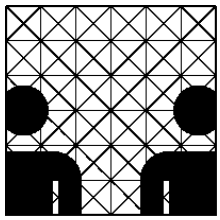
Universität Hamburg
Fachbereich Informatik



Ausführungsumgebung für FIPA Interaktionsprotokolle am Beispiel von Jadex

Diplomarbeit am Arbeitsbereich
Verteilte Systeme und Informationssysteme
von

Alexander Scheibe

Betreuer
Prof. Dr. Winfried Lamersdorf
Dr. Daniel Moldt

Tag der Abgabe:
1. Februar 2005

Inhaltsverzeichnis

1. Einleitung

Agentenorientierte Softwareentwicklung erfährt seit Ende der 90er Jahre einen immer größeren Forschungszuspruch. Im Gegensatz zur objektorientierten Softwareentwicklung, die auf Interobjektkommunikation durch Methodenaufrufe basiert, ist der Nachrichtenaustausch zwischen den Agenten die Grundlage der agentenorientierten Softwareentwicklung.

Kommunikation in Agentensystemen lässt sich in drei Bereiche aufteilen: ad-hoc, entwicklereigene und protokollbasierte Kommunikationsmechanismen [Chiab-Draa and Dignum, 2002]. Bei der Ad-hoc-Kommunikation sind keine Kommunikationsvorschriften zwischen den Kommunikationspartnern abgestimmt worden. Sie entsteht dem ausfreien und autonomen Handeln der Agenten und setzt eine große Zahl von Annahmen über die Kommunikationsfähigkeiten der Partner voraus. Bei entwicklereigenen Kommunikationsmechanismen sind Annahmen über die Fähigkeiten der Partner nicht notwendig, sodass eine Mischung aus ad-hoc und geregelter Kommunikation entsteht. Entwicklereigene Mechanismen erlauben schon vom Namen her nur die Kommunikation in einer eingegrenzten Domäne, durch ad-hoc Verfahren kann hingegen eine quasi unbegrenzte Menge von Agenten miteinander kommunizieren. Jedoch ist bereits bei einer Zahl von mehr als einhundert Agenten eine Ad-hoc-Kommunikation eher hinderlich [Walton, Robertson, 2002]. Um hierbei einen geregelten Verständigungsablauf zu erhalten, ist die dritte Form, Kommunikationsprotokolle, Grundlage des Nachrichtenaustauschs. Protokolle legen die Form und Reihenfolge von Kommunikationsakten fest und gewährleisten so die Verständigung zwischen Agenten.

Im Zuge der wachsenden Verbreitung von Agentensystemen steigt auch die Notwendigkeit für interoperable Agenten standardisierte Kommunikationsprotokolle und Infrastrukturen zu spezifizieren. Dieses wird zum Beispiel von der Foundation for Intelligent Physical Agents (FIPA) vorangetrieben. Die FIPA ist eine gemeinnützige Organisation aus Unternehmen und Universitäten, deren Ziel es ist,

> „*[...] to promote the success of emerging agent-based applications, services and equipment*“ [FIPA 2001a].

Dieses Ziel wird erreicht

> *„[...] by making available in a timely manner, international agreed specifications that maximise interoperability across agent-based applications, services and equipment."* [FIPA 2001a].

Um diese Interoperabilität zu erreichen legt, die FIPA einen ihrer Schwerpunkte auf die Festlegung von Standards bei Protokollen, Kommunikationsakten und beim Nachrichtenformat.

Um den technischen Aufwand bei der Entwicklung von protokollkonformen Agenten zu verringern, gibt es verschiedene Ansätze. Diese Ansätze reichen von Werkzeugen zur graphischen Modellierung und Codegenerierung bis hin zu in Agenten-Frameworks integrierten Ansätzen. Auch bei der Modellierung von Protokollen auf Agentenebene gibt es verschiedene Ansätze. In auf Petrinetzen basierenden Systemen ist häufig die Modellierung eines Kommunikationspartners in einem einzigen Petrinetz zu finden, so zum Beispiel auf der Mulan-Plattform [Köhler, Moldt, Rölke, 2001]. Bei anderen Systemen werden die einzelnen Partner eines Protokolls durch mehrere kleine Einheiten, die zumeist als Pläne bezeichnet werden, abgearbeitet, wie dies bei der Jadex-Plattform [Pokahr, Braubach, Lamersdorf, 2003] der Fall ist.

Ob nun bei der Modellierung als Ganzes oder in kleineren Einheiten, es muss bei den gegenwärtigen Ansätzen jeweils im Programmcode dafür Sorge getragen werden, dass Syntax und Semantik der Protokollspezifikation eingehalten werden. Entwicklungswerkzeuge können zwar in der Programmierphase zur Konformität beitragen, jedoch kann diese während der Laufzeit in kaum einem Agentensystem überprüft werden.

1.1. Zielsetzung

In diesem Zusammenhang soll im Rahmen dieser Arbeit eine Methode bzw. ein Werkzeug entwickelt werden, um drei Ziele zu verfolgen. Als erstes soll ein Werkzeug erstellt werden, das Agenten die Protokollausführung erleichtert und zugleich das protokollkonforme Verhalten der Agenten sicherstellt. Des Weiteren soll eine Trennung zwischen Protokollfluss und der Implementierung der einzelnen Protokollelemente und damit eine Trennung zwischen Protokolldesign und dessen Implementierung erreicht werden. Zusätzlich soll dieses Werkzeug für Agenten unterschiedlicher Agentenplattformen verwendbar sein.

Eine Sicherstellung der Protokollkonformität zur Laufzeit ermöglicht dabei eine Entlastung der Entwickler von Agenten in der Testphase des Entwicklungszyklus und kann diese dadurch verkürzen. Durch eine Trennung von Protokollfluss und Implementierung der einzelnen Protokollelemente im Zusammenspiel mit einem Werkzeug, das die Protokollausführung sicherstellt, kann erreicht werden, dass einzelne Protokollelemente auch für ähnliche Zwecke in anderen Protokollen verwendet werden können. Die daraus folgende Trennung zwischen Protokolldesign und Implementierung ermöglicht zudem die Spezifikation von Protokollen unabhängig vom Agentensystem, wodurch diese austauschbar ist und in anderen Systemen verwendet werden kann.

Im praktischen Teil der Arbeit soll solch ein Werkzeug entwickelt werden und es sollen seine Funktionen im Zusammenhang mit der am Fachbereich Informatik der Universität Hamburg entwickelten BDI-Agentenplattform Jadex in Form einer Prototypimplementierung unter Beweis gestellt werden.

1.2. Vorgehen

Zunächst sollen die Grundlagen des Themas, Agentensysteme, Kommunikation und Protokolle, näher betrachtet werden. Darauf aufbauend werden zunächst Werkzeuge und Methoden untersucht, die auf unterschiedliche Art und Weise eine Unterstützung für Protokolle in Agentensystemen bieten. Die Untersuchung soll die Aspekte dieser Ansätze zusammen mit dem Rahmen der Zielsetzung, zu Kriterien führen, die ein Werkzeug aus der Zielsetzung erfüllen können sollte.

Basierend darauf soll eine abstrakte Architektur entwickelt werden. Diese Architektur soll die gewonnenen Ergebnisse berücksichtigen und die gewünschte Trennung zwischen Protokollfluss und der Implementierung erfüllen. Aufbauend auf die abstrakte Architektur soll ein Prototyp erstellt werden. Um die Trennung zwischen Protokolldesign und dessen Implementierung zu ermöglichen, soll eine Protokollspezifikation ausgearbeitet werden.

Mit Hilfe des Prototypen soll dann eine Beispielprotokollausführung implementiert werden. Durch diese Implementierung sollen die Vorteile und Perspektiven durch eine Ausführungsumgebung für FIPA Interaktionsprotokolle gezeigt werden.

1.3. Gliederung

In Kapitel 2 werden zunächst die Grundlagen des Themas eingeführt. Als erstes wird eine kurze Einführung in Multi-Agentensysteme im Allgemeinen und BDI-Systeme im Speziellen gegeben. Es folgt ein Überblick über die Kommunikationsgeschichte in Agentensystemen und eine Vorstellung protokollorientierter Kommunikation. Den Abschluss des Kapitels bildet eine Einführung in die Agent UML mit ihrer Ausprägung Sequenz-Diagramme zur graphischen Darstellung von Protokollabläufen in Agentensystemen.

In Kapitel 3 wird eine abstrakte Architektur zur Implementierung von Protokollen für Agenten eines Agentensystems beschrieben. Einleitend wird eine Einteilung vorhandener Werkzeuge und Methoden zur Unterstützung protokollorientierter Kommunikation in verschiedene Kategorien, von Codegeneratoren über unterstützende und codeausführenden Tools zu Verifikationswerkzeugen, vorgenommen. Anschließend werden die Charakteristika dieser Klassen untersucht. Darauf aufbauend wird eine abstrakte Architektur entwickelt und die grundlegenden Funktionen dieser beschrieben.

In Kapitel 4 wird schließlich eine Umsetzung der abstrakten Architektur in Form eines Prototypen zusammen mit einer Protokollspezifikation auf Basis von XML vorgestellt. Gefolgt von einem Beispiel, soll die Funktion des Prototypen und des Gesamtkonzepts vorgeführt werden.

In Kapitel 5 endet diese Arbeit mit einer abschließenden Zusammenfassung und Bewertung der Ergebnisse und einem kurzen Ausblick.

2. Grundlagen

In diesem Kapitel werden zunächst die grundlegenden Aspekte des Themas – Agentensysteme, Kommunikation und Protokolle – eingeführt und erläutert.

2.1. Multi-Agentensysteme

Dem allgemeinen Trend zur Vernetzung und Verteilung von Computersystemen folgend ist eine Tendenz zur Erweiterung der Fähigkeiten dieser Systeme zu sehen. Diese Erweiterungen betreffen Fähigkeiten, die den Systemen erlauben selbständig mit anderen Systemen zusammenzuarbeiten, sowie Fähigkeiten zur Kooperation und zum Treffen von Übereinkünften mit anderen Computersystemen. Dieses sind Eigenschaften, die durch Agenten in einem Multi-Agentensystem erreicht werden können. Dabei gibt es für den Begriff Agent keine einheitliche Bestimmung in der Informatik, da er weitläufig von vielen Personen benutzt wird [Wooldridge und Jennings, 1995]. Daher wird die folgende Definition aus [Wooldridge, 2002] als Basis für den Begriff Agent in dieser Arbeit benutzt:

> An *agent* is a computer system that is *situated* in some *environment* and that is capable of *autonomous* action in this environment in order to meet its design objectives [Wooldridge, 2002].

Multi-Agentensysteme sind also Systeme aus solchen Agenten, in denen diese zusammenarbeiten, sich abstimmen und Vereinbahrungen treffen, um ihre Ziele erreichen. Dieses geschieht autonom, das heißt, ohne dass den Agenten zu jedem Zeitpunkt vorgeschrieben ist, was diese zu tun haben. Die Ziele, die von den einzelnen Agenten in einem Multi-Agentensystem verfolgt werden, müssen nicht konform, sondern können durchaus konkurrierend oder gegensätzlich sein. Dieses führt zu drei weiteren Eigenschaften von Agenten, die die Autonomie ergänzen (nach [Wooldridge und Jennings, 1995]):

- **Reaktivität**: Agenten nehmen ihre Umgebung wahr und reagieren zeitgemäß auf auftretende Veränderungen um ihre Ziele zu erreichen.

- **Proaktivität**: Agenten sind nicht nur in der Lage, auf ihre Umgebung zu reagieren, sondern haben ein zielgelenktes Verhalten, in dem sie selbst die Initiative ergreifen.

- **Soziale Fähigkeiten**: Um ihre Ziele zu erreichen sind Agenten in der Lage, mit anderen Agenten (unter Umständen auch Menschen) zu interagieren.

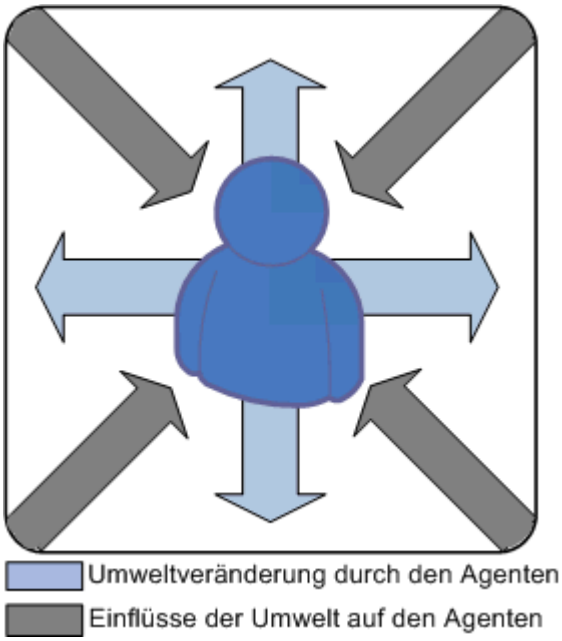


Abbildung 2.1: Agent in seiner Umwelt. Umwelt und Agent beeinflussen sich gegenseitig.

In Anlehnung an [Wooldridge, 2002] zeigt Abbildung 2.1 die Situation eines Agenten, der in Wechselwirkung mit seiner Umwelt steht, durch Signale dieser beeinflusst wird und Veränderungen in ihr verursacht. In der Regel verhält es sich für einen Agenten dabei so, dass dieser nur Teile seiner Umgebung wahrnimmt. Dieses partielle Wissen über seine Umwelt kann bewirken, dass gleichartige Aktionen des Agenten verschiedenartige Effekte in seiner Umwelt hervorrufen. Eine solche dynamische Umgebung verlangt vom Agenten, dass dieser bereits vor dem Beginn einer Aktion vorhersehen muss, ob zum Zeitpunkt, an dem diese Aktion beendet sein wird, die Bedingungen für die Aktion noch gegeben sein werden. Und selbst wenn er initiativ eine Handlung beginnt, kann es sein, dass er, veranlasst durch eine Veränderung in seiner Umwelt, reagiert und seine Handlung wieder abbricht.

Die Dynamik der Umwelt, die es dem Agenten quasi auferlegt, Vorhersagen über den zukünftigen Zustand dieser zumachen, führt zu einer weitergehenden Art von Agenten. Diese Art von Agenten führt in die Bereiche der künstlichen Intelligenz; zu Agenten, die eher mit dem Menschen vergleichbar sind. Solche Agenten folgen mentalistischen Konzepten, wie etwa Wissen, Glauben, Absicht und Pflicht [Shoham, 1990]. Ein solcher Ansatz sind Practical Reasoning Systeme. Practical Reasoning ist Schlussfolgern geleitet

durch Handlungen – der Prozess des Herausfindens, was zu tun ist [Wooldridge, 2002]. Das erfahrungsbasierte Schlussfolgern besteht aus zwei Teilgebieten, der Deliberation und dem Means-Ends Reasoning. Die Deliberation (zu Deutsch Überlegung) ist der Prozess der Entscheidungsfindung für ein Ziel für eine Aufgabe. Means-Ends Reasoning ist ein Prozess, eine Methode zum Erreichen eines Ziels bzw. zur Vollendung einer Aufgabe zu finden. Ein auf Practical Reasoning aufsetzender Ansatz ist der BDI-Ansatz, der im nächsten Abschnitt vorgestellt wird.

2.1.1. BDI-Agentensysteme

Als ein technisches Gegenstück zur menschlichen Handlungsbasis kann das Belief-Desire-Intention (BDI) Modell [Bratman, 1987] [Rao und Georgeff 1991] für Agentensysteme gesehen werden. Das BDI-Modell gilt als bekanntestes und am weitesten erforschtes Modell von Practical Reasoning Agenten [Georgeff et al., 1999].

Die mentalistischen Konzepte die dem BDI-Modell zugrunde liegen, sind die drei mentalen Attitüden Belief (Glauben), Desire (Wünsche) und Intention (Absichten). Diese bilden die Basis für ein auf Erfahrung basierendes zielorientiertes Handeln. Beliefs repräsentieren dabei Informationen, die der Agent in seiner Umwelt wahrgenommen und interpretiert hat. Die Beliefs stellen also eine Art von eigener Meinung eines Agenten über Objekte in dessen Kontext da. Dieses bedeutet insbesondere, dass der Agent seine Beliefs immer für konsistent hält, dieses aber nicht unbedingt aus Sicht eines anderen Agenten richtig sein muss. Zusätzlich können Beliefs auch über eigene oder fremde Belief reflektieren, sodass der Agent eine ihm eigene Weltanschauung besitzt.

Desires stellen Wünsche eines Agenten dar. Desires sind eine Menge abstrakter Ziele, die der Agent prinzipiell erreichen möchte, jedoch nicht aktiv verfolgt. Die Menge der Desires muss dabei keine konsistente Menge sein, sondern kann auch zueinander inkonsistente Desires enthalten, also Wünsche die nicht gleichzeitig befriedigt werden können. Desires, die zueinander konsistent sind, werden als Goals (Ziele) bezeichnet [Rao und Georgeff, 1991]. Insbesondere gilt für ein Goal, dass der Agent glaubt, es sei erreichbar. Mit Hilfe seiner Desires und (insbesondere) seiner Goals kann ein Agent zielgerichtetes proaktives Verhalten entwickeln.

Intentions repräsentieren Handlungen des Agenten, die dieser beabsichtigt zu erfüllen, also eine Menge von konkreten Zielen. Intentions müssen kompatibel mit den Beliefs und Desires eines Agenten sein. Kompatibilität bedeutet hierbei, dass ein Agent niemals etwas beabsichtigt, wenn er kein zugehöriges Goal hat und/oder seine Beliefs dagegen sprechen.

2.1.2. Pläne in BDI-Systemen

Ein weiteres Merkmal von BDI-Systemen sind Pläne. Pläne sind eine Art Rezept und beschreiben, wie der Agent bestimmte Dinge erreichen kann. Sie können als ein Teil der Beliefs des Agenten betrachtet werden [Bratman, Pollack und Israel, 1988] und befinden sich in der so genannten Plan-Library[1] des Agenten. Pläne spielen in das Means-Ends Reasoning hinein und werden vom Agenten benutzt, um seine Desires zu verwirklichen. Dabei analysiert der Agent laufend seine Möglichkeiten, um seine Desires erfüllen zu können; dieses wird als Deliberationsprozess bezeichnet. Ist er zum Entschluss gekommen, ein Desire konkret zu verfolgen, also eine Intention gebildet hat, sucht er sich im Means-Ends-Reasoning-Prozess passende Pläne zusammen. Abbildung 2.2 zeigt einen solchen Prozess.

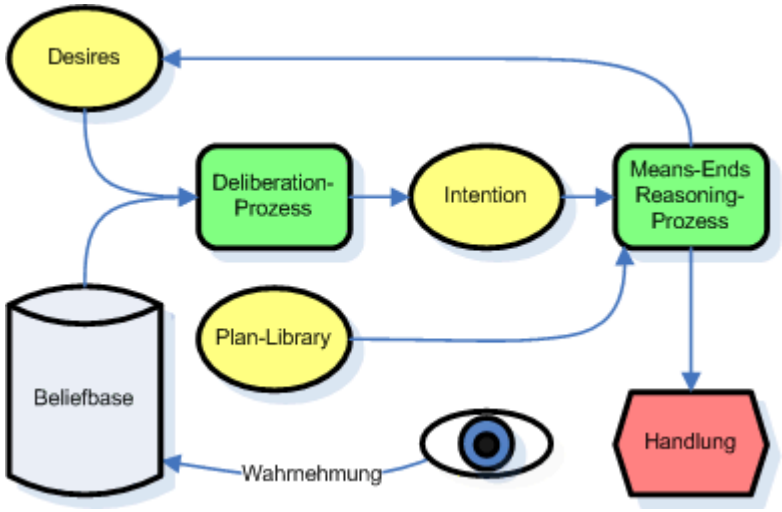


Abbildung 2.2: Reasoning Prozess

Ebenso wie für Goals gilt für Pläne, dass sie konsistent sein müssen, sowohl intern als auch zu den Beliefs. Da Pläne genauso wie das Wissen eines Agenten nicht allumfassend sein können, sollten Pläne partiell sein, dies fördert die Stabilität eines Plans [Bratman, Pollack und Israel, 1988]. Partiell bedeutet hierbei, dass in einem Plan nicht alle Umwelteinflüsse berücksichtigt werden sollten. Würde ein Plan diese berücksichtigen, könnten kleinste Veränderung, von für den Plan relevanten Umweltaspekten die Randbedingungen für dessen Ausführung verändern und dieser somit nicht mehr stabil ausgeführt werden. Pläne können aber auch partiell gegenüber dem Reaso-

[1] Dem gegenüber bezeichnet man als Beliefbase den Ort, an dem die Beliefs verwaltet werden.

ning-Prozess in dem Sinne sein, dass zum Beispiel ein Plan nicht vollständig zur Erfüllung einer Intention beiträgt. Solche Pläne können dann Subgoals (also neue Desires) erzeugen, die Ausgangspunkte für einen weiteren Prozess aus Deliberation und Means-Ends Reasoning sind, um die ursprüngliche Intentsion zu erfüllen.

2.2. Kommunikation

Kommunikation in Multi-Agentensystemen beruht traditionell auf Austausch von Nachrichten, die der Sprechakttheorie folgen. Die Sprechakttheorie geht auf John L. Austin [Austin, 1962] und John R. Searle [Searle 1969] zurück. Die drei wichtigsten Aspekte der Sprechakttheorie für die Kommunikation in Multi-Agentensystemen sind der illokutionäre und der perlokutionäre Akt sowie die Proposition. Der illokutionäre Akt wird in der Agentenkommunikation durch performative Verben explizit gemacht und gibt an, wie eine Aussage aufzufassen ist (Zum Beispiel: „Ich frage dich“). Der perlokutionäre Akt beschreibt den Effekt einer Äußerung (hier also von der angesprochenen Person eine Antwort zu erhalten). Die Proposition stellt den eigentlichen Inhalt einer Äußerung dar [ILMES].

Damit Menschen sich miteinander unterhalten können, müssen sie die gleiche Sprache sprechen für Agenten bedarf es dazu zuerst einmal eines einheitlichen Nachrichtenformates. Das Nachrichtenformat wird in Agentensystemen als Agent Communication Language, kurz ACL, bezeichnet. Eine ACL gibt Agenten die Möglichkeit, Informationen und Wissen auszutauschen. In den verbreitetsten ACL, der Knowledge Query and Manipulation Language (KQML) und FIPA ACL, sind Proposition, Illokution und Perlokution grundlegende Elemente.

KQML wurde seit 1990 federführend von der Defense Advanced Research Projects Agency (DARPA) einer Behörde des US-Verteidigungsministeriums innerhalb des Knowledge Sharing Effort Projektes (KSE) entwickelt. Ziel bei der Entwicklung von KQML war es, eine einheitliche Sprache zum Austausch von Wissen zu schaffen. KQML ist die am weitesten verbreitete ACL und kann als de facto Standard betrachtet werden. FIPA ACL ist die ACL der FIPA; FIPA ACL ist syntaktisch mit KQML verwandt und unterscheidet sich nur in wenigen Parametern und hauptsächlich in der Definition der illokutionären Performative.

Performative sind in FIPA ACL und KQML das initiale Attribut einer Nachricht (siehe Abbildung 2.3), die restlichen Elemente werden als Parameter des Performatives bezeichnet und werden als Schlüssel-Wert-Paare notiert.

```
1)
(ask-one
      :content (PRICE PORSCHE BOXSTER ?price)
      :language PROLOG
      :reply-with autoboerse
      :ontology schwacke-liste
      :sender otto
      :receiver gebrauchtwagen-karl)

2)
(tell
      :content (PRICE PORSCHE BOXSTER 50000)
      :language PROLOG
      :in-reply-to autoboerse
      :ontology schwacke-liste
      :sender gebrauchtwagen-karl
      :receiver otto)
```

Abbildung 2.3: Beispiel KQML Nachrichten: (1) Frage von Agent Otto über den Preise eines gebrauchten Boxsters und (2) die Antwort von Gebrauchtwagen-Karl

Das Performativ legt fest, wie der Empfänger einer Nachricht diese zu interpretieren hat, was die Intention des Senders ist und was dieser vom Empfänger erwartet. Bei der FIPA werden zur textlichen Beschreibung der Performative auch formale Modelle definiert. Zum Beispiel das Performativ „inform“ als:

$$\langle i, \text{inform}\ (j, \phi) \rangle$$
$$\text{FP: } B_i\phi \wedge \neg B_i(\text{Bif}_j \vee \text{Uif}_j\phi)$$
$$\text{RE: } B_j\phi$$

> Dies bedeutet: Agent i informiert Agent j über die Proposition ϕ , Agent i hält ϕ für gültig und glaubt nicht, dass Agent j irgendein Wissen über ϕ hat und als Resultat wird erwartet, dass Agent j ϕ für gültig hält.

Die FIPA bezeichnet Performative auch als kommunikative Akte (Communicative Acts) und spezifiziert 22 Akte in der Communicative Act Library [FIPA, 2002]. Die DARPA hat für KQML etwa 34 Performative vor definiert [Finn et al., 1994]. Der Unterschied in der Anzahl der Performative kommt dadurch zustande, dass die Performative der FIPA etwas allgemeiner gefasst sind, wobei bei beiden die Performative ein weites Einsatzspektrum abdecken. Beide ACL sind jedoch nicht auf die jeweilig spezifizierten Performative beschränkt, sondern erlauben auch eigene, domänenspezifische Performative zu definieren. Bei der Verwendung der vordefinierten Performative ist jedoch darauf zu achten, dass diese der Spezifikation entsprechend verwendet werden.

Zwei weitere wichtige Elemente in einer ACL-Nachricht sind der Language- und der Ontology-Parameter. Der Parameter Language spezifiziert die Sprache, in der der Inhalt ausgedrückt wird. In keiner der beiden genanten ACL ist eine bestimmte Inhaltssprache festgeschrieben worden. Innerhalb der KSE wurde jedoch eine Sprache namens KIF (Knowledge Interchange Format) entwickelt, eine logische Sprache, die als Standardsprache für den Austausch von Wissen zwischen unterschiedlichen Computersystemen betrachtet werden kann [Lobrou, Finin, Peng, 1999]. Eine KIF Spezifikation von der FIPA befindet sich ebenfalls in der Ausarbeitung. Eine in FIPA-ACL-Nachrichten häufig verwendete Inhaltssprache ist die FIPA Semantic Language (SL), eine sehr aussagekräftige Sprache, für die aufgrund ihrer Mächtigkeit drei hierarchische Untermengen für verschiedene Kommunikationsaufgaben geschaffen wurden sind (FIPA-SL0 $\subset$ FIPA-SL1 $\subset$ FIPA-SL2).

Der Ontology-Parameter ist genau wie der Language-Parameter ein inhaltsbeschreibender Parameter, in dem Sinne, dass er den Kontext, aus dem die Worte des Inhalts stammen, definiert und so deren Interpretation ermöglicht. So kann zum Beispiel die Ontologie „schwacke-liste“ aus Abbildung 2.3 dem Empfänger bei der Interpretation helfen, dass das Wort „`PRICE`“ dem Preis in einer Schwacke-Liste entspricht, „`Porsche`“ eine Automarke und „`Boxster`“ ein Modell dieser ist.

2.2.1. Protokolle

Neben der eher formlosen Kommunikation zwischen Agenten, die eher für kleinere Multi-Agentensysteme (< 100 Agenten) geeignet ist, ist gerade bei großen Systemen (≥ 100 Agenten) eine protokollorientierte Kommunikation von Vorteil [Walton, Robertson, 2002]. Nach [Walton, Robertson, 2002] wird es in großen Systemen ohne Agenten-Protokolle zu einer Flut von rational kommunizierenden Agenten kommen. Dieses beruhe darauf, dass solche Agenten zum Kooperieren eine große Anzahl von Aufgaben ausführen müssen, zum Beispiel:

1. Individuelles identifizieren relevanter Agenten aus der gesamten Gruppe.
2. Identifizieren von Agentengruppen mit besonderen Fähigkeiten.
3. Aushandeln von Vereinbahrungen unter Agentengruppen.
4. Behandeln der Möglichkeit von nicht verfügbaren und unzuverlässigen Agenten.
5. Kompensation der asynchronen Natur des Nachrichtenparsing.

Das explizite Festlegen einer Kommunikation durch ein Protokoll, zum Beispiel für eine Verhandlung, stellt dagegen sicher, dass die Nachrichten die

innerhalb dieser Verhandlung ausgetauscht, erwartet werden und formal korrekt sind, wodurch ein Agent in den oben genannten Aufgaben entlastet wird.

Das heißt in einem Protokoll:

- werden Rollen definiert, die den Kommunikationspartnern, die ein Protokoll ausführen, zugeordnet werden können. Dabei ist mit Rolle eine Menge von Agenten gemeint, die unterschiedliche Merkmalen, Interfaces, Dienstbeschreibungen oder Verhaltenweisen erfüllen [Bauer, Müller, Odell, 2001].
- gibt es genau eine initiale Nachricht, die einer Rolle zugeordnet ist.
- die Sequenz der Nachrichten ist wohldefiniert.
- der Syntax der Nachrichten ist eindeutig definiert, insbesondere referenziert jede Nachricht auf einen Sprechakt.
- entlang der Sequenz der Nachrichten ist genau bestimmt, mit welchen Nachrichten eine vorhergegangene Nachricht beantwortet werden kann. Durch die Wahl für eine der möglichen Nachrichten wird ein Pfad innerhalb der Nachrichtensequenz gewählt.
- Kommunikationsprotokolle besitzen einen oder mehrere Endzustände, wie zum Beispiel Zustände für eine erfolgreiche oder fehlerhafte Beendigung des Protokolls.

Abbildung 2.4 zeigt ein einfaches Request-Protokoll, wie es zum Beispiel für die Kommunikation aus Abbildung 2.3 verwendet worden sein könnte. Hier jedoch die Performative der FIPA, „`request`“ für „`ask-one`“ und „`inform`“ für „`tell`“, verwendet. Als Antwortalternativen (alt) auf die Request-Nachricht stehen dem Participant (Gebrauchtwagen-Karl) die Nachrichten mit den Performativen „`inform`“ oder „`failure`“ zur Verfügung.

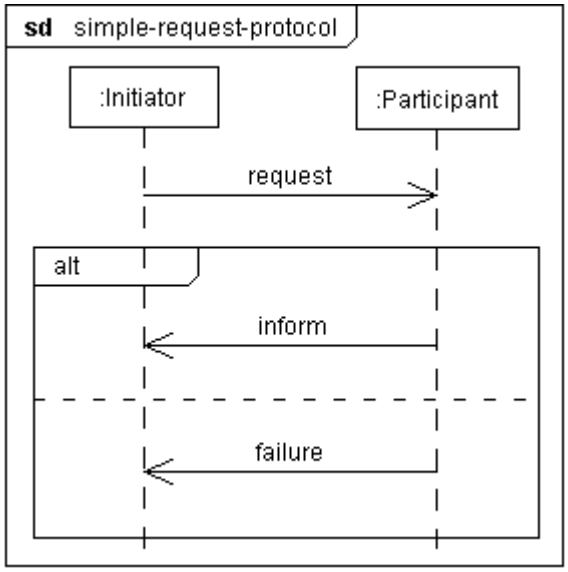


Abbildung 2.4: Einfaches Request-Protokoll[2]

[2] Für die genaue Syntax des Protokolls siehe Kapitel 2.3.1. , Agent UML Sequenz-Diagramme (Seite 21)

2.3. FIPA Agent UML

Um eine breite Akzeptanz für den Einsatz von Agentensystemen zu erhalten bedarf es Methoden, die den Entwicklungsprozess während dessen gesamten Lebenszyklusses begleiten und zugleich eine Verbindung zu Technologien aus der objektorientierten Softwareentwicklung herstellen. Agent UML (AUML) [Odell, Parunak und Bauer 2000] ist ein Ansatz für eine solche Methode, die Agenten und auf Agenten basierende Systeme mit Hilfe von UML-Ausdrücken und agentenorientierten Erweiterungen beschreibt.

Das Aufsetzen auf die Unified Modeling Language (UML) ermöglicht es, Erkenntnisse, die in der objektorientierten Softwareentwicklung gewonnen wurden, auch bei der Entwicklung von Agentensystemen zu verwenden. Zudem wird durch eine vertraute Notation die Akzeptanz von agentenorientierter Entwicklung gefördert. Die FIPA hat daher das Modeling Technical Committee[3] (TC) geschaffen, das sich mit der Ausarbeitung eines Standards für eine AUML-Notation befasst. Aufgaben dieses Komitees betreffen unter anderem die Ausarbeitung von agentenorientierten Klassen-Diagrammen, die Modellierung von sozialen Aspekten, von Goals und Mobilität, die Darstellung von Planverhalten und zeitlichen Bedingungen, sowie die Ausarbeitung von Interaktionsdiagrammen für Agenten.

Für die agentenorientierten Modellierung werden diverse Modellierungskonzepte, die mehr oder weniger Verbreitung in der Agentenwelt gefunden haben, betrachtet, damit FIPA AUML einen soliden und akzeptierte Modellierungssprache wird. Für FIPA AUML werden unter anderem betrachtet:

- UML 2.0 [OMG, 2003]

  Mit UML 2.0 will die Object Managment Group (OMG)[4] eine neue Version ihrer Unified Modeling Language einführen und durch diese die aktuellste Version UML 1.5 ersetzen. UML ist ein Sprachstandard zur Beschreibung von Strukturen und Abläufen in der objektorientierten Programmentwicklung, mit Hilfe verschiedener Diagrammtypen.

- AOR

  Agent-Object-Relationship (AOR) ist ein agentenorientierter Modellierungsansatz von Gerd Wagner. AOR soll aufgrund der agentenorientierten Kategorisierung von verschiedenen Klassen genauere Modelle von Organisationen und von organisatori-

[3] Technical Commitee; sind Gremien der FIPA die die technischen Grundlagen der FIPA schaffen und entweder neue Spezifikationen ausarbeiten oder vorhandene ändern.

[4] OMG; http://www.omg.org

schen Informationssystemen als UML erlauben und kann als eine Erweiterung von UML betrachtet werden [Wagner, 2002].

- PASSI [Cossentino und Potts, 2002]

  Process for Agent Societies Specifications and Implementation (PASSI) ist eine Methode, die schrittweise die Anforderungen des Planens und Entwickelns von Multi-Agenten-Gesellschaften zum Programmcode überführt. Die Methode vereint objektorientierte Konzepte mit Ansätzen der Künstlichen Intelligenz [Cossentino und Sabatucci, 2003].

- MESSAGE [Caire et al., 2001]

  Methodology for Engineering Systems of Software Agents (MESSAGE) ist eine agentenorientierte Softwareentwicklungsmethode, die für die Bedürfnisse der Telekommunikationsindustrie entwickelt wurde. Da sie die grundlegenden Aspekte der Multi-Agentensysteme abdeckt kann sie jedoch auch in anderen Bereichen eingesetzt werden [Cervenka, 2003].

- Tropos (einschließlich i* und GRL) [Mylopoulos, Kolp und Castro, 2001]

  Tropos[5] ist eine agentenorientiere Softwareentwicklungsmethode, die auf den Konzepten Goalbasierenderanforderungen übernommen von i* (i-Stern)[6] und GRL (Goal-oriented Requirements Language)[7] basiert. Der Tropos-Ansatz ist besonders auf BDI-Systeme fokussiert. [Cervenka, 2003b]

- ADELFE [Bernon et al., 2002]

  Atelier pour le Développement de Logiciels à Fonctionnalité Emergente (ADELFE) (Werkzeugsatz zur Entwicklung von Software mit neuer Funktionalität) ist eine Methode, die Multi-Agentensysteme als dynamische Organisation, bestehen aus verschiedenen kooperativen Agenten, betrachtet. Der Hauptunterschied von ADELFE zu anderen agentenorientierten Methoden ist, dass die Anpassungsfähigkeit innerhalb des System mit betrachtet wird, um mit unvorhersehbar auftretenden Ereignissen in der Umgebung des Agenten fertig zu werden, damit die richtige Aufgabe realisiert wird. [Huget, 2003b]

- Gaia

  Gaia ist eine Methode, die speziell auf Analyse und Design von agentenbasierenden Systemen zugeschnitten ist. Der Gaia-

[5] Tropos, http://www.cs.toronto.edu/km/tropos/

[6] i*, http://www.cs.toronto.edu/km/istar/

[7] Goal-oriented Requirements Language, http://www.cs.toronto.edu/km/GRL/

Ansatz versucht, grundlegende Divergenzen zwischen den Konzepten der objektorientierten Entwicklung und der agentenorientierten Sichtweise zu beheben [Wooldridge, Jennings und Kinny, 2000].

- Bric [Ferber, 1999]

  Block-like Representation of Interactive Components (Bric) ist eine Hochsprache für einen modularen Ansatz für Multi-Agentensystemen. Ein Bric-System besteht aus Komponenten die über Kommunikationsverknüpfungen verbunden sind. [O-dell, 2003]

- Styx

  Styx ist eine agentenorientierte Entwicklungsmethode, die Kommunikationskonzepte, Agenten-Agenten-Kommunikation und Agentenverhaltensaktivierung beschreibt, nicht aber die Entwicklung von Teilen, die für die Anwendung eines Systems spezifisch sind [Bush, Cranefield und Purvis, 2001].

- Prometheus [Padgham und Winikoff, 2002a, 2002b]

  Prometheus ist eine Methode zur Entwicklung intelligenter Agentensysteme. Es definiert eine Modellierungssprache die sich generisch zu jeder Multi-Agenten Architektur und Implementierungsumgebung verhält. Prometheus wird meist zur Entwicklung von BDI-Systemen eingesetzt. [Cervenka, 2003c]

- Madkit/Aalaadin [Ferber und Gutknecht, 1998]

  Aalaadin ist ein organisatorisch basiertes Modellierungsschema zur Beschreibung von Multi-Agentensystemen. MADKit ist ein Framework zur Unterstützung dieser Methode. Die organisatorische Perspektive von Aalaadin ermöglicht es den Entwicklern, Probleme wie die Heterogenität von Sprachen (KQML, ACL, FIPA), verschiedene Einsatzgebiete, Architekturen und Sicherheit in klarer und effizienter Weise zu betrachten [Levy, 2003].

- OPM [Dori, 2002]

  Object-Process Methodology (OPM) vereint in einem einzigen Modell sowohl statisch-strukturelle als auch dynamisch-prozedurorientierte Aspekte. OPM unterscheidet sich von UML dahingehend, dass es Entwickler nicht zwingt mehrere verschiedene Diagramme zu benutzen müssen. [Dori et al., 2003]

Das FIPA Modeling TC versucht bei der Entwicklung soweit es sinnvoll ist UML wieder zu verwenden. Dort wo es nicht sinnvoll erscheint, können jedoch auch andere oder neue Methoden benutzen oder entwickeln werden.

2.3.1. Agent UML Sequenz-Diagramme

Im Folgenden werden Sequenz-Diagramme (Protokoll-Diagramme) von AUML betrachtet wie sie in dieser Arbeit verwendet werden. Hierbei wird der „FIPA Modeling: Interaction Diagram“-Working Draft [FIPA, 2003] [Huget, 2003] zugrunde gelegt, der eine Darstellung der Diagramme auf Grundlage von UML 2.0 beschreibt. Der Draft selbst befindet sich in einer frühen Phase und ist noch Änderungen unterworfen, beschreibt aber bereits grundlegende Elemente.

Sequenz-Diagramme werden verwendet um Interaktionsprotokolle zu beschreiben. Sequenz-Diagramme werden in UML definiert als ein Diagramm welches Interaktionen in zeitlicher Reihenfolge darstellt. Insbesondere zeigt es die Teilnahme eines Objekts an der Interaktion und die Folge der Nachrichten die ausgetauscht werden. Im Gegensatz zu Kollaborationsdiagrammen[8] beinhaltet ein Sequenz-Diagramm zeitliche Abfolgen, jedoch keine Objektbeziehungen. Auf Agent UML übertragen bedeutet dies, dass Sequenz-Diagramme die Teilnahme von Agenten an der Interaktion darstellen. Damit sind Sequenz-Diagramme in Multi-Agentensystemen Diagramme, die den Nachrichtenaustausch innerhalb eines Protokolls darstellen.

2.3.1.1. Beschreibung

Sequenz-Diagramme bestehen aus zwei Dimensionen, zum einen wird durch die vertikale Dimension eine zeitliche Reihenfolge, verlaufend von oben nach unten, zum anderen in der horizontalen Dimension verschiedene Rollen oder spezifische Agenten dargestellt. Die Sequenz der Nachrichten wird durch die zeitliche Achse definiert.

Abbildung 2.5 zeigt die grundlegende Notation eines AUML Sequenz-Diagramms. Die einzelnen Rollen oder Agenten werden durch so genannte Lifelines (Lebenslinien) dargestellt. Eine Lifeline besteht aus einem Anschriftelement und der eigentlichen Linie (gestrichelt dargestellt), an der die Nachrichtensequenz notiert wird.

[8] Kollaborationsdiagramm (eng. Collaboration Diagram); Ein Diagramm, welches Objektinteraktion um die Objekte und deren Beziehungen zueinander organisiert. Kollaborations- und Sequenz-Diagramme drücken ähnliche Informationen aus, stellen diese jedoch unterschiedlich dar.

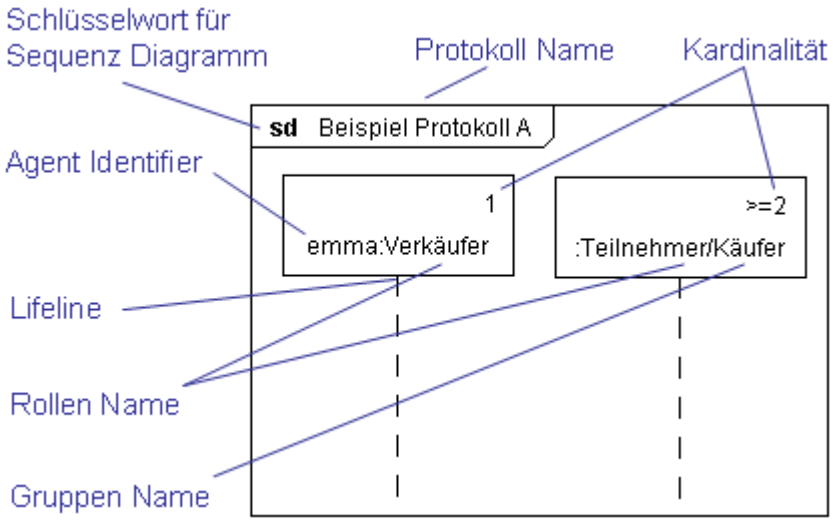


Abbildung 2.5: Agent UML- Notation 1

Im Anschriftelement muss die Rolle, die der Lifeline im Protokoll zugeordnet ist, notiert werden. Des Weiteren ist es möglich, eine Gruppe, zu der die Rolle gehört, zu notieren, sowie einen spezifischen Agenten anzugeben (z.B.: AgentId:Rolle/Gruppe). Das Anschriftelement beinhaltet außerdem die Kardinalität der Rolle – dies ist eine Zahl, eine logische Formel oder Bedingung, welche die Anzahl der Agenten, die der Lebenslinie zugeordnet sind, bestimmt. Die Kardinalität ist bei Angabe eines Agent Identifiers immer Eins. Zu jeder Rolle und jedem Paar aus einer Rolle und einer Gruppe ohne einen Agent Identifier darf nur eine Lifeline im Protokoll notiert werden.

2.3.1.2. Nachrichten

Nachrichten werden als gerichtet Pfeile von der Senderrolle zur Empfängerrolle gezeichnet. Die Eigenschaften einer Nachricht werden durch eine Pfeilspitze dargestellt (siehe Abbildung 2.6 a, b). Asynchrone Nachrichten werden mit einer offenen Pfeilspitze dargestellt, synchrone Nachrichten mit einer ausgefüllten. Wenn z.B. ein Agent eine synchrone Nachricht sendet, kann diese Rolle des Agenten erst wieder empfangen, wenn der Agent eine Antwort auf die Nachricht bekommen hat. In Agentensystemen werden in der Regel asynchrone Nachrichten verwendet [FIPA 2003]. Mit Hilfe von asynchronen Nachrichten können auch überlappende Nachrichten dargestellt werden (siehe Abbildung 2.6d), also Nachrichten die gesendet werden bevor

eine andere Nachricht ankommt. Der kommunikative Akt[9] wird über dem Pfeil bei Nachrichten von einer Rolle zu einer anderen und neben dem Pfeil bei an die gleiche Rolle gerichteten Nachrichten (siehe Abbildung 2.7 a-c) geschrieben. Um die Anzahl der oder den betroffenen Rolleninhaber einer Rolle beim Empfangen einer Nachricht darzustellen, kann eine Kardinalität an der Pfeilspitze angegeben werden, diese kann wie beim Anschriftelement der Rolle eine Zahl, logische Formel oder Bedingung sein. Ebenso kann eine Kardinalität für die Senderseite notiert werden (siehe Abbildung 2.6 c).

Zur Darstellung von Nachrichten, die von Agenten einer Rolle zu Agenten der gleichen Rolle gesandt werden, wird ein reflexiver Pfeil gegen die Zeitachse gezeichnet (siehe Abbildung 2.7). Zum Ausschluss des sendenden Agenten bei asynchronen Nachrichten wird am Beginn des Pfeils eine Diagonale durch den Pfeil gezogen (Abbildung 2.7 a). Zu beachten ist hierbei, dass bei einer synchronen Nachricht keine Diagonale durch den Pfeil gezogen wird, obwohl auch hier der Sender die Nachricht nicht erhält (Abbildung 2.7 c).

[9] Kommunikativer Akt: Aus der Sprechakttheorie [Searle, 1969] stammendes Schlüsselwort in Nachrichten, das einen illokutionären Gehalt hat. Zum Beispiel festgelegt in der Comunicative Act Library der FIPA [FIPA, 2002]

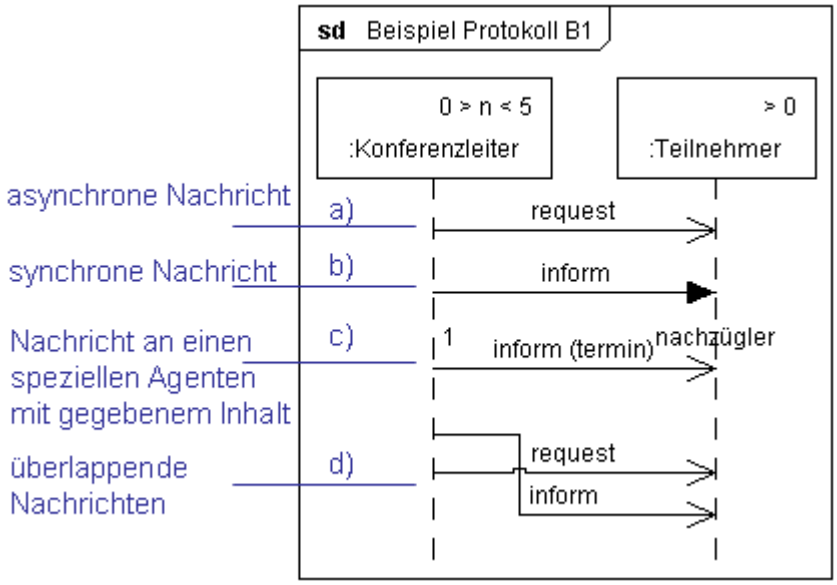


Abbildung 2.6: Agent UML 2.0 Nachrichten Notation 1

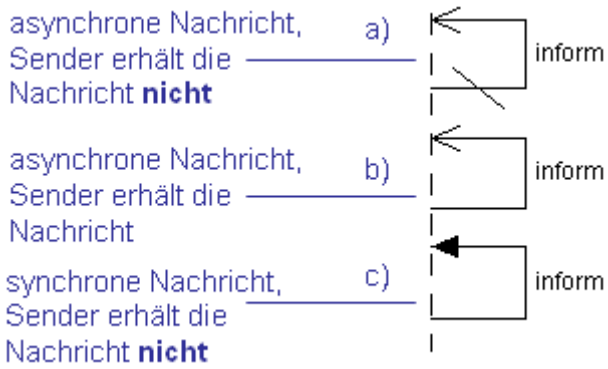


Abbildung 2.7: Agent UML 2.0 Nachrichten Notation 2

2.3.1.3. Kontrollstrukturen

Der Sequenzpfad der Nachrichten, die in einem Protokoll auftreten, kann mit Hilfe von Kontrollstrukturen an bestimmte Situationen angepasst werden. Durch Kontrollstrukturen wird dem Agenten die Möglichkeit gegeben, anhand ihrer Beliefs und Desires alternative Pfade zu wählen. Um Kontrollstrukturen darzustellen, werden die so genannte Combined Fragments aus UML 2.0 verwendet, dies sind Boxen die unterschiedliche Pfade einer Inter-

aktion je nach Operator beinhalten. Tabelle 1 zeigt eine Auflistung aller Operatoren für Pfadkontrollstrukturen und deren Bedeutung, Abbildung 2.8 (Seite 27) zeigt deren Verwendung in einem Protokoll.

Tabelle 1: Operatoren für Pfadkontrollstrukturen

Operator	Kurzform	Bedeutung
Alternative	alt	Bestimmt mehrere Pfade, von denen der Agent einen auswählen muss, um dem Protokoll zu folgen. Für die einzelnen Pfade können Guards/Constraints (Bedingungen) angegeben sein, d.h. der Agent wählt den Pfad aus, dessen Bedingung erfüllt ist. Falls keiner der Guards erfüllt ist, kann keine Alternative ausgewählt werden und das Protokoll kann nicht weiter ausgeführt werden. Zur Vermeidung kann eine Else-Alternative benutzt werden.
Option	opt	Bestimmt einen Pfad, der optional ausgeführt werden kann, beziehungsweise auszuführen ist, wenn seine Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, wird der Pfad nach der Option fortgeführt. Optionen können als Oder interpretiert werden.
Break	break	Der Break Operator definiert nach UML 2.0 [UML 2003] ein Abbruch-Szenario, welches anstelle des übrigen Szenarios im übergeordneten Combined Fragment ausgeführt wird. Zum Beispiel kann mit einem Break festgelegt werden, was zu geschehen hat, wenn eine Nachricht nicht verstanden wurde. Das Break Combined Fragment ist global im gesamten übergeordneten Combined Fragment gültig.
Parallel	par	Bestimmt mehrere Pfade, die parallel ausgeführt werden können. Die Ereignisse in den verschiedenen Pfaden können sich in jeder denkbaren Weise überlappen.

Weak Sequencing	seq	Gibt an, dass alle Nachrichten innerhalb des Combined Fragments in Bezug zu einer Lebenslinie geordnet sind, aber keine Angaben zur Ordnung dieser in Bezug zu Nachrichten für andere Lebenslinien bestehen. Weak Sequencing stellt damit nur die Sequenz für eine Lebenslinie sicher.
Strict Sequencing	strict	Erweitert das Weak-Sequencing um die Zusicherung, dass die Reihenfolge der Nachrichten innerhalb des Combined Fragments entlang der Zeitachse geordnet ist.
Negative	neg	Gibt eine Menge von Nachrichten an, die als ungültig innerhalb des Protokolls zu betrachten sind.
Critical Region	critical	Definiert eine Region im Protokoll deren Nachrichtensequenz nicht von anderen Nachrichten überlappt werden kann, das heißt die Nachrichtensequenz ist atomar zum übergeordneten Combined Fragment. Dies gilt selbst, wenn es sich beim übergeordneten Fragment um einen Parallel-Operator handelt.
Ignore	ignore	Gibt eine Menge von Nachrichten an, die nicht von Interesse sind und deren auftreten daher ignoriert werden soll. Zum Beispiel in einem Auktionsprotokoll, in dem der Auktionator nach Auktionsende sich bereits in der Zahlungsabwicklung befindet und trotzdem noch Angebote erhält. Mit Hilfe von Ignore können solche Angebote ignoriert werden. Die zu ignorierenden Nachrichten sind nach dem Operator zu notieren. Zum Beispiel: *ignore {x, y}*
Consider	consider	Consider ist das Gegenteil zu Ignore und gibt eine Menge von Nachrichten an, die zu beachten sind. Die zu berücksichtigenden Nachrichten sind nach dem Operator

		zu notieren. Zum Beispiel: *consider {x, y}*
Assertion	assert	Gibt an, dass die enthaltene Nachrichtensequenz die einzig gültige im gegenwärtigen Zustand im Protokoll ist.
Loop	loop	Der Loop-Operator gibt eine Folge von Nachrichten an, die mehrfach wiederholt werden kann. Eine Wiederholungsbedingung, die hinter dem Operator in runden Klammern notiert wird, bestimmt dabei, wie häufig die Schleife zu durchlaufen ist. Zum Beispiel: *loop (condition)*

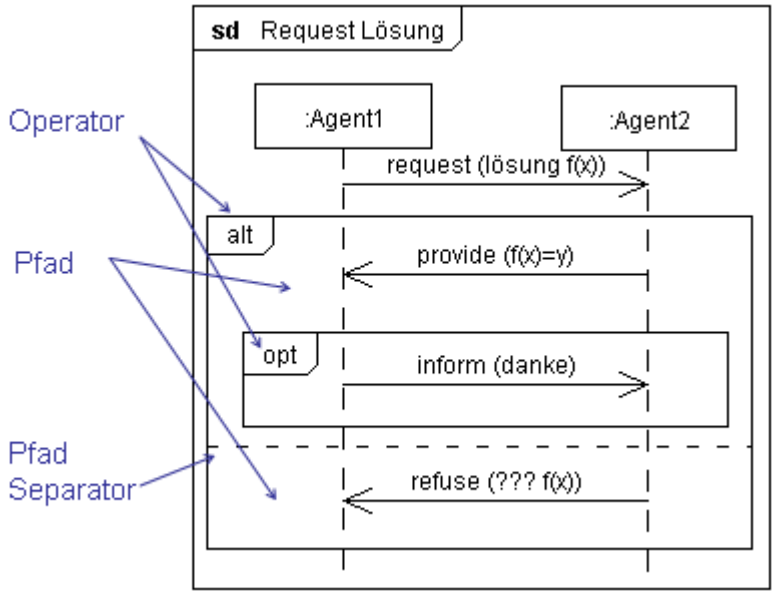


Abbildung 2.8: Pfadkontrollstrukturen

2.3.1.4. Weitere Kontrollstrukturen

Agent UML definiert weitere Kontrollstrukturen für Pfade, zum einen die Continuation, zum anderen einen Stopp-Operator (siehe Abbildung 2.9). Eine Continuation definiert einen benannten Sprungpunkt (outgoing Continuation) und Einsprungpunkt (incoming Continuation). Durch eine Continuation lässt sich zum Beispiel eine Iterationsschleife in einem Protokoll verwirklichen, die nicht mit der Loop-Anweisung durchführbar ist. Continuati-

ons werden in gerundeten Rechtecken dargestellt. Der Sprungpunkt wird durch ein linksseitig angeordnetes gefülltes Dreieck gefolgt vom Namen der Continuation dargestellt. Beim Einsprungpunkt befindet sich das Dreieck rechts vom Namen der Continuation. Zu jedem Sprungpunkt gibt es genau einen zugehörigen Einsprungpunkt.

Mit Hilfe des Stopp-Operators kann in einem Protokoll ausgedrückt werden, dass ein spezifischer Agent mit Auftreten des Stopp-Operators nicht mehr mit der Rolle, an der der Operator notiert ist, an dem Protokoll teilnimmt. Ist der Agent mit mehreren Rollen im Protokoll involviert, nimmt er weiterhin mit den übrigen Rollen teil. Der Stopp-Operator kann auch dazu verwendet werden, das Ende eines Pfades anzuzeigen, dieses erleichtert vor allem in großen Protokollen die Übersicht. Der Stopp-Operator wird durch ein **X** auf der Linie der Lifeline dargestellt.

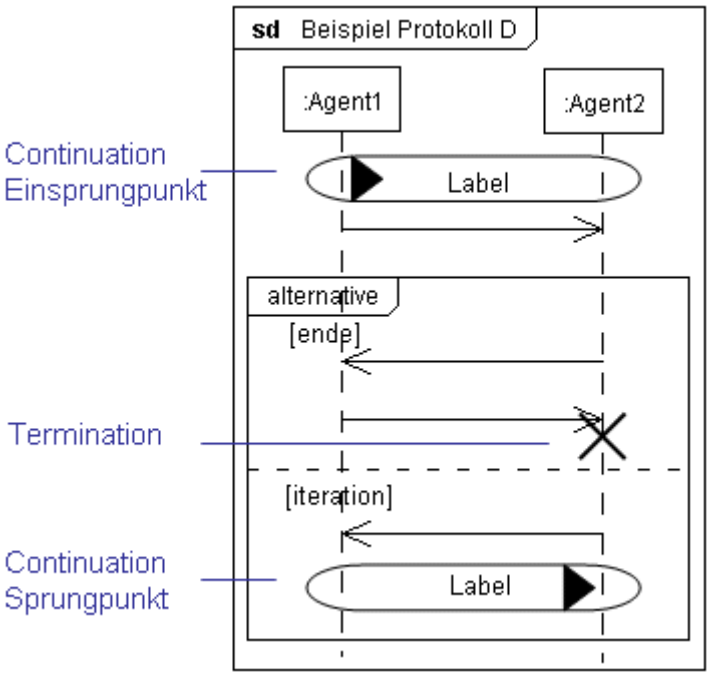


Abbildung 2.9: Agent UML Pfadkontrollstrukturen

2.3.1.5. Nebenbedingungen

In Abbildung 2.10 sind Nebenbedingungen (Constraints) angegeben, die die Auswahl einer der Pfadalternativen bestimmen. Mit Constraints kann bestimmt werden, wie Nachrichten oder Pfade in einem Protokoll verwendet

werden. Nebenbedingungen können entweder informell oder in einer formalen Sprache, zum Beispiel mit der Object Constraint Language (OCL) deklariert werden. Es werden, wie bei Nachrichten auch, bei Constraints zwei verschiede Arten angeboten, zum einen blockierende (blocking), zum anderen nicht blockierende (non-blocking). Blockierende Constraints können verwendet werden, um sicherzustellen, dass bestimmte Bedingungen erfüllt sind, bevor das Protokoll fortgesetzt wird. Nicht blockierende Constraints werden in der Regel zur Pfadauswahl in Alternativen verwendet. Blockierende Constraints werden in runden Klammern am Element, zu dem sie gehören, geschrieben, nicht blockierende Constraints werden in eckigen Klammern notiert.

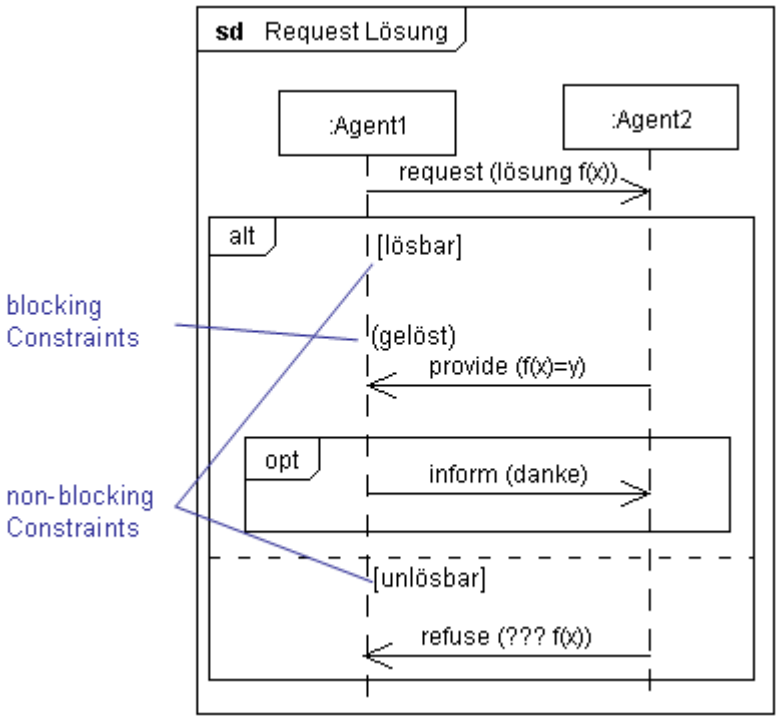


Abbildung 2.10: Agent UML Constraints Notation

2.3.1.6. Zeitliche Nebenbedingungen

Mit Hilfe von zeitlichen Nebenbedingungen kann in Sequenz-Diagrammen ausgedrückt werden, innerhalb welchen Zeitraums eine Nachricht erwartet wird. Dieses kann eine relative Angabe, zum Beispiel relativ zum Absendezeitpunkt einer vorhergegangenen Nachricht, oder eine absolute Zeitangabe sein. Abbildung 2.11 zeigt zwei verschiedene Notationsarten, die alternativ

zueinander benutzt werden können. Die Nachricht, die der zeitlichen Bedingung zugehört, muss zwischen deren Unter- und Obergrenze ankommen. Die Grenzen können entweder natürliche Zahlen (etwa {0..7}[10]) oder basierend auf einem Beobachtungszeitpunkt (etwa {0..t+3}[11]) sein. Der Beobachtungszeitpunkt muss zuvor im Diagramm definiert worden sein. Für absolute Zeitangaben ist kein bestimmtes Format vorgeschrieben, sodass Zeitangaben wie 14.6.2004 15:30 möglich sind.

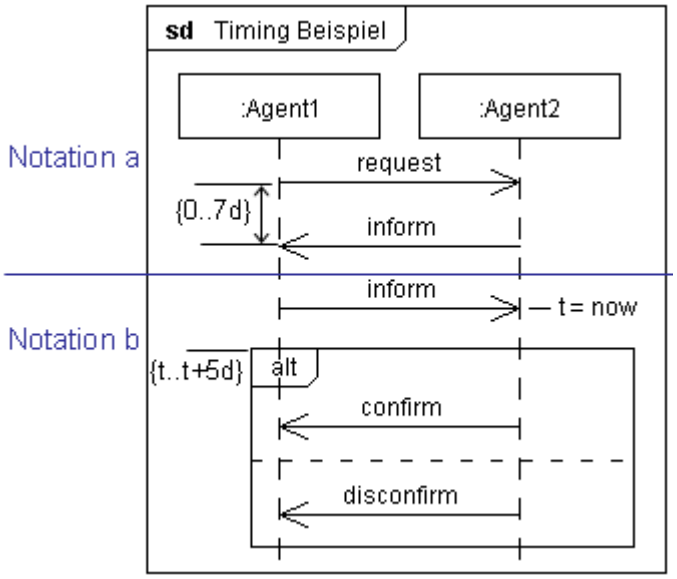


Abbildung 2.11: Notation zeitlicher Nebenbedingungen

2.3.1.7. Dynamische Rollen

Während der Protokollausführung kann es vorkommen, dass ein Rolleninhaber eine weitere oder eine andere Rolle einnehmen kann. Ein Beispiel für einen Rollenwechsel ist ein Backup-Protokoll, bei dem das aktive System das Ersatzsystem darüber informiert, dass es in die Wartungsphase übergeht, womit das Ersatzsystem die Rolle des aktiven Systems übernimmt. Rollenwechsel und das Hinzufügen einer Rolle werden durch gerichtete gestrichel-

[10] {0..7}: die Nachricht muss zwischen dem gegenwärtigen Zeitpunkt und plus sieben Zeiteinheiten ankommen. Der gegenwärtige Zeitpunkt ist der Zeitpunkt an dem der entsprechende Protokollzustand erreicht wird.

[11] {0..t+3}: die Nachricht muss zwischen dem gegenwärtigen Zeitpunkt und der Beobachtungszeit plus drei Zeiteinheiten ankommen

te Pfeile von der Ausgangs- zur Zielrolle dargestellt, mit der Anschrift „`change role`“ bzw. „`add role`“.

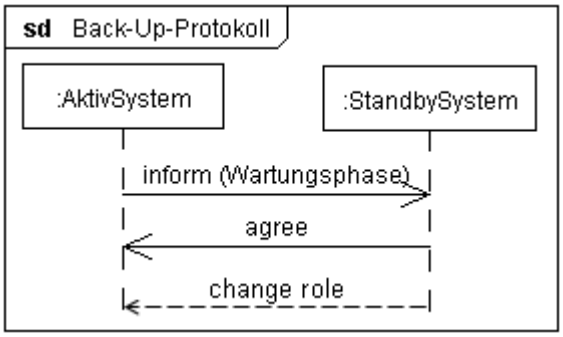


Abbildung 2.12: Dynamische Rollen in Protokollen

2.3.1.8. Protokollkombination

Vielfältig kommen in Protokollen Sequenzabschnitte vor, die bereits als einzelnes Protokoll vorkommen, so zum Beispiel ein Bezahlvorgang. Ein solches Protokoll kann in (Agent) UML 2 über ein Referenz-Element wieder verwendet werden. Das Referenz-Element wird in einer Combined Fragment Box mit dem Operator „`ref`“ dargestellt. Die Nachrichten eines solchen überlappenden Protokolls werden so gesendet, als gehörten sie zum übergeordneten Protokoll – das heißt zum Beispiel in Bezug auf Abbildung 2.13 für eine ACL-Nachricht, dass der Parameter „`protocol`“ dieser Nachricht nicht „Bezahlprotokoll“ sondern „Kaufprotokoll“ lautet. Das referenzierte Protokoll wird also so betrachtet, als würde dessen Inhalt sich im referenzierenden Protokoll befinden. Beim Referenzieren muss eventuell durch einen Kommentar angegeben werden, welche Rolle des referenzierenden Protokolls mit welcher im referenzierten Protokoll übereinstimmt. Falls ein Protokoll sich selbst referenziert, muss sichergestellt sein, dass es nicht in eine Endlosschleife gerät, es sei denn, dies ist gewollt.

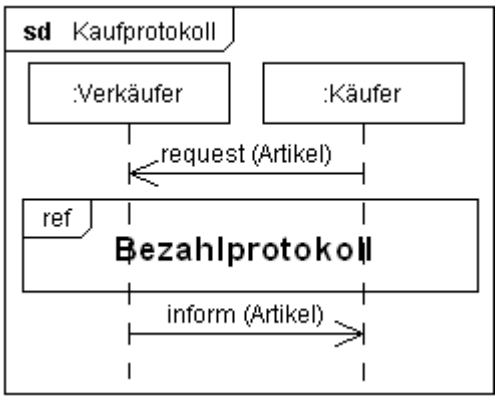


Abbildung 2.13: Überlappendes Protokoll

3. Implementierung von Protokollen

Ein verbreitetes Vorgehen bei der Entwicklung von protokollorientierten Agenten ist, während des Entwicklungsprozesses diese Protokolle in einem Dokument zu spezifizieren und dann die Implementierung der Agenten anhand dieses Dokumentes durchzuführen. Dabei wird die Spezifikation für die spätere Kommunikation nicht mehr benötigt – in dem Sinne, dass man davon ausgeht, dass sie valide implementiert wurde.

Werden die Protokoll-Spezifikationen frei zugänglich gemacht und basieren diese auf einer einheitlichen Spezifikationssprache wie zum Beispiel AUML, können zu jedem Zeitpunkt auch für Agenten anderer Agentensysteme diese Protokolle implementiert werden. Wobei bei jeder neuen Implementierung eine aufwendige Validierung des Protokollverhaltens stattfinden muss. Diese Validierung kann dabei mehr Zeit in Anspruch nehmen als die Überprüfung des individuellen Verhaltens eines Agenten bei der Abarbeitung des Protokolls.

3.1. Verwandte Arbeiten

Es gibt viele verschiedene Ansätze, um die Entwicklung von protokollorientierten Agentensystemen zu unterstützen. Diese Ansätze lassen sich in verschiedene Bereiche aufteilen: Code-Generatoren, Tools zur Unterstützung protokollorientierter Kommunikation, Tools zur Ausführung protokollspezifischen Codes und plattformbasierende Verifikation.

3.1.1. Code-Generatoren

Relativ häufig anzutreffende Entwicklungswerkzeuge für protokollorientierte Kommunikation in Agentensystemen sind Werkzeuge, die eine graphische Protokollrepräsentation in Code-Templates überführen können. Ziel solcher Tools ist es, direkt aus dem Design eines Protokolls ein Codeskelett zur Implementierung zu erhalten.

Ein Tool, das diesen Ansatz verfolgt, ist das Tool von Martin Dinkloh und Jens Nimis [Dinkloh und Nimis, 2003]. Vor der Benutzung des Tools wird zunächst ein AUML-Protokoll entworfen. Dieses AUML-Protokoll wird dann im Tool zu einem Modell einer Finite State Machine (FSM) überführt. Die Modellierung von AUML und die Überführung in eine FSM ist dabei eine manuelle Aufgabe. Ein direkter Entwurf von AUML innerhalb des Tools ist jedoch angedacht. Zur Speicherung und internen Repräsentation der FSM wurde Conversation Policy XML (cpXML) [Hanson et al., 2002] genommen. Die Repräsentation als FSM wurde dabei gewählt, weil die verwendete Agentenplattform JADE [Bellifemine et al., 2000] ein so genanntes FSMBehavior bereitstellt. Behavior dienen in JADE zur Abarbeitung von Aufgaben oder Teilaufgaben eines Agenten. Ein FSMBehavior ist ein Zusammengesetztes Behavior, dass seine Subbehavior gemäß einer vom Nutzer definierten FSM ausführt. Aus der im Tool modellierten FSM, werden diese FSMBehavior für jede Rolle des Protokolls generiert. Die generierten Behavior müssen anschließend nur noch mit anwendungsspezifischen Code vervollständigt werden. Die Architektur des Tools ist dabei so gestaltet, dass die Komponente zum Generieren des JADE-Codes gegen Komponenten zum Generieren von Code für andere Plattformen ausgetauscht werden kann.

Ein vergleichbares Tool für ein spezielles Agentensystem beschreibt Lawrence Cabac [Cabac, 2003]. Mit seinem Tool ist es möglich AUML-Protokolle zu zeichnen und daraus Petri-Netz-Skelette für das Agentensystem Mulan [Köhler, Moldt, Rölke, 2001] zu generieren.

Weitere Tools sind das bereits erwähnte PASSI [Cossentino und Potts, 2002] und SmartAgent [Griss et al., 2002] (eine Erweiterung zu JADE), aus denen Quellcode für JADE erstellt werden kann. Wobei die Quellcodegenerierung

von PASSI und SmartAgent auch die nicht protokollorientierte Kommunikation bedient.

Zu nennen sind auch Zeus [Zeus] von der British Telekom, ADK [ADK] von Tryllian und agentTool [AFIT] vom AFIT Artificial Intelligence Laboratory der U.S. Air Force, diese Tools haben gemeinsam, dass sie Teil eines Gesamtkonzepts einer Entwicklungsumgebung für Agentensysteme sind.

Die grundlegenden Eigenschaften aller Tools sind dabei:

- graphischer Protokollentwurf
- Generierung von protokollkonformen Code-Templates aus dem Designtool heraus.

Der Nachteil der genannten Tools ist allerdings, dass nach dem Generieren der Code-Templates keine Verbindung zum Protokollentwurf besteht. Ohne diese Verbindung ist es möglich, dass der Entwickler protokollkonformer Templates Fehler bei der Implementierung machen kann. Dies ist insbesondere dann problematisch, wenn alle Kommunikationspartner eines Protokolls vom selben Entwickler implementiert werden, da seine eigenen Implementierungen zumeist zueinander kompatibel sein werden. Diese Tools generieren daher eher einen Rahmen, der vorschreibt welche Nachrichten empfangen und gesendet werden sollen, jedoch kann dieses jederzeit umgangen werden. Des Weiteren muss bei Fehlern oder Änderungen, die sich im Design des Protokolls ergeben haben, eine erneute Generierung der Templates stattfinden, wodurch der geschriebene Code übertragen oder neu erstellt werden muss. Eine automatische Übertragung wird zumeist nicht beherrscht. Ein weiterer Nachteil ist, dass die Formate, in denen diese Tools ihren Protokollentwurf speichern, nicht als ein Austauschformat ausgelegt sind und daher nur von den Tools selber verstanden werden.

3.1.2. Tools zur Unterstützung protokollorientierter Kommunikation

Das Ziel dieser Tools ist es, dass ein Agent zur Laufzeit überprüfen kann, ob dass, was er innerhalb eines Protokolls senden möchte, protokollkonform ist. Dabei benutzt der Agent nur dieses Tool, ohne dass er seine Autonomie verliert.

Ein solches Werkzeug wird von Freire und Botelho [Freire und Botelho, 2002] beschrieben. Aus Protokollen, in einer AUML nahen XML Repräsentation werden dabei Java-Objekte erzeugt. Agenten sollen diese interne Protokoll-Repräsentation nutzen, um damit ihre Entscheidungsprozesse zu stützen und so eine protokollkonforme Kommunikation zu erreichen. Dabei

kann das Tool den Agenten darüber informieren, was im aktuellen Protokollzustand getan werden kann. Der Agent informiert im Gegenzug das Tool über seine Entscheidung. Das Tool übersetzt hierfür den relevanten Teil des Protokolls in Produktionsregeln, die die Korrektheit sicherstellen sollen. Bei dieser Methode arbeitet der Agent autonom in seinen Entscheidungen und muss noch selbst Sorge tragen, dass seine Entscheidungen richtig sind.

Ein weiteres Tool dieser Art wird von Yolum und Singh [Yolum und Singh, 2002] beschrieben. Bei diesem Ansatz werden Aktionen in Protokollen als Verpflichtungen betrachten, die der Agent plant zu erfüllen. Protokolle werden dabei durch Prädikate des Event Calculus [Kowalski und Sergot, 1986] spezifiziert. Aus den Protokollspezifikationen werden anschließend die möglichen Pfade berechnet, die im Protokoll durchlaufen werden können. Mit Hilfe dieser Pfade kann der Agent zur Laufzeit feststellen, welche Aktionen im gegenwärtigen Zustand des Protokolls geeignet sind.

Die Vorteile dieses Werkzeugs sind:

- Kontrolle der Protokollkonformität zur Laufzeit.
- Autonomie des Agenten bleibt erhalten.
- Protokolle können zur Laufzeit hinzugefügt werden.

Der Nachteil dieser Tools ist, dass sie in der Entwicklung des Agentenverhaltens keine Erleichterung bringen. Der Protokollfluss muss noch immer nachgebildet werden und an den Sende- und Empfangspunkten des Protokolls werden diese Tools konsultiert.

3.1.3. Tools zur Ausführung protokollspezifischen Codes

Tools dieser Art sind kaum verbreitet. Bei diesen Tools wird kein Code-Skelett aus dem Protokoll generiert, sondern das Protokoll selbst als eine Art Code-Skelett genommen.

Ein solches Tool wird von Ehrler und Cranefield [Ehrler und Cranefield, 2004][Ehrler, 2003] in ihrer zeitgleich zu dieser stattfindenden Arbeit beschrieben. Mit ihrem Plug-in for Agent UML Linking (PAUL) wird ein System beschrieben, dass es Entwicklern erlaubt, Agentenkonversationen durch Linken von anwendungsspezifischem Code auf AUML Sequenz-Diagramme zu kontrollieren. Protokolle für PAUL werden in einen eigenem XML-Metamodel Entwurf für AUML gespeichert. Die Punkte einer Rolle in einem Protokoll, an denen Nachrichten eingehen und versandt werden, werden als Execution Occurrence bezeichnet. Den einzelnen ein- und ausgehenden Nachrichten können Ausdrücke in der Objekt Constraint Language (OCL)[12]

zugeordnet werden, die als Constraints bezeichnet werden. Beim Eingang einer Nachricht wird deren Inhalt verarbeitet und gemäß der Constraints der eingehenden Nachricht werden deren Elemente als Parameter für den anwendungsspezifischen Code, der der Execution Occurrence zugeordnet ist, benutzt. Der anwendungsspezifische Code wird benutzt, um den Inhalt der Nachricht zu erzeugen. Aus dem Rückgabewert dieses Codes wird mit Hilfe der Constraints, die der ausgehenden Nachricht im Protokoll zugeordnet sind, von PAUL eine Antwort erzeugt, die dann gesendet werden kann. Der Code, den der Entwickler schreibt, besitzt dabei Variablen, mit denen Operationen der OCL-Ausdrücke auf Methoden des Codes abgebildet werden. Dabei wird eine Klasse mit Code genau einer Lifeline im Protokoll zugeordnet. Damit der Agent die Nachrichten empfangen kann, interpretiert PAUL das Interaktionsprotokoll und erzeugt, aktiviert oder deaktiviert Nachrichtenfilter, die den Agenten auf die jeweiligen Nachrichten reagieren lassen. Dadurch erlaubt PAUL dem Entwickler, Agenten auf Konversationsebene statt auf Nachrichtenebene zu entwickeln.

Vorteil dieses Systems:

- schnelle Implementierung von Protokollen.
- die Korrektheit des Protokolls wird von der Protokoll-Ausführungsumgebung überwacht.

Der Nachteil von PAUL ist, dass zur AUML-Spezifikation die OCL-Constraints gehören und so eine Einschränkung bei dem Entwurf eines Protokolls entsteht. Dieser Nachteil besteht darin, dass die Klassensignatur dadurch fest vorgeschrieben wird. Des Weiteren wird dadurch, dass eine Klasse genau für eine Lifeline in einem Protokoll bestimmt ist, die Wiederverwendbarkeit der einzelnen Methoden für andere Protokolle erschwert.

3.1.4. Plattformbasierende Verifikation

In heterogenen Multi-Agentensystemen mit einer Vielzahl von Agenten ist es nicht realistisch anzunehmen, dass Agenten immer so handeln werden, dass ihre Aktionen mit den Interaktionsprotokollen übereinstimmen. In diesen Systemen ist es (eventuell zusätzlich zu anderen Tools) notwendig die Kommunikation zwischen den Agenten mit Hilfe von Tools auf Plattformebene zu beobachten und deren Übereinstimmung mit den Protokollen zu verifizieren.

Ein Tool, das dieses ermöglicht ist SOCS-SI [Alberti et al., 2004]. Protokolle in SOCS-SI werden durch Social Integrity Constraints (ic_S) [Alberti et al., 2003] spezifiziert, ein auf Logik basierender Formalismus. Durch ic_S-Terme kann die Übereinstimmung des Agentenverhaltens bezüglich der Spezifikation verifiziert werden. SOCS-SI ist dabei eine Java-Prolog-

Implementierung, mit der zur Laufzeit die Verifikation von Protokollen, in Zusammenspiel mit einer Agentenplattform durchgeführt werden kann. Grundplattform ist dabei PROSOCS [Stathis et al. 2004], aber auch andere Plattformen können, durch Implementierung der entsprechenden Interfaces, SOCS-SI nutzen. Nachrichten in SOCS-SI werden als Events betrachtet und von einem Event Recorder von der Agentenplattform geholt. Vom Event Recorder werden die Nachrichten in einen History Manager übertragen, sodass Nachrichten im Kontext ihres Konversationsverlaufs gegen ihr Protokoll validiert werden können. Über eine graphische Oberfläche kann der Entwickler die empfangenen Nachrichten, die aktuell erfassten Agenten sowie den gegenwärtigen Stand des Prüfverfahrens beobachten. Die prüfbaren Protokolle werden dabei in das Tool geladen.

Der Nutzen einer plattformbasierenden Verifikation ist ein Geschwindigkeitsvorteil. Der Geschwindigkeitsvorteil entsteht dadurch, dass man in einem solchen Tool direkt erkennen kann, welcher Agent eine Nachricht nicht protokollkonform verschickt hat und dies auch für eine große Anzahl von Agenten. Zudem können durch eine Verifikation über die Plattform gleichzeitig mehrere fehlerhafte Agenten ausfindig gemacht werden. Eine Verifikation auf Agentenebene muss für alle Agenten eines Systems stattfinden und ist von daher wesentlich aufwändiger. Ein spezielles Problem beim SOCS-SI-Tool ist, dass eine eigenständige Protokollspezifikation für das Tool benutzt wird, welches um die Fehlerzahl möglichst gering zu halten, auch bei den Agenten verwendet werden müsste, was im Allgemeinen allerdings nicht anzunehmen ist.

3.1.5. Anforderungen an eine Protokollunterstützung

Aus den genannten Tools und ihren Eigenschaften, werden an dieser Stelle Anforderungen aufgestellt, die von der zu realisierenden Protokollunterstützung erfüllt werden sollten.

a) Das Format der Protokollspezifikation sollte geeignet sein, auch von anderen Systemen benutzt zu werden, das heißt es sollte ein Austauschformat sein.

b) Die Protokollspezifikation sollte vom Agenten direkt verwendet werden können (Das Protokoll ist selber das Template).

c) Die Validität der Nachrichten (eingehender wie ausgehender) sollte bereits innerhalb des Agenten zu Laufzeit geprüft werden können.

d) Der Code für eine Rolle, sollte so gestaltet sein, dass er nicht nur für eine Rolle in einem Protokoll verwendet werden kann, sondern auf Ebene der Nachrichten für verschiedene Rollen und Protokolle Wiederverwendung findet.

e) Die Zuweisung von Code zum Protokoll sollte außerhalb des Protokolls erfolgen und schnelle Anpassungen ermöglichen.

f) Es sollten verschiedene Verhaltensvarianten für ein und dasselbe Protokoll spezifiziert werden können.

Daraus ergibt sich zusammenfassend folgender Ansatz: Um für Agenten eine Kommunikation nach einem bestimmten Protokoll zu implementieren (zum Beispiel einem frei zugänglichen), beschränkt sich die Implementierungsarbeit nur auf das individuelle Verhalten des Agenten zur Abarbeitung des Protokolls. Dieses kann man dadurch erreichen, dass dem Agenten ein Interpreter für Protokolle zur Seite gestellt wird. Ein solcher Interpreter garantiert dann die korrekte Abarbeitung der gewünschten Protokolle. Die Entwickler des Agenten müssen dann nur noch den Code, der das individuelle Verhalten des Agenten beschreibt, mit den Ereignissen im Protokoll verknüpfen und brauchen sich somit bei der Validation nur noch auf diesen Punkt beschränken. Dieses Vorgehen reduziert die Fehlerzahl bei der Interpretation der Spezifikation und beschleunigt zugleich die Entwicklungsphase.

Bei der Zuordnung von Code zu einzelnen Ereignissen eines Protokolls, das heißt Code, der eingehende Nachrichten behandelt und Antworten auf diese erzeugt, ergeben sich zwangsläufig kleine Codefragmente, die auf die speziellen Situationen beim Empfangen und Senden abgestimmt sind. Da es häufig vorkommt, dass ähnliche Empfangssituationen in verschiedenen domänenspezifischen Implementierungen eines Protokolls und verschiedenen Protokollen wieder zu finden sind, ist es vorteilhaft, in diese Codefragmente nicht allzu viel Informationen über die zu empfangenden und zu sendenden Nachrichten einzukodieren, um sie damit vom Protokollfluss zu trennen und in anderen Implementierungen wiederverwenden zu können.

Das daraus resultierende Werkzeug ist somit ein Tool zur Ausführung protokollspezifischen Codes.

3.2. Protokoll-Ausführungsumgebung

In diesem Abschnitt wird ein System beschrieben, das die im vorherigen Abschnitt aufgestellten Eigenschaften erfüllt. Es wird auf die benötigten Komponenten eingegangen und die generellen Funktionen einer Protokoll-Ausführungsumgebung vorgestellt.

3.2.1. Grundlegende Voraussetzungen

Im Folgendem wird davon ausgegangen, dass ein Format zur Beschreibung von Protokollen zur Verfügung steht und dass Protokolle in diesem Format vorliegen. Des Weiteren wird davon ausgegangen, dass für die Protokolle jeweils ein Dokument zur Verfügung steht, in dem beschrieben ist, welcher Code welchen eingehenden Nachrichten im Protokoll zugeordnet ist. Ein solches Dokument wird als Mapping bezeichnet und spezifiziert das individuelle Verhalten eines Agenten bezüglich eines Protokolls. Der Code, der das individuelle Verhalten beschreibt, wird als Handler bezeichnet. Die Ausführungsumgebung selbst ist eine Komponente eines Agenten, an die dieser Nachrichten protokollorientierter Kommunikation direkt weiterleitet.

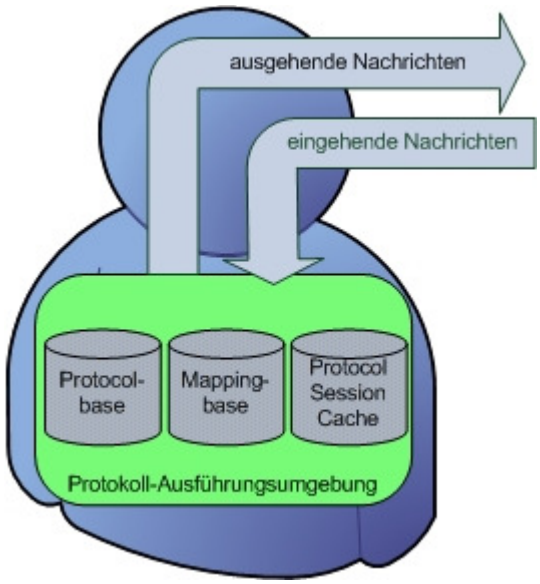


Abbildung 3.1: Protokoll-Ausführungsumgebung als Komponente eines Agenten.

Eine weitere wichtige Voraussetzung für eine funktionierende Ausführungsumgebung sind Conversation-Identifier. Ein Conversation-Identifier dient dem Agenten und in diesen Fall der Ausführungsumgebung zur Verwaltung der Kommunikationsaktivitäten. Conversation-Identifier ist ein Parameter

der ausgetauschten ACL-Nachrichten und steht im Zusammenhang mit dem Protokollparameter einer ACL-Nachricht, er sollte vom Initiator des Protokolls zugewiesen werden. Im Falle der Ausführungsumgebung kann diese auch die Aufgabe der Generierung von Conversation-Identifiern übernehmen. Für alle Antworten auf die initiale Nachricht innerhalb desselben Interaktionsprotokolls ist es für die Ausführungsumgebung wichtig, dass in den Antworten derselbe Conversation-Identifier benutzt wird. Des Weiteren ist es erforderlich, global eindeutige Werte für die Conversation-Identifier zu verwenden, um den Kommunikationspartnern, aber auch der Ausführungsumgebung zu ermöglichen, verschiedene nebenläufige Konversationen zu unterscheiden.

3.2.2. Handler

Handler beschreiben das individuelle Verhalten eines Agenten bezüglich des Empfangs einer Nachricht in einem Protokoll. Ein Handler ist Programmcode; genauer gesagt, eine Methode, die beim Nachrichtenempfang aufgerufen wird. In diesen Programmcode werden die Nachrichten verarbeitet, Antwortnachrichten erzeugt und an die Ausführungsumgebung weitergereicht, sowie interne Entscheidungsvorgänge gespeichert, die für den nächsten Handler von Interesse sein könnten.

Diese Verwendung von Handlern unterscheidet sich bis jetzt nur wenig von Plänen in Agentensystemen und ist daher noch sehr unflexibel, deswegen ist es sinnvoll, parametrisierbare Handler zu benutzen. Parameter können hierbei einfache Schlüssel-Wert-Paare sein, die den Programmfluss beeinflussen. Ebenso sind Sprachausdrücke wie ein OCL-Ausdruck[12] denkbar um eine Einschränkung für zum Beispiel für das Performativ zu erhalten, oder ein Script das die Verarbeitung übernimmt und somit der Handler nur noch als Script-Interpreter dient. Parameter werden dabei zusammen mit dem Handler im Mapping-Dokument angegeben.

Da im Handler bei der Nachrichtenverarbeitung jedoch verschiedene Entscheidungen gefällt und mehrere Nachrichten generiert werden können, kann die Flexibilisierung der Handler auch an der Stelle der Antworten ansetzten. Dieses könnte dann so aussehen, dass eine abstrakte Nachricht als Reaktion auf eine Nachricht erzeugt wird. Hierbei ist ein Handler ein allgemeiner Verarbeitungsprozess, der in verschiedenen Protokollen verwendet werden kann. Er dient der Vorverarbeitung und erzeugt ein oder mehrere Ergebnisse mit zu dazugehörigen Beschreibungen, die ausreichend spezifisch sind, um aus ihnen eine Antwort zu generieren. Eine abstrakte Nachricht ist somit ein

[12] Object Constraint Language (OCL) ist eine formale Sprache zur Beschreibung von Einschränkungen; Teil von UML

Ergebnis dieses Handlers zusammen mit dessen Beschreibung. Um aus dieser abstrakten Nachricht eine konkrete Nachricht zu erzeugen, werden im Mapping für jede Nachricht, die ein solcher Handler erzeugen kann, Reply-Filter angegeben. Eine abstrakte Nachricht, die von einem Handler an die Ausführungsumgebung weitergereicht wird, wird dann von dieser an die Reply-Filter übergeben und das Ergebnis des ersten Filters, der eine Nachricht erzeugt, wird dann als Antwortnachricht verschickt.

Ein Reply-Filter ist somit ein Verarbeitungsprozess, dessen Aufgabe es ist, aus einer abstrakten Nachricht eine Antwortnachricht zu erzeugt. Ist der Filter nicht auf die Beschreibung der abstrakten Nachricht eingestellt, lässt er diese durch, das heißt filtert sie nicht heraus und die abstrakte Nachricht wird an den nächsten Filter übergeben. Reply-Filter sollten hierbei ebenso wie ein Handler parametrisier- oder durch Sprachausdrücke steuerbar sein.

Anzumerken ist auch, dass es sowohl Handler geben kann, die keine eingehenden Nachrichten verarbeiten müssen, als auch Handler, die keine Nachrichten senden. Ersteres kann auftreten, wenn durch Constraints Zeitpunkte definiert wurden, diese abgelaufen sind, jedoch keine Nachrichte empfangen wurden. Letztere stellen die Endzustände eines Protokolls dar und verarbeiten nur die letzten ankommenden Nachrichten. Entsprechend sind auch Handler ohne jeglichen Nachrichten-Ein- und -Ausgang denkbar, zum Beispiel Fehlerhandler (siehe 3.2.3.).

3.2.3. Mapping-Dokumente

Ein Mapping stellt eine Verbindung der Handler mit Nachrichten in einem Protokoll her. Es definiert, welcher Handler ausgeführt wird, wenn eine bestimmte Nachricht im Protokoll empfangen wird (Abbildung 3.2). In der Regel wird ein Agent in einem Protokoll nur eine Rolle einnehmen, deshalb müssen in einem Mapping auch nur Handler für die Nachrichten der Rolle(n) definiert werden, die der Agent im Protokoll ausübt.

Für ein Protokoll können mehrere Mapping-Dokumente verwendet werden, so kann zum Beispiel im FIPA-Request-Protokoll für die Konversation mit einem Agent X ein spezielles Verhalten und mit einem Agent Y ein anderes Verhalten gewünscht sein. Hierbei müssen, damit die Ausführungsumgebung die richtigen Handler für Agent X und Y ausführt, im Mapping den initialen Handlern entsprechende Bedingungen zugewiesen werden, die eine Unterscheidung ermöglichen.

Beim eigentlichen Mapping werden für die zu berücksichtigenden Rollen sowie für die für die Rolle relevanten Nachrichten Klassen und Methoden angegeben, die aufzurufen sind, wenn die jeweilige Nachricht auftritt. Das Tupel aus Klasse und Methode stellt dabei den Handler dar. Die einzige Voraussetzung, die hierbei an eine Protokollspezifikation gestellt wird, ist,

dass die Nachrichten und Combined Fragments ein Id-Attribut besitzen, welches zum Referenzieren benötigt wird. Für den Handler können noch weitere Parameter oder ein Sprachausdruck definiert werden, die vom Handler verstanden werden, um dessen Einsatz flexibler zu gestalten (zum Beispiel um den selben Handler für verschiedene Nachrichten verwenden zu können). Des Weiteren gehört zur Spezifikation des Handlers die Angabe der Reply-Filter für alle Nachrichten, die von diesem Handler innerhalb des Protokolls gesendet werden können.

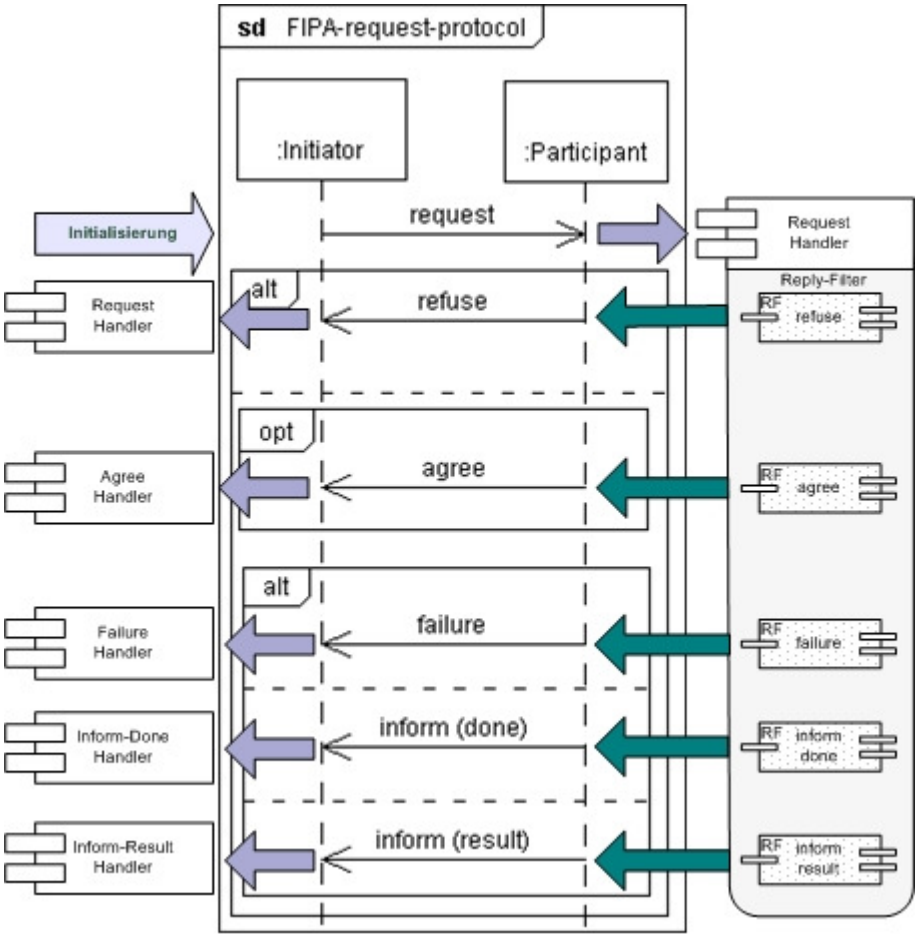


Abbildung 3.2: Graphische Darstellung eines Mappings mit Handlern und Reply-Filtern auf Nachrichtenebene.

In einigen Situationen ist das Handling auf Nachrichten jedoch unvorteilhaft, so kann durch das Id-Attribut der Combined Fragments erreicht werden auf dieser Ebene einen Handler anzumelden. So kann es zum Beispiel in be-

stimmten parallelen Abläufen, in deren Subpfaden nur jeweils eine Nachricht vorkommt, notwendig sein, dass alle parallelen Nachrichten angekommen sind, bevor mit der Verarbeitung begonnen werden kann.

Des Weiteren kann das Id-Attribut von Combined Fragments zur optionalen Notierung eines Fehlerhandlers dienen. Es ist zum Beispiel denkbar, dass Constraints an einem Alternative Combined Fragment definiert wurden, aus denen hervorgeht, dass zu einem bestimmten Zeitpunkt eine bestimmte Mindestanzahl von Nachrichten erwartet wird. Wenn zu diesem Zeitpunkt nicht die entsprechende Anzahl angekommen ist, bedeutet dies nach der Definition des Alternative Combined Fragments, dass das Protokoll beendet wird. Dieses bedeutet aber auch, dass kein Handler eines Endzustandes aufgerufen wird. Möchte man jedoch eine abschließende Bearbeitung durchführen, kann ein Fehlerhandler an einen Combined Fragment deklariert werden, der dann ausgeführt wird.

3.3. Funktionen einer Ausführungsumgebung

Ein Agent, der ein Protokoll ausführen will, teilt der Ausführungsumgebung mit, unter welchem Mapping er dieses durchführen will. Der Agent übergibt dabei die initiale Nachricht und wartet dann auf das Resultat der Protokollkommunikation. Die Ausführungsumgebung instanziiert mit der Übergabe von Mapping und initialer Nachricht eine Ablaufkontrolle für das Protokoll.

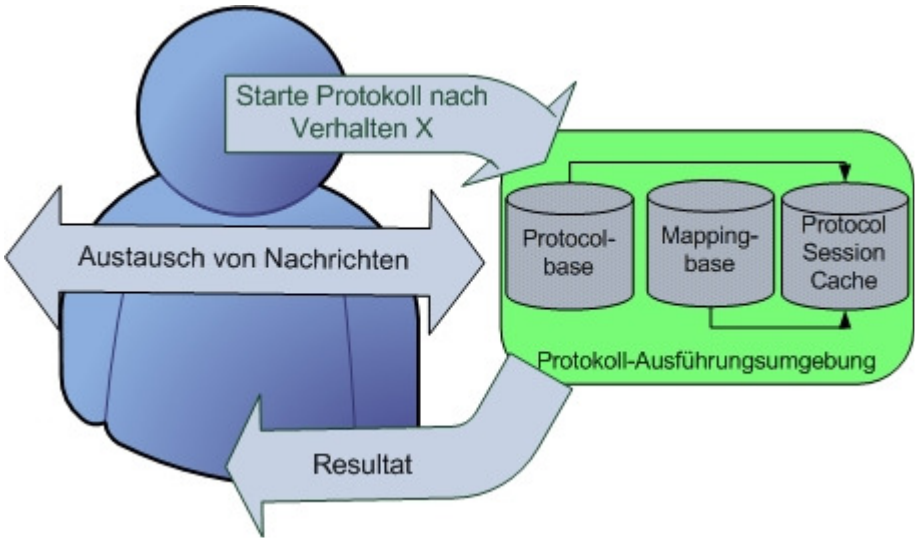


Abbildung 3.3: Transparente Protokollkommunikation über eine Ausführungsumgebung.

Die Ausführungsumgebung instanziiert beim Eingang einer Nachricht das entsprechende Protokoll (bzw. setzt es fort) und führt aus dem Mapping den entsprechenden Handler aus, in dem sie die Nachricht an den Handler über-

gibt. Der Handler verarbeitet die Nachricht und erzeugt eine oder mehrere Antwortnachrichten, die direkt bzw. über einen Reply-Filter an die Ausführungsumgebung zurückgegeben werden.

3.3.1. Fehlerkontrolle

Die von den Handlern und Reply-Filtern übergebenen Nachrichten werden von der Ausführungsumgebung dahingehend überprüft, ob die Nachrichten des Handlers innerhalb des Protokolls und des aktuellen Protokollzustands gültig sind. Der Protokollzustand ermittelt sich hierbei aus der Sequenz der relevanten[13] Nachrichten, die bis zum Zeitpunkt des Handleraufrufs ausgetauscht worden sind. Durch den Protokollzustand ist genau festgelegt, welche Nachrichten ein Handler erzeugen darf und wie die Reihenfolge dieser Nachrichten ist. Durch die gesendeten Nachrichten ergibt sich entsprechend, welche Nachrichten erwartet werden und somit, welche Protokollzustände bzw. Handler erreicht werden können.

Wird der Ausführungsumgebung eine eingehende Nachricht übergeben, muss diese zunächst bestimmen, welchem Handler diese zuzuordnen ist, und damit prüfen, ob es sich bei der Nachricht um eine erwartete, das heißt gültige Nachricht im aktuellen Zustand handelt oder nicht. Handelt es sich um eine ungültige Nachricht, kann die Ausführungsumgebung diese direkt zurückweisen, z.B. mit einer Nachricht mit dem Performativ „`not-understood`“. Handelt es sich bei einer ungültigen Nachricht um eine Nachricht innerhalb eines Protokolls, das vom Agenten der Ausführungsumgebung initiiert worden ist, kann die Ausführungsumgebung als Ergebnis einen Fehler an den Agenten zurückliefern.

Im Fehlerfall sollte eine Ausführungsumgebung im Allgemeinen das Protokoll, in dem der Fehler aufgetreten ist, beenden, da nicht mehr sichergestellt werden kann, dass die kommunizierenden Agenten synchron in ihren Protokollzuständen sind. Hierbei muss die Ausführungsumgebung jedoch berücksichtigen, ob für den jeweiligen Fehlerfall eine entsprechende Behandlung im Protokoll vorgesehen ist. Ist eine Behandlung vorgesehen, handelt es sich nicht um einen Fehler in der Protokollkommunikation sondern um eine behandelte Ausnahmesituation innerhalb des jeweiligen Protokolls. Insbesondere ist es wichtig, dass in der Protokollspezifikation mögliche Ausnahmen direkt mit angegeben sind. Zum Beispiel beim FIPA-ContractNet-Protokoll das FIPA-Cancel-Meta-Protokoll[14] direkt in der Spezifikation des Contract-

[13] relevant bezüglich der Protokollrolle des Agenten, der die Ausführungsumgebung benutzt.

[14] Das FIPA-Cancel-Meta-Protokoll beschreibt im FIPA-ContractNet-Protokoll eine Nachrichteninteraktion, die durch einen Cancel-Akt zu jedem Zeitpunkt im ContractNet-Protokoll auftreten kann und damit dieses Abbricht.

Nets angegeben wird, etwa in einem Break-Combined-Fragment. Dieses ist deshalb wichtig, um eventuell getroffene Übereinkünfte, die bis zum Auftreten des Fehlerfalls zugesagt worden sind, widerrufen zu können.

3.3.2. Constraints

Constraints bestimmen, wie Nachrichten und Pfade in einem Protokoll benutzt werden. Nicht erfüllte Constraints, die Nachrichten zugeordnet sind, können verhindern, dass diese gesendet werden und im Falle der Protokollausführung verhindern, dass ein Handler aufgerufen wird. Constraints werden im Allgemeinen eingesetzt, um einen Pfad aus mehreren auszuwählen, zum Beispiel bei Alternativen.

Da für verschiedene Bedürfnisse verschiedene Sprachen zur Deklaration von Constraints in Protokollen Verwendung finden, ist es wichtig, dass eine Protokoll-Ausführungsumgebung mit einem Sprachinterpreter für diese Zwecke erweiterbar ist. Mit Hilfe eines allgemeinen Interfaces für Sprachinterpreter ist es dann der Ausführungsumgebung möglich, zu ermitteln, welche Handler aufrufbar sind, welche Pfade gewählt werden können und auf wie viel Nachrichten gewartet werden muss.

3.3.3. Zeitliche Constraints

Die Überwachung zeitlicher Constraints ist eine der wichtigsten Aufgaben einer Protokoll-Ausführungsumgebung. Diese Constraints können hierbei in zwei verschiedene Arten unterteilt werden; zum einen solche, die bereits im Protokoll festgelegt wurden, zum anderen Constraints, die nur über den Reply-By-Parameter in ACL-Nachrichten ohne explizite Nennung in der Protokollspezifikation definiert sind. Des Weiteren lassen sich zeitliche Constraints in eigene und fremde unterscheiden, wobei eigene zeitliche Constraints solche sind, die der eigene Agent anderen Agenten auferlegt, während fremde von anderen Agenten auferlegte zeitliche Bedingungen sind.

Die Einhaltung fremder Constraints kann dabei kaum von der Ausführungsumgebung überwacht werden, in dem Sinne, dass es ihr nicht möglich ist die Abarbeitung des Handlers zu beeinflussen. Die Möglichkeiten, bei fremden Constraints liegen für die Ausführungsumgebung beim Empfang, in dem ein Protokoll zum Beispiel beendet wird, falls der Antwortzeitpunkt bereits verstrichen ist. Das Unterbinden des Versands von Nachrichten, welche die fremden zeitlichen Constraints nicht einhalten ist jedoch nicht sinnvoll, da die Entscheidung, ob eine Nachricht noch als gültig zugelassen wird, eine Entscheidung ist, die der Empfänger treffen sollte. In einem solchen Fall wäre es nämlich denkbar, dass der Empfänger eine Nachricht noch als gültig interpretiert, weil zum Beispiel die Uhren von Sender und Empfänger nicht synchron laufen.

Bei der Einhaltung eigener Constraints kann die Ausführungsumgebung zum Beispiel alle Nachrichten sammeln, die in einem Protokoll zu einem bestimmten Zeitpunkt erwartet werden und diese bei Erreichen des Zeitpunktes oder Ankunft aller erwarteter Nachrichten an den oder die zuständigen Handler übergeben. Eine solche gepufferte Bearbeitung kann bei vielen zu erwartenden Nachrichten jedoch nicht sinnvoll sein. Bei vielen Nachrichten kann es zur Beschleunigung (zum Beispiel einer Call-For-Proposal-Verhandlung) beitragen, wenn die eingehenden Nachrichten bereits beim Eingang einer Vorverarbeitung unterzogen werden und erst mit Ablauf der Frist eine Endbearbeitung stattfindet. Für eine solche zweistufige Bearbeitung sind zwei verschiedene Vorgehensweisen denkbar, eine sequentielle Vorbearbeitung bei Nachrichteneingang oder eine parallele Vorbearbeitung aller eingehenden Nachrichten. Wobei jede dieser Vorgehensweisen durch einen Endbearbeitungshandler abgeschlossen wird. Diese Endhandler werden gestartet wenn alle Vorbearbeitungshandler beendet worden sind und entweder alle erwarteten Nachrichten angekommen sind oder der Zeitpunkt des Constraints überschritten wurde. Die jeweilige Vorgehensweise ist dabei über das Mapping-Dokument spezifizierbar.

Zur Überwachung der zeitlichen Constraints ist es die Aufgabe der Ausführungsumgebung, den Reply-By-Wert der ACL-Nachrichten gemäß des Protokolls zu setzen, bzw. zu überprüfen, ob durch Handler oder Reply-Filter gesetzte Werte Protokollkonform sind. Insbesondere sind entsprechende Timer zustarten.

Gerade bei zeitlichen Constraints kann es vorkommen, dass ein Timer ausläuft und keinerlei Nachrichten eingegangen sind, die an den nächsten Handler übergeben werden können. Ist es für den entsprechenden Protokollzustand durch nichtzeitliche Constraints definiert, dass die Mindestanzahl der benötigten Nachrichten größer gleich Eins sein muss und spezifiziert das Protokoll für diese Fälle keine Alternativen, muss das Protokoll beendet werden, ohne dass ein Handler eines Endzustandes ausgeführt wird. Um dennoch eine abschließende Bearbeitung durchführen zu können, kann in einem solchen Fall ein optionaler Fehlerhandler, der am zugehörigen Combined Fragment notiert wurde, aufgeführt werden, zum Beispiel um Roll-Backoperationen durchzuführen.

3.3.4. Dienst-Propagierung

Ein Dienst ist eine Menge von funktional zusammenhängenden Mechanismen, die zum Betrieb des Agenten und anderer Dienste dienen [FIPA, 2000]. Agenten können zumeist ein Protokoll deshalb ausführen, weil sie einen Dienst anbieten, der nach diesen Protokoll abläuft, bzw. innerhalb dessen dieses Protokoll vorkommt. Der Dienst steht dabei in unmittelbarem Zusammenhang mit dem Verhalten eines Agenten bezüglich eines oder mehrerer Protokolle und damit in Beziehung zu deren Mappings.

Um den Entwicklern eine einfache Möglichkeit zu geben, einen Dienst zu propagieren und die damit verbundenen Mappings zu instanziieren, wird ein Service-Mapping eingeführt. Ein Service-Mapping besteht aus einer Service-Description und einer Menge von URLs, die auf die Mappings der verwendeten Protokolle referenzieren. Eine Service-Description besteht in diesen Zusammenhang mindestens aus einem Dienstnamen und einer Menge von Protokollen, hier die Protokolle zu den Mappings.

Durch das Instanziieren eines Service-Mappings, das heißt das Laden aller Mappings und deren Protokolle, in die Ausführungsumgebung um diese Protokolle ausführen zu können, entfällt die Notwendigkeit alle Mappings einzeln instanziieren und den Dienst selbst anmelden zu müssen. Die Anmeldung kann Ausführungsumgebung direkt nach abgeschlossener Instanziierung selbst durchführen. Des Weiteren kann durch eine Ausführungsumgebung ein Dienst abgemeldet werden. Dieses kann zum Beispiel sicher oder abrupt geschehen. Sicher heißt hierbei, aktive Protokollinstanzen können bis zu deren Beendigung fortgeführt werden, während der Aufbau neuer Protokollinstanzen unterbunden wird. Ein abruptes Abmelden hingegen, würde keinerlei Kommunikation über die Protokolle des Dienstes zulassen und laufende Instanzen radikal bzw. nach den im Protokoll spezifizierten Mechanismen abbrechen.

Durch das Einbinden von Diensten in den Aufgabenbereich einer Protokoll-Ausführungsumgebung entlastet diese damit den Entwickler bzw. den Agenten beim Service Deployment einschließlich der Service Propagierung und bei der Service Revocation.

Beispielsweise bietet ein Agent einen Zeitdienst an, dieser umfasst Protokolle zum Registrieren, Abmelden, zur Zeitübermittlung und zur Beendigung des Dienstes von Anbieterseite. Vom Entwickler werden diese vier Protokolle zusammen mit der Service-Description in einem Service-Mapping angegeben. Nun würde es ausreichen, der Ausführungsumgebung mitzuteilen „führe den Dienst nach diesem Service-Mapping aus“, wodurch die Ausführungsumgebung die Protokolle instanziieren und anschließend den Dienst veröffentlichen kann. Um den Dienst zu beenden, würde eine Anweisung „stoppe den Dienst“ genügen und die Ausführungsumgebung führt das besonders markierte Protokoll zur Beendigung des Dienstes aus und lässt während dessen keine Anmeldungen mehr zu. Damit würde sich jede dieser Aktionen auf einen Einzeiler verringern.

4. Realisierung

Nachdem im vorherigen Abschnitt auf die grundlegenden Anforderungen an eine Protokoll-Ausführungsumgebung und auf das Handlerkonzept eingegangen wurde, wird in diesen Abschnitt eine Prototyp-Implementierung beschrieben. Im Anschluss an die Vorstellung des Prototyps wird eine Protokollimplementierungen zur Bewertung des Handlerkonzepts präsentiert.

4.1. Umsetzung

Zur Umsetzung einer Protokoll-Ausführungsumgebung wurde ein Ansatz gewählt, der es Erlauben soll, die Ausführungsumgebung für jeden Agenten einer FIPA-konformen Agentenplattform aufzusetzen. Dieser Ansatz schließt dabei auch das Handlerkonzept mit ein, in dem es dieses um die Plattformunabhängigkeit erweitert.

Der plattformunabhängige Ansatz liegt darin begründet, dass beide FIPA-konformen Agentenplattformen des Fachbereichs Informatik der Universität Hamburg, Multi-Agent Nets (Mulan) [Köhler, Moldt, Rölke, 2001] und Jadex [Pokahr, Braubach, Lamersdorf, 2003], ein BDI-Aufsatz auf die JADE-Plattform [Bellifemine et al., 2000] der Telecom Italia, von einer Protokoll-Ausführungsumgebung profitieren können. Ein Vorteil, der sich aus der Plattformunabhängigkeit ergibt, ist die Möglichkeit, jederzeit die Implementierung eines Protokolls, sprich das Mapping samt Handlern, für Agenten anderer Agentenplattformen verwenden zu können.

Im Folgendem wird die FIPA Interaction Protocol Execution Engine (FIPEE) beschrieben, die Umsetzung der Protokoll-Ausführungsumgebung.

4.1.1. Funktionen von FIPEE

Einleitend wird hierzu in Abbildung 4.1 nochmals ein Überblick über Protokolle, Mappings und Handler im Zusammenspiel mit der Ausführungsumgebung gegeben.

Jede Instanziierung eines Protokolls in Form eines Mappings ist an der Ausführungsumgebung angemeldet, somit ergibt sich für eine Menge von Protokollen P eine Menge M mit $|M| \geq |P|$ Mappings. Die Mappings mappen die Nachrichten eines Protokolls auf einen Satz von Handlern, denen wiederum eine Menge von Reply-Filtern zugeordnet ist, um Antwortnachrichten zu generieren. Handler sollten dabei so modular sein, dass sie auch unter ande-

ren Mappings und anderen Protokollen Verwendung finden können, sodass die Menge der Handler H kleiner als die Anzahl der in allen Mappings ($m \cdot h$) notierten Handler ist ($|H| < m \cdot h$).

Dies bedeutet, der Agent, der die Ausführungsumgebung benutzt, teilt dieser mit, in welchen Protokollen er sich wie verhalten möchte. Dabei ist es möglich, dass er für ein und dasselbe Protokoll verschiedene Verhaltensvarianten angibt, zum Beispiel eine allgemeine und eine für eine spezifische Gruppe anderer Agenten. Die Handler, die hierbei das Verhalten darstellen, sind so programmiert worden, dass sie nicht ausschließlich das Verhalten in einem einzigen Protokoll und einem einzigen Mapping darstellen, sondern sowohl in anderen Mappings zum selben Protokoll, als auch und insbesondere in anderen Protokollen verwendet werden können. Dieses wird vor allem dadurch erreicht, dass Reply-Filter aus dem Ergebnis eines Handlers Antworten generieren.

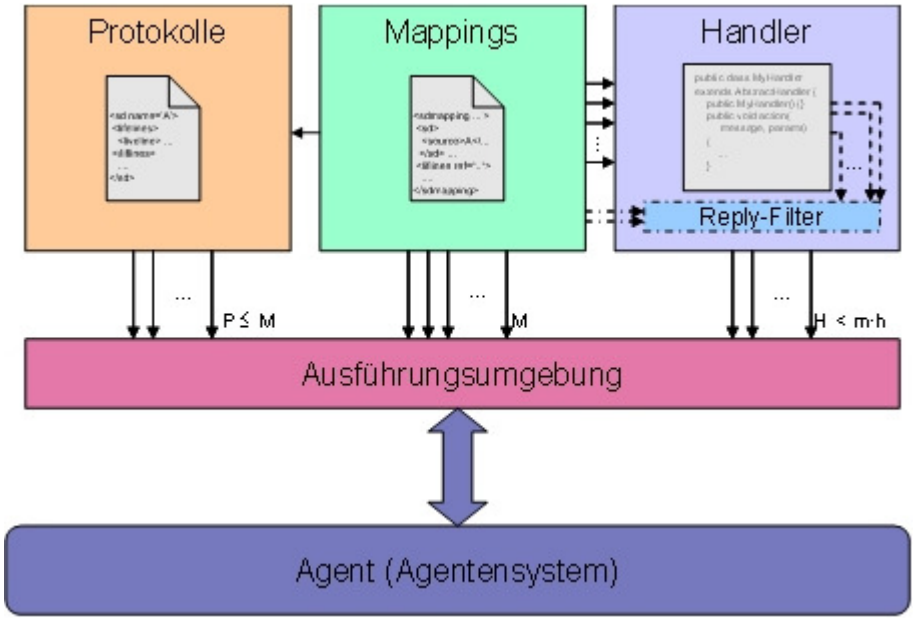


Abbildung 4.1: Architektur im Zusammenspiel von Agent, Ausführungsumgebung, Protokollen, Mappings und Handlern

Der allgemeine Ablauf beim Empfang einer Nachricht durch den Agenten sieht dabei so aus, dass dieser Nachrichten protokollorientierter Kommunikation an die Ausführungsumgebung weiterleitet (Abbildung 4.2). Die Ausführungsumgebung überprüft hierbei, ob bereits eine Konversation mit der Conversation-Id, die in der Nachricht angegeben ist, geführt wird (Punkt 1). Ist dieses nicht der Fall und es handelt sich bei der Nachricht um eine initiale Nachricht (bezüglich der Rollen des Agenten in diesen Protokoll), wird ein passendes Mapping zum Protokoll der Nachricht gesucht. Gibt es mehrere Mappings, so müssen für diese Constraints zu den initialen Nachrichten spe-

zifiziert sein, die mit Hilfe der Parameter der eingegangenen Nachricht entschieden werden können.

Gibt es eine Konversation mit der Conversation-Id, so wird von der Ausführungsumgebung die Gültigkeit der Nachricht anhand des aktuellen Protokollzustandes überprüft. Handelt es sich um keine gültige Nachricht, reagiert die Ausführungsumgebung mit einer Not-Understood-Nachricht, deren Content mit der eingegangenen Nachricht besetzt wird, und das Protokoll wird beendet.

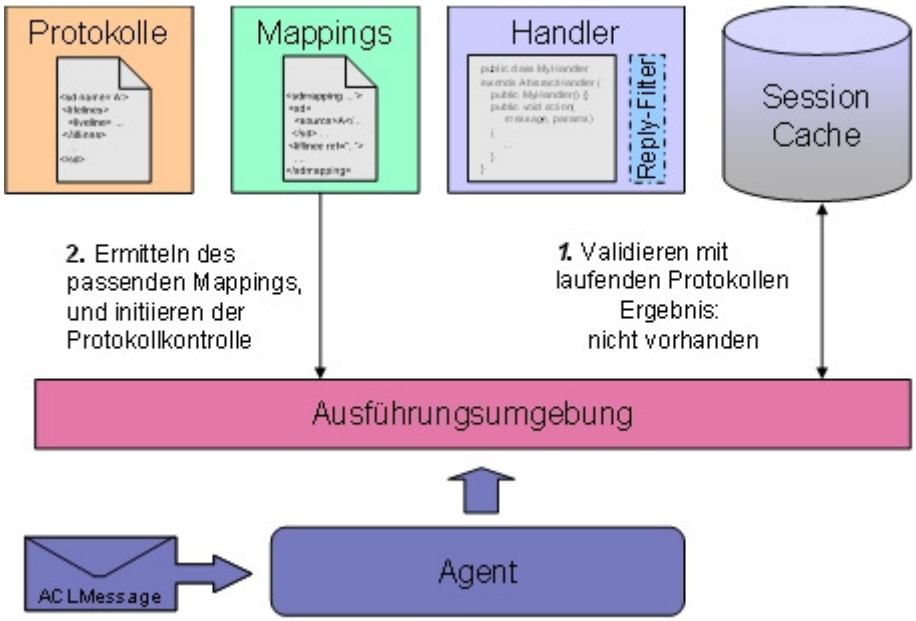


Abbildung 4.2: Instanziierung der Protokollkontrolle durch eingehenden Nachrichten.

Ist festgestellt worden, dass es sich um eine gültige Nachricht handelt, so wurde dadurch auch bestimmt, welcher Handler aufzurufen ist. Der Handler wird aufgerufen, in dem seine Klasse instanziiert wird und ihm die Nachricht übergeben wird (Punkt 2). Ein Handler kann verschiedene Ergebnisse erzeugen; handelt es sich beim Ergebnis des Handlers um ein abstraktes, so wird dieses an die Reply-Filter des Handlers übergeben. Die Übergabe geschieht dabei in der Sequenz, in der die Reply-Filter im Mapping notiert sind. Erzeugt ein Reply-Filter kein Ergebnis, dass heißt er filtert das abstrakte Ergebnis des Handlers nicht, so wird der nächste Reply-Filter aufgerufen. Dies geschieht solange, bis eine Nachricht von einem Reply-Filter erzeugt wird. Das Ergebnis von Handler oder Reply-Filter wird anschließend gegen das Protokoll validiert (Punkt 3). Bei der Validierung werden alle Parameter der Nachricht mit den Erfordernissen des Protokolls verglichen. Konnte weder

ein passender Reply-Filter gefunden noch die Nachricht validiert werden, so reagiert die Ausführungsumgebung mit dem Versenden einer Failure-Nachricht, sowie einem internen Ausnahmefehler und beendet das Protokoll. Da Fehler dieser Art in einer finalen Implementierung eines Protokolls nicht auftreten sollten, dienen die Failure-Nachricht und der Ausnahmefehler zur Validierung während der Implementierungsphase des Protokolls. Wurde ein gültiges Ergebnis erzeugt, so wird zunächst einmal ermittelt, welche Nachrichten nach dem Versand der Nachricht als nächstes zu erwarten sind (Punkt 4). Dieses dient zur beschleunigten Nachrichtenbearbeitung in Punkt 1. Anschließend wird die Antwortnachricht über den Agenten verschickt.

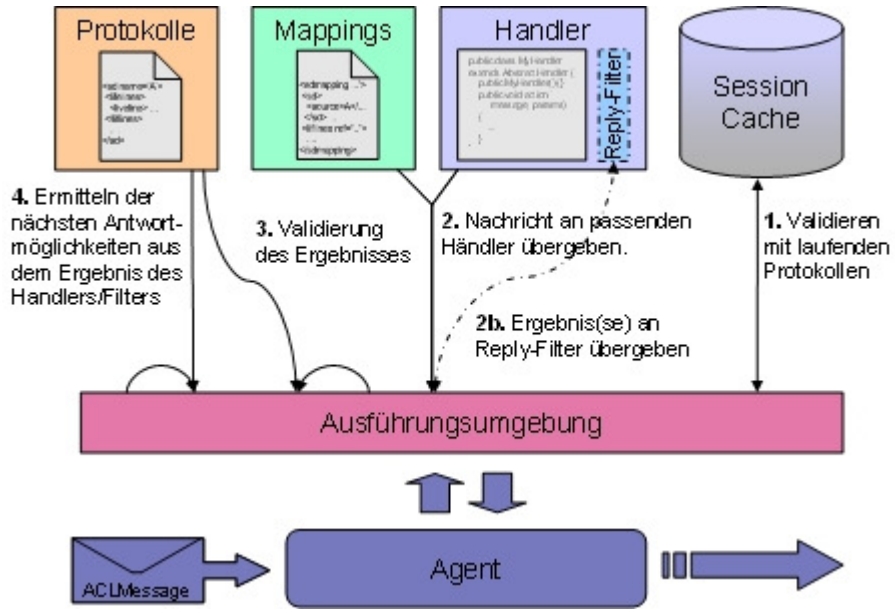


Abbildung 4.3: Ablauf bei eingehender Nachrichten

4.1.2. Komponenten von FIPEE

Die Ausführungsumgebung besteht aus verschiedenen Komponenten und Schnittstellen (Abbildung 4.4).

Hinter der Komponente XMLParser verbirgt sich ein zweistufiger Parser, zum Parsen der Mapping- und Protokoll-Spezifikationen. In der ersten Stufe wird dabei auf XMLSchema-Validität der Dokumente geprüft, in der zweiten die Übereinstimmung von Mapping und Protokoll verifiziert. Aus den Protokollen werden interne Repräsentationen im Generic-Cache gespeichert, interne Modelle der Mappings werden im Passive-Cache abgelegt. Der Ge-

neric-Cache enthält somit alle von FIPEE unterstützen generischen Protokollspezifikationen ohne Verhaltensinformationen, während die Verhaltensinformationen im Passive-Cache gespeichert sind.

Die Komponente Execution-Control steht in direkter Beziehung zum Agenten, indem beide über verschiedene Interfaces miteinander kommunizieren. Sie instanziiert Protokolle aus den Daten im Passive- und Generic-Cache, deren Instanzen im Runtime-Protocol-Cache verwaltet werden. Die Execution-Control hat des Weiteren die Aufgabe, eingehenden Nachrichten laufenden Protokollinstanzen des Runtime-Protocol-Cache zuzuweisen. Bei dieser Zuweisung ermittelt sie, welche Handler zuständig sind und instanziiert deren Instanzen, denen die Nachricht übergeben wird. Dabei sorgt sie dafür, dass keine falschen Nachrichten an die Handler übergeben und oder von diesen gesendet werden. Hierzu wird der Zustand des laufenden Protokolls über dessen generischen Repräsentation mit der Nachricht verglichen.

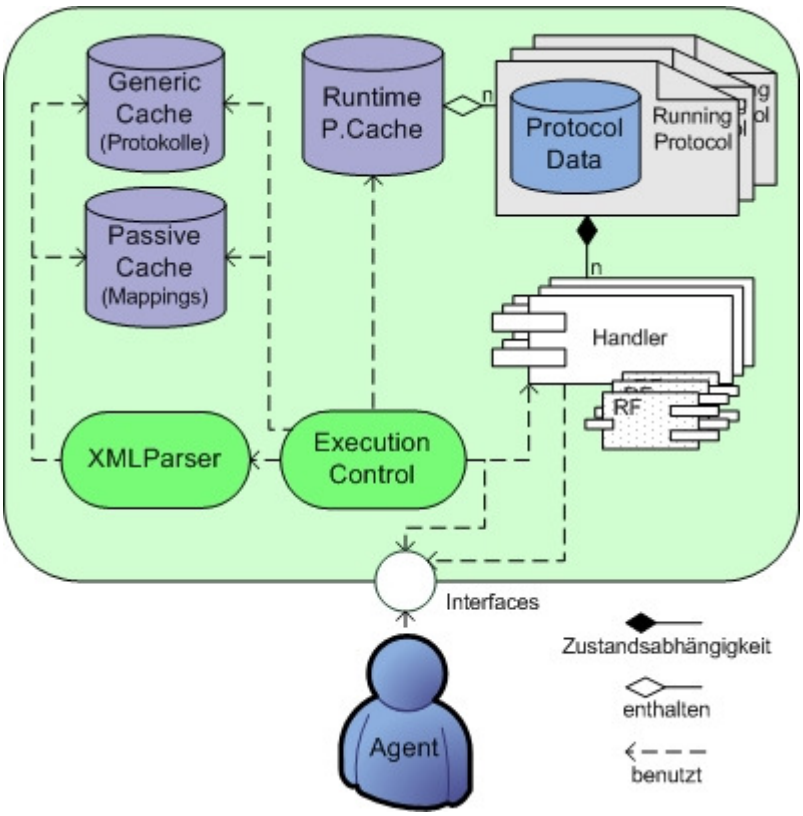


Abbildung 4.4: Hauptkomponenten von FIPEE

Den laufenden Protokollinstanzen ist ein Datenspeicher beiseite gestellt, in dem während des Ablaufs Informationen in Form von Schlüssel-Wert-Paaren gespeichert werden können. Des Weiteren werden im Datenspeicher Informationen über den Status des Protokolls, wie zum Beispiel „in Bearbeitung“, „erfolgreich“ oder „fehlgeschlagen“ hinterlegt, sowie ein eventuelles Gesamtergebnis gespeichert. Der Datenspeicher steht allen Handlerinstanzen des Protokolls zur Verfügung. Somit verringert er den Verwaltungsaufwand, der bei Benutzung der Beliefbase oder ähnlichen Speicherformen von Agenten entstehen würde, da dieser Speicher bereits sitzungsbezogen ist.

4.1.3. Interfaces zur Anbindung eines Agenten

Die Interfaces von FIPEE decken zwei Bereiche ab: Kommunikation und Funktion. Die Kommunikationsinterfaces dienen zur Kommunikation von FIPEE mit dem Agenten. Die Funktionsinterfaces stellen Funktionen für die Handler, Reply-Filter und Mapping-Constraints zur Verfügung, damit diese auf Funktionen des Agenten zugreifen können.

Zur Gruppe der Kommunikationsinterfaces gehören die Interfaces SendPort, ResultHandler, EventDispatcher und MessageConverter. Der SendPort (Abbildung 4.5) wird von der Execution-Control genutzt, um die Nachrichten der Handler und Reply-Filter zu verschicken. Die InconvertibleException kann dabei vom weiter unter besprochenen MessageConverter hervorgerufen werden.

```
void sendMessage(fipee.lang.ACLMessage m)
            throws InconvertibleException
```

Abbildung 4.5: Signatur des Interface SendPort

```
void dispatchEvent(fipee.ee.ProtocolEvent pe)
```

Abbildung 4.6: Signatur von EventDispatcher- bzw. ResultHandler-Interface

Die Interfaces ResultHandler und EventDispatcher (Abbildung 4.6) dienen beide dazu, dem Agenten, der ein Protokoll gestartet hat, das Ergebnis dieses Protokolls mitzuteilen. Die Signatur dieser beiden Interfaces ist dieselbe. Der Unterschied zwischen EventDispatcher und ResultHandler besteht darin, dass ein ResultHandler speziell für das Ergebnis einer Protokollinstanz zuständig ist, während der EventDispatcher als eine Schnittstelle zu einem

agenteninternen Event-Mechanismus zu verstehen ist, die benutzt wird, um die Ergebnisse von verschiedenen Protokollabläufen zu übermitteln. Der ResultHandler kann bei der Instanziierung eines Protokolls optional angegeben werden, wodurch diesem nach Beendigung des Protokolls das Resultat der Konversation mitgeteilt wird. Der EventDispatcher hingegen wird bei der Initialisierung der Execution-Control angegeben und anstelle eines ResultHandlers aufgerufen, falls dieser bei der Instanziierung eines Protokolls nicht angegeben wurde. Benötigt wird höchstes eines der Interfaces. Das Event, das der dispatchEvent-Methode des Interfaces übergeben wird, kann in verschiedenen Varianten auftreten: zum einen als FailureEvent, wenn bei der Protokollausführung ein interner oder nicht im Protokoll behandelter Fehler auftritt, oder als ResultEvent, wenn das Protokoll gemäß seiner Spezifikation beendet wurde. Beide Events besitzen eine Referenz des Datenspeichers der Protokollinstanz, sowie die Conversation-Id des Protokolls.

Aufgrund des plattformunabhängigen Ansatzes ist es erforderlich ein internes Nachrichtenformat zu benutzen; der MessageConverter (Abbildung 4.7) erfüllt diesen Zweck. Mit Hilfe eines MessageConverters können Agent Identifier und ACL-Nachrichten in die FIPEE-interne Repräsentation überführt werden. Die verwendete interne Repräsentation stellt dabei eine Art Obermenge der Nachrichtenformate von JADE und Mulan dar, in dem Sinne, dass beide zueinander nicht vollständig kompatiblen Formate[15] in FIPEE ACL-Nachrichten abgebildet werden können. Aus diesem Grund ist für alle Methoden des MessageConverters eine InconvertibleException deklariert worden. Der MessageConverter kann nämlich dazu benutzt werden, um z.B. JADE-Nachrichten in Mulan-Nachrichten zu konvertieren, wobei dies nicht für alle Nachrichten gelingt[15].

[15] JADE ACL-Nachrichten unterstützen nur von FIPA spezifizierte Performative, während diese auch eigene erlaubt (Stand Version 3.1). Mulan erlaubt mehre Ontologien für eine Nachricht (Version 14.2001), dies ist zwar durchaus als FIPA konform zu sehen, jedoch gibt es keine Nachrichtenrepräsentation der FIPA, die dieses berücksichtigt.

```
public abstract AID toFipeeAID(Object aid)
       throws InconvertibleException

public abstract Object fromFipeeAID(AID aid)
       throws InconvertibleException

public abstract ACLMessage
       toFipeeACLMessage(Object message)
       throws InconvertibleException

public abstract Object
       fromFipeeACLMessage(ACLMessage message)
       throws InconvertibleException
```

Abbildung 4.7: Signatur der abstrakten Klasse MessageConverter

Die Gruppe der Funktionsinterfaces umfasst einen Beliefbase-Wrapper, das Handler-Interface, sowie Reply- und Mapping-Filter-Klassen.

Der Beliefbase-Wrapper erlaubt es, ein und denselben Handler auf verschiedenen Agentensystemen über ein einheitliches Interface auf die Wissensbasen zugreifen zu lassen. Das heißt, bei der Migration eines Handlers von einer Agentenplattform zu einer anderen muss nur der Beliefbase-Wrapper angepasst werden. Dabei wird im Allgemeinen angenommen, dass Anfragen auf der Beliefbase nur eine Kopie des originalen Wertes zurückliefert. Die Struktur des Wrappers ist an die Jadex Beliefbase angelehnt, so gibt es Beliefs und Beliefsets, Funktionen zum Entfernen, Modifizieren und Erzeugen von diesen und Suchfunktionen.

Die abstrakte Klasse BasicHandler stellt Basisfunktionen für die Entwicklung von Handlern zur Verfügung. Sie stellt zum Beispiel Informationen über das aktuelle Protokoll und gültige Antwortnachrichten,dem Zugriff auf die Beliefbase des Agenten über den Beliefbase-Wrapper oder direkt, sowie Funktionen zum Nachrichtenversand bereit. Beim Nachrichtenversand wird zwischen drei Varianten unterschieden: indirektes Antworten, direktes Antworten und Senden einer einfachen Nachricht. Beim indirekten Antworten wurde vom Handler eine unvollständige Nachricht (abstrakte Nachricht) erstellt, die beispielsweise nur den Content enthält, diese wird dann von der Execution-Control an einen ReplyFilter übergeben. Beim direkten Senden muss die Nachricht des Handlers vollständig und korrekt sein; sie wird dabei ohne ReplyFilter nach Prüfung durch die Execution-Control von dieser versandt. Bei einer einfachen Nachricht handelt es sich um eine Nachricht, die nicht zu einen Protokoll gehört; sie wird direkt ohne Prüfung über die Execution-Control versandt. Implementierungen der abstrakten Klasse Basi-

cHandler werden als eine Methodensammlung betrachtet, deren Methoden die eigentlichen Handler darstellen. Diese Methoden können den verschiedenen Mappings als Handler angegeben werden. Um die Flexibilität eines Handlers zu erhöhen, können jeder Handlerinstanz Parameter übergeben werden. Die Parameter können dann für die einzelnen Handlermethoden spezifiziert und berücksichtigt werden, um deren Verhalten konfigurieren zu können.

Abstrakte Nachrichten, die für einen ReplyFilter (Abbildung 4.8) bestimmt sind, bestehen aus zwei Teilen: zum einen aus der unvollständigen Nachricht, zum anderen aus einer Beschreibung. Die Beschreibung hilft dem ReplyFilter, die Nachricht zu vervollständigen bzw. zu entscheiden, ob der angesprochene Filter für die Nachricht zuständig ist oder nicht. Ist er für die Nachricht zuständig, so erzeugt er eine vollständige Antwortnachricht die von der Execution-Control weiterverarbeitet wird. Ist der Filter nicht zuständig, erzeugt er keine Nachricht (null) und die Execution-Control ruft den nächsten Reply-Filter auf.

```
public abstract ACLMessage apply(
        fipee.lang.ACLMessage incompleteM,
        fipee.util.KeyValueList declaration);
```

Abbildung 4.8: Signatur eines ReplyFilters

Reply- und Mapping-Filter-Klassen liegen jeweils in den Formen Parameter und Expression vor. Die Parameter-Variante erlaubt die Angabe von Parametern in Form einer Schlüssel-Wert-Paar-Liste, die Expression-Varianten die Angabe von logischen Ausdrücken zur Steuerung des Filterungsprozesses. Reply-Filter und BasicHandler bilden die Grundlage des Handlerkonzepts. Mapping-Filter werden für zweierlei Zwecke benutzt:

- Um Constraints zur Auswahl eines Mappings zu definieren für den Fall, dass für eine Rolle eines Protokoll mehrere Mappings benutzt werden.

- Bei Protokollen, in denen Nachrichten angegeben sind, deren Content eine abstrakte Spezifikation enthält. Zum Beispiel steht im Protokoll: der Inhalt der Nachricht muss eine kommaseparierte Liste von Komponenten eines RGB-Farbwertes sein, in der Reihenfolge R,G,B. Der Parser würde in einem solchen Fall verlangen, dass im Mapping ein Filter angegeben wurde, der die Korrektheit des Contents der Nachricht überprüfen kann.

4.1.4. Agentenanbindung am Beispiel eines Jadex-Agenten

Die Ausführungsumgebung ist in Jadex[16] über einen Plan namens FipeeNegotiatorPlan als so genannter ThreadedPlan eingebunden, der die Interfaces SendPort und EventDispatcher implementiert. Thread Pläne sind Pläne in Jadex, die in einem eigenen Thread ausgeführt werden und in denen eine blockierende Event-Methode (waitFor) aufgerufen werden kann, die hier benutzt wird, um protokollorientierte Nachrichten weiterzuleiten. Dieser Zwischenhändlerplan ist über das Agent-Definition-File[17] (ADF) als Instant-Plan eingebunden (Abbildung 4.9) und steht damit sofort beim Start des Agenten zur Verfügung. Der FipeeNegotiatorPlan erzeugt dabei die Instanz der ExecutionControl für den Agenten und übergibt dieser den Beliefbase-Wrapper zum Zugriff der Handler auf die Jadex-Beliefbase. Um später aus Jadex leichter auf FIPEE zugreifen zu können, wird eine Referenz auf die ExecutionControl in der Beliefbase abgelegt.

Zum einfachen Laden der Protokolle, die der Agenten direkt beim Start ausführen können soll, ist ein spezielles Beliefset namens „fipee_protocol_mapping_set“ im ADF eingetragen (Abbildung 4.9, Zeile 13). Das Beliefset wird vom FipeeNegotiatorPlan durchgegangen und die entsprechenden Mappings geparst. Alternativ kann auch eine Mappingliste als Parameter des Konstruktors des FipeeNegotiatorPlans oder als Element des Beliefsets angegeben werden. Eine Mappingliste ist ein XML-Dokument, mit einer Liste von Mapping-URLs. Die Verwendung einer Mappingliste erleichtert dabei Arbeit, wenn mehrere Agenten die gleichen Protokolle verwenden. Nach dem Parsen ist der Nachrichtenfilter des Plans auf alle von FIPEE aktuell geladenen Protokolle abgestimmt, sodass die erwähnte waitFor-Methode blockiert, bis ein entsprechendes Nachrichtenereignis eintritt. Tritt ein Nachrichtenereignis ein, d.h. wurde dem Agenten eine protokollorientierte Nachricht gesandt, so wird diese zunächst von ihrer Jade-Repräsentation in die interne Nachrichtenrepräsentation von FIPEE (mit Hilfe eines Message-Converters) konvertiert und an die Ausführungsumgebung übergeben. Nachrichten, die von Handlern oder von FIPEE selbst erzeugt werden, werden über die SendPort-Funktion des Plans zurück in Jade-Nachrichten konvertiert und anschließend vom Plan versandt. Protokolle bzw. Mappings können zur Laufzeit geladen und entfernt werden.

---

[16] Für mehr Informationen zu Jadex siehe http://vsis-www.informatik.uni-hamburg.de/projects/jadex.

[17] Das Agent-Definition-File (ADF) ist eine Form der Klassenbeschreibung für Agenten: Vom ADF werden Agenten wie Objekte von deren Klassen instanziiert [Pokahr und Braubach, 2004].

```
 1.  <agent [..] name="FipeeTestAgent">
 2.    <imports>
 3.      <import>fipeex.jadex.*</import>
 4.    </imports>
 5.    <plans>
 6.      <plan name="fipeenegotiator" instant="true">
 7.        <constructor>new FipeeNegotiatorPlan(
 8.                  "http://xyz.net/MappingList.xml")
 9.        </constructor>
10.      </plan>
11.    </plans>
12.    <beliefs>
13.      <beliefset name="fipee_protocol_mapping_set"
14.               class="String">
15.       <fact>"http://xyz.net/fipa-req-map.xml"</fact>
16.       <fact>"http://xyz.net/fipa-cfp-map.xml"</fact>
17.      </beliefset>
18.    </beliefs>
19.  </agent>
```

Abbildung 4.9: Jadex ADF zur FIPEE Instanziierung

Ein Jadex-Agent, der ein Protokoll ausführen möchte, macht dies, in dem er einen ProtocolEventFilter erzeugt (Abbildung 4.10). Ein ProtocolEventFilter ist Teil der Anbindungskomponenten. Diesem Filter wird die initiale Nachricht (Zeile 4) und das Mapping zum Protokoll (Zeile 6) übergeben und der Filter selbst wird einer waitFor-Methode übergeben. Optional kann für das Protokoll bereits der zu verwendende Datenspeicher mit initialen Werten übergeben werden (Zeile 5). Dieses Vorgehen ist bei Jadex nötig, da ein einfacher Methodenaufruf mit Warten auf einen Rückgabewert das gesamte Jadex-System blockieren würde. Hier kommt auch die Funktion des FipeeNegotiatorPlans als EventDispatcher zum Tragen. Das Resultat eines Protokolls wird über den EventDispatcher in ein FipeeProtocolEvent umgewandelt und dem Jadex-Event-Mechanismus übergeben, welcher dafür sorgt, dass das Protokollereignis an der waitFor-Methode ankommt.

```
1. FipeeProtocolEvent fpe =
2.   (FipeeProtocolEvent) waitFor(
3.      new ProtocolEventFilter(
4.              messageCOnverter.toFipeeACLMessage(acl),
5.              protocolData,
6.              "http://xyz.net/fipa-cfp-map.xml",
7.              executionControl
8.       ));
```

Abbildung 4.10: Starten eines Protokolls aus einem Jadex-Plan

Abbildung 4.11 gibt noch einmal einen Überblick über die Komponenten die für Jadex implementiert wurden, um mit FIPEE zu kommunizieren, und über die Interfaces von Jadex, die implementiert worden sind, um die Ergebnisse von Protokollabläufen zu verarbeiten. Der FipeeNegotiatorPlan initiiert dabei FIPEE und stellt SendPort und EventDispatcher für die Execution-Control zur Verfügung.

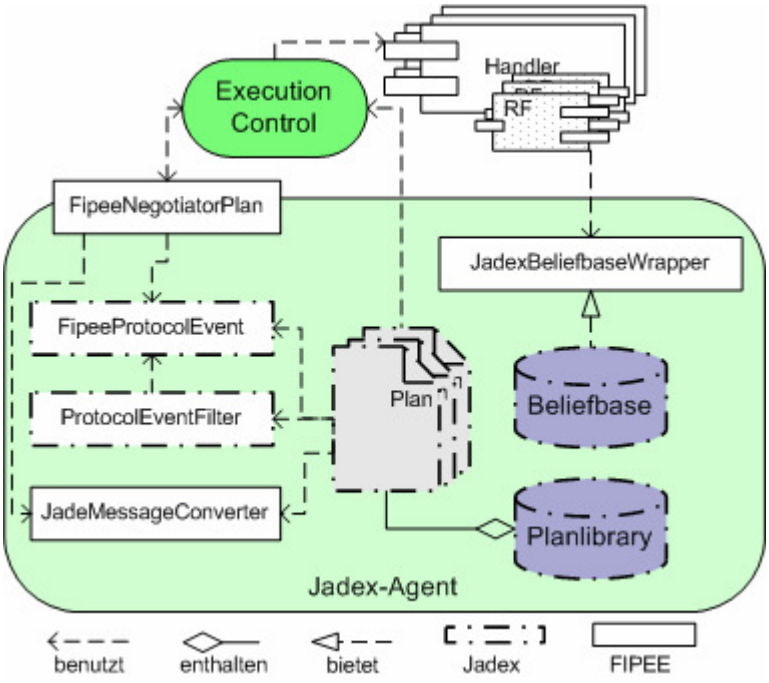


Abbildung 4.11: Anbindung eines Jadex-Agenten an FIPEE

4.1.5. AUML-Metamodelentwurf

Um AUML-Protokolle gleichermaßen spezifizieren und in der Ausführungsumgebung nutzen zu können, ist ein Metamodel erforderlich. Da zum Zeitpunkt dieser Arbeit noch kein AUML-Metamodel von der FIPA vorliegt, wurde ein eigenes Metamodel für AUML entwickelt. Dieses Metamodel wurde sehr einfach gehalten. Dennoch sind die grundlegenden Elemente, hierzu gehören alle Combined Fragments, Nachrichten, Continuations und Referenzen, darstellbar. Obwohl alle Combined Fragments darstellbar sind, werden zurzeit nur die Combined Fragments Alternative, Parallel und Option von FIPEE unterstützt. Abbildung 4.12 zeigt einen Überblick über die zurzeit von FIPEE unterstützten Elemente.

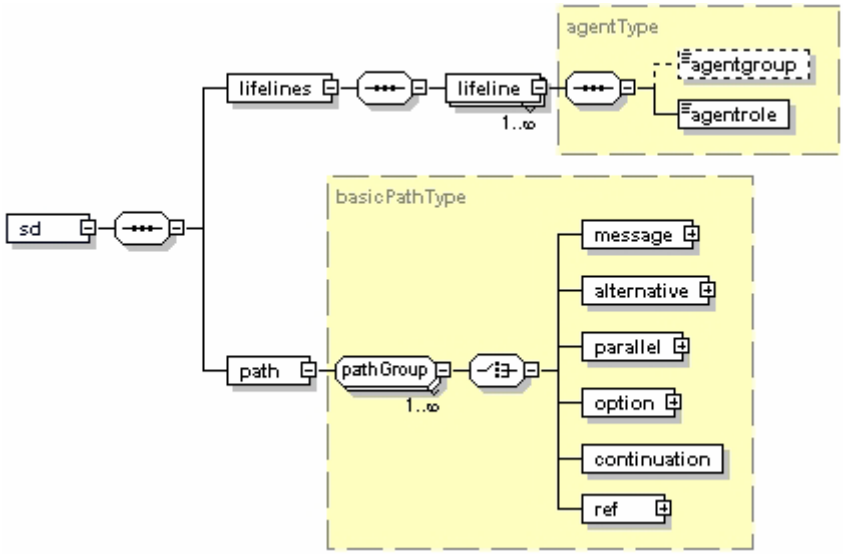


Abbildung 4.12: Von FIPEE unterstützte AUML-Elemente.

Das Element <sd>, für Sequenz-Diagramm, stellt das Wurzelelement einer Protokollspezifikation dar. Alle Elemente unterhalb von `<path>` und jede `<lifeline>` besitzen dabei ein *id*-Attribut, mit dem sie referenziert werden können. Zum Ausdruck der Kardinalität besitzt `<lifeline>` ein Attribut, mit dem diese in Form eines logischen Ausdrucks spezifiziert werden kann.

Das Element `<ref>` kann benutzt werden, um ein anderes Protokoll einzubinden, als seien dessen Elemente anstatt des `<ref>`-Elements notiert. Für eine Abbildung der Rollen des referenzierenden auf die Rollen des referenzierten Protokolls gibt es ein Unterelement `<aliases>`, welches dies ermöglicht.

Die Combined Fragments wie Alternative, Option und Parallel werden wie in Abbildung 4.13 dargestellt. Das Option-Combined-Fragment besitzt im Gegensatz zu Alternative und Parallel sinnvoller Weise nicht die Anschrift 1..∞ am `<path>`-Element, da eine Option keine Subpfade hat. In den Elementen `<condition>` und `<timecondition>` kann ein logischer Ausdruck angegeben werden, um nichtzeitliche und zeitliche Bedingungen zu spezifizieren. Die verwendete Beschreibungssprache der Bedingungen kann in einem *`lang`*-Attribut angegeben werden kann. Beim Element `<condition>` kann zusätzlich noch angegeben werden, ob es sich um eine blockierende oder nicht blockierende Bedingung handelt. Da unter Umständen keine Trennung zwischen zeitlichen und nicht zeitlichen Bedingungen gemacht werden kann, können im `<condition>`-Element auch zeitliche Bedingungen angegeben werden. In diesem Fall ist es aber sinnvoll, im gesamten Protokoll auf die Verwendung des Elements `<timecondition>` zu verzichten. Die Referenz `pathGroup` verweist auf die in Abbildung 4.12 im unteren Zweig angegebenen Elemente, welche an dieser Stelle angegeben werden können.

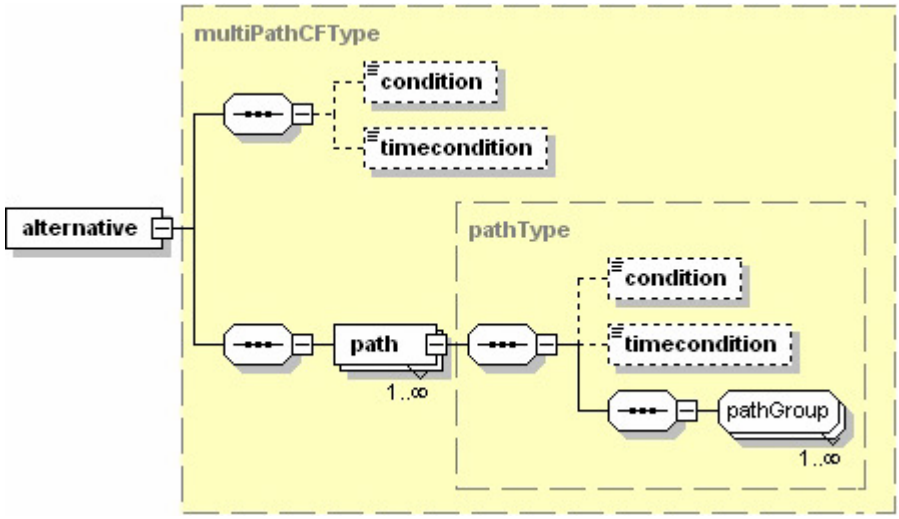


Abbildung 4.13: Alternative-Combined-Fragment im Detail

Das `<message>`-Element (Abbildung 4.14) besitzt zusätzlich zum *`id`*-Attribut noch die Attribute *`type`* und *`receivedAfter`*. Das *`type`*-Attribute dient dazu, anzugeben, ob es sich bei der Nachrichtenart um einen synchronen oder asynchronen Nachrichtenversand handelt, bzw. in dem Fall, dass Sender und Empfänger von der selben Lifeline stammt, ob es sich um asynchronen Versand ohne Einbeziehung des Senders handelt. Da asynchroner Versand für gewöhnlich die Kommunikationsart ist, die in Agentensystemen verwendet wird, ist dieses der Standardwert für das *`type`*-Attribut. Eine

asynchrone Nachricht bedeutet, dass der Agent die Nachricht sendet, ohne danach noch Kontrolle über diese zu haben. Eine synchrone Nachricht bedeutet, dass der Agent den Versand kontrolliert, zum Beispiel wird der Agent, der die Nachricht sendet, warten, bis die Nachricht empfangen wurde, und kann während dessen nichts anderes machen. Das Attribut `receivedAfter` wird dazu benutzt Nachrichtenüberlappung nachzubilden. Es referenziert dabei die ID einer Nachricht, die vor ihr an der selben Lifeline empfangen wird.

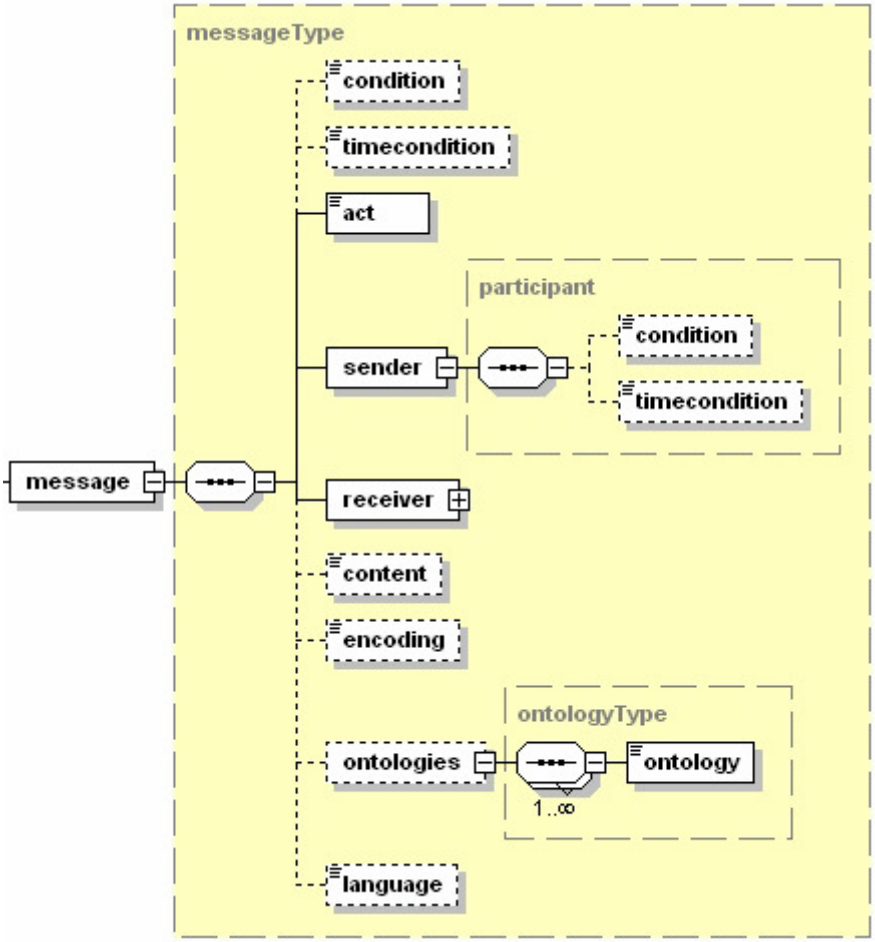


Abbildung 4.14: Das Message-Element im Detail

Die Elemente `<condition>` und `<timecondition>` entsprechen denen der Combined Fragments, sie dienen hier zur Definition allgemein geltender Constraints. Eine spezielle Definition von Constraints auf Sender- und Empfängerseite kann mit den gleichnamigen Unterelementen der Elemente `<sender>` und `<receiver>` erfolgen. Der Element `<act>` spezifiziert den

kommunikativen Akt der Nachricht, wie zum Beispiel inform oder request. Die Elemente <sender> und <receiver> besitzen ein *ref*-Attribut, mit dem die ID einer Lifeline referenziert wird. Für die Sender- und Empfängerseite kann außerdem die Kardinalität angegeben werden. Das optionale Element <content> beschreibt den Inhalt der Nachricht und zwar entweder durch eine konkrete Zeichenkette oder als abstrakte Beschreibung. Die jeweilige Form wird durch ein *type*-Attribut angegeben; der Standardwert ist „*abstrakt*“. Die Interpretation dieses Elements ist wie Folgt: wird das Element weg gelassen, ist jeder Wert als Content zulässig, ist es spezifiziert, so muss dort entweder eine abstakte Beschreibung des Nachrichteninhalts angegeben werden oder das *type*-Attribut auf den Wert „*concrete*" gesetzt werden. Die Elemente <encoding> und <language> beschreiben das Encodingverfahren bzw. die Sprache, die im <content>-Element verwendet werden darf.

Das Element <ontologies> stellt eine Besonderheit bei den inhaltsbeschreibenden Elementen dar. Beim <ontologies>-Element ist es möglich, beliebig viele <ontology>-Unterelemente zu spezifizieren, die die im Content verwendeten Ontologien beschreiben. Dieses ist auf die Spezifikationen der abstrakten FIPA-Architektur [FIPA, 2000] und auf die Spezifikation der ACL-Nachrichten-Struktur [FIPA, 2002b] zurückzuführen, wonach der Parameter Ontology in einer ACL-Nachricht, ein oder mehrere Ontologien angibt, die den Kontext des Inhalts angeben. Im Gegensatz dazu wird in vielen Implementierungen von ACL-Nachrichten der Parameter Ontology so interpretiert, dass nur eine Ontologie zugelassen ist oder nicht speziell behandelt wird[18]. Um die Interpretation von <ontologies> zu erweitern, besitzt dieses Element ein *type*-Attribut, dieses kann die Werte „*any*“, „*one*“ oder „*all*“ annehmen:

- „*all*“ bedeutet dabei, dass die Nachricht alle Ontologien enthalten muss

- „*one*“ bedeutet, dass die Nachricht nur eine dieser Ontologien enthalten darf.

- „*any*“ bedeutet, dass die Nachricht eine, mehrere oder alle Ontologien enthalten darf.

[18] Hierzu zählen die FIPA Spezifikationen SC00069 – SC0071 [FIPA, 2002 c-e] , sowie JADE [Bellifemine et al., 2000].

Die Erweiterung besteht darin, dass man diese Fälle auch durch ein Alternative-Combined-Fragment lösen könnte, dieses würde allerdings nicht die Übersichtlichkeit fördern.

4.1.6. Mapping-Metamodell

Um eine flexible Abbildung von Rollen eines Protokolls auf Code zu ermöglichen, wird ein Metamodell für ein Mapping-Dokument benötigt. Das dafür entwickelte Metamodell (Abbildung 4.15) erlaubt einen flexiblen Einsatz durch Parameter oder Expressions, wodurch der Programmieraufwand verringert werden kann.

Zur Spezifikation eines Mappings ist zunächst mit Hilfe des `<sd>`-Elements die URL des Quellprotokolls anzugeben. Als nächstes wird durch eine oder mehrere `<lifeline>`-Elemente für eine, mehrere oder alle Rollen im Protokoll, die der zu implementierende Agenten übernehmen soll, das Code-Mapping vorgenommen. Um die Rolle im Protokoll zu referenzieren, besitzt das `<lifeline>`-Element zwei Attribute *ref* und *name*, von denen genau eines verwendet werden muss. Das *ref*-Attribut referenziert dabei die im Protokoll spezifizierte ID der Lifeline, das *name*-Attribut das Paar aus Rolle und Agentengruppe. Werden beide angegeben, wird das *name*-Attribut ignoriert.

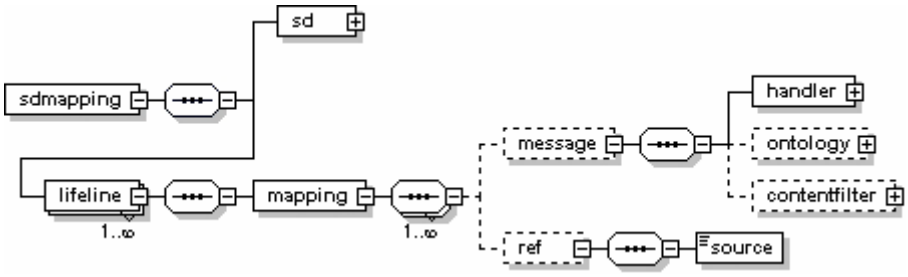


Abbildung 4.15: Übersicht Mapping-Metamodell

Um das Mapping durchzuführen, beschränkt sich der aktuelle Entwurf auf die Nachrichtenebene und bezieht dadurch keine Combined Fragments mit ein. Um das Mapping auf Nachrichtenebene durchzuführen, stehen die Elemente `<message>` und `<ref>` zur Verfügung, die beide durch ein *ref*-Attribut auf eine Nachrichten- bzw. Referenz-ID im Protokoll verweisen. Das Element `<ref>` wird benutzt, um das Mapping eines im Protokoll referenzierten Protokolls zu integrieren; alternativ können im Protokoll referen-

zierte Protokolle auch direkt im Mapping mit Hilfe der `<message>`-Elemente gemappt werden. Das `<message>`-Element legt über die Elemente `<handler>`, `<ontology>` und `<contentfilter>` die Handhabung und Constraints für das Verarbeiten einer Nachricht fest.

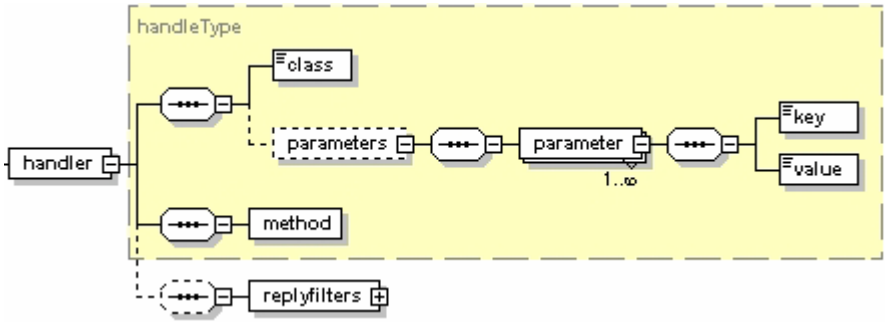


Abbildung 4.16: Handler-Schema

Die Spezifikation eines Handlers beginnt mit der Angabe der Klasse, in der der Handler spezifiziert wurde (Abbildung 4.16). Mit Hilfe des Elements `<parameters>` bzw. `<parameter>` können Parameter in Form von Schlüssel-Wert-Paaren an die Handlerinstanz übergeben werden. Das Element `<method>` dient zur Angabe der Methode, die das eigentliche Handling übernimmt. Das letzte Unterelement von `<handler>` ist das `<replyfilters>`-Element (Abbildung 4.17). Die Angabe von Reply-Filten ist nur erforderlich, falls der spezifizierte Handler diese benutzt. Der Reply-Filter wird durch eine Klassenangabe spezifiziert und kann optional entweder mit einer Parameterdefinition wie beim Handler oder durch eine Expression konfiguriert werden.

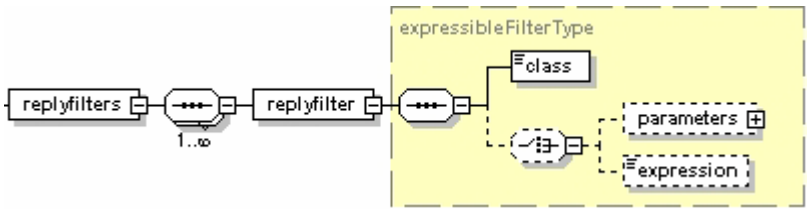


Abbildung 4.17: Replyfilter-Schema

Die Elemente `<ontology>` und `<contentfilter>` dienen als Constraint-Filter (Abbildung 4.18). <ontology> kann nur verwendet werden, falls im Protokoll keine Ontologien spezifiziert wurden und der Handler nur bestimmte Ontologien beherrscht. Alternativ zur Angabe einer Filterklasse, kann auch der Name der Ontology durch das <name>-Element spezifiziert werden. Für das Element `<contentfilter>` gilt bis auf das `<name>`-Element – das hier nicht vorhanden ist – prinzipiell das Gleiche wie für das `<ontology>`–Element, die Ausnahme ist: ein Contentfilter muss angegeben werden, wenn der Content im Protokoll als abstrakt definiert worden ist. In diesem Fall muss der Contentfilter nämlich sicherstellen, dass die abstrakte Spezifikation von der eingehenden Nachricht eingehalten wird.

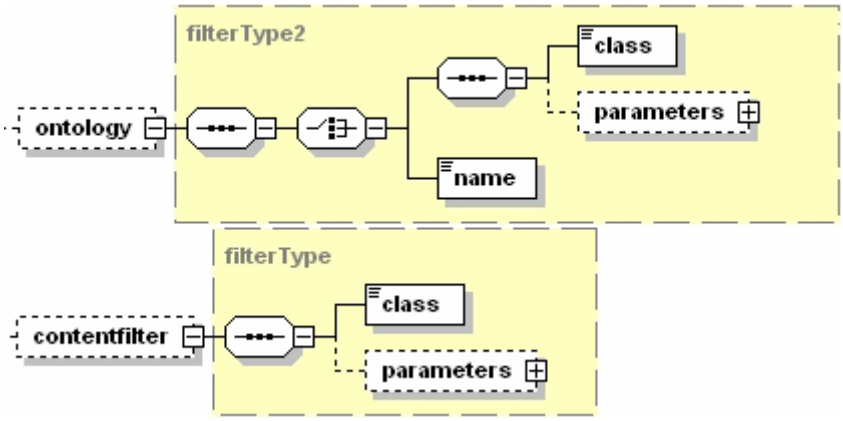


Abbildung 4.18: Ontology- und Contentfilter-Schema

4.2. Protokollimplementierungen

In diesem Abschnitt wird ein Beispiel vorgestellt, welches die Verwendung von Mappings, Handlern und Reply-Filtern in verschiedenen Protokollen und in parallel ablaufenden Protokollinstanzen zeigt.

Im Beispiel geht es darum, dass ein Agent die folgende Rechenaufgabe lösen soll: „calc 1 1 1 1 1“ – das heißt addieren der fünf Einsen. Leider kann ein Agent pro Anfrage nur dass Ergebnis von „calc x“, sowie das Ergebnis von „calc x_1“ + dem Ergebnis „calc x_2“ berechnen. Um an dass Ergebnis von „calc x_2“ zukommen, muss er dabei einen anderen Agenten fragen. Im ersten Fall also nach „calc 1 1 1 1“. Da der jeweils andere dies ebenfalls nicht berechnen kann, geht dieses so weiter bis ein Agent „calc 1“ als Aufgabe bekommt und das erste Mal ein Ergebnis liefert. Die Konversation soll dabei nach dem FIPA-Request-Protokoll ablaufen, wie in Abbildung 4.19 dargestellt. Dazu wird eine Spezifikation des Request-Protokoll in XML nötig, die in Abbildung 4.20 zu sehen ist..

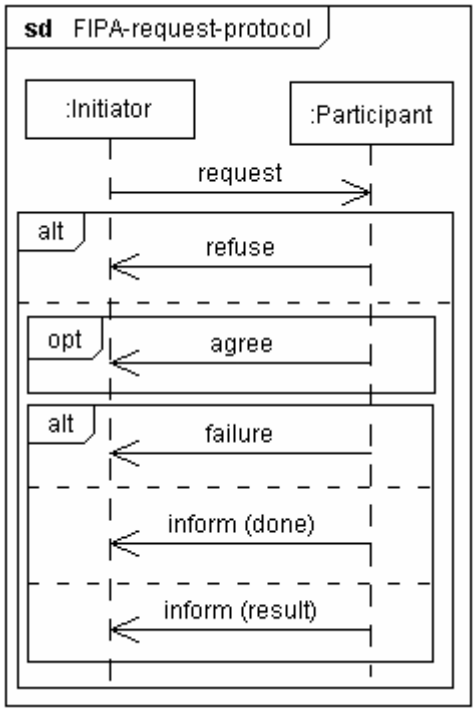


Abbildung 4.19: FIPA-Request-Protokoll

```
1. <sd [...] xsi:noNamespaceSchemaLocation="minimalSD.xsd" name="fipa-request">
2.   <lifelines>
3.       <lifeline id="LL1"><agentrole>Initiator</agentrole></lifeline>
4.       <lifeline id="LL2"><agentrole>Participant</agentrole></lifeline>
5.   </lifelines>
6.   <path>
7.       <message id="M1">
8.           <act>request</act>
9.           <sender ref="LL1"/><receiver ref="LL2"/>
10.      </message>
11.      <alternative id="ALT1">
12.          <path>
13.              <message id="M2">
15.                  <act>refuse</act>
16.                  <sender ref="LL2"/><receiver ref="LL1"/>
17.              </message>
18.          </path>
19.          <path>
20.              <option id="OPT1">
21.                  <path>
22.                      <message id="M3">
23.                          <act>agree</act>
24.                          <sender ref="LL2"/><receiver ref="LL1"/>
25.                      </message>
26.                  </path>
27.              </option>
28.              <alternative id="ALT2">
29.                  <path>
30.                      <message id="M4">
31.                          <act>failure</act>
32.                          <sender ref="LL2"/><receiver ref="LL1"/>
33.                      </message>
34.                  </path>
35.                  <path>
36.                      <message id="M5">
37.                          <act>inform</act>
38.                          <sender ref="LL2"/><receiver ref="LL1"/>
39.                      </message>
40.                  </path>
41.                  <path>
42.                      <message id="M6">
43.                          <act>inform</act>
44.                          <sender ref="LL2"/>
45.                          <receiver ref="LL1"/>
46.                          <content type="abstract">result</content>
47.                      </message>
48.                  </path>
49.              </alternative>
50.          </path>
51.      </alternative>
52.  </path>
53. </sd>
```

Abbildung 4.20: XML-Spezifikation des FIPA-Request-Protokolls

Der dabei durch parallel laufende Protokollinstanzen entstehende verschachtelte Nachrichtenverlauf ist in Abbildung 4.21 zu sehen. Farblich unter-

schiedlich markiert sind die einzelnen Protokollinstanzen, das heißt Protokolle der gleichen Konversation. Das versenden eines optionalen *agree* geschieht zufällig und wird nur bei der Konversation mit der Nummer 13 (grün) nicht vorgenommen.

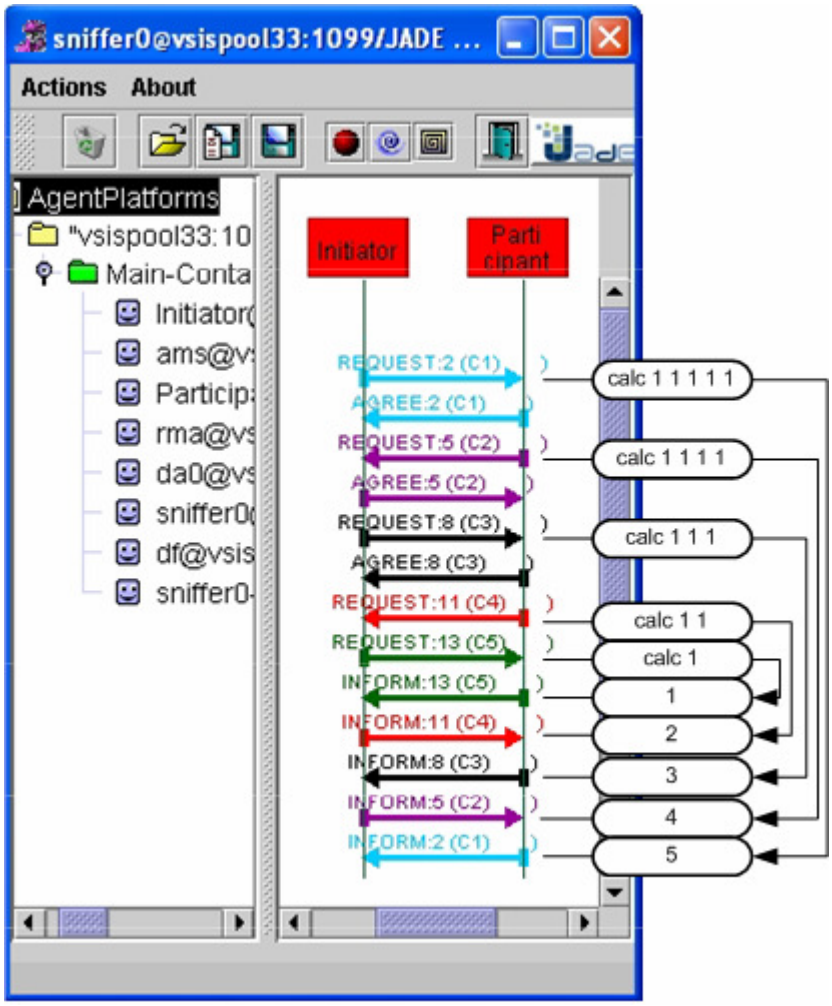


Abbildung 4.21: Nachrichtenverlauf des Beispiels, mit parallelen Protokollinstanzen

Der Ausschnitt der Mapping-Spezifikation für die Rolle des Participant des FIPA-Request-Protokolls wird in Abbildung 4.22 gezeigt. Die Request-Nachrichten werden im `fipeex.da.RequestProtocolHandler` von der Methode `request` verarbeitet (Zeilen 5 und 6). Dabei beschränken sich die Kommunikationsfähigkeiten der Agenten auf Request-Protokolle, bei denen der Inhalt der Request-Nachricht mit „calc“ beginnt (Zeile 30-38). Gemäß dem Request-Protokoll kann auf ein Request zunächst einmal mit einem

refuse oder einem *agree* geantwortet werden. Im Handler wird dazu eine abstrakte Nachricht an einen Reply-Filter weitergeleitet mit der Beschreibung „decision=negative“ bzw. „decition=positive“, wofür die Reply-Filter in Zeile 8 bzw. 12 zuständig sind. Mit der angegebenen Expression kann dann der `fipeex.da.DAExpressionReplyFilter` die abstrakte Nachricht vervollständigen und ein *failure* bzw. *agree* senden. Entsprechendes gilt auch für die restlichen Nachrichten.

```
 1. <lifeline name="Participant">
 2.   <mapping>
 3.     <message ref="M1">
 4.       <handler>
 5.         <class>fipeex.da.RequestProtocolHandler</class>
 6.         <method>request</method>
 7.         <replyfilters>
 8.          <replyfilter ref="M2"> <!-- refuse -->
 9.             <class>fipeex.da.DAExpressionReplyFilter</class>
10.             <expression>if($decision=="negative") { act="refuse"; reencode(content); }
                </expression>
11.          </replyfilter>
12.          <replyfilter ref="M3"><!-- agree -->
13.             <class>fipeex.da.DAExpressionReplyFilter</class>
14.             <expression>if($decision=="positive") { act="agree"; }</expression>
15.          </replyfilter>
16.          <replyfilter ref="M5"><!-- inform-done-->
17.             <class>fipeex.da.DAExpressionReplyFilter</class>
18.             <expression>if($process=="done") { act="inform"; }</expression>
19.          </replyfilter>
20.          <replyfilter ref="M6"><!-- inform-result-->
21.             <class>fipeex.da.DAExpressionReplyFilter</class>
22.             <expression>if($process=="processed") { act="inform"; }</expression>
23.          </replyfilter>
24.          <replyfilter ref="M4"><!-- failure -->
25.             <class>fipeex.da.DAExpressionReplyFilter</class>
26.             <expression>if(unknownKey("decition") || unknownKey("process")
                          || $process=="failed") { act="failure"; reencode(content); }
                </expression>
27.          </replyfilter>
28.        </replyfilters>
29.      </handler>
30.      <contentfilter>
31.        <class>fipee.filter.StringContentFilter</class>
32         <parameters>
33.          <parameter>
34.            <key>cis-starts-with</key><!-- cis = case in sensitive -->
35.            <value>calc</value>
36.          </parameter>
37.       </parameters>
38.      </contentfilter>
39.    </message>
40.  </mapping>
41. </lifeline>
```

Abbildung 4.22: Mapping für den Participant des FIPA-Request-Protokolls

```
1.  <lifeline name="Initiator">
2.      <mapping>
3.          <message ref="M2">
4.              <handler>
5.                  <class>fipeex.os.RequestProtocolHandler</class>
6.                  <method>refuse</method>
7.              </handler>
8.          </message>
9.          <message ref="M3">
10.             <handler>
11.                 <class>fipeex.os.RequestProtocolHandler</class>
12.                 <method>agree</method>
13.             </handler>
14.         </message>
15.         <message ref="M4">
16.             <handler>
17.                 <class>fipeex.os.RequestProtocolHandler</class>
18.                 <method>failure</method>
19.             </handler>
20.         </message>
21.         <message ref="M5">
22.             <handler>
23.                 <class>fipeex.os.RequestProtocolHandler</class>
24.                 <method>informDone</method>
25.             </handler>
26.         </message>
27.         <message ref="M6">
28.             <handler>
29.                 <class>fipeex.os.RequestProtocolHandler</class>
30.                 <method>informResult</method>
31.             </handler>
32.             <contentfilter>
33.                 <class>fipee.filter.StringContentFilter</class>
34.                 <parameters>
35.                     <parameter>
36.                         <key>min-length</key>
37.                         <value>1</value>
38.                     </parameter>
39.                 </parameters>
40.             </contentfilter>
41.         </message>
42.     </mapping>
43. </lifeline>
```

Abbildung 4.23: XML-Mapping für den Initiator des FIPA-Request-Protocol

In Abbildung 4.23 wird die Deklaration des Mappings für den Initiator des Request-Protokolls dargestellt. Zu beachten ist dabei der Contentfilter in den Zeilen 32-40, dieser beschreibt, dass der Inhalt einer Inform-Result-Nachricht nicht leer sein darf. Dies ist erforderlich, da in der Protokollspezifikation (Abbildung 4.20, Seite 70) in Zeile 46 der Content als abstrakt definiert ist, um das Ergebnis der Request-Anfrage zu beschreiben.

In Abbildung 4.24 wird ein Beispiel für eine Handler-Methode gezeigt, der Request-Handler des Participants. Um es einfach zu halten, wird von diesem

Handler keine negative Entscheidung an die Reply-Filter weitergereicht, also keine abstrakte Nachricht, die von einem Reply-Filter zu einer Refuse-Nachricht umgewandelt wird (siehe Abbildung 4.22, Zeilen 8-11), erzeugt.

```
1. public void request()
2. {
3.    ACLMessage acl = getMessage();
4.    if(doAgree()) {
5.       ACLMessage abstractReply = acl.createReply();
6.       KeyValueList description = new KeyValueList();
7.       description.add("decision","positive");
8.       postReply(abstractReply,description);
9.    }
10.   ACLMessage reply = acl.createReply();
11.   KeyValueList description = new KeyValueList();
12.   […] // Berechnungen
13.   if([…] /* Berechenbar ? */){
14.     description.add("process","processed");
15.     reply.setContent(result);
16.   } else {
17.     ACLMessage req =
18.        new ACLMessage(this.getAID(),"request");
19.     req.setProtocol("fipa-request");
20.     req.setReceiver(acl.getSender());
21.     req.setContent("calc" + remaining);
22.     ProtocolEvent pe =
23.         getExecutionControl().execProtocol(
24.             "http://[...]/request-mapping.xml",
25.              req, null);
26.     if(pe instanceof FailureEvent){
27.         description.add("process","failed");
28.     } else if(pe instanceof ResultEvent) {
29.         […] // berechne Ergebnis
30.          Belief b = getBeliefbase().getBelief("RESULT");
31.          b.modifyValue(result);
32.          reply.setContent(result);
33.          description.add("process","processed");
34.     }
35.   }
36.   postReply(reply,description);
37. }
```

Abbildung 4.24: Handler zur Behandlung des Requests.

Zum Erzeugen einer abstrakten Nachricht wird wie in Zeile 5 zunächst eine Antwortnachricht erzeugt, dieses sorgt dafür, dass die Sender- und Empfänger-Parameter der Nachricht richtig gesetzt sind. In einem so einfachen Pro-

tokoll wie dem FIPA-Request-Protokoll, bei dem sich immer nur Agenten unterhalten, wäre es zwar möglich, dass die Ausführungsumgebung dafür Sorge trägt, dass Sender und Empfänger richtig gesetzt werden, doch in Protokollen mit mehreren Teilnehmern, insbesondere mehreren der selben Rolle, kann dies nicht mehr von der Ausführungsumgebung korrekt durchgeführt werden. Ein Beispiel dafür ist das ContractNet-Protokoll: hier ist der Agent dafür zuständig, zu entscheiden, mit welchen Agenten er einen Kontrakt eingeht, also wem er welche Nachrichten sendet. Daher wurde darauf verzichtet, dass die Ausführungsumgebung für einfache Protokolle den Empfänger-Parameter der Nachricht ausfüllt.

Um die Nachricht an einen Reply-Filter übergeben zu können, benötigt dieser noch eine Beschreibung, die es ihm ermöglicht, eine vollständige Nachricht zu erzeugen. Bei der Agree-Nachricht geschieht dies in Zeile 7. Anschließend werden abstrakte Nachricht und Beschreibung durch Aufruf der Methode `postReply` (Zeile 8) an den Reply-Filter-Mechanismus übergeben.

Da Agenten bzw. Handler nicht in der Lage sind größere Zahlenreihen zu addieren, muss im Handler eine neue Protokollinstanz gestartet werden, dieses geschieht den Zeilen 22-25. Da innerhalb von FIPEE die Handler-Threads im Gegensatz zu Threaded-Plänen in Jadex echt nebenläufig laufen können, kann an dieser Stelle auf den Rückgabewert, das Ergebnis der Protokollausführung, gewartet werden. Das Ergebnis wird anschließend in die Beliefbase (Zeilen 30-32) eingetragen und mit dem Vermerk „process=processed“ an einen Reply-Filter weitergereicht (hier der Reply-Filter in den Zeilen 20-23 in Abbildung 4.22). Konnte das Protokoll nicht korrekt durchgeführt werden, so wird die Beschreibung für den Reply-Filter auf „process=failed“ gesetzt (Zeile 27) (dies wird vom Reply-Filter in den Zeilen 24-27 in Abbildung 4.22 behandelt).

Um Handler wiederverwenden zu können, würde es zur Dokumentation dieser gehören, die verwendeten Parameter und ihre Funktionen für die einzelnen abstrakten Nachrichten aufzuführen und zu erläutern.

```
1.  public void failure()
2.  {
3.      ACLMessage acl = getMessage();
4.      getData().setResult(acl.getContent(),RPData.STATE_FAILED);
5.  }
```

Abbildung 4.25: Failure-Handler des Request-Protocol-Initiators

Zum Abschluss des Beispiels noch einmal eine Handler-Methode für den Initiator (Abbildung 4.25). Es handelt sich dabei um einen Handler in einem

Endzustand des Protokolls. Die einzige Aufgabe dieses Handlers ist es, in den Datenspeicher des Protokolls das Ergebnis einzutragen und das Protokoll als „fehlgeschlagen“ zu markieren. Der Datenspeicher würde anschließend als Teil eines ResultEvents an den jeweiligen ResultHandler übergeben. So einfach dieser Handler auch ist, so vielfältig kann er eingesetzt werden und dass gilt für alle Handler des Request-Protokolls auf Initiatorseite. Das Besondere ist, dass die Nachrichtentypen, die der Initiator empfängt, in vielen anderen Protokollen ebenfalls vorkommen. Die Kombination von *refuse* und *agree* Nachrichten kommt zum Beispiel auch im FIPA-Recruiting-Protokoll [FIPA, 2002f] oder im FIPA-Brokering-Protokoll [FIPA, 2002g] vor. Die Kombination im Alternative-Combined-Fragment aus *failure*, *inform-done* und/oder *inform-result* kommt ebenfalls häufig vor, zum Beispiel im FIPA-ContractNet-Protokoll [FIPA, 2002h] oder im FIPA-Iterated-ContractNet-Protokoll [FIPA, 2002i]. Sie dienen dort, wie auch im Request-Protokoll, einfach nur zur Kenntnisnahme. Diese kleine Tatsache erlaubt es allerdings, die Handler des Request-Initiators auch in diesen und vielen anderen Protokollen wiederverwenden zu können. Bei entsprechender Programmierung ist dieses auch für Handler, die Nachrichten senden, möglich.

5. Zusammenfassung der Ergebnisse und Ausblick

5.1. Ergebnisse

In dieser Arbeit wurden zunächst die Grundlagen der Kommunikation in Agentensystemen eingeführt. Es wurden die Vorzüge nicht protokollorientierter Kommunikation mit denen der protokollorientierten Kommunikation verglichen. Dabei wurde festgestellt, dass durch die Kommunikation auf Protokollbasis Vorteile entstehen, die bei steigender Zahl der Kommunikationspartner gegenüber denen nicht protokollorientierter Systeme überwiegen. Dieses sind Vorteile wie zum Beispiel ein verringerter Aufwand beim Parsen der Nachrichten oder ein festgelegter geregelter Nachrichtenaustausch, der das Vorkommen von unverständlichen oder fehlerhaften Nachrichten minimiert.

Um die Vorteile von Protokollen auch in offenen Systemen wie dem Internet nutzen zu können, wurde FIPA Agent UML (AUML) vorgestellt. Durch die Spezifikation von Protokollen in AUML kann die Verständlichkeit von Protokollabläufen erhöht werden und dadurch Fehler bei der Implementierung der Kommunikationsfähigkeiten eines Agenten vermieden werden. Durch einen einheitlichen Standard zur Spezifikation von Protokollen können diese öffentlich zugänglich gemacht werden.

Im Vorfeld der Entwicklung wurden verschiedene protokollunterstützende Ansätze untersucht, um Anforderungen an eine Protokollunterstützung zu gewinnen. Als Ergebnis dieser Untersuchung kam heraus, dass ein Tool zur Ausführung protokollspezifischen Code am geeignetsten ist und eine Protokollspezifikation benötigt wird, die unabhängig von diesem ist.

In Anlehnung an den AUML-Entwurf des Modeling Technical Committee der FIPA wurde dazu ein Entwurf für eine Spezifikation von Protokollen in XML ausgearbeitet. Es ist allerdings nicht das Ziel dieses Entwurfs, einen Standard vorzuschlagen, sondern nur die protokollspezifische Grundlage für die entwickelte Ausführungsumgebung darzustellen. Dennoch ist der Entwurf dazu geeignet, dass Protokolle im Internet veröffentlicht werden und dadurch auch in anderen Agentensystemen verwendet werden können.

Durch die Protokoll-Ausführungsumgebung kann einem Agenten ein signifikanter Teil der Arbeit protokollorientierter Kommunikation abgenommen

werden. Eine Ausführungsumgebung übernimmt die Aufgabe der Verwaltung der Protokollsitzung und des Ablaufs. Hierzu zählen die Überwachung von Nebenbedingungen, der Schutz des Agenten vor fehlerhaften Nachrichten anderer Agenten und die Sicherstellung, dass der eigene Agent protokollkonforme Nachrichten sendet. Die Ausführung eines Protokolls über eine Protokoll-Ausführungsumgebung ist damit für einen Agenten transparent und kann durch einen einzigen Methodenaufruf ausgeführt werden.

Zum Erreichen der Zielsetzung Trennung von Protokollfluss und Implementierung wurde das Konzept der Handler eingeführt. Handler sind Programmstücke, zumeist Methoden, die von der Protokoll-Ausführungsumgebung aufgerufen werden, um eingehende Nachrichten zu verarbeiten und Antworten oder andere Nachrichten darauf zu erzeugen. Bei der Entwicklung von Handlern wird, im Gegensatz zum verbreiteten Vorgehen bei der Protokollentwicklung, nicht auf ein spezielles Protokoll hingearbeitet, sondern vielmehr auf ein Verhalten, welches für verschiedene Situationen in verschiedenen Protokollen Wiederverwendung finden kann. Die von Handlern erzeugten Nachrichten lassen sich in zwei Gruppen aufteilen, zum einen solche, die vollständig sind, d.h. alle Parameter der Nachricht sind gesetzt, zum anderen abstrakte Nachrichten. Abstrakte Nachrichten sind unvollständige Nachrichten, zum Beispiel fehlt bei ihnen der Konversationsakt. Zu diesen abstrakten Nachrichten gibt der Handler eine Beschreibung, die den Kontext, aus dem die Nachricht gesendet wird, beschreibt. Die Wiederverwendung der Handler wird insbesondere durch diese Signatur eines Handlers, also aus der Deklaration der Parameter einer abstrakten Nachricht, erreicht, die in den Reply-Filtern ihre Anwendung finden.

Reply-Filter sind Teil des Handlerkonzepts und sind den Handlern nachgeordnet. Während Handlern die Hauptaufgabe zur Bearbeitung überlassen wird, dienen Reply-Filter hauptsächlich zur Vervollständigung abstrakter Nachrichten, die von einem Handler erzeugt wurden. Dabei sind mehrere Reply-Filter einem Handler zugeordnet, die entsprechend der zur Nachricht zugehörigen Beschreibung ausgewählt werden.

Unter Verwendung einer Zuordnung von Handlern und Reply-Filtern in einem so genannten Mapping lassen sich zu Protokollen für einen Agenten schnell verschiedene Verhaltensweisen implementieren. Durch das Mapping wird zudem erreicht, dass das Protokoll selbst nicht in Verbindung mit dem Programmcode steht und somit eher einem Template gleicht, welches durch das Mapping zur Protokollimplementierung wird. Zudem können durch Constraints, die an das Mapping gebunden werden, gleichzeitig verschiedene Protokollimplementierungen von einer Ausführungsumgebung bereit gehalten werden, die dann entsprechend der erfüllten Bedingungen ausgewählt werden.

Um in der Hauptsache das Handlerkonzept zu prüfen, ist der gegenwärtige Prototyp im Vergleich zu den oben genannten Möglichkeiten beschränkt. Er eignet sich zurzeit nur für Protokolle, in denen Agenten paarweise miteinander kommunizieren. Auf eine Überwachung von zeitlichen und nicht zeitlichen Constraints wurde ebenfalls verzichtet. Für eine Implementierung von zeitlichen Constraints wurde ein Zeitkontext für eine Lifeline angedacht, auf den die Parser oder Evaluatoren für die jeweilige verwendete Sprache zugreifen können.

Durch den gewählten Ansatz einer plattformunabhängigen Ausführungsumgebung ist es möglich, plattformunabhängige Handler zu programmieren und somit eine Portierung der gesamten Protokollausführung auf andere Agentensysteme vorzunehmen. Allerdings erlaubt der plattformunabhängige Ansatz den Handlern nicht, auf die gesamt Palette der Fähigkeiten eines Agenten zuzugreifen. Bei einem Verzicht auf die Plattformunabhängigkeit der Handler kann dieses jedoch reduziert werden.

Die Beispielimplementierung des FIPA-Request-Protokolls hat gezeigt, dass die Verwaltung mehrerer nebenläufiger Protokollinstanzen gewährleistet werden kann und die notwendigen Abläufe zu deren Abarbeitung für den Agenten verborgen bleiben. Zudem wird durch den entwickelten Prototyp die FIPA Interaction Protocol Execution Engine (FIPEE) sichergestellt, dass die gesendeten Nachrichten ihrer Spezifikation im Protokoll entsprechen. Durch die Spezifikation des FIPA-Request-Protokolls in XML wurde gezeigt, wie ein einfaches und verständliches Format, das sich zum Austausch eignet, benutzt werden kann.

Mit den Spezifikationen der Mappings für Initiator und Participant des FIPA-Request-Protokolls ist gezeigt worden, dass Reply-Filter in Verbindung mit einem Handler eine einfache Anpassung der Handlerausgaben ermöglichen. Durch die Programmierung von Handlern, die auf einen speziellen Protokollkontext zugeschnitten sind, kann zudem die Wiederverwendbarkeit des Handlers in anderen Kontexten erreicht werden.

5.2. Ausblick

Im Abschnitt „Verwandte Arbeiten“ (Kapitel 3.1. auf Seite 35) wurden unter anderem Werkzeuge vorgestellt, die einen Agenten in der Protokollausführung unterstützen, ohne dass dieser seine Autonomie verliert. Um zum Beispiel bei einem BDI-Agenten, der die Ausführungsumgebung benutzt, weiterhin dessen Deliberationsprozesse nutzen zu können, könnte man Goals bei der Protokollspezifikation berücksichtigen. Diese Goals werden dann anstelle der Handler im Mapping angegeben. Goals alternativ zu Handlern zu nutzen wurde im Rahmen dieser Arbeit ebenfalls kurz untersucht. Diese Untersuchung hat ergeben, dass aufgrund der verschiedenen Goalkonzepte

der einzelnen auf dem Markt befindlichen BDI-Agentensysteme keine einheitliche Abbildung für ein plattformunabhängiges System möglich ist. Um dieses zu ermöglichen, hätte eine Obermenge aller Goalarten gebildet werden müssen, für die anschließend eine adäquate Abbildung mit Hilfe von Wrappern auf die spezifischen Goals der jeweiligen Plattform hätte vorgenommen werden müssen. Da im Allgemeinen davon ausgegangen werden kann, dass eine adäquate Abbildung nicht durchführbar ist, wurde darauf verzichtet.

Um dennoch auf Goals nicht verzichten zu müssen, wäre für eine Weiterentwicklung eine Trennung vom plattformunabhängigen Ansatz sinnvoll und notwendig. Für diese Trennung müsste die Spezifikation der Mapping-Schemata um die Möglichkeit, beliebige Goals als Handler anzugeben, erweitert werden. Der Trennpunkt wäre damit der Parser, der die Protokolle und Mappings in interne Modelle überführt. Der Parser wäre damit weiterhin eine plattformunabhängige Komponente, während die Ausführungsumgebung, Handler und Goals ohne interne Abbildungsprozesse direkt auf die Funktionen des Agenten zugreifen können.

Weitere Möglichkeiten zur Erweiterung des Systems entstehen im Zusammenhang mit einem Entwicklungswerkzeug, das die Protokollentwicklung vom Protokolldesign bis zur Implementierung der Handler begleiten kann. Mit Hilfe eines solchen Tools kann eine automatische oder semiautomatische Erstellung der Mappings vorgenommen werden. Grundlage dafür wäre ein Vorgehensmodell, bei dem die Handler und ihre Signaturbeschreibungen in einer Datenbank gespeichert sind, sodass das Entwicklungswerkzeug anhand des Protokolls die zu verwendenden Handler vorschlagen kann.

Da diese Auswahl einem Deliberationsprozess gleichkommt, wäre es in diesem Zusammenhang, interessant einen Deliberationsprozess für Handler, ähnlich wie bei Plänen, zur Verfügung zu haben. In der gegenwärtigen Implementierung ist die Entscheidung für eine bestimmte Verhaltensweise durch Constraints für die erste Nachricht an eine Rolle im Mapping vorgegeben. Durch Verwendung einer Deliberation auf Handlerebene könnte dann, wenn keiner der angegebenen Reply-Filter sich anwenden lässt, ein anderer Handler ausprobiert werden. Für die Auswahl der Handler wären zwei verschiedene Szenarien denkbar: zum einen könnten im Mapping für das zu behandelnde Ereignis mehrere Handler zur Auswahl gestellt werden. Es wäre aber auch denkbar, dass ähnlich einer Plan-Library eine „Handler-Library“ existiert, aus der die Handler aufgrund ihrer passenden Schnittstellensignatur ausgewählt und angewendet werden.

Zusammenfassend kann gesagt werden, dass die Zusammenführung des Handlerkonzepts der Ausführungumgebung und des BDI-Konzepts ein interessantes Gebiet ist, das die Möglichkeiten einer Ausführungsumgebung bereichern kann.

Ein weiterer interessanter Punkt wäre, die zweite neben Jadex am Fachbereich Informatik der Universität Hamburg existierende Agentenplattform Mulan anzubinden. Der Vorteil, der sich daraus für beide Systeme ergeben würde, wäre – da es sich bei FIPEE um ein plattformunabhängiges Tool handelt – dass die Handler die für Jadex entwickelt werden auch auf Mulan eingesetzt werden können, und umgekehrt. Des Weiteren würden dadurch auch die Protokolle und deren Mappings auf beiden Systemen eingesetzt werden können. Um Mulan anzubinden, müssten entsprechend die Interfaces, die in Kapitel 4.1.3. auf Seite 55 aufgeführt sind, implementiert werden. Für SendPort und MessageReceiver würden sich bei Mulan zwei separate Netze eignen, wobei eine MessageReceiver-Instanz für jedes auszuführende Protokoll angebracht wäre. Um Protokolle auch zur Laufzeit hinzufügen und entfernen zu können, wäre es notwendig solche Netze auch dynamisch für ein Protokoll registrieren zu können.

6. Abbildungsverzeichnis

7. Literaturverzeichnis

[ADK] Agent Development Kit (ADK), Tryllian Solutions B.V., http://www.tryllian.com/technology/product1.html (01.2005)

[Alberti et al., 2003] Marco Albert, Marco Gavanelli, Paola Mello, Evelina Lamma, Paolo Torroni; "Specification and Verification of Agent Interactions using Social Integrity Constraints", In W. van der Hoek, A. Lomuscio, E. de Vink, M. Wooldridge, Herausgeber, Electronic Notes in Theoretical Computer Science, Vol. 85 (2), Elsevier Science Publishers, 2003.

[Alberti et al., 2004] Marco Albert, Marco Gavanelli, Paola Mello, Federico Chesani, Evelina Lamma, Paolo Torroni; "Compliance Verification of Agent Interation: a Logic-based Software Tool", Technical Report, DEIS-LIA-004-03, LIA Series no. 71, University of Bologna, Italien, 2004.

[Austin, 1962] John L Austin; „How to Do Things with Words", Oxford University Press, Oxford, UK, 1962.

[AFIT] agentTool, AFIT Artificial Intelligence Laboratory, Air Force Institute of Technology, http://en.afit.af.mil/ai/agentool.htm (01.2005)

[Bauer, Müller, Odell, 2001] Bernhard Bauer, Jörg P. Müller und James Odell; „Agent UML: A Formalism for Specifying Multiagent Interaction". In Paolo Conacarini und Michael Wooldridge Herausgeber, „Agent-Oriented Software Engineering", Springer-Verlag, Seite 91-103, 2001.

[Bellifemine et al., 2000] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa; „Developing multi-agent systems with JADE". In Cristiano Castelfranchi, Yves Lespérance, Herausgeber, Lecture Notes in Computer Science; Vol. 1986, Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages (ATAL 2000), Seiten 89-103 , Springer Verlag, London, Großbritannien, Juli 2000.

[Bernon et al., 2002] Carole Bernon, Marie-Pierre Gleizes, Sylvain Peyruqueou and Gauthier Picard; "ADELFE, a Methodology for Adaptive Multi-Agent Systems Engineering"; In Third International Workshop Engineering Societies in the Agents World (ESAW-2002), 16-17 September 2002, Madrid.

[Bratman, 1987] Micheal E. Bratman; “Intentions, Plans, and Practical Reason”. Harvard University Press, Massachusetts, and London, England, 1987

[Bratman, Pollack und Israel, 1988] Michael E. Bratman, Martha E. Pollack, David J. Israel; “Plans and resource-bounded practical reasoning”, In Comupational Intelligence, Volume 4, Band 4: S. 349-355, 1988

[Bush, Cranefield und Purvis, 2001] Geoff Bush, Stephen Cranefield, Martin Purvis; “The Styx Agent Methodology”. In The Information Science Discussion Paper Series, Nummer 2001/02, Januar 2001.

[Cabac, 2003] Lawrence Cabac: “Modeling Agent Interaction Protocols with AUML Diagrams and Petri Nets”, Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Dezember 2003.

[Caire et al., 2001] Giovanni Caire, Wim Coulier, Francisco Garijo, Jorge Gomez, Juan Pavon, Francisco Leal, Paulo Chainho, Paul Kearney, Jamie Stark, Richard Evans, Philippe Massonet; ”Agent oriented analysis using MESSAGE/UML”. In Agent-Oriented Software Engineering (AOSE), 2001, S. 119-135.

[Cervenka, 2003] Radovan Cervenka: „Modeling Notation Source – MESSAGE“, Version vom 3. März 2003;
http://auml.org/auml/documents/MESSAGE.pdf, FIPA 2003

[Cervenka, 2003b] Radovan Cervenka: „Modeling Notation Source – Tropos“, Version vom 8. März 2003;
http://auml.org/auml/documents/Tropos.doc, FIPA 2003

[Cervenka, 2003c] Radovan Cervenka: „Modeling Notation Source – Prometheus “, Version vom 2. April 2003;
http://auml.org/auml/documents/Prometheus030402.pdf, FIPA 2003

[Chiab-Draa and Dignum, 2002] Brahim Chiab-Draa, Frank Dignum; “Trends in Agent Communication Language”, In Randy Goebel, Russell Greiner, and Dekang Lin, Herausgeber, Computational Intelligence, Volume 2, Number 5, Seiten 89-101, 2002.

[Cossentino und Potts, 2002] Massimo Cossentino, Colin Potts; “A CASE tool supported methodology for the design of multi-agent systems”, The 2002 International Conference on Software Engineering Research and Practice (SERP’02) – Juni 2002, Las Vegas / USA

[Cossentino und Sabatucci, 2003] Massimo Cossentino, Luca Sabatucci: “Modeling Notation Source – PASSI”, Version vom 28. April 2003; http://auml.org/auml/documents/PASSI.doc, FIPA 2003

[Ehrler, 2003] Lars Ehrler: “Declarative graphical description of Interaction Protocols for Mulit-agent Systems”. Dissertation for a prostgraduate diploma of arts. Department of Information Science, University of Otago, Neuseeland, 2003; http://larsehrler.de/veroeff/2003Paul.pdf (01.2005)

[Ehrler und Cranefield, 2004] Lars Ehrler, Stephen Cranefield, “Executing Agent UML diagrams”, In Nicholas R. Jennings, Carles Sierra, Liz Sonnenberg, and Milind Tambe, Herausgeber, Proceedings of The Third International Joint Conference on Autonomous Agents & Multi Agent Systems, AAMAS 2004, Seiten 906–913, New York, 2004.

[Finn et al., 1994] Tim Finn, Jay Weber, Gio Wiederhold, Michael Genesereth, Riechard Fritzson, Donald McKay, James McGuire, Richard Pelavin, Stuart Shapiro und Chris Beck; “Specification of the KQML Agent-Communication Language”, “The DARPA Knowledge Sharing Initiative External Interfaces Working Group” Draft, Februar 1999.

[Freire und Botelho, 2002] Joaquim Freire, Luis Botelho: „Executing explicitly represented protocols“, L.M. Proc. of the Workshop Challenges in Open Agent Systems of the 1st International Joint Conference on Atonomous Agents and Multi Agent Systems (AAMAS 2002). Juli, 2002.

[Dinkloh und Nimis, 2003] Martin Dinkloh, Jens Nimis: “A Tool for Integrated Design and Implementation of Conversations in Multiagent Systems”. 1st International Workshop on Programming Multiagent Systems languages, frameworks, techniques and tools (ProMAS 2003), gehalten bei der 2nd International Conference on Autonomous Agents & Multiagent Systems. Melbourne, Australien, Juli 2003.

[Dori, 2002] Dov Dori; „Object-Process Methodology – A Holistic Systems Paradigm”, Springer, Heidelberg (2002).

[Dori et al., 2003] Dov Dori, Marc-Philippe Huget, Iris Reinhartz-Berger, Onn Shehory, Arnon Strum: "Modeling-Notation Stource: OPM/MAS", Version vom 12. März 2003; http://auml.org.auml/documents/OPM.pdf, FIPA 2003

[Ferber und Gutknecht, 1998] Jacques Ferber, Olivier Gutknecht; „A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems". In Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98), S.128-135, Paris / Frankreich (1998).

[Ferber, 1999] Jacques Ferber; "Multi-Agent Systems: An Introduction to Distributes Artificial Intelligence". Addison Wesley Longman, Harlow / Großbritannien (1999).

[FIPA, 2000] Foundation for Intelligent Physical Agents; "FIPA Abstract Architecture Specification", FIPA SC00001(L), Genf / Schweiz (2000).

[FIPA, 2002] Foundation for Intelligent Physical Agents; "FIPA Communicative Act Library Specification", FIPA SC00037(J), Genf / Schweiz (2002)

[FIPA, 2002b] Foundation for Intelligent Physical Agents; "FIPA ACL Message Structure Specification", FIPA SC00061(G), Genf / Schweiz (2002)

[FIPA, 2002c] Foundation for Intelligent Physical Agents; "FIPA ACL Message Representation in Bit-Efficient Encoding Specification", FIPA SC00069(G), Genf / Schweiz (2002)

[FIPA, 2002d] Foundation for Intelligent Physical Agents; "FIPA ACL Message Representation in String Specification", FIPA SC00070 (I), Genf / Schweiz (2002)

[FIPA, 2002e] Foundation for Intelligent Physical Agents; "FIPA ACL Message Representation in XML Specification", FIPA SC00071(E), Genf / Schweiz (2002)

[FIPA, 2002f] Foundation for Intelligent Physical Agents; "FIPA Recruiting Interaction Protocol Specification", FIPA SC00034(H), Genf / Schweiz (2002)

[FIPA, 2002g] Foundation for Intelligent Physical Agents; “FIPA Brokering Interaction Protocol Specification”, FIPA SC00033(H), Genf / Schweiz (2002)

[FIPA, 2002h] Foundation for Intelligent Physical Agents; “FIPA Contract Net Interaction Protocol Specification”, FIPA SC00029(H), Genf / Schweiz (2002)

[FIPA, 2002i] Foundation for Intelligent Physical Agents; “FIPA Iterated Contract Net Interaction Protocol Specification”, FIPA SC00030(H), Genf / Schweiz (2002)

[FIPA, 2003] Foundation for Intelligent Physical Agents; “FIPA Modeling: Interations Diagrams” Working Draft, Version 2003-07-02, Genf / Schweiz (2003)

[Georgeff et al., 1999] Micheal Georgeff, Barney Pell, Martha Pollack, Milind Tambe, Micheal Wooldridge; “The Belief-Desire-Intention Model of Agency”. In J. Müller, M. Singh und A. Rao, Herausgeber, “Proceedings of the 5th international Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages” (ATAL-98), Seiten 1-10, Springer, 1999

[Griss et al., 2002] Martin L. Griss, Steven Fonseca, Dick Cowan, Robert Kessler; “Using UML State Machine Models for More Precise and Flexible JADE Agent Behaviors”, In Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AOSE02), Bologna, Italien, 2002.

[Hanson et al., 2002] James E. Hanson, Prabir Nandi, David Levine; “Conversation-enabled Web Services for Agents and e-Business”, In Proceedings of the International Conference on Internet Computing (IC-02) CSREA Press, 2002.

[Huget, 2003] Marc-Philippe Huget; Antwort von Marc-Philippe Huget zum “FIPA Modeling: Interations Diagrams” Working Draft, Version 2003-03-03, http://auml.org/auml/documents/ID_03-03-03-MPH1.doc (9.3.2003)

[Huget, 2003b] Marc-Philippe Huget; “Modeling-Notation Source – ADELPHE” [sic!], Version vom 13. März 2003;
http://auml.org/auml/documents/ADELPHE.pdf [sic!], FIPA 2003

[ILMES] ILMES – Internet-Lexikon der Methoden der empirischen Sozialforschung, Wolfgang Ludwig-Mayerhofer, Universität Siegen, http://www.lrz-muenchen.de/~wlm/ilmes.htm, (09.2004)

[Köhler, Moldt, Rölke, 2001] Michael Möhler, Heiko Rölke, Daniel Moldt; „Modelling the Structure and Behaviour of Petri Net Agents." In Proceedings of the 22nd Conference on Application and Theory of Petri Nets, Seiten 224-241, Juni 2001.

[Kowalski und Sergot, 1986] R. Kowalski, M.J. Sergot; „A logic-based calculus of events", New Generation Computing, Volume 4(1), Seite 67-95, 1986.

[Lobrou, Finin, Peng, 1999] Yannis Labrou, Tim Finn und Yun Peng: „Agent Communication Languages: The Current Landscape", In IEEE Intelligent Systems, Volume 14 (2), Seiten 45-52, 1999.

[Levy, 2003] Renato Levy; „Modelling-Notation Source – Alaadin/MADKit" [sic!], Version vom 3. März 2003, http://auml.org/auml/documents/MADKit.pdf, FIPA 2003

[Mylopoulos, Kolp und Castro, 2001] John Mylopoulos, Manuel Kolp, Jaelson Castro; „UML for Agent-Oriented Software Development: The Tropos Proposal". In Lecture Notes in Computer Science Volume 2185, S. 422ff, 2001.

[OMG, 2003] Object Management Group; „UML 2.0 Superstructure Final Adopted Specification", 2003

[Odell, 2003] James Odell: "Modeling-Notation Source: Bric" Version vom 27. Februar 2003; http://auml.org/auml/documents/Bric.pdf, FIPA 2003

[Odell, Parunak und Bauer 2000] James Odell, H. Van Dyke Parunak, Bernhard Bauer; „Extending UML for Agents"; AOIS Workshop at AAAI 2000, 2000.

[Padgham und Winikoff, 2002a] Lin Padgham, Michael Winikoff; "Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents". In proceedings of the workshop on Agent-oriented methodologies at OOPSLA 2002. 4. November, 2002, Seattle.

[Padgham und Winikoff, 2002b] Lin Padgham, Michael Winikoff; „Prometheus: A Methodology for Developing Intelligent Agents". In proceedings of the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS'02.

[Pokahr, Braubach, Lamersdorf, 2003] Alexander Pokahr, Lars Braubach, Winfried Lamersdorf: „Jadex: Implementing a BDI-Infrastructure for JADE Agents", in: EXP - In Search of Innovation (Special Issue on JADE), Vol 3, Nr. 3, Telecom Italia Lab, Seiten. 76-85, Turin, Italien, September 2003

[Pokahr und Braubach, 2004] Alexander Pokahr, Lars Braubach: "Jadex User Guide", Release 0.921, Distributed Systems Group, University of Hamburg, Deutschland, Juli 2004.

[Rao und Georgeff, 1991] Anand S. Rao, Michael P. Georgeff; „Modeling Rational Agents within a BDI-Architecture". In J. Allen, R. Fikes und E. Sandewall, Herausgeber, "Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning" (KR91), Seiten 473-484, Morgan Kaufmann Publishers, Inc., San Mateo, California (1991)

[Searle, 1969] John R. Searle; "Speech Acts: An Essay in Philosophy of Language" Cambridge University Press, 1996, (In Deutsch: "Sprechakte: Ein sprachphilosophischer Essay", Suhrkamp Verlag, 1971)

[Shoham, 1990] Yoav Shoham; "Agent-Oriented Programming", Department of Computer Science, Stanford University, Stanford, Kalifornien, Report No. STAN-CS-90-1335, Oktober 1990.

[Stathis et al. 2004] Kostas Stathis, Antonis C. Kakas, Wenjin Lu, Neophytos Demetriou, Ulle Endriss, Andrea Bracciali; "PROSOCS : a platform for programming software agents in computational logic", In Jörg Müller, Paolo Petter, Herausgeber, Proceedings of the 4th International Symposium from Agent Theories to Agent Implementations (AT2AI-4), Seiten 523-528, Wien, Österreich, April 2004.

[UML 2003] Object Management Group; "UML 2.0 superstructure Final Adopted specification", 2. August 2003, Needham, USA

[Wagner, 2002] G. Wagner; “The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior”. In Information Systems 28:5, 2003, S. 475-504.

[Walton, Robertson, 2002] Christopher Walton und David Robertson; “Flexible Multi-Agent Protocols”; Technical Report EDI-INF-RR-0164, University of Edinburgh, 2002.

[Wooldridge, 2002] Michael Wooldridge, “An Introduction to MultiAgent Systems”, 22. März 2002, John Wiley and Sons Ltd., West Sussex, England, ISBN: 047149691X.

[Wooldridge und Jennings, 1995] Michael Wooldridge, Nicholas R. Jennings; “Intelligent Agents: Theory and Practice”; In Knowledge Engineering Review, 10(2), S. 115-152, Cambridge University Press, Juni 1995.

[Wooldridge, Jennings und Kinny, 2000] Michael Wooldridge, Nicholas R. Jennings, David Kinny; “The Gaia Methodology for Agent-Oriented Analysis and Design”; In Autonomous Agents and Multi-Agent Systems, Volume 3, S. 285-312, Kluwer Academic Publishers, Niederlande, 2000.

[Yolum und Singh, 2002] Pinar Yolum, Munindar P. Singh; “Flexible Protocol Specification and Execution: Apply Event Calculus Planning using Commitments”, In Proceedings of the first international joint conference on Autonomous agents and multiagent systems (AAMAS’02), Seiten 527-534, Bologna, Italien, Juni 2002.

[Zeus] Zeus, British Telecommunications plc., BT Exact, http://more.btexact.com/projects/agents.htm (01.2005)

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebener Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit der Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den

Alexander Scheibe