Diplomarbeit

Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten

Michael Duvigneau

Betreuer:
Dr. Daniel Moldt
Prof. Dr. Arno Rolf

Universität Hamburg, Fachbereich Informatik

Inhaltsverzeichnis

1 Einleitung 7

2 Technische Bausteine 10
2.1 Java 10
2.2 UML 13
2.3 Referenznetze 15

3 Multiagentensysteme 22
3.1 Begriffe 22
3.1.1 Agent und Umwelt 23
3.1.2 Multiagentensystem 25
3.1.3 Ort, Plattform und Mobilität 26
3.1.4 Architekturen 27
3.2 Multiagentennetze (Mulan) 28
3.2.1 Gesamtmodell 29
3.2.2 Agentenaufbau 31
3.2.3 Agentenverhalten: Protokolle 33
3.2.4 Agentenumgebung: Plattform 34
3.2.5 Ausführbarkeit 35

4 FIPA-Standards 36
4.1 Überblick 37
4.2 Die Agentenkommunikationssprache 41
4.2.1 Kommunikative Akte 41
4.2.2 Interaktionsprotokolle 45
4.2.3 Inhaltssprachen 47
4.2.4 Ontologien 49
4.2.5 Struktur 50
4.2.6 Repräsentierung 51
4.3 Abstrakte Architektur 53
4.3.1 Grundbegriffe 54
4.3.2 Nachrichtentransport 56
4.3.3 Verzeichnisdienst 58
4.4 FIPA-2000-Architektur 60
4.4.1 Nachrichtentransport 61

4.4.2 Verwaltung und Verzeichnisdienste . . . 62
4.5 Nicht standardisiert . . . 64
4.6 Implementierungen . . . 65

5 Realisierte Plattform **68**
5.1 Plattformarchitektur . . . 69
5.1.1 Interne Schnittstelle zum Agenten . . . 72
5.1.2 Schnittstelle nach außen . . . 79
5.1.3 Nachrichten . . . 82
5.2 Implementierung . . . 84
5.2.1 Standard-Agent . . . 86
5.2.2 Verwaltungsdienste . . . 90
5.2.3 Nachrichtentransport . . . 97
5.2.4 Interne Nachrichtenrepräsentierung . . . 119
5.2.5 Repräsentierungskonversion . . . 123
5.3 Bewertung . . . 126
5.3.1 Agenten . . . 126
5.3.2 Mobilität . . . 128
5.3.3 Sicherheit . . . 130
5.3.4 Nebenläufigkeit . . . 131
5.3.5 Erweiterbarkeit . . . 133
5.3.6 Praxistauglichkeit . . . 134
5.3.7 CAPA und FIPA . . . 136
5.3.8 CAPA und MULAN . . . 137

6 Eine Feldstudie: Das Siedler-Spiel **139**
6.1 Das Spiel . . . 139
6.2 Die Agenten . . . 140
6.3 Kommunikation . . . 142
6.4 Einsatz der Plattform . . . 143

7 Ergebnis und Ausblick **146**

Glossar **148**

Literaturverzeichnis **159**

Eidesstattliche Erklärung **165**

Abbildungsverzeichnis

2.1 Elemente eines Klassendiagramms 13
2.2 Elemente eines Sequenzdiagramms 15
2.3 Kantenarten 16
2.4 Einige Anschriften 16
2.5 Instanzen und Kanäle 18
2.6 Nebenläufige Transitionen in einem B/E-Netz 19
2.7 Faltung 19
2.8 Das Netz aus Abbildung 2.6 als gefaltetes, gefärbtes Netz 20
2.9 Das Netz aus Abbildung 2.6 mit flexiblen Kanten gefaltet 20

3.1 Überblick über die Mulan-Architektur 30
3.2 Agenten-Grundmuster 32
3.3 Ein verfeinertes Mulan-Agentennetz 32

4.1 Thematische Einteilung der FIPA-Spezifikationen 38
4.2 Das FIPA-Request-Interaktionsprotokoll 46
4.3 Elemente einer Transportnachricht 56
4.4 Nachrichtentransportelemente 57
4.5 Elemente eines Verzeichniseintrags 59
4.6 Referenzmodell einer FIPA2000-Plattform 61
4.7 Der Lebenszyklus eines FIPA2000-Agenten 64

5.1 Elemente der CAPA-Plattform 72
5.2 Plattforminterne Kommunikation im Mulan-Modell 73
5.3 Plattformübergreifende Kommunikation im Mulan-Modell 73
5.4 Java-Interface zum MTS 74
5.5 Asynchrone Transportdienstimplementierung 75
5.6 Referenzrichtungswechsel in den Netzen 76
5.7 Referenzrichtungswechsel im Java-Interface 78
5.8 Der `JavaAgentAdapter` verbindet Referenznetz- mit Java-Code 80
5.9 Die Java-Package-Struktur von CAPA 85
5.10 Das Hauptnetz des Standard-Agenten 87
5.11 Die Protokollfabrik des Standard-Agenten 88
5.12 Eine einfache Wissensbasis für den Standard-Agenten 88
5.13 Die anfängliche Wissensbasis des AMS-Agenten 91

5.14 Das Registrierungsprotokoll `ams_register` des AMS-Agenten 92
5.15 Der Agenten-Lebenszyklus als Petrinetz 94
5.16 Die am Nachrichtentransport beteiligten Klassen und Netze 97
5.17 Lokaler Nachrichtentransport von Agent A an Agent B 99
5.18 Nachrichtentransport von Agent A an einen externen Agenten 100
5.19 Die Implementierung des ACC (vollständiger Quellcode) 105
5.20 ACC-Implementierung mit getrennter Speicherung der Daten 106
5.21 Das Netz `InternalMTS` (Grundprinzip) 113
5.22 Das Netz `InternalMTS` (vollständiger Quellcode) 114
5.23 Das Netz `TcpIpMTS` (vollständiger Quellcode) 117
5.24 Die Klassen zur Repräsentierung der ACL-Nachrichten und der Agent Management Ontology 120
5.25 Die Klassenstruktur des Transformationsdienstes 124

6.1 Aufgabenteilung und Zusammenarbeit der Agenten im Siedler-Projekt . 141

Kapitel 1

Einleitung

Die zunehmende Verteiltheit von Software wirft die Frage nach neuen Softwaretechniken auf, die auf die veränderten Anforderungen zugeschnitten sind. Einen vielversprechenden Ansatz bietet hier die Softwareentwicklung mit Agenten. Das Gebiet ist allerdings noch jung, so dass zur Zeit viele verschiedene Konzepte und Ansätze bestehen, die sich teilweise schon in der Definition der Eigenschaften eines *Agenten* unterscheiden.

Sollen Agenten in Software eingesetzt werden, so greift man bisher hauptsächlich auf Frameworks zurück, die eine Programmierung von Agenten in einer Hochsprache wie z.B. Java unterstützen. Mitunter verzichtet man auch auf die Unterstützung durch ein Framework und implementiert ein eigenes Agentenkonzept auf Basis einer objektorientierten Programmiersprache. Eine grafische Modellierung, wie sie sich in der Objektorientierung mit UML durchgesetzt hat, findet kaum Anwendung. Um hier Abhilfe zu schaffen, wird am Arbeitsbereich Theoretische Grundlagen der Informatik (TGI) des Fachbereichs Informatik an der Universität Hamburg seit einigen Jahren untersucht, inwiefern sich Petrinetze zur grafischen Modellierung von Agenten und Multiagentensystemen eignen. Dabei werden neben den fundamentalen Konzepten der gefärbten Petrinetze insbesondere objektorientierte Petrinetze (siehe [Mol1996]) mit Netzen als aktiven Marken (siehe [Val1998]) eingesetzt. Diese Konzepte werden vom Formalismus der Referenznetze (siehe [Kum2002]) hervorragend unterstützt und können Dank des Werkzeugs Renew (Reference Net Workshop, siehe [KWD2002]) sowohl grafisch entwickelt als auch in einem Simulator ausgeführt werden.

Aus dem Gedanken der agentenorientierten Petrinetze hat sich inzwischen das referenznetzbasierte Framework MULAN (kurz für „**Mul**ti**a**genten**n**etze“, siehe [Röl1999] und [Röl2002]) herauskristallisiert. MULAN bietet eine Architektur, die grafisch intuitiv veranschaulicht, wie sich Agenten *in* ihrer Umgebung bewegen bzw. kommunizieren und wie die Agenten durch *in ihnen* ablaufende Protokolle gesteuert werden. Die Umgebung, in der sich die Agenten befinden, wird Plattform genannt. Diese Plattform ermöglicht den Agenten, untereinander mittels Nachrichten zu kommunizieren. Es kann mehrere Plattformen in einem Multiagentensystem geben, dabei erlauben die Verbindungen eines sogenannten Ortsnetzes, *in* dem sich die Plattformen befinden, die Kommunikation zwischen Agenten über Plattformgrenzen hinweg.

Alle vier Ebenen von MULAN – Ortsnetze, Plattformen, Agenten und Protokolle – werden durch Referenznetze spezifiert. Der Simulator des Renew-Werkzeuges erlaubt die Ausführung der Netze, so dass die praktische Nutzung der MULAN-Umgebung möglich ist. Konkrete Agenten können, gesteuert durch ihre jeweiligen Protokolle, miteinander innerhalb ihrer Heimat-Plattform oder auch über das Ortsnetz mit Agenten auf anderen Plattformen kommunizieren. Die plattformübergreifende Kommunikation kann aber bisher – wegen der gemeinsamen Simulation aller beteiligten Netze auf einer physischen Maschine – nicht wirklich Entfernungen überbrücken, was für reale Multiagentensysteme wünschenswert wäre. Auch eine Kommunikation mit beliebigen anderen Agentenplattformen, die nicht der petrinetzbasierten MULAN-Architektur folgen, sollte idealerweise möglich sein.

An dieser Stelle setzt diese Diplomarbeit an: Das Ziel ist, der MULAN-Architektur zu einer Plattform zu verhelfen, welche die Kommunikationsmöglichkeiten der darauf befindlichen Agenten so erweitert, dass sie mit Agenten auf fremden Plattformen Kontakt aufnehmen können. Eine fremde Plattform kann sowohl eine physisch auf einer anderen Maschine laufende MULAN-Plattform als auch eine auf einer alternativen Architektur basierend konstruierte Plattform sein (oder natürlich auch beides: eine verschieden konstruierte Plattform auf einer anderen Maschine). Da aber die Kommunikation zwischen architektonisch verschiedenen Plattformen nur funktionieren kann, wenn alle Beteiligten eine gemeinsame Verständigungsbasis haben, muss die hier bereitgestellte MULAN-Plattform sich an definierte und bekannte Standards halten, die auch von den fremden Plattformen eingehalten werden.

Zu diesem Zweck wird auf die Arbeit der „**F**oundation for **I**ntelligent **P**hysical **A**gents“ (FIPA) zurückgegriffen. Diese Organisation hat eine Reihe von Spezifikationen erstellt, welche die Interoperabilität von Agentenplattformen unterschiedlicher Herkunft erlauben sollen. Die Spezifikationen decken etliche Bereiche der Agentenkommunikation ab, so z.B. Semantik, Struktur und Kodierung von Nachrichten, Aufbau und technische Protokolle eines netzwerkfähigen Nachrichtentransportsystems oder Verwaltungs- und Verzeichnisdienste, welche das Miteinander der Agenten auf einer Plattform in geregelte Bahnen lenken.

Die FIPA-Spezifikationen und die MULAN-Architektur ergänzen sich gut, so dass eine Verbindung der beiden mittels einer softwaretechnischen Plattformarchitektur möglich ist. Die in dieser Arbeit vorgeschlagene Architektur trägt den Namen „**C**oncurrent **A**gent **P**latform **A**rchitecture“ (CAPA). Der englischsprachige Name geht auf das parallel zu dieser Arbeit im internationalen Workshop „Agent-Oriented Software Engineering“ (AOSE’02) vorgestellte Papier [DMR2002] zurück, welches sich mit der Plattform und der in ihr vorhandenen Nebenläufigkeit auseinandersetzt.

Diese Arbeit gliedert sich wie folgt: In Kapitel 2 werden einige technische Grundlagen beschrieben und Konventionen geklärt, die für das Verständnis der realisierten Plattform notwendig sind. Kapitel 3 stellt den Aufbau von Multiagentensystemen im Allgemeinen sowie MULAN im Besonderen vor. Die diversen Spezifikationen der FIPA werden im Kapitel 4, geordnet nach den verschiedenen betroffenen Aspekten der Agentenkommunikation, beleuchtet.

Aufbauend auf den vorangehenden Kapiteln wird im Kapitel 5 CAPA vorgestellt und diskutiert. Das Kapitel gliedert sich in drei Bereiche: Der erste Abschnitt stellt die Architektur der Plattform vor und erläutert die Motive, die zu dieser Architektur geführt haben. Im zweiten Teil wird der implementierte Plattform-Prototyp beschrieben. Dabei werden auch einige Detailfragen diskutiert, die sich während der Implementierung der Plattform aufgeworfen haben. Schließlich fasst der dritte Abschnitt des Kapitels die Eigenschaften der Plattform zusammen und bewertet ihre Leistungsfähigkeit.

In Kapitel 6 wird kurz eine Anwendung vorgestellt, bei der die Plattform bereits ihre Praxistauglichkeit bewiesen hat: In einem Hauptstudiumsprojekt wurde unter Verwendung von CAPA ein Brettspiel mit Agenten im Rahmen der Mulan-Architektur modelliert. Zum Schluss fasst Kapitel 7 die Ergebnisse dieser Arbeit zusammen und diskutiert die zukünftigen Entwicklungsmöglichkeiten von CAPA.

Kapitel 2

Technische Bausteine

In diesem Kapitel werden die im Laufe der Arbeit eingesetzten Programmiersprachen, Diagrammarten und Werkzeuge vorgestellt. Dabei werden jeweils nur die Eigenschaften näher erläutert, die für diese Arbeit von Relevanz sind. Ein grundlegendes Verständnis der Objektorientierung und der Petrinetze wird dabei vorausgesetzt.

Die Vorstellung ist eingeteilt in drei Abschnitte: Zuerst wird die Programmiersprache Java nebst einiger Java-spezifischer Werkzeuge präsentiert. Anschließend wird die Bedeutung der grafischen Elemente in zwei UML-Diagrammarten in Erinnerung gerufen, welche in dieser Arbeit verwendet werden. Den umfangreichsten Abschnitt nehmen die Referenznetze und das dazugehörige Werkzeug Renew ein. Referenznetze werden in dieser Arbeit als gleichberechtigte Programmiersprache neben (und in Kombination mit) Java eingesetzt.

2.1 Java

Java (siehe [Sun2002]) ist eine objektorientierte und streng getypte imperative Programmiersprache, die von der Firma Sun Microsystems mit dem Ziel entwickelt und zur Verfügung gestellt wurde, eine weit verbreitete, plattformunabhängige Basis für Internetanwendungen zu bieten. Unter „Plattformunabhängigkeit“ wird in diesem Zusammenhang verstanden, dass in Java geschriebene und in Java-Byte-Code übersetzte Programme ohne weitere Änderung auf vielen verschiedenen Hardwareausstattungen und Betriebssystemen ausgeführt werden können. Für eine Agentenplattform wie die in dieser Arbeit vorgestellte ist die Plattformunabhängigkeit ein interessantes Merkmal, das den Einsatz der Agentenplattform an verschiedenen Orten erleichtert.

Einige weitere Eigenschaften der Sprache Java möchte ich im folgenden vorstellen, weil sie entweder Einfluss auf den Entwurf und die Implementierung der Agentenplattform haben oder aber für das Verständnis einiger Eigenschaften von Referenznetzen hilfreich sind. Eine ausführliche und exakte Definition der Sprache ist in „The Java Language Specification“ [GJS1996] zu finden.

Typsystem. Mit Ausnahme weniger primitiver Typen für numerische Werte sind alle Typen in Java Referenztypen. Ein Referenztyp kann entweder eine Klasse, ein Interface[1] oder ein auf einem beliebigen Typ aufbauendes Array sein.

Interfaces definieren einen Typ, indem sie seine Schnittstelle mittels Methodensignaturen beschreiben. Eine Methodensignatur besteht aus einem Namen sowie einer getypten Parameterliste. Es dürfen innerhalb eines Typs keine zwei Methoden definiert werden, die dieselbe Signatur aufweisen. Diese Einschränkung gilt auch dann, wenn sich die Methoden im Typ des Rückgabewertes oder in den angebrachten Modifikatoren zur Zugriffsregelung unterscheiden.

Eine Klasse definiert ebenso wie ein Interface einen Typ, implementiert aber darüber hinaus auch die Funktionalität hinter der Schnittstelle. Alle Klassen und Interfaces sind in einer Typ- oder Vererbungshierarchie partiell geordnet, an deren Wurzel die allgemeinste Klasse `Object` steht. Mehrfachvererbung ist bezüglich der Typdefinitionen aus Interfaces möglich, aber implementierte Funktionalität kann nur von jeweils einer Superklasse geerbt werden.

Java prüft Typen in der Regel statisch, d.h. zur Übersetzungszeit. Dieses Vorgehen vermeidet viele Laufzeitfehler. Für die Fälle, in denen eine dynamische Typprüfung zur Laufzeit benötigt wird, kann und muss der Programmierer eine explizite Typumwandlung an der betreffenden Stelle im Code vorsehen.

Klassen und Interfaces werden in Java in „Packages“ organisiert. Diese Organisation erlaubt eine Gruppierung inhaltlich zusammenhängender Klassen und bietet darüber hinaus Möglichkeiten, den Zugriff auf Methoden von anderen Packages aus zu beschränken.

Ausnahmebehandlung. Falls semantische Bedingungen in einem ausgeführten Java-Programm verletzt werden, wird der eigentlich vorgesehene Kontrollfluss des Programms zugunsten einer Ausnahmebehandlung unterbrochen. Von dem Punkt aus, an dem die Ausnahmesituation auftritt, wird eine `Exception` geworfen. Die `Exception` verlässt alle vom bisherigen Kontrollfluss durchlaufenen Methodenkörper, bis sie vom Code eines der Methodenkörper aufgefangen wird. Auf diese Weise kann der Kontrollfluss in Ausnahmesituationen schnell zu einer höheren Ebene übergeben werden. Auf der höheren Ebene kann dann auf übersichtliche Weise eine angemessene Fehlerbehandlung erfolgen.

Dokumentation. Zum Lieferumfang des „Java Software Development Kit“ (JDK bzw. SDK) gehört das Werkzeug JavaDoc. Damit können speziell gekennzeichnete Kommentare aus dem Quellcode von Klassen und Interfaces in eine gut les- und navigierbare Dokumentation für die späteren Nutzer umgewandelt werden. Über die statische Typprüfung bei Methodensignaturen hinaus bietet Java damit eine Möglichkeit,

[1] Den englischen Begriff „Interface“ werde ich in dieser Arbeit ausschließlich für das hier vorgestellte Java-Konzept verwenden. Die deutsche Übersetzung „Schnittstelle“ benutze ich hingegen im allgemeineren Sinn, also als Beschreibung der Kommunikations- und Interaktionsmöglichkeiten am Rand eines gekapselten Systems.

den Verwendungszweck, Vor- und Nachbedingungen, Seiteneffekte und Ausnahmebehandlung von Methoden bereits beim Programmieren zu dokumentieren.

Der im Rahmen dieser Arbeit entstandene Quellcode macht ausgiebigen Gebrauch von den JavaDoc-Kommentaren, so dass eine ausführliche Dokumentation aller Klassen und Interfaces der entwickelten Agentenplattform zur Verfügung steht. Den ursprünglichen Plan, die erzeugte Dokumentation als Anhang in diese Arbeit mit aufzunehmen, musste ich aufgrund ihres Umfangs (ca. 150 Seiten) fallen lassen.

Klassenbibliotheken. Das Java-Laufzeitsystem beinhaltet eine umfangreiche Klassenbibliothek. Diese umfasst unter anderem ein Package mit häufig benötigten Behälterklassen, Packages zur Entwicklung von grafischen Benutzungsoberflächen und Packages zur Implementierung von Netzwerkkommunikation auf verschiedenen Ebenen.

Ergänzend zu der Standard-Klassenbibliothek sind etliche weitere Klassenbibliotheken zu verschiedensten Anwendungsgebieten von Drittanbietern programmiert worden. Einige dieser Bibliotheken stehen auch als Open Source im Internet zur Verfügung, so dass sie einfach genutzt werden können. Da die Funktionalität einer Bibliothek schon fertig verwendbar zur Verfügung steht, wird die Implementierung von darauf aufbauenden Anwendungen einfacher.

Nebenläufigkeit und Synchronisation. Java unterstützt die Entwicklung nebenläufiger Anwendungen mittels Threads und Monitoren. Ein Thread ist ein sequentieller Strang der Programmausführung, mehrere Threads können nebenläufig ausgeführt werden. Sollen sich zwei oder mehr Threads synchronisieren, so kann dies in einem Monitor passieren. Jedes Objekt in Java bietet einen eigenen Monitor, in dem nur ein Thread zur Zeit ausgeführt werden kann. Im Code der Klasse werden die durch den Monitor geschützten Bereiche durch das Schlüsselwort `synchronized` textuell gekennzeichnet. Ist ein Monitor bereits von einem Thread belegt, so werden alle weiteren Threads, die den vom Monitor geschützten Code ausführen wollen, bis zum Freiwerden des Monitors blockiert.

Innerhalb des vom Monitor geschützten Bereiches kann ein Thread explizit schlafen gelegt werden. Dann wird der Monitor für andere Threads freigegeben. Da der erste Thread nun schläft, bleibt die Bedingung gültig, dass nur ein Thread im Monitor aktiv sein darf. Die anderen Threads können den schlafenden Thread wieder aufwecken.

Die durch die Monitore gebotene Sperrmöglichkeit ist im Vergleich zu Semaphoren oder anderen, ähnlich fein granulierten Synchronisationsmechanismen relativ fehlerresistent. Die textuelle Kennzeichnung der geschützten Bereiche im Quellcode garantiert, dass alle benötigten Sperren vom Java-System korrekt gesetzt und wieder freigegeben werden – es besteht kein Risiko, dass der Programmierer eine der Sperren vergisst. Dies gilt auch, wenn geschützter Code durch eine Ausnahmebehandlung auf unerwartete Weise verlassen wird. Ein Nachteil der textuellen Kennzeichnung ist allerdings, dass sie sich nicht über einen Methodenkörper hinweg erstrecken kann. Soll ein Monitor über mehrere Methodenaufrufe hinweg gesperrt werden, so muss dies außerhalb der geschützten Methoden geschehen.

2.2 UML

Die „Unified Modeling Language" (UML, siehe [UML2001]) definiert verschiedene Diagrammarten, die bei der objektorientierten Analyse, Spezifikation und Programmierung von Nutzen sind. In dieser Arbeit werden im wesentlichen zwei Diagrammarten verwendet, Klassendiagramme und Sequenzdiagramme. Daher soll die Semantik beider Diagrammarten hier kurz erläutert werden.

Klassendiagramm. Ein Klassendiagramm beschreibt die statische Struktur eines Systems: Typen (Klassen oder Interfaces), ihren internen Aufbau, ihre Schnittstellen und ihre Beziehungen untereinander. In Abbildung 2.1 sind die grundlegenden Elemente eines Klassendiagramms dargestellt.

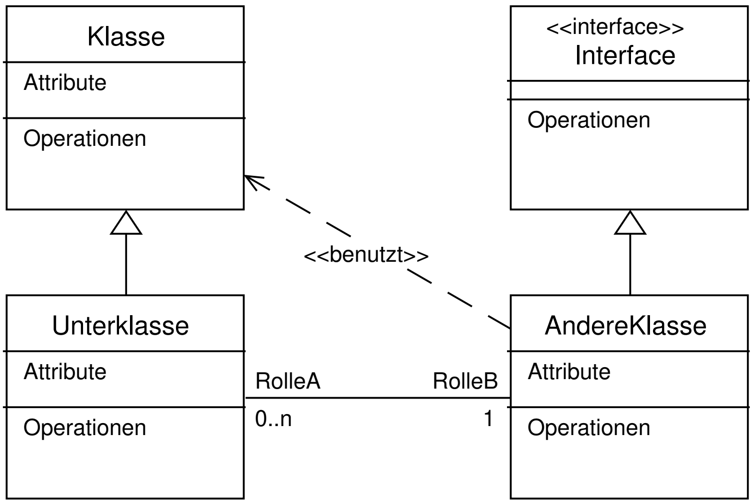


Abbildung 2.1: Elemente eines Klassendiagramms

Zu sehen sind drei Klassen und ein Interface, jeweils durch Rechtecke repräsentiert. Jedes Rechteck wird in drei Abschnitte geteilt: Der oberste Abschnitt nimmt den Namen der Klasse bzw. des Interfaces auf. Das Interface wird zusätzlich durch den in spitze Klammern gesetzten Stereotyp „«interface»" als solches gekennzeichnet. Im mittleren Abschnitt jedes Rechtecks werden die Attribute der Klassen aufgeführt. Da ein Interface als reine Schnittstellendefinition keine Attribute hat, bleibt der Abschnitt in diesem Fall leer (er könnte auch ganz weggelassen werden). Der untere Abschnitt listet die Methoden des jeweiligen Typs auf.

Der nicht gefüllte Pfeil, der von der „Unterklasse“ zur „Klasse“ weist, stellt die Generalisierungsbeziehung zwischen den beiden Klassen dar. Zusätzlich zur Typspezialisierung, die die „Unterklasse“ gegenüber der „Klasse“ vornimmt, wird die Funktionalität der Oberklasse von der Unterklasse geerbt. Demselben Prinzip folgend zeigt der unausgefüllte Pfeil zwischen „AndereKlasse“ und „Interface“, dass hier ein Interface von einer Klasse implementiert wird. Es findet also ebenfalls eine Generalisierung statt.

Die Linie zwischen „Unterklasse“ und „AndereKlasse“ beschreibt eine binäre Assoziation. Die Endpunkte der Linie definieren die Art der Verbindung genauer. Im Beispiel nimmt die „Unterklasse“ die Rolle „RolleA“ (aus Sicht der anderen Klasse) ein, während die „AndereKlasse“ die Rolle „RolleB“ ausfüllt. Die Beziehung ist anhand der Kardinalitätsangaben an den Endpunkten als 1:n-Beziehung erkennbar: Beliebig viele (auch keine) Objekte der „Unterklasse“ können an einer Instanz der Assoziation beteiligt sein, aber nur ein Objekt der anderen Klasse.

An den Endpunkten einer Assoziation können weitere Kennzeichnungen vorgenommen werden: Pfeilspitzen deuten die Navigierbarkeit der Verbindung in eine Richtung an. Ein nicht gefüllter Diamant definiert, dass ein Objekt der Klasse an dem durch den Diamanten gekennzeichneten Ende sich aus den Objekten der anderen Klasse zusammensetzt (Aggregation). Ist der Diamant ausgefüllt, so verstärkt sich die Aggregation zur Komposition.

Gestrichelte Pfeile in einem Klassendiagramm stellen eine Abhängigkeit zwischen zwei Diagrammelementen dar. Ein häufig verwendeter Abhängigkeits-Stereotyp ist die im Beispieldiagramm verwendete „«benutzt»“-Beziehung. Sie deutet an, dass Objekte der Klasse „AndereKlasse“ zur Erfüllung ihrer Aufgabe auf Objekte der Klasse „Klasse“ zurückgreifen. Dabei braucht aber keine „echte“ Beziehung im Sinne einer Assoziation zu bestehen.

Sequenzdiagramm. In einem Sequenzdiagramm wird ein beispielhafter Kommunikationsablauf grafisch dargestellt. In Abbildung 2.2 ist ein einfaches Sequenzdiagramm mit den wesentlichen Grundelementen zu sehen.

Die beteiligten Objekte werden horizontal nebeneinander arrangiert, die Zeit verläuft vertikal von oben nach unten. Zu jedem Objekt gehört eine gestrichelte Lebenslinie, an denen die durch schmale Rechtecke dargestellten Aktivierungen angebracht werden. Falls ein Objekt während der dargestellten Sequenz mehrfach geschachtelt aktiviert ist, so können weitere Aktivierungen halb überlappt auf der bestehenden Aktivierung angebracht werden. Fallunterscheidungen, etwa für Ausnahmebehandlungen, sind nicht vorgesehen.

Die horizontalen Pfeile symbolisieren Nachrichten von einem Objekt zum anderen, also Methodenaufrufe. Synchrone Aufrufe, bei denen das aufrufende Objekt bis zur Rückkehr der Methode wartet, werden mit kleinen, ausgefüllten Pfeilspitzen dargestellt. Die Rückkehr von einem Methodenaufruf symbolisiert gemäß der UML-Spezifikation ein gestrichelter Pfeil. Anders ist die Pfeilspitze aus zwei offenen Linien zu interpretieren: Hier findet ein asynchroner Aufruf statt, bei dem der Aufrufer nicht

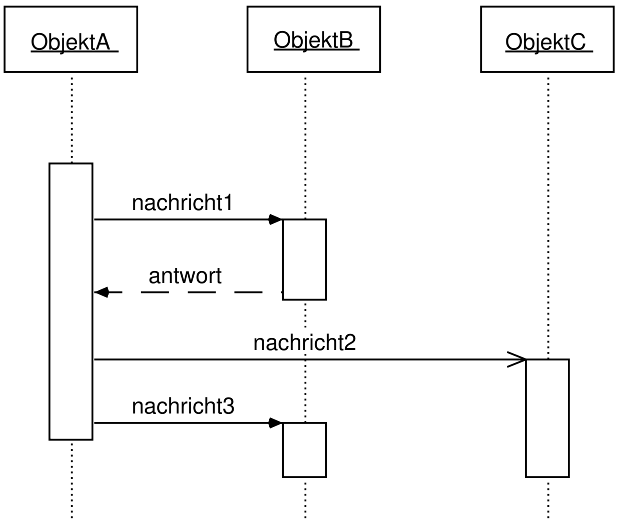


Abbildung 2.2: Elemente eines Sequenzdiagramms

auf eine Antwort wartet. Stattdessen laufen die Aktivierungen von aufrufendem und aufgerufenem Objekt nebenläufig weiter.

2.3 Referenznetze

Referenznetze sind objektorientierte Petrinetze höherer Ordnung, welche der Netze-in-Netzen-Idee von Valk (siehe [Val1998]) folgen und mit synchronen Kanälen nach Christensen (siehe [CD1992]) sowie der Anschriftssprache Java kombinieren. Für Referenznetze steht eine integrierte Entwicklungsumgebung namens Renew (**Re**ference **Ne**t **W**orkshop, siehe [KWD2002]) zur Verfügung, in der Netze gezeichnet und simuliert (d.h. ausgeführt) werden können. In diesem Abschnitt soll eine kurze Einführung in die Semantik der Referenznetze und die Fähigkeiten des Renew-Simulators gegeben werden, soweit es zum Verständnis der in dieser Arbeit dargestellten Netze notwendig ist.

Eine englischsprachige, anschauliche Einführung in die Semantik und den Hintergrund von Petri- und Referenznetzen gibt [Kum2001]. Ebenfalls anwendungsorientiert geht die englischsprachige Dokumentation zum Werkzeug Renew vor, in welcher sowohl die Semantik von Referenznetzen als auch die Bedienung der Entwicklungsumgebung behandelt werden. Der vollständige Referenznetzformalismus wird in der Dissertation von Olaf Kummer [Kum2002] definiert. Dabei werden auch alle bei der Implementierung des Renew-Simulators verwendeten Algorithmen erläutert. Einige Erfahrungen im Umgang mit Renew und Referenznetzen habe ich bereits in [Duv2001] gesammelt und dokumentiert.

Struktur und Markierung. Referenznetze bestehen zunächst wie einfache S/T-Netze aus einer statischen Beschreibung durch *Stellen*, *Transitionen* und *Kanten*. Während der Netzausführung wird in den Stellen in Form von *Marken* der dynamische Zustand des Netzes abgelegt. Wenn während der Netzausführung eine Transition *schaltet*, wird die Markierung der durch Kanten mit dieser Transition verbundenen Stellen modifiziert.

Neben den zwei klassischen Arten von Kanten, *Eingangs-* und *Ausgangskanten*, kennt der Referenznetzformalismus auch *Reservierungskanten* (grafisch mit Pfeilspitzen an beiden Enden dargestellt, siehe Abbildung 2.3), *Testkanten* (grafische Darstellung ohne Pfeilspitzen) und *flexible Kanten* (mit einer doppelten Pfeilspitze notiert).

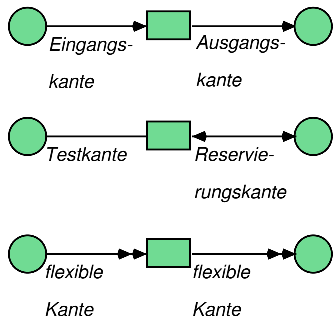


Abbildung 2.3: Kantenarten

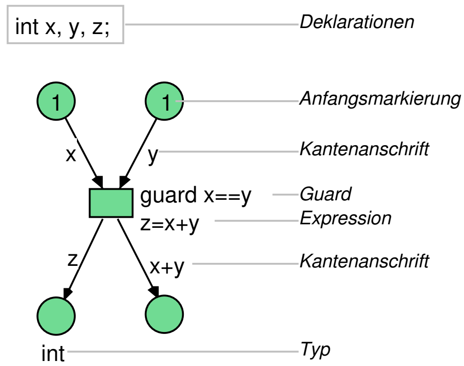


Abbildung 2.4: Einige Anschriften

Eine Eingangskante zieht eine Marke von der angebundenen Stelle ab, eine Ausgangskante legt eine Marke auf die Stelle. Die Reservierungskante ist schlichtweg eine Kurzschreibweise für die Kombination aus je einer Aus- und Eingangskante mit identischer Anschrift – es wird also eine Marke abgezogen und wieder zurückgelegt. Testkanten lassen ähnlich wie Reservierungskanten die Markierung einer Stelle unverändert, allerdings wird die Marke nicht abgezogen. Daher kann eine Marke von mehreren Testkanten gleichzeitig verwendet werden. Die flexiblen Kanten gibt es wiederum als Ein- und Ausgangskanten, ihre Semantik wird weiter unten im Zusammenhang mit der dahinterstehenden Faltungsidee erklärt.

Anschriften und Schaltregel. Der Einfluss der Strukturelemente auf die Netzmarkierung kann mittels *Stellen-*, *Transitions-* oder *Kantenanschriften* näher spezifiziert werden. Deren Notation folgt im wesentlichen der Java-Syntax. Einige Beispiele sind in Abbildung 2.4 zu sehen.

Stellenanschriften können ein Typ (d.h. eine Beschränkung auf Marken eines Typs) und eine Anfangsmarkierung für eine Stelle festlegen. *Kantenanschriften* beeinflussen das Schaltverhalten der Transition, mit der die Kante verbunden ist, indem sie die am Schaltvorgang beteiligten Marken bestimmen. An Transitionen sind drei Arten von Anschriften möglich: ungekennzeichnete *Expressions* und durch entsprechende Schlüsselwörter erkennbare *Guards* und *Actions*. Die drei Arten unterscheiden sich mehr durch Zeitpunkt und Häufigkeit ihrer Auswertung als durch unterschiedliche Ausdrucksmöglichkeiten.

Eine Transition kann nur schalten, wenn genügend Marken in den Eingangsstellen vorliegen, alle Variablen in ihrer Umgebung widerspruchsfrei an Werte (die durch Marken aus Eingangsstellen bestimmt werden) gebunden werden können und alle ihre Guards zu `true` evaluieren. Dementsprechend sind Guards auf boolesche Ausdrücke beschränkt. Actions werden genau einmal pro Schaltvorgang einer Transition ausgeführt, während Guards und Expressions schon während der Aktiviertheitsprüfung (und damit auch ohne Schaltvorgang) ausgewertet werden. Die Unterscheidung bezüglich des Evaluierungszeitpunktes ist notwendig, da in Anschriften Java-Methodenaufrufe enthalten sein dürfen, welche Seiteneffekte hervorrufen können. Seiteneffekte sollten ausschließlich in Actions verwendet werden, da nur dort die Auswertungshäufigkeit kontrollierbar ist.

Typen und Operatoren. In S/T-Netzen gibt es ausschließlich *anonyme*, also ununterscheidbare Marken. Referenznetze sind höhere Petrinetze. Das bedeutet, dass beliebige Datentypen als Marken in den Stellen verwendet werden können (anonyme Marken sind ebenfalls zulässig und werden als „[]“ dargestellt). Die Marken- und Variablentypen sind analog den Java-Typen definiert: Es gibt einige primitive numerische Typen, alles andere sind Referenztypen, also müssen die Marken bzw. Werte Referenzen auf Objekte sein.

Um Prototyping zu ermöglichen, dürfen Referenznetze auch ohne Verwendung von Typen entwickelt werden, allerdings gilt für die Typisierung von Variablen eine „Alles-oder-Nichts“-Regel: Falls es einen Deklarationsblock gibt (wie z.B. in Abbildung 2.4 zu sehen), müssen *alle* Variablen deklariert werden.

Die auf Variablen oder Werten möglichen Operationen sind ebenfalls aus Java entnommen, es gibt aber eine wesentliche Abweichung in Bezug auf den „=“-Operator, welcher in Java Zuweisungen realisiert. Da bei Referenznetzen eine feste Bindung zwischen Variablen und Marken in der Umgebung einer Transition während ihres Schaltvorgangs etabliert wird, passt eine wertändernde Zuweisung nicht ins Konzept. Stattdessen stellt der „=“-Operator eine Gleichheitsanforderung dar, die bei der Bindungssuche erfüllt werden muss. Konsequenterweise sind auch die wertändernden Java-Kurzschreibweisen „+=“, „-=“, „++“ usw. nicht erlaubt. Von diesen Anpassungen völlig unberührt bleiben die bool’schen Vergleichsoperatoren bzw. -methoden `==` und `equals()`, die sich wie gewohnt verhalten.

Um Werte in Stellen gruppieren zu können, ohne gleich eine neue Java-Klasse definieren zu müssen, wurde die Anschriftssprache um *Tupel* erweitert. Ein Tupel besteht aus mehreren Ausdrücken, die durch Kommata getrennt und durch eckige Klammern zusammengehalten werden. Tupel sind nur schwach typisiert – es sind Tupel, aber die Typen der Komponenten sind beliebig. Diese Schwäche fällt jedoch nicht ins Gewicht, da die einzig sinnvolle Operation auf Tupeln die *Unifikation* ist. Zwei Tupel können durch die eben beschriebene Gleichheitsbedingung miteinander unifiziert werden. Dazu müssen die Komponenten des einen Tupels in der richtigen Reihenfolge an die Komponenten des zweiten Tupels gebunden werden können, d.h. es muss komponentenweise Gleichheit hergestellt werden.

Referenzen, Instanzen und Kanäle. Referenznetze sind objektorientiert. Ebenso, wie in anderen objektorientierten Sprachen von einer Klasse mehrere Objekte instanziiert werden können, erlaubt der Referenznetzformalismus mehrere *Netzexemplare* (oder *Netzinstanzen*) als Instanzen eines *Netzmusters*. Eine Netzinstanz hat eine Markierung (analog dem Zustand eines Objekts), wohingegen der statische Teil – das Netzmuster – das allen Instanzen gemeinsame Verhalten definiert (analog einer Klasse). Netzinstanzen können dynamisch zur Laufzeit erzeugt werden. Zerstört werden Netzinstanzen (in Anlehnung an das Garbage-Collector-Konzept von Java) allerdings nur, wenn sie von keiner anderen Instanz mehr referenziert werden **und** alle ihre Transitionen tot sind, d.h. auf keine Art und Weise mehr aktiviert werden können.

Da Netzmuster Klassen entsprechen, können sie ebenfalls als Typen für Werte und Marken verwendet werden. Eine Marke stellt dann eine *Referenz* auf ein Netzexemplar dar. Die Idee, Netze als Marken in Netzen zuzulassen, ist von Valk unter dem Schlagwort „*Netze in Netzen*“ entwickelt worden und wird in [Val1998] vorgestellt.

Transitionen einer Netzinstanz können über einen *synchronen Kanal* (synchrone Kanäle wurden von Christensen in [CD1992] definiert) mit einer anderen Transition einer anderen (oder auch derselben) Instanz eines anderen (oder auch desselben) Netzes verschmolzen werden, so dass die an der Synchronisation beteiligten Transitionen wie eine einzige Transition schalten. Über den synchronen Kanal können in beide Richtungen Werte zur Variablenbindung und Aktiviertheitsprüfung ausgetauscht werden.

Einzige Bedingung für die Initiierung einer Synchronisation ist, dass eine der beteiligten Netzinstanzen eine Referenz auf die andere zur Verfügung hat. Mit dieser Bedingung geht eine generelle Asymmetrie der am Kanal beteiligten Transitionen einher: Die Anschrift an der Transition, welche die Referenz auf das andere Netz zur Verfügung hat, heißt *Downlink*, während die Kanalanschrift der Transition im referenzierten Netz *Uplink* genannt wird. An einer Transition können beliebig viele Downlinks, aber nur maximal ein Uplink angebracht werden.

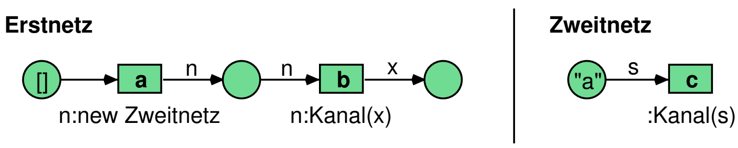


Abbildung 2.5: Instanzen und Kanäle

Zur Illustration dieser Technik mag die Abbildung 2.5 dienen. Die Instanz des Erstnetzes erzeugt zunächst durch Transition `a` eine Zweitnetz-Instanz und hält eine Referenz darauf in der mittleren Stelle (Transport über die Variable `n`). Dann nutzt

Transition `b` die Referenz, um sich über einen Kanal mit der Transition `c` des Zweitnetzes zu synchronisieren, dabei wird der Wert „a“ an die Variable `x` gebunden und in die letzte Stelle des Erstnetzes gelegt.

Faltung und flexible Kanten. Schon in einfachsten Petrinetzformalismen, wie z.B. Bedingungs-Ereignis-Netzen (B/E-Netze), können grundlegende Konzepte wie Sequenzen, Synchronisierung, Konflikte und Nebenläufigkeit ausgedrückt werden. Ein Beispiel ist in Abbildung 2.6 zu sehen, wo die Transitionen „a“ und „b“ nebenläufig schalten können. Transition „c“ synchronisiert die nebenläufigen Stränge wieder.

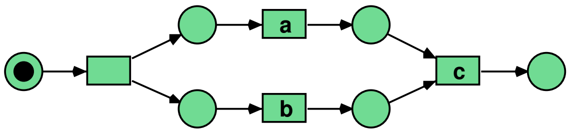


Abbildung 2.6: Nebenläufige Transitionen in einem B/E-Netz

Allerdings erfordern einfache Petrinetzformalismen einen hohen Zeichenaufwand, weil sie keine Möglichkeit bieten, Strukturen wiederzuverwenden. Beginnend mit S/T-Netzen, und noch stärker bei gefärbten oder Referenznetzen, können wiederkehrende Muster der Netzstruktur *zusammengefaltet* werden, um mehr Klarheit in die Netzgrafik zu bringen.

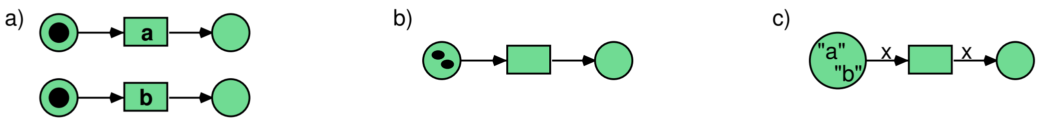


Abbildung 2.7: Faltung

In Abbildung 2.7 werden zwei Faltungsvarianten veranschaulicht: Das in Teil a) dargestellte B/E-Netz kann zum einen in das S/T-Netz in Teil b) gefaltet werden. Dadurch, dass nun zwei Marken in der Eingangsstelle liegen, kann die Transition weiterhin nebenläufig schalten. (Dabei schaltet sie nebenläufig zu sich selbst!) Es tritt aber ein Informationsverlust auf, weil eine Unterscheidung der Aktionen „a“ und „b“ nicht mehr möglich ist. Um die Unterscheidung auch nach der Faltung noch treffen zu können, bietet sich die Faltung des Originalnetzes in ein gefärbtes Netz an, wie in Teil c) dargestellt. Nun liegen nicht mehr zwei anonyme Marken in der Eingangsstelle, sondern zwei unterscheidbare, gefärbte Marken. In den Marken wird die in Variante b) verlorengegangene Information gerettet.

In Abbildung 2.8 ist das Ergebnis einer Faltung des Netzes aus Abbildung 2.6 in ein gefärbtes Netz zu sehen. Die erste Transition produziert zwei unterscheidbare Marken,

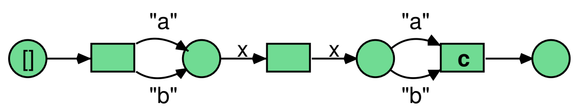


Abbildung 2.8: Das Netz aus Abbildung 2.6 als gefaltetes, gefärbtes Netz

die von der mittleren Transition nebenläufig verarbeitet werden können. Dabei ist an der verarbeiteten Marke zu erkennen, ob gerade Aktion „a“ oder Aktion „b“ durchgeführt wird. Die Transition „c“ synchronisiert weiterhin die nebenläufigen Handlungsstränge, da sie nur schalten kann, wenn beide Marken verfügbar sind.

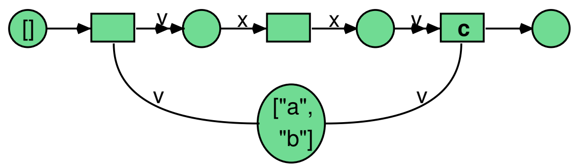


Abbildung 2.9: Das Netz aus Abbildung 2.6 mit flexiblen Kanten gefaltet

Die in Abbildung 2.8 verwendete Technik zum Aufspalten und Synchronisieren von Handlungssträngen kann nur eine feste Anzahl von Strängen behandeln, weil die entsprechenden Kanten fest in das Netz gezeichnet werden müssen. Der Referenznetzformalismus bietet als Ausweg *flexible Kanten* an, die wie in Abbildung 2.9 an doppelten Pfeilspitzen erkennbar sind. Eine flexible Kante schiebt eine Multimenge von Marken auf eine Stelle bzw. zieht sie davon ab. Die Menge muss in Form eines Java-Arrays angegeben werden.[2] Da das Array dynamisch zur Ausführungszeit berechnet werden kann, erlauben die flexiblen Kanten eine Aufspaltung in beliebig viele nebenläufige Stränge. Bei der Synchronisierung durch Transition „c“ müssen die im Array gespeicherten Elemente wieder als Marken vorliegen.

Renew und Java. *Renew* ist eine grafische Entwicklungs- und Simulationsumgebung für Referenznetze. Sie wurde von Olaf Kummer und Frank Wienberg entwickelt, während des Entstehens dieser Arbeit von mir gepflegt und wird auch in naher Zukunft hier am Fachbereich weiterentwickelt und mit neuen Fähigkeiten ausgestattet werden. Die relevante Version, die alle für diese Arbeit benötigten Fähigkeiten enthält, ist die Version 1.6 (siehe [KWD2002]).

[2] Abbildung 2.9 ist zugunsten der besseren Lesbarkeit nicht ganz korrekt: Die an der unteren Stelle notierte Anfangsmarkierung beschreibt in Referenznetz-Syntax ein Tupel. Eigentlich muss in dieser Stelle ein entsprechendes Array-Objekt vorgehalten werden, was aber in Referenznetzen nicht mit einer einfachen Stellenanschrift erzeugt werden kann.

Renew bietet zunächst geeignete Zeichenwerkzeuge, um Netze zu entwerfen. Anschriften an Transitionen, Stellen oder Kanten werden bereits beim Zeichnen des Netzes auf syntaktische Korrektheit geprüft. Darüber hinaus können die gezeichneten Netze kompiliert und an den integrierten Simulator übergeben werden. Die grafische Oberfläche erlaubt es, die laufende Netzsimulation zu beobachten und in gewissen Grenzen zu beeinflussen. So kann der automatische Simulationsablauf (sowohl manuell als auch auf vorab spezifizierte Ereignisse hin) angehalten werden, um dann gezielt einzelne Transitionen zu schalten. Die Marken können detailliert dargestellt und inspiziert, aber nicht während einer laufenden Simulation verändert werden. Damit stellt Renew eine integrierte Entwicklungs- und Debugging-Umgebung bereit.

Da Renew Java-Anschriften für die Netze verwendet, bereitet die Nutzung von Java-Klassen mit ihren Methoden und Attributen aus Referenznetzen heraus wenig Probleme. Lediglich der bequeme Ausnahmebehandlungsmechanismus, den Java mit der `Exception`-Klasse bietet, kann in Renew nicht genutzt werden. Eine zu JavaDoc äquivalente (oder noch besser: darin integrierte) Dokumentationsmöglichkeit besteht ebenfalls nicht. Dies ist insofern besonders schmerzhaft, weil Renew keine automatische statische Typprüfung auf Netz- und Kanaldefinitionen bietet.

Der „Methodenaufruf" an Referenznetzen von Java-Code aus wird in Renew durch sogenannte *Stubs* ermöglicht. Ein Netzstub ist eine Java-Klassendefinition, deren Methoden auf synchrone Kanäle an den Transitionen eines Netzes abgebildet werden. Die weiter oben erwähnte Möglichkeit, ein bestimmtes Referenznetzmuster wie eine Klasse als Typ (z.B. für Stellenanschriften) verwenden zu können, setzt in Renew ebenfalls die Erzeugung von Stubs voraus.

Anstelle der Netzinstanz-Erzeugung mittels des `:new`-Kanals tritt nun die Erzeugung eines Objekts der Stubklasse. Mit dem Stubobjekt wird gleichzeitig die Netzinstanz erzeugt und die Verbindung zwischen den Stub-Methoden und Netz-Kanälen aufgebaut. Wird eine Methode des Stubobjekts aufgerufen, so versucht der Methodenkörper, über einen synchronen Kanal eine Transition der Netzinstanz zu schalten. Ist die Transition nicht aktiviert, so blockiert die Methode, bis die Transition schalten kann.

Die Zuordnung von Methoden zu Kanälen muss keine 1:1-Zuordnung sein. Häufig wird eine Methode durch eine Sequenz von zwei Kanälen implementiert: Der erste Kanal (meist mit dem Namen der Methode benannt) übergibt die Methodenparameter an die Netzinstanz, der zweite Kanal (meist `:result` genannt) liefert den Rückgabewert zurück an den Stub. Um nebenläufige Methodenaufrufe im Netz unterscheiden zu können, sollte ein Identifikationsobjekt für den Methodenaufruf über jeden der Kanäle mitgegeben werden. Auf diese Weise sind auch komplexe „Methodenkörper" durch Netze implementierbar.

Kapitel 3

Multiagentensysteme

Multiagentensysteme sind ein Forschungsgebiet, das aus verschiedenen Ursprüngen wie der Künstlichen Intelligenz und den verteilten Systemen hervorgegangen ist, aber auch Wurzeln in der Softwaretechnik oder der Soziologie findet. Eine umfassende Einführung in dieses Gebiet kann ein Kapitel im Rahmen einer Diplomarbeit nicht bieten – dafür ist das Themenfeld zu weitreichend, zu verstreut und zu jung. Lehrbücher, die eine allgemeine Einführung in Multiagentensysteme versuchen, gibt es erst seit kurzem. Aber auch diesen Büchern kann immer noch die fachliche Herkunft der Autoren angesehen werden, wie bei den beiden als „Multiagent Systems“ betitelten Werken von Ferber [Fer2001] und Weiß [Wei2000], die schon im Untertitel den Fokus auf die verteilte Künstliche Intelligenz legen.

Daher werden in der ersten Hälfte dieses Kapitel nur die Begriffe und Konzepte des Bereichs Multiagentensysteme eingeführt, die für das Verständnis und die Konstruktion einer Agentenplattform notwendig sind. Alle Konzepte höherer Ebenen, die sich z.B. mit der Intelligenz, der Kooperation oder der Organisation von Agenten und Agentengesellschaften befassen, seien hiermit ausgeklammert.

In der zweiten Hälfte des Kapitels wird die Mulan-Architektur vorgestellt, welche ein Multiagentensystem in allen technisch relevanten Aspekten mithilfe von Referenznetzen modelliert. Die Vorstellung der Mulan-Architektur dient aber nicht nur der Veranschaulichung der Konzepte von Multiagentensystemen. Darüber hinaus ist diese Architektur auch die Umgebung, in welche die in dieser Arbeit erstellte Agentenplattform (siehe Kapitel 5) eingebettet ist.

3.1 Begriffe

Die meisten Begriffe im Bereich der Multiagentensysteme verfügen noch nicht über eine allgemein anerkannte, einheitliche Definition. Je nach Kontext wird das Hauptaugenmerk bei den Begriffsdefinitionen auf verschiedene Aspekte des jeweiligen Konzepts gerichtet. Im folgenden sollen daher die Begriffe in einer Form vorgestellt werden, wie sie für die Konstruktion einer Agentenplattform geeignet sind.

Um einen kurzen Überblick über den Zusammenhang der folgenden Begriffsdefinitionen zu geben, sei erwähnt, dass ein *Multiagentensystem* aus mehreren unabhängigen

Agenten zusammengesetzt ist, die im Zusammenspiel die Aufgaben des Systems erfüllen (wobei einzelne Agenten durchaus abweichende Ziele haben können). Unter dem Begriff des Agenten können sowohl reine Softwareprogramme als auch Hardwareeinheiten (z.B. Roboter) oder gar Menschen verstanden werden. Die Agenten sind in einer *Umwelt* angesiedelt, welche die Interaktions- und Kommunikationsmöglichkeiten der Agenten festlegt. Im Kontext von Software-Agenten besteht die Umwelt ausschließlich aus Agenten, die meist auf *Plattformen* angesiedelt sind. Unter einer Plattform ist in diesem Zusammenhang die Gesamtheit aus physischer Hardware des Rechners, dem Betriebssystem und weiterer Software zur Ausführung der Agenten zu verstehen.

3.1.1 Agent und Umwelt

Der Agentenbegriff existiert schon in der klassischen Künstlichen Intelligenz (KI) und bezeichnet dort ein einzelnes System zur Lösung von Problemen. Aus der verteilten Künstlichen Intelligenz (VKI) und den verteilten Systemen ist dann der Agentenbegriff entstanden, der im Zusammenhang von Multiagentensystemen verwendet wird. Bei der Definition dieses Agentenbegriffs fällt immer auch der Begriff „Umwelt". Neben der (V)KI und den verteilten Systemen spielen weitere Diszplinen wie die Softwaretechnik oder die Soziologie im Bereich der Multiagentensysteme eine Rolle, so dass insgesamt eine Fülle unterschiedlicher Definitionen die unterschiedlichsten Aspekte des „Agentseins" betont, von erweiterten aktiven Objekten über intelligente verteilte Problemlöser bis hin zu Mitgliedern von Gesellschaften.

Häufig zitiert wird die mehrfach von Jennings und Wooldridge verwendete und überarbeitete Definition, die aus der ausführlichen Diskussion des Agentenbegriffs in [WJ1995] hervorgegangen ist (hier aus [Jen2000, S. 280]):

> An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.

Dabei bedürfen eigentlich alle Begriffe weiterer Erklärung:

- Agenten sind *gekapselt*, d.h. sie haben klar umrissene Grenzen und Schnittstellen.
- Agenten sind in einer *Umwelt* eingebettet, die sie durch Sensoren wahrnehmen und durch Effektoren beeinflussen können.
- Agenten sind zu *flexiblen Aktionen* fähig, d.h. sie können sowohl zeitnah auf Ereignisse *re*agieren als auch von selbst agieren (d.h. *pro*aktiv handeln).
- Agenten sind *autonom*, d.h. sie können ihren Zustand und ihr Verhalten selbst bestimmen.
- Agenten sind für einen bestimmten *Zweck* geschaffen und verfolgen entsprechende Ziele.

Diese Definition, die von Jennings und Wooldridge selber als *schwache* Definition des Begriffs gesehen wird, kann in etwa als kleinster gemeinsamer Nenner für die meisten Agentensysteme gesehen werden.

Für die *starke* Definition des Agenten führen Jennings und Wooldridge in [WJ1995] zusätzliche Eigenschaften an, die in die Richtung der Künstlichen Intelligenz gehen. Dazu gehört die Annahme, dass Agenten mit menschenähnlichen Mechanismen und Begriffen modelliert werden, wie z.B. Wissen, Glauben, Zielen und Verpflichtungen.

Die in Ferbers Lehrbuch [Fer2001] gegebene Agenten-Definition geht in eine ähnliche Richtung. Über das selbständige, zielbestimmte Agieren in einer Umwelt hinaus führt Ferber explizit die Kommunikation zwischen Agenten, die Möglichkeit, Dienste zu offerieren, und den Besitz eigener Ressourcen an. Er betont, dass Agenten die Umwelt nur unvollständig wahrnehmen und repräsentieren können. Jeder Agent muss sein Verhalten darauf ausrichten, mit seinen beschränkten Informationen, Ressourcen, Fähigkeiten, Kommunikationsmöglichkeiten und kognitiven Modellen seine jeweiligen Ziele zu erreichen.

Ferber unterscheidet zwischen rein kommunikativen Agenten (sogenannten Software-Agenten) und rein situierten (physischen) Agenten. Für einen Software-Agenten, der sich in einem offenen Rechnernetz befindet, besteht die Umwelt ausschließlich aus anderen Agenten, mit denen er kommunizieren kann. Seine Effektoren und Sensoren sind somit auf den Nachrichtenempfang und -versand reduziert. Anders bei rein situierten Agenten, für die die Umwelt in der Regel ein metrischer, physikalischer Raum ist, der nur über Sensoren wahrgenommen und durch Aktionen verändert werden kann, d.h. nur indirekte Kommunikation zwischen den Agenten zulässt.

Beschäftigt man sich mit Software-Agenten, so besteht auch die Möglichkeit einer anderen, eher softwaretechnisch orientierten Herangehensweise an den Agenten-Begriff, wie z.B. von Odell in [Ode1999] diskutiert. Der bisherige Trend der Softwaretechnik geht von einer ursprünglich monolithischen Programmierung zu immer modularer aufgebauten Systemen. Der derzeitige Stand der Technik umfasst objektorientierte und komponentenbasierte Systeme, die aus mehreren, getrennt definierten, gekapselten Teilen zusammengesetzt sind. Nichtsdestotrotz werden in den aktuellen Paradigmen immer noch geschlossene Systeme entwickelt, die als Ganzes einem einzigen Ziel dienen.

Agenten lassen sich als Weiterentwicklung des Objekt- oder Komponentenkonzeptes verstehen, wobei zum Objekt im wesentlichen Autonomie, Aktivität und Verteilung hinzugefügt werden. Vereinfacht gesagt kann ein Agent als aktives Objekt von sich aus ohne äußeren Anreiz tätig werden, und er kann entscheiden, wie und ob er eine eingehende Nachricht überhaupt verarbeitet – im Gegensatz zum passiven Objekt, das einen Methodenaufruf auf jeden Fall durchführt („agents can be thought of as objects that can say 'No' as well as 'Go'“ [Ode1999, S. 2]). Außerdem können Agenten an verschiedenen Orten arbeiten und ggf. auch den Ort wechseln.

Beim Übergang vom objektorientierten zum agentenorientierten Paradigma verschiebt sich aber auch der Spezifikationsfokus von der statischen Struktur zur dynamischen Interaktion im System. Z.B. wird in den Vorschlägen zur agentenorientierten Erweiterung von UML (siehe [OPB2000]) statt des in der Objektorientierung zentralen

Klassendiagramms hauptächlich das Interaktionsdiagramm betrachtet und erweitert. Des weiteren kann ein Multiagentensystem offen sein – in dem Sinne, dass zur Entwicklungszeit des Systems noch nicht klar ist, welche Agenten mit welchen Zielen und Aufgaben einst in dem System arbeiten werden.

Für die folgenden Kapitel soll der Begriff des Agenten für einen gekapselten Software-Agenten stehen, der über Nachrichten mit seiner Umwelt, d.h. anderen Agenten, kommunizieren kann. Flexibles Verhalten und Autonomie sind möglich, aber nicht zwingend für jeden Agenten erforderlich, denn auch ein Objekt kann als Spezialfall eines Agenten gesehen werden: als rein reaktiver, nicht-autonomer Agent.

3.1.2 Multiagentensystem

Die Motivation zur Entwicklung und Verwendung von Multiagentensystemen ist, dass ein Konglomerat aus unabhängigen Softwareeinheiten genau die Probleme umschiffen helfen soll, welche der klassische monolithische Systementwurf in verteilten Umgebungen mit unvorhergesehenen oder eiligen Situationen hat. Der Begriff „monolitisch“ schließt in diesem Fall auch objektorientierte und komponentenbasierte Systeme ein, die letztendlich, trotz ihrer Modularität, als geschlossene Systeme einem übergeordneten Ziel dienen.

Ferber definiert in [Fer2001] ein Multiagentensystem operational:

> Der Begriff Multiagentensystem (MAS) bezeichnet ein System, das aus folgenden Elementen besteht:
>
> 1. Eine Umwelt E. E ist ein Raum, der im Allgemeinen ein Volumen hat.
> 2. Eine Menge von Objekten, O. Diese sind situiert, das bedeutet, dass zu einem beliebigen Zeitpunkt jedem Objekt eine Position in E zugewiesen werden kann. Objekte können von den Agenten wahrgenommen, erzeugt, modifiziert und gelöscht werden.
> 3. Eine Menge von Agenten, A. Diese repräsentieren die aktiven Objekte ($\mathsf{A} \subseteq \mathsf{O}$) des Systems.
> 4. Eine Menge von Beziehungen, R, die Objekte miteinander verbinden.
> 5. Eine Menge von Operationen, Op, damit Agenten Objekte empfangen, erzeugen, konsumieren, verändern und löschen können.
> 6. Operatoren mit der Aufgabe, die Anwendung dieser Operationen und die Reaktion der Umwelt auf die entsprechenden Veränderungsversuche darzustellen. Wir bezeichnen diese Operatoren als die Gesetze des Universums.

Weiß nimmt im Prolog zu [Wei2000] eine gröbere Sicht ein und zitiert folgende Zusammenstellung der Eigenschaften von Multiagentensystemen aus [JSW1998]:

> The characteristics of MAS are:

- each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint;
- there is no global system control;
- data is decentralized; and
- computation is asynchronous.

Darüber hinaus führt er in Anlehnung an die Diskussion in [HS1998] eine Reihe von Eigenschaften an, in denen Multiagentensysteme variieren können. Dazu gehören Fragen von der bloßen Anzahl der Agenten über die Unterschiede in ihren technischen Fähigkeiten bis zur Komplexität ihres „Denkapparats“ oder den Wahrnehmungs- und Interaktionsmöglichkeiten mit ihrer Umgebung.

Sollen ganze Multiagentensysteme modelliert werden, wird mitunter auf die Idee einer sozialen Gesellschaft zurückgegriffen, in der die Agenten bestimmte Rollen annehmen. Mit einer Rolle gehen meist bestimmte Rechte, Pflichten und Verhaltensmuster einher.

3.1.3 Ort, Plattform und Mobilität

Bei situierten Agenten, die sich in einer physikalischen Umwelt bewegen, sind die Konzepte Ort und Mobilität klar: Der Agent befindet sich an einem bestimmten Punkt im Raum. Wenn er mobil ist, kann er sich an einen anderen Ort bewegen.

Bei rein kommunizierenden Software-Agenten gibt es keine physikalische Umwelt, sondern nur andere Agenten. Dennoch können diskrete Orte eingeführt werden, an denen sich Agenten aufhalten können. Die Orte unterscheiden sich nur durch ihre Eigenschaften. Zu den Eigenschaften eines Ortes gehört nicht zuletzt die Menge der anderen an diesem Ort lebenden Agenten sowie die an dem Ort angebotenen Dienste. Die Kommunikation zwischen Agenten kann unterschieden werden in lokale und ortsübergreifende Kommunikation. Je nach Sichtweise können damit Unterschiede im anfallenden Kommunikationsaufwand oder bei den verfügbaren Interaktionsmöglichkeiten modelliert werden.

Darüber hinaus kann es aber auch für Software-Agenten einen physikalischen Aspekt des Ortes geben, wenn verschiedene logische Orte in verschiedenen physischen Maschinen abgebildet sind und der Agent sich beim Ortswechsel über eine Netzwerkverbindung bewegen muss. Dieser physikalische Aspekt von Orten wird mit der Einführung des Konzepts „Agentenplattform“ abgedeckt.

Plattformen tauchen in der abstrakten Sicht auf Multiagentensysteme nicht auf, bei der Implementierung eines solchen Systems sind sie jedoch immer präsent. Unter einer Agentenplattform ist die Gesamtheit aus Hard- und Software zu verstehen, welche den Software-Agenten die Ausführung ermöglicht. Dies beginnt beim Prozessor, schließt weitere Hardwareausstattung wie Speicher oder Netzwerkverbindungen ebenso wie die Software von Betriebssystem und Middleware ein, und geht bis zu etwaigen angebotenen agentenspezifischen Diensten wie z.B. Verwaltungs- und Kommunikationsdiensten.

Auch wenn Plattformen häufig aufgrund ihres physikalischen Aspekts als Orte gesehen werden, ist diese eins-zu-eins-Beziehung nicht zwingend. Zum einen kann eine Maschine mehrere Plattformen betreiben, ebenso wie eine Plattform auf mehrere Maschinen verteilt sein kann, zum anderen ist die Abbildung des logischen Ortskonzepts auf ein physikalisches Ortskonzept optional.

3.1.4 Architekturen

Unter der Architektur eines Systems ist dessen grundlegende Konstruktion zu verstehen. Damit beschreibt eine Architektur die Elemente des Systems, deren Aufgaben und die zwischen den Elementen stattfindende Zusammenarbeit, also die Schnittstellen und Interaktionen zwischen den Elementen.

Oft wird dem Architekturbegriff in einem Anwendungsfeld eine bestimmte Sichtweise auf das System zugeordnet. Das kann zu Missverständnissen führen, wenn sich mehrere Forschungsdisziplinen überschneiden, wie es bei Agenten und Multiagentensystemen der Fall ist. Daher sind in dieser Arbeit feinere Definitionen des Architekturbegriffs nötig, mittels derer die verschiedenen Sichtweisen explizit differenziert werden können.

Aus der Sicht der (verteilten) Künstlichen Intelligenz geht es bei der Architektur eines Agenten darum, welches kognitive Modell seinem Verhalten zugrunde liegt. Verschiedene Architekturen unterscheiden sich also darin, ob und wie der Agent Ziele und Wissen repräsentiert oder auf welche Weise seine Aktionen und Reaktionen zu Stande kommen, also welchem Organisationsprinzip der Entscheidungsprozess im Agenten folgt. Dies ist der Architekturbegriff, wie er z.B. in den VKI-orientierten Büchern von Weiss [Wei2000] und Ferber [Fer2001] verwendet wird. In dieser Arbeit wird, falls nicht aus dem Kontext ersichtlich ist, dass es sich um diese spezielle Sichtweise auf ein System handelt, die Bezeichnung *Entscheidungsarchitektur* zur Konkretisierung verwendet.

Hinter den VKI-Architekturen stehen sehr unterschiedliche Herangehensweisen an das Verhalten und die Aufgaben von Agenten. Zwei prominente Ideen sehr gegensätzlicher Art sind:

Beliefs–Desires–Intentions (BDI) Diese eng an ein Modell der menschlichen Denkweise angelehnte Idee geht davon aus, dass sich der Agent anhand seines Wissens (Beliefs) über die Umwelt verschiedene Handlungsoptionen (Desires) überlegt. Für einige dieser Optionen entscheidet sich der Agent und verfolgt sie in Zukunft, er hat also Absichten (Intentions). Entsprechend der Intentionen agiert der Agent nach der Maßgabe, mit welchen Mitteln und Aktionen er seinen Intentionen gerecht werden kann.

Subsumption Eine Subsumptionsarchitektur verwendet ein priorisiertes Reiz-Reaktions-Schema, modelliert also einen rein reaktiven Agenten. Reize höherer Priorität sorgen dafür, dass alle Reaktionen auf niedriger eingestufte Reize ausgesetzt werden, bis der höhere Reiz nicht mehr vorliegt. Die Idee wurde ursprünglich von Brooks in [Bro1986] angeregt.

Aus Sicht der Softwaretechnik oder der verteilten Systeme versteht man unter Architektur eher den technischen und logischen Aufbau des gesamten Systems, von der Hardware bis zur Software, es werden also die Elemente einer Agentenplattform und deren Beziehungen betrachtet. Im Blickfeld stehen Fragen, wie Agenten auf der Plattform ausgeführt werden, wie die Kommunikationsschnittstellen aussehen oder welche weiteren Dienste die Plattform den Agenten zur Verfügung stellt. Eine solche Architektur kann auch eine Entscheidungsarchitektur einschließen, weil mitunter die Ausführungstechnik von Agenten den internen Aufbau und damit den Entscheidungsprozess des Agenten vorgibt. Wenn in dieser Arbeit die softwaretechnische Sicht auf ein Multiagentensystem betont werden soll, benutze ich den Begriff *Plattformarchitektur*.

Im Kapitel 4 werden einige Spezifikationen der FIPA (Foundation for Intelligent Physical Agents) vorgestellt, die ebenfalls eine Architektur eines Multiagentensystems beschreiben, sogar mit der Steigerung einer „abstrakten Architektur“. Dabei liegt der Fokus auf den Kommunikationsstrukturen zwischen den Agenten eines Multiagentensystems, daher soll in dieser Arbeit der Begriff *Kommunikationsarchitektur* diese Sichtweise kennzeichnen. Die Bezeichnung *abstrakte Architektur* verwende ich als feststehenden Begriff für die „FIPA Abstract Architecture Specification“ [FIPA00001].

Die im folgenden Abschnitt 3.2 beschriebene MULAN-Architektur beschreibt ein Multiagentensystem in einer ganzheitlichen Sicht, die sowohl die Plattformarchitektur als auch die Entscheidungsarchitektur einschließt. Beide Teilarchitekturen sind aus MULAN-Sicht austauschbar, über die Kommunikationsarchitektur werden – abgesehen von der Festlegung auf Nachrichtenkommunikation – keine Aussagen gemacht. So eine Architektur, die das ganze System inklusive aller Teilarchitekturen abdeckt, werde ich *Multiagentensystem-Architektur* nennen.

Das Ziel dieser Arbeit ist die Entwicklung einer Agentenplattform, daher liegt die Verwendung des Architekturbegriffs aus der softwaretechnischen Plattformsicht nahe. Vor allem im Kapitel 5, welches die erstellte Plattform dokumentiert, ist daher unter einer Architektur gemeinhin eine Plattformarchitektur zu verstehen.

3.2 Multiagentennetze (MULAN)

Die mit dieser Arbeit bereitgestellte Agentenplattform (siehe Kapitel 5) ist in die Multiagentensystem-Architektur MULAN (siehe [Röl2002] und [KMR2001]) eingebettet. MULAN steht für **Mul**ti**a**genten**n**etze und bildet ein vollständiges Multiagentensystem mit Referenznetzen (siehe Abschnitt 2.3) ab.

Die Wahl von Referenznetzen für die Modellierung eines Multiagentensystems begründet sich zum einen darin, dass viele für Multiagentensysteme interessante Konzepte wie Nebenläufigkeit und Synchronisation, Lokalität und Mobilität, Objektorientierung und dynamisches Verhalten von diesem Petrinetzformalismus abgedeckt werden. Zum anderen erlauben die Referenznetze Dank ihrer operationalen Semantik die Ausführung des Modells. Damit kann das zunächst abstrakt spezifizierte System unter Beibehaltung der Repräsentierungssprache (d.h. weiterhin mit Netzen) ausführbar gemacht werden.

Das Mulan-Modell eines Multiagentensystems und die in diesem Modell getroffenen Designentscheidungen werden zunächst im groben Zusammenhang beschrieben. Darauf folgen detaillierte Erläuterungen der Modellierung von Agenten und Plattformen in Mulan. Da das Mulan-Modell auch eine Entscheidungsarchitektur beinhaltet, wird die Modellierung des Agentenverhaltens in Abschnitt 3.2.3 erläutert.

3.2.1 Gesamtmodell

Mulan modelliert ein Multiagentensystem für rein kommunizierende Agenten, also Software-Agenten, die keine andere Umwelt haben als ihre Kommunikationspartner (siehe Diskussion in Abschnitt 3.1.1). Dementsprechend werden Plattformen zur Modellierung von Orten herangezogen (siehe Abschnitt 3.1.3).

Bei der Betrachtung des Mulan-Modells, das ja aufgrund der Verwendung der Referenznetze auch ausführbar ist, muss zwischen den beiden Sichtweisen, der abstrakten Modellsicht und der operationalen Auswirkung, unterschieden werden. So steht zum Beispiel die Modellierung von Orten als Plattformen mit einer explizit gegebenen Kommunikationsstruktur in der Modellsicht durchaus als Repräsentierung physikalisch verschiedener Orte. In der operationalen Ausführung des Modells spielt sich aber alles auf einer einzigen physischen Maschine ab, weil die Simulation der Netze dort stattfindet.

In Abbildung 3.1 sind die vier Ebenen der Mulan-Architektur zu sehen. Die oberste Ebene (oben links im Bild) stellt einen groben Blick auf die gesamte Struktur des *Multiagentensystems* dar, wobei die Plattformen und die Kommunikationsstrukturen zwischen den Plattformen modelliert werden. Die Struktur in diesem Bild ist exemplarisch, abhängig vom abzubildenden System sind andere Plattformen und Verbindungen zu modellieren.

Jede Plattform wird durch eine Marke auf einer Stelle des Systemnetzes repräsentiert, wobei die Marke wiederum eine Netzinstanz ist. Die Kommunikationsstruktur zwischen den Plattformen wird durch Transitionen modelliert, die über synchrone Kanäle (die entsprechenden Anschriften sind in Abbildung 3.1 der Übersichtlichkeit halber ausgeblendet worden) zwei Plattformen kurzfristig miteinander koppeln, um Nachrichten auszutauschen oder Agenten zu bewegen.

Alle Netzinstanzen, die die Plattformen repräsentieren, basieren auf einem gemeinsamen Netz, das oben rechts im Abbildung 3.1 zu sehen ist. Der mit ZOOM beschriftete Pfeil soll die Vergrößerung der Marke symbolisieren (was nicht mit einer Vergröberung der die Marke enthaltenden Stelle verwechselt werden darf).

Das *Plattformnetz* enthält eine zentrale Stelle, in der alle Agenten (wiederum durch Marken repräsentiert, die Netzinstanzen sind) gelagert werden, die sich auf dieser Plattform befinden. Die mit der Stelle verbundenen Transitionen beschreiben die primitiven Dienste, die eine Plattform den Agenten zur Verfügung stellen muss. Dazu gehören Kommunikationsmöglichkeiten und die Möglichkeit, Agenten zu erzeugen und zu terminieren.

Die Agentenmarken im Plattformnetz sind wiederum Instanzen eines *Agentennetzes* (in Abbildung 3.1 rechts unten). Alle Agenten haben prinzipiell den gleichen Aufbau,

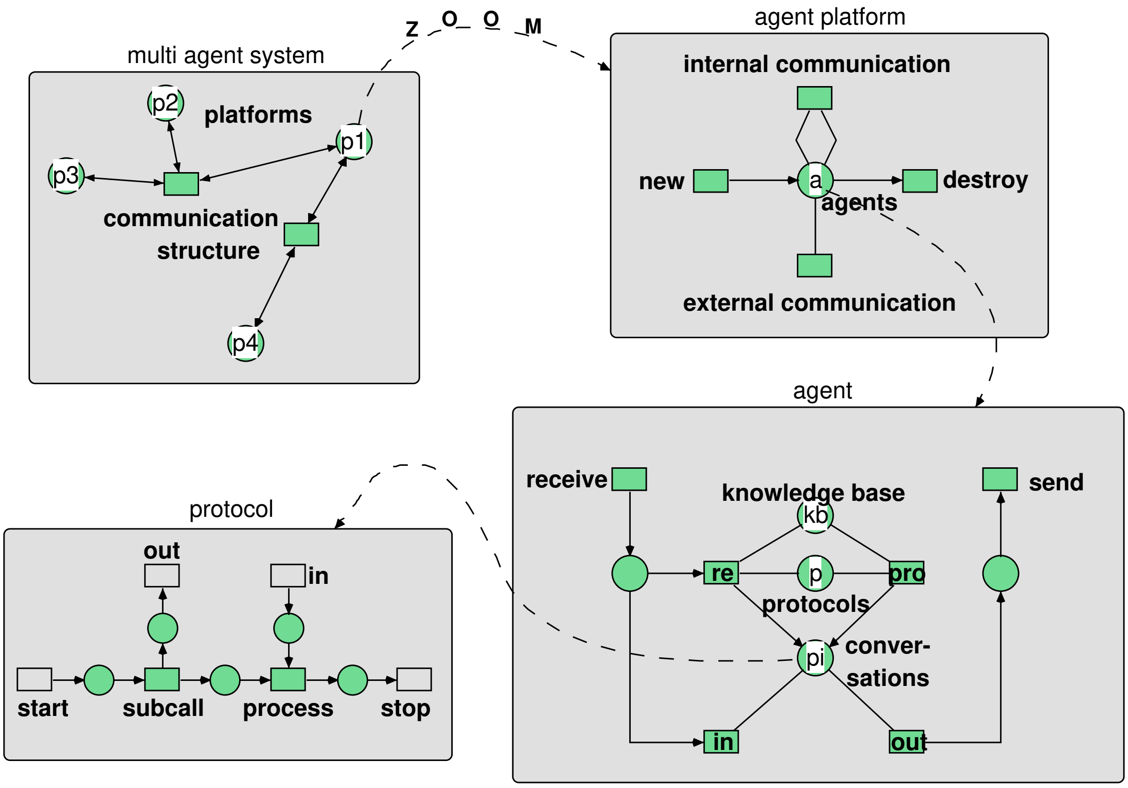


Abbildung 3.1: Überblick über die MULAN-Architektur

so dass dieses Netz vom Entwickler eines anwendungsspezifischen Multiagentensystems nicht verändert zu werden braucht. Allerdings legt dieses Netz die Entscheidungsarchitektur des Agenten fest. Das MULAN-Modell bietet hier die Möglichkeit, für andere kognitive Modelle andere Agentennetze zu verwenden, für die dann die folgenden Aussagen über Wissen, Protokolle und Konversationen nicht mehr zutreffen. Zwingend eingehalten werden muss lediglich die Schnittstelle zwischen Agentennetz und Plattformnetz, die auf die synchronen Kanäle zum Senden und Empfangen von Nachrichten festgelegt ist.

Der Zustand eines Agenten setzt sich aus zwei Teilen zusammen: dem Wissen, welches der Agent längerfristig explizit ansammelt und bewahrt, sowie den dynamisch anfallenden Informationen bei der Ausführung des Agenten (also der Markierung der Agentennetze). Das längerfristige Wissen wird von einem MULAN-Agenten in einer Wissensbasis abgelegt. Deren Modellierung lässt die Architektur mit Rücksicht auf die unterschiedlichen Bedürfnisse der Anwendungsgebiete offen.

Das Verhalten von Agenten wird in der MULAN-Architektur durch *Protokolle* modelliert. Protokolle sind wiederum Netze, die dem Agenten zur Verfügung stehen. Wenn der Agent irgendetwas tun oder denken möchte – sei es als Reaktion auf eine ankommende Nachricht, oder auch proaktiv, d.h. von sich aus – wählt er ein passendes

Protokollnetz aus und erzeugt eine Netzinstanz davon. Ein Beispiel für ein instanziiertes Protokoll, in der Mulan-Terminologie Konversation genannt, ist in Abbildung 3.1 links unten zu sehen.

Mittels der Protokollnetze und der Wissensbasen der Agenten wird der dynamische Aspekt des Multiagentensystems modelliert. Zum einen beschreibt der Entwickler eines Multiagentensystems durch die Spezifikation von Protokollnetzen das Agentenverhalten und die von den Agenten zu versendenden und zu erwartenden Nachrichten. Zum anderen kann sich das Wissen der Agenten und damit auch die Menge der einem Agenten bekannten Protokolle zur Laufzeit ändern, so dass adaptive oder lernfähige Agenten modellierbar sind.

Die vier Ebenen der Mulan-Architektur – System, Plattform, Agent und Protokoll – beschreiben ein Multiagentensystem von der äußersten bis zur innersten Sichtweise. Diese „sichtbar-in“-Beziehung wird durch das Netze-in-Netzen-Paradigma intuitiv wiedergegeben: Die Plattformen liegen als Marken im System, die Agenten wiederum liegen in der Plattform, und die Konversationen laufen im Agenten ab.

Die Kommunikation zwischen den jeweils benachbarten Ebenen wird durch synchrone Kanäle hergestellt, wobei auch eine „Weitervermittlung“ über mehrere Kanäle, z.B. vom Agenten über die Plattform zur Kommunikationsstruktur des Systems und zurück zu einem anderen Agenten auf einer anderen Plattform, möglich ist.

Die Mulan-Architektur ist durch und durch auf Nebenläufigkeit und Verteilung ausgelegt. Die Verteilung wird nur in der obersten Ebene, dem Systemnetz, explizit beschrieben, Nebenläufigkeit ist aber auf allen Ebenen vorhanden. Innerhalb einer Plattform können beliebig viele Agenten nebeneinander existieren und unabhängig voneinander arbeiten und kommunizieren. Auch innerhalb des Agenten ist das nebenläufige Führen mehrerer Konversationen möglich, auch die Initiierung von Konversationen kann nebenläufig geschehen. Synchronisation ist nur dann nötig, wenn innerhalb eines Agenten zwischen verschiedenen Handlungssträngen Informationen ausgetauscht werden müssen. Der Synchronisationspunkt ist dann die Wissensbasis des Agenten, die – je nach Aufbau – unterschiedlich feine Synchronisationsgrade beim Modifizieren und Abfragen von Informationen bieten kann.

3.2.2 Agentenaufbau

Die Agenten werden in der Mulan-Architektur durch ein Netz wie das in Abbildung 3.1 rechts unten gezeigte modelliert. Die Idee ist, dass alle Agenten in einem homogenen Multiagentensystem durch Instanzen desselben Netzes repräsentiert werden. Allerdings sind auch heterogene Systeme modellierbar, indem die Agenten durch unterschiedliche Netzmuster repräsentiert werden. Im Folgenden geht es ausschließlich um homogene Systeme.

Die grobe Grundstruktur aller Agentennetze ist in Abbildung 3.2 zu sehen. Die definierte Schnittstelle zwischen Agent und Plattform umfasst zwei synchrone Kanäle, je einen zum Senden und zum Empfangen von Nachrichten. Die Schnittstelle wird durch die orange hervorgehobenen Transitionen „send“ und „receive“ realisiert.

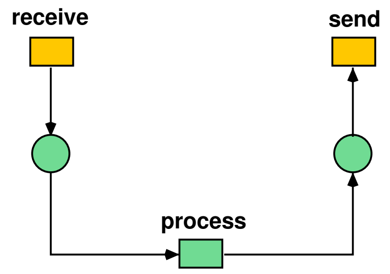


Abbildung 3.2: Agenten-Grundmuster

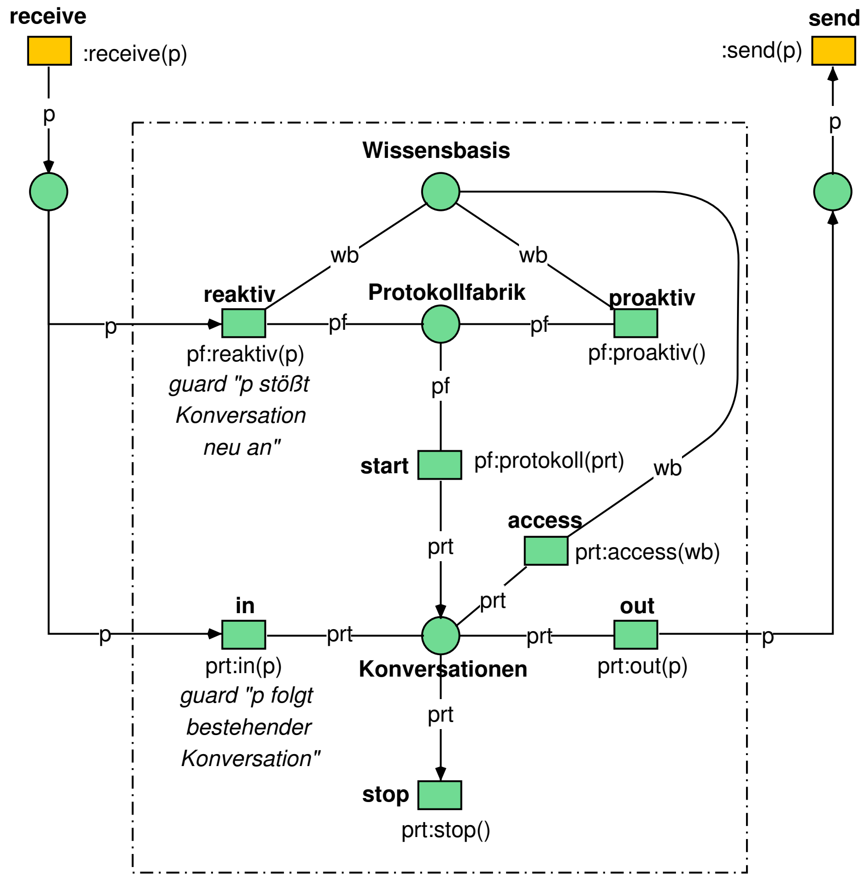


Abbildung 3.3: Ein verfeinertes MULAN-Agentennetz

Um asynchrone Nachrichtenübertragung mit den synchronen Kanälen zu modellieren, werden alle ein- und ausgehenden Nachrichten vom Agenten gepuffert. Die Stellen unter den beiden Transitionen modellieren diese Puffer. Die Pufferung könnte auch in der Plattform und der systemweiten Kommunikationsstruktur vorgenommen werden, ist aber im abstrakten Mulan-Modell auf die Agentenebene verlagert worden, um dessen Autonomie beim Nachrichtenempfang zu modellieren.

Die „process"-Transition steht stellvertretend für alle möglichen reaktiven, proaktiven oder inaktiven Verhaltensweisen des Agenten. Da die strenge Interpretation dieser Transition nur ein rein reaktives, gedächtnisloses Verhalten des Agenten zulässt, muss sie entsprechend des gewünschten kognitiven Modells verfeinert werden.[1]

Ein verfeinertes Agentennetz, das dem vorgeschlagenen, protokollbasierten Verhaltensmodell der Mulan-Architektur folgt, ist in Abbildung 3.3 dargestellt. Der gestrichelte Kasten umreißt die ehemalige, jetzt verfeinerte „process"-Transition.

Der Agent besitzt eine Wissensbasis, in der er Informationen über seinen Zustand längerfristig ablegen und dann abrufen oder ändern kann. Die Wissensbasis wird wiederum durch ein Netz modelliert, dessen Aufbau und Arbeitsweise sind allerdings nicht festgelegt. Sie kann von Agent zu Agent verschieden aufgebaut sein, je nach den anwendungsspezifischen Bedürfnissen des Agenten.

Mittels der Transitionen „reaktiv" und „proaktiv" kann der Agent ein weiteres Netz, die sogenannte Protokollfabrik, anweisen, eines der dem Agenten bekannten Protokolle auszuwählen und zu instanziieren, um eine Konversation zu führen. Während bei der proaktiven Transition die Wissensbasis die einzige Nebenbedingung zur Auswahl eines Protokolls ist, wird bei der reaktiven Protokollauswahl eine eingegangene Nachricht berücksichtigt – sofern die Nachricht nicht einer bereits laufenden Konversation zugeordnet werden kann.

Hat die Protokollfabrik ein Protokollnetz instanziiert, so wird es über die Transition „start" in die Stelle mit den laufenden Konversationen gelegt. Von hier aus kann die Konversation über synchrone Kanäle eingehende Nachrichten entgegennehmen (Transition „in"), Nachrichten versenden („out") oder auf die Wissensbasis des Agenten zugreifen („access"). Wenn eine Konversation beendet ist, kann sie mittels der Transition „stop" aus der Stelle entfernt werden.

3.2.3 Agentenverhalten: Protokolle

Mittels Protokollen wird in der Mulan-Architektur – sofern keine alternativen Überlegungs- und Entscheidungsmodelle zum Einsatz kommen – das Verhalten der Agenten beschrieben. Instanzen der Protokollnetze, in der Mulan-Terminologie Konversationen genannt, sorgen bei ihrer Ausführung dafür, dass der Agent seinen Zustand ändert, Nachrichten versendet, auf Nachrichten wartet oder andere Aktionen durchführt, die zur Anwendung des Multiagentensystems beitragen. Die Bezeichnung Konversation

[1] Die durchzuführende Verfeinerung darf natürlich *nicht* verhaltensbewahrend sein, wenn kein rein reaktiver Agent modelliert werden soll.

geht darauf zurück, dass in vielen Fällen das Protokollnetz eine Hälfte eines Dialogs zwischen zwei Agenten bestimmt.

Den Protokollnetzen stehen fünf synchrone Kanäle zum Agenten zur Verfügung, die Konversationsbeginn und -ende, Nachrichtenempfang und -versand sowie den Wissensbasiszugriff ermöglichen. Mithilfe dieser Kanäle und der gesamten Ausdrucksmächtigkeit der Referenznetze kann jedes beliebige Verhalten des Agenten implementiert werden.

Trotz der recht suggestiven Namen Protokoll und Konversation sind alle denkbaren Aufgaben modellierbar. Kein Protokoll ist gezwungen, Nachrichten entgegenzunehmen oder zu produzieren, es kann auch einfach „still" arbeiten und die Wissensbasis analysieren und anpassen. Protokolle können nicht direkt miteinander zusammenarbeiten, aber über die gemeinsam genutzte Wissensbasis des Agenten Informationen austauschen. Konversationen müssen nicht zwingend beim Ende eines Dialogs terminieren, sie können auch beliebig – sogar unendlich – oft weiterarbeiten und neue Nachrichten produzieren oder empfangen.

3.2.4 Agentenumgebung: Plattform

Die Plattform der Mulan-Architektur (siehe Abbildung 3.1, rechtes oberes Netz) stellt den auf ihr arbeitenden Agenten grundlegende Dienste zur Verfügung: Agenten können erzeugt (Transition „new") und terminiert („destroy") werden. Ferner können zwei Agenten in der Plattform mithilfe der Transition „internal communication" und den dazugehörigen synchronen Kanälen kommunizieren, d.h. Nachrichten austauschen. Die Transition „external communication" benötigt auf der Plattform nur einen Agenten, der über synchrone Kanäle und die Kommunikationsstruktur des Systemnetzes mit einem Kommunikationspartner auf einer anderen Plattform verbunden wird.

Weitere Dienste werden von Agenten auf der jeweiligen Plattform angeboten. Diese Dienste sind somit in der statischen Struktur der Architektur nicht sichtbar. Sie können zur Laufzeit ein- und ausgeklinkt werden und erlauben auch eine Modellierung von Diensten, die nur auf bestimmten Plattformen verfügbar sind.

Soll Mobilität von Agenten modelliert werden, so stehen mindestens zwei Varianten offen: Entweder wird das Plattformnetz um zusätzliche Transitionen zum Empfangen und Versenden von Agenten erweitert, die über die synchronen Kanäle der Kommunikationsstruktur des Systems die Agenten-Marke zu einer anderen Plattform befördern können. Diese sehr intuitive Modellierung ist eine direkte Umsetzung der Idee, Mobilität durch Markentransport in Petrinetzen darzustellen.

Alternativ kann aber auch eine eher generische Multiagentensystemlösung auf die Mulan-Architektur übertragen werden: Häufig wird Mobilität von Agenten dadurch ausgedrückt, dass der Agent seinen Code und seinen Zustand (in diesem Fall also seine Protokollnetze und seine Wissensbasis) zu einer anderen Plattform schickt, auf der ein neuer Agent mit genau diesen Informationen erzeugt wird. Anschließend terminiert der ursprüngliche Agent auf der ersten Plattform. Somit läuft nur noch die Kopie des Agenten auf der zweiten Plattform, also hat sich der Agent bewegt.

Diese Form der Mobilität erfordert keine Änderungen am Plattformnetz, benötigt aber die Mitarbeit des sich bewegenden Agenten. Außerdem kann nicht der gesamte Zustand des Agenten auf diese Weise transferiert werden (wie es bei einer einfachen Markenbewegung ohne Probleme möglich ist), sondern nur jene Informationen, die der Agent explizit überträgt.

3.2.5 Ausführbarkeit

Die Mulan-Architektur mit den vier Ebenen ist in der Lage, ein Multiagentensystem in allen Aspekten, von der groben Systemsicht bis zur Verhaltensmodellierung der Agenten, zu beschreiben. Durch die Verwendung der Referenznetze zur Spezifikation von Mulan ist das Modell auch ausführbar. Wenn entsprechende anwendungsspezifische Protokolle und Wissensbasen erstellt werden, entsteht eine im Referenznetzsimulator lauffähige Anwendung.

Durch die Ausführung des gesamten Modells in einem Simulator kommt aber der Verteilungsaspekt nicht zum Tragen. Zwar modelliert die Mulan-Architektur mit der Spezifikation der Kommunikationsstruktur im Systemnetz explizit ein verteiltes System, aber die Ausführung findet in *einem* Simulator, also auf *einer* Maschine statt.

Um ein reales, physisch verteiltes System zu erhalten, muss also die modellierte Kommunikation im Systemnetz durch reale Kommunikation im Netzwerk ersetzt werden. In diesem Zusammenhang ist es wünschenswert, nicht nur die Kommunikation zwischen verschiedenen Mulan-Plattformen über das Netzwerk zu ermöglichen, sondern auch die Verbindung zu fremden Agentenplattformen, die gänzlich anders implementierten Multiagentensystem-Architekturen entstammen, herzustellen.

Kommunikation zwischen den Agentenplattformen verschiedener Hersteller ist nur möglich, wenn gemeinsame Schnittstellen definiert sind, am besten in international anerkannten Standards. Solche Standards werden z.B. von der FIPA (siehe nächstes Kapitel 4) herausgegeben. Sie schreiben bestimmte Gemeinsamkeiten in den Schnittstellen der Agentenplattformen vor. Letztendlich muss also, um die plattformübergreifende Kommunikation zu ermöglichen, die Agentenplattform angepasst werden. Dies ist die Aufgabe dieser Arbeit, die Vorgehensweise und das Ergebnis werden in Kapitel 5 beschrieben.

Kapitel 4

FIPA-Standards

Wenn zwei Agenten miteinander kommunizieren wollen, so müssen sie eine gemeinsame Sprache sprechen. Solange die Agenten vom selben Hersteller stammen, kann (und muss) dieser dafür Sorge tragen, dass eine gemeinsame Verständigungsgrundlage besteht. Sollen aber Agenten von verschiedenen Herstellern miteinander kommunizieren – am besten sogar mit Partnern, die zur Erschaffungszeit noch gar nicht bekannt sind – so braucht es einen Standard, an den sich ein Agentenentwickler halten kann. So ein allgemein gültiger und weit verbreiteter Standard ermöglicht es allen Agenten, die sich an den Standard halten, zumindest grundlegende Kommunikation abzuwickeln – auch wenn dabei eventuell nur die Tatsache kommuniziert wird, dass zwei Agenten sich doch nicht verstehen, weil sie über ein zu stark abweichendes Hintergrundwissen verfügen.

Die „**F**oundation for **I**ntelligent **P**hysical **A**gents" (FIPA) hat sich genau die Schaffung einer solchen Kommunikationsgrundlage zum Ziel gesetzt. Die FIPA ist eine internationale Organisation unter Beteiligung von Firmen und Universitäten, die seit 1996 „Software-Standards für heterogene und interagierende Agenten und agentenbasierte Systeme" [FIP2002, `/about/index.html`] erarbeitet. Die seitdem von der FIPA veröffentlichten Spezifikationen decken viele Bereiche der Kommunikation zwischen Agenten ab und werden ständig iterativ überarbeitet.

Eine FIPA-Spezifikation kann sich in einem von fünf Stadien befinden: Zu Beginn ist sie ein vorläufiger („preliminary") Entwurf eines technischen Ausschusses und wird dann vom FIPA-Architektur-Gremium als experimentell („experimental") akzeptiert. In diesem Stadium dürfen nur noch geringfügige Änderungen vorgenommen werden, aber erst nach erfolgreicher Umsetzung in FIPA-konformen Plattformen kann die Spezifikation als „Standard" etabliert werden. Die verbleibenden zwei Stadien sind für abgesetzte („deprecated") oder endgültig veraltete („obsolete") Spezifikationen vorgesehen.

Während 1997 (und mit Überarbeitung auch 1998) ein mehrteiliges Dokument mit dem Titel „FIPA97" (bzw. „FIPA98") veröffentlicht wurde, welches alle Bereiche abdeckte, bringt die FIPA seit zwei Jahren unter dem Sammelbegriff „FIPA2000" einzelne, durch Nummern identifizierte Spezifikationen zu jeweils einem abgegrenzten Bereich heraus. Diese Spezifikationen standardisieren zusammengenommen die verschiedenen Aspekte der Agentenkommunikation auf einer FIPA-konformen Agentenplattform. Im folgenden wird dieser Satz von Spezifikationen als „konkrete FIPA-Architektur“ oder „FIPA2000-Standards“ bezeichnet. Neben diesen konkreten Spezifikationen steht die im Jahre 2001 veröffentlichte „Abstract Architecture“ [FIPA00001], die auf einer verallgemeinerten Ebene die notwendigen Elemente einer FIPA-konformen Architektur definiert. Im Sinne der in Abschnitt 3.1.4 diskutierten Differenzierung des Architekturbegriffs handelt es sich sowohl bei der abstrakten als auch bei der FIPA2000-Architektur um Kommunikationsarchitekturen, wobei beide auch einen groben Rahmen für Plattformarchitekturen vorgeben.

Die in den FIPA2000-Spezifikationen und der abstrakten Architektur definierten Konventionen werden in diesem Kapitel vorgestellt, soweit sie die Agentenplattform direkt betreffen. Die vorgestellten Spezifikationen sind – wenn nicht anders erwähnt – im Stadium „experimental“. Da es bisher noch keine FIPA-Dokumente im Status eines Standards gibt, wird in dieser Arbeit der Begriff „Standard“ hin und wieder als Synonym für „Spezifikation“ benutzt.

4.1 Überblick

Der Fokus der FIPA-Spezifikationen liegt auf der Kommunikation von Agenten, die sich auf verschiedenen Plattformen befinden. Agenten können innerhalb einer Plattform auf beliebigen Wegen kommunizieren, diese interne Kommunikation ist von den FIPA-Standards nicht zwingend betroffen. So sollen bei der internen Kommunikation eventuelle Vorteile aus plattformspezifischen Besonderheiten genutzt werden können, während die plattformübergreifende Kommunikation zum Zwecke der Interoperabilität auf einen gemeinsamen Nenner reduziert werden muss.

Die plattformübergreifende Kommunikation wird in vielen Aspekten standardisiert, die in diesem Abschnitt in ihrer Zusammenarbeit vorgestellt werden sollen. Die weiteren Abschnitte dieses Kapitels werden sich mit den einzelnen Spezifikationen detaillierter auseinandersetzen. Dieser Überblick richtet sich nach der Einteilung, welche die FIPA für die themenorientierte Navigation in ihrer Spezifikationsdatenbank verwendet (siehe Abbildung 4.1).

Von der Betrachtung in dieser Arbeit ausgenommen werden jene FIPA-Spezifikationen, die konkrete Gebiete für die Anwendungen („Applications“) von Agentensystemen im Blickfeld haben. Für diese Spezifikationen steht der oberste Block der Abbildung 4.1. Sie definieren, aufsetzend auf den Kommunikationsstandards der anderen FIPA-Spezifikationen, konkrete Dienste und Ontologien zu Themen wie Audiovisueller Unterhaltung, persönlichen Assistenten, Netzwerkmanagement usw. Da diese Spezifikationen in erster Linie Agenten modellieren und nicht die Funktionalität der Agentenplattform selber beeinflussen, wird hier nicht weiter auf sie eingegangen.

Der zweite Block in Abbildung 4.1 steht für die abstrakte Architektur („Abstract Architecture“), welche – wie bereits einleitend in diesem Kapitel erwähnt wird – nachträglich neben oder über die anderen Spezifikationen der FIPA2000-Architektur gestellt wurde. Da eine abstrakte Architektur eigentlich keine Elemente einführen sollte, die nicht auch in der konkreten Spezifikationen auftauchen, wird die abstrakte Sicht

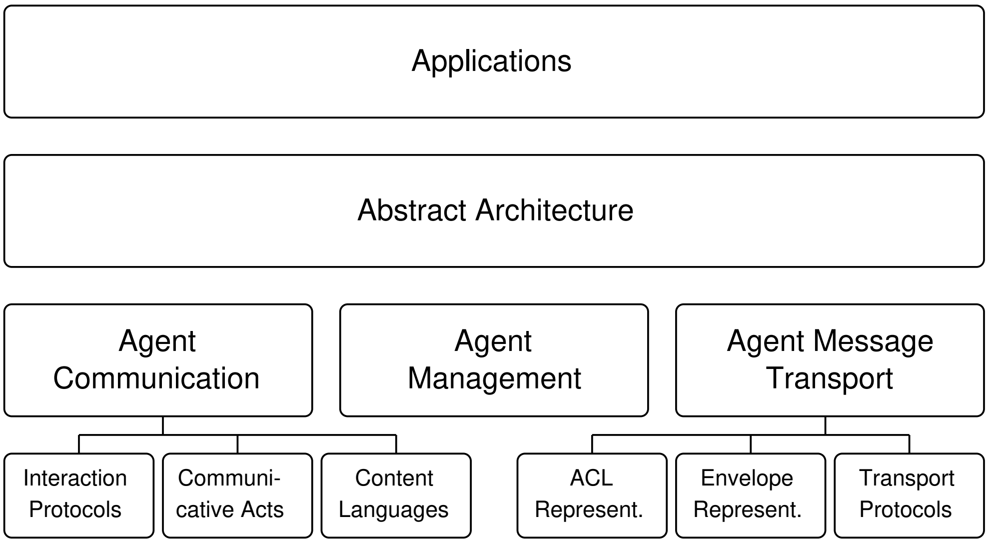


Abbildung 4.1: Thematische Einteilung der FIPA-Spezifikationen (nach [FIP2002, `/repository/bysubject.html`])

hier im Überblick nur dann erwähnt, wenn sie nicht zur konkreten Architektur passt – was leider vorkommt.

Der Schwerpunkt dieses Überblicks liegt also auf den drei Aspekten, die konkret geregelt werden müssen, wenn Agenten auf unterschiedlichen Plattformen mittels Nachrichten kommunizieren können sollen: Die Agenten müssen eine einheitliche „Agent Communication Language“ (ACL) mit bekannter Syntax, Semantik und Pragmatik sprechen. Weiterhin muss jeder Agent auf definierten Wegen erreichbar sein, d.h. sowohl das Beförderungsverfahren für Nachrichten („Agent Message Transport“) als auch die Adressverwaltung („Agent Management“) sollten bekannten Regeln gehorchen.

Agent Communication. Zwischen den Agenten wird mittels Nachrichten kommuniziert. Die FIPA-ACL definiert eine Nachricht als einen kommunikativen Akt („Communicative Act“, angelehnt an die Sprechakt-Theorie von Searle, siehe [Sea1969]). Ein kommunikativer Akt besteht aus zwei Teilen, einer inhaltlichen Aussage und einem sogenannten Performativ.

Das Performativ klärt, ob es sich bei der Nachricht um eine Mitteilung, eine Aufforderung, eine Frage oder etwas Anderes handelt. Die FIPA bietet in der „Communicative Act Library“ [FIPA00037] einige häufig gebrauchte, mit Bezeichner und Semantik fertig definierte Performative an, die zur Kommunikation verwendet werden können (siehe Abschnitt 4.2.1). Um die Kommunikationsmöglichkeiten der Agenten nicht zu sehr einzuengen, ist die Verwendung selbstdefinierter Performative erlaubt, allerdings sollten diese sich von bestehenden Typen von kommunikativen Akten sowohl in Benennung als auch Bedeutung und Verwendungszweck ausreichend unterscheiden.

Für den inhaltlichen Teil des kommunikativen Aktes muss wiederum eine Konvention bezüglich Syntax und Semantik vorgegeben werden, um ein gegenseitiges Verstehen zu ermöglichen. Allerdings ist dabei eine einzige, allgemeingültige Regelung nicht praktikabel, denn diese würde entweder nicht alle Situationen abdecken können oder aber unhandhabbar gross werden. Daher sieht die FIPA-ACL hier einen Mechanismus vor, der es den Agenten erlaubt, verschiedene Darstellungen zu verwenden. In der Nachricht muss lediglich durch eindeutige Kennzeichnung ausgewiesen werden, welche Sprache („Content Language“, siehe Abschnitt 4.2.3) mit welcher Ontologie (siehe Abschnitt 4.2.4) verwendet wird. Bei der Wahl der Inhaltssprache hilft die FIPA insoweit, als dass bereits eine kleine Auswahl von Sprachen in einzelnen Spezifikationen definiert wurde. Für die domänenspezifische Ontologie des Nachrichteninhalts kann die FIPA diese Standardisierungsleistung nicht erbringen. Es gibt aber eine Ontologiedienst-Spezifizikation, welche eine zentrale Ontologieverwaltung umreißt, die auch Bezüge zwischen verschiedenen Ontologien herstellen können soll.

Häufig reicht ein kommunikativer Akt alleine nicht aus, um ein Ziel zu erreichen, z.B. wenn eine Auktion mit zwei oder mehr beteiligten Agenten ablaufen soll. Zu diesem Zweck hat die FIPA eine Reihe von Interaktionsprotokollen definiert, die eine Abfolge von Akttypen für bestimmte Zwecke vorgeben („Interaction Protocols“, siehe Abschnitt 4.2.2). Für Interaktionsprotokolle gilt ebenso wie für Performative und Inhaltssprachen dass die Verwendung eigener, proprietärer Interaktionsprotokolle möglich ist.

Zusätzlich zur übereinstimmenden Verwendung und Bedeutung von Nachrichten muss jede Nachricht, damit sie von einem fremden Agenten verarbeitet und verstanden werden kann, sowohl eine bekannte Struktur aufweisen als auch in einer bekannten Kodierung abgefasst sein.

Die Struktur wird in der „ACL Message Structure Specification“ [FIPA00061] festgeschrieben (siehe Abschnitt 4.2.5). Eine Nachricht wird gemäß der ACL-Definition aus einer Reihe benannter Elemente zusammengesetzt, die u.a. das mit der Nachricht intendierte Performativ, die dazugehörige inhaltliche Aussage (nebst Benennung der dafür verwendeten Darstellungsform), das den Rahmen gebende Interaktionsprotokoll, einige weitere Informationen zur Konversationskontrolle sowie die Namen und Adressen der Kommunikationspartner umfassen.

Für die Kodierung bzw. Repräsentierung der gesamten Nachricht bietet die FIPA einen erweiterbaren Satz von Spezifikationen an, jeweils eine Spezifikation für eine Darstellungsform (siehe Abschnitt 4.2.6). In Abbildung 4.1 ist die Repräsentierung der Nachrichten („ACL Representation“) zwar dem Transportbereich zugeschlagen worden, weil der Nachrichtentransport auf eine feste Kodierung angewiesen ist, aber in dieser Arbeit wird die Kodierung durchaus noch als Teil der Syntaxdefinition betrachtet.

Message Transport. Für eine reale Nachrichtenkommunikation braucht es auf der technischen Seite einen Transportmechanismus, welcher Nachrichten von einem Agenten entgegen nimmt und gemäß dessen Wünschen anderen Agenten zustellt. Um Nachrichten von fremden Agenten bzw. Plattformen annehmen zu können, müssen gemeinsame Konventionen über das Transportprotokoll als auch über die Anschlusspunkte auf der Plattform etabliert sein.

Was die Transportprotokolle („Transport Protocols“) angeht, so bietet die FIPA zu diesem Thema eine Reihe alternativer Spezifikationen für verschiedene Protokolle an. Mit dem Protokoll wird natürlich auch die äußere Schnittstelle festgelegt, durch die externe Nachrichten das Transportsystem erreichen bzw. verlassen können. Ebenfalls durch das Transportprotokoll festgelegt wird wie Wahl der Kodierung für Nachrichten („ACL Representation“) und Nachrichtenumschläge („Envelope Representation“).

Bei der Standardisierung der Innenseite der Schnittstelle geht allerdings das Modell der abstrakten Architektur andere Wege als die in der konkreten FIPA2000-Spezifikationen beschrittenen. Während die konkrete Spezifikation je Plattform eine zentrale Weiterleitungsinstanz namens „Agent Communication Channel“ (ACC) vorsieht und auf eine Standardisierung der internen Transportschnittstelle zum Agenten hin verzichtet, sieht die abstrakte Architektur einen Transportdienst vor, bei dem sich jeder Agent direkt für passende Transporte („transports“, vielleicht in diesem Zusammenhang am besten als „Verkehrsmittel“ übersetzbar) registrieren kann. Die beiden Transportsystem-Modelle werden in den Abschnitten 4.4.1 und 4.3.2 beschrieben. Eine Zuordnung der Elemente der beiden Konzepte aufeinander wird bei der FIPA gerade erarbeitet und soll demnächst fertig werden (siehe [WLDG2001]) .

Agent Management. Zur Nachrichtenkommunikation über ein Transportsystem gehört auch ein Schema, nach dem Agenten adressiert werden können. Eine Konvention über die Struktur der Adressen alleine reicht aber nicht; es muss auch möglich sein, auf irgendeine Art die Adressen fremder Agenten in Erfahrung zu bringen – eine Auskunft bzw. ein Verzeichnisdienst wird benötigt.

Die abstrakte Architektur beschränkt sich in diesem Punkt auf die Definition der minimal benötigten Elemente (siehe Abschnitt 4.3.3): Ein Agent muss einen eindeutigen, unveränderlichen Namen haben, der in Nachrichten als Absender- und Empfängerangabe verwendet werden kann und muss. Ein zwingend vorhandener Verzeichnisdienst verwaltet Verzeichniseinträge, welche neben dem notwendigen Namen auch weitere Attribute über den Agenten aufnehmen können, wie z.B. die vom Agenten angebotenen Dienste.

Diese Grundelemente werden in der „Agent Management Specification“ [FIPA00023] aus der FIPA2000-Suite ebenfalls angeführt, aber durch deutlich konkretere Plattformkomponenten ergänzt (siehe Abschnitt 4.4.2). Es gibt zwei Verzeichnisdienste, die verschiedene Aufgaben wahrnehmen: Das „Agent Management System“ (AMS) einer Plattform führt Telefonbuch-artig alle Agenten auf (mit Vollständigkeitsanspruch für die lokale Plattform), während ein „Directory Facilitator“ (DF) im Stile der „Gelben Seiten“ ein Verzeichnis von angebotenen Diensten pflegt (mit freiwilligen Einträgen der anbietenden Agenten). Darüber hinaus ist das AMS für die Lebenszyklusverwaltung der Agenten auf einer Plattform zuständig, es überwacht alle Erzeugungen, Terminierungen, Pausierungen, Umzüge u.ä. aller Agenten auf der Plattform. Nutzbar werden die Dienste von AMS und DF aber erst durch die Definition der Inhaltssprache SL0 (siehe Abschnitt 4.2.3) mit einer „Agent Management Ontology", welche die von den beiden Diensten verstandenen Begriffe und Funktionen festschreibt.

In den folgenden Abschnitten werden die einzelnen hier angerissenen FIPA-Standards genauer vorgestellt. Dabei werden zunächst in einem Abschnitt die FIPA-Spezifikationen zur Syntax und Semantik von Nachrichten beschrieben, worauf in zwei weiteren Abschnitten jeweils Transport- und Verwaltungselemente aus Sicht der konkreten FIPA2000-Spezifikation und aus Sicht der abstrakten Architektur vorgestellt werden.

4.2 Die Agentenkommunikationssprache

FIPA-Agenten kommunizieren, indem sie sich Nachrichten schicken. Wie im vorigen Abschnitt bereits angerissen, kann eine solche Nachrichtenkommunikation nur funktionieren, wenn alle Agenten *dieselbe Sprache sprechen*. Als gemeinsame Sprache sieht die FIPA die „Agent Communication Language" (ACL) vor, die von mehreren Spezifikationen in verschiedenen Aspekten definiert wird. Die Vorstellung dieser Spezifikationen beginnt hier mit den eher semantisch-pragmatisch orientierten Dokumenten und geht danach zu immer technischeren Standards über.

4.2.1 Kommunikative Akte

Die „Agent Communication Language" (ACL) schreibt vor, dass mit einer Nachricht ein kommunikativer Akt ausgedrückt wird. Der Begriff des kommunikativen Akts ist angelehnt an die Sprechakt-Theorie von Searle (siehe [Sea1969]). Searle hat beobachtet, dass man mit einer Äußerung wie z.B. „In Hamburg regnet es gerade." auf mehreren Ebenen kommuniziert:

1. Man reiht Wörter, Laute oder Zeichen aneinander. Dies nennt Searle den *Äußerungsakt*.

2. Man referenziert Objekte (in diesem Fall die Stadt Hamburg) und schreibt ihnen etwas zu (dass es gerade regnet). Das ist der *propositionale Akt*.

3. Man vollzieht einen *illokutionären Akt*, weil man mit dem gesprochen Satz im jeweiligen Kontext eine bestimmte Intention verbindet, z.B. Informieren, Behaupten, Fragen, Auffordern oder Versprechen. Für das Hamburger-Regen-Beispiel können schon verschiedene illokutionäre Akte passen, abhängig vom Kontext. Beispiele sind: Informieren, Behaupten (wenn der Sprecher es gar nicht sicher weiß), Ablehnen (wenn der Sprecher vorher zum Spaziergang aufgefordert wurde) oder Warnen (wenn das Gegenüber dort Urlaub machen will). Auch eine Frage ist durch Umstellung des Satzes formulierbar („Regnet es gerade in Hamburg?"), wobei der propositionale Akt unverändert bleibt.

4. Der *perlokutionäre Akt* bezeichnet die Wirkung beim Gesprächspartner, dass man also beispielsweise durch eine Aufforderung den anderen dazu bringt, etwas zu tun, ihn durch eine Warnung alarmiert usw.

Searle stellt fest, dass der illokutionäre Akt nicht ohne Äußerungs- und propositionalen Akt vollzogen werden kann und bezeichnet den illokutionären Akt als „vollständigen Sprechakt". Dieser „vollständige Sprechakt" dürfte das Vorbild für den kommunikativen Akt aus den FIPA-Spezifikationen sein[1].

Eine ACL-Nachricht ist, da sie die programmtechnische Entsprechung eines kommunikativen Aktes darstellt, mehr als nur ein aus aneinandergereihten Symbolen und Zeichen bestehender Inhalt, der, wenn man ihn interpretiert, Objekte referenzieren und Aussagen über diese Objekte machen kann. Darüber hinaus kann jede Nachricht immer einer Kategorie wie Behaupten, Fragen, Versprechen, Auffordern, Warnen o.ä. zugeordnet werden. Diese Zuordnung soll in der ACL aber nicht, wie in der menschlichen Kommunikation, aus dem Kontext heraus geschlossen werden, sondern explizit in der Nachricht ausgewiesen sein: Wenn ein Agent eine ACL-Nachricht verschickt, muss neben dem propositionalen Inhalt eine Kennzeichnung durch ein englischsprachiges Verb vorhanden sein, welches den illokutionären Akt der Nachricht ausdrückt. In dieser Arbeit wird das Verb, welches den Typ des kommunikativen Aktes bezeichnet, Performativ genannt. Diese Begriffswahl passt zur Benennung des entsprechenden Elements der ACL-Struktur (siehe Abschnitt 4.2.5), auch wenn der Begriff Performativ in den FIPA-Spezifikationen sonst nicht verwendet wird.

In der „Communicative Act Library" [FIPA00037] werden die von der FIPA definierten bzw. akzeptierten Performative gesammelt. Ein Eintrag der Sammlung umfasst neben dem Verb und der Kurzbeschreibung der damit gekennzeichneten Kategorie von kommunikativen Akten sowohl eine natürlichsprachliche als auch eine formale Definition der Semantik der kommunikativen Akte dieses Typs mit jeweiligen Vorbedingungen und Konsequenzen. Da ein kommunikativer Akt nicht ohne propositionalen Inhalt sinnvoll ausgeführt werden kann, wird zu jedem Performativ auch die Grobstruktur des Inhalts vorgegeben.

Als Beispiel für einen Eintrag der Communicative Act Library soll hier die Definition des häufig gebrauchten `inform`-Performativs dienen (siehe [FIPA00037, S. 11]). Ein kommunikativer Akt dieser Art wird verwendet, wenn der sendende Agent dem Empfänger mitteilen möchte, dass eine bestimmte Aussage wahr ist. Der Inhalt des Akts ist somit eine Aussage. Weitergehend wird der Akt so definiert, dass der Sender

- glaubt, dass die Aussage wahr ist,
- möchte, dass der Empfänger dies ebenfalls glaubt und
- nicht glaubt, dass der Empfänger bereits etwas über die Wahrheit der Aussage weiß.

Der Empfänger der Nachricht hingegen

- kann annehmen, dass der Sender an die Wahrheit der Aussage glaubt,

[1] Die FIPA-Spezifikationen nehmen leider keine konkrete Zuordnung zwischen kommunikativem Akt und den Sprechakt-Ebenen vor.

- kann davon ausgehen, dass der Sender wünscht, dass der Empfänger auch an die Aussage glaubt und
- ist aber dennoch frei in seiner Entscheidung, ob er die Aussage als wahr in sein Wissen aufnimmt.

Formal sieht die Beschreibung des `inform`-Performativs in der Semantic Language[2] dann so aus:

$$
\begin{aligned}
&< i, \texttt{inform}(j, \phi) > \\
&\quad \mathrm{FP} : \mathrm{B}_i\phi \wedge \neg \mathrm{B}_i(\mathrm{Bif}_j\phi \vee \mathrm{Uif}_j\phi) \\
&\quad \mathrm{RE} : \mathrm{B}_j\phi
\end{aligned}
$$

Die erste Zeile beschreibt den von Agent i vollzogenen Akt, den Agenten j über die Wahrheit der Aussage ϕ zu informieren. Die Vorbedingungen („feasibility preconditions“, FP) werden in der zweiten Zeile definiert: $\mathrm{B}_i\phi$ bedeutet, dass Agent i um ϕ weiß (B steht für „belief“, meint also sicheres Glauben an ein Fakt – im Gegensatz zu U (für „uncertain“), welches unsicheres Wissen bezeichnet). Darüber hinaus darf er aber nichts darüber wissen ($\neg\mathrm{B}_i(\ldots)$), dass Agent j die Aussage als wahr oder falsch einstuft ($\mathrm{Bif}_j\phi$ als Abkürzung für $\mathrm{B}_j\phi \vee \mathrm{B}_j\neg\phi$) oder auch nur unsichere Informationen über den Wahrheitsgehalt von ϕ hat ($\mathrm{Uif}_j\phi$). Die beabsichtigte Wirkung („rationale effect“, RE) des Akts wird in der dritten Zeile beschrieben. Im Falle des `inform`-Performativs will Agent i erreichen, dass Agent j die Aussage ϕ als wahr aktzeptiert ($\mathrm{B}_j\phi$).

Mit dieser formalen Definition der Typen von kommunikativen Akten möchte die FIPA erreichen, dass klare, unzweideutige, standardisierte Aussagen über Semantik und Pragmatik der kommunikativen Akte existieren. Alle Agenten, die FIPA-konform sein sollen, müssen, wenn sie ein Performativ aus der Communicative Act Library verwenden, dieses entsprechend den definierten Richtlinien interpretieren. Nur so kann sichergestellt werden, dass – zumindest was den Typ des kommunikativen Akts angeht – keine „Missverständnisse“ zwischen FIPA-konformen Agenten auftreten.

Ein Agent ist nicht verpflichtet, sich auf diese definierten Performative zu beschränken, denn die FIPA erlaubt die Verwendung neuer, selbstdefinierter Performative von Agentenentwicklern. Der Agent kann sogar auf die Verwendung der bekannten Performative völlig verzichten und ausschließlich auf Eigenkreationen zurückgreifen. Aber solche proprietären kommunikativen Akte werden naturgemäß nicht von allen Agenten verstanden und sollten daher nur dann eingesetzt werden, wenn kein passendes Performativ in der Aktsammlung zu finden ist.

Ein wichtiger Schritt zum Erreichen der Missverständnisarmut bei der Nachrichtenkommunikation ist also die Definition ausreichend vieler Typen von kommunikativen Akten, um möglichst alle Situationen abzudecken. So werden proprietäre Eigenentwicklungen vermieden. Daher umfasst die Communicative Act Library bereits eine

[2] Die Semantic Language wird im Anhang der „FIPA Communicative Act Library Specification“ [FIPA00037] definiert, hier werden nur einige Grundzüge am konkreten Beispiel aufgezeigt.

recht vielfältige Liste von gut zwanzig Performativen. Zusätzlich kann jeder Entwickler bei der FIPA die Aufnahme von selbstdefinierten Performativen in die Sammlung beantragen, wenn er eine ausführliche Begründung und Definition mitliefert.

Neben dem bereits ausführlich beschriebenen `inform`-Performativ werden unter anderem folgende Typen von kommunikativen Akten definiert:

`confirm` **und** `disconfirm`: Diese beiden Performative ähneln sehr dem `inform`. Der wesentliche Unterschied ist, dass der sendende Agent weiß, dass der Empfänger bereits die Wahrheit der bestätigten bzw. dementierten Aussage vermutet. Dieser Fall war bei der Beschreibung des `inform`-Performativs explizit ausgeschlossen worden. Die feine Granulierung des Trios `inform`, `confirm` und `disconfirm` erlaubt es dem Absender der Nachricht, neben der eigentlichen Aussage auch etwas über den aktuellen Kontext mitzuteilen. Die meisten anderen Typen von kommunikativen Akten (außer `request`) sind eigentlich nur Sonderfälle dieser drei Informationsperformative.

`request`: Dieses Performativ steht dafür, dass der Absender den Empfänger zu einer Aktion auffordern will. Dabei kann die geforderte Aktion beliebig komplex sein, solange sie den Fähigkeiten des Empfängers entspricht (z.B. eine Reise buchen).

`agree`: Damit kann ein Agent bestätigen, dass er eine Aktion durchführen wird. Der Ausführungszeitpunkt kann in der Zukunft liegen und auch noch von anderen Bedingungen abhängen.

`refuse`: Der sendende Agent verweigert die Durchführung einer Aktion. Der Inhalt dieses Performativs ist ein Paar aus der verweigerten Aktion selbst und einer Begründung, warum die Aktion nicht durchführbar ist.

`failure`: Ein solcher kommunikativer Akt informiert den Empfänger darüber, dass der Absender zwar versucht hat, eine prinzipiell machbare Aktion durchzuführen, dies aber aus irgendeinem Grund gescheitert ist. Der Grund wird (ähnlich wie beim `refuse`-Performativ) mit übertragen.

`cancel`: Der Absender dieser Nachricht informiert den Empfänger darüber, dass ersterer nicht mehr von letzterem erwartet, eine bestimmte Aktion durchzuführen.

`not-understood`: Ein Akt dieser Art wird dann von einem Agenten (z.B. i) an einen anderen (z.B. j) verschickt, wenn i eine von j durchgeführte Aktion wahrgenommen, aber nicht verstanden hat. Der häufigste Sonderfall hiervon dürfte die Situation sein, dass i gerade eine unverständliche Nachricht von j empfangen hat.

Der kommunikative Akt `not-understood` nimmt eine gewisse Sonderstellung ein, da die ACL alle Agenten verpflichtet, auf nicht interpretierbare Nachrichten mit diesem Performativ zu reagieren (siehe [FIPA00061, S. 2]). Auf diese Weise kann auch ein gravierendes Verständnisproblem in kanalisierte Bahnen gelenkt werden.

Weitere Performative, die hier nicht detailliert aufgeführt werden sollen, dienen dem Anfordern, Akzeptieren und Ablehnen von Vorschlägen, dem Vereinbaren von Durchführungszeitpunkten für Aktionen (inklusive wiederholter Ausführung), dem Weiterleiten bzw. Streuen von Nachrichten und der gezielten Informationsabfrage.

4.2.2 Interaktionsprotokolle

Betrachtet man die Kommunikation zwischen Agenten, so fällt auf, dass im Laufe von Konversationen immer wieder typische Muster in der Abfolge der kommunikativen Akte auftreten. Statt diese Abfolgen zufällig bzw. kausal aus den (Re-)Aktionen der Agenten entstehen zu lassen, ist es wünschenswert, ein solches Muster als Interaktionsprotokoll festzuschreiben und zu benennen. Dadurch wird die Entwicklung der Agenten vereinfacht, weil wiederverwendbare Elemente entstehen. Die eindeutige Benennung und Definition eines Interaktionsprotokolls lässt kommunizierende Agenten im Voraus wissen, welche Nachricht als nächstes erwartet werden kann bzw. gesendet werden sollte. Fehlersituationen lassen sich einfacher erkennen und behandeln, weil sie schlichtweg nicht dem Ablauf des verwendeten Interaktionsprotokolls entsprechen oder aber im Protokoll bereits an festen Stellen als Möglichkeit vorgesehen sind.

In der „Interaction Protocol Library“ [FIPA00025] sammelt die FIPA Beschreibungen diverser Interaktionsprotokolle, um das Zusammenspiel von Agenten zu erleichtern. Anders als die Communicative Act Library (siehe voriger Abschnitt) ist die Interaktionsprotokollsammlung nicht monolithisch in einer FIPA-Spezifikation enthalten. Stattdessen wird jedes Interaktionsprotokoll in einer eigenen FIPA-Spezifikation definiert. Hierzu passt der im Vergleich zur Aktsammlung niedrigere Anspruch, keineswegs alle möglichen Eventualitäten und Interaktionen abzudecken, sondern nur grundlegende Interaktionsmuster aufzuzeigen, die für den jeweiligen Anwendungsfall noch feiner ausgearbeitet werden müssen. Daraus ergibt sich allerdings auch, dass die Einhaltung der definierten Interaktionsprotokolle die Interoperabilität der an einer Konversation beteiligten Agenten nicht garantiert, sondern lediglich das Erreichen dieses Ziels erleichtert.

Als Beispiel möge hier das „FIPA Request Interaction Protocol“ dienen, welches in der Spezifikation [FIPA00026] definiert wird. Das Protokoll kann immer dann zur Anwendung kommen, wenn ein Agent einen anderen auffordern möchte, eine bestimmte Aktion durchzuführen. Der aufgeforderte Agent hat entweder die Möglichkeit, die Aktion durchzuführen oder er kann auf eine von mehren Arten antworten, dass er die Aktion nicht durchführen kann bzw. will.

Die Beschreibung des Protokolls wird als AUML-Protokolldiagramm gegeben. Die Abkürzung AUML steht für „Agent UML“ und ist eine in Entwicklung befindliche Erweiterung der bereits in Abschnitt 2.2 vorgestellten „Unified Modeling Language“. Eine Einführung in die Ziele und Pläne von AUML gibt [OPB2000]. Die für FIPA-Interaktionsprotokolle verwendeten AUML-Protokolldiagramme sind eine Erweiterung der UML-Sequenzdiagramme und werden (nebst zusätzlichen abkürzenden Notationen) in der „Interaction Protocol Library Specification“ [FIPA00025] eingeführt und erläutert. Hier soll nur kurz am praktischen Beispiel die Notation vorgeführt werden.

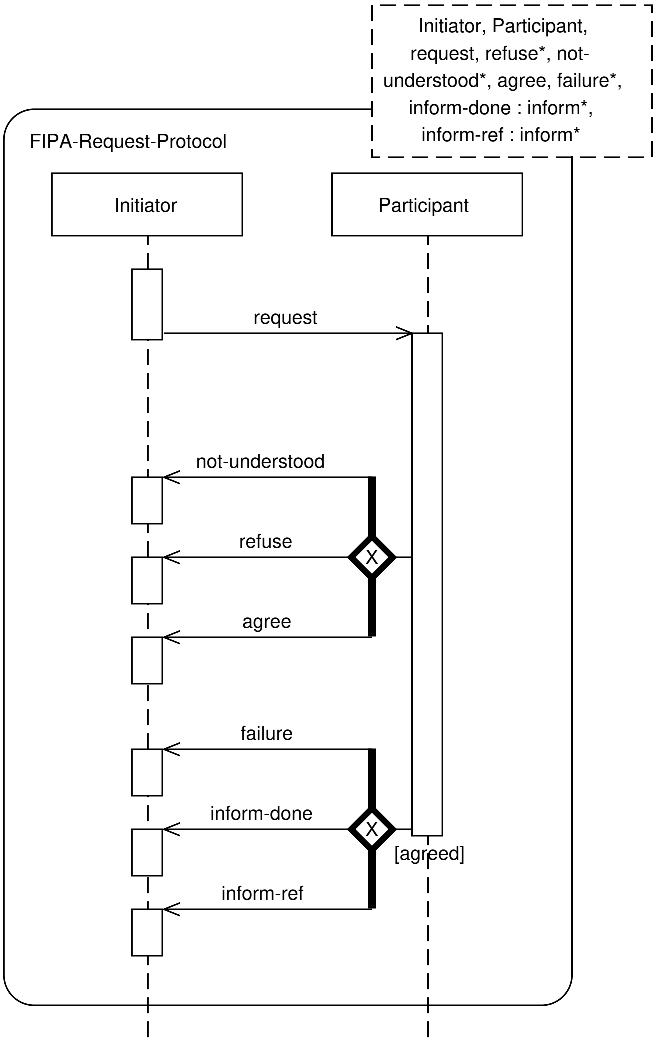


Abbildung 4.2: Das FIPA-Request-Interaktionsprotokoll (aus [FIPA00026])

Eine Kommunikation unter dem FIPA-Request-Protokoll läuft wie folgt ab (siehe Abbildung 4.2): Der Initiator der Konversation fordert den anderen Teilnehmer mittels `request`-Akt auf, eine Aktion durchzuführen. Der aufgeforderte Agent hat nun drei Möglichkeiten, darauf zu antworten: Entweder hat er die Nachricht nicht verstanden, dann sendet er einen kommunikativen Akt vom Typ `not-understood` zurück. Oder er hält die Aktion für nicht durchführbar, dann ist die Antwort ein `refuse`. Oder er ist bereit, die Aktion durchzuführen und antwortet mit `agree`.

Nur, wenn die Bedingung `[agreed]` zutrifft, verschickt der aufgeforderte Agent eine weitere Nachricht über den (Miss-)Erfolg der durchgeführten Aktion: `failure` weist den Initiator darauf hin, dass die Aktion doch nicht durchgeführt werden konnte. Die Kurznotation `inform-done` steht für ein `inform`-Performativ, welches die Bestätigung der Aktionsausführung zum Inhalt hat. Falls die Aktion daraus bestand, eine bestimmte Information nachzuschlagen oder ein Ergebnis zurückzuliefern, kann das `inform`-Performativ auch einen referentiellen Ausdruck enthalten; dafür steht die Kurznotation `inform-ref`.

Diese Protokollbeschreibung deckt offensichtlich keine Ausnahmesituationen ab. Ein vorstellbarer Fall wäre z.B. eine `cancel`-Nachricht vom Initiator, um die Aktion noch vor Ausführung abzubrechen. Solche Ausnahmesituationen sind explizit nicht im Interaktionsprotokoll modelliert, weil die FIPA darin nur ein einfaches Interaktionsmuster sieht. Die Berücksichtigung der Ausnahmesituationen hätte den Blick auf die wesentliche Struktur des Protokolls verstellt. Eine feinere Ausarbeitung dieses Musters ist daher für den praktischen Einsatz des Protokolls notwendig.

Bisher hat die FIPA noch eine Handvoll weiterer, häufig auch komplexerer Interaktionsprotokolle spezifiziert, die sich z.B. mit Ausschreibungen, Auktionen oder Zwischenhandel beschäftigen. Die Sammlung ist auf Zuwachs ausgelegt, d.h. Entwickler dürfen die Aufnahme weiterer Interaktionsprotokolle beantragen, wenn der Antrag durch ausreichende Dokumentation und eine Begründung für die Nützlichkeit des Protokolls begleitet wird. Die „Interaction Protocol Library“ hat aber nicht den Anspruch, vollständig zu sein, also alle erdenklichen Interaktionen abzudecken.

4.2.3 Inhaltssprachen

In Abschnitt 4.2.1 dieser Arbeit wird bereits erwähnt, dass ein kommunikativer Akt als Nachricht erst durch den sogenannten “Inhalt“, also den themenspezifischen, propositionalen Teil des kommunikativen Akts, vollständig wird. Dieser Inhalt kann je nach Performativ eine Aussage, eine Aktion, eine Referenz, eine Frage, ein weiterer kommunikativer Akt oder irgendein anderer Konversationsgegenstand sein. Auch die Kombination mehrerer solcher Objekte als Inhalt einer Nachricht ist möglich, so enthält z.B. das `failure`-Performativ eine Aktion und eine Begründung für deren Fehlschlag.

Der Inhalt muss so in die Nachricht eingebracht werden, dass der Empfänger ihn im Sinne des Absenders interpretieren kann. Kurz: Beide Kommunikationspartner müssen – wieder einmal – dieselbe Sprache sprechen. Es muss also eine Sprache mit möglichst exakt definierter Syntax, Semantik und Pragmatik zur Verfügung stehen. Das Problem der meisten exakter definierten Sprachen ist aber, dass sie sich für bestimmte Zwecke besser als für andere eignen und im Extremfall vielleicht für bestimmte Zwecke gar nicht geeignet sind.

Um den Agentenentwicklern sowohl konkret definierte Sprachen zur Verfügung zu stellen, als auch die Agenten nicht zu sehr in ihren Kommunikationsfähigkeiten zu beschränken, hat die FIPA die „Content Language Library“ [FIPA00007] spezifiziert. Darin sind bisher vier alternative Inhaltssprachen vertreten, namentlich SL („Semantic Language“, [FIPA00008]), CCL („Constraint Choice Language“, [FIPA00009]), KIF („Knowledge Interchange Format“, [FIPA00010]) und RDF („Resource Description Framework“, [FIPA00011]). Die Aufnahme zusätzlicher Sprachen ist auf Antrag möglich, wenn die Sprache ausreichend konkret spezifiziert ist und sich gemäß den Anforderungen der übergeordneten Spezifikationen der „Agent Communication Language“ (ACL) und der kommunikativen Akte integrieren lässt.

Auch innerhalb einer Sprachdefinition kann bereits das Problem auftauchen, einen Kompromiss zwischen Mächtigkeit und Handhabbarkeit der Sprache finden zu müssen. Die FIPA empfiehlt daher, neben der voll ausdrucksfähigen Grammatik einer Sprache auch reduzierte Teilgrammatiken anzugeben, welche von kleineren und effizienteren Parsern verarbeitet werden können. Für einige der von der FIPA definierten Sprachen sind in den jeweiligen Spezifikationen solche Unterscheidungen gemacht worden.

Weder in der abstrakten Architektur noch in den konkreten FIPA2000-Spezifikationen wird eine der Sprachen zwingend zur Verwendung vorgeschrieben, allerdings wird eine reduzierte Version von SL („Semantic Language") für die Kommunikation mit den Verwaltungsagenten der FIPA2000-Plattform festgelegt. Daher soll hier kurz die reduzierte Version SL0 vorgestellt werden. SL ist der zur Beschreibung der kommunikativen Akte verwendeten Grammatik (siehe Abschnitt 4.2.1 und [FIPA00037]) entnommen. Es handelt sich bei SL0 um die kleinste sinnvolle von der FIPA definierte Teilmenge der SL-Ausdrucksfähigkeit. SL0 kann Aktionen beschreiben, deren Vollendung ausdrücken sowie einfache binäre Aussagen und ermittelte Berechnungsergebnisse darstellen. Gegenüber den umfangreicheren SL-Varianten fehlt die Fähigkeit, aussagenlogische Ausdrücke mit boolschen Operatoren oder gar Ausdrücke höherer Logiken darzustellen.

Ein SL0-Inhaltsausdruck kann entweder ein Aktionsausdruck der Form „`(action` *`Agent Aktion`*`)`" oder eine Aussage sein. Als Aussage ist entweder eine Aktionsdurchführungsbestätigung „`(done (action` *`Agent Aktion`*`))`", ein Ergebnis einer Berechnung „`(result` *`Berechnung Ergebnis`*`)`", ein mindestens einstelliges Prädikat „`(`*`Prädikatname Parameter...`*`)`" oder ein einfaches Aussagensymbol (z.B. „`true`" oder „`false`") zulässig. Die als Agent, Aktion, Berechnung, Ergebnis oder Parameter anzugebenden Terme dürfen wiederum Zeichenketten, Zahlen, Zeitangaben, Aktionsausdrücke oder Funktionen „`(`*`Funktionsname Parameter...`*`)`" sein, wobei die Funktionsnamen `set` und `sequence` für Mengen und Listen bereits definiert sind. Um die Verwendung von benannten Funktionsparametern (anstelle der sonst nur durch ihre Reihenfolge identifizierbaren Parameter) zu ermöglichen, gibt es die Notation „`(`*`Funktionsname :Parametername Parameter`*` :...)`".

Der häufigste Anwendungsfall von SL0 (z.B. bei der Kommunikation mit den Verwaltungsagenten der FIPA-Plattform) ist das Auffordern zu bestimmten Aktionen, inklusive der darauf folgenden Erfolgs- bzw. Fehlschlagsmeldungen. Ein anschauliches Beispiel ist der folgende Versuch, dem vermeintlich schlechten Hamburger Wetter zu entkommen, indem man bei einem Reisebüro-Agenten die Buchung einer Mallorca-Reise in Auftrag gibt (in einem `request`-Performativ):

```
(action Reisebüro (buche Mallorca))
```

Darauf sollte die Antwort des Reisebüros bei erfolgreicher Durchführung der Buchung folgendermaßen lauten (in einen kommunikativen Akt des Typs `inform` eingebettet):

```
(done (action Reisebüro (buche Mallorca)))
```

Hätte die Buchung nicht geklappt, so würde die Antwort im `failure`- oder `refuse`-Performativ aus zwei Ausdrücken bestehen:

```
(action Reisebüro (buche Mallorca))
(fehler "Alles belegt!")
```

Die syntaktisch-semantische Form der konkreten Problembeschreibung im `fehler`-Prädikat ist hier eine einfache Zeichenkettenkonstante, andere Darstellungsformen sind aber durchaus denkbar.

4.2.4 Ontologien

Kommunizierenden Agenten nützt die Standardisierung von mehrfach verwendbaren Performativen, Interaktionsprotokollen und Inhaltssprachen nur wenig, wenn keine Einigkeit über das konkrete Diskussionsthema besteht. Um beim Beispiel vom Ende des vorigen Abschnitts zu bleiben: Wenn der Empfänger des Auftrags `(action Reisebüro (buche Mallorca))` den Begriff „`buche`“ als Baumart interpretiert, wird die Kommunikation für alle Beteiligten höchst unbefriedigend verlaufen.

Die FIPA erlaubt für die Festlegung der in der Kommunikation verwendeten Begriffe oder Symbole die Benennung der in einer Nachricht gültigen Ontologie. Die Ontologie definiert die Begriffe, die zur Beschreibung und Repräsentierung eines Wissensgebiets verwendet werden können. Wenn also alle an der Kommunikation beteiligten Agenten die in den Nachrichten ausgewiesene Ontologie kennen und nur Begriffe im Sinne der Ontologiedefinition verwenden, sollten keine Missverständnisse auf der inhaltlichen Bedeutungsebene mehr auftreten. Zusammen mit den in Abschnitt 4.2.1 bereits vorgestellten Performativen (welche ja auch Bestandteil einer Ontologie sind, nämlich der implizit definierten Ontologie der kommunikativen Akte) und einer der in Abschnitt 4.2.3 vorgestellten Inhaltssprachen ist so die eindeutige Interpretation einer Nachricht möglich.

Die Begriffsdefinition in einer Ontologie kann auf verschiedene Weisen geschehen. Die einfachste, aber auch unflexibelste Variante ist die *implizite* Kodierung der Ontologie im Agenten selber, d.h. der Agentenentwickler programmiert den Agenten so, dass die Begriffe im Sinne der Ontologie verwendet werden. Dabei muss also nicht die Software, sondern der Entwickler die Ontologie kennen und richtig verwenden.

Ein deutlich aufwändigeres Vorgehen schreibt die Ontologie *explizit* in einer Logik erster Ordnung auf, wobei die Begriffe als ein- oder zweistellige Prädikate, Konzepte und Relationen genannt, definiert werden. Die reine Vokabular- und Beziehungsdefinition kann durch zusätzliche Axiome ergänzt werden, welche die Interpretation und Verwendung der Begriffe konkretisieren. Somit liegen sowohl Syntax als auch Semantik der Ontologie in maschinell interpretierbarer Form vor. Dadurch ist eine größere Flexibilität beim Einsatz der Ontologien gegeben, da z.B. zur Entwicklungszeit des Agenten noch unbekannte Ontologien verwendet werden können. Ein Agent kann dann auch *über* die Ontologie reden oder mit der Ontologie arbeiten (z.B. Korrekturen vornehmen), indem er eine Meta-Ontologie verwendet.

Da Ontologien in der Regel domänenspezifisch sind, gibt es keine vorgefertigten, allgemeingültigen FIPA-Ontologie-Spezifikationen. Stattdessen wird, wenn dies zur Spezifikation von FIPA-Diensten notwendig ist, eine passende konkrete Ontologie definiert. Diese Ontologien kommen in der Regel ohne eine im eben erwähnten Sinne *explizite* Definition daher. Eine solche Ontologie ist z.B. die `FIPA-Agent-Management`-Ontologie, die bei der Kommunikation mit den Verwaltungsagenten der FIPA2000-Plattform eingesetzt wird (siehe Abschnitt 4.4.2). Diese Ontologie definiert unter anderem die Struktur von Verzeichniseinträgen (nebst Benennung und Bedeutung der einzelnen Elemente) sowie passend dazu die Namen und Aufgaben der vom Verzeichnisdienst angebotenen Funktionen.

Viele der für die „Agent Communication Language“ getroffenen Regelungen können ebenfalls als Ontologie aufgefasst werden. Das beginnt bereits bei der in [FIPA00061] definierten `FIPA-ACL`-Ontologie, welche die Namen der in Abschnitt 4.2.5 vorgestellten Nachrichtenelemente und deren Bedeutung festlegt. Diese Sichtweise lässt sich auch auf die Ontologie der kommunikativen Akte (welche die Performativbezeichnungen und ihre Bedeutung beschreibt, siehe Abschnitt 4.2.1) oder die Ontologien der Inhaltssprachen (siehe Abschnitt 4.2.3), wodurch die Bedeutung der syntaktischen Elemente und reservierten Wörter festgelegt wird, anwenden.

Eine allgemeinere Herangehensweise wird in der „FIPA Ontology Service Specification“ [FIPA00086] definiert. Ein Ontologieagent erlaubt die Kommunikation *über* Ontologien, sofern diese Ontologien explizit (s.o.) definiert sind. Zu den Aufgaben, die der Ontologieagent übernehmen kann, gehört

- das Finden öffentlicher Ontologien, um sie zu verwenden,
- die Wartung öffentlicher Ontologien,
- das Übersetzen von Ausdrücken zwischen verschiedenen Ontologien oder Inhaltssprachen,
- die Auskunft über Beziehungen zwischen Begriffen oder Ontologien, und
- die Suche nach einer gemeinsamen Ontologie für eine Agentenkommunikation.

Diese teilweise recht umfangreichen Aufgaben werden in der Ontologiedienst-Spezifikation lediglich hinsichtlich ihrer Schnittstelle und der dafür notwendigen Meta-Ontologie definiert. Aufgrund des notwendigen implementatorischen Aufwands sind die Existenz und das Dienstangebot des Ontologieagenten auf einer Plattform optional.

4.2.5 Struktur

Die Syntax von Nachrichten wird im Rahmen der FIPA-Standards auf zwei Ebenen definiert: Zum einen gibt die „ACL Message Structure Specification“ [FIPA00061] die Gliederung einer Nachricht in einzelne Elemente vor, zum anderen legen diverse Spezifikationen alternative Kodierungen der Nachricht und ihrer Elemente fest. Die Kodierung ist Gegenstand des folgenden Abschnitts 4.2.6, hier soll es zunächst um den grundlegenden Aufbau einer Nachricht gehen.

Eine Nachricht setzt sich aus einem oder mehreren Elementen zusammen, welche die einzelnen Informationen der Nachricht aufnehmen. Jedes Element hat eine eindeutige Bezeichnung und nimmt einen Wert auf, für den gegebenenfalls eine Liste von reservierten Werten existiert. Die Elemente können grob kategorisiert werden: Neben dem

Nachrichtentyp (als kommunikativer Akt) und dem Nachrichteninhalt gibt es weitere Elemente, welche den Nachrichteninhalt klassifizieren, Kommunikationsteilnehmer identifizieren oder der Steuerung des Konversationsflusses dienen. Wie bei den meisten anderen ACL-Definitionen gilt auch hier die Regel, dass eigene Elemente definiert werden können, aber die von der FIPA benannten nicht zweckentfremdet werden dürfen.

Das wichtigste Element, welches zwingend in der Nachricht vorhanden sein muss, heißt `performative`. Es nimmt das Performativ (siehe Abschnitt 4.2.1) auf, welches den Typ des kommunikativen Akts benennt, der durch die Nachricht vollzogen wird. Dieses Element legt also fest, ob die Nachricht als Frage, Aufforderung, Information oder anders zu interpretieren ist.

Das `content`-Element nimmt den zum kommunikativen Akt gehörenden Inhalt auf. Der Inhalt kann in einer von mehreren möglichen Sprachen (siehe Abschnitt 4.2.3), einer von mehreren möglichen Kodierungen und unter Verwendung einer oder mehrerer Ontologien (siehe Abschnitt 4.2.4) verfasst werden. Damit eindeutig ist, welche Sprache, Kodierung und Ontologien verwendet werden, kann dies in den Elementen `language`, `encoding` und `ontology` mitgeteilt werden. Das Verwenden dieser inhaltsbeschreibenden Felder ist nicht verpflichtend – wenn aus dem Kontext der Unterhaltung klar ist, welche Darstellung für den Inhalt gewählt werden sollte, kann man diese Felder auch ungenutzt lassen.

Die Elemente `sender` und `receiver` benennen die Teilnehmer an der mittels einer Nachricht durchgeführten Kommunikation. Die abstrakte Architektur weist in diesem Zusammenhang darauf hin, dass nicht die technische Adress-Information für den Nachrichtentransport der wesentliche Zweck dieser Elemente ist, sondern die Identifikation der Beteiligten anhand ihrer eindeutigen Agentennamen. Eine Nachricht kann nur einen Absender haben, wohl aber mehrere Empfänger – „Rundschreiben“ sind also möglich. Mittels des `reply-to`-Feldes kann vorgegeben werden, an welche Agenten eine Antwort auf diese Nachricht geschickt werden soll.

Die letzte Gruppe der von der FIPA vorgeschlagenen Nachrichtenelemente dient der Steuerung des Konversationsablaufs zwischen den kommunizierenden Agenten. Alle diese Elemente sind optional. Zur Konversationskontrolle gehört natürlich die Angabe des übergeordneten Interaktionsprotokolls (siehe Abschnitt 4.2.2) im Element `protocol` – sofern ein solches Protokoll verfolgt wird. Zwei zusammenarbeitende Elemente erlauben es einem Agenten, den Gesprächspartner um eine Kennzeichnung der Antwort-Nachricht zu bitten (`reply-with`) und eine solche erbetene Kennzeichnung einzubetten (`in-reply-to`). Wie sich die Kennzeichnung zusammensetzt, ist Sache des bittenden Agenten. Durch das Element `reply-by` kann dem Empfänger einer Nachricht eine Frist mitgeteilt werden, bis wann er spätestens geantwortet haben sollte.

4.2.6 Repräsentierung

Die Syntax von Nachrichten wird im Rahmen der FIPA-Standards auf zwei Ebenen definiert: Zum einen gibt die im vorigen Abschnitt 4.2.5 beschriebene „ACL Message Structure Specification“ [FIPA00061] die Gliederung einer Nachricht in einzelne

Elemente vor, zum anderen legen diverse Spezifikationen alternative Kodierungen der Nachricht und ihrer Elemente fest.

Bislang gibt es drei von der FIPA vorgegebene Repräsentierungen für ACL-Nachrichten: als Zeichenkette mit geklammerten Ausdrücken („ACL Message Representation in String“, [FIPA00070]), als XML-Dokument („ACL Message Representation in XML“, [FIPA00071]) und als platzsparende Bytefolge („ACL Message Representation in Bit-efficient Encoding“, [FIPA00069]). Weitere Repräsentierungen, z.B. als Parsebaum für die interne Interpretation der Nachricht im Agenten, sind natürlich möglich. Für die bis dato definierten Nachrichtentransport-Protokolle (siehe Abschnitt 4.4.1) werden aber nur diese drei Kodierungen verwendet.

Die Zeichenketten-Darstellung stellt – aufgrund ihrer guten Lesbarkeit für Menschen – die übliche Form für anschauliche Beispiele dar, daher soll sie hier grob umrissen werden. Die Notation ist an die SL-Syntax für Funktionen mit benannten Parametern angelehnt (siehe Abschnitt 4.2.3), wobei das Performativ als Funktionsname gebraucht wird. Zeilenumbrüche und Leerzeichen werden bei der Interpretation der Nachricht ignoriert. (In dieses Beispiel sind die in allen vorangegangenen Abschnitten vorgestellten Spezifikationen mit eingeflossen.)

```
(request
  :sender     (agent-identifier
                :name producer)
  :receiver   (set
                (agent-identifier
                  :name consumer))
  :content    "((action
                 (agent-identifier
                   :name consumer)
                 (consume token-42)))"
  :language   FIPA-SL0
  :ontology   (set
                producer-consumer-ontology)
  :protocol   producer-consumer-ip
  :reply-with 42)
```

Es handelt sich bei dieser Nachricht also um den kommunikativen Akt, dass ein Agent namens „producer“ einen Agenten namens „consumer“ auffordert, die Aktion, ein `token-42` zu konsumieren, durchzuführen. Der Inhalt der Nachricht ist in der Inhaltssprache SL0 verfasst und verwendet die `producer-consumer-ontology` (welche vermutlich die Aktion `consume` definiert). Der Akt findet im Kontext des `producer-consumer`-Interaktionsprotokolls statt, und der Agent „consumer“ soll mit einem Akt antworten, der durch den Wert `42` gekennzeichnet ist.

Im Beispiel gut zu erkennen ist die Tatsache, dass der Inhalt im `content`-Element einer gemäß [FIPA00070] kodierten ACL-Nachricht immer ein String ist. Das bedeutet, dass die Interpretation dieser Zeichenfolge nicht Aufgabe des Transportsystems ist, sondern in die Zuständigkeit des Agenten fällt.

4.3 Abstrakte Architektur

Die Spezifikationen der FIPA decken, wie bereits im Abschnitt 4.1 dargestellt, drei wesentliche, kommunikationsrelevante Aspekte von Agentensystemen ab: Die Kommunikationssprache, den Nachrichtentransport und die Verwaltung von Agenten und Agentenadressen. Der Agentenkommunikationssprache war der vorige Abschnitt 4.2 gewidmet, in diesem und dem folgenden Abschnitt 4.4 wird es um die beiden verbleibenden, eher technisch orientierten Themenblöcke Nachrichtentransport und Agentenverwaltung gehen.

Bei diesen beiden technischen Aspekten tritt eine Unterscheidung zu Tage, die bei der Kommunikationssprache nicht sichtbar ist: Die FIPA beschäftigt sich mit den Elementen von Multiagentensystemen auf zwei Ebenen, einer konkreteren und einer abstrakteren.

Der aus seinen Vorgängern FIPA97 und FIPA98 hervorgegangene FIPA2000-Satz von Spezifikationen beschreibt eine Agentenplattform, die den darauf befindlichen Agenten Unterstützung für die Kommunikation bietet, indem u.a. ein Grundgerüst der Kommunikationssprache, ein Nachrichtentransportprotokoll und ein Verwaltungsdienst bereitstehen. Der Entwickler der Plattform hat dabei die Wahl zwischen mehreren Transportprotokollen, Nachrichtenrepräsentierungen oder anderen Eigenschaften von Nachrichten, die Entwickler der darauf laufenden Agenten müssen allerdings mit den vom Plattformentwickler getroffenen Entscheidungen vorlieb nehmen.

Das Ziel der nachträglich entwickelten abstrakten Architektur [FIPA00001] ist es hingegen, das Konzept von Transportprotokollen, Nachrichtenrepräsentierungen, Verzeichnisdiensten usw. auf einer allgemeinen Ebene zu definieren, so dass weder Flexibilität noch Interoperabilität der Agenten auf den Plattformen eingeschränkt werden. Es soll die Integration von Agenten und Agentensystemen aus verschiedenen konkreten Architekturen oder Softwaresystemen ermöglicht werden, indem die Konzepte der verschiedenen Architekturen über die abstrakte Architektur aufeinander abgebildet werden können. Der FIPA2000-Satz von Spezifikationen soll dabei als eine mögliche Konkretisierung der abstrakten Architektur gesehen werden.

Die abstrakte Architektur erzwingt die Flexibilität in einer konkreten Architektur nicht, ein Plattformentwickler kann sich immer noch entscheiden, nur ein Transportprotokoll oder nur eine Nachrichtenrepräsentierung anzubieten. Aber die allgemeine Infrastruktur wird geschaffen, um die – falls Wahlmöglichkeiten bestehen – vom Agenten getroffene Wahl so zu dokumentieren, dass im Bedarfsfall eine Umwandlung in bzw. Anpassung an eine andere Wahlmöglichkeit vorgenommen werden kann. So bleibt die Kommunikation zwischen Agenten trotz unterschiedlicher Präferenzen prinzipiell möglich – unter der Voraussetzung, dass die entsprechenden Transformationsdienste realisiert worden sind.

Für die im vorigen Abschnitt 4.2 vorgestellten Spezifikationen zur Agentenkommunikationssprache gilt, dass sie sowohl in die FIPA2000-Suite als auch in die abstrakte Architektur eingebunden sind. Die dort vorhandene Flexibilität in einzelnen Aspekten wie den kommunikativen Akten, Interaktionsprotokollen, Inhaltssprachen oder Ontologien muss auch in der konkreten FIPA2000-Architektur gegeben sein, um die Kommunikation zwischen Agenten nicht auf ein sinnloses Maß einzuschränken.

Bei den Transport- und Verwaltungsdiensten hingegen sind Unterschiede zwischen der abstrakten und der konkreteren FIPA2000-Architektur vorhanden. Diese Unterschiede betreffen natürlich den Abstraktionsgrad der spezifizierten Elemente, gehen aber leider auch darüber hinaus. Die nachträglich über bzw. neben die FIPA2000-Spezifikationen gestellte abstrakte Architektur weicht in etlichen Definitionen schon in der Herangehensweise von der konkreten Architektur ab. Ob diese Unterschiede nun tatsächlich zu unvereinbaren Konzepten führen oder nur verschiedene Sichten auf dasselbe Konzept sind, wird noch von einer Arbeitsgruppe der FIPA untersucht (siehe [WLDG2001]). Mit einem Ergebnis dieser Untersuchung ist bald zu rechnen.

In diesem Abschnitt wird zunächst die Sichtweise der abstrakten Architektur dargestellt, gegliedert in die beiden Bereiche Nachrichtentransport und Adressverwaltung. Im darauf folgenden Abschnitt 4.4 werden dann die für beide Bereiche definierten Elemente der konkreten Architektur vorgestellt. Auf die Abweichungen zwischen den beiden Architekturen soll an dieser Stelle nicht weiter eingegangen werden, da die größten Differenzen bereits im Überblick (siehe Abschnitt 4.1) erwähnt worden sind.

4.3.1 Grundbegriffe

Die abstrakte Architektur besteht im wesentlichen aus Begriffsdefinitionen, ihr Kernbereich stellt alle Elemente der Architektur einzeln vor, definiert ihre Eigenschaften, Beziehungen und ggf. Dienstleistungen. In den folgenden beiden Abschnitten sollen hier die Nachrichtentransport- und Verwaltungselemente der abstrakten Architektur wiedergegeben werden, dafür müssen aber zunächst einige grundlegendere Konzepte aus der Architektur klar sein.

Die abstrakte Architektur definiert einen Agenten („*agent*“) als „einen Berechnungsprozess, der autonome, kommunikative Funktionalität einer Anwendung implementiert. Typischerweise kommunizieren Agenten, indem sie eine Agentenkommunikationssprache verwenden“ [FIPA00001, S. 22]. Der Fokus der Architektur liegt dabei auf der Kommunikation: Agenten kommunizieren mittels Nachrichten, die Sprechakte repräsentieren (siehe Abschnitt 4.2.1). Der Agent ist verbunden mit einigen architektonischen Elementen der Transport- und Verzeichnisdienste, diese sollen hier aber erst in den nächsten Abschnitten behandelt werden. Die technisch-konzeptuelle Realisierung des Agenten wird von der abstrakten Architektur nicht geregelt, nicht einmal ein grober Lebenszyklus ist Bestandteil der Architektur, da konkrete Architekturen hier unterschiedliche Wege gehen können und dürfen.

Bereits mehrfach gefallen ist der Begriff „Dienst“ („*service*“). Darunter versteht die abstrakte Architektur einen „funktional stimmigen Satz von Mechanismen, welche die Arbeit von Agenten und Diensten unterstützen“ [FIPA00001, S. 35]. Zu diesem Zweck bietet ein Dienst bestimmte Verhaltensweisen und Tätigkeiten öffentlich an und wird von einer Dienstbeschreibung begleitet. Ob ein Dienst als Agent implementiert und ansprechbar ist oder ob der Dienst durch andere Softwareschnittstellen wie z.B. Methodenaufrufe erreicht werden kann, lässt die abstrakte Architektur für die Definition durch konkrete Architekturen offen. Wenn aber ein Agent einen Dienst anbietet, verpflichtet er sich, einen Teil seiner Autonomie aufzugeben: Er muss die Semantik-Vorgaben des Dienstes einhalten und darf die Dienstleistung nicht willkürlich verweigern. Kann er eine Aufgabe nicht durchführen, muss er in der Fehlernachricht eine Erklärung für das Scheitern der Aktion angeben (z.B. mangelnde Zugriffsrechte, ungültige Anfrage usw.). Beispiele für Dienste sind der Nachrichtentransportdienst, der Verzeichnisdienst oder der Kodierungsumwandlungsdienst.

Für alle Architektur-Elemente, die Strukturen zur Aufnahme von Informationen darstellen (z.B. Nachrichten, Verzeichniseinträge oder Dienstbeschreibungen), verwendet die abstrakte Architektur ein gemeinsames Grundkonzept: Die Informationen werden in Schlüssel-Wert-Tupeln („*key-value-tuple*“, KVT) untergebracht. Ein KVT besteht aus einer ungeordneten Menge von Schlüssel-Wert-Paaren, wobei der Schlüssel ein sogenanntes Paar-Element („*pair-element*“) ist. Ein Paar-Element besteht wiederum aus einer Folge von Kürzeln („*token*“), die durch Punkte getrennt werden und eine Hierarchie von links (als höchste Ebene) nach rechts bilden.

Als Beispiel soll hier die in Abschnitt 4.2.5 vorgestellte ACL-Nachrichtenstruktur dienen: Die gesamte Nachricht ist ein KVT, welches Paare mit den jeweiligen Schlüsseln für das Performativ, den Inhalt, die Ontologie usw. aufnehmen kann. In voller Notation wären die genannten Schlüssel-Wert-Paare beispielsweise folgendermaßen darzustellen – wobei eine komplette Syntax für KVTs nicht Bestandteil der abstrakten Architektur ist, die Spezifikation beschränkt sich auf das allgemeine Konzept der Paar-Elemente und überlässt die Kodierungsregeln den konkreten Implementierungen:

```
org.fipa.standard.message.performative
                 org.fipa.standard.message.performative.request
org.fipa.standard.message.content "(get-description)"
org.fipa.standard.message.ontology
       org.fipa.standard.message.ontology.fipa-agent-management
```

Wenn die höheren Hierarchieebenen aus dem Kontext erschlossen werden können, darf ein Paar-Element auch aus einem einzigen Kürzel bestehen, es wird dann „unqualifiziert“ genannt. Diese Abkürzungsregel erlaubt in vielen Fällen die Vermeidung der unhandlich langen Zeilen. Die so mögliche Kurznotation erinnert stark an die bereits in Abschnitt 4.2.5 vorgestellte Zeichenkettendarstellung für ACL-Nachrichten (das folgende Beispiel gilt im Kontext `org.fipa.standard.message`):

```
performative request
content "(get-description)"
ontology fipa-agent-management
```

Im Rahmen der KVTs und Dienste ist eine eindeutige Vergabe der Kürzel, Namen, Bezeichner und ähnlicher Elemente von großer Bedeutung. Daher verspricht die abstrakte Architektur, dass die FIPA einen Namensraum verwaltet und kontrolliert, der die Eindeutigkeit der in der Architektur und anderen FIPA-Spezifikationen verwendeten Bezeichner garantiert. Herstellerspezifische Erweiterungen oder Alternativen

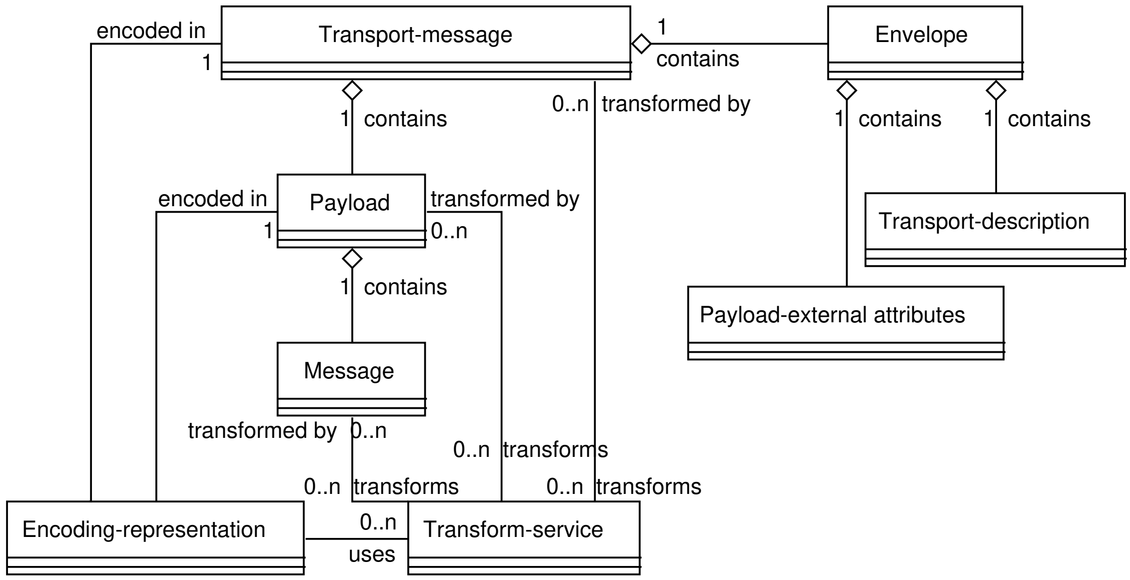


Abbildung 4.3: Elemente einer Transportnachricht (nach [FIPA00001, S. 39])

sowie proprietäre oder temporäre Ergänzungen müssen durch geeignete Präfixe kenntlich gemacht werden, so dass sie keinesfalls mit den offiziellen FIPA-Namen kollidieren können.

Problematisch ist in diesem Zusammenhang, dass es anscheinend noch kein Dokument gibt, welches alle Definitionen des FIPA-Namensraumes zusammenfasst. Einige Spezifikationen führen nicht einmal mehr vollständig qualifizierte Bezeichner für ihre Begriffe ein, so sind z.B. die `fipa-agent-management`-Ontologie oder die kommunikativen Akte bisher nur als unqualifizierte Namen definiert. Das obige Beispiel kann also falsch sein, was die voll-qualifizierten Bezeichner angeht.

4.3.2 Nachrichtentransport

Um die Interoperabilität beim Nachrichtentransport zu ermöglichen, beschäftigt sich die abstrakte Architektur mit zwei eng zusammenarbeitenden Elementen: zum einen dem Aufbau und der Verpackung einer Nachricht zu Transportzwecken, zum anderen dem Transportdienst als solchem.

Die Architektur-Elemente, aus denen sich eine Nachricht und ihre Transport-Verpackung zusammensetzen, werden in Abbildung 4.3 zueinander in Beziehung gesetzt.[3] Eine ACL-Nachricht (hier „Message“ genannt) wird als Nutzlast („Payload“) in die Transportnachricht („Transport-message“) eingebettet und um einen Umschlag („Envelope“) mit Begleitinformationen für den Transportdienst ergänzt. Dabei kann die

[3] Die Elemente in der Abbildung 4.3 sind im Gegensatz zum Original in [FIPA00001, S. 39] anders angeordnet, um die Schachtelung „Message“ – „Payload“ – „Transport-message“ grafisch zu verdeutlichen. Inhaltliche Unterschiede zum Original bestehen nicht.

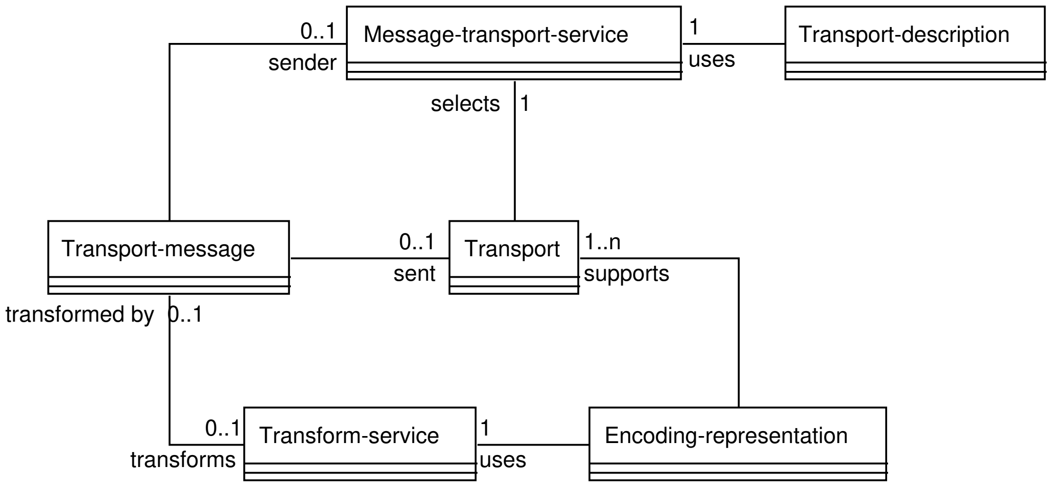


Abbildung 4.4: Nachrichtentransportelemente (nach [FIPA00001, S. 41])

Kodierung („Encoding-representation“) der Nachricht auf zwei Ebenen an die Transportgegebenheiten angepasst werden: Auf der inneren Ebene muss die Nachricht zwecks Verwendung als Nutzlast von ihrer beliebigen internen Darstellung in eine bekannte Kodierung umgewandelt werden. Dafür kommen z.B. alle in Abschnitt 4.2.6 genannten Repräsentierungen in Frage. Die abermalige Konversion der Kodierung der Nutzlast in der Transportnachricht ist optional und kann z.B. der Fehlererkennung oder Verschlüsselung dienen.

Zur Umwandlung einer Transportnachricht, ihrer Nutzlast oder auch des Nachrichteninhalts wird ein Transformationsdienst („Encoding-transform-service“) herangezogen. Dieser Dienst bietet den Agenten neben der eigentlichen Umkodierung auch Funktionen an, um die Kodierung eines gegebenen Nachrichtenteils zu erfragen oder alle bekannten Kodierungen aufzulisten.

Sowohl die Nachricht als auch der Transportumschlag enthalten Informationen über Absender und Empfänger der Nachricht. Dabei ist aber ein wichtiger Unterschied zu machen: Die Nachricht enthält die Informationen auf der Ebene der logischen Agentenkommunikation, d.h. die Angaben dienen der Identifizierung der Gesprächspartner. Demhingegen dienen die Empfängerangaben im Umschlag (als „Transport-description“) der transporttechnischen Adressierung – sie werden vom Nachrichtentransportdienst ausgewertet und gepflegt.

In Abbildung 4.4 werden diejenigen Elemente der abstrakten Architektur zueinander in Beziehung gesetzt, die am Transport einer Nachricht beteiligt sind. Das Hauptelement ist der Nachrichtentransportdienst („Message-transport-service“) selber. Dieser Dienst ermöglicht den Agenten den Versand und Empfang von transporttechnisch verpackten Nachrichten („Transport-message“). Deren Umwandlung in eine transportfähige „Encodung-representation“ durch den „Encoding-transform-service“ ist weiter oben bereits beschrieben worden.

Konkret soll der Versand und Empfang von Nachrichten laut der abstrakten Architektur durch die Erbringung der folgenden vier Dienstleistungen durch den Transportdienst möglich sein:

- Der Agent kann sich an einen Transport („Transport“ entspricht vielleicht in diesem Zusammenhang am ehesten dem Begriff „Verkehrsmittel“) *binden*, um in Zunkunft über diesen Transport Nachrichten empfangen und versenden zu können. Beim Binden wird eine Transportbeschreibung („Transport-description“) erstellt (mehr oder weniger vom Agenten vorgegeben, alle fehlenden Informationen werden vom Transportdienst nachgetragen), welche die Adressierung des Agenten über den Transport ermöglicht. Nach erfolgter Bindung wird der Transportdienst dem Agenten alle Nachrichten zustellen, die über den so geschaffenen Transport-Endpunkt eintreffen.
- Die Bindung eines Agenten an einen Transport kann wieder *gelöst* werden. Infolgedessen wird der Transportdienst dem Agenten auf diesem Wege keine Nachrichten mehr zustellen.
- Der Agent kann den Transportdienst auffordern, eine Transportnachricht gemäß der Informationen im Umschlag zu *versenden*.
- Der Transportdienst kann dem Agenten eine Nachricht *zustellen*, indem er die Zustellfunktion des Agenten nutzt.

Diese strenge Sicht als Dienst mit genau den hier aufgeführten Funktionen braucht in konkreten Architekturen aber nicht exakt umgesetzt zu werden, solange die grundlegenden Regeln des Dienstes eingehalten werden. Wenn z.B. in der konkreten Implementierung nur ein einziger Transport zur Verfügung stehen soll, kann dieser auch durch Betriebssystemschnittstellen oder andere Mechanismen realisiert werden – Bindungen sind dann vielleicht nicht nötig, die Dienstleistungen werden durch Systemfunktionen erbracht.

4.3.3 Verzeichnisdienst

Der Verzeichnisdienst ist das Verwaltungselement der abstrakten Architektur. Er stellt ein gemeinsam genutztes Lager dar, in dem Agenten ihre Namens-, Adress- und Dienstleistungsinformationen veröffentlichen und nach interessanten Informationen suchen können. Die Informationen werden in Verzeichniseinträgen organisiert, wobei jedem Agenten maximal ein solcher Eintrag zusteht. Ein Agent kann bei einem Verzeichnisdienst

- seinen eigenen Eintrag *registrieren*, wenn der Agent bei dem Dienst noch keinen Eintrag registriert hatte,
- seinen bereits im Verzeichnis registrierten Eintrag *modifizieren*,
- den eigenen Eintrag aus dem Verzeichnis *löschen*, oder

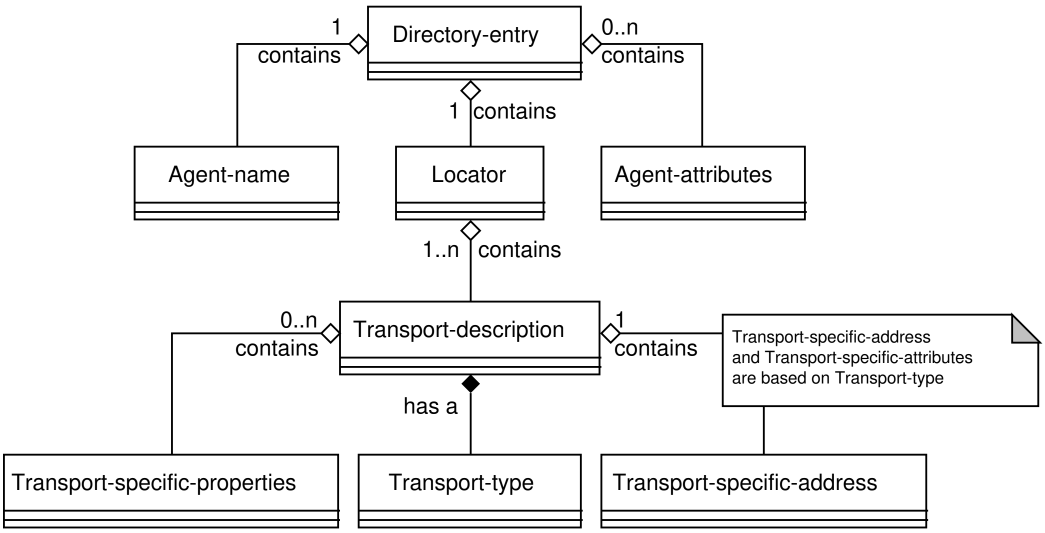


Abbildung 4.5: Elemente eines Verzeichniseintrags (nach [FIPA00001, S. 40])

- nach Verzeichniseinträgen, die auf bestimmte, vom Agenten vorzugebende Kriterien passen, *suchen*.

Diese vier Funktionen müssen von jedem Verzeichnisdienst geboten werden. Im Falle von nicht, doppelt vorhandenen oder ungültigen Einträgen oder bei Anfragen von nicht zugriffsberechtigen Agenten müssen entsprechende Fehlermeldungen generiert werden.

Es darf in einer Instanziierung der Architektur durchaus mehrere Verzeichnisdienste geben, mindestens einer ist aber erforderlich. Die Implementierung oder auch nur eine konkrete Schnittstelle der Verzeichnisdienste ist – wie üblich – nicht Gegenstand der abstrakten Architektur. Eine Einbindung von bereits bestehenden Softwaresystemen und -architekturen unter dieser konzeptuellen Dienstdefinition ist ausdrücklich erlaubt und angedacht. Damit das Zusammenspiel zwischen verschiedenen Umsetzungen eines Verzeichnisdienstes (und natürlich auch den nutzenden Agenten) möglichst reibungslos klappt, sollte eine gemeinsame Repräsentierung für Verzeichniseinträge verwendet werden. Über die Repräsentierungsdetails lässt sich die abstrakte Architektur nicht näher aus, aber ähnlich dem Vorgehen bei der Agentenkommunikationssprache wird zumindest die Grobstruktur von Verzeichniseinträgen vorgegeben.

Ein Verzeichniseintrag („Directory entry“) setzt sich aus mehren Elementen zusammen, die in Abbildung 4.5 dargestellt werden.[4] Dazu gehören der Name des Agenten („Agent-name“), genau eine Ortsangabe („Locator“) und optionale weitere Eigenschaften des Agenten („Agent-attributes“). Der Name dient der Identifikation des Agenten,

[4] Die Abbildung 4.5 ist gegenüber der Quelle [FIPA00001, S. 40] korrigiert. Im Original sind die Kardinalitätsangaben der „contains“-Beziehungen vom „Directory-entry“ zum „Locator“ und vom „Locator“ zur „Transport-description“ vertauscht. Hier sind sie an die textuelle Definition in [FIPA00001, S. 25 u. 31] angepasst.

ändert sich während der Lebenszeit des Agenten nicht und sollte im Normalfall eindeutig sein.[5] Die weiteren Eigenschaften, die im Eintrag angegeben werden können, dienen als mögliche Kriterien für eine Suche nach interessanten Einträgen im Verzeichnis. Der konkrete Inhalt dieser Attribute ist anwendungsabhängig – denkbar sind z.B. vom Agenten gebotene Dienste, beherrschte Ontologien, bekannte Protokolle, verstandene Sprachen usw.

Der Locator dient der Adressierung von Nachrichten an diesen Agenten – einer der wesentlichen Gründe für die Existenz des Verzeichnisdienstes. Zu diesem Zweck nimmt der Locator die bei der Bindung des Agenten an einen Transport (siehe voriger Abschnitt 4.3.2) erstellten Transportbeschreibungen („Transport-description") auf. Jede Transportbeschreibung legt eine Transportart („Transport-type") fest, über die der Agent erreicht werden kann. Abhängig von der Transportart spezifizieren eine Adresse („Transport-specific-address") und weitere Attribute („Transport-specific-properties") die technische Transport-Verbindung zum Agenten genauer.

4.4 FIPA-2000-Architektur

Unter dem Schlagwort „FIPA2000" wird ein Satz von Spezifikationen der FIPA zusammengefasst, die gemeinsam alle wesentlichen Aspekte der Agentenkommunikation abdecken: Die Kommunikationssprache, den Nachrichtentransport und die Verwaltung von Agenten und Agentenadressen. Hält sich der Entwickler einer Agentenplattform an diese Spezifikationen, so entsteht eine FIPA2000-konforme Plattform, deren Agenten mit Agenten auf anderen FIPA2000-konformen Plattformen kommunizieren können.

Der Agentenkommunikationssprache war der Abschnitt 4.2 gewidmet, die verbleibenden zwei Themenblöcke Transport und Verwaltung wurden im Abschnitt 4.3 bereits aus der Sicht der über bzw. neben den FIPA2000-Standards stehenden abstrakten Architektur behandelt. In diesem Abschnitt sollen nun die für die Implementierung einer FIPA2000-konformen Agentenplattform notwendigen Elemente vorgestellt werden. Diese Elemente decken ebenfalls den Bereich Transport und Verwaltung ab, beschäftigen sich damit aber auf einer deutlich konkreteren Ebene als die abstrakte Architektur.

In Abbildung 4.6 ist das Referenzmodell einer FIPA2000-konformen Agentenumgebung dargestellt. Die Agenten („Agent") als grundlegende, ausführende Einheiten werden auf der durch die Agentenplattform („Agent Platform", AP) bereitgestellten physischen Infrastruktur eingesetzt. Ein Agent kann auf die verschiedenen Dienste der Plattform und anderer Agenten zugreifen sowie externe, nicht-agentenorientierte „Software" verwenden oder auch mit menschlichen Benutzern zusammenarbeiten.

In jeder Plattform werden mindestens drei Dienste angeboten: Ein Nachrichtentransportdienst („Message Transport Service"[6], MTS) ermöglicht Agentenkommunikation so-

[5] Da es in einem Multiagentensystem kein globales Wissen gibt, kann die Eindeutigkeit des Agentennames nur soweit sichergestellt werden, wie das Wissen der namengebenden Instanz reicht.

[6] Die in Abbildung 4.6 verwendete Bezeichnung „Message Transport System" wird im Text der Spezifikation [FIPA00023] nicht verwendet. Daher wird hier auf die (vermutlich aktuellere) Bezeichnung „Message Transport Service" Bezug genommen.

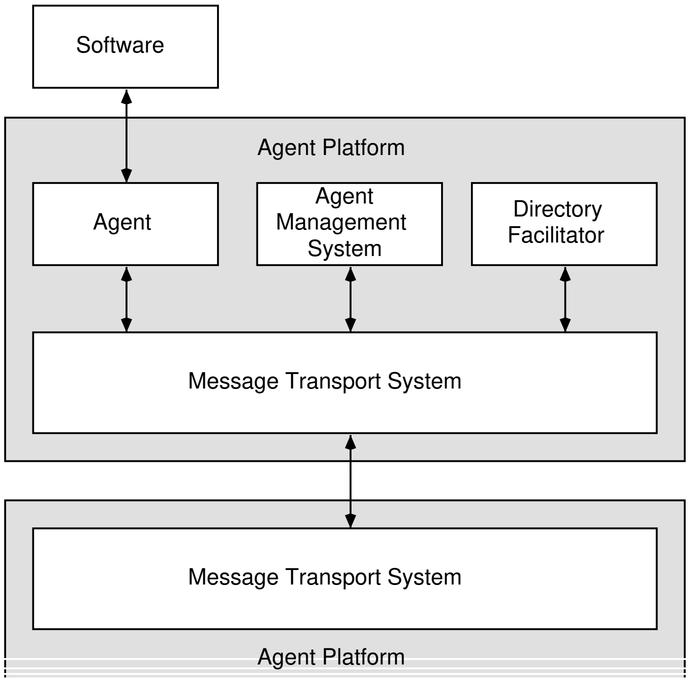


Abbildung 4.6: Referenzmodell einer FIPA2000-Plattform (aus [FIPA00023, S. 2])

wohl innerhalb der Plattform als auch plattformübergreifend. Ein Verwaltungsdienst („Agent Management System", AMS) und ein oder mehrere Verzeichnisdienste („Directory Facilitator", DF) registrieren Agenten und die von ihnen angebotenen Dienste, um das Zusammenspiel der Agenten auf der Plattform zu regeln. Alle drei Dienste werden in den beiden Spezifikationen [FIPA00023] und [FIPA00067] beschrieben und in den hier folgenden Abschnitten vorgestellt.

4.4.1 Nachrichtentransport

Die Aufgaben und Arbeitsweise des Nachrichtentransportdienstes auf einer FIPA-2000-konformen Plattform werden nebst der zugehörigen Nachrichtentransport-Ontologie in der „FIPA Agent Message Transport Service Specification" [FIPA00067] festgelegt. Der Dienst wird vom sogenannten „Agent Communication Channel" (ACC) angeboten und ist im wesentlichen für die plattformübergreifende Kommunikation zuständig – die interne Schnittstelle zu den Agenten wird ebenso wie die internen Kommunikationswege nicht spezifiziert. Damit bleibt der Weg frei für plattformspezifische Kommunikationswege, die eventuell effizienter sind oder mehr Möglichkeiten bieten als die FIPA-konforme Kommunikation mittels standardisiert kodierter ACL-Nachrichten.

Die externe(n) Schnittstelle(n) des ACC werden durch die standardisierten Nachrichtentransportprotokolle („Message Transport Protocol", MTP) festgelegt. Bisher sind in einzelnen Spezifikationen die bekannten Internetprotokolle HTTP [FIPA00084], IIOP [FIPA00075] und WAP [FIPA00076] für den FIPA-Nachrichtentransport aufbereitet worden. Welche(s) dieser Protokolle eine Agentenplattform implementiert, ist offen. Zwar gibt es mit den Spezifikationen [FIPA00077] und [FIPA00078] ausgearbeitete Transport-Profile, die die von einer Plattform zu unterstützenden Transportprotokolle vorgeben, aber diese Spezifikationen sind (anscheinend ersatzlos) als „obsolete“ eingestuft worden.

Für transporttechnische Informationen wird die „FIPA-Agent-Management“-Ontologie (siehe folgender Abschnitt 4.4.2) um einige Elemente ergänzt, darunter eine Beschreibungsmöglichkeit für die Transportfähigkeiten der Plattform. Diese `ap-transport-description` kann als Teil der Plattformbeschreibung vom AMS (siehe folgender Abschnitt 4.4.2) erfragt werden – allerdings bietet [FIPA00067] keine Lösung für das Henne-Ei-Problem, mit welchem Transportprotokoll die Anfrage nach den verfügbaren Transportprotokollen transportiert werden soll[7].

Ein zweites, wesentliches Element der Transportontologie ist der Nachrichtenumschlag (`envelope`). Ähnlich wie in der abstrakten Architektur beschrieben (siehe Abschnitt 4.3.2), werden im Umschlag die transporttechnischen Informationen zu einer Nachricht verwaltet. Dazu gehören Absender- und Empfängeradressen, Angaben über die Kodierung der Nachricht sowie Zeitstempel von jedem ACC, den die Nachricht während ihres Transports passiert hat. Im Unterschied zu allen anderen Elementen der Agent-Management-Ontologie dürfen in einem Umschlag niemals Informationen überschrieben werden – stattdessen werden weitere Werte mit demselben Schlüssel eingetragen, der jeweils jüngste Wert zählt.

Die konzeptionelle Trennung, die in der abstrakten Architektur zwischen der Agentenidentifikation und der transporttechnischen Adressierung gezogen wird, existiert in den FIPA2000-Spezifikationen nur zum Teil. Zwar arbeiten Agent und Transportdienst auf verschiedenen Ebenen (Nachricht bzw. Umschlag) mit Identifikations- oder Adressinformationen, aber für beide Zwecke wird eine Datenstruktur namens „`agent-identifier`“ verwendet. Diese Struktur nimmt sowohl den identifizierenden Agentennamen als auch eine Liste von URLs der Transportschnittstellen auf, über die der Agent erreichbar ist. Über die Ideen der abstrakten Architektur hinaus geht die Möglichkeit, im `agent-identifier` eine Liste weiterer `agent-identifier` abzulegen, welche auf Namensauflösungsdienste (siehe folgender Abschnitt 4.4.2) verweisen. Wie die technischen Informationen im `agent-identifier` ermittelt werden, wird allerdings – abweichend von der abstrakten Architektur – nicht thematisiert.

4.4.2 Verwaltung und Verzeichnisdienste

Zur erfolgreichen Kommunikation benötigt ein Agentensystem nicht nur ein einheitliches Datenaustauschformat und Konventionen zur Interpretation der Nachrichten, sondern auch feste Ansprechpartner unter bekannten Adressen. Über diese können

[7] In den Transportprofilspezifikationen [FIPA00077] und [FIPA00078] wurde jeweils ein Basis-Transportprotokoll für diesen Zweck vorgegeben. Aber diese Spezifikationen sind – wie im Text bereits erwähnt – nicht mehr gültig.

dann die Existenz anderer Agenten nebst deren Dienstleistungen und Aktivitätszustand in Erfahrung gebracht werden, um potentielle Kommunikationspartner ausfindig zu machen.

Auf einer FIPA2000-konformen Plattform stehen zu diesem Zweck gemäß der Spezifikation [FIPA00023] zwei Verwaltungsdienste zur Verfügung. Sie reagieren wie ein Agent und sind unter festen, definierten Adressen auf jeder Agentenplattform erreichbar. Die Verwaltungsagenten verwenden eine ebenfalls in dieser Spezifikation definierte „Agent-Management-Ontology", welche die von den Agenten verstandenen Funktionen und Datenstrukturen vorgibt. Um die Kommunikation mit den Verwaltungsagenten vollends zu standardisieren, wird die Ontologie mit der Inhaltssprache SL0 (siehe Abschnitt 4.2.3) und dem „Request"-Interaktionsprotokoll (siehe Abschnitt 4.2.2) zusammen verwendet.

Unter der Adresse „`df@`*hap*" (wobei „*hap*" für den Namen der Agentenplattform steht) ist der Standard-„Directory Facilitator" (DF) erreichbar. Ein DF bietet ein Diensteverzeichnis analog den „Gelben Seiten" der Telefongesellschaften an: Agenten können sich hier mit ihren eigenen Dienstleistungen registrieren und Anbieter anderer Dienste in Erfahrung bringen. Dafür stehen in der Verwaltungsontologie mehrere definierte Funktionen namens „register", „deregister", „modify" und „search" zur Verfügung.

Es darf mehrere DF-Agenten mit verschiedenen Adressen auf einer Plattform geben, der DF-Agent unter der „`df`"-Adresse stellt nur eine Minimalanforderung dar. DF-Agenten können sich mit ihrer Verzeichnisdienstleistung bei anderen DF-Agenten eintragen. Dabei besteht die Möglichkeit, eine Föderation von Verzeichnisdiensten zu bilden, so dass eine an einen DF gerichtete Anfrage automatisch an andere DF-Agenten weitergereicht werden kann. Auf diese Weise kann das Anfrageergebnis auch über mehrere Plattformen hinweg zusammengetragen werden.

Das „Agent Management System" (AMS) hat die Adresse „`ams@`*hap*" und führt eine Liste aller auf der Plattform registrierten Agenten. Jeder auf der Plattform beheimatete Agent muss beim AMS registriert sein, Agenten von außerhalb können sich zusätzlich registrieren. Damit stellt das AMS einen Verzeichnisdienst im Stile eines Telefonbuchs, auch „weiße Seiten" genannt, zur Verfügung. Die in der Verwaltungsontologie definierten Verzeichnisdienstfunktionen unterscheiden sich beim AMS von denen des DF nur im Aufbau der Verzeichniseinträge.

Über den Verzeichnisdienst hinaus stellt das AMS die verbindende Schnittstelle zwischen den Agenten und der Funktionalität der Plattform dar: Erzeugung und Terminierung sowie andere elementare Zustandswechsel eines Agenten können über das AMS angefordert werden. Die möglichen Zustandswechsel werden in der „Agent Management Specification" [FIPA00023] durch das in Abbildung 4.7 wiedergegebene Diagramm vorgegeben.

Der in diesem Diagramm dargestellte Lebenszyklus eines Agenten beginnt mit seiner Erzeugung, die in zwei Schritten („Create" und „Invoke") geschieht. Danach befindet sich der Agent im Zustand „Active", in dem er ganz normal arbeiten und Nachrichten versenden bzw. empfangen kann. Die Zustände „Waiting" und „Suspended" symbolisieren Schlafzustände, in denen der Agent keine Nachrichten empfangen kann. Weitere Auswirkungen dieser Zustände, die über den Nachrichtentransport hinausgehen,

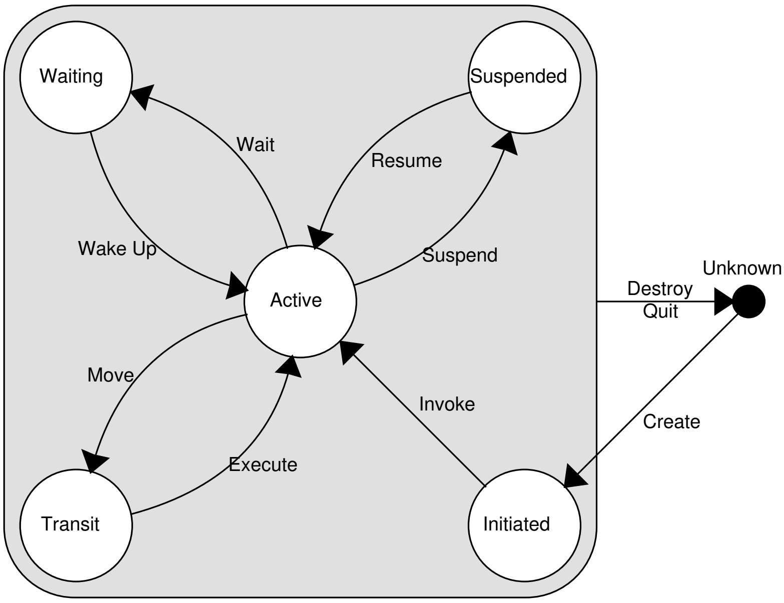


Abbildung 4.7: Der Lebenszyklus eines FIPA2000-Agenten (aus [FIPA00023, S. 9])

werden nicht spezifiziert. Der Zustand „Transit“ wird für die Migration von mobilen Agenten benötigt, wie sie in der „FIPA Agent Management Support for Mobility Specification“ [FIPA00087] grob umrissen wird.[8]

Die Beendigung eines Agenten ist eine Funktion jedes Agenten, die vom AMS angefordert werden kann. Dabei stehen zwei Varianten der Terminierung zur Verfügung: Auf eine „Quit“-Anforderung hin kann der Agent sich beenden, muss es aber nicht tun. Ein „Destroy“ hingegen wird vom AMS zwangsweise durchgesetzt.

4.5 Nicht standardisiert

Die Spezifikationen der FIPA haben das Ziel, Interoperabilität zwischen Agentensystemen unterschiedlicher Herkunft herzustellen. Zu diesem Zweck beschränken sich die Spezifikationen im wesentlichen darauf, die Kommunikation zwischen den Agenten zu standardisieren. Insbesondere in der abstrakten Architektur wird in einem eigenen Abschnitt [FIPA00001, Abschn. 2] erläutert, welche Aspekte eines Multiagentensystems abgedeckt und welche explizit ausgeklammert werden. Aber auch die konkreten Spezifikationen der FIPA2000-Suite sparen bewusst eine Definition interner Schnittstellen und Techniken einer Agentenplattform aus.

So ist ein Agentenplattformentwickler frei in der Modellierung und Implementierung der Agenten selber. Der Grad der Autonomie, Nebenläufigkeit und Flexibilität

[8] Da die Unterstützung von Mobilität bei der Entwicklung der Agentenplattform in dieser Arbeit nicht im Vordergrund steht, erfolgt eine knappe Vorstellung der entsprechenden FIPA-Spezifikation erst im Abschnitt 5.3.2.

der Agenten kann je nach den jeweiligen Bedürfnissen gewählt werden. Ob ein Agent in einem eigenen oder gar mehreren Threads ausgeführt wird oder ob alle Agenten der Reihe nach in einem Zeitscheibenverfahren zur Ausführung kommen, ist für die Interoperabilität der Agentenplattformen untereinander irrelevant. Ebenso können Agenten in unterschiedlichen Programmiersprachen implementiert werden, solange die Kommunikation mit anderen Agenten standardkonform erfolgt.

Die FIPA-Spezifikationen gehen sogar soweit, die Kommunikation zwischen Agenten innerhalb derselben Plattform aus der Standardisierung auszuklammern. Dementsprechend dürfen lokale Agenten untereinander mit abweichendem, womöglich effizienteren Nachrichtenaufbau in eigenen Repräsentierungen über plattformspezifische Wege kommunizieren. Lediglich die plattformübergreifende Kommunikation muss notwendigerweise an den Standards ausgerichtet sein.

Große Teile einer Agentenplattform sind eng mit der Implementierungsweise der Agenten verknüpft. Dazu gehört natürlich die Infrastruktur, welche die Ausführung der Agenten entsprechend der gewählten Programmiersprache ermöglicht. Des weiteren sind alle Schnittstellen zwischen Agent und Plattform implementierungsabhängig, z.B. die interne Schnittstelle zum Nachrichtentransportdienst. Dementsprechend werden diese Schnittstellen von der Standardisierung ausgenommen.

Der in Abschnitt 4.4.2 vorgestellte Lebenszyklus eines Agenten ist ebenfalls stark implementierungsabhängig. Daher beschränkt sich die Definition der Zustände und Übergänge auf eine abstrakte Ebene, die viele Freiheiten bezüglich der Auswirkung der jeweiligen Zustände auf die Agenten lässt. Die abstrakte Architektur klammert die Betrachtung des Agenten-Lebenszyklus' gar explizit aus.

Mit der Abstraktionsebene des FIPA-2000-Lebenszyklus' geht einher, dass die Funktionen des „Agent Management System“ (AMS) bezüglich der Zustandsübergänge nicht weiter konkretisiert werden. Das hat zur Folge, dass die Nachrichten, welche Zustandsübergänge vom AMS anfordern (z.B. die Erzeugung eines Agenten), keine feste Struktur aufweisen. Bei diesen Funktionen ist daher eine Kommunikation über Plattformgrenzen hinweg nicht ohne weiteres möglich, weil die Standardisierung fehlt. Lediglich im Zusammenhang mit der Mobilitätsspezifikation [FIPA00087] wird das Nachrichtenformat der zwei relevanten Zustandsübergänge genauer definiert, weil hier eine plattformübergreifende Kommunikation stattfinden muss.

4.6 Implementierungen

Es existiert bereits eine Anzahl von FIPA-konform implementierten Agentenplattformen. Eine Reihe von öffentlich verfügbaren Plattformen werden auf der Homepage der FIPA aufgeführt (siehe [FIP2002, /resources/livesystems.html]). Da die Entwicklung etlicher Plattformen bereits vor mehreren Jahren begonnen hat, unterstützen sie ursprünglich die älteren FIPA-Spezifikationssätze FIPA97 bzw. FIPA98. Der Übergang auf die FIPA2000-Spezifikationen vollzieht sich eher allmählich. Einige der auf der Programmiersprache Java aufsetzenden Implementierungen möchte ich hier kurz vorstellen.

FIPA-OS [FIP2001]. Unter der Projektbezeichnung „**FIPA O**pen **S**ource“ entstand die erste frei verfügbare FIPA-konforme Plattformimplementierung. FIPA-OS ist eine Klassenbibliothek, die Komponenten zur Implementierung FIPA-konformer Agenten bereitstellt. Die derzeitige Version 2.1.0 unterstützt viele Spezifikationen der FIPA2000-Suite. Neben der obligatorischen Agentenverwaltung wird der Nachrichtentransport mit verschiedenen Transportprotokollen und Nachrichtenrepräsentierungen ermöglicht. Dazu kommen mehrere Inhaltssprachen sowie diverse von der FIPA spezifizierte Interaktionsprotokolle.

Ein Agent wird auf Basis einer sogenannten „Shell“ implementiert, die als Java-Klasse verschiedene vorgefertigte Methoden zur Abwicklung von Konversationen enthält. Wenn eine Nachricht eintrifft oder ein Planungsprozess zu einem Ergebnis gelangt ist, wird die Shell des Agent aufgerufen. Die Reaktion im Agenten wird auf Java-Ebene implementiert, wobei die Funktionalität des Agenten in abgeschlossene Aufgaben zerlegt und mittels des vorhandenen Task-Managers nebenläufig ausgeführt werden kann. Ein Konversationsmanager bietet Ablaufsteuerung und Überwachung der Kommunikation im Hinblick auf die bekannten Interaktionsprotokolle an.

JADE [BRP+2002]. Das „**J**ava **A**gent **De**velopment Framework“ ist vor einiger Zeit ebenfalls unter eine Open-Source-Lizenz gestellt worden, so dass es jetzt frei verfügbar ist. Die derzeitige Version 2.6 bietet eine FIPA2000-konforme Agentplattform, die über mehrere Rechner verteilt laufen kann. Die Kommunikation zwischen Agenten wird vom Transportdienst der Plattform automatisch über den jeweils effizientesten Weg (z.B. lokale Methodenaufrufe, plattforminterne Netzwerkkommunikation, FIPA-konforme externe Kommunikation) zum Ziel geleitet. Zwecks Repräsentierung von Nachrichten werden für die Verwaltungsontologie und die Inhaltssprache SL Java-Klassenbibliotheken bereitgestellt, die Umsetzung in offizielle Repräsentierung erfolgt im Transportdienst automatisch.

Ähnlich wie bei FIPA-OS wird ein Agent auf Basis einer Shell implementiert, wobei jedem Agenten ein eigener Thread zugewiesen wird. Innerhalb jedes Agenten wird ein kooperatives Scheduling zwischen sogenannten „Behaviours“ durchgeführt. Einige definierte Interaktionsprotokolle werden in Form solcher Behaviours zur Verfügung gestellt.

JADE kommt mit einer grafischen Oberfläche daher, die eine Überwachung und Steuerung der Plattform und der darauf lebenden Agenten über das Netzwerk ermöglicht. Spezielle, zusätzliche Agenten dienen der Beobachtung der Agenten und der ablaufenden Nachrichtenkommunikation.

JAS [MAB+2002]. Unter dem Namen „**J**ava **A**gent **S**ervices“ läuft ein Projekt, in dem die abstrakte Architektur der FIPA in eine Java-Klassenbibliothek gegossen werden soll. Der Java-Standardisierungsprozess [JCP2002] der JAS-Klassenbibliothek ist bereits bis zur vorletzten Phase der öffentlichen Begutachtung fortgeschritten.

Da JAS im Gegensatz zu den beiden vorher vorgestellten Implementierungen nicht die konkreten FIPA-Spezifikationen sondern die abstrakte Architektur nachbildet, ergibt sich eine abweichende Funktionalität. Die Klassenbibliothek implementiert keine fertige Plattform, sondern stellt ein Gerüst bereit, in das konkrete Dienstimplementierungen hineingestellt werden können. Die Implementierungen sind nicht Bestandteil von JAS, allerdings existiert außerhalb der standardisierten Bibliothek eine beispielhafte Implementierung einfacher Dienste.

Über ein Dienstwurzelobjekt, das jedem Agenten zur Verfügung gestellt wird, kann ein Agent die für ihn benötigten Dienste erreichen. Die im Gerüst vorgesehen Dienste beinhalten den Nachrichtentransport sowie die Nachrichtenrepräsentierung und -konversion. Der Nachrichtentransport wird entsprechend der Vorgaben der abstrakten Architektur an jeden Agenten einzeln gebunden statt einen zentralen ACC für die plattformübergreifende Kommunikation vorzusehen.

Um die Menge der benötigten Transformationsdienste zwischen n Inhaltssprachen und m Repräsentierungen klein zu halten, sieht JAS eine allgemein gehaltene Austauschstruktur vor. Diese Struktur kann syntaktische Gruppierungen der Sprachelemente aufnehmen, die von jeglichen semantischen Hintergrundinformationen befreit sind. Die Repräsentierung der Strukturen kann dann unabhängig von der Sprache auf Basis der Austauschstruktur geschehen. Es werden also nur $n + m$ Transformationsmodule (anstelle von $n \cdot m$ Modulen für die Abdeckung aller Kombinationen) benötigt. Die eine Gruppe von Modulen transformiert zwischen der Austauschstruktur und der eigentlichen Sprache, die andere Gruppe zwischen der Austauschstruktur und der endgültigen Repräsentierung. Eine Umwandlung von einer Sprache in eine andere leistet dieses Konzept jedoch nicht, weil die Austauschstruktur keine semantischen Informationen mehr enthält.

Die vorgestellten Implementierungen beinhalten jeweils einen Teil der Funktionalität, die von der in dieser Arbeit implementierten Agentenplattform ebenfalls bereitgestellt werden muss. Die Übernahme einer bestehenden Implementierung als Basis für die neue Plattform scheint mir aber nicht praktikabel zu sein.

Die beiden an der FIPA-2000-Suite orientierten Plattformen FIPA-OS und JADE betreiben viel Aufwand, um den ausgeführten Agenten Nebenläufigkeit zur Verfügung zu stellen. Dieser Aufwand wird für eine in Mulan integrierte Agentenplattform nicht benötigt, weil die Mulan-Agentennetze alle Nebenläufigkeitserfordernisse abdecken. Von JADE oder FIPA-OS könnten damit nur noch einzelne unterstützende Klassen zum Netzwerktransport oder der Nachrichtenrepräsentierung übernommen werden, was mir moralisch bedenklich erscheint.

Das JAS-Gerüst würde sich hingegen aufgrund seiner Abstraktheit gut eignen, um den Rahmen für die neue Plattform zu stellen. Dagegen spricht nur, dass alle existierenden Plattformimplementierungen sich nach den älteren, konkreteren FIPA-Spezifikationen richten. Die angestrebte Interoperabilität mit bestehenden Plattformen anderer Hersteller wäre bei einer Integration in das JAS-Gerüst nicht gegeben, weil die abstrakte Architektur derzeit nicht interoperabel mit den konkreten Spezifikationen ist. Daher ist auch diese Option nicht attraktiv.

Kapitel 5

Realisierte Plattform

Die in Kapitel 3.2 beschriebene Multiagentensystem-Architektur[1] MULAN modelliert ein vollständiges Multiagentensystem. MULAN wird in Form von Referenznetzen (siehe Kapitel 2.3) spezifiziert, die über eine operationale Semantik verfügen. Somit ist das Modell in der Lage, Plattformen und Agenten mit ihrem Verhalten so zu beschreiben, dass das beschriebene System ausführbar ist.

Die Schnittstellen in der MULAN-Architektur werden in Form von synchronen Kanälen angegeben, und die darüber ausgetauschten Daten – abgesehen von der Festlegung auf die Kommunikation mittels Nachrichten – nicht näher spezifiziert. Für ein real einsetzbares Multiagentensystem, dessen Agenten auf physisch verteilten Agentenplattformen leben und trotzdem miteinander kommunizieren sollen, reicht die lokale Spezifikation von synchronen Kanälen als Schnittstelle nicht aus. Wenn zusätzlich Kommunikation mit Agentenplattformen stattfinden soll, die einer anderen Multiagentensystem-Architektur entspringen, müssen Schnittstellen definiert werden, die allen beteiligten Architekturen gemein sind.

Die in Kapitel 4 vorgestellten FIPA-Spezifikationen beschreiben eine Kommunikationsarchitektur, die genau diese Zielsetzung der Interoperabilität zwischen verschiedenen Multiagentensystem-Architekturen verfolgt. In den FIPA-Spezifikationen wird eine standardisierte Schnittstelle der Agentenplattform nach außen definiert, mit festgelegten Ansprechpartnern, Datenformaten und Kommunikationsregeln. In Bezug auf die Plattform- oder Entscheidungsarchitektur eines Multiagentensystems machen die FIPA-Spezifikationen bewusst keine Vorgaben außerhalb der natürlichen Überlappung mit einer Kommunikationsarchitektur.

MULAN hingegen ist eine Multiagentensystem-Architektur, die zwar mit einer Entscheidungsarchitektur und einer prototypischen Plattformarchitektur daherkommt, es aber erlaubt, diese auszutauschen. Zur Kommunikationsarchitektur werden kaum Vorgaben gemacht, die einzige feststehende Annahme im MULAN-Modell ist die ausschließliche Kommunikation mittels Nachrichten über synchrone Kanäle. MULAN und FIPA sind also komplementär: MULAN steuert die ganzheitliche Sicht und eine Entscheidungsarchitektur bei, FIPA legt die Kommunikationsarchitektur fest, und die

[1] Die in dieser Arbeit vorgenommene Differenzierung des Begriffs „Architektur“ wird in Abschnitt 3.1.4 erläutert.

Plattformarchitektur kann im Rahmen der Überschneidung der Vorgaben aus den benachbarten Architekturen frei gestaltet werden. Der Gedanke, die Architekturen von Mulan und FIPA miteinander zu verbinden, liegt daher nahe.

Genau dieser Weg wird in dieser Arbeit auch beschritten. Die bereitgestellte Agentenplattform trägt den Namen CAPA (für **C**oncurrent **A**gent **P**latform **A**rchitecture), um das besondere Interesse an einer hohen Nebenläufigkeit innerhalb der Plattform zu betonen.

Zunächst wird in Abschnitt 5.1 die entworfene Plattformarchitektur vorgestellt, mit besonderem Augenmerk auf die den Agenten zur Verfügung stehenden Dienste. Dazu gehören insbesondere die Übergänge zur Kommunikationsarchitektur der FIPA und zur Multiagentensystem-Architektur Mulan.

Im Abschnitt 5.2 geht es dann um die Umsetzung der Plattformarchitektur aus Abschnitt 5.1 in eine Referenznetz- und Java-Klassenstruktur. Dabei wird die Arbeitsweise und Implementierung der einzelnen Plattformelemente beschrieben. Ein wiederkehrender interessanter Punkt ist die Frage, ob ein Element als Referenznetz oder als Java-Klasse realisiert werden sollte.

Eine Bewertung und Diskussion der Eigenschaften der erstellten Plattform sowie eine Nachbetrachtung der Entscheidung für die Kombination von Mulan und FIPA folgen im Abschnitt 5.3.

5.1 Plattformarchitektur

Einer der Hauptgründe für die Neuentwicklung einer Agentenplattform im Rahmen der Mulan-Architektur ist der Schritt vom Modell eines verteilten Systems zu einem tatsächlich physisch verteilten System. Dieser Schritt bedeutet, die explizit modellierte Kommunikationsstruktur des Mulan-Modells (siehe Abbildung 3.1) durch Softwareschichten für die Netzwerkkommunikation zu ersetzen. Damit ist klar, dass die Systemsicht des Mulan-Modells zunächst funktionslos wird.[2]

Der zweite Hauptgrund ist der Wunsch, von einem homogenen Mulan-System zu einem heterogenen Multiagentensystem mit Plattformen verschiedener Architekturen wechseln zu können. Das kann nur funktionieren, wenn gemeinsame Kommunikationsstandards eingehalten werden, weswegen CAPA mit den FIPA-Spezifikationen konform gehen soll. Die FIPA-Spezifikationen schreiben die Existenz bestimmter Plattformdienste vor und legen deren öffentliche Schnittstellen fest, machen aber keine Vorschriften bezüglich der Implementierung dieser Dienste oder deren innerer Schnittstellen zu den lokalen Agenten. Die Freiheiten, die die FIPA bezüglich der Implementierung lässt, erlauben es, die prototypische Mulan-Plattform als Grundlage für die Neuentwicklung weiter zu verwenden. Allerdings werden Änderungen und Erweiterungen an den gebotenen Plattformdiensten (Agentenverwaltung, lokale Kommunikation und exter-

[2] Für die weitere Entwicklung besteht die Idee, die Systemsicht des Mulan-Modells wiederzuverwenden, um die Kommunikation zwischen CAPA-Plattformen zu beobachten (oder sogar zu konfigurieren). Da diese Funktionalität aber für die Kommunikation zwischen FIPA-konformen Plattformen weder notwendig noch hinderlich ist, wird sie erst einmal außen vor gelassen.

ne Kommunikation) in einem Umfang notwendig, der es sinnvoll erscheinen lässt, die Funktionalität des einen MULAN-Plattformnetzes auf mehrere Plattformelemente zu verteilen.

Weil es mehr Plattformarchitekturen gibt, die sich den konkreten FIPA2000-Spezifikationen oder deren Vorgängern FIPA97 und FIPA98 verschrieben haben, als Umsetzungen der abstrakten FIPA-Architektur, wurden die konkreten FIPA2000-Standards als Grundlage für CAPA gewählt. Die in der abstrakten Architektur angeregte Flexibilität in Nachrichtenrepräsentation und -transport soll dennoch nicht aufgegeben werden, um spätere Agentenentwickler nicht zu sehr in ihren Möglichkeiten zu beschränken. Diese Flexibilität kann aber nur insoweit in die Plattform integriert werden, wie sie nicht den FIPA2000-Spezifikationen zuwiderläuft. Daraus ergibt sich, dass die flexiblen Aspekte der Dienste nur über nicht-standardisierte Schnittstellen zugänglich sein können oder aber transparent den Agenten untergeschoben werden müssen.

Die FIPA2000-Suite legt, wie in Kapitel 4 ausführlich vorgestellt, für eine Plattform vor allem die Agentenverwaltungs- und Nachrichtentransportdienste fest. Die Plattform muss einen wie einen Agenten ansprechbaren Verwaltungsdienst namens *Agent Management System (AMS)* und einen ebenso erreichbaren Verzeichnisdienst namens *Directory Facilitator (DF)* bieten. Für den Nachrichtentransport ist ein *Message Transport Service (MTS)* zuständig, dessen interne Schnittstelle beliebig (auch proprietär) sein darf. Nur die externe Schnittstelle ist über den sogenannten *Agent Communication Channel (ACC)* definiert, der für die Weiterleitung von ein- und ausgehenden externen Nachrichten zuständig ist. Der ACC muss ein FIPA-konformes *Agent Message Transport Protocol (MTP)* beherrschen, über das Nachrichten in der *Agent Communication Language (ACL)* verschickt werden.

Die korrekte Verwendung von ACL-Semantik und -Syntax ist Sache der Agenten, der Nachrichtentransportdienst muss lediglich korrekt verpackte Nachrichten befördern. Dennoch halte ich es für sinnvoll, den Agentenentwicklern vorgefertigte Strukturen anzubieten. Zum einen helfen solche Strukturen, Flüchtigkeitsfehler zu vermeiden – erst recht, wenn Plausibilitätsprüfungen integriert sind. Zum anderen muss die oft benötigte Implementierung von Parsern und Generatoren für die ACL-Ausdrücke nur einmal gemacht werden, wenn die Plattform diesen „Dienst“ anbietet. In der abstrakten Architektur wird diese Idee durch die Definition eines *Encoding Transform Service* angedeutet, der zwischen verschiedenen Repräsentierungen und Kodierungen von ACL-Nachrichten oder deren Inhalten umwandeln kann.

Damit liegen folgende Rahmenbedingungen für CAPA fest:

- Die MULAN-Systemsicht mit der Kommunikationsstruktur kann nicht mehr explizit modelliert werden, sondern entsteht implizit aus der realen Netzwerkkommunikationsstruktur.

- Das MULAN-Plattformnetz wird in seiner Funktionalität aufgespalten auf mehrere Plattformelemente. Als Maßgabe dienen die Elemente der FIPA2000-Architektur.

- Gemäß den FIPA2000-Spezifikationen müssen die Dienste AMS, DF und MTS (inklusive ACC) angeboten werden. Dabei müssen AMS und DF wie Agenten auf der Plattform angesprochen werden können. Um Agenten-Lebenszyklen verwalten zu können, benötigt der AMS-Agent eine Schnittstelle zum technischen Teil der Plattform – z.B. einen „Plattform-Agenten".
- Offen ist allerdings, wie die interne Schnittstelle zum MTS gestaltet werden soll – meine diesbezüglichen Überlegungen stellt der Abschnitt 5.1.1 vor.
- Offen ist auch, welches Transportprotokoll für die externe Kommunikation verwendet werden soll – oder ob die Idee aus der abstrakten Architektur, den Agenten zur Laufzeit die Auswahl der Transportprotokolle zu überlassen, eingebunden werden kann. Dieser Punkt wird in Abschnitt 5.1.2 diskutiert.
- Es sollen vorgefertigte Strukturen zur Repräsentierung von ACL-Nachrichten und deren Inhalten bereitstehen, nebst einem Transformationsdienst zur Umwandlung zwischen verschiedenen Repräsentierungen. Meine diesbezügliche Vorgehensweise wird in Abschnitt 5.1.3 erläutert.

In Abbildung 5.1 ist die resultierende Plattformarchitektur dargestellt. Die Pfeile deuten an, an welchen Stellen der Entwickler eines Multiagentensystems, das auf dieser Plattformarchitektur aufbaut, unkompliziert Erweiterungen vornehmen können soll. Natürlich muss er anwendungsspezifische Agenten erstellen und einbinden können, dies geschieht entweder durch die Definition von Mulan-Agentenprotokollen oder durch die Verwendung einer anderen Entscheidungsarchitektur, d.h. eines anderen Agentennetzes. Zusätzlich soll es möglich sein, den Nachrichtentransport durch andere Transportprotokolle an eigene Bedürfnisse anzupassen. Auch die leichte Integrierbarkeit alternativer Nachrichtenrepräsentierungen oder neuer Ontologien in Verbindung mit passenden Transformationsmodulen ist wünschenswert.

CAPA wird nur dann eine funktionsfähige Plattform sein, wenn die Plattformarchitektur in irgendeiner Programmiersprache implementiert wird. Wegen der engen Anbindung an die mit Referenznetzen implementierte Mulan-Architektur empfiehlt sich die Beibehaltung dieser Programmiersprache auch für die Plattform. Wobei natürlich die mit dem Renew-Simulator gegebene Möglichkeit der Integration von Java-Elementen durchaus genutzt werden soll und darf, wenn die Verwendung von Java-Code für bestimmte Teile effizienter erscheint als die Verwendung von Referenznetzen.

Die Implementierungsfrage wirft sich schon an dieser Stelle auf, wo es doch eigentlich noch um eine implementierungsunabhängige Architektur geht, weil damit auch die Möglichkeiten der späteren Agentenentwickler bestimmt werden. MadKit [GFM2002] bietet zum Beispiel eine Multiagentensystem-Architektur, in der die Agenten in beliebigen Programmiersprachen verfasst werden dürfen – solange ein unter Java laufender Interpreter der Programmiersprache existiert und eingebunden ist. Das JAS-Projekt [MAB+2002] setzt die Definitionen der abstrakten Architektur [FIPA00001] in Java-Interfaces um, was wiederum einige Vorgaben – darunter die Festlegung auf die Programmiersprache Java – für darauf aufbauende Plattformen mit sich bringt. Die Frage,

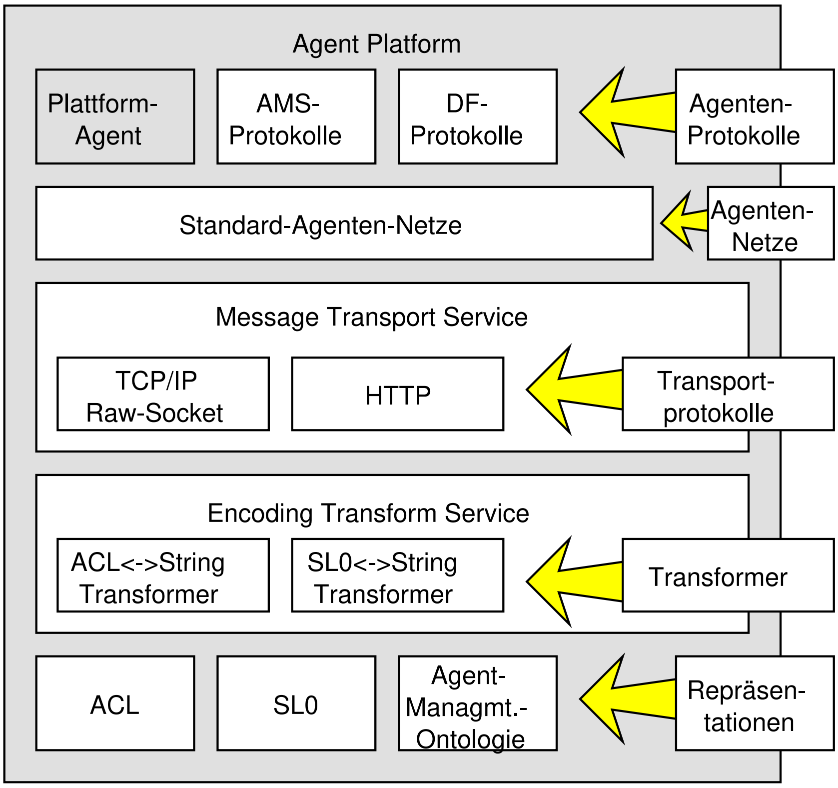


Abbildung 5.1: Elemente der CAPA-Plattform

inwieweit CAPA die auf der Plattform laufenden Agenten auf Referenznetze festlegt, oder ob auch andere Möglichkeiten der Implementierung bestehen, ist eng mit der Frage der internen Schnittstelle zwischen Agent und Plattform verknüpft und wird daher im folgenden Abschnitt 5.1.1 behandelt.

5.1.1 Interne Schnittstelle zum Agenten

Um eine für CAPA sinnvolle Schnittstelle zwischen Agent und Nachrichtentransportdienst zu ermitteln und zu definieren, sollte vorher ein Blick auf die bereits in verwandten Systemen vorhandenen Schnittstellen geworfen werden.

Im Mulan-Modell besteht die Schnittstelle zwischen Agent und Plattform aus zwei synchronen Kanälen, über die Nachrichten empfangen und versandt werden können. Die weiteren grundlegenden Dienste der Mulan-Plattform, konkret Agentenerzeugung, -terminierung und -ausführung, werden ohne zusätzliche Kanäle des Agenten realisiert. Der grundlegende „Dienst“ der Ausführung des Agentencodes erfolgt durch den Renew-Simulator, ohne Einfluss durch die Agenten oder das Plattformnetz. Im Mulan-Modell wird die Nutzung weiterer Dienste ausschließlich über den grundlegenden Kommunikationsdienst abgewickelt.

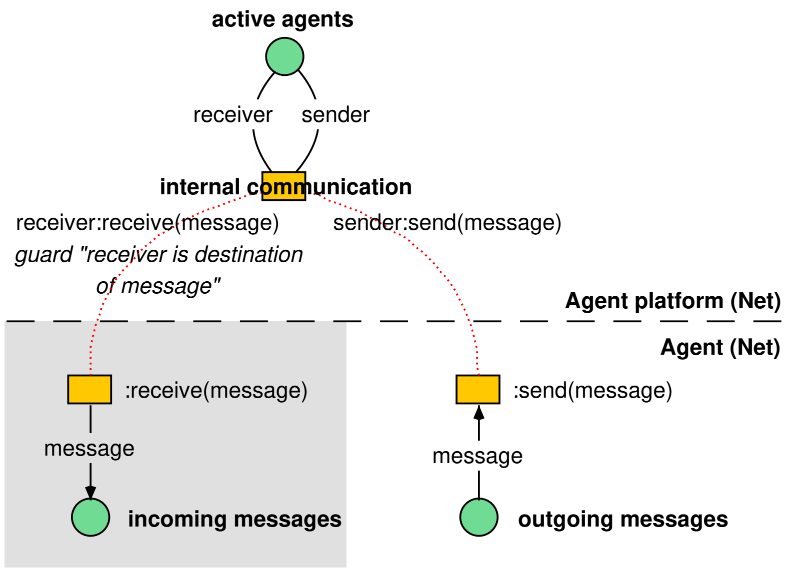


Abbildung 5.2: Plattforminterne Kommunikation im Mulan-Modell

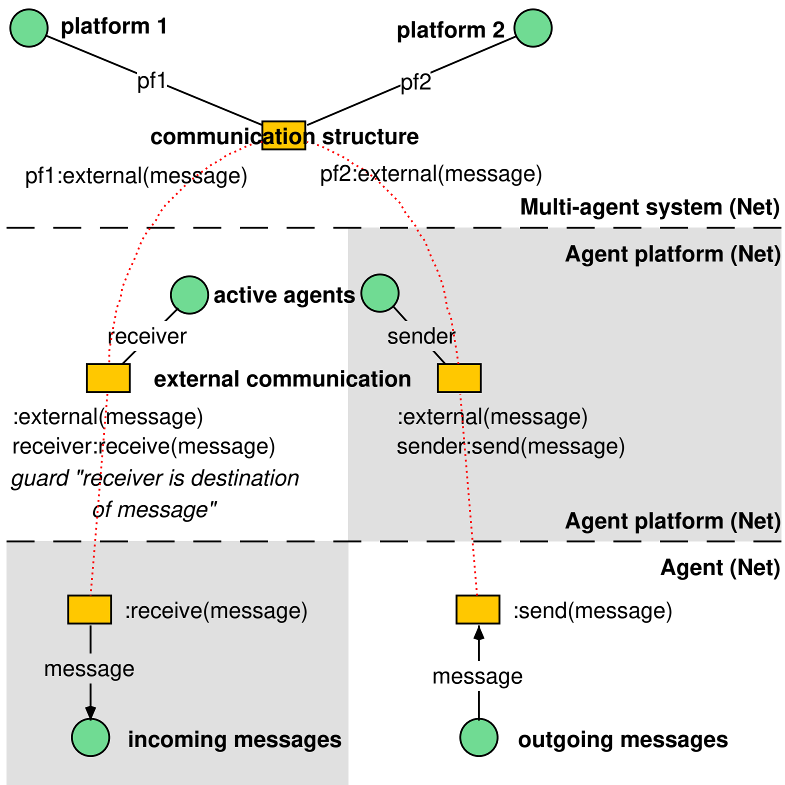


Abbildung 5.3: Plattformübergreifende Kommunikation im Mulan-Modell

```
public interface TransportService {
  public void send(AclMessage message);
}
```

Transport service (Java interface)

Agent (Java interface)

```
public interface Agent {
  public void receive(AclMessage message);
}
```

Abbildung 5.4: Java-Interface zum MTS

Der MULAN-Nachrichtentransport wird in den Abbildungen 5.2 und 5.3 veranschaulicht. Die rot gepunkteten Linien deuten die synchronen Kanäle an, die den Kommunikationsweg durch die Multiagentensystem-Architektur bilden. Die graue Hinterlegung soll darauf hinweisen, dass linke und rechte Hälfte einer Schicht von verschiedenen Netzinstanzen stammen können. Die orangefarbenen Transitionen stellen die Endpunkte der synchronen Kanäle dar (die Farbgebung dient nur der Verdeutlichung und hat keinen Einfluss auf das Schaltverhalten der Transition).

Die MULAN-Plattform unterscheidet zwischen interner und externer Kommunikation. Diese Unterscheidung ist an der Schnittstelle zum Agenten nicht spürbar, verdeutlicht Dank ihrer grafischen Darstellung aber den prinzipiellen Mehraufwand, der bei plattformübergreifender Kommunikation anfällt.

In vielen threadbasiert implementierten Agentenplattformen besteht die Schnittstelle zwischen Agent und Transportdienst aus zunächst zwei Methoden, eine zum Versenden und eine zum Empfangen von Nachrichten. Als Java-Interface könnte das z.B. aussehen wie in Abbildung 5.4.

Die Versandmethode wird vom Agententhread am Transportdienstobjekt aufgerufen. Mitunter hat der Agent eine eigene Methode, die diesen Aufruf weiterreicht, damit die Referenz auf den Transportdienst nicht überall im Code explizit angegeben werden muss.

Die Empfangsmethode wird vom Transportdienstthread am Agentenobjekt aufgerufen. Häufig verbirgt sich hinter der Empfangsmethode eine Standardimplementierung, die die angekommene Nachricht in eine Warteschlange des Agenten einreiht. Aus der Warteschlange bedient sich der Agent dann mittels weiterer eigener Methoden, wenn er gerade Zeit dazu hat.

Das Konzept der beiden vorgestellten Schnittstellen ist also gleich: Zwei Kanäle bzw. Methoden – eine(r) zum Senden, eine(r) zum Empfangen – stellen die Verbindung her. Aber dennoch gibt es Unterschiede, besonders beim Einsatz in einem nebenläufigen System. Diese Unterschiede fallen dann auf, wenn es an die Implementierung der Objekte auf beiden Seiten der Schnittstelle geht.

Bei der durch Netze spezifizierten Schnittstelle ist die Implementierung quasi schon enthalten. Das verleitet dazu, Implementierungsunterschiede als konzeptuelle Schnitt-

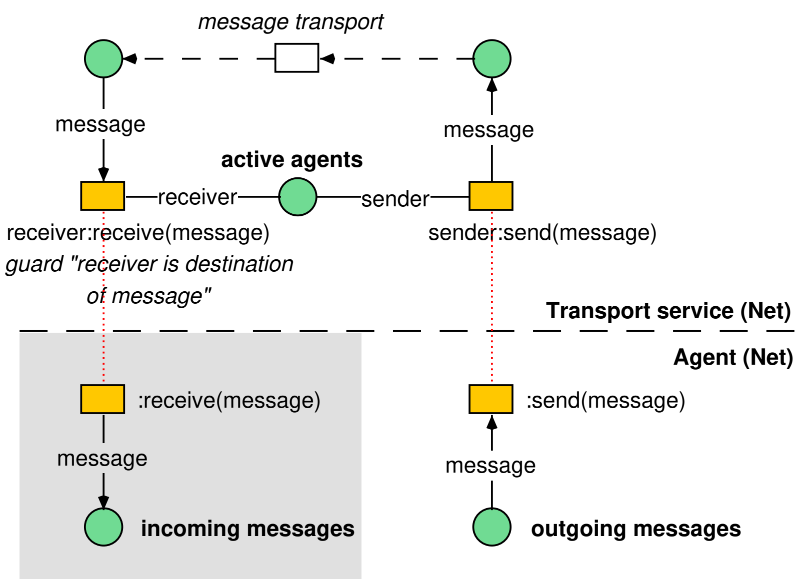


Abbildung 5.5: Asynchrone Transportdienstimplementierung

stellenunterschiede misszuverstehen. Mulan verwendet z.B. auf der Plattformseite *synchrone* Kanäle, d.h. der Nachrichtentransport von einem Agenten zum nächsten ist ein atomarer, quasi zeitloser Schritt im System. Der Autor von Mulan, Rölke, modelliert die asynchrone Nachrichtenübermittlung durch die Pufferstellen auf der Agentenseite.

Wie in Abbildung 5.5 zu sehen, kann der asynchrone Aspekt recht einfach auf die Transportdienstseite geholt werden – dieser Unterschied ist also eher ein Implementierungsproblem. Zumal die Definition der Schnittstellenkanäle ebenso wie die der Java-Interfaces aus Abbildung 5.4 keine Aussagen darüber macht, ob die Zustellung der Nachricht synchron erfolgt. Auch bei der threadbasierten Variante ist der synchrone ebenso wie der asynchrone Nachrichtentransport realisierbar. Der Unterschied wäre, ob der Transportdienst über einen eigenen Thread und eigene Nachrichtenpuffer verfügt – und genau diese zusätzlichen Elemente sind im Vergleich der beiden Netzimplementierungen aus Abbildung 5.2 und 5.5 gut zu erkennen.

Es gibt aber auch Implementierungsunterschiede, die tatsächlich in den Schnittstellendefinitionen begründet sind: Der Aufwand, Threads und Nachrichtenpuffer hinter das Java-Interface zu stellen, ist deutlich höher als der Aufwand, die entsprechenden Stellen und Transitionen in die Netze zu zeichnen. Umgekehrt erlaubt das Java-Interface eine Korrektheitsprüfung der jeweiligen Methodenaufrufe und -implementierungen zur Übersetzungszeit, inklusive Typprüfung für die übergebenen Nachrichtenobjekte. Die synchronen Kanäle der Referenznetze bieten solch eine Prüfung nicht. Wenn z.B. ein Kanalname falsch geschrieben wurde, gibt es keine Fehlermeldung, sondern die Marken bleiben in den Eingangsstellen der beteiligten Transitionen liegen. Re-

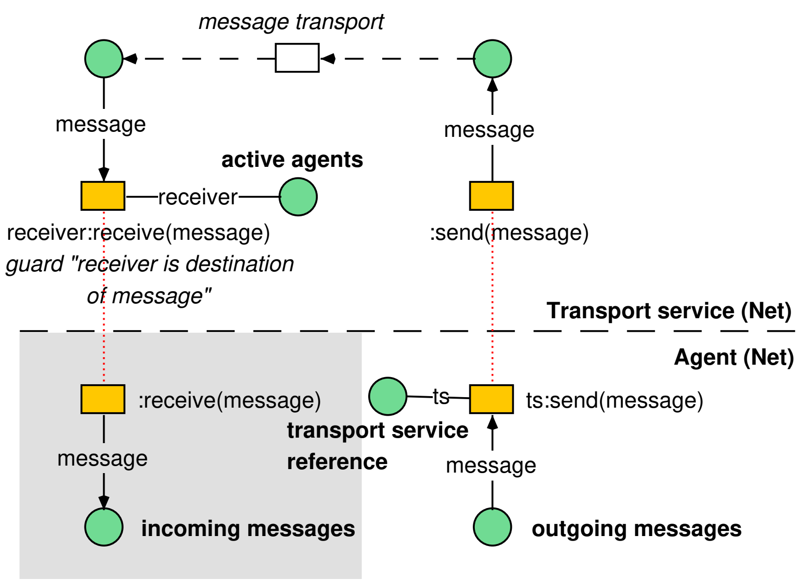


Abbildung 5.6: Referenzrichtungswechsel in den Netzen

new bietet zwar eine Kanalprüfung an, die vom Entwickler explizit auf den in der Entwicklungsumgebung geöffneten Netzen angestoßen werden kann, um unerfüllbare Downlinks zu finden – eine Art „Methodensignatur" für alle Kanalanschriften eines Netzes läßt sich damit aber nur indirekt erzwingen.

Um beide Vorteile – Typprüfung und einfache Implementierung – zu kombinieren, bietet sich die Verwendung von Renew-Stubs an, wie sie in Abschnitt 2.3 vorgestellt wurden: Das Java-Interface wird durch Netze implementiert. Jeder durch das Interface spezifizierte Methodenaufruf wird unkompliziert in einen synchronen Kanalaufruf umgewandelt.

Dabei tritt aber ein konzeptioneller Unterschied zwischen Methoden und Kanälen zu Tage: Synchrone Kanäle erlauben den unbeschränkten bidirektionalen Datenaustausch, dazu gehört auch die beidseitige Einigung über den Synchronisationszeitpunkt. Methodenaufrufe hingegen können lediglich zu Beginn Aufrufparameter entgegennehmen und am Ende einen Rückgabewert liefern, zudem bestimmt ausschließlich der Aufrufer den Aufrufzeitpunkt. Wenn also ein synchroner Kanal zum Anbieten einer Methode verwendet wird, muss auf einen Teil der Mächtigkeit des Kanals verzichtet werden.

Bei der threadbasierten Programmierung bestehen grundsätzlich drei Möglichkeiten, eine Nachricht von Objekt A zu Objekt B zu übermitteln: Die Nachricht kann *zugestellt* oder *abgeholt* werden. Bei der Abholung ist es wichtig festzulegen, ob eine Methode, die Nachrichten abholt, dies *nachfragend* oder *blockierend* tut. Der Unterschied wird interessant, wenn gerade keine Nachricht zur Abholung bereit liegt: Beim Nachfragen kehrt die Methode sofort wieder zurück – dann eben ohne Nachricht. Dafür ist der

Thread des Aufrufers wieder freigegeben, er kann sich anderen Aufgaben widmen und bei Gelegenheit erneut nachfragen – dieses Konzept ist auch als *aktives Warten* bekannt.

Beim Blockieren kehrt die Methode erst dann zurück, wenn eine Nachricht abgeholt werden kann. Der Thread des Aufrufers wird also solange angehalten, bis eine Nachricht vorliegt. Um eine blockierende Lösung implementieren zu können bietet Java das Monitor-Konzept mithilfe des `synchronized`-Schlüsselwortes an. Im Monitor wird ein Benachrichtigungsmechanismus implementiert, der die blockierten Aufrufe wieder aufweckt, wenn Nachrichten ankommen.

Bei der Nachrichtenzustellung wird von einem Thread der Nachrichtenquelle eine Benachrichtigungsmethode am Zielobjekt aufgerufen. Damit läuft im Zielobjekt ein weiterer Thread, von dem aus die Nachricht irgendwie an den Hauptthread des Zielobjekts übergeben werden muss (sofern es denn einen gibt). Alternativ kann der fremde Thread genutzt werden, um die Nachricht weiter zu verarbeiten. Damit wird ein reaktives Verhalten implementiert.

Die feine Unterscheidung, welcher Thread welche Methode aufruft und ob ein Thread zwischenzeitlich angehalten wird, ist bei synchronen Kanälen nicht notwendig. Der synchrone Kanal wird genau dann genutzt, wenn die beiden beteiligten Transitionen mit passenden Bindungen aktiviert sind, unabhängig davon, welche der Transitionen mit dem Uplink und welche mit dem Downlink beschriftet ist.

Soll ein Kanal als Implementierung einer Methode dienen, muss die Kanalanschrift ein Uplink sein, da auch die Methodensignatur keine Referenz auf den Aufrufer zulässt. Vergleicht man die Java-Interfaces aus Abbildung 5.4 mit den Kanälen aus Abbildung 5.5, so fällt auf, dass beim Senden der Uplink in den Netzen am Agenten angebracht ist, bei den Methoden jedoch am Transportdienst. Damit geht die Abweichung einher, dass im Java-Interface die Idee der Nachrichtenzustellung verfolgt wird, während die Netze eher eine Abholung (auch wenn der Begriff für den Kanal nicht ganz passt) modellieren.

Somit eignen sich die bisher gezeigten Netze nicht als Implementierung der bisher gezeigten Interfaces. Entweder müssen die Netze angepasst werden oder die Interfaces.

In Abbildung 5.6 ist die mögliche Änderung der Netze aus Abbildung 5.5 zu sehen. Beim Versandkanal haben Up- und Downlink die Plätze getauscht. Am Verhalten des synchronen Kanals ändert das nichts, aber die notwendige Referenz vom Agenten zum Transportdienst schafft Probleme. Zum einen muss diese Referenz jedem Agenten mitgegeben werden – ein Aufwand, der bei der ursprünglichen Lösung nicht nötig war. Zum anderen wäre der Missbrauch der Referenz durch den Agenten möglich, wenn der Agent sich über andere Kanäle als vorgesehen mit dem Transportdienst synchronisiert. Angenommen, der Transportdienst böte Kanäle an, die für die plattforminterne Weiterleitung von Nachrichten vorgesehen sind: dann gäbe es keinen Schutz davor, dass auch der Agent diese Kanäle nutzt. Letztendlich gründen auch die von den Mulan-Entwicklern angedachten Sicherheitskonzepte für auf der Idee, dass der Agent keine Referenz auf Plattformkomponenten benötigt. Dann kann er ganz einfach in ein „Sandkasten“-Netz gesteckt werden, welches alle seine Kanalkommunikationen auf Zulässigkeit überprüft, ohne dass das Agentennetz den Unterschied bemerkt.

```
public interface TransportService {
}
```

Transport service (Java interface)

Agent (Java interface)

```
public interface Agent {
  public void receive(AclMessage message);
  public AclMessage send();
  // send: blocking or polling?
}
```

Abbildung 5.7: Referenzrichtungswechsel im Java-Interface

Der andere Lösungsansatz für das Schnittstellenproblem ist die Anpassung des Java-Interfaces wie in Abbildung 5.7. Das dabei auftretende Problem ist der Wechsel von einer *zustellenden* zu einer *abholenden* `send`-Methode. Damit muss die Frage beantwortet werden, ob die Methode nachfragender oder blockierender Natur sein soll.

Eine Netz-Stub-Methode, die einen Rückgabewert liefern soll, blockiert immer solange, bis die Synchronisierung über den Kanal des Netzes erfolgreich war. Allerdings kann das Netz so umgestaltet werden, dass die den Kanal anbietenden Transitionen auch ohne verfügbare Nachricht feuern können – nachfragendes Verhalten ist also ebenso möglich wie blockierendes.

Der interessante Punkt bei der Frage, ob `send()` abfragend oder blockierend sein soll, ist eher, wie das Verhalten des aufrufenden Threads oder Netzes aussehen soll. Aktives Warten ist im allgemeinen eine unerwünschte Verhaltensweise, da sie – falls außer Warten nichts zu tun ist – viel Arbeitszeit für kein Ergebnis verbraucht. Im Netz sieht das so aus, dass, falls keine Vor- oder Nebenbedingungen das Feuern der aufrufenden Transition einschränken, diese Transition immer wieder schaltet – gegebenenfalls sogar nebenläufig zu sich selbst – ohne sinnvolles Ergebnis.

Riefe dieselbe Transition eine blockierende Methode auf, so begännen eine Unzahl von nebenläufigen Schaltvorgängen, aber nur ein Schaltvorgang ginge je abgeholter Nachricht zu Ende. Dieses Verhalten lässt sich nur durch Sequentialisierung der Transition vermeiden, aber dann verliert man die eigentlich gewünschte nebenläufige Behandlung beliebig vieler Nachrichten.

Beide Lösungen für die Abholung von Nachrichten sind also in ihrem nebenläufigen Verhalten nicht zufrieden stellend. Ebenso hat die Alternative, die zu versendenden Nachrichten dem Transportdienst zuzustellen, den unangenehmen Seiteneffekt, jedem Agenten eine Referenz auf den Transportdienst in die Hand geben zu müssen. Letztendlich bringen alle vorgeschlagenen Varianten, ein Java-Interface durch Netze zu implementieren, Nachteile mit sich, die aus meiner Sicht schwerer wiegen als der durch das Java-Interface gewonnene Vorteil der statischen Typprüfung.

Daher dient die alte MULAN-Schnittstelle zwischen Plattform- und Agentennetz in der neuen Plattformarchitektur unverändert als Schnittstelle zwischen MTS und Agent. Eine abgeschwächte Typprüfung lässt sich realisieren, indem nicht die Netze selber den Agenten verkörpern, sondern ein Java-Objekt, welches auf das Agentennetz verweist. Die dafür definierte Agentenklasse erlaubt der Plattform zumindest die Typprüfung, ob ein Agentenobjekt oder ein völlig anderes Objekt vorliegt. Hinzu kommt die Möglichkeit, für die Plattform interessante Attribute wie z.B. den eindeutigen Namen des Agenten in dieser Klasse unterzubringen. Die in einer Java-Klasse gegebene Dokumentationsmöglichkeiten können genutzt werden, um die Kanalschnittstelle des Agentennetzes zumindest textuell festzuschreiben.

Das zweite angestrebte Ziel bei der Idee, auf ein Java-Interface umzuschwenken, war die Anbindbarkeit von Agenten, die nicht in Netzen implementiert sind. Dies lässt sich aber auch anders erreichen. Ein Adapterobjekt (in Abbildung 5.8 dargestellt) schlägt die Brücke zwischen der Programmiersprache Java und den Referenznetzen. Der Adapter besteht selber zur Hälfte aus Netz- und aus Java-Code.[3]

Da die innere Schnittstelle zwischen Plattform und Agenten ohnehin proprietär ist, kann die Frage gestellt werden, inwieweit mehr Dienste als der asynchrone FIPA-konforme Nachrichtentransport über diese Schnittstelle angeboten werden können oder sollen. Eine Idee hierzu wäre zum Beispiel lokale, synchrone Kommunikation zwischen zwei Agenten auf einer Plattform. Diese Ideen habe ich aber hintenangestellt, um zunächst die grundlegende Funktionalität bereitzustellen.

5.1.2 Schnittstelle nach außen

Die äußere Schnittstelle einer Agentenplattform besteht aus mehreren Elementen. Zum einen müssen auf der Netzwerk-Ebene des Nachrichtentransports exakt definierte Übergabemöglichkeiten für ein- und ausgehende Nachrichten bestehen, d.h. ein Netzwerkprotokoll nebst Adressierungsschema muss festgelegt werden. Diesem Zweck dienen die in Abschnitt 4.4.1 vorgestellten FIPA-Transportprotokollspezifikationen.

Zum anderen muss auf einer höheren Ebene ein definiertes Verhalten bezüglich der Weiterleitung und Zustellung von Nachrichten festgelegt werden. Die „FIPA Message Transport Service Specification“ [FIPA00067] schreibt die Existenz eines „Agent Communication Channel“ (ACC) vor, der auf eine bestimmte Weise mit Nachrichten zu verfahren hat. Der ACC stützt sich für seine Arbeit ausschließlich auf die Informationen in einem sogenannten Nachrichtenumschlag.

[3] An diesem Beispiel kann die Behauptung, die Implementierung von Puffern und Weiterleitungsthreads sei in Java komplizierter als mit Netzen, gut nachvollzogen werden. Der Java-Code implementiert das spiegelbildliche Verhalten zum Netz-Code mit einem Puffer für empfangene Nachrichten. Der Code fällt umfangreicher aus als die Netzgrafik – und das sogar, obwohl die aufwändige Implementierung des Puffers mit den nötigen Synchronisierungsmechanismen in die Klasse `Queue` ausgelagert wurde.

Die roten Pfeile auf der rechten Seite veranschaulichen die Aufrufkette beim Versand einer Nachricht. Die Pfeile auf der linken Seite veranschaulichen den Empfang einer Nachricht, wobei die Nachricht im Puffer `inMessages` zwischengespeichert und vom Agententhread abgeholt wird. Sowohl die blockierende als auch die nachfragende Abholung stehen zur Auswahl.

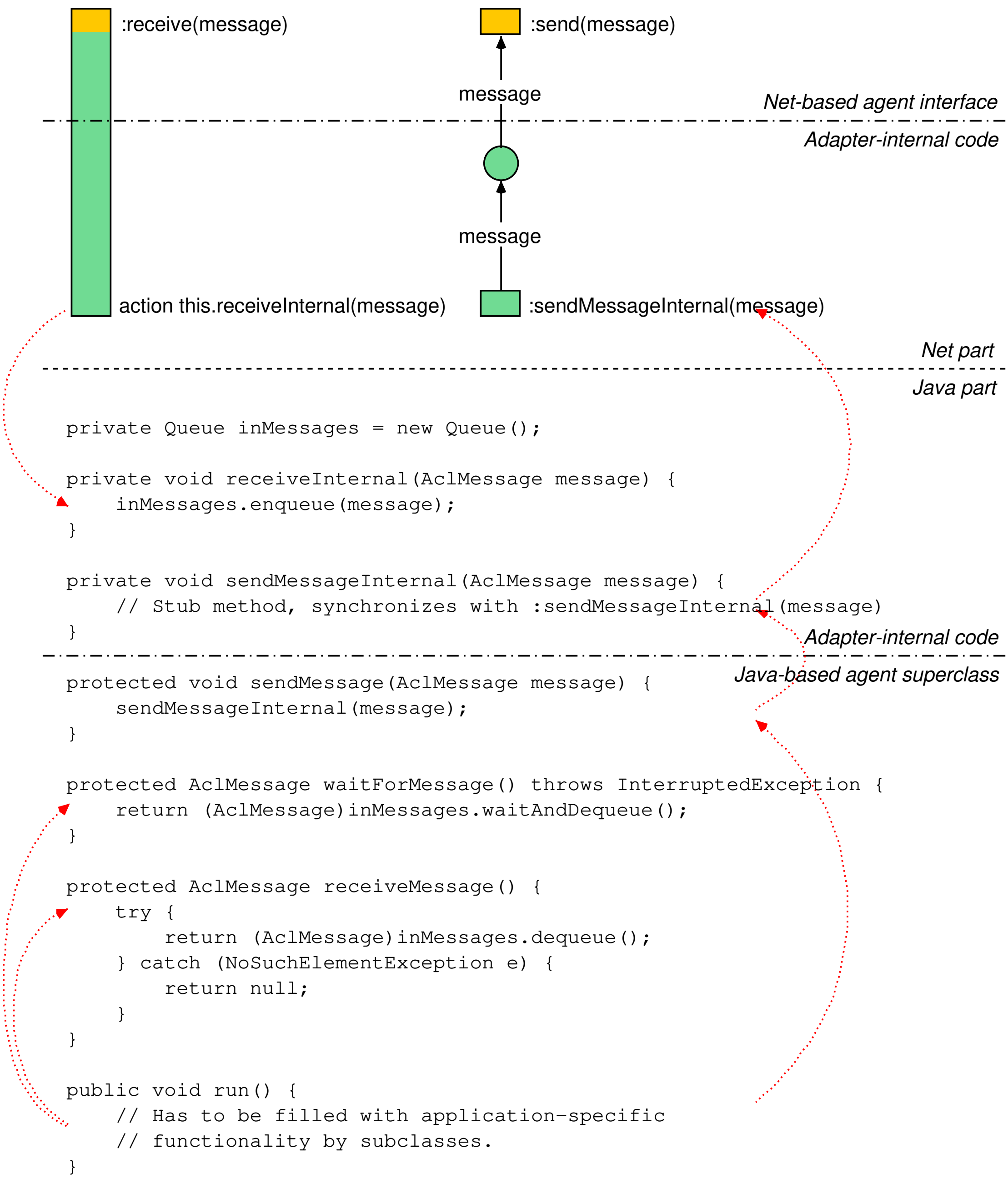


Abbildung 5.8: Der `JavaAgentAdapter` verbindet Referenznetz- mit Java-Code

Darüber hinaus muss es auf der Ebene der Agentenkommunikation feste Ansprechpartner unter bekannten Adressen geben, bei denen Auskünfte über die Namen, Adressen und Dienste der Agenten auf der Plattform eingeholt werden können. Die „FIPA Agent Management Specification“ [FIPA00023] sieht hierfür die unter den festen Namen AMS und DF ansprechbaren Verwaltungsagenten vor (siehe Abschnitt 4.4.2).

Die drei Ebenen der äußeren Schnittstelle, die eben aus Sicht der FIPA2000-Standards heraus beschrieben wurden, werden auch von der abstrakten FIPA-Architektur abgedeckt. Leider sind die Regelungen der abstrakten Architektur nur ähnlich, aber nicht identisch zu denen der FIPA2000-Suite (siehe Kapitel 4). So kennt die abstrakte Architektur keinen zentralen ACC, sondern bindet jeden Agenten individuell an ein oder mehrere Netzwerktransportprotokolle, von denen verschiedene zur Verfügung stehen können. Damit bietet die abstrakte Architektur den Agenten mehr Flexibilität, ist aber nicht kompatibel zu den FIPA2000-Standards.

CAPA soll die Flexibilität der abstrakten Architektur bieten und dennoch mit FIPA2000 kompatibel bleiben. Damit die Agenten nicht gezwungen sind, sich um die Auswahl der Transportprotokolle zu kümmern (was sie ja gemäß FIPA2000 nicht brauchen), muss diese Auswahl für die Agenten transparent geschehen. Meine Lösung sieht vor, dass der ACC, der ohnehin schon die Weiterleitung der Nachrichten übernimmt, auch ein passendes Transportprotokoll aus dem Fundus der verfügbaren auswählen kann.

Nachrichten werden mittels einer in [FIPA00023] definierten `agent-identifier`-Datenstruktur adressiert, die darin enthaltenen Informationen decken die Adressierung über alle drei Ebenen vom Netzwerk über den Transport bis zur Agentenkommunikation ab. Insbesondere ist das `addresses`-Feld der Struktur durch eine URL (Uniform Resource Locator) auszufüllen. Die erste Komponente der URL bezeichnet das zu verwendende Protokoll und kann daher vom ACC genutzt werden, das richtige Transportprotokoll auszuwählen. Damit haben die Agenten die Möglichkeit, über das `addresses`-Feld die Wahl des Transportprotokolls zu beeinflussen. Auf der anderen Seite ist der Mehraufwand über die ohnehin notwendige Adressbeschaffung hinaus optional, d.h. Agenten brauchen die Wahl nicht zu treffen, wenn sie es nicht wollen.

Damit ergibt sich für die für den Außenkontakt zuständigen Elemente der CAPA-Plattformarchitektur folgendes Bild: Ein zentraler `TransportService` nimmt die Aufgaben des ACC wahr. Er kann auf eine Auswahl von Transportprotokollen (in Anlehnung an die Terminologie der abstrakten Architektur `Transport` genannt) zurückgreifen, um die Nachrichten entsprechend ihrer Adressierung weiterzuleiten. Mit jedem Transportprotokoll wird die Art der Netzwerkkommunikation und die Adresse, unter der die Plattform mit diesem Protokoll erreichbar ist, festgelegt.

Die Menge der zur Verfügung stehenden Transportprotokolle darf von Plattform zu Plattform unterschiedlich sein. Die Auswahl soll sich sogar zur Laufzeit ändern können, wenn ein `Transport` beim zentralen `TransportService` an- oder abgemeldet wird. Kommunikation zwischen zwei Plattformen ist aber nur möglich, wenn beide ein gemeinsames Transportprotokoll kennen.

Bei den Verzeichnisdiensten AMS und DF können Adressen von Agenten in Erfahrung gebracht werden. Dabei können die Agenten (entsprechend der Definition der

Verzeichnisdienste in den FIPA-Spezifikationen) selber bestimmen, welche Auswahl ihrer Adressen sie in den Verzeichnissen veröffentlichen.

Der zentrale Transportdienst dient somit als Durchgangsstation für alle Nachrichten von und nach außerhalb. Darin eingeschlossen sind auch solche Nachrichten, die gar nicht für lokale Agenten bestimmt sind, sondern einfach nur weitergeleitet werden sollen. Wenn im Transportdienst ohnehin alle Nachrichten auf der einen Seite hineingehen und am anderen Ende wieder herauskommen, egal ob von außen nach innen, von innen nach außen oder von außen nach außen, dann können eigentlich auch die Nachrichten von innen nach innen durch diese Station laufen. Daher entstand die Idee, die im vorigen Abschnitt 5.1.1 entwickelte interne Schnittstelle zum Transportdienst gleichberechtigt mit allen externen Transportprotokollen in den Transportdienst einzuklinken. Auf diese Weise können die lokalen Agenten alle auf der Plattform lebenden Agenten und angebotenen Dienste auf die gleiche Art erreichen wie externe Agenten.

5.1.3 Nachrichten

Agenten in einer FIPA-Kommunikationsarchitektur verwenden Nachrichten, um sich gegenseitig zu informieren, zu koordinieren, zu beeinflussen usw. Diese Nachrichten müssen zwecks Verständlichkeit in einer allen Beteiligten bekannten Sprache verfasst sein. Zu diesem Zweck hat die FIPA die Agent Communication Language (ACL) definiert (siehe Abschnitt 4.2), welche die Semantik, Syntax und Kodierung von Nachrichten festlegt.

Da die Agentenplattform über spezielle Schnittstellen mit den Agenten verbunden ist, ist die ACL für die Plattform eigentlich uninteressant. Das Verständigungsproblem besteht nur zwischen Agenten. Wie Agenten Nachrichten komponieren und interpretieren, ist daher alleine ihre Sache. Wenn die Plattform Nachrichten transportiert, muss sie nur deren Umschlag interpretieren.

Dennoch gibt es Gründe, auf einer Agentenplattform vorgefertigte Nachrichtenstrukturen sowie Kompositions- und Interpretationstechniken zur Verfügung zu stellen: Ein wesentlicher Punkt ist, dass ein Großteil des Aufwands nur einmal zentral erledigt werden muss, statt dass er für jeden Agenten erneut anfällt. Die Aufwandsreduktion erlaubt es, mehr Energie in die Strukturen zu stecken und dort zusätzliche Funktionalität bereitzustellen.

Darüber hinaus reduzieren vorhandene Strukturen für häufig benötigte Nachrichtenelemente, die nur noch wie Formulare ausgefüllt zu werden brauchen, das Fehlerrisiko. Alle vorgefertigten Elemente sind garantiert korrekt, zudem kann – als zusätzliche Funktionalität – eine Plausibilitätsprüfung auf die auszufüllenden Elemente angewendet werden. Das Ausfüllen von Formularen verlagert die Fehlererkennung vom Empfänger der Nachricht zum Absender, wenn falsche oder unplausible Einträge schon beim Ausfüllen gemeldet werden.

Ein möglicher Nachteil vorgefertigter Strukturen ist die eingeschränkte Flexibilität gegenüber formlos verfassten Nachrichten. Daher gilt es, die Strukturen so flexibel wie möglich zu halten. Gegebenenfalls muss auch auf den Einsatz der Strukturen ganz und gar verzichtet werden dürfen.

Die Notwendigkeit beider Forderungen, Flexibilität und Verzichtmöglichkeit, wird klar, wenn man einen Blick auf die von der FIPA spezifizierten Nachrichtenformate wirft: Alle Formate kommen entweder als lesbare Zeichenketten oder als Byte-Sequenz daher und müssen verschickt werden können. Eine Plattform, die ausschließlich ihr eigenes Nachrichtenformat mit vorgefertigten Strukturen versenden kann, ist nicht FIPA-konform.

Für die innere Schnittstelle zu den Agenten mag eine Festlegung auf die Verwendung des Nachrichtenformats zwar erlaubt sein, wenn die Plattform für die Kommunikation nach außen eine Übersetzung in die offiziellen Formate übernimmt. So eine Festlegung würde aber dennoch eine erhebliche Einschränkung der Kommunikationsmöglichkeiten der Agenten darstellen.

Wenn die Verwendung der vorgefertigten Strukturen optional sein soll, muss auch die Umwandlungsmöglichkeit optional sein. Zudem besteht eventuell Bedarf, Nachrichten zwischen anderen Nachrichtenformaten transformieren zu können, ohne dass das plattforminterne Format beteiligt ist. Die abstrakte Architektur sieht dazu die Möglichkeit eines „Encoding Transform Service" vor (siehe Abschnitt 4.3.2). Dieser Dienst kann sowohl die Transportkodierung, Verschlüsselung oder Verpackung einer Nachricht ändern als auch zwischen den verschiedenen Nachrichtenrepräsentierungen (ob als Zeichenkette, XML-Dokument oder Byte-Sequenz) umwandeln.

Die Schnittstelle des Transformationsdienstes wird in der abstrakten Architektur nur in ihrer Funktionalität definiert. Ob der Dienst als Agent oder als Plattformkomponente ansprechbar sein soll, ist offen. Für das Agenten-Interface spricht, dass andere Agenten auf diese Weise am unkompliziertesten auf den Dienst zurückgreifen können – sie nutzen einfach die schon vorhandenen Kommunikationswege.

Andererseits wäre es durchaus praktisch, wenn der Nachrichtentransportdienst automatisiert eine Umwandlung des Nachrichtenformats vornehmen kann, um z.B. die Nachrichtenrepräsentierung an die Anforderungen eines bestimmten Transportprotokolls anzupassen. In diesem Fall darf die Kommunikation zwischen Transportdienst und Umwandlungsdienst nicht auf der Agentenebene stattfinden, weil sonst der Transportdienst bei jeder Nachrichtenweiterleitung rekursiv seinen eigenen Dienst in Anspruch nehmen müsste. Stattdessen sollte eine Schnittstelle auf der Plattform-Implementierungsebene vorhanden sein. Beide Schnittstellen können nebeneinander existieren, wenn z.B. die Agentenschnittstelle auf die Schnittstelle der niedrigeren Ebene zurückgreift.

CAPA soll also einen Repräsentierungsdienst bieten. Dieser „Dienst" besteht darin, dass entsprechende Strukturen existieren. Hinzu kommt ein Transformationsdienst, der Nachrichten oder Teile davon von einer Repräsentierung in eine andere umwandeln kann. Dieser Dienst wird auf der Plattform-Implementierungsebene bereitgestellt, weil er vom Transportdienst aus verwendbar sein soll. Die Bereitstellung auf der Agentenebene ist möglich, aber zunächst nicht nötig, wenn die Agenten auch auf die Schnittstelle auf Plattformebene direkt zugreifen dürfen. Dank des Transformationsdienstes kann die interne Repräsentierung in eine Darstellung gemäß den FIPA-Spezifikationen umgewandelt werden, ebenso wie andere Umwandlungen denkbar sind. Die Architektur soll flexibel sein, so dass alternative Repräsentierungen und damit verbundene Transformationen einfach eingeklinkt werden können.

5.2 Implementierung

Im vorigen Abschnitt 5.1 wurde die Plattformarchitektur von CAPA vorgestellt, die aus mehreren Elementen besteht. Dazu gehören das Agentenmodell von MULAN und die für eine FIPA-konforme Plattform notwendigen Dienste wie die Verwaltungsdienste AMS und DF und der Transportdienst MTS mit der Weiterleitungskomponente ACC. Es wurde begründet, warum eine interne Repräsentierung der ACL-Nachrichten durch vorgefertigte Strukturen Sinn macht, und dass ein Transformationsdienst zur Umwandlung von Nachrichtenrepräsentierungen benötigt wird.

Alle diese Elemente müssen in der lauffähigen Plattform CAPA implementiert werden. Dabei treten durchaus noch Fragen und Probleme auf, die beim Entwurf der Plattformarchitektur im Hintergrund geblieben sind. Diese Punkte sollen in den folgenden Abschnitten, gegliedert nach den Plattformelementen, aufgezeigt werden.

Die Implementierung selber ist in mehrere Java-Packages gegliedert.[4] Die Package-Struktur entspricht im Groben der Unterteilung in verschiedene Plattformelemente, wie sie in Abbildung 5.1 dargestellt ist.

Wie in Abbildung 5.9 zu sehen ist, sind alle Packages sind unter dem Präfix `de.renew.agent` in die Package-Hierarchie des Renew-Werkzeugs einsortiert. Diese Unterordnung ist darin begründet, dass die Implementierung der Agentenplattform Referenznetze verwendet und daher ohne Renew nicht laufen kann. CAPA ist allerdings – abgesehen von der Verwendung von Referenznetzen – von Renew unabhängig. Diese Unabhängigkeit könnte sich auch in einem Renew-unabhängigen Präfix für die Package-Struktur niederschlagen.

Eine Abhängigkeit in anderer Richtung, also dass Renew auf Teile der Agentenplattform angewiesen ist, ist nicht vorhanden und auf keinen Fall beabsichtigt. Ein spezieller Renew-Modus, der die Entwicklung und den Betrieb der Plattform durch eine Ergänzung der grafischen Oberfläche unterstützt, ist zwar angedacht, aber zunächst nicht realisiert. Auch der spezielle Modus soll auf keinen Fall zu einer notwendigen Komponente des Referenznetzsimulators werden.

Das Package `de.renew.agent.standard` enthält die Netze und Hilfsklassen, die die Grundimplementierung eines Standard-Agenten entsprechend der MULAN-Entscheidungsarchitektur bilden. Der Agent wird in Abschnitt 5.2.1 beschrieben.

Alle Aspekte der Plattformverwaltung sowohl auf der technischen als auch auf der Agentenkommunikationsebene sind im Package `de.renew.agent.platform` gelöst und werden im Abschnitt 5.2.2 behandelt. Im Package eingeschlossen ist die Definition der

[4] Obwohl Renew die Einordnung von Netzen in Packages noch nicht unterstützt, sind auch die Netze den Java-Packages zugeordnet. Die Zuordnung ist nicht nur logisch-inhaltlicher Art, sondern schlägt auch auf den Speicherort der Netz-Datei durch. Bei der Arbeit mit Renew müssen auf diese Art mehrere Verzeichnisse nach Netzen durchsucht werden, aber der Vorteil der klaren thematischen Zuordnung wiegt meiner Meinung nach schwerer.

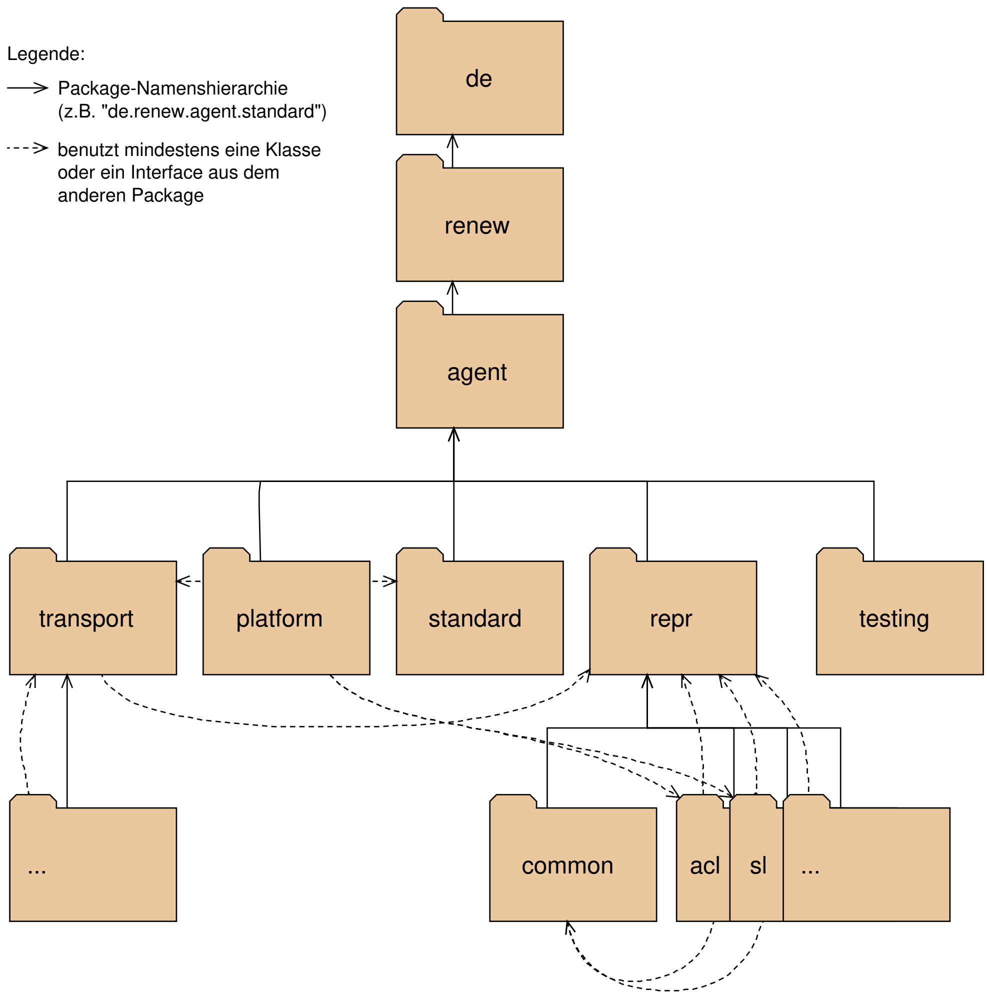


Abbildung 5.9: Die Java-Package-Struktur von CAPA

internen Schnittstelle zwischen Agent und Nachrichtentransportdienst, die hier aber wegen ihrer engen Anbindung zum Transportdienst erst in Abschnitt 5.2.3.2 erläutert werden soll.

Das Package `de.renew.agent.transport` nimmt die Schnittstellendefinition der Transportdienstelemente auf und enthält eine Implementierung des zentralen Transportdienstes. Implementierungen von konkreten Transportprotokollen sollen in eigene Packages unter dem `transport`-Package einsortiert werden. Der Abschnitt 5.2.3 wird sich mit den drei Elementen des Transportdienstes – zentrale Weiterleitung, interne Schnittstelle, externes Transportprotokoll – beschäftigen.

Das Package `de.renew.agent.repr` gibt die Schnittstelle zum Transformationsdienst zur Umwandlung von Nachrichten von einer Repräsentierung oder Kodierung in eine andere vor. Die unter dem `repr`-Package angeordneten Packages `common`, `acl` und `sl` stellen Strukturen zur internen Repräsentierung von Nachrichten bereit. Gleichzeitig enthalten sie Transformationsmodule zur Umwandlung der internen Repräsentierung in eine offizielle FIPA-Repräsentierung. Die Repräsentierungsstrukturen sind das Thema des Abschnitts 5.2.4, der Transformationsdienst wird in Abschnitt 5.2.5 erläutert.

5.2.1 Standard-Agent

Der Standard-Agent ist eine Referenzimplementierung eines Agenten, die mit kleineren Änderungen aus dem Mulan-Agentennetz hervorgegangen ist. Er ist damit im wesentlichen eine Arbeit von Heiko Rölke [Röl2002]. Der Namenszusatz „Standard“ weist auf zwei Eigenschaften dieser Agentenimplementierung hin: Zum einen stellt der Standard-Agent eine gebrauchsfähige, für allgemeine Zwecke einsetzbare Implementierung eines Agenten dar. Der Agent wird – entsprechend der im Mulan-Modell vorgeschlagenen Entscheidungsarchitektur – durch die Spezifikation von Protokollen um das anwendungsspezifische Verhalten ergänzt. Damit ist der Standard-Agent eine wiederverwendbare Grundlage für viele Agenten mit verschiedensten Verhaltensweisen.

Zum anderen kann das „Standardmodell“ auch durch andere Modelle ersetzt werden. Dies ist insbesondere dann erforderlich, wenn eine andere Entscheidungsarchitektur als die vom Standard-Agenten vorgesehene protokollgesteuerte Mulan-Architektur realisiert werden soll. Falls Agenten in anderen Programmiersprachen als den Referenznetzen implementiert werden sollen, ist ein Austausch des Standard-Agenten ebenfalls erforderlich. Ein Beispiel für einen Austausch des Agentennetzes stellt der am Ende von Abschnitt 5.1.1 vorgestellte `JavaAgentAdapter` dar.

Das Hauptnetz des Agenten ist in Abbildung 5.10 zu sehen. Eine vereinfachte Version ist in Abbildung 3.3 (S. 32) dargestellt und im zugehörigen Abschnitt 3.2.2 bereits beschrieben worden. Hier soll es also nur noch um die Ergänzungen des vollständigen Agentennetzes gehen.

Im oberen und rechten Teil des Netzes sind einige neue Elemente hinzugekommen. Oben, mittig über der bekannten Agentenstruktur, sitzen zwei Transitionen, die den Agenten initialisieren. Die beiden Hilfsnetze des Agenten, Wissensbasis und Proto-

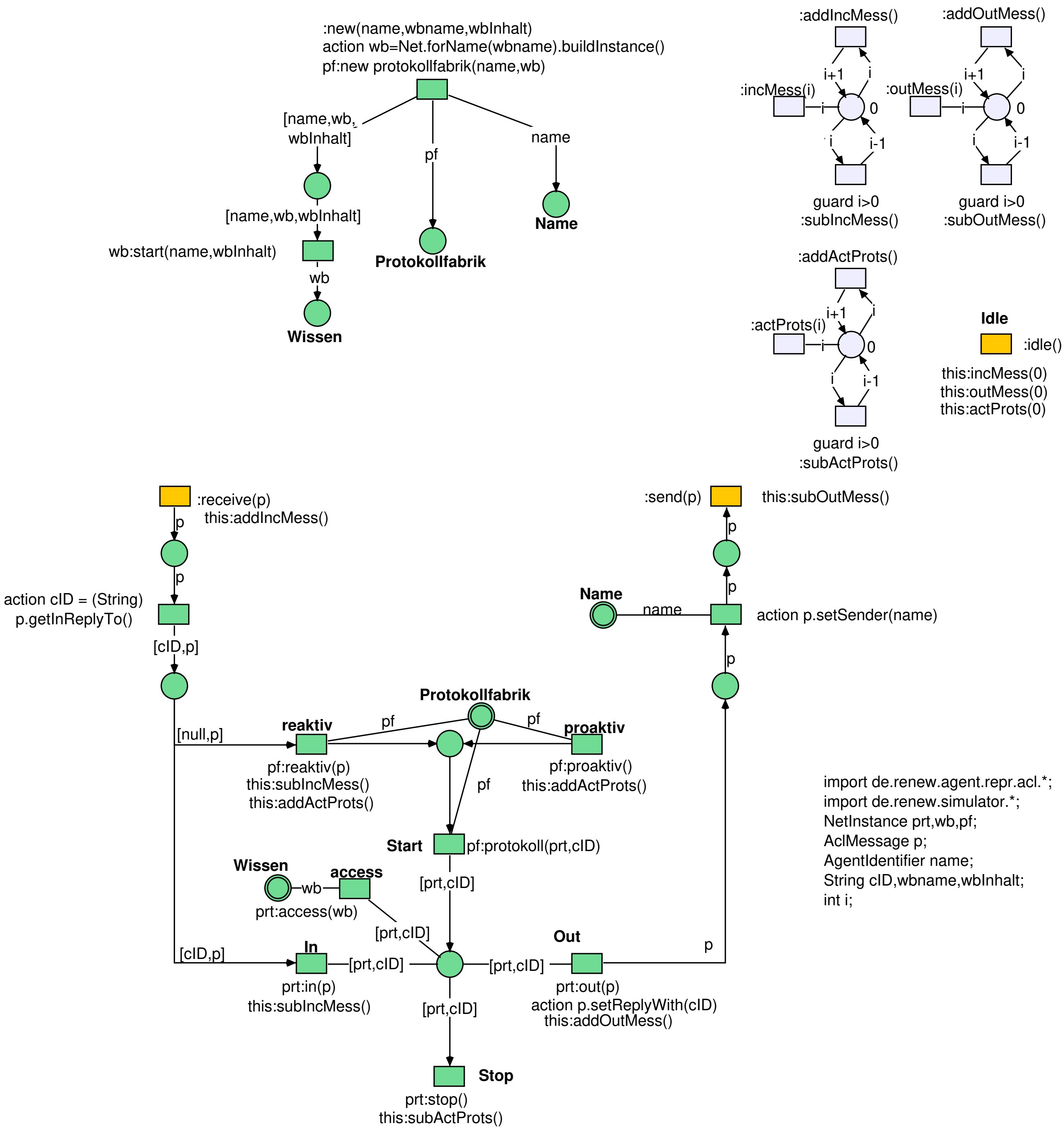


Abbildung 5.10: Das Hauptnetz des Standard-Agenten

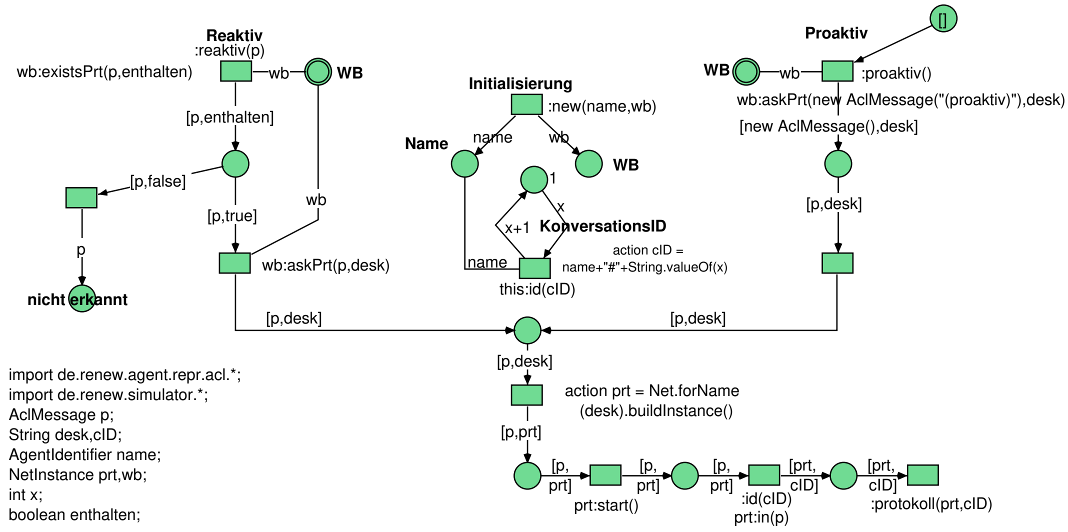


Abbildung 5.11: Die Protokollfabrik des Standard-Agenten

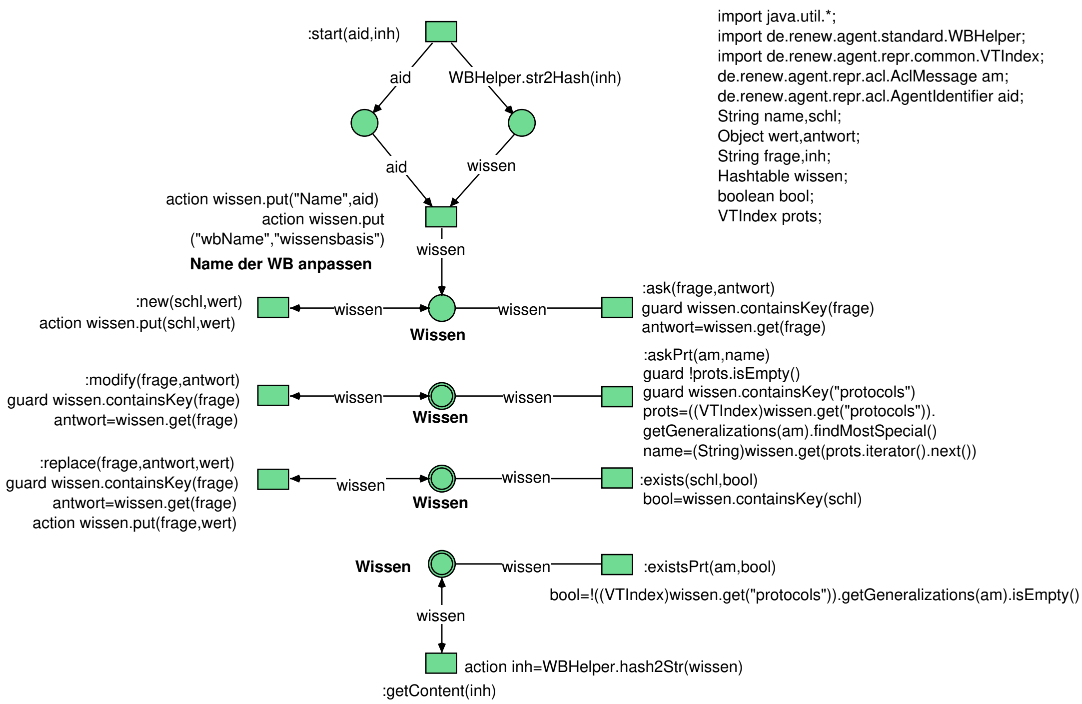


Abbildung 5.12: Eine einfache Wissensbasis für den Standard-Agenten

kollfabrik, müssen erzeugt und mit ihrem anfänglichen Inhalt gefüllt werden. Dabei werden die Wissensbasis und ihr Inhalt als Parameter bei der Erzeugung des Agenten übergeben, es steht also ein einfacher Mechanismus zur Verfügung, um dieses Element an die Bedürfnisse des Agentenentwicklers und des Anwendungsgebiets anzupassen.

Rechts finden sich drei Stellen nebst damit verbundener Transitionen (alle blassblau zurückgesetzt), die den Aktivitätszustand des Agenten protokollieren. Hier wird gezählt, an wie vielen laufenden Konversationen sich der Agent gerade beteiligt, und wie viele ein- und ausgehende Nachrichten sich im Agentennetz befinden, ohne bearbeitet bzw. versandt worden zu sein. Ziel dieser Zählerei ist es, feststellen zu können, wann der Agent inaktiv, also *idle* ist. Diese Information wird über den zur Plattformschnittstelle hinzugekommenen `:idle()`-Kanal abgefragt. Warum diese Information interessant ist, wird im folgenden Abschnitt 5.2.2 erläutert.

An der eigentlichen Agentenstruktur hat sich nichts Wesentliches verändert. Das Netz merkt sich den Namen des Agenten, um ihn automatisch als Absender in alle ausgehenden Nachrichten einzutragen. Ein weiterer Automatismus ist die Verwaltung von Konversations-Identifikatoren, die in den Variablen `cId` transportiert werden. Jeder Konversation wird bei der Instanziierung des Protokolls eine eindeutige Nummer zugewiesen, die allen ausgehenden Nachrichten im `reply-with`-Feld der ACL-Struktur mitgegeben wird. Antworten auf diese Nachrichten enthalten – sofern die Gesprächspartner die Regeln der ACL beachten – den Identifikator im `in-reply-to`-Feld. Anhand dieses Identifikators können eingehende Nachrichten laufenden Konversationen zugeordnet werden. Fehlt der Identifikator, wird dies als Beginn einer neuen Konversation interpretiert.

Neue Konversationen werden in der Protokollfabrik erzeugt. Deren Netz ist in Abbildung 5.11 zu sehen. Im oberen Bereich werden links reaktive und rechts proaktive Konversationen angestoßen. Die mittleren Transitionen und Stellen des oberen Bereichs sind für die Initialisierung der Protokollfabrik und die Vergabe der Konversationsidentifikatoren zuständig. Im unteren Bereich erfolgt, nachdem ein Protokoll im oberen Bereich ausgewählt wurde, die eigentliche Instanziierung und Initialisierung der Konversation. Dabei wird auch die den Beginn der Konversation auslösende Nachricht zugestellt.

Bei der Auswahl von zu instanziierenden Protokollen wird auf die Wissensbasis des Agenten zurückgegriffen, die eine Zuordnung von Mustern für eingehende Nachrichten auf passende Protokolle bietet. Falls eine eingehende Nachricht keinem Muster entspricht, landet sie in der Stelle „nicht erkannt“. Hier soll eigentlich eine Standardreaktion erfolgen, die aber noch nicht implementiert ist: Die FIPA sieht vor, dass Agenten auf unverständliche Nachrichten mit einem `not-understood`-Performativ reagieren (siehe Abschnitt 4.2.1).

Eine einfache Wissensbasis ist in Abbildung 5.12 zu sehen.[5] Wieder ist im oberen Teil die Initialisierung, die das anfängliche Wissen aus einer bei der Erzeugung

[5] Um die Grafik trotz der vielen Kanten zu der einen Stelle „Wissen“ übersichtlich zu halten, werden in Abbildung 5.12 virtuelle Kopien der Stelle verwendet. Die virtuellen Stellen sind an der doppelten Umrandung zu erkennen. Es handelt sich dabei lediglich um eine abkürzende grafische Notation zur Vermeidung langer, geknickter Kanten.

übergebenen Zeichenkettendarstellung in das eigentliche Schlüssel-Wert-Format umwandelt. Die Stelle „Wissen" wird von vielen Transitionen genutzt, die alle über synchrone Kanäle unterschiedliche Funktionen für den Agenten, die Protokollfabrik und die Konversationen bereitstellen. Transitionen auf der linken Seite bieten modifizierenden Zugriff, der die Wissensmarke kurzzeitig von der zentralen Stelle entfernt. Dazu gehören das Eintragen neuer Schlüssel mit Wert (`:new`), das Überschreiben eines Wertes für einen Schlüssel (`:replace`) oder das Modifizieren eines als Wert abgelegten Objektes (`:modify`).

Auf der rechten Seite befinden sich lesende Zugriffe, mit denen ein Wert eines Schlüssels (`:ask`) oder die Existenz eines Schlüssels (`:exists`) abgefragt werden kann. Der `:exists`-Zugriff ist nötig, weil der `:ask`-Zugriff über einen synchronen Kanal schlichtweg nicht schaltet, falls es den angefragten Schlüssel nicht gibt.

Beide Zugriffe werden in einer abgewandelten Form auch für die Abfrage von Protokollen in Abhängigkeit von einer eingegangenen Nachricht angeboten. Dabei wird die übergebene Nachricht gegen die bekannten Muster abgeglichen, um in Frage kommende Protokolle herauszufinden. Das Protokoll mit dem speziellsten passenden Muster wird ausgewählt. Dieser Musterabgleich stützt sich auf die in Abschnitt 5.2.4 beschriebene Subsumptionsrelation.

In Abbildung 5.13 ist ein Beispiel für die textuelle Darstellung des Wissensbasisinhalts zu sehen. Dieser Text kann bei Erzeugung der Wissensbasis in das Schlüssel-Wert-Format der Implementierung umgewandelt werden. Die Zeilen mit den Prozentzeichen trennen die einzelnen Schlüssel-Wert-Paare. Schlüssel und Wert sind durch das Gleichzeichen getrennt. Der letzte Eintrag in Abbildung 5.13 bedeutet beispielsweise, dass unter dem Schlüssel `"aNr"` ein Objekt vom Typ Integer, das mit dem Wert `0` initialisiert wurde, abzulegen ist.

Die Schlüssel, die mit `protocol` anfangen, erfahren eine gesonderte Behandlung, um den Index über Nachrichtenmuster und die dazu passenden Protokolle aufzubauen. Der erste Eintrag der AMS-Wissensbasis beschreibt zum Beispiel, dass auf eine eingehende Nachricht hin, die in der Sprache FIPA-SL0 abgefasst ist und eine Aktion namens `"move"` anfordert, das Protokollnetz `ams_move` als Vorlage für die zu beginnende Konversation verwendet werden soll.

5.2.2 Verwaltungsdienste

Die in den FIPA-2000-Spezifikationen vorgesehenen Verwaltungsdienste Agent Management System (AMS) und Directory Facilitator (DF) sollen wie Agenten ansprechbar sein, d.h. sie müssen ACL-Nachrichten unter einer fest vorgegebenen Agentenadresse entgegennehmen und beantworten können. Die in der FIPA Agent Management Specification [FIPA00023] beschriebenen Funktionen dieser beiden Dienste sind allesamt über ACL-Nachrichten auf Agentenebene abwickelbar. Daher liegt der Gedanke auf der Hand, AMS und DF als ganz normale Agenten auf Basis des im vorigen Abschnitt vorgestellten Standard-Agenten zu implementieren.

Das AMS verwaltet ein Verzeichnis, in dem alle auf der Plattform arbeitenden Agenten eingetragen sein müssen. Aber auch externen Agenten steht es frei, mittels der

```
protocol ams_move=(request :language FIPA-SL0
 :content "((action (agent-identifier) (move)))")
%%%
protocol ams_init=(inform :language FIPA-SL0
 :content "((knowledge platform))")
%%%
protocol ams_transmit=(request :language FIPA-SL0
 :content "((action (agent-identifier) (transmit-agent)))")
%%%
protocol ams_transmit=(request :language FIPA-SL0
 :content "((action (agent-identifier) (new-agent)))")
%%%
protocol ams_register=(request :language FIPA-SL0
 :content "((action (agent-identifier) (register)))")
%%%
protocol ams_deregister=(request :language FIPA-SL0
 :content "((action (agent-identifier) (deregister)))")
%%%
protocol ams_modify=(request :language FIPA-SL0
 :content "((action (agent-identifier) (modify)))")
%%%
protocol ams_search=(request :language FIPA-SL0
 :content "((action (agent-identifier) (search)))")
%%%
protocol ams_getDescription=(request :language FIPA-SL0
 :content "((action (agent-identifier) (get-description)))")
%%%
agentDesc=class de.renew.agent.common.VTIndex
%%%
aNr=class java.lang.Integer 0
```

Abbildung 5.13: Die anfängliche Wissensbasis des AMS-Agenten

Funktionen `register`, `modify` und `deregister` einen Eintrag im Verzeichnis anzulegen, zu ändern und auch wieder zu löschen. Für jede dieser Funktionen, die als kommunikativer Akt vom Typ `request` (entsprechend dem FIPA-Request-Interaktionsprotokoll) angefordert werden, verfügt der AMS-Agent über ein Mulan-Protokoll. Als Beispiel ist in Abbildung 5.14 das Registrierungsprotokoll dargestellt. Jedes der Protokolle extrahiert den zu aktualisierenden Datensatz aus der angekommenen Nachricht, sucht den zum Agentennamen des Datensatzes passenden Eintrag im Verzeichnis und führt die angeforderte Änderung durch. Das Verzeichnis selber ist in der Wissensbasis des Agenten abgelegt (siehe Eintrag `agentDesc` in Abbildung 5.13).

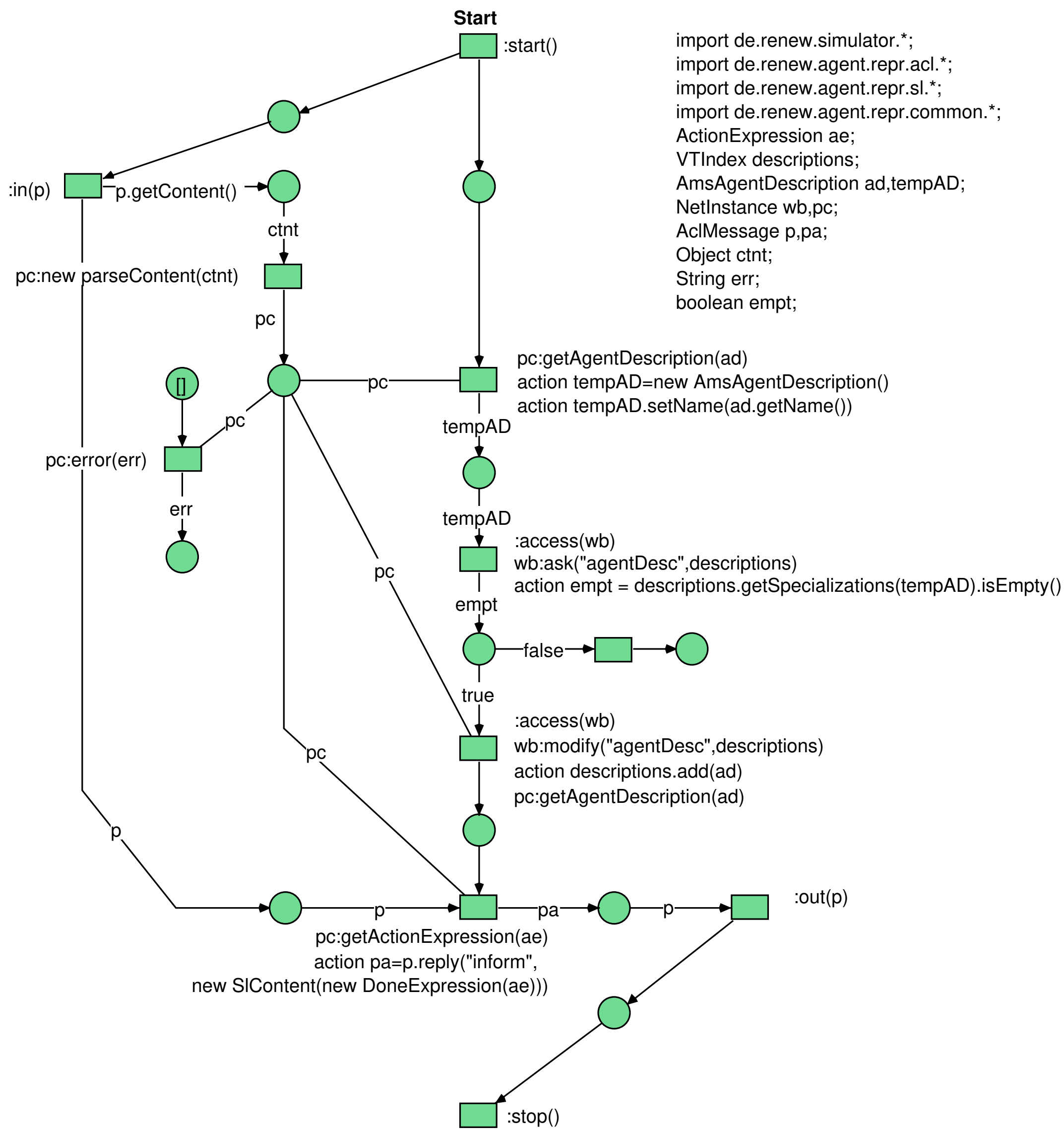


Abbildung 5.14: Das Registrierungsprotokoll `ams_register` des AMS-Agenten

Der DF verwaltet ebenfalls ein Verzeichnis von Agentenbeschreibungen, nur dass hier zusätzliche Informationen über das Dienstangebot der Agenten abgelegt werden. Da dieselbe Funktionalität wie beim AMS für die Pflege der Verzeichniseinträge benötigt wird, kann der DF einfach dieselben Protokolle wie das AMS verwenden.

Beide Verzeichnisdienste bieten natürlich auch eine Funktion zum Abfragen bzw. Suchen von Informationen aus der Datenbank. Dabei wird wieder auf ein `request`-Performativ mit dem Funktionsnamen `search` und einem Suchmuster reagiert. Das Suchmuster ist ein unvollständig ausgefüllter Verzeichniseintrag, der gegen alle vorhanden Einträge abgeglichen wird. Hierbei kommt wiederum die in Abschnitt 5.2.4 erläuterte Subsumptionsrelation zum Einsatz. Die bei FIPA2000-Plattformen vorgesehene Zusammenarbeit von DF-Diensten verschiedener Plattformen bei der Suche nach Einträgen ist noch nicht implementiert.

Das AMS hat mehr Aufgaben als nur die Verwaltung eines Verzeichnisses aller bekannten Agenten. Dazu gehört vor allem die in Abschnitt 4.4.2 vorgestellte Lebenszyklusverwaltung der Agenten auf der Plattform. Die Schnittstelle zu dieser Funktionalität ist aber in der FIPA Agent Management Specification [FIPA00023] nicht beschrieben, weil der Lebenszyklus der Agenten sehr nah mit der Implementierung der Agenten und der Agentenplattform verbunden ist. Das ist auch der Grund, warum die abstrakte Architektur der FIPA die Behandlung von Lebenszyklen explizit ausschließt.

Da auch in CAPA die Lebenszykluszustände der Agenten eng mit der Anbindung der Agenten an den Nachrichtentransportdienst und anderen technischen Aspekten der Plattform verknüpft ist, übernimmt ein spezieller Agent, „Plattformagent“ genannt, die Verwaltung dieses gesamten Komplexes. Der Plattformagent ist ein besonderer Agent, der zusätzlich zur üblichen Wissensbasis über ein Plattformobjekt verfügt, auf das von den Protokollen des Agenten zugegriffen werden kann. Der Plattformagent bietet mit seinen Protokollen dem AMS Dienstleistungen zur Plattformsteuerung an. Dazu gehören vor allem das Erzeugen und das Beenden von Agenten sowie alle anderen Zustandsübergänge im Lebenszyklus eines Agenten. Damit stellt der Plattformagent die Agenten-Hülle des technischen Teils der Plattform dar. Das AMS ist der einzige Agent, der auf die Dienstleistungen des Plattformagenten zugreifen darf.

Die Funktionalität des AMS ist über die Verzeichnisfunktionen gemäß der FIPA-Spezifikation hinaus um zunächst drei weitere Anfragen erweitert worden:

- Ein `request`-Performativ mit der Aktion `new-agent` veranlasst die Erzeugung eines Agenten auf der Plattform. Bisher werden ausschließlich MULAN-Agenten erzeugt, die auf dem im vorigen Abschnitt 5.2.1 beschriebenen Netz basieren. Übergeben werden muss eine Beschreibung der Wissensbasis des zu erzeugenden Agenten, d.h. des zu verwendenden Wissensbasisnetzes und eine Repräsentierung des anfänglichen Inhalts der Wissensbasis.

- Die Aktion `transmit-agent` hat genau dieselben Auswirkungen wie `new-agent`. Die Erzeugung eines Agenten kann auch dem Zweck dienen, eine lokale Kopie eines von einer anderen Plattform herüberziehenden Agenten herzustellen.

- Eine `move`-Aktion in der Anforderung veranlasst das AMS, den in der Anforderung bezeichneten Agenten (u.a. mittels `transmit-agent`) auf eine andere, ebenfalls in der Anforderung zu spezifizierende, Plattform zu verschieben. Die Diskussion der Mobilität auf der CAPA-Plattform geht über den Fokus dieses Abschnittes hinaus und soll daher erst in Abschnitt 5.3.2 aufgegriffen werden.

Das AMS bearbeitet diese Anfragen, indem es den Plattformagenten entsprechend instruiert. Aktionen, die die anderen Lebenszykluszustandsübergänge zum Aussetzen oder Beenden von Agenten beschreiben, werden demnächst hinzukommen.

5.2.2.1 Lebenszyklusverwaltung

Die FIPA2000-Suite definiert ein Zustandsdiagramm (siehe Abbildung 4.7) mit sechs grundlegenden Zuständen, in denen sich ein Agent in seinem Lebenszyklus befinden kann. Die Beschreibung der Zustände bleibt aber recht allgemein, es wird nur festgelegt, wie sich jeder Zustand auf die Nachrichtenzustellung zu diesem Agenten auswirkt. Eine Definition, ob der Agent auch in den nicht-aktiven Zuständen noch Nachrichten versenden kann, oder inwieweit der Agent in diesen Zuständen überhaupt ausgeführt wird, fehlt.

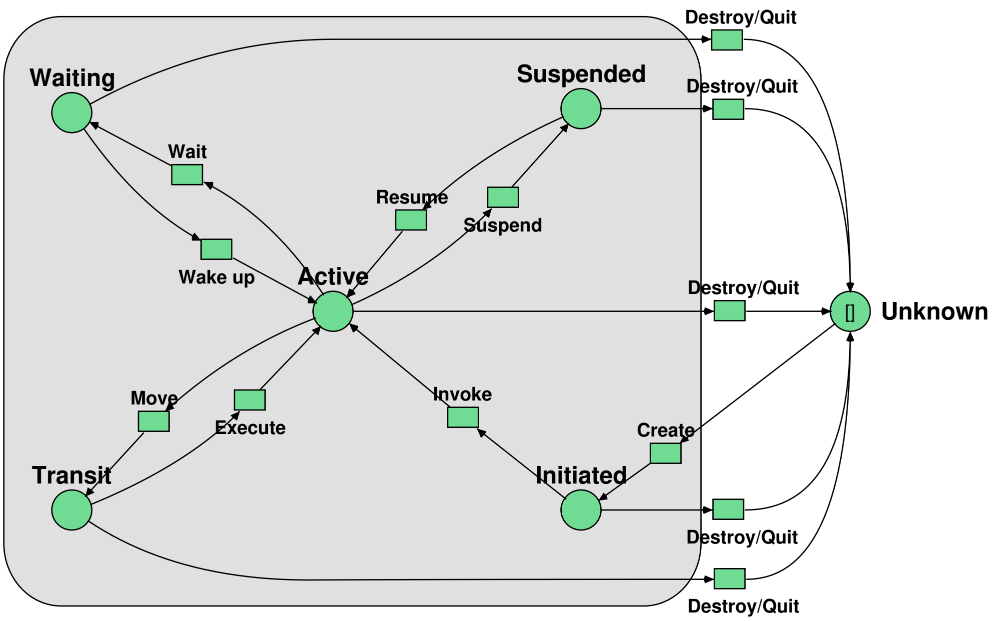


Abbildung 5.15: Der Agenten-Lebenszyklus als Petrinetz

Die Umsetzung eines Zustandsdiagramms in ein Petrinetz ist simpel, daher liegt der Gedanke nahe, die Lebenszyklusverwaltung als Referenznetz zu implementieren – wie z.B. in Abbildung 5.15 dargestellt. Allerdings hat diese Lösung – wie im Folgenden erläutert wird – starke Nachteile, so dass letztendlich die Java-Implementierung vorzuziehen ist.

Für eine vollständige Implementierung des Lebenszyklus' durch ein Petrinetz wie in Abbildung 5.15 fehlen noch Transitionen mit synchronen Kanälen zur Abfrage des aktuellen Zustands. Ferner muss entschieden werden, ob jeder Agent seine eigene Instanz dieses Netzes zugewiesen bekommt, oder ob alle Agenten als Marken in einer einzigen Instanz des Lebenszyklusnetzes verwaltet werden. Die Variante, jedem Agenten eine Lebenszyklusnetzinstanz zuzuweisen, erfordert eine besondere Behandlung des „Unknown"-Zustands: Unbekannte Agenten haben noch keine Lebenszyklusnetzinstanz, bei welcher der Zustand abgefragt werden könnte. Letztendlich würde der „Unknown"-Zustand dabei als Stelle im Netz überflüssig. Wenn alle Agenten in einer Lebenszyklusnetzinstanz verwaltet werden, gibt es beim „Unknown"-Zustand das Problem, dass nicht alle jemals erdenklichen Agenten in der entsprechenden Stelle liegen können, da es unendlich viele sind. Also ist auch hier eine besondere Behandlung des „Unknown"-Zustands erforderlich.

Darüber hinaus ist es – unabhängig von der Zuordnung von Agenten zu Lebenszyklusnetzinstanzen – in allen Zuständen nicht ohne weiteres möglich, die Tatsache abzufragen, ob sich ein Agent *nicht* in diesem Zustand befindet. Transitionen können nur auf die Existenz von Marken prüfen und reagieren auf das Nicht-Vorhandensein einer Marke mittels Nicht-Schalten. Es werden also weitere (komplementäre) Stellen und weitere Transitionen und Kanäle nötig, welche die Ermittlung des komplementären Zustands ermöglichen.

Wenn ein Zustandsübergang angefordert wird, aber der Agent sich nicht in einem Ausgangszustand für diesen Übergang befindet, ergibt sich ein ähnliches Problem: Die Synchronisation kommt aufgrund der nicht erfüllten Vorbedingung nicht zustande, d.h. die anfordernde Transition wartet, bis alle Vorbedingungen erfüllt sind. Das kann dazu führen, dass nach langer zeitlicher Verzögerung die Zustandsübergangsanforderung umgesetzt wird, obwohl der anfordernde Agent das schon längst nicht mehr wünscht. Das Abfangen von gerade nicht möglichen Übergängen würde zu weiteren Transitionen, Stellen und Kanälen führen, die das Lebenszyklusnetz bis zur Unleserlichkeit überfrachten können.

Daher habe ich für die Implementierung der Plattform die Verwaltung der Zustände und Übergänge in Java implementiert. Java-Methoden können mittels Exceptions die Unzulässigkeit von angeforderten Übergängen melden. Der dabei eingehandelte Nachteil ist aber die übliche Verkomplizierung der nebenläufigen und synchronisierten Zugriffe auf die Zustandsdatenbank.

Kooperatives Abschalten von Agenten. Eine Frage, die zuerst im Zusammenhang mit den Mobilitätsüberlegungen (siehe Abschnitt 5.3.2) aufgefallen ist, stellt sich auch bei der „gnädigen" Terminierung von Agenten: Wie kann die Plattform einen Agenten so in einen Ruhezustand versetzen, dass er keine offenen Konversationen zurücklässt? Im Falle der Mobilität geht die Frage sogar noch weiter: Wie kann sichergestellt werden, dass der Zustand des Agenten vollständig erfasst werden kann, um ihn auf eine andere Plattform zu übertragen?

Das Problem lässt sich lösen, wenn es eine Möglichkeit gibt, den Agenten den Zeitpunkt des Lebenszykluszustandswechsels exakt bestimmen zu lassen. Zu diesem Zweck wird die Agentenschnittstelle um einen dritten synchronen Kanal namens `:idle` erweitert, dessen agentenseitige Implementierung bereits im vorigen Abschnitt 5.2.1 kurz vorgestellt wurde. Falls die Plattformverwaltung einem Agenten bei einem Zustandsübergang die Chance zur Mitwirkung geben möchte, aktiviert sie den `:idle`-Kanal und wartet darauf, dass der Agent dies ebenfalls tut. Der suggestive Charakter des Kanalnamens deutet an, dass der Agent idealerweise alle Konversationen, Protokolle und andere Tätigkeiten eingestellt haben sollte, wenn er diesen Kanal aktiviert.

Dass ich mich für die Implementierung der Lebenszyklusverwaltung in Java entschieden habe, wird im Zusammenhang mit einem kooperativen Zustandswechsel problematisch. Wenn der Agent den Zeitpunkt des Zustandsübergangs bestimmen können soll, muss die Verwaltung atomar sich mit dem `:idle`-Kanal synchronisieren und den Zustand in der Datenbank aktualisieren. Dies lässt sich in Java nur realisieren, indem ein Monitor zum Einsatz kommt. Nebenläufige Zugriffe auf die Datenbank werden damit solange unterbunden, wie der Agent sich noch nicht synchronisiert hat. Der Haken ist, dass der Agent auf diese Weise die Lebenszyklusverwaltung beliebig lange außer Betrieb setzen und somit alle anderen Agenten blockieren kann. Eine aufwändige, feiner granulierte Sperrstrategie, die z.B. nur den Eintrag bezüglich des einen Agenten blockiert, würde Abhilfe schaffen. Aber die ist in Java relativ kompliziert zu realisieren. Bei einer Netzimplementierung der Lebenszyklusverwaltung wäre das ganze Problem von Anfang an bereits gelöst, weil Transitionen Marken erst dann reservieren, wenn alle Aktivierungsbedingungen zutreffen.

Auswirkungen der Lebenszykluszustände auf Agenten. Entsprechend den Minimalvorgaben der FIPA2000-Spezifikationen ist bisher nur die An- und Abkopplung der Agenten an den Nachrichtentransportdienst mit den Zustandsübergängen verknüpft. Eine Beeinflussung der den Agenten zugeteilten Rechenzeit wäre möglich, ist aber zunächst nicht implementiert. Sie würde entweder einen Eingriff in den Renew-Simulator oder eine Erweiterung des Standard-Agentennetzes um Funktionen zum Stilllegen des Netzes bedeuten.

Damit ist auch noch kein Mechanismus zum tatsächlichen Beenden von Agenten vorhanden: Der terminierte Agent wird zwar vom Nachrichtentransport abgekoppelt, kann also keine Auswirkungen in der Umwelt oder bei anderen Agenten mehr haben. Intern kann sein Netz aber noch solange weiterschalten, wie mindestens ein Protokoll im Agenten aktiv und nicht auf Nachrichten von außen angewiesen ist.

Der kooperative Zustandsübergang mittels `:idle`-Kanal wird zur Zeit nur im Zusammenhang mit Mobilität tatsächlich genutzt. Der in den Zustand `transit` übergehende Agent wird dann teilweise vom Nachrichtentransport abgekoppelt: Er darf danach ausschließlich mit dem AMS der Plattform kommunizieren. Diese Kommunikation ist notwendig, um die Wissensbasis des Agenten an die Zielplattform versenden zu können. Ein neu erzeugter Agent ist ebenfalls zunächst in der Kommunikation beschränkt, bis er endgültig aktiviert wird. In beiden Zuständen (`transit` und

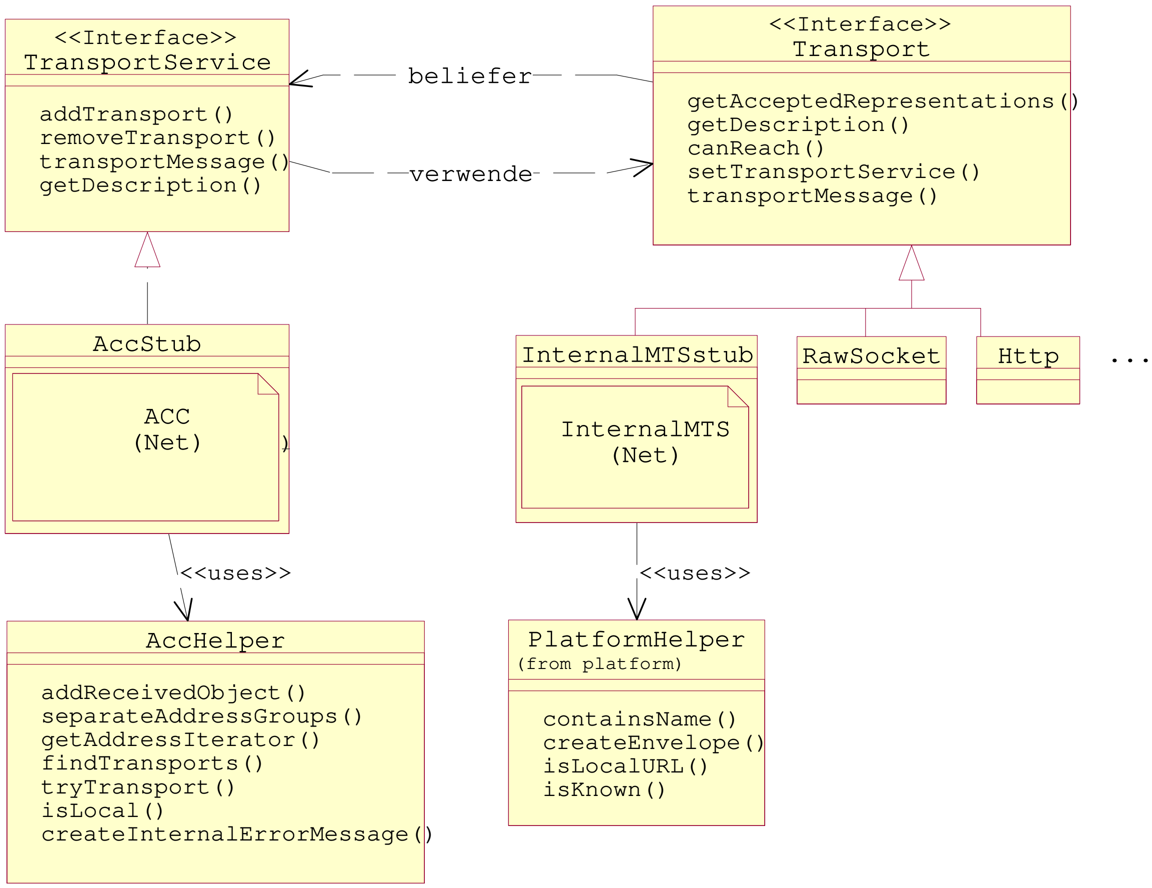


Abbildung 5.16: Die am Nachrichtentransport beteiligten Klassen und Netze

`initiated`) soll durch die Kommunikationseinschränkung sichergestellt werden, dass der Agent keine undefinierten Situationen hervorruft (z.B. durch Verwendung einer ungültigen Identifikation).

Bisher haben die Lebenszykluszustände `suspended` und `waiting` nur die Auswirkung, dass der Agent vom Transportdienst abgekoppelt wird. Ob in diesen Zuständen ebenfalls eine beschränkte Kommunikation mit dem AMS wünschenswert wäre, muss diskutiert werden, wenn diese Zustände tatsächlich eingesetzt werden sollen.

5.2.3 Nachrichtentransport

Der grundlegende Aufbau des Message Transport Services (MTS) ist bereits bei der Beschreibung der CAPA-Architektur in den Abschnitten 5.1.1 und 5.1.2 umrissen worden. Die Klassenstruktur der konkreten Umsetzung ist in Abbildung 5.16 zu sehen. Eine zentrale Instanz im MTS, in CAPA schlichtweg `TransportService` genannt, nimmt die Aufgaben des Agent Communication Channel (ACC) wahr und leitet Nachrichten von überallher nach überallhin weiter. Für den Versand und Empfang von Nachrichten über das Netzwerk können einzelne „Transporte“ in den MTS ein- und auch wieder ausgeklinkt werden. Jeder `Transport` steht für ein Transportprotokoll wie z.B. HTTP oder WAP, über das Nachrichten transportiert werden können. Die in Abbildung 5.16 aufgeführten `Transport`-Implementierungen sind allerdings exemplarisch, d.h. bis auf den `InternalMTS` existieren sie in der CAPA-Implementierung nicht.

Ein `Transport` verschickt auf Anweisung des ACC (d.h. auf einen Aufruf der durch den `Transport` implementierten `transportMessage`-Methode hin) Nachrichten an andere Plattformen und reicht von außen eingegangene Nachrichten an den zentralen `TransportService` (mittels dessen `transportMessage`-Methode) durch. Der zentrale `TransportService` wählt für jede Nachricht einen passenden `Transport` zur Weiterleitung aus. Um einen passenden `Transport` zu ermitteln, stehen dem `TransportService` die Methoden `getAcceptedRepresentations`, `canReach` und `getDescription` des `Transport`-Interfaces zur Verfügung. Der Rückgabewert der `getDescription`-Methode ist eine FIPA2000-konform aufgebaute `mtp-description`, die als Baustein für die plattformweite `ap-transport-description` verwendet werden kann und soll.

In den Abbildungen 5.17 und 5.18 sind zwei beispielhafte Transportwege einer Nachricht als UML-Sequenzdiagramm dargestellt. In der Beispielkonfiguration sind genau zwei `Transport`-Implementierungen beim zentralen `TransportService` registriert: Der `InternalMTS` wird in in Abschnitt 5.2.3.2 beschrieben werden und stellt eine Implementierung der in Abschnitt 5.1.1 entwickelten internen Transportschnittstelle zum Agenten dar. Der `TcpIpMTS` wird in Abschnitt 5.2.3.3 vorgestellt, dabei handelt es sich um eine prototypische Umsetzung eines externen Transports, die Nachrichten als Zeichenketten über eine einfache TCP-Verbindung verschickt. Der zentrale Transportdienst wird durch das im Abschnitt 5.2.3.1 erläuterte ACC-Netz bereitgestellt. Die an der ACC-Lebenslinie angebrachten Transitionssymbole bringen den hier dargestellten Ablauf mit der Netzdarstellung des ACC in Verbindung.

In Abbildung 5.17 verschickt ein Agent A eine Nachricht an einen anderen Agenten B auf derselben Plattform. Die Nachricht wird über den `:send`-Kanal an der internen Transportschnittstelle entgegengenommen und dann, ergänzt um einen Umschlag, an den zentralen Transportdienst weitergeleitet. Der Transportdienst bearbeitet den Nachrichtenumschlag und ermittelt dann die Transporte, welche die Nachricht an das gewünschte Ziel befördern können. Im Beispiel ist das nur der interne Transportdienst, da der Empfänger auf derselben Plattform beheimatet ist. Da die Nachricht bereits in einem für die lokale Zustellung geeigneten Repräsentierung vorliegt, wird sie erneut dem `InternalMTS` übergeben. Dieser stellt die Nachricht dem Empfänger über dessen `:receive`-Kanal zu und meldet an den ACC zurück, dass keine Probleme aufgetreten sind.

In Abbildung 5.18 verschickt der Agent A eine Nachricht an einen Agenten auf einer anderen Plattform. Der Ablauf unterscheidet sich ab dem Punkt, wo die Ermittlung der geeigneten Transporte den externen TCP/IP-Transportdienst ergeben hat. Der externe Transport kann die Nachricht nur in Zeichenkettendarstellung verschicken, so dass der zentrale Transportdienst zunächst eine Umwandlung mittels des in Abschnitt 5.2.5 beschriebenen Transformationsdienstes vornehmen muss. Anschließend

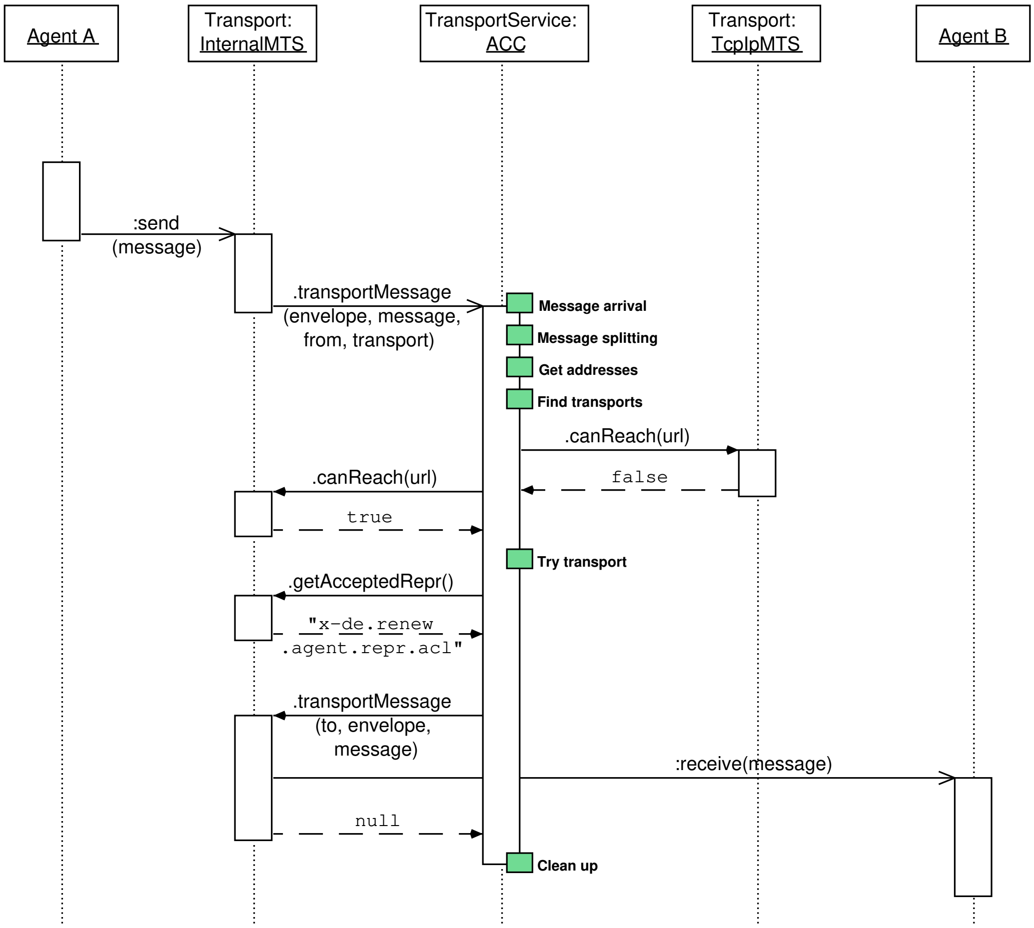


Abbildung 5.17: Lokaler Nachrichtentransport von Agent A an Agent B

kann die umgewandelte Nachricht über den `TcpIpMTS` verschickt werden, welcher den Erfolg an den ACC zurückmeldet.

Die Ankunft einer externen Nachricht im Transportsystem der Plattform unterscheidet sich nur unwesentlich von dem Versand einer Nachricht durch einen lokalen Agenten, daher wird dieser Fall hier nicht weiter behandelt. Die beiden hier gezeigten Sequenzdiagramme decken nur sehr einfache Fälle des Nachrichtentransports ab. Sie enthalten keine Information über die Behandlung von Fehlern während des Nachrichtentransports oder über die Zustellung von Nachrichten mit mehreren Empfängern. Diese Details werden aber bei der Beschreibung der einzelnen Transportdienstelemente in den folgenden Abschnitten angesprochen.

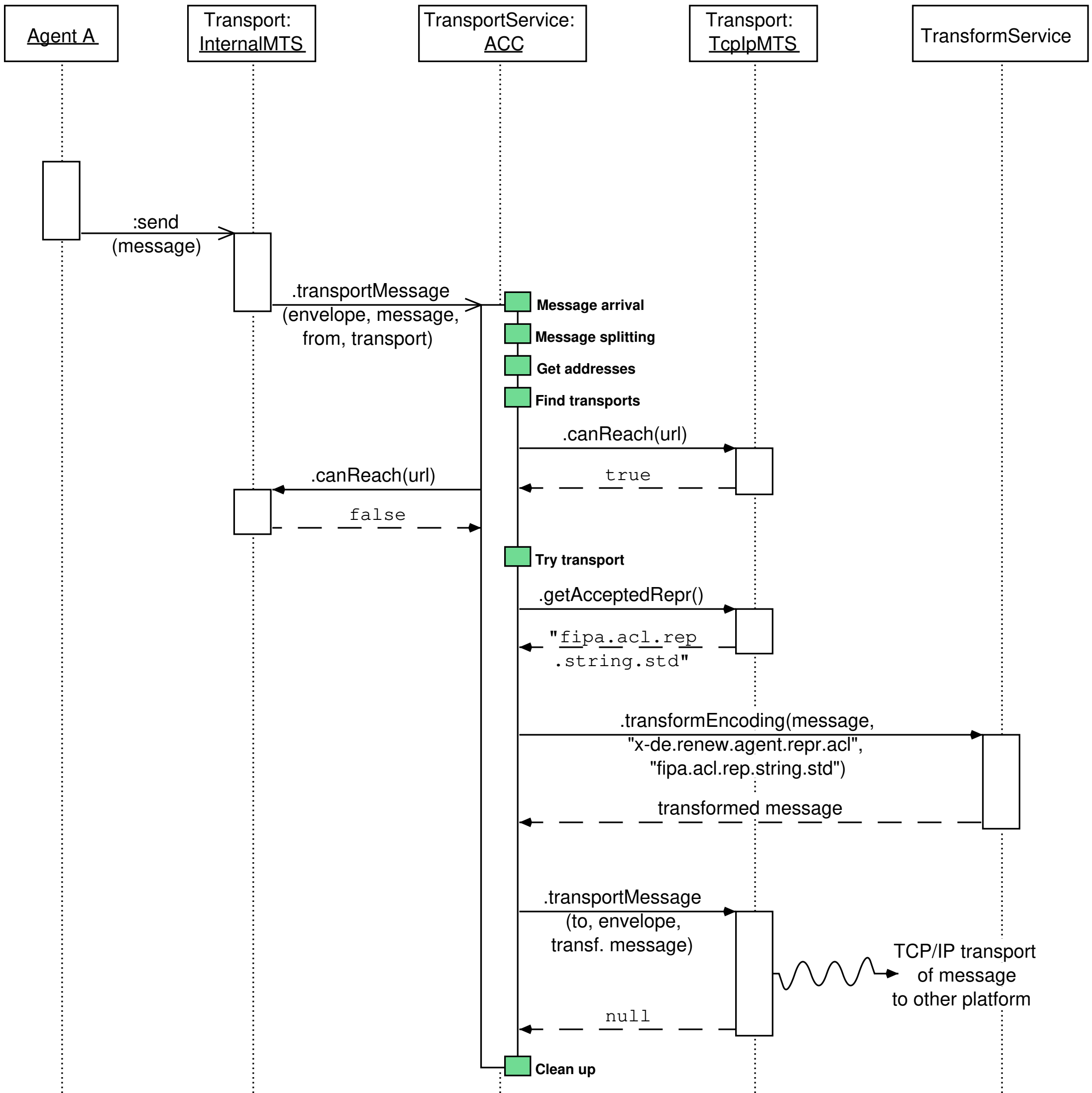


Abbildung 5.18: Nachrichtentransport von Agent A an einen externen Agenten

5.2.3.1 Zentrales Transportsystem

Der zentrale `TransportService` ist eines der wichtigsten Elemente von CAPA, da ohne diesen Dienst keine Kommunikation in der Agentenplattform stattfinden kann. Daher soll der Beschreibung seiner Implementierung hier recht viel Raum gewidmet werden. Zunächst gilt es, die Aufgabenbeschreibung des Dienstes zu konkretisieren. Danach kann die Arbeitsweise der Implementierung beschrieben werden, wobei zuvor eine Implementierungssprache (die Wahl im Rahmen dieser Arbeit besteht natürlich zwischen den Referenznetzen oder Java-Code) gewählt worden sein muss.

Ist der Arbeitsablauf der Implementierung klar, so bleiben immer noch einige Detailprobleme zu lösen. Erwähnenswert sind in diesem Zusammenhang die Zwischenspeicherung der Daten während ihrer Bearbeitung, der Aufwand zur Erzeugung aussagekräftiger Fehlermeldungen oder die Integration des Transformationsdienstes in den Weiterleitungsprozess (wie in Abschnitt 5.1.3 vorgeschlagen).

Die Aufgabe. Die Arbeitsweise des Agent Communication Channel (ACC) als zentrale Weiterleitungsinstanz des Nachrichtentransportdienstes ist in der FIPA Message Transport Service Specification[FIPA00067] im Abschnitt 3.3 recht genau beschrieben (siehe auch Abschnitt 4.4.1 in dieser Arbeit). Die Weiterleitung einer Nachricht wird bewerkstelligt, indem der ACC den Umschlag der Nachricht folgendermassen interpretiert:

1. Der Umschlag wird mit einem Empfangsstempel versehen.
2. Falls das `intended-receiver`-Feld des Umschlags leer ist, wird der Inhalt des `to`-Feldes zum Aufbau des `intended-receiver`-Feldes herangezogen.
3. Falls die Nachricht an mehrere Empfänger adressiert ist, kann der ACC mehrere Kopien der Nachricht erstellen. Der Umschlag jeder Kopie enthält dann nur noch eine Teilmenge der ursprünglichen Adressaten im `intended-receiver`-Feld. Die Kopien werden unabhängig voneinander weitergeleitet.
4. Um die Nachricht einem Adressaten zuzustellen, wird eine Liste von URLs im `addresses`-Feld aus dem im `intended-receiver`-Feld angegebenen Agentenidentifikation entnommen. Die im `addresses`-Feld angegebenen Adressen sind nach Präferenz geordnet, die erste soll zuerst probiert werden.
5. Ist die Zustellung der Nachricht an die erste Adresse nicht gelungen, wird diese Adresse aus dem `addresses`-Feld gestrichen und die nächste Adresse probiert. So werden bei beständigem Misserfolg alle Adressen zunächst probiert und dann aus dem Umschlag entfernt.
6. Wenn keine Adresse (mehr) im Umschlag angegeben ist, wird eine Fehlermeldung an den Absender der Nachricht geschickt.

Diese Vorgehensweise ist eher sequentiell als nebenläufig. Die einzige Stelle, an der eine Nachricht nebenläufig verarbeitet werden kann, sind die unter Punkt 3 erzeugten unabhängigen Kopien der Nachricht.

Wahl der Implementierungssprache. Die Sequentialität der Nachrichtenweiterleitung, zusammen mit dem hohen technischen Anteil der Nachrichtentransportimplementierung, deutet darauf hin, dass die Verwendung von Java-Code vorteilhafter sein könnte als das Zeichnen von Referenznetzen. Für diverse Transportprotokolle über das Netzwerk stehen in der Java-Klassenbibliothek bereits fertige Klassen bereit, die nur noch in die eigene Implementierung eingebunden werden müssen. Für die Weiterleitung jeder Nachricht kann ein Thread alle notwendigen Schritte sequentiell abarbeiten.

Aber: Der Nachrichtentransportdienst ist insgesamt ein hochgradig nebenläufiges System, wenn viele Nachrichten aus unterschiedlichen Quellen gleichzeitig weitergeleitet werden müssen. In der CAPA-Architektur muss der zentrale `TransportService` Nachrichten von allen bekannten Transporten entgegennehmen können, wobei jeder `Transport` auch mehrere Nachrichten unabhängig voneinander gleichzeitig abliefern kann.

Für eine Java-Thread-Implementierung bedeutet das, dass es eine Sammelstelle geben muss, an der alle eingegangenen Nachrichten zwischengelagert werden können, bis der Weiterleitungsthread für die jeweilige Nachricht gestartet ist. Diese Sammelstelle sollte möglichst nebenläufig arbeiten, um nicht zum Flaschenhals für den Transportdienst als Ganzes zu werden. Das bedeutet wiederum, dass relativ aufwändige, auf dem Java-Monitor-Konzept basierende Konstrukte notwendig werden, die erst einmal fehlerfrei implementiert sein wollen.

Da die nebenläufige Behandlung mehrerer Nachrichten beim Zeichnen eines Referenznetzes quasi „von alleine“ mit abgedeckt ist, habe ich mich für die Implementierung des `TransportService`-Interfaces durch das Netz `ACC` (siehe Abbildung 5.19) entschieden.

Die technischen Teile der Implementierung sind in Methoden einer Hilfsklasse namens `AccHelper` ausgelagert. Diese Methoden sind gedächtnislos und seiteneffektfrei, global ansprechbar und haben somit einen funktionalen Charakter. Alle Informationen, die zur Ausführung einer dieser Methoden benötigt werden, müssen beim Aufruf übergeben werden. Jede Methode hat nur genau einen Rückgabewert, die beim Aufruf übergeben Objekte bleiben unverändert.

Die `AccHelper`-Klasse stellt also eine Funktionsbibliothek im Sinne der modularen Programmierung dar. Dank der Auslagerung der technisch notwendigen Codeblöcke in eine Funktionsbibliothek bleibt die Netzzeichnung übersichtlich. Der Fokus der Implementierung kann sich also ungestört auf den großen Ablauf des Prozedur und die Nebenläufigkeit darin richten.

Die Arbeitsweise der Implementierung. Die Transitionen der mittleren, senkrechten Achse bilden die Sequenz der Weiterleitungsschritte ab (von oben nach unten):

- „Message arrival“ stellt die Übergabeschnittstelle dar, an der eine von einem `Transport` empfangene Nachricht abgegeben werden kann. Gleichzeitig wird die Nachricht mit dem in Schritt 1 geforderten Empfangsstempel versehen.
- Die Transition „Message splitting“ kombiniert die Schritte 2 und 3: Der Nachrichtenumschlag wird auf einen `intended-receiver`-Eintrag überprüft, gegebe-

nenfalls ein neuer Eintrag angelegt und/oder mehrere Nachrichtenkopien erstellt. Wenn diese Transition geschaltet hat, ist sichergestellt, dass alle Empfänger einer Nachrichtenkopie über dieselben Adressen zu erreichen sind.

- Mittels der Transition „Get addresses“ werden die Schritte 4 und 5 der Weiterleitungsprozedur vorbereitet: Aus dem Nachrichtenumschlag wird die geordnete Liste der Zieladressen der Nachricht extrahiert.
- „Find transport“ sucht zu der ersten Adresse der extrahierten Liste die Menge der passenden Transporte. Ein Transportprotokoll passt, wenn er in der Lage ist, die genannte Adresse (die ja als Teil der URL auch einen Protokollnamen enthält) zu erreichen.
- Schlussendlich vollendet die Transition „Try transport“ den Schritt 4, indem ein Transportprotokoll aus der Menge der passenden Transporte beauftragt wird, die Nachricht zu versenden. Der `Transport` meldet dann zurück, ob der Versand erfolgreich war oder aus irgendeinem Grund gescheitert ist.
- Ist eine Nachricht erfolgreich zugestellt worden, so entfernt die Transition „Clean up“ alle Überbleibsel der Weiterleitungsprozedur aus dem Netz.

Ist hingegen der Versand der Nachricht fehlgeschlagen, so kommen die Transitionen links der Hauptachse zum Zuge (von unten nach oben):

- Zunächst geht „Transport failed“ davon aus, dass nur das spezielle, eben probierte Transportprotokoll nicht in der Lage war, die Nachricht zuzustellen. Die Marke wird zurückgelegt, um den nächsten `Transport` aus der Menge der passenden Transporte auszuprobieren.
- Wenn keine passenden Transporte mehr übrig sind, wird dies von der Transition „No more Transports“ diagnostiziert. Damit ist Schritt 4 der Weiterleitungsprozedur fehlgeschlagen. Gemäß Schritt 5 wird nun die nächste Adresse den Transitionen „Find transport“ und „Try transport“ vorgelegt.
- Ist die Liste aller Adressen aus dem Nachrichtenumschlag erfolglos abgearbeitet worden, so führt die Transition „No more addresses“ den finalen Schritt 6 der Weiterleitungsprozedur aus: Es wird eine Fehlernachricht erzeugt, adressiert an den Absender der ursprünglichen Nachricht. Die Fehlernachricht wird wieder in den Weiterleitungsprozess eingeschleust.

Mit diesen Transitionen ist die Hauptaufgabe des `TransportService`, Nachrichten entsprechend den ACC-Vorschriften weiterzuleiten, abgedeckt. Die nebenläufige Behandlung unabhängiger Nachrichten ist automatisch vorhanden, da alle Nebenbedingungen über Testkanten angebunden sind, also die Transitionen nicht in ihrer inhärenten Nebenläufigkeit beschränken.

Darüber hinaus können auch die in Schritt 3 erzeugten Nachrichtenkopien nebenläufig weiterverarbeitet werden, weil jede Kopie als unabhängige Marke von der flexiblen

Ausgangskante der Transition „Message splitting“ in die nachfolgende Stelle gelegt wird. Diese einfache Aufspaltung der Nachrichtenbearbeitung in mehrere, unabhängige Kontrollflüsse hätte in einer Java-Implementierung wiederum das aufwändige Starten mehrerer Threads erfordert.

Die blassgrauen Netzelemente im unteren Bereich stellen den Verwaltungsteil der zentralen `TransportService`-Implementierung dar. Hier kann – wie in der Plattformarchitektur angedacht – ein `Transport` in das Transportsystem ein- und wieder ausgeklinkt werden. Außerdem werden hier die `ap-transport-description` der Plattform sowie der `agent-identifier` des Plattform-AMS gepflegt. Die Transportbeschreibung stellt einen wesentlichen Teil der durch die FIPA-Agent-Management-Ontologie definierten Plattformbeschreibung (siehe Abschnitt 4.4.1 bzw. [FIPA00023] und [FIPA00067]) dar. In der `ap-transport-description`-Struktur wird abgelegt, welche Transportprotokolle von einer Agentenplattform verwendet werden und unter welchen Adressen Agenten auf der Plattform erreicht werden können. Die AMS-Adresse wird beim Weiterleiten von Nachrichten benötigt, wenn Fehlernachrichten erzeugt werden müssen, in denen als Absender der AMS-Agent der Plattform anzugeben ist. Sie sollte ebenfalls alle Adressen enthalten, unter denen die Plattform zu erreichen ist.

Zustandsinformationen im Referenznetz. Während eine Nachricht durch die Weiterleitungsprozedur läuft, fallen etliche Informationen an, die kurzfristig an die Nachricht gebunden werden müssen, später aber überflüssig werden. Dazu gehören zum Beispiel die aus dem Nachrichtenumschlag extrahierte Liste von Adressen, die Menge der passenden Transporte zu einer Adresse oder – für aussagekräftige Fehlerbenachrichtigungen unerlässlich – die Diagnosemeldung des letzten fehlgeschlagenen Schritts der Weiterleitungsprozedur.

Das Netz aus Abbildung 5.19 koppelt diese Zusatzinformationen an die Nachricht, indem alle mit der Nachricht zusammenhängenden Informationsstückchen in einer Marke in Form eines Tupels gruppiert werden. Dieses Vorgehen hat den Nachteil, dass am unteren Ende der Wiederholungsschleifen zum Ausprobieren von Adressen und Transporten ein sehr umfangreiches Tupel durch die Stellen geschoben wird, dessen Informationen in dem Moment gar nicht vollständig benötigt werden. Am konkreten Beispiel: Die Transition „Try transport“ benötigt aus dem Tupel `[envelope, message, addrs, to, transports, detail]` weder die Liste der durchzuprobierenden Adressen (`addrs`) noch die letzte Fehlerdiagnosemeldung vorhergehender Durchläufe (`detail`). Die vorhergehende Transition „Find transports“ hat ein noch krasseres Missverhältnis zwischen bewegter und benötigter Information: Aus dem Tupel `[envelope, message, addrs, detail]` ist ausschließlich das erste Listenelement der Adressenliste von Interesse.

Der Versuch, die jeweils unbenötigten Informationen in Stellen auszulagern, die nicht im Hauptkontrollfluss liegen, führt zu einem Netz ähnlich dem in Abbildung 5.20. Die gelb eingefärbten Stellen („Message store“ und „Error details“ sowie zwei unbenannte Stellen innerhalb der Wiederholungsschleifen) nehmen jeweils einen Teil der Informationen auf. Mittels virtueller Kopien dieser Stellen können die Marken an einer ande-

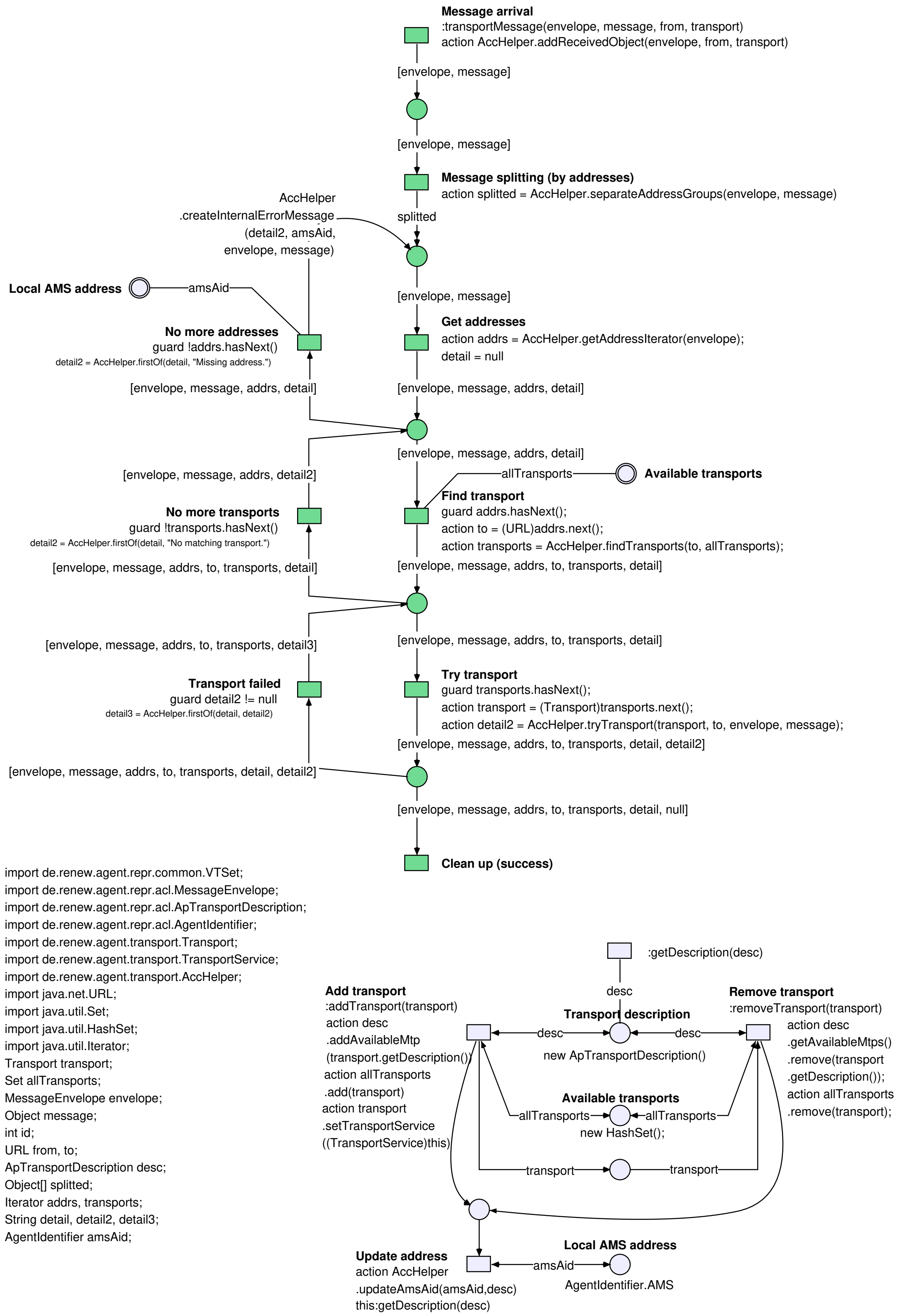


Abbildung 5.19: Die Implementierung des ACC (vollständiger Quellcode)

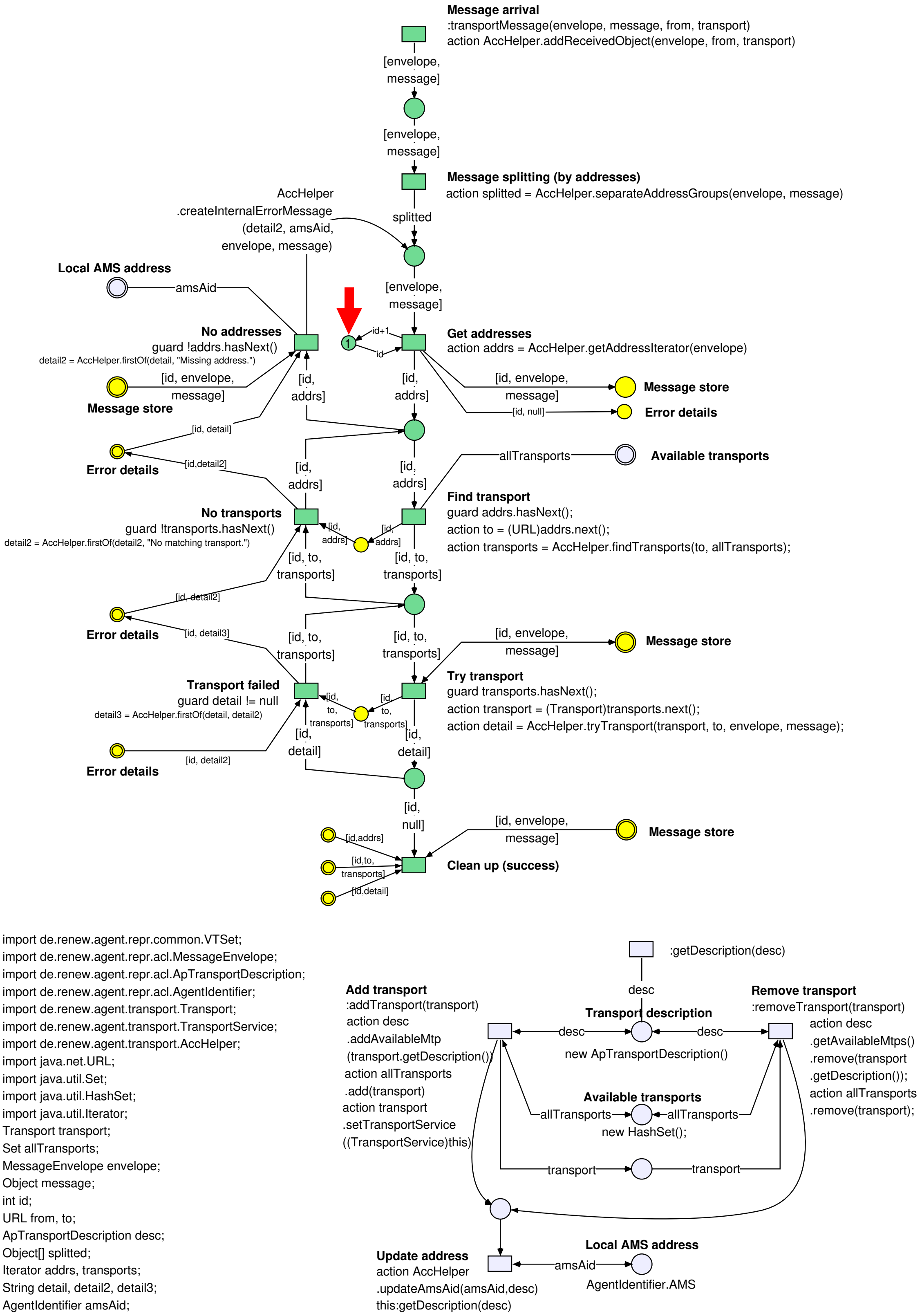


Abbildung 5.20: ACC-Implementierung mit getrennter Speicherung der Daten

ren Transition wieder abgerufen werden, ohne Kanten quer durch das Netz zeichnen zu müssen. Dennoch leidet die Übersichtlichkeit des Netzes unter diesen zusätzlichen Stellen und Kanten erheblich.

Ganz abgesehen von der optischen Komplexitätszunahme bringt die Auslagerung der nicht benötigten Informationen in andere Stellen ein Nebenläufigkeitsproblem mit sich. Um die Kontrollflussinformationen und die ausgelagerten Informationen wieder zusammenbringen zu können, muss ein gemeinsamer Schlüssel (im Netz immer in der Variablen `id`) an allen zu einem Verarbeitungsdurchgang gehörenden Informationen angebracht sein. Dieser Schlüssel muss eindeutig sein. Ein garantiert eindeutiger Schlüssel lässt sich aber nur bei Einschränkung der Nebenläufigkeit erzeugen. Die Schlüsselerzeugung muss nicht zwingend in die Serialisierung einer Transition ausarten (in Abbildung 5.20 passiert dies mit der Transition „Get addresses“, weil die mit dem roten Pfeil markierte Stelle nur eine Marke enthält), reduziert aber immer die theoretisch unendliche Nebenläufigkeit auf begrenzt viele parallele Schaltvorgänge (siehe [Mol1996, S. 418ff.]).

Daher scheint die zuerst vorgestellte Variante, trotz der umfangreichen Tupel, die bessere zu sein.

Fehlermeldungen. Ein immer im Hintergrund stehendes Ziel bei der Implementierung ist das möglichst frühe Erkennen von Fehlern und das Erzeugen aussagekräftiger Fehlermeldungen. Je früher und detaillierter ein Fehler gemeldet wird, desto einfacher und besser kann die Fehlerursache vom Entwickler gefunden und beseitigt werden.

Ein FIPA-konformer ACC schickt eine Fehlermeldung an den Absender einer Nachricht, wenn die Nachricht nicht weitergeleitet werden konnte. Der Aufbau und Inhalt der Nachricht wird von zwei Spezifikationen beeinflusst: Die FIPA Transport Service Specification [FIPA00067] schreibt eine ACL-Nachricht mit `failure`-Performativ vor, dessen Inhalt aus einem `internal-error`-Prädikat mit einer Fehlerbeschreibung als Argument besteht. Das `failure`-Performativ wird in der FIPA Communicative Act Library [FIPA00037] definiert. Dabei wird der Inhalt der Nachricht insoweit vorgegeben, dass er aus einem Zweitupel zu bestehen hat, dessen erste Komponente die fehlgeschlagene Aktion benennt. Die zweite Komponente des Tupels soll eine Aussage darstellen, die den Fehler begründet.

Die von den beiden Spezifikationen vorgegebenen Rahmenbedingungen gehen aber nicht weit genug, um Fehlernachrichten für den Empfänger garantiert verständlich zu machen. Zum einen treffen beide Spezifikationen über die zu wählende Inhaltssprache keine Aussage, damit ist die Fehlernachricht für den empfangenden Agenten ohnehin in den meisten Fällen unverständlich. Zum anderen ist selbst die Struktur des Inhalts unzureichend beschrieben. Unter der Annahme, beiden Spezifikationen gerecht werden zu wollen, müsste der Inhalt der Nachricht ein Zweitupel sein, dessen zweite Komponente das geforderte `internal-error`-Prädikat ist. Wobei die Frage auftaucht, ob hier nicht sogar ein Widerspruch zwischen den Spezifikationen besteht, denn von der ersten Komponente, die die Aktion beschreibt, wird in der Transport Service Specification gar nicht gesprochen. Woraus sich das Problem ergibt, wie denn die Aktion der Nachrichtenweiterleitung zu beschreiben ist.

Um dennoch sinnvolle Fehlernachrichten versenden zu können, habe ich ein eigenes Format für Fehlernachrichten verwendet, das hier am Beispiel zu sehen ist:

```
(failure
  :sender (agent-identifier :name "ams")
  :receiver (set (agent-identifier :name "test@plat"))
  :ontology (set "FIPA-Agent-Management")
  :language "FIPA-SL0"
  :content "(
    (action
      (agent-identifier :name \"ams\")
      (transport (inform
          :receiver (set (agent-identifier :name \"respond@plat\"))
          :sender (agent-identifier :name \"test@plat\")
          :content \"bla bla bla\")))
    (internal-error \"Unknown agent: respond@plat\"))")
```

Die Nachricht wird entsprechend der Spezifikation im Namen des AMS verschickt. Für Inhaltssprache und Ontologie habe ich die ohnehin für die Verwaltungskommunikation benötigte Sprache SL0 und die FIPA-Agent-Management-Ontologie gewählt. Implizit habe ich damit die Ontologie um die im Inhalt vorkommende Aktionsbeschreibung `transport` erweitert. Die Aktion setzt sich aus dem handelnden Agenten, in diesem Fall wieder stellvertretend das AMS, und der Tätigkeit des Transportierens der ursprünglichen Nachricht zusammen. Auf diese Weise ist die unzustellbare Nachricht in der Fehlermeldung voll enthalten, der Agent kann also genau feststellen, welche seiner Nachrichten ein Problem hatte. Die Fehlerdiagnosemeldung im `internal-error`-Prädikat besteht nur aus einer Zeichenfolge, da viele verschiedene Fehlerbedingungen denkbar sind, die ich nicht alle in Begriffe der Ontologie gießen möchte.

Die Zielgruppe für die Fehlerdiagnose-Zeichenkette sind in meinen Augen die Agenten- oder Plattformentwickler, die beim Testen ihrer Implementierung eine Hilfe brauchen, um Fehler schneller zu finden. Daher soll die Diagnosemeldung aussagekräftiger sein als ein bloßes „Nachricht nicht zustellbar“. Die konkretesten Diagnosen werden von der Transportprotokollimplementierung geliefert. Es kann aber auch passieren, dass die Nachricht unzureichend adressiert wird, so dass gar kein Transportprotokoll ausgewählt werden kann.

Für die ACC-Implementierung aus Abbildung 5.19 bedeutet dass, dass die Fehlermeldungen verschiedener Ebenen mit unterschiedlicher Priorität behandelt werden müssen, weil nach dem letzten fehlgeschlagenen Transportversuch immer auch die Situation auftritt, dass kein passender Transport mehr zu finden ist. Die umfangreichen Anschriften des linken Rücklaufstrangs im ACC-Netz sind im wesentlichen auf die Weiterleitung der Fehlermeldungen aus unteren Ebenen zurückzuführen. Darüber hinaus fügt die Fehlerdiagnoseweiterleitung in der gesamten Weiterleitungsprozedur mindestens ein weiteres Element zum Datentupel hinzu. In der alternativen ACC-

Implementierung aus Abbildung 5.20 fällt der Aufwand für die Fehlerdiagnose sogar noch größer aus.

Aber dieser Aufwand ist es wert, betrieben zu werden, weil dadurch die Fehlersuchzeit während der Nutzung des Dienstes reduziert werden kann.

Integrierte Repräsentierungskonversion. In Abschnitt 5.1.3 ist bereits die Möglichkeit angedeutet worden, während der Nachrichtenweiterleitung eine automatische Umwandlung der Nachrichtenrepräsentierung entsprechend den Anforderungen des ausgewählten Transportprotokolls vorzunehmen. Auf diese Art kann z.B. beim Nachrichtentransport über HTTP eine XML-Darstellung verwendet werden, während über WAP eine Bit-effiziente Repräsentierung zum Zuge kommt. Die Agenten können dann einfach Nachrichten in ihrem gewohnten Format verschicken, unabhängig vom verwendeten Transportprotokoll. Die automatische Umwandlung im Transportdienst erleichtert auch die Weiterleitung von Nachrichten, die über ein bestimmtes Transportprotokoll angekommen sind, über ein anderes Protokoll.

In der ACC-Implementierung von CAPA ist die automatische Umwandlung folgendermaßen integriert: Jeder konkrete Transport verfügt über ein Attribut, in dem die zulässigen Repräsentierungen bei Verwendung dieses Transportprotokolls aufgelistet werden. Die `tryTransport`-Methode der `ACCHelper`-Klasse prüft, ob eine Umwandlung der zu versenden Nachricht für das zu testende Transportprotokoll erstens nötig und zweitens möglich ist. Dabei greift der Transportdienst auf die Funktionalität des Umwandlungsdienstes zurück, der in Abschnitt 5.2.5 beschrieben wird.

Durch die vorangehende Notwendigkeitsprüfung sollen Umwandlungen vermieden werden, wenn die Nachricht bereits in einer passenden Repräsentierung vorliegt. Dies ist insbesondere dann wichtig, wenn verschiedene Repräsentierungen nicht vollständig ineinander überführt werden können – z.B. kann die plattforminterne Repräsentierung beliebige Java-Objekte transportieren, auch solche, für die keine standardisierte Zeichenkettendarstellung definiert ist.

Falls eine Umwandlung nötig ist, aber der Transformationsdienst keine Umwandlung vornehmen kann, wird diese Situation wie ein Fehler des Transportprotokolls behandelt. D.h. der ACC muss dann alternative Transportprotokolle oder Adressangaben ausprobieren. Die zurückgemeldete Fehlerbeschreibung weist auf das Konversionsproblem hin, damit der Agentenentwickler dieses Problem von anderen Netzwerkkommunikationsproblemen unterscheiden kann.

Da die automatische Nachrichtenumwandlung in der `tryTransport`-Methode der Hilfsklasse vorgenommen wird, ist von der Umwandlung in der Netzdarstellung des ACC nichts zu sehen. Das kann als Kritikpunkt gesehen werden, weil ein wesentlicher Schritt bei der Nachrichtenweiterleitung, die Repräsentierungskonversion, versteckt wird.

Wollte man die Umwandlungsprüfung auf der Netzebene modellieren, so müsste die Transition „Try transport“ vergröbert werden. Der sequentielle Charakter der Vergröberung (erst Notwendigkeit prüfen; dann Umwandeln, wenn möglich; abschließend versenden) lässt aber die Verwendung von Netzen unnötig erscheinen, da die Stärken der Netze wie Nebenläufigkeit und Synchronisation gar nicht zum Tragen kommen. Zudem muss die Nachricht im Originalzustand erhalten bleiben, um sie im Fehlerfall unverfälscht dem nächsten auszuprobierenden `Transport` vorlegen zu können. In der seiteneffektfreien Java-Methode wird die Originalnachricht nur als Kopiervorlage für die umgewandelte Nachricht verwendet – im Netz müsste dieses Vorgehen explizit durch Aufnahme der zu versenden Kopie in das Datentupel modelliert werden.

Insgesamt würde die Netzgrafik durch das Verschieben der Umwandlungsfunktionalität auf die Netzebene deutlich komplizierter und unübersichtlicher werden und den Blick auf den wesentlichen Kontrollfluss im ACC verstellen. Daher halte ich das Verstecken dieser Funktionalität im ACC-Netz für vertretbar.

Vielleicht wäre eine zusätzliche, innerhalb der Transportschleife sequentiell vor der „Try transport"-Transition angeordnete „Prepare message"-Transition ein guter Kompromiss, um sowohl das Verstecken als auch die Unübersichtlichkeit zu vermeiden. Die Vergrößerung des Datentupels fände dann freilich trotzdem noch statt. Aber die Umsetzung dieser Idee muss auf später warten.

5.2.3.2 Internes Transportsystem

Der interne Transportdienst ist für die Vermittlung von Nachrichten zwischen den lokalen Agenten und dem zentralen Transportsystem zuständig. Daher berührt er die bereits in Abschnitt 5.1.1 ausführlich diskutierte interne Schnittstelle zum Agenten, also eines der wichtigsten Elemente der Plattform.

Im Folgenden sollen zunächst die sich aus den Beziehungen des internen Transportdienstes zu den Agenten und dem zentralen Transportsystem ergebenden Rahmenbedingungen umrissen werden. Daran schließt eine Beschreibung der Arbeitsweise des Dienstes an, gefolgt von einer Diskussion einiger spezieller Fragestellungen, die im Zusammenhang mit der Implementierung auftreten.

Die Rahmenbedingungen. Die in CAPA definierte Schnittstelle zwischen Agenten und Nachrichtentransportsystem entspricht im wesentlichen der bereits in der MULAN-Architektur verwendeten Schnittstelle. Daraus ergibt sich folgendes:

- An die Schnittstelle schließt ein *Referenznetz* anstelle einer *Java-Klasse* an, weil die Schnittstelle mittels synchroner Kanäle definiert ist. Die Implementierung eines Kanal-Downlinks ist zwar auch in Java möglich, setzt aber Kenntnisse des Renew-Simulators voraus und bringt die in Abschnitt 5.1.1 diskutierte aufwändige Threadimplementierung mit sich.

- In MULAN schließt das Plattformnetz direkt an die Schnittstelle an, d.h. das Plattformnetz stellt nebst anderen Diensten die Nachrichtentransportfunktionalität bereit. In CAPA sind die anderen Dienste ausgegliedert in eigene Plattformelemente. Zum Beispiel überwacht und verwaltet der AMS-Agent in Zusammenarbeit mit dem Plattformagenten die Lebenszyklen der Agenten (siehe Abschnitt 5.2.2). Das CAPA-Netz für die interne Transportschnittstelle ist also von den zusätzlichen Aufgaben der MULAN-Plattform befreit, muss sich dafür aber ggf. mit den ausgegliederten Plattformelementen synchronisieren.

- Die FIPA-Spezifikationen sehen asynchronen Nachrichtentransport vor. Zwar lässt sich – wie in MULAN – der asynchrone Transport auch durch Ein- und Ausgangspuffer im Agentennetz modellieren. Aber da die MULAN-Agentennetze durch andere Implementierungen ersetzt werden dürfen, kann die Plattform sich darauf nicht mehr verlassen.

- Der interne Teil des Transportdienstes muss an den zentralen `TransportService` der Plattform angeschlossen werden. Es müssen sowohl Nachrichten von lokalen Agenten an den `TransportService` weitergeleitet als auch von dort aus den lokalen Agenten zugestellt werden können. Das `Transport`-Interface bietet genau diese Kommunikationsmöglichkeiten. Daher wäre es zweckmäßig, dieses Interface für die Kommunikation zwischen der internen Transportschnittstellenimplementierung und dem zentralen `TransportService` zu benutzen.

- Die MULAN-Plattform hat zwischen lokaler und plattformübergreifender Kommunikation unterschieden. Aber wie bereits gegen Ende des Abschnitts 5.1.2 angedeutet, kann die lokale Kommunikation auch als Spezialfall der allgemeinen Kommunikation über den zentralen Transportdienst abgewickelt werden. Damit ist die Unterscheidung der Kommunikationswege zunächst nicht nötig.

Die Arbeitsweise der Implementierung. Um die Struktur des internen Transportdienstnetzes, im folgenden `InternalMTS` genannt, zu entwickeln, bietet sich als Ausgangspunkt die prototypische Schnittstellenimplementierung an, wie sie bereits im Abschnitt 5.1.1 vorgestellt wurde. Betrachtet man die Abbildung 5.5 (S. 75), so stellt die obere Hälfte den Grundaufbau des internen Transportdienstes dar. Rechts werden ausgehende Nachrichten der Agenten entgegengenommen, links werden eingehende Nachrichten zugestellt. In der Mitte stehen Referenzen auf alle an den Transportdienst angeschlossenen lokalen Agenten in Form von Marken auf der Stelle „active agents“ zur Verfügung. Darüber modelliert die ohne Füllung dargestellte Transition die Anbindung zum zentralen Transportdienst auf einer abstrakten Ebene.

In Abbildung 5.21 ist die verfeinerte Grundstruktur zu sehen, in der alle ein- und ausgehenden Nachrichten der Agenten vom bzw. zum zentralen Transportdienst durchgereicht werden. Als vollständige, funktionstüchtige Implementierung des `Transport`-Interfaces des zentralen Transportsystems taugt dieses Netz noch nicht, da nur die unmittelbar am Nachrichtentransport beteiligten Elemente dargestellt sind.

Auf der linken Seite werden ausgehende Nachrichten der Agenten an den ACC weitergeleitet.[6] Die orangefarbene Transition nimmt die Nachricht in der plattforminternen Repräsentierung vom Agenten entgegen und erzeugt automatisch den für den

[6] Die Laufrichtung der Nachrichten in diesem Netz ist entgegengesetzt der Laufrichtung aus Abbildung 5.5, damit die Nachrichten der westlichen Lesegewohnheit folgend von links nach rechts befördert werden. In Abbildung 5.5 hatte ich die andere Richtung gewählt, um die Kanäle zu den Ein- und Ausgangstransitionen des Agenten darstellen zu können.

Transport notwendigen Nachrichtenumschlag. Das Paar aus Nachricht und Umschlag wird kurz zwischengepuffert, dann geklont und an den zentralen `TransportService` (dessen Referenz wird in der Stelle `ACC` bereitgehalten) weitergereicht.

Auf der rechten Seite der `InternalMTS`-Grundstruktur ist die Zustellung eingehender Nachrichten an die lokalen Agenten zu sehen. Die Nachricht wird entgegengenommen, abermals geklont (falls die Nachricht über einen anderen Transportdienst verschickt wurde, der nicht klont), und dann mittels der orangefarbenen Transition dem im Nachrichtenumschlag adressierten Agenten zugestellt.

Die Zustellung soll gemäß dem `Transport`-Interface erfolgen, daher muss eine Rückmeldung über den Erfolg oder Misserfolg der Zustellung gegeben werden. Dies übernehmen die blassblauen Elemente: War die Zustellung erfolgreich, so landet eine leere Fehlermeldung in der virtuellen Kopie der Stelle „Status“. Ist hingegen der in der Nachricht adressierte Agent nicht lokal bekannt (zunächst als abstrakte Bedingung „*agent unknown*“ ausgedrückt), so wird diese Tatsache als Fehlermeldung in die „Status“-Stelle gelegt. Damit die Rückmeldung der eigentlichen Nachricht wieder zugeordnet werden kann, wird während des gesamten Zustellungsprozesses ein identifizierendes Objekt mitgeführt, wie es bei der Interface-Implementierung durch Renew-Netz-Stubs üblich ist.

Unten in der Mitte der `InternalMTS`-Grundstruktur befindet sich die Stelle „Active agents“. Hier werden Referenzen auf alle lokalen Agenten bereitgehalten, die in der Lage sind, Nachrichten zu versenden oder zu empfangen. Falls ein Agent seinen Lebenszykluszustand ändert, wird er von der Plattformverwaltung (siehe Abschnitt 5.2.2) mittels der Kanäle `:addAgent` und `:removeAgent` an den Nachrichtentransportdienst angeschlossen oder von ihm abgekoppelt.

Das Nachrichtenobjekt. Da der Agent den Umschlag nicht selbst erzeugt und gezwungenermaßen die interne Nachrichtenrepräsentierung verwenden muss, verliert er eine Reihe von Einflussmöglichkeiten auf den Nachrichtentransport, die er gemäß den FIPA-Spezifikationen haben könnte (aber nicht muss). Der Vorteil dieser Maßnahme ist eine starke Vereinfachung der Agentenmodellierung während der Test- und Entwicklungsphase von CAPA, solange noch keine plattformübergreifende Kommunikation stattfindet. In einer späteren Weiterentwicklung sollen parallel zu den existierenden Versand- und Empfangstransitionen weitere Transitionen bereitgestellt werden, die dem Agenten die Kontrolle über den Umschlag und die Nachrichtenrepräsentierung zurückgeben.

Beim Weiterreichen an den ACC wird die Nachricht kopiert, um verdeckte Informationkanäle zwischen den Agenten zu behindern. Da die interne Nachrichtenrepräsentierung den Transfer beliebiger Java-Objekte zulässt, besteht die Möglichkeit, ein modifizierbares Objekt an einen anderen Agenten zu versenden und dann über die Modifikationen des Objektes Informationen zwischen den Agenten unter Umgehung des Nachrichtentransportdienstes auszutauschen. Dasselbe Phänomen kann beim Nachrichtenrepräsentierungsobjekt selber auftreten, da dessen Implementierung ebenfalls Modifikationen am Inhalt zulässt (siehe Abschnitt 5.2.4). Um zu vermeiden, dass ein

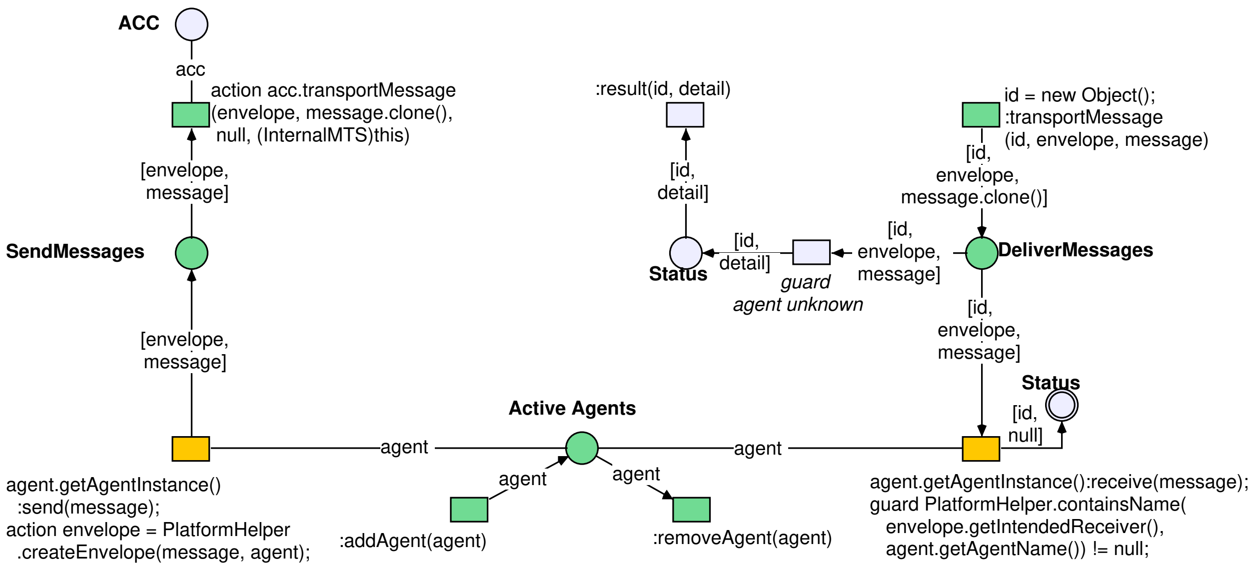


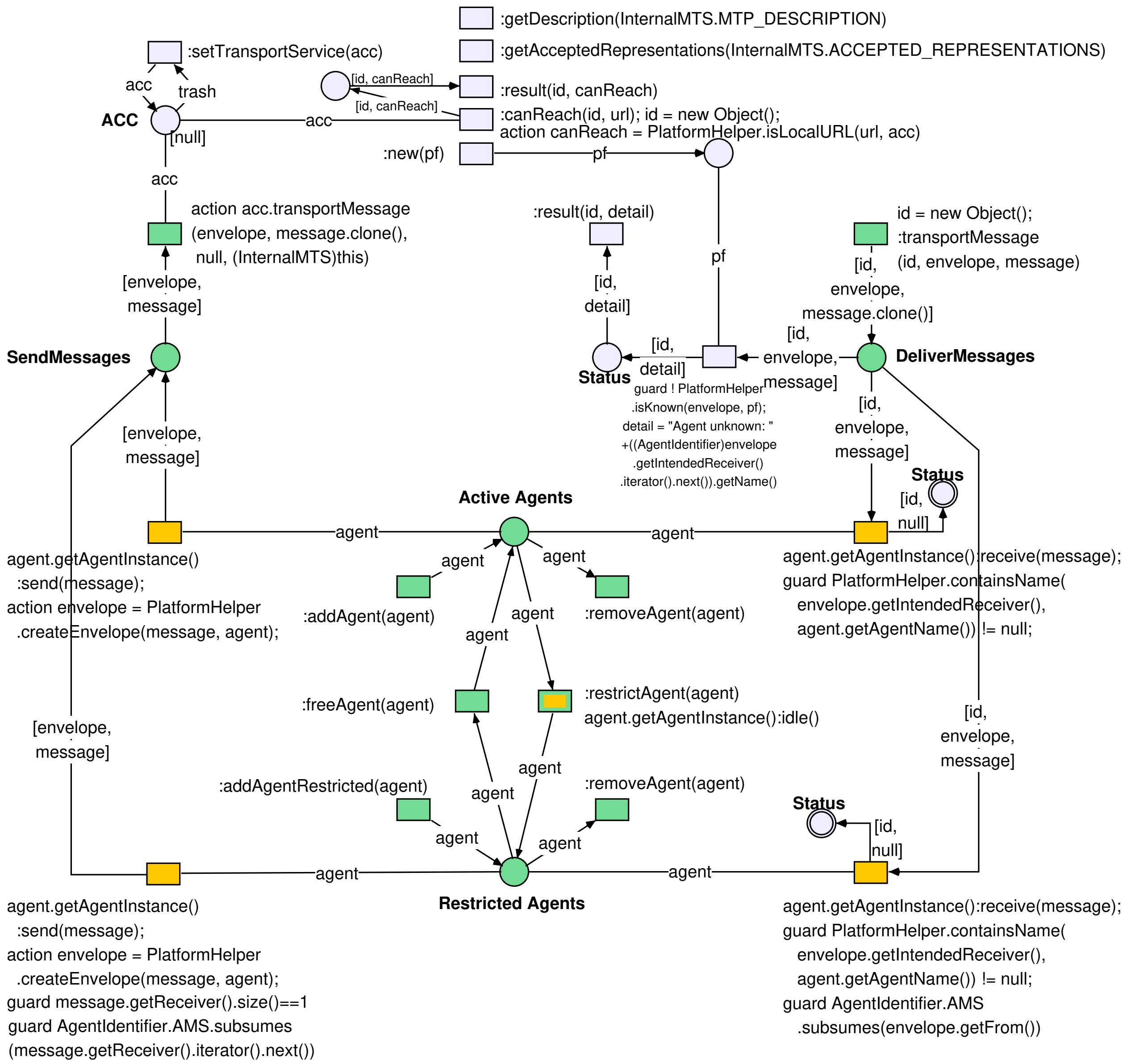
Abbildung 5.21: Das Netz `InternalMTS` (Grundprinzip)

Agent die versendete Nachricht – willentlich oder unwissentlich – nach erfolgtem Versand noch ändert, wird das Nachrichtenobjekt hier im internen Transportdienst geklont und lediglich der Klon weitergereicht.

Leider bewirkt ein Klonen des Nachrichtenobjekts nicht das Klonen der Objekte, die den Nachrichteninhalt bilden. Daher kann ein verdeckter Informationsaustausch dennoch zustande kommen. Absolute Sicherheit gibt es hier nur, wenn die Nachricht zwangsweise in eine andere Repräsentierung überführt wird, die alle Referenzen auflöst, wie z.B. eine reine Zeichenkettendarstellung. Diesen Aufwand wollte ich aber in der Entwicklungsphase der Plattform noch nicht betreiben.

Zustellung der Nachrichten an *einen* Empfänger. Die Nachrichtenzustellung durch den internen Transportdienst erfolgt nach dem 1:1-Prinzip: Jede Nachricht wird genau einem Agenten zugestellt. Daher ist es notwendig, dass die Nachrichtenumschläge nicht mehr als einen Empfänger enthalten, wenn sie dem `InternalMTS`-Netz übergeben werden. Die im vorigen Abschnitt 5.2.3.1 vorgestellte ACC-Implementierung stellt aber nur sicher, dass alle Empfänger in einem Umschlag dieselbe Adresse haben (was z.B. bei einer Nachricht an mehrere lokale Agenten auf alle Empfänger zutrifft).

Um die Ein-Empfänger-pro-Umschlag-Aufteilung sicherzustellen, wird die zentrale ACC-Implementierung um die Möglichkeit ergänzt, für speziell gekennzeichnete Transporte die Umschläge entsprechend weiter aufzusplitten. Diese Funktionalität ist im Java-Code der `tryTransport`-Methode der `AccHelper`-Klasse untergebracht. Dort wird ein zusätzliches Attribut der `transport-description` des Transports ausgewertet, welches den Namen `x-de.renew.agent.transport.local` trägt und damit eine proprietäre Ergänzung der FIPA-Agent-Management-Ontologie darstellt.




```
import de.renew.agent.repr.acl.MessageEnvelope;      Agent agent;
import de.renew.agent.repr.acl.AclMessage;           AclMessage message;
import de.renew.agent.repr.acl.AgentIdentifier;      Tuple[] deliveries;
import de.renew.agent.platform.Agent;                MessageEnvelope envelope;
import de.renew.agent.platform.InternalMTS;          TransportService acc;
import de.renew.agent.platform.PlatformHelper;       Object id, trash;
import de.renew.agent.platform.Platform;             String detail;
import de.renew.agent.transport.TransportService;    URL url;
import de.renew.unify.Tuple;                         Platform pf;
import java.net.URL;                                 boolean canReach;
```

Abbildung 5.22: Das Netz `InternalMTS` (vollständiger Quellcode)

Der Einbau einer Spezialbehandlung für den lokalen Transportdienst in den allgemeinen zentralen Transportdienst ist ein unschönes Konstrukt, das eventuell einer Korrektur bedarf. Die Gründe für diese Maßnahme lassen sich alle auf den Nenner zurückführen, dass die Implementierung einfacher scheint: Zum ersten ist die Aufsplittungsfunktionalität schon im ACC vorhanden und kann wiederverwendet werden. Und zum zweiten entstehen im internen Transportdienst keine Probleme mit der durch eine Aufsplittung verursachten n:1-Beziehung zwischen zugestellten Nachrichten und der Rückmeldung an den Transportdienst.

Der erste Grund lässt sich einfach aushebeln, weil die `AccHelper`-Klasse nur eine Funktionsbibliothek ist. Die Funktion zum Aufsplitten von Nachrichten kann also auch an anderer Stelle untergebracht werden, wo sie mehr Netzen und Klassen zur Verfügung steht als nur dem ACC. Der zweite Grund ist ebensowenig tragfähig, weil dasselbe Problem mit der n:1-Beziehung jetzt in der `tryTransport`-Methode des ACC auftritt. Der einzige Grund, der tatsächlich für eine Unterbringung der Sonderbehandlung im ACC sprechen könnte, ist die Möglichkeit, dass auch andere `Transport`-Implementierungen Nutzen daraus ziehen könnten. Nur dass mir leider kein Beispiel einfällt, bei dem die Ein-Empfänger-pro-Umschlag-Aufteilung Vorteile gegenüber der ohnehin realisierten Eine-Adresse-pro-Umschlag-Aufteilung bringt.

Die Überarbeitung der Funktionalitätszuordnung wird aber nicht mehr im Rahmen dieser Diplomarbeit stattfinden.

Vervollständigung der `Transport`-Implementierung. In Abbildung 5.22 ist die vollständige Implementierung des internen Nachrichtentransportdienstes durch das Netz `InternalMTS` dargestellt. Im Vergleich zur bisher besprochenen vereinfachten Version sind zwei Funktionsblöcke hinzugekommen: Die blassblauen Elemente im oberen Bereich stellen die für eine Implementierung des `Transport`-Interfaces notwendige Verwaltungsfunktionalität bereit. Diese Funktionalität wird auch bei dem im folgenden Abschnitt 5.2.3.3 vorgestellten externen Transport benötigt, daher sei die Erläuterung dieses Netzteils bis dahin aufgeschoben.

Restriktion der Kommunikation. Der zweite neue Block im unteren Bereich des Netzes ist ein Duplikat der bereits in der Grundstruktur beschriebenen Implementierung der internen Transportschnittstelle. Die orangefarbenen Transitionen des duplizierten Bereichs unterscheiden sich von den Originalen in jeweils einer zusätzlichen Anschrift, welche die Adressierung der Nachrichten beschränkt. Im unteren Bereich können ausschließlich solche Nachrichten versendet und empfangen werden, die entweder an das AMS adressiert sind oder vom AMS versendet wurden. Entsprechend lautet der Name der Stelle, in der die bekannten Agenten vorgehalten werden, „Restricted agents“.

Damit erweitern sich die Möglichkeiten der Plattformverwaltung, Agenten an den Nachrichtentransportdienst anzukoppeln. Die beschränkte Anbindung der Agenten an den Nachrichtentransport ist dann von Interesse, wenn ein Agent sich in einem Lebenszykluszustand befindet, in dem er eigentlich nicht kommunizieren kann oder darf, aber trotzdem mit dem AMS aushandeln können muss, wann und wie er diesen Zustand wieder verlassen will.

Insbesondere solche Lebenszykluszustandsübergänge, die von der unbeschränkten zur beschränkten Kommunikation übergehen, bedürfen besonderer Beachtung. Das dabei auftretende Problem ist, dass der Agent sich eventuell noch an Konversationen beteiligt, die plötzlich unterbrochen werden würden, wenn er auf Anweisung der Plattformverwaltung vom Nachrichtentransport abgeklemmt wird. Um eine Kooperation zwischen Agent und Plattformverwaltung zu ermöglichen, ist dieser Zustandsübergang im `InternalMTS`-Netz mit dem zusätzlichen synchronen Kanal `:idle` versehen, der die interne Schnittstelle zwischen Plattform und Agent erweitert. Wie Agent und Plattform diesen `:idle`-Kanal verwenden, ist in den Abschnitten 5.2.1 und 5.2.2.1 bereits angerissen worden.

5.2.3.3 Ein einfacher externer `Transport`

Um die Kommunikationsfähigkeit von CAPA austesten zu können, ohne sich mit dem komplexen Aufbau eines der von der FIPA ausgewählten Netzwerkprotokolle wie HTTP, WAP oder IIOP herumschlagen zu müssen, ist das in Abbildung 5.23 dargestellte Transportprotokoll in Zusammenarbeit mit Heiko Rölke entstanden.

Als Implementierungssprache haben wir Referenznetze gewählt, weil mit dem animierten Markenspiel des Renew-Simulators eine einfache Debugging-Möglichkeit zur Verfügung steht. Unter Effizienz-Gesichtspunkten sollten externe Transportprotokolle, zumal der Ablauf beim Versenden und Empfangen einer Nachricht meist sequentiell ist, in Java implementiert werden. Der Java-Aufwand bei der Implementierung nebenläufiger Versand- und Empfangsprozesse ist relativ gering, weil der zentrale Transportdienst für jede zu versendende Nachricht einen Thread zur Verfügung stellt. Die Transportprotokollimplementierung braucht also nur noch zum Empfangen von Nachrichten von anderen Plattformen einen Thread zu erzeugen, der dann unter Zuhilfenahme der in der Java-Klassenbibliothek bereitgestellten Netzwerkfunktionalität auf ankommende Nachrichten warten kann.

Entsprechend einfach sehen die beiden sequentiellen Stränge aus, mit denen im Netz TcpIpMTS Nachrichten versandt (rechte Seite) und empfangen werden (linke Seite). Lediglich an der Transition, die auf eingehende Nachrichten von außerhalb wartet („await connection“), findet in dem Moment, in dem eine Nachricht ankommt, eine Abspaltung eines „Threads“ statt: Während die Transition „await connection“ sofort wieder schalten kann, um auf die nächste Nachricht zu warten, wandert ein frisch erzeugtes Token durch die Empfangssequenz.

Die Netzwerkkommunikation erfolgt mittels einer einfachen TCP-Verbindung, über welche ein Paar aus Nachrichtenumschlag und Nachricht in Zeichenkettendarstellung übermittelt wird. Beim Versand ergibt sich das andere Ende der TCP-Verbindung aus der URL, die der zentrale Transportdienst ermittelt und an dieses Transportprotokoll übergeben hat. Am anderen Ende der Verbindung steht wiederum eine TcpIpMTS-Implementierung, welche auf Nachrichten wartet. Weitere Details bezüglich der Übertragung sollen hier nicht weiter interessieren.

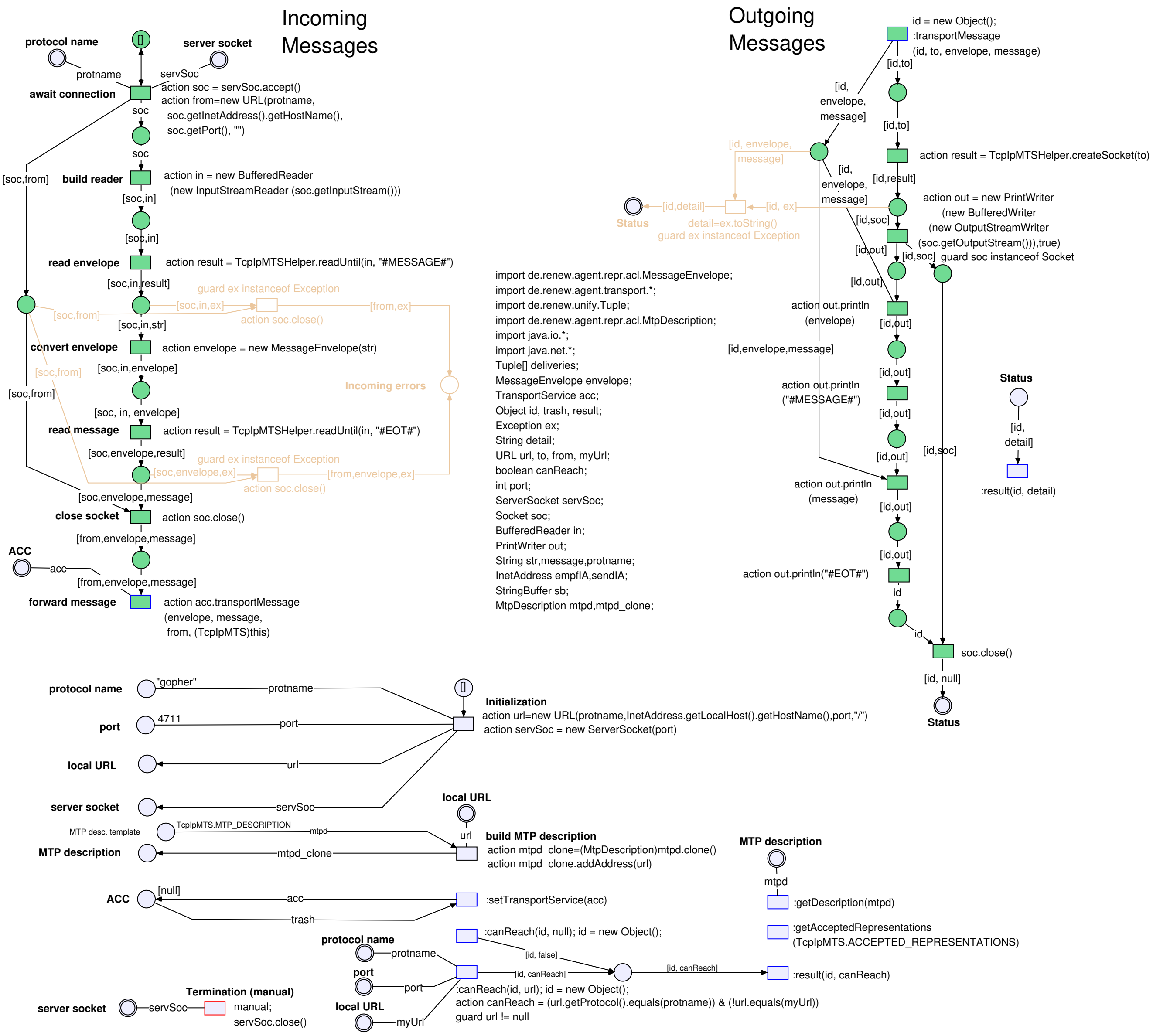


Abbildung 5.23: Das Netz TcpIpMTS (vollständiger Quellcode)

Allerdings können sowohl beim Versand als auch beim Empfang einer Nachricht jederzeit Fehler auftreten, die irgendwie behandelt werden müssen. Exemplarisch sind durch die blassbraunen Kanten und Stellen im Netz einige Fehler-Abfangmöglichkeiten angedeutet.[7] Falls Fehler beim Empfang einer eingehenden Nachricht auftreten, kann der Transportdienst nur hoffen, dass der Absender der Nachricht den Fehler ebenfalls bemerkt hat, eine weitere Behandlung der Nachricht (so es denn überhaupt eine war) vor Ort macht keinen Sinn.

Im Falle des Nachrichtenversands ist es wichtig, die aufgetretenen Fehler an den zentralen Transportdienst zurückzumelden, damit dieser das nächste Transportprotokoll bzw. die nächste Adresse ausprobieren kann. Die Rückmeldung erfolgt – wie auch schon beim internen Transportdienst (siehe Abschnitt 5.2.3.2) beschrieben – über die Stelle „Status“, wobei jede Rückmeldung mittels eines Identifikationsobjekts dem jeweiligen Aufruf wieder zugeordnet werden kann.

Um alle Anforderungen der durch das Java-Interface `Transport` definierten Schnittstelle zum zentralen Transportdienst zu erfüllen, sind über den Nachrichtenversand und -empfang hinaus einige Funktionen notwendig, die von den blassblauen Netzelementen implementiert werden. Diese allgemeinen Funktionen sind für jeden Transport erneut zu implementieren. Dazu gehören zum einen die bereits erwähnte Rückmeldung des Erfolgsstatus beim Versenden von Nachrichten. Ferner muss der Transport initialisiert werden und eine Referenz auf den zentralen Transportdienst zwecks Ablieferung eingehender Nachrichten übergeben bekommen.

Der zentrale Transportdienst muss wiederum eine FIPA-konforme Beschreibung des implementierten Transportprotokolls abfragen können, in der auch die URL steht, unter der der Transport für andere Plattformen erreichbar ist. Um dem zentralen Transportdienst die Auswahl eines passenden Transports für eine bestimmte Nachricht zu ermöglichen, müssen dessen Fragen beantwortet werden können. Die Fragen betreffen die Erreichbarkeit einer bestimmten Adresse (im wesentlichen hängt die Antwort davon ab, ob das Protokoll stimmt) und die versandfähigen Nachrichtenrepräsentierungen (in diesem Fall ausschließlich die Zeichenkettendarstellung).

Die rot umrandete Transition löst ein technisches Problem: Die Agentenplattform läuft zur Zeit innerhalb der Renew-Simulationsumgebung, welche zum Einbringen neuer Agenten- und Protokollnetze beendet und neu gestartet werden muss. Wird die Simulation beendet, ohne die gesamte Renew-Entwicklungsumgebung zu beenden, so terminiert der Schaltvorgang der „await connection“-Transition nicht. Als Folge bleibt die Netzwerkverbindung geöffnet. Abhilfe schafft die rot umrahmte, manuell vor Simulationsende zu schaltende Transition „Termination“, welche die Netzwerkverbindung schließt.

Auch wenn das Problem hier durch das unsaubere Ende der Renew-Simulationsumgebung zu Tage tritt, so offenbart sich dennoch ein Ansatzpunkt für eine Verbesserung

[7] Tatsächlich können Fehlersituationen an so ziemlich jeder Transition der Empfangs- und Versandsequenz auftreten. Die explizite Behandlung aller Fehlermöglichkeiten, die zudem noch durch die Tatsache erschwert ist, dass Renew-Netze bisher keine Java-Exceptions abfangen können, hätte zuviel Aufwand erfordert und das Netz zu unübersichtlich gemacht. Hier stellt der in Java verfügbare `try-catch`-Block einen deutlichen Vorteil gegenüber den Netzen dar.

von CAPA: Eine Nachrüstung eines expliziten Mechanismus zum Beenden von Transportprotokollen würde nicht nur beim Simulationsende helfen, sondern auch die zwischenzeitliche Deaktivierung von Transporten zulassen. Die angedachte Möglichkeit, zur Laufzeit die Kommunikationsverbindungen der Plattform umzukonfigurieren, ist ohne eine definierte Abschaltung von einzelnen Transporten nicht einsetzbar. Daher soll ein entsprechendes Konzept sobald als möglich nachgerüstet werden.

5.2.4 Interne Nachrichtenrepräsentierung

Eine interne Nachrichtenrepräsentierung als Bestandteil von CAPA ist zwar zur Implementierung einer FIPA-konformen Plattform nicht notwendig, erleichtert aber die Arbeit der zukünftigen Agentenentwickler. In Zusammenarbeit mit dem im folgenden Abschnitt 5.2.5 vorgestellten Transformationsdienst wird den Agenten durch die interne Repräsentierung die Arbeit abgenommen, standardkonform repräsentierte Nachrichten von Grund auf zu erstellen oder zu parsen.

Im Rahmen der Plattformimplementierung werden mit dem Agent Management System (AMS) und dem Directory Facilitator (DF) schon die ersten zwei Agenten erstellt, die später in der von der FIPA definierten „Agent Management Ontology“ kommunizieren können müssen. Daher bietet es sich an, zumindest für die Repräsentierung der Verwaltungsontologie und der Inhaltssprache SL0 geeignete Strukturen bereitzustellen.

In der abstrakten Architektur der FIPA wird das Konzept des Key-Value-Tupels (KVT, siehe Abschnitt 4.3.1) eingeführt. Dabei handelt es sich um eine sehr flexible Struktur zur Aufnahme verschiedenster Informationen. Mit KVTs werden all jene Konstrukte beschrieben, die in der abstrakten Architektur zur Verwaltung der Plattform und als Nachrichtenbausteine benötigt werden. In der FIPA2000-Suite dient die „Agent Management Ontology“ dem gleichen Zweck und verwendet sehr ähnliche Strukturen. Es bietet sich daher an, die KVT-Idee für vorgefertigte Nachrichtenelemente zu übernehmen und in eine Java- oder Netzimplementierung zu überführen.

Als Implementierungssprache habe ich in diesem Fall Java gewählt, und zwar aus mehreren Gründen: Zum ersten halte ich Datenhaltung und -manipulation (letztendlich geht es ja „nur“ um strukturierte Datensätze) in Java für einfacher und effizienter. Zum zweiten sind die zu definierenden abstrakten Datentypen in Java durch Interfaces gut kapselbar und verfügen über wohldefinierte, vom Compiler überprüfbare Schnittstellen. Dies kommt meiner Idee entgegen, die interne Repräsentierung mit einer integrierten Typ- oder Plausibilitätsprüfung zu versehen. Zum dritten erwarte ich keine Nebenläufigkeits- oder Synchronisationsprobleme, weil die Nachrichten immer nur von einem Agenten zur Zeit erstellt oder interpretiert werden.

Beim praktischen Einsatz der Plattform hat sich aber ergeben, dass doch Nebenläufigkeitsprobleme auftreten: Dadurch, dass in einem Agenten mehrere Konversationen zugleich ablaufen, können z.B. in der Wissensbasis nebenläufige Zugriffe auf ein dort abgelegtes Nachrichtenelement stattfinden. Zudem besteht – wie schon bei der Beschreibung des internen Nachrichtentransportdienstes in Abschnitt 5.2.3.2 festgestellt

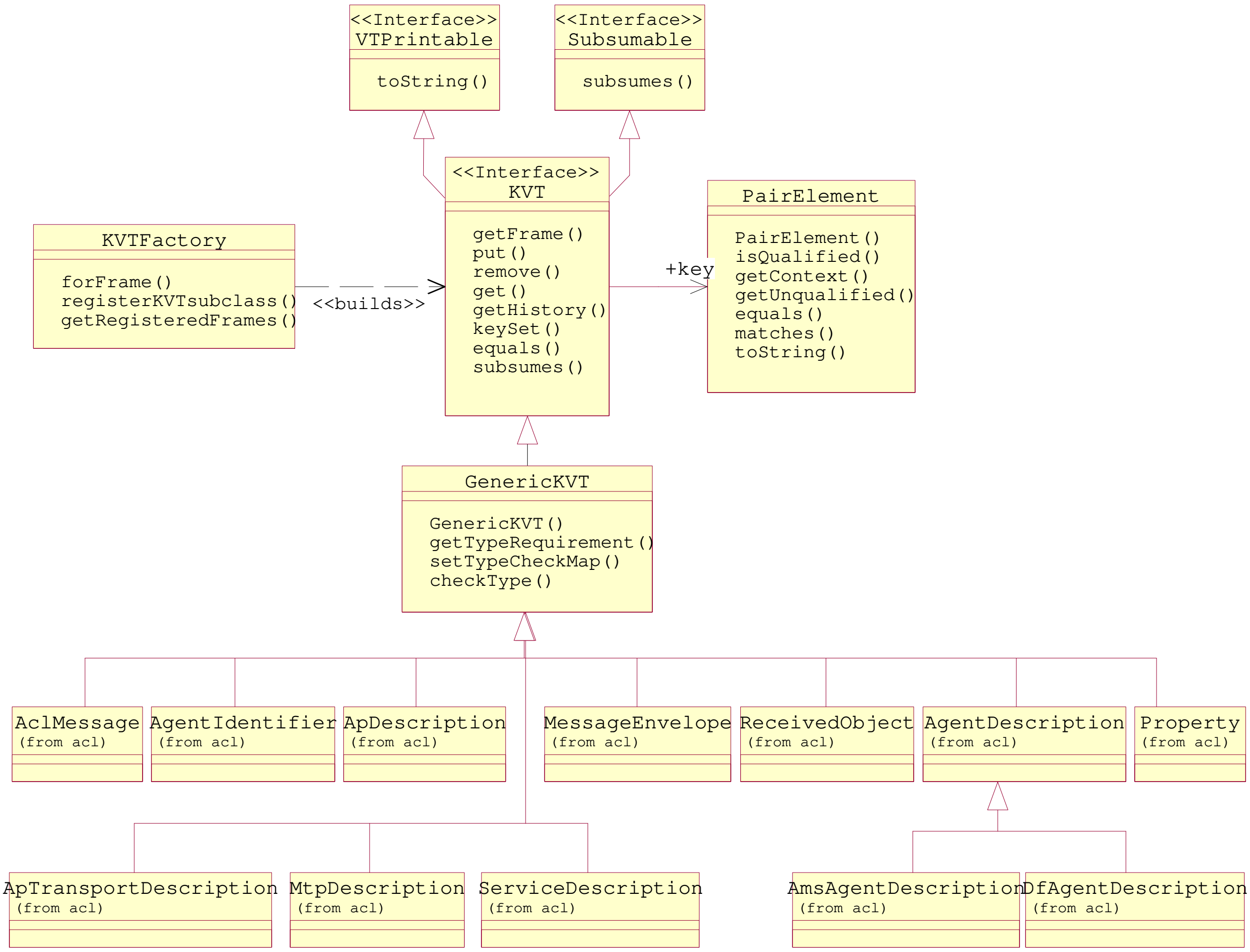


Abbildung 5.24: Die Klassen zur Repräsentierung der ACL-Nachrichten und der Agent Management Ontology

– das Risiko, dass ein Nachrichtenobjekt auch nach dem Versand noch vom Agenten referenziert und damit modifiziert werden kann.

Die Implementierung der Java-Interfaces durch Netze wäre wahrscheinlich relativ einfach möglich, zumal die Technik, Tupel aus Schlüssel und Wert in einer Stelle abzulegen, auf einfachste Weise zu einem KVT-ähnlichen Verhalten führt. Dennoch halte ich vorerst an der Java-Implementierung fest. Die Nebenläufigkeitsprobleme treten nur dann auf, wenn Agenten bzw. Protokolle in Agenten auf unschöne Weise mit Nachrichtenelementen umgehen und verdeckte Kanäle nutzen. Die Forderung, seitens der Agentenentwickler auf solche Tricksereien zu verzichten, macht also auch ohne die Nebenläufigkeitsproblematik Sinn.

Wie in Abbildung 5.24 zu sehen, wird jeder Frame (in den FIPA-Spezifikationen steht „Frame“ für Begriffe oder Bausteine, also inhaltlich zusammenhängende Strukturen) der Ontologie oder ACL-Nachricht durch eine eigene Klasse repräsentiert. Alle diese Klassen implementieren das allgemeine Interface `KVT` und erben die Funktionalität einer generischen KVT-Implementierung. Zu dieser Funktionalität gehört in erster Linie das Setzen und Abfragen von Informationen wie Frame, Schlüsseln und Werten.

Die Klassen für die einzelnen Elemente der Ontologie bzw. Nachricht stellen über die geerbte Grundfunktionalität hinaus nur noch die Namen von Frame und standardisierten Schlüsseln sowie Bequemlichkeitsmethoden zum Setzen und Abfragen der Werte zu den Schlüsseln zur Verfügung. Die Bequemlichkeitsmethoden sind entsprechend des Wertebereichs typisiert, so dass z.B. der `sender`-Slot der `AclMessage` über diese Methoden ausschließlich mit einem `AgentIdentifier`-Objekt gefüllt werden kann.

Neben den typisierten Bequemlichkeitsmethoden der spezialisierten Klasse ist aber immer auch der untypisierte Zugriff durch die generische Funktionalität der Oberklasse erlaubt, weil die FIPA die Verwendung nicht-standardisierter Schlüssel mit beliebigen Werten zulässt. Um zu verhindern, dass durch die generischen Methoden unzulässige Werttypen für einen standardisierten Schlüssel verwendet werden, habe ich die Klasse `GenericKVT` um eine Typprüfung für Werte ergänzt, die durch die spezialisierte Unterklasse konfiguriert werden kann.

Objekte der Klasse `PairElement` dienen als Schlüssel in KVTs. Die Klasse kapselt die in der abstrakten Architektur der FIPA beschriebene Struktur der Schlüssel (siehe Abschnitt 4.3.1) und stellt grundlegende Funktionen für Umwandlungen und Vergleiche zwischen qualifizierten und unqualifizierten Paar-Elementen zur Verfügung. Jede spezialisierte `KVT`-Klasse soll ihren standardmäßigen Kontext kennen und einbringen, so dass die abgekürzte Schreibweise für Schlüssel verwendet werden kann. Die abgekürzte Schreibweise passt auch auf die Bezeichnungen der Frames aus der FIPA2000-Suite, so dass die Kompatibilität zwischen beiden Architekturen in diesem Punkt sichergestellt ist.

Java verlangt, dass für alle Objekte eine Äquivalenzrelation definiert wird, welche mittels der Methode `equals` abgefragt werden kann. Die Java-Standardimplementierung definiert die feinstmögliche Äquivalenzrelation, welche der Identität von Objekten entspricht. In eigenen Klassen kann eine gröbere Relation realisiert werden, welche vom Objektzustand (oder Teilen davon) abhängen darf. Dabei ist auch eine zeitliche Änderung der Relation zulässig, wenn sich der Zustand eines Objektes wesentlich ändert.

Für KVTs habe ich eine inhaltsabhängige Äquivalenzrelation gewählt: Zwei `KVT`-Objekte sind gleich, wenn sie den selben Frame repräsentieren und die gleichen Schlüssel mit gleichen Werten enthalten. Darüber hinaus habe ich eine Subsumptionsrelation vorgesehen und implementiert, die einen einfachen Mustervergleich ermöglicht: Ein `KVT`-Objekt subsumiert ein anderes, wenn beide KVTs denselben Frame repräsentieren und alle Schlüssel aus dem ersten KVT im subsumierten KVT ebenfalls vertreten sind, wobei die beiden Werte zu einem Schlüssel sich ebenfalls (in derselben Richtung) subsumieren müssen. An einigen Stellen in dieser Arbeit habe ich bereits darauf hingewiesen, wo sich die Subsumptionsrelation nützlich einsetzen lässt, z.B. bei der Zuordnung von Protokollen zu eingegangenen Nachrichten in der Protokollfabrik des Agenten (siehe Abschnitt 5.2.1).

Für die Kommunikation mit den FIPA-Verwaltungsagenten wird die „Agent Management Ontology“ eingebettet in die Inhaltssprache „SL0“ verwendet. Daher sollten auch die Elemente dieser Sprache exemplarisch durch Repräsentierungsklassen dargestellt werden. Die meisten SL0-Elemente lassen sich durch ein dem KVT-Konzept sehr ähnliches Konzept beschreiben. Dieses Konzept ist das benannte Tupel, das wie ein KVT durch einen Frame-Bezeichner typisiert wird und die folgenden Werte nicht anhand von Schlüsseln, sondern anhand ihrer Position im Tupel identifiziert. Daher habe ich parallel zum `KVT`-Interface ein `VT`-Interface (VT steht für „Value-Tuple“, eben ohne „Key-“) nebst entsprechend übertragener Klassenstruktur definiert.

Auch VTs implementieren die inhaltsabhängigen Subsumptions- und Äquivalenzrelationen der KVTs. Allerdings muss für VTs durch den Verzicht auf Schlüssel eine neue Regel gefunden werden, welche Werte des einen Tupels beim Äquivalenz- oder Subsumptionsvergleich zu welchen Werten des anderen Tupels passen müssen. Möglich ist sowohl eine Mengensicht als auch eine Listensicht, bei der ersten spielt die Reihenfolge der Werte im Gegensatz zur Listensicht keine Rolle. Beide Modelle machen Sinn und kommen in verschiedenen Frames zum Einsatz, daher habe ich eine Frame-abhängige Äquivalenz- und Subsumptionsrelation zugelassen.

Insgesamt habe ich mit den Klassen und Interfaces im Java-Package `de.renew.agent.repr.common`, insbesondere den Klassen `GenericKVT` und `GenericVT`, eine Repräsentierungsbibliothek geschaffen, die vielseitig für beliebige Nachrichtenelemente und Ontologien eingesetzt werden kann. Die zentrale Bereitstellung der Kernfunktionalität erleichtert das Hinzufügen neuer Fähigkeiten zu allen Ontologieklassen auf einmal, was ich für die Implementierung der konfigurierbaren Typprüfung, der generischen Zeichenkettendarstellung und der Subsumptionsrelation genutzt habe.

Die generische Funktionalität ist soweit vollständig, dass die spezialisierten Klassen eigentlich gar nicht mehr benötigt werden, sie stellen dem Entwickler lediglich zusätzlichen Komfort zur Verfügung. Die Definition der Äquivalenz- und Subsumptionsrelationen ermöglicht die gemischte Verwendung von Objekten der generischen und der spezialisierten Klassen für einen Frame. Damit ist parallel zum Java-Typsystem ein neues Typsystem entstanden, dass auf den Frame-Bezeichnern basiert. Allerdings kann nun das Java-Typsystem nicht mehr zur Prüfung des Frames von Objekten herangezogen werden, da immer auch ein Objekt der generischen Klasse vorgefunden werden kann.

Um das Java-Typsystem mit der im Compiler integrierten statischen Typprüfung wieder nutzbar zu machen, gibt es die Klassen `KVTFactory` und `VTFactory`. Die Aufgabe dieser Klassen ist es, sämtliche spezialisierten Implementierungen des `KVT`- bzw. `VT`-Interfaces zu kennen und auf Angabe eines Frame-Bezeichners hin ein Objekt der entsprechend spezialisierten Unterklasse zu erzeugen. Damit ersetzen die Methoden dieser Klasse die Konstruktoren der `KVT`- und `VT`-Implementierungen. Solange KVTs und VTs ausschließlich über die jeweilige Fabrik erzeugt werden (was mehrfach in der Dokumentation der betroffenen Klassen gefordert wird), kann die Verwendung von eindeutigen Java-Typen für bestimmte Frames sichergestellt werden.

Mit der beschriebenen Klassenstruktur für die interne Nachrichtenrepräsentierung werden drei Ebenen der Repräsentierung abgedeckt: Nachrichtenobjekt und -umschlag als solche werden durch `KVT`-Objekte dargestellt, darüber hinaus lässt sich die Struktur der Inhaltssprache SL0 mit KVTs und VTs nachbilden, und nicht zuletzt sind gerade Ontologien wie die „Agent Management Ontology“ durch Repräsentierungsobjekte darstellbar.

Aus dieser Erkenntnis ergibt sich die Idee, alle Repräsentierungsebenen in einen einheitlichen Rahmen zu fassen. Von den drei Ebenen scheint mir der Ontologie-Begriff der abstrakteste zu sein, der sich auch auf Inhaltssprachen und Nachrichten als solche anwenden lässt. Daher schwebt mir vor, einen allgemeinen Ontologiedienst zu definieren, der die Aufgaben der Fabrik-Klassen übernimmt, aber dabei deutlich flexibler werden soll. Die Idee geht in die Richtung, dass sich eine Ontologiekombination je nach Bedarf zusammenstellen lässt, so dass ein Agent, der eine Nachricht an das AMS verfassen oder interpretieren möchte, sich die verfügbaren Ontologien “ACL-Message“, „SL0“ und „Agent-Managment-Ontology“ zusammenstellt. Die zusammengestellte Ontologie übernimmt dann die Erzeugung der einzelnen Nachrichtenelemente, die bisher in den global definierten Klassen `KVTFactory` und `VTFactory` erzeugt wurden. Ein weiterer Vorteil dieses Verfahrens (neben der gewonnen Flexibilität) ist die Möglichkeit getrennter Namensräume, so dass ein Frame in verschiedenen Ontologien durch verschiedene spezialisierte Klassen repräsentiert werden kann.

Die Implementierung der Ontologie-Abstraktion für die internen Repräsentierungsklassen wird aber nicht mehr im Rahmen dieser Diplomarbeit stattfinden können.

Abschließend möchte ich noch klarstellen, dass die in diesem Abschnitt beschriebene Art der Repräsentierung von Ontologien, auch mit der angedachten Abstraktion zu einem Ontologiedienst hin, eine *implizite* Kodierung von Ontologien darstellt. Die von der FIPA in der „Ontology Service Specification“ [FIPA00086] (siehe auch Abschnitt 4.2.4) angeregte *explizite* Darstellung von Ontologien in einer Metasprache wird von den hier vorgestellten Klassen nicht geleistet. Durch die explizite Beschreibung von Ontologien würde der zu betreibende Aufwand beim Erzeugen von Nachrichten eher wachsen als schrumpfen, weil mehr Flexibilität immer auch mehr explizite Entscheidungen verlangt. Das entspräche nicht meiner ursprünglichen Intention, die Nachrichtenerzeugung für Agenten zu vereinfachen.

5.2.5 Repräsentierungskonversion

Die im vorigen Abschnitt 5.2.4 vorgestellte interne Repräsentierung von Nachrichtenelementen kann den Agentenentwickler nur dann wirklich vom Aufwand des Erstellens oder Parsens standardkonformer Nachrichten befreien, wenn die Umwandlung vom bequemen internen Format in eine standardisierte Repräsentierung automatisch erfolgt. Neben dieser speziellen Umwandlung werden auch Umwandlungen zwischen standardisierten und/oder anderen Repräsentierungen benötigt, wenn Nachrichten über verschiedene Transportprotokolle verschickt werden sollen. Die „FIPA Abstract Architecture Specification“ [FIPA00001] sieht zu diesem Zweck die Einrichtung eines Transformationsdienstes vor, dessen Einbindung in eine FIPA2000-konforme Plattform ich bereits in Abschnitt 5.1.3 diskutiert habe.

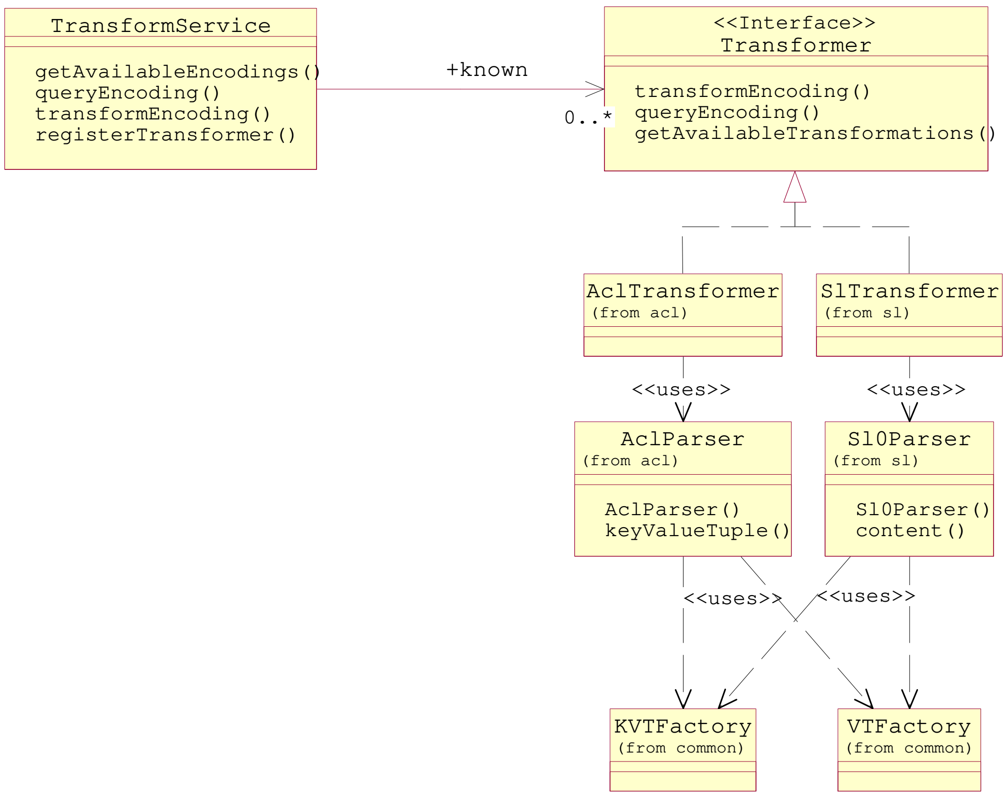


Abbildung 5.25: Die Klassenstruktur des Transformationsdienstes

In Abbildung 5.25 sind die grundlegenden Klassen und Interfaces des Transformationsdienstes inklusive zweier exemplarischer Implementierungen dargestellt. Die global bekannte Klasse `TransformService` verwaltet alle Transformationsmöglichkeiten und stellt die Schnittstelle zur Nutzung des Dienstes bereit. Entsprechend den Vorgaben der abstrakten Architektur bietet der Transformationsdienst drei Funktionen: Es kann eine Liste der bekannten Repräsentierungen angefordert, für ein gegebenes Objekt die verwendete Repräsentierung ermittelt und natürlich ein Objekt von einer Repräsentierung in eine andere umgewandelt werden. Repräsentierungen werden durch ein Namensschema analog den Frames von KVTs identifiziert, wobei die FIPA auch hier die Hoheit über den Namensraum hat.

Die eigentliche Arbeit übernehmen die Implementierungen des `Transformer`-Interfaces. Jeder Transformer kann eine oder mehrere Umwandlungen zwischen verschiedenen Repräsentierungen beherrschen sowie ein gegebenes Objekt auf Konformität zu den ihm bekannten Repräsentierungen prüfen. Alle Transfomer, die beim globalen `TransformService` registriert sind, werden von diesem zur Bereitstellung des Umwandlungsdienstes herangezogen.

Soll von einem Objekt die Repräsentierung ermittelt werden, so wird ein Rateverfahren angewendet. Der `TransformService` fragt alle registrierten Transformer, ob sie die Repräsentierungsform des Objektes kennen und sammelt die Antworten. Soll ein Objekt von einer Repräsentierung in eine andere umgewandelt werden, so sucht der `TransformService` einen Transformer heraus, der die entsprechende Umwandlung beherrscht und lässt diesen die Transformation durchführen.

Somit steht ein über die globale Klasse ansprechbarer Transformationsdienst zur Verfügung, der jederzeit um weitere Umwandlungsmöglichkeiten ergänzt werden kann, indem zusätzliche Transformer registriert werden. Welcher Art die Umwandlung ist, ob vom proprietären in ein standardisiertes Nachrichtenformat, ob zur Verschlüsselung eines Nachrichtenobjekts, ob zur Übersetzung einer Inhaltssprache oder Ontologie, wird vom Transformationsdienst nicht unterschieden. Der jeweilige Wunsch muss explizit durch Verwendung der Repräsentierungsbezeichnungen ausgedrückt werden.

Die Klassen `AclTransformer` und `SlTransformer` in Abbildung 5.25 sind zwei exemplarische Implementierungen eines Transformers. Sie beherrschen die Umwandlung zwischen der plattforminternen Nachrichtendarstellung durch Java-Objekte und der standardisierten Zeichenkettendarstellung der FIPA. Während der `AclTransformer` ganze ACL-Nachrichten umwandelt, kümmert sich der `SlTransformer` um den Nachrichteninhalt, sofern er in der Sprache SL0 kodiert ist. Beide Transformer bauen zur Umwandlung vom internen ins Zeichenkettenformat auf der Funktionalität der im vorigen Abschnitt 5.2.4 vorgestellten Repräsentierungsklassen auf. Je ein Parser ist für die Überführung der Zeichenkettendarstellung in Java-Objekte zuständig.

Beim Parsen der Zeichenkettendarstellung werden die Fabrik-Klassen `KVTFactory` und `VTFactory` genutzt, um eine möglichst anpassungsfähige Umwandlung zu ermöglichen. Dabei werden auch Informationen aus der in die `KVT`-Objekte integrierten Typprüfung mit zur syntaktischen Analyse herangezogen.[8] Die im vorigen Abschnitt 5.2.4 angedachte Abstraktion der Fabrik-Klassen zu einem Ontologiedienst hin könnte die Unterscheidung in zwei Parser, einen für die Sprache SL und einen für ACL-Nachrichten, überflüssig machen.

Beim Umwandeln von ACL-Nachrichten von einer Repräsentierung in eine andere müssen immer auch alle Schlüssel-Wert-Paare in der Nachrichtenstruktur in die neue Repräsentierung umgewandelt werden. Das erfordert im `ACLTransformer` einen Mechanismus, der in der Lage ist, den Nachrichteninhalt – der ja in den verschiedensten Repräsentierungen vorliegen kann – umzuwandeln. Um auch hier flexibel zu sein, empfiehlt sich eine rekursive Nutzung des Transformationsdienstes vom Transfomer aus. Allerdings muss dann bekannt sein, welche der beim Transformationsdienst bekannten Inhaltsrepräsentierungen denn zur umgewandelten ACL-Repräsentierung passen. Diese Einstufung (sowie die Unterscheidung von Inhalts- und ACL-Repräsentierungen) ist

[8] Das möglichst allgemeine Parsen von ACL-Nachrichtenelementen in möglichst spezielle Objekte wird leider durch eine syntaktische Altlast der ACL-Repräsentierung erschwert: ACL-Nachrichten werden in Zeichenkettendarstellung nicht – wie es der kanonischen Repräsentierung aller anderen KVTs entspräche – durch ihren Frame-Bezeichner eingeleitet, sondern durch das Performativ. Will man die Zeichenkettendarstellung möglichst allgemein parsen, so kann die Entscheidung, ob der zu Beginn eines Frames vorgefundene Bezeichner tatsächlich einen Frame oder stattdessen ein Performativ benennt, nicht eindeutig gefällt werden. Die derzeitige Implementierung zieht daher die Liste der bekannten Performative und Frames zu Rate und vermutet bei unbekannten Bezeichnern einen Frame.

bisher auf recht einfache Weise im ACLTransformer direkt kodiert. Eine erweiterbare, allgemeingültige Kategorisierung von Repräsentierungen wäre wünschenswert.

5.3 Bewertung

CAPA ist eine Agentenplattform, die sowohl den Spezifikationen der FIPA genügen als auch in die referenznetzbasierte Multiagentensystemarchitektur MULAN integriert sein soll. Die Architektur und Implementierung von CAPA wurde mit dem Ziel einer möglichst großen Nebenläufigkeit und Erweiterbarkeit entwickelt, gleichzeitig soll CAPA auch im praktischen Einsatz nutzbar sein. Auf der Plattform sollen Agenten im Sinne der Definitionen aus Abschnitt 3.1.1 leben und arbeiten. Darüber hinaus steht die Möglichkeit, dass Agenten von einer Plattform zur nächsten migrieren können, auf der Wunschliste – wobei dieser Wunsch wiederum eine Menge Fragen in Bezug auf Sicherheit aufwirft.

Ob und wie alle diese Ziele mit der in diesem Kapitel beschriebenen Agentenplattform erreicht wurden, soll in den folgenden Abschnitten diskutiert werden.

5.3.1 Agenten

In Abschnitt 3.1.1 wurden einige Merkmale von Agenten in Multiagentensystemen aufgezählt, nämlich Kapselung, Einbettung in die Umwelt, Proaktivität, Flexibilität, Autonomie und Zweckbestimmung. Im Rahmen von CAPA werden ausschließlich Software-Agenten, also rein kommunizierende Agenten betrachtet, daher reduziert sich z.B. die Umweltwahrnehmung und -beeinflussung auf die Nachrichtenkommunikation.

Will man Agenteneigenschaften in CAPA nachprüfen, so kann das auf zwei verschiedenen Ebenen geschehen. Zum einen werden die Agenten durch Netze modelliert und implementiert, so dass die Semantik der Referenznetze zur Überprüfung der Agentenmerkmale herangezogen werden kann. Diese Sichtweise entspricht der Abstraktion, mit der auch das MULAN-Multiagentensystem modelliert wird. Da CAPA eine Plattformarchitektur ist, die in die MULAN-Multiagentensystemarchitektur eingebettet ist, gelten viele Aussagen bezüglich der Agenteneigenschaften für beide Architekturen gleichermaßen.

Zum anderen werden die Referenznetze durch den in Java implementierten Renew-Simulator ausgeführt. Die Objekte und Schnittstellen des Simulators sind über die Java-Anschriften der Referenznetze ansprechbar, so dass auch auf dieser Ebene die Agenteneigenschaften beeinflusst werden können. Auf der Simulator-Ebene fällt das Lokalitätsprinzip (und damit z.B. die Kapselung) der Referenznetze weg, weil ein globaler Zugriff auf alle Simulationsobjekte möglich ist. Zudem kann auf dieser Ebene die Simulation so beeinflusst werden, dass keine garantierten Aussagen über das Schalten von Transitionen mehr getroffen werden können (z.B. kann die gesamte Simulation jederzeit abgebrochen werden). Daher soll im folgenden von der Implementierung des Simulators abstrahiert werden, alle Aussagen werden auf der Referenznetzebene getroffen.

Agenten auf einer CAPA-Plattform sind gekapselt, da ihre einzige Schnittstelle zur Umwelt aus den Nachrichtensende- und -emfangskanälen besteht. Von außen sind Anfragen und Modifikationen des Agentenzustands nicht möglich, ohne explizit über Nachrichten formuliert zu werden. Durch dieselbe Schnittstelle liegen auch die Wahrnehmungs- und Einflussmöglichkeiten der Agenten auf die Umwelt fest: Ein Agent kann nur wahrnehmen, was ihn als Nachricht erreicht. Der Agent kann die Umwelt – sprich: andere Agenten – nur über Nachrichten beeinflussen. Somit besteht in CAPA bzw. MULAN die Umwelt aus der Gesamtheit aller Agenten. Dies entspricht dem üblichen Modell für Software-Agentensysteme, die sich aus rein kommunizierenden Agenten zusammensetzen (vgl. Abschnitt 3.1.2).

Proaktivität und Flexibilität sind Eigenschaften, auf die die Plattformarchitektur keinen Einfluss hat, stattdessen kommt hier die Entscheidungsarchitektur des Agenten zum Zuge. Die mit dem MULAN-Modell vorgeschlagene, protokollgesteuerte Entscheidungsarchitektur ist flexibel und erlaubt auch das proaktive Handeln von Agenten. Allerdings ist das proaktive Starten von Konversationen bisher noch nicht in einer zufriedenstellenden Weise vom internen Zustand des Agenten abhängig.

Die Autonomie der Agenten wird von der technischen Seite durch den Referenznetzsimulator sichergestellt: Alle aktiven Transitionen eines Agenten werden irgendwann einmal schalten. Daher kann ein Agent seine Aktivitäten autonom steuern, solange er mindestens eine aktivierte Transition hat. Er kann sich auch entscheiden (indem er alle Transitionen deaktiviert), auf eine eingehende Nachricht zu warten und somit seine Autonomie begrenzt aufzugeben. Die plattformseitige Implementierung der Schnittstelle zum Agenten stellt sicher, dass die vom Renew-Simulator bereitgestellte Autonomie nicht beschränkt wird, solange der Agent sich im aktiven Lebenszykluszustand befindet.

In anderen Lebenszykluszuständen wird der Agent ganz oder teilweise vom Nachrichtentransport abgekoppelt und damit in seiner Autonomie bezüglich Nachrichtenversand und -empfang eingeschränkt. Diese Einschränkungen ergeben sich aus der Lebenszyklusdefinition in der „FIPA Agent Management Specification“ [FIPA00023] und sind zur standardkonformen Verwaltung der Agentenplattform notwendig. Die Autonomiebeschränkung wirkt ausschließlich auf den Nachrichtentransport, eine darüber hinaus gehende erzwungene Deaktivierung von Transitionen im Agenten findet nicht statt.

Beim kooperativen Abschalten von Agenten unter Einbeziehung des `:idle`-Kanals (siehe Abschnitt 5.2.2.1) geht die Plattform sogar so weit, den exakten Zeitpunkt der Kommunikationsbeschränkung in die Autonomie des Agenten zu legen. Dafür gibt die Plattform einen Teil ihrer Autonomie auf, was ein Sicherheitsrisiko für die Plattform darstellt (siehe Abschnitt 5.3.3).

Die Zweckbestimmung von Agenten, und damit einhergehend das Ziel, das ein Agent verfolgt, ist eine anwendungsspezifische Eigenschaft von Agenten. Allenfalls die Entscheidungsarchitektur hat Einfluss darauf, wie ein Agent seine Ziele ermittelt, repräsentiert und verfolgt. In der protokollgesteuerten MULAN-Entscheidungsarchitektur ist kein unmittelbares Konzept zur Darstellung von Zielen gegeben. Ein Agentenentwickler kann aber durch entsprechende Gestaltung von Protokollen und Wissensbasis eine Repräsentierung für Ziele einführen und verwenden. Beispielsweise kann (wie im Siedler-Projekt geschehen, siehe Kapitel 6) ein Prolog-System eingebunden werden, um Fakten und Ziele zu repräsentieren und Schlussfolgerungen zu ziehen.

Statt Ziele innerhalb der Mulan-Architektur mit Protokollen und Wissensbasen zu modellieren, kann ein Agentenentwickler sich auch entscheiden, die ganze protokollgesteuerte Entscheidungsarchitektur durch eine andere zu ersetzen, indem er das Agentennetz austauscht oder verfeinert. Auf diese Weise kann er jede beliebige Entscheidungsarchitektur in die Mulan-Multiagentensystemarchitektur einbinden und mit CAPA kombinieren. Der Austausch des Agentennetzes erfordert allerdings eine Anpassung der AMS- und Plattformagenten, da diese bisher fest auf das Mulan-Agentennetz programmiert sind. Eine entsprechende Parametrisierung der Agentenerzeugung sollte einer der nächsten Schritte bei der Weiterentwicklung dieser Agenten sein.

5.3.2 Mobilität

Wenn Software-Agenten mobil sind, so heißt das, dass sie von einer Agentenplattform zu einer anderen migrieren können. Ein mobiler Software-Agent muss zu diesem Zweck eine Agentenbeschreibung zur Zielplattform transferieren. Die Beschreibung kann sowohl seine statische Struktur, also seinen Code, als auch seinen dynamischen Zustand enthalten. Damit kann der Agent auf der Zielplattform seine Arbeit an dem Punkt wieder aufnehmen, wo er auf der Ausgangsplattform aufgehört hat.

Der Umfang an Code und Zustand, der zwischen den Plattformen übertragen wird, kann sehr unterschiedlich ausfallen. In [BN2001] werden drei Varianten von Code-Mobilität benannt: *Schwache* Mobilität nimmt nur den Code des Agenten mit, so dass die Ausführung des Agenten auf der Zielplattform neu beginnt. Bei der *starken* Mobilität werden Code und Zustand mitgenommen, so dass die Ausführung des Agenten von diesem Zustand aus wieder aufgenommen werden kann. Als Steigerung der starken Mobilität werden bei der *vollen* Mobilität alle bei der Ausführung des Codes benötigten Informationen wie Aufruf-Stack oder Namensraum mitgenommen, so dass die Fortsetzung der Ausführung für den Agenten völlig transparent geschieht.

In der „FIPA Agent Management Support for Mobility Specification“ [FIPA00087] werden die Rahmenbedingungen für mobile Agenten auf FIPA-konformen Agentenplattformen abgesteckt. Die FIPA lässt alle Mobilitätsvarianten zu – ob und wieviel Code und Zustand übertragen werden, ist ebenso offen wie die Verteilung der Migrationsarbeit zwischen Agent und Plattformen. Die Unterstützung von Mobilität ist ein Zusatzdienst, den eine FIPA-konforme Plattform nicht anzubieten braucht – in der Tat verzichten viele FIPA-konforme Plattformimplementierungen völlig auf Mobilität (so z.B. alle in Abschnitt 4.6 vorgestellten).

Zwei Gründe, warum Mobilität häufig nicht unterstützt wird, sind technische Schwierigkeiten und Sicherheitsbedenken. Die technischen Schwierigkeiten bestehen darin, den Code und Zustand des Agenten in eine verschickbare Beschreibung zu verpacken. Die Sicherheitsbedenken ergeben sich daraus, dass eine Plattform für mobile Agenten Code aus fremder Hand ausführt. Fremder Code kann der Agentenplattform willentlich Schaden zufügen, sofern keine Schutzmechanismen existieren. Aus beiden Gründen erhöht sich die Komplexität der Agentenplattform enorm, wenn Mobilität zugelassen wird.

Bei dem Entwurf und der Implementierung von CAPA stand die Unterstützung von Mobilität zwar mit auf der Wunschliste, aber an hinterster Stelle – die Grundfunktionalität der Agentenplattform hat zunächst deutlich höhere Priorität. Dass dennoch bereits eine einfache Migrationsmöglichkeit existiert, ist im wesentlichen auf das Engagement von Heiko Rölke zurückzuführen. Hier möchte ich nur kurz vorstellen, inwieweit die gemeinsam angestellten Überlegungen Einfluss auf CAPA hatten. Die im vorigen Abschnitt 5.2 vorgestellten Implementierungen der Plattformelemente berücksichtigen bereits die Ergebnisse der Überlegungen.

Der AMS-Agent versteht (wie in Abschnitt 5.2.2 beschrieben) die Aktionen `move` und `transmit`, mittels derer ein Agent die Migration beantragen und das AMS die Agentenbeschreibung zur Zielplattform übertragen kann. Dann koppelt das AMS den Agenten teilweise vom lokalen Nachrichtentransport ab, wobei der Agent den exakten Zeitpunkt mittels des in den Abschnitten 5.2.1, 5.2.2.1 und 5.2.3.2 erwähnten `:idle`-Kanals kontrollieren kann.

Anschließend verpackt der Agent den Inhalt seiner Wissensbasis in ein versendbares Format und erstellt somit die Beschreibung seines dynamischen Zustands. Da die Zustandsbeschreibung ausschließlich den Wissensbasisinhalt umfasst, geht jegliche Information über gerade in Ablauf befindliche Konversationen und unverarbeitete Nachrichten im Agentennetz verloren. Diese Einschränkung der Zustandsbeschreibung ist der Anlass für die Einführung der kooperativen Abschaltung des Agenten mittels `:idle`-Kanal: Durch diesen Kanal erhält der Agent die Möglichkeit, alle laufenden Konversationen zu beenden und Nachrichten abzuarbeiten, bevor er seine Zustandsbeschreibung erstellt.

Hat der Agent seine Zustandsbeschreibung an den AMS-Agenten geschickt, so wird von dort aus mittels `transmit`-Aktion die Erzeugung einer Kopie des Agenten durch das AMS auf der Zielplattform veranlasst. Die Erzeugung der Agentenkopie unterscheidet sich nicht von der Erzeugung eines neuen Agenten auf der Plattform. War die Erzeugung erfolgreich, wird der alte Agent auf der Ursprungsplattform endgültig beendet und der neue Agent an den Transportdienst der Zielplattform angekoppelt.

Die in CAPA implementierte Mobilität würde ich – auf der Referenznetzebene betrachtet – als eingeschränkt starke Mobilität bezeichnen. Es wird zwar ein aktueller Zustand des Agenten übertragen, so dass der Agent seine Arbeit auf der Zielplattform von diesem Zustand aus wieder aufnehmen kann, aber der übertragene Zustand schließt eben nicht alle bei der Ausführung des Agenten anfallenden Informationen ein. Es fehlen alle Marken in den Stellen des Agentennetzes, somit auch alle Nachrichten und instanziierten Protokollnetze, also alle Informationen über laufende Konversationen. Darüber hinaus fallen die gerade schaltenden Transitionen durchs Raster – eine Information, die auf Netzebene nicht in den Stellen sichtbar ist.

Am gravierendsten ist aber, dass gar kein Code übertragen wird. Die Wissensbasis eines Mulan-Agenten enthält nur die Namen der Protokollnetze, nicht aber die Netze selber. Ausgehend von der Annahme, dass das Mulan-Agentennetz allen Agenten gemein ist, stellen die Protokollnetze den Code eines Agenten dar. Nimmt der Agent nur die Namen der Netze mit, so ist er darauf angewiesen, dass auf der Zielplattform unter denselben Namen dieselben Netze zur Verfügung stehen.

Im Renew-Simulator ist eine Möglichkeit vorhanden, den aktuellen Simulationszustand abzuspeichern, um später davon ausgehend die Simulation wieder fortsetzen zu können. Ein abgespeicherter Simulationszustand umfasst sowohl die Netzstruktur als auch die aktuelle Markierung aller Netzinstanzen, gerade schaltende Transitionen können allerdings nicht gespeichert werden. Der Speichermechanismus sollte sich auch auf einzelne Agenten inklusive aller dazugehörigen Netzinstanzen (Wissensbasis, Protokollfabrik, Protokollnetze) anwenden lassen, so dass eine transparente Mobilität möglich wird. Diese und andere Alternativen werden in der voraussichtlich im nächsten Jahr fertiggestellt werdenden Dissertation von Heiko Rölke und Michael Köhler diskutiert werden.

5.3.3 Sicherheit

Innerhalb einer Agentenplattform gibt es verschiedene Sicherheitsaspekte, die zu berücksichtigen sind. Erstens können maliziöse Agenten versuchen, die Plattform anzugreifen, auf der sie leben. Zweitens sollten die Agenten vor Manipulationen durch die Plattform geschützt werden. Drittens bestehen bei der Nachrichtenkommunikation die üblichen Probleme der Authenzität und Integrität.

CAPA hat, da es mir in erster Line um die Bereitstellung der grundlegenden Plattformfunktionalität ging, in allen drei Punkten Nachholbedarf. Zwar ist die Plattform auf Referenznetzebene dank der Konzepte der Mulan-Architektur vor Angriffen durch Agenten gut geschützt, weil die definierte Schnittstelle zum Agenten nur einfache Nachrichten durchlässt und kein Blockieren von Plattformdiensten erlaubt (einzige Ausnahme ist der `:idle`-Kanal, durch den in der derzeitigen Implementierung die Lebenszyklusverwaltung blockiert werden kann). Aber auf der Java-Ebene ist noch kein Schutz der Plattform vor Manipulationen durch Transitionsanschriften in den Agentennetzen vorhanden. Diese Lücke müsste schließbar sein, indem die in Java integrierte klassenbasierte Sicherheitsverwaltung auf Referenznetzinstanzen ausgedehnt wird.

Der Schutz von Agenten vor Spionage oder Manipulationen durch die Plattform ist ebenfalls ebenenabhängig. Auf der Java- bzw. Simulationsebene ist quasi jede Art von Manipulation der Agentennetze möglich. Dagegen sind mir keine wirksamen Maßnahmen bekannt, weil die Agenten vom Simulator und damit von der Plattform ausgeführt werden müssen. Aus Sicht der Referenznetze kann die Plattform einen laufenden Agenten nur über die definierten Schnittstellenkanäle beeinflussen. Spionage und Manipulationen können damit nur an der Nachrichtenkommunikation stattfinden. Ebenfalls möglich ist eine Analyse und/oder Verfälschung der Agentenbeschreibung während der Agentenerzeugung sowie die Manipulation des von der Plattform verwalteten Lebenszyklus'.

Die Sicherheit der Nachrichtenkommunikation kann nur durch Maßnahmen wie Verschlüsselung und Signierung innerhalb des Agenten wiederhergestellt werden. Dabei entsteht aber das Problem, dass der Schlüssel ebenfalls von der Plattform eingesehen werden kann. Eine Lösung für Signaturen in unvertrauten Umgebungen wird z.B. in [CL2002] vorgeschlagen.

Gegen Lebenszyklus- und Agentenbeschreibungsmanipulationen seitens der Plattform (z.B. den „Mord“ an einem Agenten, indem schlichtweg dessen Ausführung verweigert bzw. abgebrochen wird) gibt es keine direkten Gegenmaßnahmen. Es kann allenfalls eine nachträgliche Diagnose des Angriffs von einem anderen Agenten auf einer anderen Plattform erfolgen.

Zusammengefasst gibt es in CAPA dieselben Sicherheitsprobleme wie sie in anderen Agentenplattformen auch bestehen. Natürlicherweise helfen auch dieselben Gegenmaßnahmen, solche sind aber bisher nicht implementiert. Eine Sonderstellung kann allenfalls durch die Abstraktion auf die Referenznetzebene erreicht werden, weil dann die Beschränkung auf wenige Schnittstellenkanäle einen gewissen Schutz zwischen Plattform und Agent darstellt.

Viele dieser Sicherheitsprobleme werden aber erst dann relevant, wenn die Plattform durch das Ausführen mobiler Agenten zum offenen System wird, welches fremden Code verarbeitet. Da in der bisherigen Mobilitätsimplementierung nur Zustandsinformationen und keine Netze (d.h. kein Code) übertragen werden, sind diese Probleme noch nicht akut.

5.3.4 Nebenläufigkeit

Der Einsatz von Petrinetzen erleichtert die Integration von Nebenläufigkeit in ein System, weil die Modellierung von Nebenläufigkeit, Kommunikation und Synchronisation im Vergleich zum thread- oder prozessbasierten Programmieren deutlich einfacher wird. Mehr Nebenläufigkeit bedeutet mehr Parallelausführung, wenn entsprechende Hardware zur Verfügung steht, und erhöht daher die Arbeitsgeschwindigkeit der Anwendung. Auch ohne parallele Hardware kann Nebenläufigkeit die Geschwindigkeit steigern: Wenn ein Schritt blockiert, weil auf ein externes Ereignis gewartet wird, können nebenläufige Schritte dennoch durchgeführt werden.

In einem Multiagentensystem kommt hinzu, dass das System von der Grundidee her schon nebenläufig ist (mehrere Plattformen, Agenten und Protokolle arbeiten nebeneinander her). Jede künstliche, also nur technisch und nicht konzeptionell begründete Beschränkung der Nebenläufigkeit bringt Abhängigkeiten ins System, die aus Agentensicht nicht zu erklären sind.

Mulan als reines Referenznetzmodell erlaubt maximale Nebenläufigkeit zwischen allen Plattformen, Agenten und Konversationen. Synchronisation über Kanäle findet nur statt, wenn Kommunikation zwischen den Ebenen stattfindet, also z.B. wenn während eine Konversation die Wissensbasis des Agenten manipuliert wird oder ein Agent eine Nachricht versendet. Die Netze des Mulan-Modells sind darauf angelegt, dass Transitionen grundsätzlich zu sich selbst nebenläufig schalten können, wenn ausreichend Marken (z.B. Nachrichten) vorhanden sind. Will ein Agent Nebenläufigkeit in bestimmten Situationen vermeiden, so muss er selber entsprechende Vorkehrungen treffen, indem er z.B. seine Konversationen über die Wissensbasis synchronisiert.

Die FIPA-Spezifikationen erwähnen Nebenläufigkeit nur sehr selten. Da in den Spezifikationen hauptsächlich die Nachrichtenkommunikation geregelt wird, beziehen sich die meisten Funktionsbeschreibungen auf die Bearbeitung jeweils einer Nachricht. Somit gibt es in den Beschreibungen kaum Ansatzpunkte für Nebenläufigkeit. Bei Diensten, die auf Nachrichten reagieren, ist allerdings immer implizit die Möglichkeit vorhanden, voneinander unabhängige Nachrichten nebenläufig zu verarbeiten.

Eine der wenigen Spezifikationen, in der Nebenläufigkeit explizit erwähnt wird, ist die „FIPA Interaction Protocol Library Specification“ [FIPA00025]. Dort wird zum einen betont, dass sich ein Agent nebenläufig an mehreren Konversationen beteiligen kann. Zum anderen werden für die Beschreibung der Interaktionsprotokolle AUML-Protokolldiagramme (siehe Abschnitt 4.2.2) verwendet, welche eine nebenläufige Aufsplittung der Lebenslinien zulassen.

Bei der Entwicklung von CAPA war die Beibehaltung einer möglichst hohen Nebenläufigkeit erklärtes Ziel. Daher soll im folgenden beleuchtet werden, inwieweit die einzelnen Elemente von CAPA diesem Ziel gerecht werden.

Die Agentennetze von Mulan sind zunächst ohne große Veränderungen übernommen worden. Daher gelten die über Mulan getroffenen Aussagen weiterhin. Die Granularität des wechselseitigen Ausschlusses bei Modifikationen der Wissensbasis könnte feiner sein (siehe Abschnitt 5.2.1).

Mit der Einführung des `:idle`-Kanals (siehe Abschnitt 5.2.2.1) ist allerdings eine Beschränkung innerhalb des Agenten entstanden: Die derzeitige Implementierung zählt alle ein- und ausgehenden Nachrichten sowie die laufenden Konversationen. Dadurch werden die Transitionen, an denen Nachrichten und Konversationen das Agentennetz betreten und wieder daraus verschwinden, gezwungenermaßen serialisiert. Nachrichten können vom Agenten nur noch der Reihe nach entgegengenommen werden, auch wenn die weitere Verarbeitung der Nachrichten im Agenten wieder nebenläufig erfolgt. Diese Beschränkung liegt voll im Verantwortungsbereich der einzelnen Agenten und beeinträchtigt die Nebenläufigkeit in der Plattform nicht. Dennoch sollte nach einer alternativen Lösung gesucht werden, welche es ohne Zählen ermöglicht, die Inaktivität eines Agenten festzustellen.

Der Nachrichtentransportdienst von CAPA ist im Abschnitt 5.2.3.1 bereits ausführlich auf seine Nebenläufigkeitseigenschaften untersucht worden. Die Weiterleitung unabhängiger Nachrichten geschieht nebenläufig, ebenso die Weiterleitung der Kopien einer Nachricht, die an verschiedene Empfänger adressiert sind. Die Schritte zum Weiterleiten einer Nachricht(-enkopie) müssen sequentiell abgearbeitet werden, damit die Nachricht den Empfänger nicht unnötig mehrfach erreicht.

Alle `Transport`-Implementierungen müssen darauf ausgelegt sein, mehrere Nachrichten nebenläufig zum Versenden übergeben zu bekommen. Ob sie den Versand auch nebenläufig bewerkstelligen, hängt von den technischen Gegebenheiten und der jeweiligen Implementierung ab. Die beiden existenten Transporte, der interne und der TCP/IP-Transport, beherrschen den nebenläufigen Versand und Empfang.

Die Java-Objekte zur internen Repräsentierung von Nachrichten können nur im wechselseitigen Auschluss manipuliert, aber jederzeit gelesen werden. Damit kann ein Agent die Objekte mit maximaler Nebenläufigkeit nutzen. Allerdings ist der Agent selber dafür verantwortlich, die bei nebenläufigem Lese- und Schreibzugriff möglichen Inkonsistenzen in den Objekten zu vermeiden.

Der Transformationsdienst ist in der Lage, alle bei ihm in Auftrag gegebenen Nachrichten bzw. Nachrichtenelemente nebenläufig umzuwandeln. Die Umwandlung einer Nachricht geschieht in den bisher implementierten Transformern sequentiell, da eine Zeichenfolge erzeugt bzw. interpretiert wird.

Die in Java implementierte Verwaltung des Agenten-Lebenszyklus' verfolgt dieselbe Synchronisierungsstrategie wie die Nachrichtenobjekte zur internen Repräsentierung: Modifikationen der Zustandsdatenbank finden im wechselseitigen Ausschluss statt, Lesezugriff ist jederzeit gestattet. Damit besteht allerdings die Gefahr von Inkonsistenzen, wenn während einer Modifikation gelesen wird. Eine Referenznetz-Implementierung könnte hier in zweierlei Hinsicht Verbesserungen mit sich bringen: Zum einen würden die Inkonsistenzen durch die Unteilbarkeit von Transitionsschaltvorgängen vermieden. Zum anderen ließe sich der wechselseitige Ausschluss auf eine feinere Granularität bringen, indem die Lebenszyklen verschiedener Agenten unabhängig voneinander modifiziert werden. Natürlich sind diese Verbesserungen auch im Java-Code möglich, werden aber deutlich aufwändiger und komplizierter.

CAPA wird ihrer Bezeichnung gerecht und bietet eine weitgehende Nebenläufigkeit. Dazu hat die Verwendung von Referenznetzen zur Modellierung aller Abläufe wesentlich beigetragen; Java-Code wird meistens nur für funktionale Aufgaben eingesetzt.

5.3.5 Erweiterbarkeit

Eine Agentenplattform stellt eine Grundlage für unterschiedliche Anwendungen in verschiedenen Einsatzgebieten dar. Daher ist es wichtig, dass die Plattform möglichst einfach an die jeweiligen Gegebenheiten angepasst werden kann. In Abbildung 5.1 wird bereits aufgezeigt, an welchen Punkten CAPA erweiterbar sein soll. Daraus ergibt sich eine Erweiterbarkeit und Anpassbarkeit der Plattform auf allen Ebenen, von der Entscheidungsarchitektur über die interne Nachrichtenrepräsentierung bis zum Netzwerktransport.

Der einfachste Fall ist, dass auf den bestehenden Fähigkeiten der Plattform aufbauend eine Agentenanwendung erstellt werden soll. In diesem Fall braucht der Entwickler nur anwendungsspezifische Protokollnetze zu erstellen und deren proaktive oder reaktive Ausführung in der Wissensbasis zu hinterlegen.

Besteht Bedarf für eine andere Entscheidungsarchitektur als das mit den Mulan-Agenten vorgegebene protokollgesteuerte Verhalten, so kann das Agentennetz ersetzt werden. Solange ein Agentennetz sich an die definierte Schnittstelle zum plattformspezifischen Transportdienst hält, hat der Entwickler alle Freiheiten bezüglich der gewünschten Entscheidungsarchitektur. Allerdings unterstützt die bisherige AMS-Implementierung noch nicht die Verwendung alternativer Agentennetze bei der Agentenerzeugung. Das entsprechende Protokollnetz des AMS-Agenten muss noch angepasst werden.

Soll eine anwendungsspezifische Ontologie verwendet werden, so kann auf die Repräsentierungsklassen aus `de.renew.agent.common` zurückgegriffen werden, um spezialisierte `KVT`- und `VT`-Unterklassen für die Ontologie zu definieren. Wenn die Klassen bei der jeweils zuständigen Fabrik registriert werden, werden sie von den bestehenden ACL- und SL0-Transformern automatisch mitgenutzt. Der durch die Fabriken eingeführte globale Namensraum für KVTs bzw. VTs führt allerdings zu Problemen, wenn verschiedene Ontologien denselben Frame-Bezeichner unterschiedlich definieren wollen. Hier kann die angedachte Ontologieverwaltung Abhilfe schaffen, indem sie die globalen Fabriken durch ontologiespezifische ersetzt.

Reicht für eine Anwendung die Frame-Darstellung einer Ontologie nicht aus oder soll eine andere Inhaltssprache verwendet werden, so steht den Entwicklern auch die Möglichkeit offen, völlig neue Repräsentierungen einzubringen. In diesem Fall müssen zusätzliche `Transformer` bereitgestellt werden, welche die neue interne Repräsentierung der Ontologie oder Sprache in ein FIPA-konformes Format um- und zurückwandeln können.

Gleichermaßen ist zu verfahren, wenn alternative Repräsentierungen zur Verfügung gestellt werden sollen. Um z.B. eine FIPA-konforme XML-Darstellung von Nachrichten in die Plattform einzubringen, müssen lediglich passende `Transformer` von und zu den bisher bekannten Nachrichtenrepräsentierungen erstellt und beim Transformationsdienst registriert werden.

Soll die Plattform über alternative Netzwerkprotokolle Nachrichten mit anderen Plattformen austauschen können, so braucht nur ein entsprechender `Transport` implementiert und beim zentralen Transportdienst angemeldet werden.

Die Erweiterungsmöglichkeiten auf den einzelnen Ebenen sind so angelegt, dass eine Anpassung der Plattform auf einer Ebene vorgenommen werden kann, ohne dass die anderen Ebenen ebenfalls angepasst werden müssen. Davon ausgenommen sind natürlich anwendungsspezifische Abhängigkeiten, so verlangt z.B. die Einbindung eines FIPA-konformen HTTP-Transportprotokolls auch die Bereitstellung einer XML-Repräsentierung für Nachrichten nebst entsprechenden Transformern.

CAPA ist somit auf allen Ebenen auf einfache Weise durch das Hinzufügen neuer Klassen (oder auch Netze) erweiterbar. Damit sollte CAPA für den Einsatz in vielen verschiedenen Gebieten gut gerüstet sein. Trotz der Erweiterbarkeit ist die bestehende Grundfunktionalität unkompliziert einsetzbar, so dass die Plattform auch für die schnelle Erstellung einer prototypischen Anwendung geeignet ist. CAPA kann dann Schritt für Schritt mit den steigenden Anforderungen bei der fortschreitenden Anwendungsentwicklung mitwachsen.

5.3.6 Praxistauglichkeit

Mulan hat in erster Linie die konzeptionelle Modellierung eines Multiagentensystems zum Ziel. Zwar stellt das Referenznetzmodell ein ausführbares System dar, aber die plattformübergreifende Kommunikation beispielsweise ist nur theoretischer Natur. Die Standardisierungsbemühungen der FIPA hingegen zielen auf den industriellen Einsatz von Multiagentensystemen. Zwar sind die Spezifikationen noch nicht stabil genug, um in einer produktiven Umgebung eingesetzt zu werden, aber eine praxistaugliche, herstellerübergreifende Standardisierung von Agentenkommunikation scheint möglich.

CAPA kombiniert MULAN mit den FIPA-Standards. Daher ist zu klären, wo zwischen Theorie und Praxis CAPA steht. Zunächst möchte ich klarstellen, dass bei der Entwicklung von CAPA der industrielle Einsatz nicht zu den Zielen gehörte. Vielmehr ging es – dem Rahmen einer Diplomarbeit angemessen – darum, die MULAN-Multiagentensystemarchitektur um standardisierte Kommunikationsmöglichkeiten zu erweitern. Zu einer erfolgreichen Erweiterung gehört natürlich auch die Praxistauglichkeit insoweit, dass die Entwicklung und Ausführung einer agentenorientierten Anwendung auf der Plattform möglich sein sollte.

Wie in Kapitel 6 vorgestellt wird, sind CAPA und MULAN in einem universitären Projekt erfolgreich als Plattform in einer Multiagentensystem-Anwendung eingesetzt worden. Damit ist konstruktiv bewiesen, dass bereits die derzeitige, prototypische CAPA-Implementierung sich für den Betrieb eines Multiagentensystems eignet. Dennoch macht sich der prototypische Charakter von CAPA noch an vielen Ecken bemerkbar.

So ist zum Beispiel die Erweiterbarkeit der Plattform nur durch Veränderung des Initialisierungscodes nutzbar. Es gibt keine bequeme Konfigurationsmöglichkeit, welche Dienste unter welchen Adressen zur Verfügung stehen sollen. Dasselbe gilt für das Einbringen der anwendungsspezifischen Agenten in die laufende Plattform. Der Zeitpunkt, an dem die Initialisierung der Plattform abgeschlossen ist, wird nicht automatisierbar gemeldet; daher muss der Nutzer den Start der Plattform manuell überwachen, bevor er die anwendungsspezifischen Agenten startet. Ebenso wie die Initialisierung ist auch die Terminierung der Plattform nicht ausgereift – vor jeder Simulation muss die gesamte Renew-Entwicklungsumgebung neu gestartet werden, weil Reste aus vorherigen Simulationen den Plattformstart verhindern würden.

Es gibt noch keine distributionsfertige Plattform-"Komponente", innerhalb derer alle Implementierungsdetails versteckt wären; alle Netze und Klassen müssen explizit in die Renew-Umgebung eingebunden werden. CAPA umfasst zur Zeit 88 Java-Klassen und 26 Referenznetze, bei der weiteren Entwicklung werden diese Zahlen noch steigen. Während der Java-Klassenlader den Umfang der Java-Implementierung vor dem Benutzer recht gut verbirgt, mussten ursprünglich alle Plattform-Netze samt ihrer grafischen Darstellung in die Renew-Entwicklungsumgebung geladen werden, um CAPA ausführen zu können. Wenn aber die Plattform schon so viele Netze in die Entwicklungsumgebung lädt, geht der Überblick verloren; die eigentlich interessanten, anwendungsspezifischen Netze müssen mühsam zwischen den Standardnetzen herausgefischt werden. Daher habe ich Renew um einen dem Java-Klassenlader ähnlichen Mechanismus ergänzt, der während der Simulation die semantische Information der Netzgrafiken einbindet, ohne den Nutzer mit der grafischen Darstellung zu belästigen.

Ein Vorteil des Renew-Simulators ist die grafische Darstellung des Markenspiels, von der aus auch gezielt Schaltvorgänge ausgelöst werden können. Damit steht ein in die Entwicklungsumgebung integrierter Debugger zur Verfügung, mittels dessen das laufende Multiagentensystem sowohl beobachtet als auch Schritt für Schritt ausgeführt werden kann. Allerdings gibt es beim Debugging keine Unterscheidung zwischen anwendungsspezifischen Protokollnetzen und den Netzen, welche die Agentenplattform implementieren. Um Agenteninstanzen mit ihren laufenden Konversationen beobachten zu können, muss sich der Entwickler erst durch die Plattformnetzinstanzen klicken. Diese Umständlichkeit wurde bei der von Timo Carl in [Car2002] im Anschluss an das in Kapitel 6 vorgestellte Projekt vorgenommenen Evaluation des Mulan/CAPA-Systems erkannt. Im Rahmen derselben Arbeit hat Carl eine während der Anwendungsausführung ständig aktuell gehaltene Überblicksdarstellung über die laufenden Agenten und Protokolle erstellt, welche die Beobachtung des Systems und Navigation durch die Netzinstanzen erheblich vereinfacht.

Doch all diese Detailschwierigkeiten bei der Verwendung von CAPA sind nur deshalb erwähnenswert, weil die Plattform die grundlegenden Anforderungen zum Betrieb einer agentenorientierten Anwendung erfüllt. Agenten können erzeugt werden, mit Verwendung von FIPA-ACL-Nachrichten miteinander kommunizieren und so eine Anwendung realisieren. Die Kommunikation ist auch über das Netzwerk möglich. Die Ausführungsgeschwindigkeit liegt in einem für einen Prototypen akzeptablen Bereich.

5.3.7 CAPA und FIPA

In die Entwicklung von CAPA sind etliche FIPA-Spezifikationen eingeflossen, von denen ich diejenigen mit größerem Einfluss im folgenden aufführen möchte:

Nachrichten werden nach den Regeln der „FIPA ACL Message Structure Specification“ [FIPA00061] aufgebaut und können gemäß der „FIPA ACL Message Representation in String Specification“ [FIPA00070] dargestellt werden. Die bereitgestellten Klassen der Agent-Management-Ontologie und der Inhaltssprache SL0 lehnen sich an die „FIPA Agent Management Specification“ [FIPA00023], „FIPA Agent Message Transport Service“ [FIPA00067] und „FIPA SL Content Language Specification“ [FIPA00008] an. Der CAPA-Dienst zur Umwandlung der Repräsentierung von Nachrichten und Nachrichtenelementen verfolgt die in der „FIPA Abstract Architecture Specification“ [FIPA00001] entwickelte Idee eines „Encoding Transform Service“.

Der Nachrichtentransport richtet sich nach der Spezifikation zum „FIPA Agent Message Transport Service“ [FIPA00067], welche eng mit der „FIPA Agent Management Specification“ [FIPA00023] verzahnt ist. Natürlich spielen hier auch die im vorigen Absatz genannten Spezifikationen zur ACL-Struktur und -Repräsentierung eine Rolle. Die „FIPA Abstract Architecture Specification“ [FIPA00001] hat die Klassenstruktur des Transportdienstes beeinflusst.

Die Verwaltung der auf der Plattform lebenden Agenten geschieht entsprechend den Vorgaben der „FIPA Agent Management Specification“ [FIPA00023]. Zur Kommunikation werden dabei die bereits erwähnte Inhaltssprache SL0 und die Agent-Management-Ontologie benötigt. Hinzu kommt, dass alle Anfragen an die Verwaltungsagenten der „FIPA Request Interaction Protocol Specification“ [FIPA00026] folgen müssen und dabei Performative aus der „FIPA Communicative Act Library Specification“ [FIPA00037] verwenden. Die „FIPA Agent Management Support for Mobility Specification“ [FIPA00087] hat aufgrund der eher experimentellen CAPA-Mobilitätsunterstützung nur geringe Auswirkungen auf Architektur und Implementierung der Plattform.

Auch wenn die detaillierte Berücksichtigung von acht Spezifikationen sowie Einflüsse aus weiteren Spezifikationen anderes suggerieren, so ist die FIPA-Konformität einer Agentenplattform damit noch nicht gewährleistet. Zum einen fehlt CAPA ein wesentliches Element, nämlich ein FIPA-konformes Nachrichtentransportprotokoll. Die Implementierung eines standardkonformen HTTP-, IIOP- oder WAP-Protokolles war im Rahmen dieser Diplomarbeit nicht mehr möglich. Aber das experimentelle, proprietäre TCP/IP-Transportprotokoll beweist, dass Kommunikation über ein Netzwerk möglich ist. Der modular aufgebaute CAPA-Transportdienst erlaubt die einfache Einbindung weiterer Transportprotokollimplementierungen, so dass ein FIPA-konformes Transportprotokoll jederzeit nachgerüstet werden kann.

Ein anderer Hinderungsgrund für die „FIPA-Konformität“ sind die FIPA-Spezifikationen selber. Sie sind an vielen Stellen noch nicht ausgereift, teilweise sogar fehlerhaft, und lassen gelegentlich Interpretationsspielräume, die bei unterschiedlicher Auslegung zur Störung der Kommunikation führen können. Eine grundlegendes Problem stellt sich schon bei der Wahl der zu berücksichtigenden Spezifikationen, weil die abstrakte Architektur nicht zweifelsfrei auf die Spezifikationen der FIPA2000-Suite abgebildet werden kann. Aber auch innerhalb der FIPA2000-Suite gibt es Schwierigkeiten, wenn zum Beispiel die Grammatik für die Zeichenkettendarstellung von Nachrichten mehrdeutig ist.[9] Ein weiteres Beispiel ist die Verwaltung des Agenten-Lebenszyklus’ (siehe Abschnitt 5.2.2.1): Zwar wird im Zustandsdiagramm jeder Zustand und Übergang eindeutig bezeichnet, aber die konkrete Übersetzung in ACL-Nachrichten mit einheitlicher Struktur fehlt. Somit muss zwangsläufig jede Agentenplattform eine proprietäre Syntax für diese standardisierten Funktionen erfinden.

Die FIPA-Konformität von CAPA ist zwar prinzipiell möglich, aber wegen des fehlenden Transportprotokolls noch nicht gegeben. Eine Prüfung, inwieweit Agenten auf einer CAPA-Plattform mit Agenten auf Plattformen anderer FIPA-konformer Architekturen zusammenarbeiten können, ist aus demselben Grund nicht möglich. Die Mehrdeutigkeiten der FIPA-Spezifikationen legen allerdings die Vermutung nahe, dass die Zusammenarbeit mit anderen FIPA-konformen Plattformen auch bei vorhandenem Transportprotokoll nicht reibungslos klappt. Eine Abstimmung der beteiligten Plattformen auf einen gemeinsamen „faktischen“ Standard dürfte nötig werden.

5.3.8 CAPA und Mulan

CAPA ist als Agentenplattform in die Mulan-Multiagentensystemarchitektur eingebettet. Dabei werden die äußeren zwei Ebenen der Mulan-Architektur, das Systemnetz und das Plattformnetz, durch CAPA und die darin angebotenen Nachrich-

[9] Konkret ergibt sich z.B. das Problem, dass ACL-Ausdrücke URLs gemäß RFC2396 [BLFM1998] enthalten können. Die durch den RFC gegebene Syntax für URLs ist aber in der Lage, den Rest des ACL-Ausdrucks bis zum nächsten Leerzeichen abzudecken.

In der Inhaltssprache SL tauchen URLs in der Grammatik gar nicht auf, dürften also nur als Zeichenkettenliterale formatiert verwendet werden. Dennoch werden in vielen Beispielen ACL-Ausdrücke unverändert in SL-Ausdrücke eingebaut, weil die Grammatik eines ACL-KVTs auf die eines SL-Funktionsterms abgebildet werden kann – mit Ausnahme des URL-Nonterminals.

tentransportprotokolle ersetzt. Die Dienste des Mulan-Plattformnetzes werden nun von mehreren verschiedenen CAPA-Elementen angeboten: Für Agentenerzeugung und -terminierung ist die Kombination aus AMS- und Plattformagenten zuständig; interne wie externe Kommunikation werden vom Nachrichtentransportdienst übernommen.

Da das Agentennetz an sich und die damit vorgegebene Verwendung von Protokollnetzen beibehalten werden, können Agenten auf CAPA auf derselben Entscheidungsarchitektur wie in Mulan aufbauen. Alle von den Mulan-Autoren angedachten Ideen zur Verhaltensmodellierung von Agenten sind damit nicht nur auf CAPA übertragbar, sondern sogar unverändert anwendbar.

Die Erzeugung und Terminierung von Agenten hat sich grundlegend verändert, da jetzt ein FIPA-konformer Agenten-Lebenszyklus vom AMS-Agenten verwaltet wird. Die in Mulan durch den simplen Aufruf eines synchronen Kanals bewerkstelligte Erzeugung eines Agenten muss nun mittels FIPA-konformer Nachrichten beim AMS beantragt werden. Dadurch entsteht zwar zusätzlicher Kommunikationsaufwand, ohne den wäre aber die Zusammenarbeit mit Plattformen und Agenten anderer Hersteller nicht möglich.

Im Fall des Nachrichtentransportdienstes hat sich die Schnittstelle zwischen Agent und Plattformdienst nicht verändert, lediglich das Nachrichtenformat wurde auf eine FIPA-ACL-Repräsentierung umgestellt. Die im Mulan-Plattformnetz getroffene Unterscheidung zwischen plattforminterner und -externer Kommunikation besteht in CAPA nicht mehr. Der Wegfall dieser Unterscheidung ist aber nur ein Implementierungsdetail und kann unabhängig von der Schnittstelle vor- und zurückgenommen werden. Der Vorteil dieser Änderung ist eine gleichberechtigte Behandlung aller Nachrichten innerhalb der Plattform, was sich allerdings auch nachteilig auf die Effizienz des lokalen Nachrichtentransports auswirkt.

Dass das Mulan-Systemnetz in einem System mit CAPA-Plattform nicht mehr benötigt wird, geht damit einher, dass die ehemals explizit modellierten Kommunikationsverbindungen zwischen mehreren Plattformen nun implizit durch die stattfindende Netzwerkkommunikation definiert werden. Ob Netzwerkkommunikation zwischen zwei Plattformen stattfindet, hängt von zwei Bedingungen ab: Zum einen müssen beide Plattformen ein gemeinsames Nachrichtentransportprotokoll beherrschen, zum anderen müssen die auf den Plattformen lebenden Agenten geeignete Adresseinträge bereitstellen – schließlich sind gemäß den FIPA-Spezifikationen [FIPA00001] und [FIPA00023] die Agenten selber für die Pflege ihrer Adresseinträge in den Verzeichnisdiensten verantwortlich. Damit sind plötzlich die Agenten für die vorher explizit als Teil des Systems modellierte Kommunikationsstruktur verantwortlich. Ob und welche Vor- und Nachteile mit dieser Verlagerung einhergehen, muss noch untersucht werden.

Die Summe aus CAPA und Mulan deckt alle Ebenen eines Multiagentensystems ab, sowohl konzeptionell als auch in einer praktisch einsetzbaren Implementierung. Von den technischen Aspekten des Nachrichtentransports wird über Nachrichtenaufbau und Repräsentierung bis hin zur Agenten- und Interaktionsmodellierung jeder Bereich der agentenorientierten Softwareentwicklung unterstützt. Die Unterstützung fällt noch recht techniknah aus und erfordert einige sich wiederholende Tätigkeiten des Entwicklers, stellt aber einen funktionstüchtigen Grundstock dar.

Kapitel 6

Eine Feldstudie: Das Siedler-Spiel

CAPA wurde im Rahmen eines Hauptstudiumsprojektes eingesetzt, das vom Arbeitsbereich Theoretische Grundlagen der Informatik an der Universität Hamburg im Wintersemester 2001/2002 angeboten wurde. Das Thema des Projektes war die agentenorientierte Softwareentwicklung: In der ersten Phase des Projektes sollte eine Anwendung agentenorientiert modelliert und implementiert werden, in der zweiten Phase konzentrierte sich die Arbeit auf Planung und Kooperation zwischen den Agenten. Der Ablauf und das Ergebnis des Projektes sind in [BGK+2002] dokumentiert.

Als Anwendungsbeispiel diente das im folgenden Abschnitt 6.1 kurz erläuterte Brettspiel „Die Siedler von Catan“. Die Abschnitte 6.2 und 6.3 stellen das in der ersten Projektphase entstandene agentenorientierte Modell vor, wobei zunächst die funktionale Aufteilung und danach die Interaktionen erläutert werden. Im Abschnitt 6.4 wird schließlich der Einsatz von CAPA im Rahmen des Projektes behandelt. Die zweite Projektphase ist aus plattformtechnischer Sicht nicht weiter interessant, da sie keine über die erste Phase hinausgehenden Anforderungen an die Plattform stellt. Daher soll sie hier nicht weiter behandelt werden.

6.1 Das Spiel

Das Brettspiel „Die Siedler von Catan“ [Teu1995] ist 1995 erschienen und hatte schnell einen außerordentlichen Erfolg auf dem deutschen Spielemarkt. Die Gründe für den Erfolg des Spieles dürften ähnlich zu den Gründen sein, die zu seiner Auswahl als Anwendungsbeispiel für das Projekt geführt haben.

Das Spielbrett wird variabel aus mindestens 30 Waben zusammengesetzt, so dass unzählige Variationen in den Anfangsbedingungen möglich sind. Die Waben sind fünf verschiedenen Landschaften zugeordnet und bieten entsprechende Rohstoffe an: Gebirge—Erz, Ackerland—Getreide, Weide—Schafswolle, Hügel—Lehmziegel sowie Wald—Holz. Den Waben werden beim Spielfeldaufbau Würfelwerte zugeordnet, die mit unterschiedlich hohen Wahrscheinlichkeiten fallen können – dadurch kann die Verfügbarkeit bestimmter Rohstoffe von Spiel zu Spiel deutlich schwanken. Außerdem gibt es Meer-Waben, welche den Rand einer Insel bilden. Auf den Meer-Waben können keine Rohstoffe abgebaut werden, einige bieten aber Handelsverbindungen in Form eines Hafens an.

Ein Spieler kann Siedlungen und Städte auf den Knotenpunkten zwischen den sechseckigen Waben bauen, die durch Strassen auf den Wabenkanten verbunden werden. Wenn die Würfelzahl einer Wabe fällt, ernten alle Spieler, die Siedlungen oder Städte an die Wabe angrenzend gebaut haben, den Rohstoff der Wabe. Die Rohstoffe werden benötigt, um weitere Siedlungen, Städte und Strassen zu bauen. Beim Bau dieser Elemente müssen das Strassen- und Siedlungsnetz jedes Spielers zusammenhängend bleiben und Mindestabstände zwischen Siedlungen und Städten aller Spieler eingehalten werden. Dadurch sind die zur Verfügung stehenden Bauplätze auf der Insel eine knappe Ressource, um die sich ein Wettbewerb zwischen den Spielern entwickelt.

In den seltensten Fällen ist ein Spieler in der glücklichen Lage, alle Rohstoffe, die er für seine weitere Expansion benötigt, durch seine eigenen Siedlungen in ausreichender Menge zu erhalten. Er kann dann auf zweierlei Wegen Rohstoffe tauschen. Zum einen besteht eine generelle, in den Spielregeln festgelegte, aber auch teure Tauschmöglichkeit zu festen Kursen mit der „Bank“. Hat ein Spieler Siedlungen oder Städte an Hafen-Waben gebaut, werden einzelne Tauschkurse bei der Bank günstiger. Zum anderen dürfen Spieler Rohstoffe untereinander tauschen, wobei das Tauschverhältnis individuell verhandelt werden kann. Diese Tauschmöglichkeit verleiht dem Spiel seinen besonderen Reiz, weil die Wertschätzung der einzelnen Rohstoffsorten von Spieler zu Spieler und von Spiel zu Spiel sehr unterschiedlich ausfällt.

Ziel des Spieles ist das Erreichen einer vorbestimmten Summe an „Siegpunkten“. Siegpunkte gibt es vor allem für Siedlungen und Städte, sie können aber auch durch eine besonders lange Straße oder – mit etwas Glück – durch den Kauf von „Entwicklungskarten“ erworben werden. Der variable Spielfeldaufbau, die unterschiedlich gute Erreichbarkeit einzelner Rohstoffsorten für die verschiedenen Spieler (abhängig von den ersten Siedlungen), die vielfältigen Tauschmöglichkeiten und die alternativen Siegpunktquellen ergeben einen komplexen Spielablauf, in dem viele unvergleichbare Strategien zum Erfolg führen können. Damit ist das Spiel prädestiniert für den Einsatz planender, verhandelnder, kooperierender und konkurrierender Agenten. Darüber hinaus bietet die Inselform des Spielplanes eine Vorlage für ein Ortskonzept, in dem sich mobile Agenten tummeln können.

6.2 Die Agenten

In der ersten Phase des Siedler-Projektes wurde die Anwendung agentenorientiert analysiert und modelliert. Die im Ergebnis entstandene Aufteilung der nötigen Aufgaben auf mehrere Agenten ist in Abbildung 6.1 dargestellt. In den Ellipsen werden die Aufgaben eines Agenten zusammengefasst, die Pfeile deuten den Informationsfluss zwischen den Agenten an.

Drei Agenten verwalten das Spielbrett und sorgen für die Einhaltung der Spielregeln: ein Spielleitungsagent, ein Insel-Agent und ein Bank-Agent. Der Bank-Agent führt Buch über den Rohstoffbesitz aller Spieler, bei ihm kann also der „Kontostand“ der

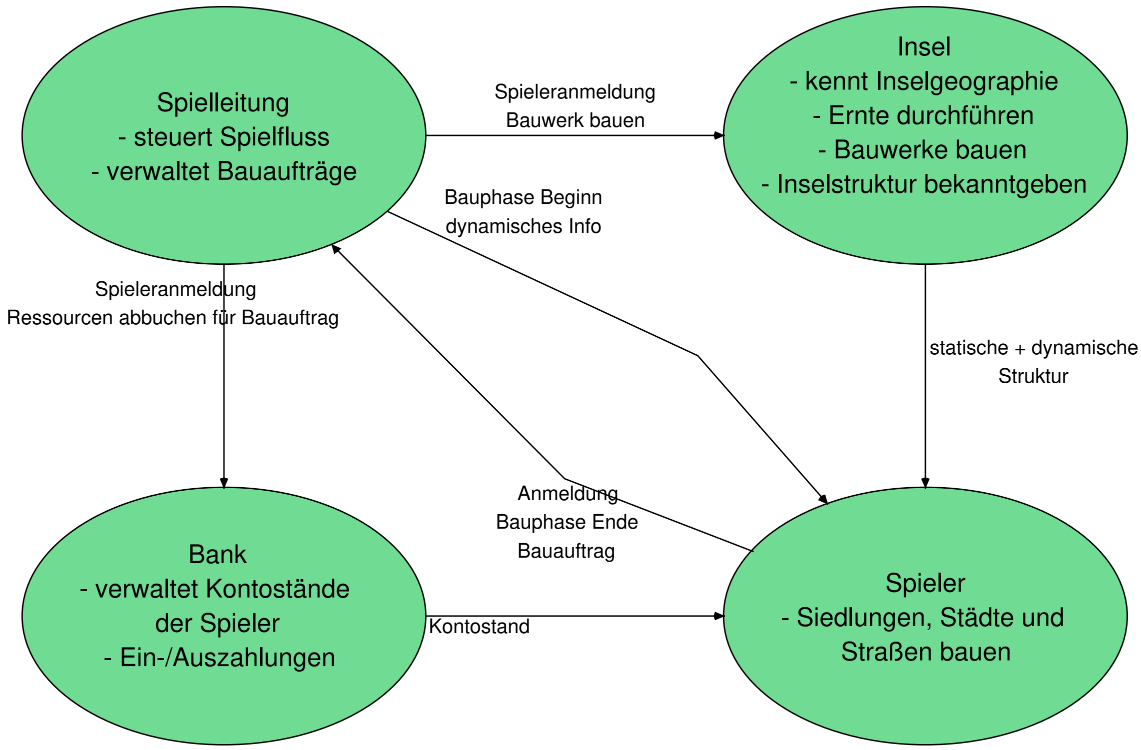


Abbildung 6.1: Aufgabenteilung und Zusammenarbeit der Agenten im Siedler-Projekt (aus [BGK+2002, S. 32])

Spieler erfragt werden. Ernteeinkünfte werden bei der Bank deponiert, Bauausgaben müssen bei der Bank abgebucht werden und Tauschgeschäfte werden ebenfalls über die Bank abgewickelt. Dabei wacht die Bank darüber, dass alle zum Tauschen oder Bauen benötigten Rohstoffe sich auch tatsächlich im Besitz des jeweiligen Spielers befinden.

Der Insel-Agent stellt das Spielbrett dar, bei ihm können Informationen über die Landschaften und Würfelwerte der Waben erfragt werden („statische Struktur“ genannt). Daneben kennt er alle Bauplätze und weiß, welcher Spieler welche Plätze belegt hat (im Projekt als „dynamische Struktur“ bezeichnet), und überwacht, ob ein Spieler auf einem gewünschten Platz bauen darf. Außerdem ist der Agent für die Rohstofferne zuständig, d.h. er bestimmt, welche Waben Rohstoffe abwerfen und zahlt die Erträge bei der Bank ein.

Der Spielleitungsagent steuert die generelle Spielphasenfolge und legt damit auch die Erntezeitpunkte fest. Ein Spieler, der neu ins Spiel einsteigen will, muss sich beim Spielleitungsagenten anmelden. Während der Bauphasen koordiniert der Agent die Insel- und Bank-Agenten: Wenn ein Spieler eine Siedlung, Stadt oder Straße bauen will, müssen sowohl die Rohstoffe bei der Bank als auch der Bauplatz auf der Insel verfügbar sein.

Spieler-Agenten kann und sollte es mehrere geben, da die Spieler auch in der agentenorientierten Umsetzung des Brettspiels miteinander um den Sieg konkurrieren und dennoch tauschenderweise kooperieren sollen. Jeder Spieler-Agent muss sich die Informationen über die statische und dynamische Struktur vom Insel-Agenten besorgen und bei der Bank Auskünfte über seinen Rohstoffbesitz einholen. Dann kann der Spieler auf beliebige Art und Weise planen und entscheiden, auf welchen Plätzen er gerne welche Bauwerke errichten würde. Dabei darf der Spieler jederzeit mit der Bank oder anderen Spielern Rohstoffe tauschen, entsprechend der im vorigen Abschnitt 6.1 beschriebenen Regeln. Im folgenden Abschnitt 6.3 wird beschrieben, mithilfe welchen Interaktionsmusters ein Spieler seine Bauwünsche in die Tat umsetzen kann.

Im Projekt sind alle Agenten mit Ausnahme der Bank parallel von mehreren Entwicklergruppen implementiert worden. Die Agenten sollten dann in allen möglichen Kombination miteinander spielen. Dabei sind drei alternative Spieler-Agenten implementiert worden. Zwei davon planen mit künstlicher Intelligenz, wobei verschiedene Planungstechniken und Strategien zum Einsatz kommen. Der dritte Agent bietet eine grafische Bedienoberfläche, damit auch menschliche Spieler am Spiel teilnehmen können.

6.3 Kommunikation

Beim Modellieren einer agentenorientierten Anwendung muss neben der Aufgabenteilung insbesondere die Interaktion und Kommunikation zwischen den Agenten betrachtet und strukturiert werden. Dazu gehört nach dem Vorbild der FIPA-Spezifikationen die Auswahl oder Definition von Interaktionsprotokollen, Inhaltssprachen und Ontologien.

Im Siedler-Projekt gab es diesbezüglich einige Schwächen. Zwar sind beispielhafte Kommunikationsabläufe mit UML-Sequenzdiagrammen modelliert worden, aber vollständige Interaktionsprotokolle wurden nicht definiert. Somit fehlte häufig die in AUML-Interaktionsdiagrammen mögliche Fallunterscheidung, mittels der alternative Reaktionen auf kommunikative Akte definiert werden können. Insbesondere die Kommunikation im Falle einer Fehlersituation ist meist vernachlässigt worden.

Die Syntax des Nachrichteninhalts wurde nur grob definiert: Nach dem Muster „*`aktion#parameter`*“ trennt ein Doppelkreuz die Aktion von den zur Aktion benötigten Parametern bzw. Detailinformationen. Die Struktur der Detailinformationen wird je nach Aktion individuell definiert. Die Zuordnung der Aktionen zu kommunikativen Akten wie „`request`“ oder „`inform`“ war nicht eindeutig. Es gab keine einheitliche Richtlinie, wie sich die Aktionsbezeichner aus der deutschen Sprache ableiten lassen – soll z.B. eine Aktion zum Bauen einer Siedlung oder Straße „`baue`“, „`bauen`“ oder „`Bauauftrag`“ heißen?

Die Inhomogenität der Nachrichtenstruktur hat im Projekt zum einen zu Mehrarbeit geführt, weil mehrere spezialisierte Parser implementiert werden mussten. Zum anderen ergaben sich Verständigungsprobleme zwischen den Agenten verschiedener Entwicklergruppen, weil Ungenauigkeiten von den Gruppen unterschiedlich interpretiert worden waren. Die Verwendung einheitlicher Repräsentierungsklassen, wie sie von CAPA unterstützt werden, hätte hier etliche Schwierigkeiten umgangen. Leider war die CAPA-SL0-Repräsentierung zu Beginn des Siedler-Projektes noch nicht implementiert.

Ein weiteres Problem, welches allerdings nicht direkt auf der Agentenkommunikationsebene liegt, stellte das Zeichnen der Mulan-Protokollnetze für die zuvor identifizierten Interaktionen dar. Zum einen waren viele Projektteilnehmer ungeübt im Umgang mit Referenznetzen, so dass das Verhalten der laufenden Netzinstanz nicht immer mit den Intentionen der Entwickler übereinstimmte. Zum anderen liegen Netze als Sprache zur Implementierung von Agenten auf einem ähnlichen Niveau wie die Sprache Assembler zur Implementierung imperativer Programme: Es gibt eine nahezu unbeschränkte Flexibilität und Mächtigkeit in der Kombination der Kontrollfluss- und Datenmanipulationsprimitive – aber die Primitive müssen in hoher Zahl verwendet und komplex kombiniert werden, um einfache Hochsprachenkonstrukte nachzubilden.

Im Anschluss an das Siedler-Projekt hat Lawrence Cabac in [Cab2002] wiederkehrende Muster in den Protokollnetzen identifiziert. Diese Muster können nun als vorgefertigte Komponenten zur Implementierung von Protokollnetzen verwendet werden. Meine eigene Forschung wird ebenfalls in diese Richtung weitergehen: Ich habe vor, die Komplexität der Nachrichtenkommunikation in CAPA hinter einer agentenorientierten Programmiersprache zu verstecken.

6.4 Einsatz der Plattform

CAPA ist parallel zum Siedler-Projekt entstanden. Daher konnte zu Beginn des Projektes nur ein kleiner Teil der gesamten Funktionalität genutzt werden. Bis zum Ende des Projektes wurden die restlichen Teile der Funktionalität unter das laufende Multiagentensystem untergeschoben. Bereits zu Beginn des Projektes waren die internen Repräsentierungsklassen für ACL-Nachrichten fertig, allerdings ohne die Inhaltssprache SL0. Das Mulan-Plattformnetz war zunächst nur gering modifiziert worden: Es wurde von synchroner auf asynchrone Nachrichtenübermittlung umgestellt und war weiterhin für den Nachrichtentransport zuständig. Die Erzeugung der Anwendungsagenten hat ein prototypischer Plattformagent übernommen, ohne jegliche weitere AMS-Funktionalität.

Die Verwendung der Repräsentierungsklassen von Anfang an war wichtig, weil jede spätere Änderung des Nachrichtenformats eine Anpassung aller Protokollnetze erfordert hätte. Der hohe Anpassungaufwand wäre zusätzlich mit dem Risiko neuer Missverständnisse zwischen den Agenten einhergegangen. Dementsprechend hat die nachträgliche Fertigstellung der SL0-Repräsentierungsklassen keine Auswirkungen mehr auf das Projekt gehabt, weil die Umstellung den Projektablauf zu sehr gebremst hätte.

Auf die Flexibilität und Ausdrucksmächtigkeit der ACL-Nachrichten wurde im Projekt weitgehend verzichtet. Zunächst reichte eine lokale Kommunikation mit bekannten Ansprechpartnern aus. Dementsprechend wurden nur die nötigsten Felder der Nachrichtenstruktur ausgefüllt, Empfänger, Performativ und Inhalt. Das Mulan-Agentennetz hat dazu die üblichen Werte in die Felder `reply-with` und `sender` eingetragen. Absender- und Empfängeradresse bestanden aus einfachsten `agent-identifier`-Strukturen, die ausschließlich den Agentennamen enthielten. Weil es noch keinen AMS- oder DF-Dienst gab, wurden feste, den Dienst beschreibende Agentennamen für „Bank“, „Insel“ und „Spielleitung“ vereinbart. Lediglich die Spieler konnten sich mit frei wählbaren Agentennamen bei der Spielleitung anmelden.

Der normierende Faktor der verwendeten Repräsentierungsklassen hat sich positiv auf die Implementierung der Agentenkommunikation im Projekt ausgewirkt. Die feste Nachrichtenstruktur, verbunden mit typsicheren Java-Methoden zum Ändern und Auslesen der Informationen, hat auf der ACL-Ebene viele Missverständnisse vermieden. Als Beleg für die Folgen fehlender einheitlicher Repräsentierungsklassen kann die Häufigkeit der syntaktischen Abweichungen beim Konstruieren und Interpretieren des Nachrichteninhalts im Projekt gelten.

Über die Normungskraft hinaus haben die Repräsentierungsklassen die Erzeugung von Nachrichten vereinfacht, indem sie wiederholt anfallende Tätigkeiten abgenommen haben. Insbesondere das Erzeugen von Antwortnachrichten durch eine spezielle Methode für diesen Zweck hat die Arbeit enorm erleichtert und viele Fehler (die z.B. durch das Übersehen der `reply-with`-Angabe hätten auftreten können) vermieden.

Gegen Ende des Siedler-Projektes war CAPA so weit gediehen, dass Kommunikation über das Netzwerk zwischen Plattformen auf verschiedenen Rechnern möglich wurde. Der ursprüngliche, einfache Transportdienst konnte unterhalb des laufenden Siedler-Projektes ausgetauscht und durch den vollständigen Transportdienst mit ACC ersetzt werden, ohne Änderungen an den Agenten zu erfordern.

Einen bei der im Projekt verwendeten Kommunikation auffälligen Verhaltensunterschied zwischen altem und neuem Transportdienst gab es nur im Fehlerfall: Der alte Transportdienst hatte unzustellbare Nachrichten liegen lassen, bis sie vielleicht irgendwann einmal zustellbar wären. Der neue Transportdienst schickt stattdessen dem Absender der unzustellbaren Nachricht eine Fehlermeldung. Dieser Unterschied führt dazu, dass die Reihenfolge der Erzeugung der Agenten relevant ist: Zuvor war auch die Anmeldung eines zu früh gestarteten Spieler-Agenten bei der Spielleitung möglich, weil die Nachricht entsprechend lange aufbewahrt wurde. Jetzt hängt der Erfolg davon ab, ob der Spielleitungsagent rechtzeitig vor Zustellung der Nachricht erzeugt wird – ein Wettrennen mit unvorhersehbarem Ausgang.

Der Austausch des Transportdienstes alleine hat zwar die Netzwerkkommunikation zwischen Plattformen ermöglicht – aber die Agenten mussten noch angepasst werden, um diese Möglichkeit auch zu nutzen. Zur Umstellung mussten alle Punkte in Agentenprotokollen überarbeitet werden, an denen Nachrichten erzeugt wurden. Die ursprüngliche Verwendung von festen Agentennamen hatte dazu geführt, dass meist unmittelbar beim Erzeugen der Nachricht auch der zur Adressierung des Empfängers benötigte `agent-identifier` erzeugt wurde. Um Netzwerkadressen in die Agenten-Identifikation einzubinden, empfiehlt sich das Ablegen der `agent-identifier`-Objekte für alle Kommunikationspartner in der Wissensbasis. Dann kann bei einer Nachrichtenerzeugung der benötigte `agent-identifier` aus der Wissensbasis geholt werden.

Die für die Netzwerkkommunikation benötigte Umstellung haben wir noch im Projekt vorgenommen, so dass beim Projektende ein verteiltes Spiel mit mehreren menschlichen Spielern möglich war. Zum Einsatz von AMS und DF ist es im Siedler-Projekt allerdings nicht mehr gekommen. Da nun aber schon die Agentenadressen in den Wissensbasen abgelegt werden, ist es relativ leicht, die fehlende Umstellung vorzunehmen: Es müssen lediglich beim Start des Agenten zum einen die eigenen Dienste beim DF registriert werden als auch mittels einer Anfrage an den DF die Adressen der Kommunikationspartner in Erfahrung gebracht werden. Die erhaltenen Adressen werden dann an die bereits definierten Plätze in der Wissensbasis gelegt.

In diesem Wintersemester 2002/2003 startet gerade eine Neuauflage des Siedler-Projektes, in dem CAPA mit seinem vollem existierenden Funktionsumfang eingesetzt werden soll. Einfache Agentenbeispiele, die sowohl SL0-Repräsentierungsklassen als auch die Verzeichnisdienste von AMS und DF nutzen, sind bereits implementiert. Parallel zum laufenden Projekt wird im Rahmen einer Diplomarbeit von Christine Reese, deren Thema die Kommunikation mit verschiedenen, international betriebenen Agentenplattformen im Rahmen des „AgentCities“-Projektes ist, ein Nachrichtentransportprotokoll über HTTP implementiert werden. Damit steht zu erwarten, dass die FIPA-Konformität von CAPA in Bälde sowohl erreicht als auch überprüft werden kann.

Kapitel 7

Ergebnis und Ausblick

Im Rahmen dieser Arbeit wird die Multiagentensystemarchitektur MULAN mit der durch die FIPA spezifizierten Kommunikationsarchitektur verknüpft. Die Verknüpfung erfolgt mittels einer Agentenplattformarchitektur namens CAPA. Ziel der Verknüpfung ist es, den petrinetzbasierten MULAN-Agenten die Kommunikation mit Agenten auf fremden, fernen oder fremden und fernen Plattformen zu ermöglichen. Dabei sollten die Eigenschaften und Fähigkeiten der MULAN-Agenten möglichst nicht beschränkt werden.

Die entstandene Agentenplattform integriert sich nahtlos in MULAN. Dabei bewahrt sie grundlegende Agenteneigenschaften wie Kapselung, Autonomie, Flexibilität oder Proaktivität soweit, wie es die FIPA-Spezifikationen zulassen. Darüber hinaus wird eine eingeschränkt starke Mobilität unterstützt, bei der Agenten mitsamt eines Großteils ihres Zustandes zu einer anderen Plattform migrieren können. Allerdings sind Sicherheitskonzepte, die Agenten und Plattform gegenseitig voreinander schützen, zwar angedacht, aber nur rudimentär vorhanden.

Die Architektur und Implementierung der Plattform ist darauf ausgelegt, größtmöglichen Vorteil aus einer der Stärken der Petrinetzmodellierung zu ziehen: Die explizite Berücksichtigung von Nebenläufigkeit im gesamten System. Alle Agenten sind – so sie nicht freiwillig auf äußere Ereignisse warten – nebenläufig und werden unabhängig voneinander ausgeführt. Gleiches gilt, wenn mehrere Konversationen unabhängig voneinander innerhalb eines Agenten ablaufen. Alle Plattformdienste wie Nachrichtentransport, Repräsentierungskonversion oder Agentenverwaltung arbeiten ebenfalls mit möglichst großer Nebenläufigkeit.

CAPA deckt ein breites Spektrum von Dienstleistungen ab, die ein Agent benötigt. Neben den grundlegenden Dienstleistungen wie der Ausführung von Agenten und dem Nachrichtentransport werden Verwaltungs-, Adressauflösungs-, Repräsentierungs- und Transformationsdienste angeboten. Wo immer es sinnvoll ist, sind diese Dienste modular aufgebaut. Damit kann zum einen für prototypische Anwendungen ein Minimalsystem auf einfache Weise zusammengestellt und genutzt werden. Zum anderen können für besondere Anforderungen spezialisierte Module erstellt und ohne großen Aufwand in die Agentenplattform integriert werden.

Die im Rahmen dieser Arbeit erstellte CAPA-Implementierung hat einen klar erkennbaren prototypischen Charakter. Will man die diversen Fähigkeiten der Plattform nutzen, fällt der dafür erforderliche Code noch relativ kompliziert aus. Um neue Module in die Plattform zu integrieren, müssen direkte Modifikationen des Initialisierungscodes der Plattform vorgenommen werden. Dennoch ist die Plattform für den praktischen Einsatz tauglich, wie das auf CAPA aufsetzende Siedler-Projekt beweist.

Die versprochene FIPA-Konformität ist bisher nicht überprüfbar, weil *noch* keines der von der FIPA definierten Nachrichtentransportprotokolle in die Plattform integriert ist. Durch ein proprietäres, prototypisches Transportprotokoll wird aber bestätigt, dass die definierten Schnittstellen zur Integration der fehlenden Transport- und Repräsentierungsmodule in der Lage sind, die Anforderungen zu erfüllen.

Der Einsatz der Agentenplattform im Siedler-Projekt hat die echte Verteilung der Anwendung auf mehrere Rechner ermöglicht. Darüber hinaus haben die in CAPA enthaltenen Repräsentierungsklassen die Kommunikationsschwierigkeiten zwischen den Anwendungsagenten reduziert. Dieser Effekt, der im Projekt aufgrund der dort verwendeten, nicht standardisierten Inhaltssprache nur teilweise zum Tragen kam, hat die Entwickler des diesjährigen Nachfolgeprojekts angeregt. Es wurden zum einen in der Vorbereitungsphase des neuen Projekts mehr Gedanken auf die Anwendungsontologie verwendet als im Vorjahr. Zum anderen zeichnet sich ab, dass verstärkt auf genormte Repräsentierungsklassen für die Nachrichteninhalte gesetzt wird.

Im Umfeld von CAPA und Mulan wird am Arbeitsbereich weiter entwickelt und geforscht. Die für die FIPA-Konformität noch fehlenden Transport- und Repräsentierungskomponenten von CAPA werden ergänzt, um anschließend das Gesamtsystem inklusive Anwendungsagenten in ein internationales Netzwerk von Agentenplattformen einzubinden. Die Plattform ist bereits um ein Anzeigemodul erweitert worden, welches dem Entwickler einen Überblick über die Agenten und Protokolle der laufenden Plattform verschafft und die Navigation durch die beteiligten Netze erleichtert. Die Modellierung von Mulan-Protokollnetzen wird darauf untersucht, ob sie durch Netzkomponenten vereinfacht werden kann. Insgesamt soll aus der Kombination von Renew, CAPA, Mulan und weiteren Ergänzungen eine integrierte Agentenentwicklungsumgebung entstehen.

Die Verknüpfung von Mulan und den FIPA-Spezifikationen mittels CAPA stellt einen Schritt auf dem Weg dar, die grafischen Ausdrucksmöglichkeiten der Petrinetze in die agentenorientierte Softwareentwicklung einzubringen. Dank der standardisierten Kommunikationsmöglichkeiten können petrinetzbasierte Agenten zukünftig mit andersartig implementierten Agenten kommunizieren, kooperieren und konkurrieren. Ein direkter Vergleich verschiedener Modellierungs- und Implementierungstechniken wird möglich.

Glossar

A

abstrakte Architektur Gemeint ist im Kontext dieser Arbeit die „→*FIPA* Abstract Architecture Specification“ [FIPA00001], welche einen allgemeineren und flexibleren Rahmen für eine FIPA-konforme →*Kommunikationsarchitektur* definiert, als es die konkreten →*FIPA2000*-Spezifikationen leisten. Die abstrakte Architektur wird in den Abschnitten 4.1 und 4.3 vorgestellt.

ACC Siehe →*Agent Communication Channel*.

ACL Siehe →*Agent Communication Language*.

Agentenplattform Unter einer Agentenplattform ist die Gesamtheit aus Hard- und Software zu verstehen, mittels derer Software-→*Agenten* ausgeführt werden. Plattform können darüber hinaus verwendet werden, um Orte zu modellieren. Siehe Abschnitt 3.1.3.

Agent Vielfältige Auffassungen machen eine Definition dieses Begriffs schwierig (siehe Abschnitt 3.1.1). In dieser Arbeit wird unter einem Agenten ein gekapselter Software-Agent verstanden, der auf einer →*Agentenplattform* angesiedelt ist und über Nachrichten mit seiner Umwelt, d.h. anderen Agenten, kommunizieren kann. Flexibles Verhalten und →*Autonomie* sind möglich, aber nicht zwingend für jeden Agenten erforderlich, denn auch ein Objekt kann als Spezialfall eines Agenten gesehen werden: als rein reaktiver, nicht-autonomer Agent.

Agent Communication Channel (ACC) Der ACC ist das zentrale Element des →*MTS* auf einer →*FIPA2000*-konformen →*Agentenplattform*. Mit seiner externen Nachrichtentransportschnittstelle ist der ACC für Versand, Empfang und Weiterleitung von Nachrichten zuständig, die Plattformgrenzen überschreiten.

Agent Communication Language (ACL) Die ACL ist eine von der →*FIPA* definierte Sprache, die mittels ihrer bekannten Syntax, Semantik und Pragmatik die Kommunikation zwischen →*Agenten* aus unterschiedlichen Systemen erlaubt, solange sich alle an der Kommunikation beteiligten Agenten dieser Sprache bedienen. Die Elemente der ACL-Sprachdefinition werden in Abschnitt 4.2 vorgestellt.

Agent Management System (AMS) Das in [FIPA00023] definierte AMS ist die zentrale Verwaltungskomponente einer →*FIPA2000*-konformen →*Agentenplattform*. Es hat die Oberhoheit über die →*Lebenszyklen* der auf der Plattform befindlichen →*Agenten* und bietet einen →*Verzeichnisdienst* zur Adressauflösung („→*weiße Seiten*“) an. Das AMS kann (muss aber nicht) als Agent mit Sonderrechten implementiert werden. Nichtsdestotrotz muss das AMS unter der fixen Adresse „`ams@`*`hap`*“ wie ein Agent ansprechbar sein (wobei *`hap`* für den Namen der Plattform steht).

AMS Siehe →*Agent Management System*.

Architektur Eine Architektur beschreibt die grundlegende Konstruktion eines Systems. Dazu gehören die Elemente des Systems, deren Aufgaben und wie die Elemente zusammenarbeiten, d.h. die Schnittstellen und Interaktionen zwischen den Elementen.

Unter einer Architektur wird je nach Anwendungsfeld eine andere Sichtweise auf ein System verstanden. Das kann zu Missverständnissen führen, wenn sich Anwendungsfelder überschneiden, wie es bei →*Multiagentensystemen* der Fall ist. Daher wird in Abschnitt 3.1.4 eine Unterscheidung der Architekturbegriffe vorgenommen. Dabei wwird zwischen den Ausprägungen →*Entscheidungsarchitektur*, →*Plattformarchitektur*, →*Kommunikationsarchitektur* und →*Multiagentensystem-Architektur* unterschieden. Ein feststehender Begriff ist die →*abstrakte Architektur* der →*FIPA*, welche eine Kommunikationsarchitektur ist.

AUML Diese Abkürzung steht für „**A**gent **UML**“; AUML ist also eine Erweiterung der →*Unified Modeling Language (UML)*. Die Zielsetzung der AUML-Entwicklung ist es, eine allgemeine Semantik, Meta-Modell und abstrakte Syntax für agentenbasierte Methoden zu finden (siehe [OPB2000]). Ein Schwerpunkt liegt dabei auf der Darstellung von →*Interaktionsprotokollen* für →*Agenten*. Die in AUML entwickelte Technik ist von der →*FIPA* in der „Interaction Protocol Library“ [FIPA00025] angewendet worden.

Autonomie Ein →*Agent* ist autonom, wenn er sein Verhalten und seinen Zustand selbst kontrollieren kann. Dies grenzt ihm vom Objekt ab, welches z.B. keine Kontrolle über den Aufruf seiner Methoden hat und nicht von sich aus – ohne äußeren Anreiz – seinen Zustand ändern kann. Allerdings hat die Autonomie immer ihre Grenzen, abhängig von den Beschränkungen, die die Umwelt dem Agenten auferlegt.

C

CCL Die **C**onstraint **C**hoice **L**anguage ist eine der von der →*FIPA* vorgeschlagenen Sprachen, mittels derer der Inhalt von Nachrichten ausgedrückt werden kann (siehe [FIPA00009] und Abschnitt 4.2.3).

D

DF Siehe →*Directory Facilitator*.

Directory Facilitator (DF) Diese Verwaltungskomponente einer →*FIPA2000*-konformen →*Agentenplattform* stellt einen →*Verzeichnisdienst* bereit. Der in [FIPA00023] definierte Dienst erlaubt die Registrierung von und Suche nach Dienste anbietenden →*Agenten* („→*gelbe Seiten*“). Es kann mehrere Anbieter eines DFVerzeichnisdienstes auf einer Plattform geben, es muss jedoch auf jeder Plattform ein Standard-DF als Agent unter der Adresse „`df@`*`hap`*“ erreichbar sein (wobei *`hap`* für den Namen der Plattform steht).

Downlink Ein Downlink ist im Kontext der →*Referenznetze* eine der beiden Transitionsanschriften, die zum Aufbau eines →*synchronen Kanals* benötigt werden. Im Gegensatz zum →*Uplink* muss die mit dem Downlink versehene Transition über eine Referenz auf das Netzexemplar verfügen, zu dem der Kanal aufgebaut werden soll.

E

Entscheidungsarchitektur Diese spezielle Ausprägung einer →*Architektur* in einem →*Multiagentensystem* legt fest, wie ein →*Agent* (re-)agiert und zu seinen Entscheidungen kommt. Diese Sicht ist in der (verteilten) Künstlichen Intelligenz weit verbreitet. Siehe Abschnitt 3.1.4.

F

FIPA Siehe →*Foundation for Intelligent Physical Agents*.

FIPA2000 Dieser Begriff bezeichnet ein Paket von Spezifikationen der →*FIPA*, die eine →*Agentenplattform* aus der Sicht einer →*Kommunikationsarchitektur* beschreiben. Das Paket besteht zur Zeit aus rund 40 Dokumenten. →*Agenten* auf →*Plattformen*, die sich an diese Spezifikationen halten, können miteinander kommunizieren.

Trotz der suggestiven Jahreszahl im Namen dieses Pakets sind einige Spezifikationen jüngeren oder älteren Datums oder wurden in der Zwischenzeit überarbeitet. Leider bestehen an einigen Stellen Diskrepanzen zur →*abstrakten Architektur* der FIPA, so dass die Interoperabilität mit anderen Instantiierungen der abstrakten Architektur noch nicht gegeben ist.

FIPA-OS Unter dem Projektnamen „FIPA **O**pen **S**ource“ entstand die erste öffentlich verfügbare, kostenfrei nutzbare Implementierung einer →*Agentenplattform* nach →*FIPA*-Standards. Siehe Abschnitt 4.6 und [FIP2001].

flexible Kante Flexible Kanten sind ein besonderes Feature der →*Referenznetze* und werden in Abschnitt 2.3 erläutert. Sie erlauben das Verschieben einer variablen Menge von Marken in einem Schaltvorgang, wobei die sich Menge der Marken deterministisch aus den anderen beteiligten Transitions- und Kantenanschriften ergeben muss.

Foundation for Intelligent Physical Agents (FIPA) Die FIPA ist eine internationale Organisation unter Beteiligung von Firmen und Universitäten, die es sich zur Aufgabe gemacht hat, durch die Entwicklung von Spezifikationen die Interoperabilität von →*Agenten* und agentenbasierten Anwendungen verschiedener Hersteller zu ermöglichen. Siehe Kapitel 4.

G

gelbe Seiten Dieses Schlagwort beschreibt einen →*Verzeichnisdienst* im Stile eines Branchenbuchs für Telefonnummern: Zu einer Dienstbezeichnung kann eine Menge von Anbietern dieses Dienstes ermittelt werden. Auf einer →*FIPA-2000*-konformen →*Agentenplattform* wird dieser Dienst vom →*DF* geboten.

H

HTTP Das **H**yper**t**ext **T**ransfer **P**rotocol (siehe [FGM+1999]) wird im Internet zur Übertragung von Webseiten verwendet. Die →*FIPA* hat in [FIPA00084] die Versendung von Agentennachrichten mittels HTTP spezifiziert.

I

IIOP Das **I**nternet **I**nter-**O**RB **P**rotocol wird im Rahmen der Common Object Request Broker Architecture [COR2002] der Object Management Group (OMG) spezifiziert. Die →*FIPA* hat in [FIPA00075] die Versendung von Agentennachrichten mittels IIOP spezifiziert.

Interaktionsprotokoll Im Kontext der →*FIPA*-Spezifikationen beschreibt ein Interaktionsprotokoll den Ablauf einer →*Konversation* zwischen zwei oder mehr →*Agenten*, z.B. eine Auktion. Das Interaktionsprotokoll gibt die möglichen Antworten auf eine eingegangene Nachricht vor und engt so die unendlich große Menge möglicher Reaktionen auf ein sinnvolles Maß ein.

Interface Obwohl eigentlich nur die englische Übersetzung des deutschen Wortes →*Schnittstelle*, wird das Wort in dieser Arbeit doch in besonderer Weise verwendet. Der englische Begriff bezeichnet in der Regel Java-Schnittstellendefinitionen, die durch das Schlüsselwort `Interface` gekennzeichnet werden.

J

JADE Das „**J**ava **A**gent **D**evelopment **F**ramework" stellt eine in Java implementierte →*Agentenplattform* nach →*FIPA*-Standards sowie einige Entwicklungswerkzeuge zur Verfügung. Siehe Abschnitt 4.6 sowie [BRP+2002] und [BPR2001].

JAS Das als „**J**ava **A**gent **S**ervices" bezeichnete, in Entwicklung befindliche Bündel von Java-Packages unter der `javax.agent`-Hierarchie definiert ein →*Interface*-Gerüst für Implementierungen der →*abstrakten Architektur* der →*FIPA*. Siehe Abschnitt 4.6 und [MAB+2002].

K

Key-Value-Tuple (KVT) KVTs werden in in vielen Spezifikationen der →*FIPA* als flexible Datenstruktur eingesetzt. In einem KVT kann eine Menge von benannten Werten (values) abgelegt werden. Die Benennung durch Schlüssel (keys) erlaubt das gezielte Abfragen bestimmter Werte, wobei neben den offiziell spezifizierten Schlüsseln auch proprietäre Schlüssel verwendet werden dürfen. In der →*abstrakten Architektur* wird darüber hinaus ein hierarchischer Namensraum für Schlüssel definiert (siehe [FIPA00001, S. 19] und Abschnitt 4.3.1).

KIF Das **K**nowledge **I**nterchange **F**ormat ist eine der von der →*FIPA* vorgeschlagenen Sprachen, um den Inhalt von Nachrichten auszudrücken (siehe [FIPA00010] und Abschnitt 4.2.3).

kommunikativer Akt Ein kommunikativer Akt ist das Grundelement der →*FIPA*-konformen Agentenkommunikationssprache (→*ACL*, siehe Abschnitt 4.2), angelehnt an die von Searle in [Sea1969] ausgearbeitete →*Sprechakt*-Theorie. Ein →*Agent*, der eine Nachricht versendet, vollzieht einen kommunikativen Akt, durch den er den Empfänger der Nachricht zu etwas auffordert, etwas fragt, über etwas informiert, vor etwas warnt usw. Eine Reihe vorgefertigter Typen von kommunikativen Akten sind von der FIPA in Form von →*Performativen* vorgegeben (siehe Abschnitt 4.2.1).

Kommunikationsarchitektur Diese spezielle Ausprägung einer →*Architektur* besteht aus Vorgaben für die Kommunikationssprachen und die Interaktion zwischen →*Agenten*. Die in Kapitel 4 vorgestellten →*FIPA*-Spezifikationen, insbesondere die →*abstrakte Architektur*, beschreiben eine solche Architektur. Siehe Abschnitt 3.1.4.

Konversation Im Allgemeinen versteht man unter einer Konversation ein Gespräch mit zwei oder mehr Beteiligten, so auch bei →*Multiagentensystemen*.

In der →*Entscheidungsarchitektur* des MULAN-Modells wird der Begriff auf technischer Ebene für ein instantiiertes →*Protokollnetz* verwendet. In den meisten Fällen entspricht dies dem intuitiven Begriff, weil das instantiierte Protokoll beschreibt, auf welche Art sich der →*Agent* an einer Konversation beteiligt. Protokollnetze können aber auch ohne Kommunikation arbeiten, wenn sie nur interne Überlegungen des Agenten beschreiben.

KVT Siehe →*Key-Value-Tuple*.

L

Lebenszyklus Die →*FIPA2000*-Spezifikation zum „Agent Management“ [FIPA00023] sieht für einen →*Agenten* sechs mögliche Zustände in seinem Lebenszyklus vor: `unknown`, `initiated`, `active`, `waiting`, `suspended` und `transit`. Zwischen diesen Zuständen sind nur bestimmte Übergänge möglich (siehe Abbildung 4.7).

M

MAS Siehe →*Multiagentensystem*.

Message Transport Service (MTS) Der Nachrichtentransportdienst stellt sowohl in der →*abstrakten Architektur* der →*FIPA* als auch in den konkreten →*FIPA2000*-Spezifikationen die grundlegenden Kommunikationsmöglichkeiten für die →*Agenten* bereit.

In der abstrakten Architektur wird der MTS durch die Beschreibung seiner Dienstleistungen spezifiziert: Der Dienst erlaubt Agenten den Versand und Empfang von Nachrichten über die →*Transporte*, an die sich der Agent vorher gebunden hat. Zwecks Versand wird jede Nachricht in einem Umschlag von einer Transportbeschreibung begleitet, die die Adresse des Empfängers enthält und bei der Bindung des Empfängers an einen Transport erstellt wird.

In den Spezifikationen [FIPA00023] und [FIPA00067] der FIPA2000-Suite wird die Vielfalt der Transporte und die Entstehung der Transportbeschreibung nicht explizit modelliert. Stattdessen wird recht genau spezifiziert, wie der MTS bestimmte Informationen im Nachrichtenumschlag zu interpretieren hat. Die plattformübergreifende Schnittstelle des MTS wird auf einer FIPA2000-konformen →*Agentenplattform* vom →*Agent Communication Channel (ACC)* zur Verfügung gestellt. Die interne Schnittstelle zu den lokalen Agenten ist ebenso wie die internen Kommunikationswege nicht Gegenstand der Spezifizikationen.

Monitor Ein Monitor ist ein Konzept, welches in einer nebenläufigen Umgebung den wechselseitigen Ausschluss für kritische Abschnitte sicherstellen kann. Im geschützten Bereich des Monitors kann nur ein →*Thread* zur Zeit arbeiten, alle weiteren an den Daten interessierten Threads werden bis zur

Freigabe des Monitors angehalten. Die Verwendung von Monitoren in der Programmiersprache Java wird in Abschnitt 2.1 erläutert.

MTS Siehe →*Message Transport Service*.

Multiagentennetze (Mulan) Die in Kapitel 3.2 vorgestellten **Mul**ti**a**genten**n**etze beschreiben eine →*Multiagentensystem-Architektur* mittels geschachtelter →*Referenznetze*. Das System wird in vier Ebenen modelliert: Von der äußersten, groben Systemsicht taucht das Modell über die →*Agentenplattform* und den →*Agenten* bis zum →*Protokoll* in immer tiefere Detaillierungsgrade ab. Dank der operationalen Semantik der Referenznetze kann das MULAN-Modell ausgeführt werden.

Multiagentensystem-Architektur Diese spezielle Ausprägung einer →*Architektur* umfasst das gesamte →*Multiagentensystem* in seinem grundlegenden Aufbau. Dabei werden nicht notwendigerweise alle detaillierten Architekturen wie die →*Entscheidungsarchitektur*, →*Plattformarchitektur* oder →*Kommunikationsarchitektur* gleich mit abgedeckt, sondern eher der grobe Rahmen für die Zusammenarbeit dieser Teilarchitekturen vorgegeben. Siehe Abschnitt 3.1.4.

Multiagentensystem (MAS) Ein Multiagentensystem ist ein System, das grob folgendermaßen charakterisiert werden kann: Die Funktionalität des Systems wird durch mehrere →*Agenten* erbracht, wobei die Agenten nur über unvollständige Informationen bezüglich des Gesamtzustandes des Systems verfügen. Es gibt keine globale Kontrolle, Synchronisation oder Datenhaltung. Eine alternative, operationale Definition wird in Abschnitt 3.1.2 vorgestellt.

Mulan Siehe →*Multiagentennetze*.

O

Ontologie Eine Ontologie definiert Begriffe, die zur Beschreibung und Repräsentierung eines Wissensgebiets verwendet werden können, sowie die Beziehungen und Eigenschaften der durch dieses Vokabular bezeichneten Objekte. In der →*FIPA*-Agentenkommunikation werden Ontologien herangezogen, um zu klären, wie der Inhalt von Nachrichten interpretiert werden soll.

P

Performativ In dieser Arbeit bezieht sich der Begriff „Performativ“ auf den Bezeichner des Typs eines →*kommunikativen Akts*. Dabei handelt es sich meist um Verben, die die Intention des kommunikativen Akts verdeutlichen, wie zum Beispiel `inform`, `request` oder `propose` (siehe Abschnitt 4.2.1). In den

→*FIPA*-Spezifikationen wird der Performativ-Begriff vermieden, lediglich das den Typ einer →*ACL*-Nachricht festlegende Element trägt den Namen `performative`.

Plattform Siehe →*Agentenplattform*.

Plattformarchitektur Diese spezielle Ausprägung einer →*Architektur* beschreibt den technischen Aufbau einer →*Agentenplattform* mit allen den →*Agenten* zur Verfügung gestellten Diensten. Diese Sicht ist eher softwaretechnischen Ursprungs und wird in dieser Arbeit vor allem in Kapitel 5 eingenommen. Siehe Abschnitt 3.1.4.

proaktiv Ein →*Agent* handelt proaktiv, wenn er von sich aus aktiv wird, also ohne äußeren Anreiz handelt. Dies kann z.B. in Folge der Ergebnisse seines eigenen Denkprozesses geschehen. Im Gegensatz dazu steht der →*reaktive* Agent, der nur auf externe Ereignisse reagiert. Mischformen aus reaktivem und proaktivem Verhalten sind üblich, die rein proaktive Form ist hingegen eher selten anzutreffen.

Protokoll Dieser Begriff ist mehrfach belegt, wobei in dieser Arbeit nach Möglichkeit die spezielleren Begriffe genutzt werden:

Im Kontext eines Standard-Agenten von →Mulan beschreiben Protokollnetze in Zusammenarbeit mit einer Wissensbasis das Verhalten des →*Agenten*.

Einige →*FIPA*-Spezifikationen definieren einzelne →*Interaktionsprotokolle*, die dem Ablauf einer Kommunikation zwischen zwei Agenten einen festen Rahmen geben.

Im Nachrichtentransportsystem einer →*FIPA2000*-konformen →*Agentenplattform* wird der Begriff Transportprotokoll alternativ zu →*Transport* verwendet.

R

RDF Das **R**esource **D**escription **F**ramework ist eine der von der →*FIPA* vorgeschlagenen Sprachen, um den Inhalt von Nachrichten auszudrücken (siehe [FIPA00011] und Abschnitt 4.2.3).

reaktiv Ein rein reaktiver →*Agent* handelt immer genau dann, wenn er Ereignisse aus der Umwelt wahrnimmt, also z.B. eine Nachricht empfängt. Im Gegensatz dazu steht →*proaktive* Agent, der Handlungen von sich aus, als Ergebnis seines Überlegungsprozesses, anstößt. Mischformen aus reaktivem und proaktivem Verhalten sind üblich.

Referenznetze Referenznetze sind objektorientierte Petrinetze höherer Ordnung mit der Anschriftssprache Java, in denen Marken wiederum Netze sein können. Zur Kommunikation zwischen Netzexemplaren steht das Konzept der →*synchronen Kanäle* zur Verfügung.

Eine Einführung in die Semantik der Referenznetze bietet Abschnitt 2.3.

Renew Der **Re**ference **Ne**t **W**orkshop ist eine grafische Entwicklungsumgebung, in der →*Referenznetze* gezeichnet und →*simuliert* werden können.

S

Schnittstelle Im Unterschied zu seinem englischen Pendant →*Interface* wird in dieser Arbeit der deutschen Begriff im allgemeinen Sinn verwendet. Er steht damit für die Beschreibung des Randes eines gekapselten Gebiets mit fest definierten Interaktions- und Kommunikationsmöglichkeiten.

Simulation In der Petrinetzgemeinde (und auch in dieser Arbeit) wird mit Simulation die Ausführung von Netzen bezeichnet. Dieses Verständnis sollte nicht mit dem weit verbreiteten Simulationsbegriff verwechselt werden, der die Nachbildung und Analyse realer Prozesse in einem Modell bezeichnet. Zwar ist die Nachbildung realer Prozesse durch die Ausführung von Petrinetzen möglich, aber der Petrinetz-Simulationsbegriff ist nicht spezifisch darauf festgelegt. Er ist eher wie die interpretierte Ausführung von Programmen zu verstehen.

SL Die **S**emantic **L**anguage ist eine der von der →*FIPA* vorgeschlagenen Sprachen, mit denen der Inhalt von Nachrichten ausgedrückt werden kann (siehe [FIPA00008] und Abschnitt 4.2.3). Für die Kommunikation mit den Verwaltungskomponenten einer →*FIPA2000*-konformen →*Agentenplattform* wird eine einfachere, in ihrer Ausdrucksfähigkeit eingeschränkte Variante namens „SL0“ verwendet.

SL wird außerdem zur formalen Beschreibung der Semantik der in der „Communicative Act Library“ [FIPA00037] definierten →*Performative* eingesetzt (siehe Abschnitt 4.2.1).

SL0 →*SL*.

Sprechakt Siehe →*kommunikativer Akt*.

synchroner Kanal Im →*Referenznetzformalismus* können mittels eines synchronen Kanals zwei Transitionen, die auch zu verschiedenen Netzinstanzen gehören dürfen, für die Dauer eines Schaltvorgangs miteinander verschmolzen werden. Während dieser →*Synchronisation* ist ein bidirektionaler Informationsaustausch zwischen den beiden Netzinstanzen möglich. Ein Kanal wird durch zwei Transitionsanschriften angefordert, den →*Uplink* und den →*Downlink*.

Synchronisation In einem nebenläufigen System müssen hin und wieder voneinander unabhängige Handlungsstränge miteinander synchronisiert werden, um z.B. Informationen auszutauschen oder die Erledigung unabhängiger Teilaufgaben abzuwarten. In einem Petrinetz kann Synchronisation sehr intuitiv modelliert werden. →*Referenznetze* erlauben darüber hinaus die Synchronisation von Transitionen aus unterschiedlichen Netzexemplaren mittels eines →*synchronen Kanals*.

T

TCP/IP Das auf dem **I**nternet **P**rotocol aufsetzende **T**ransmission **C**ontrol **P**rotocol dient als Grundlage für verbindungsorientierte Kommunikation im Internet. Viele Internet-Anwendungen bauen auf dieser kurz „TCP/IP" genannten Protokollkombination auf.

Thread Ein Thread ist ein sequentieller Strang der Programmausführung, voneinander unabhängige Threads können nebenläufig ausgeführt werden. Eine alternative Bezeichnung für „Thread" wäre „Prozess", wenn nicht der Prozessbegriff bereits von vielen Betriebssystemen belegt wäre.

Die Programmiersprache Java unterstützt Threads, ergänzt um ein →*Monitor*-Konzept zur →*Synchronisation* der nebenläufigen Stränge.

Transport Die →*abstrakte Architektur* der →*FIPA* versteht unter einem „Transport" sowohl ein Nachrichtentransportprotokoll als auch den Dienst des Nachrichtentransports über dieses Protokoll. In weiteren FIPA-Spezifikationen werden einzelne Internet-Protokolle (bisher →*HTTP*, →*WAP* und →*IIOP*) zu Transportprotokollen für Agentennachrichten ausgebaut.

In der →*abstrakten Architektur* wird ein Transportprotokoll durch einen sogenannten „Transport" (in diesem Zusammenhang kann das englische „transport" wohl am ehesten als „Verkehrsmittel" übersetzt werden) gekapselt. Der →*Agent* kann sich beim Nachrichtentransportdienst (→*MTS*) für die Nutzung verschiedener Transporte registrieren und darüber Nachrichten versenden und empfangen.

U

UML Siehe →*Unified Modeling Language*.

Unified Modeling Language (UML) Die von der Object Management Group zum Standard erhobene **U**nified **M**odeling **L**anguage (siehe [UML2001]) definiert verschiedene Diagrammarten, die bei der objektorientierten Analyse, Spezifikation und Programmierung von Nutzen sind. Die in dieser Arbeit verwendeten Diagrammarten werden in Abschnitt 2.2 kurz vorgestellt.

Uplink Ein Uplink ist im Kontext der →*Referenznetze* eine der beiden Transitionsanschriften, die zum Aufbau eines →*synchronen Kanals* benötigt werden. Im Gegensatz zum →*Downlink* hat die mit dem Uplink versehene Transition keine Referenz auf das andere am Kanal beteiligte Netzexemplar.

URL Ein **U**niform **R**esource **L**ocator adressiert eine im Internet abrufbare Ressource. Für URLs wird in [BLFM1998] eine feste Syntax vorgegeben. Das mit der Syntax vorgegebene Schema umfasst die Internet-Adresse des ressourcenanbietenden Rechners, das zum Abruf zu verwendende Transportprotokoll, den Pfad zur Ressource auf dem Rechner und ggf. Parameter für den Ressourcenabruf.

V

Verzeichnisdienst Dieser Dienst verwaltet für die ihn nutzenden →*Agenten* ein Verzeichnis, in dem Adressen von Agenten hinterlegt und – wahlweise nach dem Prinzip der →*weißen* oder →*gelben Seiten* – gesucht werden können. Während die →*abstrakte Architektur* nur die Existenz mindestens eines Verzeichnisdienstes vorschreibt und eine Grobstruktur der Verzeichniseinträge vorgibt (siehe Abschnitt 4.3.3), trennt die →*FIPA2000*-Spezifikation die Aufgabenbereiche in gelbe und weiße Seiten und sieht mindestens zwei Dienste vor: →*DF* und →*AMS* (siehe Abschnitt 4.4.2).

W

WAP Das **W**ireless **A**pplication **P**rotocol (siehe [WAP2002]) wird verwendet, um Internetkommunikation für drahtlose Geräte wie z.B. Handys zu ermöglichen. Die →*FIPA* hat in [FIPA00076] die Versendung von Agentennachrichten mittels WAP spezifiziert.

weiße Seiten Dieses Schlagwort beschreibt einen →*Verzeichnisdienst* im Stile des klassischen Telefonbuchs: Zu einem Namen kann eine Adresse (im Telefonbuch z.B. die Telefonnummer) ermittelt werden. Auf einer →*FIPA2000*-konformen →*Agentenplattform* wird dieser Dienst vom →*AMS* geboten; die erhältliche Information ist eine Agentenidentifikation mit Angaben für den →*Message Transport Service*.

X

XML Die **E**x**t**ensible **M**arkup **L**anguage gibt ein allgemeines Schema vor, wie anwendungsspezifische Informationen hierarchisch strukturiert in Textform so abgelegt werden können, das sie sowohl für Menschen als auch maschinell lesbar sind.

Literaturverzeichnis

BGK+2002 Bosch, Tobias ; Gries, Oliver ; Kausch, Heiko ; Klenski, Maxim ; Lehmann, Kolja ; Morales, Michael ; Seegert, Valentin ; Vilner, Anatolij: *Agentenorientierte Implementierung des Spiels "Die Siedler von Catan"*, Universität Hamburg, Fachbereich Informatik, Projektarbeit, April 2002

BLFM1998 Berners-Lee, T. ; Fielding, R. ; Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax / IETF. 1998. – RFC 2396. Verfügbar im Internet unter `http://www.ietf.org/rfc/rfc2396.txt`.

BN2001 Bettini, Lorenzo ; Nicola, Rocco D.: Translating Strong Mobility into Weak Mobility. In: Picco, Gian P. (Hrsg.): *Mobile Agents. 5th International Conference, MA 2001, Atlanta. Proceedings*, Springer, 2001 (Lecture Notes in Computer Science 2240), S. 182 pp.

BPR2001 Bellifemine, Fabio ; Poggi, Agostino ; Rimassa, Giovanni: Developing Multi-agent Systems with JADE. In: Castelfranchi, Cristiano (Hrsg.) ; Lespérance, Yves (Hrsg.): *Intelligent Agents VII. Agent Theories Architectures and Languages. 7th International Workshop, ATAL 2000, Boston. Proceedings*, Springer, 2001 (Lecture Notes in Artificial Intelligence 1986), S. 89–103

Bro1986 Brooks, R. A.: A robust layered control system for a mobile robot. In: *IEEE Journal of Robotics and Automation* 2 (1986), Nr. 1, S. 14–23

BRP+2002 Bellifemine, F. ; Rimassa, G. ; Poggi, A. ; Trucco, T. ; Caire, G. ; Bergenti, F.: *Java Agent Development Framework (JADE).* 2002. – Vertreten im Internet unter `http://sharon.cselt.it/projects/jade/`

Cab2002 Cabac, Lawrence: *Entwicklung von geometrisch unterscheidbaren Komponenten zur Vereinheitlichung von Mulan-Protokollen*, Universität Hamburg, Fachbereich Informatik, Studienarbeit, 2002

Car2002 Carl, Timo: *Evaluation und beispielhafte Erweiterung einer referenznetzbasierten Agentenumgebung*, Universität Hamburg, Fachbereich Informatik, Studienarbeit, 2002

CD1992 Christensen, Søren ; Damgaard Hansen, Niels: Coloured Petri Nets Extended with Channels for Synchronous Communication / University of Aarhus, Department of Computer Science. 1992 (DAIMI PB-390). – Forschungsbericht

CL2002 Cartrysse, K. ; van der Lubbe, J.C.A.: An agent digital signature in an untrusted environment. In: Fischer, Klaus (Hrsg.) ; Hutter, Dieter (Hrsg.): *Security in Mobile Multiagent Systems. 2nd International Workshop, SEMAS 2002, Bologna. Proceedings.* Kaiserslautern, Saarbrücken : Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Juli 2002 (Research Report RR-02-03), S. 12–17

COR2002 *Common Object Request Broker Architecture (CORBA).* Object Management Group (OMG). 2002. – Verfügbar im Internet unter `http://www.omg.org/technology/documents/corba_spec_catalog.htm`.

DMR2002 Duvigneau, Michael ; Moldt, Daniel ; Rölke, Heiko: Concurrent Architecture for a Multi-agent Platform. In: Giunchiglia, Fausto (Hrsg.) ; Odell, James (Hrsg.) ; Weiss, Gerhard (Hrsg.): *Agent-Oriented Software Engineering. 3rd International Workshop, AOSE 2002, Bologna. Proceedings*, ACM Press, Juli 2002. – Wird im Frühjahr 2003 in der LNCS-Serie von Springer wiederveröffentlicht., S. 147–159

Duv2001 Duvigneau, Michael: *Referenznetze bei der Erstellung der verteilten Anwendung des Börsenspiels*, Universität Hamburg, Fachbereich Informatik, Studienarbeit, 2001

Fer2001 Ferber, Jaques: *Multiagentensysteme – Eine Einführung in die Verteilte Künstliche Intelligenz.* München : Addison-Wesley, 2001. – Deutsche Übersetzung von Stefan Kirn (frz. Original von 1995).

FGM+1999 Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1 / IETF. 1999. – RFC 2616. Verfügbar im Internet unter `http://www.ietf.org/rfc/rfc2616.txt`.

FIP2001 *FIPA Open Source (FIPA-OS).* 2001. – Vertreten im Internet unter `http://fipa-os.sourceforge.net`

FIP2002 *Foundation for Intelligent Physical Agents (FIPA).* 2002. – Vertreten im Internet unter `http://www.fipa.org`

FIPA00001 Foundation for Intelligent Physical Agents: *FIPA Abstract Architecture Specification.* 29. August 2001. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00001/`.

FIPA00007 Foundation for Intelligent Physical Agents: *FIPA Content Language Library Specification.* 31. Juli 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00007/`.

FIPA00008 Foundation for Intelligent Physical Agents: *FIPA SL Content Language Specification*. 28. September 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00008/`.

FIPA00009 Foundation for Intelligent Physical Agents: *FIPA CCL Content Language Specification.* 22. August 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00009/`.

FIPA00010 Foundation for Intelligent Physical Agents: *FIPA KIF Content Language Specification.* 22. August 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00010/`.

FIPA00011 Foundation for Intelligent Physical Agents: *FIPA RDF Content Language Specification.* 18. August 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00011/`.

FIPA00023 Foundation for Intelligent Physical Agents: *FIPA Agent Management Specification*. 28. August 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00023/`.

FIPA00025 Foundation for Intelligent Physical Agents: *FIPA Interaction Protocol Library Specification.* 29. Januar 2001. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00025/`.

FIPA00026 Foundation for Intelligent Physical Agents: *FIPA Request Interaction Protocol Specification*. 29. Januar 2001. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00026/`.

FIPA00037 Foundation for Intelligent Physical Agents: *FIPA Communicative Act Library Specification.* 29. Januar 2001. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00037/`.

FIPA00061 Foundation for Intelligent Physical Agents: *FIPA ACL Message Structure Specification.* 1. August 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00061/`.

FIPA00067 Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Service Specification*. 21. Juli 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00067/`.

FIPA00069 Foundation for Intelligent Physical Agents: *FIPA ACL Message Representation in Bit-efficient Encoding Specification.* 25. Juli 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00069/`.

FIPA00070 Foundation for Intelligent Physical Agents: *FIPA ACL Message Representation in String Specification.* 3. Oktober 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00070/`.

FIPA00071 Foundation for Intelligent Physical Agents: *FIPA ACL Message Representation in XML Specification.* 13. Juni 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00071/`.

FIPA00075 Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Protocol for IIOP Specification.* 1. September 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00075/`.

FIPA00076 Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Protocol for WAP Specification.* 13. Juni 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00076/`.

FIPA00077 Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Profile Alpha Specification.* 2. August 2000. – Status „Obsolete“. Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00077/`.

FIPA00078 Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Profile Beta Specification.* 2. August 2000. – Status „Obsolete“. Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00078/`.

FIPA00084 Foundation for Intelligent Physical Agents: *FIPA Agent Message Transport Protocol for HTTP Specification.* 24. August 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00084/`.

FIPA00086 Foundation for Intelligent Physical Agents: *FIPA Ontology Service Specification.* 15. Juni 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00086/`.

FIPA00087 Foundation for Intelligent Physical Agents: *FIPA Agent Management Support for Mobility Specification.* 30. Juni 2000. – Verfügbar im Internet unter `http://www.fipa.org/specs/fipa00087/`.

GFM2002 Gutknecht, Olivier ; Ferber, Jacques ; Michel, Fabien: *MadKit (Multi-Agent Development Kit).* 2002. – Vertreten im Internet unter `http://www.madkit.org/`

GJS1996 Gosling, James ; Joy, Bill ; Steele, Guy: *The Java Language Specification.* Sun Microsystems, Inc., 1996. – Verfügbar im Internet unter `http://java.sun.com/docs/books/jls/index.html`

HS1998 Huhns, Michael N. ; Singh, Munindar P.: Agents and Multiagent Systems: Themes, Approaches and Challenges. In: Huhns, Michael N.

(Hrsg.) ; Singh, Munindar P. (Hrsg.): *Readings in Agents.* San Francisco, CA : Morgan Kaufman, 1998, S. 1–23

JCP2002 *Java Community Process, Version 2.5 (JCP 2).* Sun Microsystems, Inc. 2002. – Verfügbar im Internet unter `http://java.sun.com/aboutJava/communityprocess/jcp2.html`

Jen2000 Jennings, N. R.: On Agent-Based Software Engineering. In: *Artificial Intelligence* 117 (2000), Nr. 2, S. 277–296

JSW1998 Jennings, Nicholas R. ; Sycara, Katia ; Wooldridge, Michael: A Roadmap of Agent Research and Development. In: *Autonomous Agents and Multi-Agent Systems* (1998), Nr. 1, S. 7–38

KMR2001 Köhler, Michael ; Moldt, Daniel ; Rölke, Heiko: Modeling the behaviour of Petri net agents. In: Colom, J.-M. (Hrsg.) ; Koutny, M. (Hrsg.): *Application and Theory of Petri Nets. 22nd International Conference, ICATPN 2001, Newcastle upon Tyne. Proceedings*, Springer, 2001 (Lecture Notes in Computer Science 2075), S. 224–241

Kum2001 Kummer, Olaf: Introduction to Petri Nets and Reference Nets. In: *Sozionik aktuell* (2001), Nr. 1. – Verfügbar im Internet unter `http://www.sozionik-aktuell.de`.

Kum2002 Kummer, Olaf: *Referenznetze.* Hamburg, Universität Hamburg, Fachbereich Informatik, Dissertation, 2002

KWD2002 Kummer, Olaf ; Wienberg, Frank ; Duvigneau, Michael: *Renew – The Reference Net Workshop.* Verfügbar im Internet unter `http://www.renew.de/`. 2002. – Diese Arbeit baut auf Version 1.6 auf.

MAB$^+$2002 McCabe, Francis ; Arnold, Geoff ; Bradshaw, Jeff ; de hÓra, Bill ; Griss, Martin ; Levine, David ; Spydell, Andy ; Suguri, Hiroki ; Ushijima, Satoru ; Greenwood, Dominic (Hrsg.): *Java Agent Services Specification (JAS).* 2002. – Vertreten im Internet unter `http://www.java-agent.org/`

Mol1996 Moldt, Daniel: *Höhere Petrinetze als Grundlage für Systemspezifikationen.* Hamburg, Universität Hamburg, Fachbereich Informatik, Dissertation, August 1996

Ode1999 Odell, James: *Agents and Objects: How Do They Differ?* September 1999. – working paper v2.2

OPB2000 Odell, James ; Parunak, H. Van D. ; Bauer, Bernhard: Extending UML for Agents. In: Wagner, Gerd (Hrsg.) ; Lesperance, Yves

(Hrsg.) ; Yu, Eric (Hrsg.): *Agent-Oriented Information Systems. Workshop at the 17th National Conference on Artificial Intelligence (AAAI), AOIS 2000, Austin. Proceedings*. Austin, TX, 2000, S. 3–17

Röl1999 Rölke, Heiko: *Modellierung von Multi-Agenten-Systemen mit Referenznetzen*, Universität Hamburg, Fachbereich Informatik, Diplomarbeit, 1999

Röl2002 Rölke, Heiko: Mulan: Modellierung und Simulation von Agenten und Multiagentensystemen mit Referenznetzen / Universität Hamburg, Fachbereich Informatik. 2002. – Technischer Bericht. In Vorbereitung.

Sea1969 Searle, John R.: *Speech acts*. Cambridge Univ. Press, 1969

Sun2002 Sun Microsystems, Inc.: *The Java Platform*. 2002. – Verfügbar im Internet unter `http://java.sun.com/`. Verwendet wurde das Java 2 Software Development Kit (SDK) Version 1.2.2.

Teu1995 Teuber, Klaus: *Die Siedler von Catan*. KOSMOS Verlag, Stuttgart. 1995. – Brettspiel.

UML2001 *Unified Modeling Language (UML)*. Object Management Group (OMG). 2001. – Verfügbar im Internet unter `http://www.omg.org/technology/documents/modeling_spec_catalog.htm`.

Val1998 Valk, Rüdiger: Petri nets as token objects: An introduction to elementary object nets. In: Desel, Jörg (Hrsg.) ; Silva, Manuel (Hrsg.): *Application and Theory of Petri Nets. 19th International Conference, ICATPN '98, Lisbon. Proceedings*, Springer, Juni 1998 (Lecture Notes in Computer Science 1420), S. 1–25

WAP2002 *Wireless Application Protocol Specification*. WAP Forum. 2002. – Verfügbar im Internet unter `http://www.wapforum.org/what/technical.htm`.

Wei2000 Weiss, Gerhard (Hrsg.): *Multiagent systems: a modern approach to distributed artificial intelligence*. Cambridge, Mass. : MIT Press, 2000

WJ1995 Wooldridge, Michael ; Jennings, Nicholas R.: Intelligent Agents: Theory and Practice. In: *Knowledge Engineering Review* 10 (1995), Nr. 2, S. 115–152

WLDG2001 Willmott, Steve ; Levine, David ; Dale, Jonathan ; Greenwood, Dominic: *FIPA Convergence Work Plan*. Foundation for Intelligent Physical Agents, 8. November 2001. – Verfügbar im Internet unter `http://www.fipa.org/docs/input/f-in-00047/`.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich diese Arbeit selbstständig durchgeführt und nur die angegebenen Hilfsmittel und Quellen verwendet habe.

Hamburg, 10. Dezember 2002

Michael Duvigneau