Universität Karlsruhe (TH)

Entwicklung eines planbasierten Dialogmodells für Informationssysteme

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

von

Dennis Nienhüser, Ignaz Rutter, Stefan Ziesemer

Betreuer:
Prof. Dr. Alexander Waibel
Dipl.-Inform. Hartwig Holzapfel

Tag der Abgabe: 19. September 2005

Institut für Theoretische Informatik (ITI)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Computer haben innerhalb weniger Jahrzehnte die Welt verändert und ein neues Zeitalter eingeläutet. Heutige Modelle unterscheiden sich in Aussehen, Größe, Leistungsfähigkeit, Flexibilität und Mobilität stark von ihren Vorgängern. Die Kommunikation zwischen Mensch und Computer dagegen hat sich nur unwesentlich verändert. Bereits 1941 stellte Konrad Zuse mit der Z3 eine Rechenmaschine vor, die über eine Tastatur verfügte, und noch heute sind Tastatur und Maus mit Abstand das meistgenutzte Medium zur Dateneingabe.

Das ist umso erstaunlicher, wenn man bedenkt, dass diese Art der Kommunikation mit dem Computer nicht nur wenig intuitiv, sondern auch langsam ist. Ein Dialogsystem, das menschenähnliche Kommunikation über Sprache mit einem Computer erlaubt, hat großes Innovationspotential. Es würde Menschen, die bisher aufgrund körperlicher und geistiger Beeinträchtigungen nicht oder nur sehr eingeschränkt mit Computern umgehen können, eine nicht zu unterschätzende Hilfe sein. Auf ähnliche Weise würde es Menschen, die Computer tagtäglich in Beruf wie Freizeit einsetzen, einen schnelleren und bequemerem Umgang erlauben.

Bereits 1966 entwickelte Joseph Weizenbaum ein Programm namens ELIZA [Weiz66], mit dessen Hilfe die natürlichsprachliche Kommunikation zwischen Mensch und Maschine untersucht werden sollte. Obwohl ELIZA mit wenigen, einfachen Regeln auskam, war die Akzeptanz der Benutzer erstaunlich groß. Knapp vierzig Jahre später gibt es dennoch kein Computerprogramm, das den sogenannten Turing-Test[1] besteht.

Ein Problem vieler Dialogsysteme ist, dass mit wachsendem Dialogwissen die Anzahl der Regeln zur Identifikation von Benutzerwunsch und geeigneter Reaktion schwer handhabbar wird. Wünschenswert ist, allgemeine Regeln, wie die Reduktion einer Aufzählungsliste, auf wenige Elemente nur genau einmal definieren zu müssen, um sie dann an passender Stelle immer wieder verwenden zu können. Aus diesen Überlegungen ergibt sich die Frage, ob es möglich ist, einem Dialogsystem wenige kleine und genau definierte Bausteine zur Verfügung zu stellen, mit deren Hilfe komplexere Aufgaben gelöst werden können. Dies führt uns zu Pogy, einem System, das Planungstechniken verwendet. Als Vorteil dieses Systems erwarten wir einfachere Wartung und gute Erweiterbarkeit.

---

[1] Test, der für ein Computerprogramm als bestanden gilt, wenn ein Mensch nicht in der Lage ist, nach fünfminütiger Befragung herauszufinden, ob es sich bei seinem Gesprächspartner um Mensch oder Maschine handelt.

1.1. Problematik

Menschen kommunizieren scheinbar mühelos per Sprache. Beim Lernen einer Fremdsprache bekommt man ein erstes Gefühl dafür, dass diese Art der Kommunikation kein einfacher Prozess ist. Die Zerlegung der sprachlichen Verständigung in Teilprozesse aus Sicht eines Dialogsystems zeigt weitere Aspekte dessen Komplexität auf.

Zunächst einmal muss ein Sprachsignal erkannt, aufgenommen und in Text umgewandelt werden. An der maschinellen Spracherkennung wird seit Jahrzehnten geforscht, doch trotz beträchtlicher Fortschritte ist man von einer menschähnlichen Erkennungsleistung noch weit entfernt.

Ist diese Hürde genommen, muss die Bedeutung des Gesagten extrahiert werden. Das Gebiet des *Natural Language Understanding* (natürliches Sprachverstehen, NLU) ist ebenfalls ein eigenständiges Forschungsgebiet.

Noch komplexer wird es, wenn man nicht nur menschenähnliche Sprachkommunikation, sondern menschenähnliche Kommunikation ermöglichen will, also neben der Sprache andere Modalitäten wie Gestik und Mimik erlaubt. Neben der dazu notwendigen Sensorik und Verarbeitung der zusätzlichen Daten muss man sich dann auch darum kümmern, die unterschiedlichen Daten richtig zu fusionieren.

Ein weiteres Problem ist, Bedeutung und Intention des Gesagten zu verstehen. Ein erster Schritt ist, die Äußerung richtig in den Gesprächskontext einzuordnen. Dazu ist es nötig, Informationen mit Hilfe von Welt- und Domänenwissen in Beziehung zueinander zu setzen. Ein Kernproblem beim Entwurf eines Dialogsystems ist folglich die Repräsentation und Modellierung von Wissen. Im Anschluss muss man die Intention des Gegenübers analysieren und eine Reaktion darauf generieren. Dies ist die Aufgabe der Dialogstrategie. Zu guter Letzt muss die textuelle Antwort in ein Sprachsignal umgewandelt werden. Dies wird von sogenannten Text-To-Speech Systemen übernommen.

Neben der grundsätzlichen Funktionalität ist die Benutzbarkeit (usability) des Systems ein wichtiges Qualitätsmerkmal. Dazu gehört z. B. die Anzahl der benötigten Interaktionsschritte mit dem System bis zum Erreichen eines Ziels. Wichtig ist auch, dass das System kooperativ antwortet: Anstatt beispielsweise eine Frage mit einem simplem „Ja“ oder „Nein“ zu beantworten, sollte zusätzlich Auskunft über aufgetretene Probleme gegeben oder Alternativen vorgeschlagen werden. Um Missverständnissen vorzubeugen, sollte die Benutzeräußerung paraphrasiert[2] werden. All dies macht die Kommunikation natürlicher und damit für Menschen angenehmer.

1.2. Teamarbeit

Diese Arbeit wurde in einem Dreierteam erstellt. Die schriftliche Ausarbeitung wurde ebenso wie Entwurf und Implementierung des Systems grob in die drei

---

[2] Wiederholung des Inhalts einer Aussage mit anderem Wortlaut

Verantwortungsbereiche *Diskurs/Kontext* (Dennis Nienhüser), *Planung* (Ignaz Rutter) und *Wissensmodellierung und -repräsentation* (Stefan Ziesemer) aufgeteilt.

1.3. Gliederung der Arbeit

Im weiteren Verlauf dieser Arbeit werden in Kapitel 2 wichtige Grundlagen besprochen, die für Verständnis von Aufbau und Funktion von Dialogsystemen wichtig sind. Im Anschluss daran geben wir einen Überblick über verwandte Arbeiten und Systeme. Kapitel 4 befasst sich mit der Theorie von *planbasierten Dialogsystemen*. In Kapitel 5 wird ausführlich auf die Implementierung von *Pogy* eingegangen, gefolgt von Kapitel 6, in dem die *Ergebnisse der Evaluation* vorgestellt werden. Auf deren Basis bewerten wir in Kapitel 7 das vorliegende System und diskutieren mögliche Erweiterungen. Die Arbeit schließt mit einer *Zusammenfassung* in Kapitel 8.

2. Grundlagen

Im Folgenden sollen Begriffe, Algorithmen und Werkzeuge erläutert werden, die in dieser Arbeit von Bedeutung sind. Dabei haben wir uns bemüht, stets die deutschen Fachbegriffe zu verwenden. Nur wenn der deutsche Fachbegriff sehr ungebräuchlich ist, haben wir zu Gunsten einer besseren Lesbarkeit auf die Übersetzung verzichtet.

2.1. Dialogmanager Tapas

Tapas

Diese Arbeit benutzt den Dialogmanager *Tapas* [Holz05] und erweitert ihn um eine *planbasierte Dialogstrategie.* Tapas stellt ein Framework für verschiedene Ein- und Ausgabemodalitäten zur Verfügung. So kann z. B. über eine gewöhnliche Textkonsole oder über einen Spracherkenner mit dem System kommuniziert werden. Dabei übernimmt Tapas die Aufgabe, einen natürlichsprachlichen Text in eine *Typed Feature Structure* TFS (siehe 2.5) zu konvertieren. Dies geschieht mit speziellen Grammatiken, die für jede Sprache geschrieben werden müssen. Auch die Systemausgabe wird über Tapas realisiert. So können über verschiedene Plugins unterschiedliche Text-To-Speech Systeme (siehe 2.3) angesteuert oder eine Textausgabe auf dem Bildschirm ausgegeben werden.

2.2. Spracherkenner

Janus

Perplexität

Tapas unterstützt verschiedene Spracherkenner. Beim Einsatz auf dem humanoiden Roboter benutzen wir als Spracherkenner *Janus* [FGHK$^+$97] mit Ibis Decoder [SMFW01] und der Option, dessen Sprachmodell durch kontextfreie Grammatiken zu beschreiben. Die Grammatikregeln können in Abhängigkeit vom Kontext durch Tapas gewichtet werden [FuHW04]. Durch Gewichtung der Pogy bekannten Worte in Janus Sprachmodell sinkt die *Perplexität*[1], was wiederum eine höhere Erkennungsleistung zur Folge hat.

2.3. Sprachausgabe (Text-To-Speech)

Text-To-Speech

Ein *Text-To-Speech* System wandelt geschriebenen Text in ein akustisches Sprachsignal um. Es gibt einige frei verfügbare sowie sehr ausgereifte, kommerzielle Systeme. Wir benutzen Festival[2] sowie die Microsoft Speech API.

---

[1] Komplexitätsmaß, das man im Zusammenhang mit Spracherkennern als durchschnittliche Anzahl zur Auswahl stehender Worte zu einem Zeitpunkt betrachten kann.

[2] URL: `http://www.cstr.ed.ac.uk/`

2.4. Diskurs

Der Diskurs repräsentiert den aktuellen Gesprächszustand. Dazu gehört eine semantische Repräsentation des bisherigen Dialogverlaufs sowie verschiedene Zugriffsmöglichkeiten auf Kontextinformationen. Es gibt Mechanismen zur Einordnung neuer Äußerungen in den Dialog.

2.4.1. Kontext als Teilmenge der Wissensbasis

Wissen über Objekte der modellierten Welt und deren Beziehungen ist in der *Wissensbasis* (siehe 2.7) abgelegt. Zu einem beliebigen Zeitpunkt in einem Dialog ist immer nur eine echte Teilmenge dieses Wissens relevant. Kontext und Fokus repräsentieren diese Teilmenge auf unterschiedliche Art und Weise. *Dey und Abowd* [DeAb99] definieren Kontext als „any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves“. Sie unterscheiden zwischen Kontext 1. und 2. Art.

2.4.1.1. Kontext 1. Art

Kontext 1. Art

Dey und Abowd ordnen dem *Kontext 1. Art* Art Ort, Zeit, Identität und Aktivität zu, womit die Fragen Wo, Wann, Wer und Was beantwortet werden können.

2.4.1.2. Kontext 2. Art

Kontext 2. Art

Die Elemente des Kontexts 1. Art erlauben Zugriff auf weitere Kontextinformationen. Beispielsweise kann man über den Namen eines Mitarbeiters sein Büro ermitteln. Die Erweiterung des Kontexts um alle auf diese Weise zugänglichen Eigenschaften wird *Kontext 2. Art* genannt.

2.4.2. Fokus

Fokus

Der von Grosz [Gros86] vorgestellte *Fokus* enthält die zuletzt referenzierten Objekte, die somit „im Mittelpunkt“ stehen sowie weitere Objekte, auf die die Aufmerksamkeit gerichtet ist, ohne explizit referenziert zu werden. Wird ein Objekt längere Zeit nicht genannt, verschwindet es aus dem Fokus.

2.5. Typed Feature Structures (TFS)

Definition 2.1 (TFS)

TFS

Eine Typed Feature Structure TFS über Type und Feature nach Carpenter

[Carp92] ist ein Tupel $F = (Q, q, \phi, \delta)$ *mit einer Menge von Knoten* Q *und dem Wurzelknoten* $q \in Q$*. Weiter ist*

$$\begin{aligned}
\Phi : Q &\rightarrow Type && \textit{Funktion zur Typisierung der Knoten}\\
\delta : Feature \times Q &\rightarrow Q && \textit{weist einem Feature einen Type zu}\\
\delta(\varepsilon, q) &= q && \textit{mit dem leeren Pfad } \varepsilon\\
\delta(f\pi, q) &= \delta(\pi, \delta(f, q)) && \textit{mit } f, \pi, q \in Feature
\end{aligned}$$

TFS werden zur Repräsentation der Semantik eines Dialogs benutzt. In Abbildung 2.1 ist ein Beispiel für eine TFS dargestellt.

```
[ act_ask_food
    LINE [ prp_line
        NUMBER [ 3 ] ]
    LOCATION [ obj_location
        NAME [ "mensa" ] ]
]
```

Abbildung 2.1.: TFS für die Anfrage „Which food does the mensa serve on line three?“

2.6. Planung

Kommunikation ist für gewöhnlich ein zielorientierter Prozess. Beide Kommunikationspartner haben Ziele (zum Beispiel den Austausch von Informationen) und versuchen diese zu verfolgen. Ziel der Planung ist es, eine Abfolge von Aktionen zu generieren, die den Anfangszustand in einen Zielzustand überführt. Diese Aktionsabfolge oder Plan entspricht dabei der *intention* im BDI Agentenmodell und das Ziel dem *desire*. Beliefs werden durch Weltwissen und Kontext repräsentiert [WoJe95].

Ein Zielzustand muss dabei nicht immer das Ziel des Benutzers erfüllen. Eine Auskunft darüber, dass die gewünschte Information nicht verfügbar ist, erfüllt zwar nicht das Benutzerziel, führt aber dennoch in einen Zielzustand.

Das Finden von Aktionsabfolgen, die einen gegebenen Anfangszustand in einen Zielzustand überführen, ist eine typische Planungaufgabe. Mit Planungstechniken soll nun ein Abfolgeplan von Aktionen erstellt werden, deren Anwendung die Ausgangssituation in die Zielsituation überführt. Diese Aktionen werden als Operatoren mit Vorbedingungen und Effekten modelliert. Um die Vorbedingungen und Effekte eines Operators formulieren zu können, bedient man sich einer Beschreibungslogik.

2.6.1. Logik

Um Bedingungen und Effekte zu beschreiben, bietet es sich an, Aussagenlogik zu verwenden. Problematisch ist dabei allerdings, dass jede Aussage dann nur entweder wahr oder falsch sein kann. Damit ist aber die Modellierung von Angaben des Benutzers schwierig. Wahrheitswerte von Aussagen über Benutzerangaben sind erst dann wahr oder falsch, wenn der Benutzer die Angabe bereits gemacht hat. Der Wahrheitswert wird also erst während der Ausführung des Plans bekannt, nicht aber während der Planung. Um auch ausdrücken zu können, dass der Wahrheitswert einer Aussage unbekannt ist, kann man eine dreiwertige Logik mit den Wahrweitswerten T, F und $\perp$ verwenden.

Dabei gelten die naheliegenden Verknüpfungsregeln für die UND- (siehe Tabelle 2.1) bzw. ODER-Verknüpfung (siehe Tabelle 2.2) von Aussagen.

$\wedge$	T	F	$\perp$
T	T	F	$\perp$
F	F	F	F
$\perp$	$\perp$	F	$\perp$

Tabelle 2.1.: Wahrheitstabelle für die dreiwertige Konjunktion

$\vee$	T	F	$\perp$
T	T	T	T
F	T	F	$\perp$
$\perp$	T	$\perp$	$\perp$

Tabelle 2.2.: Wahrheitstabelle für die dreiwertige Disjunktion

Eine Beschränkung der Anzahl der Möglichkeiten kann erreicht werden, indem man in Vorbedingungen und Effekten nur Konjunktionen zulässt. Die Modellierung von Disjunktionen in Vorbedingungen kann dann indirekt über das Bereitstellen mehrerer alternativer Operatoren erreicht werden.

2.6.2. Operatoren

Im Folgenden stellen wir eine Möglichkeit zur Modellierung von Planungsoperatoren vor.

Planungsoperator
Vorbedingung

Formal ist ein Planungsoperator ein Tripel bestehend aus Vorbedingung, Aktion und Effekt. Die Vorbedingung gibt an, welche Voraussetzungen erfüllt sein müssen, um die Aktion ausführen zu können.

Aktion

Der Aktionsteil besteht aus einer Liste von Systemoperationen wie z. B. der Abfrage einer Wissensbasis. Aktionen sind atomar, d.h. sie werden entweder vollständig oder gar nicht ausgeführt. Der Aktionsteil ist nur bei der Planausführung von Interesse.

Effekt

Im Effektteil wird spezifiziert, welche Auswirkungen die Anwendung des Operators auf die Modellwelt hat. Die explizite Modellierung von Effekten erlaubt es, die Auswirkungen von Aktionen zu berechnen, ohne den Zustand der Modellwelt zu ändern.

2.7. Wissensmodellierung

In diesem Abschnitt werden die Grundlagen der Wissensmodellierung beschrieben. Die konkrete Modellierung der Wissensbasis von Pogy wird in 4.5 vorgestellt. In 5.3 gehen wir auf Implementierungsdetails ein.

2.7.1. Resource Description Framework

RDF

Das *Resource Description Framework (RDF)*wurde entwickelt, um der Vision eines „Semantischen Netzwerkes“, im englischen „Semantic Web“, näher zu kommen.

Semantisches Netz

Die Idee des Semantischen Netzes ist, Internetseiten mit maschinell lesbaren Informationen anzureichern, um diese automatisiert auffinden, verknüpfen und verarbeiten zu können.

Metainformationen

So ist es beispielsweise für einen Menschen kein Problem, die Struktur und Bedeutung eines Satzes wie „In der Mensa gibt es heute auf Linie 1 vegetarisches Gyros mit Tsatsiki und Fladenbrot für 2,05 EUR“ zu verstehen. Für eine Maschine ist die Bedeutung des Satzes nicht ohne weiteres zu erfassen, da ihr das nötige Hintergrundwissen des Menschen fehlt. Ein Mensch liest das Wort „Mensa“ und weiß, dass es dort Essen gibt. Für eine Maschine ist dieser Satz zunächst nur eine Zeichenkette ohne semantische Bedeutung. Es ist möglich, diesen Text mit Zusatzinformationen (sogenannten *Metainformationen*) anzureichern und damit der Maschine das Verständnis der Semantik zu erlauben. Dazu muss das System diese Metainformationen kennen und beispielsweise wissen, dass es in einer Mensa verschiedene Linien gibt und dort Essen ausgegeben wird. Anhand dieser Hintergrundinformationen lassen sich die Satzkomponenten in Relation zueinander setzen und damit der Inhalt erfassen. Eine Modellierung dieser Metainformationen könnte z. B. wie in Abbildung 2.2 aussehen.

```
<mensa>
 <linie>
  <nummer>1</nummer>
  <gericht>Veget. Gyros mit Tsatsiki und Fladenbrot</gericht>
  <preis>2,05EUR</preis>
 </linie>
</mensa>
```

Abbildung 2.2.: Teil eines maschinell lesbaren Mensaplans in XML Notation

Diese Modellierung ist allerdings nicht eindeutig. So zeigt Abbildung 2.3 eine weitere mögliche Modellierung dieses Sachverhalts. Die zweite Modellierung ist in diesem Fall vorzuziehen, da durch die Verknüpfung von Gericht und Preis auch mehrere Gerichte mit unterschiedlichen Preisen an einer Linie ausgegeben werden können. Beide Modellierungen haben den Nachteil einer hartkodierten Währung, in der der Preis als Zeichenkette und nicht als Zahl vorliegt. An diesem Beispiel sieht man die Bedeutung der Modellierung.

```
<mensa>
 <linie nummer="1">
  <gericht preis="2,05EUR">Veget. Gyros mit Tsatsiki und Fladenbrot</gericht>
 </linie>
</mensa>
```

Abbildung 2.3.: Andere Möglichkeit der Auszeichnung eines maschinell lesbaren Mensaplans in XML Notation

Bei dem Entwurf solcher Modelle gibt es prinzipiell zwei Freiheitsgrade: Zum einen die Strukturierung der Information (Semantik) und zum anderen die

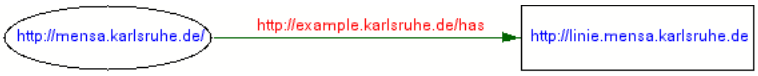


Abbildung 2.4.: Beispiel eines RDF-Statements

Kodierung der Information (Syntax). RDF ist eine ausgereifte und auf einem offenen Standard basierende Beschreibungssprache für semantische Modelle. Durch die weite Verbreitung von RDF stehen zahlreiche Werkzeuge zur Modellierung und Verarbeitung zur Verfügung. Dadurch kann sich der Entwickler ganz auf den Entwurf konzentrieren und muss sich keine Gedanken um die Serialisierung[3] machen.

Die RDF Syntax ist sehr einfach gehalten und orientiert sich an der menschlichen Sprache. Ein RDF-Statement besteht aus Subjekt, Prädikat und Objekt, womit es die Struktur eines minimalen Satzes der meisten menschlichen Sprachen widerspiegelt. Subjekt, Prädikat und Objekt werden jeweils mit einem Uniform Resource Identifier (URI) angegeben. Ein solcher URI wird grundsätzlich als Ressource bezeichnet. Als Objekt können auch definierte Datentypen gespeichert werden. Diese werden dann als Literale bezeichnet.

Abbildung 2.4 zeigt ein einfach gehaltenes Beispiel eines RDF-Statements als Graph. Er wird gelesen als „Die Ressource `http://mensa.karlsruhe.de/` hat eine Linie“. Der Graph wurde mit dem RDF Validation Service[4] des W3C erstellt. Die Linie wird mit einer URI angegeben. Auf diese Weise kann auch nach Eigenschaften (siehe 2.7.2.2) gefragt werden. Eine Serialisierung des Graphen in XML zeigt Abbildung 2.5.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:mensa="http://example.karlsruhe.de/">
  <rdf:Description rdf:about="http://mensa.karlsruhe.de/">
    <mensa:has>http://linie.mensa.karlsruhe.de</mensa:has>
  </rdf:Description>
</rdf:RDF>
```

Abbildung 2.5.: Beispielhafte Serialisierung eines RDF Modells

Auf die Serialisierung von RDF soll nicht weiter eingegangen werden, da Pogy nur auf Modellebene mit den Daten arbeitet und sich für Einlese- und Speicheraufgaben die Möglichkeiten der Java-Bibliothek Jena (siehe 2.7.3) zunutze macht. Eine detailliertere Einführung in RDF geben [Powe03] und [Lacy05].

2.7.1.1. RDF Query Language

RDQL

Die *RDF Query Language (RDQL)* ist eine Sprache, mit der die Wissensbasis abgefragt werden kann. Sie ist ähnlich aufgebaut wie die Structured Query Language (SQL). Abbildung 2.6 zeigt eine einfache Anfrage. Darin werden sowohl Subjekt wie auch Prädikat eines RDF-Statements festgelegt und anstelle

---

[3] Form der Datenspeicherung

[4] URL: `http://www.w3.org/RDF/Validator/`

des Objekts eine Variable `?RESULT` geschrieben. In diese Variable wird dann eine Liste von allen Objekten gespeichert, die in der Kombination von dem gegebenen Subjekt und Prädikat in der Wissensbasis vorkommen. Es ist auch möglich, mehrere Variablen in RDQL zu verwenden, oder mehrere Anfragetripel zu kombinieren.

```
SELECT ?RESULT WHERE
 (<http://mensa.karlsruhe.de/>,
  <http://example.karlsruhe.de/mensa#has>,
  ?RESULT)
```

Abbildung 2.6.: Beispielhafte RDQL Anfrage

2.7.2. Ontologie

Ontologie

Das Wissen über das Sein, also in obigem Beispiel das Wissen, dass eine Mensa verschiedene Linien hat, wird als *Ontologie* bezeichnet. Zur Modellierung solchen Wissens gibt es verschiedene Konzepte, die für diese Arbeit wichtigen werden der Vollständigkeit halber hier vorgestellt.

2.7.2.1. Klassen

Klasse
Individuum

Klassen stellen eine abstrakte Beschreibung der Struktur von *Individuen* oder Objekten dar. In unserem Mensabeispiel ist die konkrete Mensa einer Universität ein Individuum, während die allgemeine Beschreibung aller Mensen eine Klasse darstellt. So steht also in der allgemeinen Beschreibung der Hinweis, dass es verschiedene Linien gibt, während ein Individuum einen Verweis auf konkrete Linien besitzt. Um die Zugehörigkeit von Objekten zu Klassen zu modellieren, bietet RDF die vordefinierte *has-Type*-Relation an.

2.7.2.2. Eigenschaften

Eigenschaften

Eine Klasse kann auch sogenannte *Eigenschaften* (engl. Properties) haben. Die Klasse *Mensa* besitzt beispielsweise die Eigenschaft, verschiedene Linien zu haben. Weitere mögliche Eigenschaften sind Öffnungszeiten, Adresse oder Name der Mensa. Während der Name in einem einfachen String gespeichert werden kann, ist es sinnvoll, eine Linie wiederum als eigene Klasse zu modellieren. Dadurch kann sie selbst Eigenschaften besitzen.

2.7.2.3. Vererbung

Vererbung

Eines der wichtigsten Konzepte zur Strukturierung der Modellwelt ist die *Ist-Ein*-Beziehung (*Vererbung*). Vererbung bedeutet, dass eine Klasse die Eigenschaften einer anderen Klasse übernimmt (erbt) und wie ihre Oberklasse behandelt werden kann. Gegeben sei eine Klasse *Essen* mit der Eigenschaft *Preis* und eine Klasse *VegetarischesEssen*. Erbt *VegetarischesEssen* von *Essen*, so übernimmt sie die Eigenschaft *Preis* von ihrer Oberklasse *Essen*, kann aber auch weitere Eigenschaften definieren, die *Essen* nicht hat.

2.7.3. Jena

Jena

Jena[5] ist ein quelloffenes Semantic Web Framework von Hewlett-Packard, das es ermöglicht, Wissen semantisch (bedeutungstragend) zu verarbeiten. Jena stellt dazu eine RDF API zur Verfügung und versteht RDQL als Abfragesprache. Jena ist in der Lage, RDF aus einer XML Datei zu importieren und seine Wissensbasis in eine XML Datei zu serialisieren. Desweiteren kann sich der Anwender für eine von zwei Betriebsarten entscheiden: *Flüchtig* oder *Persistent*. Während in der ersten Betriebsart eventuelle Modifikationen nur so lange bestehen, wie das Programm ausgeführt wird, wird die Wissensbasis in der zweiten Betriebsart dauerhaft verändert. Prinzipiell bietet sich die Betriebsart *Flüchtig* während der Entwicklung an, da man z. B. bei Tests immer denselben initialen Datenbestand vorfinden möchte. Dagegen ist in einer produktiven Umgebung meist eine persistente Wissensbasis erwünscht, um neu erworbenes Wissen über einen Neustart hinaus zu erhalten. Die Betriebsart kann je nach Bedarf umgeschaltet werden.

Reasoner

Jena bietet verschiedene *Reasoner*[6] an, die bei der Abfrage verwendet werden können. Der einfachste Reasoner verwendet nur Informationen, die explizit abgelegt wurden. Es gibt aber auch komplexe Reasoner, die mit Hilfe von Regeln implizit in der Ontologie enthaltene Informationen extrahieren. Wir verwenden einen einfachen transitiven Reasoner, der die Vererbungsrelation *reflexiv* und *transitiv* interpretiert. Dadurch werden z. B. bei der Abfrage aller Essen auch die vegetarischen Essen in die Resultatliste aufgenommen, obwohl ihr eigentlicher Typ nicht *Essen* ist.

2.8. Werkzeuge

Bei der Entwicklung von Pogy wurden verschiedene Werkzeuge eingesetzt. Die beiden wichtigsten, Protégé und Eclipse, stellen wir hier vor.

2.8.1. Protégé

Protégé

Zur Erstellung der initialen Wissensbasis wurde *Protégé*[7] verwendet. Protégé ist ein unter der *Mozilla Public License*[8] veröffentlichter plattformübergreifender Ontologie- und Wissensbasis-Editor. Er wird an der Stanford University School of Medicine entwickelt. Protégé kann die erstellten Wissensbasen in verschiedene Formate exportieren, unter anderem in das von Jena unterstützte RDF. Abbildung 2.7 zeigt Protégé bei der Bearbeitung von Eigenschaften einer Person. Links im Bild ist die Klassenhierarchie von Pogy zu sehen, auf sie und die Modellierung von Pogys Wissen wird in 4.5 genauer eingegangen.

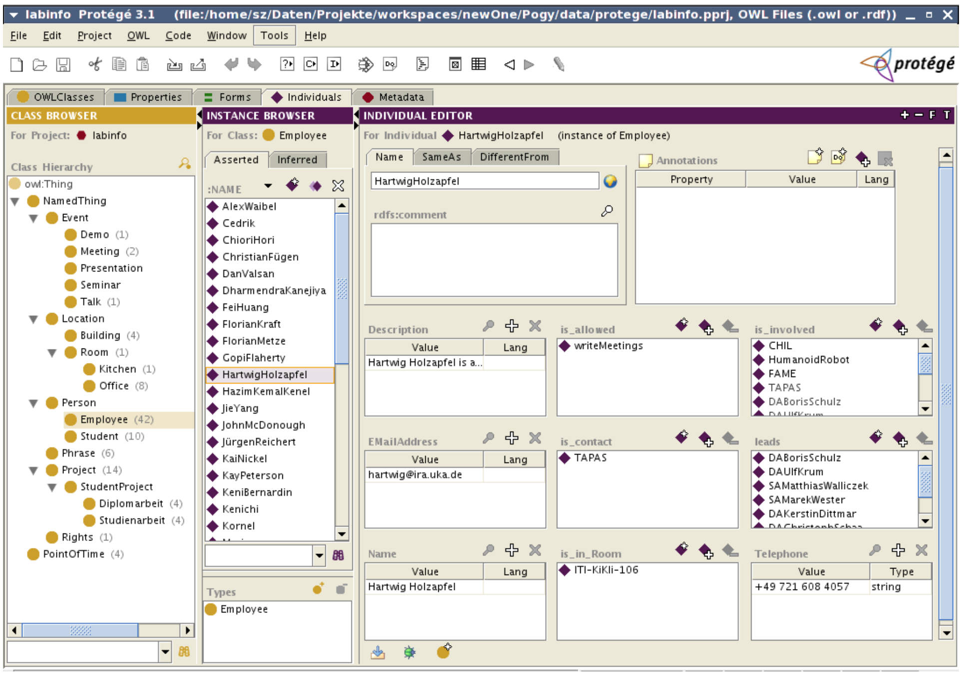
---

[5] URL: `http://jena.sourceforge.net/`

[6] Komponente, die aus vorhanden Daten neue Fakten ableiten kann

[7] URL: `http://protege.stanford.edu/`

[8] URL: `http://www.mozilla.org/MPL/`



HartwigHolzapfel

Description

Value	Lang
Hartwig Holzapfel is a...	

is_allowed: writeMeetings

is_involved: CHIL, HumanoidRobot, FAME, TAPAS, DABorisSchulz

EMailAddress

Value	Lang
hartwig@ira.uka.de	

is_contact: TAPAS

leads: DABorisSchulz, DAUlfKrum, SAMatthiasWalliczek, SAMarekWester, DAKerstinDittmar

Name

Value	Lang
Hartwig Holzapfel	

is_in_Room: ITI-KiKli-106

Telephone

Value	Type
+49 721 608 4057	string

Abbildung 2.7.: Protégé in Aktion

2.8.2. Eclipse

Pogy wurde mit Hilfe von *Eclipse*[9] entwickelt. Eclipse ist eine plattformübergreifende Entwicklungsumgebung (engl. Integrated Development Environment, IDE). Die Einrichtung von Pogy in Eclipse wird in Anhang A erklärt. *Eclipse*

2.8.3. Wissensbasisexport

Da Pogy Eingaben, die sich auf seine Antworten beziehen, verstehen muss, ist es nötig, die von Tapas verwendeten Eingabegrammatiken mit der Wissensbasis konsistent zu halten. Dazu werden Namen von Individuen aus der Wissensbasis in die Datei `database.txt` exportiert. Dies erledigt ein Start des Programms `DatabaseBuilder`, das sich im `de.uka.ira.isl.dlg.pogy.util`-Package befindet.

---

[9] URL: `http://www.eclipse.org`

3. Bestehende Arbeiten

In diesem Kapitel stellen wir einige verwandte Arbeiten mit planbasiertem Ansatz vor. Einen sehr detaillierten Überblick über vorhandene natürlichsprachliche Dialogsysteme gibt [McTe02].

3.1. Circuit-Fix-It

Das Circuit-Fix-It System [SmHi94] ist ein Dialogsystem, das Benutzern bei der Reparatur elektronischer Schaltungen zur Seite steht. Das System hat die notwendigen Kenntnisse über die Schaltung, während der Benutzer Informationen über deren Zustand liefert. Ziel ist die Reparatur der Schaltung, also das Funktionieren all ihrer Teilkomponenten. Ziel und Absicht des Benutzers sind so fest vorgegeben und müssen nicht im Dialog gefunden werden. Bei der Reparatur kooperieren Benutzer und System, indem das System versucht, in Form eines Beweises die notwendigen Schritte zur Reparatur zu finden. Aufgabe des Benutzers ist es, die dem System fehlenden Informationen über den Schaltungszustand zu liefern und Reparaturaktionen auszuführen. Auf diese Weise entsteht der Dialog. Die Aufgabe der Reparatur wird vom System in kleinere Unteraufgaben aufgeteilt, die jeweils weiter unterteilt werden können bis zu Elementaroperationen. Ein wichtiges Merkmal des Circuit-Fix-It Systems ist das Benutzermodell, das die Vertrautheit des Benutzers mit der Schaltung widerspiegelt. Es entsteht dynamisch während des Dialogs und entscheidet darüber, wie stark die einzelnen Aufgaben unterteilt werden, wenn sie dem Benutzer präsentiert werden. Dementsprechend detailliert fallen Fragen und Anweisungen des Systems aus. Auf diese Weise können sowohl unerfahrene Benutzer als auch Experten das System benutzen, ohne bei der Problemlösung überfordert oder unnötig ausgebremst zu werden.

3.2. TRAINS

Das TRAINS System [ASFH$^+$94] ist ein Dialogsystem, das den Benutzer bei Transportplanungsproblemen in einer vereinfachten Welt unterstützt. Als Beispiel dient dabei der Transport von Waren zum Beispiel von Lagerhäusern zu Fabriken. TRAINS kennt nur Züge als Transportmittel. Das System besitzt Kenntnisse über Transportrouten und mögliche Probleme. Der Benutzer gibt das Ziel vor, indem er dem System mitteilt, an welchem Ort sich Waren befinden und wohin diese gebracht werden sollen. Außerdem kann er Strategien zur Lösung vorschlagen. Das System kümmert sich um Details wie die

Verfügbarkeit von Zügen auf den gewählten Strecken. TRAINS versucht, unter Beachtung der Benutzervorgaben Lösungen zu finden und informiert den Benutzer über mögliche Probleme. Außerdem unterbreitet es Lösungsvorschläge. Dadurch ist TRAINS in der Lage, einen Dialog mit wechselnder Initiative zu führen. Die Benutzereingabe kann dabei sowohl über Sprache als auch über Tastatureingabe erfolgen. Zur Kommunikation mit dem Benutzer wird Sprachausgabe und ein graphisches Display verwendet, auf dem mögliche Transportrouten angezeigt werden. Die Planung steht bei TRAINS nicht sehr im Vordergrund, da es sich im wesentlichen um Routing-Aufgaben handelt.

3.3. TRIPS

TRIPS [FeAl98] ist eine Weiterentwicklung von TRAINS. Die Ein- und Ausgabemodalitäten wurden beibehalten. Die Logistik- und Transportwelt von TRIPS ist jedoch wesentlich komplexer, da nun eine ganze Reihe unterschiedlicher Transportmittel zur Verfügung steht[1]. Als Beispiel dient dabei die Planung einer Evakuierung mehrerer Städte. Die Planungskomponente ist sehr ausgeprägt. TRIPS kann Schlussfolgerungen über zeitlich ausgedehnte Aktionen ziehen und auch Ressourcenbeschränkungen beachten. Außerdem ist es in der Lage, die Aktionen mehrerer Agenten miteinander zu koordinieren. TRIPS besitzt eine Planungskomponente, mit der hypothetische Situationen geplant werden können. Daher ist TRIPS in der Lage „Was wäre wenn“ Situationen durchzuspielen. Es besitzt ein stochastisches Weltmodell und kann daher versuchen, seine Pläne zunächst simuliert auszuführen. Dabei sammelt es statistische Daten über den Weltzustand und kann den Benutzer so über mögliche Probleme informieren.

---

[1] Siehe auch `http://www.cs.rochester.edu/research/cisd/projects/trains/tripscompare.html`

4. Beschreibung der Arbeit und Theorie

Diese Arbeit beschäftigt sich mit der Erstellung eines planbasierten Dialogsystems für Informationssysteme. Im Umfeld des Sonderforschungsbereichs *588 Humanoide Roboter* der Universität Karlsruhe benutzen wir den Dialogmanager Tapas (siehe Abschnitt 2.1) als Framework. Die Domäne des implementierten Beispielsystems Pogy ist das Interactive System Laboratories (ISL) in Karlsruhe, zu dessen Mitarbeitern, Projekten etc. Pogy Auskunft geben kann.

4.1. Theorie

Das Verständnis einer sprachlichen Äußerung ist eine komplexe Aufgabe. Zunächst muss ein eingehendes Sprachsignal erkannt und in Text umgewandelt werden. Dadurch erhält man eine Sequenz von Worten aus einem Wörterbuch. Nun gilt es, aus dieser Wortsequenz die semantische Bedeutung zu extrahieren, die oftmals nur im Kontext des Dialogs richtig interpretiert werden kann. Zum Kontext gehören neben der Dialoghistorie auch Informationen über Dialogteilnehmer, Ort und Zeitpunkt des Dialogs.

Nachdem dieser Schritt erfolgreich abgeschlossen ist, kann eine angemessene Reaktion (Systemantwort) auf die Äußerung des Benutzers erstellt werden. Darauf liegt der Schwerpunkt von Pogy. Bevor im weiteren die dazu notwendigen Schritte und Komponenten näher erläutert werden, soll zunächst der planbasierte Ansatz grob skizziert werden. Wir gehen von Dialogen aus, die zum Zweck des Wissensaustauschs geführt werden, also Dialogen mit einem definierten Dialogziel. Die Erfüllung dieses Dialogziels betrachten wir als Planungsproblem. Der Dialog ergibt sich auf diese Weise als Nebenprodukt der Planausführung, indem Benutzereingaben den Weltzustand verändern und Systemantworten durch Planungsoperatoren ausgelöst werden.

Im Folgenden gehen wir genauer auf die Erkennung von Benutzerzielen, die Interpretation von Äußerungen im Kontext und die Definition des Begriffs Kontext ein. Die Dialogmodellierung als Planungsproblem wird vorgestellt ebenso wie die Repräsentation und Ablage von Wissen in Form einer Wissensbasis. Die Spracherkennung ist nicht Gegenstand dieser Arbeit und wird nicht weiter im Detail behandelt. Die Extraktion der Bedeutung mit Hilfe semantisch annotierter kontextfreier Grammatiken wird nur kurz angerissen und kann im Detail in 2.1 nachgelesen werden.

4.2. Introspektion

Damit ein System in der Lage ist, Aussagen über sich selbst zu treffen, muss es die Fähigkeit der Introspektion besitzen. Das bedeutet, dass ein System ein Modell seiner selbst besitzt, und so jederzeit seinen eigenen Zustand erfassen und auch modifizieren kann. Ein Planungssystem ist dadurch in der Lage, hypothetische Situationen durchzuspielen und so herauszufinden, ob eine Operatorenkombination zu einem Zielzustand führt. Es kann also „Überlegungen“ der Art „Was wäre wenn?“ anstellen.

4.3. Diskurs

Beim Einsatz auf einem humanoiden Roboter ergeben sich Einschränkungen hinsichtlich Umgebung und Anzahl an Benutzern. Zusammen mit dem Entwurfsziel eines Informationssystems resultieren daraus Vereinfachungen in der Modellwelt, die in die Modellierung einfließen und im Folgenden vorgestellt werden.

4.3.1. Kontext und Fokus

Kontext

Zur Charakterisierung einer Gesprächssituation benutzen wir vier Entitäten, die damit den *Kontext* bilden: Den *Gesprächszeitpunkt*, der vor allem zur Interpretation relativer Daten wie *heute* als auch zur Ergänzung nicht genannter Zeitpunkte genutzt wird, *Gesprächsort* und *Gesprächsverlauf*. Der Gesprächsort kann analog zum Zeitpunkt benutzt werden, um fehlende Ortsangaben zu ergänzen. Die vierte Entität sind die *Dialogteilnehmer*, d.h. Pogy als System und der Benutzer.

Um jedoch beispielsweise Anaphern (siehe 4.3.2) aufzulösen, sind im Kontext immer noch zu viele Objekte enthalten. Hier schafft der Fokus Abhilfe. Im Gegensatz zu den bisher genannten Kontextelementen, die für alle Dialogteilnehmer identisch sind, modellieren wir den Fokusbereich für Pogy als System und die jeweiligen Benutzer separat. Eine Alterungsfunktion sichert die Entfernung längerer Zeit nicht referenzierter Objekte.

Beispiel 4.1
Das vorgestellte Konzept des Kontexts *mit seinen Unterelementen* Ort, Zeit, Gesprächsverlauf *und* Fokus *soll an einem Beispiel illustriert werden. Betrachten wir folgenden Dialog zwischen* U *und* S*:*

U1 Is there a CHIL meeting today?

S2 The CHIL meeting is scheduled for 2 pm.

U3 Is there anything CHIL related afterwards?

S4 None that I'm aware of.

U5 *Ok, I see.*

S6 *Are you interested in more details on the CHIL project?*

U7 *No, thanks. The meeting will be in room 102, right?*

S8 *No, the CHIL meeting takes place in meeting room 127.*

U9 *Ok, good to know. Thanks.*

S10 *You're welcome.*

Zu Anfang besteht der Kontext nur aus Gesprächsort und -zeit, die im weiteren Verlauf als konstant angesehen werden können. Der Gesprächsverlauf baut sich sukzessive auf und enthält am Dialogende alle Äußerungen von U *und* S *zeitlich sortiert. Die Fokusbereiche von* U *und* S *entwickeln die größte Dynamik: In* U1 *bringt* U *das Objekt* CHIL meeting *in seinen Fokus.* S *übernimmt dies in seinen Fokus und fügt zusätzlich die Zeitangabe* 2 pm *hinzu. Der Zeitpunkt des Meetings wäre als Kontext 2. Art zwar auch über das Meeting-Objekt erreichbar, zur Auflösung von Ambiguitäten*[1] *ist es aber notwendig, den Zeitpunkt ebenfalls in den Fokus zu nehmen. In* U5 *überlässt* U *die Gesprächsinitiative* S, *was* S *dazu nutzt, das Objekt* CHIL project *in den Fokus zu bringen.*

4.3.2. Auflösung von Ellipsen und Anaphern

Menschen kommunizieren multimodal und benutzen Kontextreferenzen, um die Kommunikationsbandbreite zu erhöhen. Das stellt Dialogsysteme vor große Probleme, denn diese besitzen typischerweise weder eine vergleichbare Sensorik, noch können sie die vorhandenen Sensorinformation ähnlich optimal fusionieren. Auch die Auflösung von Kontextreferenzen gelingt nicht immer. Beispiele für sprachliche Kontextreferenzen sind Ellipsen und Anaphern. Eine *Ellipse* ist ein Sprachmittel, bei dem durch Auslassung von Satzteilen unvollständige Sätze gebildet werden. Demgegenüber ist eine *Anapher* eine Spracheinheit, die als Ersatz für eine vorangehende Einheit steht. Letzere wird *Antezedens*[2] *The CHIL meeting* genannt.

Ellipse

Anapher

Antezedens

Beispiel 4.2

U1 *Where is the CHIL meeting today?*

S1 *The CHIL meeting is in room 102.*

U2 *And when?*

In U2 *kann man eine Ellipse beobachten. „And when?“ ist kein grammatikalisch korrekter Satz.*

Durch eine kleine Variation von Beispiel 4.2 zeigt sich der Unterschied zwischen Ellipse und Anapher.

---

[1] Mehrdeutigkeiten

[2] sprachliche Einheit, auf die sich eine Anapher bezieht

Beispiel 4.3

U1 Where is the CHIL meeting today?

S1 The CHIL meeting is in room 102.

U2' And when is it?

Bei dem Wort it *in* U2' *handelt es sich um eine Anapher auf die Antezedens.*

Als problematisch beim Auflösen von Anaphern erweist sich die Mehrdeutigkeit der Antezedenzen. Aus grammatikalischer Sicht könnte sich *it* auch auf *room 102* beziehen. Eine simple Auflösung der Anapher in dem Sinn, dass das letztgenannte Objekt als Antezedens verwendet wird, ist folglich nicht ausreichend. Als Antezedens kommen nur Objekte in Frage, die einen Zeitpunkt als Eigenschaft besitzen. Die Menge der Objekte mit dieser Eigenschaft kann über eine Ontologie ermittelt werden. Um zusätzlich der Eigenschaft der Antezedens, für eine vorangehende Einheit zu stehen, zu genügen, wird die Suche auf die Fokusbereiche der Dialogteilnehmer beschränkt.

4.4. Planung

Da es sich bei Pogy um ein Informationssystem handelt, verfolgt es keine eigenen Ziele, sondern macht es sich zur Aufgabe, die Ziele des Benutzers zu erfüllen.

Beispiel 4.4

U1 Where is the CHIL meeting today?

S1 The CHIL meeting is in room 102.

Im Beispiel 4.4 ist es Ziel des Benutzers herauszufinden, an welchem Ort das CHIL Meeting stattfindet. Das System befindet sich zum Zeitpunkt der Anfrage im Anfangszustand.

Um Ziele zu erreichen, hat Pogy eine Anzahl von Operatoren (siehe 2.6.2) zur Verfügung, deren Voraussetzungen und Effekte bekannt sind.

4.4.1. Modellierung und Auffinden von Zielen

Um Planung durchführen zu können, muss ein Ziel bekannt sein: „If an automatic system has got the task to engage in a dialogue, first of all it must be able to recognize the user's intentions(s)“ [GBFK^{+}03]. Dazu wird eine Zieldefinition ebenso wie eine Möglichkeit, ein Ziel im Dialog zu finden, benötigt.

Ein Ziel besteht aus drei Teilen: Einem Namen, dem Typ des Resultats, das im Erfolgsfall dem Benutzer präsentiert wird, sowie einer Vorlage zur Formulierung einer Wissensbankanfrage.

Zur Zielfindung ist die speechact-Theorie wichtigstes Hilfsmittel. Dazu werden Benutzereingaben speechacts zugeordnet und auf Ziele abgebildet. Kann kein Ziel direkt zugeordnet werden, wird mit Hilfe des Diskurses die Zielsuche auf den Gesprächsverlauf ausgeweitet.

4.4.2. Planungsalgorithmus

Generell gibt es bei Planungsystemen zwei grundlegende Vorgehensweisen: Vorwärtsverkettung und Rückwärtsverkettung [RuNo03].

4.4.2.1. Vorwärtsverkettung

Vorwärtsverkettung

Bei der *Vorwärtsverkettung* (Forward-Chaining) geht man von einem Anfangszustand aus und untersucht dann alle ausführbaren Operatoren. Jeder dabei auftretende Zustand wird mit dem Zielzustand verglichen. Bei Erreichen eines Zielzustandes wird der Plan ausgegeben. Es wird also vom Anfangszustand aus eine Breitensuche im Planungsgraphen durchgeführt. Dies garantiert, dass ein kürzester Plan gefunden wird.

Der Nachteil dabei ist, dass die Vorgehensweise wenig zielgerichtet ist. Gibt es viele ausführbare Aktionen, aber nur wenige Kombinationen führen zum Ziel, so werden viele Pläne, die nichts zur Lösung beitragen, ausprobiert.

4.4.2.2. Rückwärtsverkettung

Rückwärtsverkettung

Die zweite Möglichkeit ist die *Rückwärtsverkettung* (Backward-Chaining): Man beginnt beim Zielzustand und baut den Plan rückwärts auf. Dabei werden nur Aktionen verwendet, die zum Ziel hinführen. Im nächsten Schritt wird nun versucht, die Vorbedingungen der gewählten Operationen zu erfüllen. Sobald keine Vorbedingungen mehr offen sind, ist ein gültiger Plan erreicht. Man führt also eine Breitensuche vom Zielzustand durch den Planungsgraphen aus.

Problematisch ist hierbei die Behandlung von nichtdeterministischen Operationen.

4.4.2.3. Nichtdeterministische Umgebungen

Da das Dialogsystem mit einem Menschen interagiert, kann der Erfolg einiger Operationen nicht immer garantiert werden. Wenn das System einen Operator zum Erfragen einer Information einplant, so gibt es keine Möglichkeit sicherzustellen, dass das System die gewünschte Information tatsächlich erhält. Ein anderes Beispiel für eine nichtdeterministische Operation ist die Abfrage der Wissensbasis nach gewissen Informationen. Es ist nicht garantiert, dass die Information tatsächlich vorhanden ist.

Eine Lösungsmöglichkeit wäre, bei jedem Operator alle möglichen Ausgänge zu spezifizieren und einen vollständingen Plan für alle Eventualitäten zu erstellen [RuNo03]. Allerdings ist dieses Vorgehen nicht mit der Rückwärtsverkettung vereinbar und es werden sehr große Pläne generiert, die aber nur zu einem sehr geringen Teil benötigt werden. Das Vorgehen wäre also sehr ineffizient. Außerdem ist die vollständige Spezifikation aller Operatoren sehr aufwändig und auch nicht besonders intuitiv.

4.4.2.4. Optimistische Planung

Wir haben uns entschieden, bei allen Operatoren nur den gewünschten Ausgang zu spezifizieren und weiter eine Rückwärtsverkettung durchzuführen. Dazu plant das System optimistisch und macht zwei Annahmen: Alle Operationen funktionieren und falls eine Variable bekannt wird, so wird diese auf den für den Plan erforderlichen Wert gesetzt. Dadurch werden sehr kurze Pläne berechnet, die in vielen Fällen funktionieren. Durch die optimistische Planung kann der Erfolg aber nicht sichergestellt werden. Deshalb ist es nötig, während der Planabarbeitung vor der Ausführung jedes Operators zu prüfen, ob seine Vorbedingungen erfüllt sind. Ist dies nicht der Fall, so wird die Planausführung abgebrochen. Bei einem Abbruch kann entweder neu geplant oder ein Fehlschlag gemeldet werden. Wenn der Benutzer eine Information nicht zur Verfügung stellte, so kann es helfen, einfach noch einmal nachzufragen. Ist aber eine Information in der Wissensbasis nicht vorhanden, so sind einige Ziele nicht erfüllbar und der Benutzer muss darüber informiert werden.

4.4.3. Planabarbeitung

Nach der Planung erfolgt die Planausführung. Eingabe für diese Phase ist ein vollständiger Plan, also eine Liste von Operatoren, deren Anwendung es möglich macht, das Ziel zu erfüllen.

Der Ablauf der Planabarbeitung sieht wie in Abbildung 4.1 spezifiert aus:

```
while plan not empty
    operator = pop(plan.operators)
    if operator.preconditions = true
        execute(operator.actions)
    else
        replan
    fi
endwhile
```

Abbildung 4.1.: Ablaufbeschreibung der Planausführung

4.5. Wissensmodellierung

Das Wissen von Pogy wurde mit der Hilfe von RDF (siehe 2.7.1) modelliert. Hier soll ein Überblick über dieses Wissen gegeben werden. Eine vollständige Beschreibung des Wissens würde den Rahmen dieser Arbeit sprengen. Auf der beiligenden CD befindet sich aber eine mit Protégé erstellte Dokumentation.

Pogy soll in der Lage sein, über ein Institut verschiedene Auskünfte zu erteilen. Einen ersten Überblick über das Wissen von Pogy gibt Abbildung 4.2. Dort ist die vollständige Klassenhierarchie abgebildet. Die allgemeinste Klasse ist dabei *Thing*. Davon erbt beispielsweise *NamedThing*. Diese Klasse stellt zwei Eigenschaften zur Verfügung. Zum einen eine Beschreibungsmöglichkeit, zum

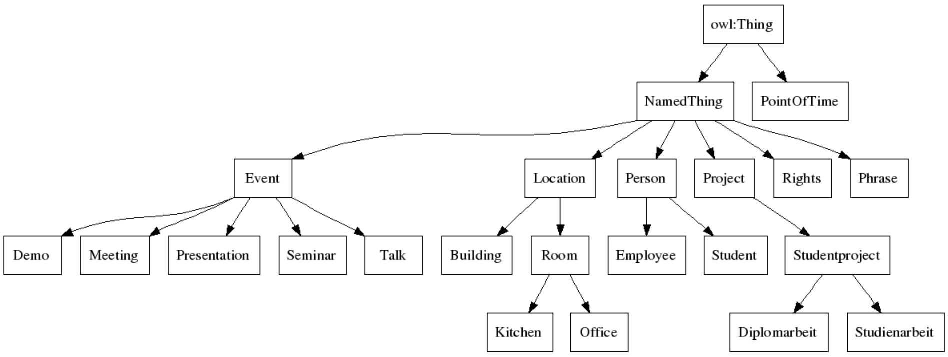


Abbildung 4.2.: Klassenhierarchie der Wissensbasis

anderen ein Feld für einen Namen. Somit ist es Pogy möglich, für jedes Individuum einer Unterklasse von *NamedThing* einen Namen und eine Beschreibung zu erfragen.

4.5.1. Beispielszenario

Exemplarisch stellen wir die Modellierung des Projekt-Szenarios vor. Pogy soll in der Lage sein, Auskunft über Projekte zu geben. Typische Informationen über Projekte sind beispielsweise Projektname, eine kurze Projektbeschreibung, evtl. eine eigene Projekthomepage, ein Ansprechpartner sowie weitere Mitarbeiter. Weiterhin sind Mitarbeiter über Telefon und E-Mail erreichbar und können ebenfalls beschrieben werden.

4.5.1.1. Aussagen

Aus dieser Beschreibung können die folgende Aussagen gewonnen werden:

- Es gibt Projekte
- Projekte haben Namen
- Projekte haben eine Beschreibung
- Projekte haben evtl. eine Homepage
- Es gibt Mitarbeiter
- Mitarbeiter haben E-Mail Adressen
- Mitarbeiter haben Telefonnummern
- Mitarbeiter können Ansprechpartner für ein Projekt sein
- Mitarbeiter können an einem Projekt arbeiten
- Zu jedem Mitarbeiter gibt es einen Beschreibungstext

Aus diesen Aussagen kann das Modell entworfen werden.

4.5.1.2. Modell

Es ergeben sich zwei Klassen: Projekte und Mitarbeiter. Dabei hat die Klasse *Projekt* die Eigenschaften Name, Beschreibung und Homepage und die Klasse *Mitarbeiter* hat die Eigenschaften E-Mail, Telefonnummer, Ansprechpartner und Mitarbeiter. Abbildung 4.3 zeigt einen ersten Entwurf des Modells als Graph. Es fällt auf, dass beide Klassen die Eigenschaften Name und Beschreibung besitzen. Diese gemeinsamen Eigenschaften werden am Besten in einer eigenen abstrakten Oberklasse zusammengefasst (vgl. Abbildung 4.4).

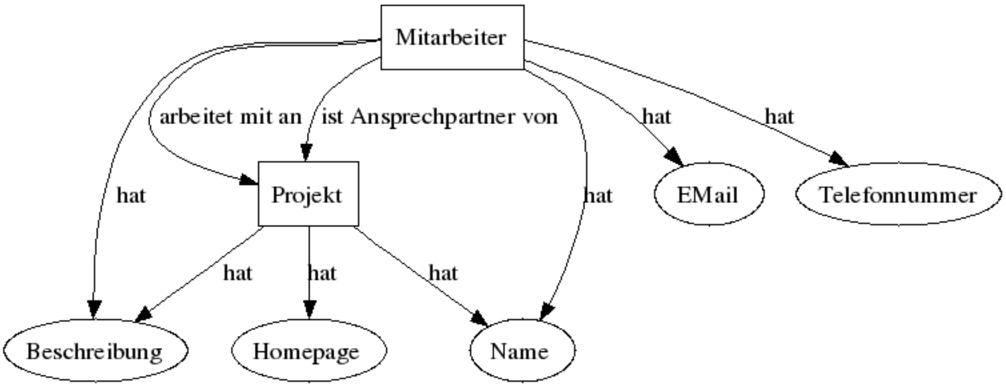


Abbildung 4.3.: Erstes Modell des Beispielszenarios

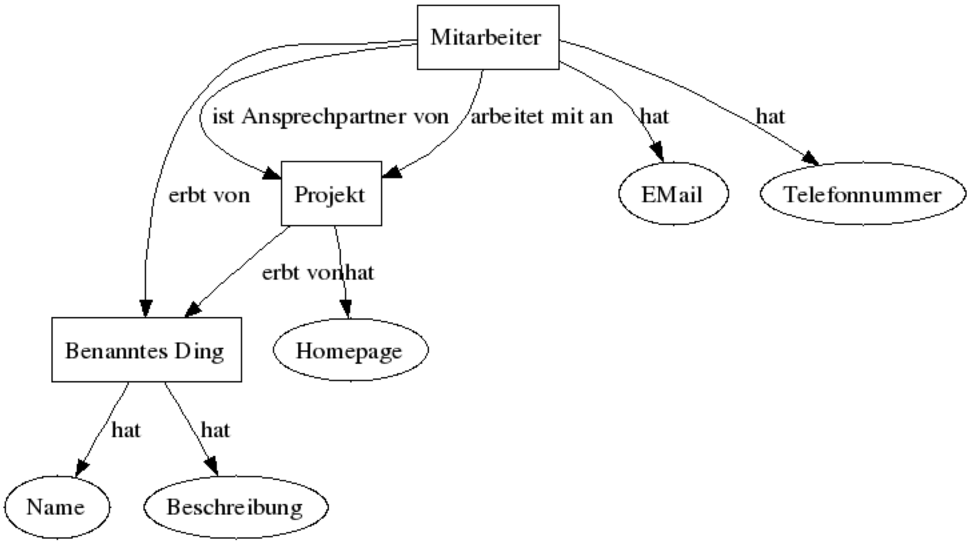


Abbildung 4.4.: Modell des Beispielszenarios mit abstrakter Klasse

Ein Punkt, der an dieser Stelle noch angesprochen werden soll, ist die Modellierung von Zeitangaben. Dafür gibt es die Klasse `PointOfTime`, die einen Zeitpunkt darstellt und als Eigenschaft einen `Timestamp` besitzt. In diesem wird die seit dem 1.1.1970 vergangene Zeit bis zum gewünschten Zeitpunkt in

Millisekunden übergeben. Dies hat den Vorteil, dass einfacher mit Datumsangaben gearbeitet werden kann. So unterstützt Jena den Vergleich von `long` - Werten in RDQL-Anfragen, jedoch nicht den von Datums-Werten[3].

---

[3] Es kann sein, dass diese Funktionalität in einer späteren Version von Jena vorhanden sein wird.

5. Implementierung

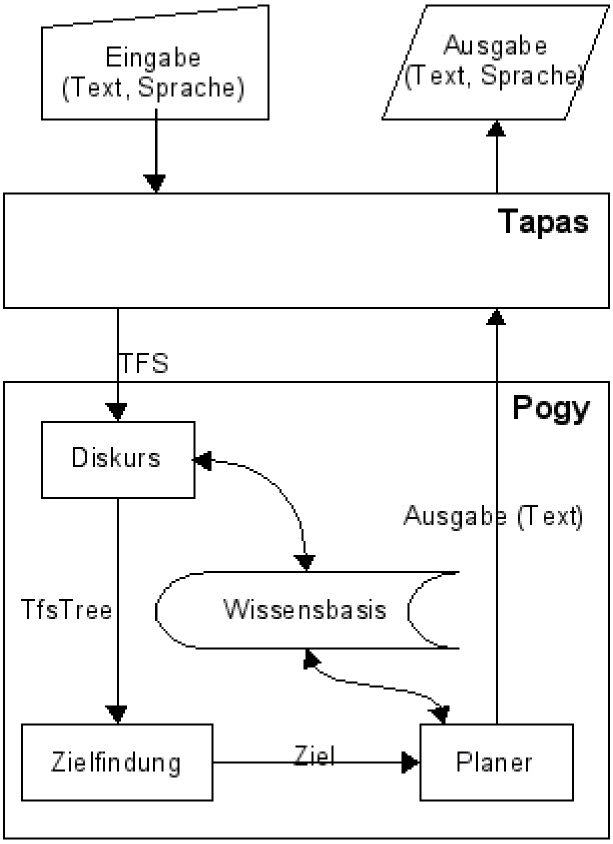


Abbildung 5.1.: Gesamtübersicht über das System

Eine eingehende Benutzeräußerung wird von Tapas gegebenenfalls mit Hilfe eines Spracherkenners in eine TFS umgewandelt. Dazu werden Ontologie und Grammatik benötigt. Die TFS wird an Pogy weitergeleitet.

Der erste Verarbeitungsschritt in Pogy ist das DiscourseUpdate. Mit Hilfe der Wissensbasis wird ein mit weiteren Informationen angereicherter *TfsTree* erzeugt und an den GoalMatcher weitergeleitet, der gegebenenfalls ein neues Ziel erstellt. Der Planer versucht, dieses Ziel zu erreichen. Dabei generiert er Anfragen an die Wissensbasis und Ausgaben an den Benutzer, die über Tapas als Sprachsignal ausgegeben werden.

5.1. Diskurs

Im Diskurs wird die Äußerung des Benutzers im Kontext interpretiert. Diese Aufgabe unterteilt sich in weitere Unteraufgaben. Zunächst müssen relative

Daten, Ellipsen und Anaphern aufgelöst werden. Danach erfolgt die Einordnung der Äußerung in den Gesprächsverlauf, was durch eine Erwartungshaltung des Systems unterstützt wird. Diese Erwartungshaltung ergibt sich aus dem bisherigen Dialog und wird dynamisch zusammen mit der Systemantwort generiert.

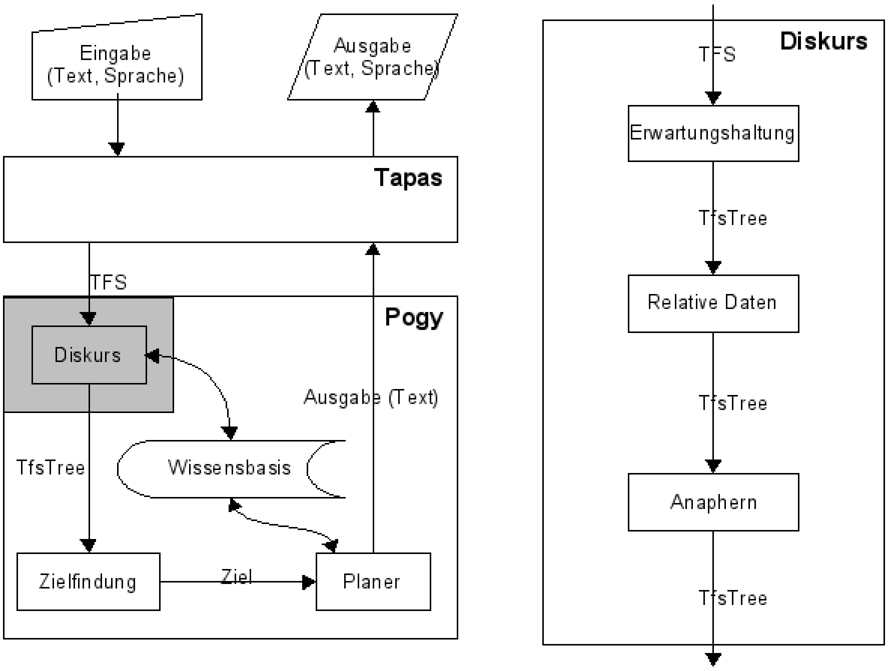


Abbildung 5.2.: Prozesse im Diskurs

5.1.1. Ersetzung relativer Daten

Menschen verwenden relative Zeitpunkte wie „jetzt“ oder „in zwei Stunden“ und relative Ortsangaben wie „hier“. Diese müssen in absolute Angaben umgewandelt werden, denn die Wissensbasis kennt nur absolute Daten, um domänenunabhängig zu sein. Um diese Umwandlung durchführen zu können, muss der Standort bekannt sein. Um den Einsatz auf einer mobilen Plattform zu ermöglichen, wird dieser nicht beim Systemstart gesetzt, sondern kann zur Laufzeit verändert werden. Relative Zeitpunkte werden durch eine Normalisierungskomponente umgewandelt.

5.1.2. Auflösung von Ellipsen

Ellipsen sind dadurch gekennzeichnet, das sie gegen grammatikalische Regeln verstoßen. Das hat zur Folge, dass wir Ellipsen explizit in der kontextfreien Grammatik vorsehen müssen, um sie verstehen zu können. Die Ellipse „And when?“ aus 4.2 wird beispielsweise gleich behandelt wie die Aussage „And when is it?“, d. h. das Problem der Auflösung von Ellipsen wird von der Grammatik insofern gelöst, als dass Ellipsen im Folgenden wie Anaphern behandelt werden können.

5.1.3. Anaphern

Wir beschränken uns bei der Auflösung von Anaphern auf formal korrekte Beziehungen zwischen Anapher und Antezedens, beide müssen also nicht zwangsläufig koreferent[1] sein. Das ist an dieser Stelle keine starke Einschränkung, da die Grammatik keine mehrdeutigen Antezedenzen in einem Satz zulässt. Über mehrere Aussagen hinweg kann es zu Ambiguitäten kommen, die durch die Heuristik des Fokusbereichs in den meisten Fällen korrekt aufgelöst werden.

Zur Erkennung einer Anapher muss die Grammatik entsprechend präpariert werden. Das ist relativ einfach, indem für einzelne Objekte außer dem konkreten Objektnamen auch Pronomen zugelassen werden. Beispielsweise werden außer dem Namen einer Person auch die Pronomen he, she, it, his, her, its, him erkannt. In der resultierenden TFS fehlt dann das entsprechende Feature und es wird versucht, dieses aus dem Fokusbereich zu ergänzen.

Um das Zusammenspiel von Grammatik, Ontologie und Fokus zu verdeutlichen, betrachten wir ein Beispiel.

Beispiel 5.1
Der Dialog sei wie folgt:

U1 Who is responsible for the CHIL meeting?

S2 That is Rainer Stiefelhagen.

U3 What is his email address?

In U3 *soll die Anapher* his *aufgelöst werden, die zugehörige Antezedens ist* Rainer Stiefelhagen. *Der für die Auflösung wesentliche (hier vereinfacht dargestellte) Teil der Grammatik:*

```
<act_describePerson,V,_> =
    ’what is’ <obj_person,N,_> { PERSON obj_person }
    ’email address’

<obj_person,N,_> = <pronoun> : <concretePerson>;
```

Abbildung 5.3.: Teil einer Grammatik

<concretePerson> enthält die Namen bekannter Personen, <pronoun> die weiter oben aufgelisteten Pronomen.

Das Feature PERSON von <act_describePerson> wird von <concretePerson> ausgefüllt, von <pronoun> nicht. Wird <pronoun> geparst, so wird versucht, die Anapher aufzulösen. Betrachten wir nun den Ablauf, der sich aus obigem Dialog ergibt.

In U1 *wird* the CHIL meeting *im Benutzerfokus abgelegt,* S2 *fügt* Rainer Stiefelhagen *dem Systemfokus hinzu. Daraufhin wird in* U3 *eine Anapher erkannt*

---

[1] Beziehung zwischen Spracheinheiten, von denen sich eine auf die andere oder beide aufeinander beziehen.

und die zugehörige Antezedens im Fokus gesucht. Die Antezedens muss vom Typ obj_person *sein, was nur für* Rainer Stiefelhagen *gilt. Die TFS kann daher an der entsprechenden Stelle ergänzt werden.*

Unifikation

Etwas komplizierter ist die Auflösung von Anaphern durch *Unifikation*. Hier wird die ursprüngliche TFS ersetzt durch eine neue, spezialisierte TFS. Das ist hilfreich, wenn für die Antezedens mehrere Typen zulässig sind und man sich die Möglichkeiten der Vererbung zunutze machen möchte. Beispielsweise wird die Frage „Where is it?“ auf diese Weise aufgelöst. *It* kann durch mehrere Typen ersetzt werden, zum Beispiel einen Raum (allgemeiner einen Ort) oder ein Meeting (allgemein Event). In der Ontologie werden die in Abbildung 5.4 dargestellten Klassen definiert.

```
class act_whereIs inherits generic:action;

class act_whereIsEvent inherits act_whereIs
{
    event : EVENT;
    point_of_time : POINT_OF_TIME;
};

class act_whereIsPerson inherits act_whereIs
{
    obj_person : PERSON;
    point_of_time : POINT_OF_TIME;
};

class act_whereIsLocation inherits act_whereIs
{
    location :  LOCATION;
};
```

Abbildung 5.4.: Ontologieauszug für „Where is it?“

Die Grammatik wird so angepasst, das die Eingabe „Where is it“ die in Abbildung 5.5 gezeigte TFS liefert.

```
[
    [ labinfo:act_whereIs,V,_ ]
    { WHERE IS IT }
]
```

Abbildung 5.5.: Vereinfachte TFS für „Where is it?“

Bei Eingang dieser TFS durchsucht Pogy die Ontologie. Dort gibt es die Spezialisierungen *act_whereIsEvent*, *act_whereIsPerson* sowie *act_whereIsLocation*. Sei der Dialog nun wie folgt:

U1 Which event is there today?

S2 I know the Pogy Event.

U3 Where is it?

Dann befindet sich im Fokus ein Meeting namens Pogy und die Unifikation ergibt die TFS aus Abbildung 5.6.

```
[ labinfo:act_whereIsEvent
  labinfo:EVENT
    [ labinfo:meeting
      generic:NAME
      [ "pogy" ]
    ]
]
```

Abbildung 5.6.: TFS für „Where is it?“ nach Unifikation

5.1.4. Erwartungshaltung

Stellt Pogy dem Benutzer eine Rückfrage, so wird eine Antwort von einem bestimmten Typ erwartet. Die Erwartungshaltung *Erwartungshaltung* wird durch die Expectation Variable (siehe 5.1.6) angezeigt, die eine Liste gesuchter Typen bereitstellt. Durch eine Lookup Tabelle werden diese Typen in TFS Pfade umgewandelt. Wird einer dieser Pfade in einer eingehenden TFS gefunden, so gilt die Erwartungshaltung als erfüllt und der Wert des TFS Pfads wird an die Expectation Variable übermittelt.

Erwartungshaltung

Beispiel 5.2
Pogy erwartet auf die Frage „Where does the meeting take place?“ eine Ortsangabe. Der zugehörige Eintrag in operators.xml *ist in Abbildung 5.7 vereinfacht dargestellt.*

```
<action>
    <say>Where does the meeting take place?</say>
    <set variable="expectation.location.name" to="statement.location" />
</action>
```

Abbildung 5.7.: Erwartungshaltung in operators.xml

Für location.name *ist in* variablePaths.xml *die in Abbildung 5.8 sichtbare Transformation in einen TFS Pfad gespeichert.*

```
<transformation var="location.name" adl="labinfo:location generic:NAME" />
```

Abbildung 5.8.: Transformation einer Variable in einen TFS Pfad

Die Benutzerantwort „In room 102“ ergibt die in Abbildung 5.9 dargestellte TFS.

```
labinfo:act_identifyLocation
  labinfo:LOCATION
  [ labinfo:location
    generic:NAME
    [ "room 102" ]
  ]
```

Abbildung 5.9.: TFS für Eingabe „In room 102“

Der gesuchte TFS Pfad ist vorhanden und sein Wert „room 102“ wird an die Expectation Variable übermittelt. Die Erwartungshaltung gilt jetzt als erfüllt.

5.1.5. Kontext und Fokus

Der Kontext kennt die aktuelle Zeit und den Aufenthaltsort, um relative Daten aufzulösen. Der Gesprächsverlauf wird in Form einer Liste verwaltet. Die Fokusbereiche sind für Benutzer und System getrennt modelliert und bestehen jeweils aus einer Liste von zuletzt referenzierten Objekten. Zu jedem Objekt wird eine time to live (TTL) gespeichert, die automatisch dekrementiert wird. Objekte mit negativer TTL werden aus dem Fokus entfernt. Die Dekrementierung der TTL erfolgt prinzipiell nach jeder Benutzer- bzw. Systemäußerung, jedoch nur bei verändertem Fokusbereich. Objekte werden dadurch nach kurzer Zeit aus dem Fokus verdrängt, falls neue Objekte in den Dialog eingebracht werden.

5.1.6. Variablen

Variablenbaum

Die in 4.2 angesprochene Introspektion realisiert Pogy durch Variablen, die in einer Baumstruktur angeordnet sind: Dem Variablenbaum. Abbildung 5.10 zeigt einen Ausschnitt aus diesem Baum. Jedem Blatt ist ein Wert zugeordnet, der über seinen Pfad abgerufen werden kann. Beispielsweise greift man per *user.name* auf den Wert der Variablen *name* im Pfad . → *user* → *name* zu. Die einzelnen Variablen besitzen unterschiedliche Bedeutungen, die wichtigsten werden jetzt vorgestellt.

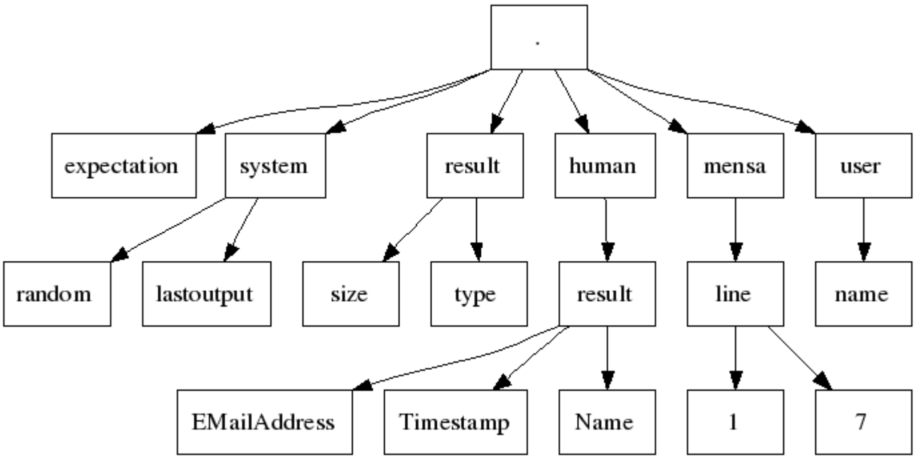


Abbildung 5.10.: Ausschnitt aus dem Variablenbaum

system Variable

system Die system Variable bietet Lesezugriff auf wichtige Systemgrößen. Dazu gehören Systemzeit, der Zufallsgenerator und die letzte Ausgabe an den Benutzer.

result Variable

result Ergebnisse von Anfragen an die Wissensbank werden in der result Variablen abgelegt. Neben den eigentlichen Daten in Form einer Liste kann die Anzahl der Ergebnisse und ihr Typ abgefragt werden.

expectation Durch Setzen einer (beliebigen) Variable der expectation Variable wird eine Erwartungshaltung gesetzt. Der Unterpfad muss dabei eine gültige Transformation in einen TFS Pfad sein. Die Transformationen in TFS Pfade werden in der Datei *variablePaths.xml* eingetragen. Der Wert der Variable ist ein Pfad zu einer Variable aus dem Variablenbaum, die als Speicherort des Wertes einer erfüllten Erwartung benutzt wird. Abbildung 5.9 verdeutlicht dies.

expectation Variable

statement Diese Variable wird in erster Linie dazu benutzt, um in Kombination mit der expectation Variable Eingaben des Benutzers zu speichern.

statement Variable

human Das interne Speicherformat vieler Variablen ist für Menschen schwer verständlich. Die human Variable sorgt für eine natürlichere Ausgabe, indem z. B. Listen durch Kommata bzw. and verknüpft werden oder Zeitangaben an die regionalen Eigenheiten angepasst werden.

human Variable

mensa Die mensa Variable ist ein Beispiel für eine domänenabhängige Erweiterung von Pogy. Hierüber kann das Essensangebot der Karlsruher Universitätsmensa zu einem beliebigen Datum abgerufen werden. Die Aktualisierung des Essensangebots erfolgt dabei automatisch über das Internet, sofern verfügbar.

mensa Variable

5.2. Planung

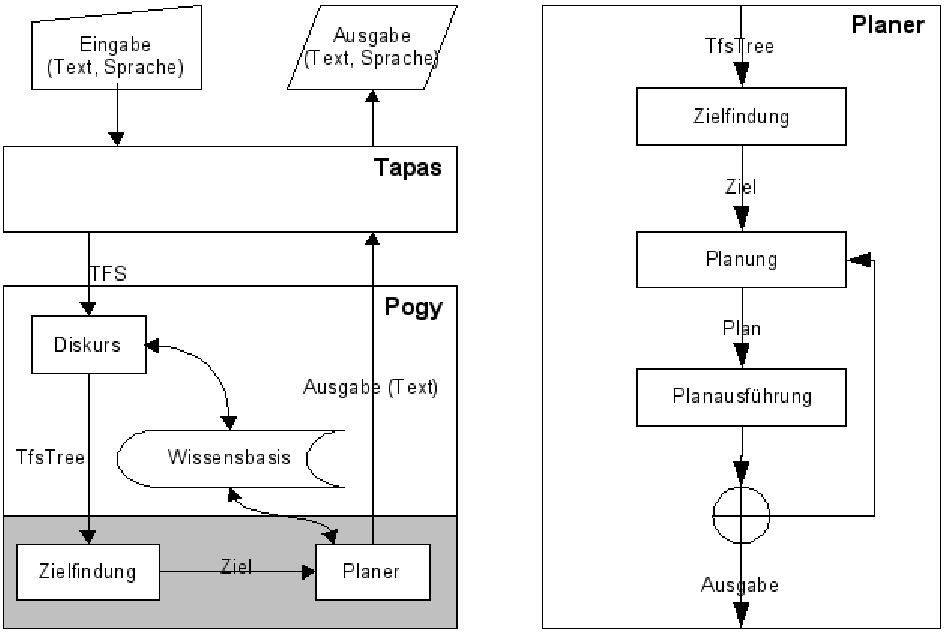


Abbildung 5.11.: Prozesse in der Planung

Die Planung unterteilt sich in drei Schritte. Zunächst wird aus dem eingehenden TfsTree ein Ziel extrahiert. Im Anschluss daran generiert der Planer einen Plan zum Erreichen des Ziels, der daraufhin ausgeführt wird. Planung und Planausführung können iteriert werden, falls die Planausführung fehlschlägt.

5.2.1. Zielfindung

Ziele werden in der XML Datei `goals.xml` spezifiziert. Abbildung 5.12 zeigt die Spezifikation für die Verabschiedung des Benutzers. Name und Ergebnistyp werden direkt im `<goal>`-Tag als Zeichenkette abgelegt. Das `<description>`-Tag dient ausschließlich Dokumentationszwecken. Im `<requirements>`-Tag wird festgelegt, unter welchen Bedingungen das Ziel gültig ist. In Beispiel 5.12 ist es lediglich erforderlich, dass der Speechact `labinfo:act_goodbye` ist, der Benutzer sich also verabschiedet hat.

Im `<requirements>`-Tag können zwei verschiedene Arten von Bedingungen stehen. Zum einen kann mit dem `<known>`-Tag die Existenz eines TFS-Pfades gefordert werden. Mit dem `<content>`-Tag wird sowohl die Existenz des Eintrags gefordert, als auch die Übereinstimmung mit dem im `value`-Attribut angegebenen Wert.

Im `<template>`-Tag steht die Vorlage für eine Wissensbasisabfrage. Im Fall der Verabschiedung ist die Vorlage leer, da keine Abfrage erforderlich ist.

```
<goal name="goodbye" type="" >
    <description>say goodbye</description>
    <requirements>
        <content path="" value="labinfo:act_goodbye" />
    </requirements>
    <template></template>
</goal>
```

Abbildung 5.12.: Zielmodellierung der Verabschiedung des Benutzers

Abbildung 5.13 zeigt ein Ziel, das im `<template>`-Tag eine Vorlage zur Abfrage von Personen mit sich bringt. Die Vorlage besteht aus einer Liste von Constraints, die optional zu Blöcken gruppiert werden können, um ihre Zusammengehörigkeit festzulegen. Das verhindert nicht vollständig spezifizierte Anfragen an die Wissensbasis.

Ein Constraint (vgl. 5.14) hat drei Parameter *var*, *pred* und *value* und bildet die Tripel-Syntax von RDF bzw. der Abfragesprache RDQL nach. Als Prädikate dienen entweder RDF-Relationen oder Vergleichsoperatoren wie `eqic` (equalsIgnoreCase). Im *value*-Parameter können feste Werte, zuvor definierte Variablen oder aus der Eingabe extrahierte Werte eingesetzt werden. Letztere werden durch das Präfix *path:* gekennzeichnet und durch den Wert der Variablen im angegebenen TFS-Pfad ersetzt.

5.2.2. Operatoren

Abbildung 5.15 zeigt die XML Darstellung eines einfachen Planungsoperators. Ein Planungsoperator wird mit einem `<operator>`-Tag eingeleitet. Dieser muss

```
<goal name="elaborate" type="Person" >
 <requirements>
  <content path="" value="labinfo:act_elaboratePerson" />
        <known   path="labinfo:PERSON generic:NAME" />
    </requirements>
    <template>
        <block>
            <constraint var="?result"
            pred="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
            value="#Person" />
            <constraint var="?result"
                        pred="#Name"
                        value="?personName" />
            <constraint var="?personName"
                        pred="eqic"
                        value="path:labinfo:PERSON generic:NAME" />
        </block>
    </template>
</goal>
```

Abbildung 5.13.: Zielmodellierung zur Ausgabe von Informationen über eine Person

```
<constraint var="?projectName" pred="eqic"
            value="path:labinfo:PROJECT generic:NAME" />
```

Abbildung 5.14.: Constraint zur Einschränkung einer Variablen auf den Wert eines TFS Pfades

ein `name` und ein `cost` Attribut enthalten. Während das `name` Attribut zur Dokumentation für den Entwickler gedacht ist, werden mit dem `cost` Attribut die Kosten für diesen Operator angegeben. Als weiteren Eintrag enthält jeder Operator eine Beschreibung, die lediglich Dokumentationszwecken dient und mit `<description>`-Tag gesetzt wird.

Außerdem enthält der Operator wie in 2.6.2 beschrieben eine Vorbedingung `<precondition>`-Tag, eine Menge von Aktionen `<action>`-Tag und einen Effekt `<effect>`-Tag.

Im Folgenden sollen die möglichen Bedingungen `<condition>`-Tag und ihre Anwendung gezeigt werden. Desweiteren werden die in Pogy vorhandenen Aktionen beschrieben.

```
<operator name="greeting␣user" cost="1">
 <description>great the user</description>
 <precondition>
 </precondition>
 <action>
  <say>Hi user!</say>
 </action>
 <effect>
  <condition variable="goal.fulfilled"
             predicate="equal"
             value="greeting" />
 </effect>
</operator>
```

Abbildung 5.15.: Beispiel eines Operators

```
<condition variable="statement.time"
     predicate="bigger" value="var:system.now" />
```

Abbildung 5.16.: Beispiel einer Bedingung

5.2.2.1. Bedingungen

Bedingungen können in einem Operator als Vorbedingung und als Effekt verwendet werden. Hierzu gibt es den `<precondition>`-Tag bzw. den `<effect>`-Tag. Eine Bedingung wird in beiden Fällen mit einem `<condition>`-Tag kenntlich gemacht und besitzt grundsätzlich die drei Attribute `variable`, `predicate` sowie `value`. Es stehen dabei die in 5.1.6 aufgeführten Variablen mit der dort erklärten Pfadschreibweise zur Verfügung. Für das `predicate` -Attribut gibt es die in Tabelle 5.1 aufgelisteten Operatoren.

smaller	vergleicht, ob der Inhalt der Variablen echt kleiner als der Wert in `value` ist
smallerEqual	vergleicht, ob der Inhalt der Variablen kleiner oder gleich dem Wert in `value` ist
equal	vergleicht, ob der Inhalt der Variablen genau gleich dem Wert in `value` ist
unequal	vergleicht, ob der Inhalt der Variablen ungleich dem Wert in `value` ist
biggerEqual	vergleicht, ob der Inhalt der Variablen gleich oder größer dem Wert in `value` ist
bigger	vergleicht, ob der Inhalt der Variablen echt größer als der Wert in `value` ist
known	überprüft, ob die Variable im Kontext bekannt ist
unknown	überprüft, ob die Variable im Kontext unbekannt ist

Tabelle 5.1.: Operationen im `predicate` -Attribut

Im Normalfall benutzt man konstante Werte für Variablen. Manchmal ist es aber nötig, dynamische Werte, wie die aktuelle Zeit zu berücksichtigen. Um dies zu ermöglichen, kann man im `value` -Attribut den Wert einer Variablen einsetzen. Dazu muss lediglich ein `var:` vor die Variable gesetzt werden. Abbildung 5.16 zeigt eine Bedingung, die genau dann wahr ist, wenn der Wert in der Variablen `statement.time` echt größer ist als der Wert in der Variablen `system.now`. Diese Bedingung ist genau dann erfüllt, wenn der angegebene Zeitpunkt in der Zukunft liegt.

5.2.3. Aktionen

Aktion

Pogy kennt verschiedene *Aktionen*, die in beliebiger Reihenfolge in den Operatoren verwendet werden können. Sie werden im `<action>`-Tag aufgeführt und lassen sich im wesentlichen in drei Gruppen aufteilen.

```
<reload component="operators"/>
<reload component="knowledgebase"/>
<reload component="goals"/>
<reload component="pathTranslator" />
```

Abbildung 5.17.: Mögliche Konfigurationen der ReloadAction

5.2.3.1. Benutzerinteraktion

SayAction

Hierbei handelt es sich um Aktionen, die zur Interaktion mit dem Benutzer dienen. In dieser Arbeit ist das die *SayAction*. Eine SayAction wird mit einem `<say>`-Tag eingeleitet und per `</say>`-Tag abgeschlossen. Der zwischenstehende Text wird bei der Planausführung als Antwort zurückgegeben. Innerhalb des Textes kann auf alle Kontextvariablen des Systems zugegriffen werden, indem der Variablenpfad von Backslashs umschlossen wird. Es sind auch weitere Aktionen zur Benutzerinteraktion denkbar, die in Kapitel 7 aufgeführt werden.

5.2.3.2. Wartungsaktionen

ReloadAction

Für Wartungszwecke steht die *ReloadAction* zur Verfügung. Mit ihrer Hilfe ist es möglich, einzelne Komponenten des Systems neu zu laden. Abbildung 5.17 zeigt die möglichen Einstellungen für diese Aktion. Dabei handelt es sich um die Systemkomponenten in Tabelle 5.2

operators	Operatoren Datei
knowledgebase	Die Wissensbasis
goals	Die Zieldefinitionen
pathTranslator	Die Übersetzungseinheit zwischen den zwei verwendeten Ontologien

Tabelle 5.2.: Wartungsaktionen

Grundsätzlich gilt: Kann eine Komponente nicht neu geladen werden, so wird die alte Komponente verwendet. Pogy informiert in jedem Fall über den Erfolg oder Misserfolg einer ReloadAction.

Auch hier sind weitere Aktionen denkbar. Diese werden in Kapitel 7 beschrieben.

5.2.3.3. Systeminterne Aktionen

Die hier aufgeführten Aktionen dienen zur Steuerung des Systems.

FocusAction

FocusAction Die *FocusAction* sorgt dafür, dass die Objekte einer erfolgreichen Wissensbasisanfrage in den Fokusbereich des Systems übernommen werden. Gleichzeitig wird dadurch die Alterungsfunktion des Fokus ausgelöst. Sie wird durch ein `<focus/>`-Tag gesetzt und benötigt keine weiteren Parameter.

LoadResultAction

LoadResultAction Eine wird eingesetzt, wenn zur Erfüllung eines Ziels eine Wissensbankanfrage notwendig ist. Mit Hilfe des Ziels wird die Anfrage erstellt, an die Wissensbank*LoadResultAction* weitergeleitet und das Ergebnis in der `result` -Variablen festgehalten. Gesetzt wird diese Aktion mit einem `<loadResult/>`-Tag.

ShrinkAction

ShrinkAction Die Anzahl der Ergebnisse, die die LoadResultAction liefert, ist nach oben nicht begrenzt. Wenn eine Ausgabe an den Benutzer erfolgen soll, muss die Liste auf eine vernünftige Länge gekürzt werden. Das erledigt die *ShrinkAction*, die mit `<shrink/>`-Tag gesetzt wird.

WriteAction

WriteAction Hiermit wird der Wissensbasis ein neues Event (z. B. ein Meeting) hinzugefügt. Voraussetzung dafür ist, dass die notwendigen Daten in der `statement` -Variablen vorliegen. Dazu gehören Veranstaltungszeit sowie Veranstaltungsdatum, die jeweils in Millisekunden seit dem 01.01.1970 in `statement.time` bzw. `statement.date` gespeichert werden. In `statement.location` wird der Name des Veranstaltungsorts und in `statement.name` der Name des Meetings erwartet. Die WriteAction berechnet bei ihrer Ausführung aus Veranstaltungszeit und -datum einen Zeitstempel und sucht in der Wissensbasis nach der URI des Veranstaltungsortes. Danach trägt sie das Meeting durch den Aufruf einer Methode der Wissensbasis in die Wissensbasis ein. Der entsprechende Ausdruck in `operators.xml` lautet `<writeStatement/>`-Tag.

SetAction

SetAction Diese Aktion ermöglicht das Setzen der Variablen im Variablenbaum. Als Parameter werden der Variablenpfad (durch Punkte getrennt) und ein Wert erwartet. Der Wert ist entweder eine Konstante oder eine Variable aus dem Variablenbaum. Letzteres wird durch das Präfix *var:* gekennzeichnet. Die SetAction wird wird mit einem `<set variable= to= >`-Tag gesetzt.

Beispiel 5.3
Abbildung 5.3 zeigt das Setzen der Variable gameSolution *auf den Wert der Variablen* system.random *(eine spezielle Variable, die bei jedem Zugriff eine Zufallszahl liefert).*

```
<set variable="gameSolution" to="var:system.random" />
```

Abbildung 5.18.: Setzen einer Variablen im Variablenbaum

UnsetAction

UnsetAction Hiermit wird eine Variable aus dem Variablenbaum gelöscht. Dazu muss lediglich ein `<unset variable= />`-Tag mit dem Pfad zur zu löschenden Variablen gesetzt werden.

5.3. Wissensbasis

In Abbildung 5.19 ist die Anbindung der Wissensbasis zu sehen. Intern verwendet die Wissensbasis Jena (siehe 2.7.3). Im rechten Teil der Abbildung ist

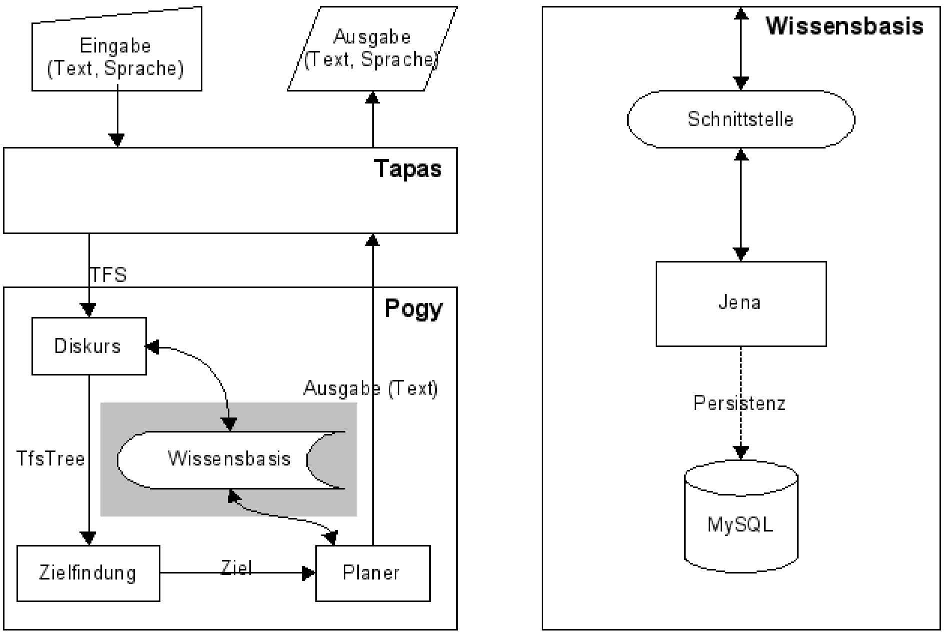


Abbildung 5.19.: Pogys Wissensbasis und ihre Anbindung an Jena

zudem angedeutet, dass Jena optional eine relationale Datenbank, wie z. B. MySQL, verwenden kann, um Persistenz sicherzustellen.

Die Wissensbasis stellt Pogy zwei Arten von Informationen zur Verfügung. Dazu gehört zum einen das Faktenwissen, also das Wissen um einzelne Individuen von Klassen, wie z. B. Personen oder Termine, und zum anderen das „Wissen des Seins“. Letzteres wird auch Ontologie genannt und beinhaltet z. B. das Wissen darüber, dass eine Person einen Namen besitzt.

Faktenwissen

Ontologie

Die Wissensbasis wird vom Diskurs sowie vom Planer über eine wohldefinierte Schnittstelle verwendet.

Der Diskurs benötigt die Wissensbasis, um die eindeutige URI eines Individuums (den Gesprächsort) zu bestimmen. Damit ist es möglich, Kontextinformationen aufzulösen. Ansonsten wird im Diskurs nur die Adl2 Ontologie verwendet, da diese das im Diskurs relevante Dialogwissen enthält.

Der Planer benötigt die Wissensbasis während der Planung, um Vorbedingungen von Operatoren zu überprüfen. Zudem wird die Wissensbasis während der Planausführung verwendet, um die vom Benutzer gewünschten Informationen zu erhalten. Dabei kann es sich beispielsweise um Auskünfte über eine Person wie E-Mail Adresse oder Telefonnummer handeln oder um eine Projektbeschreibung oder Informationen über ein Meeting.

Ziel war, eine möglichst leichtgewichtige und übersichtliche Schnittstelle für die Verwendung innerhalb von Pogy zu erstellen, um die oben erwähnten Informationsarten abfragen zu können. Dazu wurde das in 2.7.3 vorgestellte Framework Jena mit einer eigenen Schnittstelle versehen, die nur die von Pogy benötigten Funktionen zur Verfügung stellt. Diese soll hier beschrieben werden.

5.3.1. Wissensbasisanfragen

Wie bereits in 5.2.1 erwähnt, werden in der Datei `goals.xml` die Ziele definiert. Jedes Ziel kann auch eine Anfrage in Form eines RDQL-Strings[2] enthalten. Um diese Anfragen an die Wissensbasis zu stellen, muss sie lediglich als Parameter an eine Methode übergeben werden. Diese Methode gibt eine Liste mit passenden Einträgen aus der Wissensbasis zurück. Existieren keine passenden Einträge, wird eine leere Liste zurückgegeben.

Beispiel 5.4
Eine typische Anfrage an die Wissensbasis ist die Abfrage des Zeitpunkts, zu dem ein Meeting stattfindet. Im `<template>`*-Tag des dazugehörigen Ziels ist dazu die in 5.20 abgebildete Vorlage zur Abfrage der Wissensbasis definiert.*

```
<template>
 <block>
  <constraint var="?result"
   pred="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
   value="#PointOfTime" />
  <constraint var="?event"
   pred="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
   value="#Event" />
  <constraint var="?event"
   pred="#StartTime"
   value="?result" />
  <constraint var="?event"
   pred="#Name"
   value="?eventName" />
  <constraint var="?eventName"
   pred="eqic"
   value="path:labinfo:EVENT generic:NAME" />
 </block>
</template>
```

Abbildung 5.20.: Vorlage zur Wissensbasisanfrage in goals.xml

Bei der Instantiierung dieser Vorlage wird aus einem Block eine RDQL Anfrage erzeugt. Dazu werden zunächst die `pred` *Attributwerte der einzelnen* `<constraint>`*-Tags in gültige RDQL Prädikate umgeformt. Gleichermaßen wird bei vorangestelltem* `path:` *der Attributwert in* `value` *durch den Wert des angegebenen TFS Pfads ersetzt. Angenommen, die TFS enthält im Pfad* `labinfo:EVENT -> generic:NAME` *den Wert „pogy meeting“, dann werden* `pred` *und* `value` *Werte folgendermaßen ersetzt:* `= /^pogy meeting$/i`*. Dieser reguläre Ausdruck trifft genau auf die Zeichenkette „Pogy Meeting“ zu, ohne Groß- und Kleinschreibung zu beachten. Die gesamte RDQL Anfrage hat damit die in Abbildung 5.21 dargestellte Form.*

Das Lesen von RDQL Abfragen ist etwas gewöhnungsbedürftig. Eine umgangssprachliche Formulierung der Abfrage in 5.21 sieht so aus: „Gib eine Liste von Zeitpunkten zurück und berücksichtige bei der Suche all die Objekte, die die Eigenschaften Name und Startzeit besitzen und vom Typ Event *sind. Der Name der zutreffenden Individuen muss ohne Berücksichtigung von Groß- und Kleinschreibung der Zeichenkette* Pogy Meeting *entsprechen.“*

---

[2] vgl. Abschnitt 2.7.1.1

```
SELECT ?result
 WHERE
  ( ?result, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
   <http://pogy.dlg.isl.ira.uka.de/labinfo#PointOfTime> ),
  ( ?event, <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
   <http://pogy.dlg.isl.ira.uka.de/labinfo#Event> ),
  ( ?event, <http://pogy.dlg.isl.ira.uka.de/labinfo#StartTime>,
   ?result ),
  ( ?event, <http://pogy.dlg.isl.ira.uka.de/labinfo#Name>,
   ?eventName )
 AND ?eventName =~ /^pogy$/i
```

Abbildung 5.21.: Automatisch erzeugte RDQL Abfrage

Die erzeugte RDQL Anfrage wird bei der Planausführung von der LoadResultAction an die Wissensbasis übergeben. Die LoadResultAction nimmt die Ergebnisliste entgegen und speichert sie in einer Variablen im Variablenbaum, wo sie während der Planabarbeitung (z. B. für die ShrinkAction) zur Verfügung steht.

5.3.2. Abfragen von Eigenschaften

Um Eigenschaften von Individuen zu erfragen, stellt die Wissensbasis eine weitere Methode zur Verfügung. Sie benötigt die URI einer Ressource sowie den Namen der gesuchten Eigenschaft. Aus diesen Angaben wird eine RDQL-Anfrage generiert, die dann ihrerseits eine Wissensbasisanfrage startet und das Ergebnis zurückgibt.

Diese Methode wird beispielsweise im Diskurs verwendet, um die eindeutige URI des Standorts zu ermitteln, und die `result` -Variable ermöglicht mit Hilfe dieser Methode komfortablen Zugriff auf die Eigenschaften einer Klasse. In einem Operator kann durch `human.result.Name` der Name eines Individuums abgefragt werden. Bezüglich der Abfrage besteht kein Unterschied zwischen `human.result.Name` und `result.Name`. Die `human` Variable sorgt für eine „menschenlesbare“ Formatierung der im Unterpfad übergebenen Variablen `result.Name`.

Beispiel 5.5
Es soll eine Liste von Projekten ausgegeben werden. In einer ersten Anfrage an die Wissensbasis wird eine Liste von URIs aller zutreffender Projekte zurückgegeben. Bei der Ausgabe an den Benutzer soll jedoch nicht die URI des Projekts, sondern dessen Name ausgegeben werden. Dazu bedient sich die result *Variable der* `getProperty()` *Methode der Wissensbasis. Zur Ausgabe des Namens des Projekts mit der URI* `http://pogy.dlg.isl.ira.uka.de/labinfo#TAPAS` *wird dabei die in Abbildung 5.22 abgebildete RDQL Anfrage erzeugt.*

```
SELECT ?result
WHERE ( <http://pogy.dlg.isl.ira.uka.de/labinfo#TAPAS>,
 <http://pogy.dlg.isl.ira.uka.de/labinfo#Name>, ?result )
```

Abbildung 5.22.: Abfrage der Eigenschaft Name eines Individuums

5.3.3. Eintragen von Events

Änderungen an der Wissensbasis (z. B. Hinzufügen oder Löschen von Individuen) sind mit RDQL als reiner Abfragesprache nicht möglich. Stattdessen müssen Methoden, die Jena bereitstellt, benutzt werden: Die Methode `createIndividual()` legt ein Individuum an und dessen Eigenschaften werden durch Aufruf von `addProperty()` hinzugefügt. Diese Methoden arbeiten direkt auf dem Ontologiemodell von Jena.

Um von den Jena eigenen Methoden zu abstrahieren, wird in der Schnittstelle eine Methode `addEvent()` angeboten, die die notwendigen Parameter eines Events entgegennimmt und die entsprechenden Aufrufe von `createIndividual()` und `addProperty()` durchführt. Sie wird ausschließlich von der WriteAction (siehe Abschnitt 5.2.3.3) verwendet und benötigt vier Parameter, aus denen ein neuer Ereigniseintrag für die Wissensbasis generiert und in diese eingetragen wird. Der erste Parameter ist die Art des Ereignisses. So könnte es sich beispielsweise um ein Meeting oder um einen Seminarvortrag halten. Tabelle 5.3 zeigt die möglichen Ereignisse, die in der Wissensbasis definiert und an dieser Stelle einsetzbar sind.

Als weiterer Parameter wird der Name des Ereignisses benötigt. Dies kann theoretisch eine beliebige Zeichenkette sein, praktisch werden aber von Spracherkenner bzw. Eingabegrammatik nur vorher definierte Namen akzeptiert. Als dritter Parameter muss ein Veranstaltungsort als RDF Ressource (d.h. mit eindeutiger URI) übergeben werden. Zuletzt muss die Veranstaltungszeit in Millisekunden[3], die seit 01.01.1970 vergangen sind, übergeben werden.

Beispiel 5.6
Beim Eintragen eines neuen Meetings werden über verschiedene Operatoren die notwendigen Parameter wie Ort und Zeit vom Benutzer erfragt bzw. aus dem Kontext ergänzt. Nach Bestätigung durch den Benutzer trägt der in Abbildung 5.23 dargestellte Operator das neue Meeting in die Wissensbasis ein.

```
<operator name="enter a new meeting" cost="1">
 <description>enter a new meeting to knowledge base</description>
 <precondition>
  <condition variable="statement.name" predicate="unequal" value="" />
  <condition variable="statement.location" predicate="unequal" value="" />
  <condition variable="statement.date" predicate="unequal" value="" />
  <condition variable="statement.time" predicate="unequal" value="" />
  <condition variable="statement.confirmed" predicate="equal" value="true" />
 </precondition>
 <action>
  <say>Adding the new meeting \statement.name\ to my knowledge base.</say>
  <writeStatement />
 </action>
 <effect>
  <condition variable="goal.fulfilled" predicate="equal" value="addMeeting" />
 </effect>
</operator>
```

Abbildung 5.23.: Operator zum Eintragen eines neuen Meetings

Die writeAction *muss lediglich noch die in der* statement *Variablen abgelegten Benutzereingaben in die Variablen name, place und timestamp kopieren und*

---

[3] Einfacheres Format um mit Datumsangaben zu arbeiten, siehe Abschnitt 4.5.

Demo	Demonstration von Projekten
Meeting	Ein Treffen
Presentation	Eine Präsentation
Seminar	Ein Seminarvortrag
Talk	Ein Gespräch

Tabelle 5.3.: Mögliche Ereignisklassen

`knowledgeBase.addEvent('Meeting', name, place, timestamp)` ausführen.

Um auch andere Objekte als Events in die Wissensbasis einzutragen, müssen neue Methoden zur Schnittstelle hinzugefügt werden. Die Implementierung erfolgt analog zur `addEvent()` Methode. Bei einer größeren Anzahl schreibbarer Objekte ist das ständige Anpassen der Schnittstelle zur Wissensbasis allerdings unerwünscht. Abhilfe schafft eine verallgemeinerte Schnittstelle, die die Komplexität in die Operatoren verlagert. Details dazu werden in 7.4.6 vorgestellt.

6. Experimente

Um ein Dialogsystem sinnvoll betreiben zu können, müssen die Bedürfnisse der potentiellen Nutzer in Erfahrung gebracht werden. Zu diesem Zweck wurden im Rahmen dieser Arbeit zwei Experimente durchgeführt. Zunächst stellten wir Pogy im ersten Experiment über eine Weboberfläche zur Verfügung, um Daten über das Benutzerverhalten zu sammeln. Im zweiten Experiment wurde Pogy auf einem humanoiden Roboter installiert und dort im Einsatz getestet. Dort führten wir eine vollständige (End-to-end) Evaluation durch.

6.1. Experiment 1 – Pogy im Internet

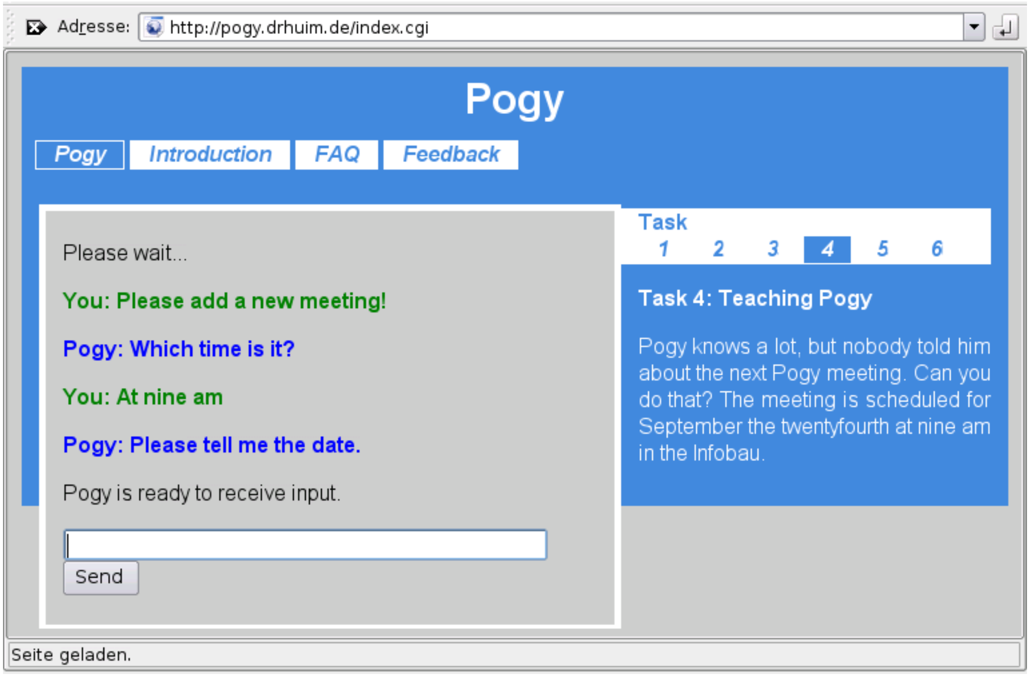
Pogy wurde zur Erweiterung und Anpassung der Eingabegrammatik sowie der Wissensbasis über ein Webinterface erreichbar gemacht. Ziel war die Anpassung des Gesamtsystems an die Bedürfnisse potentieller Benutzer. Hier sollen Aufbau und Ablauf des Experiments beschrieben werden.

6.1.1. Aufbau

Zur Realisation dieser Schnittstelle mussten einige Probleme gelöst werden. Tapas und damit auch Pogy sind für den Einsatz auf einem humanoiden Roboter optimiert. Daher ist das System darauf ausgelegt, zu einem Zeitpunkt mit genau einem Benutzer zu interagieren. Diese Annahme trifft bei menschlichen Kommunikationssituationen zu, nicht jedoch im Internet. Das parallele Starten mehrerer Instanzen von Pogy kam nicht in Frage, da dies den zur Verfügung stehenden Server zu stark ausgelastet hätte. Aus diesem Grund verhinderten wir den gleichzeitigen Zugriff mehrerer Benutzer.

Abbildung 6.1 zeigt einen Screenshot des Webinterfaces, das über `http://pogy.drhuim.de` zu erreichen war.

Pogy ist ein Experte, was die Beantwortung von Fragen zu Projekten und Mitarbeitern des ISL angeht. Um neuen Benutzern den Einstieg zu erleichtern, stellten wir einige Aufgaben (Tasks) bereit. Sie sind in Anhang B abgedruckt. Durch das Bearbeiten der Aufgaben erhielten Benutzer eine schrittweise Einführung in das System. Die Aufgabenstellung war bewusst frei gewählt, um die unterschiedliche Herangehensweise und unterschiedliche Formulierungen der Benutzer nicht einzuschränken. Aus diesem Grund präsentierten wir auch keine Beispieldialoge. Alle Benutzereingaben und Systemantworten wurden gespeichert und ausgewertet. Innerhalb kurzer Zeit erreichten wir so ein deutlich besseres Benutzerverständnis. Die neuen Erkenntnisse flossen in die Eingabegrammatik ein und halfen auch bei der Erweiterung der Wissensbasis.



Please wait...

You: Please add a new meeting!

Pogy: Which time is it?

You: At nine am

Pogy: Please tell me the date.

Pogy is ready to receive input.

Task 4: Teaching Pogy

Pogy knows a lot, but nobody told him about the next Pogy meeting. Can you do that? The meeting is scheduled for September the twentyfourth at nine am in the Infobau.

Abbildung 6.1.: Pogy im Internet

6.1.2. Ablauf

Das Experiment lief vom 04. bis 14. September 2005. Dabei wurden neue Erkenntnisse aus den Mitschnitten durch Anpassung von Grammatik und Wissensbasis zeitnah in Pogy integriert.

6.1.3. Auswertung

Es zeichnete sich sehr schnell ab, dass die initiale Grammatik für die unterschiedlichen Benutzereingaben nicht ausreichend flexibel war. Dies hatte mehrere Gründe. Zum einen wurden Rechtschreibfehler und Sonderzeichen nicht erkannt, was wir durch eine Vorverarbeitung der Eingabe umgingen. Desweiteren lassen sich viele Anfragen durch Floskeln wie ein vorangestelltes oder angehängtes „Please“ variieren. Auch Variationen wie „I want to...“, „I would like to...“ oder Ergänzungen wie „something“, „anything“ etc. wurden häufig verwendet. Um all diese Formulierungen zuzulassen, erweiterten wir die Grammatik stark. Insbesondere ließen wir an vielen Stellen die vorgestellten Floskeln optional zu.

Anfragen von Benutzern beispielsweise nach Telefonnummern veranlassten uns, die Wissensbasis um diese Informationen zu ergänzen. Auf die Integration von nicht verwandten Themen wie „Are you male or female?“ verzichteten wir.

6.2. Experiment 2 – Pogy auf Robbi

Zur Evaluation des Gesamtsystems integrierten wir Pogy in das im SFB 588 Humanoide Roboter - Lernende und kooperierende multimodale Roboter[1] verwendete one4all-System.

6.2.1. Aufbau

Dazu wurde Pogy auf einer humanoiden Roboter-Plattform installiert und in Kombination mit dem in 2.2 erwähnten Spracherkenner Janus (trainiert auf einem Datensatz von Meetings) in Betrieb genommen. Hierzu war es notwendig, die Eingabegrammatik stark einzuschränken, um die Perplexität zu verringern. Zur Sprachaufnahme wurde ein Mikrofon für Distanzaufnahmen verwendet. Auf dem verwendeten System mit 3 GHz CPU und 768 MByte Arbeitsspeicher wird Janus in Echtzeit ausgeführt.

Für die Auswertung wurde ein Evaluationsbogen entworfen. Dieser bestand aus zwei Teilen. Einerseits enthielt er, wie auch schon das erste Experiment, Aufgabenvorschläge, die die Testpersonen lösen konnten. Weiterhin wurden die Benutzer gebeten, nach dem Test eine kurze Einschätzung des Systems abzugeben. Der vollständige Evaluationsbogen ist in Anhang C abgedruckt. Obwohl Pogy nur Englisch versteht, waren die Fragen auf Deutsch verfasst, um die Formulierung der Anfragen möglichst nicht zu beeinflussen.

6.2.2. Ablauf

Die Testgruppe bestand aus sieben Personen, von denen drei Pogy bereits über die Weboberfläche (siehe 6.1) benutzt hatten. Die übrigen vier Testpersonen kannten Pogy nicht. Alle Testpersonen sprachen nacheinander und unabhängig mit Pogy und versuchten die Aufgabenvorschläge zu lösen. Nach dem Dialog sollten sie ihren Eindruck des Systems auf dem Evaluationsbogen angeben.

6.2.3. Auswertung

Die Dialoge der Testpersonen mit dem System wurden transkribiert und ebenso wie die ausgefüllten Evaluationsbögen ausgewertet. Die Ergebnisse werden hier vorgestellt.

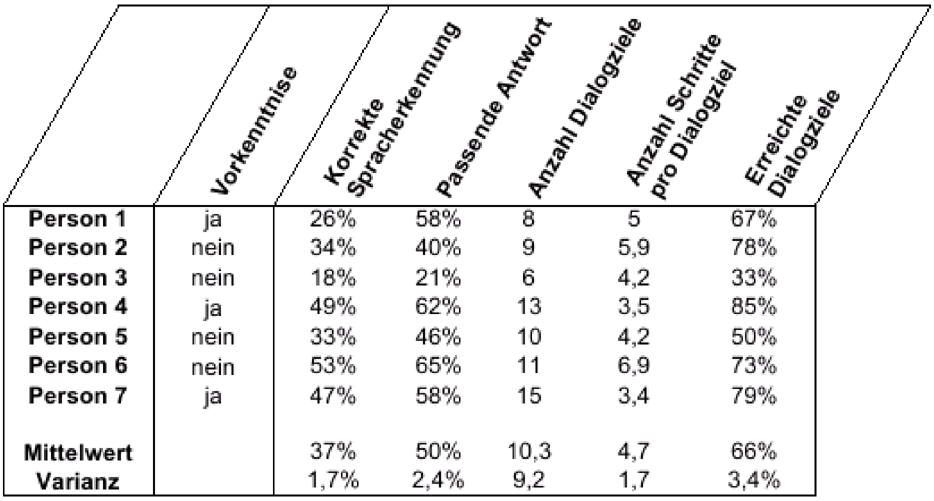
6.2.3.1. Auswertung objektiver Kriterien

Abbildung 6.2 zeigt objektive Auswertungskriterien zum 2. Experiment. Die Einträge in den einzelnen Spalten haben folgende Bedeutung:

Vorkenntnisse Gibt an, ob die Testperson Pogy bereits benutzt hatte.

---

[1] URL: `http://www.sfb588.uni-karlsruhe.de/`



	Vorkenntnise	Korrekte Spracherkennung	Passende Antwort	Anzahl Dialogziele	Anzahl Schritte pro Dialogziel	Erreichte Dialogziele
Person 1	ja	26%	58%	8	5	67%
Person 2	nein	34%	40%	9	5,9	78%
Person 3	nein	18%	21%	6	4,2	33%
Person 4	ja	49%	62%	13	3,5	85%
Person 5	nein	33%	46%	10	4,2	50%
Person 6	nein	53%	65%	11	6,9	73%
Person 7	ja	47%	58%	15	3,4	79%
Mittelwert		37%	50%	10,3	4,7	66%
Varianz		1,7%	2,4%	9,2	1,7	3,4%

Abbildung 6.2.: Objektive Auswertungskriterien zu Experiment 2

Korrekte Spracherkennung Prozentsatz der Äußerungen, die vom Spracherkenner semantisch korrekt erkannt wurden.

Passende Antwort Prozentsatz der Benutzeräußerungen, auf die das System eine passende Antwort lieferte.

Anzahl Dialogziele Anzahl von Dialogzielen, die im gesamten Gespräch aufkamen. Dialogziele sind z. B. Begrüßung oder das Erfragen von Informationen zu einem Projekt.

Anzahl Schritte pro Dialogziel Gibt die durchschnittliche Anzahl von Benutzeräußerungen bis Abbruch oder Erreichen des Dialogziels an.

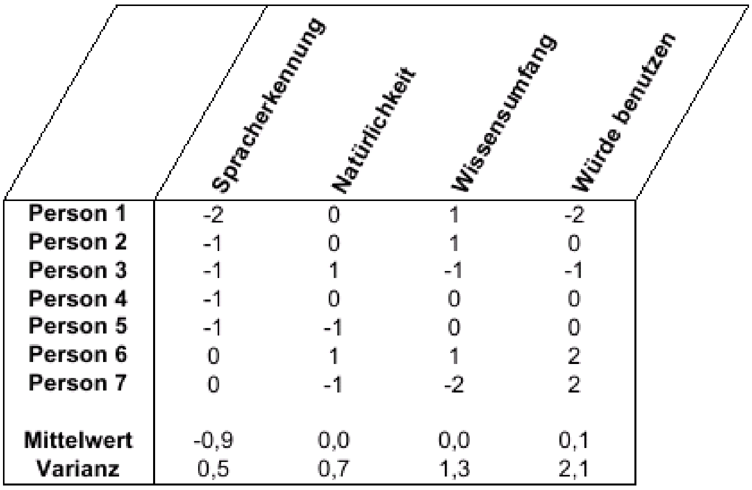
Erreichte Dialogziele Prozentsatz der Dialogziele, die erfolgreich abgeschlossen wurden.

6.2.3.2. Auswertung subjektiver Kriterien

Abbildung 6.3 zeigt subjektive Auswertungskriterien zum 2. Experiment. Als Antwortmöglichkeiten waren Bewertungen von -2 (sehr schlecht) bis +2 (sehr gut) möglich. Die Einträge in den einzelnen Spalten haben folgende Bedeutung:

Spracherkennung Benutzereinschätzung der Güte der Spracherkennung.

Natürlichkeit Bewertung der Natürlichkeit der Interaktion.



	Spracherkennung	Natürlichkeit	Wissensumfang	Würde benutzen
Person 1	-2	0	1	-2
Person 2	-1	0	1	0
Person 3	-1	1	-1	-1
Person 4	-1	0	0	0
Person 5	-1	-1	0	0
Person 6	0	1	1	2
Person 7	0	-1	-2	2
Mittelwert	-0,9	0,0	0,0	0,1
Varianz	0,5	0,7	1,3	2,1

Abbildung 6.3.: Subjektive Auswertungskriterien zu Experiment 2

Wissensumfang Einschätzung des Benutzers über den Wissensumfang des Systems.

Würde benutzen Angabe der Testperson darüber, ob sie das System benutzen würde.

6.2.3.3. Anmerkungen der Testpersonen

Neben den fest vorgegebenen Kriterien gab es die Möglichkeit, freie Kommentare und Verbesserungsvorschläge abzugeben. Diese sind hier zusammengefasst.

Ein Großteil der Benutzer bemängelte die Spracherkennungsleistung. Gerade schnelle Sprache oder sehr lange Sätze würden schlecht verstanden. Dadurch komme oftmals kein richtiger Dialog zustande. Wenn das System die Angaben verstehe, so könne man sich aber sinnvoll unterhalten. Auch die geringe Auswahl an Formulierungen, die das System verstünde, wurde als einschränkend empfunden. Zudem fiel auf, dass das System bei Nichtverstehen recht häufig den Dialog abbreche. Außerdem fehle an einigen Stellen im Dialog die Rückmeldung darüber, was das System verstanden hat.

Die freundliche Interaktion sowie das verständliche Text-To-Speech System wurden positiv bewertet. Allerdings bestand der Wunsch nach einer deutschen Sprachausgabe für die Mensagerichte, da das englischsprachige System diese sehr unverständlich ausgibt. Auch die Fähigkeit des Systems, den Gesprächszusammenhang zu erkennen und so Referenzen auf zuvor erwähnte Dinge und Personen aufzulösen, gefiel den Benutzern.

7. Ergebnisse und Diskussion

In diesem Kapitel stellen wir Ergebnisse der Auswertung der durchgeführten Experimente vor und diskutieren diese.

7.1. Experiment 1 – Pogy im Internet

Durch die eventbasierte Kommunikation von Tapas über Sockets stellte sich die Anbindung an einen Webserver als schwierig heraus. An einigen Stellen mussten Timeouts verwendet werden, um eine Reaktion des Systems zu ermöglichen.

Wie schon in 6.1.3 beschrieben ist die kontextfreie Grammatik nicht für die Texteingabe geeignet. Besser wäre beispielsweise ein ChunkParser, der einzelne Stücke parsen kann, aber nicht die vollständige Eingabe akzeptieren muss. Es mussten sehr umfangreiche Regeln geschrieben werden, um ein akzeptables Verständnis des Systems zu erreichen. Das war zwar aufwändig, aber ansonsten relativ problemlos.

Die Auslagerung des Wissens in externe Datenquellen wie XML/RDF ermöglichte eine einfache Erweiterung des Systems um neues Wissen. Durch die ReloadAktion war es dabei nicht einmal nötig, das System bei der Einbindung von neuem Wissen neu zu starten. Das System zeigte sich sehr robust. Es lief über den gesamten Zeitraum von zehn Tagen stabil.

Insgesamt stellte sich Pogy als flexibel und einfach erweiterbar heraus. Das Hinzufügen von Wissen und neuen Aktionen ist mit geringem Aufwand möglich.

7.2. Experiment 2 – Pogy auf Robbi

In diesem Abschnitt analysieren und bewerten wir die im 2. Experiment gesammelten Daten. Dazu betrachten wir die in Kapitel 6 vorgestellten objektiven und subjektiven Kriterien.

7.2.1. Bewertung objektiver Kriterien

Bei Betrachtung der Daten in Abb 6.2 fällt zunächst auf, dass die Spracherkennung relativ schlecht funktioniert und nur jede zweite oder dritte Äußerung inhaltlich richtig wiedergegeben wurde. Das hat mehrere Gründe: Nur eine

Testperson hatte Erfahrung mit Spracherkennern und demzufolge beeinträchtigten Probleme wie ein zu großer Abstand zum Mikrofon, zu schnelle oder zu langsame Sprechweise und Spontansprache die Erkennungsleistung.

Dennoch liegt der Prozentsatz der passenden Antworten höher, da Pogy oftmals aus dem Kontext Informationen ableiten konnte. Gelegentlich wurden Halbsätze oder Hintergrundgeräusche als Sprache erkannt, was auch daran liegt, das wir auf den Einsatz eines Nahbesprechungsmikrofons verzichteten. Die Information, dass Pogy die Eingabe nicht verstand, wurde ebenfalls als passende Antwort gewertet und verbessert das Ergebnis zusätzlich.

Die durchschnittliche Anzahl der Schritte zum Erreichen eines Dialogziels variiert relativ stark zwischen den Testpersonen.

Es zeigt sich, dass Personen, die das System bereits kannten, im Durchschnitt eine niedrigere Anzahl an Interaktionsschritten zum Erreichen von Dialogzielen benötigten. Diese wurden fast immer erfolgreich abgeschlossen. Auch manche Testpersonen ohne Vorkenntnisse erreichten kleine Werte bei der durchschnittlichen Anzahl Schritte pro Dialogziel. Sie brachen jedoch viele Dialogziele aufgrund schlechter Spracherkennung frühzeitig ab und erreichten so einen geringen Prozentsatz erfolgreich abgeschlossener Dialogziele.

Die beiden Personen mit der höchsten Anzahl an Schritten pro Dialogziel waren nicht mit dem System vertraut. Sie schlossen dennoch etwa den gleichen Prozentsatz an Zielen erfolgreich ab wie die erfahrenen Benutzer.

7.2.2. Bewertung subjektiver Kriterien

Keine Testperson bewertete die Leistung des Spracherkenners als gut oder sehr gut (siehe Abbildung 6.2). Natürlichkeit und Wissensumfang des Systems wurde im Mittel als zufriedenstellend empfunden. Uneinig waren sich die Benutzer darüber, ob sie ein solches System benutzen würden. Zwei würden es sehr gerne benutzen, einer auf keinen Fall. Insgesamt konnten keine Unterschiede in den Antworten der Testpersonen mit und ohne Vorkenntnisse festgestellt werden.

7.3. Vergleich der Experimente

Die Bereitstellung von Pogy im Internet und die Auswertung der daraus resultierenden Dialoge half uns bei der Erweiterung der Wissensbasis und gab interessante Einblicke in das Verhalten potentieller Benutzer. Die größten Änderungen erfuhr jedoch die Eingabegrammatik, um eine möglichst flexible Eingabe von Text zu ermöglichen. Die resultierende Grammatik war allerdings für den Einsatz mit einem Spracherkenner ungeeignet, so dass wir im zweiten Experiment nahezu alle Grammatikänderungen wieder rückgängig machen mussten. Desweiteren setzten Benutzer bei der Eingabe von Text über die Weboberfläche andere Formulierungen ein als bei der natürlichsprachlichen Kommunikation. Insofern war das erste Experiment für die nachfolgende Tests auf einem humanoiden Roboter nur bedingt hilfreich.

7.4. Ausblick

Aus dieser Arbeit geht ein System hervor, das leicht erweitert und auf neue Anforderungen angepasst werden kann. Im Folgenden werden mögliche Veränderungen und Erweiterungen diskutiert.

7.4.1. Erweiterung der Erwartungshaltung

Pogy verwendet bei den meisten Fragen eine Erwartungshaltung, die die erwartete Benutzerantwort eingrenzt (siehe 5.1.4). Janus bietet die Möglichkeit, die Gewichtung im Sprachmodell während der Laufzeit dynamisch an den Kontext anzupassen [FuHW04]. Pogy könnte dies ausnutzen und seine Erwartungshaltung an Janus weitergeben, was eine geringere Fehlerrate in der Spracherkennung erwarten lässt.

7.4.2. Nutzung des Kontexts 2. Art zum Ergreifen der Gesprächsinitiative

An manchen Gesprächspunkten könnte Pogy die Initiative übernehmen, beispielsweise bei unverstandener Eingabe oder nach erfolgreicher Erfüllung eines Dialogziels. Zur Auffindung eines passenden Gesprächsthemas bietet sich der Kontext 2. Art (siehe 2.4.1.2) an. Eine erfolgreiche Anfrage zu einer Person könnte Pogy beispielsweise nutzen, um über den Kontext 2. Art Projekte zu lokalisieren, an denen diese Person beteiligt ist. In einem nächsten Schritt würde Pogy die Gesprächsinitiative übernehmen und anbieten, Informationen zum gefunden Projekt auszugeben.

7.4.3. Verbesserung der Erkennungsleistung des Spracherkenners

Die Fehlerrate des Spracherkenners im zweiten Experiment war relativ hoch. Die Durchführung einer größeren Evaluation könnte Daten zur Verbesserung der Grammatikabdeckung liefern. Noch erfolgsversprechender wären der Einsatz von Nahbesprechungsmikrofonen und die Unterdrückung jeglicher Störgeräusche, was allerdings dem Szenario des Gesprächs mit einem humanoiden Roboters nur bedingt gerecht wird.

7.4.4. Internationalisierung

Pogy kann sich nur auf Englisch unterhalten. Es wäre aber wünschenswert, dass verschiedene Sprachen verstanden und auch in verschiedenen Sprachen geantwortet werden kann.

Um Pogy mehrsprachenfähig zu machen, müssen einige Änderungen durchgeführt werden. Diese beschränken sich auf die Eingabegrammatik sowie auf die

Generierung der Ausgabe, da Pogy selbst auf semantischer Ebene arbeitet. In den beiden folgenden Schritten beschreiben wir eine mögliche Umsetzung.

7.4.4.1. Eingabesprache

Zum Verständnis der Eingabe müssen die entsprechenden Grammatiken in jede zu unterstützende Sprache übersetzt werden. Je nach Verwendung wird desweiteren ein entsprechender Spracherkenner benötigt. Die Funktionalität, mit Spracherkennern in verschiedenen Sprachen und den entsprechenden Grammatiken zu arbeiten, stellt Tapas (siehe [Holz05]) zur Verfügung.

7.4.4.2. Ausgabesprache

Zur Ausgabe in unterschiedlichen Sprachen sind weitere Änderungen erforderlich: Zeichenketten in der Wissensbasis müssen gegebenenfalls in verschiedenen Sprachen gespeichert werden. Einige Methoden wie die Generierung von Listen oder die ReloadAction müssen eine konfigurierbare Ausgabe anbieten. Desweiteren müssen die Texte der möglichen Ausgaben, die sich in der Datei `operators.xml` befinden, in alle gewünschten Ausgabesprachen übersetzt werden.

Die tatsächliche Implementierung der Internationalisierung ließe sich dann auf zwei Arten bewerkstelligen. Zum einen könnte in jedem Operator mit einem zusätzlichen Attribut die Sprache angegeben werden, zum anderen kann die übliche Methode der Internationalisierung verwendet werden. Dazu wird die Rohform des ursprünglichen Textes (üblicherweise englisch) in eine Tabelle eingetragen und in weiteren Spalten die entsprechende Übersetzung in die gewünschten Sprachen. Durch einfaches Suchen des Ausgangstextes in der Tabelle wird die gewünschte Sprachversion nachgeschlagen und ausgegeben. Tapas bietet Funktionen zur mehrsprachigen Ausgabe an [Holz05], so dass die Implementation relativ einfach sein sollte.

7.4.5. Änderungen an der Wissensbasis

Hier werden mögliche Erweiterungen an der Wissensbasis diskutiert, die die Entwicklung und Wartung von Pogy vereinfachen.

7.4.5.1. Konsistente Wissensbasen

In der aktuellen Arbeit sind zwei Wissensbasen mit unterschiedlichen Ontologien vorhanden. Das ist zum einen die Wissensbasis, die für die Generierung der Eingabegrammatik nötig ist, im Folgenden Eingangswissensbasis genannt, und zum anderen die Wissensbasis, die Pogy zur Planung und Antwortgenerierung benutzt, im folgenden Hauptwissensbasis genannt. Diese Aufteilung bringt einige Probleme mit sich. An einer Stelle im Programm muss ein Abgleich beider Ontologien vorgenommen werden. Zudem muss nach einer größeren Änderung der Hauptwissensbasis die Eingangswissensbasis angepasst werden.

Dieses Problem kann auf zwei Arten gelöst werden. Entweder könnte die Eingangswissensbasis aus der Hauptwissensbasis generiert oder das System auf eine zentrale Wissensbasis umgestellt werden.

Export der Hauptwissensbasis Hierzu muss die Hauptwissensbasis in das entsprechende Format der Eingangswissensbasis umgewandelt werden. Ein erster Schritt in diese Richtung ist der bereits erwähnte `DatabaseBuilder` (s. 2.8.3). Dieses Tool ist in der Lage, einzelne Instanzen von Klassen in die Datei `database.txt` zu exportieren. Diese Datenbank kann von Tapas eingelesen werden. Es müsste außerdem ein Werkzeug geschrieben werden, das in der Lage ist, die Ontologie der Hauptwissensbasis in die Eingangswissensbasis zu überführen. Das bedeutet, es muss die benötigten Teile der Klassenhierarchie in das `.adl2` Dateiformat exportieren können. Auf diese Art wäre es einfacher, die Eingangswissensbasis konsistent mit der Hauptwissensbasis zu halten.

Zentrale Wissensbasis Eine andere Lösung stellt eine zentrale Wissensbasis dar. Eine zu definierende Schnittstelle könnte eine Liste aller Individuen einer Klasse liefern. Damit wäre keine Eingabeontologie mehr nötig, und man könnte beim Schreiben der Grammatiken, die ja für einen Spracherkenner unerlässlich sind, einfach die Klassennamen aus der Wissensbasis benutzen. Dies würde das Schreiben von Grammatiken stark vereinfachen und eine Eingangsontologie überflüssig machen.

7.4.5.2. Persistenz

Jena kann auch in einem persistenten Modus betrieben werden (siehe hierzu auch 2.7.3). Dazu ist eine Datenbank wie beispielsweise MySQL[1] nötig. Weitere Änderungen sind nicht nötig, lediglich die Betriebsart von Jena muss umgestellt werden. Auf diese Weise würde neu erworbenes Wissen erhalten bleiben.

7.4.6. Erweiterung der Schnittstelle zum Ändern der Wissensbasis

Die derzeitige Schnittstelle erlaubt beliebige RDQL Abfragen und ist damit sehr flexibel. Änderungen an der Wissensbasis können allerdings nicht per RDQL vorgenommen werden, da es sich hierbei um eine reine Abfragesprache handelt. Die Schnittstelle zur Wissensbasis soll von den Jena eigenen Methoden abstrahieren. Dazu sind zwei unterschiedliche Herangehensweisen denkbar.

Eine Möglichkeit ist, zum Eintragen verschiedener Objekttypen jeweils eigene Methoden anzubieten. Auf diese Weise wird derzeit das Eintragen von Events realisiert (siehe 5.3.3). Diese Vorgehensweise hat den Nachteil, dass die Schnittstelle für jeden neuen schreibbaren Typen verändert werden muss.

Das ständige Anpassen der Schnittstelle würde verhindert werden, wenn man Methoden zum Anlegen von Individuen und Hinzufügen ihrer Eigenschaften

---

[1] URL: `http://dev.mysql.com/`

in der Schnittstelle anbietet. Dadurch wird die Komplexität in die Operatoren verlagert und die Schnittstelle müsste nur das Erzeugen einer eindeutigen URI übernehmen. Abbildung 7.1 zeigt einen möglichen Operator, der das Eintragen einer neuen Person erlaubt. Neben der geänderten Schnittstelle und diesem Operator müsste eine *create* Aktion implementiert werden, die die notwendigen Schnittstellenaufrufe zum Erzeugen eines Individuums und Setzen seiner Eigenschaften durchführt.

```
<operator name="add a person" cost="1">
 <precondition>
  <condition variable="statement.name" predicate="unequal" value="" />
 </precondition>
 <action>
  <create type="Person">
   <property type="Name" value="var:statement.name" />
   <property type="Gender" value="male" />
  </create>
 </action>
 <effect>
  <condition variable="goal.fulfilled" predicate="equal" value="addPerson" />
 </effect>
</operator>
```

Abbildung 7.1.: Operator zum Eintragen einer Person in die Wissensbasis

7.4.7. Wissensbasisanfragen mit Operatoren

In Pogy werden nötige Wissensbasisanfragen bereits bei der Zielfindung definiert. Das System würde durch eine Aufteilung dieser Anfragen noch flexibler. Die einzelnen Anfragen könnten in Operatoren mit entsprechenden Vor- und Nachbedingungen verankert werden. Eine neue Aktion wäre nötig, mit deren Hilfe es möglich ist, eine RDQL-Anfrage zu formulieren. Abbildung 7.2 zeigt eine mögliche Syntax. Eingebettet in einen eigenen Operator würde Pogy die Anfrage bei Bedarf automatisch einplanen.

```
<lookup result="variablenpfad">
    SELECT ?RESULT WHERE (<http://pogy.dlg.isl.ira.uka.de/labinfo#PersonUri>,
    <http://pogy.dlg.isl.ira.uka.de/labinfo#Name>, ?RESULT)
</lookup>
```

Abbildung 7.2.: Mögliche Aktion zur Abfrage der Wissensbasis

7.4.8. Verschiedene Ausgabemodalitäten

Je nach Einsatzumgebung und Aufgabenbereich von Pogy können weitere Ausgabemodalitäten von Nutzen sein. Soll Pogy beispielsweise Wegbeschreibungen zu Räumen oder Lokalitäten liefern, könnten diese als Karte auf einem Monitor angezeigt oder auf einem Drucker ausgegeben werden. Auch eine Ansteuerung von Extremitäten wäre denkbar, um beispielsweise auf Gegenstände zeigen zu können.

7.4.9. Selbstauskunft

Zu Wartungszwecken könnten weitere Variablen entwickelt werden, die Systeminformationen über die verwendete Hardware enthalten. Stehen solche Variablen auch in der Planung zur Verfügung, kann Pogy anhand seines Systemzustandes andere Entscheidungen treffen. Dadurch wäre es beispielsweise möglich, die beste Modalität der Ausgabe (Sprache, Gestik) automatisch in Abhängigkeit von der zur Verfügung stehenden Hardware auszuwählen.

7.4.10. Mehrere Benutzer

Pogy ist darauf ausgelegt, mit nur einem Benutzer zur gleichen Zeit zu kommunizieren. Desweiteren wird nicht zwischen verschiedenen Benutzern unterschieden. Eine Mehrbenutzerunterstützung ist bislang nicht möglich, da das vorhandene System keine Möglichkeit zur Benutzeridentifikation besitzt und durch einen Sprachdialog keine Identifikation möglich ist.

Es interessieren uns insbesondere zwei Methoden der Mehrbenutzerinteraktion: Kooperative Umgebungen und die gleichzeitige, aber unabhängige Nutzung.

7.4.10.1. Unabhängige Nutzung des Systems

In diesem Szenario benutzen mehrere Nutzer das System gleichzeitig, aber unabhängig voneinander. Um dies zu unterstützen müssten Variablenbaum, Kontext und Fokusbereiche getrennt für die Benutzer verwaltet werden. Gleiches gilt für die erstellten Pläne.

7.4.10.2. Kooperative Nutzung des Systems

In einer kooperativen Umgebung verfolgen mehrere Benutzer das gleiche Dialogziel und arbeiten gemeinsam an einer Lösung. Dazu ist es nötig, dass mehrere Benutzer gleichzeitig verwaltet werden können. Im Gegensatz zur unabhängigen, gleichzeitigen Nutzung des Systems wären bei der kooperativen Nutzung Teile des Variablenbaums und Kontext weiterhin nur einmal vorhanden. Fokusbereiche müssten für jeden Benutzer angelegt werden.

7.4.10.3. Rechtemanagement

Eine Unterstützung von mehreren Benutzern erlaubt auch die Einführung von Berechtigungen. Beispielsweise wäre es sinnvoll, das Schreiben in die Wissensbasis (z. B. durch Eintragen neuer Meetings) nur einem bestimmten Benutzerkreis zu erlauben. Eine solche Rechteverwaltung ist bereits in Pogy integriert.

7.4.10.4. Benutzeradaption

Eine Erweiterung des Benutzermodells würde eine Adaption auf den jeweiligen Benutzer ermöglichen. Dazu gehört das Speichern persönlicher Daten wie Name oder Geburtstag. Weiter wäre es möglich, zu Beginn eines Gesprächs alte Gespräche mit diesem Benutzer in den Kontext zu laden. Ein Erkennen von Benutzervorlieben würde einen impliziten Gesprächskontext vorgeben. Beispielsweise könnte die Anfrage „Which food is there today?“ zu „Which food is there today in the mensa on line six?“ aufgelöst werden, wenn eine entsprechende Benutzervorliebe bekannt ist.

7.4.11. Unterstützung von Subdialogen

Derzeit führt das Erkennen eines neuen Dialogziels dazu, dass das alte Ziel verworfen wird. Handelt es sich nur um kurze Klärungsfragen, dann ist eine Rückkehr zum alten Dialogziel gewünscht. Die Verwendung eines Goal-Stacks[2] würde diese Subdialoge erlauben.

7.4.12. Editor für Operatoren

Mit wachsendem Dialogwissen steigt auch die Zahl der verwendeten Operatoren. Da das manuelle Editieren der Operatoren leicht unübersichtlich und damit fehleranfällig wird, wäre ein graphischer Editor hilfreich.

---

[2] Ein Stack (Keller), auf den Ziele temporär abgelegt werden können, um sie bei Bedarf später wieder aufzunehmen.

8. Zusammenfassung

In dieser Arbeit haben wir Entwurf und Entwicklung eines planbasierten Dialogmodells für Informationssysteme vorgestellt.

Es ist uns gelungen, ein System zu entwickeln, das kleine Bausteine (Planungsoperatoren) intelligent verknüpft und daraus den Weg zu einem komplexen Ziel (Dialogziel) konstruiert. Durch optimistische Planung wird der kürzeste Weg gefunden.

Benutzern gefiel der natürliche Umgang mit dem System und dessen Fähigkeit, Aussagen im Kontext zu verstehen. Dies erreichten wir durch verschiedene Kontextmodelle und darauf operierenden Funktionen. Dadurch ist das System in der Lage, Äußerungen sowohl für sich als auch im Gesprächsverlauf zu interpretieren.

Die sorgfältige Modellierung des Wissens in einer isolierten Wissensbasis sorgt für eine saubere Trennung des Weltwissens von der Implementierung und legt damit die Grundlage für einen einfachen Transfer des Systems in eine neue Domäne. Durch die Verwendung des allgemein anerkannten RDF Standards ist zudem Zukunftssicherheit und Erweiterbarkeit gegeben, ebenso wie die Möglichkeit, Daten mit bewährten Werkzeugen verarbeiten zu können.

Die entworfene Architektur stellte sich als robust und flexibel heraus. Erweiterungen können leicht eingepflegt werden und dank semantischer Repräsentation des Wissens ist eine einfache Erweiterung z. B. auf Mehrsprachenfähigkeit garantiert.

A. Einrichten von Pogy

Um Pogy einzurichten, sind einige Einstellungen nötig. In diesem Anhang soll erklärt werden, wie Pogy in der Entwicklungsumgebung Eclipse eingerichtet wird und wie Pogy auf einem System direkt ausgeführt werden kann.

A.1. Pogy in Eclipse

Pogy wurde mit Hilfe von Eclipse entwickelt, deshalb wird die Einrichtung in Eclipse an dieser Stelle beschrieben.

A.1.1. Benötigte Bibliotheken

Abbildung A.1 listet die von Pogy benötigten Bibliotheken auf. Bei der Einrichtung ist zu beachten, dass `tools.jar` vor `tapas-1.2.0.jar` geladen wird. Abbildung A.2 zeigt das entsprechende Dialogfenster unter Eclipse.

```
lib/commons-logging.jar
lib/concurrent.jar
lib/icu4.jar
lib/jena.jar
log4j.jar
lib/tapas-1.2.0.jar
tools.jar
lib/xecesImpl.jar
junit.jar
JRE System Library [j2sdk1.5-sun]
```

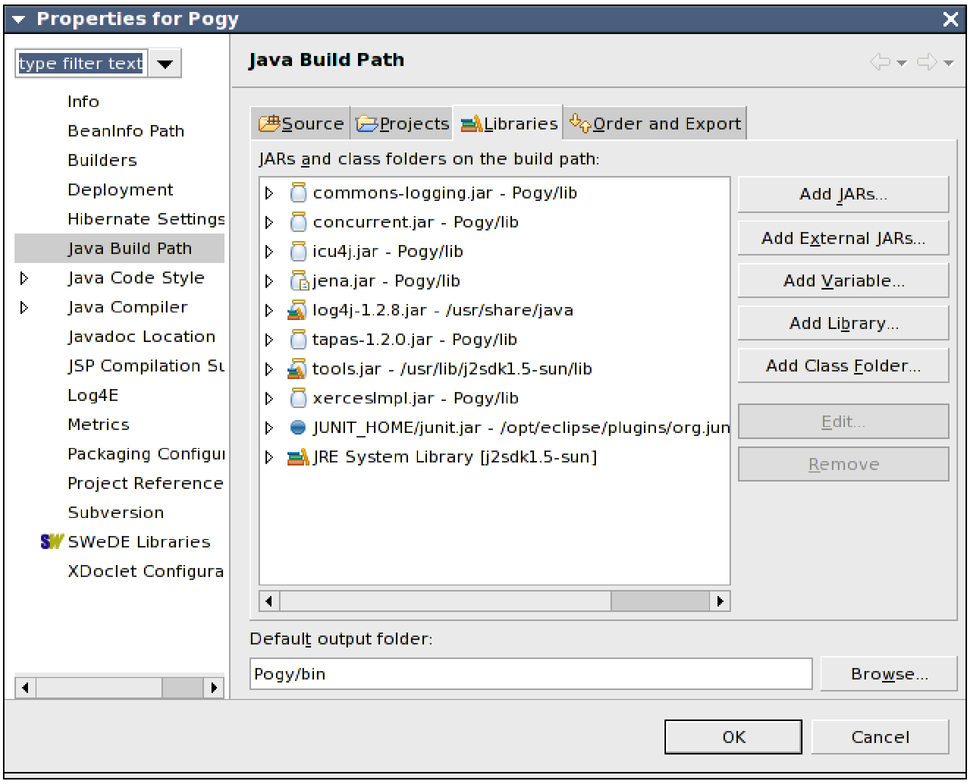
Abbildung A.1.: Benötigte Bibliotheken

A.1.2. Pogy Run-Konfiguration

In der Run-Konfiguration von Pogy müssen die in A.3 aufgelisteten Einstellungen vorgenommen werden.

A.2. Pogy Standalone

Um Pogy auf einem System ohne Eclipse zu starten, kann unter Linux das Shellskript `pogy.sh` verwendet werden.



JARs and class folders on the build path:

- commons-logging.jar - Pogy/lib
- concurrent.jar - Pogy/lib
- icu4j.jar - Pogy/lib
- jena.jar - Pogy/lib
- log4j-1.2.8.jar - /usr/share/java
- tapas-1.2.0.jar - Pogy/lib
- tools.jar - /usr/lib/j2sdk1.5-sun/lib
- xercesImpl.jar - Pogy/lib
- JUNIT_HOME/junit.jar - /opt/eclipse/plugins/org.jun
- JRE System Library [j2sdk1.5-sun]

Default output folder: Pogy/bin

Abbildung A.2.: Eclipse Einstellung der benötigten Bibliotheken

```
Program arguments:
  -conf ${workspace_loc:Pogy}/pogy.properties -gui
VM arguments
  -Djava.library.path=${workspace_loc:Pogy}/lib/
  -Dfile.encoding=UTF8
Classpath
  /data/
  /data/protege/
  /lib/ext/
```

Abbildung A.3.: Pogy Run Konfiguration

B. Aufgaben aus Experiment 1

Die Aufgaben aus dem Experiment 1:

B.1. Task 1 – Find a list of research projects

There are lots of different projects at ISL. Ask Pogy about them!

B.2. Task 2 – Project responsibilities

Did you learn about the FAME project already? Try to find out who is the contact person for it.

B.3. Task 3 – Contact details

That CHIL projects looks interesting, doesn't it? Suppose you want to write an email to it's contact person, what would be the address?

B.4. Task 4 – Teaching Pogy

Pogy knows a lot, but nobody told him about the next Pogy meeting. Can you do that? The meeting is scheduled for September the twentyfourth at nine am in the Infobau.

B.5. Task 5 – Getting hungry

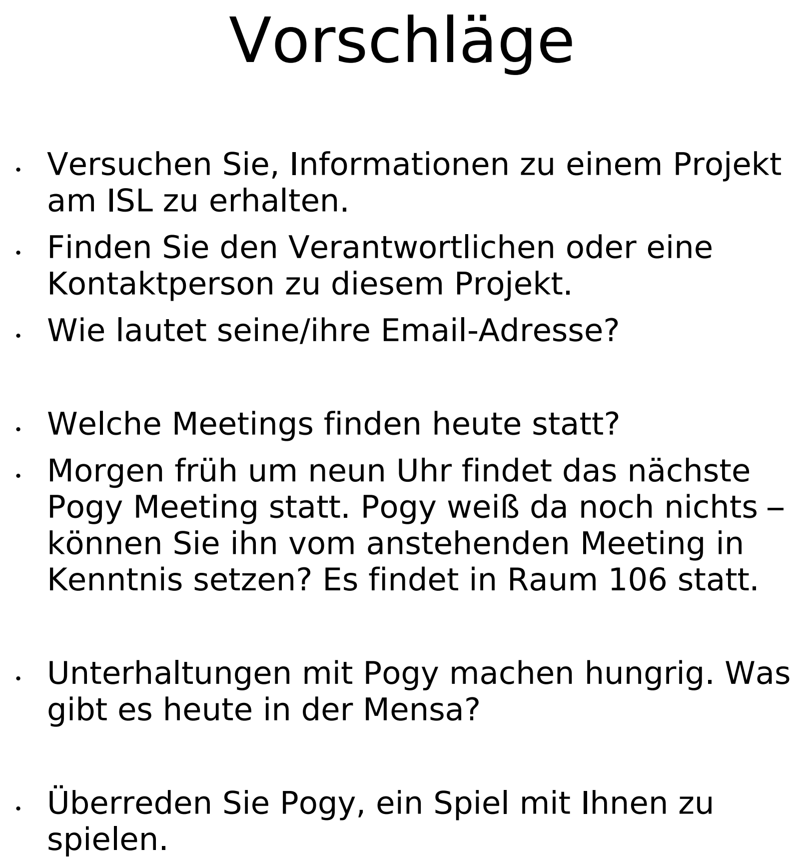
The Karlsruher Mensa offers cheap food (literally). Make Pogy look up today's food for you!

B.6. Task 6 – Feeling bored

Time for some recreationals? Ask Pogy to play a game with you!

C. Evaluationsbogen

Auf den folgenden beiden Seiten ist der Evaluationsbogen für das Experiment 2 (siehe 6.2) abgebildet.



Vorschläge

- Versuchen Sie, Informationen zu einem Projekt am ISL zu erhalten.
- Finden Sie den Verantwortlichen oder eine Kontaktperson zu diesem Projekt.
- Wie lautet seine/ihre Email-Adresse?

- Welche Meetings finden heute statt?
- Morgen früh um neun Uhr findet das nächste Pogy Meeting statt. Pogy weiß da noch nichts – können Sie ihn vom anstehenden Meeting in Kenntnis setzen? Es findet in Raum 106 statt.

- Unterhaltungen mit Pogy machen hungrig. Was gibt es heute in der Mensa?

- Überreden Sie Pogy, ein Spiel mit Ihnen zu spielen.

Abbildung C.1.: Evaluationsbogen Vorderseite

Evaluationsbogen - Pogy

Frage	**nein**	**ja**
Haben Sie Pogy bereits im Internet getestet?	○	○

Frage	**--**	**-**	**~**	**+**	**++**
Wie gut funktioniert die Spracherkennung?	○	○	○	○	○
Wie natürlich empfinden Sie die Interaktion?	○	○	○	○	○
Wie umfangreich ist das Wissen des Systems?	○	○	○	○	○
Würden Sie das System benutzen?	○	○	○	○	○

Was gefällt Ihnen, was nicht?

Kommentare, Verbesserungsvorschläge

BITTE NICHT AUSFÜLLEN	# erkannter Phrasen	korrekt		fehlerhaft	
	# parsbarer Phrasen bei korrekter Erkennung	korrekt		fehlerhaft	
	# passender Antworten	passend		unpassend	
	# erkannter Intentionen	erkannt		nicht erkannt	
	# Dialogziele	erreicht		abgebrochen	

BITTE NICHT AUSFÜLLEN	# Schritte / Dialogziel	Erfolg	Abbruch

Abbildung C.2.: Evaluationsbogen Rückseite

D. Beispieldialog

In diesem Beispieldialog steht steht *U* für den Benutzer und *P* für die Antwort von Pogy.

U Hi Pogy!

P Hi user!

U Please tell me something about ISL.

P The scientific focus of the Interactive Systems Laboratories (ISL) is on technologies that facilitate the human experience, human mutual understanding, and communication. Examples of such technologies are translation, speech, language, vision technologies, multimodal and cross-modal perceptual interfaces, smart rooms, or pervasive computing.

U What can you do for me?

P Let me outline some of the things I can do for you. I am an information system. You can ask me for events like meetings or demos. I know research projects and even some student projects. That's by far not everything, for example I'm even able to remember new meetings if you tell me about them.

U Which projects do you know?

P Oh, there are 22 results. So I will tell you just a few. There are the following Projects Humanoid Robot, Tapas, Steem and CHIL.

U Who is responsible for the CHIL Project?

P Rainer Stiefelhagen is responsible for this.

U Who is he?

P Rainer Stiefelhagen is working at professor Waibel's Interactive Systems Laboratories. His research interests include focus of attention and tracking of facial features and eye gaze.

U What is his phone number?

P The phone number is +49 721 608 6385.

U And what is his email address?

P The email address is stiefel@ira.uka.de

U Who is Hartwig Holzapfel?

P Hartwig Holzapfel is a PhD Student at the University of Karlsruhe, working in the field of dialogue management. His research interests include cognitive architectures for dialogue systems to enable different kind of learning tasks in the context of multilingual and multimodal systems. At the moment Hartwig is a visiting researcher at Carnegie Mellon.

U What do you know about TAPAS?

P TAPAS is a multimodal and multilingual dialogue manager. It provides a framework to easily create dialogue applications. Pogy is using a part of this system.

U Where is the office of Alex Waibel?

P The office is 113.

U What is his email address?

P The email address is waibel@cs.cmu.edu

U What is his phone number?

P The phone number is +49 721 608 4730.

U Which meetings are there?

P There are the following Events yesterday talk, Pogy, Pogy Demo and Pogy Release.

U Which meeting is today?

P I know the Pogy Release Event.

U Who is Petra Gieselmann?

P Her research interests are in dialogue systems for natural language interaction, speech understanding, human-robot-interaction and learning systems.

U There will be a meeting.

P Which time is it?

U At twelve o'clock.

P What kind of meeting is it?

U It is a pogy meeting.

P Please tell me the date.

U Today

P Ok, where does the pogy take place?

U In the mensa.

P Please confirm the following: Add the pogy meeting which takes place at mensa to the knowledgebase. The meeting takes place at today at 12:00 AM. Is everything right?

U Yes

P Adding the new meeting pogy to my knowledge base.

U Which meeting is today?

P There are the following Events pogy and Pogy Release.

U What does the mensa serve today?

P There are seven lines in the mensa. Which one are you interested in?

U Line three

P Please tell me the date.

U today

P Line 3 serves Gnocchipfanne Provencale mit mediterranem Gemüse.

U I want to play a game.

P Okay, let's play a game: I'm thinking of a number between 1 and 99. It is... oh no, you have to guess it!

U Is it fifty?

P No sorry, my number is smaller than 50. Guess again.

U Is it twentyfive?

P No sorry, my number is bigger than 25. Guess again.

U Is it thirtyone?

P No sorry, my number is bigger than 31. Guess again.

U Is it fourty

P No sorry, my number is smaller than 40. Guess again.

U It must be thirtysix

P No sorry, my number is smaller than 36. Guess again.

U So it must be thirtyfour.

P Yeah, that's it! It's 34!

U Bye pogy.

P Bye, bye! Have a nice day!

Literatur

[ASFH+94] James F. Allen, Lenhart K. Schubert, George Ferguson, Peter Heeman, Chung Hee Hwang, Tsuneaki Kato, Marc Light, Nathaniel G. Martin, Bradford W. Miller, Massimo Poesio und David R. Traum. The TRAINS project: A case study in building a conversational planning agent. Technischer Bericht, Computer Science Dept, University of Rochester, September 1994.

[Carp92] B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press. 1992.

[DeAb99] Anind K. Dey und Gregory D. Abowd. Towards a Better Understanding of Context and Context-Awareness. *2000 Conference on Human Factors in Computing Systems*, 1999.

[FeAl98] George Ferguson und James F. Allen. TRIPS: An Integrated Intelligent Problem-Solving Assistant. In *Proceedings of the Fifteenth National Conference on AI (AAAI-98)*, Juli 1998.

[FGHK+97] M. Finke, P. Geutner, H. Hild, T. Kemp, K. Ries und M. Westphal. The Karlsruhe-Verbmobil Speech Recognition Engine. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP-97*, Munich, Germany, 1997.

[FuHW04] Christian Fuegen, Hartwig Holzapfel und Alex Waibel. Tight Coupling of Speech Recognition and Dialog Management - Dialog-Context Grammar Weighting for Speech Recognition. In *Proceedings of the International Conference on Spoken Language Processing, ICSLP 2004*, 2004.

[GBFK+03] Günther Görz, Kerstin Bücher, Yves Forkl, Martin Klarner und Bernd Ludwig. Speech Dialogue Systems A „Pragmatics-First“ Approach to Rational Interaction. In *DFG Workshop on Theoretical Computer Science*, Oktober 2003.

[Gros86] Barbara J. Grosz. Attention, Intentions, and the Structure of Discourse. *Computational Linguistics* 12(3), July-September 1986.

[Holz05] Hartwig Holzapfel. Towards Development of Multilingual Spoken Dialogue Systems. In *Proceedings of the 2nd Language and Technology Conference*, 2005.

[Lacy05] Lee W. Lacy. *OWL: Representing Information Using the Web Ontology Language*. Trafford Publishing. 2005.

[McTe02] Michael F. McTear. Spoken Dialogue Technology: Enabling the Conversational User Interface. *ACM Computing Surveys* 34(1), 1 März 2002, S. 90–169.

[Powe03] Shelly Powers. *Practical RDF Solving Problems with the Resource Description Framework.* O'Reilly. Juli 2003.

[RuNo03] Stuart Russell und Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ. 2. Auflage, 2003.

[SMFW01] H. Soltau, F. Metze, C. Fuegen und A. Waibel. A One pass-Decoder based on Polymorphic Linguistic Context Assignment. In *Proceedings of the Automatic Speech Recognition and Understanding Workshop, ASRU-2001*, Madonna di Campiglio, Trento, Italy, December 2001.

[SmHi94] R. Smith und D.R. Hipp. *Spoken Natural Language Dialog Systems: A Practical Approach.* Oxford University Press. 1994.

[Weiz66] Joseph Weizenbaum. ELIZA–A Computer Program For the Study of Natural Language Communication Between Man and Machine. *Communications of the ACM* 9(1), Januar 1966, S. 35–36.

[WoJe95] Michael Wooldridge und Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review* 10(2), 1995, S. 115–152.

Index