Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Research Report
RR-99-01

Ein System zur Definition und Ausführung von Protokollen für Multi-Agentensysteme

Stefan Philipps und Jürgen Lind

April 1999

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 2080
67608 Kaiserslautern, FRG
Tel: +49 (631) 205-3211
Fax: +49 (631) 205-3210
E-Mail: info@dfki.uni-kl.de

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel: +49 (631) 302-5252
Fax: +49 (631) 302-5341
E-Mail: info@dfki.de

WWW: http://www.dfki.de

Deutsches Forschungszentrum für Künstliche Intelligenz

DFKI GmbH

German Research Centre for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest non-profit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focusing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialisation.

Based in Kaiserslautern and Saarbrücken, the German Research Centre for Artificial Intelligence ranks among the important "Centres of Excellence" world-wide.

An important element of DFKI's mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

DFKI has about 115 full-time employees, including 95 research scientists with advanced degrees. There are also around 120 part-time research assistants.

Revenues for DFKI were about 24 million DM in 997, half from government contract work and half from commercial clients. The annual increase in contracts from commercial clients was greater than 37\% during the last three years.

At DFKI, all work is organised in the form of clearly focused research or development projects with planned deliverables, various milestones, and a duration from several months up to three years.

DFKI benefits from interaction with the faculty of the Universities of Saarbrücken and Kaiserslautern and in turn provides opportunities for research and Ph.D. thesis supervision to students from these universities, which have an outstanding reputation in Computer Science.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI's six research departments are directed by internationally recognised research scientists:

- ❏ Information Management and Document Analysis (Director: Prof. A. Dengel)
- ❏ Intelligent Visualisation and Simulation Systems (Director: Prof. H. Hagen)
- ❏ Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- ❏ Programming Systems (Director: Prof. G. Smolka)
- ❏ Language Technology (Director: Prof. H. Uszkoreit)
- ❏ Intelligent User Interfaces (Director: Prof. W. Wahlster)

In this series, DFKI publishes research reports, technical memos, documents (e.g. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster

Director

Ein Systems zur Definition und Ausführung von Protokollen für Multi-Agentensysteme

Stefan Philipps und Jürgen Lind

DFKI-RR-99-01

© Deutsches Forschungszentrum für Künstliche Intelligenz 1999

This work may not be copied or reproduced in whole or part for any commercial purpose. Permission to copy in whole or part without payment of fee is granted for non-profit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Deutsche Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

ISSN 0946-008X

1 Einleitung

Direkte Kommunikation unter Agenten in einem Multi-Agentensystem wird über sog. Protokolle abgewickelt. Diese Protokolle legen explizit fest, wie die Kommunikation ablaufen soll und welche Rollen die verschiedenen Agenten dabei einnehmen. So beschreibt z. B. das Contract-Net-Protokoll ([M1] - [M4]) die Kommunikation zwischen einem Agenten in der Rolle des sogenannten Managers und weiteren Agenten in der Rolle der Bieter (engl. bidder). Hier schreibt der Manager einen Auftrag an andere Agenten aus, diese ermitteln dann die Kosten, die ihnen die Erfüllung dieses Auftrags verursachen würde und senden diese als Angebot in der Rolle des Bieters zurück. Der Manager wählt dann unter den Angeboten eines aus und schickt dem entsprechenden Agenten die Bestätigung. Dieser bearbeitet den Auftrag und sendet das Ergebnis an den Auftraggeber.

In der Regel werden solche Protokollabläufe fest in die Agentenimplementierungen hineincodiert. In diesem Report wird eine Möglichkeit beschrieben, Protokolle explizit außerhalb der konkreten Implementierung zu repräsentieren, anstatt implizit im Programmcode. Das entwickelte Protokollausführungssystem wird in ein bestehendes Programm integriert werden und, ausgehend von einer Protokollrepräsentation, die Kommunikation des Agenten abwickeln. Dadurch ergibt sich die Möglichkeit, in verschiedenen Anwendungen die selbe Protokollrepräsentation zu verwenden und Protokolle bestehender Agenten auszutauschen, um so ihr Verhalten zu verändern.

2 Grundlagen

2.1 InteRRaP

Die am DFKI entwickelte Agentenarchitektur InteRRaP [M5] ist eine hybride Architektur, d.h. InteRRaP-Agenten haben reaktives, deliberatives und kooperatives Verhalten. Sie können also auf Anforderung einfache Aktionen ausführen (reaktives Verhalten), selbsttätig aufgrund ihrer Wissensbasis komplexere Aktionen planen (deliberatives Verhalten) und mit anderen Agenten kooperieren (kooperatives Verhalten). Als Schnittstelle des Agenten zur Außenwelt dient ein sogenanntes World Interface, über das der Agent Zugriff auf Sensoren und Aktoren hat, und über das er mit anderen Agenten kommunizieren kann.

2.2 Zielsprache

Die Zielsprache für den Compiler ist Oz 2.0, ein Portierung auf Mozart 1.0 ist vorgesehen.

2.3 Estelle

Estelle [S1] ist, so wie Lotos [S1] und SDL [S1], eine Spezifikations-Sprache für Systemverhalten. Sie wurde entwickelt, um Abläufe exakt zu spezifizieren, da die natürliche Sprache für diesen Zweck meist zu ungenau ist. Sie wird meist zur Spezifikation von Protokollen im Telekommunikations-Bereich eingesetzt. Estelle verwendet das Konzept der erweiterten, endlichen Automaten. Dies sind Automaten, die zusätzlich Variablenwerte speichern können. Jeder Automat, der eine eigenständige Einheit im Gesamtsystem repräsentiert, wird als ein Modul beschrieben. Innerhalb des Moduls werden die Transitionen des Automaten in einer Pascal-ähnlichen

Syntax angegeben. Ein Modul kann beliebig viele Interaktionspunkte haben, mit denen es mit der Außenwelt kommunizieren kann. Interaktionspunkte verschiedener Module werden über Kanäle miteinander verbunden. Dabei werden den Modulen an den Enden des Kanals verschiedene Rollen zugeordnet, die sie bei der Kommunikation über den Kanal spielen, z.B. Sender und Empfänger. Module sind hierarchisch aufgebaut. Ein Vatermodul kann mehrere Sohnmodule haben und deren Interaktionspunkte und Kanäle dynamisch während des Ablaufs verändern. Mit Estelle lassen sich Protokolle und Dienste beschreiben. Ein Protokoll beschreibt einen konkreten Ablauf, ein Dienst jedoch nur dessen Schnittstelle nach außen. Wird z.B. in einem Protokoll beschrieben, daß ein Verbindungsaufbau-Wunsch ggf. mehrmals gesendet wird, bis eine Bestätigung empfangen wird, kann das in einer zugehörigen Dienstbeschreibung weggelassen werden. So sendet ein Benutzer des Dienstes nur den Verbindungsaufbau-Wunsch ab und wartet auf die Bestätigung. Das mehrmalige Senden eines Verbindungsaufbau-Wunsches wird für den Anwender unsichtbar vom zugehörigen Protokoll erledigt. Mit Estelle lassen sich sehr komplexe Dienste und Protokolle beschreiben, die in mehreren Schichten aufeinander aufbauen. Auch nichtdeterministisches Verhalten kann beschrieben werden. Eine automatische Code-Generierung aus einer Estelle-Spezifikation ist jedoch aufgrund der Komplexität der beschreibbaren Abläufe nicht möglich.

2.4 Lex und Yacc

Lex und Yacc [C5] sind Werkzeuge, die helfen, Programme zur Transformation strukturierter Eingaben zu erstellen. Ein mögliches Einsatzgebiet ist die Erzeugung von Compilern. In einer speziellen Syntax wird die zu erkennende Sprache angegeben. Aus dieser Beschreibung wird dann C-Code generiert, der Eingaben in dieser Sprache lesen und erkennen kann. Der von Lex generierte Teil hat dabei die Aufgabe, die Eingabe auf der Basis regulärer Ausdrücke in Einheiten einzuteilen (parsen) und an den von Yacc generierten Teil weiterzugeben. Die erkannten Einheiten können z.B. Schlüsselworte der Sprache, Zahlen oder Bezeichner sein. Die Zeilen

```
[1-9][0-9]*   {/* c-code */ yylval.int = atoi(yytext); return INT_VALUE;}
+             {/* c-code */ return PLUS_MARK;}
```

bedeuten z.B., wenn eine, nicht mit Null beginnende, Zahlenfolge in der Eingabe erkannt wird, wird der Wert INT_VALUE an das aufrufende Programm (i.d.R. von Yacc erzeugt) zurückgegeben. Zusätzlich wird der Wert der Zahl (als String in `yytext` enthalten) in der Struktur `yylval` als Integer-Wert gespeichert. Wird ein Plus-Zeichen erkannt, wird der Wert PLUS_MARK zurückgegeben.

Mit Yacc wird nun auf Basis der von Lex erkannten Einheiten eine kontextfreie Grammatik beschrieben, die die Syntax der zu erkennenden Sprache darstellt.

In unserem einfachen Beispiel könnte z.B. eine Regel in einer Yacc-Spezifikation lauten:

```
addition: INT_VALUE  PLUS_MARK  INT_VALUE       {/* c-code */};
```

`INT_VALUE` und `PLUS_MARK` stellen dabei die terminalen Symbole der Grammatik dar. `addition` ist ein nicht-terminales Symbol. Liest der Parser (von Lex erzeugt) eine Zahl, ein Pluszeichen und wieder eine Zahl in der Eingabe, gibt er die Terminal-Symbole `INT_VALUE`, `PLUS_MARK` und `INT_VALUE` in dieser Reihenfolge an den Compiler (von Yacc erzeugt) zurück. Hier kann das nicht-terminale Symbol `addition` abgeleitet werden. Dabei wird der in geschweiften Klammern angegebene C-Code ausgeführt (hier nur ein Kommentar). Dort kann auch auf die vorher in der Struktur `yylval` gespeicherten Integer-Werte zugegriffen werden.

3 Anforderungsanalyse

Ein neues Protokoll soll von einem Anwender leicht und verständlich dargestellt (repräsentiert) werden können. Aus diesem Grund soll bewußt auf unnötige Sprachelemente verzichtet werden, die für eine entsprechende Spezifikation nicht benötigt werden und nur die Lesbarkeit eines Protokolls erschweren.

Andererseits sollen dennoch genügend Sprachmittel zur Verfügung stehen, so daß möglichst viele gängige Multi-Agentenprotokolle damit dargestellt werden können. Dazu gehören auch Protokolle wie das Contract-Net, das Interaktionen zwischen mehr als 2 beteiligten Agenten beschreibt.

Es muß sichergestellt werden, daß eine exakte Trennung zwischen Protokoll und Anwendung besteht. Als Anwendung ist hier eine Agentenarchitektur zu verstehen, die eine Protokollrepräsentation mit Hilfe des Ausführungssystems benutzt. Dazu sollen klar definierte Schnittstellen zur Anwendung festgelegt werden. So ist es möglich, daß verschiedene Anwendungen über das Protokollausführungssystem die selbe Protokollrepräsentation benutzen können.

In einem Multi-Agentensystem kann es vorkommen, daß ein Agent nicht mehr auf eine Anfrage eines anderen Agenten antworten kann, weil z.B. ein Kommunikationskanal zusammengebrochen ist. Damit ein Agent in diesem Fall nicht unendlich auf Nachrichten wartet, soll ein Timeoutmechanismus eingebaut werden, mit dem solche Ausnahmesituationen abgefangen werden können.

Aufgrund von Kommunikationsstörungen können falsche Nachrichten eintreffen. Es muß festgelegt werden, wie das System auf solche Nachrichten reagiert.

Schließlich sollen die Komponenten leicht in die beim DFKI entwickelte Agentenarchitektur InteRRaP ([M5] und Kapitel 2.2) unter der Programmiersprache Oz ([O1] - [O4] und Kapitel 2.3) integrierbar sein.

3.1 Bestehende Konzepte

In der Literatur zu Multi-Agentensystemen werden Protokolle häufig in Form endlicher Automaten repräsentiert. Ein Beispiel hierfür ist die Darstellung des Joint-Plan-Negotiation-Protokolls [M4] in Abb. 1, die sehr übersichtlich die Kommunikation zwischen zwei Agenten zeigt. Komplexere Protokolle mit mehr als zwei beteiligten Agenten sind jedoch u.U. nicht mehr als endlicher Automat beschreibbar. In Abb. 2 ist der Ablauf des Contract-Net-Protokolls ansatzweise in dieser Form dargestellt. In einem endlichen Automaten kann man nur

durch den aktuellen Zustand beschreiben, welcher Stand der Abarbeitung das Protokoll gerade hat. Im Automaten aus Abb. 2 wird auf Nachrichten (Inform) von den Agenten gewartet, die am Anfang mit Announce angesprochen wurden. Da das Protokoll erst weiterläuft (Nachrichten Grant und Reject), wenn alle geantwortet haben und nicht bestimmt werden kann, in welcher Reihenfolge sie das tun werden, müßten für jede mögliche Reihenfolge des Eintreffens der Nachrichten Zustände vorgesehen werden. Mit der Anzahl der an einem Protokoll beteiligten Agenten, steigt die Anzahl der benötigten Zustände dabei exponentiell. Dies wird bereits bei wenigen angesprochenen Bietern sehr unübersichtlich. Wird jedoch die Ausschreibung (Announce) am Anfang mittels Broadcast[1] an eine unbekannte Anzahl Agenten gesendet, kann nicht mehr bestimmt werden, wieviele Zustände benötigt werden, weil die Anzahl der möglicherweise antwortenden Bieter unbekannt ist. Damit sind die Darstellungsmöglichkeiten eines endlichen Automaten erschöpft.

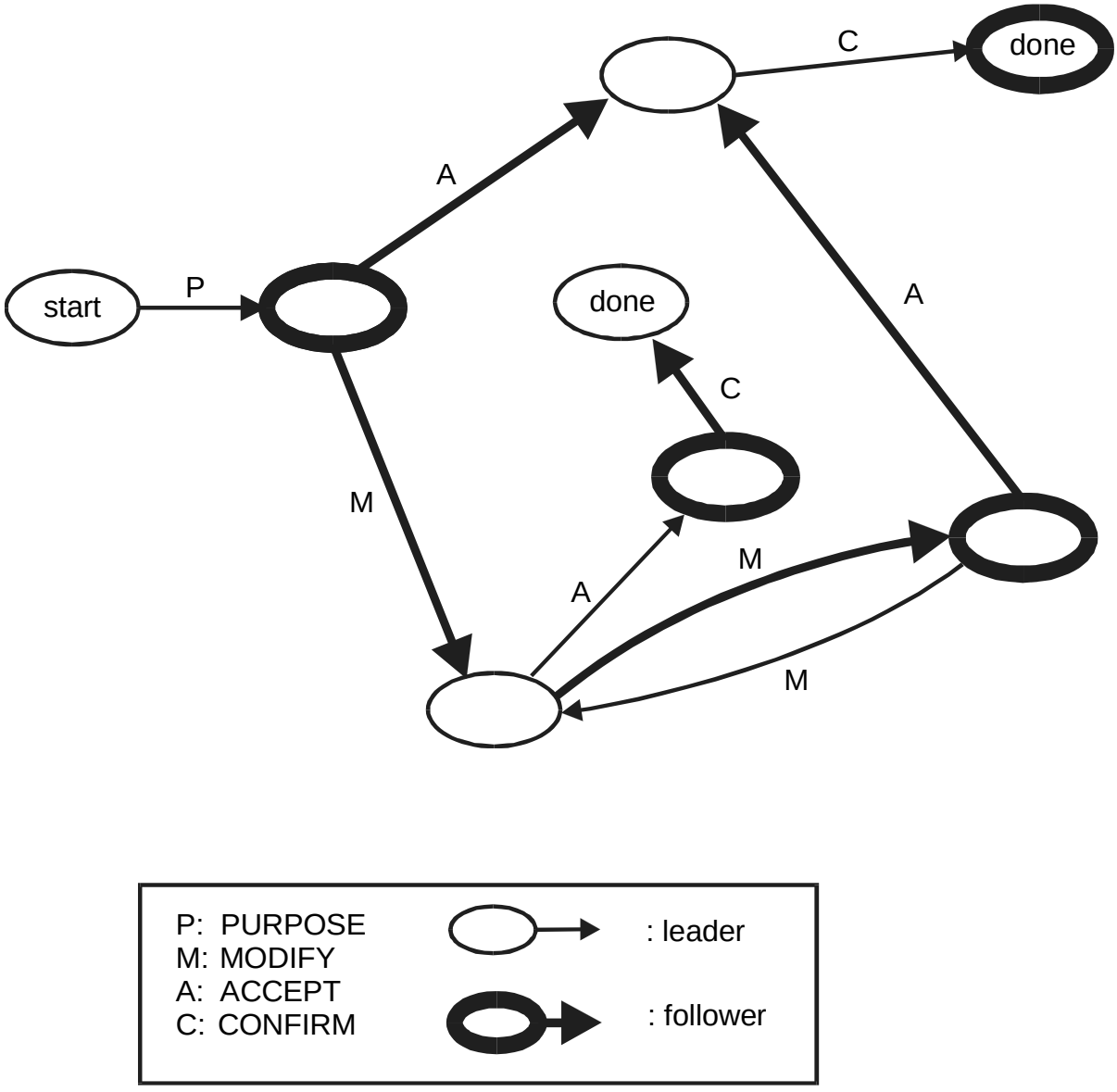

Abb. 1: Darstellung des Joint-Plan-Negotiation-Protokolls als NDA [M4]

[1] Beim Broadcast wird wie beim Rundfunk an alle in einem bestimmten Bereich gesendet. Wieviele genau diese Nachricht, bzw. das Radioprogramm empfangen, ist unbekannt.

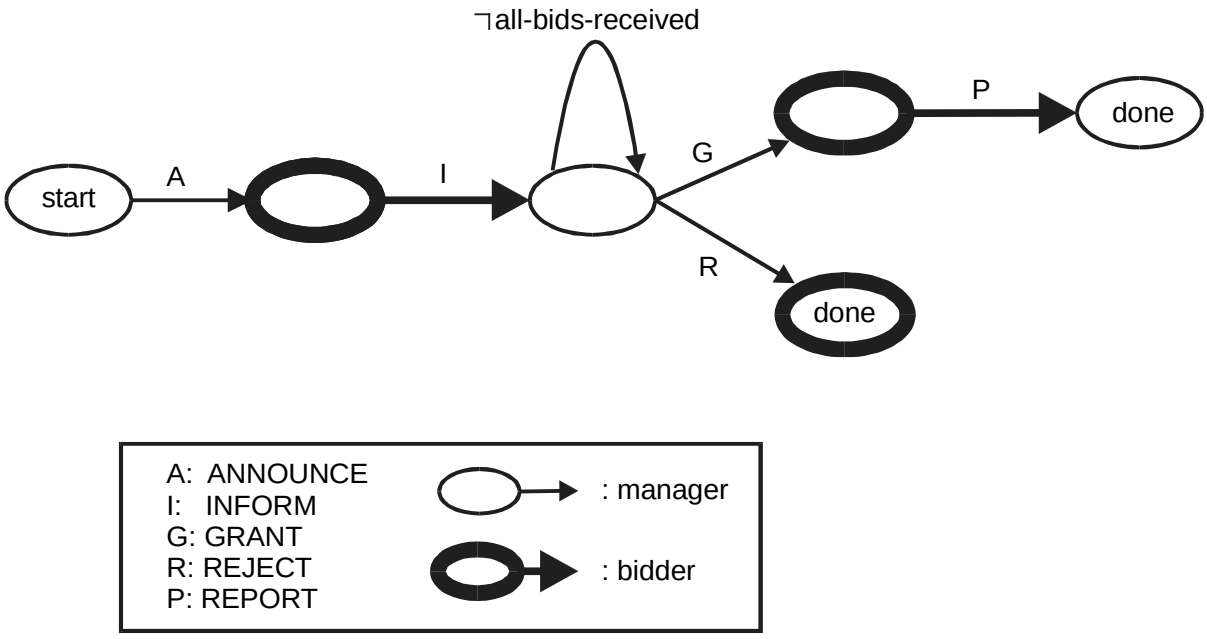

Abb. 2: Darstellung des Contract-Net-Protokolls [M4]

Komplexere Spezifikationsmöglichkeiten finden sich in anderen Gebieten der Informatik. So basieren z.B. die hauptsächlich im Gebiet der Telekommunikation eingesetzten Spezifikationssprachen für verteilte Systeme Estelle und SDL (siehe [S1] und Kapitel 2.4) auf dem Konzept der erweiterten, endlichen Automaten. Hier können in den Zuständen der Automaten zusätzlich Variablenwerte gespeichert werden. Dadurch ist eine Modellierung komplexerer Vorgänge möglich. Jedes beteiligte System wird hier durch einen solchen erweiterten, endlichen Automaten dargestellt. Die Automaten aller Beteiligten bilden zusammen das Protokoll. Der Sprachumfang bietet sehr umfangreiche Möglichkeiten zur Darstellung sehr komplexer Protokolle und Dienste, die in mehreren Schichten aufeinander aufbauen können. Die Spezifikation erfolgt textuell. Zum Teil existiert auch eine entsprechende grafische Darstellung. Bei Estelle und SDL handelt es sich jedoch nur um reine Spezifikationssprachen, die exakt einen Ablauf beschreiben. Eine Schnittstelle zu einer Anwendung, die an bestimmten Stellen zum Zug kommt, kann hier nicht festgelegt werden.

4 Grundsätzliche Entwurfsentscheidungen

Von Estelle (siehe Kapitel 2.4) wird das Konzept der erweiterten, endlichen Automaten übernommen, d.h. Automaten, die zusätzlich Variablenwerte speichern können. Die Repräsentation als Automat liegt nahe, da ein Protokollablauf durch die eintreffenden Nachrichten Sprünge in der Ausführung enthält, die als Zustandsübergänge eines Automaten modelliert werden können. Die Verwendung von Variablen stellt sicher, daß die Protokollsprache mächtig genug ist, um komplexere Protokolle zu beschreiben (siehe Kapitel 3.2). Jede beteiligte Rolle wird durch einen Automaten dargestellt. Alle Automaten der Agenten bilden zusammen das Protokoll. Transitionen werden durch eintreffende Nachrichten ausgelöst. Nach einer Transition können Nachrichten versendet und Variablenwerte verändert werden. Die Möglichkeit einer Dienst-Beschreibung, wie in Estelle wird nicht vorgesehen, da dies für die gängigen Multiagenten-Protokollen nicht benötigt wird. Die in

Estelle verwendeten Elemente wie Interaktionspunkte und Kanäle werden in ihrer Anwendung modifiziert und auf den Anwendungsbereich der Multiagenten-Protokolle zugeschnitten, damit die hier auftretenden Abläufe leicht beschrieben werden können. Dadurch, daß die zu entwickelnde Sprache weniger Möglichkeiten als Estelle bietet (z.B. kein dynamisches Verändern von Kanälen und kein Nichtdeterminismus), wird eine automatische Code-Erzeugung möglich.

Um die bei einem Multiagenten-Protokoll auftretenden Daten leicht speichern zu können, stehen neben den aus anderen Programmiersprachen gewohnten Datentypen `integer`, `boolean` und `string` auch Typen zur Speicherung von Nachrichten (Typ `message`), beliebigen Nachrichteninhalten (Typ `any`) und Agentenreferenzen (Typ `agent`) zur Verfügung. Als Agentenreferenz ist hier ein Wert zu verstehen, über den ein Agent identifiziert werden kann, z.B. eine laufende Nummer, ein Name oder eine Speicheradresse. Dies kann von Anwendung zu Anwendung verschieden sein und wird innerhalb des Protokolls nicht betrachtet. Außerdem können Variablenlisten der verschiedenen Datentypen erzeugt werden. Die hierfür vorgesehenen Datentypen beginnen jeweils mit dem Namen des Grundtyps, gefolgt von `List`, z.B. `agentList` oder `messageList`. Es kann angegeben werden, für welche Variablen Initialwerte bei der Instantiierung eines Protokolls übergeben werden sollen, z.B. der Wert für eine Timeoutfrist oder eine Liste der anzusprechenden Agenten.

Der Benutzer implementiert anwendungsabhängige Abläufe in speziellen Prozeduren, den sog. Anwendungs-Prozeduren (engl.: application procedures), auf die im Protokoll zugriffen werden kann. Sie dienen als Schnittstelle zwischen Protokoll und Anwendung und werden in der Klasse `ApplicationProc` zusammengefaßt. So wird z.B. beim Contract-Net-Protokoll der Ablauf Announce, Bid, Grant und Reject im Protokoll angegeben. Es ist jedoch sinnvoll, die Entscheidung über Grant oder Reject in einer Prozedur zu treffen, da sie von der konkreten Agenten-Implementierung abhängt. Wird während des Protokollablaufs direkt auf eine Prozedur zugegriffen, so ist die Ausführung des Protokolls für diesen Agenten für die Zeit der Prozedurausführung suspendiert. Soll jedoch in diesem Zeitraum weiterhin auf Nachrichten von anderen Agenten gewartet werden, weil z.B. eine größere Berechnung durchgeführt wird, die von anderen Agenten unterbrochen werden kann, so besteht die Möglichkeit, Prozeduren aufzurufen, indem im Protokoll eine Nachricht mit dem Namen der Prozedur und ihren Parametern an die Klasse `ApplicationProc` gesendet wird. Das Ergebnis der Prozedur kommt dann auch als Nachricht von dieser Klasse zurück. Diese Nachrichten werden genauso behandelt, wie Nachrichten von oder zu anderen Agenten. So werden z.B. auch durch eintreffende Nachrichten von Anwendungs-Prozeduren Transitionen im Protokollablauf ausgelöst. So können flexible Protokolle mit paralleler Ausführung realisiert werden, ohne neue Sprachkonstrukte einzuführen.

Damit Agenten nicht unendlich auf Nachrichten warten, besteht die Möglichkeit, Zustandswechsel in Abhängigkeit einer Timeoutfrist auszuführen.

Ein Agent nimmt während der Ausführung eines Protokolls eine bestimmte Rolle ein. Im Contract-Net-Protokoll sind dies z.B. die Rollen Manager und Bieter (engl. manager und bidder). Die verschiedenen Rollen können untereinander nur bestimmte Nachrichten verschicken. So ist es z.B. unsinnig, wenn ein Manager im Contract-Net-Protokoll ein Angebot an einen Bieter sendet. Deshalb besteht eine Protokollspezifikation aus der Beschreibung der verschiedenen Rollen und der sog. Kanäle. Nachrichten zwischen den verschiedenen Rollen müssen über diese Kanäle laufen. Hierüber wird festgelegt, welche Nachrichten zwischen den verschiedenen Rollen ausgetauscht werden können (siehe dazu Kapitel 5.1).

Es wird hier nur eine textuelle Protokolldarstellung entworfen, die direkt compiliert werden kann. Auf den Entwurf einer grafischen Darstellungsform wird aufgrund der zu erwartenden Komplexität der Darstellung und Verarbeitung verzichtet.

Um Erweiterungen und Variationen bestehender Protokolle besser beschreiben zu können, wurde in einem ersten Entwurf, analog zum Konzept der objektorientierten Programmierung, ein Vererbungsmechanismus zwischen Protokollen vorgesehen. So sollte ein abgeleitetes Protokoll die Transitionen eines Oberprotokolls erben und sie durch weitere ergänzen können. In der objektorientierten Programmierung besteht außerdem die Möglichkeit, eine Methode einer Oberklasse in einer abgeleiteten Klasse durch eine Methode mit der selben Signatur (Anzahl und Typ der Ein- und Ausgabeparameter) zu überschreiben, um so deren Verhalten zu verändern. Übertragen auf eine Protokollbeschreibung würde das bedeuten, daß eine Transition in einem Oberprotokoll in einem abgeleiteten Protokoll durch eine Transition mit der selben Signatur überschrieben werden kann. Zu dieser neuen Transition könnte dann ein anderes, für das abgeleitete Protokoll charakteristisches Verhalten angegeben werden, z.B. weitere Agenten anzusprechen. Die Ausführbarkeit einer Transition wird jedoch, vergleichbar mit einer If-Verzweigung, von Fall zu Fall überprüft (aktueller Zustand, eintreffende Nachricht und weitere Bedingungen). Sie besitzt keine, mit einer Prozedur vergleichbare Signatur. Wegen diesen mangelnden Voraussetzungen wurde ein Vererbungsmechanismus zwischen Protokollen wieder verworfen.

Die Protokolle sollen mittels eines Präcompilers in den Code einer bestehenden Programmiersprache übersetzt werden. Hierfür ist zunächst Oz vorgesehen. Die Protokollsprache wird jedoch, mit Ausnahme von syntaktischen Ähnlichkeiten, unabhängig von der Zielsprache sein, so daß auch eine Übersetzung in eine andere höhere Programmiersprache, z.B. Java, möglich ist. Die Möglichkeit der Protokollverarbeitung mittels eines Interpreters wurde verworfen. Durch den direkten Zugriff des Interpreters auf die Protokollrepräsentation würde zwar der zusätzliche Schritt des Compilierens eingespart, die Interpretation würde jedoch den Protokollablauf verlangsamen. Der Laufzeitvorteil wurde hier höher bewertet.

5 Die Protokollsprache

Die Protokolle werden in einer hier eigens definierten Sprache spezifiziert. Sie ist in ihrer Syntax und Semantik an die Programmiersprache Oz angelehnt. In diesem Kapitel wird zunächst eine nicht-formale Sprachbeschreibung angegeben. Die Anhänge 3 und 4 enthalten den Quellcode für die Parser- bzw. Compilergeneratoren Flex und Bison (GNU-Versionen von Lex und Yacc), die eine formale Beschreibung der Sprache darstellen. Es ist zu beachten, daß alle vom Anwender eingeführten Bezeichner, wie Protokollname, Rollenname oder Variablennamen mit einem Großbuchstaben beginnen müssen. Schlüsselworte beginnen generell mit einem Kleinbuchstaben. Kommentare werden mit % begonnen. Der darauffolgende Text wird vom Compiler bis zum Zeilenende ignoriert. Nachfolgend werden allgemeine Angaben, die im konkreten Fall ersetzt werden müssen, in spitzen Klammern (< >) geschrieben. Optionale Angaben stehen in eckigen Klammern ([]). Als Protokollinstanz wird im Folgenden ein für einen konkreten Agenten in Ausführung befindliches Protokoll bezeichnet. So existieren zur Laufzeit eines Contract-Net-Protokolls i.d.R. eine Protokollinstanz für den Manager und mehrere Instanzen für die Bieter.

5.1 Protokolldefinition

Eine Protokolldefinition wird mit dem Schlüsselwort `protocol` gefolgt vom Protokollnamen begonnen. Darauf folgt die Beschreibung der Kanäle und der beteiligten Rollen. Beendet wird die Definition mit `end` (siehe Beispielspezifikation in Kapitel 6).

Eine Kanaldefinition ist eingeschlossen in die Schlüsselworte `channel` und `end.` Dazwischen stehen die Bezeichner der Rollen, die über diesen Kanal Nachrichten austauschen sollen. In Klammern hinter dem Rollenbezeichner werden die Nachrichten angegeben, die von dieser Rolle geschickt werden können. Der Kanal für das Contract-Net-Protokoll wird z.B. folgendermaßen deklariert.

```
channel
  Manager(Announce Grant Reject)
  Bidder(Bid NoBid Report)
end
```

Wird, wie in diesem Fall, nur der Name der Nachricht angegeben, so handelt es sich um einfache Nachrichten, die nur das Inhaltsfeld `content` vom Typ `any` (Datentyp für einen beliebigen Nachrichteninhalt, siehe Kapitel 4.2.1) aufweisen. Der Anwender kann jedoch einem Nachrichtentyp auch andere Inhaltsfelder zuweisen, um so spezielle Informationen, z.B. bool'sche Werte, übertragen zu können, auf die die empfangende Protokollinstanz dann wieder zugreifen kann. Im Beispiel des Extended-Contract-Net-Protokolls geschieht dies z.B. folgendermaßen:

```
channel
  Manager(Announce Grand Reject Refine NoRefine)
  Bidder( Bid(any TheBid, boolean WantToRefine)
          NoBid Refine Report)
end
```

Hinter dem Nachrichtennamen `Bid` werden hier, getrennt durch Kommata, die Namen der Parameter mit ihren zugehörigen Datentypen angegeben. In diesem Beispiel kann der Bidder durch die bool'sche Variable `WantToRefine` dem Manager mitteilen, ob er noch eine Nachricht vom Typ `Refine` schicken wird. Für ein Inhaltsfeld einer Nachricht können alle Datentypen außer dem Typ `message` verwendet werden. Indem der Typ `message` ausgeschlossen wird, soll verhindert werden, daß Nachrichten als Inhalte in andere Nachrichten gesteckt und so, unter Umgehung des Deklarationszwangs innerhalb des entsprechenden Kanals, doch versendet werden. Dies würde die Verständlichkeit eines Protokolls stark vermindern.

Der Wert eines Inhaltsfeldes einer Nachricht wird zurückgegeben, indem hinter dem Namen der entsprechenden Variable vom Typ `message` ein Punkt und der Name des Inhaltsfeldes angegeben wird. Enthält z.B. die Variable `NewMessage` (vom Typ `message`) eine Nachricht vom oben deklarierten Nachrichtentyp `Announce`, so kann durch

```
NewMessage.content
```

auf das Inhaltsfeld der Nachricht zugegriffen werden. Ist die Nachricht z.B. vom Nachrichtentyp `Bid`, so wird durch

```
NewMessage.WantToRefine
```

der entsprechende bool'sche Wert zurückgegeben.

5.2 Rollendefinition

Die Beschreibung einer Rolle wird durch das Schlüsselwort `role` gefolgt vom Rollenbezeichner eingeleitet und durch Angabe von `end` beendet. Hinter `role` wird in Klammer die Anzahl der möglichen Instanzen (>= 1) der Rolle angegeben, die an diesem Protokoll beteiligt sein können. Das Zeichen * steht für eine unbestimmte Anzahl. Sollen bei der Instantiierung einer Rolle Parameter übergeben werden, so werden diese in Klammern hinter dem Rollenbezeichner genannt. Da hier, anders als in Oz, eine Variable explizit mit ihrem Datentyp deklariert werden muß, wird dieser vor dem Variablenbezeichner angegeben. Mehrere Parameter werden durch Kommata getrennt. Bei der Beispielimplementierung des Contract-Net-Protokolls aus Kap. 6 sieht dies z.B. folgendermaßen aus:

```
role(1) Manager(agentList BidderList,
                any Task, integer TimeoutValue)
```

Eine Rollendefinition besteht aus einem Deklarationsteil und der Beschreibung der Transitionen. Innerhalb einer Transition, dem sog. Transitionsblock, können verschiedene Aktionen ausgeführt, z.B. Nachrichten gesendet oder Variablenwerte manipuliert, werden. Es besteht die Möglichkeit, solche Aktionen in einem Makro zusammenzufassen. Siehe hierzu Kapitel 5.2.3.

Die Teile Deklaration, Transition und Aktion werden im Folgenden näher beschrieben.

5.2.1 Deklarationen

Die zu verwendenden Anwendungs-Prozeduren, Zustände und Variablen müssen zu Beginn einer Rollendefinition deklariert werden.

Dazu werden zunächst die Anwendungs-Prozeduren über einen Kanal zwischen der Protokollinstanz (Schlüsselwort `protocol`) und der Protokoll-Prozedur-Klasse (Schlüsselwort `applicationProc`) deklariert. Dabei gelten die selben syntaktischen Regeln wie bei der Kanaldefinition zwischen zwei Protokoll-Rollen. Jede Prozedur, die im Protokoll verwendet werden soll, wird mit ihrem Namen und ihren Parametern als Nachricht von `protocol` angegeben. Die Protokollinstanz schickt, analog zur Sichtweise der objektorientierten Programmierung, den Prozeduraufruf als Nachricht an die Klasse, die die Prozedur enthält, hier die Klasse `ApplicationProc`. Wird ein Ergebnis von einer Prozedur erwartet, so wird dieses Ergebnis hier als Nachricht von `ApplicationProc` deklariert. Die Nachricht wird mit dem Namen der Prozedur mit einem

vorgestellten `Result` bezeichnet. Auch bei der Angabe von Parametern gelten die selben Regeln, wie bei den Rollen-Kanälen.

Der Manager im Contract-Net-Protokoll deklariert seine Anwendungs-Prozeduren z.B. folgendermaßen:

```
channel
    protocol(ChooseBid(any Bid1, any Bid2)
             Inform(string Message)
             GetResult)

    applicationProc(ResultChooseBid)
end
```

Die Prozeduren `ChooseBid` und `Inform` haben hier eigens definierte Eingabe-Parameter. `GetResult` hat nur den Standardparameter `content` vom Typ `any`. Die Prozedur `ChooseBid` hat außerdem einen Rückgabewert, der nach Ausführung der Prozedur in der Rückgabenachricht `ResultChooseBid` an die Protokollinstanz gesendet wird. Da für diese Nachricht keine weiteren Parameter angegeben wurden, enthält sie den Standardparameter `content` vom Typ `any`. Es können jedoch auch bei den Rückgabenachrichten andere Parameter definiert werden.

Nach der Deklaration der Anwendungs-Prozeduren werden die Zustände deklariert. Dazu werden die Bezeichner der benutzten Zustände zwischen den Schlüsselworten `states` und `end` angegeben. Manchmal kann es sinnvoll sein, daß innerhalb eines Zustandes alle nicht erwarteten Nachrichten, also Nachrichten, für die keine Transitionen aus diesem Zustand angegeben wurden, zunächst in einem FIFO-Puffer zwischengespeichert (gestaut) und erst abgearbeitet werden, wenn dieser Zustand wieder verlassen wird. Dies kann z.B. der Fall sein, wenn in einem Zustand auf das Ergebnis einer Anwendungsprozedur gewartet wird, jedoch auch Nachrichten von anderen Agenten eintreffen können, die danach bearbeitet werden sollen. Um solche nicht erwarteten Nachrichten zu stauen, muß innerhalb der Zustandsdeklaration hinter dem Zustandsnamen in Klammern `queueUnexpectedMessages` angegeben werden.

Zwischen `declare` und `end` werden die Variablen der verschiedenen Datentypen deklariert. Dazu werden jeweils hinter dem Namen des Datentyps die einzuführenden Variablen angegeben. Die Deklaration wird durch `end` abgeschlossen.

Es sind folgende Datentypen vorgesehen:

`boolean` : kann die Wahrheitswerte `true` und `false` annehmen; Initialwert: `false`

`integer` : natürliche Zahlen; Initialwert: `0`

`string` : eine Zeichenkette; Initialwert: `nil` (leerer String)

message : eine Nachricht; Initialwert: `nil`

agent : Eine Referenz auf einen Agenten. Welcher Art diese Referenz ist, ist anwendungsabhängig (z.B. eine Nummer oder eine Referenz auf ein Oz-Objekt, das den Agenten darstellt) und wird innerhalb der Spezifikation nicht betrachtet. Initialwert: `nil`

any : Variablen des Typs any können beliebige, anwendungsabhängige Nachrichteninhalte aufnehmen, auf die innerhalb des Protokolls nicht zugegriffen werden muß. Initialwert: nil

booleanList, integerList, stringList, messageList, agentList, anyList: Listen, die Werte der jeweiligen Datentypen aufnehmen können. Initialwert: nil

Die auf den Datentypen definierten Operationen werden im Kapitel 5.2.3 beschrieben.

Hier das Beispiel des Deklarationsteils der Rolle Manager innerhalb des Contract-Net-Protokolls:

```
channel
  protocol(ChooseBid(any Bid1, any Bid2)
           Inform(string Message)
           GetResult)
  applicationProc(ResultChooseBid)
end

%my states
states
  WaitingForAnswer
  WaitingForReport
end


declare
  %my variables
  message
    BestBid % the currently best bid
    Result
  end
  agent
    Contractor
  end
end
```

5.2.2 Transitionen

Durch `initialize to <Zustand>` wird die erste Transition des Protokollablaufs für eine Rolle beschrieben (initiale Transition). Wie bei den übrigen Transitionen kann die Ausführung durch Angabe von `provided <Bedingung>` von einem bool'schen Wert abhängig gemacht werden. Danach können zwischen `begin` und `end` Aktionen angegeben werden, die bereits nach dem Übergang zum Startzustand ausgeführt werden sollen. Kann keine initiale Transition ausgeführt werden, kommt es zu einem Laufzeitfehler (siehe Kapitel 5.3.2). Beim Contract-Net-Beispiel aus Kapitel 6, gibt es folgende initialen Transitionen:

```
initialize to WaitingForAnswer
provided (length(BidderList) >= 1)
begin
  % ...
end

initialize to done
provided (length(BidderList) < 1)
begin
  % ...
end
```

Der Bezeichner `WaitingForAnswer` wurden zu Beginn der Rollendefinition als Zustand und `BidderList` als Übergabeparameter vom Typ `agentList` deklariert. `done` steht für den Endzustand. Die Operation `length` gibt die Anzahl der Elemente in einer Liste zurück. Nach einer Transition zum Endzustand `done` wird die Protokollbearbeitung für diese Protokollinstanz abgebrochen. Mit diesen initialen Transitionen wird also überprüft, ob die übergebene Liste der anzusprechenden Bieter (in der Variable `BidderList`) mindestens ein Element enthält. Ist dies nicht der Fall, stellt dies einen Fehler dar, und die Protokollbearbeitung wird sofort abgebrochen, ansonsten wird in den Zustand `WaitingForAnswer` übergegangen.

Nun folgen, eingeschlossen von `trans` und `end`, die übrigen Transitionen. Diese haben i.A. die Form:

```
from <AusgangsZustand> to <ZielZustand>
[provided <bool'scher Wert>]
when <Rolle>.<Agent> sends <Nachricht>
begin
  <Aktionen>
end
```

Bei <Rolle> wird die Protokollrolle angegeben, von der eine Nachricht erwartet wird. Für <Agent> kann eine Variable vom Typ agent stehen, die bereits eine konkrete Agentenreferenz enthält. Der Übergang wird dann nur ausgeführt, wenn eine Nachricht von diesem Agenten eintrifft. Soll auf mehrere Agenten gewartet werden, kann eine Variable vom Typ agentList, oder anyAgent für beliebige Agenten angegeben werden. Für <Nachricht> stehen eine oder mehrere Nachrichtentypen oder anyMessage. In der entsprechenden Kanaldefinition muß angegeben worden sein, daß der Agent, von dem die Nachricht erwartet wird, diesen Nachrichtentyp schicken kann.

Bei Transitionen, die durch Nachrichten von Anwendungs-Prozeduren ausgelöst werden, wird applicationProc anstatt <Rolle>.<Agent> angegeben. In folgendem Beispiel wird auf das Ergebnis der Prozedur CheckTask gewartet.

```
from CheckingTask to WaitingForAnswer
when applicationProc sends ResultCheckTask
   %received bid from applicationProc
begin
  % ...
end
```

Als Ausgangszustand kann außerdem anyState für einen beliebigen Zustand angegeben werden. Dies kann bei der Behandlung von Fehlern im Protokollablauf sinnvoll sein (siehe Kapitel 5.3). Steht das Schlüsselwort same für den Zielzustand, wird wieder in den bisherigen Zustand übergegangen.

Es ist möglich, daß beim Eintreffen einer Nachricht mehrere Transitionen gleichzeitig ausführbar sind. Um hier nichtdeterministisches Verhalten zu verhindern, werden beim Eintreffen einer Nachricht die Transitionen, die vom aktuellen Zustand ausgehen, sequentiell von oben nach unten überprüft. Die erste ausführbare Transition wird dann ausgewählt und alle weiteren nicht mehr beachtet.

5.2.3 Aktionen

Als Aktionen, die zu einer Transition angegeben werden können, sind das Senden von Nachrichten und das Manipulieren von Variablen möglich. Diese Aktionen können mit Schleifen und Verzweigungen verknüpft werden. currentMessage steht für die zuletzt empfangene Nachricht, die die Transition ausgelöst hat und sender für den zugehörigen Agenten.

Aktionen können auch in einem Makro zusammengefaßt werden. Dazu wird hinter den Transitionen das Schlüsselwort macro, gefolgt vom Bezeichner des Makros angegeben. Nach den eigentlichen Aktionen wird das Makro durch end abgeschlossen. Das Makro wird durch insert(<Makroname>) aufgerufen.

5.2.3.1 Versenden von Nachrichten

Nachrichten werden durch den Befehl

```
send <Nachricht> to <Rolle>.<Agent>
```

an andere Agenten verschickt.

Dabei kann `<Nachricht>` entweder eine explizit angegebene Nachricht, z.B. `Bid`, oder eine Variable vom Typ `message` oder `messageList` sein. Der Inhalt der Nachricht wird in Klammern hinter dem Nachrichtennamen angegeben. Leere Klammern stehen für eine Nachricht ohne Inhalt.

Für `<Agent>` wird eine Variable des Typs `agent` angegeben, die eine Referenz auf den Agenten enthält, an den die Nachricht gesendet werden soll. Soll an mehrere Agenten gesendet werden, kann hier eine Variable vom Typ `agentList` angegeben werden, oder das Schlüsselwortes `all` für alle anderen Agenten. `<Rolle>` bezeichnet die Rolle, die der Empfänger einnimmt. Dabei muß in der Kanaldefinition festgelegt worden sein, daß an diese Rolle diese Nachricht gesendet werden kann. Im Beispiel aus Kapitel 6.1 sendet der Manager die Ausschreibung (`Announce`) folgendermaßen an die Bieter:

```
send Announce(Task) to Bidder.BidderList
```

`Task` ist vom Datentyp `any` und enthält die Daten zur Ausschreibung. Da der Nachrichtentyp ohne weitere Inhaltsfelder deklariert wurde, wird damit das Feld `content` der Nachricht gefüllt. `BidderList` ist vom Typ `agentList` und enthält die Referenzen auf alle anzusprechenden Agenten, die die Rolle `Bidder` einnehmen sollen.

Auch Anwendungs-Prozeduren können durch das Versenden von Nachrichten aufgerufen werden. So reicht ein im Beispiel angesprochener Bieter durch

```
send CheckTask(MyTask) to applicationProc
```

die Ausschreibung (in der Variable `MyTask`) an die Protokoll-Prozedur `CheckTask` weiter. Da in der Kanaldefinition zwischen der Protokollinstanz und der Klasse `ApplicationProc` auch die Nachricht `ResultCheckTask` deklariert wurde, wird das Ergebnis der Prozedur als Nachricht diesen Nachrichtentyps an die Protokollinstanz zurückgesendet.

Die Parameter der Prozeduren werden in den Inhaltsfeldern der Nachrichten übergeben. Da hier die selben Regeln gelten, wie bei Nachrichten zwischen verschiedenen Protokoll-Rollen, hat eine Prozedur als Standard-Wert den Parameter `content`. Dieser kann jedoch auch weggelassen werden, wenn eine Prozedur ohne Parameter aufgerufen werden soll.

Soll die Ausführung des Protokolls für eine Protokollinstanz stoppen, solange eine Prozedur ausgeführt wird, kann das Senden an die Klasse `ApplicationProc` und das Empfangen des Ergebnisses auch in einem Schritt zusammengefaßt und damit in einem Aktionsblock einer Transition ausgeführt werden. Die Syntax gleicht hierbei Prozeduraufrufen herkömmlicher Programmiersprachen. Im Beispiel des Contract-Net-Protokolls

aus Kapitel 6.1 wählt der Manager z.B. folgendermaßen das beste Angebot aus zwei bestehenden Angeboten aus:

```
Result = applicationProc(ChooseBid(BestBid.content
                                   currentMessage.content))
```

Die Prozedur `ChooseBid` wurde mit zwei Parameter im Prozedur-Kanal der Rolle deklariert. Die dort deklarierte Rückgabenachricht `ResultChooseBid` wird als Ergebnis der Prozedur der Variable `Result` vom Typ `message` zugewiesen.

5.2.3.2 Variablenmanipulation

Auf den Datentypen sind folgende Operatoren definiert. Der Bezeichner vor der Prozedur zeigt den Rückgabetyp an. `void` steht für "kein Rückgabewert".

`message`:

`agent` **`getSender`**`(message)`: Liefert den Absender einer Nachricht.

`List` (Gemeinsame Operationen für Listen aller Datentypen, hier durch den Typ `List` zusammengefaßt. `<Element>` steht für ein Element des jeweiligen Datentyps.) :

`void` **`put`**`(<Element> List)`: Der Wert wird in die Liste geschrieben.

`<Element>` **`get`**`(List)`: Der Wert des zuerst in die Liste eingefügten Elements wird zurückgegeben. Die Liste bleibt erhalten.

`void` **`remove`** `(<Element> List)`: Das Element wird aus der Liste entfernt.

`boolean` **`member`** `(<Element> List)`: Prüft, ob das Element in der Liste enthalten ist.

`integer` **`length`**`(List)`: Gibt die Anzahl der Elemente in der Liste zurück.

Den Variablen der verschiedenen Datentypen werden durch den Zuweisungsoperator '=' Werte zugewiesen. Variableninhalte werden durch '==' und '\=' verglichen. Integerwerte können zusätzlich durch '<', '=<', '>=' und '>' verglichen werden. Die Rückgabewerte sind vom Typ `boolean`. Durch '!' vor einer bool'schen Variable wird der Wahrheitswert umgekehrt.

Durch Angabe des Variablennamens kann auf den Wert einer Variablen zugegriffen werden.

5.2.3.3 Schleifen und Verzweigungen

Zur Realisierung von Schleifen steht folgendes Konstrukt zur Verfügung:

```
while <Wahrheitswert>
  <Aktionen>
end
```

Hier werden die angegebenen Aktionen solange ausgeführt, bis der angegebene Wahrheitswert vom Typ `boolean` den Wert `false` hat.

Für Verzweigungen kann folgendes Konstrukt verwendet werden:

```
if <Wahrheitswert1> then
  <Aktionen1>
elseif <Wahrheitswert2> then
  <Aktionen2>
else
  <Aktionen3>
end
```

Hier wird in Abhängigkeit der Wahrheitswerte zu den verschiedenen Aktionen verzweigt.

5.3 Fehlerbehandlung

Zur Fehlerbehandlung zählen der Timeoutmechanismus und die Ausnahmebehandlung (Exceptions). Wurde im Protokoll ein Fehler erkannt, kann durch `exit` die Ausführung des Protokolls für diese Protokollinstanz beendet werden.

5.3.1 Timeoutmechanismus

Der Anwender kann Timeoutfristen vereinbaren, um nach einer gewissen Zeit das Warten auf eine oder mehrere Nachrichten abzubrechen. Nachrichten können z.B. ausbleiben, weil der Kontakt zu einem anderen Agenten abgebrochen ist, oder dieser wegen anderer Aufgaben nicht in der erwarteten Zeit reagieren konnte.

In einem ersten Entwurf wurde eine an Estelle (siehe [S1] und Kapitel 2.4) angelehnte Timersyntax vorgesehen. Dort wird z.B. mit

```
from WARTEN to UNTERBROCHEN
  delay (5)
begin
...
end
```

vom Zustand WARTEN mit einer Verzögerung von 5 Sekunden in den Zustand UNTERBROCHEN übergegangen, sofern der Zustand WARTEN nicht bereits verlassen wurde. Diese Syntax suggeriert jedoch, daß es sich hier um eine gewöhnliche Transition handelt, die lediglich zeitverzögert ausgeführt wird. Der Timeout-Fall stellt jedoch in der Protokollsprache eine Ausnahme dar, z.B. wenn ein Agent nicht mehr reagiert. Dies sollte auch in der Syntax ersichtlich sein. Eine Möglichkeit besteht darin, den Timer im Aktionsblock einer Transition explizit zu starten. Der Timer wird dann entweder automatisch gestoppt, sobald der Zustand, in dem der Timer gestartet wurde, wieder verlassen wird, oder das Timersignal trifft als Nachricht vom System ein. Da jedoch die Timeoutfrist für einen bestimmten Zustand gelten soll, muß, wenn mehrere Transitionen in einen timerabhängigen Zustand übergehen, der Timer auch in jeder dieser Transitionen gestartet werden. Dies macht den Mechanismus unübersichtlich und schwer handhabbar. Außerdem kann eine Timeoutfrist, die für mehrere Zustände gilt, so nur schwer realisiert werden.

Aus diesen Gründen, werden Timeoutfristen separat und in Abhängigkeit der Zustände, für die sie gelten sollen, angegeben. Eine Frist gilt dann für den oder die angegebenen Zustände. Der Timer wird gestoppt, sobald ein anderer Zustand erreicht wird. Die Timeoutfristen werden zwischen dem allgemeinen Deklarationsteil und den initialen Transitionen angegeben. Der Manager im Contract-Net-Beispiel aus Kapitel 6 führt z.B. Timeoutfristen für die Zustände `WaitingForAnswer` und `WaitingForReport` ein.

```
timeout
  Timeout1 TimeoutValue startOnEnter in WaitingForAnswer end
  Timeout2 TimeoutValue startOnEnter in WaitingForReport end
end
```

Hier sind `Timeout1` und `Timeout2` die Namen der Timeoutfristen. Läuft eine Frist ab, wird eine Nachricht mit dem Namen der Frist an diese Protokollinstanz gesendet. Diese Timeout-Nachricht kann dann wie andere Nachrichten behandelt werden. Die Variable `TimeoutValue` vom Typ `integer` enthält den Zeitraum in Millisekunden. `startOnEnter` bedeutet, daß der Timer nur einmal beim Erreichen des Zustandes gestartet wird. Der Bieter aus dem Beispiel des Extended-Contract-Net verwendet hier `startOnTransition`.

```
timeout
  Timeout1 12000
    startOnTransition   % reset time on every transition
    in WaitingForAnswer WaitIfRefine
  end
end
```

Dabei wird die Timeoutfrist bei jeder Transition innerhalb der angegebenen Zustände neu gestartet, also auch bei Transitionen der Zustände zu sich selbst und zwischen `WaitingForAnswer` und `WaitIfRefine`.

5.3.2 Ausnahmebehandlung

Ausnahmen (Exceptions) dienen dazu, Informationen über Fehler im Protokollablauf zwischen den verschiedenen Protokollinstanzen auszutauschen. Dazu wird die vordefinierte Nachricht `Exception(string Text)` verwendet. Sendet eine Protokollinstanz z.B. eine Nachricht, die beim Empfänger nicht erwartet wird (keine passende Transition aus dem aktuellen Zustand), so wird vom System eine Exception mit dem Text "message not expected" zurückgeschickt. Hat der Partner die Protokollausführung bereits beendet, so wird dies durch den Text "partner ended protocol execution" mitgeteilt. Aber auch innerhalb eines Protokolls können Exceptions gesendet werden, wenn ein anderer Agent mit einer eigenen Nachricht über eine Ausnahme informiert werden soll. So informiert der Manager im Beispiel des Contract-Net-Protokolls einen Bieter, dessen Angebot zu spät, d.h. nachdem durch einen Timeout das Warten auf Angebote beendet wurde, eintrifft, folgendermaßen mittels einer Exception-Nachricht über den Sachverhalt:

```
send Exception("bid arrived too late") to sender
```

Trifft eine Exception-Nachricht ein, so wird im Normalfall die Prozedur `handleException`, die in der Klasse `ApplicationProc` vom Anwender implementiert wird, mit dem Text der Exception aufgerufen. Da eine Exception i.d.R. einen schwerwiegenden Fehler im Protokollablauf darstellt, wird daraufhin die Abarbeitung des Protokolls für diese Instanz beendet. Der Abbruch kann jedoch verhindert werden, indem die Exception im Protokoll abgefangen wird. Dazu muß eine Transition angegeben werden, die durch das Eintreffen der Exception-Nachricht ausgelöst wird. So fängt z.B. der Bieter im Beispiel des Extended-Contract-Net-Protokolls folgendermaßen die Nachricht "bid arrived too late" ab und geht in den Endzustand über.

```
from WaitingForAnswer WaitIfRefine to done
when Manager.MyManager sends Exception
provided (currentMessage.Text == "message arrived too late")
begin
   %
end
```

Die Variable `MyManager` enthält dabei die Referenz auf den Agenten in der Rolle `Manager`, mit dem bisher kommuniziert wurde.

Folgende Exceptions können vom System gesendet werden:

Text	**Bedeutung**	**Empfänger**
no suitable initialisation found	keine der initialen Transitionen konnte ausgeführt werden	`ApplicationProc` der neu erzeugten `PEU`[2]

[2] PEU: Protocol Excecution Unit, dt.: Protokoll-Ausführungs-Einheit, siehe Kapitel 7.1

no suitable transition found	zur eingehenden Nachricht konnte keine passende Transition gefunden werden	PEU, die die Nachricht gesendet hat
can't send message	CU[3] konnte Nachricht der PEU nicht senden	PEU, die die Nachricht senden wollte
no suitable PEU	zur eingehenden Nachricht konnte keine passende PEU gefunden werden; evtl. hat die entsprechende PEU die Protokollverarbeitung bereits beendet	PEU, die die Nachricht gesendet hat
can't create new PEU	ein Agent erhielt die erste Nachricht einer neuen Protokollausführung, konnte jedoch keine passende PEU erzeugen	PEU, die die Nachricht gesendet hat

6 Eine Beispielspezifikation

Als Beispiel wird hier eine Spezifikation des Contract-Net-Protokolls und des Extended-Contract-Net-Protokolls angegeben. Es gibt viele Variationen dieser Protokolle, die sich in ihrer Implementierung, z.B. im Umfang ihrer Fehlerbehandlung unterscheiden. Allen gemeinsam ist jedoch der in Kapitel 2.1 beschriebene grundsätzliche Ablauf. Hier wurde darauf geachtet, das Protokoll möglichst realitätsnah zu beschreiben, z.B. im Hinblick auf die Fehlerbehandlung. Dadurch konnten viele der hier entworfenen und in der Praxis benötigten Konstrukte, z.B. Timeout und Exceptions, in der Beispielspezifikation verwendet werden. In Abb. 3 bis 6 sind die Abläufe als Automaten dargestellt. Maßnahmen zur Ausnahmebehandlung wurden hier zur Übersichtlichkeit weggelassen. An den Transitionen ist jeweils hinter dem Bezeichner `IN` die eintreffende und damit auslösende und hinter `OUT` die entsprechende abgehende Nachricht dargestellt. Fehlt eine solche Nachricht, ist `nil` angegeben. Jede Nachricht wird durch den Nachrichtentyp und den vorgestellten Namen der Rolle des Absenders bei eingehenden Nachrichten, bzw. der Rolle des Empfängers bei abgehenden Nachrichten bezeichnet. Nachrichten in eckigen Klammern [ ] sind optional. Beim Übergang zum ersten Zustand können noch keine Nachrichten empfangen werden. Die hier gezeigt Darstellungsweise deckt noch nicht alle Abläufe ab, da die Aktionen auf Variablen hier nicht gezeigt sind. Sie ist jedoch gut geeignet, sich einen ersten Überblick zu verschaffen. Zum besseren Verständnis sind teilweise Erläuterungen angegeben. Sie sind eingeschlossen in die Kommentar-Zeichen `/*` und `*/`. Auf die Grafiken folgt die Spezifikation in der Protokollsprache.

[3] CU: Communication Unit: dt.: Kommunikations-Einheit, siehe Kapitel 7.1

6.1 Das Contract-Net-Protokoll

Der Manager (Abb. 3) sendet zuerst die Nachricht Announce an n Bieter. In dieser Implementierung des Contract-Net-Protokolls (siehe Kapitel 2.1) prüft der Manager daraufhin, sobald ein Angebot (Nachricht `Bid`) eingeht, ob bereits ein Angebot vorliegt. Ist dies der Fall, vergleicht er die beiden Angebote und sendet dem Bieter mit dem schlechteren Angebot umgehend eine Ablehnung (Nachricht `Reject`). Es steht also zu jedem Zeitpunkt fest, welches das aktuell beste Angebot ist. Signalisiert ein Bieter mit der Nachricht NoBid, daß er kein Angebot abgeben kann, muß der Manager, wenn es sich nicht um die Nachricht des letzten Bieters handelt, keine Absage-Nachricht (Nachricht `Reject`) versenden. Erst wenn die letzte Nachricht eines Bieters eingeht, wird in den Zustand `WaitingForReport` übergegangen. Ist die letzte Nachricht ein Angebot (Nachricht `Bid`), wird wiederum aus den beiden vorliegenden Angeboten das beste ausgewählt. Dem schlechteren Bieter wird eine Ablehnung geschickt (Nachricht `Reject`), dem besseren eine Zusage (Nachricht Grant). Ist die letzte Nachricht eines Bieters eine NoBid-Nachricht, muß wiederum lediglich dem Absender des aktuell besten Angebots eine Zusage geschickt werden. Im Zustand WaitingForReport, wird darauf gewartet, daß der Bieter, dem die Zusage erteilt wurde, die Ergebnis-Nachricht (Nachricht `Report`) zu dem ihm erteilten Auftrag zurückschickt. Empfängt er diese Nachricht, sendet er das Ergebnis an seine Anwendungs-Prozedur `GetResult` (Nachricht `applicationProc.GetResult`) und beendet die Protokollausführung durch den Übergang in den Endzustand `done`.

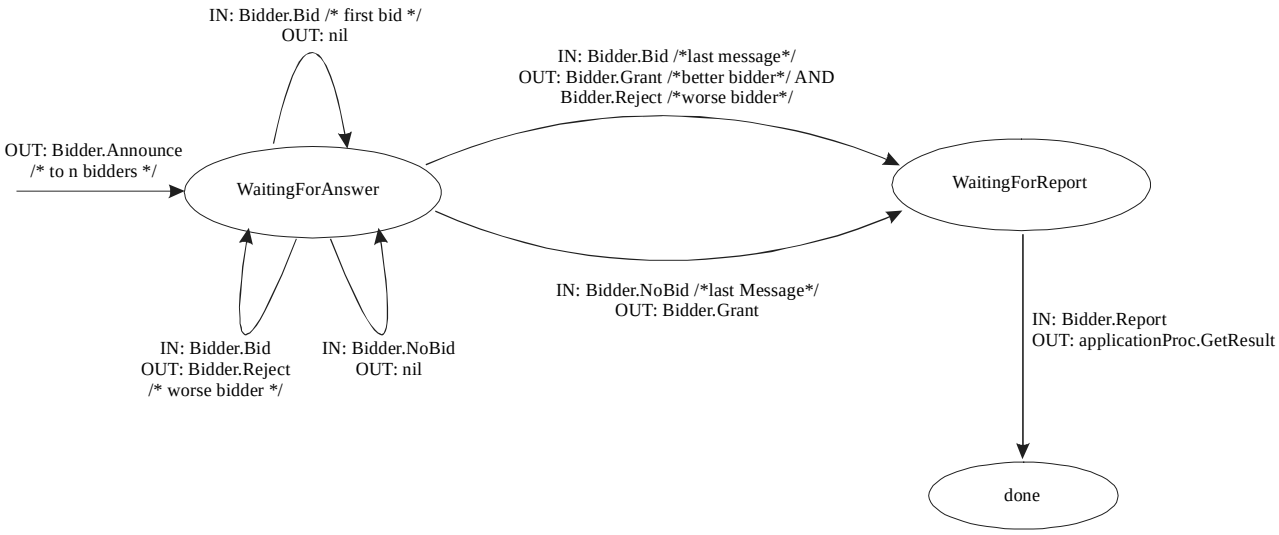

Abb.3: Der Manager im Contract-Net-Protokoll

Der Bieter im Contract-Net-Protokoll (Abb. 4) sendet den Inhalt der empfangenen Ausschreibung (Nachricht `Announce`) an seine Anwendungs-Prozedur `CheckTask`. Diese ermittelt, ob der Agent ein Angebot abgeben kann und sendet das Ergebnis in der Nachricht `ResultCheckTask` zurück. Kann kein Angebot abgegeben werden, schickt der Bieter die Nachricht `NoBid` an den Manager. Ist ein Angebot möglich, so wird es in der Nachricht `Bid` an den Manager gesendet. Wird das Angebot des Bieters abgelehnt (Nachricht `Reject`), informiert er seien Anwendung über die Anwendungsprozedur `Inform` darüber. Wird das Angebot jedoch vom Manager angenommen, wird die Anwendungs-Prozedur `DoTheJob` aufgerufen, die den Auftrag ausführt.

Kommt das Ergebnis der Ausführung in der Nachricht `ResultDoTheJob` zurück, wird es in der Nachricht `Report` an den Manager weitergeleitet und die Protokollausführung daraufhin beendet. Da die Prozeduren `CheckTask` und `DoTheJob` des Bieters eine längere Ausführungszeit benötigen, werden sie nicht direkt aufgerufen, sondern durch eine Nachricht an die Klasse `ApplicationProc` gestartet (siehe Kapitel 5.2.3.1). Dadurch können in dieser Zeit weitere Nachrichten empfangen werden. Dies ist in dieser Implementierung nicht vorgesehen. Man kann sich jedoch vorstellen, daß der Manager hier z.B. die Bearbeitung der Aufgabe (Zustand `DoTheJob`) durch eine entsprechende Nachricht abbrechen kann.

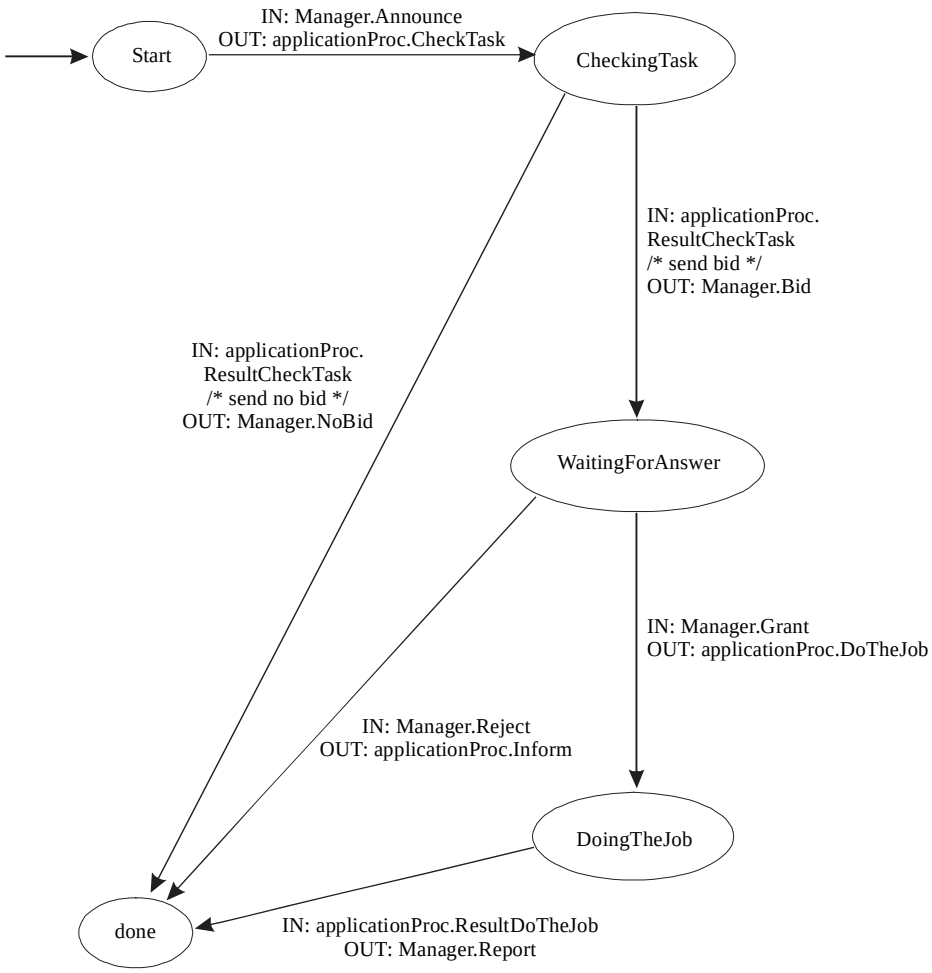

Abb. 4: Der Bieter im Contract-Net-Protokoll

Im Folgenden wird dieser Ablauf in der Protokollsprache beschrieben. Hier sind zusätzlich Transitionen für Ausnahmefälle vorgesehen, wenn z.B. alle Bieter die Nachricht `NoBid` senden, oder ein Agent nicht mehr reagiert.

```
protocol ContractNet

  % channel for messages between manager and bidder

  channel
     Manager(Announce Grant Reject)
     Bidder(Bid NoBid Report)
  end


  % role manager; one instance is allowed within one
  % protocol execution

  role(1) Manager(agentList BidderList,
                  any Task, integer TimeoutValue)


    % channel between the protocol and the applicationProcedures
    % of the agent that has instantiated this role
    %
    % names and parameters of messages to applicationProc correspond
    % with names and parameters of the procedures
    %
    % messages from applicationProc are named as the procedures with
    % the word 'Result' in front

    channel
      protocol(ChooseBid(any Bid1, any Bid2)
               Inform(string Message)
               GetResult)

      applicationProc(ResultChooseBid)
    end



    %my states
    states
      WaitingForAnswer
      WaitingForReport
    end


    declare

      %my variables
      message
        BestBid % the currently best bid
       Result
      end

      agent
        Contractor
      end

    end


    timeout
      Timeout1 TimeoutValue startOnEnter in WaitingForAnswer end
      Timeout2 TimeoutValue startOnEnter in WaitingForReport end
    end

    initialize to WaitingForAnswer
    provided (length(BidderList) >= 1)
    begin

      send Announce(Task) to Bidder.BidderList % send the message to all the
                                               % agents in the list

    end


    initialize to done
    provided (length(BidderList) < 1)

    % to less agents to announce to; there must be 2 or more
    begin
```

```
  applicationProc(Inform("bidderList must have at least 2 elements"))
end


trans


  % transition 1

  from WaitingForAnswer to same
  when Bidder.BidderList % one of the agents in the list
  sends Bid
  provided (BestBid == nil)

  % this the first bid
  begin
     remove(sender BidderList)
    BestBid = currentMessage
  end


  % transition 2

  from WaitingForAnswer WaitingForReport to same
  when Bidder.BidderList sends Bid
  provided ((BestBid \= nil)  && (length(BidderList) > 1))

  %there was a bid before
  begin
    remove(sender BidderList)
    insert(CompareBids)    % insert macro
  end


  % transition 3

  from WaitingForAnswer to same
  when Bidder.BidderList sends NoBid
  provided (length(BidderList) > 1)  % not the last reply

  begin
    remove(sender BidderList)
  end


  % transition 4

  from WaitingForAnswer to WaitingForReport
  when Bidder.BidderList sends Bid
  provided (length(BidderList) == 1)

  % last reply
  begin
    remove(sender BidderList)
    insert(CompareBids)
    Contractor = getSender(BestBid)
    send Grant() to Bidder.Contractor
  end


  % transition 5

  from WaitingForAnswer to WaitingForReport
  when Bidder.BidderList sends NoBid
  provided ((length(BidderList) == 1) &&
            (BestBid \= nil))

  % last reply
  begin
    remove(sender BidderList)
   Contractor = getSender(BestBid)
    send Grant() to Bidder.Contractor
  end


  % transition 6

  from WaitingForAnswer to WaitingForReport
```

```
  when system sends Timeout1
  provided (BestBid \= nil)    % there is a bid
  begin
    applicationProc(Inform("timeout in WaitingForAnswer, taking currently best bid"))
    Contractor = getSender(BestBid)
    send Grant() to Bidder.Contractor
  end


  % transition 7

  from WaitingForAnswer to done
  when system sends Timeout1
  provided (BestBid == nil)    % there is no bid
  begin
    applicationProc(Inform("timeout in WaitingForAnswer and no bid received"))
  end


  % transition 8

  from WaitingForReport to done
  when Bidder.Contractor sends Report
  begin
    send GetResult(currentMessage.content) to applicationProc
  end


  % transition 9

  from WaitingForReport to done
  when system sends Timeout2
  begin
    applicationProc(Inform("timeout in WaitingForReport and no report received"))
  end


  % transition 10

  from WaitingForReport to same
  when Bidder.BidderList sends Bid
  begin
    send Exception("bid arrived to late") to Bidder.sender
  end


  % transition 11

  from WaitingForAnswer to done
  when Bidder.BidderList sends NoBid
  provided ((length(BidderList) == 1) &&
            (BestBid == nil))

  % last reply but currently no bid
  begin
    applicationProc(Inform("no bid received"))
  end


end




macro CompareBids
begin

  Result = applicationProc(ChooseBid(BestBid.content currentMessage.content))

  if (Result.content == BestBid.content)  % the new bid is worse
  then
    send Reject() to Bidder.getSender(currentMessage)

  elseif (Result.content == currentMessage.content) % the new bid is better
  then
    send Reject() to Bidder.getSender(BestBid)
    BestBid = currentMessage
```

```
    else    % error
      applicationProc(Inform("wrong result of method chooseBid"))
      exit
    end
  end


end



% role bidder; multiple instances are allowed within
% one protocol execution

role(*) Bidder


  %my methods

  channel
    protocol(CheckTask
             DoTheJob
             Inform(string Message))

    applicationProc(ResultCheckTask ResultDoTheJob)
  end



  %my states
  states
    Start CheckingTask WaitingForAnswer
    DoingTheJob
  end



  declare

    %my variables
    any MyTask end
    agent MyManager end

  end


  timeout
    Timeout1 12000 startOnEnter in WaitingForAnswer end
  end


  initialize to Start
  begin
  end

  trans

    % transition 1

    from Start to CheckingTask
    when Manager.anyAgent sends Announce
    begin
      MyManager = sender
      MyTask = currentMessage.content    % remember task
      send CheckTask(MyTask) to applicationProc
    end


    % transition 2

    from CheckingTask to WaitingForAnswer
    when applicationProc sends ResultCheckTask
    provided (currentMessage.content \= nil)

    % content is not nil; send bid
    begin
      send Bid(currentMessage.content) to Manager.MyManager
    end
```

```
      % transition 3

      from CheckingTask to done
      when applicationProc sends ResultCheckTask
      provided (currentMessage.content == nil)

      % content is nil; send no bid
      begin
        send NoBid() to Manager.MyManager
      end


      % transition 4

      from WaitingForAnswer to done
      when Manager.MyManager sends Reject
      begin
        applicationProc(Inform("received reject"))
      end


      % transition 5

      from WaitingForAnswer to DoingTheJob
      when Manager.MyManager sends Grant
      begin
        send DoTheJob(MyTask) to applicationProc
      end


      % transition 6

      from WaitingForAnswer to done
      when system sends Timeout1
      begin
        applicationProc(Inform("timeout and no reaction from manager"))
      end


      % transition 7

      from DoingTheJob to done
      when applicationProc sends ResultDoTheJob
      begin
        send Report(currentMessage.content) to Manager.MyManager
      end
    end

  end

end
```

6.2 Das Extended-Contract-Net-Protokoll

Im Extended-Contract-Net-Protokoll kann der Manager die Aufgabenstellung nachträglich über die Nachricht `refine` verändern. Außerdem kann ein Bieter einen Vorschlag für ein `refine` abgeben. In dieser Implementierung des Protokolls gibt dazu ein Bieter mit seinem Angebot (Nachricht `Bid`) an, ob er eine Refine-Nachricht senden will. Dadurch weiß der Manager, daß diese Nachricht noch aussteht. Die Transitionen sind zum Teil identisch mit denen des Contract-Net-Beispiels. Der Manager des Extended-Contract-Net-Protokolls (Abb. 5) sammelt jedoch zusätzlich im Zustand `WaitingForAnswer` `Refine`-Nachrichten ein, die von den Bietern eintreffen. Nach der letzten eingetroffenen Nachricht (Nachricht `Bid`, `NoBid` oder `Refine`), überprüft er außerdem mit Hilfe der Anwendungs-Prozedur `ConsiderRefine`, ob er ein `Refine` an alle senden soll. Lautet das Ergebnis dieser Prozedur (in der Nachricht `ResultConsiderRefine`), daß kein Refine gesendet werden soll, sendet der Manager die Nachricht `Grant` an den besten Bieter und `NoRefine` an alle anderen. Dadurch wissen diese, daß das Protokoll damit für sie beendet ist. Soll jedoch ein `Refine` vom Manager gesendet werden, so wird zuerst dem Bieter mit dem bisher besten Angebot eine Ablehnung (Nachricht `Reject`) geschickt und dann eine `Refine`-Nachricht an alle Bieter gesendet. Der Ablauf beginnt damit von vorne.

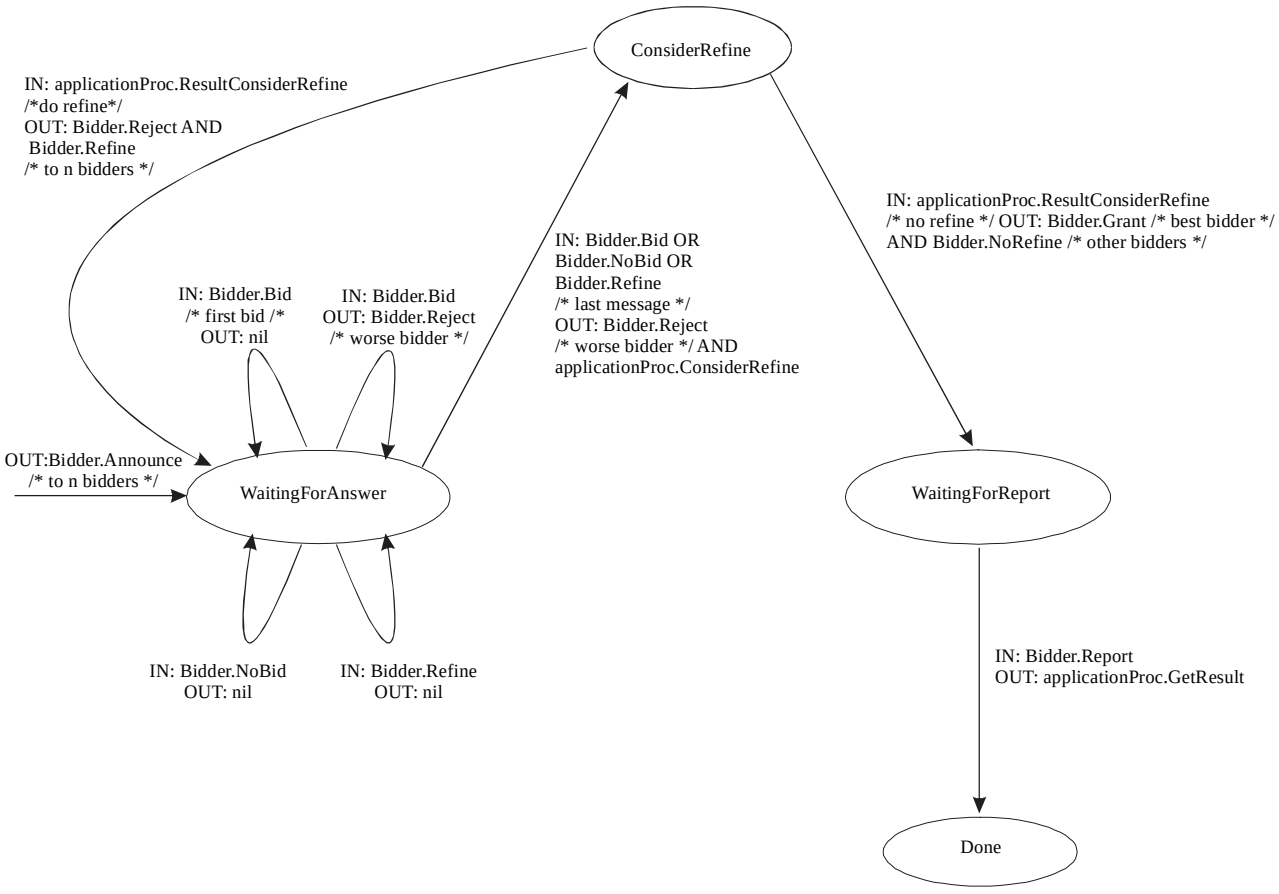

Abb.5: Der Manager im Extended-Contract-Net-Protokoll

Auch beim Bieter des Extended-Contract-Net-Protokolls (Abb. 6) sind Zustände und Transitionen hinzugekommen. Da sich der Ablauf Ausschreibung, Angebot, Ablehnung beim Extended-Contract-Net-Protokoll mehrmals wiederholen kann, wird, nachdem die Nachricht `NoBid` an den Manager gesendet, bzw. eine Ablehnung (Nachricht `Reject`) vom Manager empfangen wurde, nicht in den Endzustand `done`, sondern in den neuen Zustand `WaitIfRefine` übergegangen. Dort wird gewartet, ob der Manager eine Refine-Nachricht sendet. Ist dies der Fall, wird wie bei der ersten Ausschreibung, die Nachricht überprüft und in Abhängigkeit des Ergebnisses `Bid` oder `NoBid` zurückgeschickt. Sendet der Manager NoRefine wird schließlich in den Endzustand `done` übergegangen.

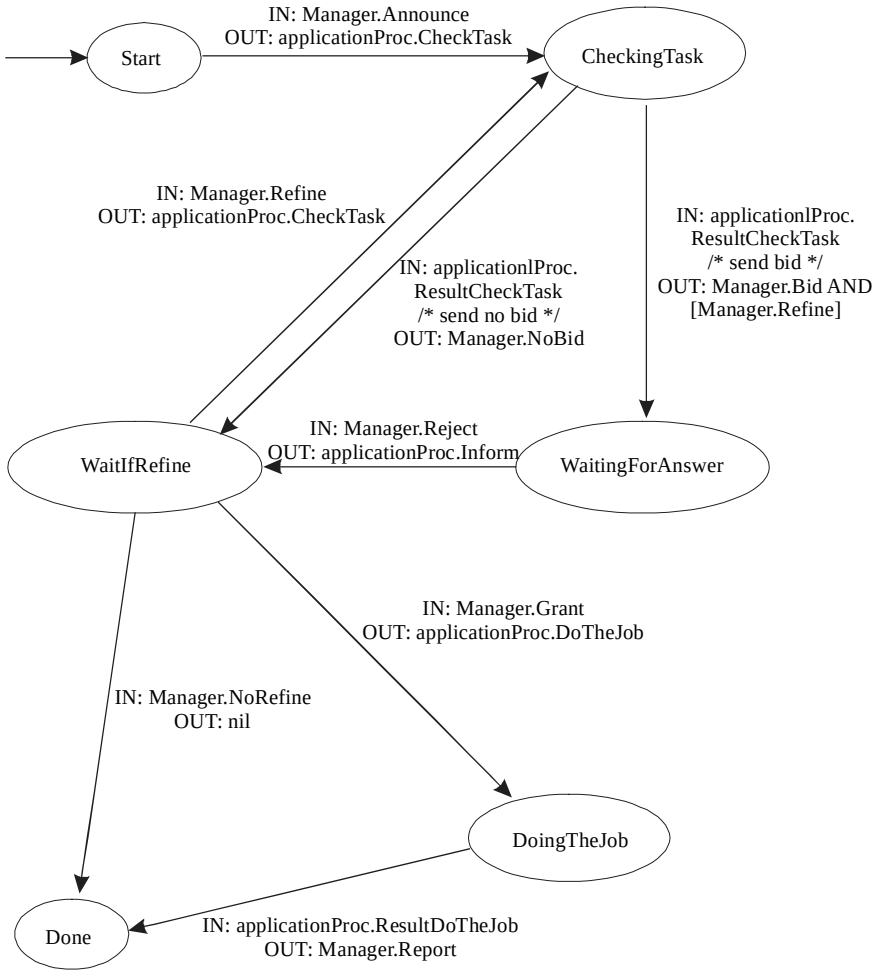

Abb.6: Der Bieter im Extended-Contract-Net-Protokoll

```
protocol ExtendedContractNet

 channel
     Manager(Announce Grant Reject Refine NoRefine)
     Bidder( Bid(any TheBid, boolean WantToRefine)
             NoBid Refine Report)
 end


 role(1) Manager(agentList BidderList, any Task, integer TimeoutValue)

   %my methods

   channel
     protocol(ChooseBid(any Bid1, any Bid2)
              Inform(string Message)
              GetResult
              ConsiderRefine(any BestBid, anyList RefineList))

     applicationProc(ResultChooseBid ResultConsiderRefine)
   end



   %my states
   states
     WaitingForAnswer
     ConsiderRefine
     WaitingForReport
   end


   declare

     %my variables
     message
       BestBid % the currently best bid
       Result
     end

     anyList
       RefineList
     end

     agent
       Contractor
      Bidder
      BestBidder
     end

     agentList
       FullBidderList
       WantToRefineList
     end

   end


   timeout
     Timeout1 TimeoutValue startOnEnter in WaitingForAnswer end
     Timeout2 TimeoutValue startOnEnter in WaitingForReport end
   end


   initialize to WaitingForAnswer
   provided (length(BidderList) >= 1)
   begin
       FullBidderList=BidderList %remember all bidders
       send Announce(Task) to Bidder.BidderList  % send the message to all the
                                                 % agents in the list
   end

   initialize to done
   provided (length(BidderList) < 1)

   % to less agents to announce to; there must be 2 or more
   begin
     applicationProc(Inform("BidderList must have more than 2 elements"))
```

```
end


trans


  % transition 1

  from WaitingForAnswer to same
  when Bidder.BidderList % one of the bidders
  sends Bid
  provided (BestBid == nil )

  %this is the first bid
  begin
    remove(sender BidderList)
    BestBid = currentMessage
    if currentMessage.WantToRefine
    then
      put(sender  WantToRefineList)
    end
  end


  % transition 2

  from WaitingForAnswer to same
  when Bidder.BidderList sends Bid % one of the bidders
  provided ((BestBid \= nil)  && (length(BidderList) > 1))

  %there was a bid before
  begin
    remove(sender BidderList)
    if currentMessage.WantToRefine
    then
      put(sender  WantToRefineList)
    end

    insert(CompareBids)
  end


  % transition 3

  from WaitingForAnswer to ConsiderRefine
  when Bidder.BidderList sends Bid
  provided ((length(BidderList) == 1) &&
            !currentMessage.WantToRefine &&
            (WantToRefineList == nil))

  %this bid is the last reply
  begin
    remove(sender BidderList)
    insert(CompareBids)
    send ConsiderRefine(BestBid.TheBid RefineList) to applicationProc
  end


  % transition 4

  from WaitingForAnswer to same
  when Bidder.BidderList sends Bid
  provided ((length(BidderList) == 1) &&
            currentMessage.WantToRefine)

  %this is the last bid, but there is at least one refine to come
  begin
    put(sender  WantToRefineList)
    remove(sender BidderList)
    insert(CompareBids)
  end



  % transition 5

  from WaitingForAnswer to same
  when Bidder.WantToRefineList sends Refine
```

```
provided ((length(WantToRefineList) > 1) ||
          (length(BidderList) >= 1))

% this is not the last reply
begin
  put(currentMessage.content RefineList)
  remove(sender WantToRefineList)
end


% transition 6

from WaitingForAnswer to ConsiderRefine
when Bidder.WantToRefineList sends Refine
provided ((BidderList == nil) && (length(WantToRefineList) == 1))

% this refine is the last reply
begin
  put(currentMessage.content RefineList)
  remove(sender WantToRefineList)
  send ConsiderRefine(BestBid.TheBid RefineList) to applicationProc
end


% transition 7

from WaitingForAnswer to same
when Bidder.BidderList sends NoBid
provided ((length(BidderList) > 1) ||
          (length(WantToRefineList) > 1))


% this is not the last reply
begin
  remove(sender BidderList)
end



% transition 8

from WaitingForAnswer to ConsiderRefine
when Bidder.BidderList sends NoBid
provided ((length(BidderList) == 1) &&
          (WantToRefineList == nil))

% this is the last reply
begin
  remove(sender BidderList)
 send ConsiderRefine(BestBid.TheBid RefineList) to applicationProc
end



% transition 9

from WaitingForAnswer to ConsiderRefine
when system sends Timeout1
provided (BestBid \= nil)    % there is a Bid
begin
  send ConsiderRefine(BestBid.TheBid RefineList) to applicationProc
end


% transition 10

from WaitingForAnswer to done
when system sends Timeout1
provided (BestBid == nil)    % there is no Bid
begin
  applicationProc(Inform("timeout and no bid received"))
end


% transition 11

from ConsiderRefine to WaitingForReport
when applicationProc sends ResultConsiderRefine
```

```
  provided (currentMessage.content == nil) % no refine needed
  begin
    Contractor = getSender(BestBid)
    send Grant() to Bidder.Contractor
   insert(FillBidderList)
    remove(Contractor BidderList)
    send NoRefine() to Bidder.BidderList
  end


  % transition 12

  from ConsiderRefine to WaitingForAnswer
  when applicationProc sends ResultConsiderRefine
  provided (currentMessage.content \= nil) % refine needed
  begin
   BestBidder = getSender(BestBid)
    send Reject() to Bidder.BestBidder
    BestBid = nil
    % fill BidderList again and send refine with new task
    insert(FillBidderList)
    send Refine(currentMessage.content) to Bidder.BidderList
  end


  % transition 13

  from WaitingForReport to done
  when Bidder.Contractor sends Report
  begin
    send GetResult(currentMessage.content) to applicationProc
  end



  % transition 14

  from WaitingForReport to done
  when system sends Timeout2
  begin
    applicationProc(Inform("timeout and no report received"))
  end


  % transition 15

  from ConsiderRefine WaitingForReport to same
  when Bidder.BidderList sends Bid
  begin
    send Exception("message arrived to late") to Bidder.sender
  end


  % transition 16

  from ConsiderRefine WaitingForReport to same
  when Bidder.BidderList sends Refine
  begin
    send Exception("message arrived to late") to Bidder.sender
  end


end


macro CompareBids
begin

  Result = applicationProc(ChooseBid(BestBid.TheBid currentMessage.TheBid))

  if (Result.content == BestBid.TheBid)  % the new bid is worse
  then
    send Reject() to Bidder.sender

  elseif (Result.content == currentMessage.TheBid) % the new bid is better
  then
    send Reject() to Bidder.getSender(BestBid)
    BestBid = currentMessage
```

```
    else    % error
      applicationProc(Inform("wrong result of method chooseBid"))
      exit
    end

  end



  % fill BidderList with active bidders

  macro FillBidderList

  begin

    % remove remaining (not reacting) bidders from
    % FullBidderList

    while (BidderList \= nil)
      Bidder = get(BidderList)
      remove(Bidder BidderList)
    end


    % fill BidderList again
    BidderList = FullBidderList

  end




end



role(*) Bidder



  %my methods

  channel
    protocol(CheckTask
             DoTheJob
             Inform(string Message))

    applicationProc(ResultCheckTask(any BidContent, any RefineContent)
            ResultDoTheJob)

  end


  %my states
  states
    Start CheckingTask WaitingForAnswer
    DoingTheJob WaitIfRefine
  end



  declare

    %my variables
    any MyTask end
    agent MyManager end
    boolean IWantToRefine end
  end


  timeout
    Timeout1 12000
      startOnTransition   % reset time on every transition
    in WaitingForAnswer WaitIfRefine
    end
```

```
end


initialize to Start
begin
end

trans

  % transition 1

  from Start to CheckingTask
  when Manager.anyAgent sends Announce
  begin
    MyManager = sender
    MyTask = currentMessage.content
    send CheckTask(MyTask) to applicationProc
  end


  % transition 2

  from CheckingTask to WaitingForAnswer
  when applicationProc sends ResultCheckTask
  provided (currentMessage.BidContent \= nil)

  % content is not nil; send bid
  begin
    IWantToRefine = (currentMessage.RefineContent \= nil)
    send Bid(currentMessage.BidContent IWantToRefine) to Manager.MyManager
    if IWantToRefine
    then
      send Refine(currentMessage.RefineContent) to Manager.MyManager
    end
  end

  % transition 3

  from CheckingTask to WaitIfRefine
  when applicationProc sends ResultCheckTask
  provided (currentMessage.BidContent == nil)

  % content is nil; send no bid
  begin
    send NoBid() to Manager.MyManager
  end


  % transition 4

  from WaitingForAnswer to WaitIfRefine
  when Manager.MyManager sends Reject
  begin
    applicationProc(Inform("received reject"))
  end


  % transition 5

  from WaitingForAnswer to DoingTheJob
  when Manager.MyManager sends Grant
  begin
    send DoTheJob(MyTask) to applicationProc
  end


  % transition 6

  from WaitIfRefine to done
  when Manager.MyManager sends NoRefine
  begin
  end


  % transition 7

  from WaitIfRefine to CheckingTask
  when Manager.MyManager sends Refine
```

```
      begin
        MyTask = currentMessage.content
        send CheckTask(currentMessage.content) to applicationProc
      end


      % transition 8

      from WaitingForAnswer WaitIfRefine to done
      when system sends Timeout1
      begin
        applicationProc(Inform("timeout and no reaction from manager"))
      end


      % transition 9

      from WaitingForAnswer WaitIfRefine to done
      when Manager.MyManager sends Exception
      provided (currentMessage.Text == "message arrived to late")
      begin
        applicationProc(Inform("message arrived to late at manager"))
      end


      % transition 10

      from DoingTheJob to done
      when applicationProc sends ResultDoTheJob
      begin
        send Report(currentMessage.content) to Manager.MyManager
      end



    end

  end

end
```

7 Das Protokollausführungssystem

Das Protokollausführungssystem (Protocol Execution System, PES) verarbeitet eine Protokollspezifikation und steuert die Ausführung des Protokolls. Dazu wird zunächst ein gegebenes Protokoll in eine Hochsprache übersetzt. Als Zielsprache wird dabei zunächst Oz verwendet. Die Syntax der Protokollsprache ist jedoch von der Zielsprache unabhängig. Aufbauend auf ein protokollunabhängiges Grundsystem, in dem z.B. Nachrichten versendet und empfangen werden, werden von einem Protokoll-Compiler (Präcompiler im Bezug auf die spätere Oz-Compilierung) Klassen erzeugt, in denen die protokollabhängigen Abläufe in Oz implementiert sind.

7.1 Grundsystem

In Abb. 7 ist die Klassenstruktur des Grundsystems dargestellt. Darauf aufbauend werden weitere protokollspezifische Klassen erzeugt (siehe Kapitel 7.2.1). Die Struktur orientiert sich an der Agentenarchitektur InteRRaP (siehe Kapitel 2.2). Dadurch soll eine leichte Integration in dieses Zielsystem ermöglicht werden. Die Klassen des Grundsystems liegen im Verzeichnis `PES`. Durch Einbinden der Datei `PES.oz`, z.B. durch die Oz-Anweisung `\insert './PES/PES.oz'` aus dem übergeordneten Verzeichnis, steht das System einer Agentenarchitektur zur Verfügung.

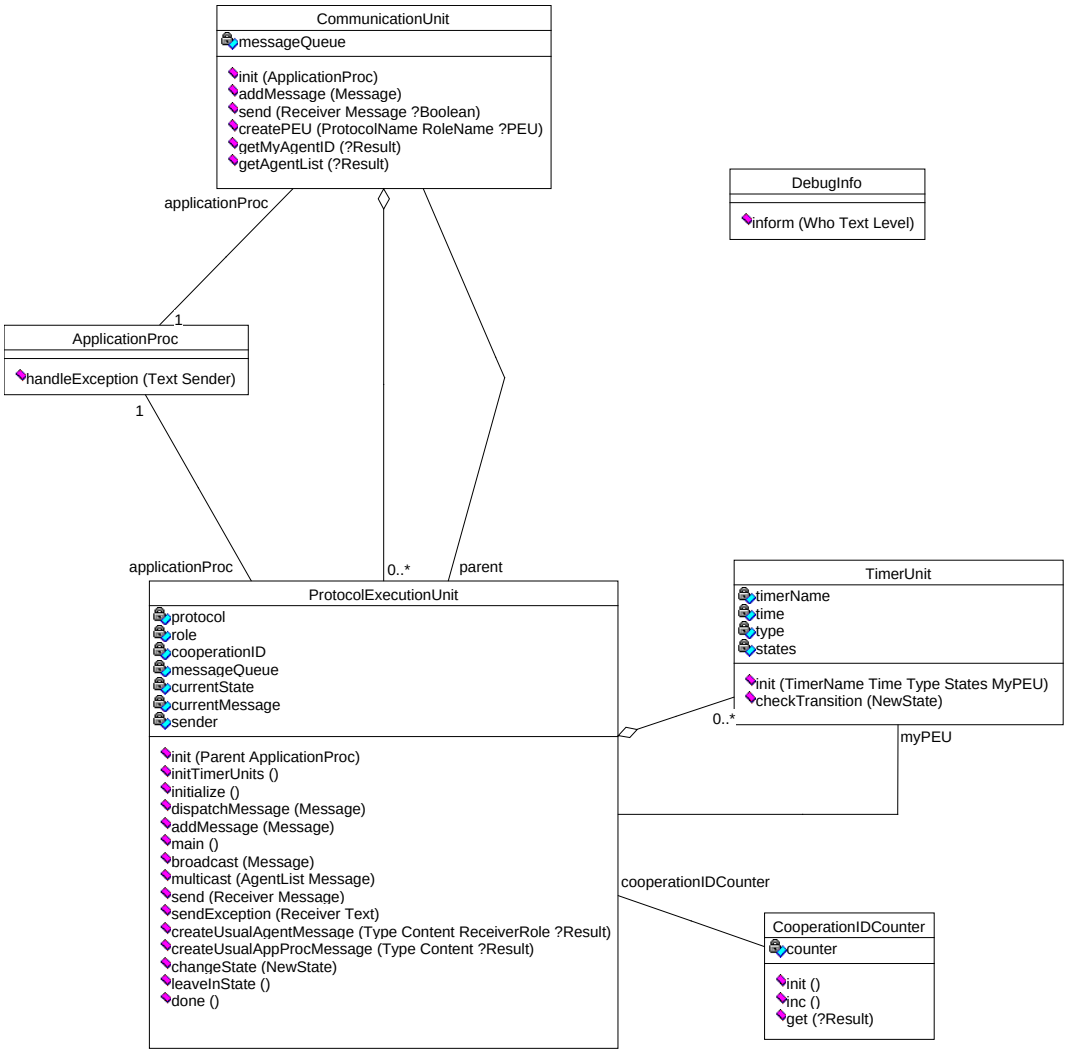

Abb. 7: Allgemeines Klassenmodell

Die Klasse `ProtocolExecutionUnit` (PEU) stellt die Einheit dar, die für einen Agenten ein laufendes Protokoll bearbeitet (Protokoll-Ausführungs-Einheit). Dies können mehrere Protokolle parallel sein. So ist es möglich, daß ein Agent zur selben Zeit in einem laufenden Contract-Net-Protokoll als Manager auf Angebote wartet und in einem anderen als Bieter gerade ein Angebot abgegeben hat. Auch können sich mehrere verschiedene Protokolle zur gleichen Zeit im Ablauf befinden. Ein Agent, der eine neue Protokollausführung starten will, instantiiert eine Unterklasse von `ProtocolExecutionUnit` und ruft darin die Prozedur `main` auf. In Kapitel 8 wird eine Beispielanwendung vorgestellt.

Jede Protokoll-Ausführungs-Einheit hat eine Unterklasse von `ApplicationProc`, in der der Anwender, die im Protokoll definierten Prozeduren implementiert. Zusätzlich implementiert er die dortige Methode `handleException`, die bei einer Ausnahmesituation im Protokollablauf aufgerufen wird.

Die Klasse `CommunicationUnit` (CU) stellt die Kommunikationsschnittstelle des Agenten dar (Kommunikationseinheit). Von ihr werden die Nachrichten der ggf. parallel ablaufenden Protokolle empfangen und gesendet. Dies ist anwendungsabhängig und kann z.B. direkt über Prozeduraufrufe geschehen, wenn die Agenten als Threads innerhalb eines Prozesses laufen, oder über TCP/IP, wenn die Agenten auf verschiedene Rechner verteilt sind. Diese anwendungsabhängigen Abläufe implementiert der Anwender in einer eigenen Unterklasse von `CommunicationUnit`. Er überschreibt dazu u.a. die Prozedur `send`, mit der eine Nachricht an einen bestimmten Agenten verschickt wird. Um Fehler (Exceptions) auf der Kommunikationsebene zu behandeln, hat auch jede Instanz der Klasse `CommunicationUnit` eine Referenz auf ein Objekt einer `ApplicationProc`-Klasse. Auf die Prozeduren der Kommunikationseinheit greift eine Protokoll-Ausführungs-Einheit über ihre Referenz `parent` zu. Trifft eine Nachricht für eine Protokollinstanz ein, so wird sie der entsprechenden `ProtocolExecutionUnit` über die Prozedur `addMessage` übergeben. Die Ausführung eines Protokolls wird durch Aufruf der `main`-Methode gestartet.

Über die Klasse `TimerUnit` wird der Timeout-Mechanismus realisiert. Die Klasse `CooperationIDCounter` generiert eindeutige Kooperations-Nummern (CooperationIDs) zur Unterscheidung der verschiedenen Protokollabläufe. Über die Kooperations-Nummer ordnet die Kommunikations-Einheit dann eingehende Nachrichten den entsprechenden Protokoll-Ausführungs-Einheiten zu.

Mit Hilfe der Klasse `DebugInfo` können über die Angabe eines Debug-Levels mehr oder weniger detaillierte Informationen über den Protokollablauf ausgeben werden. Wird hier das Debug-Level 0 eingestellt, so erscheinen sehr detaillierte Informationen über die Abläufe im PES auf der Standardausgabe. Beim Debug-Level 1 werden nur noch wichtige Informationen, wie das Empfangen oder Versenden von Nachrichten ausgegeben.

Abb. 8 zeigt das Modell der Nachrichtenklassen. Die Klasse `Message` ist die Oberklasse für alle anderen Nachrichtenklassen. Sie enthält die Attribute für den Nachrichtentyp und den Absender der Nachricht (`type` und `sender`). Von ihr abgeleitet sind die Klassen für Nachrichten von den Anwendungs-Prozeduren (`AppProcMessage`), Nachrichten vom Protokoll-Ausführungs-System (`SystemMessage`), z.B. für Timeout-Nachrichten, und Nachrichten die zwischen den verschiedenen Agenten verschickt werden

(`AgentMessage`). Diese Klassen enthalten noch kein Inhaltsfeld (Content), da sich diese Felder für verschiedene Nachrichten unterscheiden können. Wenn z.B. ein Anwender in einer Rollen-Kanal-Definition eine Nachricht mit eigenen Inhaltsfeldern definiert hat, so wird vom PES eine Unterklasse von `AgentMessage` generiert, die die gewünschten Inhaltsfelder enthält. Für gewöhnliche Nachrichten (Inhaltsfeld `content`), wird zusätzlich die Klasse `UsualMessageContent` abgeleitet. Exceptions haben immer das Inhaltsfeld `Text` vom Typ `String`. Dieses Attribut ist in der Klasse `ExceptionContent` enthalten, von der die Klassen `SystemException`, für Exceptions vom lokalen System (z.B. `"can't send message"`) und `AgentException` für Exceptions von anderen Agenten (z.B. `"no suitable transition found"`) erben. Die Nachrichten-Klassen haben einen Konstruktor (Methode `init`), in dem Werte für die Attribute übergeben werden, ggf. `set`-Methoden zum setzen weiterer Werte und `get`-Methoden zum Auslesen der Werte einer Nachricht.

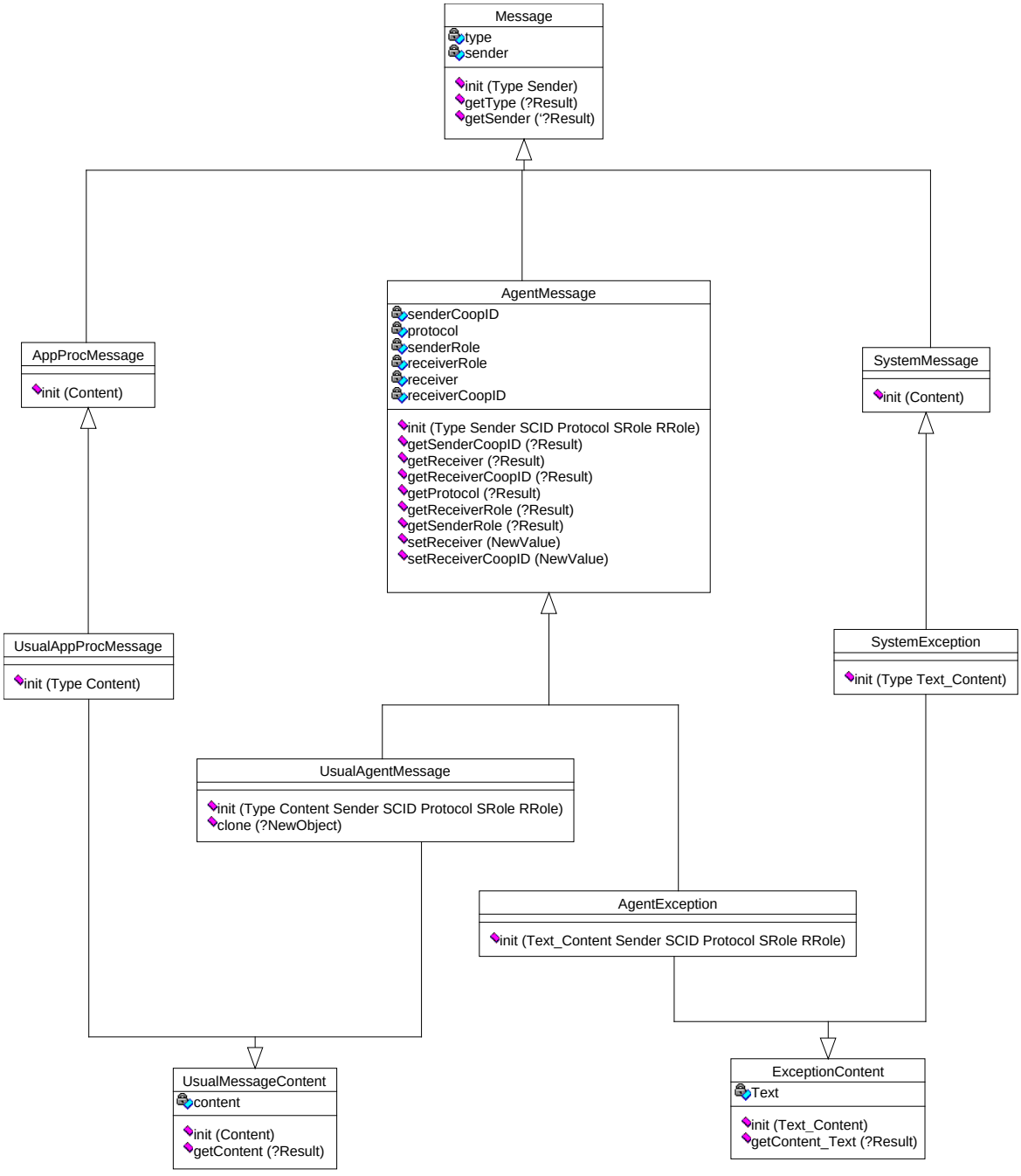

Abb. 8: Das Modell der Nachrichtenklasse

In der Klasse `AgentMessage` sind Attribute für die Absender- und Empfänger-Kooperations-Nummer (`CooperationID`) enthalten, die zur Unterscheidung der verschiedenen ablaufenden Protokolle (Kooperationen) dienen. Außerdem werden hier der Empfänger, der Name des Protokolls und die Bezeichnungen der Absender- und Empfänger-Rolle angegeben. Die Angabe von `receiverRole` ist notwendig, damit z.B. der Empfänger der Nachricht `Announce` beim Contract-Net-Protokoll weiß, daß er eine Protokollinstanz des Contract-Net-Protokolls in der Rolle des Bieters (Bidder) und nicht z.B. des Managers erzeugen muß.

Genaue Klassenbeschreibung

CommunicationUnit
(z.T. von InteRRaP übernommen)

Die Kommunikations-Einheit des Agenten. Hier wird die Kommunikation abgewickelt. Der Anwender leitet diese Klasse ab und überschreibt die anwendungsabhängigen Methoden, z.B. zum Senden.

Attribute und Features

`MessageQueue`	Schlange der noch nicht abgearbeiteten Nachrichten

Methoden

`init(ApplicationProc)`	Konstruktor mit Referenz auf die jeweilige ApplicationProc-Klasse
`AddMessage(Message)`	Die Nachricht wir dem Agenten übergeben.

Methoden, die überschrieben werden

`send (Receiver Message ?Boolean)`	Die Nachricht wird dem Agenten `Receiver` geschickt.
`CreatePEU(ProtocolName RoleName ?PEU)`	Eine neue Protokollausführungseinheit (PEU) für das angegebene Protokoll und die angegebene Protokoll-Rolle wird erzeugt. Das ist z.B. notwendig, wenn ein Agent die erste Nachricht eines neuen Protokolls erhält.
`GetMyAgentID(?Result)`	Die eigene Agentenreferenz wird zurückgegeben.
`GetAgentList(?Result)`	gibt die Liste aller ansprechbaren Agenten zurück

ProtocolExcecutionUnit
(z.T. von InteRRaP übernommen)

Für jede Protokollrolle wird eine Unterklasse von `ProtocolExecutionUnit` erzeugt, in der z.B. die Transitionen implementiert sind. Will ein Agent an einem Protokoll teilnehmen, so erzeugt er eine Instanz einer solchen Klasse, die dann für die Ausführung des Protokolls für diesen Agenten sorgt.

Attribute und Features

`Protocol`	das ausgeführte Protokoll
`Role`	die zugewiesene Rolle im Protokoll

`CooperationID`	Identifikationsnummer zur Unterscheidung der verschiedenen PEU's
`MessageQueue`	Schlange der Nachrichten für diese PEU
`CurrentState`	der aktuelle Zustand in der Protokollausführung
`CurrentMessage`	die zuletzt empfangene Nachricht
`Sender`	Referenz auf den Absender der zuletzt empfangenen Nachricht

Methoden

`init(Parent ApplicationProc)`	Konstruktor mit Übergabe der Referenzen auf die zugehörige `CommunicationUnit` und die `ApplicationProc`-Klasse
`AddMessage(Message)`	Die Nachricht `Message` wird der Protokoll-Ausführungs-Einheit übergeben.
`main()`	Die Protokollausführung wird gestartet.
`Broadcast(Message)`	Die Nachricht `Message` wird an alle ansprechbaren Agenten verschickt. Dies wird über die Methoden `getAgentList` und `send` der Klasse CommunicationUnit realisiert.
`Multicast(AgentList Message)`	Die Nachricht `Message` wird an die Agenten in `AgentList` verschickt.
`send(Receiver Message)`	Die Nachricht `Message` wird an den Agenten `Receiver` verschickt. Die geschieht über Aufruf der Methode `send` der Klasse `CommunicationUnit`.
`SendException(Receiver Text)`	Eine Exception-Nachricht wird an den Agenten `Receiver` geschickt.
`CreateUsualAgentMessage(Type Content ReceiverRole ?Result)`	Eine `UsualAgentMessage`-Klasse wird erzeugt und zurückgegeben.
`CreateUsualAppProcMessage (Type Content ?Result)`	Eine `UsualAppProcMessage`-Klasse wird erzeugt und zurückgegeben.
`ChangeState(newState)`	Eine Transition zum Zustand `newState` wird ausgeführt. Die Timer werden über den Zustandswechsel informiert.
`LeaveInState()`	Es wird eine Schleife ausgeführt. Die Timer werden informiert.
`done()`	Die Protokoll-Ausführungs-Einheit beendet die Protokollausführung. Es werden keine weiteren Nachrichten verarbeitet.

Methoden, die überschrieben werden

`InitTimerUnits()`	Die Timer zur Überwachung der Timeoutfristen werden initialisiert und gestartet.
`Inititalize()`	Ausführung der initialen Transition
`DispatchMessage(Message)`	Die Nachricht wird bearbeitet.

ApplicationProc

Für jede Protokollrolle wird eine Unterklasse von ApplicationProc erzeugt, in der die leeren Anwendungs-Prozeduren enthalten sind. Der Anwender leitet diese Klasse ab, überschreibt die Anwendungs-Prozeduren und übergibt der entsprechenden Protokoll-Ausführungs-Einheit eine Instanz dieser Klasse.

Methoden, die überschrieben werden

`HandleException(Text Sender)`	Diese Methode wird aufgerufen, wenn eine Exception-Nachricht eingeht und nicht innerhalb des Protokolls abgefangen wird.

TimerUnit

Für jede deklarierte Timout-Frist hat die entsprechende PEU eine Instanz von `TimerUnit`, die diese Frist überwacht.

Attribute und Features

`TimerName`	der in der Protokoll-Definition angegebene Name der Timeout-Frist
`Time`	der Wert der Timeout-Frist
`Type`	`startOnEnter` oder `startOnTransition`
`States`	die Zustände, für die diese Timeout-Frist gilt

Methoden

`init(TimerName Time Type States MyPEU)`	Konstruktor der Klasse
`checkTransition(NewState)`	Es wird überprüft, ob der Timer aufgrund dieser Transition gestartet, gestoppt oder zurückgesetzt werden muß.

CooperationIDCounter

(von InteRRaP übernommen)

erzeugt neue, eindeutige Kooperations-Nummern

Attribute und Features

`Counter`	der aktuelle Zählerwert

Methoden

`init()`	Konstruktor
`inc()`	erhöht den Zählerwert
`get(?Result)`	gibt den aktuellen Zählerwert (Kooperations-Nummer) zurück

DebugInfo

Mit Hilfe dieser Klasse, können zu Testzwecken Informationen über den Protokollablauf ausgegeben werden. Den Meldungen sind Prioritäten zugeordnet, so daß über die Klasse `DebugInfo` der Detailierungsgrad der Ausgabe gesteuert werden kann. Anwendungen können diese Klasse mitbenutzen, um eigene Testausgaben mit den Informationen zum Protokollablauf auszugeben.

Methoden

`inform(Who Text Level)`

Der `Text` wird mit Angabe des Absenders (`Who`) ausgeben, wenn der übergebene Wert für `Level` über dem in der Methode voreingestellten Wert liegt. Da die Methode in der Regel statisch aufgerufen wird, wurde der Wert des eingestellten Levels nicht als Attribut modelliert, sondern in der Methode `inform` belassen.

7.2 Protokoll-Compiler

Der Protokoll-Compiler übersetzt eine Protokoll-Definition, die in einer Text-Datei mit der Erweiterung `.pro` abgelegt ist, in Oz-Code. Dabei werden Dateien mit Oz-Klassen-Beschreibungen generiert, die gemeinsam mit dem PES-Grundsystem eingebunden, einen Protokollablauf steuern.

Der Compiler wurde mit Hilfe des Scanner-Generators Flex und des Parser-Generators Bison erzeugt. Dies sind die GNU-Versionen der entsprechenden Generatoren Lex und Yacc. Die formale Definition der Protokollsprache, die den Generatoren als Eingabe dient, befinden sich in Anhang 3 und 4. In dieser Definition sind zu den verschiedenen Syntaktischen Regeln C-Prozeduraufrufe angegeben, mit denen die Übersetzung ausgeführt wird. Die während der Übersetzung erzeugten Code-Teile werden in sogenannten String-Listen (verkettete Listen mit Strings als Inhalt) gespeichert, weitergereicht und zusammengesetzt. Ist der vollständige Code für eine Zieldatei erzeugt, so wird die Datei mit einem entsprechenden Vemerk über die Erzeugung geschrieben und ihr Dateiname abgespeichert. Aus diesen Daten wird schließlich eine weitere Datei generiert, die den Namen des Protokolls trägt und dazu dient, die neu erzeugten Oz-Klassen einzubinden (siehe Kapitel 7.2.1).

7.2.1 Klassengenerierung

Ausgehend von der Protokollspezifikation werden vom Protokoll-Compiler protokollspezifische Klassen erzeugt. Sie werden im aktuellen Verzeichnis abgelegt Außerdem wird darin eine Datei generiert, die den Namen des Protokolls und die Erweiterung `.oz` trägt. Durch Einbinden dieser Datei, z.B. durch die Oz-Anweisung '`\insert ContractNet\ContractNet.oz`', stehen einer Agentenarchitektur alle zu diesem Protokoll neu erzeugten Klassen zu Verfügung. Die Klassen, die aus dem Beispiel aus Kapitel 6 generiert wurden, sind in Abb. 9 dargestellt. Hier wurden aus Gründen der Übersichtlichkeit keine Attribute und Methoden der Oberklassen angegeben. Für jede Protokoll-Rolle wird eine Unterklasse von `ApplicationProc` erzeugt. Sie

enthalten die leeren Methoden der Anwendungs-Prozeduren. Der Anwender leitet diese Klassen ab, überschreibt die Methoden und implementiert so das anwendungsabhängige Verhalten des entsprechenden Protokolls.

Jeder Nachrichtentyp, der in einem Kanal mit eigenen Inhaltsfeldern deklariert wurde, wird durch eine neu erzeugte Unterklasse von `AgentMessage` bzw. `AppProcMessage` dargestellt, die diese Inhaltsfelder und entsprechende Zugriffsfunktionen enthält.

Für jede Protokoll-Rolle wird eine Unterklasse von `ProtokollExcecutionUnit` erzeugt. Hier sind die rollenspezifischen Abläufe aus der Protokollspezifikation enthalten. Alle vom Anwender deklarierten Variablen werden als Attribute in die Klasse eingefügt. Die Schreibweise mit einem großen Anfangsbuchstaben wird dabei beibehalten. Die Attribute sind dadurch privat in der Oz-Klasse. Wenn, wie im Beispiel des Contract-Net-Protokolls, Initialparameter für eine Rolle angegeben wurden, wird in der generierten Klasse ein neuer Konstruktor erzeugt (Methode `init`), in dem, neben den üblichen Parametern `Parent` und `ApplicationProc`, die Werte für die Initialparameter übergeben werden. Wurden Timeout-Fristen deklariert, wird die Methode `initTimerUnits()` überschrieben. Die initiale Transition wird in der Methode `initialize()` und alle weiteren in der Methode `dispatchMessage` implementiert. Zusätzlich wird für jedes Makro eine Methode erzeugt. Auch hier wird die Schreibweise mit dem großen Anfangsbuchstaben beibehalten. Wie Transitionen und Aktionen der Protokollsprache in die Programmiersprache Oz umgesetzt werden, wird im nächsten Kapitel beschrieben.

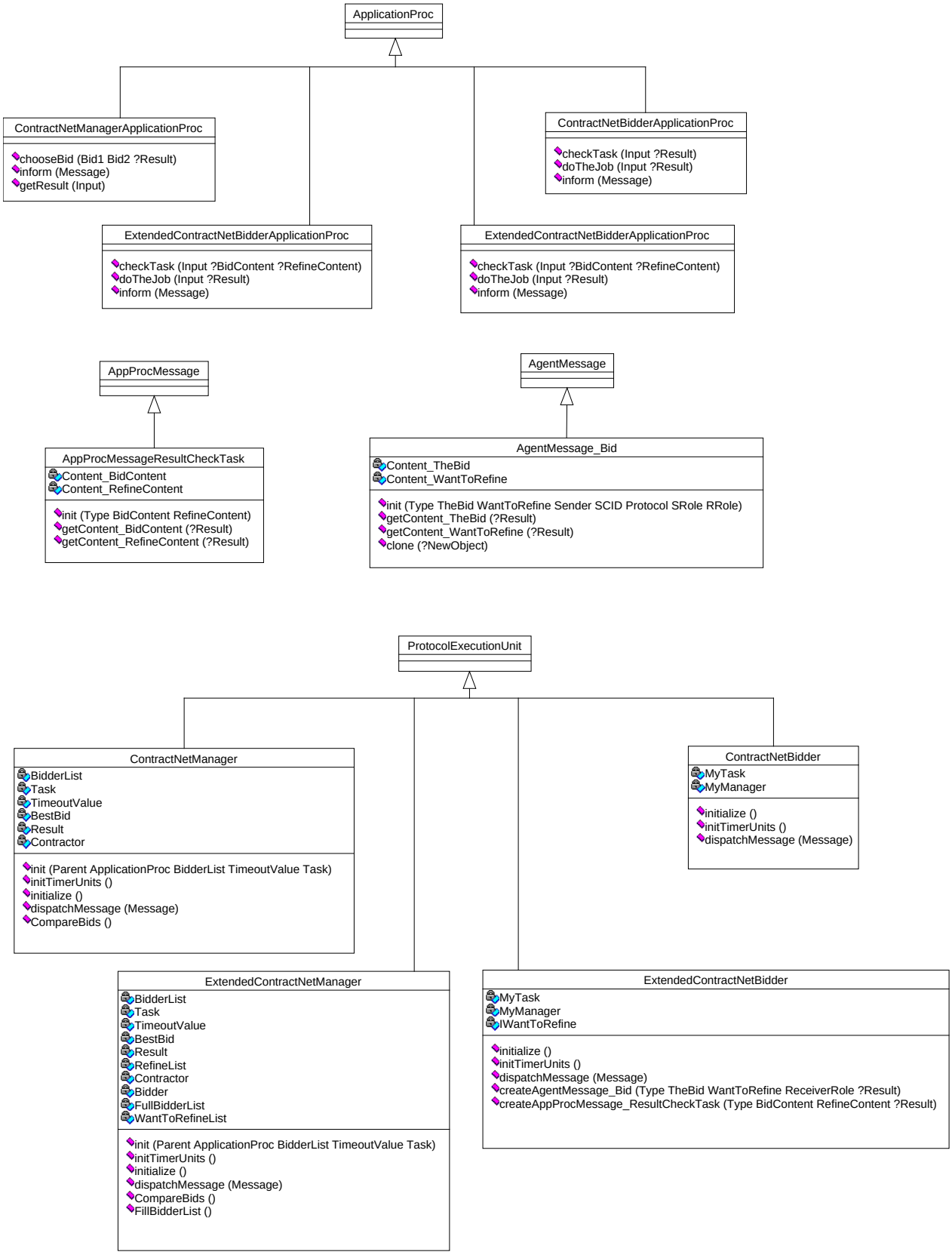

Abb. 9: Die aus dem Contract-Net-Beispiel erzeugten Klassen.

7.2.2 Protokollübersetzung

In den erzeugten Unterklassen von `ProtocolExcecutionUnit` werden für alle deklarierten Zustände Features mit dem Namen des jeweiligen Zustands generiert, die mit einem eindeutigen Integerwert initialisiert werden. Innerhalb der Methoden kann dann mit `self.<Zustandsname>` darauf zugriffen werden. Das Attribut `currentState` enthält den Wert des aktuellen Zustands, `currentMessage` die zuletzt empfangene Nachricht und `sender` den Absender dieser Nachricht. Zur Zeit werden für die Sprachelemente `QueueUnexpectedMessages` und die Angabe der maximal möglichen Rolleninstanzen (siehe Kapitel 5.2) keine funktionalen Einheiten generiert. Dies schränkt die Mächtigkeit der Protokollsprache nicht ein, da diese Funktionen auch innerhalb des Protokolls realisiert werden können. Diese Sprachelemente sollten jedoch in Zukunft unterstützt werden, da sie in manchen Fällen eine Protokoll-Repräsentation vereinfachen können.

7.2.2.1 Transitionen

Die Transitionen befinden sich in der Methode `dispatchMessage`. Lediglich die initialen Transitionen sind in der Methode `initialize()` enthalten. Der Transitionsblock mit den Transitionen der Form

```
from <AusgangsZustand> to <ZielZustand>
when <Rolle>.<Agent> sends <Nachricht>
[provided <bool'scher Wert>]
begin
  <Aktionen>
end
```

wird in eine `case`-Anweisung übersetzt. Dabei sind für jede Transition, die durch eine Nachricht von einem anderen Agenten ausgeführt werden soll, folgende Bedingungen zu prüfen:

aktueller Zustand
Absender der Nachricht
ggf. Rolle des Absenders
Nachrichtentyp
ggf. die `provided`-Bedingung

Ob der aktuelle Zustand mit einem der Ausgangszustände der Transition übereinstimmt, wird durch den Ausdruck

```
{Member @currentState [ self.<AusgangsZustand1>
                        self.<AusgangsZustand2> ... ] $}
```

überprüft. Diese Bedingung entfällt, wenn für den Ausgangszustand `anyState` angegeben wurde. Die 2. Bedingung wird in Abhängigkeit des Typs von `<Agent>` überprüft. Steht hier eine Variable vom Typ `agent`, so wird lediglich der Wert der Variable mit dem Wert des Attributs `sender` verglichen. Ist eine Variable vom Typ `agentList` angegeben, so wird mit `{Member @sender @<AgentList>}` überprüft, ob der Absender in der Liste enthalten ist. Bei `anyAgent` muß lediglich mit (`@sender \= 'applicationProc' orelse @sender \= 'system'`) überprüft werden, ob der Absender nicht eine Anwendungs-Prozedur, oder das PES-System ist. Die Rolle des Absenders und der Nachrichtentyp werden mit Hilfe der Methoden `getSenderRole` und `getType` der Nachrichtenklasse ermittelt und mit den angegebenen Werten verglichen. Die Übersetzung des bool'schen Wertes der `provided`-Bedingung wird später beschrieben. Die Bedingungen werden mit `andthen` verknüpft. Bei der initialen Transition ist nur ggf. die `provided`-Bedingung zu prüfen.
Im Beispiel des Contract-Net-Managers aus Kapitel 5 wird z.B. die erste Transitionsbedingung der Form

```
from WaitingForAnswer to same
when Bidder.BidderList % one of the agents in the list
sends Bid
provided (BestBid == nil)
```

in die `case`-Bedingung

```
elsecase
    % test current state
    {Member @currentState [ self.WaitingForAnswer ] $}
    % test message sender
    andthen {Member @sender @BidderList $}
    % test sender role (if agent message)
    andthen {@currentMessage getSenderRole($)} == 'Bidder'
    % test message type
    andthen {@currentMessage getType($)} == 'Bid'
    % provided clause (if given)
    andthen @BestBid == nil
```

übersetzt.
Soll eine Transition aufgrund der Nachricht einer Anwendungs-Prozedur oder des Systems ausgeführt werden, wird der Absender auf den Wert `applicationProc`, bzw. auf `system` überprüft. Die Überprüfung der Absender-Rolle entfällt, statt dessen steht hier `true`.
Die zweite Transition des Contract-Net-Bidders aus Kapitel 5 wird z.B. von

```
from CheckingTask to WaitingForAnswer
when applicationProc sends ResultCheckTask
provided (currentMessage.content \= nil)
```

nach

```
elsecase
    % test current state
    {Member @currentState [ self.CheckingTask ] $}
    % test message sender
    andthen @sender == applicationProc
    % test sender role (if agent message)
    andthen true
    % test message type
    andthen {@currentMessage getType($)} == 'ResultCheckTask'
    % provided clause (if given)
    andthen {@currentMessage getContent($)} \= nil
```

übersetzt.

Wird mit der Transition in einen anderen Zustand übergegangen, so wird als erste Anweisung der Methodenaufruf `{self changeState(self.<ZielZustand>}` ausgeführt. Eine Schleife wird durch den Aufruf `{self leaveInState}` realisiert. Beim Übergang in den Endzustand wird als letzte Anweisung dieser Transition `{self done}` ausgeführt. In diesen Methoden werden u.a. die Timer über die Transition informiert.

7.2.2.2 Aktionen

Aktionen der Protokollsprache, die in einem Makro zusammengefaßt sind, stehen in der erzeugten Klasse in einer Methode, die den Namen des Makros trägt. Der Makro-Aufruf wird so durch `{self <MakroName>}` ersetzt.

7.2.2.2.1 Versenden von Nachrichten

Die Anweisung

```
send <Nachricht>(<Inhalt>) to <Rolle>.<Agent>
```

wird in den Aufruf

```
{self send(@<Agent>
       {self createUsualAgentMessage(<Nachricht> <Inhalt> <Rolle> $)} )}
```

übersetzt.

Die Methode `createUsualAgentMessage` gibt dabei ein neues Nachrichtenobjekt (Unterklasse von `Message`) zurück. Hier werden außerdem die Angaben zur Rolle des Absenders und die Kooperationsnummern eingetragen. Soll eine Nachricht eines neu definierten Nachrichtentyps verschickt werden, so wird die dafür erzeugte Methode `createAgentMessage_`<NachrichtenTyp> aufgerufen. Ist `<Agent>` eine Variable vom Typ `agentList`, so wird anstatt `send` die Methode `multicast` aufgerufen. Steht `anyAgent` für `<Agent>`, so wird `broadcast` ausgeführt.

Wird eine Nachricht an eine Anwendungs-Prozedur geschickt, von der keine Ergebnis-Nachricht erwartet wird (keine zugehörige Result-Nachricht im Kanal von `applicationProc` definiert), so wird die `send`-Anweisung in einen Prozeduraufruf innerhalb eines Thread-Blocks übersetzt. So übersetzt z.B. der Präcompiler die Anweisung

```
send GetResult(currentMessage.content) to applicationProc
```

in

```
thread {@applicationProc getResult({@currentMessage getContent($)} )} end
```

Wird eine Rückgabe-Nachricht erwartet, so werden zusätzlich Variablen deklariert, die den Rückgabewert aufnehmen. Bei einer nicht benutzerdefiniert Nachrichten (keine weiteren Inhaltsfelder) ist dies lediglich die Variable PES_RC. Nach Ausführen der Prozedur wird eine Rückgabe-Nachricht erzeugt (Methode `createUsualAppProcMessage, bzw createAppProcMessage_<NachrichtenTyp>) und d`er eigenen Protokollinstanz übergeben (Methode `addMessage`).

So übersetzt der Präcompiler z.B. die Anweisung

```
send CheckTask(MyTask) to applicationProc
```

in

```
thread
    PES_RC
in
    {@applicationProc checkTask(MyTask PES_RC)}
    {self addMessage({self createUsualAppProcMessage('ResultCheckTask'
                                                      PES_RC $)} )}
end
```

Soll eine Anwendungs-Prozedur ohne Ergebnis-Nachricht direkt ausgeführt werden, so wird die Prozedur analog aufgerufen.

So wird z.B.

```
applicationProc(Inform("timeout in WaitingForReport and no report
                        received"))
```

nach

```
{@applicationProc inform("timeout in WaitingForReport and no report
                          received" )}
```

übersetzt.

Ist eine Ergebnis-Nachricht definiert, so wird diese zusätzlich der angegebenen Variablen vom Typ `message` zugewiesen.

```
Result = applicationProc.ChooseBid(BestBid.content
                                   currentMessage.content)
```

wird so nach

```
local
    PES_RC
in
    {@applicationProc chooseBid({@BestBid getContent($)}
                                {@currentMessage getContent($)}
                                PES_RC)}
Result <- {self createUsualAppProcMessage('ResultChooseBid' PES_RC $)}
end
```

übersetzt.

7.2.2.2.2 Variablenmanipulation

Die Operatoren auf den Datentypen werden folgendermaßen in Oz realisiert.

`message`:

```
agent getSender(message): {message getSender($)}
```

List

```
void put(<Element> <List>)              :   <List>   <-   {Append   <List>
<Element> $}
<Element> get(<List>)                   : {Nth <List> {Length <List> $}}
void remove(<Element> <List>)        :  <List>  <-  {List.subtract  <List>
                                        <Element> $}
boolean member(<Element> <List>)    : {Member <Element> <List> $}
integer length(<List>)              : {Length <List> $}
```

7.2.2.2.3 Schleifen und Verzweigungen

Da es in Oz kein zu `while` äquivalentes Konstrukt gibt, wird

```
while <Wahrheitswert>
  <Aktionen>
end
```

in eine rekursive Version der Form

```
local
While = proc{$}
           case  <Wahrheitswert> then
              <Aktionen>
              {While}
           else
              skip
           end
        end
in
   {While}
end
```

umgesetzt.

`if`-Verzweigungen werden in `case`-Verzweigungen übersetzt.

Protokoll:

```
if <Wahrheitswert1> then
  <Aktionen1>
elseif <Wahrheitswert2> then
  <Aktionen2>
else
  <Aktionen3>
end
```

Oz-Implementierung:

```
case <Wahrheitswert1> then
  <Aktionen1>
elsecase <Wahrheitswert2> then
  <Aktionen2>
else
  <Aktionen3>
end
```

8 Ein Anwendungsbeispiel

In Abb. 10 wird eine Anwendung gezeigt, die die aus dem Beispiel aus Kapitel 6 erzeugten Klassen verwendet. Zur Übersichtlichkeit wurden in der Darstellung Attribute und Methoden der Klassen weggelassen. Die Klasse Agent besitzt eine Kommunikationseinheit (`MyCommunicationUnit`), die von der Klasse `CommunicationUnit` abgeleitet ist. Die Klasse `ADS` (Agent Directory Service) dient zur Verwaltung der Referenzen auf die restlichen Agenten der Anwendung. Die Kommunikationseinheit erzeugt bei Bedarf Instanzen der Protokollspezifischen Klassen `ContractNetManager`, `ContractNetBidder`, `ExtendedContractNetManager` und `ExtendedContractNetBidder`, die `ProtocolExceutionUnit` ableiten. Der Anwender hat außerdem die `ApplicationProc`-Klassen der verschiedenen Protokollrollen abgeleitet. Sie erben außerdem von `MyApplicationProc`, worin die Methode `handleException` implementiert und das Verhalten der Einzelnen Agenten beim Protokollablauf, z.B. die Höhe der Angebote und Werte für Zeitverzögerungen, festgelegt sind.

In dieser einfachen Testanwendung werden 5 Agenten verwendet, die das Contract-Net, bzw. das Extended-Contract-Net-Protokoll ausführen. Das Verhalten der Agenten kann in der Klasse `MyApplicationProc`

verändert werden. Eine compilierte Stand-Alone-Anwendung, die z.B. mit `ozsac main.oz Test` erzeugt wird, kann mit dem Namen des Protokolls als Parameter gestartet werden, z.B. `Test ContractNet`. Die Informationen über den Protokollablauf erscheinen auf der Standard-Ausgabe.

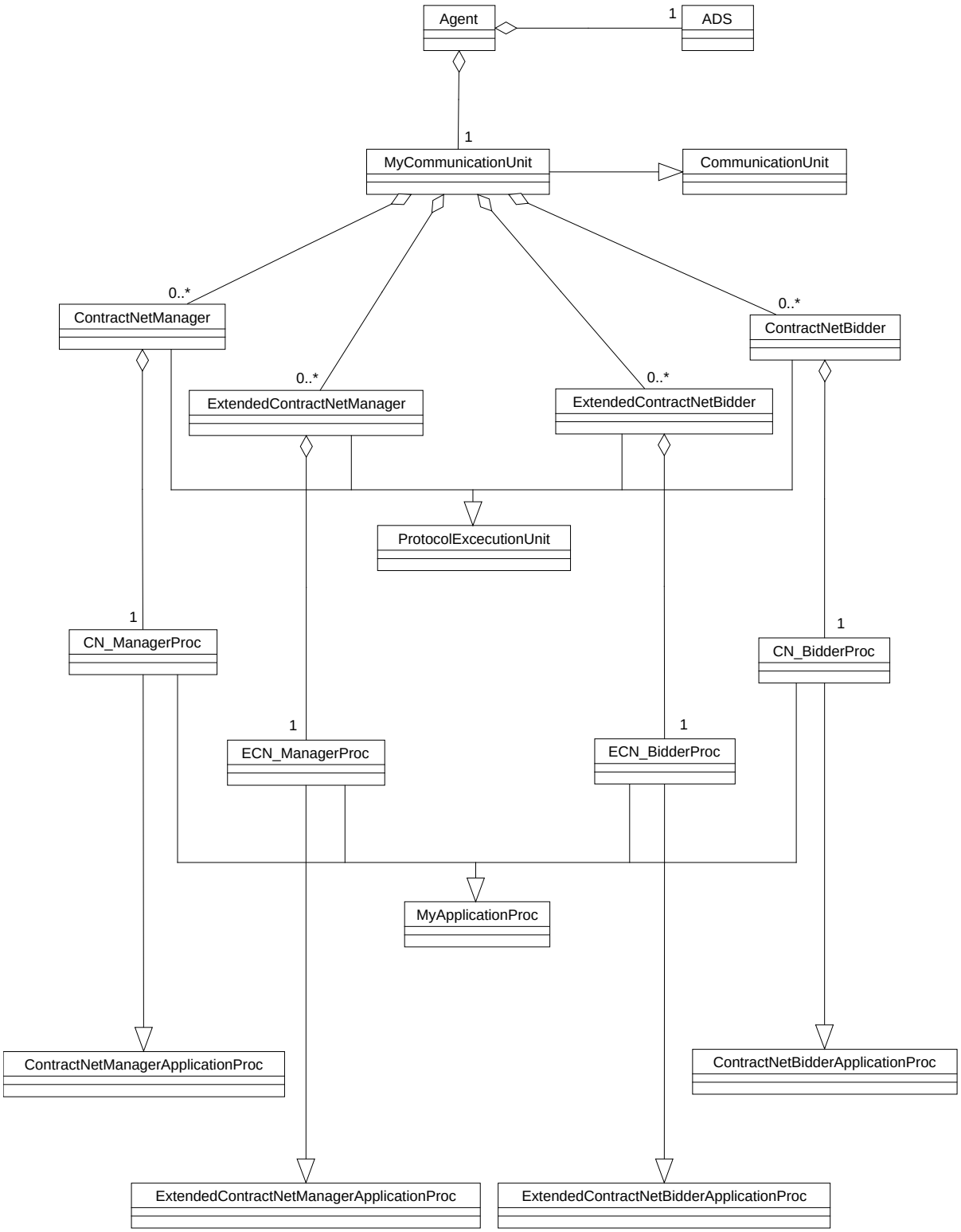

Abb. 10: Struktur einer Anwendung, die auf die aus dem (Extended-)ContractNet-Beispiel erzeugten Klassen aufbaut.

9 Zusammenfassung und Ausblick

Mit der hier entwickelten Protokollsprache können Multiagenten-Protokolle klar und eindeutig spezifiziert werden. Durch die automatische Code-Generierung aus einer Protokoll-Spezifikation entfällt der fehleranfällige Arbeitsschritt der Implementierung des festgelegten Protokoll-Ablaufs. Das Protokoll-Ausführungs-System übernimmt die Steuerung des Ablaufs, erkennt Fehler (z.B. falsche Nachrichten) und überwacht Timeout-Fristen. Ein Programmierer muß nach der Festlegung eines Protokolls lediglich den Code für die anwendungsabhängien Abläufe (Anwendungs-Prozeduren) und die Agentenumgebung (Zugriff auf andere Agenten, Zustellen der Nachrichten) zu Verfügung stellen. Durch die Trennung von Protokoll und Anwendung können so einmal entworfene Protokolle für unterschiedliche Anwendungen verwendet werden, ohne den eigentlichen Protokoll-Code zu verändern. Durch dieses System ergibt sich für das Deutschen Forschungszentrum für Künstliche Intelligenz die Möglichkeit, neue Protokolle schneller zu entwickeln und einzusetzen.

10 Anhang 6: Literaturverzeichnis

Multiagentensysteme

[M1] Jürgen Müller (Hrsg.): Verteilte Künstliche Intelligenz; BI Wissenschaftsverlag Mannheim, Leipzig, Wien, Zürich; 1993

[M2] G.M.P. O'Hare / N.R. Jennings: Foundations of Distributed Artificial Intelligence; John Wiley & Sons Inc., N.Y.; 1996

[M3] Afsaneh Haddadi: Communication and Cooperation in Agent Systems; Springer-Verlag Berlin, Heidelberg, New York; 1996

[M4] Jörg H. Siekmann, Klaus Fischer: Vorlesung Multiagentensysteme, WS 96/97;
URL: http://www.dfki.uni-sb.de/~kuf/mas.html

[M5] Christoph G. Jung, Klaus Fischer: Methodological Comparison of Agent Models; Universität des Saarlandes & DFKI GmbH, April 1998;
URL: http://www.dfki.de/~jung/publications/comparison98.ps.gz

[M6] Hans-Jürgen Bürckert, Klaus Fischer, Gero Vierke: TELETRUCK: A Holonic Fleet Management System; DFKI GmbH
URL: ftp://ftp.dfki.uni-kl.de/pub/Publications/TechnicalMemos/97/TM-97-03.ps.gz

[M7] Bachem, W. Hochstättler, M. Malich: Simulated Trading: A New Approach For Solving Vehicle Routing Problems; Mathematisches Institut der Universität zu Köln;
Technical Report No 92.125; 1992

Oz

[O1] Mehl, T. Müller, K. Popov, R. Scheidbauer, C. Schulte: DFKI Oz User's manual;
DFKI 1997; URL: http://www.ps.uni-sb.de/oz2/documentation/

[O2] Henz, M. Müller, T. Müller, C. Schulte, J. Würtz: The Oz Standard Modules;
DFKI 1997; URL: http://www.ps.uni-sb.de/oz2/documentation/

[O3] Henz, M. Müller, J. Würtz: Tips on Oz;
DFKI 1997; URL: http://www.ps.uni-sb.de/oz2/documentation/

[O4] Seif Haridi: A Tutorial of Oz 2.0

Compilerbau

[C1] Niklaus Wirth: Grundlagen und Techniken des Compilerbaus; Addison-Wesley Bonn, Paris; 1996

[C2] Reinhard Wilhelm, Dieter Maurer: Übersetzerbau: Theorie, Konstruktion, Generierung; Springer-Verlag Berlin, Heidelberg; 1992

[C3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann: Compilerbau, Teil 1; Addison-Wesley Bonn, München, u.A. ; 1988

[C4] Jim Welsh, Michael McKeag: Structured System Programming; Prentice-Hall International, Inc., London; 1980

[C5] John R. Levine, Tony Mason, Doug Brown: Lex & Yacc; O'Reilly & Associates, Inc.;1995

Sonstiges

[S1] Dieter Hogrefe: Estelle, LOTOS und SDL: Standard-Spezifikationssprachen für verteilte Systeme; Springer-Verlag Berlin, Heidelberg; 1989

Ein Systems zur Definition und Ausführung von Protokollen für Multi-Agentensysteme

Stefan Philipps und Jürgen Lind

RR-99-01

Research Report