Agentenkommunikation - Einsatz von FIPA/ACL zur Modellierung der Kommunikation zwischen Dialogmodell und Applikation

Studienarbeit am Institut für Theoretische Informatik
Prof. Dr. Alex Waibel
Fakultät für Informatik
Universität Karlsruhe (TH)

von

cand. inform.
Tobias Kluge

Betreuung:

Prof. Dr. Alex Waibel
Dipl. Inform. Hartwig Holzapfel

Tag der Anmeldung: 1. Juli 2005
Tag der Abgabe: 30. September 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1 Zielsetzung der Arbeit

Dialogsysteme und Anwendungen sind sehr eng miteinander verbunden, oft werden Anwendungen direkt an Dialogsysteme gekoppelt. Allerdings sollte ein Dialogsystem unabhängig von einer Domäne und verwendeten Applikationen sein. Das ist schwierig, wenn die Anwendung fest an das Dialogsystem gebunden ist.
In dieser Arbeit wird die lose Kopplung von Anwendungen an ein Dialogsystem untersucht. Als Grundlage werden Agentensysteme verwendet, die eine dynamische Verbindung zwischen Dialogsystem und Anwendungen ermöglichen.
Die Implementierung dieser Arbeit verwendet das Tapas-Dialogsystem; als Anwendung wird die CHIL Smartroom-Anwendung zur Steuerung von Geräten in einem multimodalen Konferenzraum genutzt.

1.2 Gliederung der Arbeit

Im Kapitel 2 werden die Grundlagen vorgestellt, die für das Verständnis dieser Arbeit nötig sind. Bereits bestehende Arbeiten auf verwandten Gebieten werden im Kapitel 3 diskutiert.
Der im Rahmen dieser Arbeit entwickelte Ansatz wird in den Kapiteln Entwurf (Kapitel 4), Implementierung (Kapitel 5) und damit erreichte Ergebnisse in Kapitel 6 vorgestellt. Eine Zusammenfassung und ein Ausblick auf mögliche Erweiterungen werden in Kapitel 7 gegeben.
Im Anhang A wird die verwendete Smartroom-Anwendung vorgestellt, die erstellten Dialogstrukturen sind im Anhang B abgedruckt.

2. Grundlagen

In diesem Kapitel werden grundlegende Forschungsthemen und Technologien diskutiert, die im Rahmen der Studienarbeit verwendet wurden.

2.1 Theoretische Grundlagen

Die theoretischen Grundlagen liegen im Bereich der Agententheorie, FIPA/ACL und Ontologien. Diese Konzepte werden nachfolgend vorgestellt.

2.1.1 Agententheorie

Eine sehr gute Einführung liefern Michael J. Wooldridge und Nicholaes R. Jennings in ihrer Arbeit „Agent Theories, Architectures, and Languages: A Survey“ aus dem Jahr 1994 (siehe [WoJe95]). Die folgende kurze Einführung basiert auf dieser Veröffentlichung, gibt aber nur einen kurzen Überblick. Für detailierte Informationen ist das Studium dieser Ausarbeitung empfohlen.

Was sind Agenten

Agenten sind unabhängige, rational handelnde Funktionseinheiten. Diese können sowohl mit Menschen als auch mit anderen Agenten interagieren. Sie werden unter anderem im Bereich der verteilten Intelligenz (Distributed Artified Intelligence, DAI) verwendet, um Probleme zu lösen.

Ein Agent besitzt Wissen über die Welt um den Agenten („belief“, „knowledge“) sowie Wissen, welches seine Handlungen beeinflusst („desire“, „intention“, „obligation“, „commitment“, „choice“, ...). Anhand dieses Wissens handelt der Agent, so kann er zum Beispiel aus bestehendem Wissen neues Wissen ableiten.

Es gibt dabei zwei Probleme bzw. Schwerpunkte, die bei der Verwendung von Agentensystemen zu beachten sind:

- Das **syntaktische Problem** bezeichnet die Sprache, in der das Wissen und die Kommunikation ausgedrückt wird.
- Das **semantische Problem** bezeichnet das Modell, mit dem das Wissen modelliert wird

Wooldridge führt als Lösung für das syntaktische Problem modale Sprachen sowie Meta-Sprachen (eine Sprache, die Aussagen über die zugrunde liegende Sprache macht) an. Das semantische Problem kann durch „possible worlds semantics“ oder durch den Ansatz der interpretierten symbolischen Strukturen („sentential, or interpreted symbolic structures approach“) gelöst werden.
Die „possible worlds semantics“ beschreibt einen Ansatz, bei dem die Welten („worlds“) die Menge aller möglichen Zustände beschreiben, wobei nur die zulässigen Zustände enthalten sind. Das Wissen oder auch der „Glaube“ des Agents ist dann das, was in allen Welten wahr ist.

2.1.2 FIPA/ACL

Die FIPA (Foundation of Intelligent Physical Agents) ist das Standardisierungsorgan für Agentensysteme. Seit der Gründung 1996 in der Schweiz wurden verschiedene Standardisierungen veröffentlicht, so zum Beispiel auch die Agentenkommunikation (Agent Communication), die als FIPA/ACL (Agent Communication Language, ACL) bekannt geworden ist. Ausgewählte Kommunikationsprotokolle werden nachfolgend detailliert besprochen.

FIPA Referenzmodell

In 2.1 ist ein Überblick der Architektur des FIPA Referenzmodells dargestellt. Die Einheit, in der sich Agenten aufhalten, wird als **Agentenplattform** bezeichnet. Sie befindet sich auf einem Rechner und kann mit anderen Agentenplattformen auf anderen Rechnern kommunizieren. Dabei ist die Agentenplattform selbst nicht standardisiert, sie kann zwischen den verschiedenen Implementierungen des Standards variieren.

Das Gesamtsystem setzt sich nach [fInt02] aus den folgenden Komponenten zusammen.

- **Agenten** („Agent“) - Sie implementieren die eigentliche Funktionalität und kommunizieren mittels der Agentenkommunikationssprache ACL. Sie können Dienste sowohl anbieten als auch konsumieren. Jeder Agent besitzt eine eindeutige Identifikationsnummer („AID“).
- **Dienst-Verzeichnis** („Directory Facilitator“, DF) - Es verwaltet ähnlich zu den „Gelben Seiten“ eine Liste von Diensten und Agenten, die diese anbieten.
- **Agentenverwaltungssystem** („Agent Management System“, AMS) - Es ist für die Verwaltung der Agenten einer Agentenplattform zuständig. Dabei erfolgt eine Abbildung von logischen Agentennummern(AID) auf reelle Adressen der Agenten. Es ist genau ein Agenten-Verwaltungssystem pro Agentenplattform zulässig.

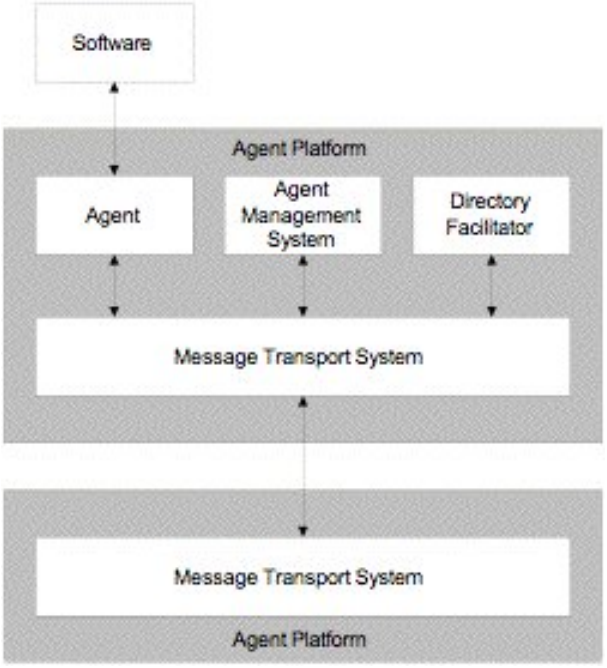

Abbildung 2.1: FIPA-Request Protokoll; Quelle: [fInt02]

- **Nachrichtenübertragungsdienst** („Message Transport Service“, MTS) - Er verwaltet die externe Kommunikation zu weiteren Agentenplattformen.
- **Software** - Sie wird durch Agenten an das System angebunden und implementiert externe Funktionalität.

In der wissenschaftlichen Literatur wurde dieser Standard diskutiert. So führt [PaPS00] an, das die Spezifikation der internen Kommunikation innerhalb der Agentenplattform nicht ausreichend ist.

FIPA Request

Das FIPA Request Protokoll definiert die Aufforderung eines Agenten an einen anderen Agenten, eine Aktion auszuführen. In Abbildung 2.2 ist der Ablauf in UML-Notation grafisch veranschaulicht.

Der fragende Agent („Initiator“) sendet eine Anfrage („request“) an den Empfänger („Participant“). Dieser kann die Anfrage ablehnen („refuse“) oder ihr zustimmen („agree“). Wenn er zugestimmt hat, muss er die Aktion ausführen. Tritt bei der Ausführung ein Fehler auf, schickt er eine Fehlerbenachrichtigung („failure“) an den Aufrufer zurück. Tritt kein Fehler auf, antwortet der Empfänger entweder damit, das die Anfrage ausgeführt wurde („inform-done“) - ohne Rückgabe des Ergebnisses - oder mit dem Ergebnis („inform-result“).

FIPA Query

Das FIPA Query-Protokoll wird für Anfragen verwendet. Dabei kann erfragt werden, ob eine Aussage wahr/falsch ist - was als QUERY-IF bezeichnet wird. Mit QUERY-REF kann nach einer Menge, die eine Anfrage erfüllt, gefragt werden. Grafisch wird

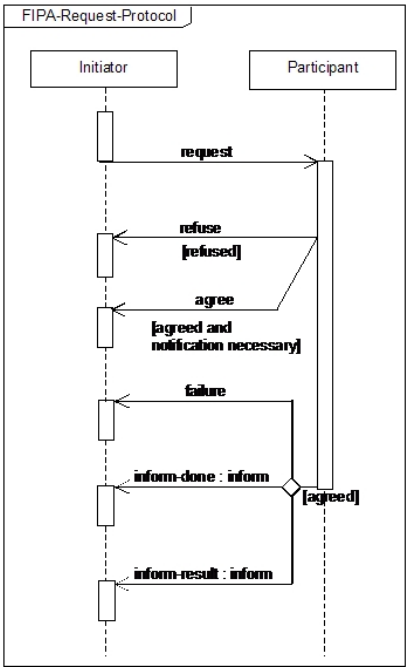

Abbildung 2.2: FIPA-Request Protokoll; Quelle: [fInt03b]

der Ablauf in Abbildung 2.3 dargestellt.

Der fragende Agent („Initiator“) sendet entweder eine Anfrage, ob etwas wahr oder falsch ist („query-if“), oder nach einer Menge, die die gegebene Bedingung erfüllt(„query-ref“). Der Empfänger(„Participant“) kann diese Anfrage ablehnen („refuse“) oder beantworten („agree“). Beim Beantworten sendet er entweder - wenn ein Fehler bei der Abarbeitung der Anfrage aufgetreten ist - eine Fehlernachricht („failure“) oder ein Ergebnis. Dieses kann entweder die Erfüllung der Anfrage (ist sie wahr(„inform-t“) oder falsch(„inform-f“)) oder eine Menge zurückliefern, die die Anfrage erfüllt („inform-result“).

2.1.3 Ontologien

Durch Ontologien wird ein gemeinsames Wissen durch Konzepte und Relationen zwischen diesen Konzepten definiert. Entwickelt wurden sie im Bereich der Künstlichen Intelligenz, werden heute aber sogar im Internet - zum Beispiel für die Kategorisierung von Internet-Seiten - verwendet. Eine sehr gute Einführung in die Verwendung von Ontologien wird in [NoMc01] gegeben.
Ontologien werden in dieser Arbeit verwendet, um das Wissen und die Kommunikation der Agenten zu modellieren. Auch in Dialogsystemen werden Ontologien genutzt, um das interne Wissen und Zusammenhänge darzustellen.

2.2 Technologien

In diesem Abschnitt werden die verwendeten Technologien und Programme vorgestellt. Als Agentenplattform wird JADE verwendet, Ontologien werden mit Protege modelliert.

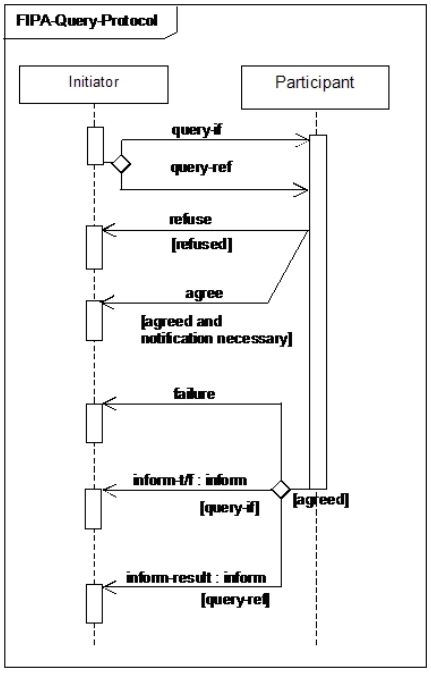

Abbildung 2.3: FIPA-Query Protokoll; Quelle: [fInt03a]

2.2.1 JADE

Eine der bekanntesten Agentenplattformen ist JADE (Java Agent DEvelopment Framework, [1]). Diese ist in Java geschrieben und konform zum FIPA - Standard. Die Implementierung eines Agentensystems wird vereinfacht, da die Kommunikation und Funktionalität bereits in Java - Klassen gekapselt sind, die nur noch erweitert und an die jeweiligen Bedürfnisse angepasst werden müssen.
Die Agenten können auf verschiedenen Rechnern in einem Netzwerk verteilt werden, die Kommunikation erfolgt dabei transparent. JADE bietet Programme an, mit denen sich der Kommunikationsfluss und die vorhandenen Agenten verwalten und überwachen lassen.
JADE wurde als Agentenplattform ausgewählt, da es in der Programmiersprache Java geschrieben und ausgereift ist. Es wird weiterhin aktiv entwickelt und ist unter LGPL lizensiert, kann also frei und ohne Lizenzkosten verwendet werden.
Die Kommunikation zwischen den Agenten kann verschieden kodiert werden. Der SL-Codec konvertiert Nachrichten in ein menschenlesbares Format. Durch Kodierung einer Nachricht mit dem LEAP-Codec können auch mobile Geräte, die nur eine beschränkte Rechen- und Übertragungskapazität besitzen, über das JADE-System kommunizieren. Diese Nachrichten sind allerdings nicht menschenlesbar sondern bytekodiert.
JADE verwendet spezielle Klassen, die das Verhalten der Agenten implementieren - die Behaviour-Klassen. Diese können für den einmaligen Versand einer Nachricht verwendet werden, aber auch alle eingehenden Nachrichten verarbeiten und weitere Funktionen aufrufen. Diese Architektur ist sehr flexibel und kann sehr vielfältig eingesetzt werden.

[1] http://jade.tilab.com/

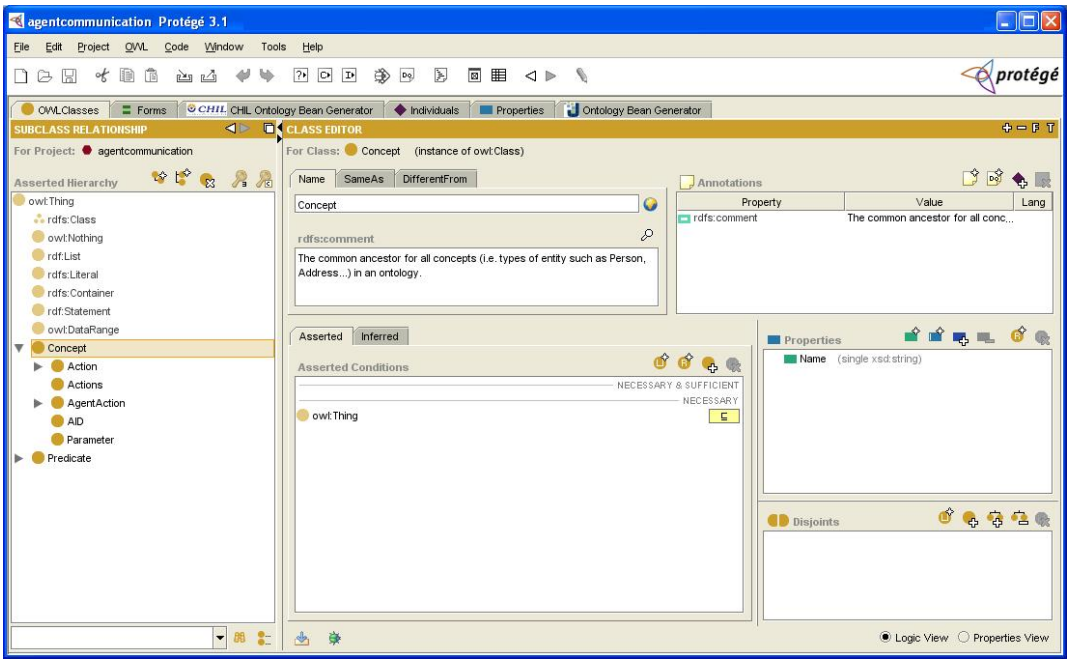

Abbildung 2.4: Oberfläche von Protege

2.2.2 Verwendung von Ontologien in JADE

JADE bietet eine einfache Möglichkeit, die Kommunikation durch Ontologien zu unterstützen. Das hat den Vorteil, dass eine Nachricht typisiert verwendet werden kann; JADE übernimmt dabei die Umwandlung der internen Kommunikationsnachricht (der Inhalt besteht nur aus ASCII-Text) in den entsprechenden Typ der Ontologie und belegt die Werte anhand der übertragenen Nachricht.

Um eigene Konzepte und Beziehungen hinzuzufügen, müssen diese JADE-Schnittstellen implementieren. Agenten-Kommunikationsnachrichten müssen dabei die Schnittstelle *jade.content.AgentAction* implementieren, Ontologie-Konzepte die Schnittstelle *jade.content.Concept* und Ontologie-Beziehungen die Schnittstelle *jade.content.Predicate*. Eine gute Einführung in die Verwendung von Ontologien in JADE geben G. Caire and D. Cabanillas in [CaCa04].

2.2.3 Protege und BeanGenerator

Da die Erstellung von Ontologien allgemein und im Speziellen für JADE ohne geeignete Programme sehr aufwändig sein kann, ist die Verwendung von Hilfsprogrammen empfehlenswert. Für die Verwaltung von Ontologien hat sich Protege der Universität Stanford durchgesetzt. Es ist kostenlos und frei verfügbar, kann durch Plugins erweitert werden und bietet viele Werkzeuge zum Bearbeiten von Ontologien. Abbildung 2.4 stellt die Oberfläche von Protege zum Bearbeiten von Ontologien dar.
Für die Erzeugung von JADE-Ontologien gibt es ein Plugin (*BeanGenerator*, [2]), welches direkt aus der Ontologie die entsprechenden Java-Klassen für JADE erzeugt. Das Plugin wird nicht mehr weiterentwickelt, mit Protege 3.1 und JADE 3.3 funktioniert es jedoch.

[2] http://acklin.nl/page.php?id=34

3. Bestehende Arbeiten

Die Verwendung von Agenten in Dialogsystemen sowie die Anbindung von Applikationen an Dialogsysteme wurden oft untersucht. Es wird kein Überblick über alle bisherigen Ansätze gegeben, sondern es werden nur ausgewählte Arbeiten kurz vorgestellt.

3.1 Agentenbasierte Dialogsysteme

Agentensysteme in Dialogsystemen werden unter anderem in den Ansätzen Jaspis und TRIPS verwendet.

TRIPS

In Allens Veröffentlichung [ABDF+01] von TRIPS wird ein Überblick über das dort verwendete Dialogsystem, Beispiele und Auswertungen von Benutzerstudien sowie ein kleiner Einblick in die Architektur(siehe Abbildung 3.1) des Systems gegeben. Er verwendet Agenten zur internen Kommunikation im Dialogsystem sowie zwischen Dialogsystem und den Anwendungen („Service Providers“), die durch einen Dienstvermittler („Service Broker“) verwaltet werden. Dazu wird ein Problemlösungsmodell entwickelt, das aus einem abstrakten und verschiedenen speziellen Domänenmodellen besteht. In seiner Veröffentlichung führt er auf, das zum abstrakten Modell Ziele (dazu werden von ihm Goals, Subgoals und Constraints gezählt), Lösungen (Aktionen, die weiter in Richtung der Ziele führen), Ressourcen (Objekte und weitere Abstraktionen (wie zum Beispiel Zeit), die in Lösungen verwendet werden) und Situationen (Zustände des Weltmodells) gehören.

Domänen-spezifische Modelle geben Abbildungen zwischen dem abstrakten Modell und den Domänen-abhängigen Operatoren an. Dazu gehören die Objekte, die in einer speziellen Domäne als Ressource zählen, die verfügbaren Lösungen und Ziele.

In der Architektur werden externe Anwendungen mittels einem Verhaltensagenten („behavioral agent“) angebunden, die sowohl über weitere Agenten als auch direkt

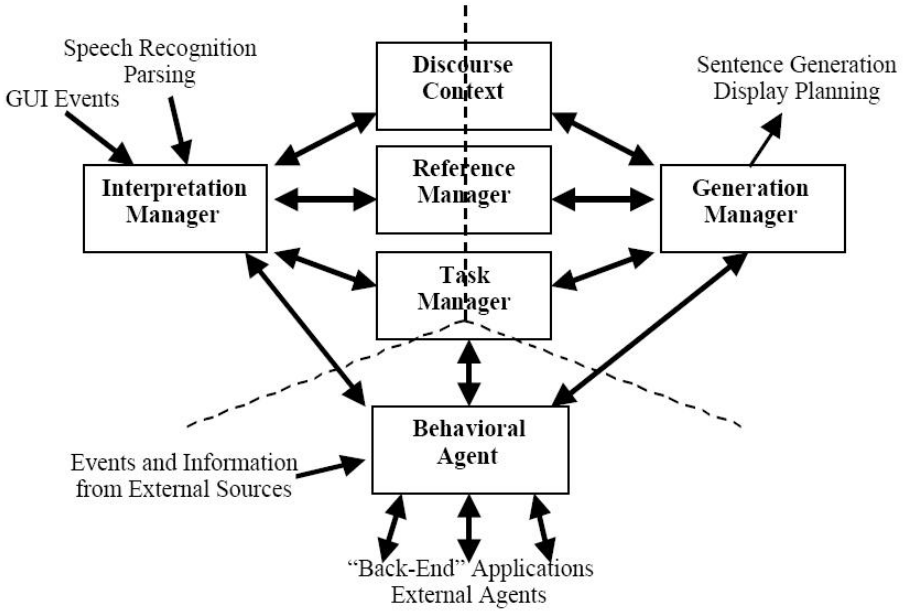

Abbildung 3.1: Architektur TRIPS; Quelle: [ABDF+01]

angesprochen werden können. Leider werden keine Details zur Anbindung der Applikationen veröffentlicht.
In einer weiteren Ausarbeitung „A Problem Solving Model for Collaborative Agents“ (siehe [AlBF02]) werden leider ebenfalls keine Details zur Kommunikation mit den angebundenen Anwendungen gegeben.

Jaspis

Turunen setzt Agenten im Dialogsystem Jaspis ein ([TuHa01]), um es modular und flexibel zusammenbauen zu können. Er unterteilt das Gesamtsystem in die Module Eingabe („Input“), Dialog und Ausgabe („Presentation“). Innerhalb der Module (siehe Abbildung 3.2) gibt es jeweils einen Verwalter („Manager“), Agenten und Gutachter („Evaluator“). Die Gutachter wählen die jeweils zu verwendenden Agenten innerhalb eines Moduls, die Verwalter koordinieren die Agenten und Gutachter. Agenten werden in Jaspis eingesetzt, um die Entscheidungsfindung dezentral auszuführen. Nicht eine zentrale Komponente entscheidet, sondern die Agenten in Verbindung mit den jeweiligen Gutachtern agieren. Sie können mit Hilfe des aktuellen Dialogzustands selbst entscheiden, ob sie ein Ergebnis liefern können.

3.2 Anbindung von Anwendungen an Dialogsysteme

Der einfachste Ansatz zur Anbindung einer Anwendung an ein Dialogsystem ist, diese direkt vom Dialogsystem an bestimmten Stellen im Dialog aufzurufen. Abhängig davon, wie gut diese Anbindung im Dialogsystem implementiert ist, funktioniert das einfach und schnell.

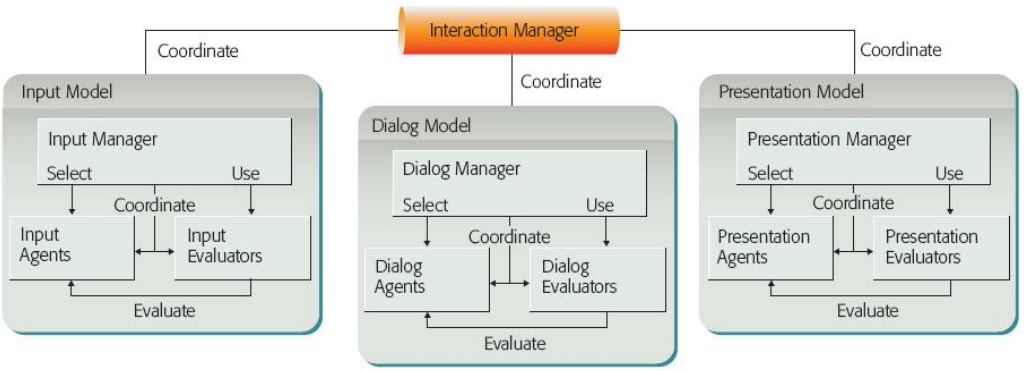

Abbildung 3.2: Architektur Jaspis; Quelle: [TuHa01]

Tapas

Im verwendeten Dialogsystem Tapas[1] muss eine Anwendung gewissen Anforderungen entsprechen (Konstruktor ohne Parameter, get- und set-Funktionen für den Zugriff auf Variablen), in eine JAR-Datei übersetzt werden und in den Konfigurationsdateien eingefügt werden. Dann kann die Anwendung über einen Funktionsaufruf, wie in 3.1 dargestellt, angesprochen werden.

Listing 3.1: Definition eines Funktionsaufrufes einer externen Anwendung aus Tapas

```
jpkg://localhost:5454/playSong   $sem.[generic:ARG|FILENAME],
        $objs.[generic:ARG|generic:NAME];
```

Weitere Informationen zur Erweiterbarkeit im Dialog sind in [Dene02] zu finden. Das Tapas-Tutorial ([HoGi05]) stellt die Verwendung von Tapas Schritt für Schritt vor.

Webservices

Eine andere Möglichkeit Anwendungen an ein Dialogsystem anzubinden, stellt die Verwendung von Webservices dar. Diese werden dynamisch zur Laufzeit, beim Start des Dialogsystems oder bei der Entwicklung des Dialogs eingebunden.
Portabella beschreibt in [PoRa04] einen Ansatz, bei dem semantisch beschriebene Webservices aus einem Verzeichnis ausgewählt und direkt zur Laufzeit ausgeführt werden. Weitere Informationen, insbesondere zur Implementierung sind noch nicht veröffentlicht.

[1] http://isl.ira.uka.de/ hartwig/tapas.html

4. Entwurf

4.1 Idee

Die Anbindung externer Anwendungen soll durch die Verwendung von Agenten flexibel und dynamisch erfolgen. Die Verfügbarkeit der Dienste soll sich zur Laufzeit ändern dürfen - so sollen neue Dienste hinzugefügt werden können, andere können aber während der Laufzeit (temporär) ausfallen und nicht mehr erreichbar sein.

Der entwickelte Ansatz sollte diese Anforderungen in Betracht ziehen - außerdem soll die Entwicklung von Dialogmodellen vereinfacht werden. Wenn möglich, soll die Entwicklungsdauer und der menschlich-intellektuelle Aufwand verringert sowie die benötigten Datenstrukturen vereinfacht werden.
Ontologien sollen verwendet werden, um die Dienste der Anwendungen zu beschreiben und diese Informationen in der Dialogerzeugung verwenden zu können.

4.2 Architektur

Das Dialogsystem und die Anwendungen werden durch Agenten an das Agentensystem gekoppelt. Diese kommunizieren miteinander, um die verfügbaren Aktionen zu bestimmen, ausführbare Dienste zu bestätigen, fehlende Informationen zu bestimmen und Dienste auszuführen.
Abbildung 4.1 stellt dies grafisch dar. Weitere Informationen sind im Kapitel Implementierung unter 5.1 zu finden.

4.3 Kommunikation

Die Kommunikation zwischen Dialogsystem und Anwendung wird in diesem Absatz vorgestellt. Anhand eines Beispiels wird die Idee, welche dabei verfolgt wird, erläutert. In Abbildung 4.2 und 4.3 wird die Kommunikation schematisch dargestellt. Es sind in diesem Beispiel drei Objekte dargestellt - der Benutzer, das Dialogsystem und die Applikation. Je nach Aufbau des Systems können mehrere Applikationen und ggf. mehrere Benutzer vorhanden sein; dies wird im hier betrachteten Beispiel

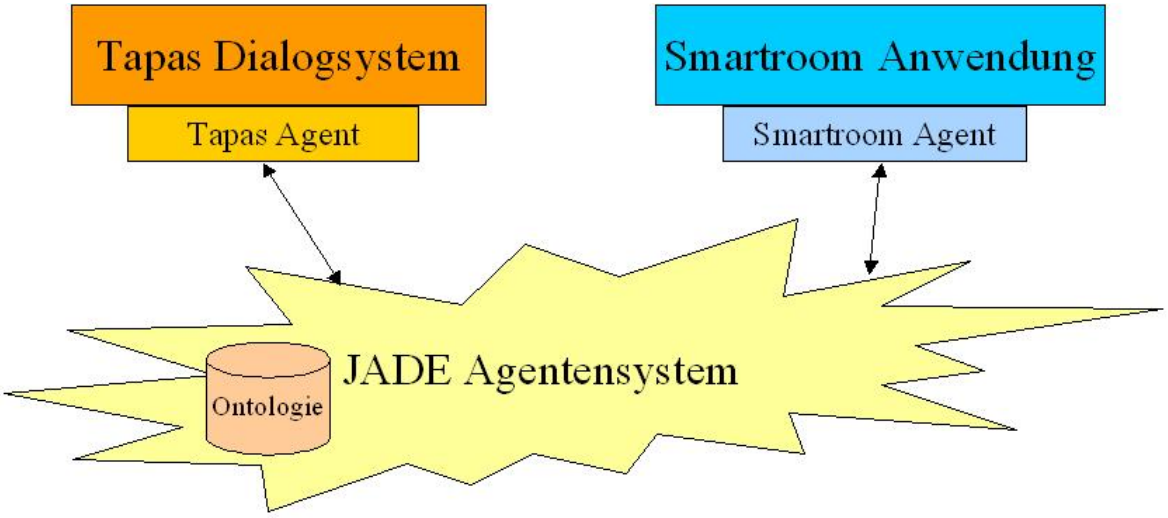

Abbildung 4.1: Übersicht Architektur

vernachlässigt, um einen ersten, einfachen Ablauf darzustellen.

In diesem Beispiel startet der Benutzer die Interaktion mit dem Dialogsystem durch die Aussage „Führe Aktion *inform* aus“. Diese Aussage wird vom Spracherkenner erkannt und an das Dialogsystem weitergeleitet. Nach dem Parsen wird diese Information verarbeitet. Dazu wird eine interne Funktion aufgerufen, in der überprüft wird, ob die vom Nutzer gewünschte Aktion *inform* ausführbar ist. Das Dialogsystem fragt dies bei der angemeldeten Applikation mittels einer FIPA-Nachricht an. Die Anwendung weiß, welche Aktionen sie ausführen kann und welche Informationen zur Ausführung benötigt werden.
Die Antwort der Applikation gibt an, ob die Aktion verfügbar ist. In diesem Beispiel ist sie es. Das Dialogsystem verarbeitet diese Information und fragt bei der Anwendung nach fehlenden Informationen. Für das Ausführen der Aktion *inform* wird die Information *Nachricht* benötigt, welche ebenfalls in einer FIPA-Antwortnachricht an das Dialogsystem gesendet wird.
Das Dialogsystem kann die fehlende Information *Nachricht* nicht aus vorhandenem Wissen mit einem Wert belegen und startet eine Rückfrage beim Benutzer mittels Sprachausgabe. In Abbildung 4.3 antwortet der Nutzer mit „Ich verspäte mich“, was vom Dialogsystem als die fehlende Information *Nachricht* erkannt wird. Da damit alle benötigten Informationen zum Ausführen der Aktion *inform* mit Werten belegt sind, kann die Aktion ausgeführt werden. Dazu sendet das Dialogsystem eine FIPA-Nachricht mit der auszuführenden Aktion und den benötigten Informationen an die Applikation, welche die Aktion ausführt und eine Bestätigung an das Dialogsystem zurück sendet. Dem Nutzer wird durch eine Sprachausgabe mitgeteilt, dass die von ihm gewünschte Aktion ausgeführt wurde.
Weitere Details werden im Kapitel Implementierung (siehe 5.2) vorgestellt.

Aus diesem Beispiel lassen sich folgende Agenten-Aktionen ableiten, auf welchen die Kommunikation basiert:

- **IsActionAvailable** fragt die Verfügbarkeit einer Aktion an.

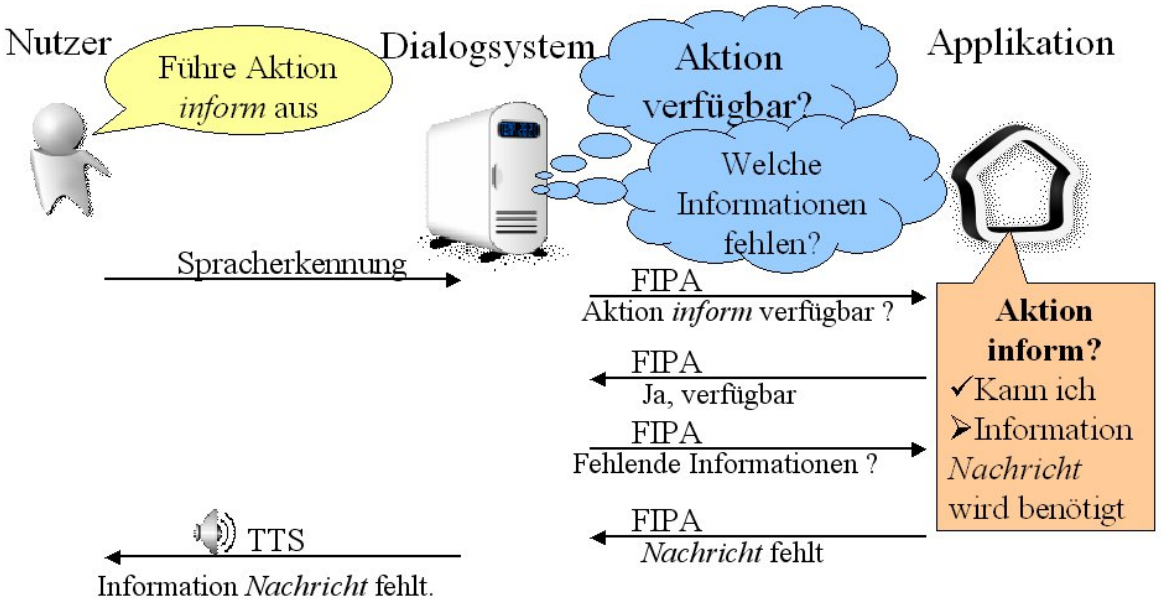

Abbildung 4.2: Beispielkommunikation Teil 1 - Anfrage und Rückfrage

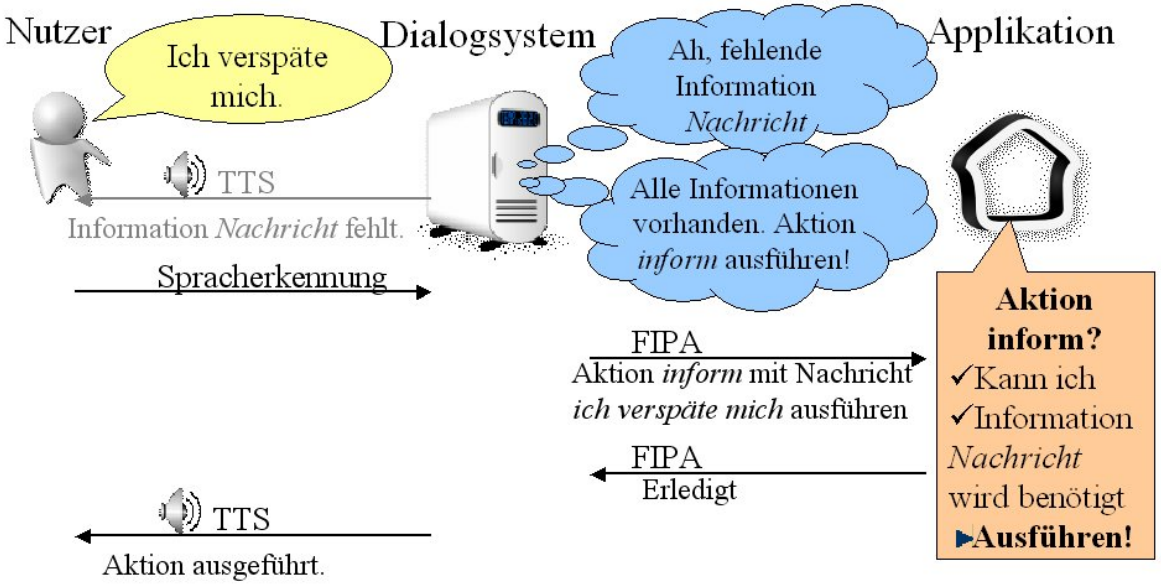

Abbildung 4.3: Beispielkommunikation Teil 2 - Antwort und Ausführung

- **GetMissingParameter** erfragt fehlende Informationen zur Ausführung einer Aktion.
- **InvokeAction** führt eine Aktion mit angegebenen Informationen aus.

Als Parameter dieser Agenten-Aktionen werden folgende Konzepte verwendet.

- **Action** bezeichnet eine Anwendungsaktion (Dienst) und enthält weitere Informationen: den Namen der Aktion; eine kurze Beschreibung; eine Menge von Informationen, die zur Ausführung benötigt werden.
- **Parameter** bezeichnet die Information, welche zum Ausführen einer Anwendungsaktion benötigt wird. Das Konzept umfasst einen Namen, eine Beschreibung, einen Wert und - sofern vorhanden - eine Menge von Werten, welche für diese Information möglich sind.

Die Agenten-Aktionen und die damit verbundenen Konzepte werden in einer gemeinsamen Ontologie definiert.

4.4 Anbindung des Dialogsystems

Die Anbindung an das Dialogsystem soll die im Kapitel 4.3 vorgestellte Kommunikation möglichst einfach, schnell und geeignet kapseln. Es bietet sich an, die Kommunikation und den Dialogsystem-Agenten unabhängig von dieser speziellen Agentenkommunikation zu implementieren. Dieser allgemeine Agent wird dann um die spezifische Kommunikation erweitert, welche für die Kommunikation mit den Applikationen zur Anfrage im Rahmen des Kommunikationsprotokolls benötigt werden.

Das Dialogsystem muss in der Lage sein, Anwendungen nach der Verfügbarkeit von Aktionen zu fragen, fehlende Informationen von Aktionen zu erfragen und Aktionen mit den benötigten Informationen auszuführen. Außerdem müssen die Antworten der Anfragen verarbeitet und für das Dialogsystem geeignet aufbereitet werden.
Beim Systemstart sollte die Menge der ausführbaren Aktionen bei allen angemeldeten Applikationsagenten erfragt werden, um damit ein initiales Dialogmodell aufbauen zu können. Die zum Aufbau der Datenstrukturen benötigten Informationen müssen bei den Applikationen erfragt und übernommen werden. Sollten Dienste von Anwendungen ausfallen oder neue hinzukommen, müssen diese in das Dialogmodell übernommen werden.

4.5 Anbindung der Anwendungen

Anwendungen, die mit dem Dialogsystem kommunizieren möchten, müssen auf die im Kapitel 4.3 und 4.4 vorgestellten Kommunikationsaktionen valide Antworten senden.
Beim Starten einer Anwendung müssen sie die Dienste, welche sie anbieten, dem Dialogsystem mitteilen.

5. Implementierung

Die bisher vorgestellten Konzepte und Software-Komponenten werden in diesem Kapitel zu einem Gesamtsystem verbunden, welches die gestellten Forderungen erfüllt und praktisch eingesetzt werden kann.

5.1 Architektur

Als Agentensystem wird das in Kapitel 2.2 vorgestellte JADE verwendet. Da sowohl das Dialogsystem als auch die verwendete Applikation in Java geschrieben sind, können diese durch Agenten direkt an das Agentensystem angebunden werden. Anwendungen, die nicht in Java geschrieben sind, müssen mittels eines in Java geschrieben Agentenstellvertreters an das Agentensystem angebunden werden. Dies stellt eine zusätzliche Belastung des Systems durch eine verzögerte Kommunikation und eine aufwändige Transformation der Datenstrukturen dar.
Die Jade-Agenten müssen eine gemeinsame Kommunikationsontologie verwenden, um miteinander kommunizieren zu können. Es werden „Behaviour“-Klassen, welche die Verarbeitung der Jade-Nachrichten erledigen, für die Agentenaktionen erstellt und beim Initialisieren der Agenten geladen.

5.2 Kommunikation

Die zur Kommunikation verwendeten Typen werden in der Ontologie beschrieben. Dabei sind zwei Arten von Aktionen zu unterscheiden: die von JADE definierten Agentenaktionen („AgentAction“), die für die Agentenkommunikation verwendet werden, und die Anwendungsaktionen (Dienste), welche als „Nutzlast“ dieser Agentenkommunikation benutzt werden und in der Ontologie als Konzept „Action“ definiert wurden.

In Abbildung 5.1 ist die Ontologie dargestellt; farbig hinterlegte Einträge werden dabei von JADE vorgegeben.

Die bereits in Kapitel 4.3 vorgestellten ontologischen Konzepte werden nachfolgend detailiert vorgestellt:

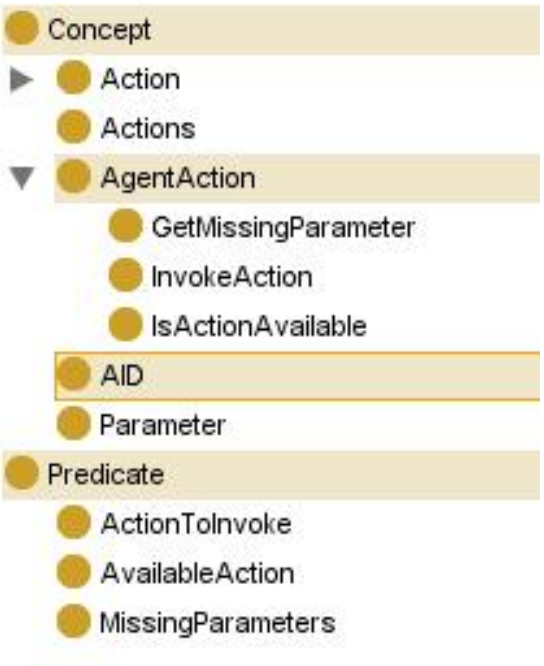

Abbildung 5.1: Ontologie

Anwendungsaktion

Eine Anwendungsaktion (Ontologie-Typ „Action“) besteht dabei aus einem Namen und einer Menge von Parametern. Ein Parameter besitzt einen Namen und eine Beschreibung („Description“), die zur Dialogerzeugung und intern für die Smartroom-Anwendung benötigt werden. Da die Smartroom-Software intern ein dynamisches Aktionsmodell verwendet (siehe Anhang A.3) und für dieses spezielle Informationen über die fehlenden Parameter zur Laufzeit benötigt, wurde der Typ Parameter um die Eigenschaften *Classname* (der Klassenname des Parameters - typischerweise String) und *Value* (dem Wert des Parameters) erweitert.

Agentenaktionen

Es existieren Funktionen, um die Verfügbarkeit einer Aktion zu erfragen, die fehlenden Parameter einer Aktion herauszufinden und eine Funktion mit gegebenen Parametern auszuführen.

Aktion verfügbar Das Dialogsystem kann erfragen, ob eine Aktion aktuell verfügbar ist. Dafür wurde die Agentenaktion *IsActionAvailable* definiert, die eine Anwendungsaktion als Parameter aufnimmt. Die Applikation antwortet entweder mit wahr oder falsch. Der Rückgabewert für eine verfügbare Aktion ist dabei die angefragte Aktion; ist die Aktion nicht ausführbar, dann wird diese mit einem NOT-Konstrukt gekennzeichnet.
Um mit Jade eine FIPA-konforme Nachricht für diese Anfrage zu erzeugen, muss ein Prädikat in der Ontologie definiert werden und dieses zur Laufzeit mit den entsprechenden Parametern ausgefüllt werden. Der für diese Anfrage verwendete FIPA-Nachrichtentyp ist QUERY-IF.

Parameter der Aktion erfragen Die Agentenaktion *GetMissingParameter* dient dazu, bei einer Anwendung die fehlenden Parameter für das Ausführen einer Aktion zu erfragen. Dabei können zusätzlich bereits belegte Parameter zur Aktion gesendet werden, die von dieser dann beachtet werden. Die Antwort auf diese Anfrage enthält eine Menge von Parametern, die dann vom Dialogsystem ausgefüllt werden müssen. Analog zur vorherigen Agentenaktion wird ebenfalls ein Prädikat benötigt, um mittels Jade eine FIPA-konforme Anfragenachricht zu erzeugen. Der Nachrichtentyp dieser Anfrage ist QUERY-REF.

Aktion ausführen Ist die Aktion verfügbar und alle benötigten Parameter mit Werten belegt, kann diese ausgeführt werden. Dazu wird die Agentenaktion „InvokeAction“ vom Dialogsystem aufgefüllt, an die Anwendung gesendet, von dieser ausgeführt und die Ausführung an das Dialogsystem zurückgeliefert. Tritt bei der Ausführung ein Fehler auf, wird die Ausführung der Aktion abgebrochen und eine Fehlerbenachrichtigung an das Dialogsystem gesendet.
Diese Aktion kann direkt als AgentAction in Jade in die Nachricht eingepackt werden, allerdings führte das zu nicht behebbaren Fehlermeldungen. Selbst die Jade-Entwicklermailingliste konnte keine Lösung dafür liefern. Deswegen wurde diese Nachricht ebenfalls als Prädikat in eine FIPA-konforme Nachricht eingepackt. Diese Nachricht wird als Nachrichtentyp FIPA-REQUEST versendet.

Alternative

Eine Alternative zu der hier verwendeten Kommunikation, welche durch das Dialogsystem gesteuert wird, wurde ebenfalls untersucht. In diesem Kommunikationsszenario führt das Dialogsystem die Aktion direkt aus, die Anwendung erfragt selbstständig bei dem Dialogsystem fehlende Informationen. Dieses Vorgehen kollidiert mit dem Dialogmodell, in welches die Rückfragen der Anwendung aufwändig integriert und mit der normalen Dialogabarbeitung synchronisiert werden müssen. Die Anpassungen dafür sind tiefgreifend.
Die ursprüngliche Idee, alle Anwendungsaktionen in die Ontologe aufzunehmen, wurde schlussendlich nicht umgesetzt. Im Rahmen dieser Arbeit würde es nur die Ontologie mit einer Vielzahl von Konzepten erweitern, deren Modellierung den Rahmen dieser Arbeit sprengen würde. Im Kapitel 7 wird darauf eingegangen.

Bewertung

Aktuell werden die von JADE versendeten Nachrichten mit dem SL-Codec kodiert. Dieser hat den Vorteil, dass er menschenlesbar und interoperabel mit anderen FIPA-konformen Systemen einsetzbar ist. Es gibt weitere Codecs, welche die Nachrichten maschinenlesbar kodieren. Diese sind aber für die Entwicklung ungeeignet, da die versendeten Nachrichten nicht menschenlesbar sind und deswegen nicht kontrolliert werden können. Außerdem ist die Interoperabilität mit anderen FIPA-konformen Systemen, die nicht diesen Codec verarbeiten können, nicht mehr möglich.

Ausblick

Aktuell wird die Ontologie beim Starten des Dialogsystems geladen und daraus die Grammatiken generiert. Das könnte auch dynamisch zur Laufzeit erfolgen, dazu

müsste eine weitere Agentenaktion definiert werden. Das Dialogsystem könnte diese bei der Initialisierung senden, um die internen Dialogstrukturen aufzubauen. Sollte eine Anwendung während der Laufzeit hinzu kommen, müsste es eine solche Nachricht versenden, die von dem Dialogsystem gelesen und verarbeitet wird. Eine andere Lösung wäre, dass das Dialogsystem beim Verarbeiten von Nutzereingaben eine solche Nachricht an alle verfügbaren Agenten schickt, die dann jeweils ihre Aktionen an das Dialogsystem senden; der Nachteil dieses Ansatzes ist offensichtlich: der Kommunikationsaufwand ist sehr hoch und fügt eine Verzögerung für das Dialogsystem hinzu.

5.3 Dialog

Das Dialogsystem Tapas ist sehr modular aufgebaut, neue Dienste können sehr einfach und schnell entwickelt und eingebunden werden. Tapas wurde dahingehend erweitert, dass es über JADE mit anderen Agenten kommunizieren kann und diese Informationen zur Laufzeit in das Dialogsystem übernehmen kann. Weiterhin wird dynamisch die Ausführbarkeit von Aktionen abgefragt, fehlende Parameter erfragt und durch Rückfragen mit dem Nutzer ausgefüllt und anschließend die Aktion mit den ausgefüllten Parametern ausführt.
Es sind dabei zwei Änderungen am Kern des Dialogsystems vorzunehmen:

- die **Anbindung an JADE**, Kapselung des Dialogsystems durch einen Agenten und Erstellung von Funktionen, die die Agentenkommunikation (insbesondere der drei Agentenaktionen) implementieren
- die **Einbindung von verfügbaren Anwendungsaktionen**, die aus der Ontologie entnommen und in die erzeugende Grammatiken übernommen werden soll

Diese Funktionalitäten sollen orthogonal zum bisherigen Dialogsystem und deren Definition erfolgen. Insbesondere sollen diese unabhängig vom Dialogsystem nutzbar sein, das Dialogsystem auch ohne die neuen Funktionen wie bisher verwendbar sein. Vom Dialogsystem benutze Dateien im ADL2- und Grammatik-Format zur Spezifikation des Dialogs sollen weiterhin verwendbar sein, die neue Funktionalität muß ohne Erweiterung dieser Formate einfach und transparent genutzt werden können.
In den nächsten Abschnitten wird die Umsetzung der Änderungen im Detail vorgestellt.

Anbindung an JADE

Das Dialogsystem wird durch einen JADE-Agenten repräsentiert, der sich mit dem Agentensystem registriert. Er stellt keine Dienste zur Verfügung, er stellt Anfragen an Anwendungsagenten und konsumiert die Antworten.
Es wird jeweils eine Funktion für die Agentenaktionen implementiert: *sendIsActionAvailable*, *sendGetMissingParameters*, *sendInvokeAction*. Darin wird jeweils die Kommunikationsnachricht erzeugt, die entsprechende Anwendungsaktion und - wenn vorhanden - Parametern gesetzt, diese Nachricht an verfügbare Agenten geschickt und die erhaltene Antwort an das Dialogsystem zurück gegeben.

Für Anfragen nach der Verfügbarkeit einer Aktion (*IsActionAvailable*) wird das FIPA Query-IF Protokoll verwendet und implementiert. Die Nachricht wird mittels einem abstrakten, beschreibendem Konzept und einem Prädikat, welches die entsprechende Aktion einbindet, kodiert; dies ist notwendig, damit die negative Antwort, welche mit NOT kodiert wird, FIPA-konform zurückgegeben und verarbeitet werden kann.
Analog dazu wird für die Anfrage nach fehlenden Parametern (*GetMissingParameters*) das FIPA Query-REF Protokoll verwendet. Dafür müssen ebenfalls abstrakte Typen und ein Prädikat verwendet werden, um die Antwort, welche eine Menge von Parametern enthält, FIPA-konform zu kodieren.
Die Nachricht, welche für das Ausführen der Anwendungsaktion an die Anwendungsagenten geschickt wird, enthält die Aktion als Prädikat und die entsprechenden Parameter - ohne spezielle Kodierung durch abstrakte Datentypen.

Dialogerzeugung

Die vom Dialogsystem verwendeten Strukturen und Grammatiken werden in verschiedenen Dateien spezifiziert, welche beim Start geladen und verarbeitet werden. Diese Dateien wurden angepasst, um die Aufnahme der verfügbaren Anwendungsaktionen in den Dialog möglich zu machen. Tapas bietet die Möglichkeit, Objektdatenbanken zu spezifizieren, welche zur Laufzeit abgefragt werden und in das Dialogmodell übernommen werden können. Es wurde eine Adapterklasse implementiert, die die Schnittstelle zur Objektdatenbank von Tapas implementiert und die verfügbaren Anwendungsaktionen zurück liefert. Damit können für die Erzeugung der Grammatiken die verfügbaren Aktionen zur Laufzeit eingelesen werden.

In Listing 5.1 wird die verwendete Grammatik dargestellt.

Listing 5.1: Verwendete Grammatik im Dialogsystem

```
public <act_selectaction,VP,_> = 'please'* <execute> 'the'*
        'action' <obj_action,N,_> { generic:ARG obj_action } 'please'*;
public <act_setParameter,VP,_> = <setParam_before>*
       <obj_information,VP,_> { PARAMETER obj_information }
       <setParam_after>*;

<execute> = 'invoke' : 'execute' : 'run';
<setParam_before> = 'the'* 'information' 'is';
<setParam_after> = 'thank' 'you' : 'thats' 'it';

<obj_information,VP,_> = <obj_message,VP,_>
        : <obj_parameter_fixedValue,VP,_>
        : <Message_User>;
<obj_message,VP,_> =  'i' 'will' 'be' 'late' {VALUE "i will be late"}
        :'i' 'am' 'late'        { VALUE "i am late" };

<Message_User> = 'call' 'me' { VALUE "call me"}
        : 'please' 'call' 'me' { VALUE "please call me"}
        : 'call' 'me' 'please' { VALUE "call me please"};

<obj_action,N,_> = import jpkg://localhost:5454/AgentCommunication?jpkg
        Action name { NAME import };
<obj_parameter_fixedValue,VP,_> = import
        jpkg://localhost:5454/AgentCommunication?jpkg FixedValue
        value { VALUE import };
```

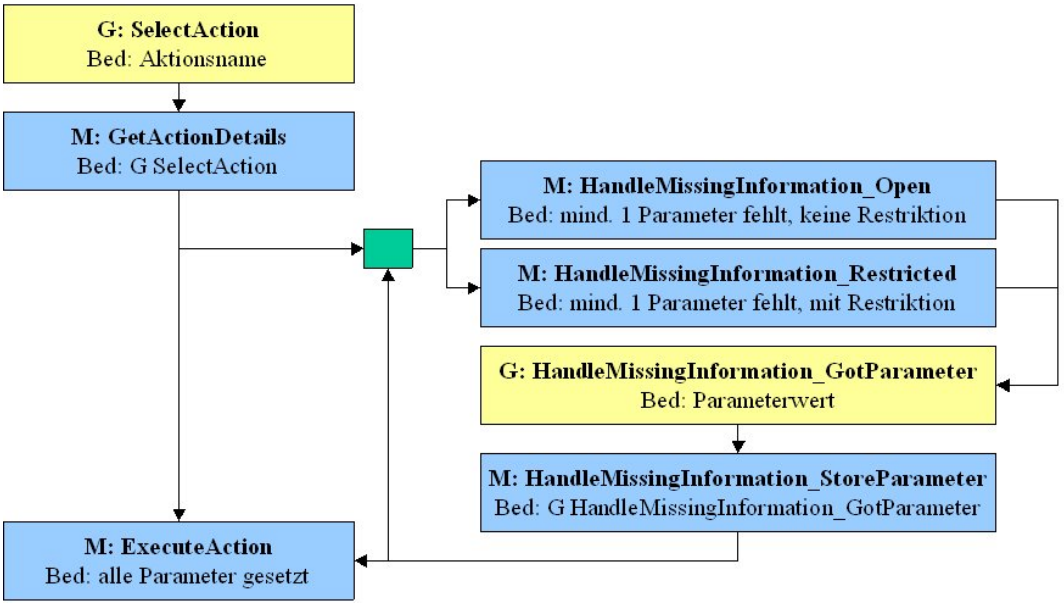

Abbildung 5.2: Interner Ablauf im Dialogsystem

Die beiden letzten Zeilen spezifizieren den Zugriff auf die Objekt-Datenbank (erkennbar durch *import*), aus welcher die verfügbaren Aktionen dynamisch ausgelesen werden. Eine Aktion kann durch die Aussage „invoke action ...“ ausgeführt werden. Dabei wird das in 4.3 entworfene Agentenkommunikationsprotokoll ausgeführt. Die Fähigkeit des Dialogsystems, durch kleine Skripte das Dialogsystem zu steuern, wurde verwendet, um die Logik des Agentenkommunikationsprotokolls zu implementieren. In Abbildung 5.2 wird die erstellte interne Struktur im Dialogsystem dargestellt.

Fordert der Nutzer das Dialogsystem auf, eine Aktion auszuführen, wird das Goal (in der Grafik als *G* gekennzeichnet) *SelectAction* ausgewählt und ausgeführt. Sobald dieses Dialogziel finalisiert ist, wird der Move (in der Abbildung als *M* gekennzeichnet) *GetActionDetails* aufgerufen, der die fehlenden Parameter dieser Aktion bei der Applikation anfragt und diese in der internen Datenstruktur abspeichert. Müssen fehlende Parameter vom Benutzer erfragt werden, werden die Moves *HandleMissingInformation_Open* oder *HandleMissingInformation_Restricted* ausgeführt. Der „offene“ Move (HandleMissingInformation_Open) greift dann, wenn ein Parameter keinen Wert besitzt und keine vorbelegten Werte anbietet. Daraufhin kann der Nutzer eine beliebige Antwort geben. Im Gegensatz dazu wird der einschränkende Move (*HandleMissingInformation_Restricted*) dann aufgerufen, wenn ein Parameter keinen Wert besitzt, aber eine Menge von möglichen Werten anbietet.
In diesen Moves wird die Beschreibung des Parameters vom Dialogsystem ausgegeben und - sofern vorhanden - die Liste der möglichen Werte. Anschließend wartet das Dialogsystem auf die Antwort des Nutzers. Antwortet der Nutzer, wird das Goal *HandleMissingInformation_GotParameter* aufgerufen - sofern der Nutzer einen gültigen Wert eingegeben hat. Anschließend feuert der Move *HandleMissingInformation_StoreParameter*, in dem dieser Wert dem zugehörigen Parameter zugeordnet und gespeichert wird. Bei einer ungültigen Antwort wird der Nutzer aufgefordert, die Information noch einmal einzugeben.
Diese Abfrage wird für alle Parameter wiederholt, bis der Nutzer alle fehlenden

Informationen eingegeben hat oder die Abarbeitung abbricht. Sind alle fehlenden Informationen gesetzt, wird der Move *ExecuteAction* aufgerufen. In dessen Abarbeitung werden die Informationen in die ontologische Konzepte verpackt und an die Anwendung zur Ausführung gesendet.

Der Quellcode dieser Aktionen ist im Anhang B zu finden.

Bewertung

Die in diesem Absatz vorgestellte Anbindung erfüllt die gestellt Aufgabe. Eine direkte Einbindung in den Kern des Dialogsystems wäre wünschenswert, konnte aber im Rahmen dieser Arbeit nicht umgesetzt werden. Das Dialogsystem ist sehr modular und flexibel aufgebaut, insbesondere die Verwendung von Jython innerhalb der Dialogbeschreibung in einer ADL2-Datei ist sehr gut für diese einfache und flexible Anbindung der Kommunikation an das Dialogsystem geeignet.

Ausblick

Die bereits im Absatz *Bewertung* erwähnte direkte Einbindung in das Dialogsystem ist der nächste Schritt für eine weitere Entwicklung auf Seite des Dialogsystems. Auch könnte untersucht werden, wie die Rückfrage nach fehlenden Informationen weiter und besser in das Dialogsystem integriert werden kann.

5.4 Applikation

Anwendungen, die mit dem Dialogsystem kommunizieren wollen, müssen die in 5.2 definierten Agentenaktionen verstehen und beantworten können. Die Smartroom-Anwendung wird - ebenfalls wie das Tapas-Dialogsystem - durch einen Agenten im Agentensystem repräsentiert.

Für die Referenz-Implementierung der Smartroom-Anwendung wurde dessen Besonderheit genutzt, Aktionen nicht im Quellcode hart zu kodiert sondern zur Laufzeit dynamisch auswählen zu können. Dadurch muss der Zugriff auf diese Aktionen nicht für jede einzelne Funktion durch Quellcode ausformuliert werden, sondern kann zentral über eine Funktion erfragt werden.

Normale Anwendungen, die diese Funktionalität nicht anbieten, müssen - wie bei anderen Middleware-Ansätzen - diese Zugriffe auf Aktionen direkt im Quellcode ausformulieren. Das ist langwierig und umständlich, wird meist nur durch Kopieren-und-Einfügen der einzelnen Quellcode-Abschnitte durchgeführt und führt dadurch oft zu versteckten Fehlern.

Für die drei Agenten-Kommunikationsnachrichten wurde jeweils eine JADE-Behaviour-Klasse implementiert, die für die Bearbeitung dieser Nachrichten zuständig sind. Beim Starten des Smartroom-Agenten werden diese für eingehende Nachrichten registriert und automatisch vom Agentensystem aufgerufen, wenn entsprechende Nachrichten eintreffen.
In den folgenden Absätzen werden die Details der Umsetzung der Agentenkommunikation auf in der Applikation vorgestellt.

Verfügbarkeit einer Aktion

Die eingehende Agentenaktion wird entpackt, aus der Nutzlast dieser Nachricht die Anwendungsaktion extrahiert und bei der Smartroom-Anwendung die Verfügbarkeit dieser Anwendungsaktion angefragt. Im positiven Fall - die Aktion ist verfügbar - wird die Anwendungsaktion zurück geschickt; im negativen Fall wird die Antwort NOT-kodiert zurück gesendet.

Anfrage nach fehlenden Parametern

Analog zum vorhergehenden Absatz wird die Anwendungsaktion entpackt, bei der Smartroom-Anwendung die Liste der Parameter dieser Aktion angefragt, diese in das entsprechende ontologische Konzept umgewandelt und anschließend FIPA-konform kodiert zurück gesendet.

Ausführen von Aktionen

Die zuständige Behaviour-Klasse entpackt die Agentennachricht, überprüft die Ausführbarkeit der Aktion - trifft dies zu, dann wird eine Bestätigung zurück geschickt; trifft dies nicht zu, wird die Ausführung abgelehnt. Wenn die Ausführung bestätigt wurde, ruft das Agentensystem automatisch eine Funktion auf, in der dann die eigentliche Ausführung der Aktion stattfindet. Dieses Vorgehen wurde verwendet, da die Ausführung einer Aktion länger dauern kann - das Dialogsystem aber sofort eine Nachricht benötigt, um den Nutzer informieren und die weitere Dialogverarbeitung entsprechend anstoßen zu können.
Wurde die Aktion erfolgreich beendet, wird dies dem Dialogsystem mitgeteilt - tritt hingegen ein Fehler bei der Ausführung innerhalb der Smartroom-Anwendung auf, dann wird die Ausführung der Aktion abgebrochen und eine Fehlermeldung an das Dialogsystem gesendet.

Bewertung

Die Anbindung an das Agentensystem ist relativ einfach zu lösen. Die geforderte Funktionalität wird implementiert.

Ausblick

In dieser Arbeit wird nur eine Applikation verwendet. Werden mehrere Applikationen genutzt, müssen ggf. ähnliche Dienste mit unterschiedlichen benötigten Informationen genauer untersucht werden. Dabei spielt auch die Dienstgüte, im Englischen als *quality of service* (QoS) bezeichnet, eine große Rolle.

6. Ergebnisse und Diskussion

Ein dynamisches Dialogsystem muss sich mit einem normalen, händisch erzeugten und optimierten Dialogsystem (nachfolgend *statisch* genannt) messen. Es ist davon auszugehen, dass dieses optimal ist und von einem automatisch erzeugten nur schwer nachgebildet werden kann. Das Ziel dieser Evaluation ist es, den Unterschied zwischen diesen beiden Dialogsystemen zu messen.

6.1 Szenario

Den Testprobanden wurden folgende Aufgaben gestellt:

1. Senden Sie eine Nachricht „i am late“ an alle Personen im Smartroom.
2. Senden Sie eine Nachricht an Frank, welcher im Smartroom ist. Diese lautet „call me, please“.

Diese sollten per Sprache auf Englisch mit Hilfe der zur Verfügung stehenden Aktionen ausgeführt werden. Dabei konnte die Ausführung der Aktionen direkt im Smartroom live verfolgt werden. Die Nutzer erhielten durch das Dialogsystem eine akustische Rückmeldung.

Dieses Szenario wurde ausgewählt, da die Anwendung diese Aktionen direkt zur Verfügung stellt, das Dialogsystem kompakt gebaut werden konnte und einen abgeschlossenen Aufgabenbereich abdeckt. Die relativ großen Restriktionen (Vorgabe der zu übermittelnden Nachrichten, Vorgabe der zur Verfügung stehenden Aktionen und die geringe Anzahl von Aktionen) war nötig, um das System im Rahmen dieser Studienarbeit stabil und benutzbar bauen und testen zu können.

Die Dialogsysteme waren bezüglich den Rückmeldungen an den Nutzer ähnlich, aber nicht komplett gleich. Das dynamische System gab detaillierte Informationen über darüber aus, was vom Benutzer erwartet und von ihm eingegeben wurde; das statische System hingegen war zurückhaltend mit Äußerungen an den Nutzer.

6.2 Ergebnisse

An der Benutzerstudie haben acht Testprobanden teilgenommen, männlich und weiblich. Die meisten der Personen hatten keine Erfahrung mit Dialogsystemen und deren Fähigkeiten und Grenzen.

Die Teilnehmer mussten die zwei Aufgaben jeweils mit dem statischen und dem dynamischen System ausführen. Um die Lerneffekte beim zweiten Ausführen der Aufgaben zu minimieren, wurden die Dialogsysteme in unterschiedlicher Reihenfolge verwendet. Dabei begann der erste Teilnehmer mit dem statischen System und benutzte anschließend das dynamische, der zweite Teilnehmer begann mit dem dynamischen und verwendete anschließend das statische. Dieses Vorgehen wurde bei allen Teilnehmern umgesetzt.

Bei der Auswertung der aufgenommenen Audiodaten wurde festgestellt, dass auf den aufgenommenen Nutzeräußerungen zum Teil Ausgaben des Dialogsystems vom Segmentierer des Audiostroms als Sprache erkannt und an den Spracherkenner weitergeleitet wurden - das führte zu Erkennungsfehlern, die das Dialogsystem gestört haben. Allerdings traten diese Artefakte nur bei zwei Probanden und dort auch nur bei wenigen Aussagen auf.

Des weiteren wurde das Mikrofon unterschiedlich von den Teilnehmern getragen: einige setzten das Headset auf den Kopf, andere hielten es in der Hand - deswegen sind einige Aufnahmen lauter und mit mehr Störgeräuschen behaftet.

Wie bereits in der Einleitung dieses Kapitels gesagt, geht es darum, den Unterschied zwischen beiden Dialogsystemen sowohl messbar als auch durch die Empfindung der Benutzer zu untersuchen. Messbar wird dies am besten durch den Vergleich der erreichten Dialogziele (Goals) und die durchschnittliche Länge bis zum Erreichen eines Goals. Wichtig ist dabei, die Anzahl der abgebrochenen Dialogziele zu beachten - ist diese hoch, gab es Probleme im Dialogsystem oder des Nutzers mit dem Dialogsystem. Durch die Probleme mit der Aufnahmequalität ist es wichtig, die Anzahl der korrekt erkannten Sätze (sentence correctness rate, SCR) in diese Betrachtung mit einzubeziehen.

In Tabelle 6.1 sind die Ergebnisse der ersten Aufgabe für das statische System angegeben, in Tabelle 6.2 für das dynamische. Die wichtigste Kennzahl durchschnittliche Länge bis zum Erreichen eines Dialogziels unterscheidet sich um 0.6 zwischen statischem(2.81%) und dynamischen(3.42%) System. Allerdings ist die SCR des dynamischen Systems(57.74%) um 31% schlechter als die des statischen Systems(76.10%). So wurden beim statischen System die Äußerungen von drei Benutzern komplett richtig erkannt, beim dynamischen nur von einem.
Die optimale Länge für das Erreichen des Dialogziels bei dieser Aufgabe beträgt zwei Schritte.

In Tabelle 6.3 (statisches System) und in Tabelle 6.4 (dynamisches System) findet sich ein ähnliches Bild. Vier Benutzer beim statischen und nur zwei Benutzer beim

Nutzer	Goals	erreicht	abgebrochen	Länge	SCR
1	1	1	0	2.00	100.00%
2	1	1	0	2.00	100.00%
3	2	1	1	2.00	75.00%
4	2	1	1	4.00	57.14%
5	1	1	0	5.00	60.00%
6	2	1	1	3.00	100.00%
7	3	2	1	2.50	50.00%
8	4	3	1	2.00	66.67%
AVG	2	1.375	0.625	2.81	76.10%

Tabelle 6.1: Ergebnisse Aufgabe 1 - statisches System

Nutzer	Goals	erreicht	abgebrochen	Länge	SCR
1	5	1	4	2.00	22.22%
2	1	1	0	2.00	100.00%
3	1	1	0	6.00	83.33%
4	4	2	2	3.00	37.50%
5	1	1	0	6.00	28.57%
6	5	4	1	3.33	68.75%
7	5	1	4	2.00	54.55%
8	2	2	0	3.00	66.67%
AVG	3	1.625	1.375	3.42	57.70%

Tabelle 6.2: Ergebnisse Aufgabe 1 - dynamisches System

Nutzer	Goals	erreicht	abgebrochen	Länge	SCR
1	3	1	2	3.00	71.43%
2	1	1	0	3.00	100.00%
3	1	1	0	4.00	50.00%
4	2	2	0	3.00	66.67%
5	1	1	0	3.00	100.00%
6	2	2	0	3.00	100.00%
7	1	1	0	3.00	100.00%
8	3	1	2	4.00	34.62%
AVG	1.75	1.25	0.5 3.25	77.84%	

Tabelle 6.3: Ergebnisse Aufgabe 2 - statisches System

Nutzer	Goals	erreicht	abgebrochen	Länge	SCR
1	3	2	1	3.50	60.00%
2	1	1	0	6.00	50.00%
3	1	1	0	3.00	100.00%
4	1	1	0	3.00	50.00%
5	2	1	1	4.00	33.33%
6	1	1	0	3.00	100.00%
7	1	1	0	4.00	75,00%
8	2	2	0	4.50	60.00%
AVG	1.5	1.25	0.25	3.88	66.04%

Tabelle 6.4: Ergebnisse Aufgabe 2 - dynamisches System

dynamischen System wurden korrekt erkannt. Der Unterschied in der Länge beträgt ebenfalls wieder 0.6 Schritte - bei einer optimalen Länge von drei Schritten. Der Unterschied in der Spracherkennung fällt hier mit ca. 18% weniger extrem aus; bei beiden Systemen gab es allerdings jeweils einen Probanden mit einer SCR-Rate von ca. 34%.
Insgesamt sind die Ergebnisse der 2. Aufgabe etwas besser als die der ersten. Das hängt sicherlich damit zusammen, dass die Nutzer bei der 2. Aufgabe garantiert schon von ersten Erfahrungen profitieren konnten, was bei der ersten Aufgabe nicht gleichmäßig gegeben war.

In Abbildung 6.1 und 6.2 sind die Werte grafisch veranschaulicht.

Nachdem die Probanden die Aufgaben ausgeführt hatten, wurden sie noch zu den Systemen befragt.

1. Wie gut funktionierte die Spracherkennung(ASR)? - Antwort von -2 bis +2
2. Wie gut hat das System die Aufgaben umgesetzt? - Antwort von -2 bis +2
3. Eignet sich das System für diese Aufgabe? - Antwort von -2 bis +2
4. Wie natürlich fanden Sie die Interaktion? - Antwort von -2 bis +2
5. Halten Sie das System für Intelligent? - Antwort von -2 bis +2
6. Haben Sie Unterschiede zwischen beiden Systemen festgestellt? Wenn ja - welche? - Antwort Ja / Nein; zusätzlich freie Antwort möglich
7. Was hat Ihnen gefallen? - freie Antwort
8. Was hat Ihnen nicht gefallen? - freie Antwort

Die Ergebnisse der Fragen 1 bis 5 sind in Abbildung 6.3 dargestellt. Die schlechten Ergebnisse in der Spracherkennung schlagen sich auch in der Nutzerbefragung nieder - allerdings auch die (wenigen) erfolgreichen. Die Umsetzung der Aufgabe wurde von allen mit gut bis sehr gut bewertet; die Eignung dieses Systems wurde - bedingt

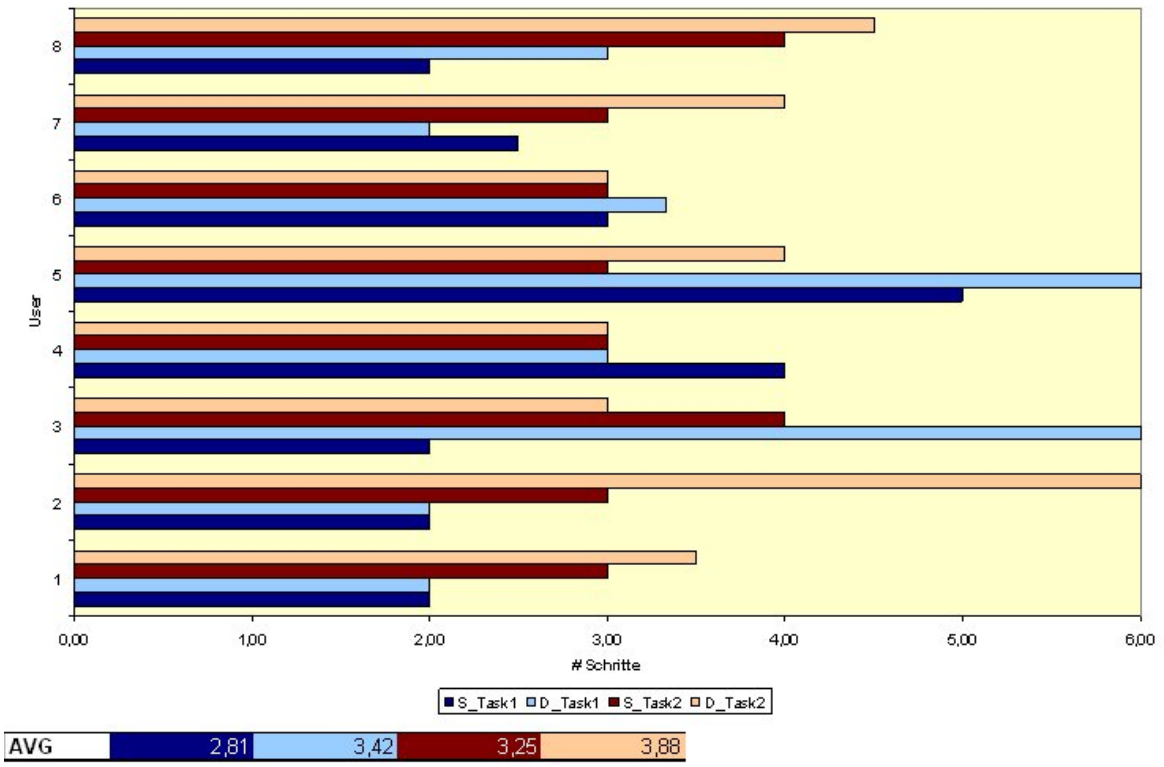

Abbildung 6.1: Durchschnittliche Anzahl Schritte bis zum Erreichen eines Dialogziels

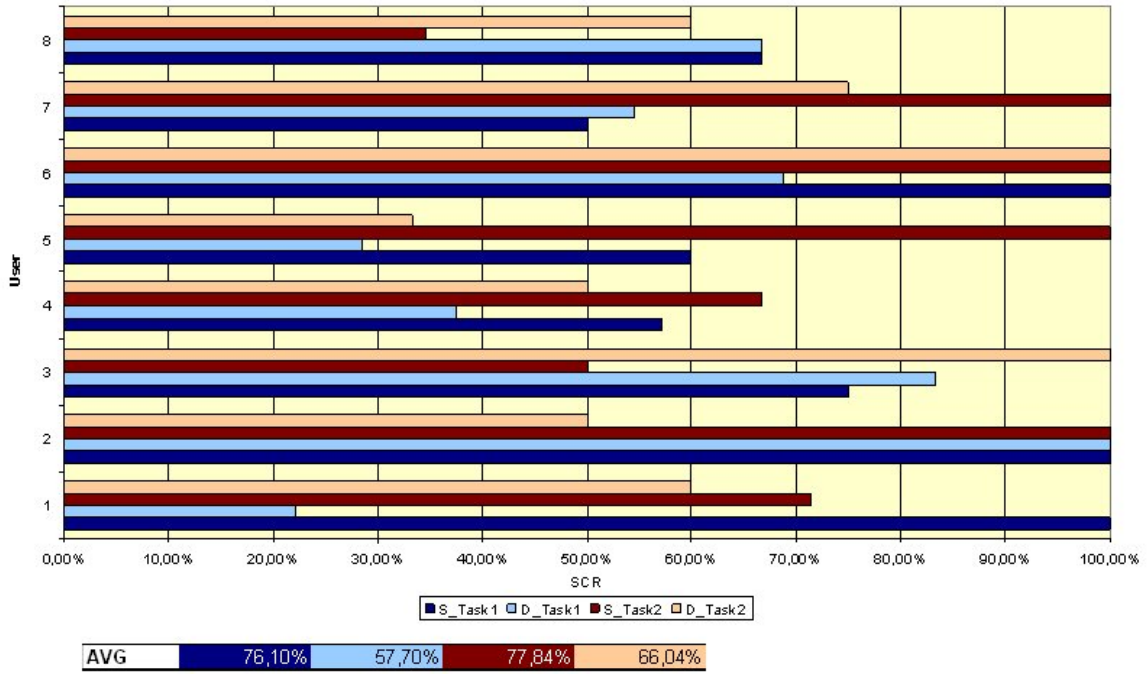

Abbildung 6.2: Anteil der vom Spracherkenner korrekt erkannten Sätze (SCR)

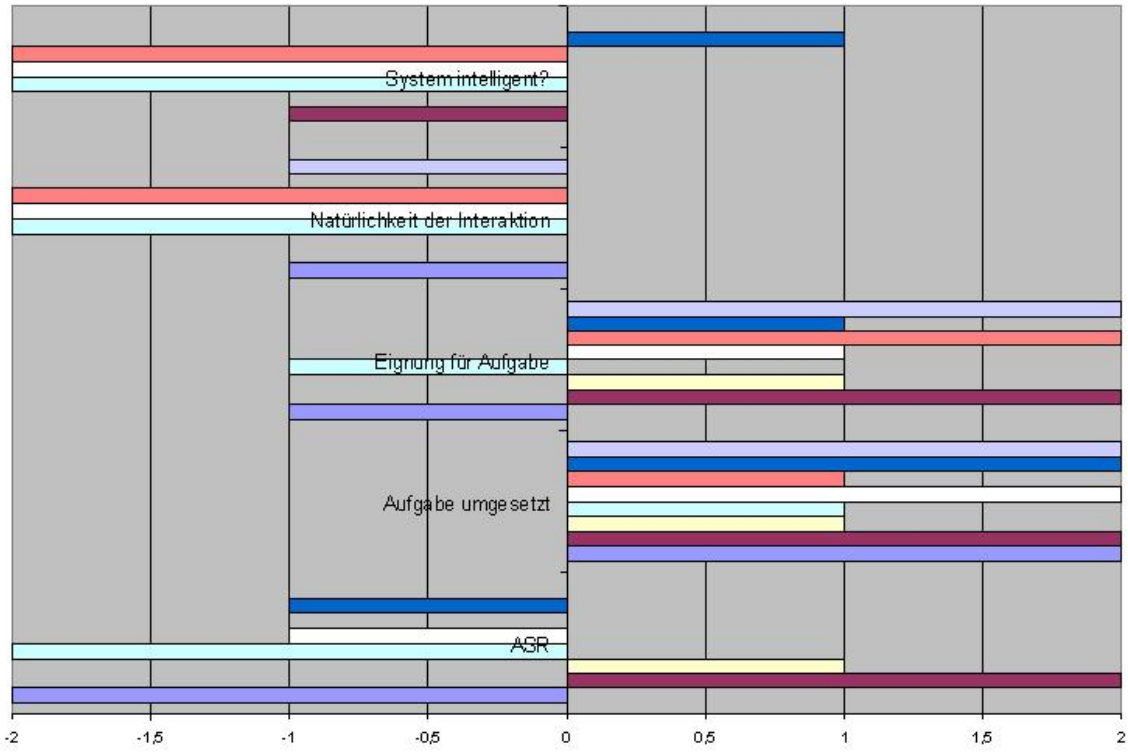

Abbildung 6.3: Ergebnisse der Nutzerbefragung

durch die Probleme - unterschiedlich eingeschätzt. Die Natürlichkeit wurde aufgrund des restriktiven Systems und Grammatiken weniger gut eingeschätzt, ähnlich verhält es sich mit der Systemintelligenz.

Alle Nutzer haben einen Unterschied zwischen beiden Systemen festgestellt - allerdings nur die unterschiedliche Länge der Antworten des Dialogsystems (dynamisch länger, statisch kürzer), welche sowohl positiv (mögliche Antworten wurden vorgelesen) als auch negativ (zu lang; Nutzer weiß, welche Aufgabe er ausführen und welche Werte er setzen möchte) bewertet wurde.
Gefallen hat den Nutzern die Ausführung der Aktionen; die Nachfrage bei Unverständnis; das der Proband von der Spracherkennungskomponente verstanden wurde; die Idee, frei sprechen zu können und darauf eine Antwort zu bekommen sowie die schnellere Umsetzung von Befehlen des statischen Systems.

Kritisiert wurde die Spracherkennung; die Sprachausgabe des Dialogsystems („zu unnatürlich“); das Ausführen der Aktionen durch Kommandos; die Eingeschränktheit der Grammatiken; die zu ausführlichen Antworten des dynamischen Systems und der daraus resultierenden langen Wartezeiten.

Dialogstrukturen

Die Größe der Dialogstrukuren unterscheidet sich stark: die Beschreibungsdatei des dynamischen Systems hat eine Länge von 240 Zeilen (ADL2-Datei) inklusive Kommentare - das statische hingegen nur 135 Zeilen. Die Grammatiken sind von der Größe ähnlich, beide besitzen zehn Einträge.

6.3 Analyse der Ergebnisse

Beim Betrachten der Ergebnisse fällt auf, dass die Dialogsysteme vom Benutzer nicht unterschieden werden können. Der einzige festgestellte Unterschied war die unterschiedliche Länge der Antworten des Dialogsystems. Da das Szenario und das Dialogsystem allerdings relativ eingeschränkt sind, waren diese Ergebnisse zu erwarten.

Die Unterschiede in der Spracherkennungsleistung können teilweise auf die unterschiedlichen grammatikalischen Strukturen zwischen den beiden Systemen zurückgeführt werden. Diese sind ähnlich, aber nicht komplett gleich. Außerdem ist die Anbindung des Dialogsystems an das Agentensystem und die Applikationskommunikation nicht optimal, deswegen ist das gebaute und integrierte System nicht so robust wie das statische.

Der Umfang der Dialogstruktur für das dynamische System scheint schlechter bzw. länger zu sein. Würde die Anbindung an das Agentensystem direkt in den Quellcode des Dialogsystems übernommen, würde die Länge und der Umfang der Dialogstrukturen sehr viel kleiner werden. Damit wäre dieser Vorteil des dynamischen Systems bereits bei diesem kleinen Szenario sichtbar.

Die Antwortzeit des Dialogsystems und die Ausführungszeit der Applikationsaktionen wird durch die Agentenkommunikation nicht merklich verlangsamt. Dies ist sehr gut, da damit der Nutzer nicht durch eine längere Wartezeit belastet wird.

7. Zusammenfassung und Ausblick

Die Ergebnisse aus Kapitel 6 stimmen zuversichtlich, dass dieser Ansatz ein Schritt in Richtung einer automatisierten, vereinheitlichten und vereinfachten Anbindung von Applikationen an Dialogsysteme darstellt. Er wird von den Benutzern angenommen und kann - in diesem Szenario - nicht von einem normalen Dialogsystem unterschieden werden.

Die erstellte Anwendung kann Dienste nach Wünschen des Benutzers dynamisch ausführen. Das dazu implementierte Protokoll erfragt zuerst bei den verfügbaren Applikationen, ob der Dienst verfügbar ist. Bei einer positiven Antwort werden die zur Ausführung benötigten Informationen bei der Applikation angefragt. Fehlende Informationen können aus dem Wissen des Dialogsystems beantwortet werden. Sind Informationen darin nicht gesetzt, wird eine Rückfrage an den Nutzer des Systems generiert und dessen Antwort verwendet. Nachdem alle benötigten Informationen vorhanden sind, wird die Aktion vom Dialogsystem gestartet und dem Nutzer eine Bestätigung der Ausführung angezeigt.
Es wurden weitere Möglichkeiten des Kommunikationsprotokolls untersucht, die hier beschriebene hat sich als am besten geeignete herausgestellt.

Die Benutzerstudie, die nur mit acht Testprobanden durchgeführt wurde, könnte - in Verbindung mit einer größeren Applikation und einem entsprechendem Szenario - mit einer größeren Anzahl von Nutzern noch einmal wiederholt werden. Dabei sollten die akustischen Probleme, die zu den relativ schlechten Ergebnissen des Spracherkenners geführt hatte, vermieden werden.
Die Ergebnisse der Benutzerstudie sind nur beschränkt aussagekräftig, es fehlt eine Untersuchung größerer Anwendungen und damit verbunden eines umfangreicheren Dialogsystems. Dort sollte dieser Ansatz seine Vorteil sehr deutlich ausspielen und einem normalen Dialogsystem klar überlegen sein - zumindest im Hinblick auf den Aufwand, mit dem Anwendungen angebunden und verwendet werden.

Im Rahmen dieser Arbeit wurde die Dienstgüte nicht untersucht. Diese ist besonders dann wichtig, wenn mehrere Anwendungen ähnliche Dienste anbieten. Möglicherweise besitzen diese Dienste unterschiedliche Merkmale und damit verbunden eine besse-

re oder schlechtere Eignung für die Ausführung der Aufgabe aus Sicht des Benutzers.

Die Verwendung mehrerer Applikationen wurde in dieser Arbeit nicht explizit untersucht. Die verwendete Kommunikationsontologie und die Datenstrukturen wurden entworfen, um mit mehrere Agenten kommunizieren zu können.

Ein weiterer Punkt, der einer umfassenderen Untersuchung bedarf, ist die Erweiterung der Ontologie, um Aktionen direkt darin beschreiben zu können. Dies ist nötig, um das Dialogsystem bestmöglich für die dynamische Anbindung von Applikationen anpassen zu können. Interessant ist dabei insbesondere die Art, wie fehlende Informationen zur Ausführung von Diensten behandelt werden. Die im Rahmen dieser Arbeit verwendeten Ansätze - explizite Regeln im Dialogsystem sowie eine Menge von festgelegten Werten, aus denen der Nutzer eins auswählt - sind eine Möglichkeit.

A. Smartroom Anwendung

Die Smartroom-Anwendung wird im Rahmen des EU-Forschungsprojektes CHIL entwickelt. Sie ermöglicht die Steuerung der in einem Raum vorhandenen elektronischen Geräte durch Software. Im Smartroom des Interactive Systems Labs an der Universität Karlsruhe können dadurch ein beweglicher Videoprojektor(Abbildung A.1), ein beweglicher Lautsprecher („Targeted Audio“, Prototyp entwickelt von Daimler-Chrysler; siehe [DFBM+05]; Abbildung A.2), eine Lichtsteuerung über das X10-Protokoll (über das Stromnetz) sowie normale Videoprojektoren und Lautsprecher kontrolliert werden.

Bei der Entwicklung wurde Wert darauf gelegt, die Architektur offen und leicht erweiterbar zu gestalten. So ist die Ansteuerung der steuerbaren Geräte unabhängig von der Funktionalität der Geräte. Des weiteren wird eine auf RMI basierende Kommunikation genutzt, um die Komponenten auf unterschiedlichen Rechnern ausführen zu können. Diese Komponenten werden beim Starten der Software auf den jeweiligen Rechnern abhängig von der Konfiguration automatisch geladen und gestartet. Die Definition von Aktionen, die mit diesen Komponenten ausgeführt werden können, geschieht in XML-Dateien, welche zur Laufzeit eingelesen und verarbeitet werden. Die Aktionen können mittels einer Java-Schnittstelle ausgeführt oder in einer grafi-

Abbildung A.1: Steuerbare Projektor-Kamera - Kombination

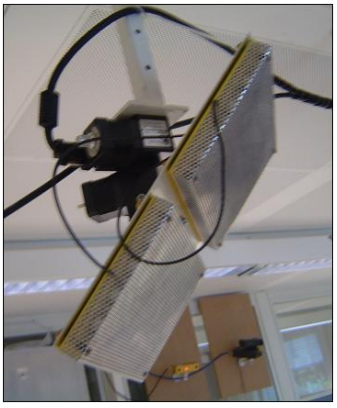

Abbildung A.2: Targeted Audio

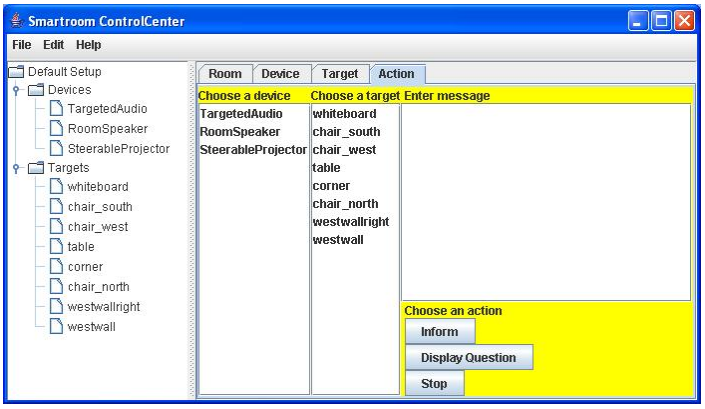

Abbildung A.3: Grafische Oberfläche der Smartroom Applikation

schen Oberfläche(Abbildung A.3) ausgewählt und aktiviert werden. Allerdings sind noch nicht alle Aktionen, die mittels Java-Schnittstelle ausgeführt werden können, in der Oberfläche anzeig- und ausführbar.

A.1 Architektur

In Abbildung A.4 ist ein grober Überblick der Architektur der Smartroom-Anwendung dargestellt. Die Smartroom-Anwendung ist modular aus Software-Komponenten aufgebaut, hier der Steuerung von Geräten und deren entsprechender Anzeige- und Ausgabekomponenten. Diese kommunizieren mittels Java RMI (*Remote Method Invocation*, einem direkt in Java eingebauten entfernten Methodenaufruf). Komponenten melden sich an einem zentralen Registrierungsserver (*RMI Registry Server*) an und können dort von dem SmartroomManager gefunden werden. Dieser SmartroomManager implementiert die Schnittstelle für die Steuerung der Komponenten.
Durch die Kommunikation mittels Java-RMI können die Komponenten auf verschiedenen Rechnern in einem Netzwerk gestartet werden, die Steuerung durch den SmartroomManager kann sogar über das Internet erfolgen.

A.2 Komponenten

Im Rahmen dieser Architektur werden Komponenten als Software-Module bezeichnet, die bestimmte Aktionen zur Verfügung stellen; sie werden zur Laufzeit mit Werten aus den Konfigurationsdateien initialisiert und mittels RMI beim Gesamtsystem

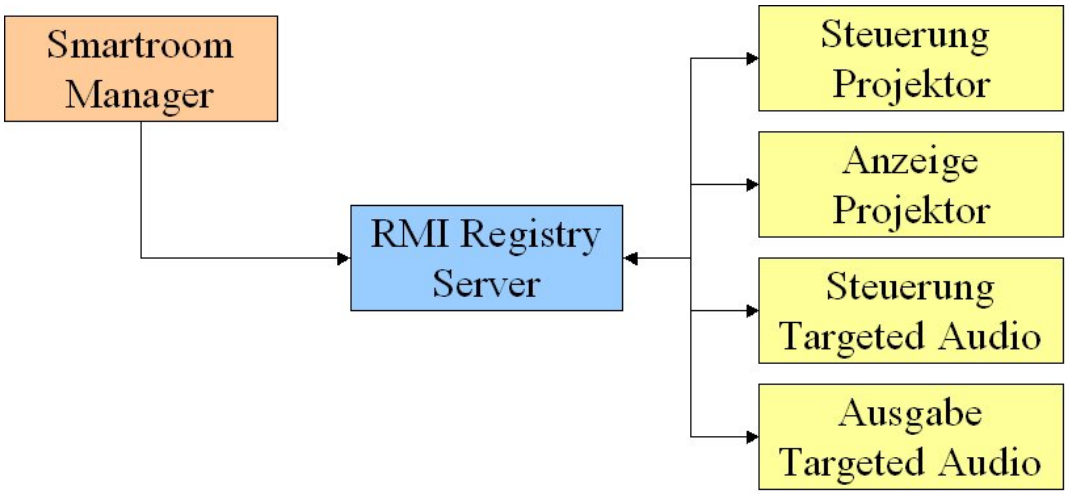

Abbildung A.4: Übersicht Architektur der Smartroom-Anwendung

registriert. Die Definition muss unterhalb von */Smartroom/Components* (XPath-Notation) erfolgen. Im Listing A.1 wird eine Komponente definiert, die auf dem PC mit dem Hostname *i13pc240* läuft (und nur dort gestartet wird) und vom Typ *de.uka.ira.isl.chil.smartroom.component.tts.TTSComponent* ist. Beim Erzeugen einer Komponente wird von dieser Klasse eine neue Instanz erzeugt und die Parameter, die innerhalb des *init*-Tags definiert werden, als String übergeben.

Listing A.1: Definition eines vollständigen Move

```
<component>
  <host>i13pc240</host>
  <type>
    de.uka.ira.isl.chil.smartroom.component.tts.TTSComponent
  </type>
  <init>
    <name>RoomSpeaker</name>
    <ttsEngine>
      /home/smartroom/programs/swift/bin/swift
    </ttsEngine>
  </init>
</component>
```

Des weiteren können mittels PTU (Pan-Tilt-Unit; steuerbare Geräte, die durch zwei kleine Elektromotoren bewegt werden) der Video-Projektor und das Targeted Audio aus dem CHIL Smartroom an verschiedene Positionen im Raum bewegt werden. Mittels des X10-Protokolls lassen sich verschiedene Geräte an- und ausschalten, aber auch das Licht dimmen. Auf die Definition und Implementierung dieser Module wird nicht eingegangen, da dies hier nicht relevant ist.

A.3 Aktionen

Aktionen, intern als *Move* bezeichnet, beschreiben eine Abfolge von Interaktionen mit Komponenten, die zur Laufzeit eingelesen und nach erfolgreicher Verarbeitung ausgeführt werden können. Die Beschreibung erfolgt in XML. Im Nachfolgenden wird der Name *Move* verwendet, um eine Anwendungsaktion zu bezeichnen. Ein Move besteht aus Unteraktionen, die mit *Action* bezeichnet werden. Innerhalb einer Action werden die eigentlichen auszuführenden Funktionen definiert: die Funktion auf

einem Objekt, die mittels Java Reflection ausgeführt werden soll, sowie gegebenfalls benötigte Parameter. Definiert werden muss ein Move unterhalb von */Smartroom/-Moves* (XPath-Syntax).

Listing A.2: Definition eines vollständig spezifizierten Move

```
<move>
  <name>SwitchOffLittleLamp</name>
  <action>
    <device>SmartroomX10</device>
    <methodName>switchOff</methodName>
    <parameter>
      <class>String</class>
      <name>name</name>
      <description>
        name of the device to switch off
      </description>
      <value>little lamp</value>
    </parameter>
  </action>
</move>
```

Ein einfacher Move wird im Listing A.2 definiert. Der Name lautet *SwitchOffLittleLamp* - dieser Name referenziert diesen Move global, deswegen muss er innerhalb aller definierten Aktionen eindeutig sein. In diesem Move wird eine Action definiert, die auf der Komponente („device“ genannt) SmartroomX10 die Methode („methodName“) switchOff aufruft. Der dafür benötigte Parameter trägt den Namen „name“ und ist vom Typ String. Die dazugehörige Beschreibung („description“) lautet „name of the device to switch off“. Dieser Parameter ist bereits vorbelegt mit dem Wert „little lamp“. Damit ist dieser Move vollständig definiert und kann ohne Rückfragen ausgeführt werden.

Listing A.3: Definition eines unvollständigen Move

```
<move>
  <name>InformSmartroomTTS</name>
  <action>
    <device>RoomSpeaker</device>
    <methodName>display</methodName>
    <parameter>
      <class>String</class>
      <description>text to speak</description>
      <name>text</name>
      <value></value>
    </parameter>
  </action>
</move>
```

Im Gegensatz dazu kann der Move InformSmartroomTTS (Listing A.3) nicht sofort ausgeführt werden, der Parameter „text“ ist nicht belegt und muss vor der Ausführung mit einem Wert belegt werden.

A.4 Weitere Besonderheiten

Erweiterbarkeit

Die verwendete Architektur erlaubt eine einfache und schnelle Erweiterung durch neue Funktionen und Komponenten. Bestehende Komponenten können weitere Funk-

tionen anbieten, die - nachdem diese programmiert wurden - nur noch in den XML-Beschreibungsdateien definiert werden müssen. Anschließend stehen sie zur Verfügung und können verwendet werden.
Neue Komponenten können - nachdem sie implementiert wurden - einfach durch einen Eintrag in der entsprechenden Konfigurationsdatei verwendet werden.

Starten des Systems

Zum Starten der Komponenten muss nur eine Anwendung gestartet werden, die automatisch die Komponenten des jeweiligen Rechners erzeugt, initialisiert und beim RMI Registry-Server anmeldet. Das wird durch die Konfiguration in den XML-Konfigurationsdateien gesteuert.
Dadurch kann das System - auch auf mehreren Rechnern verteilt - sehr schnell gestartet werden.

A.5 Ausblick

Das Gesamtsystem funktioniert, allerdings gibt es viele Möglichkeiten, das System zu verbessern. Die einzelnen Komponenten erfüllen ihre Aufgabe, allerdings könnte noch weitere Funktionalität hinzugefügt werden.

Die Steuerung der beweglichen Ausgabegeräte wird insbesondere dann interessant, wenn die Ausgabeziele sich zur Laufzeit bewegen. So ist es sinnvoll, das Targeted Audio für eine sich im Raum bewegende Person nachzuführen. Da es aber noch keine Software gibt, welche die Bewegung von Personen im Raum mit ausreichender Genauigkeit angeben kann, konnte dies bisher noch nicht getestet werden. Durch die Steuerung über die grafische Oberfläche der Smartroom-Anwendung lässt sich dies allerdings - wenn auch nur manuell - durchführen.

B. Dialoganbindung

B.1 Goal SelectAction

Listing B.1: Implementierung des Goals SelectAction

```
goal SelectAction {
precondition:
 [ act_selectaction
  generic:ARG    [ obj_action
   tapas_ac:NAME   [ base:string ]
  ]
 ]
->
bindings:
 internal://dialogue/say "action",
  $sem[generic:ARG|NAME], "selected";
};
```

B.2 Move GetActionDetails

Listing B.2: Implementierung des Move GetActionDetails

```
move GetActionDetails on variable Intention changed to finalized {
goal: ( SelectAction = finalized ),
script: %{
 from tapas.services.jadeconnector import JADEConnector
 from tapas.services.jadeconnector import AgentCommunicationHelper
 action = AgentCommunicationHelper.getVanillaAction()
 action.setName(sem.getTypeValue("generic:ARG|NAME"))
 params = JADEConnector.sendJADEMessage("queryref",
  "MissingParameters", [sem.getTypeValue("generic:ARG|NAME")])
 result = []
 for param in params:
  if (param.getName() != None) and (param.getValue() == None):
   action.addParameter(param)
 AgentCommunicationHelper.setCurrentAction(action)
 constraint = 1
}%
->
```

```
bindings:
};
```

B.3 Move HandleMissingInformation_Restricted

Listing B.3: Implementierung des Move HandleMissingInformation_Restricted

```
move HandleMissingInformation_Restricted {
script: %{
 constraint = 0
 from tapas.services.jadeconnector import AgentCommunicationHelper
 result = AgentCommunicationHelper.getNextMissingParameter()
 if (result != None) and (result.getFixedValue().size() > 0):
  constraint = 1
}%
->
bindings:
 internal://dialogue/say "the␣information␣", %{
  from tapas.services.jadeconnector import AgentCommunicationHelper
  result=AgentCommunicationHelper.getNextMissingParameterDescription()
 }% , "is␣␣missing.";
 internal://dialogue/say %{
  from tapas.services.jadeconnector import AgentCommunicationHelper
  values=AgentCommunicationHelper.getNextMissingParameterFixedValues()
  if len(values) > 0:
          result = "possible␣values␣are␣"
          i = 0
          for value in values:
    i = i + 1
    if i > (len(values) -1):
     result = result + value # remove last "or"
    else:
     result = result + value + "␣or␣"
 }%;
 internal://dialogue/target [PARAMETER|VALUE],
  HandleMissingInformation_GotParameter;
};
```

B.4 Move HandleMissingInformation_Open

Listing B.4: Implementierung des Move HandleMissingInformation_Open

```
move HandleMissingInformation_Open {
script: %{
 constraint = 0
 from tapas.services.jadeconnector import AgentCommunicationHelper
 result = AgentCommunicationHelper.getNextMissingParameter()
 if (result != None) and (result.getFixedValue().size() == 0):
        constraint = 1
 }%
->
bindings:
 internal://dialogue/say "the␣information␣", %{
  from tapas.services.jadeconnector import AgentCommunicationHelper
  result=AgentCommunicationHelper.getNextMissingParameterDescription()
 }% , "is␣␣missing.";
 internal://dialogue/say "please␣speak␣now.";
 internal://dialogue/target [PARAMETER|VALUE],
```

```
  HandleMissingInformation_GotParameter;
};
```

B.5 Goal HandleMissingInformation_GotParameter

Listing B.5: Implementierung des Goals HandleMissingInformation-GotParameter

```
goal HandleMissingInformation_GotParameter {
precondition:
 [ act_setParameter
  PARAMETER [ obj_information
   VALUE        [ base:string ]
  ]
 ]
->
bindings:
 internal://dialogue/say "the value",
  $sem[PARAMETER|VALUE], "has been stored";
};
```

B.6 Move ExecuteAction

Listing B.6: Implementierung des Move ExecuteAction

```
move ExecuteAction on variable Intention changed to finalized {
script: %{
 constraint = 1 # default: do not invoke this move
 from tapas.services.jadeconnector import JADEConnector
 from tapas.services.jadeconnector import AgentCommunicationHelper
 params = AgentCommunicationHelper.getParameter()
 result = []
 for param in params:
 if (param.getName() != None) and (param.getValue() == None):
  constraint = 0
 else:
  result.append(param)
  constraint = 1

 if constraint == 1:
 res = [AgentCommunicationHelper.getAction().getName()]
 res.extend(result)
 success = JADEConnector.sendJADEMessage("request","InvokeAction",res)
 AgentCommunicationHelper.reset()
}%
->
bindings:
 internal://dialogue/say "the selected action has been executed.";
};
```

Literatur

[ABDF+01] J. Allen, D. Byron, M. Dzikovska, G. Ferguson und L. Galescu. Towards Conversational Human-Computer Interaction. *AI Magazine*, 2001.

[AlBF02] J. Allen, N. Blaylock und G. Ferguson. A Problem Solving Model for Collaborative Agents. In *First International Joint Conference on Autonomous Agents and Multiagent Systems*, Bologna, Italy, 2002.

[CaCa04] Giovanni Caire und David Cabanillas. JADE Tutorial - Application-defined content languages and ontologies, 2004.

[Dene02] Matthias Denecke. Rapid Prototyping for Spoken Dialogue Systems. In *19th international conference on Computational linguistics*, Taipei, Taiwan, 2002.

[DFBM+05] Maria Danninger, Gopi Flaherty, Keni Bernardin, Robert Malkin, Rainer Stiefelhagen und Alex Waibel. The Connector - Facilitating Context-aware Communication. 2nd Joint Workshop on Multimodal Interaction and Related Machine Learning Algorithms, Edinburgh, UK, 2005.

[fInt02] Foundation for Intelligent Physical Agents. FIPA Agent Management Specification, 2002.

[fInt03a] Foundation for Intelligent Physical Agents. FIPA Query Interaction Protocol Specification, 2003.

[fInt03b] Foundation for Intelligent Physical Agents. FIPA Request Interaction Protocol Specification, 2003.

[HoGi05] Hartwig Holzapfel und Petra Gieselmann. Tapas Tutorial 1.2. Technischer Bericht, Interactive Systems Labs, 2005.

[NoMc01] N. Noy und D. L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Technischer Bericht KSL-01-05, Knowledge Systems Laboratory, Stanford University, 2001.

[PaPS00] Maurizio Panti, Loris Penserini und Luca Spalazzi. A critical discussion about an agent platform based on FIPA specification. In *Sistemi Evolui per Basi di Dati*, 2000, S. 345–356.

[PoRa04] David Portabella und Martin Rajman. A Dialogue-based Grounding mechanism and new Service Description Features for adapting Semantic (Web) Services to Personal Assistants. Technischer Bericht IC/2004/85, Ecole Polytechnique Fédérale de Lausanne, 2004.

[TuHa01] Markku Turunen und Jaakko Hakulinen. Agent-Based Adaptive Interaction and Dialogue Management Architecture for Speech Applications. *Lecture Notes in Computer Science* Band 2166, 2001, S. 357–364.

[WoJe95] Michael J. Wooldridge und Nicholas R. Jennings. Agent Theories, Architectures, and Languages: A Survey. In Michael J. Wooldridge und Nicholas R. Jennings (Hrsg.), *Workshop on Agent Theories, Architectures & Languages (ECAI'94)*, Band 890 der *Lecture Notes in Artificial Intelligence*, Amsterdam, The Netherlands, Januar 1995. Springer-Verlag, S. 1–22.

Index