Universität Rostock

Traditio et Innovatio

Konzeption und Implementation eines Kontrollsystems für instrumentierte Umgebungen auf Basis von BDI-Agenten

eingereicht am 01.07.2011

von

Robin Nicolay | Kurt-Schumacher-Ring 52 | 18146 Rostock

Matrikel-Nr.: 5203974

Betreuer: Dr. rer. nat. Sebastian Bader

Gutachter:

Prof. Dr.-Ing. Thomas Kirste

Universität Rostock

Albert-Einstein-Str. 21

18059 Rostock

Zweitgutachter:

Prof. Dr.-Ing. Alke Martens

Pädagogische Hochschule Schwäbisch Gmünd

Oberbettringer Str. 200

73525 Schwäbisch Gmünd

Diplomarbeit

Institut für Elektrotechnik und Informatik | Diplom Informatik

Abstract

The paradigm of *BDI-Agents* introduced in [RG95] describes an agent consisting of a set of *Beliefs* as informative components, *Desires* representing the motivational state and *Intentions* capturing the deliberative components. *AgentSpeak(L)*, introduced in [Rao96], provides logical base to build these agents.

This thesis describes the design of a *BDI-Controller* (*BDI-SmartControl*) for instrumented environments using the *Jason-Framework* [BH06] and the „*helferlein*" middleware system [BRK11].

The control system handles entities of its environment as individual *BDI-Agents*. While functionality and behavior of these agents depend on their type, connections between entities are determined by abstract, local zones. The final behavior to support a user is modeled inside situational distributed person roles. The goal deliberation is based on environmental events and contexts. Distribution can be performed by addressing entities or types and is coordinated by priorities, goaltypes and costs. These techniques enable a dynamic typebased handling of devices and a direct development of user-supporting behaviours, which is dependent of specific situation and location. At the same time there is a very low dependency on behaviors to parameters of the smart environment.

Kurzfassung

Das in [RG95] vorgestellte Paradigma des BDI-Agenten beschreibt einen Agenten, bestehend aus einer Menge von *Beliefs* als informative Komponente, *Desires* als Repräsentation der Motivation und *Intentions* zur Kapselung deliberativer Bestandteile. *AgentSpeak(L)* (vorgestellt in [Rao96]) bietet eine logische Grundlage für die Entwicklung solcher Agenten.

Diese Diplomarbeit erläutert die Entwicklung eines Kontrollsystems für instrumentierte Umgebungen auf Basis von BDI-Agenten (*BDI-SmartControl*) unter Verwendung des *Jason*-Frameworks [BH06] und dem „*helferlein*"-Middlewaresystem [BRK11].

Das Kontrollsystem handhabt die *Entitäten* seiner Umgebung als individuelle BDI-Agenten. Während die Funktionalität und das Verhalten dieser Agenten durch ihren Typ bestimmt wird, werden Verbindungen zwischen *Entitäten* durch die Verwendung abstrakter, lokaler Zonen festgelegt. Das letztendliche Verhalten zur Unterstützung eines Nutzers wird in situationsabhängig verteilten Personenrollen modelliert. Die Deliberation des Systems geschieht durch die Auswertung von Ereignissen und Zuständen der Umgebung. Die Verteilung von Zielen kann durch die Adressierung einer *Entität* oder eines Typs geschehen und wird durch die Verwendung von Prioritäten, Zieltypen und Kosten koordiniert. Der Ansatz dieses Kontrollsystems ermöglicht eine dynamische Handhabung von Gerätemengen und eine direkte situations- und ortsabhängige Entwicklung unterstützender Verhaltensweisen. Gleichzeitig wird eine sehr geringe Abhängigkeit zu realen Parametern der *Intelligenten Umgebung* erreicht.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Verwandte Arbeiten . . . 3

2 Grundlagen **5**
2.1 Agentensysteme . . . 5
2.1.1 Agentenbegriff . . . 5
2.1.2 Abstrakte Agentenarchitektur . . . 6
2.2 BDI-Agenten . . . 8
2.2.1 Historie des BDI-Paradigmas . . . 8
2.2.2 Motivation des BDI-Paradigmas . . . 8
2.2.3 Vorstellung einzelner BDI-Komponenten . . . 10
2.2.4 AgentSpeak(L) . . . 11
2.3 Das Smart-Appliance-Lab der Universität-Rostock . . . 16
2.4 BDI-Frameworks . . . 18
2.4.1 Jack . . . 18
2.4.2 Jam . . . 20
2.4.3 Jadex . . . 21
2.4.4 Jason . . . 23
2.4.5 Sim_Agent . . . 24
2.5 Zusammenfassung . . . 27

3 Konzeption **28**
3.1 Nutzungsszenarien . . . 28
3.2 Annahmen . . . 30
3.2.1 Verhaltensannahmen . . . 30
3.2.2 Technische Annahmen . . . 31
3.3 Anforderungsanalyse . . . 32
3.3.1 Diskussion der Annahmen . . . 32
3.3.2 Konkrete Anforderungen für die Realisierung des Kontrollsystems . 34
3.4 Evaluation verfügbarer Middlewares . . . 37
3.4.1 Auswahlkriterien für die Untersuchung einer Middleware . . . 37
3.4.2 Untersuchung der Middleware-Konzepte . . . 37
3.4.3 Verwendung von Tuplespace, NetworkProxy und Infrastructure . . 39
3.5 Evaluation verfügbarer Frameworks . . . 40
3.5.1 Auswahlkriterien für die Untersuchung eines Frameworks . . . 40

3.5.2 Untersuchung der Frameworks . . . 40
3.5.3 Verwendung der Jason-Architektur . . . 42
3.6 Konzeptionsbeschreibung des Kontrollsystems . . . 45
3.6.1 Umgebungsschnittstellen . . . 45
3.6.2 Wissen in globalen Worldstates . . . 45
3.6.3 AgentSpeak in RegelMengen . . . 46
3.6.4 Geräte und Agenten als Entitäten . . . 46
3.6.5 Funktionalität durch EntitätsTypen . . . 47
3.6.6 Verhalten durch Rollen . . . 47
3.6.7 Erweiterung des Kontextes . . . 48
3.6.8 Ausführung im Agenten-Rohling . . . 49
3.6.9 Organisation von Agenten, Entitäten und EntitätsTypen . . . 50
3.6.10 Koordination von Verhaltensweisen und Ressourcen . . . 50
3.6.11 Verteilung von Zielen und Aufgaben . . . 53
3.6.12 Konzeptueller Aufbau des Kontrollsystems . . . 55
3.7 Zusammenfassung . . . 56

4 BDI-SmartControl - Umsetzung 57
4.1 Technische Umsetzung . . . 57
4.1.1 Informationsdarstellung . . . 57
4.1.2 Javabasierte Umgebungsschnittstellen . . . 58
4.1.3 Agenten, Entitäten, EntitätsTypen und ihre Organisation . . . 63
4.1.4 Agenten-Rohling, Koordination und Zielverteilung . . . 66
4.1.5 Erweiterung des Kontextes . . . 71
4.2 Richtlinie zur Erstellung neuer Agenten . . . 73
4.2.1 Hinzufügen eines Gerätes bekannten Typs . . . 73
4.2.2 Modellierung neuer Verhaltensweisen . . . 73
4.3 Realisierte Verhaltensweisen . . . 78
4.3.1 EntitätsTypen für Geräte . . . 78
4.3.2 EntitätsTypen zur Situationserkennung . . . 80
4.3.3 EntitätsTypen als Personenrollen . . . 81
4.4 Zusammenfassung . . . 82

5 Zusammenfassung und Ausblick 83
5.1 Zusammenfassung der Konzeption und Umsetzung . . . 83
5.2 Die Verwendung des BDI-Paradigmas . . . 84
5.3 Fähigkeiten, Probleme und weitere Entwicklungsziele . . . 85
5.3.1 Fähigkeiten . . . 85
5.3.2 Offene Probleme und weitere Entwicklungsziele . . . 87
5.4 Vergleich verwandter Arbeiten . . . 89

Literaturverzeichnis 91

Abbildungsverzeichnis

2.1 Der *Reasoningcycle* eines *AgentSpeak(L)*-Agenten [BBJ+02, S. 3]. 15
2.2 Das *Smart-Appliance-Lab* der Universität Rostock [MMI11]. 16
2.3 Teamstruktur und Teampläne einer Fußballmannschaft [LLJ08, S. 647]. . . 19
2.4 Eine Übersicht der *Jam*-Architektur nach [Hub99, S. 237]. 22
2.5 Der *Reasoningcycle* eines *Jason*-Agenten [BH07]. 26
2.6 Kapselung von Regelmengen in Regelfamilien. [MB01, S. 8] 26

3.1 Eine Übersicht der Evaluation betrachteter Middlewares. 38
3.2 Eine Übersicht der Evaluation betrachteter Frameworks. 42
3.3 Multi-Agenten Organisationskonzepte [Das08, S. 82]. 51
3.4 Zieltypen in BDI-Systemen nach [BPML04, S. 51] erweitert um *Jason*. . . 52

4.1 Abstrakte *Zonen* für Geräteabhängigkeiten und Personenverteilungen. . . 79

1 Einleitung

Das in „*BDI Agents: From Theory to Practice*" [RG95] vorgestellte Paradigma eines BDI-Systems definiert einen Agenten, bestehend aus einer Menge von *Beliefs, Desires und Intentions*. Während *Beliefs* dem Agenten als informative Komponente über den Systemzustand dienen, repräsentieren *Desires* seine Motivation durch aktive Ziele und Wünsche. *Intentions* beschreiben die tatsächliche Menge der Aktionen eines Agenten, um diese Ziele zu erreichen.

Das Ziel dieser Arbeit ist die Entwicklung eines Kontrollsystems für instrumentierte Umgebungen auf Basis des BDI-Paradigmas. Als Testumgebung und Referenz einer instrumentierten Umgebung wird *Smart-Appliance-Lab* der *Universität Rostock* (beschrieben in 2.3) zum Einsatz kommen. Das *Smart-Appliance-Lab* wird täglich für Forschung und Lehre verwendet und bietet einen realen Einsatzort für die Entwicklung eines Kontrollsystems. Darüber hinaus verfügt es über ein umfangreiches Sensoren- und Aktorenensemble und ermöglicht eine Betrachtung verschiedenster Aspekte zur Aufnahme von Informationen und Steuerung von Geräten. Die Nutzung des *Smart-Appliance-Labs* schafft eine weit gefächerte Menge von Anforderungen und Möglichkeiten und dient somit der folgenden Arbeit als Grundlage für die Konzeption des Kontrollsystems.

Im folgenden Kapitel werden die einzelnen Aspekte des vorgestellten Themas näher erläutert und ein Überblick über den Aufbau dieser Arbeit gegeben. Anschließend werden einige verwandte Arbeiten im Bereich des *Ubiquitous Computing* und *Intelligenter Umgebungen* beschrieben.

1.1 Motivation

Das Thema der folgenden Arbeit ist die Konzeption und Implementation eines Kontrollsystems für instrumentierte Umgebungen auf Basis von BDI-Agenten. Als Kontrollsystem wird hierbei ein System für die autonome Unterstützung von Nutzern einer *Intelligenten Umgebung* im Sinne des *Ubiquitous Computing* bezeichnet. Der Begriff *Ubiquitous Computing* wurde erstmals 1991 von Mark Weiser in seinem Werk „*The computer for the 21st century*" [Wei91] geprägt und kann mit folgenden Worten umschrieben werden:

> „Ubiquitous computing enhances computer use by making many computers available throughout the physical environment, while making them effectively invisible to the user." [Wei93, S. 1]

Für die Arbeitsweise eines Kontrollsystems für *Intelligente Umgebungen* bedeutet dies: Eine autonome Steuerung von Komponenten mit dem Ziel, die notwendige Menge an Interaktionen zwischen Nutzer und Umfeld auf ein Minimum zu reduzieren und gleichzeitig die Arbeit in einem Bereich mit unbekanntem Funktionsumfang zu unterstützen.

Der Funktionsumfang einer instrumentierten Umgebung ist maßgeblich durch die in ihr vorhandenen Geräte bestimmt. Der Begriff einer instrumentierten Umgebung wird von Lucia Terrenghi [Ter08, S. 1] als Raum mit eingebetteter Technologie zur Darstellung und Erfassung von Informationen beschrieben. Das für diese Arbeit entwickelte System wird im *Smart-Appliance-Lab* des Lehrstuhls für *Multimediale Informationssysteme* an der Universität Rostock (vorgestellt in 2.3) zum Einsatz kommen. Das *Smart-Appliance-Lab* ist eine Multidisplay-Umgebung und verfügt über eine Vielzahl von Sensoren und Aktoren zur Aufnahme und Darstellung von Informationen. Neben der technischen Seite des *Smart-Appliance-Labs* wird auch die Dynamik einer *Intelligenten Umgebung* betrachtet. Eine Beschreibung der Charakteristika dynamischer Umgebungen und eine einhergehende Motivation des BDI-Paradigmas für die Realisierung eines Kontrollsystems wird in Abschnitt 2.2.2 vorgestellt.

Das in dieser Arbeit entwickelte Kontrollsystem wird als Multiagentensystem realisiert. Eine umfangreiche Beschreibung des Paradigmas wird in [Woo09] erläutert. Eine Übersicht über aktuelle Forschungsgebiete und Ansätze für den Einsatz in *Intelligenten Umgebungen* liefert [Coo09]. Agenten besitzen die Fähigkeit autonomen und rationalen Handelns und bilden eine gute Grundlage für die Entwicklung selbständiger Systeme. Agentenbegriffe und ihre abstrakte Architektur werden in Abschnitt 2.1 dieser Arbeit beschrieben.

Eine spezielle Form des Agenten stellt der BDI-Agent dar. Ein BDI-Agent besitzt Zustände zur Speicherung des Umgebungswissens (*Beliefs*), Funktionen zur Generierung von Wünschen (*Desires*) und Methoden zur Auswahl und Ausführung von Aktionen (*Intentions*). Abschnitt 2.2 gibt einen Überblick über die Historie, Definitionen und Arbeitsweisen des BDI-Paradigmas. Aktuelle Frameworks zur Entwicklung von BDI-Systemen werden in 2.4 vorgestellt. Die Struktur dieser Arbeit gliedert sich wie folgt:

Der nächste Abschnitt dieses Kapitels gibt einen Überblick über verwandte Arbeiten in den Bereichen *Ubiquitous Computing* und *Intelligenten Umgebungen*. Kapitel 2 beschreibt die technischen und konzeptionellen Grundlagen dieser Arbeit. Zusätzlich wird ein Überblick über technische Hilfsmittel für die Entwicklung von BDI-Systemen gegeben. Das anschließende Kapitel 3 dient der Beschreibung der einzelnen Schritte zur Konzeption des Systems. Hierzu werden Nutzerszenarien im alltäglichen Umgang mit dem *Smart-Appliance-Lab* aufgeführt und abstrakte Annahmen für die Realisierung des erwarteten Umgebungsverhalten getroffen. Auf Basis dieser Annahmen werden anschließend konkrete Anforderungen für die Entwicklung eines BDI-basierten Kontrollsystems getroffen. Mithilfe dieser Kriterien werden verfügbare Frameworks und Middlewares evaluiert und konkrete Konzepte für die Realisierung diskutiert. Im anschließenden Kapitel 4 wird die Umsetzung des in dieser Arbeit entwickelten BDI-Kontrollsystems beschrieben. Hierbei werden sowohl die technische Realisierung als auch notwendige Schritte für eine Erweiterung und Konfiguration des Systems beleuchtet. Im letzten Kapitel werden die Erkenntnisse und Ergebnisse dieser Arbeit zusammengefasst. Hierzu werden Vor- und Nachteile des Systems und einige Konzepte für weiterführende Entwicklungen erläutert.

1.2 Verwandte Arbeiten

Für den Bereich der *Intelligenten Umgebung* existiert eine große Vielfalt an Forschungsbereichen. Der folgende Abschnitt gibt einen kleinen Überblick über unterschiedliche Ansätze und Forschungsgebiete im Bereich des *Ubiquitous Computing*.

Neuronale Netze zur Steuerung eines Wohnhauses Das Paradigma der *Neuronalen Netze* beschreibt künstliche Systeme, deren Arbeitsweise durch die Funktionsweise des menschlichen Gehirns inspiriert ist. Sie bestehen im Allgemeinen aus einer sehr großen Anzahl einfacher Auswertungseinheiten (Neuronen). Große Neuronenmengen sind in der Lage, durch Interaktionen über gewichtete Verbindungen komplexe Verhaltensweisen zu erzeugen. Eine Stärke *Neuronaler Netze* ist das Erlernen und die Anpassung von Verhaltensweisen anhand praktischer Beispiele.

Das *Adaptive House* [Moz05] beschreibt ein Projekt zur Verwendung *Neuronaler Netze* in *Intelligenten Umgebungen*. Ziel dieses Projekts ist die Entwicklung eines selbstständig durch Beobachtungen lernenden Kontrollsystems. Beobachtet werden Änderungen von Gerätezuständen durch Hausbewohner. Das Lernen findet über Verstärkung (*Reinforcement Learning* beschrieben in [SB98]) entsprechender Neuronenverbindungen statt und erstellt Modelle zur Voraussage zukünftiger Nutzeraktionen. Arbeiten die erlernten Modelle zuverlässig führt der *Controller* diese Aktionen selbstständig aus. Auf diese Weise werden Bewohner nach und nach von der manuellen Steuerung der Umgebung befreit. Ein Nebenziel des Systems ist die Minimierung des Energieverbrauchs durch eine intelligente Regulierung von Heiz- und Beleuchtungssystemen. Der aktuell entwickelte Prototyp des Hauses besitzt über 75 Sensoren und Aktoren.

Rein reaktive Systeme zur Steuerung Intelligenter Umgebungen Reaktive Systeme handeln ausschließlich auf Basis aktueller, lokaler Wahrnehmungen. Vorangegangene Umgebungszustände haben keinen Einfluss auf die Aktionsausführung. Eine spezielle Form des reaktiven Systems ist die 1985 von Rodney A. Brooks vorgestellte *Subsumption-Architektur* [Bro85]. Die *Subsumption-Architektur* spezifiziert eine hierarchische Koordination reaktiver Verhaltensweisen. Höhere Verhaltensebenen beinhalten komplexes Verhalten und sind in der Lage, das Verhalten niederer Ebenen zu unterdrücken oder zu überschreiben.

Ein reaktiver *SuBaSLC-Controller* [Dyr11] kommt aktuell im *Smart-Appliance-Lab* der Universität Rostock zum Einsatz. *SuBaSLC* besteht aus einer Menge von *Behaviours* (Verhaltensweisen), die abhängig von ihrem Zustand und eintreffenden Ereignissen gewünschte Zielzustände für die Geräte des Labors formulieren. Die Zielzustände sind zu Koordination des Verhaltens mit Prioritäten und Gewichten versehen. Eine zentrale Schaltkomponente erfragt in regelmäßigen Zeitabständen die aktuellen Ziele aller Verhaltensweisen, filtert diese nach der Priorität und sendet sie weiter an *Materialiser* (Geräteschnittstellen). Diese berechnen auf Basis von *Merging* (Vermischungen) und *Voting* (Auswahlverfahren) letztendliche Werte für Gerätezustände und setzen diese um. Geräte und *Behaviours* können dynamisch zur Laufzeit hinzugefügt und entfernt werden.

Agenten-Workspaces in Intelligenten Umgebungen Omicini beschreibt in „*Smart Environments as Agent Workspaces*" [ORV07] Konzepte zur Verwendung von Artefakten bei der Entwicklung eines Kontrollsystems für *Intelligente Umgebungen*. Artefakte werden typischerweise vom Entwickler eines Multiagentensystems zur Kapselung bestimmter Funktionen realisiert. Als „Werkzeuge" für Agenten können sie dynamisch von Agenten des Systems initialisiert und genutzt werden. Artefakte stellen eine Menge von Eigenschaften und Funktionen zur Verfügung und dienen unter anderem als transparente Schnittstelle zu realen Objekten der Umgebung. Darüber hinaus unterstützen sie das verteilte Arbeiten mehrerer Agenten an einer Aufgabe und ermöglichen die Verknüpfung von Informationen unterschiedlicher Quellen. Für die Entwicklung eines Kontrollsystems für *Intelligente Umgebungen* unterstützen sie die Bereitstellung von Sensorinformationen, Gerätezustände und die Ausführung von Geräteaktionen. Darüber hinaus können Artefakte zur Koordination von Agentenmengen verwendet werden. Ein Beispiel hierfür ist das in 2.4.4 beschriebene *MOISE*-Artefakt.

Artefakte zur Anreicherung einer realen Umgebung Ein etwas anderer Forschungsbereich beschäftigt sich mit der Anreicherung von Gegenständen der Umgebung. *Smart Artifacts* beschreiben eine Augmentation realer Objekte durch Informationen, Services und die Einbettung von Computertechnik.

Ambient Arogas [SSR+03] erhöht die funktionale Darstellung (Affordance) realer Objekte durch eine Zuordnung intelligenter Artefakte. Diese Artefakte bieten situationsbezogene Services und Informationen, deren Inhalte über eingebettete Anzeigen und mobile Geräte dargestellt werden. Das Entfernen von Computern aus der Wahrnehmung des Menschen geschieht zum einen durch eine Verkleinerung der verwendeten Hardware, zum anderen durch eine Transformation in alltägliche, mit Informationen und Funktionen angereicherte Objekte. Die Interaktion mit diesen Objekten wird durch eine Vielzahl von Sensoren unterstützt.

Eine beispielhafte Anwendung ist die Verbindung des Frauenhofer Instituts IPSI in Darmstadt mit der EDF-LDC in Paris. Hierbei werden unter Verwendung einer *Hello.Wall* Anwesenheiten und Stimmungen eines Ortes wahrgenommen und über Lichtpunkte auf *Hello.Walls* anderer Orte übertragen. Darüber hinaus bieten die *Walls* auch Möglichkeiten zum Austausch von Informationen oder dem Abhalten von Videokonferenzen. Ein persönliches Artefakt, das „*Personal Aura*", dient der Konfiguration der eigenen Wahrnehmung im System. Mit ihm kann unter anderem die Sichtbarkeit und die eigene soziale Rolle festgelegt werden.

2 Grundlagen

Das folgende Kapitel gibt einen Überblick über die Grundlagen und die verwendeten technischen Ansätze dieser Arbeit. Begonnen wird mit der Erläuterung der Agentenarchitektur. Anschließend wird der BDI-Agent und das mit ihm verbundene BDI-Paradigma vorgestellt. Zusätzlich werden das *Smart-Appliance-Lab* der Universität Rostock als Testumgebung und verfügbare Frameworks für die Entwicklung eines BDI-Systems beschrieben.

2.1 Agentensysteme

Agenten existieren in den unterschiedlichsten Ausprägungen und bilden die Grundlage für das in dieser Arbeit entwickelte System. Dieser Abschnitt liefert eine Beschreibung des Agentenbegriffs und der abstrakten Architektur eines Agenten.

2.1.1 Agentenbegriff

Wooldridge, Jennings und Nicholas beschreiben in [WJ95, 117-118] die Notwendigkeit einer eindeutigen Definition, um ein „Verrauschen" des Agentenbegriffs zu verhindern. Zu diesem Zweck definieren sie die folgenden Stufen eines Agentenbegriffs:

Schwacher Agentenbegriff Der schwache Agentenbegriff beschriebt ein Stück Hard- oder Software mit folgenden Eigenschaften:

- ▷ Autonomität: Agenten handeln selbstständig. Sie kontrollieren ihre Aktionen und interne Zustände.
- ▷ Soziale Fähigkeiten: Agenten interagieren mit anderen Agenten oder Menschen über eine Agenten-Kommunikationssprache.
- ▷ Reaktivität: Agenten nehmen ihre Umgebung und deren Änderungen wahr und reagieren auf sie.
- ▷ Proaktivität: Agenten sind in der Lage selbstständig und zielorientiert zu handeln.

Starker Agentenbegriff Der starke Agentenbegriff erweitert den schwachen Agentenbegriff um weitere auf die Menschlichkeit bezogene Konzepte wie: Wissen, Glauben, Absichten und emotionale Zustände. Der starke Agentenbegriff besitzt zusätzlich folgende Attribute:

- ▷ Mobilität: Agenten können sich in elektronischen Netzwerken bewegen.

- ▷ Aufrichtigkeit: Agenten werden nicht wissentlich falsche Informationen verbreiten.
- ▷ Wohlwollend: Agenten erzeugen keine Konflikte. Jeder Agent versucht seine Aufgabe zu erfüllen.
- ▷ Rationalität: Ein Agent arbeitet auf Basis von Annahmen immer mit dem Ziel, seine Aufgaben zu erfüllen.

2.1.2 Abstrakte Agentenarchitektur

Der folgende Abschnitt gibt einen kurzen Überblick über die in [Woo00, S. 12-17] vorgestellte abstrakte Agentenarchitektur. Hierbei soll die Einordnung des BDI-Agenten ins System der Agenten beschrieben werden.

Laut Wooldridge kann die Umgebung eines Agenten durch eine Menge von Umgebungszuständen charakterisiert werden. Zu jedem Zeitpunkt befindet sich die Umgebung aus Sicht des Agenten in einem dieser Zustände.

- ▷ Umgebungszustände: $S = \{s_1, s_2, ...\}$

Für Agenten in realen Umgebungen ist die Aufnahme von Weltzuständen durch eine Wahrnehmungsfunktion (*perception*) beschrieben. Diese bildet Umgebungszustände auf eine Menge möglicher Wahrnehmungen ab.

- ▷ Wahrnehmungen: $P = \{p_1, p_2, ...\}$
- ▷ Wahrnehmungsfunktion: $perception : S \rightarrow P$

Die Fähigkeiten des Effektors eines Agenten kann in einer Menge von Aktionen zusammengefasst werden.

- ▷ Aktionen: $A = \{a_1, a_2, ...\}$

Abstrakt kann ein Agent also als eine Funktion (*action*) betrachtet werden, die aufgrund bestimmter Umgebungszustände bestimmte Aktionen auslöst.

- ▷ Agentenfunktion: $action : P^* \rightarrow A$

Die oben beschriebene Agentenfunktion bezeichnet laut Wooldridge einen *Standard Agenten*. Es existieren mehrere Möglichkeiten der Abbildung von Umgebungszuständen auf Aktionen. Diese sind in unterschiedlichen Agententypen beschrieben.

Rein reaktiver Agent Die Agentenfunktion reaktiver Agenten ist lediglich von der aktuellen Wahrnehmung abhängig. Vergangenen Zustände der Umgebung haben keinen Einfluss auf die Wahl der Aktion. Rein reaktive Agenten sind gleich dem *Standard Agenten*.

- ▷ Agentenfunktion: $action : P^* \rightarrow A$

Aktionswahl mit internem Zustand Agenten können mit internen Zuständen versehen werden. Interne Zustände repräsentieren die Vergangenheit des Agenten. Das Verhalten ist somit durch die Zustände der Umgebung bis zum aktuellen Zeitpunkt und der aktuellen Wahrnehmung bestimmt. Die Menge der möglichen internen Zustände eines Agenten beinhaltet aktuelle Informationen über die Umgebung und Wissen über vergangene Zustände.

- ▷ Interne Zustände: $I = \{i_1, i_2, ...\}$
- ▷ Folgezustand: $next : I \times P \rightarrow I$

Die Funktion *next* beschriebt einen Übergang vom aktuellen Zustand zu einem Folgezustand mithilfe aktueller Wahrnehmungen.

Unter Verwendung interner Zustände ergibt sich folgende Abbildung für die Agentenfunktion

- ▷ Agentenfunktion: $action : I \rightarrow A$

Der im anschließenden Abschnitt vorgestellte BDI-Agent ist ein Agent mit internen Zuständen. Eine umfangreiche Einführung in das Paradigma des Multiagentensystems inklusive bestehender Ansätze und Techniken beschreibt Michael Wooldridge in [Woo09].

2.2 BDI-Agenten

Der folgende Abschnitt beschreibt das für diese Arbeit verwendete BDI-Paradigma. Begonnen wird mit einer Motivation und Definition seiner Komponenten auf abstrakter Ebene. Im weiteren Verlauf wird die Semantik des Paradigmas durch aktuelle Realisierungen, der logischen Programmiersprache *AgentSpeak(L)* und einem *AgentSpeak(L)*-Agenten beschrieben. Die Darstellung durch *AgentSpeak(L)* dient zugleich als Grundlage für das in dieser Arbeit entwickelte System.

2.2.1 Historie des BDI-Paradigmas

Die ersten Grundgedanken des Paradigmas wurden 1987 von Michael. E. Bratman in den philosophischen Theorien „*Intention, plans, and practical reason*" [Bra87] verfasst. Bratman diskutierte in ihm die signifikante Rolle von *Intentions* im Umgang mit praktischem Reasoning. Wenige Jahre später beschrieben Cohen und Levesque [CL90] eine der ersten logischen Formalisierungen von *Intentions* und dem Begriff *Committment* als Grad der Festlegung auf Aktionsausführungen. Weitere Formalismen wurden anschließend von Konolige und Polack [KP93] und Singh und Asher [SA90] veröffentlicht.

2.2.2 Motivation des BDI-Paradigmas

Es existieren eine Vielzahl unterschiedlicher Anwendungsgebiete für heutige Softwaresysteme. Das in dieser Arbeit entwickelte Steuerungsystem wird in einer dynamischen Umgebung tätig sein. Dieser Abschnitt gibt einen kurzen Überblick über die Eigenschaften eines Echtzeit-Anwendungsgebietes und liefert eine Motivation für die Nutzung des BDI-Paradigmas.

2.2.2.1 Merkmale dynamischer Umgebungen und Echtzeitsysteme

Eine Charakterisierung von dynamischen Umgebungen und darin arbeitender Echtzeitsysteme beschrieben in [RG95, S. 313] lautet wie folgt:

1. Zu jedem Zeitpunkt gibt es mehrere potenzielle Möglichkeiten, wie sich eine Umgebung entwickeln kann. Die Umgebung ist nicht deterministisch.

2. Zu jedem Zeitpunkt gibt es potenziell mehrere mögliche Aktionen oder Prozeduren, die ein System ausführen kann. Das System ist nicht deterministisch.

3. Zu jedem Zeitpunkt gibt es potentiell mehrere verschiedene Ziele, die vom System erreicht werden sollen. Hierbei sind nicht alle Ziele simultan realisierbar.

4. Die Wahl von Aktionen und Prozeduren zur optimalen Realisierung gegebener Ziele ist abhängig vom derzeitigen Zustand der Umgebung (im Folgenden als Kontext bezeichnet) und unabhängig vom internen Zustand des Systems.

5. Der Zustand einer realen Umgebung kann in einem Sensorschritt nur lokal und nicht vollständig erfasst werden.

6. Die Rate mit der Berechnungen und Aktionen ausgeführt werden, steht in vernünftiger Relation zur Geschwindigkeit, mit der sich die Umgebung entwickelt.

Aktionsausführung in einer Echtzeitumgebung unterliegen häufig einer maximalen Ausführungsdauer, genannt Echtzeit. Echtzeit kann in harte Echtzeit und weiche Echtzeit unterteilt werden. Während die Überschreitung einer vorgegebenen maximalen, realen Reaktionszeit bei einer harten Echtzeitsystems als Fehler gewertet wird, definiert die weiche Echtzeit lediglich einen groben Richtwert [Bau11].

Charakteristika (3.) und (4.) der Eigenschaften eines Echtzeitanwendungsgebietes beschreiben die Verfolgung mehrerer Ziele und eine Unabhängigkeit von internen Zuständen. Diese Kriterien sprechen, nach der in Abschnitt 2.1.2 beschriebenen abstrakten Agentenarchitektur, sowohl für die Verwendung reaktiver- als auch zustandsbasierter Agentensysteme für die Arbeit in dynamischen Umgebungen. Das Konzept dieser Arbeit basiert auf zustandsbasierten BDI-Agenten. Die Realisierung eines Kontrollsystems unter Verwendung rein reaktiver Agenten ist in Abschnitt 1.2 und [Dyr11] beschrieben.

Weitere Untersuchungen zu agentenorientierten Systemen in komplexen, dynamischen Umgebungen sind unter anderem in [BIP88, Doy94, RG91, RK86, Sho93] verfasst.

2.2.2.2 Motivation einzelner BDI-Komponenten

Rao motiviert in [RG95, S. 314] die einzelnen Komponenten eines BDI-Systems anhand eines in [vL92, (S. 995-1072)] beschriebenen verzweigten Suchbaumes wie folgt:

Belief Laut Attribut *(4.)* ist es essentiell, dass das System zu jedem Zeitpunkt Informationen über den aktuellen Kontext besitzt. Schwierigkeiten treten durch unvorhersehbare Änderungen *(1.)* und einer lediglich lokalen Erfassung der Umgebung *(5.)* auf. Zu diesem Zweck soll das Kontextwissen in einer Komponente gespeichert und durch aktuelle Informationen aktualisiert werden. *Beliefs* können als Informationskomponente über den Systemzustand betrachtet werden.

Desire Das System benötigt Informationen über mögliche Ziele (und ihre Kosten und Nutzen) bei einem gegebenem Umgebungskontext (Charakteristika *(3.)* und *(4.)*). Dieses Wissen kann spontan durch Funktionen generiert werden und benötigt daher keine Zustandsrepräsentation. *Desires* können als Anregungszustand des Systems betrachtet werden.

Intentions Eine Schwierigkeit bei der Auswahl und Ausführung von Aktionen bilden spontane Änderungen im Kontext einer dynamischen Umgebung *(1.)*. Die folgenden zwei Fälle werden hierbei betrachtet:

Änderungen während der Aktionswahl Änderungen während der Auswahl von Aktionen erfordern zeitaufwendige Neuberechnungen des Auswahlverfahrens. Diese könnten durch die Verwendung schnellerer (und möglicherweise weniger optimaler) Entscheidungsfunktionen beschleunigt werden.

Änderungen während der Aktionsausführung Änderungen des Umgebungskontextes während der Ausführung von Aktionen erzeugen ein weitaus größeres Problem. Bisher wurde diesem Problem auf zwei unterschiedliche Arten begegnet:

- ▷ *Klassische Entscheidungstheorie*: Bei Änderungen der Umgebung wird eine neue Neuberechnung durchgeführt. Bei Umgebungen mit hoher Dynamik verbraucht dieses Vorgehen viel Rechenzeit und verzögert möglicherweise erheblich die Vollendung eines Aktionspfades durch häufige Unterbrechungen.
- ▷ *Klassische Informatik*: Klassische Computerprogramme führen, einmal initialisiert, ihre Aktionsausführung bis zum Ende durch. Hierbei können signifikante Veränderungen der Umgebung ein Ziel vor seiner Vollendung hinfällig werden lassen.

Eine umfangreichere Beschreibung des Problems kann in [Wei00, S. 31] nachgelesen werden. Nach Rao ist unter Beachtung des Attributs *(6.)* keiner der beiden Ansätze zufriedenstellend. Eine dritte Komponente übernimmt hier die Auswahl und Ausführung von Aktionen und entscheidet je nach Festlegungsgrad (*Committment*) über eine Neuberechnung. *Intentions* realisieren eine Balance zwischen zu vielen oder zu wenigen Neuberechnungen. Zusammengefasst beschreibt sie die deliberative[1] Komponente des Systems.

2.2.3 Vorstellung einzelner BDI-Komponenten

Im folgenden Abschnitt gibt einen Überblick über die Definitionen und Realisierungsmöglichkeiten der einzelnen Komponenten eines BDI-Systems.

2.2.3.1 Beliefs

Grob umschrieben repräsentieren *Beliefs* das aktuelle Wissen des Agenten über seine Umgebungen. Hierzu gehören Informationen aktueller Sensordaten, Nachrichten anderer Agenten und interne Zustände [TPH02, S. 1]. Das in *Beliefs* gespeicherte Wissen, ist nicht notwendigerweise immer wahr [MBC+06, S. 2]. Zusammenfassend können *Beliefs* als Informationskomponenten über den aktuellen Systemzustand beschrieben werden [RG95, S. 313].

Es existiert eine Vielzahl unterschiedlicher Realisierungen von *Beliefs*. Einige von ihnen sind: Einfache Variablen (*OASIS* [RG95, S. 317]), Prädikate (*AgentSpeak(L)* 2.2.4), modallogische Formeln (*Practionist* [MBC+06]) und objektorientierte Datenstrukturen (*Jam* 2.4.2 und *Jadex* 2.4.3).

Die Speicherung von *Beliefs* kann entweder in einer gemeinsamen Beliefbasis oder individuell im einzelnen Agenten stattfinden. Die Verwendung einer gemeinsamen Wissensbasis beugt Inkonsistenzen im Verhalten mehrerer Agenten aufgrund verschiedener Annahmen vor (ein Ansatz für *Joint-Intentions* und *Joint-Beliefs* ist beschrieben in [CL91]) und kann zugleich als gemeinsames Kommunikationsmedium dienen. Eine individuelle Wissensbasis umgeht das Bottleneck einer globalen Speicherstruktur und ist besser für die Handhabung individueller Zustände geeignet.

[1] Deliberation (engl. Beratung) bezeichnet im praktischen Reasoning den Vorgang der Zielbestimmung [Wei00, S. 29].

2.2.3.2 Desires

Desires beschreiben Zustände, die ein Agent zu erreichen versucht [TPH02, S. 1]. Sie können als Anregungszustand des Systems betrachtet werden [RG95, S. 313f].

Mögliche Repräsentationen von *Desires* sind unter anderem: Planlabels in *XML*-Dateien (*Jadex*), event- und kontextabhängige Pläne (*AgentSpeak*) und typisierte Ziele und *BNF*-Spezifikationen (*Jam*). Ziele können darüber hinaus mit bestimmten Eigenschaften versehen werden. Beispiele hierfür sind die Spezifikation von Relationen und Abhängigkeiten zwischen Zielen (*Practionist*) oder eine Unterscheidung von Zieltypen ([BPML04, S. 51] gibt einen Überblick über verwendete Zieletypen in BDI-Systemen).

2.2.3.3 Intentions

Intentions beschreiben den Weg, den ein Agent wählt, um sein Ziel zu erreichen [TPH02, S. 1]. Sie beschreiben zudem eine Komponente zur Entscheidungsfindung [RG95, S. 314] und bestimmen den Grad der Festlegung auf einen aktiven Handlungskurs (*Committment*).

Ein Beispiel für Intentionsmechanismen ist unter anderem die Verwendung eines *Means-End Reasoning*[2] für die Auswahl eines optimalen Aktionspfades. Das Reasoning kann hierbei unter anderem auf Basis von Kosten oder dem Wahrheitsgehalt modallogischer *Beliefs* (*Practionist*) geschehen. Eine weitere Möglichkeit ist die Verwendung von Selektionsfunktionen und eine Verwaltung teilweise initialisierter Pläne in parallelen *Runtimestacks* (*Jason* 2.4.4) oder die Koordination durch ein entscheidungstheoretisches Aufgaben-Scheduling in Planbäumen (*AgentSpeak(XL)* [BBJ+02]).

Die Intensität der Festlegung auf einen Aktionspfad kann nach [RG91, S. 3] in drei Bereiche eingeteilt werden. *Blindly-Committed* beschreibt eine unabhängige Zielverfolgung. Ein Agent lässt sein Ziel nur dann fallen, wenn es erreicht wurde. *Single-Minded* Agenten verfolgen ihre Ziele in Abhängigkeit zur aktuellen Beliefmenge. Ein solcher Agent lässt sein Ziel fallen, wenn ein Erreichen unmöglich scheint. *Open-Mindet* Agenten lassen Ziele dann fallen, wenn sie erreicht wurden oder nicht länger Teil der gewünschten Zielmenge sind. *Open-Minded* Agenten handeln abhängig von ihrer aktuellen *Belief*- und *Desire*-Menge.

2.2.4 AgentSpeak(L)

Der folgende Abschnitt erläutert die Verwendung des BDI-Paradigmas anhand von *AgentSpeak(L)*. *AgentSpeak(L)* ist eine BDI-agentenorientierte Entwicklungssprache auf Basis logischer Programmierung und wurde erstmals durch Anand S. Rao in „*AgentSpeak(L) BDI agents speak out in a logical computable language*" [Rao96] vorgestellt. Nach [MB01, S. 1] dient *AgentSpeak(L)* (in der Literatur oft nur als *AgentSpeak* bezeichnet) als Brücke zwischen der theoretischen BDI-Logik und implementierten Systemen. Inspiriert wurde *AgentSpeak* durch die Arbeit an der BDI-Architektur [RG95] und BDI-Logik [Rao98] und

[2] Means-End Reasoning bezeichnet im praktischen Reasoning die Auswahl eines optimalen Realisierungspfades für das Erreichen eines gegebenen Ziels[Wei00, S. 29].

einigen praktischen Implementationen, unter anderem am *Procedural Reasoning System* (beschrieben in [GL87]) und *dMARS* (beschrieben in [LW04]). Einige BDI-Architekturen auf Basis von *AgentSpeak* sind: *Jason* (vorgestellt in 2.4.4) und *Sim_ Agent* (beschrieben in 2.4.5). Eine Erweiterung von *AgentSpeak(L)* zu *AgentSpeak(XL)* wird in [BBJ+02] dargestellt.

2.2.4.1 Aufbau AgentSpeak(L)

Dieser Abschnitt erläutert anhand praktischer Beispiele den in [Rao96] vorgestellten Aufbau eines *AgentSpeak*-Agenten und seiner Komponenten. In den folgenden Beispielen werden Variablen durch großgeschriebene Ausdrücke symbolisiert.

Agent Ein Agent setzt sich zusammen aus Mengen von *Events, Beliefs, Plänen, Intentions, Aktionen*, den Auswahlfunktionen S_e, S_o, S_i und kann durch das folgende Tupel dargestellt werden:

$$\langle \textit{Events, Beliefs, Pläne, Intentions, Aktionen, } S_e, S_o, S_i \rangle$$

Beliefs *Beliefs* repräsentieren das aktuelle Wissen des Agenten durch atomare Formeln.
Beispiel: Beliefs einer angeschalteten Lampe

```
worldstate(lamp1, on, true)
worldstate(lamp1, dimValue, 1.0)
```

Ziele *Ziele* dienen zum Aufrufen von Unterzielen. *AgentSpeak* unterscheidet hierbei zwischen („?") Test- und („!") Ausführungs-Zielen. Testziele dienen unter anderem zum Erfragen von Werten, Ausführungsziele zur Ausführung von Unterplänen. Der Aufbau eines *Ziels* folgt hierbei dem eines *Beliefs*, erweitert um das zusätzliche Präfix „?" oder „!" entsprechend seines Typs.
Beispiel: Das testen und dimmen des Helligkeitswerts der Lampe

```
?worldstate(lamp1, dimValue, Value)
     → worldstate(lamp1, dimValue, 1.0)
!achieveWorldstate(lamp1, dimLight, 0.5)
```

Events *Events* werden durch das Hinzufügen („+") oder Entfernen („-") von *Beliefs* oder das Aufrufen von Unterzielen erzeugt. Es wird unterschieden zwischen *externen Events*, unter anderem ausgelöst durch das Hinzufügen oder Entfernen von *Beliefs*, und *internen Events* durch das Aufrufen von Unterplänen.
Beispiel: Auftretende Events beim Dimmen der Lampe

```
+!achieveWorldstate(lamp1, dimLight, 0.5)
-worldstate(lamp1, dimValue, 1.0)
+worldstate(lamp1, dimValue, 0.5)
```

Aktionen *Aktionen* beschreiben die letztendliche Ausführung von Agentenhandlungen. Sie sind zusammengesetzt aus einem Aktionssymbol und einer Menge von Formeln erster Ordnung. Sie können unter anderem *Beliefs* hinzufügen und entfernen oder Agentenhandlungen auslösen.
Beispiel: Die Aktion zum Dimmen einer Lampe und aktualisieren von Beliefs

```
achieve(lamp1, dimValue, 0.5)
-worldstate(lamp1, dimValue, 1.0)
+worldstate(lamp1, dimValue, 0.5)
```

Pläne Pläne in *AgentSpeak* bestehen aus einem *Head* und einem *Body*.

- ▷ *Head*: Setzt sich zusammen aus einem auslösenden *Event* (`e`) und einem *Kontext* aus konjunktiv verknüpften Bedingungen ($\mathtt{b}_1$ `&` ... `&` $\mathtt{b}_n$). Tritt das passende *Event* ein, wird der Plan in die Menge der *relevanten Pläne* aufgenommen. Gilt der *Kontext* eines *relevanten Plans*, wird dieser in die Menge der *ausführbaren Pläne* aufgenommen.
- ▷ *Body*: Beschreibt den Ausführungsteil eines Plans und besteht aus einer Sequenz von *Aktionen* und *Zielen* ($\mathtt{h}_1$`;` ...`;` $\mathtt{h}_m$).

Der Aufbau eines Plans kann wie folgt dargestellt werden:

$\mathtt{e}$ `:` $\mathtt{b}_1$ `&` ... `&` $\mathtt{b}_n$ `<-` $\mathtt{h}_1$`;` ...`;` $\mathtt{h}_m$`.`

Beispiel: Die Funktion zum Dimmen einer angeschalteten Lampe

```
+!achieveWorldstate(lamp1, dimLight, Value)
  :  worldstate(lamp1, on, true)
     & not worldstate(lamp1, dimValue, Value)
  <- achieve(lamp1, dimLight, Value);
     -worldstate(lamp1, dimValue, OldValue);
     +worldstate(lamp1, dimValue, Value).
```

Intentions *Intentions* beschreiben die Menge von *Aktionen* und *Zielen*, auf die sich ein Agent festgelegt hat. Realisiert werden sie durch parallele *Stacks* teilweise instantiierter Pläne. Externe *Events* erzeugen das Ausführen eines Planes in einem neuen *Stack*. Unterziele (Aufrufe interner *Events*) an der Spitze eines Planes werden durch den selektierten „Unterplan" ersetzt.

S_e Wählt ein *Event* aus dem *Eventbuffer*.

S_o Wählt einen *ausführbaren Plan* aus und fügt ihn der *Intention*-Menge hinzu.

S_i Wählt eine Aktion aus der Menge der *parallelen Intentionstacks* und führt diese aus.

2.2.4.2 Arbeitsweise eines AgentSpeak(L)-Agenten

Im Folgenden wird die Arbeitsweise eines *AgentSpeak(L)*-Systems anhand des in [BBJ+02] vorgestellten *AgentSpeak(L)*-Agenten beschrieben.

Abbildung 2.1 zeigt den *Reasoning*-Kreislauf eines *AgentSpeak*-Agenten. *Rechtecke* stellen die jeweilige Mengen dar. *Diamanten* beschreiben Selektionsfunktionen und *Kreise* die Prozesse des Systems.

1. Zu Beginn jedes Kreislaufs aktualisiert der *AgentSpeak*-Agent die Liste von *Events* und *Beliefs*. Die *Belief-Revision-Function*[3] (BRF) erzeugt hierbei neue *Events* bei jeder Änderung der Beliefmenge.

2. Die Auswahlfunktion S_e wählt ein *Event* für die Bearbeitung dieses Kreislaufs aus. Hierbei können Priorisierungen von Ereignissen realisiert werden.

3. Im nächsten Schritt wird das selektierte *Event* mit den *Events* der Pläne der *Plan-Library* unifiziert und die Menge der *relevanten Pläne* erstellt.

4. Die Kontextbedingungen aller *relevanten Pläne* werden mit der aktuellen Beliefbasis unifiziert. Hierbei wird die Menge der *ausführbaren Pläne* generiert. Diese Menge entspricht den aktuellen *Desires* des Agenten.

5. $S_o$wählt einen Plan der *ausführbaren Pläne* und fügt ihn der *Intention*-Struktur hinzu. Bei dieser Auswahl können Kosten-, Nutzen- oder Erfolgsinformationen beachtet werden. War das auslösende *Event* ein internes *Event* (Unterziel), wird der Ausführungsteil des neuen Plans an die Spitze des aufrufenden *Intention-Stacks* gelegt. Externe Events werden in neuen, individuellen *Stack* bearbeitet.

6. Abschließend wählt S_i eine *Aktion* aus einem der *parallelen Stacks* aus. Hierbei kann eine Aktionskoordination für *Intentions* realisiert werden (Abbildung 2.5 des *Jason*-Interpreters zeigt ein Beispiel für die Unterdrückung von *Intentions*).

7. Anschließend wird die gewählte *Aktion* ausgeführt. Eine atomare *Aktion* wird sofort ausgeführt. Interne *Events* werden der *Eventmenge* hinzugefügt und stehen dem nächsten *Reasoning*-Schritt zur Verfügung.

[3] Funktion zur Aktualisierung einer Wissensbasis bei Änderungen der Welt (vorgestellt in [RC04, S. 446]).

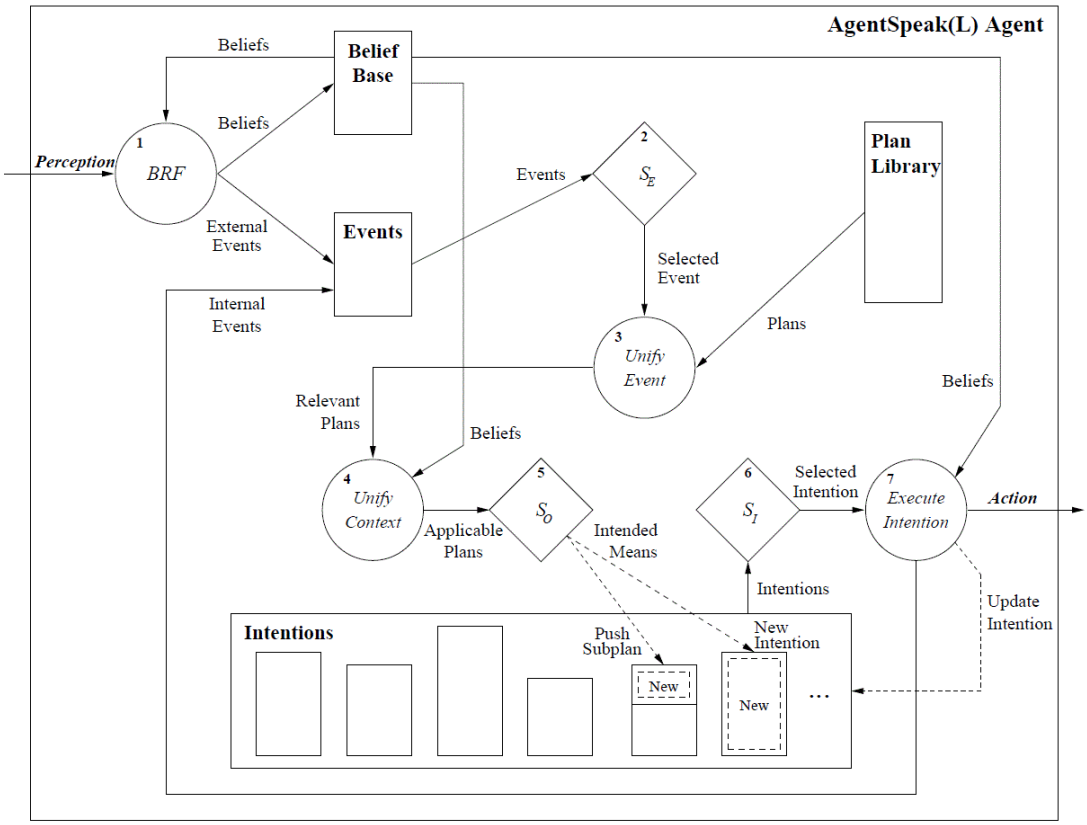

Abbildung 2.1: Der *Reasoningcycle* eines *AgentSpeak(L)*-Agenten [BBJ+02, S. 3].

Abbildung 2.2: Das *Smart-Appliance-Lab* der Universität Rostock [MMI11].

2.3 Das Smart-Appliance-Lab der Universität-Rostock

Das *Smart-Appliance-Lab* (kurz: *SmartLab*) der Universität Rostock dient der Forschung und Lehre im Bereich des *Ubiquitous Computing*. Das *SmartLab* ist eine instrumentierte, dynamische Umgebung. Für das in dieser Arbeit entwickelte System dient es als Testumgebung. Die folgenden Paragraphen beschreiben einige Eigenschaften des *SmartLabs*. Auf ihrer Grundlage werden im weiteren Verlauf dieser Arbeit Nutzungsszenarien und technische Annahmen für die Realisierung eines Kontrollsystems entwickelt. Abbildung 2.2 zeigt einen Rundumblick des Labors.

Sensoren Das *Smart-Appliance-Lab* verfügt über eine Vielzahl von Sensoren. Hierzu gehören unter anderem bewegliche Kameras in der Decke, Bodensensoren für Personenbewegungen, Bewegungs-, Licht- und Temperatursensoren. Für eine genaue Positionsbestimmung verfügt das Labor zusätzlich über ein Ubisense System [Ubi11]. Einzelne Sensoren sind mit *Java-Services*[4] versehen und können über die Infrastruktur des *SmartLabs* abgefragt werden.

Aktoren Das *SmartLab* ist eine Multidisplay-Umgebung. Es besitzt Projektoren, automatische Leinwände und Verdunkelungen für alle Wände des Raumes. Die Verwaltung von Anschlüssen geschieht über eine servicegesteuerte *Extron*-Einheit [Ext11]. Für die Verbindung von Notebooks sind die Tische des Labors mit sensorüberwachten VGA-Ports versehen. Die Geräte des Labors sind über ein EIB-System (beschrieben in [Fra09]) mit der Servicestruktur des Labors verbunden.

Middleware Die *Middleware* des *Smart-Appliance-Labs* kann unter dem Begriff „*helferlein*" zusammengefasst werden. Die Website [BRK11] beschreibt die *helferlein-Middleware* des *Smart-Appliance-Labs*. Der folgende Abschnitt dient als Überblick über die im *helferlein*-System realisierten *Middleware*-Konzepte und stellt die Grundlage für die Evaluation verfügbarer Umgebungsschnittstellen in 3.4 dar.

GenericPublisher Der *GenericPublisher* ist in der Lage, beliebige *Java*-Objekte für die Umgebung zu veröffentlichen. Veröffentlichungen können unter anderem im *Tuplespace*, via *Socket*, *Telnet* und *HTTP* oder als *Publish/Subscribe* geschehen. Die

[4] *Services* bezeichnen in diesem Zusammenhang eine Softwareschnittstelle zur Nutzung eines Geräts

Veröffentlichung eines Objekts via Socket ermöglicht unter Verwendung eines *NetworkProxies* die transparente Nutzung des Objekts durch verbundene *Network-Proxy*-Objekte (gleichen Interfaces) auf verschiedenen Computern. Die Veröffentlichung eines Geräte-*Services* via *Tuplespace* erzeugt Tupel für alle Zustände des Geräts und stellt eine Steuerungsschnittstelle als *CommandQueue*[5] bereit.

Infrastructure Die *Infrastructure* dient als verteiltes Verzeichnis veröffentlichter Objekte. Mit ihrer Hilfe können aktive *Services* in der Umgebung erfasst und überwacht werden. Die *Infrastructure* empfängt und verarbeitet hierfür Netzwerknachrichten über die Anwesenheit von Computern und Objekten und bietet Funktionen für das assoziative Auffinden passender *Services*.

Publish/Subscribe Das *Publish/Subscribe*-System dient der unabhängigen Verteilung von *Events*. Produzenten können mithilfe des Systems Nachrichten an Interessenten versenden. Diese abonnieren sich für *Events* bestimmten Typs und selektieren diese unter Verwendung von Filtern.

Tuplespace Der *Tuplespace* ist ein globaler, asynchroner, assoziativer Tupelspeicher. Der Zugriff auf ist entsprechend der in Linda (einer Sprache zur Koordination und Kommunikation paralleler Prozesse beschrieben in [EZ92]) definierten Methoden über *write*-, *read*- und *take*-Funktionen realisiert. Die assoziative Adressierung des gesuchten Inhalts geschieht über die Verwendung von *Template*-Tupeln. Für *Events* bei neuen Tupeln im *Tuplespace* kann sich per *Notifikation* unter Verwendung eines *Templates* beim *Tuplespace* angemeldet werden. Die Arbeit zur initialen Entwicklung und Verwendung eines *Tuplespaces* für das *Smart-Appliance-Lab* ist in [Nic09] beschrieben. Der aktuelle *Tuplespace* unterstützt umfangreiche Serialisierungsmechanismen und *Java*-Schnittstellen für eine direkte Nutzung abstrakter *Tuplespace*-Paradigmen wie beispielsweise *CommandQueues*.

Das SmartLab als dynamische Umgebung Abschnitt 2.2.2.1 beschreibt die Eigenschaften einer dynamischen Umgebung. In Bezug auf die vorangegangene Aussage, das *SmartLab* sei eine dynamische Umgebung, werden im Folgenden einige Charakteristika genannt, um diese Aussage zu untermauern.

Die Umgebung des *Smart-Appliance-Labs* kann durch unvorhergesehene Nutzerinteraktionen zu jedem Zeitpunkt nichtdeterministisch geändert werden. Zur Bestimmung von Nutzerintentionen werden zusätzliche Umgebungsparameter verwendet. Die Sensoren des *Smart-Appliance-Labs* bieten hierfür eine lokale Erfassung der Umgebung. Die Intention des Nutzers sind nicht immer eindeutig bestimmbar. Hierdurch existieren häufig mehrere (auch konkurrierende) Möglichkeiten einen Nutzer zu unterstützen. Die Ausführung sollte hierbei in angemessener Zeit stattfinden.

[5] *CommandQueues* bezeichnen hier im *Tuplespace* realisierte Queues für die Aufnahme von *Command-Tuple*. Der Inhalt eines *Command-Tuples* kann von veröffentlichten *Services* des *SmartLabs* als Funktionsaufruf interpretiert und ausgeführt werden.

2.4 BDI-Frameworks

Die folgende Übersicht von BDI-Frameworks dient als Grundlage der in Abschnitt 3.5 beschriebenen Auswahl einer Architektur zur Entwicklung eines Kontrollsystems für *Intelligente Umgebungen*. Fünf Frameworks wurden aufgrund ihrer Ansätze, Fähigkeiten und Einsatzgebiete aus der Menge verfügbarer BDI-Frameworks ausgewählt.

2.4.1 Jack

Jack ist ein kommerzielles Produkt der AOS Group[6]. Das Framework (beschrieben in [HRHL01]) ist ein umfangreiches *Java*-Tool zur Entwicklung von BDI-Agenten. Entworfen wurde es 1997 von ehemaligen Mitgliedern des „Austrailian Artificial Intelligence Institute[7]".

Jack arbeitet laut [AOS08a] auf Basis von Agenten mit individuellen *Belief*-Mengen. Agenten handeln auf eintreffende Ereignisse durch die Ausführung zustandsgesteuerter Pläne. Pläne werden in einer speziell für *Jack* entwickelten *Java*-Erweiterung „*Jack Plan Language*" (JPL) modelliert. Die internen Zustände des Agenten (*Beliefs*) sind in Form von Tupel realisiert. Ihre Abfrage geschieht entsprechend eines assoziativen Speichers (beispielsweise *Tuplespaces*) durch die Spezifikation des gesuchten Inhalts durch terminale Werte und Wildcards in *Template*-Tupel. Die Semantik von *Jack* liegt laut [HRHL01] zwischen logischer Programmiersprachen und embedded SQL.

Eine besondere Eigenschaft des Frameworks stellt die Koordination komplexer Agentenstrukturen in Gruppen dar. *Jack-Teams* ermöglicht die Verwaltung von Agentengruppen in hierarchischen Teams und die Verfolgung von Teamplänen (Abbildung 2.3 zeigt dies am Beispiel einer Fußballmannschaft). Das Konzept stützt sich auf die Annahme, dass jedem Teammitglied abstrakte Rollen zugewiesen werden können. Rollen dienen hierbei der Spezifikation von Verantwortlichkeiten, Fähigkeiten, Beziehungen und Abhängigkeiten. Die logische Gruppierung von Agenten hat das Ziel, den Kommunikations- und Verhandlungsaufwand zwischen einzelnen Agenten zu senken. Darüber hinaus soll die Nutzung von Teams die Entwicklung des Verhaltens großer Agentenmengen aus einer „Vogelperspektive" ermöglichen [HRHL01, S. 2].

Kommunikationsschnittstellen der Agenten können frei über Adapter implementiert werden und erlauben Realisierungen von Kommunikationsparadigmen wie KQML und ACL.
Bei der Entwicklung des BDI-Systems wird der Entwickler von folgenden in [AOS11b] beschriebenen Tools und einer detaillierten Dokumentation [AOS08b] unterstützt:

Jack-Agent-Language: Die *Jack Agent Language* (JAL) ist eine Programmiersprache für die Entwicklung von Agentensystemen. Sie erweitert *Java* um agentenorientierte, syntaktische Konstrukte.

[6] ©Agent Oriented Software Pty Ltd, Agent Oriented Software Limited (UK) and Agent Oriented Software Inc handeln als "AOS" und gemeinsam als "AOS Group".

[7] Das „Austrailian AI Institute" war ein 1988 von Bob Hawke ins Leben gerufenes Regierungsprojekt für die Erforschung und Kommerzialisierung künstlich intelligenter Systeme. Das Projekt wurde 1999 eingestellt.

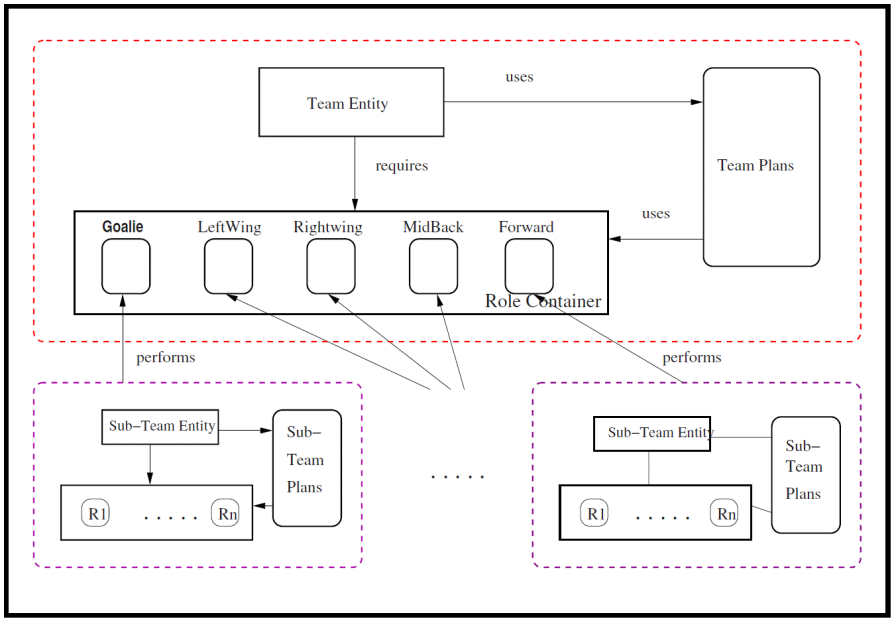

Abbildung 2.3: Teamstruktur und Teampläne einer Fußballmannschaft [LLJ08, S. 647].

Jack-Agent-Compiler: Konvertiert JAL Ressourcen zu *Java*-Code. Dieser kann dann von der JVM auf der Zielplattform ausgeführt werden.

Jack-Agent-Kernel: Die Laufzeitumgebung für Programme in JAL. Enthält eine Menge von Klassen für Hintergrundfunktionalitäten.

Jack-Development-Environment: Das JDE bietet dem Entwickler eine graphische Oberfläche für die Entwicklung von Plänen, die Verbindung von Plänen mit Agenten und der Konfiguration und Überwachung der Interagentenkommunikation, sowie der Verfolgung von Planausführungsschritten (beschrieben unter [AOS08b]).

Jack-Eclipseplugin Unterstützt die Entwicklung von *Jack*-Systemen in der Entwicklungsumgebung Eclipse.

Jack ist unter anderem im militärischen Bereich im Einsatz. Hier wird es für die Unterstützung taktischer Entscheidungen in Kampfszenarien verwendet. *Jack* dient hier zur Vorhersage menschlichen Verhaltens in Verteidigungssituationen und dem Management militärischer Ressourcen. Ein weiteres Beispiel ist die Verwendung zur Steuerung der *KI* im Computerspiel *Unreal-Turnament*.

C-BDI Ein weiteres kommerzielles Produkt der AOS-Goup befindet sich aktuell in der Entwicklung. *C-BDI* ist ein auf der Website [AOS11a] vorgestelltes *C++*-Framework, das den Anforderungen eines echtzeitsicheren und verifizierbaren Systems entsprechen und unter anderem in Fahrzeugen zum Einsatz kommen soll. Das „C" in *C-BDI* steht

folglich für „Certificable". Eine vorläufige Version des Systems ist zum gegenwärtigen Zeitpunkt (Anfang 2011) in C implementiert und wird aktuell von einer Gruppe von Nutzern evaluiert. Einige der Schlüsseneigenschaften des Systems sind die Identifikation von Zielkonflikten und eine stetige Steigerung der Entscheidungsprozessessicherheit, die Möglichkeit der Verifikation und Validation des Systems und die Einhaltung bestehender allgemeiner Standards. Für die Realisierung dieser Eigenschaften vertraut das System auf die Nutzung zertifizierter Compiler und der Verwendung von C++ als Implementationssprache. Eine Veröffentlichung von *C-BDI* soll im Laufe des Jahres 2011 stattfinden.

2.4.2 Jam

Jam ist ein 1999 von Marcus J. Huber in *„Jam: A BDI-theoretic mobile agent architecture"* [Hub99] vorgestelltes BDI-Framework. Als Produkt einer kleinen Firma mit dem Namen „Intelligent-Reasoning-Systems" (I.R.S.) in Südkalifornien ist es kostenfrei für einen nicht kommerziellen Einsatz nutzbar. Die Arbeit am *Jam*-Framework wurde 2008 mit der Auflösung des Unternehmens eingestellt. Die Ressourcen des Frameworks sind weiterhin auf der Internetseite [Hub08] verfügbar. *Jam* bietet eine hybride Agentenarchitektur und gründet auf den Paradigmen des *Procedural Reasoning Systems*[8] (PRS), einer Ausführung in *Structured Circuit Semantic*[9] (SCS) und einer Planbeschreibungssprache in *Act Plan Interlingua*[10] Die initiale *Belief*- und Planmenge der *Jam*-Architektur werden in BNF-Form[11] in Textdateien spezifiziert. Sie ermöglichen die Festlegung einer Menge von Fakten, bestehend aus Relationsname und Argumenten und die Spezifikation von Plänen zur Ausführung von *Java*-Funktionen.

Eine Besonderheit von *Jam* stellt die Möglichkeit dar, auf Basis von *Java* Serialisierungsmechanismen, Agenten zur Laufzeit auf andere Hostsysteme zu migrieren. Mithilfe einer *AgentGo*-Funktion kann jeder Agent zu einem beliebigen Zeitpunkt angehalten, migriert und an anderer Stelle fortgeführt werden.

Eine weitere Besonderheit des Frameworks ist die Verwendung verschiedener Zieltypen zur Ausführung von Aktionen:

- ▷ *Achieve*-Ziele erreichen einmalig ein gegebenes Ziel.
- ▷ *Maintain*-Ziele erreichen und erhalten einen Zielzustand.
- ▷ *Perform*-Ziele verfolgen permanent ein Ziel. Ein Beispiel hierfür ist das Auskundschaften einer Umgebung.

[8] Ein in [GL87, IGR92] vorgestelltes Framework zur Erstellung von Real-Time Reasoning Systemen.

[9] Eine Erweiterung zur Strukturierung und Kontrolle von atomaren nicht abbrechbaren Aktionen einer Ausführungsschleife, beschrieben in [LHDK94].

[10] Eine Sprache zur Formalisierung einer Wissensrepräsentation für komplexe Pläne und ihrer reaktiven Ausführung in dynamischen Umgebungen beschrieben in [WM95, MW97].

[11] Backus-Naur-Form beschreibt eine Metasprache zur syntaktischen Beschreibung einer Programmiersprache [MR03].

Das *Jam*-System besteht im Überblick aus den folgenden in [Hub99] und in Abbildung 2.4 dargestellten Komponenten:

Weltmodell: Eine *Belief*-Datenbank für *Java*-Datentypen und Objekte.

Planbibliothek: Die Planbibliothek für Pläne aus Vorbedingung und Ausführungsteil mit einfachen Aktionen, komplexen funktionalen Strukturen und argumentierten Unterplanaufrufen. Zudem können Pläne mit Nutzenfunktionen für ein nutzenbasiertes *Means-End Reasoning*[12] versehen werden.

Intentionsstruktur: Ausführbare Pläne (abhängig von Planvorbedinung und *Belief*-Menge des Agenten) werden in der *Applicable-Plan-List* (APL) verwaltet. Pläne in der APL werden auf Basis ihrer Nützlichkeit als *Intention* gewählt. Ein Runtimestack bestehend aus teilweise initialisierten Plänen verwaltet die Ziele des Agenten und unterstützt das Reasoning über mehrere gleichzeitig verfolge *Intentions*.

Interpreter: Das *Jam*-System arbeitet im Gegensatz zur in Abschnitt 2.4.1 beschrieben *Jack*-Architektur nicht auf Basis von Events sondern besitzt eine Interpreter Ausführungsschleife. Die Aufgabe des Interpreters ist die Selektion von Plänen aus der APL, die Festlegung selbiger als *Intention* und die anschließende Koordination und Ausführung einzelner Planschritte. Eine beispielhafte Implementation des *Jam*-Interpreters kann [Hub99] entnommen werden.

Observer: Der Observer ist eine leichte vom Nutzer spezifizierte Komponente. Sie ist in der Lage Aktionen zwischen den einzelnen Durchläufen des Interpreters auszuführen. Hierzu gehört unter anderem das Zwischenspeichern von Nachrichten. Realisiert ist der Observer als ein mit der Reasoningfähigkeit von *Jam* verknüpfbaren Plan ohne Bedingungsteil.

Einsatzgebiete von *Jam* liegen laut [Hub08] unter anderem im Bereich der Robotersteuerung und Entwicklung intelligenter Nutzerinterfaces.

2.4.3 Jadex

Jadex ist das BDI-Framework der „Verteilte Systeme und Informationssysteme"-Gruppe an der Universität Hamburg (Website: [UH11]). Es ist unter der *GNU LGPL* Lizenz (Lesser General Public License [FSF07]) verwendbar. *Jadex* basiert auf *Java* und *XML* und kann als Erweiterung des Multiagenten-Frameworks JADE (*Java* Agent DEvelopment Framework[CPR08]) gesehen werden. Die folgenden Attribute des *Jadex*-Systems sind unter [BP10] beschrieben. Die *Belief*-Menge eines *Jadex*-Agenten sind in Form von beliebigen *Java*-Objekten in einer *Belief*-Basis organisiert. Der Zugriff und die Verwaltung der Wissenselemente findet entweder direkt oder über eine *OQL*[13]-ähnliche Anfragesprache statt. Die Ziele (*Desires*) eines *Jadex*-Agenten beschreiben auf deklarative Weise was

[12] Einer zielbasierten Auswahl von Realisierungspfaden beschrieben in [Bra81] und [Wei00]

[13] Object Query Language ist eine in [CBB00] spezifizierte SQL ähnliche Anfragesprache für Objektdatenbanken.

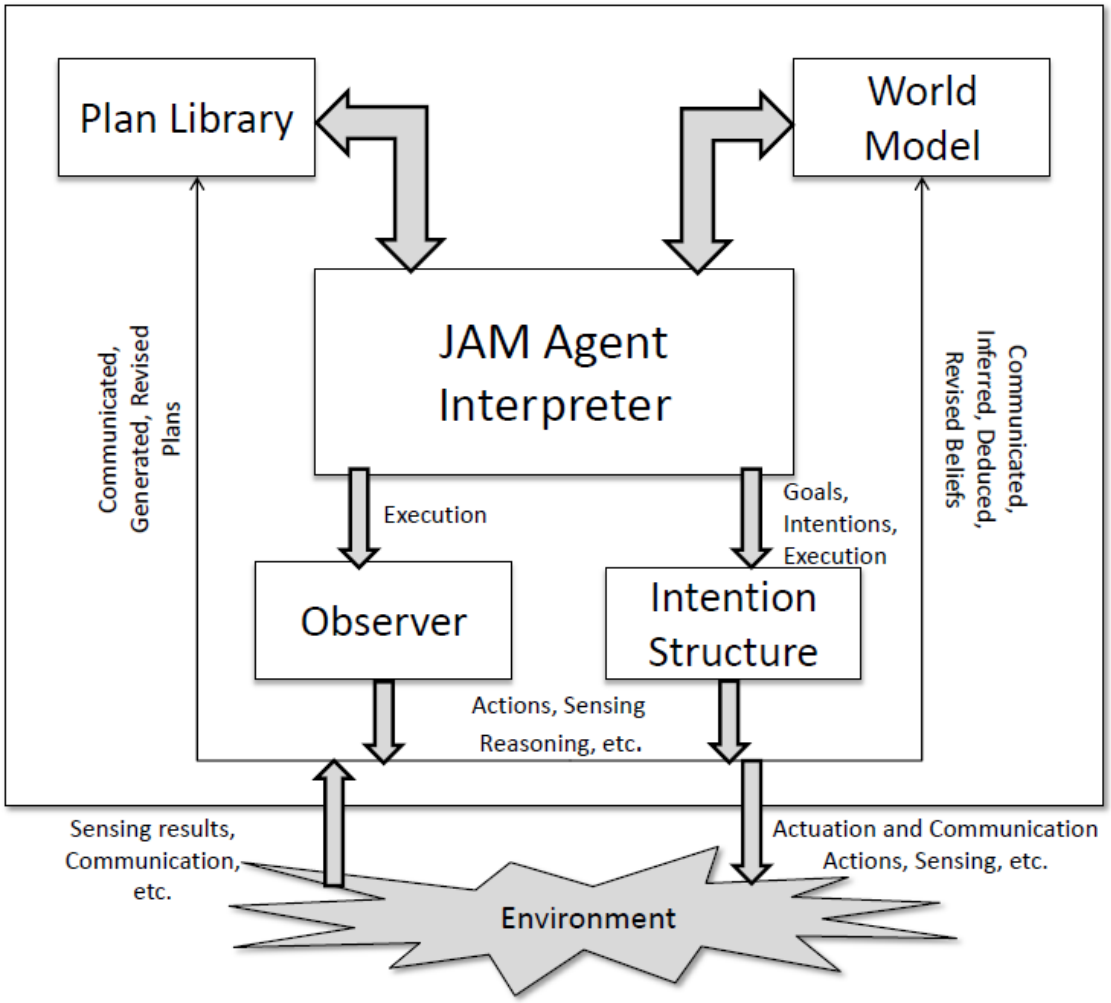

Abbildung 2.4: Eine Übersicht der *Jam*-Architektur nach [Hub99, S. 237]. .

ein Agent erreichen kann und enthalten keine Informationen darüber, wie dieses Ziel erreicht werden soll. Pläne (*Intentions*) sind realisiert als *Java*-Objekte mit vererbten Hilfsfunktionen zum Nachrichtenversand, zur Arbeitsverteilung an Unterzielen und dem Empfang von *Events*. Sie besitzen einen Ausführungsteil für das Erreichen abstrakter Ziele und Arbeiten nach dem Prinzip des Means-End Reasoning.

Während *Java* bei der Implementation der Agenten zum Einsatz kommt dienen *XML*-Inhalte in *Agenten-Definitions-Dateien* (ADF) zur abstrakten Spezifikation initialer *Beliefs*, *Desires* und *Intentions*. Die Arbeit auf abstrakten Zielen erhöht hierbei die Fehlersicherheit und senkt den Entwicklungsaufwand. Auch *Jadex* ermöglicht die Verwendung von *Achieve*- und *Maintain*-Zielen. Zusätzlich dienen *Query*-Ziele zur Erfassung von Wissensinhalten.

Die Agenten eines *Jadex*-Systems können ohne Code-Änderungen als Standalone oder auf der *Jade*-Plattform ausgeführt werden. Für die Entwicklung und das Überwachen des Agentensystems stehen unter anderem umfangreiche Ressourcen mit vordefinierten, generischen Funktionalitäten, zielorientierten Interaktionsprotokollen und das *Jadex-Control-Center* (JCC) als Laufzeitumgebung zur Verfügung.

Das JCC bietet unter anderem Funktionen zur Überwachen der Agentenkommunikationen, Ausführung von Simulationen, Unit-Tests und Agentenintrospektion. Die Nutzung der *Jade*-Plattform ermöglicht zudem eine standardisierte Kommunikation über FIPA-Protokolle.

Jadex kommt laut [UH11] unter anderem für die Entwicklung zielgerichteter Geschäftsprozesse [BP09] und für die Koordination von Behandlungsplänen in Krankenhäusern zum Einsatz [HL06].

2.4.4 Jason

Jason ist ein BDI-Framework basierend auf objektorientierten *Java*-Funktionalitäten und Prologreasoning. Entwickelt wurde es unter anderem von Jomi F. Hübner und Rafael H. Bordini. Veröffentlicht ist es unter der GNU LGPL Lizenz und steht kostenlos auf [HBF+10] zum Download bereit.

Jason ist nach Aussage der Entwickler der erste vollwertige Interpreter für AgentSpeak(L) (beschrieben in Abschnitt 2.2.4) . Darüber hinaus erweitert *Jason* die in AgentSpeak(L) spezifizierte Semantik um Annotationen von *Beliefs* (vorgestellt in [MVB02]) und Plänen und bietet Mechanismen zur Bearbeitung von Planfehlern. Das Framework besitzt die folgenden im Überblickskapitel von [HBF+10] beschrieben Eigenschaften:

Die Arbeit des *Jason*-Interpreters basiert auf der formalen Semantik von AgentSpeak. Eintreffende *Events* erzeugen kontextabhängige Planausführungen. Die Selektionsfunktionen des AgentSpeak-*Cycles* können unter Verwendung von Annotationsinformationen frei in *Java* angepasst werden. Agenten und Umgebungsschnittstelle des *Jason*-Systems sind in *Java* implementiert und bieten eine große Anzahl vorgefertigter interner Funktionen. Zu diesen Funktionen gehören: Eine sprachaktbasierte Interagentenkommunikation nach *SACI*[14], mit deren Hilfe Agenten *Beliefs* austauschen, Ziele verteilen und Pläne

[14] „Simple Agent Communication Infrastructure" ist ein in [HS03] beschriebenes *Java*-Framework für die Realisierung einer KQML basierten Interagentenkommunikation.

erfragen können. Die Umgebungsschnittstelle ermöglicht den Aufruf beliebiger, parametrisierter *Java*-Funktionen innerhalb des Ausführungsteils von AgentSpeak-Plänen und Funktionen für die Verteilung externer Wahrnehmungen auf alle oder einzelne Agenten.

Das Verhalten des Agenten wird auf logischer Ebene spezifiziert. *Beliefs* werden als annotierte Literale erster Ordnung und Pläne in Form annotierter AgentSpeak-Regeln dargestellt. Annotationen ermöglichen die Speicherung von Meta-Informationen und können bei der logischen Unifikation zur Auswertung von Kontexten und *Belief*-Mengen mit einbezogen werden. *Intentions* werden in Form von parallelen Stacks koordiniert. Das *Reasoning* des Systems entspricht, bis auf die zusätzliche Verarbeitung von Nachrichten und Koordinationsmechanismen für *Intentions*, dem in 2.2.4.2 vorgestellten Verfahren.

Jason unterstützt die Entwicklung eines Agentensystems mit einer eigenen Entwicklungsumgebung. Das Debugging geschieht über die Möglichkeit einer schrittweisen Ausführung, der Verwendung von Breakpoints und Introspektion für Agenten. Darüber hinaus existieren Plugins für Entwicklungsumgebungen wie Eclipse. *Jason* ist kompatibel zur *Ja*de-Plattform und kann ohne Änderung des eigenen Codes auf selbige migriert werden.

Für *Jason* existieren Schnittstellen zu *MOISE* und *Cartago*. *MOISE* (beschrieben in [HSB07]) erweitert das Framework um eine Spezifikation von Rollen, Gruppen und Teammissionen und einer zentralen Organisationskomponente zur Koordination von Agentengruppen. *Cartago* (beschrieben in [RVO06, RS10] erweitert das *Jason*-Framework um die Verwendung von Artefakten. Seit 2010 ist das *MOISE*-Framework für *Jason* in *Cartago* als Artefakt enthalten. Eine Kombination dieser drei Frameworks kann als *JaCaMo*-Projekt unter [BBH+11] bezogen werden.

Abbildung 2.5 zeigt den Reasoning-Kreislauf eines *Jason*-Systems. Dieser erweitert den in Abbildung 2.1 dargestellten AgentSpeak-Agenten um ein *SACI*-basiertes Kommunikationssystem und Mechanismen zur Intentionskoordination. Eine Umfangreiche Beschreibung des *Jason*-Systems ist in [BHV+05] beschrieben.

Jason kam unter anderem bei der Entwicklung von *MAS-SOC* [BOO+04] zum Einsatz. Hierbei ging es um die Simulation sozialer Aspekte des Vorstadtwachstums. *Jason* diente hierbei unter anderem zur Definition des Verhaltens virtueller Charaktere.

2.4.5 Sim_Agent

Auch *Sim_Agent* ist ein auf AgentSpeak(L) basierendes Toolkit. Entwickelt wurde es als Teil des „*Cognition and Affect Project*"[15] an der Universität von Birmingham. *Sim_Agent* ist ein Framework für die Entwicklung von Multi-BDI-Agentensystemen und als Ansatz einer allgemeinen *Kognitiven Architektur* im Bereich von ACT[16] und SOAR[17] einzuordnen. Ein Überblick über die Hauptmerkmale des *Sim_Agent* Toolkits ist auf der Internetseite der Universität von Birmingham [US11] zu finden.

[15] Ein 1991 in Birmingham gestartetes Forschungsprojekt zur Untersuchung von *Kognitiven Architekturen*, möglichen Designs, Anforderungen und ihren Korrelationen [Uni10]).
[16] Adaptive Control of Thought. Beschrieben in [And].
[17] State Operator And Result. Beschrieben in [LLR96].

Sim_ Agent bietet eine Generalisierung einer großen Vielfalt von Architekturen und umfangreiche Möglichkeiten der Modifikation und des Redesigns (syntaktische und semantische Erweiterungen). Als *Kognitive Architektur* bietet es unterschiedlichste kognitive Mechanismen (unter anderem Selbstwahrnehmung) zur Modellierung menschlichen Verhaltens.

Eine weitere Besonderheit des *Sim_ Agent*-Frameworks ist die Kapselung von Regeln in Regelmengen und Regelmengen in Regelfamilien. Jeder Reasoningschritt findet auf Basis einer einzelnen Regelmenge einer Regelfamilie statt. Zwischen einzelnen Reasoningschritten kann die verwendete Regelmenge einer Regelfamilie gewechselt werden (dargestellt in Abbildung 2.6).

Die Regelbasierte Wissens- und Verhaltensmodellierung findet mithilfe einer Erweiterung der objektorientierten Programmiersprache Pop-11[18] statt. Unterstützt werden Paradigmen von eventbasierter, konventioneller, prozeduraler und funktionaler Implementationen bis hin zur Verwendung *Neuronaler Netze* und logischer Programmierung. Für das Debugging ermöglicht das Toolkit prozedurale Änderungen zur Laufzeit, einen dynamischen Wechsel von Modulen und ein sogenanntes *Reflection Meta-Management* für den Zugriff auf eigene Regeln und die Aufzeichnung eigener Prozesse.

[18] Eine an der Universität von Sussex entworfene und mittlerweile von der Universität in Birmingham weiterentwickelte Programmiersprache (beschrieben in [BRS85]).

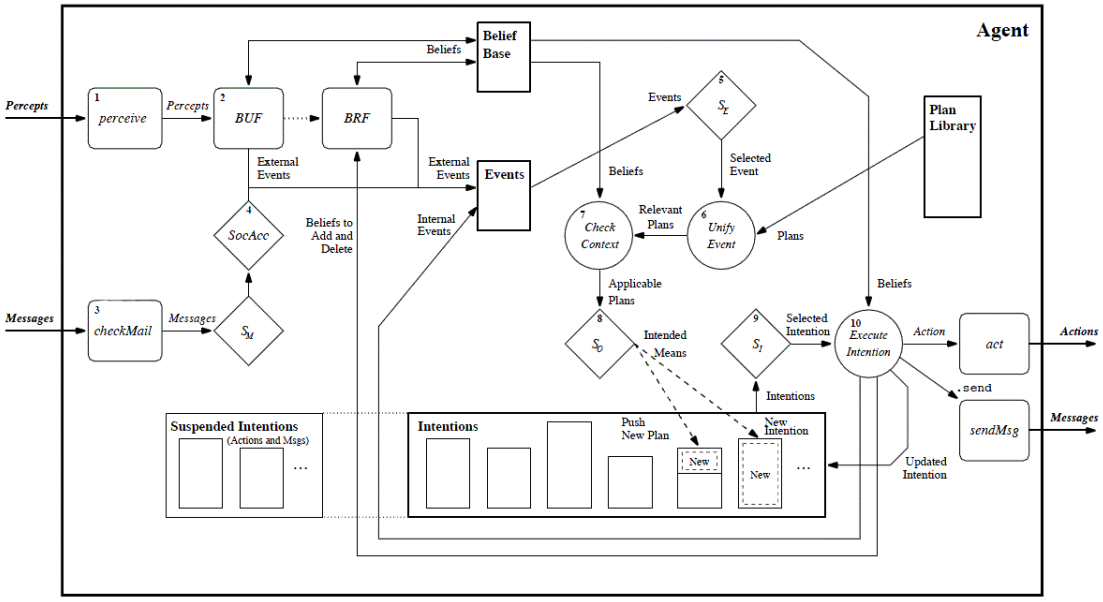

Abbildung 2.5: Der *Reasoningcycle* eines *Jason*-Agenten [BH07].

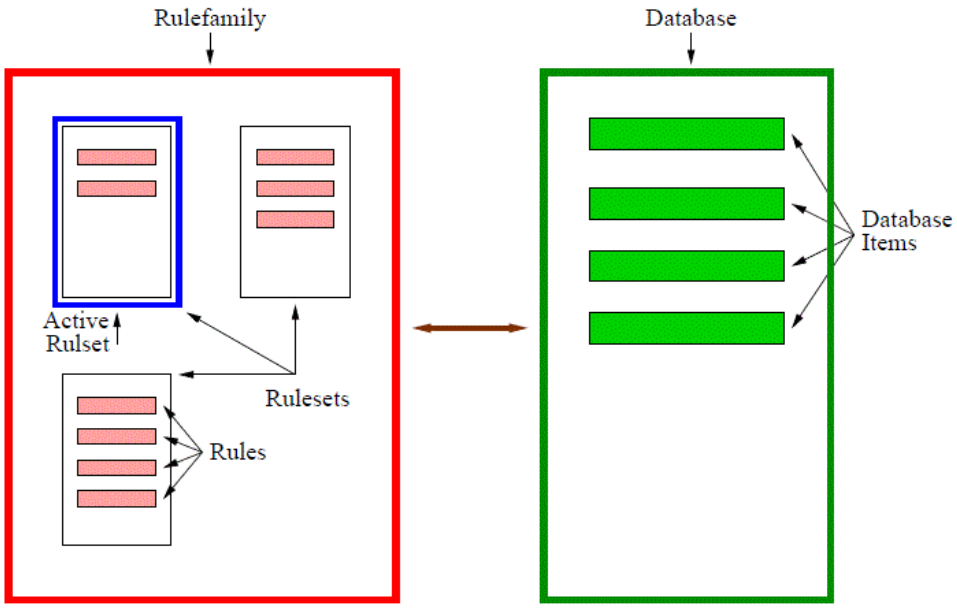

Abbildung 2.6: Kapselung von Regelmengen in Regelfamilien. [MB01, S. 8]

2.5 Zusammenfassung

In diesem Kapitel wurden die konzeptionellen und technischen Grundlagen für die Entwicklung des Kontrollsystems für instrumentierte, dynamische Umgebungen unter Verwendung des BDI-Paradigmas beschrieben. Begonnen wurde mit der Erläuterung von Agentensystemen und der Vorstellung des BDI-Agenten. Für die Beschreibung der Semantik und der praktischen Umsetzung des Paradigmas wurde das Konzept der logischen, BDI-agentenorientierten Programmiersprache *AgentSpeak(L)* verwendet. Zusätzlich wurde das *Smart-Appliance-Lab* der Universität Rostock und in ihm realisierte *Middlewares* als die Testumgebung des Kontrollsystem vorgestellt. Abschließend wurde ein Überblick über mögliche Architekturen für die Realisierung eines Kontrollsystems auf Basis von BDI-Agenten gegeben.

3 Konzeption

Das folgende Kapitel beschreibt die konzeptionelle Entwicklung des Kontrollsystems. Begonnen wird mit der Aufzählung einiger Szenarien, die die alltägliche Nutzung des *Smart-Appliance-Labs* (vorgestellt in Abschnitt 2.3) beschreiben. Anschließend werden allgemeine und abstrakte Annahmen für die Realisierung des erwarteten Umgebungsverhaltens und über die verfügbaren technischen Mittel des Labors getroffen. Mithilfe dieser Annahmen werden konkrete Anforderungen für die Entwicklung eines BDI-Multiagentensystems zur Steuerung einer *Intelligenter Umgebungen* entwickelt. Auf Basis dieser Anforderungen werden anschließend die in 2.3 beschriebenen *Middlewares* des *Smart-Appliance-Labs* und die BDI-Frameworks aus Abschnitt 2.4 evaluiert und Konzepte für eine konkrete Umsetzung vorgestellt und begründet.

3.1 Nutzungsszenarien

In den folgenden Szenarien werden alltägliche Situationen bei der Nutzung des *Smart-Appliance-Labs* dargestellt. Die durch „▷" gekennzeichneten Unterpunkten beschrieben das erwartete Raumverhalten. Auf der Grundlage dieser konkreten Szenarien werden im Anschluss abstrakte Annahmen für das erwartete Verhalten eines intelligenten Steuerungssystems modelliert.

S 1 Personen betreten einen Raum:

- ▷ Der Raum wird optimal beleuchtet.

S 2 Alle Personen verlassen einen Raum:

- ▷ Geräte schalten sich nach gewisser Zeit automatisch ab.
- ▷ Abhängige Geräte werden automatisch konfiguriert.
- ▷ Die Beleuchtung wird ausgeschaltet.

S 3 Ein Notebook wird mit einem VGA-Port verbunden:

- ▷ Der Inhalt der angeschlossenen Quelle wird automatisch angezeigt.
- ▷ Notwendige Konfigurationen werden automatisch durchgeführt.

S 4 Ein VGA-Port wird gelöst:

- ▷ Abhängige Geräte werden automatisch stillgelegt.
- ▷ Nicht mehr verwendete Ressourcen werden freigegeben.

S 5 Aktive Gerätekonfiguration wird von anderer Quelle überschrieben:

- ▷ Bestehende Gerätekonfiguration wird auf freie Ressourcen verteilt.

S 6 Geräte können Anweisungen nicht ausführen:

- ▷ Abhängige Gerätekonfigurationen werden auf freie Ressourcen verteilt.

S 7 Wenige, zusammensitzende Personen schließen ein Notebook an:

- ▷ Die gegenüberliegende Darstellungsfläche wird verwendet.

S 8 Mehrere verteilt sitzende Personen schließen ein Notebook an, um gemeinsam ein Dokument zu betrachten:

- ▷ Eine für alle gut sichtbare Darstellungsfläche wird verwendet.
- ▷ Bereits angezeigte Inhalte werden für alle gut sichtbar umgelegt.

S 9 Viele Leute verteilen sich in einen Raum, eine Person im Vortragsbereich verbindet ein Notebook mit einem VGA-Port:

- ▷ Eine optimale Vortragskonfiguration des Raumes wird hergestellt.
- ▷ Falls notwendig werden belegte Ressourcen akquiriert.
- ▷ Bestehende Gerätekonfigurationen werden entsprechend den Wünschen vorheriger Nutzer auf übrige Ressourcen verteilt.

S 10 Während eines Vortrages verbindet ein Zuhörer sein Notebook mit einem umliegenden VGA-Port:

- ▷ Eine für alle gut sichtbare Darstellungsfläche wird verwendet.
- ▷ Die Konfiguration des Vortragenden bleibt davon unberührt.

S 11 Ein Vortragender möchte während seines Vortrages das Whiteboard nutzen:

- ▷ Behinderungen werden entfernt.
- ▷ Abhängige Gerätekonfigurationen werden auf freie Ressourcen verteilt.

3.2 Annahmen

Der folgende Abschnitt beschreibt eine Menge abstrakter Annahmen, die für die Entwicklung eines Kontrollsystems für das *Smart-Appliance-Labs* getroffen wurden. Die Annahmen sind unterteilt in *Verhaltensannahmen*, gewonnen aus den in Abschnitt 3.1 beschriebenen Nutzungsszenarien und *technischen Annahmen* aus den in Abschnitt 2.3 beschriebenen technischen Spezifikationen des *Smart-Appliance-Labs*.

3.2.1 Verhaltensannahmen

Abschnitt 3.1 beschreibt einige Nutzungsszenarien und erwartete Verhaltensweisen der Umgebung. Der folgende Bereich definiert unter Verwendung dieser Szenarien eine Menge allgemeiner Annahmen zur Realisierung des beschriebenen Raumverhaltens. Mithilfe dieser Annahmen wird im Anschluss eine Menge konkreter Anforderungen spezifiziert.

VA 1 Die Notwendigkeit einer Handlung wird durch ein Ereignis erzeugt:

- ▷ *Personen betreten einen Bereich oder verbinden Geräte.*

VA 2 Die Art der Handlung ist abhängig vom Ereignis und Umgebungskontext:

- ▷ *S 7, S 8, S 9 und S 10: Beschreiben die Belegung und Rekonfiguration von Ressourcen beim Verbinden eines Notebooks abhängig von Personenverteilung und Ressourcennutzung.*

VA 3 Die Ausführung von Aktionen ist abhängig vom einzelnen Gerät:

- ▷ *S 2: Die Auswahl eines abhängigen Gerätes.*

VA 4 Die Ausführung von Aktionen ist abhängig vom Gerätetyp:

- ▷ *S 5: Die Auswahl freier Ressourcen mit gleichem Funktionsumfang.*

VA 5 Änderungen von Gerätezuständen haben Kosten:

- ▷ *S 8: Die Auswahl einer optimalen Darstellungsfläche.*

VA 6 Erwartete Aktionen sind abhängig von der aktuellen Situation im Raum:

- ▷ *S 9: Eine Vortragssituation erfordert eine umfangreichere Konfiguration der Umgebung.*

VA 7 Beginn und Ende einer Situation kann über Ereignisse und Zustände erkannt werden:

- ▷ *S 9: Eine Menge von Personen verteilt sich auf bestimmte Bereiche. Bestimmte Ereignisse, wie das Anschließen eines Notebooks, treten auf.*

VA 8 Situationen beinhalten Personenrollen mit unterschiedlichen Zielen und Prioritäten:

- ▷ *Szenario 10: Ein Zuhörer verbindet sein Notebook. Die Vortragskonfiguration bleibt davon unberührt.*

3.2.2 Technische Annahmen

Die folgenden technischen Annahmen wurden auf Basis der in 2.3 beschriebenen Spezifikationen des *Smart-Appliance-Labs* getroffen. Sie dienen als Grundlage für die Entwicklung konkreter infrastruktureller Anforderungen. Diese Anforderungen sollen unter anderem zur Auswahl einer optimalen Umgebungsschnittstelle dienen.

TA 1 Das *Smart-Appliance-Lab* ist eine dynamische Umgebung.

TA 2 Die Geräte des Labors besitzen eindeutige Zustände.

TA 3 Das Labor verfügt über eine *Middleware* mit folgendem Funktionsumfang:

1. Verbindungsmöglichkeiten zu allen Geräten.
2. Bereitstellung von Informationen zur Identifikation und Typisierung verbundener Geräte.
3. Bereitstellung von Gerätezuständen.
4. Mitteilung von Zustandsänderungen verbundener Geräte.
5. Möglichkeit zur Steuerung verbundener Geräte.
6. Erreichbar über Kommunikationsinfrastruktur des *Smart-Appliance-Labs.*

TA 4 Es existieren Sensoren und Schnittstellen für das Erfassen von Positionsinformationen.

TA 5 Nicht vorhandene Geräte liefern keine Informationen.

3.3 Anforderungsanalyse

In diesem Abschnitt werden die Anforderungen für das in dieser Arbeit entwickelte System zur Steuerung einer intelligenten, dynamischen Umgebung spezifiziert. Die im Folgenden entwickelten Aussagen beziehen sich auf die in Abschnitt 3.2 getroffenen Annahmen. Die am Ende dieses Abschnittes vorgestellte Anforderungsliste beschreibt die Menge der konkret getroffenen Anforderungen für die Entwicklung des Kontrollsystems auf Basis des BDI-Paradigmas.

3.3.1 Diskussion der Annahmen

Um eine Menge von Anforderungen für die Realisierung des Kontrollsystems zu entwickeln und zu begründen, werden die in 3.2 beschriebenen abstrakten Annahmen im einzelnen diskutiert

VA 1 *Die Notwendigkeit einer Handlung wird durch ein Ereignis erzeugt*

Die selbstständige Ausführung von Aktionen als Reaktion auf Ereignisse (im Folgenden als *Event* bezeichnet), spricht für eine Realisierung als eventbasiertes Agentensystem.

VA 2 *Die Art der Handlung ist abhängig vom Ereignis und Umgebungskontext*

Die Wahl einer Handlung basiert auf *Event*- und Umgebungszustand (*Kontext*). Das Kontrollsystem sollte informationsbehaftete Ereignisse verarbeiten können und über eine aktuelle, konsistente Repräsentation des Weltzustandes für all seine deliberativen Komponenten verfügen. Eine Deliberation auf Basis von *Event* und *Kontext* wäre hierfür von Vorteil.

VA 3 *Die Ausführung von Aktionen ist abhängig vom einzelnen Gerät*

Da der Handlungsbedarf für jedes Gerät einzeln bestimmbar sein muss, sollte das Kontrollsystem über die Fähigkeit einer gerätebasierten *Event*- oder Zielverteilung verfügen. Ein eigener Deliberationsprozess für jedes Gerät unterstützt die individuelle Ausführung von Aktionssequenzen. Einzelne Geräte sollten hierfür in der Lage sein, eigene interne Zustände verwalten. Hierfür bietet sich die Realisierung des Steuerungssystems in Form eines Multi-Agentensystems an ([Coo09] gibt eine Übersicht von Forschungsgebieten über die Nutzung von MAS in *Intelligenten Umgebungen*).

VA 4 *Die Ausführung von Aktionen ist abhängig vom Gerätetyp*

Nach dieser Annahme wird der Handlungsbedarf und Funktionsumfang eines Gerätes durch seinen Typ bestimmt. Das Kontrollsystem benötigt Informationen über die Art eines Geräts, um die Handlungen und Fähigkeiten entsprechend seines Typs festzulegen. Darüber sollte eine typbasierte *Event*- oder Aktionsverteilung und Mechanismen zur Handhabung von Ressourcenkonflikten realisiert werden.

VA 5 *Änderungen von Gerätezuständen haben Kosten*

Änderungen haben Kosten. Aus diesem Grund sollte das System ein kostenbasiertes Means-End Reasoning unterstützen.

VA 6 *Erwartete Aktionen sind abhängig von der aktuellen Situation im Raum*

Da die Menge an Situationen im Raum stetigen Veränderungen unterliegt, muss auch das Verhalten des Kontrollsystems dynamisch anpassbar sein. Das System muss jederzeit in der Lage sein, Handlungsweisen zu entfernen oder hinzuzufügen.

VA 7 *Beginn und Ende einer Situation kann über Ereignisse und Zustände erkannt werden*

Das Kontrollsystem sollte das parallele Erkennen und Verwalten unterschiedlicher Situationen unterstützen. Der Wechsel zwischen Situationen und abhängigen Systemparametern sollte in angemessener Zeit stattfinden.

VA 8 *Situationen beinhalten Personenrollen mit unterschiedlichen Zielen und Prioritäten*

Das System benötigt Mechanismen zur dynamischen Anpassung und Koordination logisch gruppierter Handlungsmengen. Hierzu gehört die situationsbedingte Zuordnung zu Personen und eine dynamische Verwaltung aktiver Zielmengen, Aufgabenbereiche und Verantwortlichkeiten. Die Realisierung eines exklusiven Ressourcenzugriffs könnte die Koordination unterstützten. Die Definition der Handlungsweise sollte zur Erhöhung der Fehlersicherheit und Senkung des Entwicklungsaufwands auf einer hohen Abstraktionsebene geschehen. Gleichzeitig muss eine ausreichende Mächtigkeit gegeben sein, um auch komplexe Bedingungen und Vorgänge abzudecken.

TA 1 *Das Smart-Appliance-Lab ist eine dynamische Umgebung*

Das Steuerungssystem soll als BDI-Multiagentensystem implementiert werden. Diese Entscheidung setzt sich zusammen aus der bereits entstandenen Anforderung zur Realisierung eines Multi-Agentensystems und den Vorteilen des BDI-Paradigmas für den Einsatz in dynamischen Umgebungen. Die Dynamik des *Smart-Appliance-Labs* ist in 2.3 erläutert. Eine Motivation für die Verwendung eines BDI-Systems in dynamischen Umgebungen kann Abschnitt 2.2.2.2 entnommen werden. Die Definition des Paradigmas ist in Abschnitt 2.2 beschrieben.

TA 2 *Die Geräte des Labors besitzen eindeutige Zustände*

Diese Annahme bildet die Grundlage für das Weltwissen des Kontrollsystems. Der für die Deliberation verwendete Umgebungskontext sollte sich aus der Menge aller Zustände der im Raum befindlichen aktiven Geräte und eigener interner Zustände zusammensetzen. Das System sollte in der Lage sein, Gerätezustände zu erfassen und in einer einheitlichen Form zu verwalten.

TA 3 *Das Labor verfügt über eine Middleware...*

Das Kontrollsystem muss sich mit einer geeigneten Middleware verbinden können. Es sollten alle notwendigen Lese- und Schreibmechanismen unterstützt werden. Neue Geräte müssen dynamisch gefunden, identifiziert und gesteuert werden können. Darüber hinaus sollte es eine einheitliche Form der Informationsdarstellung und Aktionsausführung geben. Die Aktionsausführung muss hierbei in angemessener Zeit und transparent geschehen. Der aktuelle Systemzustand muss jederzeit ersichtlich sein.

TA 4 *Es existieren Sensoren und Schnittstellen für das Erfassen von Positionsinformationen*

Das System sollte in der Lage sein, schnell eintreffende, multimodale Sensorwerte zu verarbeiten und gegebenenfalls auf diese reagieren können. Hierzu sind unter Umständen eine Menge heterogener Schnittstellen nötig. Rohdaten könnten auf ein höheres, einheitliches Abstraktionslevel gehoben werden um eine Verarbeitung zu erleichtern und die Abhängigkeit zur realen Umgebung zu verringern.

TA 5 *Nicht vorhandene Geräte liefern keine Informationen*

Da die Menge an Geräten einem dynamischen Wechsel unterliegt, existieren im Allgemeinen keine negativen Informationen für nicht vorhandene Geräte. Die Verwendung einer Closed-World-Assumption, zur richtigen Interpretation fehlender Zustände, würde sich anbieten.

3.3.2 Konkrete Anforderungen für die Realisierung des Kontrollsystems

Das Kontrollsystem für dynamische, intelligente, instrumentierte Umgebungen wird als BDI-Multiagentensystem realisiert. Im Zusammenhang mit BDI-Systemen werden im Folgenden die Begriffe *Belief* (Wissensbasis), *Desire* (Zielmenge) und *Intention* (Aktionswahl und Ausführung) verwendet. Eine genauere Erläuterung dieser Begriffe kann Abschnitt 2.2.3 entnommen werden. Die folgende Anforderungsliste wurde aus den in 3.3.1 diskutierten Annahmen gewonnen und beschreibt die technischen Anforderungen für die Entwicklung des Kontrollsystems im Rahmen des BDI-Paradigmas.

3.3.2.1 Belief

1. Geräte und Sensordaten:
   a) Das System muss fähig sein, dynamische Geräte zu finden und zu identifizieren.
   b) Es müssen Informationen über Gerätetypen, Zustände und Zustandsänderungen erfasst werden können.
   c) Geräte sollten über individuelle, interne Zustände verfügen.
   d) Das Kontrollsystem muss über Schnittstellen für das Empfangen heterogener, multimodaler Sensordaten verfügen.
   e) Unterschiedlicher Sensordaten gleichen Typs sollten auf einer höheren Abstraktionsebene zusammengefasst werden.
2. Zustände:
   a) Das System muss über eine geeignete Lese- und Schreibschnittstellen für die *Middleware* einer *Intelligenten Umgebung* verfügen.
   b) Es sollte eine einheitliche Form der Darstellung von Zuständen besitzen.
   c) Die Wahrnehmung informationsbehafteter Ereignisse sollte möglich sein.
   d) Das System sollte in der Lage sein, übergeordnete *Kontexte* zu generieren.
   e) Das System muss eine globale, aktuelle Repräsentation des Weltzustandes besitzen.

3.3.2.2 Desire

1. Die Deliberation sollte auf Basis einer Closed-World-Assumption geschehen.
2. Geräteverwaltung:
   a) Das System muss neue Geräte automatisch in die Zielbestimmung mit einbeziehen.
   b) Es sollte eine geräte- und gerätetypbasierte *Event*- oder Zielverteilung realisieren.
   c) Gerätefunktionalität und Verhalten sollte anhand des Gerätetyps festgelegt werden.
   d) Geräte sollten über einen eigenen Deliberationsprozess verfügen.
   e) Ressourcenkonflikte sollten behoben werden können.

3. Deliberation und Verhaltensdynamik:
   a) Die Zielbestimmung sollte auf Basis von *Event* und *Kontext* funktionieren.
   b) Die Definition einer Verhaltensweise sollte mächtig und abstrakt sein.
   c) Es muss eine Möglichkeit der dynamischen Manipulation von Zielmengen existieren.
   d) Es sollte eine logische Gruppierung von Zielmengen möglich sein.
   e) Das System sollte über Mechanismen zur Zuordnung von Zielmengen und Autoritäten zu Personen verfügen.

3.3.2.3 Intention

1. Aktionswahl:
   a) Der Ausführungspfad zum Erreichen eines Ziels sollte durch ein Means-End Reasoning geregelt werden.
   b) Die Koordination aktiver Ziele sollte realisiert werden.
   c) Das System sollte einen Exklusivzugriff für Ressourcen unterstützen.
   d) Es sollte kein unerklärliches Verhalten auftreten.
2. Ausführung:
   a) Es sollte eine parallele, unabhängige Ausführung von Geräteaktionen möglich sein.
   b) Aktionsausführungen sollten durch eine einheitliche Schnittstelle gesteuert werden.
   c) Die Aktionsausführung muss in angemessener Zeit stattfinden.
   d) Aktionsausführungen sollten transparent sein.

3.4 Evaluation verfügbarer Middlewares

Ein wichtiger Punkt für die Realisierung eines Kontrollsystems ist die Wahl einer geeigneten Schnittstelle zur Umgebung. Für instrumentierte Umgebungen existiert eine große Anzahl von *Middlewares* (engl. Dienstschicht). Einige von ihnen sind *Corba* [MR97], *Jini* [Sun11], *UPnP* [JHN04] und das in 2.3 vorgestellte *helferlein*-System.

Das *helferlein*-System realisiert viele *Middleware*-Konzepte in einem Projekt. Für eine Verteilung von Anwendungen und *Services* können (analog zu *Corba* und *Jini*) transparente Proxyverbindungen zu veröffentlichten Objekten verwendet werden. Für das Auffinden von *Services* ist ein dezentrales Verzeichnis (ähnlich dem *Jini Registrar*) vorhanden. Ein *Tuplespace* (vergleichbar mit dem in *Jini* vorhandenen *Javaspace*) ist ebenfalls verfügbar. Zusätzlich bietet das *helferlein*-System eine Kompatibilität zwischen einzelnen Konzepten, die es unter anderem ermöglicht *Services* sowohl über das Netzwerk, als auch über einen *Tuplespace* zu veröffentlichen und transparent durch Proxyverbindungen zu nutzen. Da das *helferlein*-System im *Smart-Appliance-Lab* zur Anwendung kommt kann es für die Umsetzung eines Kontrollsystems direkt verwendet werden.

Bei der Wahl einer geeigneten Schnittstelle wurde sich, aufgrund vorangegangener Argumente auf die Komponenten des *helferlein*-Systems beschränkt. Eine Evaluation einiger Konzepte des Systems in Bezug auf seine Verwendung in Forschung und Lehre kann [BRK10] entnommen werden. Im folgenden Abschnitt werden die einzelnen Teile der *Middleware* des *Smart-Appliance-Labs* untersucht. Hierfür wurden aus der in 3.3.2 beschriebenen Anforderungsliste relevante Kriterien für die Wahl einer geeigneten Schnittstelle extrahiert.

3.4.1 Auswahlkriterien für die Untersuchung einer Middleware

Die Untersuchung der in [BRK11] vorgestellten Frameworks des *Smart-Appliance-Labs* basiert auf den in der folgenden Liste spezifizierten Kriterien. Sie beschreibt abstrakte Anforderungen für die Realisierung der in Abschnitt 3.3.2 getroffenen Aussagen.

1. Finden und Bereitstellen von Identifikations- und Typinformationen neuer Geräte.
2. Eine Möglichkeit zur Abfrage von Gerätezuständen.
3. Eine automatische Mitteilung von Zustandsänderungen.
4. Eine automatische Verteilung von Sensordaten.
5. Stellen einer Schnittstelle zur Gerätesteuerung.
6. Eine angemessene Geschwindigkeit.

3.4.2 Untersuchung der Middleware-Konzepte

Die derzeit verfügbaren *Middleware*-Konzepte des *Smart-Appliance-Labs* sind zusammengefasst als das „*helferlein*"-System und beschrieben in [BRK11]. Der folgende Abschnitt stellt eine Zusammenfassung der Evaluation einzelner Konzepte dar.

	Infrastructure	NetworkProxy	Publish/Subscribe	TupleSpace
Finden, Identifizieren, Typisieren neuer Geräte	+	-	0	+
Abfrage von Gerätezuständen	0	+	+	+
Mitteilung von Zustandsänderungen	0	+	+	0
Verteilung von Sensordaten	0	+	+	0
Schnittstelle zur Aktionsausführung	-	+	-	+
Geschwindigkeit	+	+	+	+
Summe	1	4	3	4

Abbildung 3.1: Eine Übersicht der Evaluation betrachteter Middlewares.

Infrastructure Die *Infrastructure* dient in erster Linie dem Auffinden und Identifizieren aktiver Komponenten der Umgebung. Für die Steuerung von Geräten oder eine Übertragung dynamischer Gerätezustände und Sensordaten ist die *Infrastructure* bislang nicht vorgesehen.

NetworkProxy/GenericPublisher Der *GenericPublisher* bietet die Möglichkeit beliebige Objekte automatisch zu veröffentlichen. Mithilfe des *NetworkProxies* und einem passenden Interface kann anschließend innerhalb des Netzwerkes ein lokaler Proxy für eine transparente Remotesteuerung dieses Objekts verwendet werden. Die Schnittstelle bietet eine effiziente Möglichkeit zur Interaktion mit Geräteservices. Für das Erstellen einer Proxyverbindung ist Kenntnis über die implementierende Klasse und den Host des Geräteservices erforderlich. Durch die Kombination mit der *Infrastructure* können diese Informationen gewonnen werden. Eine Übersicht über die Funktionen des *NetworkPublishers* zur Veröffentlichung von Socketschnittstellen ist in [Koc10] beschrieben.

Publish/Subscribe Das *Publish/Subscribe*-System dient dem unabhängigen, gefilterten Empfang von Ereignissen. Das System bietet eine effiziente Möglichkeit der Veröffentlichung und Überwachung von Zustandsänderungen. Für die Steuerung von Geräten ist es nicht vorgesehen. Eine Beschreibung des Systems kann [Nyo10] entnommen werden.

Tuplespace Der *Tuplespace* ist ein assoziativer Tupelspeicher. Er enthält Zustände und Funktionsbeschreibungen verbundener Geräte. Die Verbindung mit dem *Tuplespace* kann mithilfe eines *NetworkProxies* realisiert werden. Für eine Ausführung von Aktionen stehen *CommandQueues* einzelner Geräte zur Verfügung. Das Eintreffen neuer Zustände kann über *Events* empfangen werden. *Events* liefern hierbei lediglich die Art des Zustandes. Für die Verarbeitung hochfrequentierter Sensordaten ist der *Tuplespace* nicht vorgesehen.

Zusammenfassung der Ergebnisse Abbildung 3.1 zeigt eine Einschätzung der verfügbaren Realisierungen bezüglich dieser Kriterien. Mit einem „+" gekennzeichnete Kriterien werden direkt erfüllt, „0" deutet auf eine teilweise Übereinstimmung der Anforderungen und „-" auf einen nicht direkt vorgesehenen Anwendungsfall hin.

3.4.3 Verwendung von Tuplespace, NetworkProxy und Infrastructure

Die Wahl geeigneter *Middlewares* fällt auf die kombinierte Verwendung von *Network-Proxies, Tuplespace* und *Infrastructure*. Mithilfe von *NetworkProxies* können Verbindungen zu *Tuplespace* und Sensorschnittstellen erstellt werden. Der *Tuplespace* bietet eine Möglichkeit der direkten Abfrage und Identifikation aktiver Geräte, Gerätetypen und Zustände. Die Änderung eines Zustandes kann durch das Scannen der Zustandsmenge mit dem BDI-System abgeglichen werden. Darüber hinaus bietet der *Tuplespace* durch die Implementation des *CommandQueues* eine textbasierte Schnittstelle zur parallelen Steuerung von Geräten. Diese Form der Steuerung ermöglicht eine direkte Weiterleitung textueller, parametrisierter Steuerungsbefehle aus *AgentSpeak(L)*. Die Verwendung eines *Tuplespaces* begünstigt zudem die Realisierung einer öffentlichen Schnittstelle für die Konfiguration des BDI-*Controllers*. Die *Infrastructure* soll während der Initialisierungsphase zum Auffinden des *Tuplespaces* und Sensorservices dienen.

3.5 Evaluation verfügbarer Frameworks

Es existiert eine Vielzahl von Möglichkeiten für die Realisierung eines BDI-Systems. Der folgende Abschnitt zeigt eine Evaluation der in 2.4 beschriebenen Frameworks bezüglich der in Abschnitt 3.5.1 definierten Anforderungen. Im Anschluss werden die Unterschiede bei der Evaluation zusammengefasst und die Wahl eines Frameworks begründet.

3.5.1 Auswahlkriterien für die Untersuchung eines Frameworks

Die Untersuchung der in Abschnitt 2.4 vorgestellten Frameworks basiert auf den in der folgenden Liste spezifizierten Kriterien. Sie beschreibt abstrakte, softwaretechnische Anforderungen für die Realisierung der in Abschnitt 3.3.2 getroffenen Aussagen.
Für die Einschätzung eines Frameworks wurden folgende Kriterien untersucht:

1. *Belief*
   a) Einfache, frei konfigurierbare Darstellung von *Beliefs*
   b) Unterstützung einer allgemeinen und einer individuellen assoziativen *Belief*-Basis
2. *Desire*
   a) Deliberation auf Basis von *Event* und *Kontext*
   b) Unterstützung einer logischen Gruppierung und einem dynamischen Wechsel von *Desires*
   c) Arbeit auf einer *Closed-World-Assumption*
   d) Abstrakte Verhaltensdefinition
3. *Intention*
   a) Parallele Verfolgung mehrerer Ziele
   b) Koordinationsmechanismen für mehrere aktive Ziele
   c) Unterstützung eines kostenbasierten Means-End Reasoning
   d) Transparente Arbeitsweise
4. Kompatibilität zur aktuellen Middlewarestruktur des *Smart-Appliance-Labs*
5. Unterstützung für die Entwicklung eines Multiagentensystems
6. Unabhängige Ausführbarkeit
7. Einfache Entwicklung und Wartung

3.5.2 Untersuchung der Frameworks

Im Folgenden werden die Vor- und Nachteile einzelner evaluierter Frameworks bezüglich der in 3.5.1 definierten Anforderungen beschrieben.

Jack Eine Besonderheit des *Jack*-Frameworks ist die hierarchische Koordination von Rollen und Teamzielen. Die Verwendung dieses Frameworks würde eine direkte Umsetzung des Rollenverhaltens ermöglichen. Nach [AOS11b] muss die Verteilung von *Beliefs* an alle Agenten vom Entwickler selbst realisiert werden. Die Pläne des BDI-Frameworks werden in einer eigenen Plansprache (JPL) oder der graphischen Entwicklungsumgebung spezifiziert. Nachträgliche Anpassung des Verhaltens erfordern die Verwendung der *Jack*-Entwicklungsumgebung oder das Schreiben von *Java*-Code.

Jam Eine Besonderheit des *Jam*-Frameworks ist die Nutzung verschiedener in Abschnitt 2.4.2 beschriebener Zieltypen. Die Verwendung dieses Frameworks könnte die Fixierung eines gewünschten Weltzustandes erleichtern. Pläne werden in einer „*Applicable-Plan-List*" zusammengefasst, die je nach Umgebungskontext aktualisiert wird. Die Auswahl eines Ziels geschieht anhand von Meta-Reasoning Prozessen oder Nutzenfunktionen [Hub99, S. 237]. Die Realisierung einer „*Event* x *Kontext*"-Deliberation, einer Gruppierung von Verhaltensweisen und ein dynamischer Austausch müsste über diese Prozesse realisiert werden.

Jadex Das *Jadex*-Framework verfügt über eine XML-basierte Initialkonfiguration. Mit ihrer Hilfe können Beliefmengen und Pläne mit abstrakten Zielen spezifiziert werden. Die interne Repräsentation von *Beliefs* wird durch *Java*-Objekte in einer *Belief*-Basis realisiert. Die Abfrage geschieht über eine API durch eine OQL-ähnliche Anfragesprache (beschrieben in Abschnitt 2.4.3). Auch Jadex unterstützt die Verwendung unterschiedlicher Zieltypen.

Jason Das *Jason*-Framework ist laut Entwickler die erste Implementation eines Interpreters für AgentSpeak (beschrieben in 2.2.4). *Jason* verwendet eine prologbasierte Unifikation für die *Event* und *Kontext* gesteuerte Deliberation. Die Spezifikation von AgentSpeak ermöglicht eine prädikative Definition von *Beliefs* inklusive der Nutzung variabler Felder. Die Festlegung von Verhaltensweisen geschieht in ASL-Textdateien in Übersichtlicher AgentSpeak-Notation. Auf diese Weise kann Agentenverhalten ohne das Schreiben von *Java*-Code modelliert werden. Für das Aufnehmen von Informationen und das Ausführen von Aktionen steht eine schlanke *Java*-Schnittstelle zur Verfügung. *Jason* bietet keine vorimplementierte Realisierung für eine hierarchische oder typisierte Koordination mehrerer aktiver Ziele. Könnte aber hierfür mit der MOISE-Architektur (näher erläutert in 2.4.4) erweitert werden.

Sim_Agent Auch *Sim_ Agent* basiert auf der Interpretation von AgentSpeak. Als Ansatz einer allgemeinen Kognitiven Architektur besitzt es eine vielzahl vorgefertigter kognitiver Modelle. Da das in dieser Arbeit entwickelte System das Ziel rationaler (nicht menschlicher) Entscheidungen hat, sind diese Modelle für eine Realisierung weniger relevant. Eine weitere Besonderheit ist die Koordination von Verhaltensmengen in Regelfamilien und Regelmengen. Mit ihrer Hilfe könnte eine logische Gruppierung von Verhaltensmengen realisiert werden. *Sim_ Agent* ist in Pop-11 (beschrieben in 2.4.5) implementiert

	Jack	Jam	Jadex	Jason	Sim_Agent
Belief					
Mächtige und flexible Beliefdefinition	+	+	0	+	+
Globale assoziative Belief-Basis	0	+	+	+	+
Desire					
Deliberation auf Event und Kontext	+	0	+	+	+
Logische Gruppierung von Desires	+	0	+	+	+
Dynamischer Wechsel von Desires	+	0	+	+	+
Intention					
Koordination mehrerer aktiver Ziele	+	+	+	0	+
Allgemein					
Kompatibilität	+	+	+	+	-
Unabhängige Ausführung, einfache Wartung	0	+	+	+	0
Summe	6	5	7	7	5

Abbildung 3.2: Eine Übersicht der Evaluation betrachteter Frameworks.

und müsste daher unter Verwendung spezieller Schnittstellen mit der überwiegend javabasierten Infrastruktur des Labors verbunden werden. Durch die Nutzung dieser Sprache ist die anschließende Wartung möglicherweise mit zusätzlichem Aufwand verbunden.

Zusammenfassung der Ergebnisse Abbildung 3.2 zeigt die Unterschiede einer subjektiven Aufwandsabschätzung bezüglich der vorab spezifizierten Auswahlkriterien. Die mit einem „+" gekennzeichneten Punkte sind direkt durch die Verwendung des Frameworks abgedeckt. Mit „0" gekennzeichnete Felder erfordern eine Erweiterung des Funktionsumfangs. Ein mit einem „-" gekennzeichneter Punkt beschreibt die Notwendigkeit umfangreicher Anpassungen.

3.5.3 Verwendung der Jason-Architektur

Bei der Untersuchung der Frameworks fällt die Wahl auf *Jason*. Bezogen auf die in Abschnitt 3.5.1 beschriebenen Anforderungen sind hierfür folgende Gründe zu nennen:

Kompatibilität Das Framework bietet eine schlanke, javabasierte Umgebungsschnittstelle. Sie implementiert Funktionen zur Aufnahme von Informationen und argumentierten Ausführung von *Java*-Code durch *AgentSpeak(L)*-Pläne. *Java* bietet als objektorientierte Programmiersprache eine weitverbreitete Grundlage für die Entwicklung plattformunabhängiger, mobiler Systeme. Der Hauptteil der Middlewarekonzepte des *Smart-Appliance-Labs* ist in *Java* realisiert.

Belief Wissenselemente können über annotierte Prädikate, inklusive einer beliebigen Anzahl von Argumenten (auch Variablen), dargestellt und global verteilt werden. Sie bilden die Wissensbasis der Prologresolution (Deliberation). Annotationen können unter anderem verwendet werden, um die Herkunft eines Wissenselements zur erfahren.

Desire Die Deliberation findet auf Basis der Beliefmenge eines Agenten und dem auslösenden *Event* statt. Das *Matching* geschieht hierbei durch die Unifikation logischer Aussagen. Annotierte Informationen werden in den Prozess mit einbezogen. Das Framework bietet darüber hinaus eine Menge vorgefertigter interner Funktionen. Mit diesen ist es unter anderem möglich annotierte Pläne zur Laufzeit hinzuzufügen oder zu entfernen. Für die Spezifikation von Planbedingungen kann, sowohl eine einfache Negation (*Closed-World-Assumption*), als auch eine starke Verneinung (*Open-World-Assumption*) verwendet werden.

Intention Intentionen sind über parallele Intentionsstacks realisiert. Von hier aus werden Unterpläne geladen und Aktionen ausgeführt. Für ein Nutzenbasiertes Means-End Reasoning können Pläne mit Annotationen versehen werden. Spezielle interne Funktionen ermöglichen die Ausgabe von Konsolentexten während jedes Arbeitsschritts. Zur Ausführung von Aktionen in der Umgebung können beliebige parametrisierte *Java*-Funktionen verwendet werden.

MAS-Unterstützung Zur Unterstützung eines Multiagentensystems verfügt das Framework über Funktionen für eine SACI-basierte Interagentenkommunikation und die Möglichkeit zum Erstellen und Entfernen von Agenten zur Laufzeit.

Unabhängige Ausführung Das *Jason*-System besteht aus einem *Java*-Archiv. Es ist keine Installation für die Ausführung notwendig.

Entwicklung und Wartung Für die Entwicklung bietet *Jason* ein Eclipseplugin und ein *Jason*-Development-Environment. Mit ihnen ist eine Introspektion und schrittweise Ausführung der Agenten möglich. Die anschließende Anpassung von Verhaltensweisen erfordert lediglich das Bearbeiten übersichtlicher *AgentSpeak(L)*-Pläne in ASL-Textdateien.

JaCaMo-Erweiterung *JaCaMo* [BBH+11] (*Jason*, *Cartago*[RVO06] und *MOISE* [HSB07]) erweitert das *Jason*-Framework um zwei zusätzliche allgemeine Architekturen für die Entwicklung von Multiagentensystemen. Das *MOISE*-Framework beschreibt ein allgemeines Framework für die Organisation von Agentengruppen. *Cartago* ist eine allgemeine Architektur für die Arbeit mit virtuellen Artefakten (beschrieben in 1.2).

MOISE Hübner beschreibt in [HSB07, S. 2] die zwei Hauptpunkte der in [LE98] vorgestellten Sichten auf Organisationsstrukturen. Zum einen die agentenorientierte Sicht, bei der jeder Agent seinen eigenen Beitrag erfüllt und die Organisationsstruktur als emergentes Phänomen entsteht (als Beispiel verweist Hübner hier auf einen Ameisenhügel). Zum anderen beschreibt er die systemorientierte Sicht, in der die Organisation selbst als eine vom Entwickler modellierte Entität existiert und von einzelnen Agenten erfragt und befolgt werden kann.
MOISE realisiert nach Hübner eine systemorientierte Sicht der Organisation. Das Framework ermöglicht die Modellierung von Agentengruppen, Rollen, Gruppenzielen und Abhängigkeiten in Form von Spezifikationen und eine anschließende Nutzung in Organisationsentitäten. Die Verwendung der *MOISE*-Architektur wird in [HSB07, S. 7] am Beispiel einer Fußballmannschaft beschrieben.
Das Beispiel der Fußballmannschaft zeigt ein Anwendungsfeld mit relativ statischer Gruppenstruktur, Rollenverteilung und klar definierten langfristigen Zielen.
Die Arbeit in *Intelligenten Umgebungen* besitzt eine hohe Rollen- und Gerätedynamik. Das dynamische Hinzufügen und Entfernen (teilweise unbekannter) Geräte, Gerätetypen, Geräteabhängigkeiten und Personenrollen, sowie der situationsbedingte Wechsel aktiver Ziele und Kostenvorstellungen müsste vorab bei der Entwicklung globaler Spezifikationen abgedeckt werden. Die hohe Dynamik des Geräteensembles und der Organisationsstrukturen ist vorab schwer definierbar. Aus diesem Grund wird für dieses Kontrollsystem eine dezentrale, agentenbasierte Sicht der Organisation angestrebt und von der Verwendung der *MOISE*-Architektur abgesehen.

Cartago Die Verwendung von Artefakten ermöglicht eine Kapselung von Funktionsweisen zur Steuerung und Überwachung von Geräten (Objekten) der Umgebung. Zusätzlich unterstützen Artefakte das verteilte Arbeiten mehrerer Agenten an einer Aufgabe. Artefakte können als objektorientierte, verteilt nutzbare „Werkzeuge" eines Agenten betrachtet werden. Weitere Beispiele für den Einsatz von Artefakten wurden in Abschnitt 1.2 vorgestellt.
Die Verwendung des *Tuplespaces* als „Universalartefakt" ermöglicht die direkte Übertragung von Gerätezuständen beliebiger Geräte in die Beliefbasis des Systems. Darüber hinaus realisiert die Nutzung von *CommandQueues* eine einzelne, einheitliche Schnittstelle zur Steuerung aller notwendigen Geräte. Die Bearbeitung eines Problems durch mehrere Agenten wird durch die Anforderung einer globalen assoziativen Beliefbasis abgedeckt. Nach einer subjektiven Einschätzung erfüllt die Nutzung der in 3.4.3 beschriebenen *Middleware*-Konzepte des *Smart-Appliance-Labs* bereits alle relevanten Anforderungen, die durch die Nutzung der *Cartago*-Architektur abgedeckt werden könnten. Die Auslagerung von Agentenfähigkeiten in Artefakte würde den Aufwand der Wartung und Konfiguration des Systems durch komplexere *AgentSpeak*-Prozesse und das Schreiben von *Java*-Artefakten erhöhen. Aus diesem Grund wird *Cartago* für die Entwicklung des Kontrollsystems nicht verwendet.

3.6 Konzeptionsbeschreibung des Kontrollsystems

Durch die Wahl adäquater *Middleware*-Konzepte (3.4) und eines geeigneten Frameworks (3.5) wurde bereits ein umfangreicher Teil der in Abschnitt 3.3.2 beschriebenen Anforderungen erfüllt. Im Folgenden werden die Konzepte zur Realisierungen der Restmenge konkreter, offener Anforderungen diskutiert. Am Ende jedes Abschnitts werden die einzelnen entwickelten Konzepte zusammengefasst. Mithilfe dieser Zusammenfassungen wird im Anschluss (3.6.12) der konzeptionelle Aufbau des Kontrollsystems beschrieben.

3.6.1 Umgebungsschnittstellen

Das *Smart-Appliance-Lab* bietet eine Vielzahl von Schnittstellen zur Erfassung und Manipulation von Zuständen. Die für diesen *Controller* verwendeten Schnittstellen wurden in 3.4.3 zusammengefasst. Die Konfiguration und Verbindung wird über sogenannte *„Adapter"* erfolgen. *Adapter* beschreiben *Java*-Implementationen einzelner Schnittstellen und bieten Funktionen für das Erfassen und Versenden von Informationen. Für die Übertragung von Zuständen wird die Umgebungsschnittstelle des *Jason*-Frameworks derart erweitert, dass eine Zischenspeicherung globaler Wissensmengen erfolgt. Auf diese Weise wird eine konsistente gemeinsame Beliefbasis aller Agenten realisiert.

- ▷ *Adapter* = Schnittstelle zu einer Middleware

3.6.2 Wissen in globalen Worldstates

Das System soll über eine gemeinsame und individuelle Beliefmengen verfügen. Philip Cohen und Levesque beschreiben in [CL91] das Konzept der *Joint-Beliefs*. Hierbei hat jeder Agent Informationen über das Wissen seines Nachbarn. Die Verwendung einer globalen Beliefbasis ermöglicht unter anderem eine allgemeine Verteilung von *Events* und Umgebungszuständen und erhöht die Konsistenz im Verhalten mehrerer Agenten durch gleiche Voraussetzungen. Zusätzlich sollen einzelne Agenten des Systems die Möglichkeit besitzen individuelle Zustände in einer eigenen Beliefbasis zu speichern.

Zur Förderung von Kompatibilitäten und Verwendung klarer Schnittstellen sollte die Darstellung von Zuständen und Zielwünschen in einheitlicher Form geschehen. Eine Möglichkeit zur Kapselung einzelner Zustände ist die in [Bad10, S. 3] für die Realisierung eines Kontrollsystems auf Basis einer logischen Programmiersprachen konzipierte Darstellung durch *Worldstates*. *Worldstates* realisieren eine eindeutige Zuordnung einer Zustandsmenge zu einem Objekt. Die Verwendung einer Menge separater Fakten ermöglicht einen direkten Austausch einzelner Zustände eines Objekts. Die Repräsentation von Zuständen durch *Worldstates* ähnelt zusätzlich dem strukturellen Aufbau der Geräte-Tupel des *Tuplespaces*.

Die folgende Darstellung ähnlich [Bad10, S. 3] beschreibt den Aufbau eines *Worldstates*. Das voran beschriebene Objekt wurde in diesem Konzept durch *Subjekt* ersetzt. Die Bezeichnung *Subjekt* kapselt hierbei neben Objekten auch Vorgänge als Träger von Attributen (beispielsweise Kosten).

- ▷ *Worldstate* = ⟨*Subjekt, Key, Value*⟩
- ▷ *Subjekt* = Beliebiger Träger von Attributen
- ▷ *Key* = Eindeutiger Bezeichner einer Eigenschaft
- ▷ *Value* = Wert dieser Eigenschaft

Die Prädikate zur Repräsentation interner *Beliefs* eines Agenten können frei gewählt werden. Sie sollten sich aber dennoch vom Schema des *Worldstates* unterscheiden, um die Trennung globalen und individuellen Wissens zu verdeutlichen.

- ▷ *Internes Belief* = Beliebiges individuelles Wissensprädikat

3.6.3 AgentSpeak in RegelMengen

Die Verwaltung von Funktionalitäten des Systems soll unter anderem durch das Hinzufügen und Entfernen von Zielmengen erfolgen. Das *Jason*-Framework beschreibt die aktive Zielmenge eines BDI-Agenten durch die Menge ausführbarer *AgentSpeak*-Pläne. Das Framework unterstützt zusätzlich die Annotation einzelner Pläne durch eindeutige *Label*- und Herkunftsangaben. Die Agenten des *Jason*-Systems können zur Laufzeit Pläne hinzufügen oder entfernen. Für das Zusammenfassen dieser Pläne zu Verhaltensweisen und Funktionsmengen wird für die Entwicklung des Kontrollsystems eine Zuordnung mehrerer Pläne zu einer *RegelMenge* realisiert. Die Identifikation einer *RegelMenge* geschieht über eine *RegelMengenID*.

- ▷ *RegelMengenID* = Eindeutiger Bezeichner einer *RegelMenge*
- ▷ *RegelMenge* = ⟨*RegelMengenID*, {AgentSpeak-Plan}⟩

3.6.4 Geräte und Agenten als Entitäten

Die Gerätemengen des *Smart-Appliance-Labs* können sich dynamisch zur Laufzeit ändern. Der *Controller* sollte fähig sein, das Verhalten gemäß der aktuellen Menge von Geräten anzupassen. Zusätzlich sollen einzelne Geräte über eigene interne Zustände verfügen. S. Bader beschreibt in [Bad10, S. 2] unter anderem die Notwendigkeit der Unabhängigkeit einzelner Komponenten eines Kontrollsystems. Aufgrund dieser Anforderungen wird für dieses System jeder aktiven, eindeutig identifizierbaren *Entität*[1] ein eigener Agent zugeordnet.

[1] *Entität* (lat. "entitas" Substantiv zu "sein") dient in diesem Zusammenhang der Erweiterung der Menge aller Geräte zur Menge aller „Dinge"

Agenten besitzen die Fähigkeit des autonomen Handelns und können selbstständig die Realisierung notwendiger Aktionssequenzen zur Erreichung eines gewünschten Zielzustands übernehmen. Zusätzlich ermöglichen sie die Modellierung individuellen zustandsbasierten und reaktiven Verhaltens. Einzelne Agenten können eigene Zustände besitzen und selbstständig auf das Eintreffen von *Events* und Zielen reagieren. Das Management von Ressourcen und die Konfiguration abhängiger Komponenten können autonom durch den Agenten einer *Entität* geregelt werden. Zur gerätebasierten Verteilung von Aufgaben ist es notwendig einzelne Agenten zu identifizieren. Hierfür nehmen Agenten die Identität ihrer *Entität* an. Diese Identität wird im Folgenden als *EntitätsID* bezeichnet.

- ▷ *EntitätsID* = Eindeutiger Bezeichner einer *Entität*
- ▷ *Entität* = Ein beliebiges, eindeutig identifizierbares Ding

3.6.5 Funktionalität durch EntitätsTypen

Vorab wurde die Zuordnung von Agenten zu *Entitäten* beschrieben. Eine weitere Anforderung beschreibt die Abhängigkeit von Gerätefunktionalität und Gerätetyp. Diese Anforderung basiert auf der Annahme, das der Funktionsumfang eines jeweiligen Gerätes durch seinen Typ spezifiziert werden kann. In Kombination mit einer agentenbasierten Steuerung schließt diese Abhängigkeit das Verhalten eines Agenten mit ein. Zu diesem Zweck werden *Entitäten* mit *EntitätsTypen* versehen.

EntitätsTypen definieren Fähigkeiten und Verhaltensweisen. Sie übernehmen zum einen die Kapselung von *RegelMengen* (siehe: 3.6.3) und hierarchischer *Prioritäten* (siehe: 3.6.10.2), und dienen zum anderen zur funktionalen Gruppierung von *Entitäten*. Eine Menge von *Entitäten* mit gleichem *EntitätsTyp* besitzt demnach die gleiche typabhängige Funktionsmenge. Die *Entität* eines Agenten kann mit mehreren *EntitätsTypen* versehen werden. Das folgende Tripel beschreibt den Aufbau eines *EntitätsTyps*.

- ▷ *EntitätsTypID* = Eindeutiger Bezeichner eines *EntitätTyps*
- ▷ *EntitätsTyp* = $\langle$*EntitätsTypID*, $\{$*RegelMengenID*$\}$, *Priorität*$\rangle$

3.6.6 Verhalten durch Rollen

Annahmen Sechs bis Acht beschreiben eine Abhängigkeit erwarteten Verhaltens von Situationen und Personenrollen. Die hieraus entstandenen Anforderung beschreiben die Fähigkeit zur dynamischen Anpassung des Verhaltens und Zuordnung von Zielmengen zu Personen. Laut [Lin01, S. 140] bezeichnen Rollen eine logische Gruppierung atomarer Aktivitäten. Nach Dastani in [Das08, S. 81-86] ermöglicht die Verwendung von Personenrollen die Definition gesammelter Fähigkeiten, die notwendig sind um globale Ziele zu erreichen. Zusätzlich dienen sie der Beschreibung von Passfähigkeiten zu Teamaktivitäten, einer Festlegung autoritärer Hierarchien und Verwaltung von Zugriffsberechtigungen. Die Verwendung abstrakter Rollen zur Beschreibung von Verantwortlichkeiten und Fähigkeiten bewirkt laut [HRHL01, S. 2f] zusätzlich eine Minimierung von Kommunikations- und Verhandlungskosten.

Jinsong Leng beschreibt in [LLJ08, S. 644] einige Anforderungen für die Realisierung eines Agententeams. Diese beinhalten Mechanismen zur Koordination, Kooperation und Strukturierung von Rollenmengen und Verfahren der Wissensteilung und Rollenzuweisung.

Die Anpassung eines Agentenverhaltens wird durch eine dynamische Verteilung von *EntitätsTypen* (in diesem Zusammenhang auch nur als *Rollen* bezeichnet) auf *Entitäten* entsprechenden Typs realisiert. Die Modellierung eines den Nutzer unterstützenden Verhaltens geschieht demnach direkt, innerhalb seiner situationsbezogenen zugeordneten *Rolle*. Das Ziel hierbei ist es, jeder Person in der Umgebung eine abstrakte *Rolle* und somit eine Menge unterstützender Verhaltensweisen und *Prioritäten* (gemäß dem Aufbau eines *EntitätsTyps* 3.6.5) zuzuordnen. Die Konzepte dieser Arbeit zur Koordination und Kooperation von Agenten werden in Abschnitt 3.6.10 vorgestellt. Die Strukturierung der Rollenmenge wird in 3.6.10.2 beschrieben. Die Verteilung von Informationen ist durch eine gemeinsame Wissensbasis, beschrieben in 3.6.2 , gewährleistet.
Das Konzept dieses Kontrollsystems zur parametrisierten Verteilung von *Rollen* beinhaltet folgende Aspekte:

1. Festlegung auf einen *EntitätsTyp*, als Träger einer *Rolle*
2. Spezifikation der *Rolle* (des zu verwendenden *EntitätsTyps)*
3. Festlegung einer Menge von erlaubten Aufenthaltsorten (*Zonen*) einer *Rolle*
4. Definition einer maximalen Anzahl von verteilten *Rollen*
5. Festlegung von Inkompatibilitäten zwischen verschiedenen *Rollen*

Über diese Art der Verteilung können beispielsweise Ubisense-*Entitäten* mit dem *EntitätsTyp „person"* als Träger einer *Rolle* spezifiziert werden. Eine Verwendung von *Rollen* ermöglicht eine hohe Abstraktionsebene für die Modellierung einer Unterstützung dynamischer Nutzerziele.

Die Verwendung abstrakter *Rollen* kommt unter anderem im *Jack*-Framework (beschrieben in 2.4.1) zum Einsatz. Eine detaillierte Auflistung von Konzepten und Beispielen zur Modellierung von *Rollen* kann [Rie00] entnommen werden.

▷ *Rolle* = Ein dynamisch verteilter *EntitätsTyp*

3.6.7 Erweiterung des Kontextes

Im vorherigen Abschnitt wurde bereits die Verwendung von Situations- und Zonenwissen erwähnt. Zur Erweiterung der globalen Wissensmenge und Erhöhung der Abstraktionsstufe formulierter Kontexte wird die Wissensbasis des Systems durch die Verarbeitung bestehender Kontexte um zusätzliches Wissen erweitert. Die folgenden zwei Verhaltensweisen sollen diese Aufgabe erfüllen.

Abstrahierung von Positionswissen Die Position eines Objektes im Raum ist ein zentraler *Kontext* bei der Steuerung *Intelligenter Umgebungen*. Die zonenbasierte Verarbeitung realer Positionsinformationen erhöht die Abstraktionsebene bei der Spezifikation von Handlungsbedingungen und verringert erheblich die Abhängigkeit der Verhaltensmodellierung zur realen Umgebung. Zu positionsbehafteten *Kontexten* zählen unter anderem Personenverteilungen oder lokale Abhängigkeiten von Geräten.

Die Verhaltensweise *zoneMapper* hat die Aufgabe vorhandene Wissensmengen über absolute Positionen um eine Zuordnung zu abstrakten *Zonen* zu erweitern. *Zonen* werden durch globale *Worldstates* von beliebigen Agenten definiert.

- ▷ *zoneMapper* = *EntitätsTyp* zur Abstraktion von Positionsinformationen
- ▷ *Zone* = Durch reale Positionsinformationen spezifizierter abstrakter Bereich

Wahrnehmung von Situationen Nach den in Abschnitt 3.2 beschriebenen Annahmen ist das erwartete Verhalten einer *Intelligenten Umgebung* abhängig von der aktuellen Umgebungssituation. Eine weitere Annahme beschreibt die Möglichkeit der Wahrnehmung dieser Situationen durch Ereignisse und Zustände. Der in dieser Annahme genannte Ansatz zur Situationserkennung wird in einer Menge paralleler *„situationPerceptor"*-Verhaltensweisen realisiert. Jede erkennbare Situation besitzt einen eigenen Agenten. Diese haben jeweils die Aufgabe das Eintreten und Verlassen eines bestimmten Nutzungsszenarios zu erkennen.

Eine weitere Aufgabe ist durch Annahme Acht beschrieben: *„Situationen beinhalten Personenrollen [...]"*. *situationPerceptors* sollen entsprechend einer Situation ihr Wissen über Rollenverteilungen im System veröffentlichen. Eine genauere Erläuterung zur Verwendung von *Rollen* wurde in Abschnitt 3.6.6 beschrieben.

- ▷ *situationPerceptor* = *EntitätsTyp* zur Verwaltung einer Umgebungssituation

3.6.8 Ausführung im Agenten-Rohling

Abschnitt 3.6.4 beschreibt das Konzept zur Verwendung eines Agenten für jede *Entität*. Hierzu wird für jede *Entität* des Systems ein eigener Agent synthetisiert. Da die Fähigkeiten und Verhaltensweisen einer *Entität* ausschließlich durch verbundene *EntitätsTypen* bestimmt sind, besitzt ein neu erstellter *Entitäts*-Agent lediglich ein latentes Grundverhalten und wird im Folgenden konzeptuell als *Agenten-Rohling* bezeichnet. Wooldridge beschreibt in [Woo09, S. 42] eine Funktion zur Synthese fähiger Agenten. Das in dieser Arbeit verfolgte Konzept ordnet auch unbekannten *Entitäten* einen Agenten zu. Auf diese Weise können *Agenten-Rohlinge* ohne Typ bereits als intelligente Schnittstelle zur Aktionsausführung in der realen Umgebung genutzt werden. Hinzugefügte Typen bestimmen anschließend den Umfang seiner letztendlichen Fähigkeiten.
Ein *Agenten-Rohling* besitzt die folgenden statischen *Grundfunktionen*:

1. Annahme einer *EntitätsID* (3.6.4)
2. Aufnahme von *EntitätsTypen* und mit ihm verknüpfter Informationen (3.6.5)

3. Laden, Entladen und Sortieren von *RegelMengen* (3.6.10.2)
4. Mechanismen zur entitäts- und entitätstypbasierten Verteilung von Zielen (3.6.11)
5. Veröffentlichung von Konfigurationsinformationen
6. Mechanismen für die Verwaltung von Wissensmengen

Der Aufbau eines *Agenten-Rohlings* kann durch das folgende Tripel dargestellt werden. Die Menge der *RegelMengen* (3.6.3) beschreibt hierbei die organisierte (3.6.10.2) Gesamtmenge typspezifischer Verhaltensweisen.

▷ *Agenten-Rohling*
$= \langle EntitätsID, \{EntitätsTyp\}, \{RegelMenge\}, \{Internes\ Belief\}, Grundfunktionen \rangle$

▷ *Grundfunktionen* = Statisch implementiertes Grundverhalten

3.6.9 Organisation von Agenten, Entitäten und EntitätsTypen

Die Organisation zur dynamischen Verwaltung von *Entitäten* und *EntitätsTypen* wird ähnlich dem in [HSB07] vorgestellten *MOISE*-Frameworks zur Koordination von Agentensystemen realisiert. Das *MOISE*-Framework bedient sich einer Schemaspezifikation von Rollen, Teams und Teamplänen. Die Verwendung dieses Frameworks ermöglicht die Verteilung und Koordination von Aufgaben auf einzelne Mitglieder eines Teams. Der Vorgang der Organisation wird durch eine zentrale Instanz namens OrgManager verwaltet. Aufgrund des dynamischen Charakters dieses Systems wurde von einer Nutzung statischer Schemata abgesehen. Für die Konzeption eines Kontrollsystems wurde das Prinzip einer zentralen Instanz zur Verteilung von *Entitäten* und *EntitätsTypen* übernommen.

Ein spezieller Agent (im Folgenden als *orgManager* bezeichnet) erstellt und beendet gemäß der aktuellen *Entitäts*-Menge *Agenten-Rohlinge* und übernimmt die dynamische Verteilung von *Entitäten* und *EntitätsTypen*.

▷ *orgManager* = Zentraler Agent zur Verwaltung von Agenten und *Entitäten*

3.6.10 Koordination von Verhaltensweisen und Ressourcen

Es existieren eine Menge Ansätze zur Koordination von Agentensystemen. Cohen und Levesqueue beschreiben in [CL91] den Ansatz der *Joint Intentions* und *Joint Beliefs*. Hierbei verfolgen Agenten so lange ein Ziel bis alle übereinstimmten, dass es erreicht, unmöglich oder irrelevant ist. Durch die globale Verteilung von Wissen, haben alle Agenten Informationen über die Konfiguration ihrer Umgebung. Eine Koordination durch eine zeitliche Anordnung von Plänen und einer Einteilung von Agenten in *Command-* und *Controlteams* wird in [Tid93] beschrieben. Hübner beschreibt in [HSB02] eine Koordination über Organisationsstrukturen, Fähigkeiten und Kontrolle von Zielen und Zugriffsberechtigungen. Abbildung 3.3 zeigt eine Übersicht über Organisationskonzepte in Multi-Agentensystemen.

		Joint intentions	Roles	Shared beliefs	Joint commitments	Global goals	Communication	Organisation/Team abstraction	Mutual beliefs	Social structure	inter-[team/group] relationships	Sub-teams	Social commitment	Groups	Interactions/scenes	Task structure	Organisational centred	Agent centred
Cohen & Levesque	*Joint intentions*	✔		✔		✔		✔										✔
Esteva et al	*Institutions*		✔			✔	✔		✔			✔		✔	✔	✔		
Ferber	*Roles and groups*		✔				✔	✔		✔				✔		✔	✔	
Fisher et al.	*Groups*						✔							✔				✔
Hübner	*Roles and permissions*		✔			✔	✔	✔		✔	✔	✔	✔	✔		✔	✔	
Pynadath & Tambe	*Team-oriented*		✔	✔			✔	✔		✔						✔	✔	
Tidhar, Cavedon & Rao	*Team-oriented*	✔	✔		✔	✔		✔		✔	✔	✔	✔					✔

Abbildung 3.3: Multi-Agenten Organisationskonzepte [Das08, S. 82].

Das in dieser Arbeit entwickelte Konzept folgt dem Ansatz der Koordination durch Organisationsstrukturen und Zugriffsberechtigungen zur Realisierung der in 3.3.2 geforderten Zuordnung von Autoritäten und des exklusiven Ressourcenzugriffs. Alle Agenten verfügen über eine gemeinsame *Beliefmenge.* Die Steuerung von Zugriffsberechtigungen wird über Zieltypen (ähnlich der *Jam*-Architektur (2.4.2) und entsprechend der sozialen Struktur (ähnlich der *Jack*-Architektur (2.4.1)) durch *Prioritäten* realisiert. Die Ausführung des Ansatzes soll innerhalb eines jeden Agenten geschehen und ohne zentrale Instanz auskommen. Im Folgenden werden beide Koordinationsmechanismen vorgestellt.

3.6.10.1 Zieltypen

Für die Koordination des Zugriffs auf *Entitäten* wird eine Unterscheidung von Zieltypen vorgenommen. Das *Jam*- und *Jadex*-Framework (beschrieben in 2.4.2 und 2.4.3) unterscheidet zwischen *Maintain*-, *Achieve*- und *Performzielen.* Für die Koordination von Zugriffen werden für dieses System *Achieve*- und *Maintainziele* verwendet. *Performziele* beinhalten das permanente Ausführen von Aktionen. Durch die eventbasierte Arbeitsweise des hier entwickelten Kontrollsystems wird dieser Zieltyp nicht realisiert. *Achieveziele* realisieren das einmalige Erreichen eines Entitätszustandes. Die Verwendung eines *Maintainzieles* beschreibt das Erreichen und Halten eines Entitätszustandes. Realisiert wird dies durch das Sperren einer *Entität* für nicht erlaubte Zugriffe. Die Verteilung von Berechtigungen für den Zugriff auf gesperrte *Entitäten* geschieht über *Prioritäten.* Durch die individuelle Verwaltung von Sperren, Zielarten und Zielverteilungen sollte das Kontrollsystem ohne eine zentrales Ressourcenmanagement auskommen. Abbildung 3.4 zeigt eine Übersicht über verwendete Zieltypen in BDI-Systemen.

- ▹ *Achieveziel* = Einmaliges Erreichen eines Entitätszustandes
- ▹ *Maintainziel* = Erreichen und Halten eines Entitätszustandes

	KAOS	Gaia	Jack	PRS	Jam	Jadex	Jason
achieve	x	x	x	x	x	x	x
maintain	x	x		x	x	x	
chease	x						
avoid	x						
optimise	x						
test			x	x			x
query					x	x	
perform					x	x	
preserve			x	x			

Abbildung 3.4: Zieltypen in BDI-Systemen nach [BPML04, S. 51] erweitert um *Jason*.

3.6.10.2 Hierarchien durch Prioritäten

Annahme Acht des Abschnitts 3.2 beschreibt eine Unterscheidung von Personenrollen in gewünschter Zielmenge und *Priorität*. Die Zielmenge wird durch die in 3.6.6 erläuterte dynamische Verteilung von *EntitätsTypen* realisiert. Die *Priorität* einer Personenrolle kann als Maß der Autorität im System betrachtet werden und dient zur Koordination von Berechtigungen und Hierarchien. *Prioritäten* werden zur Erzeugung einer absoluten Ordnung durch absolute Werte repräsentiert.

Priorisierte Sperrung von Entitäten *Rollen* werden durch *EntitätsTypen* realisiert. Zur Priorisierung einer *Rolle* werden *EntitätsTypen* mit absoluten *Prioritäten* versehen. Das Ausführen eines *Maintain*-Zieles führt zu einer Sperrung der Zielentität entsprechend der *Priorität* seines Absenderstyps. Auf diese Weise sind *Rollen* in der Lage Ressourcen entsprechend ihrer Autorität zu sperren. Nur eine *Rolle* mit mindestens gleicher *Priorität* ist somit Berechtigt diese Ressource zu nutzen. Verschwindet ein *EntitätsTyp*, werden die durch ihn gesperrten *Entitäten* automatisch entsperrt.

Priorisierte Organisation von RegelMengen Das Verhalten eines *Entitäts*-Agenten ist durch die Summe seiner typabhängigen *RegelMengen* bestimmt. Für eine Organisation mehrerer *RegelMengen* innerhalb eines Agenten wird eine Anordnung von Verhaltensmengen, auf Basis verknüpfter *Prioritäten* durchgeführt. Bei Überschneidungen von Auslöseereignissen und Kontextbedingungen in mehreren *RegelMengen* einer *Entität* wird somit immer das Verhalten des *EntitätsTyps* mit höchster *Priorität* ausgeführt. Bei der Verwendung mehrerer Personenrollen unterschiedlicher *Prioritäten* innerhalb eines Agenten wird somit stets die Reaktion der höchsten Personenrolle verwirklicht. Auf diese Weise kann das Verhalten mehrerer *Rollen* kombiniert und Verhaltensweisen niederer *Rollen* überschrieben werden.

- ▷ *Priorität* = Maß der Autorität zur Koordination von Berechtigungen

3.6.10.3 Koordination identischer EntitätsTypen

Die Ausführung von Aktionen geschieht durch eine Menge asynchron arbeitender Agenten. Durch die Verteilung von Personenrollen können mehrere Agenten den gleichen aktiven *EntitätsTyp* besitzen. Hierbei besteht die Gefahr, dass alle Agenten mit gleichem *EntitätsTyp* beim Eintreffen eines passenden Ereignisses die gleiche *Intention* besitzen. Eine Möglichkeit dieses Problem zu umgehen wäre das beschränken der Verwendung aller Personenrollen auf maximal eine aktive *Rolle* pro Typ. Dieser Ansatz würde eine direkte Modellierung von Rollenkombinationen und Gruppendynamiken erschweren. Eine weitere Möglichkeit ist das Sammeln und Koordinieren von Zielen an zentraler Stelle. Diese Art der Realisierung könnte durch die hohe Entitätsdynamik und die asynchrone Arbeit der einzelnen Agenten die Ausführung von Aktionen verzögern oder gar ein zusätzliches *Bottleneck* erzeugen. Die Anforderung einer parallelen, unabhängigen Aktionsausführung wäre zusätzlich nur eingeschränkt erfüllt.

Das in dieser Arbeit entwickelte System überträgt einzelnen Agenten die Koordination eintreffender und formulierter Ziele. Die in 3.6.11 beschriebenen Funktionen realisieren eine Abspeicherung des zuletzt erhaltenen Ziels. Es obliegt der Entscheidung des jeweiligen *EntitätsTyp* das mehrfache Eintreffen identischer Anweisungen zu unterdrücken oder zu verarbeiten. Für die indirekte Verteilung von Zielen wird die Ausführung durch eine einzige Personenrolle realisiert, um die gleichzeitige Verteilung von Zielen auf unterschiedliche *Entitäten* zu vermeiden. Eine weitere Möglichkeit der Koordination identischer Ziele wird in 5.3.2 beschrieben.

3.6.11 Verteilung von Zielen und Aufgaben

Die Verteilung von Zielen und Aufgaben innerhalb eines Multiagentensystems sollte mächtig und dennoch übersichtlich sein. Für dieses System wird eine identitäts- und typbasierte Zielverteilung unter Beachtung von *Prioritäten*, Sperrungen und Kosten realisiert.

3.6.11.1 Means-End Reasoning

G. Weiss beschreibt in [Wei00, S.29] eine Unterteilung des praktischen Reasoning in zwei Schritte. Zum einen die Zielgenerierung (*Deliberation*) zum anderen die Auswahl eines geeigneten Handlungspfades zur Erfüllung dieses Ziels (*Means-End Reasoning*). Der Prozess der Deliberation wird in *Jason* durch die Generierung der Menge ausführbarer Pläne realisiert. Für das *Means-End Reasoning* zur anschließenden Auswahl von Aktionen für das Erreichen eines Ziels können Pläne mit zusätzlichen Kosten- oder Nutzeninformationen versehen werden. S. Bader beschreibt in [Bad10, S. 8f] die Verwendung von Kosten für eine Modellierung von Nutzereinstellung.

Die Eingrenzung event- und kontextabhängiger abstrakter Nutzerziele wird durch eine situationsbedingte Zuordnung von Personenrollen realisiert. Aus diesem Grund bezieht sich die Verwendung von Kosten nicht auf die Wahl eines geeigneten abstrakten Ziels (*Deliberation*), sondern auf die Verwendung günstiger Ressourcen zu seiner Verwirklichung (*Means-End Reasoning*).

Eine Verwendung von Kosten im Deliberationsprozess würde individuelle Pläne für unterschiedliche *Entitäten* (Ressourcen) bedeuten. Aus diesem Grund wird das *Means-End Reasoning* nicht über die kostenbasierte Differenzierung von entitätsabhängigen Plänen sondern durch eine kostenbasierte Verteilung von Unterzielen an *Entitäten* realisiert. Kosten können abhängig von Zielen, Gerätemengen und *EntitätsTypen* definiert werden. Die Auswertung dieser Kosten geschieht als Teil der im Folgenden beschriebenen typbasierten Zielverteilung. Da der Aufwand des Systems für das automatische Erreichen eines Zustandes weniger relevant für eine Person ist, verkörpern Kosten eine hierarchische Anordnung bevorzugter Geräte bei bestimmten Zielen im Sinne eines Nutzers.

- ▷ *Kosten* = Aufwand für das Beseitigen eines bestimmten Zustandes

3.6.11.2 Konzeption einer entitäts- und entitätstypbasierten Zielverteilung

Ziele verfolgen das Erreichen eines teilweise definierten Weltzustandes. Durch die Anforderung einer einheitlichen Schnittstelle zur Aktionsausführung in Kombination mit der in 3.6.2 beschriebenen Darstellung von Informationen und der Verwendung der in 3.6.10.1 beschriebenen Zieltypen ist eine Zielverteilung durch *achieve*- und *maintainWorldstate* für das einmalige Erreichen und das Erreichen und Halten eines Zustandes entstanden.

Die Funktionalität zur Verteilung und Weiterleitung von Zielen ist in der Menge der *Grundfunktionen* eines *Agenten-Rohlings* verankert. Durch die Adressierung einer *EntitätsID* wird die Ausführung direkt an die entsprechende *Entität* übergeben. Wird anstelle einer *EntitätsID* eine *EntitätsTypID* verwendet erfolgt eine Weiterleitung des Ziels an eine *Entität* des entsprechenden Typs. Bei der Wahl dieser *Entität* werden Informationen über Sperrungen (3.6.10.2) und Kosten (3.6.11.1) mit einbezogen. Der *EntitätsTyp* des Absenders dient hierbei als Referenz auf seine *Priorität*. Zusätzlich soll eine Verteilung von Zielen an alle *Entitäten* eines entsprechenden Typs möglich sein. Die letztendliche Verarbeitung des Ziels durch die *EntitätsTypen* einer Zielentität erfolgt durch das interne Weiterleiten von *achieve*$\langle E, F, P, S \rangle$ an die Menge der *RegelMengen* angenommener *EntitätsTypen*.

- ▷ *achieveWorldstate* = $\langle E, F, P, S \rangle$ Für das einmalige Erreichen eines *Worldstates*
- ▷ *maintainWorldstate* = $\langle E, F, P, S \rangle$ Für das Erreichen und Halten eines *Worldstates*
- ▷ *achieve* = $\langle$*EntitätsID*, $F, P, S \rangle$ Für das direkte Ausführen innerhalb einer *Entität*
- ▷ E = Adressierung über *EntitätsID* oder *EntitätsTypID*
- ▷ F = Bezeichner eines Ziels
- ▷ P = Parametermenge eines Ziels
- ▷ S = Absender *EntitätsTypID*

3.6.12 Konzeptueller Aufbau des Kontrollsystems

Der folgende Abschnitt beschreibt den konzeptuellen Aufbau des Kontrollsystems unter Verwendung vorangegangener Zusammenfassungen.

Kontrollsystem = ⟨*orgManager*,{*Agenten-Rohling*},{*Worldstate*}, {*Adapter*}⟩

- ▷ *orgManager* = Zentraler Agent zur Verwaltung von Agenten und *Entitäten*
- ▷ *Agenten-Rohling*
  = ⟨*EntitätsID*,{*EntitätsTyp*},{*RegelMenge*},{*Internes Belief*},*Grundfunktionen*⟩
  - ▷ *EntitätsID* = Eindeutiger Bezeichner einer *Entität*
    - ▷ *Entität* = Ein beliebiges, eindeutig identifizierbares Ding
  - ▷ *EntitätsTyp* = ⟨*EntitätsTypID*,{*RegelMengenID*},*Priorität*⟩
    - ▷ *EntitätsTypID* = Eindeutiger Bezeichner eines *EntitätsTyps*
    - ▷ *RegelMengenID* = Eindeutiger Bezeichner einer *RegelMenge*
    - ▷ *Priorität* = Maß der Autorität zur Koordination von Berechtigungen
  - ▷ *RegelMenge* = ⟨*RegelMengenID*,{AgentSpeak-Plan}⟩
  - ▷ *Internes Belief* = Beliebiges individuelles Wissensprädikat
  - ▷ *Grundfunktionen* = Statisch implementiertes Grundverhalten
- ▷ *Worldstate* = ⟨*Subjekt*,*Key*,*Value*⟩
  - ▷ *Subjekt* = Beliebiger Träger von Attributen
  - ▷ *Key* = Eindeutiger Bezeichner einer Eigenschaft
  - ▷ *Value* = Wert dieser Eigenschaft
- ▷ *Adapter* = Schnittstelle zu einer Middleware

3.7 Zusammenfassung

In diesem Abschnitt wurde die konzeptuelle Entwicklung des Kontrollsystems beschrieben. Zu diesem Zweck wurden mithilfe alltäglicher Nutzungsszenarien im *Smart-Appliance-Lab* abstrakte Annahmen für die Realisierung eines beschriebenen Raumverhaltens getroffen. Unter Verwendung dieser Annahmen wurden konkrete Anforderungen diskutiert, mit denen anschließend verfügbare Frameworks für die Entwicklung von BDI-Systemen und Middlewares des *Smart-Appliance-Labs* evaluiert und selektiert wurden. Anschließend wurde auf Basis der nicht bereits abgedeckten Anforderungen ein Konzept zur Entwicklung eines BDI-Kontrollsystems mit folgendem Aufbau modelliert:

Für die Implementation des Kontrollsystems wird das *Jason*-Framework verwendet. Das Framework verfügt über eine javabasierte Schnittstelle für die Umgebung und ermöglicht die Entwicklung eines BDI-Multiagentensystems auf Basis von *AgentSpeak*. Als Schnittstelle zu veröffentlichten Geräte-*Services* und Sensorschnittstellen dienen *NetworkProxies*. Ihre Aufgabe ist unter anderem die Verbindung des Kontrollsystems mit dem *Tuplespace* des *helferlein-Systems*. Dieser übernimmt die Bereitstellung von Zustandsmengen und Steuerungsschnittstellen dynamischer Gerätemengen in Form von Zustandstupel und *CommandQueues*.

Jede *Entität* der Umgebung erhält einen individuellen, dynamisch synthetisierten Agenten. Diese Agenten besitzen initial nur eine Grundmenge von Mechanismen zur Verteilung von Zielen und der Organisation interner Zustände und *RegelMengen*. Die *Rolle* eines Agenten ist durch die *EntitätsTypen* seiner *Entität* definiert. Verknüpft mit diesen Typen erhält ein Agent seine Verhaltensweisen, Fähigkeiten über zugeordnete *RegelMengen* und eine Autorität in Form einer absoluten *Priorität*. Zur Erweiterung des *Kontextes* werden absolute Positionsinformationen abstrakten *Zonen* zugeordnet und Situationen im Raum erkannt.

Die Koordination des Verhaltens geschieht durch den Einsatz von *Prioritäten* und der Verwendung unterschiedlicher Zieltypen und Sperrzustände. Die Adressierung von Agenten für eine Verteilung von Zielen erfolgt direkt über die Verwendung von *Entitäten* oder indirekt durch *EntitätsTypen* und unter Beachtung von Kosten, Sperrzuständen und abstrakter *Zonen*. Die letztendliche Unterstützung eines Nutzers ist in priorisierten Personenrollen modelliert. Diese werden entsprechend aktueller Raumsituationen und *Zonen* auf die Personen der Umgebung verteilt.

4 BDI-SmartControl - Umsetzung

Dieses Kapitel befasst sich mit der Realisierung der in Kapitel 3 entwickelten Konzepte unter dem Namen *BDI-SmartControl*. Der folgende Abschnitt beschreibt die technische Umsetzung von Komponenten und Arbeitsweisen. Im Anschluss werden die notwendigen Arbeitsschritte zur Konfiguration und Erweiterung des *Controllers* anhand eines praktischen Beispiels beschrieben. Der letzte Abschnitt dieses Kapitels gibt einen Überblick über die bereits implementierten Verhaltensweisen dieses Systems.

4.1 Technische Umsetzung

Die technische Umsetzung des *BDI-SmartControl*-Systems basiert auf der Nutzung des *Jason*-Frameworks (beschrieben und evaluiert in 2.4.4 und 3.5) und der *Middleware*-Konzepte *Tuplespace*, *NetworkProxy* und *Infrastructure* (beschrieben und evaluiert in 2.3 und 3.4). Während die Umgebungsschnittstelle des Kontrollsystems innerhalb der *Java*-Umgebung des *Jason*-Frameworks realisiert wurde, basiert die Implementation der Agenten auf der logischen, agentenorientierten Programmiersprache *AgentSpeak* (vorgestellt in 2.2.4). Ausgelöst durch Ereignisse, bei entsprechenden Umgebungszuständen, formulieren und verteilen sie Ziele zur Steuerung und Konfiguration der Umgebung. Die folgenden Abschnitte beschrieben die implementationstechnische Umsetzung der einzelnen Komponenten des Systems. Begonnen wird mit der javabasierten Realisierung der Umgebungsschnittstellen. Anschließend, werden die entwickelten Mechanismen des Agentensystems vorgestellt. Zur Verdeutlichung der Zusammenhänge und Funktionsweisen umgesetzter Konzepte werden die abstrakten Erläuterung der folgenden Abschnitte durch konkrete Code-Beispiele begleitet. Die Syntax dieser Beispiele folg der in 2.2.4 vorgestellten *AgentSpeak*-Notation.

4.1.1 Informationsdarstellung

Die globale Wissensbasis des Kontrollsystems besteht aus einzelnen *Worldstate-Beliefs* (im Folgenden oft nur als *Worldstate* bezeichnet). Entsprechend der in 3.6.2 beschrieben Darstellung eines Zustands folgen die in *BDI-SmartControl* verwendeten Prädikate folgender Spezifikation:

`worldstate(Subjekt, Key, Value)`

`Subjekt`	Ein Prädikat als Träger eines Attributs
`Key`	Ein atomarer Term als Bezeichner des Zustands
`Value`	Ein Prädikat zur Beschreibung des Parameters

Beispiel Die Repräsentation einer angeschalteten Lampe mit *EntitätsID* `lamp1`.

```
worldstate(lamp1, on, true)
```

Vom Agenten erzeugte interne *Beliefs* besitzen das Präfix „`my`". Für die Spezifikation interner *Beliefs* in *RegelMengen* sind keine weiteren Richtlinien vorgegeben. Auf diese Weise können *EntitätsTypen* beliebige Ausdrücke als Variablen verwenden.

4.1.2 Javabasierte Umgebungsschnittstellen

Adaptoren dienen der Aufnahme von Informationen aus der Umgebung und der Realisierung von Aktionsausführungen. Die Funktionalität der *Java*-Schnittstelle des *Jason*-Frameworks ist in der Klasse `jason.environment.Environment` verankert. Die Umgebungsschnittstelle des *BDI-SmartControl*-Systems erbt und erweitert diese in der Klasse `SmartLabEnv`. Mit ihr sind die Verwaltung der globalen Wissensmengen realisiert und die, in den folgenden Abschnitten beschriebenen, *Adaptoren* verbunden.

Für die Verbindung eines *Adaptors* mit einem Entsprechenden Geräte-Service wurde der in 2.3 beschriebene *NetworkProxy* verwendet. Das folgende Beispiel zeigt die Schritte, der Verbindung eines *Ubisense-Adapters* unter Verwendung des *NetworkProxies* für einen *Ubisense*-Service mit Klasse `Ubisense.class` auf Host `codd` und Port `55556`. Der Service des Ubisense-Systems stellt hierbei die Position seiner Tags in Form von `UbisenseEvent`s zur Verfügung:

```
//Benötigte Klassen der SmartLab-Infrastruktur
import de.mmis.core.base.event.Observable;
import de.mmis.core.base.event.Observer;
import de.mmis.core.pojop.NetworkProxy;
import de.mmis.devices.ubisense.server.Ubisense;
import de.mmis.devices.ubisense.server.UbisenseEvent;

//Erstellen eines Proxies des Ubisense-Services
Ubisense ubisenseProxy = NetworkProxy.createProxy(Ubisense.class, "codd", 55556);

//Hinzufügen eines Listeners für den Empfang von Positions-Events
ubisense.addObserver(new Observer<UbisenseEvent>() {

    //Implementation der Observer-Methode.
    //In ihr werden Events empfangen und ausgewertet.
    public void notify(Observable<? extends UbisenseEvent> s, UbisenseEvent event){

        //Überprüfen des Events
        if (event == null) return;
        event.getEventType().equals(UbisenseEvent.Type.LOCATION_MESSAGE);

        //Auslesen von Positionsinformationen eines Ubisense-Tags
        String tagID = event.getTag_id_string();
        float[] position = {event.getX(), event.getY(), event.getZ()};

    }
});
```

4.1.2.1 Verwaltung globaler Wissensmengen

Das *Jason*-Framework verfügt direkt über keine gemeinsame Belief-Basis aller Agenten, besitzt aber Mechanismen zur globalen Verteilung von Wissen. Hierbei muss sicher gestellt werden, dass die globale Wissensmenge aller *Entitäten* konsistent ist. Zu diesem Zweck speichert die Umgebungsschnittstelle alle globalen Wissensinhalte und übernimmt die Verteilung an alle Agenten.
Für eine koordinierte Verbreitung von Wissen stehen dem Agenten folgende Aktionen zur Verfügung.

```
tellPercept(Worldstate)
untellPercept(Worldstate)
deletePerceptionsOfAgent(AgentName)
getPerceptions
```

Durch die Verwendung von `tellPercept` und `untellPercept` werden Wissensinhalte auf alle aktuellen und zukünftigen Agenten verteilt beziehungsweise entfernt. Die Aktion `deletePerceptionsOfAgent` entfernt alle *Worldstates* die von einem Agenten veröffentlicht wurden. Um alle aktuellen *Worldstates* zu erhalten wird `getPerceptions` verwendet.

4.1.2.2 Tuplespace-Adapter

Der folgende Abschnitt beschreibt die einzelnen Arbeitsbereiche des *Tuplespace-Adapters*. Der *Tuplespace-Adapter* stellt die Hauptschnittstelle zur Umgebung dar. Mit seiner Hilfe werden aktive Geräte im Raum gefunden, identifiziert und gesteuert. Zusätzlich ermöglicht der *Tuplespace* eine externe Steuerung und Konfiguration des Systems.

Einlesen von Geräten, Zuständen, Regeln und Anweisungen Der *Tuplespace-Adapter* bildet die Schnittstelle zum *Tuplespace* des *Smart-Application-Labs*. Mit seiner Hilfe werden Zustandstupel und Geräteinformationen aus dem *Tuplespace* gelesen. Für die Ausführung von Aktionen ist eine Verwaltung von *CommandQueues* realisiert. Während der Initialisierung des *Jason*-Frameworks wird unter Verwendung des *NetworkProxies* und der *Infrastructure* eine Verbindung zum *Tuplespace* hergestellt. Anschließend wird der *Tuplespace* mit folgenden Templates[1] gescannt:

(:status $?_{EntityID*}$ active)
$\rightarrow$ worldstate(room, hasEntity, EntityID)

(:device $?_{EntityTypeID*}$ $?_{EntityID*}$) und ($?_{EntityTypeID*}$ implements $?_{EntityTypeID*}$)
$\rightarrow$ worldstate(EntityID, entityTypeDefinition, EntityTypeID)

(:property-map-elem $?_{EntityID*}$ $?_{Key*}$ $?_{Value*}$)
$\rightarrow$ worldstate(EntityID, Key, Value)

[1] Tiefgestellte Werte verdeutlichen die Verteilung des Informationsgehalts aus Wildcard-Feldern („?")

`(:bdi-worldstate` $?_{worldstate(EntityID*, Key*, Value*)}$`)`
$\rightarrow$ `worldstate(EntityID, Key, Value)`

`(:bdi-ruleset` $?_{RuleSetID}$ $?_{(Rules)}$`)`
$\rightarrow$Neue *RegelMenge* mit *RuleSetID* und *Rules*

Aktive Geräte werden im *Tuplespace* durch ein `:status`-Tupel dargestellt. Statustupel enthalten die Identifikation eines aktiven, veröffentlichten Gerätes im *Tuplespace*. Sie werden verwendet, um die Menge der Geräte-*Entitäten* der Umgebung zu bestimmen. Gerätetypen werden durch `:device`-Tupel spezifiziert. Sie enthalten Informationen über die implementierten Serviceklassen des Gerätes. Abhängigkeiten unter Serviceklassen werden durch ein `implements`-Tupel beschrieben. Die Bezeichnungen von Serviceklassen wird zur Identifikation des Gerätetyps verwendet. Alle Serviceklassen eines im *Tuplespace* veröffentlichten Gerätes werden auf *EntitätsTypen* und mit ihnen verbundenen *RegelMengen* (beschrieben in 4.1.3.2) abgebildet. Durch die Abbildung der gesamten Klassenhierarchie auf *EntitätsTypen*, kann die Spezifikation typspezifischen Verhaltens auf jeder beliebigen Serviceebene erfolgen. Gerätezustände in `:property-map-elem`-Tupel werden direkt als *Worldstates* in das System übertragen.

Für eine direkte Spezifikation von Weltzuständen kann ein `:bdi-worldstate`-Tupel verwendet werden. Die verfügbaren *RegelMengen* der im folgenden Abschnitt 4.1.2.4 beschriebenen *RuleLibrary* werden durch die Nutzung eines `:bdi-ruleset`-Tupel erweitert. Hierzu wird die Menge der textbasierten *AgentSpeak*-Pläne hintereinander in das innere *Rules*-Tupel geschrieben. Die direkte Formulierung von Zielen geschieht durch die Nutzung eines `(:bdi-achieve GOAL)`-Tupel. Dieses Tupel wird als `worldstate(orgManager, perform, Goal)` vom Organisationsmanagement (4.1.3.3) aufgenommen und verarbeitet.

Der *AgentSpeak*-Interpreter des *Jason*-Systems arbeitet auf Basis einer prologähnlichen Programmiersprache. Variablen werden in *AgentSpeak* durch einen Großbuchstaben an erster Stelle symbolisiert. Terminale Symbole und Agentennamen müssen mit einem kleinen Buchstaben beginnen. Aus diesem Grund werden alle durch einen „*" gekennzeichnete Werte entsprechend angepasst. In Abschnitt 5.3.2 wird dieser Punkt näher erläutert.

Beispiel Ein im *Tuplespace* veröffentlichte Lampe besitzt folgende Tupel:

```
(:status Lamp1 active)
(:device ($type de.mmis.devices.eib.devices.lamp) Lamp1)
(:property-map-elem Lamp1 On true)
(:property-map-elem Lamp1 dimValue 1.0)
```

Nach der erfolgreichen Verarbeitung dieser Tupel entstehen folgende *Worldstates*:

```
worldstate(room,hasEntity,lamp1)
worldstate(lamp1,entityTypeDefinition,de.mmis.devices.eib.devices.lamp)
worldstate(lamp1,on,true)
worldstate(lamp1,dimValue,1.0)
```

Erfassen von Zustandsänderungen Nach dem erfolgreichen Auslesen des Umgebungszustands, werden *Notifications* für alle erwähnten *Templates* erstellt. Das Eintreffen eines neuen entsprechenden Tupel erzeugt eine Überprüfung des jeweiligen Templatebereichs und ein Update geänderter Zustände im Kontrollsystem. Durch das Scannen des kompletten Templatebereichs soll das Erfassen entfernter Zustände ermöglicht werden. Das Hinzufügen („+") und Entfernen („-") von Weltzuständen löst globale *Events* der folgender Form im BDI-*Controller* aus.

```
+worldstate(EntityID, Key, Value)
-worldstate(EntityID, Key, Value)
```

Events dieser Form können in *BDI-SmartControl* als Initiator für eventbasiertes Verhalten durch Gerätezustandsänderungen verwendet werden.

Ausführung von Geräteaktionen Der *Tuplespace-Adapter* verwaltet die *Command-Queues* aller Geräte, die über `:status`-Tupel dem System hinzugefügt wurden. Als Referenz auf einen jeweiligen *CommandQueue* dient die im System vergebene *EntityID*. Auf diese Weise ist eine *EntitätsID*-adressierte Aktionsausführung möglich. Die Verwendung der internen Funktion `achieve(EntityID, FunctionName, ParameterList)`im Aktionsteil eines *AgentSpeak*-Plans, veranlasst das System ein Tupel folgender Form in den durch `EntityID` referenzierten *CommandQueue* zu schreiben.

```
(:command-queue-elem EntityID* QueueID[2] (FunctionName ParameterList))
```

4.1.2.3 Ubisense-Adapter

Das *Ubisense*-System misst Entfernungen zu *Ubisense*-Tags und dient zur dreidimensionalen Erfassung von Positionsinformationen beweglicher Objekte (genauer erläutert in [Ubi11]). Der *Ubisense-Adapter* dient als Empfänger für *Events* des *Ubisense*-Systems. Um die Belastung für die Agenten des *Jason*-Frameworks gering zu halten, sammelt der *Adapter* Positionsinformationen und aktualisiert das Wissen des Agentensystems in kurzen Zeitintervallen durch den Mittelwert erfasster Positionen in folgendem *Worldstate*:

```
worldstate(UbisenseTagID, position, [X, Y, Z])
```

Neu erkannte Tags werden als eigene *Entitäten* und dem *EntitätsTyp* `person` im System angemeldet. *Entitäten* mit dem *EntitätsTyp* `person` werden im Allgemeinen als Halter einer Personenrolle verwendet. *Ubisense-Tags* dienen somit, ähnlich dem in 1.2 beschriebenen „*Personal Aura*"-Artefakt, zur Wahrnehmung von Personen im System und als Träger dynamisch verteilter *Rollen*.

[2] `QueueID` beschreibt eine fortlaufende Nummer zur sequenziellen Anordnung der Elemente eines *Tuplespace-Queues* erläutert in [BRK11].

4.1.2.4 RuleLibrary-Adapter

Die Aufgabe der *RuleLibrary* besteht in der Verwaltung von *RegelMengen* und in der Unterstützung von Agenten beim Laden und Entladen von Verhaltensweisen. Sie besitzt alle dynamisch verfügbaren *AgentSpeak-RegelMengen* und *RegelMengenIDs* in einer `HashMap`. Agenten können durch das Ausführen der internen Funktion `loadRules(RuleSetID, EntityTypeID)` *RegelMengen* erfragen. Die `RuleSetID` wird hierzu mit den *RegelMengenIDs* (*Key*) der `HashMap` abgeglichen. Besitzt die *RuleLibrary* eine passende *RegelMenge* wird sie dem Absenderagent des Funktionsaufrufs in Form von *Beliefs* zur Verfügung gestellt. Das Laden und Entladen von Regeln erzeugt (+) und entfernt (-) hierbei folgende *Beliefs* innerhalb eines Agenten.

```
loadRules(RuleSetID,EntityTypeID)
          → {+dynamicRule(Rule, Label, TypedRuleSetID)}

unloadRules(RuleSetID,EntityTypeID)
          → {-dynamicRule(Rule, Label, TypedRuleSetID)}
```

`Rule` Beinhaltet einen mit `Label` annotierten *AgentSpeak*-Regelterm

`Label` Beinhaltet die `TypedRuleSetID` und einen Indexwert

`TypedRuleSetID` Identifikation zusammengesetzt aus `EntityTypeID+''_''+RuleSetID`

Die Referenz auf den *EntitätsTyp* ermöglicht eine Trennung gleicher *RegelMengen* unterschiedlicher *EntitätsTypen*. Das Entladen von *RegelMengen* geschieht durch Aufrufen von `unloadRules(RuleSetID, EntityTypeID)` und dem Entfernen entsprechender *Beliefs*. Weiterführende Schritte zum Laden und Entladen von *RegelMengen* innerhalb eines Agenten werden in Abschnitt 3.6.8 beschrieben.

Ist eine *RegelMengenID* nicht in der *RuleLibrary* vorhanden, wird versucht eine passende *RegelMengen*-Datei mit Bezeichnung ***RuleSetID***`.asl` im Ordner „ `./planSets/`" des *Controllers* einzulesen. Die Datei zur Spezifikation einer *RegelMenge* besteht aus statischen *Worldstates* und einer Menge von *AgentSpeak*-Regeln. Die statischen *Worldstates* können zu Beginn einer Datei definiert werden. Beim Einlesen einer Datei werden diese automatisch der allgemeinen Beliefmenge des Agentensystems hinzugefügt. Anschließend folgt die Menge von *AgentSpeak*-Regeln. Diese werden in die Verwaltung *der RuleLibrary* geladen. Die Entwicklung einer *RegelMenge* kann durch *Eclipse* [The11] und dem *Jason*-Eclipseplugin [BH11] unterstützt werden.

In Kombination mit dem *Tuplespace* kann für eine Erweiterung der *RuleLibrary* auch das in Abschnitt 4.1.2.2 beschriebene `(:bdi-ruleset` $?_{RuleSetID}$ $?_{(Rules)}$`)`-Tupel verwendet werden. Die Pläne im *Rules*-Tupel sind, solange das Tupel im *Tuplespace* liegt, unter der angegebenen *RuleSetID* verfügbar. Dieses Verfahren ermöglicht eine Veröffentlichung und Verwendung von Verhaltensweisen zur Laufzeit und durch beliebige, mit der Kommunikationsinfrastruktur verbundene, Quellen.

4.1.3 Agenten, Entitäten, EntitätsTypen und ihre Organisation

Der folgende Abschnitt beschreibt die Umsetzung der konzeptuell beschriebenen *Entitäten* (3.6.4), *EntitätsTypen* (*3.6.5*) und *Rollen* (3.6.6). Zusätzlich wird das System zur Organisation dieser Elemente (3.6.9) vorgestellt.

4.1.3.1 Entitäten

Entitäten bezeichnen laut Abschnitt 3.6.4 ein beliebiges identifizierbares Ding. Das Vorhandensein einer *Entität* in *BDI-SmartControl* wird durch den folgenden Weltzustand repräsentiert.

```
worldstate(room, hasEntity, EntityID)
```

Die `EntityID` ist hierbei der eindeutiger Bezeichner der *Entität*. *Entitäten* selbst werden im Kontrollsystem lediglich als eine Identität betrachtet. Aus diesem Grund wird *EntitätsID* im Folgenden nur als *Entität* bezeichnet.

Beispiel Die Anwesenheit einer Lampe mit der Bezeichnung `lamp1` wird wie folgt veröffentlicht.

```
worldstate(room, hasEntity, lamp1)
```

4.1.3.2 EntitätsTyp und abhängige Attribute

EntitätsTypen dienen laut 3.6.5 zur Kapselung von Verhaltensweisen und *Prioritäten*. Die Zuordnung eines *EntitätsTyps* zu einer *Entität* wird durch folgenden *Worldstate* beschrieben:

```
worldstate(EntityID, entityTypeDefinition, EntityTypeID)
```

Die dynamische Zuordnung eines *EntitätsTyps* zu einem *EntitätsTyp* im Sinne einer *Rolle* (beschrieben in 3.6.6) geschieht durch die Verwendung von:

```
worldstate(AType, roleDefinition, role(RType,Zones,Max,ITypes))
```

Die genaue Bedeutung der einzelnen Parameter und ein Beispiel für die Verwendung des *Worldstates* wird in Abschnitt 4.1.3.3 beschrieben.

`EntityTypeID`, `AType` und `RType` sind Bezeichner (*EntitätsTypID*s) eines *EntitätTyps*. Ein *EntitätsTyp* wird durch keine feste Datenstruktur repräsentiert, sondern definiert eine aus abhängigen *Worldstates* entstandenen Wissensmenge. Der Aufbau dieser Wissensmenge entspricht der in 3.6.5 beschriebenen Form. Die Abhängigkeit einzelner *Worldstates* ist durch die Verwendung der *EntitätsTypID* realisiert. Attribute wie *RegelMengen* und *Prioritäten* werden durch folgende *Worldstates* hinzugefügt.

```
worldstate(EntityTypeID, priority, Priority)
worldstate(EntityTypeID, ruleSet, RuleSetID)
```

Die `Priority` eines *EntitätsTyps* ist als absoluter numerischer Wert definiert. Werden mehrere *Prioritäten* eines *EntitätsTyps* spezifiziert, gilt automatisch der höchste aller Werte. *Prioritäten* dienen gemäß dem in 3.6.10.2 beschriebenen Konzept zur Koordination von Zugriffsberechtigungen und der Anordnung von *RegelMengen.* Die `RulesSetID` ist eine Referenz auf eine zusätzliche *RegelMenge* eines *EntitätTyps.* Durch sie werden das Verhalten und die Fähigkeiten eines Typs gebildet. Typen ohne ein `RuleSet`-*Worldstate* besitzen mindestens die `RuleSetID` identisch zur `EntityTypeID`. Für eine eindeutige Zuordnung interner *Beliefs* zu *EntitätsTypen,* können abhängige Wissensinhalte mit der Annotation `[entityType(EntityTypeID)]` versehen werden. Auf diese Weise können alle *Beliefs,* unabhängig von ihrem Inhalt, durch die Angabe der passenden Annotation selektiert werden.

Neben der Angabe von *RegelMengen* und *Prioritäten* existiert die Möglichkeit typabhängige Kosten als globale *Worldstates* zu spezifizieren. Das Subjekt einer Kostendefinition bezieht sich hierbei aber auf die Realisierung eines Ziels und bietet nur zusätzlich die Möglichkeit *EntitätsTypen* als Parameter zu spezifizieren. Kostendefinitionen besitzen folgendes Schema:

```
worldstate(Achieve, cost, Cost)
```

`Achieve` Einem Ziel, dessen Parameter denen eines `achieve`-Ziels (erläutert in 4.1.4) entsprechen. Bei der Festlegung der Parameter können an beliebigen Stellen Wildcards („_") verwendet werden.

`Cost` Ein absoluter numerischer Wert

Die *EntitätsTypID* ist konzeptuell der eindeutiger Bezeichner eines *EntitätsTyps.* Das Kontrollsystem betrachtet einen *EntitätsTyp* als Identität, der eine Menge von *Worldstates* zugeordnet ist. Aus diesem Grund wird die *EntitätsTypID* im Folgenden auch als *EntitätsTyp* bezeichnet.

Beispiele Der *EntitätsTyp* `de.mmis.devices.eib.devices.lamp` mit *Priorität* `0` und der *RegelMenge* `lampRules` für `lamp1` besteht aus folgenden *Worldstates:*

```
worldstate(lamp1, entityTypeDefinition, de.mmis.devices.eib.devices.lamp)
worldstate(de.mmis.devices.eib.devices.lamp, priority, 0)
worldstate(de.mmis.devices.eib.devices.lamp, ruleSet, lampRules)
```

Die Verwendung von `lamp1` hat für einen *EntitätsTyp* `person` die Kosten `3`:

```
worldstate(achieve(lamp1, _, _, person), cost, 3)
```

4.1.3.3 Organisation

Abschnitt 3.6.9 beschreibt die Nutzung eines zentralen Organisationsmanagements (*orgManager*) zur Koordination von Agentenmengen und Verknüpfung von Agenten, *Entitäten* und *EntitätsTypen.*

Der *orgManager* ist als *AgentSpeak*-Agent implementiert und verarbeitet die folgenden globalen *Beliefs* des Systems.

```
worldstate(room, hasEntity, EntityID)
worldstate(EntityID, entityTypeDefinition, EntityTypeID)
worldstate(EntityTypeID, priority, Priority)
worldstate(EntityType, ruleSet, RuleSet)
worldstate(orgManager, perform, Goal)
```

Entitäten, die nicht bereits an einen Agenten vergeben sind, werden automatisch neu erstellten *Agenten-Rohlingen* (`roleWorker`) zugeordnet. Anschließend werden alle zur *Entität* gehörenden Informationen über *EntitätsID, EntitätsTypen*, *Prioritäten* und *RegelMengen* durch spezielle *Beliefs* der internen *Belief*-Menge des Agenten hinzugefügt. Dieser Vorgang ermöglicht die selbstständige Steuerung interner Initialisierungsprozesse durch den Agenten (genauer erläutert in 4.1.4). Die Verwendung des `Goal-`*Worldstates* veranlasst den *orgManager* zur Ausführung des Beschriebenen `Goals`. Der *orgManager* verfügt hierfür über die Mechanismen der Zielverteilung eines `roleWorker`, beschrieben in 4.1.4.3.

Abschnitt 3.6.6 beschreibt das Konzept zur Verhaltensdynamik durch *Rollen*. Neben der direkten Verteilung von *EntitätsTypen* auf *Entitäten* beherrscht der *orgManager* (entsprechend der in 3.6.6 beschriebenen Aspekte zur Verteilung von *Rollen*) die Fähigkeit zur dynamischen Vergabe von *Rollen*. Hierzu verarbeitet er folgendes *Belief*.

```
worldstate(AType, roleDefinition, role(RType, Zones, Max, ITypes))
```

`AType` Beschreibt den *EntitätsTyp*, der von einer *Entität* besessen werden muss, um Träger dieser *Rolle* zu sein.

`RType` Beschreibt den *EntitätsTyp*, der als *Rolle* vergeben werden soll.

`Zones` Definiert eine Liste abstrakter *Zonen,* in denen sich die *Entität* aufhalten darf, um Träger dieser *Rolle* zu sein.

`Max` Gibt die maximale Anzahl der gleichzeitig aktiver *Rollen* dieses Typs an.

`ITypes` Definiert eine Liste von *Rollen*, die von dieser *Rolle* verdrängt werden.

Beispiel Das folgende Beispiel beschreibt die dynamische Verteilung einer Vortragsrolle `lec` auf eine Person `pers`. Der Vortragende darf sich im Vortragsbereich `rost` oder im Bereich der Sitze `seat` aufhalten. Zu jedem Zeitpunkt existiert maximal `1` Vortragender im Raum. Ein Vortragender ist nicht gleichzeitig ein Zuhörer `list`.

```
worldstate(pers, roleDefinition, role(lec, [rost, seat], 1, [list]))
```

4.1.4 Agenten-Rohling, Koordination und Zielverteilung

Abschnitt 3.6.4 beschreibt eine Zuordnung von Agenten zu jeder *Entität* des Systems. Hierfür werden vom *orgManager* (beschrieben in 4.1.3.3) *AgentSpeak*-Agenten des Typs *roleWorker* (definiert in `roleWorker.asl`) synthetisiert. Die Aufgabe des *roleWorkers* ist die Aufnahme einer *Entität*, *EntitätsTypen*, *Prioritäten* und *RegelMengen*. Die Aufnahme von *Entität*, *EntitätsTypen* und abhängiger Informationen geschieht durch das Hinzufügen folgender interner *Beliefs* durch den *orgManager:*

```
+myEntityID(EntityID)
+myEntityType(EntityTypeID)
+myEntityTypePriority(EntityTypeID, Priority)
+myEntityRuleSet(EntityTypeID, RuleSetID)
```

Beliefs für die Konfiguration eines *roleWorkers* beginnen mit einem „`my`". Auf diese Weise soll eine Abgrenzung zwischen *EntitätsTyp*-Wissen und Konfigurations-*Beliefs* möglich sein. Die internen Konfigurations-*Beliefs* bilden neben dem Auslösen von Agentenprozessen eine Menge von Referenzen, mit deren Hilfe Identitätsinformationen in den entitätsunabhängig modellierten *EntitätsTyp-RegelMengen* genutzt werden können.

Grundfunktionen Der *AgentSpeak-Agent roleWorker* besitzt eine Menge statischer, latenter *Grundfunktionen* (konzeptionell beschrieben in 3.6.8). Die *Grundfunktionen* realisieren eine Vielzahl der in 3.6 beschriebenen Verfahren zur Verhaltenskoordination. Ziel dieser Grundfunktionalität ist es, das anschließende Modellieren dynamisch hinzugefügter Verhaltensweisen so abstrakt und übersichtlich wie möglich zu gestalten. Die folgenden Abschnitte beschreiben die einzelnen Funktionsbereiche und verweisen auf die entsprechend verwendeten Konzepte.

4.1.4.1 Veröffentlichung von Konfigurationsinformationen

Die Veröffentlichung von Konfigurationsinformationen bildet eine Menge interner *Beliefs* auf eine Menge globaler *Worldstates* ab. Mit ihrer Hilfe haben alle Agenten des Systems Informationen über die *Entitäten* ihrer Umgebung. Folgende *Beliefs* werden als *Worldstates* veröffentlicht.

```
myEntityID(E)
          → worldstate(AgentName, committedEntity, E)

myEntityType(EType)
          → worldstate(AgentName, committedEntityType, EType)

myEntityPriority(EType, Prio)
          → worldstate(EType, priority, Prio)

myMaintain(Prio)
          → worldstate(AgentName, maintained, Prio)
```

```
myUsedBy(UserAgentName)
          → worldstate(AgentName, usedBy, UserAgentName)

myActiveRuleSet(RuleSetID)
          → worldstate(AgentName, hasRuleSet, RuleSetID)
```

Der Name eines Agenten (`AgentName`) und seine *EntitätsID* (`E`) sind im Allgemeinen identisch. Sollte die Systembezeichnung des Agenten dennoch abweichen, kann dies durch die Verwendung von `AgentName` dargestellt werden. Die globale Verteilung von Konfigurationen dient, ähnlich der in [CL91] beschriebenen *Joint-Beliefs*, zur Realisierung einer Wahrnehmung der globalen Systemkonfiguration durch einzelne Agenten.

4.1.4.2 Laden, Entladen und Sortieren von RegelMengen

Ein *roleWorker* ist in der Lage selbstständig *RegelMengen* zu laden oder zu entladen. Initiiert wird dieses Verhalten durch das Hinzufügen und Entfernen des folgenden *Beliefs*.

```
myEntityRuleSet(EntityTypeID, RuleSetID)
```

Das Hinzufügen eines derartigen *Beliefs* veranlasst den Agenten die in Abschnitt 4.1.2.4 beschriebene `loadRules(RuleSetID, EntityTypeID)`-Funktion der *RuleLibrary* aufzurufen. Diese füllt die interne *Belief*-Menge des Agenten mit der entsprechenden Menge folgender *Beliefs*

```
dynamicRule(Rule, Label, TypedRuleSetID)
```

Die in diesen *Beliefs* spezifizierte `Rule` wird automatisch der *RegelMenge* des Agenten hinzugefügt. Zusätzlich zur direkten Konfiguration von *RegelMengen* versucht jeder Agent beim Erhalt eines neuen *EntitätTyps* eine *RegelMenge* mit *RegelMengenID* gleich *EntitätsTypID* zu laden.

Im Anschluss an das Hinzufügen einer *RegelMenge* werden alle *RegelMengen* des Agenten gemäß der *Prioritäten* ihrer *EntitätsTypen* sortiert. Dieser Vorgang sorgt für die in 3.6.10.2 beschriebene prioritätsbasierte Staffelung von Verhaltensweisen.

Durch das Entfernen eines `myEntityRuleSet(EntityTypeID, RuleSetID)`-*Beliefs* wird die entsprechende `unloadRules(RuleSetID, EntityTypeID)`-Funktion aufgerufen. Diese entfernt zugehörige `dynamicRule(Rule, Label, TypedRuleSetID)`-*Beliefs*, woraufhin abhängige Regeln im Agenten entfernt werden.

Für das Erfüllen von Zielen beim Laden oder Entladen einer *RegelMenge* können folgende zwei Unterziele innerhalb einer *RegelMenge* spezifiziert werden. Diese werden automatisch beim Hinzufügen und Entfernen von Regelsätzen ausgelöst.

```
+!init(RuleSetID)
+!exit(RuleSetID)
```

4.1.4.3 Entitäts- und entitätstypbasierten Verteilung und Koordination von Zielen

Abschnitt 3.6.11 beschreibt das Konzept zur entitäts- und entitätstypbasierten Verteilung von Zielen. Die folgenden Mechanismen implementieren die Konzepte zur Koordination von Verhaltensweisen durch Zieltypen (3.6.10.1) und *Prioritäten* (3.6.10.2), einem kostenbasierten Means-End Reasoning (3.6.11.1) und der Handhabung identischer Ziele gleicher *EntitätsTypen* (3.6.10.3). Ein *roleWorkers* besitzt unter anderem die folgenden, nutzbaren Unterziele und *Beliefs* zur Steuerung der entitäts- und entitätstypbasierten Verteilung von Zielen:

```
+!achieveWorldstate(E, Key, Value, SenderEntityType)
+!maintainWorldstate(E, Key, Value, SenderEntityType)
+myMaintain(Priorität)
```

Je nach Zieltyp kann ein sperrendes `maintain`- oder entsperrendes `achieveWorldstate`-Unterziel genutzt werden. Die Annahme eines `maintainWorldstate`-Ziels durch die Zielentität erzeugt automatisch entsprechend priorisierte Sperrzustände (`myMaintain`) innerhalb der *Entität*. Sperrzustände können durch das Ausführen eines `achieveWorldstate`-Ziels in einer gesperrten *Entität* oder das Ausführen eines Ziels mit `unlockMaintain` als `Key` aufgehoben werden. In beiden Fällen benötigt der aufrufende Agent eine entsprechend hohe *Priorität*. Für die Adressierung einer Ziel-*Entität* kann für `E` eine *EntitätsID* oder eine *EntitätsTypID* genutzt werden.

Entitätsbasierte (direkte) Zielverteilung Durch die Angabe einer *EntitätsID* erfolgt eine direkte Zielvergabe. Die Verteilungsfunktion übernimmt hierbei automatisch das Auffinden der *Entität* und das Weiterleiten entsprechender `Key-Value`-Ziele an ihren Agenten. Der `SenderEntityType` dient als Referenz auf die *Priorität* des Absenders. Die Bestimmung der Berechtigung zur Ausführung von Zielen in gesperrten *Entitäten* erfolgt gemäß dem in 3.6.10.2 beschriebenen Verfahren.

EntitätsTypbasierte (indirekte) Zielverteilung Die Adressierung eines *EntitätsTyps* veranlasst die Weiterleitung des Ziels an eine *optimale Entität* entsprechenden Typs. Die Auswahl einer *optimalen Entität* geschieht unter Beachtung von Sperrzuständen und Kosteninformationen. Die indirekte Adressierung betrachtet nur *Entitäten*, die unterhalb der eigenen Prioritätshöhe gesperrt wurden. Auf diese Weise werden eigene oder durch ebenbürtige *Rollen* gesperrte Geräte nicht verändert. Das mehrmalige Ausführen eines `achieve`-Ziels, zeigt somit immer auf die gleiche, günstigste nicht selbst gesperrte *Entität*. Das mehrmalige Ausführen eines `maintain`-Ziels sperrt nach und nach alle freien *Entitäten*. Die Erweiterung `All` ermöglicht eine Verteilung des Ziels an alle nicht gesperrten *Entitäten* entsprechenden Typs.

```
+!achieveWorldstateAll(EntityType, Key, Value, SenderEntityType)
+!maintainWorldstateAll(EntityType, Key, Value, SenderEntityType)
```

Für das manuelle Erfragen der *optimalen EntitätsID* zur Vergabe eines Ziels können die folgenden Testziele formuliert werden (Aus Platzgründen wurde `EntityType` durch `EType` ersetzt):

```
+?getCheapestUnlockableEntity(EType,Key,Value,SenderEType,ER,CR)
+?getCheapestLowerUnlockableEntity(EType,Key,Value,SenderEType,ER,CR)
```

Das erste der beiden Ziele bezieht durch die eigene *Priorität* gesperrte *Entitäten* bei der Auswahl des Optimums mit ein. Das zweite Ziel betrachtet nur gesperrte *Entitäten* für alle *Prioritäten* unterhalb der eigenen Autorität. Die Parameter `ER` und `CR` enthalten die resultierende *EntitätsID* und zugehörige *Kosten*.

Die indirekte Zielverteilung unterliegt einer zusätzlichen Richtlinie. Die Nutzung einer indirekten Zielverteilung innerhalb eines mehrfach verteilten *EntitätsTyps* wird vom System unterdrückt, sofern die *EntitätsID* des Agenten nicht das Präfix einer sortierten Liste aller *EntitätIDs* mit gleichem Typ ist. Auf diese Weise wird sichergestellt, das beim Eintreffen eines *Events*, lediglich ein Vertreter mehrerer identischer *EntitätsTypen* reaktiv ein Ziel verfolgt. Die eigene Unterdrückung kann innerhalb eines *EntitätsTyps* durch das Hinzufügen oder Entfernen von `myUnsuppressed(EntityType)` aktiviert und deaktiviert werden.

Interne Zielverarbeitung Die interne Zielverarbeitung durch die *EntitätsTypen* einer *Entität* erfolgt durch die interne Weiterleitung eines `achieve`-Ziels. Die Parameter des `achieve`-Ziels entsprechen hierbei denen des erhaltenen `maintainWorldstate`- beziehungsweise `achieveWorldstate`-Ziels nach einer erfolgreichen Überprüfung der Berechtigung des Absenders und Konfiguration von Sperrzuständen.

```
+!achieve(EntityID, Key, Value, SenderEntityType)
```

Für die Verarbeitung innerhalb eines *EntitätsTyps* ist es lediglich notwendig diese `achieve`-Ziele als auslösende *Events* eines Plans zu nutzen. Eine abgelehnte Zielvergabe wird als `achieveFailed`-Unterziel zum Absenderagenten zurückgesendet.

```
+!achieveFailed(EntityID, Key, Value, SenderEntityType)
```

Das interne *Belief* `lastAchieved` wird automatisch der erfolgreichen Ausführung eines Ziels aktualisiert („`-+`").

```
-+lastAchieved(EntityID, Key, Value, SenderEntityType)
```

Mit ihm ist es möglich das gleichzeitige Eintreffen identischer Ziele zu erkennen. Zusätzlich veröffentlicht eine *Entität* mit `worldstate(EntityID, usedBy, AgentName)` den Namen des letzten Nutzers.

Beispiel Ein *Entität* `Ubi1` mit *EntitätsTyp* `listener` besitzt *Priorität* `4`, *Entität* `Ubi2` mit `lecturer` *Priorität* `5`:

```
worldstate(listener, priority, 4)
worldstate(lecturer, priority, 5)
```

Entitäten `lampX` mit $X=\{1,...,6\}$ besitzen *EntitätsTyp* `lamp`:

```
worldstate(room, hasEntity, lampX)
worldstate(lampX, entityTypeDefinition, lamp)
```

EntitätsTyp `lamp` kann folgendes Ziel verarbeiten:

```
+!achieve(EntityID, dimLight, Dimvalue, SenderEntityType) <- ...
```

Für den *EntitätsTyp* `lecturer` existieren die folgenden Kosten für die Nutzung einiger Lampen:

```
worldstate(achieve(lamp5, _, _, lecturer), cost, 1)
worldstate(achieve(lamp6, _, _, lecturer), cost, 2)
```

`lecturer` möchte die günstigste `lamp` und zusätzlich `lamp4` dimmen und sperren:

```
lecturer: !maintainWorldstate(lamp, dimLight, 0.5, lecturer)
          !maintainWorldstate(lamp4, dimLight, 0.5, lecturer)
```

Die Zielentitäten aktualisieren die folgenden internen Zustände:

```
lamp5:    -+myMaintain(5)
          -+myUsedBy(Ubi2)
          -+lastAchieved(lamp5, dimLight, 0.5, lecturer)
lamp4:    -+myMaintain(5)
          -+myUsedBy(Ubi2)
          -+lastAchieved(lamp4, dimLight, 0.5, lecturer)
```

`listener` möchte alle Lampen voll beleuchten:

```
listener: !achieveWorldstateAll(lamp, dimLight, 1.0, listener)
```

Und erreicht alle Lampen ($Y = X \setminus\{4,5\}$), die nicht durch den höher priorisierten `lecturer` gesperrt sind.

```
lampY:    -+myUsedBy(Ubi1)
          -+lastAchieved(lampY, dimLight, 1.0, listener)
```

4.1.4.4 Verwaltung von Wissensmengen

Ein neu synthetisierter *roleWorker*-Agent ruft automatisch die Aktion `getPerceptions` (beschrieben in 4.1.2.1) auf um alle aktuellen globalen *Worldstates* zu erhalten. Zusätzlich besitzt der Agent die folgenden aufrufbaren Unterziele zum Entfernen von *EntitätsTypen* und dem Wissen anderer Agenten.

```
+!die
+!removeEntityType(EntityType)
+!removeKnowledgeAboutAgent(AgentName)
```

Die `+!die` Funktionalität eines Agenten entlädt alle *RegelMengen* und löscht alle Informationen über und von diesem Agenten. Abschließend wird der Agent vom System entfernt. `+!removeEntityType` entfernt alle mit dem *EntitätsTyp* (`[entityType(EntityType)]`) annotierten Informationen innerhalb eines Agenten, `+!removeKowledgeAboutAgent` das gesamte veröffentlichte Wissen eines Agenten `AgentName`.

4.1.5 Erweiterung des Kontextes

Der folgende Abschnitt beschreibt zwei realisierte *EntitätsTypen* zur Erweiterung der Wissensmenge. Die konzeptuelle Beschreibung dieser beiden Typen ist unter 3.6.7 zusammengefasst.

4.1.5.1 Abstraktion von Sensorinformationen

Zur Abstraktion von Positionsinformationen existiert im Kontrollsystems der Typ *zoneMapper*. Der *zoneMapper* verarbeitet *Worldstates* vom Typ

```
worldstate(EntityID, position, [X, Y, Z])
```

zur folgenden Menge von *Beliefs*

```
worldstate(EntityID, atZone, Zone)
worldstate(Zone, peopleAtZone, Number)
worldstate(room, hasMovement, true)
```

Die Definition bekannter *Zonen* findet über die Verwendung globaler Zonenbeliefs statt.

```
worldstate(Zone, zoneBorders, [Xmin, Xmax, Ymin, Ymax, Zmin, Zmax])
```

Die Erfassung von Zonenzugehörigkeiten geschieht unabhängig für jede *Zone*. Auf diese Weise können sich *Zonen* überlagern und *Entitäten* gleichzeitig mehrere *Zonen* belegen. *Worldstate* `peopleAtZone` beschreibt für jede *Zone* die Anzahl der *Entitäten* mit *EntitätsTyp „person"*. Dieses *Belief* wird unter anderem für die Auswertung von Personenverteilungen verwendet. Bewegungen im Raum erzeugen ein `hasMovement`-*Worldstate*

wird für einen längeren Zeitraum keine Bewegung wahrgenommen, wird dieses *Worldstate* entfernt. Die Definition von *Zonen* durch globale *Worldstate-Beliefs* ermöglicht es jedem Agenten eigene abstrakte *Zonen* festzulegen und auswerten zu lassen. Abstrakte *Zonen* ermöglichen unter anderem die umgebungsunabhängige Spezifikation von Geräteabhängigkeiten. Leinwände können beispielsweise Lampen in der Nähe dimmen oder Projektoren Leinwände in ihrer Projektionsrichtung steuern (Beispiele hierfür sind in 4.3 beschrieben).

4.1.5.2 Situationswahrnehmung

Zur Wahrnehmung von Situationen verwendet das Kontrollsystem sogenannte *situation-Perceptors* (beschrieben in 3.6.7). Für jede Situation existiert ein individueller *Entitäts-Typ*. Ihre Aufgabe ist es das Eintreffen und Beenden von Situationen zu erkennen und entsprechend Rollendefinitionen zu veröffentlichen oder zu entfernen. Für die Wahrnehmung von Situationen kann das Wissen über Personenverteilungen (4.1.5.1), Zustandsänderungen in *Worldstates* (4.1.2.2) oder Zielaufrufe anderer Agenten (4.1.4.3) verwendet werden. Ein Beispiel für die Realisierung einer Situationserkennung wird in 4.2.2.3 vorgestellt.

4.2 Richtlinie zur Erstellung neuer Agenten

Der folgende Abschnitt gibt einen Überblick über notwendige Schritte zur Erweiterung und Konfiguration des Kontrollsystems. Hierbei werden verfügbare Funktionaltäten und Vorgehensweisen für das Hinzufügen neuer Geräte, *Zonen* und Personenrollen anhand von Beispielen beschrieben.

4.2.1 Hinzufügen eines Gerätes bekannten Typs

Durch die Veröffentlichung eines Geräts im *Tuplespace*, wird automatisch eine Agent für dieses Gerät erstellt und *EntitätsTypen* entsprechend der Serviceklassen geladen. Die Zustände des Gerätes werden in die globale *Worldstate*-Menge von *BDI-SmartControl* aufgenommen. Sind die Serviceklassen des Gerätes bekannt und entsprechende *EntitätsTyp-RegelMengen* definiert, ist die *Entität* einsatzbereit.

4.2.2 Modellierung neuer Verhaltensweisen

Der folgende Abschnitt beschreibt das Vorgehen bei der Modellierung eigener Verhaltensweisen am Beispiel einer Erweiterung der `lecture`-*Rolle*. Das folgende Verhalten soll mithilfe des Kontrollsystems realisiert werden:

Der Vortragende einer Vorlesung soll für eine Aufzeichnung gefilmt werden. Alle automatischen Kameras sollen hierbei dem Vortragenden folgen. Ist kein Vortragender im Raum soll die Kamera auf die Vortragsleinwand zeigen.

Voraussetzungen für die Realisierung des Beispiels Die Positionsdaten aller Personen im Raum werden durch das Ubisensesystem erfasst und vom Ubisense-*Adapter* (4.1.2.3) in das Kontrollsystem übertragen. Für die Aufzeichnung werden bewegliche Kameras (`Cam1` und `Cam2`) in der Decke des *Smart-Appliance-Labs* verwendet. Die Vortragsleinwand ist `LW4`. Weiterhin werden die folgenden im *Tuplespace* veröffentlichten[3] Geräte-Services angenommen.

`Cam1 und Cam2` Beide Kameras besitzen den Service `devices.Movecam` der die Funktion `showPos(float x, float y, float z)` zur Ausrichtung der Kamera auf einen Punkt im dreidimensionalen Raum implementiert.

`LW4` Ist die Vortragsleinwand mit Service `devices.Lw` und der Eigenschaft `float[] position` = { 1, 2, 3 }.

[3] Für die Veröffentlichung von Objekten im *Tuplespace* sollte der in 2.3 beschriebene *GenericPublisher* verwendet werden.

4.2.2.1 Veröffentlichte Tupel und entstandene Worldstates

Die Veröffentlichung der Geräte im *Tuplespace* erzeugt die folgenden relevanten Tupel:

```
( :status LW4 active )
( :status Cam1 active )
( :status Cam2 active )
( :device ( $type devices.Lw) LW4 )
( :device ( $type devices.Movecam) Cam1 )
( :device ( $type devices.Movecam) Cam2 )
( :property-map-elem LW4 position ( 1f, 2f, 3f ))
( :command-queue LW4 1 0)
( :command-queue Cam1 1 0)
( :command-queue Cam2 1 0)
```

Durch sie werden in *BDI-SmartControl* automatisch folgende *Worldstates* veröffentlicht:

```
worldstate( room, hasEntity, lw4 )
worldstate( room, hasEntity, cam1 )
worldstate( room, hasEntity, cam2 )
worldstate( lw4, entityTypeDefinition, devices.lw)
worldstate( cam1, entityTypeDefinition, devices.movecam)
worldstate( cam2, entityTypeDefinition, devices.movecam)
worldstate( lw4, position, [1, 2, 3])
```

Durch das Hinzufügen der *Worldstates* werden vom *orgManager* (4.1.3.3) automatisch Agenten für `lw4` mit Typ `devices.lw` und `cam1` und `cam2` mit Typ `devices.movecam` erzeugt. Zusätzlich werden entsprechende *RegelMengen* geladen und *CommandQueue*-Referenzen angelegt.

4.2.2.2 Modellierung eines Geräteverhaltens

Für die Spezifikation von *RegelMengen* können im Ordner „`./planSets/`" des *Controllers* die folgenden ***RuleSetID***`.asl`-Dateien hinterlegt werden. Sie werden bei der Aufnahme des entsprechenden *EntitätsTyps* (mit: ***EntityType*** = ***RuleSetID***) automatisch vom Agenten geladen.

```
devices.lw.asl
devices.movecam.asl
```

Der *EntitätsTyp* `devices.lw` und `devices.movecam` besitzen für dieses Beispiel lediglich die Aufgabe der Ausführung von Anweisungen in der Umgebung. Hierfür wird jeweils folgende Regel definiert:

```
+!achieve(EntityID, Key, Value, SenderEntityType)
   :  myEntityID(EntityID)
   <- achieve(EntityID, Key, Value).
```

Die Übergabe von Zielen einer *Entität* an die *RegelMengen* ihrer *EntitätsTypen* geschieht durch `+!achieve(EntityID, Key, Value, SenderEntityType)`. Die Bedingung, des *AgentSpeak*-Planes überprüft, ob die *Entität* des Agenten `myEntityID(EntityID)` mit der adressierten *Entität* `EntityID` des Ziels übereinstimmt. Die Aktion `achieve(EntityID, Key, Value)` (ohne „`!`" und `SenderEntityType`) leitet die entsprechende Anweisung zur Ausführung an die Umgebungsschnittstelle weiter.

Beispiel Ein Agent hat folgende Kosten spezifiziert und verfolgt folgendes Ziel:

```
worldstate(achieve(cam1,_,_,_), cost, 1).
worldstate(achieve(cam2,_,_,_), cost, 2).
... <- !maintainWorldstate( devices.movecam, showPos, [X,Y,Z] )
```

Die Kosten dieses Agenten beschreiben, das `cam1` günstiger ist als `cam2` (unabhängig von Aktion, Parameter und Absender). Die Ausführung dieses indirekt verteilbaren Ziels würde die günstigere *Entität* `cam1` veranlassen ein *Command*-Tupel `(:command-queue-elem Cam1 QueueID (showPos (X Y Z)))` in den *CommandQueue* von `Cam1` zu schreiben und sich selbst mit der höchsten *Priorität* des aufrufenden Agenten zu sperren (fehlende Absender-Typen bei der Formulierung von Zielen werden automatisch durch den höchsten Absender-Typ ersetzt).

4.2.2.3 Modellierung einer Situationserkennung

Das folgende Beispiel zeigt die Erkennung einer Vortragssituation und die Verteilung der *Rolle* des Vortragenden (`lecturer`). Als Verhaltensweise wird der *EntitätsTyp* `lecturePerceptor` mit folgender *RegelMenge* `lecturePerceptor.asl` versehen:

```
worldstate(rostum, zoneBorders, [4.8, 7.1, 0, 6.8, 0, 2.9]).
worldstate(seats, zoneBorders, [0, 4.6, 2.6, 6.8, 0, 2.8]).

+worldstate(Zone, peopleAtZone, Num)
   :  worldstate(rostum, peopleAtZone, Roster)
      & worldstate(seats, peopleAtZone, Sitter)
      & Roster = 1
      & Sitter > 3
      & not situationLecture
   <- +situationLecture.

-worldstate(rostum, peopleAtZone, Num)
   :  not worldstate(rostum, peopleAtZone, Roster)
      & not worldstate(vgasensor, vgaConnect, 3)
      & not worldstate(vgasensor, vgaConnect, 6)
      & situationLecture
   <- -situationLecture.
```

```
+situationLecture
   <- tellPercept(
       worldstate(person, roleDefinition, role(lecturer, [rostum], 1, []))
      ).

-situationLecture
   <- untellPercept(
       worldstate(person, roleDefinition, role(lecturer, [rostum], 1, []))
      ).
```

Die Wahrnehmung von Situationen basiert in diesem einfachen Beispiel auf der Verteilung von Personen in abstrakten *Zonen* und Verbundener VGA-Ports. Das Eintreten der Situation benötigt mindestens vier Personen im Sitzbereich (`seats`) und genau eine Person im Vortragsbereich (`rostum`). Das Verlassen der Situation erfolgt, wenn kein VGA-Port im Vortragsbereich angeschlossen ist und sich dort auch keine Person aufhält (VGA-Anschlüsse sind beweglich und besitzen daher keine Positionsinformationen, die für eine Ortsabhängigkeit genutzt werden können). Der `lecturePerceptor` unterteilt den realen Raum in zwei zusätzliche Bereiche. Zu diesem Zweck veröffentlicht er vor seiner Ausführung die individuellen *Zonen* `rostum` und `seats` als statische *Worldstates.* Das Eintreten und Verlassen einer Situation hat die dynamische Verteilung von Rollentyp `lecturer` auf eine *Entität* des Typs `person` im Vortragsbereich zur Folge. Das Starten dieses *EntitätsTyps* in einer neuen *Entität* kann beispielsweise durch folgende Tupel geschehen.

```
(:bdi-worldstate „worldstate(room, hasEntity, AgentName)”)
(:bdi-worldstate
   „worldstate(AgentName, entityTypeDefinition, lecturePerceptor)”
)
```

4.2.2.4 Modellierung eines Rollenverhaltens

Für die Realisierung des Kameraverhaltens wird eine zusätzliche Datei im Ordner „`./planSets/`” erstellt. Diese wird `videoTarget.asl` um das Verhalten eines *EntitätsTyps* `videoTarget` zu spezifizieren. Die *RegelMenge* dieser Datei enthält folgenden Inhalt.

```
worldstate(videoTarget, priority, 10).

+!init(videoTarget)
   :  myEntityID(EntityID)
      & worldstate(EntityID, position, [X,Y,Z])
   <- !achieveWorldstateAll(devices.movecam, showPos, [X,Y,Z]).

+!exit(videoTarget)
   :  worldstate(lw4, position, [X,Y,Z])
   <- !achieveWorldstateAll(devices.movecam, showPos, [X,Y,Z]).
```

```
+worldstate(EntityID, position, [X,Y,Z])
   :  myEntityID(EntityID)
   <- !achieveWorldstateAll(devices.movecam, showPos, [X,Y,Z]).
```

Wird diese *RegelMenge* von der *RuleLibrary* eingelesen, wird der statisch definierte *Worldstate* `worldstate(videoTarget, priority, 10)` der globalen Wissensbasis hinzugefügt. Das Laden und Entladen der *RegelMenge* innerhalb eines Agenten erzeugt *Events* der Form `+!init(videoTarget)` und `+!exit(videoTarget)` (beschrieben in 4.1.4.2). Das *Event* `worldstate(EntityID, position, [X,Y,Z])` wird durch das Hinzufügen neuer Positionsinformationen (unter anderem durch den Ubisense-*Adapter*) erzeugt. Die eigene *EntitätsID* kann über das interne *Belief* `myEntityID` ermittelt werden (beschrieben in 4.1.4). Mit seiner Hilfe und Positions-*Beliefs* über die eigene *Entität* wird unter Verwendung von `!achieveWorldstateAll` allen *Entitäten* mit *EntitätsTyp* `devices.movecam` (`cam1` und `cam2`) das Ziel gegeben auf Position `[X, Y, Z]` zu schauen. Beim Entladen der *Rolle* werden alle Kameras auf die Position von `lw4` ausgerichtet.

Erweiterung der Vortragsrolle Für die Erweiterung der *Rolle* des Vortragenden (`lecturer`) wird folgender *Worldstate* in den *Tuplespace* eingefügt:

```
(:bdi-worldstate
   „worldstate(lecturer,roleDefinition,role(videoTarget,[room],1,[]))"
)
```

Existiert eine aktiver `lecturer`-*EntitätsTyp* im System, veranlasst diese Rollendefinition die dynamische Verteilung des *EntitätsTyps* `videoTarget` auf Maximal `1` *Entität* vom Typ `lecturer` in *Zone* `room`. Durch die höhere *Priorität* von `10` wird das Verhalten von `videoTarget` nicht überschrieben.

Rollenabhängigkeiten *Rollen* niederer *Prioritäten* benötigen kein Wissen über *Rollen* höherer *Priorität*. *Rollen* höherer *Priorität* können das Rollenverhalten niederer *Rollen* überschreiben (3.6.10.2), oder sie als inkompatible *Rollen* ganz verdrängen (4.1.3.3). Die Rollenhierarchie der `videoTarget`-*Entität* würde im Kontrollsystem aus folgenden, in Abschnitt 4.3.3 beschriebenen, *EntitätsTypen* bestehen:

```
person < member < lecturer < videoTarget
```

Die *Rolle* `person` erzeugt ein Aufleuchten der Lampen im Bereich des Nutzers. Die *Rolle* `member` überschreibt unter anderem dieses Verhalten (4.3.3). Alle für den Vortrag relevanten Nutzerunterstützungen werden von `lecturer` abgedeckt. Soll nun der Vortragende für eine Aufnahme optimal beleuchtet werden, muss lediglich die das `person`-Verhalten unterdrückende `member`-*Rolle* entfernt werden. Hierzu kann der *Worldstate* zur Aktivierung der `videoTarget`-*Rolle* folgendermaßen angepasst werden:

```
worldstate(lecturer,roleDefinition,role(videoTarget,[room],1,[member]))
```

4.3 Realisierte Verhaltensweisen

Dieser Abschnitt beschreibt realisierte Verhaltensweisen für die Arbeit im *Smart-Appliance-Lab*. Auch der *zoneMapper* (vorgestellt in 4.1.5.1) ist als *EntitätsTyp* implementiert. Abbildung 4.1 zeigt ein Beispiel für Geräteabhängigkeiten durch dynamisch erstellte *Zonen* und Raumbereiche für die Erfassung von Personenverteilungen.

4.3.1 EntitätsTypen für Geräte

Geräte-*Entitäten* besitzen *Priorität* 0. Für eine klassenunabhängige Adressierung durch andere *Entitäten* nehmen Geräte-Agenten zusätzliche, innerhalb der klassenabhängigen *EntitätsTypen* spezifizierte, abstrakte Typen an. Zusätzliche *EntitätsTypIDs* sind in Klammern vermerkt. Im *CommandQueue* interpretierbare Steuerungsanweisungen und Properties aus Service-Klassen sind automatisch im Kontrollsystem verwendbar. Die folgenden Paragraphen beschreiben lediglich die Erweiterungen von Fähigkeiten und Verhaltensweisen durch *EntitätsTyp-RegelMengen*.

de.mmis.devices.eib.devices.lamp (lamp)

- ▷ Veröffentlicht eine abstrakte Beleuchtungszone um die eigene Position. Der Name dieser *Zone* entspricht der eigenen *EntitätsID*.

de.mmis.devices.eib.devices.screen (screen)

- ▷ Veröffentlicht eine abstrakte Leinwandzone um die eigene Position. Der Name dieser *Zone* entspricht der eigenen *EntitätsID*.
- ▷ Heruntergefahrene Leinwände dimmen das Licht aller Lampen in ihrer *Zone*.

de.mmis.devices.eib.devices.windowsensor (windowsensor)

- ▷ Nimmt zusätzlich den *EntitätsTyp* `windowsensor` mit *Priorität* 100 an.
- ▷ Fahren gefährdete Verdunkelungen hoch und sperren sie.

de.mmis.devices.extron.extron (extron)

- ▷ Kann bekannte *EntitätIDs* auf *Input*-Nummern abbilden.

de.mmis.devices.pensensor.pensensor (pensensor)

- ▷ Erzeugt `worldstate(pensensor, penUsed, I)` für aufgenommene Stifte aus einer Stiftbox `I`.

de.mmis.devices.vgasensor.vgasensor (vgasensor)

- ▷ Erzeugt `worldstate(vgasensor, vgaConnect, I)` für einen angeschlossen VGA-Port `I`.

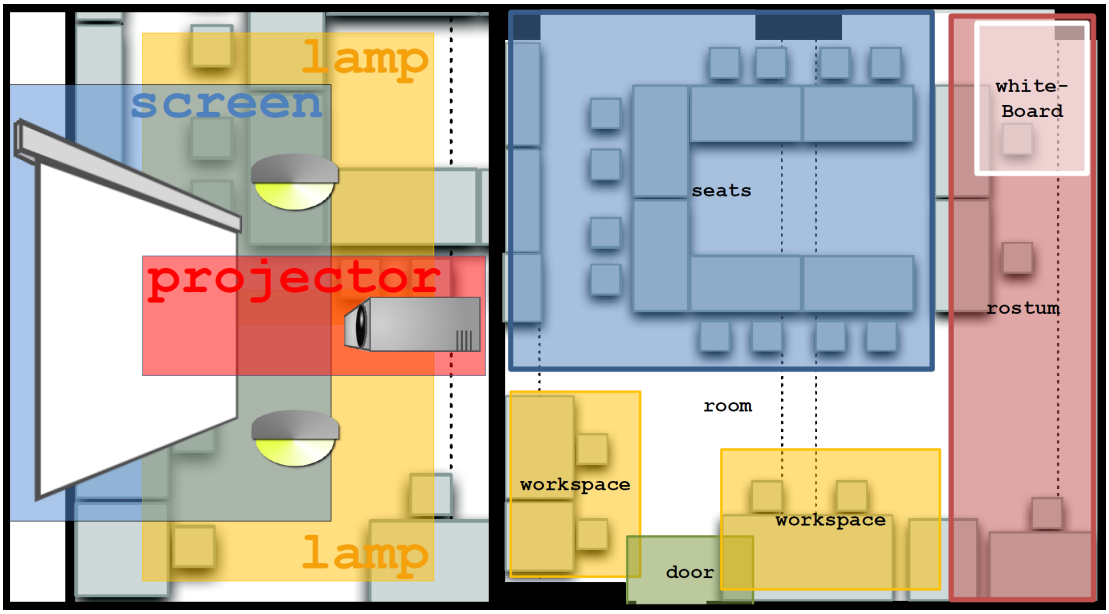

Abbildung 4.1: Abstrakte *Zonen* für Geräteabhängigkeiten und Personenverteilungen.

de.mmis.devices.projector.projector (projector)

- ▷ Veröffentlicht eine schmale Projektionszone anhand seiner Position und Ausrichtung. Der Name dieser *Zone* entspricht seiner *EntitätsID*.
- ▷ Steuert automatisch Leinwände und Lampen in seiner Projektionszone.
- ▷ Steuert die Extron im Raum.
- ▷ Kann mit dem Ziel `display` und einer VGA-Port Quelle `Source` alle Konfigurationen für eine Darstellung durchführen.
- ▷ Bestehende Inhalte können gemäß den Kosten des Nutzers auf andere Projektoren umgeleitet werden.
- ▷ Veröffentlicht die aktuelle Darstellungskonfiguration als *Worldstate*.

4.3.2 EntitätsTypen zur Situationserkennung

Die Folgenden *EntitätsTypen* implementieren Verhaltensweisen zur Situationserkennung. Ihre Aufgabe ist es, den Beginn und das Ende einer Situation zu ermitteln und entsprechende Anpassungen des *Controller*-Verhaltens durchzuführen. Im Gegensatz zu Geräte- oder Rollentypen werden diesen Verhaltensweisen keinen realen Objekten zugeordnet.

lecturePerceptor

- ▷ Ermittelt den Beginn einer Situation durch die Verteilung mehrerer Personen im Sitzbereich und einer Person im Vortragsbereich.
- ▷ Ermittelt das Ende einer Situation durch das Verlassen des Vortragsbereichs bei unverbundenen VGA-Ports.
- ▷ Ist eine Vortagssituation aktiv, wird die *Rolle* `lecturer` an eine `person` im Vortragsbereich vergeben.

memberPerceptor

- ▷ Ermittelt den Beginn einer Situation durch Verteilt sitzende Personen im Sitzbereich.
- ▷ Ermittelt das Ende einer Situation wenn die Anzahl der Personen im Sitzbereich unter 3 sinkt.
- ▷ Veröffentlicht zwei abstrakte *Zonen* zur Unterteilung des Sitzbereiches.
- ▷ Ist eine Gruppensituation aktiv, bekommen alle `person` die *Rolle* `member` und der Raum wird beleuchtet.

resetPerceptor

- ▷ Wird für längere Zeit keine Bewegung im Raum festgestellt, werden alle Geräte bekannten *EntitätsTyps* abgeschaltet.

4.3.3 EntitätsTypen als Personenrollen

Personenrollen sind durch dynamisch verteilte *EntitätsTypen* realisiert. Mit ihnen ist es möglich das unterstützende Verhalten des Kontrollsystems direkt für priorisierte Personen in bestimmten Situationen und *Zone* zu modellieren.

person (Priorität 0)

- ▷ Standardtyp eines Ubisense-Tags und allgemeiner Träger weiterer Personenrollen.
- ▷ Wird ein Knopf des Ubisense-Tags gedrückt wird die höchste *Rolle* des Tags ausgegeben.
- ▷ Betritt eine Person die Beleuchtungszone einer Lampe, wird diese aktiviert.
- ▷ Verlassen alle Personen die Beleuchtungszone einer Lampe, wird diese gedimmt.
- ▷ Verbindet eine Person einen VGA-Port, wird die ihm gegenüber liegende Leinwand bevorzugt.

member (Priorität 2)

- ▷ Deaktiviert das dynamische Beleuchtungsverhalten der *Rolle* `person`.
- ▷ Verbindet ein member einen VGA-Port, wird eine allgemein gut sichtbare Leinwand bevorzugt.

lecturer (Priorität 5)

- ▷ Verteilt die *Rolle* listener auf alle übrigen Personen.
- ▷ Verbindet ein lecturer einen VGA-Port im Vortragsbereich, wird eine definierte Vortragskonfiguration erstellt und fixiert.

listener (Priorität 4)

- ▷ Konzept zur Unterdrückung unerwünschten Teamverhaltens.
- ▷ Unterdrückt das Verhalten zu Auslöseereignissen, die von einem Vortragenden bearbeitet werden, für alle Personenrollen bis Priorität Vier.

whiteBoardUser (Priorität 10)

- ▷ Ist eine Rollenerweiterung für eine Person in der *Whiteboard-Zone*.
- ▷ Wird ein Stift aufgenommen, werden verdeckende Leinwände hochgefahren.

4.4 Zusammenfassung

In diesem Kapitel wurde die technische Realisierung eines BDI-Kontrollsystems für *Intelligente Umgebungen* beschrieben. Die Konzeption und Realisierung des in dieser Arbeit entwickelten Systems wird zu Beginn des folgenden Kapitels zusammengefasst. Im Anschluss an die Erläuterung der technischen Umsetzung wurden anhand eines Beispiels die notwendigen Schritte zur Konfiguration und Erweiterung des Systems vorgestellt. Hierzu wurde eine einfache Realisierung eines Geräteverhaltens beschrieben und ein direkter Anwendungsfall einer kostenbasierten Zielverteilung dargestellt. Anschließend wurde eine mögliche Verhaltensweise zur Erkennung einer Vortragssituation und Verteilung der Personenrolle eines Vortragenden erläutert. Als Erweiterung der Personenrolle des Vortragenden diente das Beispiel der *Rolle* eines Kameraziels. Zusätzlich wurden Richtlinien bei der Entwicklung priorisierter Rollenhierarchien vorgestellt. Der abschließende Abschnitt dieses Kapitels beschrieb bereits realisierte Verhaltensweisen im Bereich des Geräteverhaltens, der Situationserkennung und der Unterstützung von Personen für das *Smart-Appliance-Lab* der Universität Rostock.

5 Zusammenfassung und Ausblick

Das abschließende Kapitel dieser Arbeit dient der Zusammenfassung des Konzepts und der Umsetzung. Abschnitt 5.2 gibt einen Überblick von Punkten, die für die Verwendung des BDI-Paradigmas für eine Realisierung eines Kontrollsystems für *Intelligente Umgebungen* ermittelt wurden. Darüber hinaus werden in 5.3 offene Probleme und weitere Entwicklungsziele dieses konkreten Ansatzes beschrieben. Abschließend wird das *BDI-SmartControl*-System einigen verwandten Arbeiten gegenübergestellt.

5.1 Zusammenfassung der Konzeption und Umsetzung

Zu Beginn von Kapitel 3 wurden auf Basis vorgestellter Nutzungsszenarien allgemeine Annahmen zur Realisierung des erwarteten Raumverhaltens entwickelt (3.2). Mithilfe dieser Annahmen wurden anschließend konkrete Anforderungen für die Entwicklung eines BDI-Kontrollsystems entworfen (3.3.2). Unter Verwendung dieser Anforderungen wurden verfügbare BDI-Frameworks (3.5) und die Middleware des *Smart-Appliance-Labs* (3.4) evaluiert. Die anschließende Konzeption (3.6) und Implementation (4) eines BDI-Kontrollsystems realisierte ein System mit folgendem Aufbau:

Das Kontrollsystem besteht aus einer dynamischen Menge *AgentSpeak*-basierter Agenten. Als Schnittstelle zur Umgebung verfügt es über eine Menge javabasierter *Adaptoren* (4.1.2). Diese realisieren unter anderem Verbindungen zum Tuplespace, zur Aufnahme von Informationen und Realisierung von Aktionen und dem *Ubisense*-System zur Erfassung von Positionen.

Wissen werden in Form globaler *Worldstates* (4.1.1) verwaltet. Geräte und Meta-Prozesse des Systems werden durch *Entitäten* (4.1.3.1) repräsentiert. Jede *Entität* besitzt eine eindeutige Identität und wird durch einen eigenen Agenten (4.1.4) verwaltet. Agenten besitzen initial nur latentes Verhalten. Verhaltensweisen und Fähigkeiten einer *Entität* werden durch ihren Typ bestimmt. Zu diesem Zweck wird einer *Entität* eine Menge von *EntitätsTypen* (4.1.3.2) zugeordnet.

EntitätsTypen dienen der Kapselung von Verhaltensweisen und Berechtigungen durch *RegelMengen* und *Prioritäten*. Verwendet werden sie zur Spezifikation von Gerätenfunktionalitäten und zur dynamischen Verteilung von Personenrollen. Personenrollen ermöglichen eine direkte, situationsbedingte und personenbezogene Modellierung unterstützender Verhaltensweisen. Aufgenommene *EntitätsTypen* dienen darüber hinaus der typbasierten Gruppierung von *Entitäten*.

Organisiert wird die Verteilung von *EntitätsTypen* und die Verwaltung von *Entitäts-* und Agenten-Mengen durch einen zentralen Organisationsagenten (4.1.3.3). Die Koordination von Verhaltensweisen und Auflösung von Ressourcenkonflikten wird innerhalb der statischen Funktionsmenge eines jeden Agenten realisiert (4.1.4).

Ziele werden in Achieve- und Maintainziele unterteilt. Während Achieveziele das einmalige Erreichen eines Zustandes ermöglichen, sperren Maintainziele eine *Entität* um einen erreichten Zustand zu erhalten. Jede *Entität* besitzt die Fähigkeit zur Verteilung dieser Ziele (4.1.4.3). Die Zielverteilung kann durch Angabe einer *Entität* (direkt) oder eines *EntitätsTyps* (indirekt) erfolgen. Bei der typbasierten Zielverteilung wird das Ziel automatisch an eine optimale *Entität*, der durch den *EntitätsTyp* spezifizierten Funktionsgruppe, vergeben. Die Auswahl einer optimalen *Entität* geschieht unter Beachtung von Kosten und Sperrinformationen entsprechend des Absendertyps.

Kosten dienen dem Means-End Reasoning bei der Verteilung von Unterzielen. Die Verwaltung von Sperrungen und Zugriffsberechtigungen wird durch *Prioritäten* koordiniert. Verhaltensweisen mehrerer *EntitätsTypen* innerhalb einer *Entität* wird durch eine priorisierte Sortierung organisiert (4.1.4.2). Zusätzlich wurden folgende Verhaltensweisen zur Erweiterung des Kontextwissens implementiert:

Situationsperceptoren erkennen Situationen anhand von Ein- und Austrittsereignissen. Entsprechend ihrer Situationen verteilen sie Rolleninformationen im System. Positionen werden auf abstrakte *Zonen* abgebildet. Mit ihrer Hilfe können, unabhängig von den Parametern der realen Umgebung, Personenverteilungen und Geräteabhängigkeiten spezifiziert werden.

5.2 Die Verwendung des BDI-Paradigmas

Zur Verwendbarkeit des BDI-Paradigmas sind die Meinungen unterschiedlich. Während klassische Entscheidungstheoretiker und Planungswissenschaftler die Notwendigkeit der einzelnen BDI-Komponenten hinterfragen, prüfen Soziologen und Bereiche der verteilten Intelligenz, ob diese ausreichend sind [RG95, S. 312].

Einige Vorteile des BDI-Systems sind in [RG95, S.318] beschrieben: BDI-Systeme ermöglichen das Entwerfen von hierarchischen Plänen zur Reaktion auf spezifische Situationen. *Intentions* schaffen eine gute Balance zwischen reaktivem und zielorientiertem Verhalten. Die Entwicklung eines BDI-Systems geschieht unter direkter Verwendung abstrakter Ziele.

Ein Beispiel ist die Ersetzung von *Fortan* als Beschreibungssprache eines Pilotenverhaltens durch abstrakte Pläne. Der Zeitaufwand zur Änderung eines taktischen Verhaltens wurde hierbei von zwei Monaten auf unter einen Tag reduziert [RMSM92].

Abschnitt 2.2.2 des Grundlagenteils dieser Arbeit beschreibt eine Motivation des BDI-Paradigmas für die Arbeit in dynamischen Systemen. Für die Realisierung eines Kontrollsystems ist die Speicherung und Aktualisierung von Informationen in Form von *Beliefs* von Vorteil. Zusätzlich bieten *Beliefs* eine gute Grundlage zur Erweiterung des Wissens und der Speicherung interner Zustände. Die Formulierung von Zielen ermöglicht eine direkte, dynamische und lokale Modellierung der Nutzerunterstützung. Die Selektion attribuierter Aktionspfade ermöglicht die Implementation komplexer Reasoning-Mechanismen. Ein Hauptvorteil des Paradigmas ist die intentionsgesteuerte Festlegung (Committment) auf die Ausführung von Prozessen.

Ein Einsatzbeispiel ist das in [RG95, S. 313] beschriebene OASIS-System zur Über-

wachung eines Flugplatzes. Die Verwendung des BDI-Paradigmas begünstigt hier unter anderem das Abbrechen langwieriger Aktionen wie beispielsweise Landeanflüge durch unvorhergesehene Umgebungsänderungen.

Beim direkten Einsatz des Kontrollsystems im *Smart-Appliance-Lab* ist diese Fähigkeit nach subjektiver Einschätzung weniger relevant. Nach den in 3.3.2 beschriebenen Anforderungen, werden Nutzerwünsche spontan durch Ereignisse gebildet und können im Allgemeinen in sehr kurzer Zeit durch die Geräte der Umgebung realisiert werden.

Das BDI-Paradigma folgt dem symbolorientieren Ansatz der Wissensverarbeitung. Im Vergleich zu konnektivistischen Ansätzen wie beispielsweise *Neuronalen Netzen* liegen die Stärken eines Systems in der direkten, abstrakten Modellierung von Verhaltensweisen. Neuronale Netze hingegen bieten einen klaren Vorteil bei der Realisierung lernender und sich anpassender Systeme.

5.3 Fähigkeiten, Probleme und weitere Entwicklungsziele

Im folgenden Abschnitt werden Fähigkeiten und Grenzen des Systems aufgezeigt. Zusätzlich werden offene Entwicklungsziele formuliert.

5.3.1 Fähigkeiten

Der folgende Abschnitt fasst die Fähigkeiten des entwickelten *BDI-SmartControl*-Systems, verwendeter Frameworks und *Middlewares* zusammen.

5.3.1.1 Das Jason-Framework

Das *Jason*-Framework vereint die logische, agentenorientierte Programmiersprache *AgentSpeak* mit der objektorientierten Programmiersprache *Java*. *AgentSpeak* bietet als agentenorientierte Entwicklungssprache eine übersichtliche und mächtige Grundlage für die Entwicklung komplexer Verhaltensweisen. Die integrierte *Java*-Schnittstelle ermöglicht eine direkte Implementation von *Adaptoren* für die javabasierte Infrastruktur des *Smart-Appliance-Lab* und eine Integration beliebigen *Java*-Codes zur Erweiterung des Funktionsumfangs. Die parallele Arbeit dynamischer Agentenmengen ermöglicht eine asynchrone und unabhängige Entwicklung und Arbeit einzelner Intelligenzen. Die Verwendung einer *SACI*-basierten Kommunikation unterstützt die Interaktion mehrerer Agenten. Durch die Kompatibilität zum Agenten-Framework *Jade* werden Verbindungen zu anderen Agentensystemen und die Verwendung von *FIPA* Kommunikationsstandards ermöglicht. Das *Jason*-System besteht aus einem *JAR*-Archiv und kann auf nahezu jedem Computer ohne Installation direkt ausgeführt werden. Die Entwicklung von *AgentSpeak* und *Java*-Code wird durch ein *Eclipse-Plugin* oder *Jason-JDE* unterstützt [BH11].

5.3.1.2 Tuplespace, Infrastructure und NetworkProxy

Die Verwendung von *NetworkProxies* ermöglicht eine schnelle und einfache Verbindung zu allen veröffentlichten *Services* im *Smart-Appliance-Lab* (inklusive *Tuplespace*). Der

Tuplespace bietet die Möglichkeit der direkten Abfrage von Identifikationen aktiver Geräte, Gerätetypen, Zuständen und der Mitteilung von Zustandsänderungen. Durch die Implementation des *CommandQueues* existiert eine direkte textbasierte Schnittstelle für die Steuerung von Geräten durch *AgentSpeak*-Agenten. Darüber hinaus bietet der *Tuplespace* eine erreichbare Schnittstelle für eine tupelbasierte Konfiguration des Kontrollsystems. Durch die Verwendung der *Infrastructure* und des *GenericPublishers* können alle relevanten *Services* direkt für *BDI-SmartControl* veröffentlicht und gefunden werden.

5.3.1.3 *BDI-SmartControl*

Der folgende Abschnitt beschreibt die Fähigkeiten durch die Realisierung der in 3.6 vorgestellten Konzepte in 4.

Dynamische Geräteverwaltung Das *BDI-SmartControl*-System nimmt zur Laufzeit beliebige im *Tuplespace* veröffentlichte Geräte als *Entität* auf. Diese können automatisch durch fähige[1] *Entitäten* unter Verwendung aller realisierten Koordinations- und Zielverteilungsmechanismen genutzt werden. Geräte bekannten Typs (verknüpfte *EntitätsTypen* mit *RegelMengen* und *Prioritäten*) übernehmen automatisch entsprechende Verhaltensweisen und Fähigkeiten. Wird beispielsweise eine Tischlampe mit dem entsprechenden Geräte-*Service* für *EntitätsTyp* `lamp` (4.3.1) veröffentlicht, erzeugt ihre *Entität* bei vorhandenen Positionsinformationen eine Beleuchtungszone und wird von einem sich nähernden *Ubisense-Tag* mit Typ `person (4.3.3)` angeschaltet. Durch die Verwendung einzelner Agenten arbeitet jedes Gerät hierbei unabhängig und parallel. Die Nutzung von Geräteagenten ermöglicht eine selbstständige Bewältigung individueller Ressourcenkonflikte unter Verwendung interner Zustände und verfügbarer Koordinationsmechanismen (beispielsweise zur Verteilung angezeigter Inhalte unter Projektoren).

Sammlung und Abstraktion von Informationen heterogener Quellen Die Verwendung javabasierter *Adaptoren* ermöglicht eine übersichtliche Erweiterung des Systems zur Informationsaufnahme und Steuerung heterogener Komponenten des *Smart-Appliance-Labs*. Die Verwendung einer einheitlichen Informationsdarstellung in Form von *Worldstate-Prädikaten* vereinfacht die Abbildung von Inhalten ins Kontrollsystem und erzeugen eine kompatible Darstellung. Die einheitliche Darstellung durch *Worldstates* versetzt interne *EntitätsTypen,* wie beispielsweise den *zoneMapper*, in die Lage Positionsinformationen beliebiger *Adaptoren* (*Ubisense*, *Sensefloor*, *Kameratracking*) abstrakten *Zonen* zuzuordnen.

Modellierung von Verhaltensweisen Die zusätzliche Erhöhung der Abstraktionsebene des Umgebungskontextes vereinfacht die Modellierung des Verhaltens. Das Erzeugen übergeordneter *Kontexte* wie beispielsweise „Vier Personen im Sitzbereich" erhöht gleichzeitig die Mächtigkeit bei der Definition von Bedingungen. Die Modellierung von Verhal-

[1] Fähig bedeutet in diesem Sinn Kenntnis über mögliche Steuerungsbefehle des veröffentlichten Geräte-Services

tensweisen erfolgt durch *AgentSpeak*-Pläne. Diese werden zu *RegelMengen* zusammengefasst. Eine *RegelMenge* repräsentiert eine abstrakte Verhaltens- und Funktionsmenge. Die Modellierung erfolgt lokal und direkt durch Angabe eines auslösenden Ereignisses und notwendiger Bedingungen für ein Handeln des Agenten. Mechanismen zur entitäts- oder typbasierten Ansteuerung von Geräten unter Beachtung von Sperrungen und Kosten sind in wenigen Zielaufrufen realisiert und nehmen Entwicklern einen erheblichen Teil des Koordinationsaufwands ab. Die Übertragung eines Steuerungsbefehls über den *CommandQueue* zum letztendlichen Geräteservice ist in einer einzelnen Plan-Aktion realisierbar. Die dynamische Verteilung prioritätsabhängiger Rollen ermöglicht eine direkte Entwicklung situationsbezogener Unterstützungen individueller Personentypen. Kombinationen von Personenrollen realisieren hierarchisch organisierte Verhaltenskompositionen. Durch das hohe Abstraktionslevel, die Verwendung der Zielverteilungsmechanismen und *AgentSpeak* als Programmiersprache umfassen entwickelte Personenrollen oft nur wenige Zeilen. Als Beispiel zeigt der folgende Code das Anschalten einer Lampe beim Betreten ihrer Beleuchtungszone (4.3.1), unabhängig von den Parametern einer realen Umgebung:

```
+worldstate(EntityID, atZone, LampZone)
   :  myEntityID(EntityID)
      & worldstate(LampZone, committedEntityType,lamp)
   <- !achieveWorldstate(LampZone, turnOn, [], person).
```

Unabhängigkeit zur Umgebung Die Arbeit auf abstrakten *Zonen* und einer typbasierten Zielverteilung geschieht weitestgehend unabhängig von der tatsächlichen Umgebung. Für einen Wechsel des derzeitigen Kontrollsystems in eine neue Umgebung müssen lediglich gerätebezogene Kosten, Zonengrenzen und direkt spezifizierte Auslöseereignisse (beispielsweise für `lecturer` die VGA-Portnummern des Vortragsbereichs) angepasst werden.

5.3.2 Offene Probleme und weitere Entwicklungsziele

Im folgenden Abschnitt werden die offenen Probleme, die im Rahmen dieser Arbeit nicht ausgearbeitet wurden erläutert, weiterführende Arbeitsziele formuliert und Lösungsansätze genannt.

Groß- und Kleinschreibung Das *AgentSpeak*-basierte Kontrollsystem nutzt die Unterscheidung von Groß- und Kleinschreibung für eine Trennung von variablen und terminalen Ausdrücken. Das aktuelle Kontrollsystem behebt dies, indem alle eintreffenden terminalen Ausdrücke der Umgebung in kleingeschriebene Ausdrücke umgewandelt werden. Die Veröffentlichung eines Gerätes mit gleicher Bezeichnung aber unterschiedlicher Schreibweise wird vom Kontrollsystem als eine *Entität* behandelt. Eine weiterführende Entwicklung kann dies durch eine separate Kennzeichnung unterschiedlicher Schreibweisen beheben.

Identisches Verhalten identischer Rollen Durch die Verteilung von *Rollen* auf Personen besitzen oft mehrere Personen-Agenten den gleichen *EntitätsTyp*. Da Auslöseereignisse nur unter größerem Aufwand einer einzelnen Person zugeordnet werden können, reagieren alle *Entitäten* mit gleicher *Rolle* simultan und identisch auf eintreffende Ereignisse. Das aktuelle System realisiert die Kompensation mehrerer identischer Ziele innerhalb der Zielentität. Zusätzlich werden bei der indirekten Verteilung von Zielen alle *EntitätsIDs der Entitäten* entsprechenden Typs nach Namen sortiert. Das Präfix der entstandenen Namensliste führt das Ziel aus, alle anderen werden unterdrückt. Da dieser Ansatz nicht auf das Ziel sondern auf die Verteilung von Typen bezogen ist, kann hierbei auch ein individuelles Ziel einer „Nichtpräfix"-*Entität* unterdrückt werden. Eine weitaus bessere Lösung bietet hier die Realisierung eines Auktionsverfahrens, bei dem sich mehrere Rollen mit identischem Ziel auf eine ausführende *Entität* einigen. Ein möglicher Auktionsgegenstand wäre hierbei die Zufallszahl.

Verwendung quaderförmiger abstrakter Zonen *Zonen* werden im Kontrollsystem in Form von Quadern definiert. Für die Spezifikation von Geräteabhängigkeiten ist diese Form auf Dauer nicht ausreichend. Langezogene Projektionszonen eines Beamers können mit diesem Verfahren nur entlang der X-, Y- oder Z-Achse angelegt werden. Das System verfügt über die Möglichkeit mehrere Quader unter gleicher Zonenbezeichnung zusammenzufassen. Für komplexe Strukturen ist diese Methode aber kaum praktikabel. Eine weiterführende Entwicklung könnte die Spezifikation abstrakter *Zonen* durch komplexere Formen ermöglichen.

Skalierbarkeit Die Verwendung einzelner *Worldstates* erzeugt eine große Anzahl von Zuständen. Diese ergeben sich aus der Zuweisung gesammter Servicehierarchien in Form von *EntitätsTypen* für jede *Entität*, der globalen Veröffentlichung von Agentenkonfigurationen, der dynamischen Verbreitung abstrakter *Zonen* und damit einhergehenden unidirektionalen Zuweisung von Geräten und Personen. Zusätzlich wird die gesammte Geräte-Zustandsmenge des *Tuplespaces* ins Kontrollsystem übertragen.

Agenten benötigen für die Deliberation unter Verwendung des *Kontextes* eine logische Wissensbasis. Die globale Wissensbasis (beschrieben in 4.1.2.1) ist durch eine zentrale Verteilung aller globalen *Worldstates* in die Wissensmenge jedes einzelnen Agenten realisiert. Für die aktuelle Konfiguration des *Smart-Appliance-Labs* erzeugt dies für circa 30 *Entitäten* ungefähr 700 *Worldstates* pro Agent. Ein Ziel für zukünftige Arbeiten wäre die Reduzierung der Zustandsmenge eines jeden Agenten auf die Menge der relevanten Zustände.

Lernen von Verhaltensweisen Für die Realisierung eines lernenden BDI-Systems existieren nur einige wenige Ansätze. Die in diesem Kontrollsystem modellierten Verhaltensweisen werden ausschließlich von Entwicklern entworfen. Emergentes Verhalten entsteht hierbei durch dynamische Geräte-Mengen, Rollen und der Komposition von Verhaltens-Mengen. S. Bader beschreibt in [Bad10, S. 8-9] das Lernen durch Anpassungen von Kostenfunktionen. In Kombination mit dem in dieser Arbeit entwickelten Kostenmodell

ist ein Erlernen bevorzugter Geräte zügig realisierbar. Das Erlernen neuer abstrakter Verhaltensweisen hätte nach erster Betrachtung eine automatische Erstellung von *AgentSpeak*-Plänen zur Folge. Die Entwicklung dieser Art des Lernens ist nur unter erhöhtem Aufwand möglich. Einige Mechanismen für das Lernen in BDI-Systemen sind in [LLJ08, S. 646] und [PWP05] beschrieben.

5.4 Vergleich verwandter Arbeiten

Nachfolgend werden einige verwandte Arbeiten im Bereich der Intelligenten Umgebungen vorgestellt und die verfolgten Ziele und realisierten Konzepte mit dem BDI-SmartControl-System verglichen. Die allgemeinen Vor- und Nachteile des in dieser Areit entwickelten Kontrollsystems wurden in Abschnitt 5.3 beschrieben.

Zentrale Planung auf Basis verteilter Operatoren Thomas Heider beschreibt in [HT02], als Teil des *EMBASSI*-Projekts [HKS01], ein System zur Realisierung einer multimodalen, zielbasierten Steuerung komplexer, technischer Infrastrukturen. Ziel ist es die alltägliche funktionale Interaktionen zwischen Menschen und einer Vielzahl externer Objekte durch eine globale Systemsicht und abstrakte Ziele zu ersetzten. Durch die multimodale Formulierung eines abstrakten Ziels soll hierbei die notwendige Konfiguration dynamischer Gerätemengen vom System realisiert werden. Einige Designziele der Architektur beschreiben die Erkennung neuer Gerätefunktionalitäten und eine dezentrale Verwaltung unabhängiger, dynamisch austauschbarer Komponenten. Das System arbeitet als Pipeline-Ansatz, bestehend aus mehreren Protokoll- und Aufgabenschichten. Die Formulierung abstrakter Ziele geschieht durch eine Transformation physikalischer, multimodaler Nutzeraktionen in atomare Events (lexikalische Schicht), einer Umwandlung atomarer Events in amodale Sätze (syntaktische Schicht) und einer Transformation amodaler Sätze zu abstrakten Zielen (semantische Schicht). Die Realisierung von Zielen wird durch eine planende Strategieschicht und eine koordinierende Ausführungsschicht ermöglicht.

Im System dieser Arbeit sind die abstrakten Ziele zur Unterstützung eines Nutzers in dynamisch verteilten Personenrollen modelliert. Das Kontrollsystem übernimmt automatischen im Hintergrund die Ausführung von Aktionen. Das Hauptaugenmerk liegt hierbei in der Koordination dynamischer, aktiver Zielmengen mehrerer typisierter Nutzer und Geräte einer Umgebung. Während die Realisierung des *EMBASSI*-Projekts die Komposition von Gerätefunktionen zu abstrakten Zielen und eine multimodale, dialogbasierte Interpretation von Nutzerwünschen ermöglicht, dient das *BDI-SmartControl* der autonomen Unterstützung individueller Nutzertypen und zielbasierten Steuerung dynamischer Gerätemengen. Das Kontrollsystem verfügt hierfür über keine direkte Interaktionsschnittstelle für Nutzer. Zur Konfiguration des Systems können spezielle Tupel im *Tuplespace* verwendet werden.

Dezentrale Planung durch Verbreitungen von Aktivierungen Christiane Plociennik beschreibt in [RT08] einen dezentralen Planungsansatz zur Selbstorganisation dynamischer Gerätemengen. Ziel des Ansatzes ist das Finden optimaler Aktionssequenzen zur komplexen Unterstützung von Nutzeraktivitäten bei gegebenen Zielen. Hierbei soll das System ohne direkt modellierte Kooperationsstrategien auskommen. Als Grundlage dient der in [Mae90] beschriebene *Maes*'-Algorithmus. In ihm werden mögliche Aktionen eines Gerätes durch sogenannte „*compmodinst*" (*CMI*) repräsentiert. *CMIs* bestehen aus Vorbedingung und Effekt. Je nach Übereinstimmung von Vorbedingung und Effekt sind sie in der Lage, untereinander positive oder negative Aktivierungsenergien auszutauschen. Zusätzlich zum Austausch, werden diese Energien auch durch Wahrnehmungen und Nutzerziele generiert. Die Verteilung dieser Energien ist in einzelnen Durchläufen synchronisiert. Am Ende eines Durchlaufs wird eine Aktion ausgeführt.

Das *BDI-SmartControl*-System realisiert die Verteilung von Zielen durch direkt modellierte Kooperationsstrategien. Im Gegensatz zum voran beschriebenen System werden die Aktionssequenzen für das Erreichen eines abstrakten Ziels nicht emergent erzeugt, sondern sind fest in EntitätsTypen modelliert. Lediglich die Auswahl von Geräten und Bestimmung lokaler Abhängigkeiten zur Realisierung abstrakter Ziele wird hierbei kontextabhängig koordiniert. Anders als das *Maes*'-basierte System arbeiten die einzelnen Agenten asynchron und reagieren direkt auf eintreffende Ereignisse. Durch die Verwendung von Kostendefinitionen und abstrakten Zonen können hierfür beliebige situationsspezifische, dynamische Verhaltensmengen des BDI-Systems lokal und direkt modelliert werden. Im Gegensatz hierzu erzeugt das System von Plociennik komplexe abstrakte Ziele adhoc aus einzelnen, atomaren, möglichen Aktionen eines aktuellen Geräteensemles.

Reaktive Systeme zur Steuerung Intelligenter Umgebungen Abschnitt 1.2 beschreibt das von Martin Dyrba in [Dyr11] vorgestellte reaktive System (*SuBaSLC*) zur Steuerung *Intelligenter Umgebungen*. Ereignisse der Umgebung werden durch eine Menge aktiver, unabhängiger Verhaltensweisen in eine Menge von Zielzuständen umgewandelt. Die Auswertung teilweise konkurrierenden Zielmengen ermöglicht hierbei die Erzeugung von „Zwischen"-Zielzuständen. Beispielsweise ermöglicht das gleichzeitige Ein- und Ausschalten einer Lampe durch konkurrierende Verhaltensweisen einen gedimmten Zustand.

Die Ziele des *BDI-SmartControl*-Systems erzeugen nur absolute Zustände. Zu diesem Zweck werden die den Nutzer unterstützenden Aktionssequenzen koordiniert auf entsprechende Geräte verteilt. Rein reaktives Verhalten kann durch das Weglassen der Vorbedingung in Plänen nachgebildet werden. Ein Vorteil des BDI-Systems liegt bei der lokalen, absoluten Modellierung abstrakter, komplexer, situationsabhängiger und dynamischer Verhaltensweisen unter Verwendung aktueller und vergangener Zustände. Ein Vorteil des reaktiven Systems gegenüber der *BDI-SmartControl*-Realisierung liegt in der Skalierbarkeit. Abschnitt 5.3.2 beschreibt dieses Problem im Detail.

Literaturverzeichnis

[And] J. R. Anderson. Act: A simple theory of complex cognition. *American Psychologist*, 1996(51):355–365.

[AOS08a] AOS Group. Jack intelligent agents teams manual: http://www.aosgrp.com/documentation/jack/jack_teams_manual_web/, 2008.

[AOS08b] AOS Group. Jack whitepaper: Jack an agent infrastructure for providing the decision-making capability required for autonomous systems, 2008.

[AOS11a] AOS Group. C-bdi website: http://www.agent-software.com/products/c-bdi/index.html, 2011.

[AOS11b] AOS Group. Practical jack beliefset relations: http://www.aosgrp.com/documentation/jack/practicals/jack_jde/html/p2n.html, 2011.

[Bad10] S. Bader. A logic (programming) based controller for smart environments, 2010.

[Bau11] M. Bauer. Uni-protokolle.de: http://www.uni-protokolle.de/lexikon/echtzeit.html, 2011.

[BBH+11] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Jacamo: http://jacamo.sourceforge.net/, 2011.

[BBJ+02] R. H. Bordini, A. L. C. Bazzan, R. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. Agentspeak(xl): Efficient intention selection in bdi agents via decision-theoretic task scheduling. *AAMAS'02*, 2002.

[BH06] R. Bordini and J. F. Hübner. Bdi agent programming in agentspeak using jason (tutorial paper), 2006.

[BH07] R. H. Bordini and J. F. Hübner. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley series in agent technology. Wiley, Chichester, 2007.

[BH11] R. H. Bordini and J. F. Hübner. Jason development files: http://sourceforge.net/projects/jason/files/, 2011.

[BHV+05] R. H. Bordini, J. F. Hübner, R. Vieira, P. O. Naso, and M. Golding. Chapter 1 jason and the golden fleece of agent-oriented programming. *Multiagent systems, artificial societies, and simulated organizations.*, (15):3–37, 2005.

[BIP88] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence.*, 4(3):349–355, 1988.

[BOO+04] R. H. Bordini, F. Y. Okuyama, D. de Oliveira, G. Drehmer, and R. C. Krafta. The mas-soc approach to multi-agent based simulation. *Lecture notes in artificial intelligence.*, pages 70–91, 2004.

[BP09] L. Braubach and A. Pokahr. Go4flex: http://jadex.informatik.uni-hamburg.de/go4flex/bin/view/about/overview, 2009.

[BP10] L. Braubach and A. Pokahr. Jadex: Bdi agent system, 2010.

[BPML04] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for bdi agent systems. pages 9–20, 2004.

[Bra81] M. E. Bratman. Intention and means-end reasoning. *The philosophical review Vol. 90, No. 2.*, (90):252–265, 1981.

[Bra87] M. E. Bratman. *Intention, plans, and practical reason.* Harvard University Press, Cambridge, Mass, 1987.

[BRK10] S. Bader, G. Ruscher, and T. Kirste. A middleware for rapid prototyping smart environments: experiences in research and teaching. In *Ubiquitous Computing/Handheld and Ubiquitous Computing*, pages 355–356, 2010.

[BRK11] S. Bader, G. Ruscher, and T. Kirste. The helferlein-system, a middleware for dynamic and heterogeneous ensembles: http://code.google.com/p/helferlein/, 2011.

[Bro85] R. A. Brooks. A robust layered control system for a mobile robot, 1985.

[BRS85] R. Barrett, A. Ramsay, and A. Sloman. *POP-11: A practical language for artificial intelligence.* E. Horwood and Halsted Press, Chichester, New York, 1985.

[CBB00] R. G. G. Cattell, D. K. Barry, and M. Berler. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, San Francisco, 2000.

[CL90] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial intelligence.*, 42:213–261, 1990.

[CL91] P. R. Cohen and H. Levesque. Teamwork, 1991.

[Coo09] D. J. Cook. Multi-agent smart environments. *Ambient intelligent smart environments*, 1:51–55, 2009.

[CPR08] G. Caire, A. Poggi, and G. Rimassa. Jade: A software framework for developing multi-agent applications. lessons learned. *Information and software technology.*, 50(1-2):10–21, 2008.

[Das08] M. Dastani. *Languages, methodologies and development tools for multi-agent systems: First International Workshop, LADS 2007, Durham, UK, September 4-6, 2007.* Springer, Berlin ;, New York, 2008.

[Doy94] J. Doyle. Rationality and its roles in reasoning. *Computational intelligence.*, 8:376–409, 1994.

[Dyr11] M. Dyrba. Design and implementation of a smartlab controller based on the subsumtion architecture: Diplomarbeit, 01.03.2011.

[Ext11] Extron. Extron electronics: www.extron.com, 2011.

[EZ92] S. Ericsson-Zenith. *Process Interaction Models.* PhD thesis, Universität Paris, Paris, 1992.

[Fra09] K. Frank. *EIB/KNX: Grundlagen Gebäudesystemtechnik.* Huss-Medien, Berlin, 4., aktualis edition, 2009.

[FSF07] Inc Free Software Foundation. Gnu lesser general public license: http://www.gnu.org/licenses/lgpl.html, 2007.

[GL87] M. P. Georgeff and A. L. Lansky, editors. *Reactive reasoning and planning*, AAAI'87. AAAI Press, 1987.

[HBF+10] J. F. Hübner, R. H. Bordini, M. Fisher, Martins J., Á. F. Moreira, R. Vieira, W. Visser, and M. Wooldridge. Jason a java-based interpreter for an extended version of agentspeak: http://jason.sourceforge.net/jason/jason.html, 2010.

[HKS01] T. Herfet, T. Kirste, and M. Schnaider. Embassi: Multimodal assistance for infotainment and service infrastructures. *Comupter & graphics*, (25):581–592, 2001.

[HL06] A. Heinzl and W. Lamersdorf. Medpage: http://vsis-www.informatik.uni-hamburg.de/projects/medpage/index.php, 2006.

[HRHL01] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - summary of an agent infrastructure. *Fifth international conference on autonomous agents.*, 2001.

[HS03] J. F. Hübner and J. S. Sichman. Simple agent communication infrastructure (saci), 2003.

[HSB02] J. F. Hübner, J. S. Sichman, and O. Boissier. A model for the structural, functional, and deontic specification of organizations in multiagent systems, 2002.

[HSB07] J. F. Hubner, J. S. Sichman, and O. Boissier. Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. *Agent-oriented software engineering*, 1:370–395, 2007.

[HT02] T. Heider and Thomas K. Supporting goal-based interaction with dynamic intelligent environments. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 596–600, Lyon, France, 2002.

[Hub99] M. J. Huber. Jam: A bdi-theoretic mobile agent architecture. *in proceedings of the third international conference on autonomous agents.*, pages 236–243, 1999.

[Hub08] M. J. Huber. Intelligent reasoning systems: http://marcush.net/irs/download_jam.html, 2008.

[IGR92] F. F. Ingrand, M. P. Georgeff, and A. Rao. An architecture for real-time reasoning and system control. *Intelligent systems and their applications.*, 7:34–44, 1992.

[JHN04] L. Jain, R. J. Howlett, and M. G. Negoita. *Knowledge-based intelligent information and engineering systems: Proceedings.* Springer, Berlin ;, New York ;, Paris, 2004.

[Koc10] C. Koch. Networkpublisher handbuch, 2010.

[KP93] K. Konolige and M. E. Pollack. A representationalist theory of intention. *in proceedings of the thirteenth international joint conference on artificial intelligence.*, pages 390–395, 1993.

[LE98] C. Lemaître and C. B. Excelente. Multi-agent organization approach. *Proceedings of second iberoamerican workshop on DAI and MAS*, 1998.

[LHDK94] J. Lee, M. J. Huber, E. H. Durfw, and P. G. Kenny. An implementation of the procedural reasoning system for multirobot applications. *In conference on intelligent robotics in field, factoy, service, and space.*, pages 1232–1237, 1994.

[Lin01] J. Lind. *Iterative software engineering for multiagent systems: The massive method.* Springer, Berlin ;, New York, 2001.

[LLJ08] J. Leng, J. Li, and L. C. Jain. A role-based framework for multi-agent teaming. pages 642–649, 2008.

[LLR96] J. F. Lehman, J. Laird, and P. Rosenbloom. A gentle introduction to soar, an architecture for human cognition. *In invitation to cognitive science*, 1996.

[LW04] M. Luck and M. Wooldridge. The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous agents and multi-agent systems.*, 9:1–2, 2004.

[Mae90] P. Maes. Situated agents can have goals. *Designing autonomous agents*, 6:49–70, 1990.

[MB01] R. Machado and R. H. Bordini. Running agentspeak(l) agents on sim agent. *In pre-proceedings of the eighth international workshop on agent theories, architectures, and languages.*, 2001.

[MBC+06] V. Morreale, S. Bonura, F. Centineo, M. Cossentino, and S. Gaglio. Reasoning about goals in bdi agents: the practionist framework. *In proceedings of joint workshop "'From objects to agents"'*, 2006.

[MMI11] MMIS Universität-Rostock. Mmis universität-rostock: http://mmis.informatik.uni-rostock.de/index.php?title=mmis_home, 2011.

[Moz05] M. C. Mozer. 12 lessons from an adaptive home, 2005.

[MR97] T. J. Mowbray and W. A. Ruh. *Inside CORBA: Distributed object standards and applications.* Addison-Wesley, Reading, Mass, 1997.

[MR03] D. D. McCracken and E. D. Reilly. Backus-naur form (bnf). pages 129–131, 2003.

[MVB02] Á. F. Moreira, R. Vieira, and R. H. Bordini. Extending the operational semantics of a bdi agent-oriented programming language for introducing speech-act based communication. pages 135–154, 2002.

[MW97] K. L. Myers and D. E. Wilkins. The act formalism, version 2.2., 1997.

[Nic09] R. Nicolay. Smartspace: Eine kommunikations-infrastruktur für intelligente umgebungen: Studienarbeit, 30.09.2009.

[Nyo10] M. Nyolt. Publish-subscribe: Praktikumsbericht universität rostock, 2010.

[ORV07] A. Omicini, A. Ricci, and G. Vizzari. Building smart environments as agent workspaces. *Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 92–97, 2007.

[PWP05] T. Phung, M. Winikoff, and L. Padgham. Learning within the bdi framework: An empirical analysis. pages 282–288, 2005.

[Rao96] A. S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. *Lecture notes in computer science.*, (1038):42–55, 1996.

[Rao98] A. S. Rao. Decision procedures for bdi logics. *Journal of logic and computation.*, 8(3):293–342, 1998.

[RC04] S. Russell and J. F. Canny. *Künstliche Intelligenz: Ein moderner Ansatz.* Pearson Studium, München ;, Boston [u.a.], 1 edition, 2004.

[RG91] A. S. Rao and M. P. Georgeff. Modeling rational agents within a bdi-architecture, 1991.

[RG95] A. S. Rao and M. P. Georgeff. Bdi agents: From theory to practice. *In proceedings of the first international conference on multi-agent systems.*, pages 312–319, 1995.

[Rie00] D. Riehle. *Framework design: A role modeling approach.* [s.n.], [S.l.], 2000.

[RK86] S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In *Proceedings of the 1986 conference on Theoretical aspects of reasoning about knowledge*, TARK '86, pages 83–98, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[RMSM92] A. S. Rao, D. Morley, A. Selevestrel, and G. Murray. Representation, selection, and execution of team tactics in air combat modelling. *Proceeding of the 5th Australian Joint Conference on Artificial Intelligence*, pages 185–190, 1992.

[RS10] A. Ricci and A. Santi. Cartago by example, 01.08.2010.

[RT08] C. Reisse and Thomas K. A distributed mechanism for device cooperation in smart environments. In *Advances in Pervasive Computing. Adjunct proceedings of the 6th International Conference on Pervasive Computing*, pages 53–56, Sydney, Australia, 2008.

[RVO06] A. Ricci, M. Viroli, and A. Omicini. Cartago: An infrastructure for engineering computational environments. *Third international workshop "'environments for mulit-agent systems"'*, pages 102–119, 2006.

[SA90] M. Singh and N. Asher. Towards a formal theory of intentions. *Logics in AI*, (478):472–486, 1990.

[SB98] R. S. Sutton and A. G. Barto. Reinforcement learning i: Introduction, 1998.

[Sho93] Y. Shoham. Agent-oriented programming. *Artificial intelligence.*, 60:51–92, 1993.

[SSR+03] N. A. Streitz, N. Streitz, C. Röcker, T. Prante, R. Stenzel, and D. van Alphen. Situated interaction with ambient information: Facilitating awareness and communication in ubiquitous work environments, 2003.

[Sun11] Sun Microsystems. Jini.org: http://www.jini.org/wiki/main_page, 2011.

[Ter08] L. Terrenghi. Design for interaction in instrumented environments, 2008.

[The11] The Eclipse Foundation. Eclipse downloads: http://www.eclipse.org/downloads/, 2011.

[Tid93] G. Tidhar. Team-oriented programming: Preliminary report, 1993.

[TPH02] J. Thangarajah, L. Padgham, and J. Harland. Representation and reasoning for goals in bdi agents. *Australian computer science communications*, January(24):259–265, 2002.

[Ubi11] Ubisense. Ubisense real-time-location-system: http://www.ubisense.net/, 2011.

[UH11] Verteilte Systeme und Informationssysteme Universität Hamburg. Jadex bdi agent system: http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/about/overview, 2011.

[Uni10] University of Birmingham. The birmingham cognition and affect project: http://www.cs.bham.ac.uk/research/projects/cogaff/, 2010.

[US11] Universität von Birmingham and A. Sloman. The simagent toolkit – for philosophers and engineers (and some biologists, psychologists and social scientists): http://www.cs.bham.ac.uk/research/projects/poplog/packages/simagent.html, 2011.

[vL92] J. van Leeuwen, editor. *Handbook of theoretical computer science*. Elsevier and The MIT Press, Amsterdam [etc.], Cambridge, Mass, 2nd impression. edition, 1992.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific american.*, 265(3):94–104, 1991.

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36:75–84, 1993.

[Wei00] G. Weiss, editor. *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 2000.

[WJ95] M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge engineering review.*, 10:115–152, 1995.

[WM95] D. E. Wilkins and K. L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of logic and computation.*, 5(6):731–761, 1995.

[Woo00] M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*, pages 27–77. MIT Press, 2000.

[Woo09] M. J. Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, Chichester, U.K, 2nd edition, 2009.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe. Ferner versichere ich, keine anderen Hilfsmittel als die angegebenen Quellen benutzt zu haben. Alle Ausführungen in der Arbeit, die wörtlich oder sinngemäß aus fremden Quellen entnommenen wurden, sind als solche gekennzeichnet.

Rostock, den 29. Juni 2011 Robin Nicolay