweiterten Alphabet auf ein gängiges Alphabet.

3. Auswahl und Propagation einer Antwortmenge AWM zum *belief set* BS. Da sich Antwortmengen und das *belief set* strukturell nicht unterscheiden (Menge von Literalen), entspricht dieser Schritt einer Zuweisung $BS = AWM$. Durch die *belief state construction* ist sichergestellt, das ein konsistentes logisches Programm erzeugt wird, somit ist das Vorhandensein einer (evtl. leeren) Antwortmenge garantiert. Im Falle mehrerer Antwortmengen muss das verwendete Verfahren zur Konstruktion von P^* definieren, welche Antwortmenge allen anderen vorzuziehen ist.

2.3.7 Beispiel: Revision in einer Kachelwelt

Wie bereits zuvor in Abschnitt 2.3.6 erwähnt, wird das Wissen eines Agenten aus dem *belief state* gefolgert. Dies ist ein logisches Programm P^*, welches aus einer Folge von logischen Programmen $P_1, \ldots, P_n$ zusammengebaut wird. Mit der *super rejection* wurde eine Möglichkeit vorgestellt, P^* zu konstruieren, welche zudem die Möglichkeit bietet, störendes Wissen mittels Constraints zu verwerfen. Wir beschreiben zunächst wünschenswerte Eigenschaften an ein Programm, das Wahrnehmungen eines Agenten möglichst sinnvoll revidiert.

Behandlung von Zellen

Ein Schwerpunkt liegt auf der adequaten Behandlung der Umwelt, ein Agent nimmt hierbei neue Informationen über Zellen in Form von Fakten wahr. Folgende Anforderungen stellen sich an einen Wissensoperator, der das Wissen von Agenten revidiert:

- Verwerfung redundanter (veralteter) Informationen. Beobachtet ein Agent eine leere Zelle (eine Zelle gleichen Inhalts) über mehrere Zeittakte, so soll nur die aktuellste Beobachtung Teil seines Wissens sein.

- Trägheitsprinzip für nicht neu erfasste Zellen. Die Information, dass eine konkrete Zelle existiert, bleibt bestehen, selbst wenn sie nicht im aktuellen Sichtfeld des Agenten ist. Dies bezieht sich ebenfalls auf die Position einer Kuh, ein Agent glaubt so lange, dass eine Kuh auf einem Feld steht, bis er eine aktuellere Positionsangabe der Kuh wahrnimmt.

- Zellwiederherstellung, ein Spezialfall der bei beweglichen Objekten wie Kühen auftreten kann. Dieser Fall tritt immer am Rande des Sichtfelds eines Agenten auf und geht davon aus, das nur ein bewegliches Objekt auf einer Zelle gesichtet wurde. Bewegt sich der Agent und das beobachtete Objekt mit dem Agenten, wird aufgrund von Eigenschaft 1 die alte Information über das Objekt verworfen. Ist dies die einzige Information über die Zelle, nimmt ein Agent an, die Zelle sei leer (*empty*). Abbildung 2.5 zeigt diesen Spezialfall.

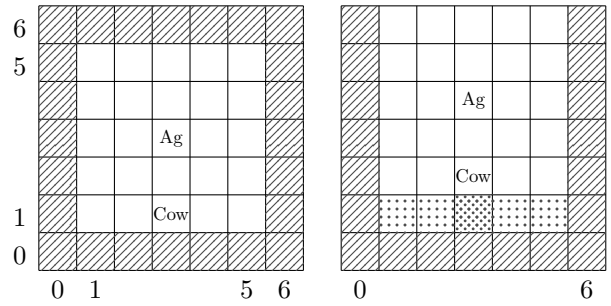

Abbildung 2.5: Die Abbildung zeigt einen Sonderfall. Über Zelle (3,1) weiss der Agent in Schritt 1 lediglich, dass dort eine Kuh ist. Bewegen sich Agent und Kuh einen Schritt nach oben, revidiert der Agent sein Wissen und verwirft die Information, dass bei (3,1) eine Kuh war. Stattdessen wird die Zelle (3,1) als leere statt unbekannte Zelle wiederhergestellt.

Alternative Behandlung von Zellen

Die vorgestellten Verfahren der *belief state construction* haben einen Nachteil. Sie sind für allgemeine Programmsequenzen ausgelegt, sie garantieren ein syntaktisch konsistentes Programm. Konflikte zwischen Regeln werden auf Basis einer Ordnung der Eingabeprogramme aufgelöst. Allerdings kann es sinnvoll sein, angepasste Verfahren zu entwickeln, die sich speziell auf ein Problem anstatt allgemeiner Problemsequenzen zugeschnitten sind. Dies erlaubt die Nutzung von Strukturinformationen über ein zu lösendes Problem, das in der Regel in den Prädikaten selbst kodiert ist. Der Nachteil dieser Methode ist selbstverständlich, dass man kein allgemeines Verfahren nutzt, sondern ein Programm oder einen speziellen Wissensoperator für eine Klasse von Problemen einschränkt. Desweiteren sind solche Verfahren sehr zeitaufwendig in der Wartung, wenn es zu Änderungen in der Wissensrepräsentation kommt.

Algorithmus für Zellen Im folgenden wird ein logisches Programm vorgestellt, was die mittels *ison* beschriebenen Zellinhalte behandelt. Zudem werden die in Abschnitt 2.3.7 erörterten Sonderfälle abgefangen.

für alle $ison(ID, X, Y, T)$ Fakten:

$$r_1 : ison'(ID, X, Y, T).$$

Hilfsregeln über Zellinhalte:

$$r_2: cowseen(C,T) \leftarrow ison(C,\_,\_,T), cow(C).$$
$$r_3: cowgone(C,T) \leftarrow cowseen(C,T), cowseen(C,T'), T < T'.$$
$$r_4: celltime(X,Y,T) \leftarrow ison(\_,X,Y,T)'.$$
$$r_5: newer(X,Y,T) \leftarrow celltime(X,Y,T), celltime(X,Y,T'), T < T'.$$

Inferenzregeln

$$r_6: ison(ID,X,Y,T) \leftarrow not\, rej(X,Y,T), ison'(ID,X,Y,T).$$
$$r_7: ison(empty,X,Y,0) \leftarrow recovercell(X,Y), not\, knowncell(X,Y).$$

reject und *recovery* Regeln:

$$r_8: rej(X,Y,T) \leftarrow newer(X,Y,T).$$
$$r_9: rej(X,Y,T) \leftarrow cowgone(ID,T), cow(ID), ison'(ID,X,Y,T).$$
$$r_{10}: recovercell(X,Y) \leftarrow rej(X,Y,\_).$$
$$r_{11}: knowncell(X,Y) \leftarrow ison(\_,X,Y,T), T \geq 1.$$
$$r_{12}: ison(empty,X,Y,0) \leftarrow recovercell(X,Y), not\, knowncell(X,Y).$$

In Regel $r1$ wird ein Fakt umgeschrieben, eine ähnliche Vorgehensweise wie zum Beispiel bei *causal rejection*, allerdings wird keine *reject*-Regel für das umgeschriebe Fakt erzeugt. Die Regeln r_2 bis $r5$ erzeugen Hilfsinformationen, wie aktuell der Inhalt einer Zelle ist. Mit Hilfe dieser Informationen kann sehr schnell entschieden werden, welche Belegung eine Zelle haben soll (die Idee dieser Regeln ist es, den Suchraum für DLV einzuschränken, um unnötige Annahmen von vornherein auszuschliessen). Die Regeln r_6 und r_7 propagieren die *ison*-Literale, r_6 ist dabei die Standard-Inferenzregel, r_7 behandelt explizit den in Abbildung 2.5 erörterten Spezialfall. Die Regeln r_8 bis r_{12} erzeugen nach dem üblichen Muster *reject*, respektive *recovery* Hilfliterale.

weitere Anmerkungen Der vorgestellte Algorithmus ist nur für sehr spezielle Szenarien geeignet. Das Programm ist an genau eine Wissensdarstellung fest gebunden, eine Änderung dieser hat auch zur Folge, dass Regeln neu geschrieben werden müssen. Erschwerend ist zudem die Erweiterung um zusätzliche Zellausprägungen einer Kachelwelt, auch hier müssen die Revisionsregeln von Hand angepasst werden.

Implementierung der Revisionskomponente

Die Implementierung wird in Abschnitt 7.3.3 beschrieben. Während der Entwicklung der Revisionskomponente fiel auf, dass einige Wissensoperationen, die eigentlich Teil der *belief state construction* und Inferenz sind, in die *belief base* verlagert werden können. Dies spiegelt sich in den Wissensänderungsoperatoren

wieder, die nicht nur Expansion, sondern beliebig komplexe Operationen auf der *belief base* ausführen können. Dies betrifft insbesonders das Überschreiben von *ison*-Fakten, die ohnehin immer verworfen werden. Auf diese Weise erhält man einerseits eine schlanke *belief base*, und andererseits kommt dies der Laufzeit- und Speicherbedarf von Inferenzoperationen mit DLV entgegen.

2.4 Desire Generation

S. Broszeit, D. Hoelzgen

2.4.1 Einordnung

Die *desire generation* dient dazu dem Agenten eine Menge von Möglichkeiten vorzugeben, denen er in der Umwelt nachgehen kann. Verschiedene *desires* werden im klassischen BDI Modell als gleichwertig betrachtet, was es nachfolgenden Komponenten erschwert, eine intelligente Auswahl auf diesen zu treffen. Außerdem wird im klassischen BDI Modell nicht darauf eingegangen, auf welche Weise die *desires* generiert werden. Eine fest vorgegebene Generierung von *desires* aufgrund der aktuellen *beliefs* des Agenten birgt die Gefahr, dass ein reaktiv handelnder Agent entsteht. Um einen autonomen, rational handelnden Agenten konstruieren, soll die Komponente zur *desire generation* des klassischen BDI Modells daher um eine Motivationskomponente erweitert werden. Ziel dieser Komponente ist es, aufgrund einer statischen Menge von Motiven *desires* zu generieren, und deren Intensität aufgrund der aktuellen Situation des jeweiligen Agenten anzupassen.

2.4.2 Beschreibung

Die Motivation ist Gesichtspunkt vieler geisteswissenschaftlicher Teilgebiete. Sie dient dazu den treibenden und richtunggebenden Zustand im menschlichen wie animalischen Organismus zu modellieren. Die Psychologie spricht einem Menschen eine Menge von Motiven zu. Diese bezeichnen den Hang bestimmte Ziele zu verfolgen und werden durch bestimmte Reize der Umgebung angeregt. Deshalb tritt der Hang zu bestimmten Zielen erst dann in Erscheinung, wenn die äußeren Umstände das entsprechende Motiv ausreichend aktiviert haben.

Abstrakt betrachtet entspricht diese Beschreibung unserer bisherigen Anforderungsanalyse an die Generierung von *desires*. Wir können das Verhalten unserer Agenten ausreichend dynamisch formen und geben ihnen zusätzlich statische Persönlichkeitsmerkmale, anhand dieser ihr Verhalten ausgerichtet wird.

2.4.3 Erster Ansatz nach Meneguzzi und Luck

Um eine Methode zu finden, die die Generierung von *desires* nach dem BDI Modell mittels abstrakter Motive ermöglicht, sei zunächst die Methode nach Meneguzzi und Luck [18] vorgestellt. Nach dieser wird eine statische Menge von

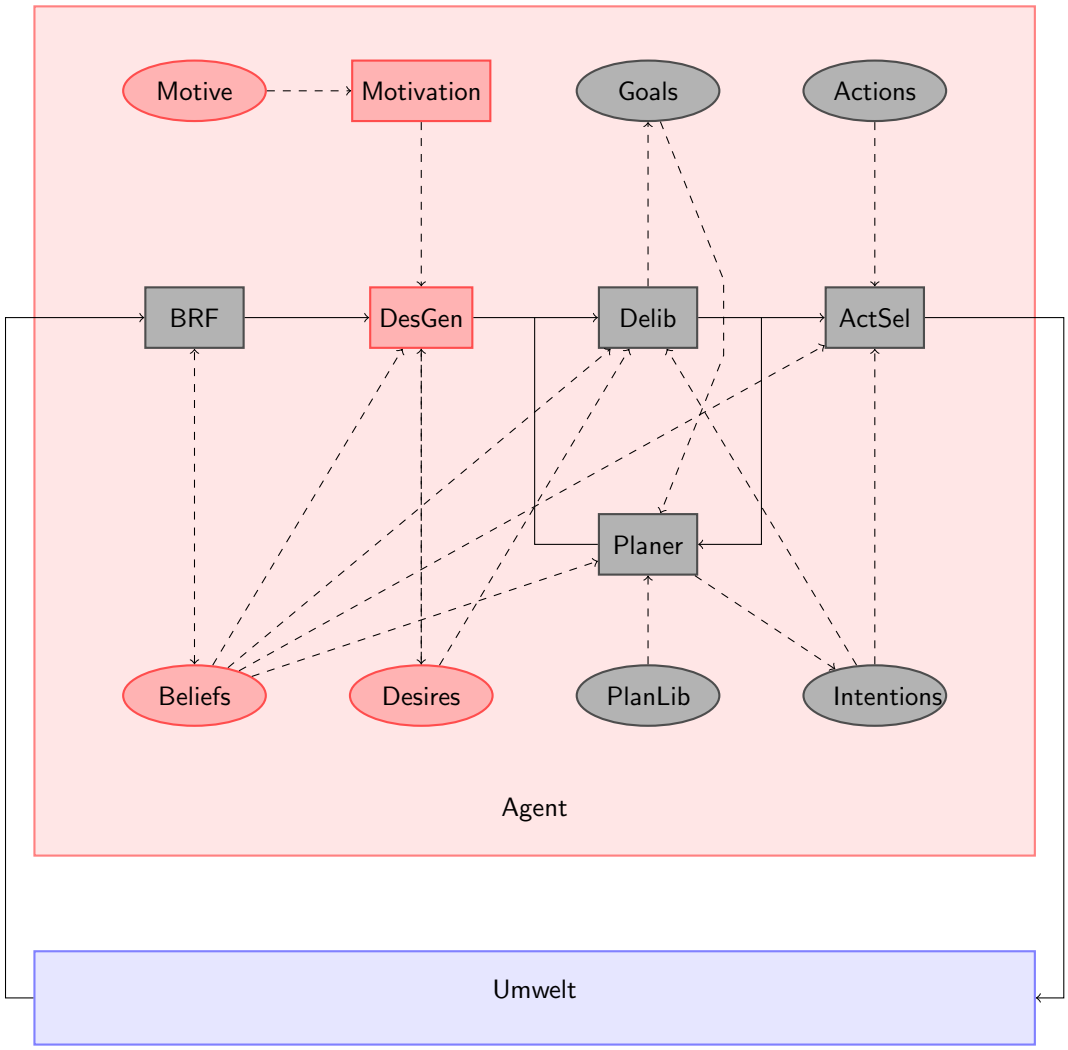

Abbildung 2.6: Einordnung der Motivation in das Cowbot BDI Modell

Motiven mit variabler Intensität genutzt, um bei Überschreiten eines Schwellwertes ein dem Motiv zugehöriges *desire* zu generieren. Formal betrachtet besteht diese Menge von Motiven aus Tupeln der Form $< m, i, t, f_i, f_g, f_m >$, mit

m als Name des Motivs

$i \in \mathbb{N}$ als Intensität im Sinne eines Maß der aktuellen Wichtigkeit

$t \in \mathbb{N}$ als Schwellwert, welcher überschritten werden muss, damit das Motiv in Kraft tritt

$f_i : Bel \rightarrow \mathbb{N}$ als Funktion, welche die eine Intensitätsänderung abhängig von den aktuellen *beliefs* des Agenten zurückgibt

$f_g : Bel \rightarrow Des$ als Funktion, welche eine Menge von *desires* zu diesem Motiv generiert

$f_m : Bel \rightarrow \mathbb{N}$ als Funktion, welche die „Sättigung“ eines Motivs durch Verringerung der Intensität realisiert

In jedem BDI Durchlauf werden die Intensitäten mittels der Funktion f_i aktualisiert, und im Falle der Erfüllung des zugehörigen *desires* durch f_m zurückgesetzt. Übersteigt ein Motiv seinen Schwellwert, so wird f_g zur Generierung eines *desires* genutzt, dessen Erfüllung dem Handeln im Sinne des entsprechenden Motivs entspricht. Auf diese Weise wird der *desire state* in jedem BDI Durchlauf neu erzeugt, dessen *desires* zusätzlich mit einer Intensität versehen werden, so dass man diese entsprechend der Motivation ordnen kann.

2.4.4 Kritik am Ansatz von Meneguzzi & Luck

Die enge Bindung zwischen dem abstrakten Motiv mit entsprechender Intensität auf der einen und den durch dieses generierten, konkreten *desires* auf der anderen Seite erschwert jedoch die gewünschte Funktion der *desire generation* mittels Motiven. Sollten etwa mehrere, unabhängige *desires* entsprechend eines Motivs generiert werden, deren Intensitäten voneinander verschieden sind, weil ihre Erfüllung im Sinne des Motivs eine unterschiedliche Güte hat, so ist dies nach dieser Methode unnötig komplex. Im Massim Szenario würden die Agenten beispielsweise über ein Motiv zum Kühefangen verfügen, das für jede Kuh, die gesichtet wird, ein entsprechendes *desire* erzeugt. Wenn die Intensität aller dieser *desires* allerdings durch das Motiv vorgegeben wird, dann wären alle Kühe gleichwertige Optionen. Da allerdings Distanz zum eigenen Pferch und zum Agenten, Clustering sowie die Anzahl der zu überquerenden Hindernisse eine nicht unerhebliche Rolle spielen, ist diese Lösung nicht sinnvoll. Unabhängig davon könnte die Kopplung einer sich durch die aktuelle Situation des Agenten ändernden Intensität an ein abstraktes Motiv vom Konzept her ungeschickt sein, da diese als statische Komponente das Wesen des Agenten spezifizieren sollen, und nur die durch diese aufgrund der aktuellen Situation generierten *desires* sollen dynamisch und unterschiedlich intensiv sein.

Aus diesen Gründen wird das Erzeugen von *desires* aufgrund abstrakter Motive und der Umgang mit diesen *desires* aufgeteilt. Die Motive können nun als statische Menge ohne Intensität, abhängig von der aktuellen Situation des Agenten beliebig viele, konkrete *desires* erzeugen, die das Handeln nach den durch die abstrakten Motive modellierten Persönlichkeitsmerkmale des Agenten bedeuten. Diese *desires* haben wiederum Funktionen um ihre eigene Intensität abhängig von der aktuellen Situation zu ändern. Auf diese Weise können verschiedene *desires*, die durch ein einzelnes Motiv generiert wurden, unterschiedlich intensiv sein.

Der Unterschied zwischen den beiden Methoden sei kurz anhand eines kleinen Beispiels erläutert: Gehen wir von einem Agenten aus, der nach dem Motiv handelt, reich zu werden. Ihm sind jedoch gesellschaftliche Regeln bekannt, dass man Geld nicht einfach stehlen darf. Weiterhin gehen wir davon aus, dass eine Summe existiert, bei der dieser Agent das Geld dennoch stehlen würde, wenn die Gelegenheit ausreichend günstig wäre.

Nach Meneguzzi und Luck existiert nun das Motiv reich werden. Ein mögliches Desire könnte sein, dass der Agent arbeitet wenn sich die Möglichkeit bietet, ein anderes jedoch, das Geld einfach zu stehlen. Doch auch wenn die Intensität bei einer Möglichkeit, sehr wenig Geld zu stehlen nur leicht ansteigen würde, so würde sie bei ausreichend vielen Gelegenheiten den Schwellwert überschreiten und das entsprechende *desire* generieren, auch oder insbesondere dann, wenn die Intensität auch durch ihm angebotene Arbeit angestiegen ist.

Nach der geänderten Methode könnten unabhängige *desires* aus dem abstrakten Motiv erzeugt werden, deren Intensität sich auch unabhängig anpasst. So könnte bei einem Arbeitsangebot die Intensität des *desires* zu Arbeiten steigen, doch das hätte keinen Einfluss auf das Stehlen von Geld. Zwar werden auch *desires* zum Stehlen generiert, doch geschieht dies für die einzelnen konkreten Gelegenheiten separat, so dass diese keinen Einfluss aufeinander haben, und ihre Intensität so gering ist, dass eine Auswahl eines der *desires* als *goal* unwahrscheinlich ist, insbesondere dann, wenn der Agent an Arbeit kommt. Auf diese Weise soll es leichter sein, dem Agenten das gewünschte und am Vorbild des Menschen angelehnte Verhalten zu geben.

2.4.5 Umsetzung der Kritik

Um die zuvor beschriebene Methode zur Generierung von *desires* durch eine abstrakte Motivmenge zu realisieren, wurde das Motivationskonzept von Meneguzzi & Luck zu der folgenden formalen Beschreibung der Komponenten erweitert.

Motiv

Ein Motiv sollte die abstrakte Beschreibung der einer den Agenten antreibenden Kraft sein. Da durch ein Motiv *desires* erzeugt werden sollen, benötigt es eine Funktion, die die aktuellen *beliefs* des Agenten auf neue *desires* abbilden kann, eine Intensität ist wie zuvor begründet hingegen nicht mehr notwendig. Demnach wird ein Motiv als ein Tupel $< m, f_{dgen} >$ definiert, mit

m als eindeutiger Name des Motivs

$f_{dgen} : Bel \rightarrow Des$ als Funktion, die aufgrund von *beliefs* ein *desire* erzeugen kann. Hierbei ist zu beachten, dass die Funktion für alle passenden Literale in den *beliefs* ein konkretes *desire* erzeugt.

Die Funktion f_{dgen} zur *desire generation* soll konkrete *desires* erzeugen können, welche abhängig von der aktuellen Situation des Agenten ihre Intensität als

Grad der Motivation anpassen können. Zu diesem Zweck werden bei dem Zugriff auf die *beliefs* des Agenten zusätzliche Regeln ausgewertet, um anschließend die Intensitätsfunktionen des erzeugten *desires* als erweiterte logische Programme instantiieren zu können. Dieses Vorgehen sei kurz an einer beispielhaften Funktion zur Generierung von *desires* eines einzigen Typs gezeigt:

$$f_{dgen}(b \in Bel) = bedingung(X) \mapsto < id_X, i, u_X, m_X >$$

Hierbei kann für alle Belegungen von X, für welche $bedingung(X)$ zutrifft, ein *desire* mit entsprechend parametrisierten Funktionen erzeugt werden. Es ist zu beachten, dass id_X den eindeutigen, parametrisierten Namen des *desires* als Literal darstellt, um doppelte *desires* erkennen zu können.

Motivationskomponente

Die Motivationskomponente, in Abbildung 2.6 als Motivation bezeichnet, verfügt über die Motivmenge des Agenten und wird von der *desire generation* dazu genutzt, anhand der aktuellen *beliefs* mittels der Motive des Agenten neue *desires* zu generieren. Dazu werden iterativ alle Motive durchlaufen, und für alle Möglichkeiten, f_{dgen} anzuwenden die entsprechenden *desires* erzeugt. Sie werden allerdings nicht direkt zum *desire state* des Agenten hinzugefügt, sondern an die *desire generation* weitergegeben. Die Motivationskomponente kann somit durch die Funktion der Form $m : Mot \times Bel \to Des$ formal beschrieben werden.

Desire

Nach der vorgestellten Methode müssen nun die *desires* um eine Intensität als Maß der Stärke des Verlangens sowie Funktionen um diese zu aktualisieren erweitert werden. Aus diesem Grund wird auch ein *desire* als Tupel der Form $< id_X, i, u_X, m_X >$ dargestellt, mit

- id_X als Darstellung des *desires* als eindeutig zuzuordnendes Literal
- $i \in \mathbb{N}$ als aktuelle Intensität des *desires*, wobei eine hohe Intensität einem *desire* Vorrang vor anderen *desires* geringerer Intensität gibt
- $u_X : Bel \to \mathbb{N}$ als Funktion, welche die Intensität des *desires* aufgrund der aktuellen *beliefs* des Agenten anpasst
- $m_X : Bel \to \mathbb{N}$ als Funktion, welche unabhängig von u_X die Intensität des *desires* im Falle dessen Erfüllung herabsetzt

Die Funktionen u_X zum Intensitätsupdate und m_X zur Sättigung sind voneinander getrennt, da man so unabhängig von der Erfüllung des *desires* die aktuelle Intensität bestimmen kann. Ein vereinfachtes Beispiel ist das *desire*, die Kuh X in das Gatter zu bringen. Für das Update dieses *desires* wird die Intensität ausschließlich über der Abstand zu dieser Kuh bestimmt. Die Sättigung prüft hingegen, ob die Kuh schon am gewünschten Punkt ist. Dieses Vorgehen soll die Funktionen in erster Linie vereinfachen und wird im Abschnitt der Komponente zur *desire generation* verdeutlicht.

Desirestate

Der *desire state*, in Abbildung 2.6 als *desires* bezeichnet, beinhaltet die aktuellen *desires* des Agenten. Da diese über eine Intensität verfügen, befinden sie sich in einer totalen Quasiordnung, und können von nachfolgenden Komponenten innerhalb des BDI Durchlaufs genutzt werden, um das Handeln des Agenten in eine autonome, motivationsgetriebene Richtung zu leiten. Es ist zudem sichergestellt, dass sich nach Ablauf der *desire generation* im BDI Modell keine *desires* mit negativer Intensität im *desire state* befinden Das bedeutet, dass nur solche *desires* im *desire state* bestehen bleiben, welche tatsächlich das konkrete Verlangen des Agenten gemäß seiner ihm zur Verfügung stehenden Optionen beschreiben.

Es wurde darüber nachgedacht, auch hier einen Schwellwert hinzuzufügen, der nachfolgende Komponenten davon abhält, auf *desires* mit zu geringer Intensität zuzugreifen. Dies ist jedoch nicht notwendig, da *desires* mit sehr geringer Intensität im weiteren Durchlauf ohnehin nur dann Beachtung finden werden, wenn keine *desires* mit höherer Intensität vorhanden ist. In einem solchen Fall macht es jedoch Sinn, dass der Agent sich, wenn auch wenig motiviert, den ihm zu Verfügung stehenden Optionen widmet.

Zugriff andere Komponenten auf den Desire State

Um dem zugrundeliegenden BDI Modell gerecht zu werden, entspricht der Zugriff auf den *desire state*, dem Zugriff auf die *desires* im klassischen BDI Modell. Demzufolge liegt anderen Komponenten zunächst nur die Menge der eindeutig zugeordneten Literale vor. Um diese auf eine Intensität abzubilden, stellt der *desire state* eine Funktion $Int : Des' \to \mathbb{N}$ zur Verfügung.

Desire Generation

Die *desire generation*, in Abbildung 2.6 als DesGen bezeichnet, greift auf die vorgestellten Komponenten zu, um den *desire state* des Agenten gemäß seiner aktuellen Situation zu erzeugen bzw. anzupassen. Formal kann sie als eine Funktion der Form $d : Bel \times Mot \times Des \to Des$ aufgefasst werden: Die aktuellen *beliefs* des Agenten und dessen aktuelle *desires* werden mittels der Motive in einen neuen *desire state* überführt. Hierbei ist zu beachten, dass der *desire state* des Agenten konsistent gehalten und nicht in jedem BDI Durchlauf neu generiert wird. Die einzelnen Schritte des dazu notwendigen internen Ablaufs sehen dabei wie folgt aus:

Generierung von Desires durch Motive Die Motivationskomponente wird genutzt, um abhängig von der aktuellen Situation, in welcher sich der Agent befindet, neue *desires* zu generieren. Initialisiert werden diese - allerdings nicht notwendigerweise - mit einer Intensität von 0. Durch die Bedingung, dass *desires* ein eindeutig zugeordnendes Literal besitzen, kann erkannt werden, ob es sich wirklich um ein neues *desire* handelt oder ob dieses bereits im aktuellen *desire*

state des Agenten vorhanden ist. Im ersten Fall wird das neue *desire* dem *desire state* hinzugefügt.

Intensitätsupdate Anschließend an die Generierung der *desires* wird für alle *desires* im nun erweiterten *desire state* die Funktion zum Intensitätsupdate aufgerufen und das Ergebnis dieser Funktion als neue, vorläufige Intensität des *desires* festgelegt.

Desires auf Sättigung prüfen Um erfüllte Ziele aus dem *desire state* zu entfernen, wird als letzter Schritt vor dem Löschen von *desires* mit zu geringer Intensität die Sättigungsfunktion aufgerufen. Stellt diese fest, dass ein *desire* erfüllt ist, setzt sie üblicherweise die neue Intensität auf 0 zurück. Dies muss explizit nach dem eigentlichen Intensitätsupdate geschehen, damit ein eigentlich erfülltes *desire* nicht aufgrund einer vereinfachten Intensitätsupdatefunktion eine höhere Intensität erhält und im *desire state* bestehen bleibt.

Desires löschen Im letzten Schritt werden alle *desires*, deren Intensität kleiner oder gleich 0 ist, aus dem *desire state* entfernt. Auf diese Weise ist sichergestellt, dass erfüllte oder aufgrund der aktuellen Situation unpassende oder unmögliche *desires* nicht mehr ausgewählt werden können. Dies kann auch schon geschehen, wenn das *desire* in demselben BDI Durchlauf erzeugt worden ist. So wird realisiert, dass aufgrund von neuem Wissen zwar der Wunsch entsteht, dieser jedoch in der aktuellen Situation des Agenten als nicht möglich bzw. sinnvoll eingestuft wird.

2.4.6 Alternative Ansätze

Die vorgestellte Methode wurde insbesondere aus dem Grund entwickelt, verschiedenen aus einem Motiv resultierenden *desires* unterschiedliche Intensitäten zuweisen zu können. Bei der Entwicklung dieser Methode wurden einige alternative Ansätze zum Umgang mit diesem Problem besprochen, auf die im Folgenden kurz eingegangen sei.

Konkretisierung der Motive

Anstelle abstrakter Motive für mehrere konkrete *desires* könnten die Motive selbst konkretisiert werden, um diese und damit auch ihre Intensitäten den einzelnen *desires* besser zuordnen können. Dies widerspricht jedoch der Idee von Motiven als abstrakte Beschreibung der antreibenden Kraft des Agenten.

Mehrfachzuordnung von Motiven und Desires

Eine sehr realitätsnahe Erweiterung des vorgestellten Konzepts ist es, die bestehende Zuordnung von jeweils einem Motiv zu verschiedene *desires*, die es generiert und mit instanziierten Intensitätsfunktionen ausstattet, zu einer Mehrfachzuordnung zu erweitern. Bei einer solchen Mehrfachzuordnung hat nicht nur

ein Motiv Einfluss auf die Intensität eines *desires*. So kann man Motivation entsprechend von Fähigkeit, Gelegenheit, Unterstützung, etc modular konstruieren. Auch dieser Ansatz vervielfacht die Komplexität unserer Komponente.

2.5 Deliberation

E. Böhmer, M.Kruse

Die Deliberationskomponente hat die Aufgabe ein *desire* auszuwählen und dies zu einem *goal* zu erheben.

Goals Formal ist ein *goal* ein Literal $G(v_1, \ldots, v_n)$. Dabei ist G der Name des *goals*, welcher sich möglichst von dem *desire* ableiten sollte, zu welchem es erstellt wurde. $v_1, \ldots, v_n$ sind Variablen bzw. Konstanten.

Ein Beispiel für ein *goal* ist: $InCorral(O, P)$ mit der Bedeutung, dass das Ziel lautet, ein Objekt O in den Pferch P zu bringen. Konkret hätte dann ein Agent das *goal* $InCorral(cow3, ourCorral)$, was für ihn heißt, dass er das Ziel hat, die Kuh $cow3$ in den Pferch $ourCorral$ zu bringen.

Intentions Eine *intention* ist formal ein Literal $I(v_1, \ldots, v_n)$. Der Unterschied zwischen einer *intention* und einem *goal* liegt in deren Semantik: ein *goal* ist ein ausgewähltes *desire*, während eine *intention* ausdrückt, was der Agent beabsichtigt, um dieses *goal* zu erreichen. Damit sind *intentions* schon viel konkreter und existieren erst, wenn der Agent bereits für ein Ziel geplant hat.

Ein Beispiel für eine *intention* ist: $On(X, Y)$. Die Bedeutung dieser *intention* ist, dass sich der Agent zu den Koordinaten (X, Y) begeben möchte. In einem konkreten Fall hat ein Agent vielleicht das *goal* einem anderen zu helfen und muss dazu erst einmal in seine Nähe, z.B. nach (5,8). Dann könnte dieser Agent die *intention* $On(5, 8)$ haben.

Zusammenhang von Goals, Subgoals und Intentions Wenn eine *intention* direkt durch eine Aktion erfüllt werden kann, nennen wir sie *atomic intention*.

Ein Beispiel dafür wäre, wenn ein Agent eine Aktion `sendto(dest, msg)` ausführen kann, womit er einem Empfänger `dest` eine Nachricht `msg` zukommen lassen kann. Damit wäre $Told(ag1, agentsCatchCows)$ eine atomare *intention*, die durch die Aktion `sendto(ag1, agents catch cows)` erreicht wird.

Um eine nicht atomare *intention* zu erfüllen, muss diese als *subgoal* ausgewählt werden. Dies passiert analog dazu, dass die Deliberationsfunktion *desires* als *goals* wählt mit dem Umterschied, dass die Deliberationsfunktion jedes *subgoal* auswählen, damit das darüber liegende Ziel erreicht werden kann. Auch der *planner* kann genau so, wie er für ein *goal* planen kann, auch für ein *subgoal* planen.

Dieser Vorgang, dass nicht *atomare intentions* für *subgoals* ausgewählt werden, wird so oft wiederholt, bis für das *goal* schließlich nur noch *atomare intentions* exisitieren, sodass der Agent mittels Aktionen das Ziel erreichen kann.

Goalstate Ein *goalstate* ist ein sortiertes Tupel von *goals* und *subgoals*. Der *Planner* plant jeweils für das an erster Stelle stehende *goal* bzw. *subgoal*.

Pläne und Intentionstate Ein Plan ist ein Tupel $(G, K, Body)$. Wobei G ein *goal* ist und K konjugierte Literale mit der Möglichkeit der Default-Negation `not`. Diese logische Formel muss konsistent mit der *belief base* sein, damit der Plan ausführbar wird. *Body* ist ein Tupel von atomaren *intentions* oder *subgoals*.

Der *intention state* ist ein geordnetes Tupel von Plänen. Diese Pläne sind zur Ausführung ausgewählt und können bereits teilweise abgearbeitet sein. Der aktuell zur Ausführung ausgewählte Plan ist immer der erste Plan.

Deliberationsfunktion Die *desire generation* erzeugt einen *desire state*, auf dem man mit $\Phi(D)$ (2.4.5) zugreifen kann. Die Elemente werden von der Deliberation ausgelesen und i.d.R wird das höchstmotivierte *desire* d_{m0}, das das erste Element in $\Phi(D)$ ist und die Intensität $I(d_{m0})$ hat, ausgewählt.
Außerdem muss beachtet werden, dass ein *desire* nur ausgewählt wird, wenn es erfüllbar ist. Dies prüft die Deliberation mithilfe der *beliefs*.
Nachdem die Deliberationsfunktion ein *desire* gewählt hat, erstellt sie ein passendes *goal* und fügt es dem *goal state* als erstes Element hinzu.

Eine weitere Aufgabe der Deliberationsfunktion ist es, zu erkennen, ob das Ziel bereits erfüllt ist oder der Kontext nicht mehr passt und deshalb ein Plan nicht mehr ausgeführt werden muss oder darf.

Wenn das erste Literal im *Body* des ersten Plans eines *intention states* ein *subgoal* ist, muss die Deliberationsfunktion dieses *subgoal* aus dem *Body* enfernen und dem *Goalstate* an erster Stelle als neues *goal* hinzufügen. Danach erfolgt eine Übergabe des Kontrollflusses an den *planner*. Sollte das erste Literal im *Body* des ersten Plans eines Intentionstates eine atomare *intention* sein, gibt die Deliberationsfunktion die Kontrolle an die Actionselection weiter.

Goalwechsel Wenn eine *intention* teilweise ausgeführt wurde, kann die *desire generation* neue *desires* generieren und mit einer Motivation belegen. Daher muss die Deliberation bei der Auswahl eines neuen *desires* überprüfen, ob das aktuell höchstmotivierte *desire* d_{m0} das gleiche *desire* ist, wie im vorausgegangenen Durchlauf. Dieses höchstmotivierte *desire* des letzten Durchlaufs nennen wir d_{m1}. Sind d_{m0} und d_{m1} gleich, kann die aktuelle *intention* weiter ausgeführt werden. Im anderen Fall muss ein neuer Plan berechnet werden, was jedoch mit Aufwand verbunden ist. Daher wird auf die Motivation von d_{m1} ein konstanter Summand k_{Plan} addiert, was in einer erhöhten Motivation resultiert, die aktuelle *intention* weiter zu verfolgen. Der Summand k_{Plan} soll dabei die Kosten der Neuberechnung eines Plans darstellen und dient somit der Beibehaltung einer schon teilweise durchgeführten *intention*.

- $I(d_{m1}) + k \geq I(d_{m0}) \Rightarrow$ ausgewähltes *desire* d_{m1}
- $\mathrm{I}(d_{m1}) + k < I(d_{m0}) \Rightarrow$ ausgewähltes *desire* d_{m0}

Verwerfen/Revidieren von Intentions Wenn der Fall eintritt, dass trotz des Faktors k_Plan eine bereits begonnene *intention* nicht mehr weiter ausgeführt, d.h. unterbrochen wird, unterscheiden wir zwei Fälle:

1. „kurzfristige *desires*“: Wenn das neu ausgewählte *desire* d_{m0} aus der Kategorie „kurzfristige *desires*“ stammt, kann nach Erfüllung des zugehörigen Ziels mit der Erfüllung von d_{m1} fortgefahren werden. Es ist hierbei weniger zu befürchten, dass sich der Kontext geändert hat, da Desires aus dieser Kategorie durch einefache Ziele und zugehörigen Plänen erfüllt werden können. Nach deren Erfüllung kann der ursprüngliche Plan noch anwendbar sein und dadurch wieder aufgenomemn werden.

2. „langfristige *desires*“: Wenn das neu gewählte *desire* d_{m0} aus der Kategorie „langfristige *desires*“ stammt, muss die zugehörige *intention* gelöscht werden. Es ist in diesem Fall sehr wahrscheinlich, dass sich der Agent nach Erfüllung von d_{m0} in einem ganz anderen Kontext befindet, in dem die alten zu d_{m1} gehörigen *intentions* keinen Sinn mehr machen. Daher muss auch, wenn d_{m1} wieder zum ausgewählten *desire* wird, dafür neu geplant werden.

„kurzfristige *desires*“ sind *desires* werden als einfach und in wenigen Schritten erfüllbar angenommen. Daher haben diese weniger Einfluss auf die Umwelt, sodass eine alte *Intention*

Zugriff anderer Komponenten auf den Intention State

Der *planner* (2.6) modifiziert den *intention state* eines Agenten. Aber auch andere Komponenten müssen darauf zugreifen. Die *action selection* greift auf diesen zu, um die nächste atomare *intention* und passende dazu Aktion auszuwählen.

2.6 Planner

E. Böhmer

Warum planen? Wenn wir uns mit dem Thema „Planen“ beschäftigen ist zunächst die Frage interessant, warum wir dies benötigen. Aus unserer menschlichen Perspektive scheint es doch recht einfach: Wenn wir eine Kuh sehen, laufen wir so hinter ihr her, dass sie vor uns wegläuft und so in unseren Pferch gelangt. Möglicherweise versuchen wir unterwegs sogar noch weitere Kühe einzusammeln und so schneller an unser Ziel zu gelangen, möglichst viele Kühe in unseren Pferch zu bringen.

Leider ist das für unsere Agenten nicht so einfach. Die Ideen, die wir Menschen recht intuitiv entwickeln können, müssen für unsere Agenten programmiert werden, damit sie nicht „planlos“ über das Feld laufen, vielleicht einmal eine Kuh entdecken und diese für eine Weile irgendwo hin treiben.

Damit die Handlungen unserer Agenten also *zielgerichtet* sind, müssen sie planen können. Bevor wir uns aber Strategien überlegen, wie wir automatisiert planen können, müssen wir uns erst einmal Gedanken darüber machen, was man überhaupt unter Planen versteht.

Was ist Planen? Planung hat mehrere Aspekte. Eine Definition ist beispielsweise: (aus [26])

> *Planung ist die gedankliche Vorwegnahme von Handlungsschritten, die zur Erreichung eines Zieles notwendig scheinen.*

Planung ist also ein abstrakter Vorgang, bei dem noch nicht gehandelt wird. Dennoch müssen die Effekte der gedachten Handlungsschritte sehr wohl berücksichtigt werden. Die Aussage

> *Planning is the reasoning side of acting.*

([11]) drückt das sehr gut aus: Planen und Handeln gehören zusammen.
Planung hat immer ein Ziel; ohne dieses Ziel zu haben, hätte Planung keinen Zweck.
Bei uns Menschen läuft Planung oft unbewusst ab. Häufig müssen wir, gerade bei alltäglichen Aufgaben, gar nicht planen, da wir durch Gewohnheiten auch ohne vorheriges Planen zum Ziel kommen. Wir müssen beispielsweise morgens nicht überlegen, wie wir den Tag beginnen, sondern stehen auf, gehen ins Bad und frühstücken anschließend, ohne vorher im Bett gelegen und diese Handlungschritte geplant zu haben. Ein weiteres Beispiel, bei dem wir Menschen ohne Planung auskommen, sind die Reflexe. Zwar können reaktive Agenten fest verdrahtet sein und so Reflexe simulieren, aber intelligente Agenten, wie wir sie brauchen, müssen explizit planen können um reagieren zu können.
Ein weiterer Aspekt des Planens sind die Kosten. Planen ist zeitaufwändig und damit teuer. Wir Menschen planen daher nur, wenn es notwendig ist und eben nicht in alltäglichen Situationen, um so Kapazitäten für andere Gedanken zu haben. Für unsere Agenten ist Planung aber notwendig und wir müssen berücksichtigen, dass sie Zeit zum Planen brauchen.
Das Ergebnis des Planens ist eine Folge von auszuführenden Handlungsschritten, die nach Ausführung das Ziel erreichen sollen. Dabei ist eine wichtige Frage, wie „gut“ der Plan ist. Gut bedeutet hierbei beispielsweise, wie schnell wir mit dem Plan an das Ziel gelangen. Auch wenn wir in der Informatik meist nach optimalen Ergebnissen suchen, sollten wir bedenken, dass sich Menschen oft mit brauchbaren statt mit optimalen Plänen zufrieden geben, also mit Plänen, die ans Ziel führen, aber eventuell etwas aufwändiger sind als nötig wäre. Der Grund liegt ganz einfach darin, dass es für uns Menschen kaum möglich ist einen optimalen Plan zu finden, da in der realen Welt zu viele Faktoren eine Rolle spielen,

als dass man sie alle berücksichtigen könnte. Da auch das automatisierte Planen sehr komplex ist, sollten wir uns vielleicht auch mit brauchbaren Plänen zufrieden geben, anstatt noch mehr Zeit zu investieren, um optimale Pläne zu finden.

Klassische Planung

Planungprobleme werden oft als ein Zustandsüberführungssystem (state-transition-system) Σ modelliert.
$\Sigma = (S, A, E, \gamma)$ mit
$S = \{s_1, s_2, \ldots\}$ Menge von Zuständen,
$A = \{a_1, a_2, \ldots\}$ Menge von Aktionen,
$E = \{e_1, e_2, \ldots\}$ Menge von Ereignissen,
$\gamma : S \times A \times E \to 2^S$ Zustandsüberführungsfunktion
Die Menge von Ereignissen ist dabei nicht immer erforderlich und kann zum Beispiel Beobachtungen darstellen, was später für dynamische Systeme nützlich sein kann. γ ist meist nur partiell definiert, da nicht jede Aktion in jedem Zustand ausgeführt werden kann, sondern nur unter bestimmten Bedingungen.

Eigenschaften Bei klassischer Planung werden oft eine Menge Vereinfachungen vorgenommen, um das Prinzip von Planungsmechanismen besser darstellen zu können. Bei klassischer Planung gilt:

- Σ ist endlich
- Σ ist deterministisch, also $|\gamma(s, a, e)| = 1$
- Σ ist statisch, d.h. das System hat keine Eigendynamik
- Σ ist vollständig beobachtbar
- nur beschränkte Ziele (ein zu erreichender Zielzustand) sollen erreicht werden
- nur sequentielle Pläne sind zugelassen
- Aktionen sind nicht zeitaufwändig
- Planung geschieht offline

Obwohl das Problem bei so vielen Restriktionen fast trivial erscheint, können Planungsprobleme mit sehr großer Zustandsmenge und vielen möglichen Aktionen sehr komplex werden.

Bei genauerem Hinsehen wird schnell klar, dass für die Ziele der PG Algorithmen, die auf derart beschränkten Systemen arbeiten, kaum nutzbar sind; Weder unsere reale Welt, noch die der Kühe sind so, wie oben angenommen.

Schauen wir uns die Punkte noch einmal an.

- Σ ist endlich. Bei automatisierter Planung haben wir natürlich nur endlich viel Speicherplatz zur Verfügung, daher sind tatsächlich auch nur endlich viele Zustände und Aktionen darstellbar.

- Σ ist deterministisch, das heißt eine Handlung a im Zustand s führt in genau einen Zustand s'. Das ist nicht realistisch. Es gibt zwar immer wieder Theorien gibt, die behaupten unsere Welt wäre deterministisch, wir Menschen hätten nur noch nicht alle Zusammenhänge erkannt, aber wir sollten von Nichtdeterminismus ausgehen und unser System dementsprechend modellieren.

- Σ ist statisch, d.h. das System hat keine Eigendynamik. Ein Multiagentensystem ist aus Sicht eines einzelnen Agenten immer dynamisch, da auch durch die anderen Agenten die Welt verändert wird.

- Σ ist vollständig beobachtbar. Selbst wenn eine Welt voll beobachtbar wäre, ist es evtl. sinnvoll auf nur unvollständige Information zurückzugreifen, da die Mengen an Informationen sonst schnell extrem groß werden.

- Nur beschränkte Ziele sollen erreicht werden. Mit beschränkten Zielen ist gemeint, dass ein einzelner Zielzustand erreicht werden soll. Oft ist es aber sinnvoller, zu versuchen mehrere wünschenswerte Ziele zu erreichen und Prioritäten dafür zu vergeben oder mehrere Ziele abhängig von der Historie zu erreichen.

- Nur sequentielle Pläne sind zugelassen. Möglicherweise wollen wir nicht immer nur eine Handlung zu einem Zeitpunkt ausführen, sondern beispielsweise während der Bewegung zusätzlich auch noch unsere Umgebung beobachten. Dazu brauchen wir Planungssysteme, die Parallelität erlauben.

- Aktionen sind nicht zeitaufwändig. In der Realität benötigen wir Zeit, um eine Handlung auszuführen, Zeit, in der sich die Welt verändern kann. Dies müssen wir ebenfalls berücksichtigen.

- Planung geschieht offline. Bei klassischer Planung wird der Zustand der Welt beobachtet und ein Plan entwickelt, der anschließend ausgeführt wird. Auf Grund der oben aufgeführten Restriktionen funktioniert dies auch immer. Bei uns kann sich aber die Welt ändern oder Handlungen nicht den gewünschen Effekt haben, sodass Pläne während der Ausführung geändert oder angepasst werden müssen.

Um also den Anforderungen unseres Problems gerecht zu werden, können wir nicht auf klassische Planungsalgorithmen zurückgreifen, sondern müssen uns überlegen, wie wir in dynamischen Systemen unter unvollständigem Wissen planen können.

Planen in dynamischen Systemen bei unvollständigem Wissen Auch für Planung ohne einige dieser Restriktionen existieren eine Fülle von Ansätzen, die jeweils *einige* der Aspekte realer Systeme berücksichtigen. Ein Ansatz dafür ist DLV^K, den wir in 2.6.1 genauer betrachten. Zuvor wollen wir jedoch den *planner* in unser Modell einfügen.

Die Planungskomponente

Die Aufgabe des *planners* ist, für ein von einem Agenten verfolgtes Ziel zu planen. Er bildet eine wichtige Komponente des BDI-Modells und bietet dem Agenten die Möglichkeit sein Know-How aus *plan library* auszunutzen.

Arbeitsweise Wie man der Abbildung unseres BDI-Models und der Definition des *planners* entnehmen kann, nutzt der *planner* als Eingabe die aktuellen *beliefs*, *goals* sowie die *plan library* und modifiziert den *intention state* eines Agenten. Dieser Prozess wird im Folgenden genauer beschrieben. Um ein *goal* erreichen zu können, muss ein Agent einen passenden Plan für dieses Ziel aussuchen oder bauen.
In der *plan library* kann es für ein Ziel mehrere Pläne geben, die sich unter anderem im Kontext unterscheiden. Die meisten Pläne sind kontextabhängig, das heißt auf eine Situation abgestimmt. Anhand der aktuellen *beliefs* bestimmt der *planner*, ob ein passender Plan in der *plan library* vorhanden ist. Im positiven Fall besteht die Arbeit des *planners* lediglich darin, diesen Plan an erster Stelle dem *intention state* hinzuzufügen. Wenn kein Plan in der *library* vorliegt, so wird eine DLV^K-Komponente (2.6, s.u.) aufgerufen, die einen Plan berechnet. Analog zum ersten Fall wird dieser vom *planner* zum *intention state* hinzugefügt. Die eigentliche Herausforderung beim Planen besteht in unserem Fall darin, dass die Umwelt der Agenten diskret, dynamisch, unzugänglich und nicht deterministisch ist. Dadurch wird es nötig sein, auch Pläne zu akzeptieren, die nicht optimal sind oder nicht sicher zum Ziel führen. Daher muss möglicherweise auch während der Planausführung abgebrochen und neu geplant werden. Außerdem sind die erzeugten Pläne wahrscheinlich meistens unsicher, da die Umgebung hochdynamisch ist und die Agenten recht wenig über diese wissen. Daher ist *monitoring*, also die Überprüfung, ob ein Agent erfolgreich im Umsetzen seiner Pläne war, unerlässlich.

Monitoring Theoretisch gibt es mehrere Möglichkeiten, welche Komponente wann entscheidet, dass *monitoring* durchgeführt werden soll. Uns erscheint es jedoch am sinnvollsten, wenn der *planner* dafür verantwortlich ist. Dabei soll er in gewissen Abständen ein *subgoal* „Monitoring“ in die Pläne einfügen. Dieser Abstand darf nicht zu groß sein, da sonst nicht rechtzeitig festgestellt werden kann, wenn ein Plan fehlschlägt. Zu häufig sollte *monitoring* aber auch nicht ausgeführt werden, da sonst möglicherweise sehr viel Zeit darauf verwendet wird.
Das *monitoring* selbst besteht im Überprüfen, ob der aktuelle Zustand der Welt dem entspricht, wie der Agent es vorgesehen hat. Zum Beispiel, ob er sich an der Stelle befindet, an der er sich befinden möchte.

Planlibrary Unter einer *plan library* verstehen wir eine Sammlung von statischen Plänen eines Agenten, die sein Know-How beschreibt. Die Gesamtheit der Pläne ermöglichen dem Agenten seine Ziele zu verfolgen und zu erreichen. In der *plan library* befindet sich eine Reihe von Plänen, auf die der *planner* immer zugreifen kann. Jeder dieser Pläne enthält auf jeden Fall auch Informationen über den Kontext, in dem er ausgeführt werden kann, sowie das Ziel, welches er erfüllt.

Die DLV^K-Komponente Wenn kein passender Plan in der *library* vorhanden ist, ruft der *planner* seine DLV^K-Komponente (siehe nächster Abschnitt 2.6.1) auf. Dieser übergibt er den Kontext als Startzustand, das Ziel als Zielzustand, eine Instanziierung, sowie ein „passendes Regelwerk“. Mit „passendes Regelwerk“ ist folgendes gemeint:

- in einem Regelwerk sind Fluent- und Aktionsdeklarationen sowie Kausalregeln und Regeln für bedingte Ausführbarkeit enthalten
- je nach Situation, also zum Beispiel Art des *goals*, werden nicht alle verfügbaren Regeln übergeben, sondern nur solche die auch relevant erscheinen

Die Art des Regelwerks bestimmt sich aus dem erwünschten Zielzustand. Beispielsweise wird für das Ziel „die Herde an Punkt Q treiben“ ein anderes Regelwerk notwendig sein, als für das Ziel „zum Punkt P gelangen“. Ähnlich bestimmt sich die Instanziierung aus dem Startzustand bzw. Kontext: Je nachdem welche Agenten und andere Objekte beteiligt sind, müssen Variablen vorkommen.
Durch die Wahl des passenden Regelswerks erhalten wir außerdem die Möglichkeit zunächst für *goals subgoals* zu planen und in einem weiteren Planungsschritt die *subgoals* weiter zu zerlegen.
Aus der Ausgabe der DLV^K-Komponente muss der *planner* einen Plan erstellen, der für die Agenten umsetzbar ist. Wenn DLV^K beispielsweise ausgibt, ein Tor zu öffnen, muss eventuell zunächst ein Schalter gesucht werden, bevor dieser dann betätigt werden kann, um das Tor zu öffnen.

2.6.1 DLV^K

E.Böhmer

Ein Programm in K beginnt mit der Deklaration von Fluents und Aktionen, die benötigt werden. Es folgen eine Reihe von (Kausal-)Regeln, dem wichtigsten Teil der Sprache. Außerdem kann man Bedingungen, die für die Ausführung einer Regel erfüllt sein müssen, aufschreiben. Es muss natürlich der Startzustand (zumindest in Teilen) angegeben werden und der zu erreichende Zielzustand. Die folgenden Beschreibungen der Sprache sind [7] entnommen. Weiterführende Informationen sowie eine lauffähige Version des DLV^K-Systems finden sich auf [22] .

Planungsproblem

Um ein Planungsproblem zu beschreiben brauchen wir zunächst eine Aktionsbeschreibung $AD = \langle D, R \rangle$. D ist dabei eine Menge von Aktions- und Fluentdeklarationen und R eine Menge von Regeln und Ausführungsbedinungen. Ein Planungsbereich ist dann ein Paar $PD = \langle \Pi, AD \rangle$. Π ist das statische Hintergrundwissen und formal ein normales stratifiziertes Datalogprogramm. Ein Planungsproblem schließlich ist wiederum ein Paar $\langle PD, q \rangle$, wobei q eine Anfrage ist.
Dargestellt wird ein Planungsproblem als ein Programm der Form:

- `fluents:`
- `actions:`
- `initially:`
- `always:`
- `goal:`

In den Teilen `fluents` und `actions` werden Fluents bzw. Aktionen deklariert. Der `initially`-Teil enthält (soweit bekannt) Bedingungen des Anfangszustandes. Im `always`-Teil werden Kausalregeln und Bedingungen zur Ausführbarkeit von Aktionen angegeben. Der Teil `goal` enthält schließlich die Anfrage. Die Syntax der einzelnen Teile beschreibt der folgende Abschnitt.

Syntax

Deklarationen In einem K-Programm müssen Aktionen, Fluents und Typen deklariert werden. Dazu seien $\sigma^{act}, \sigma^{fl}, \sigma^{typ}$ disjunkte Mengen von Namen (für Aktionen, Fluents und Typen) und $\sigma^{con}, \sigma^{var}$ disjunkte Mengen von Konstanten-/ Variablensymbolen. Ein Aktionsatom ist dann definiert als: $p(t_1, \ldots, t_n)$ für $p \in \sigma^{act}$, $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var}$; n ist die Stelligkeit von p. Fluentatome und Typenatome werden analog definiert.
Beispiele für:

- eine Aktion: `move(B,L)`
- ein Fluent: `occupied(B)`
- ein Typ: `block(a)`

Ein Literal heißt grundiert, wenn es keine Variablen enthält. Die Menge L umfasst alle Typen-, Fluent-, und positiven Aktionsliterale
Im Programm wird mit $p(X_1, \ldots, X_n)$ `requires` $t_1, \ldots, t_m$, eine Aktion deklariert, wobei p pos. Literal, $X_1, \ldots, X_n \in \sigma^{var}$, $t_1, \ldots, t_m$ sind Typen, alle X_i erscheinen auch in $t_1, \ldots, t_m$ und $m \geq 0$, Fluents wieder analog.
Ein Beispiel ist: `move(B,L) requires block(B), location(L).`

Regeln und Aufführungsbedinungen Der wichtigste Bestandteil der Sprache K sind Kausalregeln, sie definieren Abhängigkeiten in der Welt und haben die Form:
`caused` f `if` $b_1, \ldots, b_k$, `not` $b_{k+1}, \ldots,$ `not` b_l
`after` $a_1, \ldots, a_m$, `not` $a_{m+1}, \ldots,$ `not` a_n
Dabei ist $f = false$ oder f ist aus L, a_i ist auch aus L $\forall 0 \leq i \leq n$ und b_j ist ein Aktions- oder Fluentliteral $\forall 0 \leq j \leq l$. Eine solche Regel besagt, dass f gilt, wenn im aktuellen Zustand auch der `if`-Teil erfüllt ist und im vorherigen Zustand auch der `after`-Teil erfüllt war. Sowohl der `if`-Teil als auch der `after`-Teil können weggelassen werden, wenn sie nicht benötigt werden.
Beispiele sind: `caused occupied(B) if on(B1,B), block(B)`
und `caused on(B,L) after move(B,L)`
Wenn die Regel nur für Anfangszustände gelten soll, setzt man das Schlüsselwort `initially` davor.
Eine Ausführungsbedingung gibt an, was erfüllt sein muss, damit eine Aktion in einem Zustand ausgeführt werden kann und hat die Form:
`executable` a `if` $b_1, \ldots, b_m$, `not` $b_{m+1}, \ldots, b_n$.
a ist hier ein positives Aktionsliteral, b_i ist aus L $\forall 0 \leq i \leq n$. Eine solche Bedingung besagt, dass die Aktion a ausführbar ist, wenn der `if`-Teil erfüllt ist. Dieser kann auch leer sein, dann ist a unbedingt ausführbar. Analog kann mit `nonexecutable` ausgedrückt werden, wann eine Aktion nicht ausführbar ist. Ein Beispiel für bedingte Ausführbarkeit ist:
`executable move(B,L) if not occupied(B), not occupied(L), B <> L`

Parallelität und sichere Pläne Normalerweise wird nach Plänen gesucht, die Parallelität enthalten können. Um dies zu vermeiden, muss man das Schlüsselwort `noConcurrency` angeben.
Ebenso gibt es die Möglichkeit nur sichere Pläne (s.u.) berechnen zu lassen, indem man das Schlüsselwort `securePlan` angibt. Andernfalls muss man evtl. von Hand überprüfen, ob ein Plan sicher oder optimistisch ist.

Anfragen Eine Anfrage beschreibt den zu erreichenden Zielzustand und hat die Form:
$g_1, \ldots, g_m$, `not` $g_{m+1}, \ldots$, `not` g_n `?` (i).
i ist dabei die Anzahl an Schritten, die bis zum Erreichen des Ziel erlaubt sind, die g_j sind Fluents, die im Zielzustand gelten sollen. Ein Zustand s erreicht dann das Ziel, wenn gilt: $\{g_1, \ldots, g_m\} \subseteq s$ und $\{g_{m+1}, \ldots, g_n\} \cap s = \varnothing$ Ein Beispiel für eine Anfrage ist:
`on(c,b), on(b,a), on(a,table)? (3)`

Semantik

In einer korrekten Instantiierung eines Planungsproblems müssen alle Aktionen und Fluents mit ihrer Deklaration übereinstimmen und Typenliterale müssen mit dem statischen Hintergrundwissen übereinstimmen.
Ein Zustand ist dann eine konsistente Menge grundierter Fluentliterale. Eine

Transition (Überführung) ist ein Tupel $t = \langle s, A, s' \rangle$, wobei s, s' Zustände vor bzw. nach der Menge A von Aktionen sind.
Ein Startzustand ist legal, wenn er alle `initially`-Regeln und die Regeln im `always`-Teil, die keinen `after`-Teil haben erfüllt. Eine Transition ist legal, wenn die Aktionsmenge A ausführbar ist in s.
Eine Überführungssequenz $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$ ist legal, wenn s_0 ein legaler Startzustand und jede Transition in T eine legale Transiton ist.

Pläne

Ein Plan, der von K berechnet wird, ist eine Sequenz von Aktionsmengen $\langle A_1, \ldots, A_i \rangle, i \geq 0$. Wenn keine parallelen Pläne gewünscht sind, enthält jede Menge A_j nur eine Aktion.
Es können zwei Arten von Plänen erzeugt werden: optimistische und sichere Pläne, je nach gewählter Option.
Einen Plan nennt man optimistisch, wenn eine legale Überführungssequenz $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$ existiert, sodass T das Ziel erreicht.
Ein Plan ist sicher, wenn für jeden legalen Startzustand s_0 und jede legale Überführungssequenz $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{j-1}, A_j, s_j \rangle \rangle$ das Ziel in s_j erreicht wird, oder es eine Menge A_{j+1} gibt, die ausführbar ist in s_j, sodass die Sequenz legal fortgesetzt wird. Ein sicherer Plan führt also unabhängig von allen möglichen Startzuständen ins Ziel.

Antwortmengensemantik

Wie bereits erwähnt, kann man ein Programm der Sprache K in ein erweitertes logisches Programm umformen und dann mittels Antwortmengensemantik einen Plan erstellen. Dazu folgen wir im Wesentlichen [8], Kapitel 3.
Es stellt sich zunächst das Problem, dass für Kausalregeln mit `after`-Teil zwei verschiedene Zeitpunkte betrachtet werden müssen: der aktuelle und der vorherige Zustand. Daher führen wir Zeitmarken ein: `time(0), ..., time(i)`, wobei i der in der Anfrage erlaubten Anzahl von Schritten entspricht. Außerdem benötigen wir Prädikate `next(0,1), ..., next(i-1,i)` um Übergänge von einem Zustand zum Folgezustand zu modellieren.
Damit können wir nun das Programm transferieren. Das statische Hintergrundwissen ist bereits als erweitertes logisches Programm gegeben, wir müssen hier nichts ändern.
Für Regeln vom Typ r:`caused` H `if` B `after` A gilt für die neue Regel r':

$$h(r') = \begin{cases} \varnothing, & \text{wenn } H = false \\ f(t, T1), & \text{wenn } H = f(t) \text{ und } f(t) \in \sigma^{fl} \end{cases}$$

wobei $T1$ eine neue Variable ist. In den Rumpf der Regel r' kommen alle Typliterale aus r, Fluentliterale `(not)b(t,T1)` wenn `(not)b(t)` aus B ist, Aktions- und Fluentliterale `(not)a(t,T0)` wenn `(not)a(t)` aus a ist, wobei $T0$ wieder eine neue Variable ist. Für die Zeit brauchen wir außerdem `time` T_1, wenn A leer, sonst `next` (T_0, T_1). Typinformationen, die in K im Deklarationsteil auftauchen

werden außerdem mit in den Regelrumpf genommen, sodass der Deklarationsteil nicht mehr benötigt wird.
Hier zwei Beispiele: `caused occupied(B) if on(B1,B), block(B)` wird zu
`occupied(B,T1) :-on(B1,B,T1), block(B), time(T1).`
und `caused on(B,L) after move(B,L)` wird zu
`on(B,L,T1) :-move(B,L,T0), block(B), location(L), next(T0,T1).` .
Bei der Transformation von Ausführungbedingungen e:`executable` $a(t)$ `if` B wird der Kopf der neuen Regel e': $a(t,T0) \vee \neg a(t,T0)$. Dadurch kann, aber nicht muss, eine Aktion ausgeführt werden, wenn der Rumpf erfüllt ist. In den Rumpf der Regel kommen Typliterale und Typinformationen sowie alle Fluent- und Aktionsliterale wie bei Kausalregeln. Außerdem fügt man `next(T0,T1)` hinzu, damit sichergestellt ist, dass noch ein weiterer Schritt ausgeführt werden darf.
Ein Beispiel: `executable move(B,L) if B <> L` wird zu
`move(B,L,T0)` $\vee\neg$ `move(B,L,T0):- B <> L, block(B), location(L), next(T0,T1).`
Regeln des `initially`-Teils bekommen die feste Zeitmarke 0, die in die Aktions- und Fluentliterale integriert wird.
So wird zum Beispiel
`on(a,table)` zu `on(a,table,0): -block(a), location(table).`
Zuletzt wird die Anfrage `goal: G ? (i)` umgeformt, wobei der Kopf der Regel ein neues Prädikat `goal_reached` ist. In den Rumpf kommt jedes Fluent aus G mit Zeitmarke i. Zusätzlich benötigen wir ein Constraint `:-not goal_reached`
Zum Beispiel wird die Anfrage `on(c,b), on(b,a), on(a,table)? (3)` zu

```
goal_reached :-on(c,b,3), on(b,a,3), on(a,table).
             :-not goal_reached.
```

Beispiel

Zum Schluss betrachten wir noch ein Blockwelt-Beispiel, das die bekannte Sussman-Anomalie beschreibt.
Dazu enthält das statische Hintergrundwissen folgende Informationen:

```
location(table) :- true. true.
location(B) :- block(B).

block(a). block(b). block(c).
```

Die Konstanten a, b und c sind also Blöcke und sowohl Blöcke als auch die Konstante *table* sind Locations.
Die Aktionsbeschreibung enthalte folgende Deklarationen, Regeln und Ausführungsbedingungen:

```
fluents: on(B,L) requires block(B), location(L).
         occupied(B) requires location(B).

actions: move(B,L) requires block(B), location(L).
```

```
always: executable move(B,L) if not occupied(B), not occupied(L), B <> L.
        caused on(B,L) if not -on(B,L) after on(B,L).
        caused occupied(B) if on(B1,B), block(B).
        caused on(B,L) after move(B,L).
        caused -on(B,L1) after move(B,L), on(B,L1), L <> L1.
```

Es gibt also die Fluents *on* und *occupied* sowie die Aktion *move*, die einen Block auf eine Location bewegen kann, wenn beide nicht belegt sind. Außerdem werden die Effekte dieser Aktion beschrieben.
Zuletzt brauchen wir noch die Beschreibungen Anfangs- und Zielzustand:

```
initially: on(a,table). on(b,table). on(c,a).
goal:      on(c,b), on(b,a), on(a,table)? (3)
```

Das DLV erzeugt damit folgenden Plan:

```
STATE0: occupied(a), on(a,table), on(b,table), on(c,a)
ACTIONS: move(c,table)
STATE1: on(a,table), on(b,table), on(c,table), -on(c,a)
ACTIONS: move(b,a)
STATE2: occupied(a), on(a,table), on(b,a), on(c,table), -on(b,table)
ACTIONS: move(c,b)
STATE3: on(a,table), on(b,a), on(c,b), -on(c,table), occupied(a), occupied(b)
PLAN: move(c,table); move(b,a); move(c,b)
```

Anschließend kann man, da die Option `securePlan` nicht gewählt wurde, noch testen lassen, ob der Plan sicher ist.

2.7 Action Selection

A.Löwen

Die *action selection* bildet die Schnittstelle zwischen Agent und Umwelt. Sie bildet die mentalen Vorgänge von Wissensverarbeitung, Motivation, Revision und Planung auf eine tatsächliche Aktion in der Umwelt ab.

Funktionsweise

Die *action selection* wählt gemäß des *intention state* die nächste atomare Intention zum aktuell verfolgten *goal* von dem *intention state* und wählt eine Aktion aus, die diese Intention erfüllt. Typischerweise sind alle möglichen Aktionen, die ein konkreter Agent in einer gegebenen Umgebung ausführen kann vordefiniert und unveränderlich. So können zum Beispiel unsere Cowbot-Agenten nur neun Aktionen ausführen.

Bewegungen in acht möglichen Himmelsrichtungen north, south, east, west, northwest, northeast, southwest, southeast

Auslassen von einer Aktion - Stehenbleiben skip

Dabei gibt es zu einer atomaren Intention stets nur eine mögliche Aktion. Somit ist diese Komponente in diesem Szenario recht einfach und wird durch eine Eins-zu-eins-Zuordnung realisiert. Hat aber ein „hochgradig flexibler“ Agent für ein und dieselbe Intention mehrere Aktionen zur Verfügung, wird die Auswahl zwischen diesen erheblich schwerer und kann beliebig komplex gestaltet werden.

Beispiel

Stellen wir uns einen Agenten zur Erforschung eines Planeten vor. Beschränken wir uns auf die Betrachtung der Bewegungen von diesem Roboter. „North“ ist die nächste verfolgte atomare Intention von diesem Agenten. Er muss sich beispielsweise einen Meter in Richtung Norden bewegen, um sie zu erfüllen. Der Roboter hat folgende Möglichkeiten in dieser Situation:

1. Ein Schritt in Richtung Norden machen
2. Einen Sprung in Richtung Norden machen
3. Sich auf diese Position zu teleportieren
4. Die Räder um 180 Grad drehen, um diese Position zu erreichen

Welche Aktion soll denn nun unser Roboter wählen und worauf soll diese Auswahl basieren? Zum einen kann der Agent die Energieeffizienz der möglichen Aktionen betrachten. Zum anderen kann er für jede ausgeführte Aktion eine Erfolgsstatistik führen und Aktionen wählen, die laut dieser mehr Chancen auf Erfolg haben. Damit der Agent die Wahl clever treffen kann, braucht er eine hinreichend komplexe *action selection* Komponente.

2.8 Vergleich der beiden Modelle

S.Broszeit

Nachdem das der PG-Arbeit zugrundeliegende klassische BDI Modell im Allgemeinen sowie die von der PG ausgearbeiteten Komponenten im Detail vorgestellt wurden, soll nun hier ein Überblick darüber gegeben werden, welche Aspekte des BDI Grundgerüsts weiterentwickelt wurden.

Motivated BDI-Modell unserer Cowbots

Die *beliefs* sind im klassischen BDI Modell nur als Menge definiert. In unserem Agentenkonzept entscheiden wir uns diese als erweiterte logische Programme zu formulieren.

Formal gesehen erfüllt die *beliefs revision function* (vorgestellt in 2.3) genau die auch im klassischen BDI Modell gültige Funktionsvorschrift $\mathcal{P}(Bel) \times P \rightarrow \mathcal{P}(Bel)$. Im Gegensatz zum klassischen BDI Modell, bei dem der tatsächliche Übergang von Perzeptionen zu *beliefs* offengelassen wird, ist in der *belief revision function* klar definiert, wie die erweiterte logische Programmierung genutzt

werden soll, um aus eingehenden Informationen und bereits bestehenden Wissen mithilfe von Revisionsmethoden wie Update, Expansion oder Bounded-History neues Wissen erschlossen werden kann.

Eine weitere deutliche Abweichung vom klassischen BDI Modell ist die Nutzung von Motiven. Während die *desires* im zugrundeliegenden Modell aus *intentions* und *beliefs* gefolgert werden ($\mathcal{P}(Bel) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Des)$), spielt im Motivated BDI Modell die zusätzliche Motivationskomponente eine Rolle. Die Funktionsvorschrift entspricht also $\mathcal{P}(Bel) \times \mathcal{P}(Int) \times \mathcal{P}(Mot) \rightarrow \mathcal{P}(Des)$, wobei Mot die Menge aller möglichen Motive ist. Die Motivation erlaubt es die *desire generation* (vorgestellt in 2.4) besser zu strukturieren. Durch Motive können die Wünsche, die ein Agent zu seiner Lebenszeit aufstellen kann, besser umrissen werden und verschiedene Agententypen können allein durch geänderte Motivzusammenstellung entworfen werden, statt ihre funktionalen Komponenten verändern zu müssen. Außerdem entsteht eine Rangfolge der aufgestellten *desires* aufgrund ihrer Intensität. Dies ist ein weiterer Vorteil gegenüber der vorherrschenden zufälligen Auswahl eines *desires*.

Auch die *deliberation*(vorgestellt in 2.5) weicht von ihrer ursprünglichen Definition ($\mathcal{P}(Bel) \times \mathcal{P}(Des) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Int)$) ab. Der Prozess der *deliberation* wird durch die neue Datenkomponente *goal* in Zielauswahl und Planung aufgeteilt. Ein *goal* ist ein als Ziel ausgewähltes Desire, für das im weiteren Durchlauf der *deliberation* eine Vorgehensweise aufgestellt wird, deren Resultat mit der *intention* des klassischen BDI Modells korrespondiert. Für diese Planung wird zudem dynamische Planung verwendet. Die *deliberation* besteht nunmehr aus zwei Komponenten, von denen die eigentliche *deliberation* für die Zielauswahl und der *planner* für das Aufstellen der *intentions* verantwortlich ist. Es ergeben sich also die beiden Funktionsvorschriften *deliberation:*$\mathcal{P}(Bel) \times \mathcal{P}(Des) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Goal)$ und *planner:*$\mathcal{P}(Bel) \times Goal \times \mathcal{P}(Plan) \rightarrow \mathcal{P}(Int)$, wobei $Goal$ die Menge aller möglichen erfassten Ziele und $Plan$ die Menge aller Pläne ist. Wenn man die Funktionsvorschriften beider Komponenten kombiniert, erhält man also eine erweiterte Funktionsvorschrift der Form *erw. deliberation:*$\mathcal{P}(Bel) \times \mathcal{P}(Des) \times \mathcal{P}(Int) \times \mathcal{P}(Plan) \rightarrow \mathcal{P}(Int)$.

Kapitel 3

Multi-Agenten-System

M.Kruse

Ein Agent ist ein Programm (Software), der in einer Umgebung agiert in der er Umwelteinflüsse wahrnimmt und in der er Umweltaktionen ausführen kann ([24]). Insofern mehrere Agenten zusammenarbeiten um miteinander Aufgaben innerhalb dieser Umwelt zu lösen, spricht man von Multi-Agenten-Systemen. Jeder einzelne Agent ist vollkommen autak in dieser Umgebung und macht seine eignen „Erfahrungen".

Da sich die Umwelt im Zeitverlauf ändert, muss sich die Agenten den neuen Gegebenheiten anpassen. Dies wird duch grundlegende Prozesse (sog. *key processes*):

- Problemlösung,
- planen,
- entscheiden,
- lernen

Ein Hauptziel eines *Multi-Agenten-Systems* ist die Koordination von Zielen und Aufgaben der Agenten. Dies wird in der Regel durch durch Zusammenarbeit der Agenten erreicht um Aufgaben zu erfüllen, die nicht durch einen Agenten allein zu bewältigen wären. Im Idealfall ermöglicht das *Multi-Agenten-System* die Interaktion der Agenten wie die Interaktion zwischen Menschen. Diese Ziele eines Systems sind schwer zu realisieren und stellen die Hauptherausforderung an das Gesamtsystem dar.

Die Hauptcharakteristiken eines *Multi-Agenten-Systems* sind:

- jeder Agent besitzt nur begrenzt Wissen über seine Umwelt und ist eigeschränkt in seinen Möglichkeiten,
- es wird eine verteilte Systemkontrolle ausgeübt,
- die Daten sind dezentral verteilt und

- die Berechnungen eines jeden Agenten ist asynchron.

Die Systeme die in unserem System diese Anforderungen erfüllen werden im folgenden konzeptuell erklärt. Beginned mit dem Phasenmodell (siehe 3.1) der *BDI-Durchläufe* über die MAS-Achritektur (siehe 3.2) bis hin zur Implementierung des Systems (siehe 7).

3.1 Das Phasenmodell

3.1.1 Einordnung des Phasenmodells

Das Phasenmodell dient zur Zusammenfassung mehrerer BDI-Modell-Durchläufe. Da die Komponenten *belief revision*, *desire generation*, *deliberation*, *planner* und *action selection* sequenziell ablaufen, wird während eines Durchlaufs durch die BDI-Komponenten genau eine Aktion gewählt, die zur Ausführung gebracht wird.

3.1.2 Benutzte Begriffe

- BDI-Durchlauf
  Dies stellt einen Durchlauf durch das BDI-Modell (siehe 2.1und 2.8) dar. Dies ist die kleinste Einheit in diesem Teil des Dokuments.

- Phase = 0...n BDI- Durchläufe
  Die hier beschriebenen Phasen, setzen sich aus mehreren BDI-Durchläufen zusammen. Sie dienen dazu mehrere BDI-Durchläufe in Sinnabschnitte zu unterteilen.

- Zyklus = 4 Phasen bzw = 6 Phasen
  Wenn alle vier bzw sechs Phasen durchlaufen sind, ist ein Zyklus beendet und ein neuer wird mit dem Start der ersten Phase eingeleitet.

Da ein Agent seine Zustände endlos wiederholt, wird nach einem Zyklus direkt der nächste Zyklus gestartet.

3.1.3 Zwei Arten von Phasenmodellen

Da unser MAS zwei unterschiedliche Arten von Aufgaben für die Agenten kennt, unterscheiden wir auch in zwei verschiedene Phasenmodelle. Sollte eine Aufgabe eines Agenten ohne Hilfe von anderen Agenten zu erledigen sein, benutzen wir einen *Single-Agent-Task*. Diese Aufgaben werden durch ein vier-Phasen-Modell (siehe 3.1.5) abgebildet. Sollten Aufgaben nur von einer Gruppe von Agenten gelöst werden können, wird das erweiterte Phasenmodell, sechs-Phasen-Modell (siehe 3.1.8), benutzt werden. Das erweiterte Phasenmodell stützt sich auf das vier-Phasenmodell und erweitert dies um Phasen, die zur Bildung von Gruppen und den gemeinsamen Ablauf der Pläne, notwendig sind.

3.1.4 Entwicklung des Phasenmodells

Ein Phasenmodell wird in [24] beschrieben um Phasen für verteiltes Planen festzulegen. Dieses Phasenmodell wurde bezüglich des verteilten Planens als 6-Phasenmodell übernommen. Dieses Phasenmodell kann nur bedingt auf Aufgaben angewendet werden, die ein Agent allein bewerkstelligen kann. Da es durchaus sinnvoll erscheint Phasen für diese einfacheren Aufgaben zu definieren, haben wir das *sechs-Phasenmodell* gekürzt und die einzelnen Phasen mit einer teilweise anderen Bedeutung belegt. Das Ergebnis ist das vier-Phasenmodell.

3.1.5 Vier-Phasen-Modell

Die vier Phasen des Modells sollen für Aufgaben benutzt werden, die durch einen Agenten allein, ohne Zuhilfenahme anderer Agenten, bewältigt werden können.

Informationsaustausch

Die erste Phase dient dem Austausch von *relevanten beliefs* also Informationen über die Umwelt, wie zum Beispiel die aktuelle Position oder wichtige wahrgenommene Informationen. Es ist zu beachten, dass die weitergegebenen Informationen geschickt gewählt werden müssen, um die Komplexität nicht unnötig zu erhöhen. Diese Informationen werden in Abschnitt (siehe 4.1) beschrieben.

Beispiel Alle Agenten berichten über ihre aktuelle Position sowie die Position und Status von feindlichen Agenten, Kühen und Schaltern. **Über unbewegliche Gegenstände wie Hecken, Bäume wird nur informiert, wenn diese noch nicht bekannt sind.** Über freie Felder wird gar nicht informiert, diese können aus fehlenden anderweitigen Informationen gefolgert werden.

Absprache

Die zweite Phase dient der Absprache von *goals*. Dazu teilen sich die Agenten ihre aktuellen *desires* mit. Danach können die Agenten ihre *desires* verändern, damit Konflikte zwischen den Agenten vermieden werden. Hierbei ist zu beachten, dass die aktuelle Menge von *desires* eines Agenten nur aus ausreichend motivierten *desires* besteht, und somit keine übermäßige Komplexität entstehen sollte.

Planen

Die Planungsphase dient dem Erstellen von realisierbaren Plänen für das gewählte *goal*. Diese wird von jedem Agenten selbstständig übernommen und wird in der Regel in einem BDI-Durchlauf fertiggestellt. Die Pläne werden so ausgelegt, dass nach bestimmten Aktion eine Prüfaktion durchgeführt wird. Dadurch soll sichergestellt werden, dass eventuelle Probleme zu einem frühen Zeitpunkt der Planausführung erkannt werden. Zur Fehlerbehandlung werden zusätzliche

Pläne erzeugt, die in einem solchen Fall aktiviert werden müssen. In der Regel sind dies konstante Pläne der *plan library*. Es werden aber auch Pläne mit *DLV-K* erstellt, damit die Agenten auf eine Umwelt reagieren können, die bei der Erstellung der statischen *plan library* nicht berücksichtigt wurden. Die konstanten Pläne werden gemäß dieser Methodik im Vorfeld geschrieben. Sollte ein konstanter Plan zur Ausführung kommen, ist keine wirkliche Aktion in der Phase Planen notwendig.

Ausführung und Monitoring

In der letzten Phase werden die erstellten Pläne ausgeführt. Bei jedem BDI-Durchlauf wird eine Aktion durchgeführt, wodurch diese Phase in der Regel mehrere BDI-Durchläufe benötigt, da ein Plan i.d.R. mehrere Aktionen beinhaltet. Außerdem soll nach bestimmten Aktionen ein Monitoring auf diese Aktionen ausgeführt werden, was bedeutet, dass die Aktion auf Erfolg geprüft wird. Im Fehlerfall muss der Ersatzplan bzw. Fehlerplan aktiviert werden, was im Allgemeinen zu einer Neuplanung führen wird, da anscheinend vorher angenommene Gegebenheiten nicht mehr erfüllt sind. Dies kann zum Beispiel auftreten, wenn zur 'Laufwegsberechnung' ein Gatter offen war und daher ein direkter Weg zu einem Wegpunkt geplant wurde. Wenn das Gatter jedoch geschlossen ist und der Agent daher nicht 'durchgehen' kann, muss der Agent einen neuen Weg berechnen. Somit wäre in diesem Fall eine vollkommene Neuplanung notwendig.

Zusammenhang der Phasen

In Abbildung 3.1 sind die Phasen innerhalb eines Zyklus und mehrere BDI-Durchläufe innerhalb einer Phase dargestellt. Eine Phase kann übersprungen werden, indem in ihr keine BDI-Durchläufe durchgeführt werden. Dies ist der Fall, wenn die einzelnen Komponenten vorher die Notwendigkeit einer Aktion prüfen (siehe 3.1.6).

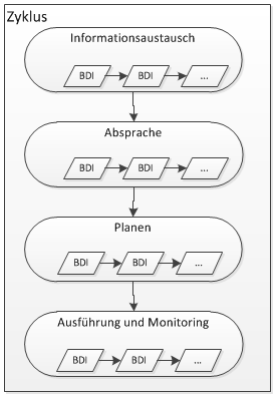

Abbildung 3.1: vier-Phasenmodell

3.1.6 Realisierung

Die Phasen werden durch *subgoals* realisiert, die in jedem Plan vorhanden sind. Als Beispiel betrachten wir eine einfache Aufgabe: *goal* 'Ich möchte 10 Schritte geradeaus gehen'. Dies muss im ersten Schritt zu einem Plan mit den Phasen als *subgoals* aufgelöst werden: Diese *subgoals* würden dann, dem eigentlichen

```
i_move10forward=
(InfomationExchanged,
Agreed,
Planned,
ExecutionMonitoring)
```

Plan entsprechend, aufgelöst werden. Dies würde bedeuten, dass das *subgoal* 'Informationsaustausch' zu einem oder mehreren Kommunikationsakten (atomare Intentionen) aufgelöst wird, in welchen die *beliefs* (teilweise oder vollständig siehe 4.1) allen Agenten mitgeteilt werden. Alternativ können auch *subgoals* genutzt werden, die in einem weiteren Schritt aufgelöst werden. Letztendlich werden atomare Intentionen für die Kommunikation erzeugt. Somit kann sichergestellt werden, dass alle Agenten über das relevante Wissen der anderen Agenten verfügen.

Spätestens nach dem letzen Kommunikationsakt des 'Informationsaustauschs' würde das *subgoal* 'Absprache', ebenfalls in mehrere Kommunikationsakte, aufgelöst (oder *subgoals*, die weiter aufgelöst werden). Hierbei würden die entstandenen *desires* (teilweise oder vollständig siehe 4.1) übertragen werden. Dies wird zum einen für eine etwaige Zusammenarbeit in Gruppen benötigt (3.1.8) und zum anderen zur Absprache von *desires*, welche u.U. nur von einem Agenten erreicht werden können, jedoch aktuell von mehreren Agenten verfolgt werden. Im letzterem Fall könnten die überzähligen Agenten das *desire* aufgeben.

Wenn die Phase 'Planen' aktiv wird (spätestens nach dem letzten *BDI-Durchlauf* der Phase 'Absprache') würde das entsprechende *subgoal* aktiv werden. Dieses *subgoal* würde den Planprozess im eigentlichen Sinne, Bestimmen von Aktionen in der Umwelt, anstoßen und dafür sorgen, dass ein Plan auf den *intention stack* gelegt wird. Dabei würde, an geeigneten Stellen, ein *subgoal* für das Monitoring bis zu dieser Position, erzeugt werden. Außerdem muss ein entsprechender Plan zur Fehlerbehandlung erstellt werden.

Schlussendlich würde das *subgoal* 'AusführungMonitoring', für die Abarbeitung des Plans in der Umwelt, aufgelöst werden. Dabei wird der Plan für die Umwelt und den Zusatzplänen für Überprüfung (Monitoring) und Fehlerbehandlung ausgeführt werden. Dies entspricht der Abarbeitung des Plans aus der Planungsphase.

3.1.7 Konventionen für das Phasenmodell

Da alle Agenten im System das gleiche allgemeine Ziel verfolgen, sind hierfür keine speziellen Mittel notwendig, um die Zusammenarbeit der Agenten zu si-

chern, da dies ihrer festen Struktur entsprechen soll. Aus Komplexitätsgründen sind jedoch Konventionen notwendig, um das Erstellen von Plänen sowie deren Ausführung zu erleichtern. Diese dienen in erster Linie dazu, den komplexen Austausch von Argumenten, zum Beispiel beim geschickten Verteilen von Rollen eines gemeinsamen Plans, zu vermeiden oder ganz unnötig zu machen. Konventionen können z.B. in Plänen oder der Intensitätsberechnung für Motive umgesetzt werden. Eine genauere Definition ist im Abschnitt 6.1.1 zu finden.

3.1.8 Verteiltes Planen

A.Löwen

Wenn ein Agent ein Ziel hat, von welchem er weiß, dass er es nicht alleine lösen kann, beginnt er mit den folgenden Phasen des verteilten Planens:

1. **Gruppenbildung** Im ersten Schritt versucht der Agent eine Gruppe zu finden, mit deren Hilfe er das Ziel erreichen kann. Dabei sendet er eine Anfrage an andere Agenten und fragt nach Hilfe bei einem bestimmten *goal*. Falls diese Anfrage akzeptiert wurde, teilt der Antragsteller-Agent den Helper-Agenten mit, ob diese bei dem Ziel helfen werden. (Es könnte ja sein, dass sich mehr Agenten melden, als für die Aufgabe nötig sind.) Die restlichen Agenten bekommen eine Absage.

2. **Task Decomposition** In diesem Schritt zerlegt der Agent sein Ziel solange in Teilziele, bis sie von einem einzelnen Agenten bewältigt werden können. Dieses macht der Antragsteller-Agent. Die Teilziele werden in der *planlibrary* gespeichert. Der einzige Unterschied besteht dabei darin, dass diese Pläne anstatt von sequenziell auszuführenden SubGoals, die parallele Tasks enthalten, die von mehreren Agenten gleichzeitig ausgeführt werden. Ausserdem ist es möglich DLV^K dafür zu verwenden.

3. **Task allocation** Die eben erstellten Teilziele werden unter den Agenten der Gruppe verteilt. Wir haben uns dafür entschieden, dass der Antragsteller-Agent die Teilziele den anderen Agenten zuweist und zusendet.

4. **Individuelles Planen** Jeder Agent plant eigenständig für das ihm zugeteilte Teilziel. Dabei greift er wie gewohnt auf eigene Pläne-Bibliothek und evtl. auf die DLV-Komponente zu.

5. **Konfliktlösung** Da die so erstellten Pläne eventuell Konflikte enthalten können, müssen die Pläne in Übereinstimmung gebracht werden, damit sie zusammen ausgeführt werden können. Deswegen bekommt der Antragsteller-Agent die Aufgabe, diese Pläne von evtl. entstandenen Konflikten zu befreien. Ausserdem gibt es eine andere Möglichkeit die Konflikte zu lösen. Dieses macht jeder Helfer-Agent lokal, in dem er für ihm zugewiesenes Ziel umplant. So kann es unter anderem vorkommen, dass ein Agent seine Position in der Formation nicht belegen kann, da diese augenblicklich von einem anderem belegt ist. Eine einfache Lösung für dieses

Konflikt besteht darin, eigene Position in der Formation umzurechnen und sich neben der Ursprungsposition zu stellen.

6. **Planausführung und Monitoring** Bevor die Agenten ihre Pläne ausführen, werden diese mit Monitoring-Abfragen (Aufgaben) versehen. Die Agenten führen ihre Pläne aus und überwachen die Effekte. Falls es zu einem schwerwiegendem Planfehler bei einem der Agenten kommt, teilt er das umgehend dem Antragsteller-Agenten mit. Dieser entscheidet dann, ob man umplanen muss. Einfache Fehler, wie z.B. Fehler beim Laufen werden nicht kommuniziert und lokal gelöst.

3.2 Die MAS-Architektur

D.Hoelzgen, F.Bienek

In diesem Kapitel soll die Konzeption der Architektur des Multiagentensystems beschrieben werden. Hierzu werden zunächst das Szenario und die Umwelt beschrieben, um anschließend die mit der Methode Gaia erstellte Konzeption vorzustellen.

3.3 Szenariobeschreibung

Im behandelten Szenario treten zwei gegnerische Teams gegeneinander an, welche jeweils aus der gleichen Anzahl von Agenten bestehen. Ziel ist es, die auf der Karte vorhandenen Kühe in den jeweils eigenen Pferch zu treiben. Dies kann durch den Umstand erreicht werden, dass sich die Kühe von den Agenten wegbewegen, die Agenten diese also durch geschickte Positionierung treiben können. Neben freien Feldern und Hindernissen befinden sich auf der Karte Zäune, welche durch das Positionieren eines Agenten auf einer Schalterposition geöffnet werden kann.

Ziel ist es, nach einer festen Anzahl von Zügen die meisten Kühe im eigenen Pferch zu haben, daher müssen diese nicht nur in diesen getrieben, sondern auch dort gehalten werden. Zu beachten ist, dass bei der Bewegung der Kühe neben dem Zufall und der Position der Agenten auch die Position anderer Kühe eine Rolle spielt, so dass diese zu einer Bewegung in Gruppen tendieren.

3.4 Umwelt

Die Umwelt in der sich die Agenten befinden ist diskret, dynamisch, unzugänglich und nicht-deterministisch. Konkret bedeutet dies, dass die Umwelt eine in quadratische Felder aufgeteilte, rechteckige Karte darstellt, wobei ein Agent oder eine Kuh ein jeweils freies Feld belegen kann. Felder können zudem auch durch statische Hindernisse oder öffenbare Zäune belegt sein. Die Interaktion der Agenten mit der Umwelt erfolgt durch Bewegung in die acht möglichen Richtungen

und in diskreten Zeitschritten. Die Agenten haben einen bestimmten Sichtradius, indem sie die Umwelt wahrnehmen. Zu Beginn des Szenarios ist die Welt, inklusive Hindernissen und Zäunen, nur teilweise bekannt. Die Reaktionen der Umwelt auf die Aktionen der Agenten sind nicht deterministisch, da die Bewegung der Kühe vom Zufall abhängt, und auch die gewählten Aktionen der Agenten selbst nicht immer erfolgreich ausgeführt werden. Ferner ist es möglich, dass auch im Sichtbereich explizit fehlerhafte Informationen wahrgenommen werden.

3.5 Konzeption

Im Zuge der Konzeption der Architektur des Multiagentensystems wird zunächst die High Level Methode Gaia vorgestellt, um ausgehend von dieser Methode, aufgeteilt in eine Analyse- und Entwurfsphase, die Architektur zu entwerfen.

3.6 Die Gaia-Methode

Die Gaia Methode [27], Generic Architecture for Information Availability, ist eine High Level Methode zur Analyse und zum Entwurf agentenorientierter Systeme. Wir haben diese Methode zur Konzeption des Cowbot MAS gewählt, da sie auf ein breites Spektrum an Einsatzgebieten ausgelegt ist und wir somit unabhängig von Entscheidungen bei der Softwareentwicklung sind, die Ergebnisse aber konkret genug sind, um entsprechend unserer späteren Bedürfnisse weiter verarbeitet werden zu können.

Ziel dieses Aufsatzes ist es daher, ein besseres Verstehen der umzusetzenden MAS Architektur zu entwickeln und in diesem Zusammenhang auftretende Probleme zu identifizieren.

3.6.1 Vorbemerkungen

Es sei an dieser Stelle darauf hingewiesen, dass die Umwelt in diesem Konzept nicht explizit modelliert wird. Vielmehr wird davon ausgegangen, dass jede Rolle, welche später Bestandteil des Agententypen wird, implizit Zugriff auf dessen Wahrnehmung als Informationsquelle erhält. Aus diesem Grund ist die durch den Agenten selbst wahrgenommene Umwelt, auch wenn die aus ihr erhaltenen Informationen mit in die angebotenen Services und Kommunikation mit einfließen, nicht explizit aufgeführt. Handelt es sich jedoch um eine durch einen Kommunikationsakt weitergegebene Information, so wird diese jedoch aufgeführt.

3.6.2 Analyse

Ziel der Analysephase ist es, ein Verstehen des Systems und dessen Struktur zu entwickeln. Hierzu werden abstrakte Modelle verwendet und Details der Implementierung ausgeblendet.

Identifikation der Schlüsselrollen

Die Identifikation der Schlüsselrollen hat die vier folgenden Rollen ergeben, die Agenten im Cowbot MAS einnehmen können. Hierbei ist zu beachten, dass ein Agententyp mehrere dieser Rollen annehmen kann.

Cowbot Diese Rolle dient dazu, allgemeine Interaktionen und Attribute für alle Agenten zu definieren. Sie wird von jedem Agententypen angenommen.

Driver Der Driver ist dafür zuständig, Kühe in eine gewünschte Richtung zu treiben. Hierbei wird davon ausgegangen, dass es entsprechend der Anforderungen des Systems üblich ist, dass mehrere Driver zugleich eine Herde von Kühen in eine bestimmte Richtung treiben.

Scout Der Scout dient der Auskundschaftung der Umgebung, sei es die Beschaffenheit derselben zu erkunden, oder aber um Position von Kühen und Feinden zu ermitteln.

Leader Der Leader dient zum Lösen von Aufgaben, welche noch nicht von anderen Agenten bearbeitet werden. Die Rolle dient einzig und allein der Koordination anderer Rollen, sei es in Form von anderen Agenten oder anderer Rollen des Agenten selbst. Der Leader übernimmt zu diesem Zweck eine Leitungsrolle, indem er Aufgaben zuteilt und koordiniert.

Door Opener Der Door Opener dient dazu, Zäune zu öffnen, indem er den zugehörigen Switch betätigt.

Auf eine weitere, denkbare Rolle wird an dieser Stelle noch nicht eingegangen, sie könnte für ein weiter fortgeschrittenes System jedoch in Betracht gezogen werden

Disturber Der Disturber dient dazu, die Anstrengungen des gegnerischen Teams möglichst effizient zu stören, indem er etwa von diesem getriebene Kuhherden auseinander treibt.

Interaktionsmodell

Im Interaktionsmodell werden die Interaktionen zwischen den Rolleninhabern in Form von Kommunikationsprotokollen definiert. Für jede mögliche Interaktion existiert ein solches Kommunikationsprotokoll, wobei dieses die Interaktionsattribute, keine konkreten Nachrichten beschreibt.

Hierbei gelten als eingehende Informationen diejenigen Informationen, welche von allen Teilnehmern zur Kommunikation beigesteuert werden, als ausgehende Informationen hingehen gelten, welche im Zuge der Kommunikation selbst, egal von welchem Teilnehmer, erzeugt werden.

Gemäß der Notation in Gaia wird jedes Interaktionsprotokoll in Form einer Tabelle notiert. Hierbei besteht die Kopfzeile aus dem Namen des Protokolls, die zweite Zeile aus den als Initiator und Empfänger beteiligten Rollen. Die letzte

Zeile wird dazu genutzt, die oben erwähnten ein- und ausgehenden Informationen aufzulisten.

Broadcast Information Dieses Protokoll dient der Weitergabe von relevanten Informationen. Es ist zu beachten, dass es sich hierbei um solche Informationen handelt, die 'für sich selbst' übertragen werden, zur Weitergabe von Informationen im Zuge gemeinsamer Planung oder Aktion existieren andere Protokolle (Abb.3.2).

Broadcast Information	
Initiator: Scout	Empfänger: Cowbot
Eingehend:	- supplied EnvironmentInformation
Ausgehend:	-

Abbildung 3.2: Interaktion Broadcast Information

Request Help Dieses Protokoll wird zum Anfordern von weiteren Agenten genutzt, wenn ein Leader eine Aufgabe nicht alleine bewältigen kann. Er fordert Hilfe bei den entsprechenden Rollen an und verteilt Aufgaben an die gewählten Agenten (Abb.3.3).

Request Help	
Initiator: Leader	Empfänger: Driver/Door Opener
Eingehend:	- supplied HelpInformation
Ausgehend:	- Taskinformation

Abbildung 3.3: Interaktion Request Help

Task Monitoring Dieses Protokoll dient dem Monitoring von gemeinsam ausgeführten Tasks. Sofern notwendig, ist es möglich den Task entsprechend der geänderten Umstände anzupassen (Abb.3.4).

Task Monitoring	
Initiator: Driver / Door Opener	Empfänger: Leader
Eingehend:	- supplied TaskMonitoringInformation
Ausgehend:	- TaskInformation

Abbildung 3.4: Interaktion Task Monitoring

Global Monitoring Dieses Protokoll dient dem globalen Monitoring, um gegensätzlich wirkende oder unvereinbare Anstrengungen der Agenten zu vermeiden (Abb.3.5).

Global Monitoring	
Initiator: Cowbot	Empfänger: Cowbot
Eingehend:	- supplied GlobalMonitoringInformation
Ausgehend:	- ConflictResolveInformation

Abbildung 3.5: Interaktion Global Monitoring

Rollenmodell

Im Rollenmodell werden die im System beteiligten Rollen definiert. Zu diesem Zweck wird für jede Rolle ein Rollenschema ausgearbeitet, welches durch Rollenattribute eine abstrakte Beschreibung der erwarteten Funktion liefert. Diese Attribute umfassen die Rechte und Verantwortlichkeiten der Rolle, sowie die zur Verfügung stehenden Interaktionsprotokolle und Aktivitäten.

Cowbot Diese allgemeine und von allen Agententypen angenommene Rolle dient als allgemeiner Empfänger für die von einem Scout verbreiteten Informationen sowie zur Durchführung und Teilnahme am globalen Monitoring, welches Konflikte unter den Aktivitäten aller Agenten vermeiden soll (Abb.3.6).

Cowbot	
Protokolle	Broadcast Information, Global Monitoring
Berechtigungen	- generates GlobalMonitoringInformation - reads supplied EnvironmentInformation - reads ConflictResolveInformation
Lebendigkeit	(Broadcast Information | Global Monitoring)$^{\omega}$
Sicherheit	true

Abbildung 3.6: Rolle Cowbot

Driver Der Driver dient dazu, nach Reaktion auf ein Hilfegesuch Kühe in die gewünschte Richtung zu treiben. Während des Treibens der Kühe ist er zudem in der Lage, am aufgabenbezogenen Monitoring teilzunehmen (Abb.3.7).

Scout Die einzige Aufgabe der Scout-Rolle ist das Auskundschaften der Umgebung. Sollten Informationen als relevant erachtet werden, können diese an andere Agenten weitergegeben werden (Abb.3.8).

Leader Diese Rolle dient der Koordination anderer Rollen. Zu diesem Zweck ist es dem Leader möglich, Hilfe beim Lösen von Aufgaben anzufordern und die Ausführung dieser zu überwachen (Abb.3.9).

Door Opener Der Door Opener dient dazu, nach Reaktion auf ein Hilfegesuch Zäune zu öffnen, indem er den zugehörigen Switch betätigt. Während

Driver	
Protokolle	Drive Cow, Task Monitoring, Request Help
Berechtigungen	- generates TaskMonitoringInformation - reads supplied HelpInformation - reads EnvironmentInformation, TaskInformation
Lebendigkeit	(RequestHelp. (Task Monitoring | Drive Cow)+)$^{\omega}$
Sicherheit	true

Abbildung 3.7: Rolle Driver

Scout	
Protokolle	Broadcast Information, Scout
Berechtigungen	- generates EnvironmentInformation
Lebendigkeit	(Scout. Broadcast Information)$^{\omega}$
Sicherheit	true

Abbildung 3.8: Rolle Scout

dieser Aktion nimmt auch diese Rolle am aufgabenbezogenen Monitoring teil (Abb.3.10).

Leader	
Protokolle	Request Help, Task Monitoring
Berechtigungen	- generates HelpInformation - reads supplied TaskMonitoringInformation - reads TaskInformation
Lebendigkeit	(RequestHelp. (Task Monitoring)+)$^{\omega}$
Sicherheit	true

Abbildung 3.9: Rolle Leader

Door Opener	
Protokolle	Open Door, Task Monitoring, Request Help
Berechtigungen	- generates TaskMonitoringInformation - reads supplied HelpInformation - reads EnvironmentInformation, TaskInformation
Lebendigkeit	(RequestHelp. (Task Monitoring | Open Door)+)$^{\omega}$
Sicherheit	true

Abbildung 3.10: Rolle Door Opener

3.6.3 Entwurf

Ziel der Entwurfsphase ist es, die in der Analysephase gewonnenen abstrakten Modelle in konkretere Modelle zu überführen. Diese sollen den Implementierungsprozess direkt oder als Eingabe für weitere Methoden unterstützen und dienen zudem der Identifikation von Schwachstellen im Konzept.

Agentenmodell

Das Agentenmodell dient der Auflistung aller im System vorhandenen Agententypen, welche durch die von ihnen gespielten Rollen definiert werden, und bein-

haltet Angaben zur Anzahl der Instanzen. Wie oben beschrieben wird zunächst davon ausgegangen, dass ein einziger Agententyp alle Rollen übernimmt, um so maximale Flexibilität zu erreichen. Sollte dies zu Performanzeinbußen oder anderen Problemen führen, kann eine andere Aufteilung erstellt und die dadurch entstehenden Engpässe in der Kommunikationsstruktur mit Hilfe des Beziehungsmodells identifiziert werden (Abb.3.11).

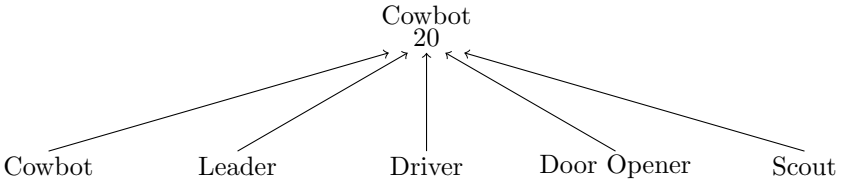

Abbildung 3.11: Agentenmodell, hier für 20 Agenten

Servicemodell

Das Servicemodell identifiziert die den Rollen zugeordneten Dienste, welche wie Funktionen und Methoden bei Objekten verstanden werden können. Der Unterschied ist, dass sie bei Aufruf nicht zwangsläufig ausgeführt werden. Für alle Protokolle und Aktivitäten einer Rolle werden solche Dienste über die Ein- und Ausgaberessourcen sowie die Vor- und Nachbedingungen spezifiziert.

Da in Gaia nur ungenau beschrieben ist, wie Input und Output der einzelnen Services verstanden werden sollen, gelten folgende Konventionen: Der Input eines Services besteht aus den im Laufe der Inanspruchnahme übermittelten Informationen an den Agenten sowie aus dem verwendeten Wissen des Agenten selbst, der Output besteht aus der durch die Ausführung des Services auf direktem Wege erhaltene Information für den aufrufenden oder ausführenden Agenten. Hierbei kann die Reihenfolge von Input und Output variieren und ist nicht wie Parameter und Rückgabewert von Methoden zu verstehen. In diesem Kontext gilt durch Ausführung einer Aktivität erzeugte Änderung an der Umwelt nicht als Output des Services, da dies nur indirekt nach der Ausführung durch den Agenten wahrgenommen werden kann.

Cowbot Services

Umweltinformationen empfangen Empfängt Umweltinformationen von anderen Rollen, unterliegt keinerlei Bedingungen. Hierbei handelt es sich in der Regel um von anderen Agenten als relevant erachteten Informationen, wie zum Beispiel die Position von Kühen und Gegnern.

Input
EnvironmentInformation

Output

Vorbedingungen
$true$

Nachbedingungen
$EnvironmentInformation \neq nil$

Globales Monitoring Identifiziert mit Hilfe der globalen Monitoring Informationen Konflikte und erzeugt gegebenenfalls Informationen zu deren Lösung.

Input
GlobalMonitoringInformation

Output
ConflictResolveInformation

Vorbedingungen
$true$

Nachbedingungen
$(foundConflict \Rightarrow ConflictResolveInformation \neq nil) \vee noConflict$

Teilnahme an Globalen Monitoring Gibt für das globale Monitoring benötigte Informationen und erhält gegebenenfalls Informationen zur Konfliktlösung.

Input
ConflictResolveInformation

Output
GlobalMonitoringInformation

Vorbedingungen
$GlobalMonitoringInformation \neq nil$

Nachbedingungen
$(foundConflict \Rightarrow ConflictResolveInformation \neq nil) \vee noConflict$

Driver Services

Kuh treiben Interne Aktivität, welche das Ziel hat eine Kuh in die gewünschte Richtung zu treiben.

Input
TaskInformation, EnvironmentInformation

Output
TaskMonitoringInformation

Vorbedingungen
$TaskInformation \neq nil$

Nachbedingungen
$TaskInformation \neq nil$

Teilnahme an Task Monitoring Gibt für das aufgabenbezogene Monitoring benötigte Informationen und erhält gegebenenfalls Informationen zur Konfliktlösung.

Input
TaskInformation

Output
TaskMonitoringInformation

Vorbedingungen
$TaskMonitoringInformation \neq nil$

Nachbedingungen
$TaskMonitoringInformation \neq nil$

Auf Hilfegesuch reagieren Ermöglicht, auf ein Hilfegesuch zu reagieren und bei optionaler Zusage entsprechende Informationen zur Aufgabe zu erhalten.

Input
HelpInformation, TaskInformation

Output

Vorbedingungen
$true$

Nachbedingungen
$((HelpInformation \neq nil \wedge willHelpRequester) \Rightarrow$
$TaskInformation \neq nil)$
$\vee\, willNotHelpRequester$

Scout Services

Erkunden Interne Aktivität, welche das Ziel hat relevantes Wissen über die Umgebung zu erlangen. *Das Wissen, was durch Ausführen dieser Aktivität erlangt wird, ist absichtlich nicht unter Output aufgeführt, weil es nicht direkt durch diese Aktivität erzeugt wird, und die Umwelt nicht explizit modelliert ist.*

Input
EnvironmentInformation

Output

Vorbedingungen
$true$

Nachbedingungen
$true$

Umweltinformationen weitergeben Dient der Weitergabe von als relevant erachteten Informationen über die Umwelt.

Input

Output
EnvironmentInformation

Vorbedingungen
$EnvironmentInformation \neq nil$

Nachbedingungen
$EnvironmentInformation \neq nil$

Leader Services

Hilfe anfordern Fordert Hilfe von weiteren Agenten an und übermittelt bei Zusage Informationen zur (Teil-)Aufgabe

Input

Output
HelpInformation, TaskInformation

Vorbedingungen
$HelpInformation \neq nil$

Nachbedingungen
$((HelpInformation \neq nil \wedge willGetHelp) \Rightarrow$
$TaskInformation \neq nil)$
$\vee\, willNotGetHelp$

Task Monitoring Identifiziert mit Hilfe der aufgabenbezogenen Monitoring Informationen Konflikte und erzeugt gegebenenfalls Informationen zu deren Lösung.

Input
TaskMonitoringInformation

Output
TaskInformation

Vorbedingungen
$TaskInformation \neq nil$

Nachbedingungen
$TaskInformation \neq nil$

Door Opener Services

Tür öffnen Interne Aktivität, welche das Ziel hat eine Tür durch einnehmen einer bestimmten Position zu öffnen.

Input
TaskInformation, EnvironmentInformation

Output
TaskMonitoringInformation

Vorbedingungen
$TaskInformation \neq nil$

Nachbedingungen
$TaskInformation \neq nil$

Teilnahme an Task Monitoring Gibt für das aufgabenbezogene Monitoring benötigte Informationen und erhält gegebenenfalls Informationen zur Konfliktlösung.

Input
TaskInformation

Output
TaskMonitoringInformation

Vorbedingungen
$TaskMonitoringInformation \neq nil$

Nachbedingungen
$TaskMonitoringInformation \neq nil$

Auf Hilfegesuch reagieren Ermöglicht, auf ein Hilfegesuch zu reagieren und bei optionaler Zusage entsprechende Informationen zur Aufgabe zu erhalten.

Input
HelpInformation, TaskInformation

Output

Vorbedingungen
$true$

Nachbedingungen
$((HelpInformation \neq nil \wedge willHelpRequester) \Rightarrow$
$TaskInformation \neq nil)$
$\vee\, willNotHelpRequester$

Beziehungsmodell

Das Beziehungsmodell dient der Spezifikation der zwischen den Agententypen bestehenden Kommunikationsverbindungen. Solange nicht mehrere verschiedene Agententypen im System existieren, ist dieses Modell überflüssig.

Kapitel 4

Kommunikation

4.1 Kommunikation

S. Broszeit

4.1.1 Motivation

Das Konzept sieht vor, dass die Agenten unseres Multiagentensystems zeitweise unterschiedliche Aufgaben übernehmen, beispielsweise um die Umwelt zu erkunden oder um Teilziele zu erfüllen. Damit unsere Agenten dabei koordiniert agieren können, ist es unumgänglich, dass sie Informationen untereinander austauschen. Aber auch wenn die Agenten kooperieren, um gemeinsam ein Teilziel zu verfolgen, müssen sie kommunizieren.

4.1.2 Sprechakttheorie

Um ihre Kommunikation zu strukturieren verwenden wir die Sprechakttheorie [2, 21]. Die Sprechakttheorie teilt die Kommunikation in einzelne Sprechakte auf. Der am einfachsten zu identifizierende Sprechakt ist der „tell“ Sprechakt. Dieser dient allein dazu einem anderen Agenten eine Information zukommen zu lassen. Es können außerdem weitere Sprechakte identifiziert werden wie zum Beispiel der „ask“ Sprechakt, der eine Information von einem anderen Agenten erfragt, oder der „untell“ Sprechakt, der eine Information aus der Wissensbasis des Empfängers entfernt, sowie die Broadcastvarianten der beiden erstgenannten Sprechakte „ask-all“ und „stream-all“. Die Sprechakte lassen sich bei Bedarf erweitern, um die Kommunikation besser zu strukturieren. Im Rahmen eines Multiagentensystems bietet sich der „achieve“ Sprechakt an, mit Hilfe dessen ein Agent einem anderen ein Ziel in Auftrag geben kann.

4.1.3 Szenariobezug

Die eingangs vorgestellten Sprechakte, mittels derer eine Kommunikation zwischen den Agenten stattfinden soll, finden sich auch in unserem Ansatz wieder. Anders als in KQML [23] werden die genutzten Sprechakte jedoch nicht explizit durch Schlüsselwörter innerhalb der übertragenen Information festgelegt, sondern sind vielmehr implizit über die Semantik der übertragenen Literale festgelegt. Während in KQML die Semantik einer Nachricht von den verschiedenen Parametern, des *message layer's* abhängig ist, so hängt diese in unserem Ansatz allein von den auf dem *content layer* codierten Informationen ab. Die der Kommunikation zugrundeliegenden Sprechakte sind jedoch in beiden Fällen dieselben.

4.1.4 Kommunikationsverlauf

In unserem Szenario ist es notwendig verschiedene Kommunikationsvorgänge durchzuführen, sei es um Informationen über das Spielfeld auszutauschen oder um eine Gruppe zusammenzuschließen. KQML bietet dafür zwar bereits Sprechakte an, diese stellen jedoch teilweise starke Eingriffe in die Autonomie der Agenten dar, zum Beispiel führt der Sprechakt *achieve* direkt dazu, dass der Empfänger den Inhalt der Nachricht zum *goal* nimmt. Unsere Agenten sollen jedoch jederzeit in der Lage sein, solche Entscheidungen selbst zu treffen. Deshalb bedienen wir uns nur der Sprechakte, die Einträge in die Wissensbasis vornehmen: *tell* und *untell*.

4.1.5 Interpretation der Kommunikation

Die Interpretation der übertragenen Literale seitens des Agenten geschieht auf verschiedenen Ebenen. Zum einen werden die Motive des Agenten gemäß der übertragenen Literale angepasst, zum anderen werden diese auch in den statischen oder dynamisch generierten Plänen des Agenten berücksichtigt. Zudem werden auch die in der BRF des Agenten genutzten Revisionsregeln dahingehend angepasst, dass diese das Wissen des Agenten entsprechend revidieren. Im Folgenden seien für die einzelnen Bereiche Beispiele zum besseren Verständnis angegeben:

Anpassen der Motive. Das Literal *help*(...) stellt das Wissen des Agenten dar, dass ein anderer Agent Hilfe beim Lösen einer Aufgabe angefordert hat. Der diesem Literal am ehesten entsprechende Sprechakt ist *achieve*, der anfragende Agent möchte, dass der empfangende Agent eine Aufgabe übernimmt. Die Motive des Agenten können nun dahingehend angepasst werden, dass sie gemäß dieser Semantik und ausgehend von der Konvention, dass die Agenten grundsätzlich versuchen, sich gegenseitig zu helfen, ein *desire* erzeugen, welches das Verlangen des Agenten darstellt, dieser Anfrage nachzukommen.

Anpassen der Pläne. Während gemeinsamer Aktionen ist ein durch Kommunikation koordiniertes Vorgehen notwendig. In diesem Fall könnte ein Agent beispielsweise mittels *in_position*(...) mitteilen, dass er eine angeforderte Position erreicht hat und nun mit der Aufgabe fortfahren kann. Der diesem Literal entsprechende Sprechakt ist *ready*, und wird in den Plänen entsprechend behandelt, so dass nach Eingang einer solchen Meldung von allen beteiligten Agenten mit der gemeinsamen Bewältigung der Aufgabe fortgefahren werden kann.

4.1.6 Konsequenzen unseres Kommunikationsmodells

Die Wahl unseres Ansatzes liegt darin begründet, dass dieser besser mit dem von uns vorgestellten Modell vereinbar ist. Das explizite Angeben der Sprechakte in KQML macht ein Schachteln der übertragenen Information notwendig, was von unserer Wissensdarstellung nicht unterstützt wird. Der Vorteil von KQML, eine standardisierte und damit auch für andere Multiagentensysteme verständliche Form der Kommunikation, ist in unserem Szenario nicht notwendig, da alle an der Kommunikation beteiligten Agenten aus dem selben Multiagentensystem stammen und mit dem zur Interpretation der Kommunikation notwendigen Initialwissen sowie angepassten Motiven und Plänen versehen werden können.

Kapitel 5

Wissensdarstellung

F. Bienek, B. Jablkowski

5.1 Wissensrepräsentation im Cowbot-BDI-Modell

Wir wollen nun beschreiben, wie Wissen im Cowbot-Szenario konkret realisiert wird.

Zunächst ist zu beachten, dass unsere Agenten sich in einer *grid world*, das heißt in einer zweidimensionalen Gitterwelt aufhalten. Wir definieren deshalb ein Prädikat, welches die aktuelle Belegung einer Zelle symbolisiert (1).

Desweiteren muss ein Agent Belegungen der Zellen eindeutig identifizieren können. Diese Identifikatorprädikate werden genutzt, um jeder Id eine Bedeutung zu geben, die auch in *answer set solvern* genutzt werden können (2-10).

Für einen Großteil der Zellenbelegungen im Cowbotszenario reichen die obigen Prädikate aus. Es kann jedoch vorkommen, dass den Agenten ein Hindernis in Form eines Zauns gestellt wird. Diese Zäune können entweder im geschlossenen oder im offenen Zustand auftauschen. Um zu verhindern, eine zweite Id für den gleichen Zaun zu verwenden, definieren wir ein Prädikat, dass pro Zelle festhält, in welchem Status sich der Zaun befindet (11). Diese Prädikat wird nur für Zäune generiert, es ist also nicht für alle Zellen vorhanden.

Damit sind die grundlegenden sensorischen Daten eines Agenten in einem Szenario modelliert. Zwei weitere Daten werden dem Agenten zu Beginn der Simulation übergeben: Die Größe der Karte und die Dauer der Simulation. Wir speichern diese Daten in entsprechenden Prädikat-Konstrukten (12 und 13). Außerdem sollte der Agent in jedem Takt der Welt über die Zeit Bescheid wissen (14).

Die Wissensrepräsentation sieht also wie folgt aus:

1. **ison(Id,X,Y,T)** - Id steht in Zeitpunkt T auf dem Feld mit den Koordinaten (X,Y).

2. **owncorral(Id)** - Id ist unser Pferch.
3. **enemycorral(Id)** - Id ist ein feindlicher Pferch.
4. **cow(Id)** - Id ist eine Kuh.
5. **fence(Id)** - Id ist ein Zaun.
6. **iam(Id)** - Id bin ich.
7. **obstacle(Id)** - Id ist ein Hinderniss.
8. **switch(Id)** - Id ist ein Schalter.
9. **ally(Id)** - Id ist ein verbündeter Agent
10. **enemy(Id)** - Id ist ein Feind.
11. **isopen(X,Y,B,T)** - Die Zelle mit der Koordinate (X,Y) ist zum Zeitpunkt T offen (B = True) oder geschlossen (B = false)
12. **worldSize(X,Y)** - Breite X und Höhe Y des Spielfeldes, in Kacheln
13. **endTime(T)** - Dauer des Szenarios, in Zeittakten
14. **time(T)** aktueller Zeittakt T der Welt
15. **lineofsight(X)** Sichtweite, in Kacheln pro Halbachse
16. **simulation(X)** Szenario-ID
17. **gate(X,Y,SW,DIR)** Struktur aus Zaunzellen, die mit Schalter mit Id SW geöffnet werden können. DIR beschreibt die orthogonale Zaunausrichtung (*horizontal* oder *vertical*).
18. **fencegroup(X,Y,DIR)** Koordinaten zusammenhängender Zaunzellen, zu denen ein Agent noch nicht den Schalter kennt.

Es gilt $X, Y \in \mathbb{N}_0$, $T \in \mathbb{N}_0$, $B \in \{true, false\}$, $Id \in \{unexplored, empty, unknown, owncorral, enemycorral, fence, switch, ally, enemy, obstacle, self, cowA\}$ *mit* $A \in \mathbb{N}_0$

Anmerkungen

- Das Wissen über die Umwelt, welche der Agent durch eigene Wahrnehmung erfasst, soll immer nur den zuletzt gesehenen Ist-Zustand darstellen. Dies muss nicht zwangsweise in allen Zellen der Welt die Information des aktuellen Zeittakts sein, da ein Agent nur über eine begrenzte Blickreichweite verfügt.

- Entitäten der Welt sind freie Zellen, Mauern, Kühe, befreundete Agenten, feindliche Agenten, Schalter und Zäune. Außer für Kühe gilt für alle, dass sie nicht eindeutig identifiziert werden können (das Szenario unterscheidet nicht zwischen ihnen). Differenzierbare Individuen sind lediglich Kühe (MASSIM annotiert sie mit einer Id, welche wir in *cowA* als *A* speichern). Agenten sind lediglich anhand ihrer Fraktion (Freund oder Feind) unterscheidbar (daher haben *ally* und *enemy* keine Zahlen-Suffixe).
- Einige Konstrukte erscheinen redundant (wie zB "*ally(ally)*"). Wir haben sie trotzdem eingeführt, da es so mit einem *answer set solver* einfacher ist, dieses Wissen auszuwerten.
- Der Agent ist sich stehts seiner eigenen Position in der Welt bewusst. *ison(self,X,Y,T)* und *iam(self)* sind also immer Menge des Agentenwissens.
- Die Id *self* existiert in dieser Form nicht im System. Jedem Agenten wird zu Beginn eine Id zugewiesen (wir legen das in der ASL-Datei des Agenten fest). *Self* ist hier für diese Id zu verwenden.
- Die Prädikate *ison* und *isopen* besitzen einen Zeitstempel, um verschiedene Informationen für das gleiche Feld speichern zu können. Die Identifikator-Prädikate benötigen dies nicht, da sich während einer Simulation Entitäten nicht ändern (eine Kuh ist nicht ab einem bestimmten Zeitpunkt eine Mauer).

5.2 Agentenprogramme

T. Vengels

Dieser Abschnitt erläutert einige der verwendeten logischen Programme, auf Basis derer ein Agent Wissen generiert. Es werden die Programme vorgestellt, die jedem Agenten zugeordnet sind. Eine rollenspezifische Wissensdarstellung wird in späteren Kapiteln (6.1.1, 6.1.2, 6.1.3) erörtert, wo ebenfalls eine detaillierte Vorstellung der Fähigkeiten und Aufgaben der Agenten erfolgt.

Logisches Programm 5.2.1 (Grundannahme über die Welt)**.** *Zu Beginn ist den Agenten lediglich die Weltgrösse und die Lage ihres Pferchs bekannt. Daraus schliessen sie, dass ihre Welt weitestgehend nicht erkundet ist.*

```
cellcoord(X,Y) :- worldsize(MX,MY),
                  #int(X), X >= 0, X < MX,
                  #int(Y), Y >= 0, Y < MY.

cellexplored(X,Y) :- ison(_,X,Y,T), T > 0.

ison(unexplored,X,Y,0) :- cellcoord(X,Y),
                          not cellexplored(X,Y).
```

Hier ist anzumerken, dass cellcoord, cellexplored Hilfsliterale zur Berechnung sind und deshalb nicht in der Wissensrepräsentation explizit auftauchen. Verzichtet man explizit auf unerkundete Zellen, können leere Zellen auf diese Weise abgeleitet werden.

Logisches Programm 5.2.2 (Umweltaktualisierung)**.** *Das folgende Beispiel aktualisiert die Umweltwahrnehmungen, sowie den Zeitstempel. Diese Daten bekommen die Agenten von Umweltsimulatoren geliefert:*

```
:- time(X), time(Y), X<Y.
:- ison(ID,X1,Y1,T1), ison(ID,X2,Y2,T2), cow(ID), T1<T2.
:- ison(ID,X,Y,T1), agent(ID), time(T2), T1<T2.
:- ison(ID,X,Y,T1), ison(ID,X,Y,T2), switch(ID), T1<T2.
:- ison(ID,X,Y,T1), ison(ID,X,Y,T2), fence(ID), T1<T2.
:- ison(ID,X,Y,T1), ison(ID,X,Y,T2), corral(ID), T1>T2.
:- ison(ally,X,Y,T1), ison(ID,X,Y,T2), iam(ID), T1<T2.
:- isopen(X,Y,S1,T1), isopen(X,Y,S2,T2), T1<T2.
```

Das Programm stellt sicher, dass ison-Aussagen über bewegliche Objete (Kühe, Agenten) sowie Kacheln nur einmalig im Wissen auftauchen. Zusätzlich müssen noch Regeln für Behandlung freier Zellen angegeben werden, falls man explizit unerkundete Zellen berücksichtigt und leere Zellen seiten der Umwelt mitgeteilt werden:

```
:- ison(empty,X,Y,T1), ison(empty,X,Y,T2), T1 < T2.
:- ison(empty,X,Y,T1), ison(AG,X,Y,T2), agent(AG).
:- ison(empty,X,Y,T1), ison(COW,X,Y,T2), cow(COW).
:- ison(empty,X,Y,T1), ison(OBS,X,Y,T2), obstacle(OBS).
```

Logisches Programm 5.2.3 (Zaunausrichtung)**.** *Aus dem Wissen über Zaunzellen kann Wissen über Tore (zusammenhänge Zaun-Schalter Komplexe) abgeleitet werden. Das folgende Program hilft dem Agenten, dies zu wissen:*

```
gate(X,Y,SW,horizontal) :- ison(SW,X,Y,BLANK), switch(SW),
                           ison(F,A,Y,BLANK2), fence(F), A=X+1.
gate(X,Y,SW,horizontal) :- ison(SW,X,Y,BLANK), switch(SW),
                           ison(F,A,Y,BLANK2), fence(F), X=A+1.

gate(X,Y,SW,vertical) :- ison(SW,X,Y,BLANK), switch(SW),
                         ison(F,X,A,BLANK2), fence(F), A=Y+1.
gate(X,Y,SW,vertical) :- ison(SW,X,Y,BLANK), switch(SW),
                         ison(F,X,A,BLANK2), fence(F), Y=A+1.

gate(X,Y ,SW,vertical) :- gate(X,A,SW,vertical), ison(F,X,Y,BLANK),
                          switch(SW),fence(F), Y = A + 1.
gate(X,Y ,SW,vertical) :- gate(X,A,SW,vertical), ison(F,X,Y,BLANK),
                          switch(SW),fence(F), A = Y + 1.
```

```
gate(X,Y ,SW,horizontal) :- gate(A,Y,SW,horizontal),
                            ison(F,X,Y,BLANK), switch(SW),
                            fence(F), X = A + 1.
gate(X,Y ,SW,horizontal) :- gate(A,Y,SW,horizontal),
                            ison(F,X,Y,BLANK), switch(SW),
                            fence(F), A = X + 1.

gatefound(X,Y) :- gate(X,Y,T,BLANK).

noswitch(X,Y) :- ison(F,X,Y,BLANK), fence(F), not gatefound(X,Y).
```

Logisches Programm 5.2.4 (Zaunzustand)**.** *Mit Hilfe der gate Regeln erkennt der Agent zusammenhängende Zaunzellen. Dies ermöglicht einem Agenten, den Zustand über zusammenhängende Zaunzellen anhand einer Information über eine Zaunzelle zu revidieren.*

```
gatetime(SW,T) :- gate(X,Y,SW,DIR), isopen(X,Y,_,T).

gatehigher(SW,T) :- gatetime(SW,T), gatetime(SW,T2), T2>T.
gatemax(SW,T) :- gatetime(SW,T), not gatehigher(SW,T).
gatestate(SW,STA,T) :- gatemax(SW,T), gate(X,Y,SW,DIR),
                       isopen(X,Y,STA,T).

isopen(X,Y,S,T) :- gatestate(SW,S,T), gate(X,Y,SW,DIR).
```

Logisches Programm 5.2.5 (Pferche)**.** *Zu Beginn einer Simulation bekommt jeder Agent das Wissen über die Position des eigenen Pferches mitgeteilt. Pferche sind stehts rechteckige Zellbereiche, anstatt einzelner Zellkoordinaten kann der Pferchmittelpunkt von Bedeutung sein. Dieser stellt einen guten Zielpunkt für das Treiben von Kühen dar.*

```
corMinX(MinX) :- #min{X:ison(owncorral,X,_,_)} = MinX.
corMaxX(MaxX) :- #max{X:ison(owncorral,X,_,_)} = MaxX.
corMinY(MinY) :- #min{Y:ison(owncorral,_,Y,_)} = MinY.
corMaxY(MaxY) :- #max{Y:ison(owncorral,_,Y,_)} = MaxY.

corCX(X) :- corMinX(A), corMaxX(B), #int(W), B=W+A,
            #int(Wo2), W=Wo2+Wo2, X=A+Wo2.
corCX(X) :- corMinX(A), corMaxX(B), #int(W), B=W+A,
            #int(Wo2), #int(WTemp), WTemp=Wo2+Wo2,
            W=WTemp+1, X=A+Wo2.

corCY(Y) :- corMinY(A), corMaxY(B), #int(H), B=H+A,
            #int(Ho2), H=Ho2+Ho2, Y=A+Ho2.
corCY(Y) :- corMinY(A), corMaxY(B), #int(H), B=H+A,
            #int(Ho2), #int(HTemp), HTemp=Ho2+Ho2,
            H=HTemp+1, Y=A+Ho2.
```

```
corralcenter(owncorral, X, Y) :- corCX(X), corCY(Y).
```

Statt der Konstante owncorral kann das Program auch generisch erweitert werden, um sowohl den eigenen als auch den feindlichen Pferch aufzuspüren. Die doppelte Auslegung der corCX, corCY Regeln ist notwendig, um sowohl gerade wie ungerade Längen auf den Spielfeldachsen abzufangen.

Logisches Programm 5.2.6 (Kommunikation - Agentenposition)**.** *Aus den Wahrnehmungen allein erfährt der Agent lediglich seine eigene Position, und welche Agenten seines oder des gegnerischen Teams in seiner Umgebung sind. In den ison-Prädikaten ist allerdings nicht kodiert, um welchen Freund es sich handelt. Hierfür leitet jeder Agent seine Position explizit ab, und erstellt eine Nachricht, die er verschicken möchte.*

```
agentposition_msg(ID,X,Y,T) :- ison(ID,X,Y,_), iam(ID), time(T).

agentposition(ID,X,Y) :- agentposition_msg(ID,X,Y).

:- agentposition_msg(ID,X,Y,T1), agentposition_msg(ID,X,Y,T2),
   T1 < T2.
```

Der Fakt agentposition_msg wird von jedem Agenten zu Beginn einer Spielrunde der Agentengesellschaft mitgeteilt. Das Constraint stellt sicher, das keine veralteten Nachrichten über Agentenpositionen herangezogen werden.

Kapitel 6

Strategien

6.1 Strategien im Szenario

D. Hölzgen, M. Kruse

6.1.1 Kühe treiben

In diesem Abschnitt soll das Vorgehen zum Eintreiben von Kühen in den eigenen Pferch beschrieben werden. Hierbei werden zunächst grundlegende Abläufe, Begriffe und Vorgehensweisen erklärt, auf deren Grundlage anschließend konkret auf die Generierung der notwendigen Desires und deren Erfüllung eingegangen werden kann.

Beteiligte Rollen

Am Treiben der Kühe sind drei verschiedene Rollen beteiligt, wobei hier beachtet werden muss, dass sich nicht nur die vom Agent übernommene Rolle ändert, sondern dieser auch mehrere Rollen gleichzeitig übernehmen kann.

Die erste Rolle, die ein Agent für die Durchführung der Strategie zum Kühe treiben übernimmt, ist die des Leaders. Dieser stellt die Gruppe zusammen und koordiniert deren Abläufe. Die beiden anderen beteiligten Rollen sind der Driver und Door Opener, welche von weiteren Agenten je nach aktueller Situation übernommen werden. Hierbei dient der Driver dem Treiben von Kühen durch das Einnehmen einer Position nahe der Kühe, der Door Opener hingegen dem Öffnen der Gatter, welche sich auf dem Weg und vor dem Pferch befinden. Üblicherweise übernehmen alle an der Gruppe beteiligten Agenten, auch jener Agent, welcher die Leader Rolle übernimmt, die Driver Rolle, und nur bei Bedarf übernehmen ein oder zwei Agenten temporär die Door Opener Rolle.

Die beteiligten Rollen sind im Kapitel 3.6.2 zur MAS Konzeption näher beschrieben.

Ablauf

Im Folgenden sei zum besseren Verständnis ein Überblick über den gesamten Ablauf gegeben. Das Treiben von Kühen beginnt damit, dass ein Agent ein Desire Kühe zu treiben zu seinem Goal auswählt. Dadurch übernimmt er automatisch die Leaderrolle und beginnt, abhängig von der Größe der Herde, eine Gruppe zusammenzustellen, indem er Hilfeanfragen an andere Agenten sendet.

Diese Anfragen können darin resultieren, dass bei einigen anderen Agenten das Desire, dem anfragenden Agenten zu helfen ausreichend stark motiviert wird, um als Goal ausgewählt zu werden. Diese nehmen nun an der Gruppenbildung teil, indem sie zunächst ihre Bereitschaft zu Helfen kommunizieren. Aus diesen Hilfeangeboten kann der Leader nun wählen und den entsprechenden Agenten zusagen, wodurch die Gruppe für diese Aufgabe gebildet ist.

Anschließend zerlegt der Leader die Aufgabe in Teilschritte in Form von Wegpunkten, und übermittelt jedem Gruppenmitglied neben einigen Zusatzinformationen seine vorläufige Rolle sowie den ersten Wegpunkt. Hierbei ist zu beachten, dass auch der Leader selbst weitere Rollen übernehmen kann und wird. Diese Informationen kann jeder Agent nun nutzen, um seinen individuellen Plan zu erstellen, den ersten Wegpunkt zu erreichen, was zunächst üblicherweise das Einnehmen der Position in der Formation bedeutet. Nachdem dies abgeschlossen ist, meldet jeder Agent den Erfolg seines Teilziels.

Dieser Vorgang wird nun für alle vom Leader gefundenen Teilschritte wiederholt, bis entweder die Aufgabe abgeschlossen ist oder ein Agent bei der Ausführung seiner Aufgabe auf Probleme gestoßen ist, welche er alleine nicht lösen kann. In diesem Fall gilt die komplette Aufgabe als gescheitert, was jedoch, die entsprechende Situation vorausgesetzt, im Normalfall in einer kompletten Neuaufnahme und damit Neuplanung dieser Aufgabe endet.

Fehlerbehandlung

Bei der Ausführung von Plänen können zwei unterschiedliche Typen von Fehlern auftauchen. Zum einen kann es immer mal wieder vorkommen, dass ein Einzelschritt eines einzelnen Agenten fehlschlägt. Dies hängt mit der dynamischen, nicht deterministischen Umgebung zusammen und ist üblicherweise einfach zu korrigieren, ohne dass die Gruppe über den Fehlschlag informiert werden sollte. Tritt jedoch ein Fehlschlag auf, zu dessen Korrektur sich der Agent nicht selbst imstande sieht, so informiert dieser den Leader, welcher daraufhin die gesamte Gruppe informiert, dass die Erfüllung der Aufgabe gescheitert ist[1]. Solche Fehler können auftreten, wenn der Plan einen Weg durch ein unbekanntes Gebiet beinhaltet. Muss in diesem beispielsweise ein Zaun durchquert werden, so muss die Planung unter Beachtung dieser neuen Information neu angestoßen werden.

[1] Sollte aufgrund der aktuellen Situation dennoch das gleiche Desire am stärksten motiviert sein, so kann es bei grundsätzlicher Erfüllbarkeit erneut von der Deliberation als Goal ausgewählt werden, um ein erneutes Planen für die ganze Gruppe zu veranlassen.

Individuelle Fehlerbehandlung Mit der individuellen Fehlerbehandlung beginnt ein Agent, sobald ein (Teil-)Plan fehlschlägt. Im Normalfall wird dies durch einen nicht erfolgten Schritt oder einen durch eine Kuh blockierten Weg verursacht, und kann durch ein einfaches Neuplanen des Teilziels des Agenten behoben werden. Fehlschläge dieser Art werden nicht kommuniziert.

Fehlerbehandlung in der Gruppe Ist ein Neuplanen jedoch nicht erfolgreich durchführbar, so wird der Leader der Gruppe benachrichtigt. Hierzu wird die Information

$planfail(self, task, time)$

übertragen, welches wiederum den Leader veranlasst, der gesamten Gruppe die Information

$taskabort(self, task, time)$

zu übermitteln. Dies hat ein Auflösen der Gruppe und das Verwerfen, von allen zur Planausführung erzeugten Informationen, zur Folge. Sollte der Leader selbst einen Fehlschlag feststellen, den er durch individuelle Neuplanung nicht lösen kann, entfällt der erste Schritt. Die Entscheidung, bei einem solchen Fehlschlag komplett aufzugeben ist darin begründet, dass die notwendige Neuplanung der Aufgabe, sofern sinnvoll und möglich, durch Wiederaufnahme des Desires als Goal diese Funktion ohnehin übernimmt.

Konventionen

Um das gemeinsame Lösen der Aufgaben weitestgehend zu unterstützen, sind gewisse Konventionen festgelegt, deren Erklärung keiner weiteren Kommunikation bedarf und welche von allen Agenten bei der Planung und Ausführung der Pläne berücksichtigt werden.

Gruppenbildung Die hier vorgestellte Gruppenbildung geht davon aus, dass wie in Abschnitt 6.1.1 beschrieben, zwei verschiedene Motive zum Ausführen einer Aktion, für deren Erfolg ein Agent nicht ausreichend ist, und zum Helfen bei einer solchen Aktion, existieren. Der Ablauf der Gruppenbildung sieht dabei wie folgt aus:

- Der Leader stellt eine Hilfeanfrage indem er die Information

  $help(self, task, obj, intensity, time)$

  weitergibt. Hierbei steht die *intensity* für die Intensität seines als Goal ausgewählten Desires, welches er verfolgt. Dies ist für die anderen Agenten notwendig, um festzustellen bei welchem Agenten die Hilfe den größten Effekt hätte.

- Die Helper (Driver und Door Opener) können, sofern sie dazu bereit sind, ihr Angebot zu helfen mittels

  $ack(self, task, obj, intensity, time)$

übermitteln. Auch hierbei ist die Intensität des zugehörigen Desires notwendig, um festzustellen, welcher Agent am ehesten für die Hilfe ausgewählt werden sollte.

- Der Leader wählt aus den Hilfeangeboten diejenigen Agenten mit der höchsten Intensität aus. Diesen kann er mittels

  $accepted(self, task, obj, time)$

  mitteilen, dass er sie ins Team aufgenommen hat, den anderen mittels

  $rejected(self, task, obj, time)$

  mitteilen, dass sie anderen Aufgaben nachgehen können. In diesem Fall kann das Wissen über die Hilfeanfrage wieder verworfen werden. Sollten nicht genügend Agenten gefunden worden sein, so muss er allen Agenten eine Absage erteilen. Um sicherzustellen, dass nicht sofort versucht wird diese Aufgabe erneut zu lösen, geschieht die Absage in diesem Fall mittels

  $rejectedall(self, task, obj, time)$.

  Diese Information soll für eine gewisse Zeit im Wissen des Agenten verbleiben und ihn von der Wiederaufnahme des Desires abhalten, wobei dies sowohl für den Leader als auch für alle anderen Agenten gilt[2]. Um auch diejenigen Agenten, die bisher kein Hilfegesuch abgeschickt haben, das Ende der Gruppenbildung mitzuteilen, wird folgende Information an alle Agenten weitergegeben:

  $groupformed(self, task, obj, time)$.

  Dies ist notwendig, damit nicht andere Agenten nach Abschluss ihrer Aufgaben versuchen, an dem schon abgeschlossenen Gruppenbildungsprozess teilzunehmen.

Auch nach der erfolgreichen Gruppenbildung werden keine Informationen über die Gruppenmitglieder verbreitet, da die Koordination einzig über den Agenten erfolgt, welcher die Leader Rolle übernommen hat.

Identifikation von Herden Um die Komplexität und den notwendigen Informationsaustausch möglichst gering zu halten, wird nur eine einzige Kuh als Repräsentation der Herde ausgewählt. Diese sollte den Mittelpunkt einer als kreisförmig angenommenen Herde darstellen und als solche die meisten anderen Kühe in ihrer Nachbarschaft aufweisen. Hierzu werden die Motive, wie in Abschnitt 6.1.1 beschrieben, dahingehend angepasst, dass die Intensität der generierten Desires diese Eigenschaft ausdrückt.

Der Radius des Kreises um die zum Treiben gewählte Kuh als Herdenmittelpunkt lässt sich demnach aus der Intensität des zugehörigen, als Goal ausgewählten Desires ableiten:

$r = \lceil i * 0.05 \rceil$

[2]Die Intensität des Desires selbst wird hierbei nicht angepasst, da dies nicht dem Konzept der Motivation entspricht

Dieser Radius ist zusammen mit der Information über die ausgewählte Kuh als Herdenmittelpunkt ausreichend, um ein eindeutiges Wissen über die Position der Herde und deren Ausmaß sicherzustellen. Die grundsätzliche kreisförmige Betrachtung der Herde erleichtert zudem die Ausrichtung der Agenten zum Treiben der Kühe in eine gewünschte Richtung.

Wegpunkte Der Weg zum Treiben der Kühe wird anhand von Wegpunkten bestimmt. Diese richten sich jedoch nicht nach der Position der beteiligten Agenten, sondern nach der Position der Herde selbst, welche durch die ausgewählte Kuh als Mittelpunkt repräsentiert wird. Somit muss insbesondere bei Wegpunkten hinter Gattern und im eigenen Pferch darauf geachtet werden, dass nicht nur die Herde, sondern auch die Positionen der beteiligten Agenten auf der richtigen Seite des Pferchs sind. Die zum Treiben notwendige Position sowie der konkrete Weg zum Ziel, können die einzelnen Agenten aus der Position des Wegpunktes, dem Radius[3], der Richtung des *aktuellen* Wegpunkts, in welche die Herde getrieben werden soll sowie der Position, welche der Agent, in der unter Abschnitt 6.1.1 vorgestellten Formation, einnehmen soll.

Ein Wegpunkt wird wie folgt übermittelt:

$defaultwp(self, task, obj, step, x, y, direction, formationpos, agentcount, time)$,

wobei

step für die aufsteigende Nummer des Wegpunkts steht

x, y die Koordinaten der Kuh darstellen

direction die Richtung bestimmt, in welche sich nach Einnehmen des Wegpunkts gedreht werden soll, um die Herde anschließend in diese Richtung weitertreiben zu können. Hierbei sei explizit darauf hingewiesen, dass dies nicht die Richtung darstellen muss, aus welcher die Kühe zu diesem Wegpunkt getrieben wurden. Mögliche Richtugsangaben sind die von Massim verwendeten Angaben zur Bewegung: *north*, *south*, *east*, *west*, *northeast*, *northwest*, *southeast* und *southwest*.

$formationpos, agentcount$ die Position des Agenten in der Formation sowie die Anzahl aller Agenten in dieser angibt, um die konkrete Position wie in Abschnitt 6.1.1 beschrieben unabhängig von Informationen über die Formationsposition der anderen Agenten einnehmen zu können.

Ein Wegpunkt gilt als erreicht, wenn die als Herdenmittelpunkt gewählte Kuh die gewünschte Position bis auf ein Feld genau erreicht hat, und als abgeschlossen, wenn nach Aufforderung alle Agenten die neuen Positionen gemäß der Richtung des Wegpunkts eingenommen haben.

Neben diesem Wegpunkttyp, ohne besondere zusätzliche Bedeutung, gibt es weitere Typen von Wegpunkten, welche zusätzliche Implikationen mit sich bringen und im Folgenden kurz vorgestellt sein:

[3]Der Radius wird jedoch nicht für jeden Wegpunkt, sondern zu Beginn der Aufgabe vom Leader übermittelt.

startwp Dieser Wegpunkt stellt den ersten Wegpunkt der Strecke dar. Die Kühe müssen demnach nicht zum Wegpunkt getrieben werden, so dass es ausreicht wenn die einzelnen Agenten ihre Positionen in der Formation einnehmen. Der Wegpunkt gilt als abgeschlossen, wenn alle Agenten ihre Position eingenommen haben.

fenceopenwp Dieser Wegpunkt stellt einen Wartepunkt vor Gattern dar. Während ein oder zwei Agenten die Rolle des Door Openers übernehmen, um das Tor zu öffnen, muss der Rest der Agenten auf diese Aktion warten. Zudem geben alle Agenten gemäß der Konvention bei Erreichen dieses Wegpunktes ihre Position zum Schalter des Gatters an, sofern diese bekannt ist, um dem Leader die Auswahl der Agenten für das Öffnen des Gatters zu erleichtern. Hierbei gilt der Wegpunkt als abgeschlossen, wenn das Gatter geöffnet wurde. Es sei angemerkt, dass die Formation des nächsten Wegpunktes berücksichtigen sollte, dass für diesen nun ein Agent weniger zur Verfügung steht.

fenceclosewp Dieser Wegpunkt stellt den Wartepunkt nach dem Passieren eines Gatters dar. Hier wird ebenso wie beim *fenceopenwp* vorgegangen, da ein weiterer Agent das Gatter aufhalten muss, damit der vorherige Door Opener selbiges passieren kann. Der Wegpunkt gilt als vollständig abgeschlossen, wenn alle Agenten zurück in der Formation sind.

endwp Statt einem *fenceclosewp* kann auch dieser Wegpunkttyp auf einen *fenceopenwp* folgen. Er steht für den Sonderfall, dass das geöffnete Gatter das Gatter des eigenen Pferchs war. Dieser Wegpunkt gilt als abgeschlossen, wenn die Kuhherde die gewünschte Position erreicht hat.

Bewegen in einer Formation Zum Treiben der Kühe gehen die Agenten in einer festgelegten Position vor. Zur exakten Bestimmung seiner Position in der Formation muss ein Agent folgendes wissen:

- Die gewählte Kuh als Mittelpunkt der Herde
- Die Richtung, in welche sich die Kuh bewegen soll
- Den Radius, welcher für die Herde um die Kuh im Mittelpunkt errechnet wurde
- Die zugewiesene Positionnummer nach dem in der Abbildung 6.1 verdeutlichten Schema
- Die Anzahl der beteiligten Agenten

Bis auf den Radius der Herde werden alle weiteren Informationen für jeden Wegpunkt neu übermittelt, da sie einer steten Änderung unterliegen[4]. Die Wahl

[4]Dies tut der Radius praktisch auch, er wird jedoch bei diesem Vorgehen nicht neu berechnet.

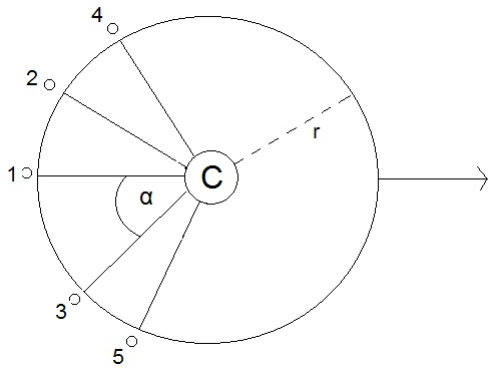

Abbildung 6.1: Bewegen in einer Formation

der Nummerierung ist darin begründet, dass sie auf der einen Seite unabhängig von der Anzahl der Agenten einheitliche Schlüsse zulässt und auf der anderen Seite die Formation nur wenig geändert werden muss, wenn die am Rand liegenden Agenten aus dieser austreten.

Der Winkel sollte konfigurierbar sein, als Startwert der Winkelsumme werden 150° genutzt. Diese werden per Konvention dem Initialwissen des Agenten hinzugefügt, und bedürfen keinerlei Kommunikation.

Einnehmen der Positionen. Während die Agenten ihre Positionen in der Formation einnehmen, muss darauf geachtet werden, die Herde selbst nicht auseinander zu treiben. Zu diesem Zweck sollte die gesamte Herde, also entweder das ermittelte Cluster oder der geschätzte Kreis, als virtuelles Hindernis mit in die Planberechnung einbezogen werden. Hierbei kann es sinnvoll sein, nach Möglichkeit einen etwas größeren Abstand zu halten.

Positionsfindung in Engstellen. In Engstellen kann es vorkommen, dass die Positionen nicht wie gewünscht eingenommen werden können. In diesem Fall sollte jeder Agent wie in Abbildung 6.2 veranschaulicht, seine Position jeweils entlang des Kreises, weg von der gewünschten Bewegungsrichtung, möglichst genau einzunehmen versuchen.

Treiben in der Formation. Sind die Agenten wie beschrieben ausgerichtet, genügt es zum Treiben der Kühe in die gewünschte Richtung die Agenten in diese Richtung zu bewegen.

Drehen der Formation. Neben der Bewegung in die gewünschte Richtung ist es üblicherweise notwendig, diese Richtung von Zeit zu Zeit anzupassen. Hierzu werden von den Agenten wie in Abbildung 6.3 gezeigt, nach Erreichen des Wegpunktes, zunächst entlang des Kreises, die neuen Positionen eingenommen, anschließend kann mit dem Treiben fortgefahren werden. Hierbei ist darauf

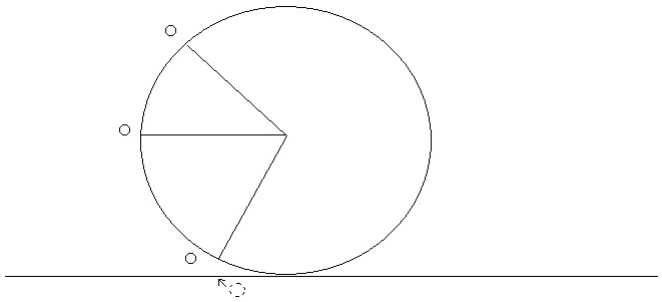

Abbildung 6.2: Formation an Engstellen

zu achten, bei der Wahl Wegpunkte für die Herde den Umfang dieser mit einzuberechnen, wie Abbildung 6.4 zeigt.

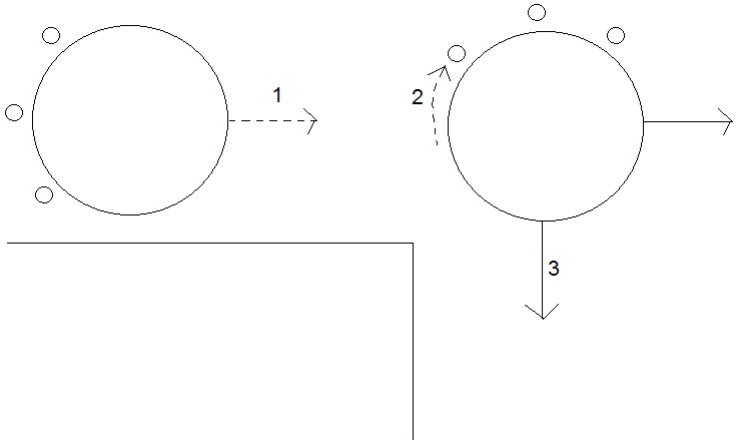

Abbildung 6.3: Richtungsaenderung der Formation

Generierung der Desires durch Motive

Da die Wünsche das den Agenten antreibende Element darstellen, müssen diese entsprechend angepasst werden, um die hier vorgestellte Strategie zu unterstützen. Um sicherzustellen, dass jeweils der Agent Hilfe beim Treiben der Kühe erhält, bei dem sie am effektivsten eingesetzt ist, und auch diejenigen Agenten zur Hilfe ausgewählt werden können, bei welchen dies den geringsten Schaden erzeugt, werden zwei verschiedene Motive eingeführt. Während das eine das Verlangen beschreibt, die Kühe in den Pferch zu treiben, beschreibt das andere den Antrieb, den befreundeten Agenten zu helfen.

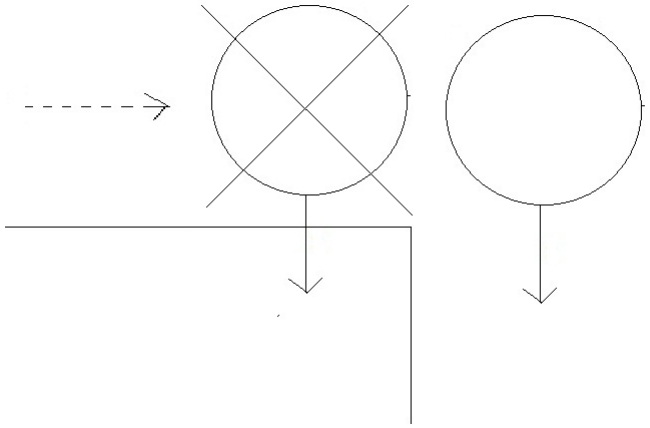

Abbildung 6.4: Wahl der Wegpunktberechnung

Leader. Dieses Motiv steht für das Verlangen, die Kühe in den eigenen Pferch zu treiben. Die hieraus erzeugten Desires werden immer nur für eine einzelne Kuh erzeugt, wobei sich die Intensität nach der Distanz des Agenten zu dieser Kuh sowie der sich in der Nachbarschaft zur Kuh befindlichen anderen Kühe errechnet. Durch Zusagen anderer Agenten wird die Intensität zusätzlich erhöht, jedoch *nicht* durch Hilfeangebote. Ein durch dieses Motiv generierte Desire sieht wie folgt aus: $drivencow(X)$

Helper. Dieses Motiv steht für den Antrieb befreundeten Agenten zu helfen. Die Intensität dieses Motivs leitet sich direkt aus der Intensität der Hilfeanfrage ab, wobei diese exakt um den von der Deliberation verwendeten Bias zum Goal Wechsel höher ist, als wenn es sich um das eigene Leader Motiv handeln würde. Die Intensität steigt zusätzlich, wenn eine Zusage gegeben wurde oder diese akzeptiert wurde. Um eine effiziente Auswahl der helfenden Agenten zu unterstützen, wird die Intensität jedoch abhängig von der Entfernung zur Kuh wieder herabgesetzt.

Rangordnung. In seltenen Fällen kann es vorkommen, dass sich zwei Agenten in so ähnlichen Situationen befinden, dass ihre Leader Desires und die daraus für den jeweils anderen Agenten resultierenden Helper Desires exakt gleich motiviert sind. Für diesen speziellen Fall wird bei der Konfiguration eine Rangordnung der Agenten eingegeben, wobei der in der Rangordnung höhere Agent die Hilfe des anderen Agenten erhält.

Treiben von Kühen

In diesem Abschnitt werden die einzelnen Vorgänge beim Treiben von Kühen im Zusammenhang erläutert, unter Berücksichtigung der vorgestellten Vorge-

hensweisen und Konventionen.

Gruppenbildung Bei der Gruppenbildung wird, wie in Abschnitt 6.1.1 vorgestellt, zunächst ein Hilfegesuch an die übrigen Agenten geschickt. In Abhängigkeit vom Radius werden nun die n höchst motivierten Agenten für die Gruppe ausgewählt, wobei

$n = 3$, für $intensity < 60$

$n = 4$, für $intensity \geq 60 \wedge intensity < 80$

$n = 5$, für $intensity \geq 80$

Diese Zahlen sind aus den Erfahrungen in Testspielen abgeleitet. Mit weniger als drei Agenten ist es insbesondere beim Öffnen von Gattern unmöglich die Herde weiter treiben zu können, mehr als fünf Agenten werden ineffizient und sind auf vielen Karten an Engstellen nicht mehr gut zu platzieren. Zudem ermöglicht die Beschränkung auf fünf Agenten das Zustandekommen von insgesamt vier starken Teams von Agenten.

Nach der Gruppenbildung muss für die weitere Vorgehensweise zunächst die Information über die zu treibende Herde ausgetauscht werden. Der Leader informiert daher alle an der Gruppe beteiligten Agenten mittels

$herd(obj, radius, time)$

über den von ihm ermittelten Radius. Dieser ist später notwendig, um die richtige konkrete Position in der Formation zu errechnen und bei Bewegungen nicht die Herde auseinander zu treiben.

Treiben anhand von Wegpunkten Zu Beginn, bei unbekannter Strecke auch nach erreichten Teilzielen, zerlegt der Leader die zum Treiben der Herde aus der Planung hervorgegangene Strecke zunächst in Teilziele in Form von Wegpunkten. Diese können, neben dem Standardtyp, auch von anderen der in Abschnitt 6.1.1 vorgestellten Typen sein, sind zunächst nur dem Leader bekannt und unterliegen einer festen Reihenfolge.

Der jeweils nächste Wegpunkt wird vom Leader an die Gruppe übermittelt, was gleichzeitig als Startbefehl dient. Jeder an der Gruppe beteiligte Agent errechnet sich aus den im Wegpunkt enthaltenen Informationen sowie aus dem zu Beginn übermittelten Radius sein nächstes Ziel sowie den Weg dorthin aus. Nun werden zunächst zwei Schritte ausgeführt:

- Zunächst legt der Agent den errechneten Weg zum Wegpunkt zurück. Das Ziel gilt hierbei als erreicht, wenn er seine eigene Zielposition, unabhängig von der Herde, erreicht hat. Dies übermittelt er mittels

  $inposition(self, step, time)$

  an den Leader. Stellt dieser nach einer gewissen Wartezeit fest, dass nicht nur alle Agenten ihre Position erreicht haben, sondern auch die Herde

selbst in die gewünschte Position getrieben wurde, so gilt der Wegpunkt als erreicht und er weist die übrigen Agenten mittels

$getinformation(self, step, time)$

an, ihre Position gemäß der Richtung des erreichten Wegpunktes einzunehmen. Dies geschieht wie in Abschnitt 6.1.1 beschrieben unter Berücksichtigung der Herde als virtuelles Hindernis, um die Kühe nicht auseinander zu treiben.

- Die Agenten nehmen nun ihre neue Position ein, und informieren den Leader mittels

  $information(self, step, time)$

  über den Erfolg dieser Aktion. *Sonderfall Startwegpunkt:* Da der Startwegpunkt der erste Wegpunkt ist, genügt es hier, direkt die Position gemäß der Richtung des Wegpunktes einzunehmen, und anschließend über den Erfolg zu informieren.

Die Sonderbehandlung für spezielle Wegpunkttypen außer Acht gelassen, ist hiermit die Behandlung eines Wegpunkts abgeschlossen, und der Leader kann den nächsten Wegpunkt übermitteln, bis die Gruppe die Herde in den eigenen Pferch getrieben hat. Zusätzlich sollten nach jedem Wegpunkt neue Informationen weitergegeben werden. Dies ist insbesondere dann hilfreich, wenn der Weg durch unbekanntes Terrain führt, wird jedoch auch genutzt, um die Position von Kühen mitzuteilen, die außerhalb des Sichtfeldes von anderen Agenten liegt.

Sollte das Treiben fehlschlagen, weil etwa die Herde nicht zur gewünschten Position getrieben wurde, so schlägt der Gesamtplan fehl, und so bricht der Leader wie in Abschnitt 6.1.1 beschrieben, die Aufgabe ab, und gegebenenfalls mit einer Neuplanung begonnen werden kann.

Koordiniertes Fortbewegen. Leider kann man nicht genau vorhersagen, wie sich die Kühe bewegen. Dies ergibt das Problem, dass man nicht genau weiß, wie schnell man sich fortbewegen muss, damit man die Kühe treibt, ohne diese auseinander zu treiben. Aus diesem Grund wird bei der Planerstellung darauf geachtet nicht nur den Weg, sondern auch die Wartezeiten einzuplanen. In dem Fall würde nur dann neu geplant werden müssen, wenn der Plan fehlschlägt (etwa weil eine Kuh nicht schnell genug gelaufen ist und den Weg versperrt), was im Zuge der individuellen Fehlerbehandlung ohne Auswirkungen auf die Gruppe geschehen würde.

Vorgehensweise bei Gattern. Zur Behandlung von Gattern wurden vor und nach einem Gatter spezielle Wegpunkte zu deren besonderer Behandlung hinzugefügt. Der Ablauf hierbei ist der folgende:

- Zunächst erreichen die einzelnen Agenten den Wegpunkt vor dem Zaun und übermitteln dem Leader, sofern bekannt, mittels

  $distancetoswitch(self, obj, dist, time)$

über ihre Distanz zum Switch[5]. Nun gibt es zwei Möglichkeiten:
Entweder der Switch ist bekannt, dann ist auch der Agent bekannt, dann wird mittels

$openswitch(self, obj, time)$

der Agent mit der geringsten Distanz zu diesem aufgefordert, den Switch zu öffnen. Ist der Switch nicht bekannt, so werden die beiden Randagenten mittels

$startlookforswitch(self, direction, time)$

aufgefordert, den Switch zu suchen, wobei sich die Richtung aus Änderungen der Wegpunktrichtung um jeweils 90° im und gegen den Uhrzeigersinn ableiten. Anschließend wird derjenige Agent, welcher den Switch gefundet hat, zum Öffnen aufgefordert, der andere mitttels

$getinformation(self, step, time)$

in die Formation zurückbeordert.

- Da nicht sichergestellt ist, dass sich das Gatter im Sichtfeld des Leaders befindet, teilt der Door Opener den Status des Zauns gemäß der normalen Informationsweitergabe nach dem Öffnen mit. Für den Fall, dass der Switch zunächst gesucht werden musste, wird nun noch auf die Information

  $information(self, step, time)$

  des anderen Suchers gewartet, anschließend übermittelt der Leader den nächsten Wegpunkt an die Gruppe.

- Ist der Wegpunkt erreicht, wird die obige Prozedur mit den übrigen Agenten erneut durchgeführt, um das Gatter von der anderen Seite offen zu halten. Ist dies sichergestellt, wird zunächst der erste Door Opener, und nach dessen Erfolg der Zweite zurück in die Formation beordert. Hierbei ist zu beachten, dass der Erste den Wegpunkt übermittelt bekommt, der Zweite jedoch nur zurück in die Formation beordert wird.

- Nachdem alle Agenten wieder in der Formation sind, kann mit dem nächsten Wegpunkt fortgefahren werden.

Vorgehensweise beim eigenen Pferch. Die Vorgehensweise im eigenen Pferch unterscheidet sich in der Behandlung des Wegpunkts auf der anderen Seite des Gatters, was sich auch durch den anderen Typ des Wegpunkts zeigt. Hierbei ist die Behandlung des Wegpunkts vor dem Zaun die selbe, die Behandlung des Endwegpunkts innerhalb des Pferchs läuft dabei wie folgt ab:

- Hat ein Agent den Wegpunkt erreicht, so übermittelt er dies dem Leader wie gewohnt

[5] Es wird hier zunächst davon ausgegangen, dass dies durch eine gewisse räumliche Nähe zum Gatter direkt aus dem Grundwissen des Agenten ableitbar ist, sofern der Switch bekannt ist.

- Stellt der Leader fest, dass der Wegpunkt als erreicht gilt[6], so übermittelt er allen Agenten außer dem Door Opener die Information

  $leavecorral(self, time)$,

  was die Agenten dazu veranlasst den Pferch zu verlassen. Den Erfolg dieser Aktion können diese mittels

  $leftcorral(self, time)$

  an den Leader übermitteln, was ein Verlassen der Gruppe impliziert. Befindet sich kein Agent mehr innerhalb des Corrals, so weist der Leader den Door Opener mittels

  $closecorral(self, time)$

  an, das Gatter zu schließen. Auch hier übermittelt der Door Opener die Information über den geänderten Gatterstatus an den Leader, was auch in diesem Fall ein Verlassen der Gruppe impliziert.

Durch den Erhalt der letzten Information verlässt auch der Leader die Gruppe und das Ziel gilt als erreicht.

6.1.2 Scout

B.Jablkowski, F.Bienek

Nachdem die Strategie des Treibens einer Kuhherde erläutert wurde, soll nun etwas näher auf das Erkunden der Welt eingegangen werden. Die Rolle des scouts besteht darin, noch nicht erforschte Weltzellen zu erkunden und nach Kühen zu suchen. Wir haben uns dazu entschlossen, diese Aufgabe in Gruppen auszuführen. Dies hat den Vorteil, dass ein Rollenwechsel (von *scout* zu *cowdriver*) weniger Probleme in der Gruppe macht. Außerdem gehen wir von einer statischen Gruppe aus. Welcher Agent der *leader* dieser Gruppe ist, wie groß diese ist und wer zu dieser Gruppe gehört, wird in der entsprechenden Konfigurationsdatei festgelegt. Der Vorteil dieser Vorgehensweise ist, dass man keine Zeit für die Gruppenbildung und die damit verbundenen Probleme verliert (frühere Implementierungen haben gezeigt, dass eine saubere Gruppenbildung in diesem Szenario kaum zu erreichen ist). Ein Agent, der eine Rolle wechseln und einer anderen Gruppe beitreten möchte, muss viele Dinge beachten. Eine davon ist, seine *beliefs* für die bis gerade ausgeführte Rolle zu verwerfen und gegebenenfalls auch andere Agenten über seinen Rollenwechsel zu informieren. Der Rollenwechsel und die Gruppenbildung erfordern viel Kommunikation und Zeit. Der Nachteil von einer nicht vorhandenen dynamischen Gruppenbildung ist, dass man mit einer statischen Anzahl an Gruppenmitgliedern nur bestimmte, aber beim weiten nicht alle Probleme lösen kann.

[6] In Praxistests könnte sich herausstellen, dass hierzu weniger strikte Kriterien notwendig sind.

Gruppenstruktur Der statische *multiscout* unterscheidet zwischen zwei internen Rollen, die des *leaders* und die des *helpers*. Jede Gruppe, unabhängig von der Anzahl der Agenten, die sie bilden, hat einen leader. Der Rest der Agenten, die zu dieser Gruppe gehören, werden zu *helpern*. Das Wissen darüber, welche Unterrolle man in der Gruppe ausübt, wird in den *beliefs* des Agenten gespeichert.

Kontextsensitivität In der *multiscout*-Rolle werden zwei Fälle unterschieden. Der erste Fall beschreibt die Situation in welcher die Welt noch nicht ganz erkundet wurde. Der zweite Fall befasst sich mit dem Kontext einer vollkommen oder zum Teil erforschten Welt. Im ersten Fall glaubt der *scout*, die Welt sei noch nicht komplett erkundet und versucht deswegen, sich hauptsächlich auf die unerforschten Felder zu konzentrieren. Im Falle, dass der *scout* glaubt, die Welt sei schon erkundet, verwirft er das Kriterium und scoutet die schon zuvor erkundeten Felder, um über diese die aktuelle Informationen zu erhalten.

Die Zielfindung

Bevor eine Gruppe von Scouts mit der Erkundung der Welt anfangen kann, muss diese erst ein Ziel dafür auswählen. Für diese Bestimmung ist der *leader* verantwortlich. An dieser Stelle müssen ein paar Kriterien berücksichtigt werden. Das ausgewählte Ziel darf weder gerade von anderen erkundet werden noch ein potentielles Ziel eines höher priorisierten Agenten sein. Wurde ein Ziel ermittelt und es existieren keine Konflikte bezüglich dieses Ziels muss der *leader* den anderen *scout leadern* mitteilen, dass er beabsichtigt, dieses Gebiet zu erkunden. Agenten sollen sich sinnvoll verteilen. Um dies zu gewährleisten, muss jeder Agent eine Menge von Ausweichpositionen besitzen. Sollten alle Positionen abgelehnt werden, so nimmt der Agent an, dass die Welt in diesem Zeitpunkt bereits optimal erforscht wird. Abhängig vom Kontext werden zwei Vorgehensweisen zur Zielfindung unterschieden. Befindet sich die Gruppe in einem Kontext, indem die Welt noch nicht erkundet worden ist, so verhält sich der *leader* wie folgt:

1. Der *scout* sucht randomisiert x Punkte in der Welt die er als *unexplored* annimmt.

2. Der *scout* berechnet für jeden Punkt ein Gewicht anhand seiner Umgebung. Die Gewichte erzeugen eine Ordnung. Das Gewicht K_i eines Punktes i wird berechnet durch die Summe aller anliegenden Felder, die maximal eine Distanz von δ zum Zielpunkt haben. δ kann frei gewählt/angepasst werden.

$$K_i = \sum_{i-\delta}^{i+\delta} \sum_{j-\delta}^{j+\delta} F_{ij}$$

   mit $F_{ij} = 1$, falls $ison(unexplored, i, j, \_)$, sonst 0

3. Für alle Punkte mit gleichem Gewicht berechnet er eine Unterordnung anhand der Distanz zum Pferch. Je kleiner die Distanz, desto größer die Priorität.

   Anmerkung zu den Gewichten Als erstes Kriterium wurde das Gewicht gewählt, welches die unerforschten Felder zusammenrechnet, weil es für einen *scout* plausibler ist, möglichst große Ansammlungen von unerforschten Feldern zu erforschen. Das zweite Kriterium drückt aus, dass unerforschte Felder nahe des eigenen Pferchs wichtiger erscheinen, als Felder die weiter entfernt sind, weil man so schneller Kühe in den Pferch bekommt.

4. Der am stärksten gewichtete Punkt wird als potentielles Ziel ausgewählt.

5. Der *scout* kommuniziert allen anderen *scout leadern* seinen Wunsch zu dem gewählten Punkt zu gehen. Das zu übermittelnde Prädikat ist von der Form:

   $$scoutOption(AG_{ID}, X, Y)$$

   wobei X, Y die Koordinaten des gewünschten Punktes und AG_{ID} die ID des Senders ist. Das Prädikat drückt das potentielle Ziel des Agenten aus.

6. Konfliktbehebung:

   (a) Falls kein anderer *scout leader* diesen Bereich scouten möchte oder bereits scoutet, wird der Wunsch zum eigentlichen Ziel. Im diesen Falle kommuniziert der *scout* den anderen *scout leadern* sein ausgewähltes Ziel. Das zu übermittelnde Prädikat ist von der Form:

   $$scoutDestination(AG_{ID}, X, Y)$$

   wobei X, Y die Koordinaten des gewünschten Punktes und AG_{ID} die ID des Senders ist. Das Prädikat drückt das ausgewählte Ziel des Agenten aus.

   (b) Sonst werden alle Konfliktpunkte überprüft:

      i. Wird schon mindestens einer der Konfliktpunkte von einer anderen *scout*-Gruppe erforscht, so wird das eigene potentielle Ziel verworfen. Falls es noch Kandidaten gibt, wird das nächste potentielle Ziel untersucht.

      ii. Sind die Konfliktpunkte nur Kandidaten anderer *scout leaders*, so wird die eigene Priorität mit den Prioritäten anderer Agenten abgeglichen:

         A. Hat der *scout* selbst die höchste Priorität, so kommuniziert er den anderen Agenten sein Ziel.

         B. Sonst wird das potentielle Ziel verworfen und der nächste Kandidat ausgewählt. Der Agent muss sich selbst sowie alle seine Verbündeten davon informieren.

7. Falls der *scout* keine potentiellen Ziele mehr hat, wird die Motivation zum scouten angepasst.

Glaubt der Agent, die Welt sei schon ausreichend erkundet worden ändert sich seine Strategie bezüglich der Auswahl der Ziele. Das Wissen des Agenten darüber das die Welt schon erforscht wurde wird durch das folgenden Prädikat ausgedrückt:

$$worldknown$$

In diesem Kontext verfährt der Agent wie folgt:

1. Der *scout* sucht randomisiert x Punkte in der Welt.
2. Der *scout* wählt einen beliebigen Punkt aus.
3. Vorgehensweise wie im ersten Fall ab Punkt 5.

Das Erkunden

Mit der Auswahl des Zieles beginnt die Erkundungsphase. Diese besteht darin, das ausgewählte Ziel dadurch zu erreichen, dass spezielle Wegpunkte abgelaufen werden.

Wegfindung mit waypoints Der Weg von der Startposition bis zum Ziel wird in *waypoints* unterteilt. Es sind Koordinaten, welche nacheinander von den Agenten abgelaufen werden sollen. Das hat unter anderem den Vorteil, dass die *scouts* nicht sofort den gesamten Weg planen und ablaufen müssen, sondern nur Abschnitte des gesamten Weges erkunden können. *Waypoints* werden so gewählt, dass Richtungsänderungen der *scouts* so selten wie möglich auftreten sollten. Dies hat den Vorteil, dass die Formation, in welcher die Agenten laufen, beibehalten wird. Außerdem werden *waypoints* für das Überqueren von Zäunen genutzt. Vor jedem solchen Hindernis wird ein spezieller Wegpunkt gesetzt, der sogenannte *open fence waypoint*. Hat der *leader* diesen als nächsten Wegpunkt, weiß er, dass er und seine Gruppe einen Zaun bewältigen müssen und kann die damit verbundenen Funktionalitäten des *dooropeners* aufrufen 6.1.3. Hinter dem Hindernis liegt der sogenannte *close fence waypoint*. Dieser wiederum hilft dem leader im Abschließen der Funktionalitäten des *dooropeners* und der damit verbundenen Aktionen. *Waypoints* werden nur vom *leader* der Gruppe erzeugt und müssen dann den *helpern* mitgeteilt werden. Diese brauchen jedoch nur über den nächsten Wegpunkt zu wissen, da sie in das Planen nicht involviert werden.

Pfadfindung mit dem Astar-Algorithmus Die Abschnitte zwischen den Wegpunkten werden mithilfe des Astar-Algorithmus abgelaufen. Dieser liefert dem Agenten den eigentlichen Pfad, entlang welchen dieser sich bewegen soll. Jeder *scout* berechnet den optimalen Pfad nur für sich. Dieser hängt nämlich nicht nur von den Zielkoordinaten ab aber auch von den Startkoordinaten und

da sich die Agenten in einer Formation fortbewegen, sind sie in der Welt immer relativ zum *leader* verschoben.

Fehlerbehandlung In einer unbekannten und indeterministischen Umgebung gibt es viele Situationen in welchen der *scout* sein Ziel nicht erreichen kann. Im ersteren kann es vorkommen, dass die gesetzten *waypoints* durch Gebiete verlaufen, die nach dem Erkunden sich als nicht direkt oder gar überhaupt nicht erreichbar offenbaren. Ein anderer damit verbundener Fehler ist, dass sich der *waypoint* auf einer Mauer oder einem anderen nicht begehbaren Feld befindet. Zur Zeit der Planung gab es ja kein Wissen über dieses noch unerforschte Feld. Der indeterministische Charakter der Umgebung erzeugt hingegen Probleme, auch im Falle, wo die Welt in dem zu erkundenden Bereich schon erforscht wurde. Andere Entitäten wie feindliche Agenten, Kühe oder ein zuvor geöffneter Zaun können dem *scout* den Weg versperren, obwohl dieser zur Zeit des Planens begehbar war. Eine zusätzliche Hürde ist die in der Massim-Umgebung explizit eingeplante Fehlerquote der auszuführenden Aktionen. Das bedeutet, dass jede n-te Aktion vom System aus nicht richtig ausgeführt wird. Alle diese Situationen müssen richtig erkannt und behandelt werden. Diese Probleme werden vom *scout leader* anders als von den *scout helpern* behandelt.

Fehler nahe am Ziel Eine Ausnahme von dieser Regel ist der Fall, indem sich der *scout*, sowohl *leader* als auch *helper*, so nahe am Ziel befindet, dass dieses Ziel als erreicht markiert werden kann. Wie groß diese Distanz sein darf, die den *scout* vom Ziel trennt, kann mit dem δ-Parameter bestimmt werden. Das Prädikat, mit welchem sich der Agent die Distanz merkt, sieht wie folgt aus:

$$distanceToCP(\delta)$$

Dieses Wissen wird immer dann aktualisiert, wenn ein Fehlschritt aufgetreten ist. Nach dem Erreichen des *waypoints* wird dieses Wissen aus den *beliefs* entfernt.

Allgemeine Fehlerbehandlung Im Allgemeinen löst jedoch der *scout leader* die restlichen Probleme, indem er das Neuplanen anstößt. Falls eine Laufaktion k-mal hintereinander gescheitert ist wird der aktuelle Plan verworfen und es wird neu geplant. Der k-Parameter kann auch frei bestimmt werden. Das Prädikat, welches dies ausdrückt, sieht wie folgt aus:

$$walkingfailstolerance(k)$$

Beim Neuplanen werden auch alle damit verbundenen Aktionen ausgeführt, wie das Verwerfen der alten Informationen und das Informieren der *scout helper* über die Änderungen. Das Verwerfen des alten Plans wird dadurch realisiert, dass ein speziell dafür vorgesehener Plan $!abolishScoutPlan$ ausgeführt wird. Im Falle eines *scout helpers* wird nur der Plan für das Laufen zum aktuellen *waypoint* verworfen. Der *helper* wartet dann auf weitere Anweisungen von seinem *leader*.

Offene Probleme Die obigen Ansätze sind leider nicht im Stande, alle Problemfälle zu beheben. Es hängt damit zusammen, dass die Funktionalitäten des *scouts* nur für die Zielfindung und die Erkundung angeschnitten sind. Der *scout* kann sehr gut Fehler erkennen, die durch das Scheitern einzelner Aktionen generiert wurden, es ist für ihn jedoch nicht möglich, abstrakte Sachverhalte wahrzunehmen und diese bei der Fehlerbehandlung und dem Neuplanen zu berücksichtigen. Nicht triviale Fälle, wie zum Beispiel das Erkennen, das ein Gebiet abgeschlossen ist und nicht erreicht werden kann, sollten durch die Deliberationskomponente erkannt und behandelt werden. Man kann sich leicht mehrere Fälle vorstellen, in welchen der *scout* ohne eine Unterstützung der Planerkomponente, vor allem der dynamischen Planung und einer Deliberationskomponente, welche Pläne verifiziert und vor allem auf die mögliche Ausführbarkeit testet, scheitern wird.

6.1.3 Strategieentwicklung *dooropener*

*(*S. Broszeit)

Motivation

Während des Treibens von oder Suchen nach Kühen können Agentengruppen auf Zäune, die über einen Schalter geöffnet werden können, stoßen. Um diese Hindernisse zu passieren, wird der *dooropener* verwendet. Er ist im Wesentlichen dafür zuständig den Schalter zu suchen und zu betätigen. Er ist als eine „Blackbox“ gedacht, die von den Agenten aufgerufen wird, sobald man an einem Gatter ankommt.

Entwicklung der Rolle

Türöffnen Der erste Ansatz des *dooropeners* sah vor, dass dieser einen Schalter betätigt. Dieser - im Vergleich zu den anderen Rollen - simple Ansatz ist durch eine simple Wegfindungsroutine zu realisieren.

Schaltersuche Um allerdings das dazu notwendige Wissen zur Verfügung zu haben, ist eine Suche nach dem Schalter, der zu einem Gatter gehört, erforderlich. Da sich der Schalter an beiden Seiten eines Gatters befinden kann, ist es sinnvoll zwei Agenten loszuschicken. Diese laufen zu den bekannten Endpunkten des Gatters bis endweder sie oder ein anderer Agent den Schalter gefunden haben.

Koordination Zur Koordination dieses Suchvorgangs wurde die Rolle um einen *dooropenerLeader* erweitert, der die Agenten, die losgesendet werden, um den Schalter zu suchen, auswählt, ihren Fortschritt überwacht und sie ggf. zum Schalter bzw. zurück in die Formation schickt.

Integration Der *dooropenerLeader* sollte außerdem die Anbindung an die anderen Rollen erleichtern, da diese ebenfalls auf einem *leader-helper*-Modell aufbauen und er die dynamische Gruppengröße verwalten konnte.

Regelwissen Damit der *dooropener* die beschriebene Funktionalität durchführen kann, muss er Wissen über seine Umwelt erschliessen. Um zu wissen in welche Richtung der Agent nach einem Schalter suchen soll, muss er das zugehörige Gatter als Zusammenhang erkennen und entsprechend seiner Ausrichtung zum Gatter zu dessen bekannten Endpunkten laufen. Auch um den Schalter zu betätigen, muss der Agent erkennen auf welcher Achse das Gatter liegt, da er auf ein an den Schalter angrenzendes Feld laufen muss.

Durchlaufen Um das Gatter erfolgreich zu durchlaufen, muss der *dooropenerLeader* überprüfen, ob alle Agenten das Gatter passiert haben, wenn dies - z.B. formationsbedingt - nicht der Fall ist, so müssen die Agenten weiter laufen.

Außerdem traten erstmals Schwierigkeiten mit der Integration in andere Rollen auf. Die eingangs erwähnte „Blackbox“ ließ sich schlecht mit einer unterschiedlichen Behandlung des Durchlaufens des Gatters vereinbaren, denn der *scout* sollte schnell durch das Gatter gelangen, während der *cowdriver* möglichst gleichmäßig laufen sollte, um keine Kühe zu verlieren. Da die Entwicklung des Dooropeners nicht zufriedenstellend fortgeschritten war und ein modularer Aufruf nicht fehlerfrei funktionierte, entschied sich die PG dafür, dass der *scout* einen eigenen Dooropener verwenden sollte und der ursprüngliche Dooropener sich auf den *cowdriver* konzentrieren sollte.

Gruppierung Auch wenn die Agenten das Gatter bereits passiert haben, so verbleibt bislang noch der Türöffner auf dem Schalter. Damit er wieder zur Gruppe aufschließen kann, ist es nötig, dass ein weiterer Agent den Schalter von innen betätigt und den Nachzügler zurück zur Gruppe schleust. Für die Koordination und das anschließende Zurückrufen des „Schleusers“ ist erneut der *dooropenerLeader* verantwortlich.

Abschluss Nachdem alle Agenten das Gatter passiert haben und die Gruppierung abgeschlossen ist, können die *dooropener* ihre Rolle ablegen und ihre ursprüngliche Rolle fortsetzen.

Bewertung

Die ursprünglich in der Strategie geplanten Aufgaben des *dooropeners* das Öffnen des Gatters und Suchen des Schalters werden durch die *dooropener*-Rolle realisiert. Nicht abgesehene Probleme, wie z.B. die Erkennung von Gattern wurden nicht früh genug erkannt und führten dazu, dass diese später sowohl in AgentSpeak als auch in DLV (von der BRF und dem HigherPlaner) realisiert wurden.

Unabhängig davon ist der gesamte Funktionsumfang des *dooropenerLeader* eine (notwendige) Erweiterung der ursprünglichen Strategie, deren Schnittmenge mit den Zuständigkeiten anderer Rollen leider stellenweise zu Problemen führte.

Kapitel 7

Implementierung

In diesem Kapitel stellen wir die Umsetzung des vorgestellten Multi-Agenten-Systems vor. Zur Implementierung nutzen wir Jason, einen java-basierten *AgentSpeak-Interpreter*. Dieser Interpreter wurde um einige Funktionalität, wie den *answer set solver* DLV und eine Anbindung an eine Multi-Agenten-Umgebung, erweitert. Diese Erweiterungen, nebst der Klassen zur Realisierung des Cowbot-BDI Modells, werden in Abschnitt 7.3 vorgestellt. Die Agentenspezifischen Implementierungen werden in Abschnitt 7.4 beschrieben.

7.1 Jason

7.1.1 Aufbau und Funktion des Interpreters

M.Kruse

Der Jason Interpreter führt seinen Agenten mit Hilfe von *reasoning cycles* [4] aus. Dabei durchläuft jeder Agent folgende Phasen:

- Wahrnehmung der Umgebung
- Aktualisierung der Beliefs
- Empfang von Nachrichten
- Auswahl von Nachrichten
- Auswahl eines Ereignisses
- Auswahl der relevanten Pläne für dieses Ereignisses
- Bestimmung von anwendbaren Plänen
- Auswahl eines anwendbaren Plans
- Auswahl einer Intention für die weitere Ausführung

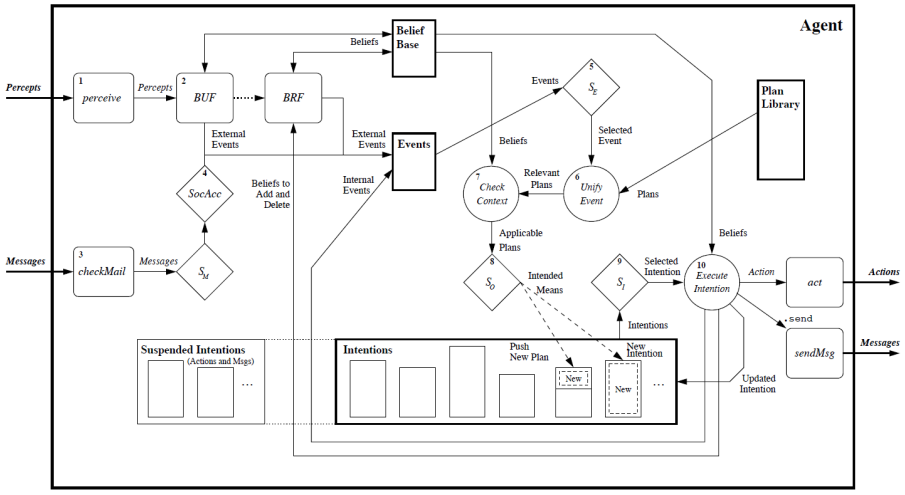

Abbildung 7.1: Jason-Interpreter

- Ausführung eines Schrittes der Intention

Zu Beginn eines jeden Zyklus werden die Events aktualisiert, die von der Umgebung durch die Beliefs und durch den Empfang von Nachrichten ausgelöst werden. Danach wählt die *select event* Funktion ein Event zur weiteren Bearbeitung aus. Diese Auswahl aus allen Events der aktuellen Event-Liste kann angepasst werden, damit das Verhalten des Agenten geändert werden kann (z.B. Priorisierung von Events).

Nach der Auswahl eines Events muss ein Plan (*Intention*) aus den anwendbaren Plänen ausgewählt werden. Ein Plan ist anwendbar, wenn das entsprechde Event ausgelöst wird und der Kontext des Plans erfüllt ist. Dies bewerkstelligt die *select operation*, die ebenfalls angepasst werden kann um eigenes Verhalten zu implementieren. Nachdem ein anwendbarer Plan ausgewählt wurde, wird dieser dem Intention-Stack hinzugefügt. Der Stack verwaltet alle zur Ausführung gebrachten Intentionen. Die Methode *select intetion* wählt wiederum aus allen Intentionen auf dem Intention-Stack eine Intention aus, damit von dieser die oberste Aktion abgearbeitet werden kann. Dies entspricht dann einem Schritt, den der Agent an seine Umwelt schickt.

Weitere Aufgaben des Interpreters

Die Belief-Updates der Umgebung/Umwelt werden durch den Interpreter gesetzt, damit die Agenten mit den aktuellen Umweltinformationen versorgt werden. Außerdem werden in diesem Fall neue Events in das Eventset aufgenommen. Da diese Prozesse nicht zu einem Agentenzyklus gehören, müssen diese Informationen von dem Interpreter gesetzt werden.

7.1.2 Komponentenmodell von Jason

T. Vengels

Nach der Vorstellung des Jason-Interpreters in Abschnitt 7.1.1 folgt eine kurze Übersicht über den Aufbau von Jason als Multi-Agenten-System. Der Überblick ist nötig, um ein Verständnis für die in nachfolgenden Abschnitten erläuterten Jason-Modifikationen zu erlangen. Vereinfacht lässt sich ein Multi-Agenten-System in Jason in drei Komponenten gliedern, zwischen denen jeweils aufeinander abgestimmte Schnittstellen definiert sind. Dies ermöglicht den Austausch einzelner Komponenten, ohne die anderen anpassen zu müssen (*bridge* Entwurfsmuster).

Infrastruktur Die Infrastruktur übernimmt die Verwaltung von Agenten, deren Kommunikation und Anbindung an eine Umgebung auf einer unteren Ebene, bildet also die Architektur eines MAS. Die Verwaltung der Agenten umfasst deren Lebenszyklus, konkret bedeutet dies das Erzeugen von Agenten-Objekten. Diese Softwareagenten führen ihren *reasoning cycle* jeweils in einem eigenen Thread aus. Der Infrastruktur überwacht hierbei die Ausführungs- bzw. Schlafphasen. Ein Agent, der keine weiteren Befehle mehr abarbeiten kann, versetzt seinen Thread in den Schlafmodus. Externe Ereignisse wie Kommunikation oder Umweltwahrnehmungen veranlassen die Infrastruktur, schlafende Agenten wieder aufzuwecken. Jason untertützt sowohl eine zentrale (auf einem Rechner) als auch verteilte (mittels JADE oder SACI) Infrastrukturen. Anpassungen auf dieser Ebene oder Eigenentwicklungen waren für die Projektgruppe nicht relevant.

Umgebung In Jason können eigene Umgebungen auf Basis der `environment`-Klasse erstellt werden. Im Rahmen der Projektgruppe konzentrierte man sich auf Umgebungen basierend auf MASSim. Eine angepasste Umgebung wurde lediglich als Monitor zur graphischen Wiedergabe der von MASSim erstellen SVG-Dateien genutzt. Eine Bearbeitung von Wahrnehmungen und Handlung findet in dieser Implementierung nicht statt.

Agenten Agenten werden in Jason durch zwei wichtige Klassen definiert, die zwecks Implementierung des vorgestellten Cowbot-BDI Konzeptes auch angepasst wurden:

- Die Klasse `Agent` im Paket `jason.asSemantics` setzt das von Jason definierte Agentenmodell um und beherbergt alle benötigten Datenstrukturen, um das Wissen und Pläne und ausführende Intentionen zu halten und durch den Interpreter abarbeiten zu lassen. Auf dieser Klasse basiert der Cowbot-BDI-Agent, die von der PG entwickelten Komponenten klinken sich hier in Methoden zur Wissensmanipulation und Planausführung ein (siehe 7.3.9).
- Die Klasse `AgArch` im Paket `jason.architecture` bildet die Anbindung zwischen Agenten und der Infrastruktur, auf Seite des Agenten. Für die Umsetzung der Implementierung erfolgt hier die Anbindung

an MASSim. Jeder Cowbot-Agent verfügt über seine eigene Verbindung. Das gewählte Softwaredesign erlaubt prinzipiell die Anpassung an weitere Umgebungen (siehe 7.3.8).

7.2 MASSim

T. Vengels

Bei MASSim (kurz für *multi-agent systems simulation* handelt sich um einen Simulator für Multi-Agenten Umgebungen. Hierbei stellt MASSim ein Software-Framework zur Verfügung, um Agenten mit Wahrnehmungen über eine simulierte Umgebung zu versorgen und Handlungen an diese Umgebung entgegenzunehmen und auszuwerten. MASSim wurde von der Technischen Universtität Claustahl im Rahmen des seit 2005 jährlich stattfindenden *multi-agent programming contest* (kurz: MAPC) [19] entwickelt. Im Rahmen der Projektgruppe verwendeten wir die Simulatoren der Jahre 2009 bzw. 2010.

7.2.1 Umwelt

Im Rahmen der PG betrachteten wir die Kuhfang-Szenarien der Jahre 2009 und 2010, die nahezu identisch sind (abgesehen von Variablen wie Rundenzeit, Teamgrösse, etc) und Ausgangspunkt der entworfenen Agentenrollen (siehe 3.5) und Strategien (siehe 6.1.1) darstellen. Die Umwelt wird als zweidimensionales Kachelfeld dargestellt, Agenten werden rundenbasiert in festen Zeitabständen mit Wahrnehmungen versorgt. Pro Runde (dies ist in der simulierten Umgebung gleich der Welttakt) kann ein Agent genau eine Aktion ausführen. Die Simulation lässt jeweils zwei Agententeams gegeneinander antreten, Gewinner ist das Team, welches im Durchschnitt über alle Züge der Simulation die meisten Kühe im eigenen Pferch halten konnte. Simuliert wird eine Umgebung mit folgenden Eigenschaften:

Spielfeld Die Umgebung wird als endliches, zweidimensionales Kachelfeld simuliert. Jede Kachel (Zelle) wird durch ihre Belegung beschrieben (zum Beispiel frei, freundlicher Agent, feindlicher Agent, etc.). Für bestimmte Zellen wird zudem ihr Status übertragen. So gilt für Zaunzellen, dass für diese zusätzlich die Information über ihren Zustand (offen oder geschlossen) gesendet wird. Der Status eines Zaunes wechselt auf offen, falls ein Agent neben einem Schalter steht, der an den Zaun angrenzt. Zäune sind zudem immer orthogonal ausgerichtet. Für Pferchzellen wird ebenfalls übermittelt, ob es der eigene oder feindliche Pferch ist. Zu beachten ist, dass über Zellen auf denen Kühe oder Agenten stehen nicht übermittelt wird, ob es sich um Pferch oder Zaunzellen handelt. Insofern muss ein Agent unterscheiden können zwischen Entitäten, welche sich über die Kachelwelt bewegen, und Objekten die bauliche Bestandteile der Welt sind. Die einzelnen Elemente des Spielfeldes kurz zusammengefasst:

- *obstacle* Ein sich nicht veränderndes, permanentes Hindernis der Welt. Auf diese Felder können keine Agenten oder Kühe mittels einer Bewegung ziehen.

- *switch* Schalter sind wie *obstacle* unbegehbar. Allerdings können Agenten, welche orthogonal neben einen Schalter stehen, angrenzende *fence*-Zellen öffnen (dieser Zustand bleibt so lange erhalten wie Agenten neben einem Schalter stehen).

- *fence* Hindernisse, die Agenten und Kühe nicht durchlassen, solange sie geschlossen sind. Schalter können Zäune öffnen und somit für Agenten und Kühe passierbar machen. Schliesst ein Zaun, während sich Agenten oder Kühe auf diesen Zellen befinden, werden sie zufällig an eine freie Zelle teleportiert.

- *empty* Eine Zelle, die frei ist. Agenten und Kühe können auf freie Zellen laufen.

- *corral* Eine Pferchzelle, die prinzipiell für Agenten und Kühe begehbar ist (im Sinne der Spiellogik sich wie *empty* verhält). Die Position der eigenen Zellen wird den Agenten beim Betreten einer Simulation mitgeteilt. Während der Simulation erhalten Agenten Informationen, ob es sich um eigene oder feindliche *corral*-Zellen handelt.

- *agent* Agent, bei denen zusätzlich die Information übermittelt wird, ob es sich um Freunde oder Kontrahenten handelt.

- *cow* Eine Kuh, zu der zusätzlich eine während der Simulation gleichbleibende, eindeutige Identifikationsnummer übermittelt wird.

Aktionen Jeder Agent kann entweder stehenbleiben (die Aktion *skip*), oder in der acht angrenzenden Zellen gehen (die Richtung erfolgt durch Angabe einer Himmelsrichtung, zum Beispiel *north*, *northwest*, etc.). Pro Runde wird höchstens eine Aktion pro Agent gewertet, im Falle keiner Aktion wird dies als *skip* interpretiert. Ein Agent darf eine Zelle wechseln, falls sie nicht von einem Hinderniss (festes Hinderniss, Schalter, Kuh oder Agent) oder einem geschlossenen Zaun blockiert wird. Eine besondere Einschränkung gilt Zellen orthogonal adjazent zu einem Schalter: von diesen Zellen aus darf ein Agent keine diagonale Bewegung ausführen (dies ist mit der Implementierung der Spielregeln im Simulator zu erklären: ein Agent könnte sonst leicht selbt einen Zaun öffnen, dann diagonal auf die nun offene Zaunzelle angrenzend am Schaltern wandern. Dies würde eine Koordinaten der Agenten beim Überqueren von Zäunen überflüssig machen). Das Treiben von Kühen realisieren Agenten indirekt, Kühe laufen vor Agenten weg. Das Treiben ist also durch Positionierung der Agenten gegenüber den Kühen umzusetzen. Zusammengefasst die erlaubten Aktionen mit der Umwelt:

- *north*, *northeast*, *east*, *southeast*, *south*, *southwest*, *west*, *northwest* - dies sind die prinzipiell erlaubten Aktionen, die ein Agent in der Umgebung wirken kann.

- *skip* - eine Aktion, bei der ein Agent nicht seine Position ändert. Kann zufällig vom Simulator als Alternative für jede andere Agentenaktion gewählt werden.

Umwelteigenschaften Nach der kurzen Beschreibung der simulierten Umgebung folgen noch Umwelteigentschaften. Hierzu wird eine Einordnung nach Russell and Norvig, aus [25] Kapitel 1:

accessibility Die simulierte Umgebung ist nicht vollständig wahrnehmbar für einen Agenten. Stattdessen wird lediglich ein Ausschnitt der Welt pro Welttakt an den Agenten übermittelt.

determinism Die simulierte Umgebung ist nicht deterministisch. Aktionen von Agenten können von der Simulation zufällig unterbunden werden (eine Aktion wird dann mit *skip* überschrieben). Deterministisch ist lediglich das Zeitintervall, dies ist in einer Simulation immer konstant (beim Wettbewerb: zwei Sekunden). Die Reihenfolge, in der Agenten oder Kühe ihre Bewegungen ausführen, wird eines Rundenüberganges zufällig ermittelt. Dies hat den Effekt, das aneinanderstehende Akteure sich gegenseitig blockieren können.

episodicity Die simulierte Umgebung ist nicht episodisch. Zum einen ist die Simulation auf endlich viele Runden (in der Regeln 1000 bis 1500) begrenzt. Zudem wird per Zufallsgenerator entschieden, ob Aktionen von Agenten erfolgreich ausgeführt werden. Haben Kühe mehrere gleichwertige Optionen, wohin sie laufen wollen, wird dies ebenfalls per Zufallsgenerator ermittelt. Somit laufen auch wiederholt gespielte Karten niemals gleich ab.

dynamisn Die simulierte Umwelt ist dynamisch. Aus Sicht eines Agenten sind insbesonders andere Agenten und Kühe Prozesse, welche die Welt verändern.

Abschliessend sei erwähnt, dass einige Parameter pro Szenario variieren können. Die Sichtweite der Agenten wird in der Szenariobeschreibung festgelegt. Entspricht diese der Breite oder Höhe des Spielfeldes, gilt die obige Bewertung der *accessibility* nicht mehr. Genauso kann die Rundenzeit auf mehr als zwei Sekunden gesetzt sind. Während der Entwicklung waren dies praktische Möglichkeiten, den Agenten schrittweise Fähigkeiten wie Erkundung des Terrains oder Planungsstrategien zum Kuhtreiben zu verleihen, um letztendlich wettkampftaugliche Agenten zu erhalten.

7.2.2 Verbindungsprotokoll

Die Kommunikation mit dem MASSim selbst geschieht über ein XML-Protokoll. Zunächst müssen sich interessierte Agenten bei dem Simulator registrieren. Bei Erfolg bekommt jeder Agent einmalig Informationen über die aktuell simulierte Runde zugespielt, anschliessend in festen Zeitabständen Wahrnehmungen über sein aktuelles Umwelt. Eine Beschreibung des Protokolls ist in [3] zu finden, die Protokollimplementierung und Übersetzung in die gewählte Wissensdarstellung erfolgt durch die Klasse MassimAdapter (siehe 7.3.8). In Abbildung 7.2 verdeutlicht die Protokollzustände, die ein Agent (bzw. dessen technische Infrastruktur) nach erfolgreichem Anmelden der Simulation verarbeiten muss.

- `SIM-START` Kündigt den Start einer Simulation an. Übertragene Daten sind die Grösse des Spielfeldes, als auch die Position der eigenen Pferchzellen.
- `ACTION-REQUEST` Kündigt den Beginn eines Spielzuges an. Der Server übermittelt Position und wahrgenommene Umwelt eines Agenten. Der Server erwartet, dass innerhalb des Rundentaktes mittels `ACTION` ein Agent eine auszuführende Aktion übermittelt.
- `SIM-END` Kündigt das Ende der Simulation an, übermittelte Daten ist der Ausgang eines Wettkampfes.

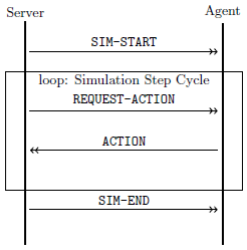

Abbildung 7.2: Verhaltensdiagramm zum Protokoll zwischen MASSim und einem Agenten. Aus dem MAPC-Dokument über das Verbindungsprotokoll [3].

7.2.3 SVG Ausgabe

Der Simulator legt pro Runde eine Visualisierung der gesamten Umwelt an. Dies erfolgt in Form von SVG-Dateien (*scalable vector graphics*), einem XML-basierten Standard zur Beschreibung von Vektorgrafiken. Diese können in gängigen Webbrowsern oder Betriebssystemen angezeigt werden. Darüber hinaus wurde eine angepasste Jason-Umgebung (aufbauend auf die `environment`-Klasse) implementiert (siehe 7.3.10). Diese überwacht während einer laufenden Simulation die Dateiausgabe von MASSim stellt das komplette Spielfeld graphisch dar.

7.2.4 Spieleclient

Zum Kennenlernen und Erfahrungen sammeln wurde ein Spieleclient entwickelt. Dieser erlaubt es, einen Agenten in einer Massim-Simulation von einem Menschen steuern zu lassen. Der Spielclient wurde bereits im ersten PG-Semester fertiggestellt, dies ermöglichte der Projektgruppe noch während der Konzeptionsphase wertvolle Strategien und Erfahrungen für die Entwicklung eines kooperativen Multi-Agenten-Systems zu sammeln, entwerfen und zu testen.

Implementierung

Der Spieleclient basiert im Kern auf einem Adapter, der eine Verbindung zu einer Massim-Umgebung erlaubt, und einer graphischen Anzeige zur Visualisierung des für den Agenten aktuell sichtbaren Teils der Umwelt. Die Verbindung zum Massim-Server wird von den Klassen AbstractAgent und ClientAgent geregelt. AbstractAgent ist eine abstrakte Basisklasse aus dem frei verfügbaren MASSim Package und stellt grundlegende Verbindungsfunktionalität bereit, ClientAgent implementiert darauf aufbauend Methoden zur Auswertung der empfangen XML Daten passend zu den Kuhfang-Szenarien. Diese Daten werden auf Instanzen der Klassen ClientWorld und WorldObject verarbeitet, ClientWorld modelliert hierbei ein einfaches zweidimensionales Kachelfeld, WorldObject modelliert Entitäten wie Hindernisse, Kühe oder Agenten. Die Anzeige selber erfolgt über ein JPanel, gekapselt in der Klasse masclient. Die Klasse DirectionalControl verwirklicht einen Cursor, der sich die letzte Kachel merkt, auf der ein menschlicher Spieler seinen Agenten stellen will. ThreadMessage ist eine nebenläufige Queue, die für Nachrichtenaustausch zwischen dem Netzwerkthread, der Anwendung selber und der Anzeige genutzt wird. Abbildung 7.3 zeigt die Klassenstruktur, Abbildung 7.4 eine Beispielszene aus einer Karte.

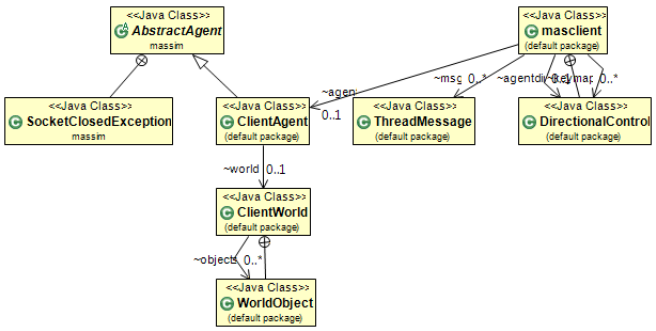

Abbildung 7.3: Strukturdiagramm des Spieleclients

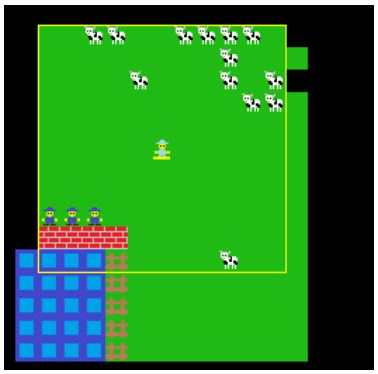

Abbildung 7.4: Beispielszene Spieleclient, Agenten sind die blauen Figuren, der eigene Agent ist hellblau hervorgehoben. Das gelbe Quadrat stellt das aktuelle Sichtfeld dar. Schwarze Bereiche sind unerkundet, grüne Bereiche freie Felder, Hindernisse sind rote Mauern.

7.2.5 MapTool

Das *map tool* ist ein kommandozeilen-basiertes Hilfsprogramm zur Erstellung von MASSim Karten. Das Programm transformiert hierbei in Klartext vorliegende Kartenbeschreibungen in MASSim 2010 kompatible XML-basierte Kartendateien. Die erzeugten Kartendateien beschreiben jeweils ein Spielfeld, die Erzeugung von Szenarien mit mehreren Karten (respektive Wettkampfrumden) wird nicht unterstützt. Die Entwicklung von MapTool erfolgte aus der Tatsache, dass der Karteneditor im MASSim-Entwicklungspaket keine hohe Benutzerfreundlichkeit aufweist und ein Erstellen von Karten damit als nicht sinnvoll eingestuft wurde (es ist reine Glückssache ob Karten geladen oder gespeichert werden. In der Regel stürzte der Editor ohne nachvollziehbare Fehlermeldung einfach ab).

7.3 Architektur

7.3.1 Überblick

T. Vengels

Die folgenden Abschnitte beschreiben die Implementierung des Agentenkonzepts. Die externen genutzten Programme und das Agentenframework wurden bereits in Abschnitt 7.1 und 7.2 vorgestellt. Die folgenden Abschnitte beschreiben das Design einzelner Erweiterungen und das Zusammenspiel des implementierten Cowbot-BDI Modells, sowie Anpassungen an Jason, um dieses als Grundlage für die Realisierung des Konzeptes zu nehmen.

- *Umweltanbindung* Hier stand die Anbindung an MASSim7.2 im Vordergrund. Andererseits sollte MASSim nicht die einzige Umwelt sein, in der

Agenten agieren können. Dementsprechend wurde zunächst ein grobes Strukturmodell entworfen, sodass die Umweltanbindung an weitere Bedürfnisse anpassbar ist. Die Umsetzung dieser Idee wird in Abschnitt 7.3.8 genauer beschrieben.

- *BDI Modell* Hier stand die softwaretechnische Realisierung durch Schnittstellen und Klassen der einzelnen konzeptuellen Komponenten *BRF*, *DesGen*, *Delib*, *Planner* und *ActSel* im Vordergrund. Notwendig hierfür war ebenfalls der Entwurf eine Logikbibliothek, welche die in Abschnitt 2.3.3 vorgestellten logischen Programme und Einbindung von DLV in Java ermöglicht.

Die nachfolgenden Abschnitte erläutern sowohl die Implementierung der einzelnen BDI-Komponenten, als auch deren Integration in Jason.

7.3.2 Logik Bibliothek

T. Vengels

Um erweiterte logische Programme in Jason zu nutzen, war die Anbindung eines *answer set solvers* notwendig. Desweiteren wurden Klassen, die die Erzeugung von ELPs gezielt unterstützen, entworfen. Hier greifen wir aus dem einfachen Grunde nicht auf die in Jason vorhande Logikklassen zurück, da diese mächtigere Konstrukte (Verschachtelte Terme, Listen, Disjunktionen in einem Regelkörper) erlauben als zum Beispiel DLV versteht, andererseits stehen in Jason Konstrukte wie Constraints nicht zur Verfügung. Hierzu entwarfen wir eine Klassenstruktur, die ELPs wie in Abschnitt 2.3.3 realisiert und *answer set solver* als externe Programme über die Java-Methode Runtime.getRuntime().exec als Prozess ausführen kann. Um in Textdateien codierte logische Programme mit den Klassen der Logik-Bibliothek nutzen zu können, wurde ein Parser mit Hilfe des Werkzeuges *javacc* entwickelt.

Paket- und Klassenstruktur

Die Logikbibliothek wurde in den Paketen `edu.udo.cs.ie.cowbots.logic` und `edu.udo.cs.ie.cowbots.logic.solver` implementiert. Im ersten Paket befinden sich Klassen, um logische Programme mit Java zusammenzubauen, im zweiten Paket befindet sich die Anbindung an DLV. Die Strukturdiagramme sind in Abbildung 7.5 und 7.6 abgebildet.

Klassen und Interfaces

Die Logikbibliothek definiert eine Reihe von Schnittstellen, hiermit ist die grundlegende Struktur von logischen Programmen festgelegt. In Fällen, wo es zu Namenskonfikten mit in Jason definierten Klassen kommen konnte, wurde vor dem Klassennamen ein „ELP“-Präfix gesetzt. Anderfalls hätte man in Methoden, die Objekte der Jason-Logik und ELP-Logik benutzen, den vollständigen Paketnamen vor jede Variablendeklaration setzen müssen. Insofern ist das Präfix ein

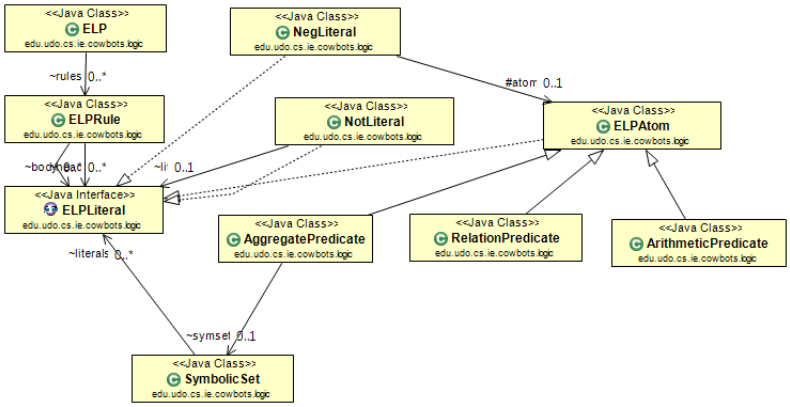

Abbildung 7.5: Strukturdiagramm des edu.udo.cs.ie.cowbots.logic Paketes

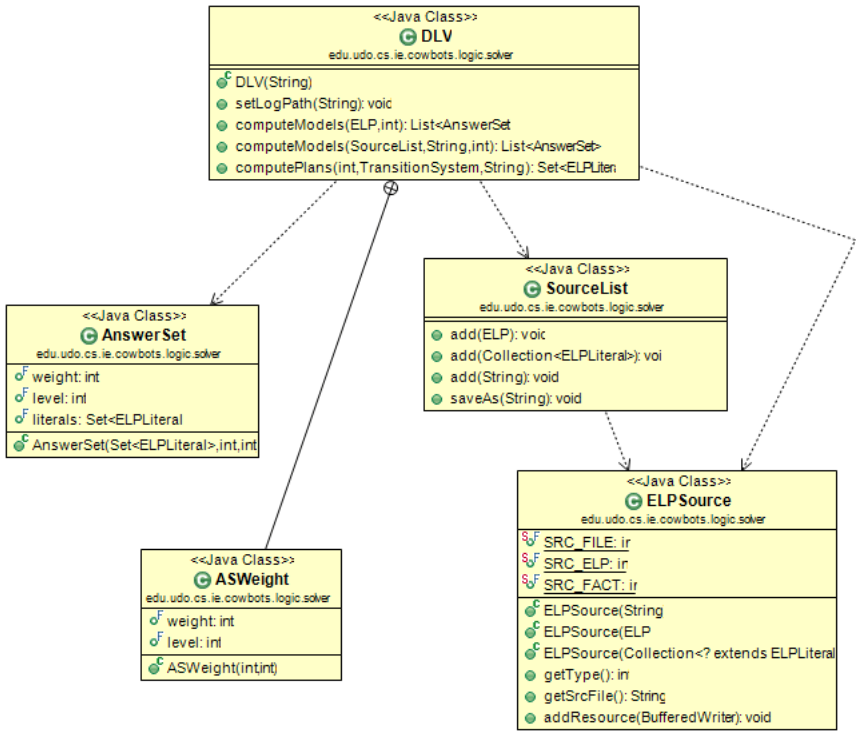

Abbildung 7.6: Strukturdiagramm des edu.udo.cs.ie.cowbots.logic.solver Paketes

vertretbarer Kompromiss, die Quelltexte leserlich zu halten. Es sind folgende Klassen und Interfaces definiert:

ELPLiteral Schnittstelle, die ein Literal als grundlegenden Baustein für die Konstruktion von Regeln in logischen Programmen definiert. Diese Schnittstelle bietet Zugriffsmöglichkeiten auf an das Literal gebundene Atom, als auch optional gebundene Negationsoperatoren. Die Schnittstelle weisst drei Implementierungen auf:

- NegLiteral - Eine Klasse, die ein strikt negiertes Atom (der Klasse ELPAtom) kapselt und somit den unären Operator - darstellt, also Ausdrücke der Form $\neg L$ über ein Atom L darstellt.
- NotLiteral - Eine Klasse, die ein *default* negiertes Literal darstellt, also Ausdrücke der Form *not* L über ein Literal L.
- ELPAtom - Vereinigt die Eigenschaften von Atomen und positiven Literalen, siehe nächster Abschnitt.

ELPAtom Ein ELPAtom repräsentiert die grundlegenden Bausteine für Regeln in erweiterten logischen Programmen. Es implementiert das Interface `ELPLiteral`. Darüber hinaus stellt es Manipulatoren für Prädikatsymbol lesen sowie Terme setzen und auslesen. Dies erlaubt es Instanzen dieser Klasse direkt als positive Literale in Regeln zu verwenden, als auch direkt auf das Prädikatsymbol und gebundene Terme zuzugreifen. Beispiel:

```
ELPAtom a = new ELPAtom("a", "1");
```

erstellt ein einstelliges Prädikat $a(1)$, dieses kann im Kontext der Programmkonstruktion ohne Weiteres in eine Regel eingesetzt werden. Zusätzlich unterstützt die Klasse spezielle, in DLV eingebaute Prädikate (*build-in predicates*). Dies sind in der Regel arithmetische oder vergleichende Operationen, oder Aggregatfunktionen. Ausdrücke dieser Form können aus der `ELPAtom`-Klasse heraus direkt instanziiert werden. Folgendes Beispiel

```
ELPAtom comp = ELPAtom.Gtr("X", "Y");
```

instanziiert ein `ELPAtom`, welches den Ausdruck $X > Y$ darstellt. Die `ELPAtom`-Klasse fungiert in diesem Fall als Fabrik für Instanzen von Klassen, welche speziell an eingebaute Prädikate angepasst sind. Dies sind im Einzelnen:

- `ArithmeticPredicate` - diese Klasse repräsentiert Ausdrücke, die Addition und Multiplikation verwenden. DLV akzeptiert logische Programme, in denen das Ergebnis einer Addition (Multiplikation) einer Variablen zugewiesen wird.
- `RelationPredicate`, dies sind zweistellige Prädikate, die zwei Argumente bezüglich ihrer Ordnung über die gängigen Operatoren $<, \leq, =, \neq, \geq, >$ vergleichen. DLV erlaubt sowohl infix- als auch präfix-Notation, die implementierte Logikbibliothek unterstützt lediglich Ausgabe der infix-Notation.

- `AggregatePredicate`, dies sind eingebaute Prädikate, die über ein *symbolic set* eine Aussage machen. Ein *symbolic set* ist ein Ausdruck der Form $\{V_i : L_i\}$, die lokalen Variablen V_i werden an eine Konjunktion von Literalen L_i gebunden. Gängige Aggretfunktionen sind $\#min$, $\#max$, $\#sum$, $\#times$, $\#count$.

ELPRule Instanzen der Klasse `ELPRule` repräsentieren Regeln in logischen Programmen. Jede `ELPRule` kann über einen Kontainer mehrere Instanzen der Klasse `ELPAtom` referenzieren. Es wird zwischen Kopf- und Körperliteralen unterschieden, und entsprechende Zugriffsmöglichkeiten auf diese Elemente, respektive Kontainer, bereitgestellt. Regeln werde sukzessive durch Hinzufügen von Kopf- oder Körperliteralen erzeugt. Folgendes Beispiel

```
ERLPRule r = new ELPRule();
r.addHead( new ELPAtom("a"));
r.addBody( new ELPAtom("b"));
r.addBody( new NotLiteral( new ELPAtom("c"));
```

instanziiert die Regel $a \leftarrow b, not\, c$.

ELP Die Klasse `ELP` stellt ein erweitertes logisches Programm dar. Realisiert wird dies über einen Kontainer für Instanzen der Klasse `ELPRule`. Die Antwortmengen zu diesen Klassen kann über die `DLV`-Klasse gemacht werden. ELPs werden durch sukzessives Hinzufügen von Regeln erstellt. So erzeugt

```
ELP program = new ELP();
program.addRule(r);
```

ein erweitertes logisches Programm mit vormals erstellter `ELPRule` r. Mittels der Hilfsklasse `Alphabet` ist es zudem eine Möglichkeit gegeben, die in einem Programm benutzen Prädikatsymbole in Köpfen- und Regeln wiederzugeben.

DLV Die `DLV`-Klasse ermöglicht den Aufruf von DLV [22], einem *answer set solver*, aus Java heraus. Hierzu wird das Kommandozeilenprogramm DLV als neuer Prozess mittels *Runtime.getRuntime().exec* ausgeführt. Als Eingabe kann entweder ein ELP, oder eine Liste von ELP-Quellen, und Kommandozeilen-Optionen übergeben werden. Das Ergebnis, die Antwortmengen, sind eine Menge von `ELPLiteral`-Instanzen und können dann in Java ausgewertet werden. Die DLV-Klasse erwartet im Konstruktor eine absolute Pfadangabe zur ausführbaren DLV-Programmdatei. Die Berechnung einer Antwortmenge eines Programms zeigt folgendes Beispiel:

```
DLV dlv = new DLV("path/to/dlv.exe");
List<AnswerSet> = dlv.computeModels(elp, 1);
```

.Darüber hinaus erlaubt die `DLV`-Klasse die Berechnung von Antwortmengen eines Programms, das über mehrere Quellen verteilt ist (zum Beispiel zusätzliches

Hintergrundwissen, zusätzliche Inferenzregeln). In dem Fall können die Quellen über die Hilfsklasse *SourceList* effizient an DLV übermittelt werden, anstatt diese Quellen zunächst von Hand zu einem Gesamtprogramm zu verschmelzen. Das Beispiel:

```
ELP src1;
String src4 = "src2.lp";
SourceList sl = new SourceList();
sl.add(src1);
sl.add(src2);
dlv.computeModels(sl,"-nofacts",2);
```

berechnet maximal zwei Antwortmengen der fiktiven Quellen *src1* und *src2*, ohne die initialen Fakten (DLV Kommandozeilenoption *„-nofacts“*) in die Antwortmengen zu übernehmen.

Parser und akzeptierte logische Sprache Als Ergänzung der Logik-Bibliothek wurde ein Parser mit Hilfe des Werkzeuges javacc [12] implementiert. Die zu Grunde liegende Grammatik spiegelt sowohl den Aufbau von erweiterten logischen Programmen, die mit der Logik-Bibliothek erzeugt werden können wieder. Zu Beachten ist, dass nicht jedes vom Parser akzeptierte Programm auch von DLV akzeptiert wird, da Symbole (Prädikat-, Konstanten-, Variablennamen und Zahlen) lediglich als Strings interpretiert werden. Folgende BNF verdeutlicht die akzeptierte Sprache der Logik-Bibliothek und des Parsers:

$$
\begin{aligned}
ELP &\rightarrow Rule* \\
Rule &\rightarrow (LiteralList\ '.') \mid \\
&\quad\ (LiteralList)?\ ':\text{-}'\ LiteralList \\
LiteralList &\rightarrow Literal(',' Literal)* \\
Literal &\rightarrow \text{'not'}?\ '\text{-}'?\ Atom \\
Atom &\rightarrow Symbol\ ('(' Symbol(',', Symbol) * ')')? \mid \\
&\quad\ (Symbol\ Op)?\ Aggregate\ Op\ Symbol \mid \\
&\quad\ Symbol\ Op\ Aggregate\ (Op\ Symbol)? \\
Aggregate &\rightarrow '\#'\ AgSymbol\ SymbolicSet \\
SymbolicSet &\rightarrow '\{'\ Symbol\ (',' Symbol)*\ :\ LiteralList '\}' \\
AgSymbol &\rightarrow \text{min|max|sum|times|count} \\
Symbol &\rightarrow (a \ldots z | A \ldots Z | 0 \ldots 9 | \_)+ \\
Op &\rightarrow < \mid \leq \mid = \mid \neq \mid \geq \mid >
\end{aligned}
$$

Die Umsetzung des Parsers erfolgte mit dem Parser-Generator *javacc* [12], welcher die Erzeugung von Java-Code aus Script-Dateien, in denen Grammatik und benötigter Programmcode zur Umstetzung von Parsern aufgestellt werden kann.

7.3.3 Wissensrevision

T. Vengels

In diesem Abschnitt wird die Implementierung der in Kapitel 2.3 vorgestellten Wissensrevision beschrieben. Zunächst wird die Paket- und Klassenstruktur beschrieben, darauf aufbauend wird der interne Ablauf anhand von Verhaltensdiagrammen schematisch dargestellt.

Paket- und Klassenstruktur

In den Paketen
`edu.udo.cs.ie.cowbots.bdi.brf` und
`edu.udo.cs.ie.cowbots.bdi.brf.operator`
befinden sich die Klassen und Schnittstellen der Wissenskomponente, hier erfolgt die Umsetzung der Revisionsfunktion und des epistemischen Zustands aus dem Cowbot-BDI Modell. Abbildung 7.7 zeigt die Klassenstruktur, im folgenden werden die Klassen und Schnittstellen beschrieben. Für eine genaue Dokumentation der Schnittstellen sei die JavaDoc-Dokumentation zu betrachten.

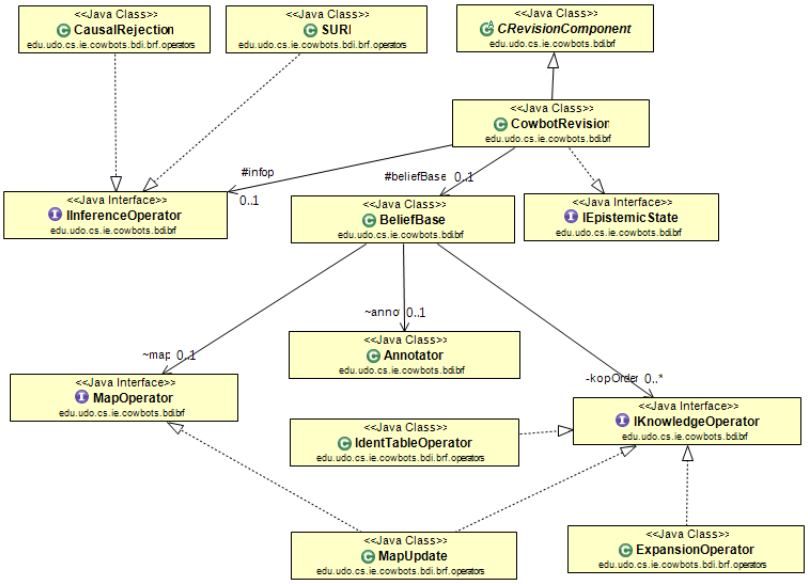

Abbildung 7.7: Strukturdiagramm des edu.udo.cs.ie.cowbots.bdi.brf Paketes

CRevisionComponent abstrakte Basisklasse, welche grundlegende Schnittstellen zur Verbindung mit der Jason-Agentenklasse bereitstellt. Die Aufgabe der Klasse ist zum einen die Initialisierung des Revisionsmoduls, als auch Wisssensänderungen in Jason (in der Jason-Agentenklasse) abzufangen und zu verarbeiten.

IEpistemicState Schnittstelle, über die sowohl andere Cowbot-BDI Komponenten, als auch interne Aktionen, Zugriff auf das Wissen eines Agenten erlangen. Die Klasse ermöglicht den Zugriff auf das logische *belief set* eines Agenten.

CowbotRevision die eigentliche Umsetzung des in Abschnitt 2.3 vorgestellten Revisionskonzept. Die Klasse erbt (respektive implementiert) die zuvor beschriebenen Schnittstellen, kapselt wichtige Komponenten wie die die logische `BeliefBase`- und `BeliefSet`-Klasse und verwendet Wissensoperatoren zur Generierung des *belief state* und dem daraus resultierenden Wissen eines Agenten. Der allgemeine Ablauf der Revisionsmethode ist hierbei immer gleich:

- Laden aller `InformationObject`-Instanzen aus der Wissenbasis. Diese Liste wird vorsortiert geliefert.
- Erstellen des *belief states* mit Hilfe von Methoden des später vorzustellenden Interfaces `IInferenceOperator`.
- Generierung der Antwortmengen mit Hilfe von DLV.
- Filterung der Antwortmenge, dies erzeugt ein neues *belief set*
- Differenzbildung zum vorherigen *belief set*. Die Revisionsklasse ermittelt somit Listen über neu hinzugekommene und weggefallene Literale. Auf Grund dieser Listen wird die Jason-interne *belief base*, das Gegenstück des logischen *belief sets* aktualisiert. Somit ist gewährleistet, dass auch in ASL-Plänen das revidierte Wissen des Agenten verfügbar ist.

Die Revisionsklasse selber wird mittels einer XML-Konfigurationsdatei 8.3 eingerichtet. Dort ist es möglich, Klassen für Wissensänderungs- und Inferenzoperatoren anzugeben, welche mittels dem Java *class loader* während der Initialisierung eines Agenten geladen werden.

BeliefBase eine Klasse, welche eine Wissensbasis für logische Programme modelliert. Logische Programme werden durch die in Abschnitt 7.3.2 vorgestellte ELP-Klasse gespeichert in Instanzen der Klasse InformationObject gespeichert, welche in Java-Kontainern des Types `List<InformationObject>` gehalten werden. Das Einpflegen von `InformationObject`-Instanzen erfolgt durch Wissensänderungsoperatoren, welche zunächst bei der BeliefBase registriert werden und danach lediglich auf ihnen zugewiesenen Listen arbeiten können.

BeliefSet eine Kontainer-Klasse, welche das Wissen des Agenten in Form von ELPLiteral-Instanzen hält. Neben üblichen Kontainern-Operationen wie Aufzählung von Elementen beherrscht die Klasse auch die Extraktion von ELPLiteral-Teilmengen basierend auf Prädikatsymbolen, sodass andere Komponenten gezielt Teilwissen abfragen können. Neben der Iteration aller Elemente beherrscht diese Klasse zwei oft genutzte Methoden:

findFirst Die Methode *findFirst* liefert zu einem gegebenen Prädikatsymbol das erste auffindbare Literal im BeliefSet, oder null.

finAll analog zu *findFirst* liefert diese Methode eine (möglicherweise leere) Liste aller Literale, die Instanzen eines Prädikatsymbols sind.

IKnowledgeOperator Die Modifikation der *belief base* erfolgt durch Klassen, welche diese Schnittstelle implementieren. Hierzu ist folgende Methode definiert:

- *update* Die `BeliefBase` übergibt einem Operator eine Liste von `InformationObject`-Instanzen, welche von einem Operator in eine `BeliefBase` eingepflegt werden. Das Verhalten dieser Operation hängt von der konkreten Implementierung eines Operators ab.

Darüber hinaus definiert die Schnittstelle einige optionale implementierbare Methoden, welche von der Revisionsfunktion genutzt werden können.

- *processBeliefSet* Erlaubt Komponenten, sich über die Schnittstelle direkt in die Revisionsfunktion einzuklinken. Nach Aktualisierung des *belief sets* wird allen Operatoren die Möglichkeit gegeben, ihren Zustand auf Grundlage des revidierten Wissens zu ändern. Implementieren andere Komponenten wie zum Beispiel *path finding*-Routinen lediglich diese Methode, können sie jederzeit ohne zusätzlichen Pflegeaufwand an anderen Programmstellen auf dem aktuellen Wissen des Agenten arbeiten.
- *deleteFacts* wird von der Revisionsfunktion aufgerufen, Grundlage sind hierfür zurückgewiesene Fakten (respektive Regelköpfe). Geeignet, um die Ausdehnung der Wissensbasis eines Agenten aufzuräumen und in Laufzeitkritischen Umgebungen anwachsende *belief state* Programme zu vermeiden.

Im Zuge der Implementierung wurden folgende Operatoren umgesetzt, die allesamt ein unterschiedliches Verhalten aufweisen.

- *ExpansionOperator* - fügt neues Wissens durch Expansion (Erweitern der Liste). Eine Instanz dieser Klasse wird standardmässig von der BeliefBase angelegt, womit automatisch die einfachste, aber auch konzeptionell treueste Wissensbasisänderung erfolgt. Dieser Operator implementiert lediglich die `update`-Methode der Schnittstelle.
- *MapOperator* - eine Schnittstellenerweiterung, welche Wissen über die Agentenumwelt direkt auf Basis einer räumlichen Datenstruktur modifiziert. Ein MapOperator überschreibt somit redundante Informationen über Kachelzellen direkt auf Ebene der Wissensbasis. Bezüglich der *belief state construction* ergeben sich kleinere Update-Sequenzen, welche sich positiv auf die Laufzeit der Antwortmengen-Berechnung auswirken können.
- *IdentTable* - verwaltet eine *lookup*-Tabelle über die in der Wissensrepräsentation (5.1) vorgestellten Identifikatoren. Dies erleichtert in Java die Evaluation von *ison*-Prädikaten (wird zum Beispiel von graphischen Oberfläche zur Darstellung oder von Pfadfindungs-Algorithmen benutzt.).

IInferenceOperator Die Konstruktion von konsistenten logischen Programmen, und anschliessende Wahl einer Antwortmenge, erfolgt in Klassen der Schnittstelle `IInferenceOperator`. Die Schnittstelle definiert drei Methoden:

- *createBeliefState* erzeugt aus einer Liste von `InformationObject`-Instanzen ein konsistentes erweitertes logisches Programm.
- *setSavePredicates* ermöglicht die Angabe einer Menge von Prädikatsymbolen, welche von eventuellen Regeltransformationen ausgeschlossen werden. In Fällen, wo Literale als ohnehin konfliktfrei gelten, führt dies zu kleineren logischen Programmen. Die Liste der als immer konfliktfrei anzusehenden Symbole muss bei der Initialisierung der Revisionsklasse gesetzt werden.
- *selectAnswerSet* wählt aus einer Liste von Antwortmengen eine Antwortmenge aus, die von der Revisionskomponente als *belief set* übernommen wird. In dieser Methode erfolgt ebenfalls die Bereinigung einer Antwortmenge von zusätzlichen Literalen aus der *belief state construction*. Die genaue Arbeitsweise der Methode hängt stehts implementierten Wissensoperator ab.

Die Schnittstelle wird von zwei Klassen implementiert, welche die aus Abschnitt 2.3 vorgestellten Methoden zur *belief state construction* umsetzen.

Hilfsklassen Im Revisionspaket finden sich noch einige Hilfsklassen, welche zur korrekten Funktionsweise der Revisionskomponente notwendig sind:

- `InformationObject` Eine Klasse, welche ein Tupel aus Metainformationen und logischen Programmen modelliert.
- `Meta` Meta-Informationen, in unserer Implementierung wird neben der Zeit zusätzlich ein Zähler gespeichert. Der Zähler wird pro eingefügten *information object* in die *belief base* erhöht und erlaubt einen Vergleich zwischen *information objects* gleichen Zeitstempels.
- `Annotator` Der Annotator ist für die Erzeugung von *information objects* zuständig, er generiert zu einem logischen Programm passende Meta-Informationen.

7.3.4 Desire-Generierung

D. Hoelzgen

In diesem Abschnitt wird die Implementierung der *desire generation* beschrieben, welche im Kapitel 2.4 konzipiert ist. Zu diesem Zweck werden zunächst die an der *desire generation* beteiligten Klassen beschrieben, anschließend wird auf das Anlegen von Motiven für einen Agenten eingegangen.

Paket- und Klassenstruktur

Alle zur *desire generation* notwendigen Klassen und Interfaces befinden sich im Package *edu.udo.cs.cowbots.bdi.desgen*. Es beinhaltet die Möglichkeit zur Darstellung des aktuellen *desire states* des Agenten und bietet Methoden um auf diesen zuzugreifen und gemäß der dem Agenten zugewiesenen Motive in Abhängigkeit der aktuellen Situation zu aktualisieren.

Das Herzstück der *desire generation* ist die DesireComponent. Diese verfügt mit dem MotiveSet über die Motive des Agenten, und kann diese nutzen, um den DesireState in Abhängigkeit der übergebenen *beliefs* zu aktualisieren. Dies umfasst sowohl das Hinzufügen von neuen *desires*, als auch das Anpassen der Intensität der schon vorhandenen *desires*. Andere Komponenten im Framework greifen nutzen die DesireComponent, um Zugriff auf den aktuellen DesireState zu erhalten.

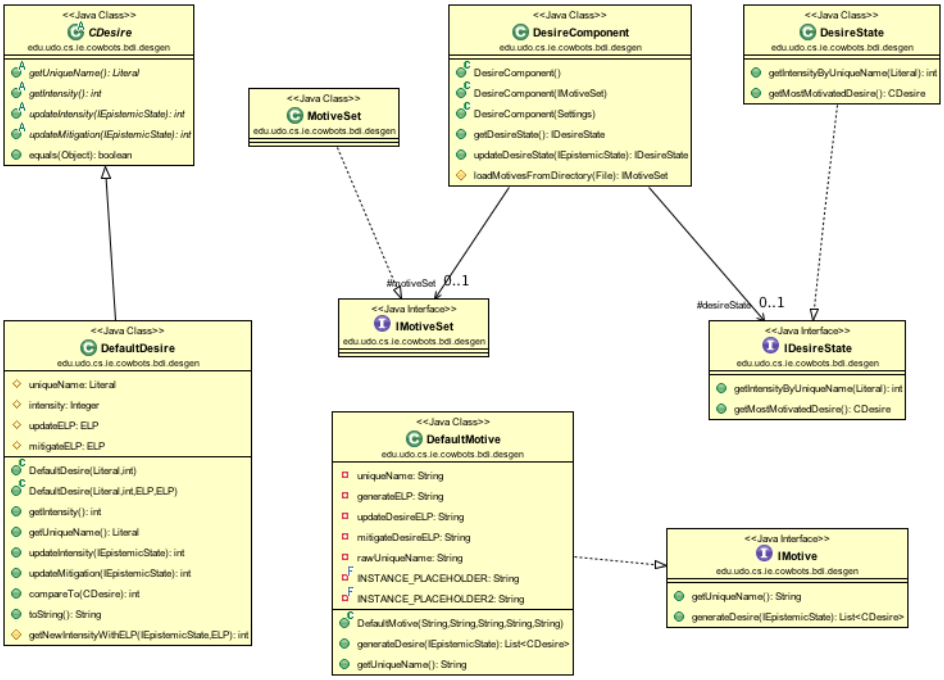

Abbildung 7.8: Strukturdiagramm des edu.udo.cs.ie.cowbots.bdi.desgen Pakets

IMotive Dieses Interface kann von einer Klasse implementiert werden, welche ein Motiv des Agenten darstellt. Hierbei ist zu beachten, dass diese Klasse als Container für die das Motiv darstellenden ELPs genutzt wird. Neben einer Methode, die abhängig von der aktuellen Situation des Agenten neue *desires*

erzeugt, definiert das Interface eine Methode, um den Namen des Motivs auszulesen. Das Interface wird gemäß des Konzepts von der Klasse *DefaultMotive* implementiert.

IMotiveSet Dieses Interface wird von einer Klasse implementiert, welche die aktuelle Motiv Menge des Agenten fasst. Das Interface selbst erbt seine Mengeneigenschaften von der Java Klasse *List*, um das Vorhandensein aller notwendigen Mengenoperationen sicherzustellen. Dieses Interface wird von der Klasse *MotiveSet* implementiert, einer Unterklasse der Java Klasse *ArrayList*.

CDesire Diese abstrakte Klasse dient als Grundlage für Klassen, welche ein *desire* des Agenten darstellen. Sie wird mit ELPs zum Anpassen der Intensität instantiiert, und bietet darauf aufbauend Methoden zum Anpassen und Abschwächen der Intensität, abhängig von der aktuellen Situation des Agenten. Neben der Möglichkeit, auf die Darstellung des *desires* als Literal zuzugreifen implementiert diese Klasse auch das *Compareable* Interface, um eine Sortierung gemäß der Intensität zu ermöglichen. Diese abstrakte Klasse wird gemäß des Konzepts von der Klasse *DefaultDesire* implementiert.

IDesireState Dieses Interface wird von einer Klasse implementiert, welche den aktuellen *desire state* des Agenten darstellt. Obwohl dieser im Konzept als Menge definiert ist, erweitert das Interface das in Java vorhandene Listen Interface *List*. Dies ist darin begründet, dass der Vergleich der Intensität nicht als Grundlage zum Vergleich auf Gleichheit zweier Desires herangezogen werden kann - Dieser Umstand ist in sortierbaren Mengen in Java nicht berücksichtigt. Dieses Interface wird von der Klasse *DesireState* implementiert, einer Unterklasse der Java Klasse *LinkedList*.

DesireComponent Diese Klasse stellt das Herzstück der *desire generation*. Es verwaltet das *motive set* und den *desire state* des Agenten, und wird gemäß des Konzepts im BDI Durchlauf aufgerufen, wenn die Arbeit der *belief revision* beendet ist. Der aktualisierte *desire state* des Agenten kann anschließend von den folgenden Komponenten als Grundlage für die Handlungen des Agenten genutzt werden, insbesondere von der *deliberation* bei der Goal Auswahl.

Erzeugung von Desires durch die Motive des Agenten

In diesem Abschnitt soll die Funktionsweise der Motive zur Erzeugung von *desires* erklärt werden. Jeder Agent kann mit einer beliebigen Menge von Motiven initialisiert werden. Hierzu wird für jeden Agent der Pfad zu dem Ordner, in welchem sich die seine Motive beinhaltenden Dateien befinden, angegeben. Ein Motiv besteht dabei aus drei verschiedenen ELPs, die jeweils einen unterschiedlichen Zweck verfolgen

- Erzeugen von neuen *desires*

```
#desgen
motivegeneratorfound(X) :- cow(X), not catched(X).
catched(C) :- ison(owncorral, X, Y, _), ison(C, X, Y, _).
#desgen end

#intupdate
task_accepted :- accepted(_, drivencow, $INSTANCE$, _).
newintpart(d,S) :- dist($INSTANCE$, S).
newintpart(w,S) :- weight($INSTANCE$, S).
desirefunctionnewint(S) :- #sum{I,C : newintpart(C, I)} = S.
#intupdate end

#intmitigation
desirefunctionnewint(0) :- catched($INSTANCE$).
desirefunctionnewint(0) :- not knowcow($INSTANCE$).
catched(C) :- ison(owncorral, X, Y, _), ison(C, X, Y, _), cow(C).
knowcow(C) :- ison(C, _, _, _), cow(C).
#intmitigation end

#uniquename
drivencow($INSTANCE$)
#uniquename end
```

Abbildung 7.9: Beispielhaftes Motiv: drivencow

- Aktualisieren der Intensität als Grad der Motivation
- Abschwächen der Intensität bei Erfüllen des zugehörigen *goals*

Der Inhalt einer solchen ein Motiv beschreibenden Datei ist, zum besseren Verständnis leicht vereinfacht, in Abbildung 7.9 dargestellt.

In dieser werden die einzelnen ELPs jeweils durch eine mit einer Raute beginnenden Zeile begonnen bzw. beendet. Das erste ELP dient hierbei dem erzeugen von neuen *desires*. In diesem Fall soll ein *desire* für jede dem Agenten bekannte Kuh erzeugt werde, die sich nicht in dem eigenen Pferch befindet. Die nächsten beiden ELPs beschreiben die Aktualisierung und Abschwächung der Intensität. Hierbei ist zu beachten, dass die beiden Programme für die bei der Erzeugung des *desires* gefundene Kuh instantiiert wird. Der letzte Abschnitt beschreibt das Literal, welches genutzt wird um das *desire* im Wissen des Agenten darzustellen.

Der Ablauf zum Aktualisieren des *desire states* des Agenten wird somit analog zum Konzept wie folgt durchgeführt:

- Die Desire Component wird aufgerufen, um den *desire state* des Agenten zu aktualisieren.
- Für alle Motive im *motive set* werden die zugehörigen *desires* erzeugt.

Falls in diesem noch nicht vorhanden, werden sie dem *desire state* des Agenten hinzugefügt.

- Die Intensität als Grad der Motivation aller *desires* im *desire state* wird zunächst aktualisiert, anschließend bei Erfüllung des zugehörigen *goals* abgeschwächt.
- Ist die Intensität auf Null gefallen, so wird das *desire* entfernt.

Anschließend kann der aktualisierte *desire state* von den anderen Komponenten genutzt werden um das weitere Handeln des Agenten zu bestimmen.

7.3.5 Deliberation

A.Löwen

Die Deliberationskomponente
edu.udo.cs.ie.cowbots.bdi.delib.DeliberationComponent.java
bildet einen Adapter zwischen der eventgesteuerten Jason-Plattform, den in AgentSpeak geschriebenen Plänen und der Agenten Klasse. Außerdem ist diese Klasse für das Wechsel des Ziels zuständig. In folgenden Abschnitten wird auf die einzelnen Attribute Methoden eingegangen, wobei auf die Beschreibung der *getter/setter* verzichtet wird.

k Der Wert dieses Attributes stellt fest, um welchen Faktor sich die Intensität des zuletzt verfolgten Ziels (*lastGoal*) und dem aktuell am höchsten motivierten *Desire* mindestens unterscheiden müssen, damit ein Zielwechsel stattfindet. Dieses soll es ermöglichen bei einem verfolgten und bereits geplanten Ziel zu bleiben, obwohl ein anderes etwas mehr motiviert wird.

internalActionDesire Mit Hilfe einer in Jason bereits vordefinierten Methode *desire(Literal l)* prüfen wir, ob ein Agent ein Ziel verfolgt, welches er laut der Motivation verfolgen sollte. Falls einer der Pläne für das Erreichen eines Ziels fehlschlägt, der Agent dieses aber immer noch erreichen möchte, wird ein neues *event* dafür erzeugt.

lastGoal Dieses Attribut enthält das zuletzt als Ziel ausgewählte *desire*.

reconsider Die Methode (*reconsider(Queue¡Event¿ events, IDesireState desireState, TransitionSystem ts)*) greift auf den *DesireState* zu, wählt das am meisten motivierte *Desire* aus, entscheidet ob es ein Zielwechsel passieren soll und ob und welches *Event* neu erzeugt werden soll. Falls kein neues *Event* nötig ist, wird die Warteschlange mit auf andere Weise erzeugten *events* abgearbeitet. Diese stammen aus den in AgentSpeak geschriebenen Plänen, durch die Kommunikation oder Wissensänderungen der Agenten. Außerdem war es notwendig ein *dummy-event* zu benutzen. Dieses dient zur Aufrechterhaltung des Lebenszyklus eines Agenten, falls er noch keine Ziele und Intentionen hat, bzw. noch keine *desires* erzeugt sind.

reconsiderMessage Bei dieser Methode handelt es sich um eine Erweiterung der Behandlung der Kommunikation zwischen Agenten. Anders als bei der Jason-Plattform werden alle eingegangenen Nachrichten sofort behandelt. Dieses ist notwendig, da ansonsten die Nachrichten die *Events* Warteschlange überfluten und es dazu kommt, dass ein Agent nie zu etwas anderem kommt bzw. noch mit dem veraltetem Wissen arbeiten muss und evtl. auf die Anfragen der anderen Teammitgliedern nicht reagieren kann.

7.3.6 Dynamisches Planen

E. Böhmer

Das Paket *edu.udo.cs.ie.cowbots.planer* beinhaltet die Planungskomponenten der Agenten, mit deren Hilfe sowohl ganze Pläne dynamisch erzeugt werden können, als auch kleinere Teilschritte berechnet werden können. Dazu werden sowohl andere interne Aktionen der Agenten, als auch die DLV^K-Komponente aufgerufen.

Die Planerklassen werden in .asl-Plänen als interne Aktion aufgerufen. Die Schnittstelle zum dynamischen Planen bietet hierbei ein Aufruf von `makePlan` mit einem Schlüsselwort, das die Art des Planens bestimmt.

makePlan

Für die folgenden Situationen ist ein Aufruf von `makePlan` vorgesehen:

1. Kühe sollen in den Pferch gebracht werden und der Leader braucht Wegpunkte dafür.
2. Ein Scout will zu seiner *ScoutDestination* und benötigt dafür Wegpunkte.
3. Der Agent will zu einem bestimmten Punkt gelangen und möchte eine Folge von Aktionen, die ihn zu diesem Punkt bringen
4. Der Agent möchte dynamisch planen, das heißt es ist ihm unbekannt, ob er einfach weiter laufen muss oder eventuell eine Aufgabe wie das Öffnen eines Gatter bevorsteht.

Zu Fall 1: In diesem Fall ist ein Aufuf `makePlan(driving, X, Y, Name, N)` nötig. X und Y sind dabei die Zielkoordinaten, also die Koordinaten des Zentrum des Pferches. $Name$ ist der Name der Kuh. In N stehen nach dem Aufruf die Anzahl der Wegpunkte, die berechnet wurden. Die Wegpunkte selbst werden bei dem Aufruf den Beliefs hinzugefügt, nachdem die alten und nun nicht mehr benötigten Wegpunkte aus den Beliefs gelöscht wurden.

Zu Fall 2: Dieser Fall läuft analog zum ersten Fall, mit der Ausnahme dass bei der Erzeugung der Wegpunkte die Sonderbehandlung des eigenen Pferches genutzt wird. Außerdem geben X und Y die Zielkoordinaten des Agenten, nicht die einer Kuh, an. Der Aufruf hat dann die Form
`makePlan(scouted, X, Y, Name, N)`.

Zu Fall 3: In diesem Fall ist ein Aufruf `makePlan(reach, X, Y, Me)` nötig. X und Y sind wie eben die Zielkoordinaten des Agenten, Me der Name des Agenten. Ein Plan, der den Agenten an sein Ziel führt, wird erzeugt und der *PlanLibrary* des Agenten hinzugefügt. Aufgerufen werden kann der neue Plan mit `!reach;`.

In diesen drei Fällen ruft *makePlan* die `generate`-Methode der Klasse *PlanGenerator* (s.u.) auf.

Zu Fall 4: Es ist ein Aufruf `makePlan(dynamic,X,Y,C)` nötig, wobei C wieder die Kuh ist, die getrieben werden soll, X und Y die Koordinaten des aktuell ersten Wegpunktes. Der neu erzeugte Plan wird mit `!dynamic` aufgerufen.

Für diesen Fall wird die Klasse *HigherPlanner* genutzt, die eine Schnittstelle zur DLV^K-Komponente bietet, mit deren Hilfe der dynamische Plan erstellt wird.

Bevor die in den Fällen drei und vier erzeugten .asl-Pläne der Planlibrary hinzugefügt werden können, müssen eventuell vorhandene Pläne mit gleichem Namen gelöscht werden.

PlanGenerator

Im Folgenden werden die Methoden der Klasse *PlanGenerator* erläutert.

generate Die Hauptmethode des PlanGenerators,
`generate(ts, goal, task)`, wird aufgerufen mit dem *TransitionSystem*, einem ELPLiteral, welches das aktuelle Ziel des Agenten angibt, und einem String[]. Letzteres beinhaltet das Keyword des `makePlan`-Aufrufes (*scouted, driving* oder *reach*) sowie das Individuum (Kuh, Agent) für das geplant werden soll.

Für die Keywords *scouted* und *driving* sollen neue Wegpunkte berechnet werden. Daher werden zunächst die alten Wegpunkte gelöscht und anschließend wird die `generateWaypoints`-Methode der Klasse *createWaypoints* aufgerufen. Die dort erzeugten Wegpunkte werden den Beliefs des Agenten hinzugefügt und gezählt. Die Rückgabe von `generate` ist dann ein String der `"numberOfWaypoints"` und die Anzahl der Wegpunkte beinhaltet.

Für das Keyword *reach* soll eine Folge von atomaren *intentions* erzeugt werden, die jeweils einer Bewegungsaktion in der Umwelt entspechen und den Agenten an den in `goal` angegebenen Zielpunkt bringen. Hierfür wird die `getPath`-Methode von *astar* aufgerufen. Die Rückgabe von `generate` ist dann ein String, der einen .asl-Plan kodiert, welcher der *PlanLibrary* des Agenten hinzugefügt werden kann. Dieser Plan hat stets die Form
`!reach: true <- ?iam(Me); direction;... direction.`
wobei die *directions* die Richtungsangaben sind, die *astar* berechnet hat.

HigherPlanner

Der hier erzeugte Plan beinhaltet häufig `.send`-Anweisungen, über die den anderen Agenten mitgeteilt werden kann, wohin sie gehen sollen.

KPlanner

Zusätzlich zu der festgeschriebenen, statischen Planung, für die verschiedenen Agententypen, ist es aber auch notwendig in der Situation dynamisch planen zu können, dazu wird eine Erweiterung von *DLV* genutzt: DLV^K [7].

Ein solches K-Programm besteht aus einer Reihe von Fluent- und Aktionsdeklarationen, gefolgt von einer Reihe von Regeln, die die Ausführbarkeit von Aktionen sowie deren Folgen beschreiben. Außerdem enthält es eine Beschreibung des Startzustandes sowie den gewünschten Endzustand der Welt. Zusammen mit einem ELP (s.o.) als statisches Hintergrundwissen über die Welt, kann das *Planning Frontend* des bereits vorgestellten *answer set sovlers DLV* Pläne für die Agenten erstellen.

Klassen In den folgenden Abschnitten werden die Klassen erläutert, die nötig sind, um ein K-Programm darzustellen und DLV^K zu nutzen.

KDeclaration Jedes *K* Programm enthält eine Reihe von Fluent- und Aktionsdeklarationen, die aus dem zu definierenden Atom gefolgt von dem Schlüsselwort `requires` und den benötigten Typen bestehen. Die Atome und Typen werden durch ELPAtome dargestellt. Zur Erzeugung einer *KDeclaration* wird dann ein Atom und eine Liste der Typen benötigt.

```
ELPAtom fluent = new ELPAtom("incorral", "Ag");
ArrayList<ELPAtom> types = new ArrayList<ELPAtom>();
types.add(new ELPAtom("agent", "Ag"));
KDeclaration d = new KDeclaration(fluent, types);
```

würde die Fluentdeklaration `incorral(Ag) requires agent(Ag).` erzeugen.

KRule Der Hauptteil des K Programms besteht aus zwei Typen von Regeln: *causation rules* und *executability conditions*. Beiden Arten ist gemein, dass sie im *always*-Teil des Programmes auftreten können und sie syntaktisch ähnlich aufgebaut sind. Die abstrakte Klasse *KRule* bietet daher Funktionalitäten, die von beiden Regelarten gebraucht werden. Dazu gehört vor allem die Möglichkeit, dass ein *if*- und *afterpart* vorhanden sind.

KCausationRule Eine *causation rule* zeigt, welche Abhängigkeiten in der Welt bestehen. Der Kopf, *caused*, ein einzelnes Literal, der Regel beschreibt, was passiert wenn der *ifpart* der Regel erfüllt ist und zuvor der *afterpart* der Regel erfüllt war.

Ein Beispiel: Durch

```
KRule krule = new KCausationRule(new NegLiteral(
new ELPAtom("incorral", "Ag1")));
krule.addAfter(new ELPAtom("ask_free", "Ag1", "Ag2"));
```

wird die Regel `caused ~incorral(Ag1) after ask_free(Ag1, Ag2).` erzeugt.

KExCondition Eine *executability condition* gibt Bedingungen zur Ausführung einer Aktion an. Der Kopf, *executable*, ein Aktionsliteral, ist ausführbar, wenn der *ifpart* der Regel erfüllt ist. Ein Beispiel: Durch

```
KRule krule = new new KExCondition(
new ELPAtom("ask_free", "Ag1", "Ag2"));
krule.addIf(new ELPAtom("incorral", "Ag1"));
krule.addIf(new NotLiteral(new ELPAtom("incorral", "Ag2")));
```

wird die Regel `executable ask_free(Ag1,Ag2)`
`if not incorral(Ag2), incorral(Ag1).` erzeugt.

KProgram Ein KProgram stellt ein Programm dar, mit dem DLV^K planen kann. Alle für ein vollständiges Programm erforderlichen Komponenten können mit einer passenden `add`-Methode hinzugefügt werden.

Die `saveAs`-Methode von KProgram bietet, zusammen mit den `toString`-Methoden aller K-Klassen, die Möglichkeit das Programm syntaktisch korrekt abzuspeichern. Damit ist es anschließend möglich, mit der DLV^K-Komponente zu planen.

DLV Die in 7.3.2 bereits vorgestellte DLV Klasse enthält auch die Funktionalität, ein K-Programm ausführen zu lassen. Der Hauptunterschied in der Ausführung von DLV zu DLV^K liegt darin, dass die Kommandozeile für den Aufruf anders aufgebaut werden muss. Für den K-Planer ist es notwendig, das K Programm und ein passendes statisches Hintergrundwissen (in Form eines ELPs) abzuspeichern und als Parameter in der Kommandozeile zu übergeben. In der `computePlans`-Methode wird diese Kommandozeile aufgebaut und in der `runDLVK`-Methode findet der DLV^K-Aufruf statt.
Anschließend wird das Ergebnis geparst, damit es für die weitere Verwendung nutzbar ist.

7.3.7 Action Selection

A.Löwen

Im Packet *edu.udo.cs.ie.cowbots.bdi.actsel* ist die *ActionSelection* Komponente untergebracht. Diese beinhaltet ein Interface (*CActionComponent*), und eine Klasse (*CActionComponent*), die dieses implementiert. Für jede atomare Intention in diesem Szenario gibt es eine einzige Aktion, die diese Intention erfüllt. Deswegen beinhaltet die Implementierung lediglich eine Methode *selectActionForIntention(Literal atomicIntention)*, die einen Literal für atomare Intention als ausgewählte Aktion zurückliefert. Wie im Kapitel 2.7 angemerkt ist, kann man sich Szenarios vorstellen, bei denen die Aktionsauswahl deutlich komplexer ist. Damit unser System erweiterbar bleibt, gibt es das oben genannte Interface.

7.3.8 Cowbot Agenten Architektur

T. Vengels

Ein wichtiger Aspekt bei der Implementierung des Cowbots-Multi-Agenten-Systems war die Anbindung der Massim-Umgebung. Die Behandlung von Aktionen und Versorgung mit Wahrnehmungen geschieht in Jason agentenseitig durch die Agentenarchitektur-Klasse von Jason, die im `jason.architecture.AgArch` angesiedelt ist. Die dafür zuständigen Methoden wurden überschrieben, und durch eine `EnvProxy` Schnittstelle abstrahiert. Auf diese Weise ist die Anbindung an unterschiedliche Agenten-Umwelten möglich. Abbildung 7.10 zeigt die Struktur der Architekturklasse.

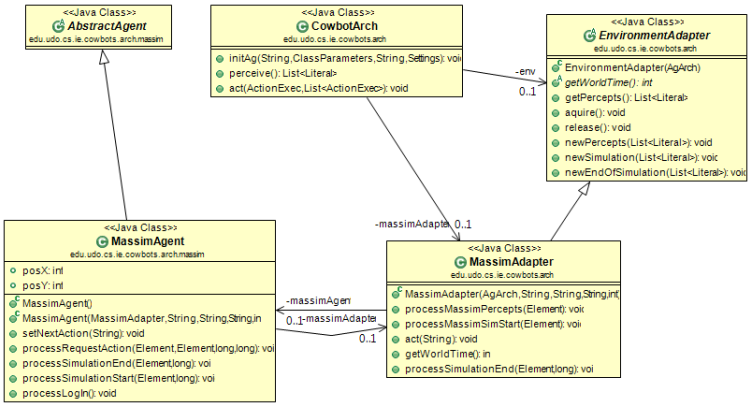

Abbildung 7.10: Strukturdiagramm des edu.udo.cs.ie.cowbots.arch Paketes

Klassenbeschreibung und Funktion

Folgende Klassen und Interfaces wurden definiert:

CowbotArch Die CowbotArch-Klasse erbt von der Jason-Klasse `jason.architecture.AgArch` und stellt die Schnittstelle zwischen Agent und Infrastruktur auf Agentenseite dar. Hier wurden folgende Methoden angepasst:

- *act* dieser Methode wird die Aktion, codiert als ein Literal, übergeben, welche ein Agent in der Umgebung wirken will. Dazu wird das übergebene Literal in seine Textdarstellung überführt (dieser Prozess wird von der *action selection* übernommen), und mittels einer `EnvironmentAdapter`-Instanz an eine Umgebung gesendet.

- *perceive* statt Umweltinformationen aus der Jason-Infrastruktur anzunehmen, wird die Versorgung über Instanzen der Klasse `EnvironmentAdapter` gewährleistet. Hierzu wird geprüft, ob neue Umweltinformationen vorliegen, was in den betrachteten zeit-diskreten Szenarien lediglich einmal pro

Welttakt passieren kann. Liegen neue Umweltinformationen vor, werden diese an den Agenten weitergeleitet.

EnvironmentAdapter eine abstrakte Basisklasse, um neue Wahrnehmungen für die CowbotArch bereitzustellen und Handlungen von der CowbotArch an eine Umgebung zu senden. In spezialisierten, erbenden Klassen erfolgt die spezifische Anbindung an Massim.

MassimAdapter Erbt und implementiert Methoden der EnvironmentAdapter-Klasse. Hier findet der Austausch von Informationen zwischen einem Agenten und der Massim-Umgebung statt.

MassimAgent Die `MassimAgent`-Klasse implementiert die Interpretation des Protokolls von MASSim, hier erfolgt die Transformation in die Wissensdarstellung der Projektgruppe. Die nötigen Rohdaten werden von der Klasse `AbstractAgent` geliefert, die in Literale transformierten Informationen dann über die Klasse `MassimAdapter` der Agentenarchitektur und somit dem Agenten selbst zur Verfügung gestellt.

AbstractAgent Diese Klasse realisiert eine nebenläufige, TCP-basierte Kommunikation mit einem MASSim-Server. Bei Verbindungsabbrüchen wird diese automatisch (mit Hilfe eines Threads der den Verbindungsstatus überwacht) wiederhergestellt. Die eigentliche Auswertung der Daten, die ein Agent seitens der Simulation erhält, erfolgt in der Klasse `MassimAgent`. Hier findet die Übersetzung des XML-basierten Protokolls in die in Abschnitt 5.1 vorgestellte Wissensrepräsentation statt.

7.3.9 Cowbot Agenten-Klasse

T. Vengels

Dieser Abschnitt beschreibt die Modifikationen der Jason-Klasse `jason.agent.Agent`, welche Grundlage für die Implementierung des Cowbot-BDI-Modells ist. Um die erweiterte Funktionalität der Cowbot-BDI-Agenten in Java und Jason nutzen zu können, wurde zunächst ein Interface entworfern, welches Zugriff auf alle Kernkomponenten bildet.

ICowbot Definiert das grundlegende Interface eines Cowbot-Agenten auf Java-Ebene. Es besteht im Wesentlichen aus *getter*-Methoden, welche den Zugriff wichtige Module in Java, insbesonders bei der Implementierung interner Aktionen, ermöglichen. Verkürzt seien die wichtigsten Methoden umrissen:

- *getEpistemicState* liefert die `IEpistemicState` Instanz eines Agenten. Für die Beschreibung dieser Schnittstelle siehe 7.3.3
- *getCowDesires* liefert eine Instanz der `DesireComponent` Schnittstelle. Für die Beschreibung dieser Schnittstelle siehe 7.3.4

- *getCowDelib* liefert eine Instanz der `DeliberationComponent` Schnittstelle. Für die Beschreibung dieser Schnittstelle siehe 7.3.5
- *getActionSelection* liefert eine Instanz der `IActionComponent` Schnittstelle. Für die Beschreibung dieser Schnittstelle siehe 7.3.7
- *getDLV* liefert eine Instanz des *answer set solvers* DLV. Für die Beschreibung dieser Schnittstelle siehe 7.3.2

Sofort wird klar, dass jede Cowbot-BDI kompatible Agentenimplementierung diese Schnittstelle erfüllen muss, symbolisiert sie doch die objektorientierte Umsetzung des konzeptionellen Entwurfs. Darüber hinaus bietet die Schnittstelle eine hilfreiche Methode bei der Entwicklung von Agenten basierend auf der `lib_cowbots`:

- *getUI* liefert eine Instanz der Debug-Oberfläche, vorgestellt in Abschnitt 7.3.10.

CowbotAgent Die `CowbotAgent` Klasse ist die zentrale Verbindung zwischen den Cowbot-BDI-Komponenten und Jason. Nach der Vorstellung von Jason und dem AgentSpeak-Interpreter, folgt die Erläuterung welche Methoden wie angepasst wurden.

- *buf* Diese Methode wird in jedem *reasoning cycle* aufgerufen. Hier prüft der Agent, ob neue Umweltwahrnehmungen vorliegen. Falls neue Informationen vorliegen, erfolgt an dieser Stelle nacheinander der Aufruf folgender Cowbot-BDI-Methoden:

  1 `CRevisionComponent` die implementierte Revisionsfunktion, neue Umweltinformationen werden hierüber in die logische Wissensbasis eingepflegt. Darauf revidiert ein Agent sein Wissen. Die Änderungen gegenüber dem vorherigen *belief set*, werden mit der Jason-`BeliefBase` abgeglichen.

  2 `CDesireComponent` nachträgliche Aktualisierung und Feintuning von *desires*, insofern die benötigten Rechenregeln nicht vollständig durch logische Programme abgedeckt wurden konnten.

  3 `CDeliberationComponent` die Deliberation wird aufgerufen und entscheidet auf Grundlage der Wissensänderung und der *desires*, ob ein *goal* erzeugt und als Intention verfolgt wird.

- *brf* Diese Methode ist von Jason zur Revision von Wissen vorgesehen. Diese Methode wird von Jason bei Wissensänderungen mittels + und − Anweisungen aus ASL-Plänen heraus. Da die Deliberation und Planausführung auf Basis des mittels ELPs generierten Wissens steuert, gehen wir davon aus dass diese Wissensänderungen notwendige Berechnungen des Interpreters zur Planausführung sind. Insofern reicht es hier, die logische Wissensbasis mit den Änderungen abzugleichen, anstatt automatisch einen vollen Revisionszyklus auszuführen.

- *socAcc* diese Methode ermöglicht es einem Agenten, eingehende Nachrichten anderer Agenten zu akzeptieren oder abzulehnen. Die Umsetzung des Konzeptes sieht ein kooperatives Agentensystem vor, von daher können alle Nachrichten akzeptiert werden.

- *selectMessage* diese Methode ermöglicht dem Agenten, aus noch nicht bearbeiteten Nachrichten anderer Agenten eine bevorzugte auszuwählen. Der in der Nachricht kodierte Sprachakt wird dann von dem AgentSpeak-Interpreter umgesetzt. In diese Methode klinkt sich die Deliberations-Komponente ein, für Details siehe 7.3.5

- *selectEvent* ermöglicht in Jason die Auswahl eines Ereignisses, welches der Interpreter als nächstes bearbeiten soll. In diese Methode klinkt sich die Deliberation ein, für Details siehe Kapitel 7.3.5

- *selectIntention* ermöglicht es, dem Jason-Interpreter die nächste abzuarbeitende Intention vorzugeben. Standardmässig werden Intentionen (und die dort abzuarbeitenden Pläne) Schritt für Schritt im *round robin*-Verfahren abgearbeitet. In diese Funktion klinkt sich die Deliberation ein und übernimmt das Scheduling von Intentionen, für Details siehe Kapitel 7.3.5

7.3.10 Graphische Oberfläche

T. Vengels

Zur Visualisierung des inneren Zustandes eines Agenten wurde eine graphische Oberfläche implementiert. Diese informiert den Benutzer über den Zustand der Cowbot BDI Erweiterungen, wie den *belief state*, das *belief set*, oder den *desire state*. Darüber hinaus wird die Umwelt des Agenten graphisch, als Kachelwelt, dargestellt. Die Oberfläche dient in der Überwachung des Agenten und implementierter Methoden wie *path finding* oder *clustering* (eben Algorithmen wo ein *live feedback* sinnvoll ist), während der Entwicklung und Testläufen. Sie ist als Ergänzung zur bestehenden Jason-Oberfläche oder als Debug-Hilfe anzusehen, es handelt sich nicht um eine Oberfläche zur Interaktion mit Agenten.

Implementierung

Zentrale Schnittstelle der graphischen Oberfläche ist die Schnittstelle `ICowbotUI`, diese bietet den zuvor definierten mittels derer die Cowbot-BDI Komponenten ihren internen Zustand an die graphische Oberfläche übertragen können. Die Schnittstelle wird von den Klassen `CowUI` und `DummyUI` implementiert. `CowUI` ist eine graphische Anzeige basierend auf Java Swing, die `DummyUI` verwirft alle Aktualisierungen und stellt selbst keine graphische Ausgabe her. Letzteres ist sinnvoll für Systeme, die über keine graphische Ausgabe verfügen (zum Beispiel ein Linux-Server ohne X11). Die Aktualisierung des UI erfolgt durch die Cowbot-BDI Komponenten. Abbildung 7.11 bietet eine Übersicht über die Klassenstruktur des Paketes.

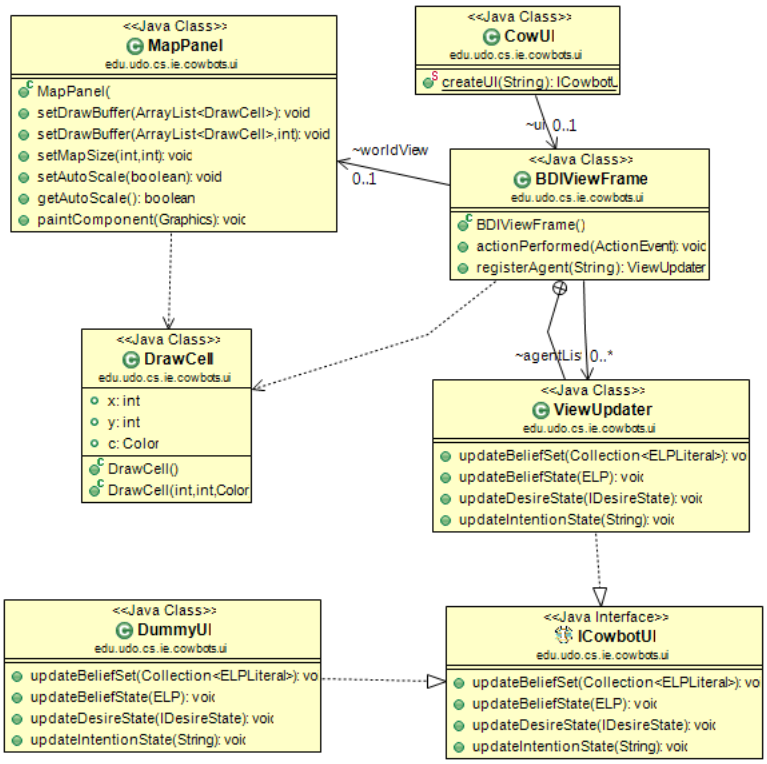

Abbildung 7.11: Strukturdiagramm des edu.udo.cs.ie.cowbots.ui Paketes

7.4 Agentenimplementierung

Nachdem die Architektur beschrieben wurde, soll nun näher auf die einzelnen Aufgaben der Agenten eingegangen werden. Diese sind in 3 Abschnitte unterteilt: Der *scout*, welcher die Welt erkundet, der *cowdriver*, welcher für das Treiben von Kühen zuständig ist, und die Funktion des *dooropeners*, der in beiden Rollen Anwendung findet.

7.4.1 Scout-Implementierung

F.Bienek, B.Jablkowski

Die vorgestellte Implementierung entspricht der Strategie in 6.1.2. Die aktuellen Versionen des Scouts können in den ASL-Dateien *exScoutHelper* und *exScoutLeader* eingesehen werden.

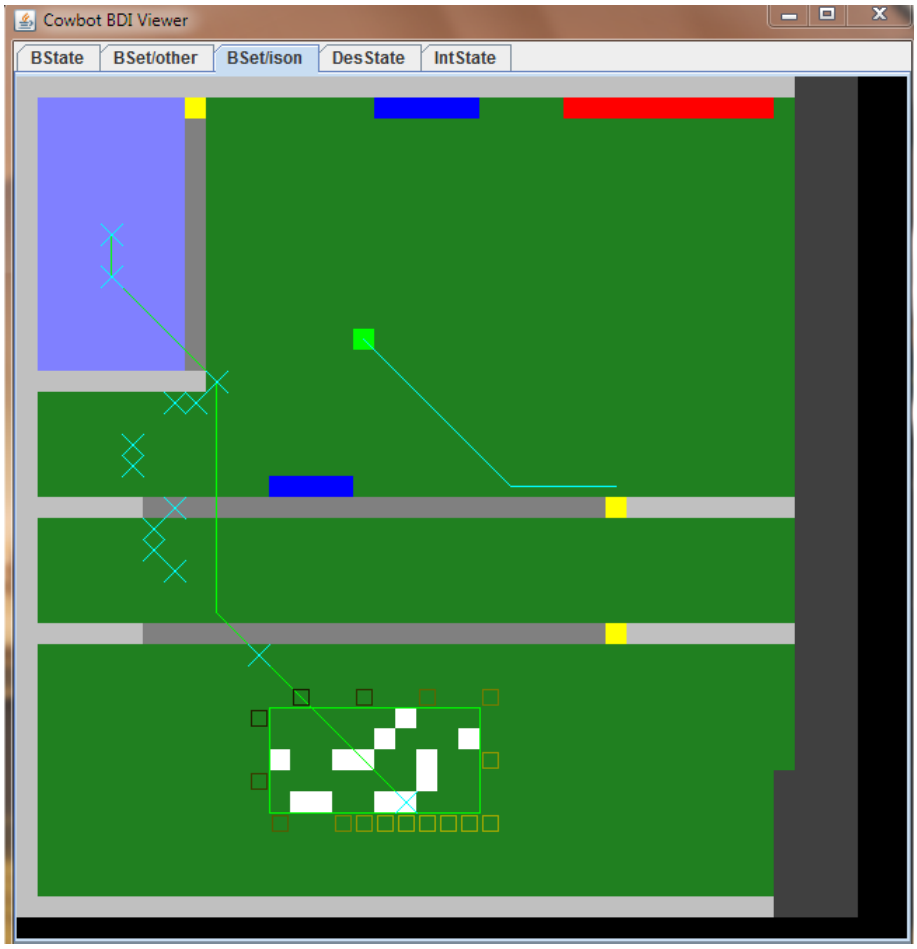

Abbildung 7.12: Bildschirmaufnahme der Cowbot-BDI Oberfläche

Struktur

Die ASL-Pläne eines Agenten werden durch die Generierung von Events angeworfen. Die Deliberationskomponente erzeugt anhand der *desires* ein Event, welches in den ASL-Plänen abgearbeitet wird. Wird in den *desires* der Wunsch, die Welt zu erkunden, bevorzugt, so entspricht das erzeugte Event dem Event "*scouted*". Dieses Event bildet den Startschuss der *scout*-Pläne.

Wir haben uns dazu entschieden, die Pläne des *scouts* in einem Baum zu strukturieren, da so die Aktionen der Agenten besser getrennt werden können. Zentrale Aufgaben eines *scout-leaders* beinhalten unter Anderem das Finden eines geeigneten Ziels in der Welt, das Generieren der Wegpunkte dorthin und die Verteilung selbiger an alle *scout-helper* der eigenen Gruppe. Der Aufbau der Bäume für *scout-leader* und *scout-helper* werden in den Abbildungen 7.13 und

7.14 skizziert.

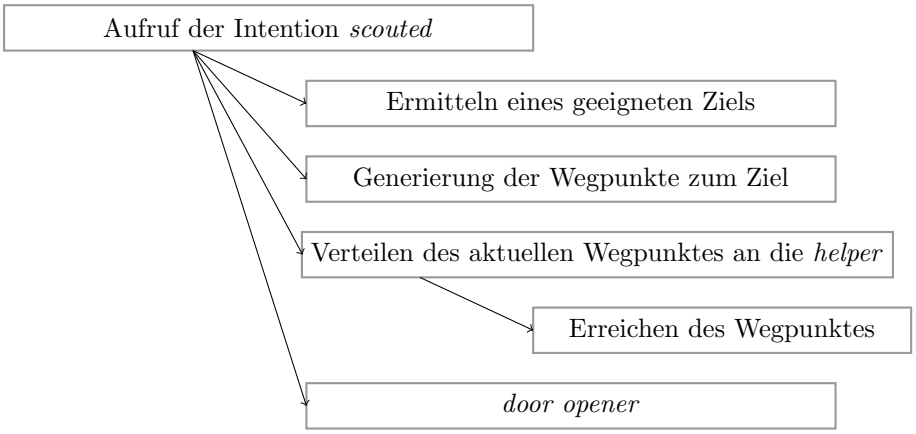

Abbildung 7.13: Skizze der Struktur des *scoutleaders*

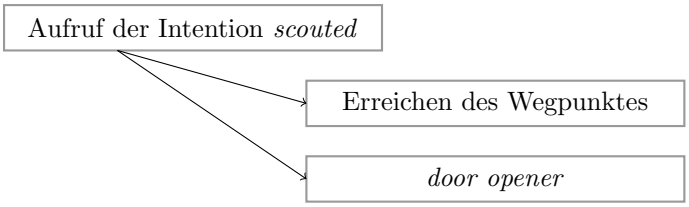

Abbildung 7.14: Skizze der Struktur des *scouthelpers*

Interaktion zwischen *helper* und *leader* Wie zu erkennen ist, ist die Struktur des *helpers* sehr simpel. Ein interessanter Punkt bildet die Interaktion zwischen dem *leader* und den Agenten, die ihm helfen. Frühere Implementationen haben gezeigt, dass die Kommunikation über ASL-Pläne äußert fehleranfällig ist, sobald die Anzahl der kommunizierten Nachrichten „zu groß“ wird. Wir haben uns daher dazu entschieden, den *scout-helper* so zu strukturieren, dass er den Kommandos seines *leaders* folgt, sobald er die Aufforderung dazu bekommt (d.h. wenn er einen neuen Wegpunkt von seinem *leader* bekommt). Der Großteil der Aufgabenverteilung wird vom *leader* übernommen und verteilt. Der *helper* übernimmt neue Ziele, indem er den bisherigen Wegpunkt löscht und den neuen seinem Wissen hinzufügt. Er ruft dazu einen Plan auf, welcher auf das Hinzufügen eines *beliefs* wartet. Dieser Plan sorgt sowohl für den Wechsel zwischen altem und neuem Wegpunkt, als auch dafür, dass der Agent seine zur Zeit verfolgten Aktionen abbricht (hier realisiert mit der Jason-internen Aktion „.succeed_goal“, siehe Abb.7.15).

Die Ausführung der neuen Aktion des *helpers* wird dann wieder mit dem

```
+waypoint(TYPE, scouted, _, WC, XCOOR,
                 YCOOR, FromDir, ToDir, _, _, _) : true
<-
   //Aktualisierung des Wegpunktes
   [...]

    //Beenden der momentan aktiven Lautroutine
   .succeed_goal(walkToDest(_,_,_)).
```

Abbildung 7.15: Ausschnitt aus dem Plan zur Aktualisierung des Wegpunktes

```
+!scouted: wpoint(fenceclosewp,scouted,_,WC,XCOOR,YCOOR,_,_,_,_,_,_)
<-
  //Aktionen für die Behandlung von Wegpunkten hinter einem Zaun
  [...]

  //Aktionen für die Behandlung von Wegpunkten vor einem Zaun
+!scouted: wpoint(scoutswitchoutside, scouted, _, WC,
 XCOOR, YCOOR, _, _, _, _, _)
<-
  [...]
```

Abbildung 7.16: Ausschnitt aus dem den Planköpfen des *scouthelpers*

scouted-Event gesteuert. Der *helper* unterscheidet dabei die Typen der Wegpunkte (7.16).

Bewegung der *scouts*

Die aktuelle Version kann in der ASL-Datei *exScoutWalker* eingesehen werden. Die Grundidee der Lauf-Routine sieht in Stichpunkten wie folgt aus:

- Aufruf des *walker*-Plans
- Überprüfung, ob der Agent sich bereits auf dem Zielpunkt befindet.
- Falls ja:
    1. Beenden des Plans mit einem Erfolg.
- Falls nein:
    1. Aufruf des a*-Algorithmusses und Versuch, einen Schritt in die erwünschte Richtung zu machen.
    2. Überprüfung: War die Aktion erfolgreich?
    3. Falls nein:

   (a) Hochzählen eines Zählers, welcher die Anzahl der Schrittfehler mitzählt, um 1.
   (b) Beenden des Plans mit einem Fehlschlag, wenn der Zähler zu viele Fehler anzeigt. Sonst erneuter Aufruf des *walker*-Plans.
4. Falls ja:
   (a) Erneuter Aufruf des *walker*-Plans, Setzen des Zählers auf 0.

Der eigentliche Ablauf entspricht also einem rekursiven Aufruf des *walker*-Plans, bis entweder das Ziel erreicht ist, oder mehrere Aktionen in der Welt am Stück fehlschlagen. Wir haben uns dazu entschieden, diesen Plan um ein paar Funktionen zu erweiteren:

Nähe zum Zielpunkt Für die *scout*-Rolle ist es wünschenswert, schnell von A nach B zu kommen. In frühen Implementationen hat sich herausgestellt, dass das vollständige Erreichen eines Zielpunktes nicht notwendig ist, wenn die Agenten stattdessen schneller sind. Es reicht, „sehr nah dran“ zu sein. Wir haben daher die Bedingung für ein erfolgreiches Beenden der Laufroutine aufgelockert. Ein Agent beendet das Laufen jetzt erfolgreich, wenn er entweder seinen Zielpunkt erreicht hat, oder einen Schrittfehler macht, während der Abstand zum Zielpunkt kleiner als ein Schwellwert ist.

Positionen kommunizieren Die Absprache zwischen *leader* und *helper* erfordert, dass Ersterer die Positionen seiner *helper* kennt, um Aufgaben korrekt verteilen zu können. Im vorliegenden Szenario kann es aber sehr schnell vorkommen, dass der *leader* die Sicht auf seine Helfer verliert (jeder Agent hat ja eine beschränkte Sichtweite). Wir haben die Pläne daher so angepasst, dass jeder *helper* nach einem Schritt seine Position an seinen *leader* schickt. Dieser aktualisiert diese mit einem Plan, der durch die Addition eines *beliefs* aufgerufen wird. So wird das Wissen eine Agenten zeitnah aktualisiert, sodass der *leader* jederzeit die Positionen seines Teams kennt.

Überqueren von Hindernissen

Während die Agenten die Welt erkunden, kann es häufig dazu kommen, dass die Gruppe ein Hindernis überqueren muss. Im vorliegenden Szenario handelt es sich dabei um einen Zaun (*fence*), welcher von einem Schalter (*switch*) geöffnet und geschlossen werden kann. Der Zaun ist immer dann geöffnet, wenn ein Agent direkt neben dem Zaun steht (der Schalter selbst ist ein Hindernis). Befindet sich kein Agent an einer solchen Stelle, schließt sich der Zaun. Daraus ergeben sich für die *scouts* eine Reihe von Aufgaben:

- Identifizieren des Zauns, der überquert werden soll.
- Finden des Schalters, der zum relevanten Zaun gehört.
- Einen Helfer-Agenten bestimmen, welcher den Schalter bedient.

- Sobald der Zaun geöffnet ist, den Zaun überqueren.
- Einen Helfer-Agenten bestimmen, welcher den Zaun von innen offen hält.
- Die restlichen Helfer auf die richtige Seite des Zauns holen

Wir wollen hier nicht detailliert auf jeden Schritt eingehen. Nähere Ideen zur Implementierung finden sie in 7.4.3. Stattdessen wollen wir noch einmal darauf eingehen, wie wir die Nutzung von erweiterten logischen Programmen in den ASL-Plänen nutzen können.

Erkennen eines zusammenhängenden Zauns Wir haben uns dazu entschlossen, die Erkennung von zusammenhängenden Zäunen mit erweiterter logischer Programmierung zu lösen. Da sich die Informationen über die Welt nur einmal pro Welttakt ändern können, reicht es hier, das Regelwissen ebenso oft anzuwenden.

Das relevante Programm kann in 5.2 eingesehen werden („Zaunausrichtung“). Wird dieses Programm von *dlv* ausgewertet, so schließt der Agent von seinen bekannten Zelleninformationen auf zusammenhängende Zäune. Die *gate*-Regeln erkennen einen Zaun und den anliegenden Schalter, wenn dieser bekannt ist. Dazu verwenden die Regeln die Wissensliterale der Welt (*ison*,*fence* und *switch*). Sollte der Agent zu einer ausgewählten Zelle einen Schalter finden, der sich direkt neben der ausgesuchten Zelle befindet, so bildet er daraus ein *gate*. Weitere Zaunzellen werden dann rekursiv hinzugefügt. Häufig kann es jedoch vorkommen, dass die Umgebung der Agenten nicht vollständig aufgedeckt ist. Dies wird dann problematisch, wenn zu einem bekannten Zaunfeld noch kein Schalter gefunden wurde. In einem solchen Fall kann der Agent nicht auf ein *gate*-Literal schließen. Dadurch können die *fencegroup*-Regeln aktiv werden, welche Zaungruppen bilden, die noch keinen assoziierten Schalter besitzen .

Schließt ein Agent aus Regelwissen und Weltwissen nun ein *gate*- oder *fencegroup*-Literal, so ermöglicht es das, in den ASL-Plänen darauf zu reagieren (Abb.7.17).

```
+!scoutGetSwitch(FENCEX,FENCEY) : gate(FENCEX,FENCEY,SW,FD)
<-
   //Ausführung der Aktionen bei bekanntem Schalter
   [...]

+!scoutGetSwitch(FENCEX,FENCEY) : fencegroup(FENCEX,FENCEY,FD)
 <-
   //Ausführung der Aktionen bei unbekanntem Schalter
   [...]
```

Abbildung 7.17: Planköpfe zur Reaktion der *gate*- und *fencegroup*-Literale

Interne Aktionen des Scouts

Die internen Aktionen der Rolle können in folgendem Paket gefunden werden:

- *edu.udo.cs.ie.cowbots.internalActions.scout*

Desweiteren bedienen wir uns einzelnen Funktionen aus den folgenden Paketen:

- *edu.udo.cs.ie.cowbots.internalActions.experimental*
- *edu.udo.cs.ie.cowbots.internalActions*
- *edu.udo.cs.ie.cowbots.planer*.

Es folgt eine kurze Beschreibung der einzelnen Funktionen:

getScoutCandidate Diese interne Aktion ermittelt für den Agenten ein Koordinatenpaar, welches der Agent im Zuge des Erkundens erreichen möchte. Die genaue Strategie kann in nachgelesen werden.

getRandomScoutCandidate Diese Funktion ermittelt für den Agenten randomisiert ein Koordinatenpaar. Dies entspricht der Strategie in einer erforschten Welt (siehe 6.1.2).

checkScoutArea Sobald ein Agent sein Ziel ermittelt hat, muss er prüfen, ob es nicht in Konflikt mit anderen Zielen anderer Agenten steht (Wir wollen unsere Scouts ja verteilen). Dazu nutzen wir *checkScoutArea*. Diese Funktion überprüft alle bekannten Ziele anderer Agenten, und gleicht sie mit dem eigenen Ziel ab.

getRelevantFence Es ist in der Welt nicht immer einfach, ein zu durchschreitendes Tor auch als solches zu erkennen. *getRelevantFence* ermittelt anhand der Ausrichtung der Wegpunkte und der Position der Tore, welches Tor für die Agentengruppe relevant ist.

getOpeningPosition Im Zuge des Zaun-Öffnens muss es eine Möglichkeit geben, die Position zu finden, die eine Bedienung des Schalters ermöglicht. Dazu wird diese Funktion aufgerufen.

getNearestHelper Sobald die Position des Schalters einer Tür bekannt ist, wird durch diese Funktion der Helfer-Agent ausgesucht, der die geringste Reichweite zu dieser Position hat.

getScoutSwitchFinder Ein Problem einer nicht vollständig aufgedeckten Welt ist, ein Tor durchlaufen zu wollen, ohne die Position des passenden Schalters zu kennen. In einer solchen Situation müssen Helfer ausgesucht werden, welche zunächst die Umgebung unteruschen.

checkHelpersNearCWP Hierbei handelt es sich um eine Hilfsfunktion, die es dem *leader* der Gruppe ermöglicht, zu überprüfen, ob seine Helfer das Hindernis überwunden haben (check, if Helpers are near the CloseWayPoint"). Befinden sich alle relevanten Helfer (also alle außer des Toröffners und des TT-or-Offen-Halters") nahe genug am *closewaypoint*, so kann der *leader* fortfahren, den Öffner des Tores zu sich zu rufen.

distanceToCP Funktion, welche im *exScoutWalker* überprüft, ob ein Wegpunkt trotz fehlerhaften Schrittes eines Agenten in der Nähe liegt (näheres hierzu in 7.4.1)

Aus den weiteren Paketen bedienen wir uns folgender Funktionen:

exEasyFormation Jeder Agent sollte in seiner Gruppe eine Formation halten können. Diese Funktion berechnet anhand derPosition des Wegpunktes die neuen Positionen des Agenten.

makePlan Diese Funktion wird von uns genutzt, um Wegpunkte zum Ziel zu gewinnen.

getExpectedCoordinates Funktion, welche im *exScoutWalker* überprüft, auf welches Feld der nächste Schritt führen soll. Dies wird benötigt, um fehlerhafte Schritte zu erkennen, und entsprechend zu handeln (näheres hierzu in 7.4.1).

7.4.2 Driver-Implementierung

A.Löwen

Driver ist eine der wichtigsten Rollen bei unserem Multi-Agenten-System. Für die Implementierung ist diese in zwei Teilrollen aufgeteilt - *leader* und *helper*. Sollte einer der Agenten eine Kuh gefunden haben und sich für das Treiben dieser stark genug motivieren, so wird er zu einem Anführer einer Agentengruppe, die er dynamisch zusammen stellt und leitet. Er sendet anderen Agenten in der Gruppe für das Treiben der Kühe erforderliche Teilaufgaben zu, koordiniert das Vorgehen der Gruppe und löst diese auf, sobald die Aufgabe erfüllt ist oder diese aufgegeben wird. Die Helfer planen selbständig für die Teilziele, die sie bekommen haben und versuchen diese zu erfüllen. Damit die Agenten als ein Team handeln und koordiniert vorgehen können, kommunizieren die Agenten mit dem *leader*, der für die Synchronisation der Abarbeitung der Aufgaben sorgt. Den gesamten Ablauf für das Treiben einer Kuh zum eigenem Pferch kann man wie folgt grob zusammenfassen:

1. Gruppenbildung
2. Sammeln der Agenten bei der zu treibender Kuh
3. Abarbeitung der Wegpunkte(das eigentliche Treiben)

4. Auflösen der Gruppe

Im Folgenden wird auf die wichtigsten Pläne und deren Zusammenspiel eingegangen.

Driver-Leader

drivencow(Cow) ist der Einstiegspunkt für den *leader*. Sollte ein Agent sich für das Treiben der Kuh entscheiden, wird dieses *event* erzeugt und damit der Rollenwechsel eingeleitet. *Cow* ist dabei eine Variable und enthält die *id* der zu treibender Kuh. Als erstes wird für einen sauberen Rollenwechsel gesorgt. Zum einen werden Destruktoren der anderen Rollen aufgerufen. Diese sorgen für das Löschen von nicht mehr aktuellen (wegen dem Rollenwechsel) *mental notes*. Zum anderen wird eine bereits in Jason implementierte interne Aktion

```
succeed_goal
```

benutzt, um die Intentionen der anderen Rollen von dem *stack* zu entfernen. Als letzte Teilaufgabe dieses Plans wird *!multiAgentTask(drivencow,Cow)* angestoßen.

!multiAgentTask(drivencow,Cow) sorgt für das Starten mehrerer Teilziele. Der *leader* soll dafür sorgen, dass die zu treibende Kuh immer im eigenen Sichtradius bleibt. Andererseits soll man die Kuh auch nicht erschrecken. Deswegen folgt der Gruppenleiter der Kuh auf einer Distanz. Dieses wird von einem Teilziel *!!followCow(Cow)* gewährleistet, was als eigene Intention gestartet wird(notiert mit dem doppelten Ausführungszeichen) und erst wenn das eigentliche Treiben beginnt als erfüllt angesehen wird und mit Hilfe von

```
succeed_goal
```

gestoppt wird. Parallel dazu findet die Gruppenbildung statt - *!groupFormed(drivencow,Cow,HelperList)*. Sobald die Gruppe formiert ist und alle Helfer angekommen sind, fangen die Agenten an, die Kuh zu treiben.

!groupFormed(drivencow,Cow,HelperList) diese Methode ermöglicht eine dynamische Gruppenbildung. Der *leader* sendet an alle Agenten aus eigenem Team eine Hilfeanfrage, berechnet anhand der Intensität des *drivencow-desires* die benötigte Gruppengrösse und wartet bis genügend Antworten angekommen sind. Die Agenten, die für das Helfen zugestimmt haben, antworten mit *ack(Task, Cow, MyGoalIntensity, Time)*. Mit Hilfe von einer internen Aktion

```
.count( ack(_,_,_,_)[source(_)],N)
```

wird die Anzahl der zugesagten Agenten ermittelt. Sollten sich zu wenig Agenten gemeldet haben, bleibt es dem *leader* nichts übrig als die Aufgabe abzubrechen. Er teilt dieses anderen Agenten mittels *rejectedall(drivencow, Cow, Time)* mit

und wird für andere Aufgaben frei. Falls sich genug Agenten für diese Aufgabe bereit erklärt haben, werden zwei Listen gebildet. In eine kommen Agenten, die für das Helfen ausgewählt sind, in die andere der Rest. Die Auswahl basiert auf der mitgeteilten Intensität des Wunsches zu Helfen. Da für die Intensitätberechnung die Distanz zu der Kuh eine Rolle spielt, werden diejenigen Agenten bevorzugt, die sich am nächsten zu der zu treibenden Kuh befinden. Mit Hilfe von folgenden Anweisungen werden diese Listen gebildet.

```
    .findall(howMuchDoesTheyWant(Agent,GoalIntensity),
      ack(_, _, GoalIntensity, _)[source(Agent)], AgentsList );
edu.udo.cs.ie.cowbots.internalActions.Leader.makeAckRejLists(
  AgentsList,GroupSize,HelperList,RejectedList);
```

Als erstes wird eine Liste(AgentsList) mit Literalen gebildet, die Namen der Agenten und ihre Intensität für das Treiben *desire* enthält. Diese Liste wird an die interne Aktion *makeAckRejLists* weitergegeben und in zwei Listen(HelperList, RejectedList) aufgespaltet. Die ersten bekommen eine Zusage und dürfen beim Treiben dieser Kuh helfen, die anderen werden abgelehnt und können sich somit für eine andere Aufgabe entscheiden. Den Helfer Agenten werden Informationen über die Herde(*herd(Cow, Radius, Time)*) mitgeteilt und somit ist die Formierung der Gruppe abgeschlossen.

!waitTillAgentsArrived(HelperList) Nachdem die Gruppe formiert ist, müssen sich die Helfer-Agenten zu der zu treibenden Kuh bewegen. Der *leader* wartet bis alle Agenten in der Nähe sind und leitet danach die Abarbeitung der Wegpunkten ein.

!forEachWP sorgt für die Abarbeitung der Wegpunkte. Dabei sendet der *leader* die von ihm berechneten Wegpunkte den anderen Agenten aus seiner Gruppe zu, koordiniert und synchronisiert das Laufen von einem Wegpunkt zu anderem in Formation, achtet auf die Sonderbehandlung bei einigen Wegpunkten(wie z.B. Passieren von Toren). Jeder der Agenten errechnet selbständig eigene Koordinaten in der Halbkreis-Formation. Dafür wird folgende interne Aktion benutzt.

```
edu.udo.cs.ie.cowbots.planer.getMyNextPositionsCoordinates
 (T, XCoord, YCoord, FromDirection, ToDirection, Formationpos,
 AgentsCount, Radius, MyNextXCoordinate, MyNextYCoordinate)
```

Für das Hinlaufen zu den berechneten Koordinaten wird *walkTo subgoal* benutzt. Dieser ermöglicht das Hinlaufen zu einer Position mit einer festgelegter Anzahl der Fehler beim Laufen. Es ist nämlich oft der Fall, dass ein Schritt (Aktion) fehlschlägt, und man nicht unbedingt sofort umplanen muss.

!taskabortLeader ist der Destruktor vom *leader*. Mit Hilfe von einer internen Aktion *.abolish* werden die in diesen Plänen genutzte *mental notes* entfernt.

Driver-Helper

helped(Leader, T, Cow) leitet die Ausführung von einem Helfer-Agenten ein. Die Pläne für diese Rolle sind analog zu denen von *leader* aufgebaut. Da die Helfer oft auf die nächsten Anweisungen von ihrem Anführer angewiesen sind, finden sich in diesen Plänen oft Synchronisationsschleifen wieder. Damit sind kurze Pläne gemeint, die sich selbst rekursiv aufrufen und in denen auf ein bestimmtes Literal gewartet wird. Dieser kann entweder über Kommunikation kommen, oder aber auch durch die Änderungen in der Umwelt in die *beliefbase* des Agenten erscheinen.

7.4.3 Door Opener-Implementierung

S. Broszeit

Der *door opener* ist dafür zuständig, dass eine Agentengruppe ein Gatter passiert, indem sie den zugehörigen Schalter finden, einer der Agenten diesen betätigt, die übrigen das Gatter durchlaufen, ein zweiter Agent den Schalter von der Innenseite des Gatters betätigt und der Außenstehende somit ebenfalls hinter das Gatter gelangen kann. Sind alle Agenten durch das Gatter gelaufen, ist die Aufgabe des *door openers* abgeschlossen.

Der *door openers* entspricht analog zu den anderen Agentenrollen dem *leader-helper*-Konzept. Der *door opener leader* koordiniert das Verhalten der Agentengruppe, während die *door opener helper* auf Nachfrage kleinere Aufgaben übernehmen, wie etwa die Schaltersuche oder das Schalterbetätigen. Die Pläne des *door opener leaders* bestehen entsprechend des beschriebenen Ablaufs aus den Phasen Schaltererkennung, Schalterbetätigung, Gatterdurchlaufen und Neugruppierung.

Da Absprache in eine Multi-Agenten-System sehr zeitaufwendig ist, geht der *door opener leader* von verschiedenen Heuristiken aus, um die Aufgaben geschickt zu verteilen: Da die Agenten eine Formation einhalten, werden für die Suche nach dem Schalter sowie das letztendliche Öffnen nur die äußersten Agenten in Betracht gezogen, da diese sowohl den geringsten Abstand zu ihrem Auftragsgebiet, sowie - bei ihrem Austreten aus der Formation - den geringsten Schaden an ihrer Aufrechterhaltung haben. Aus demselben Grund wird das das Gatter später von dem direkten Formationsnachbarn des Schalter betätigenden Agenten von innen aufgehalten.

Wenn Agenten an einem Wegpunkt ankommen, der anzeigt, dass ein Gatter den Weg versperrt, wird überprüft, welches der zugehörige Zaun ist, indem vom Wegpunkt aus alle Felder in Richtung des nächsten Wegpunktes überprüft werden, bis eines einen Zaun beinhaltet.

```
relateDirection(north, 1, 0).
relateDirection(south, 1, 2).
relateDirection(west, 0, 1).
relateDirection(east, 2, 1).
relateDirection(northeast, 2, 0).
```

```
relateDirection(northwest, 0, 0).
relateDirection(southeast, 2, 2).
relateDirection(southwest, 0, 2).

+!getFence(Wx, Wy, Direction) :
ison(F, Wx, Wy, _) & (switch(F) | fence(F))
<--+currentFence(F, Wx, Wy).

+!getFence(Wx, Wy, Direction) :
relateDirection(Direction, Xoff, Yoff)
<-
!getFence((Wx + (Xoff - 1)), (Wy + (Yoff -1)), Direction).
```

Für diesen Zaun wird nun im eigenen Wissen überprüft, ob man bereits weiss, wo sich Schalter dieses Gatters befindet. Wenn kein Wissen über einen Schalter vorliegt, so werden die beiden äußersten Agenten der Formation aufgefordert nach diesem zu suchen. Dazu bewegen diese sich solange auf direkt neben den äußersten bekannten Zaunfeldern liegende Felder, bis sie einen Schalter, der zum Gatter gehört, wahrnehmen oder der *door opener leader* ihnen mitteilt, dass der in der anderen Richtung suchende Agent bereits fündig geworden ist.

```
locateBorderGate(xaxis, east, 2, 1).
locateBorderGate(xaxis, west, 0, 1).
locateBorderGate(yaxis, north, 1, 2).
locateBorderGate(yaxis, south, 1, 0).

toggleSwitch(outside, north, 1, 2).
toggleSwitch(outside, east, 0, 1).
toggleSwitch(outside, south, 1, 0).
toggleSwitch(outside, west, 2, 1).

getGateOrthogonal(xaxis, south, south).
getGateOrthogonal(xaxis, north, north).
getGateOrthogonal(xaxis, northeast, north).
getGateOrthogonal(xaxis, northwest, north).
getGateOrthogonal(xaxis, southeast, south).
getGateOrthogonal(xaxis, southwest, south).

getGateOrthogonal(yaxis, east, east).
getGateOrthogonal(yaxis, west, west).
getGateOrthogonal(yaxis, southeast, east).
getGateOrthogonal(yaxis, southwest, west).
getGateOrthogonal(yaxis, northeast, east).
getGateOrthogonal(yaxis, northwest, west).
```

```
getAgentSide(east, right, north).
getAgentSide(east, left, south).
getAgentSide(west, right, south).
getAgentSide(west, left, north).

getAgentSide(north, right, east).
getAgentSide(north, left, west).
getAgentSide(south, right, west).
getAgentSide(south, left, east).

+!lookForSwitch :
startLookingForSwitch(F, Fx, Fy, GD, AD)[source(A)]
& switchFound(F, Fx, Fy, S, X, Y)
<-
.print("Ein Switch wurde (von einem anderen Agenten) gefunden");
-switchFound(F, Fx, Fy, S, X, Y);
-startLookingForSwitch(F, Fx, Fy, GD, AD)[source(A)].



+!lookForSwitch :
startLookingForSwitch(F, Fx, Fy, GD, AD)[source(A)]
& ison(S,X,Y,T) & switch(S)
& gate(X,Y ,S,_) & gate(Fx,Fy ,SW,_)
<-
.print("sehe Switch");
.send(A, tell, switchDetected(S, X, Y, T));
-startLookingForSwitch(F, Fx, Fy, GD, AD)[source(A)].



+!lookForSwitch :
startLookingForSwitch(F, Fx, Fy, GD, AD)[source(A)]
& fencegroup(X, Y, Axis)  & fencegroup(Fx, Fy, Axis)
& locateBorderGate(Axis, Direction, Xoff, Yoff)
& not (fencegroup(Xg, Yg, Axis)
& (Xg = X + (Xoff-1)) & (Yg = Y + (Yoff-1)))
& toggleSwitch(outside, Ortho, Xt, Yt)
& getGateOrthogonal(Axis, GD, Ortho)
& getAgentSide(Ortho, AD, Direction)
<-
  !setWalkingModi(0,0,0);
!walkTo((X + (Xt - 1)),(Y + (Yt - 1)), Direction).
```

Wenn ein Schalter zum Tor bekannt ist, wird der nächststehende der beiden

äußersten Agenten zu diesem geschickt.

Sobald dieser das Tor geöffnet hat, laufen die übrigen Agenten zum nächsten Wegpunkt.

Wenn alle Agenten den Zaun passiert haben, wird der Schalter von innen betätigt und der immer noch außenstehende Agent kann zur Gruppe aufschließen.

Kapitel 8

Bedienungsanleitung

A.Löwen

8.1 Vorwort

Diese Bedienungsanleitung hilft Ihnen unseres Mutli-Agenten-System zu installieren, konfigurieren und auszuführen. Sie werden dabei auf unterschiedliche Technologien stoßen, welche bei einem komplexen System zusammenarbeiten. Eine Vielzahl an Konfigurationsoptionen ermöglicht es Ihnen das System optimal euren Wünschen anzupassen.

8.2 Benötigte Komponenten

Folgende Komponenten werden für dieses Multi-Agenten-System benötigt.

8.2.1 Java

Unseres System basiert auf Java und ist somit plattformunabhängig. Für die Ausführung wird eine aktuelle Java-Laufzeitumgebung benötigt. Die Umgebungsvariable *JAVA_HOME* soll auf den Java-Installationspfad verweisen.

8.2.2 Massim

Eine aktuelle Version der Massim-Umgebung kann von der *multi-agent-contest* Seite heruntergeladen werden. Im Paket ist ein Simulationsserver, Beispiel-Agenten und ein Server-Monitor enthalten.

8.2.3 DLV

Im Projekt wird ein *ASP-Solver* benutzt - DLV. Dieses Programm kann von der Entwickler-Homepage heruntergeladen werden.

8.2.4 Jason

Unseres Multi-Agenten-System wird auf Jason-Plattform ausgeführt. Diese kann von der *sourceforge.net* Webseite bezogen werden.

8.2.5 Cowbot Multi Agent System

Die eigentliche Implementierung der Agenten ist im Cowbots-Projekt enthalten. Sie besteht aus zwei Teilen: *cowbot library* und das Test MAS Projekt. Diese können aus dem SVN der Projektgruppe ausgecheckt werden.

8.3 Installation und Konfiguration

Nachdem Ihre Java-Laufzeitumgebung installiert ist, alle anderen Komponenten heruntergeladen und falls nötig entpackt sind, können Sie mit der Konfiguration des Projektes beginnen.

8.3.1 mas2j

Die Struktur des Multi-Agenten-Systems wird in einem *system definition file* festgelegt. Diese Projektdatei hat die Endung „.mas2j“. Man kann dabei nicht nur die Anzahl der Agenten mit unterschiedlichen Typen, sondern auch Ausführungsmodi etc. für den Interpreter angeben. So kann man z.B. die Agenten synchron oder asynchron ausführen, im Netzwerk mit Hilfe von einer „host“-Angabe verteilen. Mit Hilfe der mas2j-Datei wird das System definiert und auf der Jason-Plattform gestartet.

8.3.2 agent.xml

In der Datei agents.xml können Agenten über die Angaben in der mas2j Datei hinaus feiner konfiguriert werden. Hier wird insbesonders festgelegt, welche logischen Programme ein Agent initial als Wissen oder für spezialisierte Komponenten wie die Desire-Generierung lädt. Die Datei hat folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<agents>

  <agent default="yes">
    <belief src="FILENAME" />
    <belief src="FILENAME2" norevise="true" />

    <program name="PRGNAME" src="FILENAME" />
  </agent>

  <agent name="alice">
  ...
```

```
  </agent>

</agents>
```

- In einem *agent tag* können erweiterte logische Programme angegeben werden, die ein Agent wärend seiner Initialisierungsphase lädt. Es kann eine namenslose Standard-Konfiguration geben. Diese wird jedem in einer mas2j Datei benannten Agenten zugewiesen, der keine eigene Konfiguration in der agents.xml besitzt.

- Program-Tag deklariert Programme, die ein Agent lediglich lädt. Diese können über das IEpistemicState Interface mit *getProgram* abgerufen werden.

- PRGNAME ist ein eindeutiger Name unter dem das Programm intern gespeichert wird.

- Besitzt ein *agent tag* ein name-Attribut, und exisitiert so ein Agent in dem über die mas2j Datei definierten Agentensystem, werden die Programme aus dem benannten Block, und nicht die aus dem Default-Block, geladen.

- Mit Hilfe von *belief tag* werden Programme angegeben, die ein Agent initial in seine Belief Base lädt. Dabei beschreibt *src* den Dateinamen des logischen Programms, das ein Agent laden soll. Mit *norevise=“true“* kann man optional das angegebene Programm nach der *belief state construction* dem *belief state* hinzufügen. Hierbei muss man zwingend aufpassen, dass keine Konflikte erzeugt werden.

Folgende Konfiguration erfüllt, aus Sicht der Revision, eine Mindestkonfiguration an die agents.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<agents>
  <agent default="yes">
    <!-- belief-tag is used to give an agent initial knowledge -->
      <belief src="bel_map.elp" />
      <belief src="bel_fence.elp"/>
      <belief src="bel_fenceorientation.elp"/>
      <belief src="bel_rejectedall.elp"/>
  </agent>
</agents>
```

Standardmässig lädt jeder Agent die agents.xml. In der mas2j Datei kann man jedoch für jeden Agenten einzeln mittels des Usersettings *agcfg=“’eine.xml.datei“’* eine Konfigurationsdatei angegeben werden.

8.3.3 local_config.xml

In dieser Konfigurationsdatei werden lokale Dateisystempfade der Komponenten eingetragen. (DLV, Massim, usw.) Ausserdem werden hier die Server, auf denen das System läuft, festegelegt.

8.3.4 brf_cfg.xml

Das ist eine Konfigurationsdatei für Feintuning der Revisionsfunktion. Hier wird festgelegt, über welche Wissensoperatoren ein Agent verfügen kann. Von besonderer Interesse ist hier der UpdatePolicy Bereich. Im *norewrite tag* können per Komma getrennte Prädikate angegeben werden, die nicht transformiert in die Jason-BB übertragen werden(die Standard-Transformation entfernt den ersten Term und fügt diesen als Source Annotation einem Jason-Literal hinzu). Eine *brf_cfg.xml* hat folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<brf>
  <knowledgeoperator name="KNAME" class="CNAME"> +

  <inferenceoperator name="INAME" class="UNAME"> +

  <updatepolicy name="UNAME">
    <operator name="KNAME">
     <trigger>functor-Liste</trigger>
     <config />
    </operator> +
    <norewrite>functor-Liste</norewrite>
  </updatepolicy> +

  <config name="CNAME">
    <update policy="UNAME"/>
    <revision operator="INAME"/>
  </config> +
</brf>
```

Mit *knowledgeoperator* und *inferenceoperator* werden eindeutig mittels $KNAME$ oder $INAME$ benannte Wissensoperatoren deklariert. Das Feld *class* muss hierbei ein gültiger, dem Java *class loader* bekannter Klassenname sein. In der *update policy* wird festgelegt, welche Operatoren zur Modifikation der *belief base* geladen werden sollen. Die Felder *trigger* bestimmen hierbei disjunkte Alphabete über Literale, auf denen die Operatoren arbeiten dürfen. Anschliessend wird mit dem Feld *config* festgelegt, welche *update policy* und welchen Revisionsoperator ein Agent benutzt.

8.3.5 Motives

Im Ordner Motives befinden sich Textdateien mit der *.motives* Erweiterung. Diese beschreiben Motive der Agenten und werden bei der Erzeugung der *desires* genutzt.

8.3.6 logging.properties

In dieser Datei werden Einstellungen für den *Logger* gespeichert.

8.4 Starten des Systems

Unseres System kann je nach Konfiguration auf unterschiedliche Arten gestartet werden. Für einfacheres Testen auf verschiedenen Rechnern ist ein kleiner MAS Starter im test_mas Projekt vorhanden. Startet man die MASCowbots.java, erscheint eine Auswahl aller verfügbaren mas2j Projekte, und ein paar Knöpfe auf welchem Rechner diese ausgeführt werden sollen. Man kann das MAS auch von der Konsole starten. Hierzu muss eine mas2j Datei angegeben werden. Ausserdem kann man optional noch den Rechner auf dem Massim ausgeführt wird angeben.

- Windows:

```
java -cp jason.jar;cowbots.jar;launcher.jar MASCowbots
 <mas2j-Datei> <massim-host>
```

- Unix:

```
java -cp jason.jar:cowbots.jar:launcher.jar MASCowbots
 <mas2j-Datei> <massim-host>
```

Kapitel 9

Anpassungen für den Contest

D. Hoelzgen, T. Vengels

Im Rahmen der Teilnahme an dem von der TU Clausthal veranstalteten Wettbewerb für Multiagentenprogrammierung [19] wurden, vor allem in Hinblick auf das Einhalten der Geschwindigkeitsvorgaben, einige Änderungen am Framework sowie dem Multiagentensystem vorgenommen, welche in diesem Kapitel vorgestellt werden.

Zum Schluss dieses Kapitels sei zudem auf einige Probleme und mögliche Verbesserungen eingegangen, welche sich im Verlauf des Wettbewerbs aus den Diskussionen ergeben hatten.

9.1 Änderungen am MAS

Die Implementierung der Rollen gemäß der erstellten Strategie in Form von logischen Programmen, AgentSpeak Code sowie k-Plans zeigte sich in der Praxis unter Wettbewerbsbedingungen als zu schwerfällig. Die dynamische Gruppenbildung und die damit verbundene Absprache kostete viel Zeit, insbesondere die Entscheidung, nur einen einzigen Agententypen zu entwickeln, welcher alle konzipierten Rollen implementiert, erzeugte sehr viel Rechenaufwand. Somit wurden zunächst mehrere Agententypen erstellt, von denen nur die Leader alle notwendigen Berechnungen für die *Desire Generation* und dynamische Planung durchführten. Die dadurch zu einem großen Teil wegfallende Absprache zur Gruppenbildung hat das System zusätzlich beschleunigt, die Agenten waren schneller in der Lage, auf die aktuelle Situation zu reagieren.

Als ein weiteres Problem unter Wettbewerbsbedingungen stellten sich die teils komplexen, aber starren Strategien heraus. Im Laufe der Konzeption und Entwicklung des Frameworks wurden seitens des Wettbewerbsveranstalters die Geschwindigkeiten der Kühe im Szenario verdreifacht. Dadurch wurde es not-

wendig, das Handeln viel öfter an die sich schnell ändernde Situation anzupassen, zudem wurden bestimmte Vorgehensweisen zu riskant - es war beispielsweise nicht mehr ohne weiteres möglich, mit den Kühen zusammen vor einem Tor zu warten, bis dieses geöffnet wurde. Aus diesem Grund wurden auch die Strategien angepasst:

- Komplexe Manöver, welche ohnehin nicht in der Form durchgeführt werden konnten, wurden aufgeteilt oder durch einfachere Aktionen ersetzt. Auf diese Weise konnte schnell der nächste Schritt entschieden werden, und Fehlschläge erforderten weniger Kommunikation zur Abbruchbehandlung.
- Das synchronisierte Vorgehen wurde größtenteils aufgehoben. Da die Kühe sich nun genau so schnell bewegen konnten wie die Agenten, hat es sich in zahlreichen Testläufen als sinnvoll herausgestellt Aktionen, sobald möglich, sofort auszuführen. Auch dies verminderte die Kommunikation, die sonst zur Absprache notwendig gewesen war. Als positiver Nebeneffekt mussten die Agenten weniger oft auf andere Gruppenmitglieder warten.

Gemäß der stark vereinfachten Strategien wurden zudem nur die direkt folgenden Schritte konkret geplant, da eine vollständige Planung des Vorgehens aufgrund der sich schnell ändernden Situation ohnehin nicht zur Anwendung gekommen wäre.

9.2 Änderungen am Framework

Zunächst wurden an der *Belief Revision* Änderungen bezüglich Geschwindigkeitsoptimierungen vorgenommen. Auf Basis der in Abschnitt 7.3.3 vorgestellten Klassen `CowbotRevision` und `IKnowledgeOperator` wurden performantere, hauptsächlich java-lastige Algorithmen entwickelt, die äquivalente Ergebnisse zu den wichtigsten Wissensoperationen auf ELP-Basis ermöglichen. Problem war im Wesentlichen, dass es nicht gelang vernünftige Laufzeiten von DLV bei 20 Agenten auf einem System zu Stande zu bekommen. Die Laufzeit pro DLV Aufruf schwankte zwischen 1,5 und 3 Sekunden (System: Dual-Quadcore XEON 16GB RAM), was bezüglich eines Rundentakts von zwei Sekunden als Risikofaktor für ausreichend schnell handelnde Agenten angesehen wurde. Allgemein wurden die Agenten so angepasst, das zumindest die einfachen Regeln bezüglich der Revision der Kachelwelt in Java realisiert wird.

- `exRevision` Auf Basis der `CowbotRevision` abgeleitete Klasse. Hier wurde der Schritt der *belief state construction* und anschliessender Berechnung des *belief sets* geändert. Die Java-basierte Version aktualisiert lediglich die Zeit, Agenten- und Kuhpositionen. Die restlichen, für die Contest-Version relevanten Wissensaktualisierungen wurden auf angepasste Wissensänderungsoperatoren verteilt.
- `exMapOperator` Wissensänderungsoperator speziell für Kartenwissen, basierend auf den in Abschnitt 7.2 vorgestellen Szenario. Der Wissensopera-

tor inspiziert eingehende *ison*-Prädikate mit Hilfe der `IdentTable` Schnittstelle, und speichert lediglich Agenten- und Kuhpositionen in der Wissensbasis. Andere Kachelinformationen werden direkt mit Hilfe einer spatialen Datenstruktur gemäss dem aktuellsten Zeitstempel erneuert. Die Erkennung von *gate*-Strukturen wird mittels eines einfachen Abtastung über alle Schalter und Zaunzellen realisiert. Zaunzellen, die hierbei nicht als *gate* gruppiert werden konnten, werden in einem zweiten Schritt mittels Abtastung zu $fencegroup$ zusammengefasst.

- `exKnowledge` Wissensänderungsoperator für die Aktualisierung der Agentenpositionen. Jeder Agent übermittelt pro Welttakt seine Position mit Hilfe einer $agentposition_{msg}$, die jeweils die aktuellste Position wird als *agentposition* per *belief set postprocessing* dem Wissen des Agenten zur Verfügung gestellt.

Neben den umfangreichen Änderungen an der *Belief Revision* mussten auch andere Komponenten beschleunigt werden. Die *Desire Generation* erhielt durch Java Operationen unterstützte Implementierungen der in Abschnitt 7.3.4 vorgestellten abstrakten Klasse *CDesire*. Diese führte komplexere Berechnungen im Vorhinein aus, so dass die ELPs unter Beachtung des zusätzlich generierten Wissens ausgewertet werden konnten. Da auch dies auf großen Karten und vielen Agenten nicht den Anforderungen des geringen Rundentakts gerecht werden konnte, wurden weitere Funktionalitäten, die zuvor durch logische Programme gestellt wurden, durch Java-Äquivalente übernommen.

Die letzte Komponente, welche umfangreiche Änderungen erfahren hat, ist der dynamische *Planner*. Mangelnde Geschwindigkeit und eine hohe Komplexität beim Anpassen der vorhandenen Pläne machten auch hier eine äquivalente Komponente notwendig, die auf Java-Ebene direkt auf einer neuen Kartendarstellung und mit speziell dafür entwickelter Pfadberechnung Aufgaben entsprechend der aktuellen Situation des Agenten und dessen Zielen erstellen und zuweisen kann. Ergebnis dieser Komponente ist, wie bei der ursprünglichen dynamischen Planung auch, ein vom Leader der Gruppe genutzter Plan zur Aufgabenübermittlung an seine Teammitglieder.

9.3 Gesammelte Erfahrungen

Die Teilnahme am Wettbewerb motivierte viele weiterführende Diskussionen über Probleme und Verbesserungsmöglichkeiten in unserem System. Das Spielen gegen andere Teams, welche zum Teil vollkommen andere Strategien als das unsere verfolgten, stellte sich als sehr gute Möglichkeit heraus, das eigene Multiagentensystem in einer unbekannten Situation zu erleben. Im Folgenden seien nun drei allgemeine Ansatzpunkte für Verbesserungen vorgestellt, welche oft Gegenstand der Diskussionen waren.

Hier ist zunächst eine gute Koordination der Gruppen zu nennen. In der Regel reicht es nicht aus, wenn die einzelnen Gruppen verschiedene Ziele haben.

Vielmehr muss hier auf Mögliche unbeabsichtigte Auswirkungen auf die Aktionen anderer Gruppen geachtet werden. So kann eine Gruppe, welche nach dem erfolgreichen Eintreiben von Kühen zu ihrer nächsten Position läuft, eine andere Gruppe extrem beeinträchtigen, wenn sie die von ihr getriebene Herde auf dem Weg zum Ziel durchkreuzt und so auseinander treibt. Die Beobachtungen vieler Spiele haben hierbei gezeigt, dass es oft nicht möglich ist, dieses Problem durch die Wahl eines anderen Ziels oder durch Abwarten zu umgehen. Vielmehr sollten die Agenten geschickt so positioniert werde, dass sie die höher priorisierte Gruppe durch ihre Position schlechtestenfalls nicht beeinträchtigen, bestenfalls jedoch sogar unterstützen. So könnte eine Gruppe, welche einen Zaun passieren möchte, durch den gerade von einer anderen Gruppe Kühe getrieben werden, ihre Position derart ändern, dass die Kühe nach Passieren des Zauns in die gewünschte Richtung gelenkt werden.

Ein weiteres Problem war die Beachtung von zu wenigen Faktoren bei der Bewertung von Situationen. So wurde beispielsweise zur Bewertung, welche Kühe als nächstes in den eigenen Pferch getrieben werden sollten, nur die eigene Position, die Position der Kühe und die des Pferches betrachtet. Auf diese Weise wurden unsere Teams jedoch nahezu lahmgelegt, wenn der Gegner stetig Kühe aus dem eigenen Pferch getrieben hatte. Ein Unterscheiden der Situation, dass die Agenten selbst gerade eine Kuh auf den letzten Schritten in den eigenen Pferch treiben, oder eine Kuh durch die Aktionen des Gegners aus dem Pferch entwicht, hätte hier eine sinnvolle Aufgabenzuweisung vereinfachen können.

Zuletzt sei ein Problem genannt, welches oft Gegenstand von Diskussionen war, welche nicht in einem konkreten Lösungsansatz endeten: Das Nichterkennen von negativen Begleiterscheinungen bei eigentlich erfolgreicher Planausführung. Sowohl die Karten als auch die Strategie des gegnerischen Teams waren unbekannt. Auf diese Weise kam es immer wieder vor, dass eine eigentlich erfolgreiche Planausführung negative Begleiteffekte mit sich brachte. So wurden Kühe durch den gegnerischen Pferch getrieben, gegnerische Teams in ihren Bemühungen indirekt unterstützt wurden und ein Weg durch den eigenen Pferch gewählt. Unbeachtet dieser konkreten Beispiele geht es bei diesem Problem nicht um ein Anpassen der Strategie im Vorhinein, sondern ein Erkennen von unbeabsichtigten, negativen Auswirkungen während der Ausführung. Die Diskussionen um Möglichkeiten, die Ursache für die erkannten Auswirkungen sicher zu identifizieren und das Vorgehen bei der Suche nach Alternativen zeigten gut die Komplexität dieses Themengebiets und bieten Ansatzpunkte für weitere Auseinandersetzungen mit diesem Thema.

Kapitel 10

Erfahrungsbericht

D. Hoelzgen

10.1 Erfahrungsbericht

10.1.1 Seminarphase

Noch vor dem eigentlichen Beginn im Wintersemester wurde im Rahmen von Seminarvorträgen das zur Arbeit in der Projektgruppe vorausgesetzte Wissen vorgestellt und gemeinsam erarbeitet. Die Gruppe entschied sich zu diesem Zweck zwei gemeinsame Tage im Universitätskolleg Bommerholz zu verbringen. Gegenüber von mehreren Treffen in den Räumen der Universität bot dies zum einen den Vorteil einer ungestörten Atmosphäre für das Seminar, zum anderen bot sich in Pausen zwischen den Vorträgen sowie beim gemeinsamen Besuch der Bar des Kollegs am Abend genügend Zeit und Raum, um ein näheres Kennenlernen der Teilnehmer untereinander zu ermöglichen.

Die von den Teilnehmern der Projektgruppe vorbereitete Themen waren:

- Agenten und das BDI-Modell
- Methoden der Multiagentenprogrammierung
- Agentenorientierte Programmierung
- Logische Programmierung
- Kommunikation und Kooperation
- Planen
- Wissensdynamik in Multiagentensystemen
- AgentSpeak

- Jason

Ursprünglich sollte in einem weiteren Vortrag das Thema Multiagentensysteme im Allgemeinen betrachtet werden. Dieser Vortrag entfiel jedoch, da der Vortragende die Projektgruppe vor Beginn verlassen hatte. Zusätzlich zu den von den Teilnehmern vorbereiteten Themen hielten auch die Betreuer der Projektgruppe Vorträge. Die Themen dieser Vorträge waren:

- Wissensrepräsentation, Inferenz und Wissensdynamik
- Aktionen, Beobachtungen und Wissensdynamik
- Umgebungen für Multiagentensysteme

Jeder Vortrag dauerte in der Regel 45 Minuten, so dass anschließend genug Zeit blieb, um das Thema sowie dessen Relevanz für die Arbeit der Projektgruppe zu diskutieren. Zudem wurde zu jedem Vortrag eine Ausarbeitung bereitgestellt, welche eine tiefere Auseinandersetzung mit dem jeweiligen Thema ermöglichte.

10.1.2 Wintersemester

Tutorium

Um zum einen den Inhalt des ausgefallenen Vortrags zu Multiagentensystemen nachzuholen, und um zum anderen Erfahrungen im praktischen Umgang mit logischer Programmierung sowie dem Agentspeak Interpreter Jason zu erlangen, wurde zunächst mit einer dreiwöchigen Tutoriumsphase begonnen. Durch das gemeinsame Erarbeiten der Themen sowie dem Lösen ausgesuchter Aufgaben sollte so das notwendige und in der Seminarphase aufgebaute Wissen vervollständigt werden, welches für eine geordnete Konzeption und anschließende Implementierung eines Multiagentensystems notwendig war.

Konzeptionsphase

Die Hauptaufgabe im Wintersemester bestand in der Konzeption des Multiagentensystems sowie eines Frameworks zu dessen Ausführung. Ausgehend von dem in der Seminarphase vorgestellten BDI Modell wurden Gruppen gebildet, welche sich mit den einzelnen Aspekten des Modells beschäftigten und Möglichkeiten zu deren konkreter Konzeption und Erweiterungen finden sollten. Die drei gebildeten Gruppen beschäftigten sich zunächst mit den folgenden Themen:

- Belief Revision
- Desire Generation
- Deliberation und Dynamische Planung

Die Gruppenergebnisse wurden einander einmal wöchentlich vorgestellt, zudem wurde gemeinsam ein Konzept für das Multiagentensystem, insbesondere in Hinblick auf gemeinsame Planung und Interaktion der Agenten untereinander, entwickelt. Beim Zusammenfügen der Gruppenergebnisse traten jedoch durch die gleichbleibend beibehaltenen Gruppen und die sich dadurch gebildete, auf nur einen Aspekt des Modells fokussierte Expertise, Probleme auf. Daraufhin wurde beschlossen, die Zahl der gemeinsamen Treffen zu erhöhen, zudem wurde ein Projektleiter gewählt, um die Übersicht über die Fortschritte der verschiedenen Gruppen zu behalten.

Testspiele

Da sich die Gruppe schon früh entschieden hatte, als Test der Implementierung an einem Wettbewerb zur Multiagentenprogrammierung der TU Clausthal [19] teilzunehmen, wurde neben der Arbeit am Konzept ein Java Programm entwickelt, mit dem es einem Anwender möglich war die Kontrolle über einen Agenten innerhalb der vom Veranstalter des Wettbewerbs zur Verfügung gestellten Simulation zu übernehmen. Auf diese Weise wurden in Form von Testspielen erste Erfahrungen in dem Szenario gesammelt, welche später als Grundlage für die Ausarbeitung von Strategien dienen sollten.

10.1.3 Sommersemester

Implementierung des Frameworks

Das im Wintersemester erstellte Konzept sollte im Sommersemester nun in Form eines Frameworks zur Ausführung von Multiagentensystemen implementiert werden. Nach einer Planung des Frameworks sowie der Schnittstellen zwischen den einzelnen Komponenten wurden diese von den selben Gruppen implementiert, welche diese im Semester zuvor konzipiert hatten. Leider stellte sich heraus, dass es Unstimmigkeiten zwischen unserem Konzept und den Möglichkeiten, diese in dem Agentspeak Interpreter Jason, welchen wir als Grundlage für unser Framework nutzen, zu integrieren. Weitere Anpassungen am System wurden notwendig, wodurch sich die Entwicklung des Frameworks länger als geplant hinauszögerte, und somit auch die Entwicklung des eigentlichen Multiagentensystems auf Grundlage dieses Frameworks nach hinten verschob.

Entwicklung des Multiagentensystems

Bei den anfänglichen Überlegungen zu grundlegenden Strategien sowie zur Wissensrepräsentation im Multiagentensystem wurde schnell klar, dass eine Aufteilung auf die gewohnten Gruppen keine zufriedenstellenden Ergebnisse liefern würde. Aus diesem Grund wurden für diese Arbeiten die bestehenden Gruppen gemischt, um möglichst Kenntnisse aus allen Bereichen mit in die Überlegungen einfließen lassen zu können. Anschließend konnte ausgehend von einer Konzeption des Multiagentensystems in Gaia damit begonnen werden, die einzelnen Rollen in unserem Framework zu implementieren.

Multiagent Contest

Während der Entwicklung des Systems zeichnete sich immer deutlicher ab, dass das resultierende System, wenn es vollständig gemäß der Konzeption implementiert werden würde, die eng gesteckten Zeitschranken für den Wettbewerb nicht einhalten können wird. Aus diesem Grund wurde eine weitere Entwicklung abgezweigt, welche bestehende Komponenten, unabhängig von der Konzeption durch speziell auf dieses Szenario angepasste, schnellere Komponenten ersetzen sollte. Zudem wurden die Strategien vereinfacht, um den sich sehr schnell ändernden Situationen im Szenario gerecht werden zu können.

Den vier Tage andauernden Wettbewerb verließen wir auf dem fünften von acht Plätzen. Angesichts der Tatsache, dass unsere Gruppe eines der wenigen Teams war, welche zum ersten Mal an diesem Wettbewerb teilgenommen hatten, ist dieses Ergebnis für uns sehr zufriedenstellend. Die Teilnahme stellte sich als sehr spannende Möglichkeit heraus, das entwickelte Multiagentensystem in unbekannten Situationen zu erleben, und bot der Gruppe so eine starke zusätzliche Motivation, über mögliche Probleme und Verbesserungen ausgiebig zu diskutieren.

10.2 Organisation der PG

Zusätzlich zu der oben erwähnten Gruppenarbeit wurden zur Organisation weitere Abmachungen getroffen und Werkzeuge genutzt, welche im Folgenden kurz vorgestellt sein.

10.2.1 Projektleiter

Um den Überblick über die Arbeit in den einzelnen Gruppen nicht zu verlieren, wurde von der Gruppe ein Projektleiter bestimmt. Dieser hatte die Aufgabe, die Handlungen der einzelnen Gruppen zu koordinieren und unbehandelte Probleme frühzeitig zu erkennen. Er war zudem damit beauftragt, das Einhalten des Zeitplans sowie die damit verbundene Aufgabenverteilung zu überwachen und fungierte als Ansprechpartner für die Betreuer der Projektgruppe.

10.2.2 Gemeinsame Sitzungen

In gemeinsamen Plenarsitzungen mit den Betreuern der Projektgruppe wurden aktuelle Entwicklungen und Ergebnisse vorgestellt und diskutiert. In den Diskussionen mit allen an der Projektgruppe beteiligten Personen sollten auftretende Probleme möglichst frühzeitig erkannt und behoben werden. Zudem sollte allen Mitgliedern der Projektgruppe die Möglichkeit gegeben werden, ihr Wissen und ihre Ideen in die aktuelle Entwicklung von anderen Gruppen mit einfließen zu lassen. in diesen Sitzungen wurde zudem das weitere Vorgehen beschlossen und mit den Betreuern abgestimmt.

Zusätzlich zu den Plenarsitzungen wurden je nach Bedarf zusätzliche Sitzungen veranstaltet, in denen die Studenten Zwischenstände der Gruppen abglei-

chen konnten, was insbesondere dann hilfreich war, wenn neue Entwicklungen für die weitere Arbeit vorgestellt werden mussten.

10.2.3 Wiki

Um die während den Sitzungen verfassten Protokolle sowie weitere Informationen, Anleitungen und gemeinsam erarbeitete Ergebnisse schnell und einfach festhalten zu können, wurde ein Wiki eingesetzt. Jedes Mitglied der Projektgruppe konnte hier an gemeinsamen Dokumenten mitarbeiten, zudem wurden Seiten zur Übersicht über die Implementierung mit Hinweisen zur Nutzung der neu erstellten Komponenten eingerichtet.

10.2.4 SVN

Das für die gemeinsame Arbeit an der Implementierung ohnehin notwendige SVN wurde neben seiner Hauptfunktion auch für die gemeinsame Arbeit am Zwischen- und Endbericht sowie zum Austausch von während der Arbeit erstellten Präsentationen und Ausarbeitungen genutzt.

10.2.5 Tasktime

In Ermangelung eines möglichst einfachen Werkzeugs zur Aufgabenverwaltung, welches das Zuweisen von Aufgaben an mehrere Arbeiter ermöglichen sollte, wurde eine Aufgabenverwaltung auf PHP Basis entwickelt. Diese wurde von der Projektgruppe benutzt, um einen schnellen Überblick über die Aufgaben, die damit verbundenen Ressourcen sowie die einzuhaltenden Deadlines zu erhalten, um so die Planung des weiteren Vorgehens zu unterstützen.

Kapitel 11

Fazit und Ausblick

S. Broszeit

Das von der PG im ersten PG Semester erarbeitete Konzept war in die Bereiche Belief Revision Function, Desire Generation und Deliberation gegliedert. Jeder dieser Bereiche beinhaltete gute Ansätze, wie beispielsweise Revision mit Hilfe von Causal Rejection in der BRF, Motivation in der Desire Generation und dynamisches sowie verteiltes Planen in der Deliberation.

Während der Realisierung dieser guten Ansätze mit dem Bestreben nach Autonomie der Agenten, Dynamik der Aufgabenverteilung und Gruppenzusammensetzung und Anpassbarkeit der Funktionalität konnten die einzelnen PG-Mitglieder Erfahrungen mit dem Framework Jason und der Sprache AgentSpeak sowie einen Gesamtüberblick über das Agentenkonzept sowie den Aufbau eines MAS gewinnen.

Für die Erprobung beim Multi Agent Programming Contest wurden einzelne Komponenten der Implementierung letztendlich durch simplere, leichter debugbare und - im Hinblick auf die Laufzeit - effizientere Alternativen ersetzt.

Der Erfahrungsgewinn der Implementierung hätte allerdings bereits früher erreicht werden können. Erhöhte Kommunikation zwischen den Gruppen des ersten Semester hätte den Überblick über das Agentenkonzept und im Zuge dessen dessen Zusammenspiel früher erreicht. Durch die Aufgliederung in drei Bereiche hatten sich dementsprechende Spezialistenteams gebildet, die ihren Bereich zwar gut überschauten, denen aber das Wissen fehlte, um die Schnittstellen entsprechend zu gestalten. Wissensaustausch hätte durch vermehrte Kommunikation zwischen den Gruppen oder Rotation der Gruppenmitglieder stattfinden können. Auch die Erfahrungen mit Jason und AgentSpeak hätten bereits früher gesammelt werden können. Das Tutorium zu Beginn des ersten Semesters gab zwar bereits einen guten Überblick über AgentSpeak. Aber die Komplexität, die für das letztendliche Agentensystem erforderlich war, war so nicht zu erfassen. Eine spielerische Annäherung mit einer naiven Implementierung in unserem Szenario, wäre eine mögliche Alternative, um sowohl technische als auch praktische Erfahrungen früher zu sammeln.

Kapitel 12

Danksagung

M. Kruse

Die PG-Teilnehmer möchten sich bei folgenden Institutionen, Programmierteams, Personen bedanken:

- der Technischen Universität Dortmund für die Räumlichkeiten,
- dem IRB für die Bereitstellung von Computerresourcen für den Contest,
- der TU-Clausthal für einen sehr guten Contest,
- den Jason Autoren Jomi F. Hübner und Rafael H. Bordini,
- den Eclipse Autoren für eine Java- Entwicklungsumgebung,
- unseren Betreuern Herrn M. Thimm, Herrn P. Krümplemann und
- unserer Professorin Frau G. Kern-Isberner.

Ohne die Beteiligung dieser Personen, Institutionen und Softwaresystemen wäre die PG in dieser Art und Weise nicht möglich gewesen.

Literaturverzeichnis

[1] Alchourròn, C. E., P. Gärdenfors und D. Makinson: *On the logic of theory change: Partial meet contraction and revision functions.* Journal of Symbolic Logic, 50:510–530, 1985.

[2] Austin, J. L.: *How to do things with words.* Harvard University Press, Cambridge, Mass., 1975.

[3] Behrens, T, J Dix, J Hï¿½bner und M Kï¿½ster: *Multi-Agent Programming Contest Protocol Description and Further Informations (2010 Edition)*, 2010. [Online; Stand 21. Januar 2010].

[4] Bordini, R. H., J. F. Hubner und M. Wooldridge: *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley, J, 2007.

[5] Dell'Armi, Tina, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone und Gerald Pfeifer: *Aggregate Functions in DLV.* In: *Theory and Practice of Logic Programming*, Seiten 545–580. Cambridge University Press, 2003.

[6] Dragoni, Aldo, Paolo Giorgini und Marco Baffetti: *Distributed belief revision vs. belief revision in a multi-agent environment: First results of a simulation experiment.* In: Boman, Magnus und Walter Van de Velde (Herausgeber): *Multi-Agent Rationality*, Band 1237 der Reihe *Lecture Notes in Computer Science*, Seiten 45–62. Springer Berlin / Heidelberg, 1997.

[7] Eiter, Thomas, Wolfgang Faber, Nicola Leone, Gerald Pfeifer und Axel Polleres: *Planning under incomplete knowledge.* Proceedings First International Conference on Computational Logic, Knwoledge Representation and Non-monotonic Reasoning Stream, 1861 in LNCS/LNAI:807–821, 2000.

[8] Eiter, Thomas, Wolfgang Faber, Nicola Leone, Gerald Pfeifer und Axel Polleres: *A Logic Programming Approach to Knowledge-State Planning, II: The DLV^K System.* Technischer Bericht, TU Wien, 2003.

[9] Eiter, Thomas, Michael Fink, Giuliana Sabbatini und Hans Tompits: *On properties of update sequences based on causal rejection.* Theory Pract. Log. Program., 2(6):711–767, 2002.

[10] Gelfond, Michael und Vladimir Lifschitz: *Classical Negation in Logic Programs and Disjunctive Databases.* New Generation Computing, 9:365–385, 1991.

[11] Ghallab, Malik, Dana Nau und Paolo Travers: *Automated Planning: Theory and Practice.* Morgan Kaufmann, 2004.

[12] javacc, 2010. [Online; Stand 2. September 2010].

[13] Katsuno, Hirofumi und Alberto Mendelzon: *On the difference between updating a knowledge base and revising it.* In: James F. Allen, Richard Fikes und Erik Sandewall (Herausgeber): *Principles of Knowledge Representation and Reasoning*, Seiten 387–394. Morgan Kaufmann, San Mateo, California, 1991.

[14] Kern-Isberner, Gabriele: *Vorlesung Wissensdynamik und Informationsfusion, Foliensatz AGM-Revisionstheorie*, 2007. [Folien zur Vorlesung "Wissensdynamik und Informationsfusion"].

[15] Kfir-Dahav, Noa E. und Moshe Tennenholtz: *Multi-agent belief revision.* In: *TARK '96: Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, Seiten 175–194, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[16] Liu, Wei und Mary-Anne Williams: *A Framework for Multi-Agent Belief Revision.* Studia Logica, 67(2):291–312, 2001.

[17] Marek, V. W. und Mirosaw Truszczynski: *Revision Programming.* THEORETICAL COMPUTER SCIENCE, 190, 1994.

[18] Meneguzzi, Felipe und Michael Luck: *Motivations as an Abstraction of Meta-level Reasoning.* In: *CEEMAS '07: Proceedings of the 5th international Central and Eastern European conference on Multi-Agent Systems and Applications V*, Seiten 204–214, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] multi-agent programming contest, 2010. [Online; Stand 27. August 2010].

[20] Pivkina, Inna, Enrico Pontelli und Tran Cao Son: *Revising Knowledge in Multi-agent Systems Using Revision Programming with Preferences.* In: *CLIMA IV*, Seiten 134–158, 2004.

[21] Searle, John R.: *Speech Acts.* Cambridge University Press, 1969.

[22] The DLV Project, 2009. [Online; Stand 25. September 2009].

[23] Tim Finin, Don McKay, Rich Fritzson: *An Overview of KQML: Knowledge QUery and Manipulation Language.* Technischer Bericht, KQML Advisory Group, 1992.

[24] Weiss, Gerhard: *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1999.

[25] Weiss, Gerhard (Herausgeber): *Multiagent systems: a modern approach to distributed artificial intelligence.* MIT Press, Cambridge, MA, USA, 1999.

[26] Wikipedia: *Planung — Wikipedia, Die freie EnzyklopÃ¤die*, 2009. [Online; Stand 15. September 2009].

[27] Wooldridge, M., N. R. Jennings und D. Kinny: *The Gaia Methodology for Agent-Oriented Analysis and Design.* Journal of Autonomous Agents and Multi-Agent Systems, Seiten 285–312, 2000.

Abbildungsverzeichnis

2.1 Der BDI-Prozess [25] 12
2.2 Überblick über das Cowbot BDI Modell 14
2.3 interner Prozessfluss der BRF- und *beliefs*-Komponenten 23
2.4 Schematischer Ablauf der BRF Funktions-Komponenten 25
2.5 Spezialfall Zellwiederherstellung 34
2.6 Einordnung der Motivation in das Cowbot BDI Modell 37

3.1 vier-Phasenmodell 62
3.2 Interaktion Broadcast Information 68
3.3 Interaktion Request Help 68
3.4 Interaktion Task Monitoring 69
3.5 Interaktion Global Monitoring 69
3.6 Rolle Cowbot 70
3.7 Rolle Driver 71
3.8 Rolle Scout 71
3.9 Rolle Leader 72
3.10 Rolle Door Opener 72
3.11 Agentenmodell, hier für 20 Agenten 73

6.1 Bewegen in einer Formation 95
6.2 Formation an Engstellen 96
6.3 Richtungsaenderung der Formation 96
6.4 Wahl der Wegpunktberechnung 97

7.1 Jason-Interpreter 110
7.2 MASSim Verbindungsprotokoll 115
7.3 Strukturdiagramm des Spieleclients 116
7.4 Bildschirmaufnahme Spieleclient 117
7.5 Strukturdiagramm des edu.udo.cs.ie.cowbots.logic Paketes 119
7.6 Strukturdiagramm des edu.udo.cs.ie.cowbots.logic.solver Paketes 119
7.7 Strukturdiagramm des edu.udo.cs.ie.cowbots.bdi.brf Paketes . . . 123
7.8 Strukturdiagramm des edu.udo.cs.ie.cowbots.bdi.desgen Pakets . 127
7.9 Beispielhaftes Motiv: drivencow 129
7.10 Strukturdiagramm des edu.udo.cs.ie.cowbots.arch Paketes 135

7.11 Strukturdiagramm des edu.udo.cs.ie.cowbots.ui Paketes 139
7.12 Bildschirmaufnahme der Cowbot-BDI Oberfläche 140
7.13 Skizze der Struktur des *scoutleaders* 141
7.14 Skizze der Struktur des *scouthelpers* 141
7.15 Ausschnitt aus dem Plan zur Aktualisierung des Wegpunktes . . 142
7.16 Ausschnitt aus dem den Planköpfen des *scouthelpers* 142
7.17 Planköpfe zur Reaktion der *gate*- und *fencegroup*-Literale 144