Spezifikation eines Dialogsystems für den Multimediaraum unter Verwendung des Dialogmanagers „ariadne“

Studienarbeit

Elin Anna Topp

Betreuer:
Dipl.–Inform. C. Fügen · Dipl.–Inform. M. Denecke
Prof. Dr. A. Waibel

Institut für Logik, Komplexität und Deduktionssysteme

Universität Karlsruhe, 4. Juli 2002

Vorwort

Diese Studienarbeit ist in der Zeit von Juni 2001 bis Juni 2002 am Institut für Logik, Komplexität und Deduktionssysteme der Fakultät für Informatik, Universität Karlsruhe entstanden. Sie stützt sich auf ein von Matthias Denecke an der Carnegie Mellon University in Pittsburgh, USA, entwickeltes Dialogmanagementsystem und beschäftigt sich mit der Spezifikation eines Dialogsystems zur Steuerung einer multimedial ausgestatteten Räumlichkeit. Als Beispiel stand dabei im Hintergrund der Multimediahörsaal der Fakultät für Informatik, der mittlerweile viele verschiedene miteinander agierende Geräte zur Verfügung stellt, so die miteinander agieren, daß eine dialoggeführte Steuerung die Arbeit in diesem Hörsaal sehr vereinfachen würde.

Ich möchte an dieser Stelle meinen Betreuern Christian Fügen und Matthias Denecke sehr herzlich für die von ihnen immer wieder gezeigte Hilfsbereitschaft danken, wenn ich mich einem scheinbar unlösbaren Problem gegenüber sah.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Wenn Du glaubst, Du hättest das Licht eingeschaltet, fährt sicher die Jalousie runter!

(Frei nach „Murphy's Law")

1.1 Die Aufgabenstellung

Wohin man sich auch dreht und wendet, ist man umgeben von Geräten aller Art, deren Bedienerführung angeblich selbsterklärend ist. Für einzelne Geräte wie Lampen, Videorecorder und -projektoren mag das zutreffend sein. Interessant wird es jedoch dann, wenn viele solche Geräte in einem Raum vorhanden sind, beispielsweise in sogenannten Multimediasälen. Bisher verlangt die Steuerung des in der vorliegenden Arbeit als Beispiel herangezogenen Multimediahörsaales der Fakultät für Informatik der Universität Karlsruhe von der Benutzerin, daß sie sich mit der „Sprache" des Raumes auseinandersetzt. Zum Beispiel sollen in einer Vorlesung bestimmte Lampen ein- oder ausgeschaltet, evtl. die Verdunkelung heruntergelassen und bei Bedarf einer (oder mehrere) der Projektoren eingeschaltet werden, um die Ausgabe des mitgebrachten Laptops oder die des im Raum befindlichen Computers an die Wand zu projizieren. Dazu müssen eine Schalttafel (für Lampen und Verdunkelung) und ein menügestütztes Steuerpult (Touchscreen) für die Einstellungen der Projektion benutzt werden. Für die Vortragende wäre es deutlich angenehmer, einem Steuerungssystem natürlichsprachlich mitteilen zu können, welche Geräte sie verwenden möchte. Hier setzt nun die Arbeit an, in der versucht wurde, ein Beispielsystem zu erstellen, das die Möglichkeit bietet, einem „intelligenten Raum" möglichst frei formulierte Befehle geben zu können.

Zu diesem Zweck sollte mit Hilfe des Dialogmanagers „ariadne" ein Dialogsystem erstellt werden, das Anfragen einer Benutzerin verarbeitet und, sofern Klarheit über die auszuführende Aufgabe herrscht, an eine Applikation weitergibt, die einen solchen „intelligenten Raum" simuliert. Eine direkte Anbindung an die Steuerung des

vorliegenden Raumes war nicht vorgesehen, lediglich die als Beispiel herangezogene Zusammenstellung der Geräte entstammt dem konkreten Raum. Da die hier angenommenen Anfragen natürlichsprachlich formuliert werden sollten, muß das System in der Lage sein, entsprechende Klärungsfragen zu stellen, um sich so an die Absicht der Benutzerin herantasten zu können. Die vorliegende Arbeit sollte als Studie zum Einsatz des Dialogmanagers „ariadne" verstanden werden, deren Ziel eher die Entwicklung einer sinnvollen Herangehensweise als die Erstellung eines uneingeschränkt einsatzbereiten Systems war. Die Studienarbeit umfaßt die Erstellung des Dialogsystems und die Implementierung der genannten Applikation (in Java). Abbildung 1.1 zeigt den Gesamtaufbau des Systems.

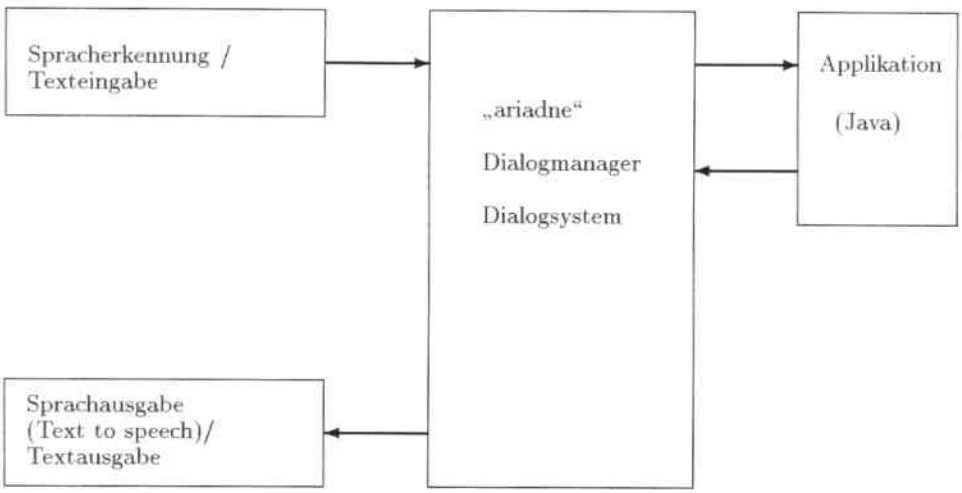

Abbildung 1.1: Der Dialogmanager als Vermittler

1.2 Dialogsystem / Dialogmanager?

An dieser Stelle soll zunächst eine Unterscheidung zwischen dem im Rahmen der Studienarbeit erstellten Dialogsystem und dem zugrunde liegenden Dialogmanager „ariadne" gemacht werden. Grundsätzlich muß für einen maschinellen Dialog unterschieden werden zwischen domänenspezifischem Wissen („worauf bezieht sich der mögliche Dialog?") und dem Wissen über die zu verwendenden Interaktionsmuster („wie soll der Dialog geführt werden?"). An dieser Stelle kann insofern eine starke Trennung erfolgen, da die Interaktionsmuster eines Systems vom eigentlichen Inhalt eines Dialogs völlig unabhängig sind. Diese Interaktionsmuster werden vom Dialogmanager zur Verfügung gestellt und können damit für die eigentlichen Dialogsysteme verwendet werden. Hiermit wird die Erstellung eines Systems sehr stark vereinfacht, da nicht für jedes einzelne, eine bestimmte Domäne abdeckende System definiert werden muß, wie der Dialog zu führen ist. Die zugrundeliegende Idee einer Entwicklungsumgebung für natürlichsprachliche Dialogsysteme wird in [1] vorgestellt.

Das hier untersuchte Dialogsystem stellt also lediglich das domänenspezifische Wissen bereit, das sich wie folgt untergliedern läßt in

1. eine Ontologie, die die Zusammenhänge der in der Domäne „Multimediaraum" auftretenden Objekte und Aktionen herstellt, angegeben durch das Domänenmodell,

2. eine Reihe von Dialogzielen, die, sobald sie hinreichend genau beschrieben sind, der Applikation für weitere Aktionen benötigte Parameter liefern, bzw. Datenbankanfragen auslösen,

3. eine Grammatik, die es dem System ermöglicht, die Anfragen der Benutzerin auf ihren Zusammenhang mit den vorgegebenen Dialogzielen zu prüfen,

4. eine Beschreibung der Datenbank, um eine Verknüpfung der in Applikation und Dialogsystem verwendeten Objekte und Eigenschaften herstellen zu können, so wie

5. eine Liste von Schablonen (Templates) die dem Dialogmanager (abhängig vom Zustand des Dialoges) die Möglichkeit zur Bildung domänenspezifischer Klärungsfragen geben.

1.3 Aufbau der Ausarbeitung

Die vorliegende Ausarbeitung umfaßt insgesamt 5 Kapitel. Im folgenden Kapitel 2 werden, soweit zum Verständnis notwendig, der Aufbau von „ariadne" und die grundsätzliche Vorgehensweise zur Erstellung eines Dialogsystems erläutert. Weiterhin wird in Kapitel 3 das im Rahmen der Studienarbeit erstellte System beschrieben, Kapitel 4 berichtet über die Ergebnisse der Arbeit und Probleme bei der Implementierung. Kapitel 5 liefert dann eine Zusammenfassung sowie Anknüpfungspunkte für Verbesserungen und zukünftige Arbeiten.

Kapitel 2

Spezifikation eines Dialogsystems mit „ariadne“

Dein Wunsch sei mir Befehl!

(ALLG. GEBR.)

Das folgende Kapitel soll die Grundprinzipien des Dialogmanagers so weit erläutern, wie es zum Verständnis des später beschriebenen Systems notwendig erscheint. Es hat somit nicht den Anspruch, ein vollständiges Handbuch zu sein.

2.1 Das Zusammenspiel einzelner Komponenten

Ein Dialogsystem besteht aus mehreren Komponenten, die - sofern domänenabhängig - von der Benutzerin zu spezifizieren sind. „ariadne“ selbst stellt vor allem die Kommunikationswege zwischen den einzelnen Komponenten des Dialogsystems her, kann also sozusagen als „black box“ betrachtet werden, und stellt selbst einige Komponenten zur internen Dialogverarbeitung bereit.

Abbildung 2.1 auf Seite 6 zeigt die einzelnen Komponenten, die abgerundeten Kästen stellen die durch "ariadne“ bereitgestellten domänenunabhängigen Komponenten, die „black box“, dar. Die eckigen Kästen sind die jeweils nur für das betrachtete System gültigen Komponenten, die domänenabhängig erstellt werden müssen. Die domänenunabhängigen Komponenten sollen in dieser Arbeit nicht genauer behandelt werden, da von ihnen nur die jeweilige Schnittstelle zum domänenspezifischen Bereich interessant ist. Durch die Pfeile sollen die Beziehungen der einzelnen Komponenten untereinander repräsentiert werden. Die von den beiden Komponenten „Dialogziele“ und „Klärungsfragen“ ausgehenden Pfeile repräsentieren nicht die eigentliche Kommunikation, sondern die Schnittstelle, die es dem System ermöglicht, die benötigte Information weiterzuleiten.

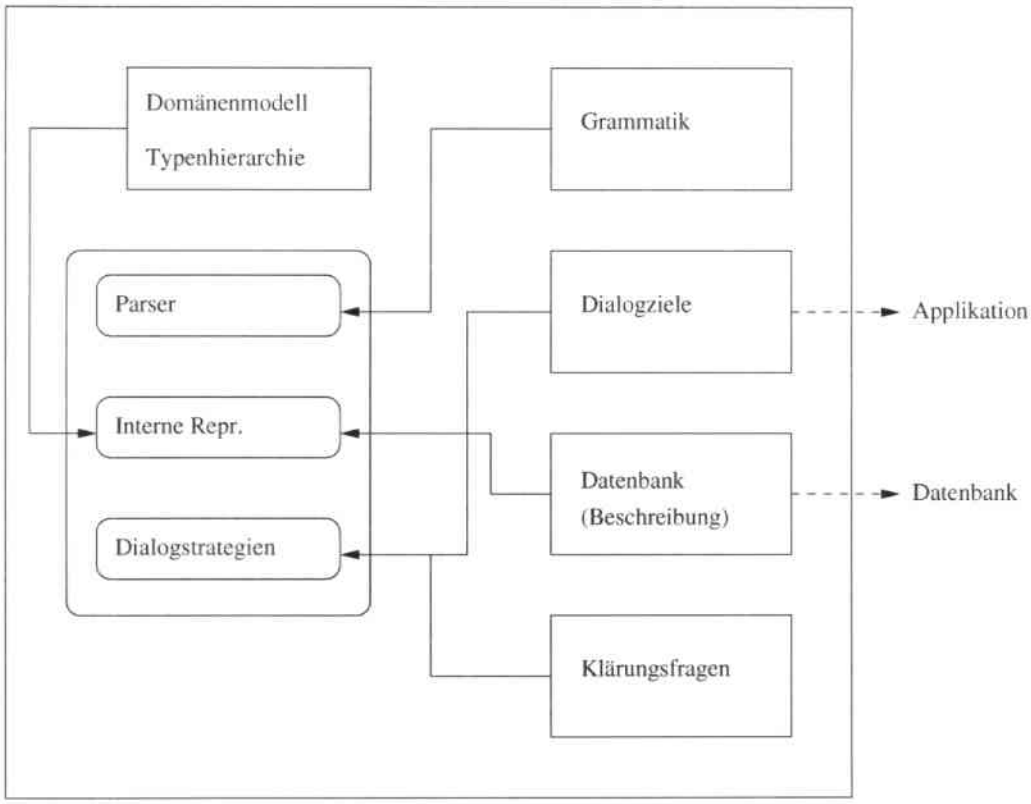

Abbildung 2.1: Aufbau des Dialogsystems

Die Kommunikation des Dialogmanagers mit der Datenbank und der Applikation erfolgt über Sockets. Die in den folgenden Abschnitten beschriebenen Komponenten des Dialogsystems werden alle in einer Datei nacheinander angegeben, am Anfang wird neben dem Namen noch festgelegt, mit welcher Applikation (Package) das System kommunizieren soll und welcher Port, d.h. welches Socket dafür vorgesehen ist.

Von „ariadne“ wird unter anderem ein Parser bereitgestellt, der mit Hilfe der anzugebenden Grammatik in der Lage ist, die von der Benutzerin stammende Eingabe zu verarbeiten und in eine interne Repräsentation umzusetzen, die wiederum durch das Domänenmodell strukturiert wird. Diese interne Repräsentation wird mit den Dialogzielen verglichen und bei ausreichender Übereinstimmung als Eingabe für eine im jeweils ausgewählten Dialogziel festgelegte Funktion der Applikation verwendet. Zusätzlich werden Informationen aus der Datenbank mit Hilfe der zugehörigen Beschreibung ebenfalls in die interne Repräsentation übernommen. Dabei kann die Datenbank auch durch die Applikation simuliert sein. Falls ein Dialogziel nicht vollständig erreicht werden konnte, greift das System auf die bereitgestellten Schablonen für Klärungsfragen zurück, um so weitere Informationen zu erhalten.

2.2 Das Domänenmodell (Typenhierarchie)

Das Domänenmodell beschreibt mit Hilfe einer objektorientierten Typenhierarchie die Konzepte der Domäne, d.h. die vorhandenen Objekte, ihre Eigenschaften und die mit ihnen durchführbaren Aktionen. Die Typenhierarchie wird intern durch typisierte Merkmalsstrukturen (typed feature structures, TFS) repräsentiert. Die TFS sind in [2], bzw. [4] genauer dargelegt, an dieser Stelle soll die Beschreibung nur soweit gehen, wie zum späteren Verständnis der Implementierung notwendig.

Typisierte Merkmalsstrukturen sind Strukturen (Konzepte) von Merkmalen, denen Typen zugeordnet werden, die wiederum selbst eine solche Struktur sein können. Ein solches Konzept stellt dann die Beschreibung eines Objektes, einer Eigenschaft oder einer Aktion dar. Ein sehr einfaches Konzept wäre z.B. das eines abstrakten Objektes, das den Namen OBJECT trägt. Ein Merkmal des OBJECT–Konzeptes wäre dann der Name, der vom Typ STRING ist. Generell kann ein Konzept Merkmale besitzen, die „klassischen" Datentypen wie INT, STRING oder BOOL entsprechen, aber auch solche, deren Typ ein bereits spezifiziertes Konzept ist

„ariadne" arbeitet mit Domänenmodellen, die nach dem Prinzip der Objektorientierung aufgebaut sind, d.h. die in einem Domänenmodell erscheinenden Konzepte sind in einer Vererbungshierarchie angeordnet. Ausgehend von bereits in einer generischen Hierarchie bereitgestellten Basiskonzepten können die benötigten Konzepte spezifiziert werden. Die wichtigsten Basiskonzepte sind OBJECT, ACTION und PROPERTY, von denen ausgehend die domänenspezifischen Konzepte erben können. Ein Konzept OBJ_BOOK, das ein Buch beschreibt, würde folgendermaßen angegeben sein:

```
desc obj_book inherits object {
    string    :    TITLE;
    string    :    AUTHOR;
    int       :    ISBN;
};
```

Dabei erbt das Konzept OBJBOOK von OBJECT das Merkmal *NAME* vom Typ STRING. Ein Beispiel für ein Merkmal, das vom Typ eines bereits definierten Konzeptes ist, wäre:

```
desc obj_book inherits object {
    string        :    TITLE;
    obj_person    :    AUTHOR;
    int           :    ISBN;
};
```

In diesem Fall wäre dann OBJ_PERSON ein Konzept, das von OBJECT geerbt hat. Auf diese Weise werden die Beziehungen **has-a** und **is-a** aus der objektorientierten

Programmierung übernommen und umgesetzt. Bei der Vererbung von Merkmalen können diese auch durch Überschreiben spezialisiert werden. Beispielsweise stellt das Basiskonzept ACTION ein Argument *ARG* vom Typ OBJECT bereit. Ein Aktionskonzept ACT_DOSOMETHINGWITHABOOK könnte dann dieses Merkmal überschreiben:

```
desc act_doSomethingWithABook inherits action {
    obj_book    :    ARG;
};
```

Beim Aufbau der Hierarchie ist auch Mehrfachvererbung möglich, allerdings mit einer Einschränkung gegenüberdem aus der objektorientierten Programmierung bekannten Prinzip der Mehrfachvererbung. Das System sucht mittels Typunifikation zu zwei gegebenen Konzepten (hier z.B. OBJ_BOOKNOVEL, OBJ_BOOKENGLISH) solche, die diese zwei repräsentieren (beispielsweise OBJ_BOOKENGLNOVHARDCOVER und OBJ_BOOKENGLNOVPAPERBACK). Um nun Mehrdeutigkeiten bei der vorgenommen Typunifikation zu vermeiden, die die verwendeten Algorithmen deutlich langsamer machen würden, muß ein eindeutiges Zwischenkonzept eingefügt werden, für das dann alle erbenden Konzepte die beiden Grundkonzepte repräsentieren. Gibt es zum Beispiel die Konzepte OBJ_BOOKENGLISH und OBJ_BOOKNOVEL, könnte aus diesen beiden das Konzept OBJ_BOOKENGLISHNOVEL entstehen:

```
desc obj_bookEnglNov inherits obj_bookNovel, obj_bookEnglish {
    ...;
};
```

Gäbe es nun zwei gleichwertige vererbte Konzepte, z.B. OBJ_BOOKENGLNOVHARDCOVER und OBJ_BOOKENGLNOVPAPERBACK, könnten diese nicht beide direkt von den beiden Grundtypen OBJ_BOOKENGLISH und OBJ_BOOKNOVEL erben, sondern es müßte ein „Zwischenkonzept“ eingeführt werden, von dem dann beide erben können. Abbildung 2.2 zeigt die „falsche“

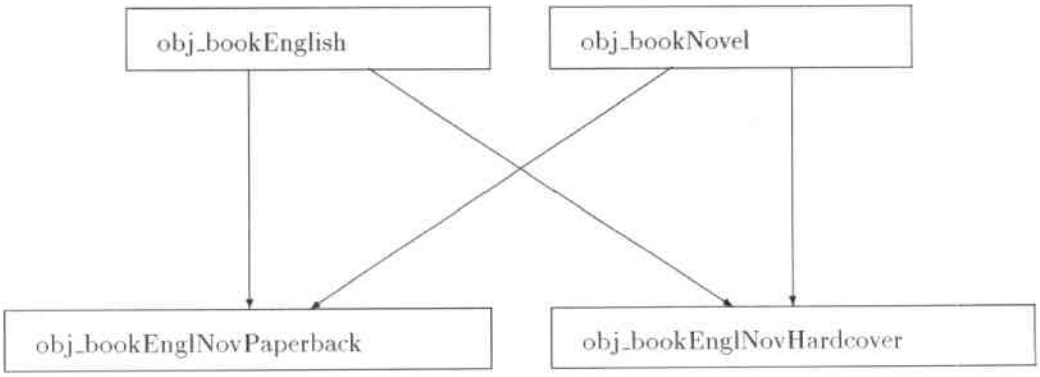

Abbildung 2.2: Zu Mehrdeutigkeiten führende Mehrfachvererbung

Vererbungsstruktur, die zu Schwierigkeiten führt, da die Konzepte sich später nicht

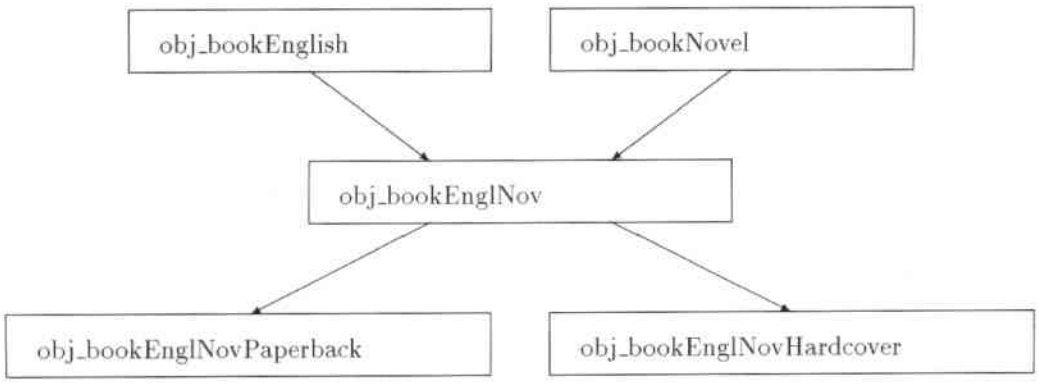

Abbildung 2.3: Korrekte Mehrfachvererbung

eindeutig weiterverarbeiten lassen. Abbildung 2.3 zeigt die korrekte Lösung mit dem eingefügten Konzept OBJ_BOOKENGLNOV. Durch diese von „ariadne" verlangte Besonderheit entstehen bei der Aufstellung des Domänenmodells häufig nicht weiterverwendete Hilfskonzepte, die die Lesbarkeit der Hierarchie etwas verringern.

2.3 TFS und Dialogziele

Der nächste von der Benutzerin zu spezifizierende Teilbereich ist die Auflistung der Dialogziele. In den meisten Fällen soll das Erreichen eines Dialogzieles die Erledigung einer Aufgabe nach sich ziehen, das kann zum Beispiel die Suche nach bestimmten Daten in einer Datenbank sein oder aber die Ausführung einer Funktion der Applikation, die den Zustand der Daten oder der Umgebung verändert. Die Beschreibung eines Dialogzieles verwendet unterspezifizierte typisierte Merkmalsstrukturen (underspecified typed feature structures), die in [4] näher beschrieben werden. Hierbei ist es möglich, das Dialogziel mit der Angabe bestimmter Teile einer TFS zu beschreiben, so daß eine exakte Entsprechung von Aktion und Dialogziel in allen Details nicht benötigt wird. Ein Dialogziel, das die Suche nach einem bestimmten Buch nur mit Hilfe des Titels veranlaßt, könnte folgendermaßen definiert sein:

```
goal SearchBook {
    description:
        [ act_searchBook ]
            ARG     [ obj_book ]
                      TITLE    [ string ]
                    ]
        ]

        .
        .
        .
};
```

Würde das Dialogziel nur die vollständige TFS für ACT_SEARCHBOOK akzeptieren, müßte die gesamte Konzeptstruktur für OBJ_BOOK wie im Beispiel auf Seite 7 angegeben sein. Dies würde eine Suche nach Büchern nur unter Angabe des Titels unmöglich machen, da in diesem das Dialogziel erst als vollständig spezifiziert gelten könnte, wenn alle Angaben zum Buch vorhanden sind, nach denen im allgemeinen aber gerade gesucht werden soll.

Weiterhin wird im Dialogziel angegeben, wie viele Objekte des angegebenen Typs „verarbeitet“ werden können, bzw. wie oft die mit dem Dialogziel verbundene Funktion der Applikation aufgerufen werden kann. Ist durch die Eingabe das gesuchte Objekt nicht exakt, z.B. durch seinen Namen, spezifiziert und sind nicht mehr Objekte des Typs (hier OBJ_BOOK) bekannt, wird die benötigte Funktion für alle diese Objekte nacheinander ausgeführt. Sind mehr Objekte vorhanden, als das Dialogziel akzeptiert, muß durch einen Klärungsdialog das spezielle Objekt spezifiziert, bzw. die Objektmenge eingeschränkt werden.

Es ist außerdem möglich, dem Dialogziel eine Vorschrift anzuhängen, was im Falle der Zurücknahme (cancel) des vorherigen Wunsches zu tun ist. Im Allgemeinen stellt dazu die Applikation entsprechende Funktionen bereit, die die direkt vorher ausgeführte Aktion rückgängig machen. Am Ende des Dialogziels steht die Bindung (binding) an die entsprechende Funktion der Applikation mit den durch das „Füllen“ des Dialogziels gegebenen Aufrufparametern. Für das Beispieldialogziel könnte das angegeben sein durch:

```
goal SearchBook{
    .
    .
    .
    binding:
        BookPackage::searchBook individual : [ARG|NAME];
};
```

Damit wird in der Applikation *BookPackage* die Funktion *searchBook* mit dem Parameter *name* aufgerufen.

2.4 Die Grammatikregeln

Die dritte Komponente ist die sprachenabhängige Grammatik, die für jede Sprache, die das System verarbeiten soll, mit ensprechender Kennzeichnung anzugeben ist. Der Aufruf des Systems erfolgt dann jeweils unter Angabe der gewünschten Sprache. Verwendet wird eine vektorisierte Grammatik, d.h. mit den angegebenen Regeln wird

für den Parser eine Grammatik angegeben und zudem definiert, wie eine Konvertierung der Parserausgaben in die interne Repräsentation, d.h in die TFS, möglich ist. Die Grammatik ist kontextfrei, d.h. auf der linken Seite der Regeln steht ein einzelnes Nichtterminalsymbol (`NT`). Der Formalismus erlaubt strukturelle Regeln der Form `NT -> NT NT NT` sowie lexikalische Regeln `NT -> t` (mit `t` Terminalsymbol). Terminalsymbole sind Teilsätze oder einzelne Wörter, sie werden in Hochkommata (') eingeschlossen. Nichtterminale können unterteilt werden in vektorielle Nichtterminalsymbole und Hilfssymbole. Die vektoriellen Symbole enthalten semantische Information über die Konzepte der Domäne sowie syntaktische Information über die das Konzept sprachlich realisierende Konstituente. Für die vektoriellen Symbole wird hierbei die Form `<Konzept:syn1:syn2>` verwendet, wobei `syn1` die syntaktische Hauptkategorie (N für Nomen, NP für Nominalphrase, entsprechend V und VP für Verb und Verbphrase), `syn2` die syntaktische Unterkategorie angibt. Letztere kann verwendet werden, um mehrteilige Verbphrasen zu beschreiben, was im Deutschen relativ häufig, im Englischen dagegen eher selten vorkommt. Ein Beispiel wäre das Verb `abfahren`, das in konjugierter Form zerlegt vorkommt. Um den Satz `Ich fahre heute ab` verarbeiten zu können, würde man die Nichtterminale `<act_abfahren:V:Sep1>` (für `ich fahre`) und `<act_abfahren:V:Sep2>` (für `ab`) verwenden können.

Hilfssymbole dienen der Entwicklerin als Vereinfachung und sind eigentlich kein integraler Bestandteil der Grammatik. Mit ihnen können zum Beispiel für die Semantik irrelevante, synonym verwendetet Füllwörter zusammengefaßt werden. Sie stehen ohne Angaben syntaktischer Kategorien in spitzen Klammern.

In einer generischen Regel wird der Einstiegspunkt in die Grammatik, d.h. das Startsymbol festgelegt. Dies kann zum Beispiel das Grundkonzept ACTION sein, so daß nur Eingaben, die sich mit einem Aktionskonzept unifizieren lassen, weiterverabeitet werden.

In den Grammatikregeln werden auch Konvertierungsregeln mit angegeben, die es ermöglichen, die sprachlichen Konstrukte auf Konzepte der Domäne zu beziehen. Die Regeln für die Aktion ACT_SEARCHBOOK könnten beispielsweise lauten:

```
<act_searchBook:VP:_>    = 'search for'
                             <obj_book:NP:_> { ARG obj_book }
                         : 'find' <obj_book:NP:_> { ARG obj_book };
<obj_book:NP:_>          = <gq:_:_>* { QUANTIFIER gq }
                           <obj_book:N:_>
<obj_book:N:_>           = 'book'
                         : 'Moby Dick' { NAME "Moby Dick" }
                         : ...;
```

Die Angabe von `{ ARG obj_book }` besagt, daß ein an dieser Stelle stehendes Objekt vom Typ OBJ_BOOK als das Merkmal *ARG* des Aktionskonzepts ACT_SEARCHBOOK angenommen werden soll. Optional kann diesem noch ein Artikel oder Zahlwort vorangestellt sein, was durch `<gq:_:_>* { QUANTIFIER gq }` beschrieben wird. Steht an Stelle des Objektes ein spezieller, eingetragener Buchtitel, kann das System sogar einen Schritt weitergehen, und den Titel des Buches ebenfalls in die interne Repräsentation übernehmen. Es besteht auch die Möglichkeit, Merkmale von Objekten, hier speziell die Buchtitel, in einer Datenbank abzulegen und diese über eine direkte Anbindung in den Grammatikregeln verfügbar zu machen. Wird kein spezieller Titel angegeben, (also z.B. nur `the book`), hängt das weitere Vorgehen davon ab, wie das Dialogziel *SearchBook* definiert ist. Grundsätzlich wird zunächst die Eingabe als dem Konzept OBJ_BOOK entsprechend erkannt. Läßt z.B. das Dialogziel eine Verarbeitung mehrerer Bücher zu und ergibt die Anfrage an die Datenbank für das Konzept OBJ_BOOK nicht mehr Buchtitel, als das Ziel verarbeiten kann, wird die angebundene Funktion der Applikation für jeden Buchtitel aufgerufen.

2.5 Die Datenbankanbindung

Es bestehen zwei Möglichkeiten, dem Dialogsystem Informationen über die tatsächlich vorhandenen Ausprägungen eines Objekts und die Umwelt zu geben. Eine Möglichkeit ist die Benutzung einer Datenbank (MS Access), deren Dateien von „ariadne“ gelesen werden können. Die zweite Möglichkeit ist die Simulation einer Datenbank durch die Applikation. Ersteres ist dann sinnvoll, wenn das ganze System eine Form von Auskunftssytem ist, also tatsächlich Datenbankanfragen aus der natürlichsprachlichen Eingabe extrahieren soll. Existieren insgesamt nur wenige, womöglich noch sehr unterschiedliche Objekte in einer speziellen Umgebung, die mit der Applikation verändert werden kann, ist es sinnvoll, die Datenbank zu simulieren. Für „ariadne“ ist es dabei unerheblich, welche Art von Datenbankanbindung verwendet wird. Ein wichtiger Punkt ist allerdings, daß MS Access Datenbanken keine Zustände von Objekten repräsentieren können, was den Vorteil der Java–Simulation für veränderliche Umwelten erklärt

Unabhängig von dieser Wahl ist die vierte Komponente des Dialogsystems die Spezifikation der „Datenbank“, bzw. ihrer Schnittstelle. Für die einzelnen Datenbanken, bzw. Tabellen wird angegeben, wo sie zu finden sind, ob in einer Datei oder über eine Anfrage an die Applikation. Gleichzeitig wird hier auch die Verbindung zur Typenhierarchie hergestellt. Die Merkmale der Konzepte werden den entsprechenden Merkmalen der Datenbank zugeordnet, so daß bei Bedarf eine Anfrage zu einem gewünschten Merkmal gestellt werden kann. Für eine simulierte Datenbank würde ein Eintrag für Bücher wie folgt aussehen:

```
database Books obj_book internal BookPackage:Books.db
{
    dbtable Books obj_books {
        dbfield Title     = [TITLE];
        dbfield Author    = [AUTHOR];
        .
        .
        .
    };
};
```

2.6 Schablonen (Templates) für Klärungsfragen

Eine wichtige Komponente eines Dialoges sind die Klärungsfragen, mit deren Hilfe das System Informationslücken schließen kann. Damit die Fragen sinnvoll und zur Domäne passend formuliert werden, muß festgelegt werden, wie nach welchem fehlenden Merkmal gefragt werden kann. Die fünfte und letzte Komponente des Dialogsystems ist also eine Auflistung von Schablonen (Templates), mit denen verschiedene Fragen generiert werden können. Diese Templates bilden die zweite der beiden sprachenabhängigen Komponenten. Hierbei sind drei Typen von Templates zu unterscheiden: solche, die eine Frage mit völlig freier Antwortmöglichkeit erzeugen, bezeichnet als *infoqst*, andere, die eine Auflistung der möglichen Antworten präsentieren (*enumqst*) und Aussagen (*statement*), die als Bestätigung verwendet werden können.

Die Auswahl der Templates hängt vom Zustand des Dialogs und evtl. auch von den Werten einzelner Merkmale ab. Mit der Grammatik und der Konvertierungsregel wird versucht, die erhaltene Eingabe mit den Dialogzielen (bzw. einem davon) zu unifizieren. Zunächst stehen alle Dialogziele mit dem gleichen Status *deselected* nebeneinander. Wird nun eine Gruppe (oder ein einzelnes) der Ziele als möglich angenommen, erhalten diese den Status *selected*. Ist genau ein Dialogziel als das gewünschte ermittelt worden, aber noch nicht alle nötigen Merkmale erkannt, erhält dieses Ziel den Status *determined*. Ist ein Dialogziel vollständig spezifiziert und wird somit in der Applikation die zugehörige Funktion aufgerufen, hat das Ziel (wie auch insgesamt der Dialog) den Status *finalized*.

Das Template zur Bestätigung des Aufrufs der Suchfunktion

```
statement {
    state:(finalized = SearchBook) ->
        text: "Looking in database for ##objs@[ARG|TITLE]."
};
```

würde also dann ausgewählt werden, wenn das Dialogziel *SearchBook* erkannt worden wäre und zudem alle benötigte Information verfügbar ist.

Die interne Repräsentation besteht aus zwei Teilen, der Objekt- und der semantischen Repräsentation. Zunächst wird eine mit `##sem@[...]` abrufbare semantische Struktur erzeugt, mit der direkt versucht wird, die Eingabe mit den TFS der Typenhierarchie zu unifizieren. Sind mit Hilfe des Parsers bestimmte Konzepte identifiziert worden, werden diese in die Objektrepräsentation übernommen, aus der sie mit dem Konstrukt `##objs@[...]` abgerufen werden können. Soll zum Beispiel überprüft werden, ob ein bestimmtes Objekt, bzw. Konzept nicht erkannt wurde, ist es sinnvoll, dies in der semantischen Repräsentation abzufragen, da die eigentliche Objektrepräsentation erst dann mit Einträgen gefüllt wird, wenn hinreichend viel Information über das gewünschte Objekt bekannt ist. Das Template

```
infoqst {
    state:(determined = SearchBook),
    path:(undefined = ##sem@[ARG|TITLE]) ->
        text: "What is the title of the book you are looking for?"
        location: [ARG] <obj_book:NP:_>
};
```

würde genau dann gewählt werden, wenn zwar aufgrund der Eingabe klar ist, daß ein Buch gesucht werden soll, aber nicht welches. D.h. der Parser hat in seinem Baum die Eingabe als ACT_SEARCHBOOK identifiziert, konnte aber kein OBJ_BOOK erkennen. Im Template kann mittels *location* angegeben werden, an welche Stelle in der Struktur die zur erwartende Eingabe zu schreiben ist, damit der Parser einen Einstiegspunkt hat.

2.7 Die Schnittstelle zur Applikation

In den vorangegangenen Abschnitten wurde bereits mehrfach der Ausdruck Package (im Beispiel *BookPackage*) für die Applikation gebraucht. Dies rührt daher, daß eine Schnittstelle in Form eines Interfaces (*DialoguePackage*, in Java) vorgegeben ist. Dazu wird noch eine, die Implementierung vereinfachende Klasse (*DialoguePackageAdapter*) angeboten. Diese Klasse reduziert die Anzahl der im jeweiligen Package zu implementierenden Schnittstellen–Methoden. Diese Methoden ermöglichen der eigentlichen Schnittstellen-Applikation (ebenfalls vorhanden), dem *ServerPackage*, die gewünschten Funktionen und Informationen zwischen Dialogsystem und Package auszutauschen. Unter anderem gehören dazu auch die Aufrufe der in den Dialogzielen angegebenen Funktionen, die wiederum von der eigentlichen Applikation zur Verfügung gestellt werden. In Abbildung 2.4 ist das Zusammenspiel der Packages verdeutlicht. In jedem Fall müssen

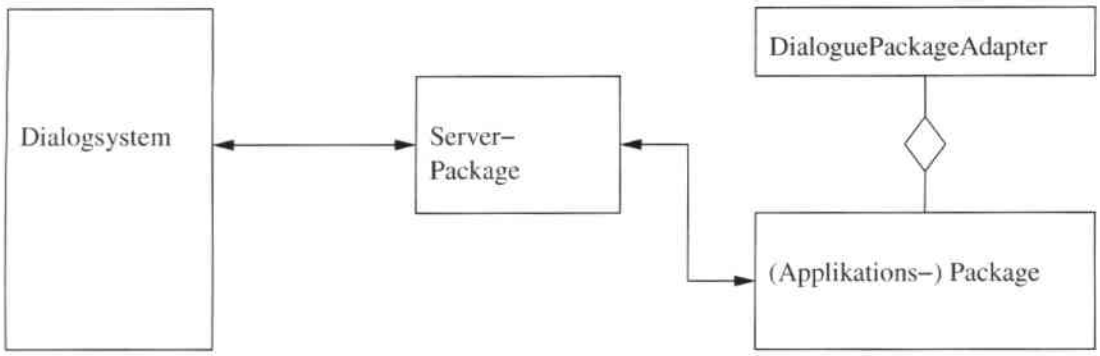

Abbildung 2.4: Zusammenhang Dialogsystem und Applikation

- der Konstruktor des Packages,
- eine Methode `getObjects`, die aus einer „Datenbanktabelle" die dort gehaltenen Objekte ausliest, sowie
- die Methoden `start` und `stop`, die das Package in den Startzustand („Betriebsbereitschaft") versetzen, bzw. es von der Schnittstelle abkoppeln,

implementiert werden. Wird die Datenbank simuliert, müssen die Namen der Datenbank(en), Tabellen und Felder über Stringkonstanten bekannt gemacht werden, genauso wie die Namen der innerhalb des Packages aufrufbaren Methoden.

Das Package, d.h. die Applikation ist letztendlich auch für die Bereitstellung des Weltwissens verantwortlich, d.h. Zustände von Umwelt und Objekten werden nicht im Dialogmanager gehalten. Dementsprechend muß die Applikation Möglichkeiten bereitstellen, z.B. das zuletzt behandelte Objekt zu memorieren oder Entscheidungen zu treffen, ob eine gewünschte Funktion für ein spezielles Objekt ausführbar ist. Bezogen auf das Beispiel „Buchdatenbank" könnte es die Funktion „Buch ausleihen" geben, für die dann die Applikation die Entscheidung treffen müßte, ob sie ausführbar ist (das angefragte Buch könnte bereits verliehen oder ein Präsenzexemplar sein).

Kapitel 3

Das erstellte Dialogsystem

Dein Wort in Gottes Ohr?

(ALLG. GEBR.)

3.1 Vorgehensweise

Ziel der Arbeit war es, ein Dialogsystem für einen Multimediaraum zu spezifizieren. Dementsprechend wurde zuerst das Applikationspackage (*MultimediaRoomPackage*) zur Simulation eines solchen Raumes in Java implementiert. Im Anschluß wurde dann das Dialogsystem der Applikation angepaßt. Dabei wurden zuerst das Domänenmodell, die Dialogziele, eine sehr einfach gehaltene Grammatik sowie die Beschreibung der Datenbank erstellt. Die Templates für die Klärungsfragen wurden erst in einem letzten Schritt, in dem zusätzlich noch die Grammatikregeln erweitert wurden, angefügt.

Diese Vorgehensweise hat sich allerdings als nicht optimal herausgestellt, worauf in Kapitel 4 noch einmal detatillierter Bezug genommen werden soll. In Kapitel 5 wird eine andere Herangehensweise vorgeschlagen.

3.2 Der Multimediahörsaal

Die im nächsten Abschnitt beschriebene Java–Applikation simuliert die Schnittstelle zu einer echten Steuerung verschiedener Geräte. Exemplarisch wurde hier der Multimediahörsaal der Fakultät für Informatik der Universität Karlsruhe als Grundlage verwendet. Allerdings kann hierbei kein Anspruch auf Vollständigkeit bestehen, da nicht die eigentlichen technischen Daten des Hörsaals herangezogen wurden. Die Simulation umfaßt folgende Elemente:

- die zwei Lampensysteme Raumlicht und Tafelbeleuchtung, wobei letztere als stufenlos regelbar angenommen wurde

- drei Video–Projektoren, dabei ist der Hauptprojektor vom Betrachter aus auf die linke Seite des Raumes ausgerichtet, rechts befindet sich ein kleinerer Zweitprojektor und ein dritter strahlt an die Rückwand des Hörsaals
- eine Kamera (dreh- und schwenkbar)
- ein Lautsprechersystem
- ein Mikrophon
- zwei Computer, wobei einer der im Saal fest installierte PC ist, der andere den Anschluß für einen mitgebrachten Laptop meint
- ein Videorecorder
- ein Audiorecorder, bzw. CD-Player
- ein Verdunkelungssystem innen
- die Jalousien an der Außenseite des Gebäudes
- zwei Projektionswände, davon eine (auf die der Hauptprojektor gerichtet ist) fest installiert, aber mit veränderlicher Neigung und Drehmöglichkeit, die andere aufrollbar.

3.3 Die Java–Applikation

Die Klassenstruktur der Java-Applikation entspricht im Wesentlichen den Typen der im vorigen Abschnitt beschriebenen Geräte und Objekte. Abbildung 3.1 auf Seite 19 zeigt diese Struktur. Die abstrakte Adapterklasse `DialoguePackageAdapter` deklariert eine Reihe von Methoden, die der Dialogmanager braucht, um mit dem Package selbst kommunizieren zu können. Diese werden in Abschnitt 2.7, Seite 14 angegeben. Die Klasse MultimediaRoomPackage gibt in einem konstanten Array nach außen zusätzliche Methoden bekannt:

```
private static final String[] methodNames_ =
    { "switchOn", "switchOff", "connect",
      "changeBrightness", "changeVolume", "changeBass",
      "changeTreble", "changeEnrollState", "changeSlant",
      "changeRotation", "changePosition", "switchMode",
      "switchChannel"};
```

Ursprünglich war nur eine allgemeine Methode zur Änderung von Eigenschaften (*changeProperty*) vorgesehen. Im Zuge der Spezifikation der Schnittstelle auf Seiten des Dialogsystems stellte sich aber heraus, daß dies aus in Abschnitt 3.4.1 näher

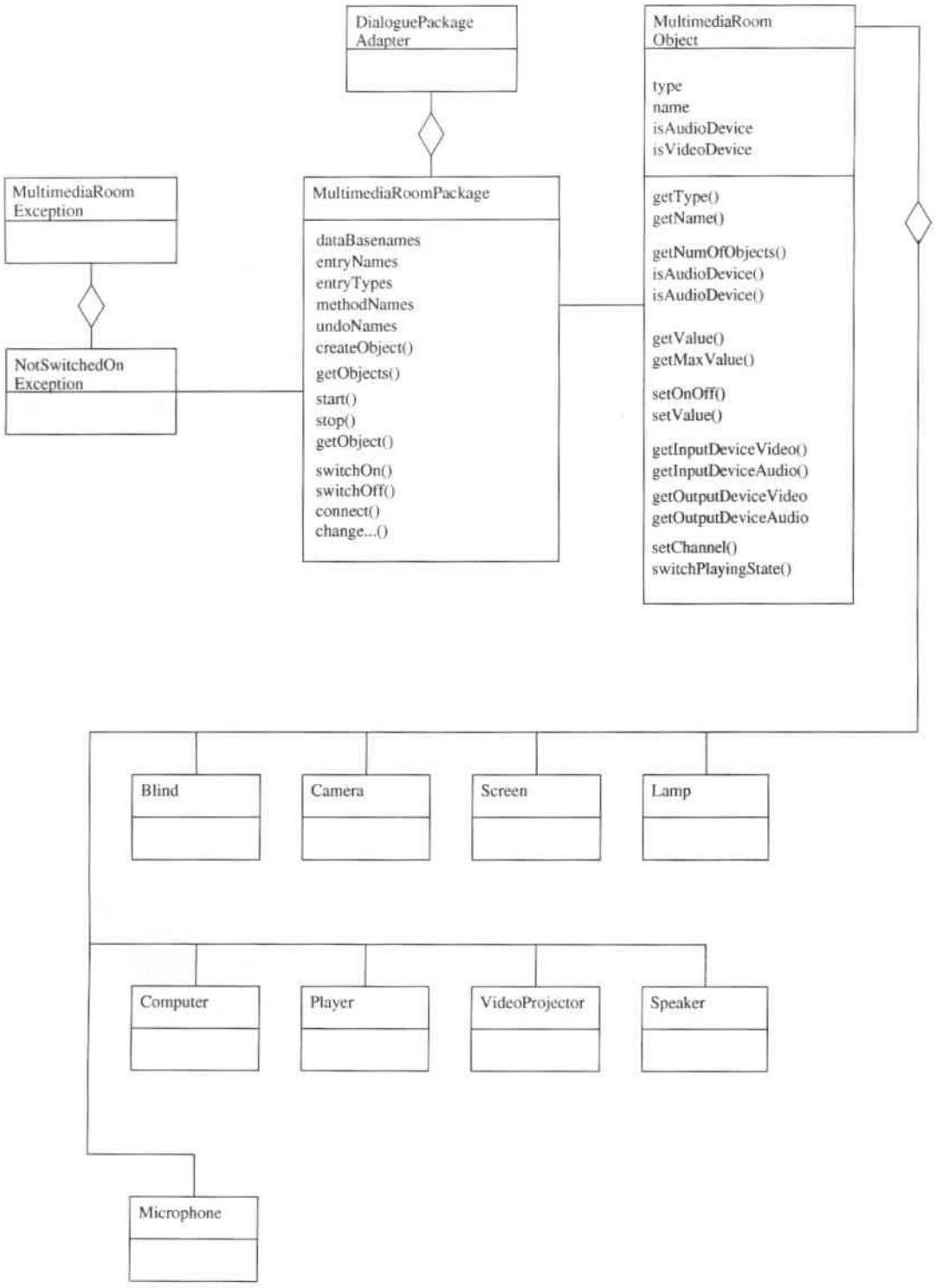

Abbildung 3.1: Klassenstruktur der Java–Applikation. Zur besseren Übersicht wurde auf die Darstellung der einzelnen Elemente und Methoden der Geräteklassen verzichtet. Siehe dazu auch die Beschreibung auf Seite 20 im Text

erläuterten Gründen nicht möglich war. Dementsprechend mußte im Nachhinein die Schnittstelle der Applikation an das Dialogsystem angepaßt und mit Methoden für jede einzelne Aktion ausgestattet werden.

Die Datenbanksimulation wird ebenfalls durch Stringkonstanten realisiert. Mit

```
private static final String[] databaseNames_ =
    { "VideoProjectors", "Computers", "Microphones",
      "Speakers", "Cameras", "Players",
      "Screens", "Blinds", "Lamps"};
```

werden die Datenbanknamen angegeben,

```
private static final String[][] entryNames_ = {
    { "Type", "Name", "OnOff", "InputDevice"},
    { "Type", "Name", "OnOff", "OutputDeviceVideo",
      "OutputDeviceAudio"},
    { "Type", "Name", "OnOff", "Volume", "OutputDevice"},
    { "Type", "Name", "OnOff", "Volume", "InputDevice"},
    { "Type", "Name", "OnOff", "Rotation", "Slant",
      "OutputDevice"},
    { "Type", "Name", "OnOff", "Volume", "Treble", "Bass",
      "Playingstate", "Channel", "InputDeviceVideo",
      "OutputDeviceVideo", "InputDeviceAudio",
      "OutputDeviceAudio"},
    { "Type", "Name", "Enroll", "UpDown", "Rotation", "Slant"},
    { "Type", "Name", "Enroll"},
    { "Type", "Name", "OnOff", "Brightness"}
};
```

beschreibt die einzelnen Tabellen. Dazu kommt noch ein Stringarray, das den Einträgen ihren jeweiligen Typ (STRING, BOOLEAN oder INT, weitere wurden hier nicht verwendet) zuordnet.

Die eigentlichen Geräte, bzw. Objekte sind zunächst durch ein generisches MULTIMEDIAROOMOBJECT modelliert. Auf diese Weise kann das Package auch allgemeinere Informationen liefern und unabhängig vom jeweiligen Objekttyp auf die Objekte zugreifen. Von diesem erben die einzelnen Gerätetypen, die dann die jeweils benötigten Funktionen überschreiben. Zum Beispiel ist nicht für jedes Gerät eine Methode *switchOn* zum Einschalten notwendig. Diese wird dann jeweils in der Klasse des entsprechenden Gerätes definiert. Für das generische MULTIMEDIAROOMOBJECT sind alle Methoden leer. Das heißt sie können beliebig aufgerufen werden und verändern nichts, wenn sie von der angesprochenen Klasse nicht definiert wurden.

Die Vererbungshierarchie ist relativ flach, da bei der Implementierung der Java–

Applikation mehr auf das Vorhandensein verschiedener Gerätetypen als auf die Erstellung einer Hierarchie von Eigenschaften geachtet wurde. Im Vergleich zur später deutlich detaillierter spezifizierten Typenhierarchie des Dialogsystems wird klar, daß eine solch detaillierte Struktur auch auf der Java–Seite sinnvoll gewesen wäre, um eine vollständige Entsprechung zu erreichen. Damit hätte dann ein direkter Bezug von Merkmalen im Dialogsystem zu Eigenschaften und Zuständen hergestellt werden können, so daß auch für Klärungsfragen die Zustandsinformationen zu allen Konzepten (d.h. Geräten) zur Verfügung stünden. Wären von vornherein beide Komponenten gleich modelliert gewesen, wäre auch die Fehlerbehandlung deutlich klarer der Applikation zuzuschreiben gewesen. Momentan wird zwar der Fehler „Gerät ist nicht eingeschaltet" mit der NOTSWITCHEDONEXCEPTION behandelt, andere Fehler werden aber schon dadurch vermieden, daß die Dialogsystemseite nur bestimmte Objekttypen für zugehörige Aktionen zuläßt. Zu dieser Vermischung wäre es bei geschickterer Modellierung nicht gekommen.

3.4 Das Dialogsystem

Das erstellte Dialogsystem soll im folgenden Abschnitt in der Reihenfolge der benötigten Komponenten erläutert werden.

3.4.1 Das Domänenmodell des Systems „multimediaroom"

Anhand der durch die Java–Applikation vorgegebenen Klassen wurden drei Konzepthierarchien für das Domänenmodell aufgebaut. Diese leiten sich jeweils von den drei Basiskonzepten OBJECT, PROPERTY und ACTION ab.

Objektkonzepte

An der Wurzel der in Abbildung 3.2 auf Seite 22 dargestellten Objekthierarchie steht das allgemeine OBJECT, von dem ausgehend zunächst die Grundobjektkonzepte OBJ_SWITCHABLE für alle Geräte, die ein- bzw. ausschaltbar sind und OBJ_WITHVOLUMECONTROL für alle Geräte mit regelbarer Lautstärke abgeleitet sind. Auf der anderen Seite stehen die Konzepte für positionierbare, drehbare, auf- und entrollbare, sowie kippfähige Objekte. Weiterhin steht noch das virtuelle Gerät des Kanals (z.B. für Videokanäle) zur Verfügung. Die Eigenschaft „ein"- bzw. „ausschaltbar" wirkt eventuell etwas sinnlos, wenn man von Geräten spricht, allerdings muß hier beachtet werden, daß es durchaus Geräte wie zum Beispiel elektrische Jalousien gibt, die nicht erst explizit eingeschaltet werden müssen, um herauf- oder heruntergefahren zu werden. Mit dem Konzept OBJ_CONNECTABLE der nächsten Stufe werden alle die Geräte erfasst, die mit einem oder mehreren anderen Geräten

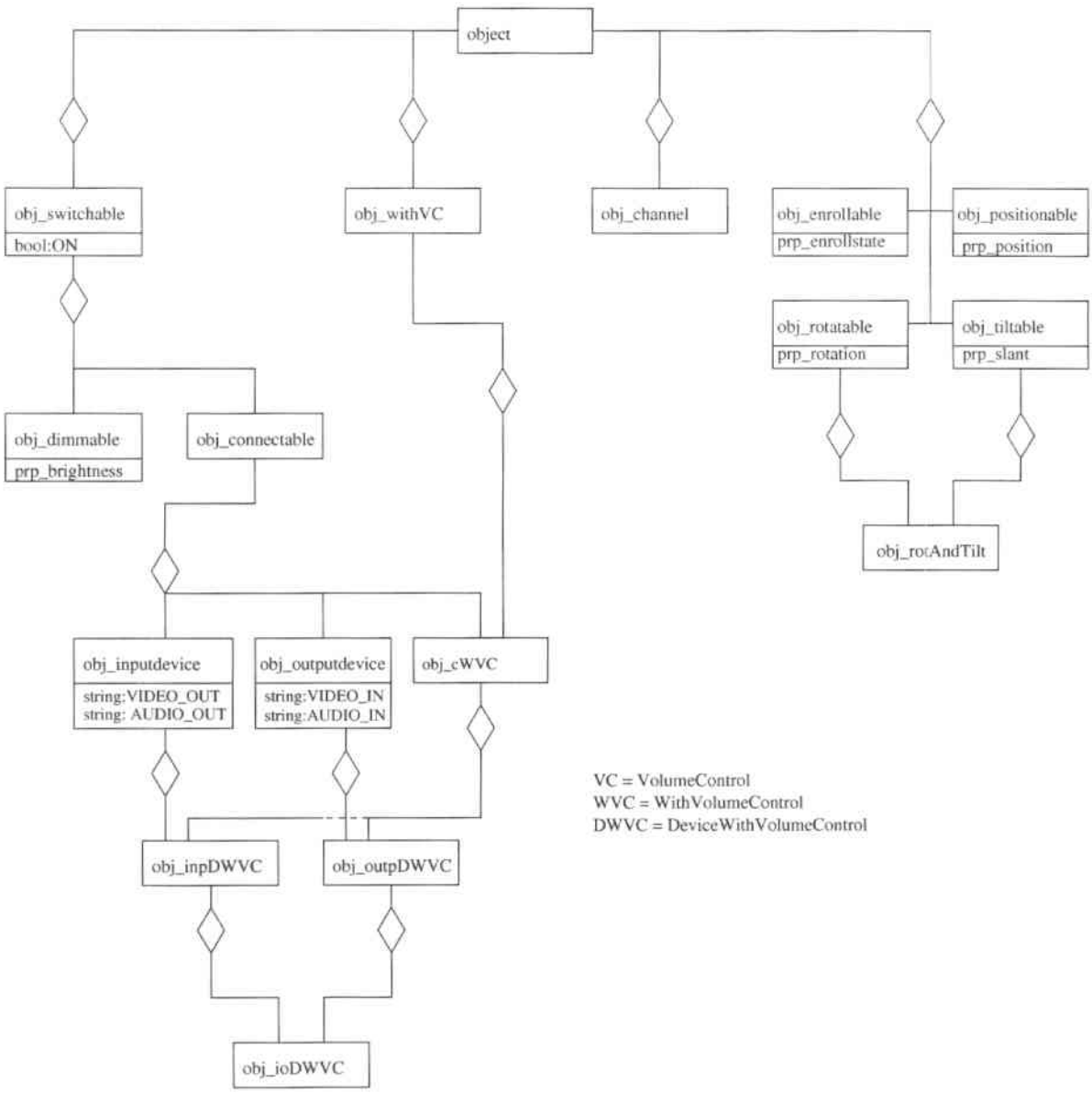

Abbildung 3.2: Die Objekthierarchie

verbunden werden können. „Verbinden" meint in diesem Falle das Umlenken der jeweiligen Signale eines OBJ_INPUTDEVICE zu einem OBJ_OUTPUTDEVICE. Auf diese Weise kann vermieden werden, daß eine sinnlose Verbindung versucht wird. Eine weitere Unterscheidung, die jedoch hauptsächlich innerhalb der Applikation genutzt wird, ist die Unterscheidung zwischen Audio- und Video-Signalen. Für Geräte, die sowohl Ein- als auch Ausgabe verarbeiten, bzw. erzeugen können, wie z.B. ein Audiorecorder, wurde noch das Konzept eines kombinierten Ein-/Ausgabegerätes gebraucht. Zusätzlich sind dann wiederum einige der Geräte mit einer Lautstärkenkontrolle ausgestattet, was insgesamt zu einer relativ verschlungenen Hierarchie der Objekte führt. Dies hat allerdings seinen Grund hauptsächlich in der im vorigen Kapitel erläuterten

Besonderheit bei der Mehrfachvererbung. Aus Gründen der Übersichtlichkeit wurde in der Abbildung auf die Hinzunahme der endgültigen Konzepte für die einzelnen Objekte verzichtet. Die einzelnen Gerätetypen sind wie folgt definiert:

```
desc obj_blind inherits obj_enrollable;
desc obj_microphone inherits obj_inputdeviceWithVolumeControl;
desc obj_speaker inherits obj_outputdeviceWithVolumeControl;
desc obj_camera inherits obj_inputdevice, obj_rotateAndTiltable;
desc obj_projector inherits obj_outputdevice;
desc obj_computer inherits obj_inputdevice;
desc obj_screen inherits obj_enrollable, obj_positionable,
     obj_rotateAndTiltable;
desc obj_lamp inherits obj_dimmable;
desc obj_player inherits obj_iodeviceWithVolumeControl {
     prp_bass         :   BASS;
     prp_treble       :   TREBLE;
     prp_playerstate  :   STATE;
};
```

Im Normalfall ist sicherlich auch ein Computer ein Ein- und Ausgabegerät, es schien allerdings zunächst einmal sinnvoll, ihn nur als signalerzeugendes Gerät zu betrachten. Eine Änderung an dieser Stelle kann aber ohne großen Aufwand durchgeführt werden.

Eigenschaften und Zustände

Eine weitere Subkomponente des Domänenmodells sind die Eigenschaften und Zustände der Objekte, die in einer eigenen Hierarchie modelliert sind. Abbildung 3.3 stellt diese Hierarchie der Eigenschaften und möglichen Zustände der Objekte dar. Ausgehend vom Basiskonzept PROPERTY sind dort der Grad einer Veränderung (PRP_DEGREE), die Grundeigenschaft „veränderlich" (PRP_CHANGEABLE) mit untergeordneten Spezialeigenschaften und der Modus eines allgemeinen Players („play", „stop", „fast forward", etc.) beschrieben. Dazu kommt noch die Absicht (PRP_INTENTION), die angibt, welchen Zweck der Sprecher mit seiner Äußerung verfolgen könnte. Diese Umständlichkeit rührt daher, daß es, wie später noch beschrieben wird, sehr viele verschiedene Aktionen gibt, von denen teilweise zwei den gleichen Zweck verfolgen. Zum Beispiel sind sowohl die Aktion „Licht einschalten" als auch die Aktion „Jalousien hochziehen" eine sinnvolle Reaktion auf die Aussage „Hier ist es aber dunkel". Um einen solchen Satz mit den entsprechend in Frage kommenden Aktionen in Verbindung zu setzen, wurde die „Absicht" als Merkmal eingeführt.

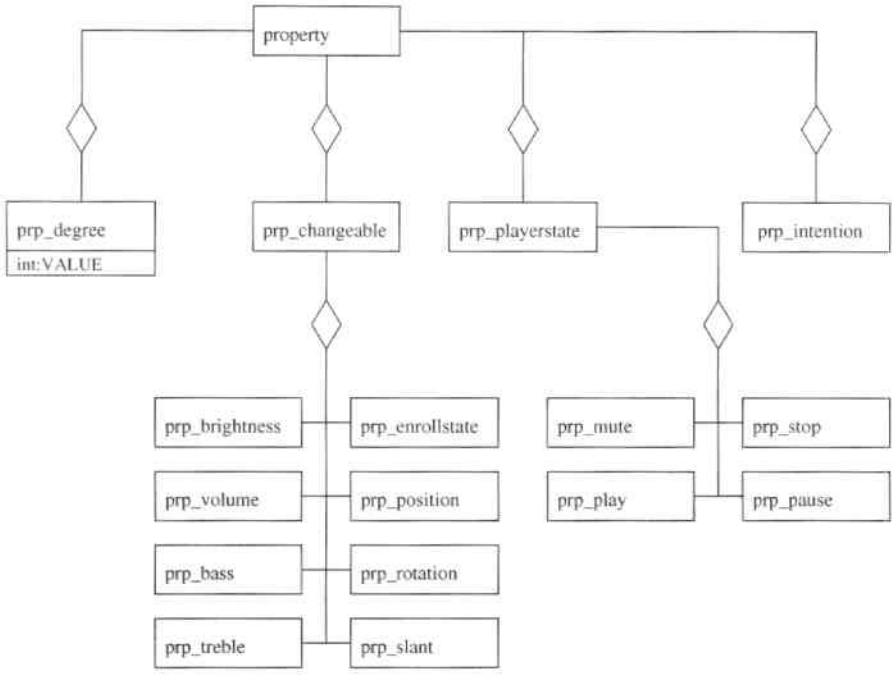

Abbildung 3.3: Eigenschaften und Zustände

Aktionskonzepte

Das dritte verwendete Basiskonzept ist die Aktion (ACTION). Es enthält schon von vornherein ein Argument *ARG* vom Typ OBJECT. Allerdings wurde hier auf die Verwendung dieses Arguments verzichtet und ein weiteres eingeführt, das den Namen *OBJ* trägt. Ohne diesen Umweg ließ sich später keine für das Aktionskonzept ACT_CONNECTDEVICES geeignete generische Grammatikregel finden. Aus diesem Grund wurde auch der Umweg über ein zusätzlich in die Hierarchie eingehängtes allgemeines Aktionskonzept ACT_DOSOMETHING gewählt. Abbildung 3.4 zeigt die Hierarchie der möglichen Aktionen, die auch jeweils die Grundlage für den Abgleich mit den Dialogzielen bilden. Ausgehend von der allgemeinen Aktion ACT_DOSOMETHING sind hier die Aktionen zunächst aufgegliedert nach der Anzahl der Objekte auf die sie sich beziehen. Im Normalfall beziehen sich die möglichen Aktionen auf ein Objekt (ACT_WITHONEOBJECT). Lediglich eine Aktion (ACT_CONNECTDEVICES), die eine Verbindung zwischen einem Ein- und einem Ausgabegerät herstellt, ist eine Aktion mit zwei Objekten (ACT_WITHTWOOBJECTS). Dieses Zwischenkonzept mußte wegen der im Abschnitt 2.2 beschriebenen Besonderheit bei der Mehrfachvererbung eingebaut werden. Den größten Anteil der Aktionen bilden die unter ACT_CHANGEPROPERTY in der Hierarchie eingehängten Aktionen zur Veränderung eines Objektes. Um die Abbildung nicht noch unübersichtlicher werden zu lassen, sollen sie hier einfach alle nur aufgelistet werden:

```
desc act_changeBrightness inherits act_changeProperty {
```

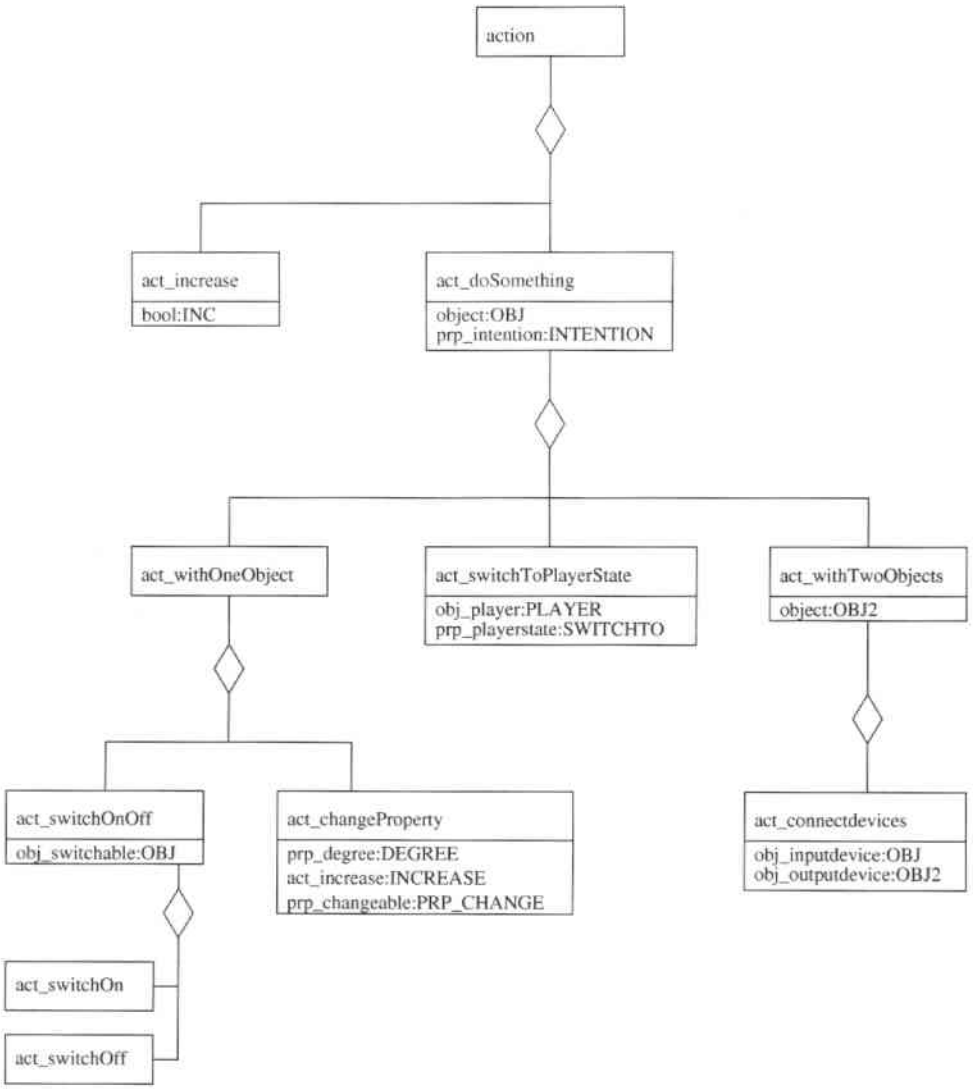

Abbildung 3.4: Hierarchie der Aktionen

```
    obj_dimmable        :    OBJ;
};
desc act_changeVolume inherits act_changeProperty {
    obj_withVolumeControl:    OBJ;
};
desc act_changeBass inherits act_changeProperty {
    obj_player          :    OBJ;
};
desc act_changeTreble inherits act_changeProperty {
    obj_player          :    OBJ;
};
desc act_changeEnrollState inherits act_changeProperty {
    obj_enrollable      :    OBJ;
};
desc act_changePosition inherits act_changeProperty {
```

```
    obj_positionable     :    OBJ;
};
desc act_changeRotation inherits act_changeProperty {
    obj_rotatable        :    OBJ;
};
desc act_changeSlant    inherits act_changeProperty {
    obj_tiltable         :    OBJ;
};
```

Anfangs wurde versucht, es bei einer allgemeinen Aktion für alle Eigenschaften zu belassen, in der dann ein Merkmal für die gewünschte zu verändernde Eigenschaft vorgesehen gewesen wäre. Dazu hätte in der Grammatik ein Wort oder Satzteil auf mehrere Merkmale konvertiert werden müssen. Ein einfaches Beispiel im Deutschen wäre der Begriff „heller". Implizit werden hiermit zwei Informationen gegeben: Es geht zum einen um die Helligkeit (PRP_BRIGHTNESS), zum anderen soll diese Eigenschaft positiv verändert werden (ACT_INCREASE mit `INC = true`).

Fälschlicherweise wurde nach mehreren gescheiterten Versuchen eine solche Doppelkonvertierung als nicht möglich angenommen. In diesen Versuchen war nicht sofort klarzustellen, ob ein Fehler in der Spezifikation auftrat, oder aber „ariadne" selbst nicht in der Lage war, die Konvertierung vorzunehmen. Das Problem wurde dann mit einer veränderten Typenhierarchie und entsprechend veränderten Dialogzielen umgangen. Dies wirkte sich wiederum auf die Schnittstellen (auch in der Java–Applikation) aus, wie bereits in Abschnitt 3.3 angedeutet. Erst im Nachhinein stellte sich heraus, daß die Möglichkeit, eine solche Doppelkonvertierung eines Wortes vorzunehmen, gegeben gewesen wäre. Eine Umarbeitung der Spezifikation erfolgte jedoch nicht, da diese in sämtlichen Komponenten des Systems Auswirkungen gehabt hätte, was einen sehr großen Aufwand bedeutet hätte. Auf diesen Punkt soll in Kapitel 5 auch noch einmal Bezug genommen werden.

Die Aktion ACT_INCREASE steht quasi außerhalb der eigentlichen Hierarchie, da sie nur zur Beschreibung der eigentlichen Veränderungsaktionen benötigt wird. Ist das Attribut *INC* vom Typ BOOL auf `true` gesetzt, wird eine Veränderung in positiver Richtung (increase), bei `false` in negativer Richtung (decrease) angenommen. Der Grad der Veränderung wird jeweils in Prozentschritten der Skala einer veränderbaren Eigenschaft angegeben. Dazu wird das Markmal PRP_DEGREE verwendet.

3.4.2 Die Dialogziele

Durch die bereits beschriebene Vorgehensweise ist das gesamte System sehr stark an der Applikation ausgerichtet, d.h. die Dialogziele stellen ein Abbild der aufrufbaren Funktionen der Java–Applikation dar. Jedes Dialogziel wird in erster Linie durch das

in ihm verwendete Aktionskonzept identifiziert. An dieser Stelle soll nur eines der Ziele exemplarisch ausformuliert werden, die übrigen entsprechen diesem Dialogziel in ihrem Aufbau.

```
goal ChangeBrightness {
    description:
        [ act_changeBrightness
            OBJ      [ obj_dimmable
                       NAME [ string ]
                     ]
            DEGREE   [ prp_degree
                       VALUE [ int ]
                     ]
            INCREASE [ act_increase
                       INC    [ bool ]
                     ]
        ]
    min:
        1
    max:
        10
    cancel:
        [ bot ]
    binding:
        MultimediaRoomPackage::changeBrightness individual :
                    [OBJ|NAME],[INCREASE|INC],[DEGREE|VALUE];
};
```

Das Dialogziel (goal) CHANGEBRIGHTNESS wird beschrieben durch eine Aktion ACT_CHANGEBRIGHTNESS, für die wiederum das Objekt *OBJ*, der Grad der Veränderung *DEGREE* und die Richtung *INCREASE* der Änderung angegeben werden müssen. Die angebundene Funktion der Applikation *changeBrightness* kann, falls das Objekt nicht eindeutig bestimmt werden konnte, für mindestens ein (`min:1`) und maximal zehn (`max:10`) Objekte gleichen Typs (hier OBJ_DIMMABLE) aufgerufen werden. Sind mehr Objekte vorhanden, als das Dialogziel verarbeiten darf, muß mit einer entsprechenden Klärungsfrage das konkrete Objekt erfragt, bzw. die Menge eingeschränkt werden. Dies ist in den später beschriebenen Templates festzulegen. In der vorliegenden Spezifikation wurde allerdings nur die Unterscheidung zwischen einem Objekt und allen vorhandenen Objekten (Geräten) dieses Typs gemacht. Dies geschah dadurch, daß für die Dialogziele SWITCHONOBJECT und SWITCHOFFOBJECT eine Maximalzahl angegeben wurde, bei der sicher war, daß nicht so viele Geräte gleichen Typs vorhanden sind. Auf die Angabe einer Funktionsanbindung zum Rückgängigmachen des Aufrufes (`cancel:`) wurde hier verzichtet, da die Auswirkung der Funktion durch einen erneuten Aufruf mit invertiertem Parameter

INCREASE rückgängig gemacht werden kann.

Die übrigen Dialogziele lauten

- SwitchOnObject (Einschalten eines Gerätes)
- SwitchOffObject (Ausschalten eines Gerätes)
- ConnectDevices (Verbinden zweier Geräte)
- ChangeBrightness (Verändern der Helligkeit)
- ChangeVolume (Verändern der Lautstärke)
- ChangeBass (Verändern der Basseinstellung)
- ChangeTreble (Veränderung der Höhen)
- ChangeEnrollState (Auf-, bzw. Entrollen einer Leinwand/Jalousie)
- ChangeRotation (Drehen eines Objektes)
- ChangePosition (Verändern einer Position z.B. einer Tafel)
- ChangeSlant (Verkippen eines Gerätes, z.B. einer Kamera)
- SwitchPlayerState (Umschalten eines Players in einen anderen Modus)

Auch hier wurde zunächst versucht, für alle Ziele, die in ihrem Namen den Teil „Change" enthalten, nur eines zu wählen, was aber unter der erläuterten fälschlichen Annahme nicht möglich war.

3.4.3 Die Grammatikregeln

Die Implementierung des Systems erfolgte bisher nur in englischer Sprache, d.h. die Grammatikregeln und Templates für die Klärungsfragen liegen nur in Englisch vor, könnten jedoch auch noch in deutscher Sprache spezifiziert werden. Wie bereits in Kapitel 2 erwähnt, kann für jede gewünschte Sprache eine eigene Grammatik und eine Template–Liste spezifiziert werden, die dann je nach Startparameter verwendet werden. Diese beiden Komponenten sind die beiden einzigen sprachenabhängigen Komponenten des Systems, damit wäre eine Hinzunahme weitere Sprachen nur eine Erweiterung des Systems.

Die Regeln für die Grammatik und die Konvertierung sind ebenfalls sehr stark an den Aktionen orientiert, die den Hauptbestandteil der Dialogziele ausmachen. Die verwendete `generic rule`

```
public <action:VP:Imp> implements SpeechActVP :
       action = 0 { "" action }, object = 1 { OBJ object };
```

bildet den Einsprung in die Grammatik des Moduls „MultimediaRoom". Damit wird grundsätzlich angenommen, daß die Regeln für Aktionskonzepte in diesem Modul zu finden sind. Durch die Konvertierung des allgemeinen OBJECT auf das Merkmal *OBJ* ist auch nur die Angabe eines Objektes ohne Bezug zu einer Aktion ausreichend, um die Konvertierung in das verwendete allgemeine Aktionskonzept ACT_DOSOMETHING zu erreichen. Damit wird dann eine weitere Verarbeitung unter der Annahme, daß eine der in diesem Modul spezifizierten Aktionen ausgeführt werden soll, möglich

Die erstellten Regeln für die Grammatik legen zunächst die Konvertierungen für allgemeine Aussagen fest. Die Regeln

```
<act_doSomething:V:_>   = 'Hello'
                        : 'I would like to give a talk'
                          { INTENTION prp_intTalk };
```

setzen zum Beispiel die Eingabe `Hello` in die allgemeinste Aktion des Moduls (ACT_DOSOMETHING) um, so daß eine Klärungsfrage benötigt wird, um herauszufinden, was die Benutzerin wünscht. Eine andere Möglichkeit ist es, mit der Eingabe `I would like to give a talk` zumindest eine Absichtserklärung zu erkennen, was die Anzahl der möglichen (anzubietenden) Aktionen stark einschränkt.

Der weitere Teil der Grammatikregeln legt im Wesentlichen die Umsetzung der Aktionskonzepte fest. Hierbei wurde versucht, möglichst viele verschiedene Sätze aufzunehmen, die als Eingaben vorkommen könnten, was sich allerdings als recht schwierig erwies, da eine Einzelperson immer in einem relativ beschränkten Rahmen nach Formulierungen sucht. Zusätzlich sind noch die möglichen Eingaben für die eingesetzten Objektkonzepte und Eigenschaften berücksichtigt, die dann zu einer entsprechenden Konvertierung führen.

Die für die mit der Eingabe `Make the blackboard light a little brighter` aufgerufene Aktion „Tafelbeleuchtung heller machen" benötigten Regeln lauten:

```
<act_changeBrightness:V:_>  = <help>* { ignore }
                              <obj_dimmable:NP:_>* { OBJ obj_dimmable }
                              <prp_degree:A:_>* { DEGREE prp_degree }
                              'brighter' { INCREASE|INC true };

<prp_degree:A:_>            = 'a little'    { VALUE 10 }
                            : 'a lot'       { VALUE 50 };

<obj_dimmable:NP:_>         = <gq:_:_>* { QUANTIFIER gq }
```

```
                              <obj_dimmable:N:_>;

<obj_dimmable:N:_>          = 'light'
                            : 'blackboard light'
                              { NAME "blackboard light" };

<help>                      = 'make'
                            : 'do'
                            : 'put';
```

Dabei spezifiziert der Beispielsatz das Dialogziel *ChangeBrightness* bereits vollständig. Aus Gründen der Übersichtlichkeit sind hier nur die zur Bearbeitung des oben angeführten Beispielsatzes nötigen Einträge aufgeführt. Der Zusatz `ignore` gibt an, daß das eventuell vorangegangene Hilfskonzept für die Konvertierung unerheblich ist, damit also schlicht ignoriert werden kann. Mit der Angabe von „the blackboard light" wird die gewünschte Lampe identifiziert. Dabei wird zunächst für die Eingabe festgestellt, daß es sich bei `blackboard light` um ein OBJ_DIMMABLE handelt, das (mit einem vorangestellten Artikel) an der passenden Stelle für das Aktionskonzept ACT_CHANGEBRIGHTNESS steht. Durch `a little` wird die Schrittweite `DEGREE|VALUE` auf 10 (Prozent) gesetzt. `brighter` setzt die Richtung der der Änderung `INCREASE|INC` auf `true`.

3.4.4 Die Anbindung an die simulierte Datenbank

In der Definition der Datenbankeinträge werden die einzelnen Merkmale der Konzepte aus der Typenhierarchie mit den jeweiligen Feldern der Datenbankeinträge in Verbindung gebracht. Für die Objekte vom Typ OBJ_PLAYER sieht diese Definition beispielsweise folgendermaßen aus:

```
database Players obj_player
    internal MultimediaRoomPackage:Players.db {
    dbtable Players obj_player {
        dbfield Name              = [NAME];
        dbfield OnOff             = [ON];
        dbfield Volume            = [VOLUME];
        dbfield Treble            = [TREBLE];
        dbfield Bass              = [BASS];
        dbfield Playingstate      = [STATE];
        dbfield InputDeviceVideo  = [VIDEO_IN];
        dbfield InputDeviceAudio  = [AUDIO_IN];
        dbfield OutputDeviceVideo = [VIDEO_OUT];
        dbfield OutputDeviceAudio = [AUDIO_OUT];
    };
};
```

Die „Datenbank" `Players.db` existiert als solche nicht, sie wird durch die Angabe `internal MultimediaRoomPackage` direkt aus der Java-Applikation ausgelesen.

Für jedes Objektkonzept aus der Typenhierarchie, bzw. für jede ein Gerät beschreibende Klasse der Java-Applikation existiert eine solche Umsetzung der Einträge. Beim Start von „ariadne" werden zunächst einmal alle Einträge überprüft, später können dann Anfragen an das System mit den Einträgen der Datenbank abgeglichen werden.

Das System stellt eine Aktion zur Verbindung zweier Geräte, d.h. Objekte bereit, die in beliebiger Reihenfolge angegeben werden können. Um hier einen Datenbankaufruf für beide Objekte zu gewährleisten, müssen die Definitionen der Schnittstelle für Konzepte, die mit der Aktion ACT_CONNECTDEVICES verbunden werden können, mehrfach angegeben werden. Dieses Phänomen läßt sich mit dem folgenden Beispiel erklären:

Verknüpft werden sollen ein Videoprojektor und ein Computer. Von beiden Gerätetypen gibt es mehrere im Raum, die Anfrage `connect the projector to the computer` läßt für beide Geräte die genaue Bezeichnung offen. Erkannt werden sollte, um welche Objekttypen es sich handelt und daß in beiden Fällen nur eine so kleine Anzahl an Geräten vorhanden ist, daß eine Aufzählung vertretbar ist, die dann zur Auswahl der Geräte angeboten werden sollte. Die Typen werden erkannt, es erfolgt die Datenbankanfrage für das zuerst erkannte Objekt, also hier den Computer. Als nächstes müßte die Anfrage für den Projektor erfolgen, da aber in der Auflistung der Datenbanken das Konzept OBJ_PROJECTOR **vor** OBJ_COMPUTER steht, wird dieser Block nicht mehr gelesen, weil das System immer nur einmal die Liste der Datenbankanbindungen durchsucht, bis es einen passenden Eintrag gefunden hat.

Um dieses Problem zu umgehen, wurden die Definitionen mehrfach angegeben:

```
database VideoProjectors obj_projector internal
  MultimediaRoomPackage:VideoProjectors.db  {
  ...
};
database Blinds obj_blind internal
  MultimediaRoomPackage:Blinds.db {
  ...
};
database Computers obj_computer internal
  MultimediaRoomPackage:Computers.db  {
  ...
};
```

```
database Lamps obj_lamp internal
  MultimediaRoomPackage:Lamps.db {
  ...
};
database Microphones obj_microphone internal
    MultimediaRoomPackage:Microphones.db  {
  ...
};
database Players obj_player internal
  MultimediaRoomPackage:Players.db {
  ...
};
database Screens obj_screen internal
  MultimediaRoomPackage:Screens.db {
  ...
};
database Speakers obj_speaker internal
 MultimediaRoomPackage:Speakers.db  {
  ...
};
database VideoProjectors obj_projector internal
    MultimediaRoomPackage:VideoProjectors.db  {
  ...
};
database Microphones obj_microphone internal
    MultimediaRoomPackage:Microphones.db  {
  ...
};
database Players obj_player internal
  MultimediaRoomPackage:Players.db {
  ...
};
database Computers obj_computer internal
  MultimediaRoomPackage:Computers.db  {
  ...
};
```

3.4.5 Templates für Klärungsfragen

Bei den Templates für die Klärungsfragen wurde versucht, alle Möglichkeiten der Unklarheit einzelner Bestandteile einer Anfrage abzufangen. Ist zum Beispiel in der Anfrage klar, daß ein Objekt eingeschaltet werden soll, wird aber das Objekt, oder gar der Typ nicht erkannt, so wird eine entsprechende Frage nach dem einzuschaltenden Gerät generiert. Das Template

```
infoqst {
```

```
        state:(determined = SwitchOnObject),
        path:(undefined = ##sem@[OBJ]) ->
        text: "What should be switched on?"
        options: "videorecorder lamp stereo projector audio"
        location: [OBJ] <obj_switchable:NP:_>
    };
```

springt also dann an, wenn beispielsweise die Eingabe `switch on` erkannt wird, das Objekt aber unklar ist. Generiert wird die Frage `What should be switched on?`, als Antwort wird ein OBJ_SWITCHABLE erwartet. `The blackboard light` wäre also eine gültige Antwort. Die Einträge unter *options* geben den Sprachraum an, in dem ein eventuell eingesetzter Spracherkenner nach der Eingabe suchen kann, um diese Suche zu erleichtern. Entsprechend gibt es weitere Templates für die Nachfrage nach dem Objekt für die Dialogziele *SwitchOffObject* und alle *Change*-Ziele. Es wäre möglich gewesen, an dieser Stelle ein Template für alle Nachfragen nach dem Objekt anzugeben, bei dem im Text eine „Variable" der Form `##goals@[]:<action:V:_>` verwendet worden wäre, die je nach Ziel den jeweils ersten Regeleintrag der Grammatikregel für das entsprechende Aktionskonzept zum Inhalt gehabt hätte. Dieser Eintrag hätte zum Beispiel für die Aktion ACT_CHNAGEBRIGHTNESS `'be dimmed'` lauten können, um die Frage `What should be dimmed` zu erzeugen. Da aber für jede Aktion ein Dialogziel vorhanden ist, erschien es sinnvoll, hier auch für jedes spezielle Ziel ein Template zu verwenden und auf die Verwendung solcher Variablen zu verzichten.

Weitere Templates sorgen für eine entsprechende Frage, wenn eine Veränderung eintreten soll, aber nicht klar ist, um wieviel diese Änderung erfolgen soll. Alle bisher genannten Templates sind vom Typ *infoqst*, d.h. sie geben nicht durch eine Auflistung vor, wie die Antwort formuliert sein sollte.

Eine weitere Gruppe von Templates soll mit Hilfe einer Aufzählung bekannter Objekte ermitteln, welches konkrete Objekt gemeint ist.

```
enumqst {
    state:(determined = ConnectDevices),
    path:(defined = ##objs@[OBJ2]),
    path:(ambiguous = ##objs@[OBJ|NAME]) ->
    text:   "I have
            ##objs^num@[OBJ2]
            ##objs^first@[OBJ]:<obj_inputdevice:N:_>,
            they are named
            ##objs^first@[OBJ|NAME]
            ##objs^middle@{ , [OBJ|NAME]}
            ##objs^last@{ and [OBJ|NAME]}.
```

```
            Which one would you like to  use?"
        location: [OBJ] <obj_inputdevice:NP:_>
};
```

sorgt dafür, daß, wenn zwei Geräte miteinander verbunden werden sollen, von denen das Ausgabegerät (*OBJ2*) eindeutig, vom Eingabegerät (*OBJ*) aber nur der Gerätetyp bekannt ist, eine Aufzählung der passenden Geräte erfolgt. Die Benutzerin kann dann aus dieser Liste eines der Geräte genau benennen.

Falls aus einer Eingabe das gewünschte Dialogziel gar nicht ermittelt werden konnte, oder zumindest mehrere Ziele den Status *selected* haben, gibt es ebenfalls entsprechende Templates. Mit

```
infoqst {
    state:(ambiguous = ##goals@[]) ->
    text: "How can I help you?"
};
```

wird nachgefragt, wenn alle Ziele möglich sind. Ist mit einer durch PRP_INTENTION bekannten „Absicht", die Liste der möglichen Ziele schon ein wenig eingegrenzt, gibt es Templates, die aus dieser Liste das gewünschte Ziel zu ermitteln suchen:

```
infoqst {
    state:(ambiguous = ##goals@[]),
    equals:(prp_intShowMovie = ##sem@[INTENTION]) ->
    text: "Do you need the vcr, or do you have the film on a computer?"
};
```

Im vorliegenden Beispiel würde eine Eingabe wie `I would like to show a film` zur oben stehenden Klärungsfrage führen.

Die Definition der Templates wird abgeschlossen durch Aussagen (*statement*), die dann aufgerufen werden, wenn ein Dialogziel vollständig erfüllt werden konnte, für die Aktion „Licht einschalten" wäre dies

```
statement {
    state:(finalized = SwitchOnObject) ->
    text: "Switching on
            ##objs^first@[OBJ|NAME]
            ##objs^middle@{ , [OBJ|NAME]}
            ##objs^last@{ and [OBJ|NAME]}."
};
```

In diesem Fall werden alle eingeschalteten Geräte noch einmal aufgezählt. `##objs^first@[OBJ|NAME]` greift auf den Namen ersten Objekt der Liste aller n einzuschaltenden Geräte zu, `##objs^middle@{ , [OBJ|NAME]}` auf die

Namen der Objekte `2` bis `n-1`, die durch Kommata getrennt werden und `##objs^last@{ and [OBJ|NAME]}` beschließt die Aufzählung mit dem `n`-ten Objektnamen.

Die Reihenfolge der Templates ergibt sich durch ihre jeweilige Spezifität, da das erste passende Template beim Durchlauf ausgewählt wird. Somit müssen allgemeinere Templates immer unter den spezielleren stehen, da diese sonst nicht mehr erreicht werden.

Kapitel 4

Ergebnisse

Theorie ist, wenn nichts funktioniert,
aber alle wissen, warum.

(ALLG. GEBR.)

In diesem Kapitel sollen die Ergebnisse der Arbeit zusammengefaßt und die während der Anfertigung der Spezifikation aufgetretenen Probleme diskutiert werden.

4.1 Allgemeines

Der Dialogmanager „ariadne" befand sich während der gesamten Spezifizierungsphase des Dialogsystems selbst noch in der Entwicklung, woduruch sich die folgenden Schwierigkeiten ergaben.

Die Hauptschwierigkeit lag in der nicht immer mit dem Entwicklungsstand des Dialogmanagers konsistenten Dokumentation. Diese Inkonsistenz entstand aus einem sehr schnellen Wachstum des Dialogmanagers, dessen Funktionalitätsumfang zu Beginn der Arbeit noch gar nicht vollständig abzusehen war. Daraus ergab sich oftmals das Problem, daß nicht bekannt war, ob und wie bestimmte Ideen umgesetzt werden könnten. Ein solches Problem war die sich durch alle Bereiche hindurchziehende Umständlichkeit, die entstand, weil nicht bekannt war, wie die einfachere Umsetzung möglich gewesen wäre. Das Problem wurde bereits in Abschnitt 3.4.1 auf Seite 24 näher erläutert. Gerade dieses Beispiel zeigt, daß bei einer anderen Umsetzung, d.h. einer veränderten Modellierung, das gesamte System deutlich übersichtlicher geworden wäre, weil viele Dialogziele und demnach auch Templates einzusparen gewesen wären. Ein weiteres Problem, das die fortlaufende Entwicklung von „ariadne" mit sich brachte, war die Schwierigkeit, in einer Fehlersituation zu entscheiden, ob ein Spezifikationsfehler im Dialogsystem oder ein Implementierungsfehler im Dialogmanager („ariadne") selbst vorlag.

Aus Sicht der Verfasserin ist anzumerken, daß der Dialogmanager mit den Templates doch unerwartet viel vom Dialogmanagement an die Benutzerin abgibt, da in ihnen sehr detailliert beschrieben ist, unter welchen exakt angegebenen Bedingungen nach einem Merkmal gefragt werden kann und wie die Klärungsfrage formuliert sein soll. Damit wird ein großer Teil des Interaktionswissens für den Dialog doch wieder in der domänenabhängigen Wissensbasis des Systems verankert, obwohl die Intention von „ariadne" dahingehend interpretiert wurde, daß genau diese Bereiche voneinander getrennt werden sollten.

4.2 Zur Vorgehensweise

Bereits in Abschnitt 3.1 wird die zur Spezifikation des Systems verwendete Vorgehensweise erläutert. Auch dort wird schon erwähnt, daß sich diese Herangehensweise als nicht optimal erwies.

Die wesentliche Problematik ergibt sich daraus, daß im Falle der zuerst implementierten Java–Applikation natürlich versucht werden muß, die Schnittstelle des Dialogsystems an die Applikation anzupassen. Dies führt dazu, daß bei der Umsetzung der eigentliche Dialog leicht aus dem Auge verloren wird, und nur auf die Übertragung der Möglichkeiten des Java–Programms in das Dialogsystem geachtet wird. Im vorliegenden Fall hat dies dazu geführt, daß tatsächlich sehr viel mehr Wert auf die Behandlung von aufgabenstellenden Anfragen wie `Switch on the light` gelegt wurde, als auf eine allgemeine Dialogführung, die natürlich auch allgemeine Anfragen wie `What can I do in this room?` berücksichtigen sollte. Sicherlich ist hier auch nicht die strikt umgekehrte Vorgehensweise anzuraten, da dann die eigentlichen Ziele des Systems zu sehr in den Hintergrund rücken könnten. Sinnvoll erscheint der Verfasserin dieser Ausarbeitung ein schrittweise rotierendes Vorgehen, indem zuerst nur eine Vorstellung von einer Applikation und ihrer Schnittstelle existiert, dann der Dialog aufgebaut wird und erst im Anschluß die eigentliche Applikation erstellt wird.

Durch die verwendete Vorgehensweise wurde unter anderem versäumt, das für den Dialog benötigte Weltwissen gleich zu Beginn mit in die Java–Applikation aufzunehmen, was letztlich zu den beiden ungleichen Vererbungshierarchien führte (siehe auch Abschnitte 3.3 und 3.4.1). Auch aus diesem Grund erscheint es sinnvoller, zunächst durch die Spezifikation des Dialoges festzulegen, welches Wissen benötigt wird, um dieses dann bereitstellen zu können.

4.3 Fähigkeiten des Dialogsystems

In der vorliegenden Fassung ist das System in der Lage, konkrete Aufforderungen zur Erledigung einer Aufgabe korrekt zu verarbeiten und die Ausführung durch Textausgabe zu simulieren. Folgende und ähnliche Eingaben führen zu einer direkten Umsetzung in die Java–Applikation:

- `Switch on <DEVICE>`, `DEVICE` bezeichnet das einzuschaltende Gerät, kann aber auch unspezifisch sein, z.B. `the light`, dann werden alle verfügbaren Geräte des Typs „Lampe“ (OBJ_LAMP) eingeschaltet.
- `Switch off <DEVICE>`, analog zu `Switch on`.
- `Connect <DEVICE 1> to <DEVICE 2>`, hier muß mindestens eines der Geräte ein Eingabe- und mindestens eines ein Ausgabegerät gleichen Typs (Audio oder Video) sein.
- `Make <LAMP> a little brighter`, anstelle von „a little“ können auch andere Angaben („a lot“, „10 percent“, etc.) stehen. Ist `LAMP` nicht regelbar (dimmable), wird dies an entsprechender Stelle (in der Java–Applikation) abgefangen und eine Fehlermeldung erzeugt.
- `Make <AUDIO> a lot louder` entspricht im Wesentlichen dem vorherigen Eintrag, d.h für ein lautstärkenregelbares Gerät wird die Lautstärke größer.
- `A lot more bass on <PLAYER>` setzt für z.B. einen Audiorecorder den Bassregler hoch
- `A little less treble on <PLAYER>` (analog zu `bass`)
- `Roll up <SCREEN> a little`, `SCREEN` steht hier stellvertretend für alle auf- und abrollbaren Objekte.
- `Rotate <CAMERA> left`, auch hier steht `CAMERA` für alle Geräte die drehbar angebracht sind.
- `Tilt <SCREEN> forward a little` verändert die Neigung eines Objektes, das entsprechend kippbar ist.
- `Set <PLAYER> to <PLAYMODE>` schaltet z.B. einen Videorecorder in den Modus `PLAYMODE`, beispielsweise „Fast Forward“.

Für die jeweiligen Aufforderungen stellt die Grammatik noch weitere Formen bereit, die dann sofort verarbeitet werden können. Teile der oben aufgelisteten

Sätze erzeugen unvollständige interne Repräsentationen und führen dann zur Suche nach einem passenden Template für eine Klärungsfrage. So würde die Eingabe `Connect the projector to the laptop` zur Frage

```
I have four projectors, they are named: <ONE>, <TWO>,
<THREE> and <FOUR>. Which one would you like?
```

führen. Dabei stehen die Bezeichner in eckigen Klammern nur als Platzhalter für die tatsächlichen Namen der Geräte. Bei dem bisherigen Entwicklungsstand des Dialogmanagers ist es an dieser Stelle leider nicht möglich, eine Antwort wie `The second` korrekt zu verarbeiten, dies soll aber in zukünftigen Implementierungen möglich sein. Unter Umständen erfordert ein entsprechendes Konstrukt auch Veränderungen in der Spezifikation. Zur Zeit muß also noch der in der Liste erwähnte Name des Projektors angegeben werden, um den Dialog zu einem korrekten Endzustand zu führen.

Weitere Eingaben sind relativ freie Aussagen, die sich nicht direkt auf eine der möglichen Aktionen beziehen.

- `I would like to give a talk` (und ähnliche Absichtserklärungen) läßt das System vermuten, daß ein Projektor (oder mehrere) mit der Ausgabe eines Rechners verbunden werden soll. Die Benutzerin wird gefragt `Do you need a projector and a computer?`. Auch hier war es leider nicht möglich, die Antwort „yes" zu verarbeiten. Dies wäre grundsätzlich möglich, funktioniert aber nur dann, wenn aufgrund des vorherigen Dialogs bereits eine Auswahl von Geräten in einer Auflistung vorhanden ist, die der Reihe nach abgefragt werden können. Um dies zu erreichen, müßten eine Reihe von Veränderungen in den Grammatikregeln und Konvertierungsregeln durchgeführt werden.

- `It is too dark in here` (u.ä.) führt zur Nachfrage, ob die Verdunkelung des Raumes hochgezogen, oder aber das Licht eingeschaltet werden sollte. Hier fehlt derzeit noch der Bezug zur Realität, d.h. die Anfrage an die Applikation, bzw. die Datenbank, ob die Jalousien tatsächlich unten sind, die natürlich erfolgen sollte, bevor eine Klärungsfrage ausgegeben wird.

Eine wesentliche Komponente eines Dialogsystems, nämlich Spracheingabe und -erkennung wurden bisher noch nicht eingesetzt, da es zunächst nur um den Dialog als solchen gehen sollte. Die Erkennung einzusetzen sollte allerdings auch an der Dialogführung nichts ändern, bisher arbeitet das System quasi auf simulierten Hypothesen eines Erkenners und würde bei der Hinzunahme der echten Spracherkennung an dieser Stelle eben echte Hypothesen erhalten. Bei der Erstellung der Grammatikregeln und Templates wurde deswegen auch versucht, solche Fälle, in denen ein

eventuell eingesetzter Spracherkenner nur Halbsätze o.ä. liefert, abzufangen. Bei der textuellen Eingabe würde eine Benutzerin in den seltensten Fällen `Switch on`, also die Aufforderung ohne das Objekt, eingeben, was aber eventuell bei einer automatischen Erkennung passieren könnte, wenn das Objekt nicht erkannt und nur das erste Satzfragment an den Parser weitergereicht wird.

4.4 Verhalten im Test

Ein kleinerer Test mit zwei Personen, die mit dem System nicht vertraut waren und nur die Information hatten, daß sie verschiedene Anfragen zur Verknüpfung und Ansteuerung von Geräten stellen könnten, machte deutlich, daß gerade im Bereich des „freien Dialogs" noch sehr viele Möglichkeiten übersehen wurden.

Dies lag vermutlich an der recht starken Einschränkung auf die Betrachtung von Anfragen, die sich direkt auf die vorhandenen Geräte beziehen. So fehlt z.B. völlig die Möglichkeit, allgemeine Anfragen der Art `What can I do here` zu verarbeiten oder überhaupt erst einmal in der Begrüßung eine Auflistung der Systemoptionen zu geben. Diese Art von Mängeln dürfte allerdings ohne großen Aufwand zu beheben sein, ebenso ist die Hinzunahme von alternativen Formulierungen für bereits vorhandene Eingaben mit nur wenig Aufwand verbunden.

Kapitel 5

Zusammenfassung und Ausblick

Aus Fehlern wird man klug...

(ALLG. GEBR.)

5.1 Fazit

In der vorliegenden Ausarbeitung wurde immer wieder auf die während der Spezifikation des Dialogsystems entstandenen Probleme hingewiesen. Die meisten dieser Probleme resultierten aus der Tatsache, daß der Dialogmanager „ariadne" während der Studienarbeit immer noch weiterentwickelt wurde, so daß zu Beginn der Arbeit der volle Funktionsumfang noch gar nicht bekannt war. Zu Beginn der Studienarbeit waren interne Komponenten, wie zum Beispiel die Generierung der Klärungsfragen, noch gar nicht oder nur sehr rudimentär implementiert und teilweise fehlerbehaftet. Dies und die angesprochenen Mißverständnisse (teils hervorgerufen durch mangelnde Dokumentation) führten zu einem System, das in der vorliegenden Fassung nur als bedingt einsatzbereit gelten kann.

Um das System zu einem funktionstüchtigen Dialogsystem für die Domäne Multimediaraum zu machen, ist es möglich einige Teile der Spezifikation zu übernehmen, andere bieten die Möglichkeit zur relativ einfachen Erweiterung oder Überarbeitung. Einige Umsetzungen von Ideen können wiederum eher als Übersicht dienen, welche Probleme entstehen können und Anhaltspunkte liefern, wie geschickter vorzugehen wäre. Die Verfasserin der Ausarbeitung versteht die Arbeit somit als Studie des Dialogmanagers „ariadne" in der Benutzung durch eine nicht mit den Interna des Managers vertraute Person, sowie als ersten Ansatz zur Erstellung eines Modells für einen Multimediaraum.

Unter Verwendung einer anderen, sinnvolleren Vorgehensweise ist es aus Sicht der Verfasserin möglich, mit dem Dialogmanger „ariadne" ein funktionstüchtiges Dialog-

system zu spezifizieren. Dabei sollte allerdings versucht werden, die Spezifikation zu modularisieren, da in der vorliegenden Arbeit deutlich wurde, daß diese auch schon bei kleineren Systemen relativ schnell unübersichtlich wird.

5.2 Ausblick für weiterführende Arbeiten

Das System läßt durchaus eine große Anzahl von Möglichkeiten der Weiter- oder auch Neuentwicklung offen. Einige grundlegende Ideen dazu sollen in diesem Abschnitt aufgezeigt werden.

5.2.1 Allgemeines

Das Dialogsystem und die Applikation sollten insgesamt realistischer gestaltet werden. Dazu ist es sinnvoll, in weiteren Arbeiten die tatsächlichen Gegebenheiten der Umgebung, sowie auch die potentiellen Nutzerinnen des Systems in stärkerem Maße zu berücksichtigen, als im vorliegenden Fall geschehen. Entsprechende Daten ließen sich aus Benutzerstudien und technischen Beschreibungen der Umgebung gewinnen.

Hilfreich als Referenz wäre zunächst einmal ein Gesamtkatalog der bereits von „ariadne" bereitgestellten Funktionalitäten mit entsprechender Beschreibung der zur Umsetzung zu verwendenden Syntax. Dies würde es ermöglichen, bereits bei der Spezifikation grundlegender Komponenten, z.B. im Domänenmodell, Vorbereitungen zu treffen, die dann später die Umsetzung bestimmter Ideen in anderen Komponenten, z.B. in den Templates, extrem erleichtern.

5.2.2 Vorgehen

Wie bereits im vorigen Abschnitt angedeutet, besteht aus Sicht der Verfasserin eine sinnvolle Vorgehensweise darin, zunächst ohne Berücksichtigung der technischen Gegebenheiten mögliche Eingaben zu sammeln, die dann auch Aufschluß über die gewünschten Funktionalitäten geben können. Dabei sollten bei einer Benutzerstudie keinerlei Vorgaben bezüglich der möglichen Anfragen gemacht werden. Die gegebene Information sollte lediglich beinhalten, daß sich die Person in einem Raum mit sehr vielen Geräten und Medien befindet und daß alle vorhandenen Geräte natürlichsprachlich gesteuert werden können. Von den Ergebnissen eines solchen „Versuchs" ausgehend, beschränkt sich das zu erstellende System nicht nur auf

a) direkte Aufforderungen an die Gerätesteuerung und

b) den Betrachtungshorizont derjenigen, die die Implementierung vornimmt.

Sicherlich ist bei einer solchen, eventuell mit einem „Wizard of Oz"-System durchzuführenden Befragung „ins Blaue hinein" mit einer großen Anzahl von nicht direkt umsetzbaren Wünschen zu rechnen, aber es fällt sicherlich einfacher, ein realistisches Dialogsystem zu erstellen.

Der nächste Schritt könnte sein, die technische Seite eines solchen Raumes (sofern schon „in Hardware" vorhanden) zu betrachten. Aus dieser Betrachtung kann dann ein entsprechendes Domänenmodell erstellt werden, wobei die Typenhierarchie dann auch bereits aus der Befragung entnommene Elemente aufgreifen kann. Darauf aufbauend könnte ein erstes Dialogsystem mit Zielen, die ebenfalls auf Wünsche der Befragten gegründet sein können, erstellt werden. An dieses sollte dann erst die Applikation angepaßt werden, die wiederum die technischen Gegebenheiten und deren Spezifikationen umsetzt. Ein soweit erstelltes System kann dann zu einer erneuten Benutzerstudie eingesetzt werden, um Verbesserungen vornehmen zu können.

Bei einer solchen Vorgehensweise stünden die Wünsche der Benutzerin mehr im Vordergrund, denn nicht der Aufruf der Funktion „Licht einschalten" ist letztlich interessant, sondern eher das Ziel „Raum erhellen", egal, ob dies durch das Einschalten des Lichts oder Öffnen der Jalousien erreicht wird. Allerdings muß natürlich trotzdem die Möglichkeit gegeben sein, einzelne Geräte direkt anzusprechen, d.h. es müßte möglich sein, Dialogziele hierarchisch zu gliedern. Solche Überlegungen erscheinen der Verfasserin allerdings nur auf Domänen anwendbar, die eine Umwelt mit veränderbaren Zuständen wiedergeben. Bei einem reinen Auskunftssystem (z.B. der in Kapitel 2 immer wieder als Beispiel herangezogenenen Bibliothek) scheinen zunächst keine sinnvollen Anwendungen für zusammenfassende Dialogziele zu bestehen. Eine genauere Überprüfung erscheint jedoch angeraten.

Weiterhin wäre es mit einer anderen Vorgehensweise eher möglich, dem natürlichen wie auch natürlichsprachlichen Dialog näherzukommen, als das mit einem System möglich ist, in dem wiederum die Technik im Vordergrund steht. Im vorliegenden Fall muß sich immer noch die Benutzerin in ihrer Denkstruktur und Sprache an das System anpassen, was mit einem Dialogsystem zur Verarbeitung natürlicher Sprache ja gerade vermieden werden soll.

Auch unabhängig von einer neuen Vorgehensweise, die eine komplette Neustrukturierung der Spezifikation mit sich bringen würde, kann natäurlich auch die vorliegende Arbeit als Grundlage für Erweiterungen und Veränderungen dienen.

5.2.3 Java–Applikation

In Abschnitt 3.3 wurde bereits vorgeschlagen, die Java–Klassenstruktur entsprechend der Typenhierarchie des Domänenmodells zu modellieren. Eine exakte Entsprechung wird hier natürlich nicht möglich sein, da Java Mehrfachvererbung nicht im gleichen Maße wie „ariadne" ermöglicht. Zudem würde jede zukünftige Änderung in einer der beiden Komponenten auch Änderungen in der jeweils anderen nach sich ziehen. Dies ist allerdings auch bei einer relativ flachen Hierarchie der Fall, wenn eine Erweiterung der Merkmale in der Typenhierarchie des Domänenmodells auch eine änderung der Funktionalitäten (also auch der Zustandsbeschreibungen) der Java–Applikation nach sich zieht. Insgesamt erscheint es der Verfasserin also sinnvoll, eine so weit wie möglich konsistente Struktur zu erzeugen, die auch in der Java–Applikation die Geräte eher nach ihren Eigenschaften klassifiziert, als nach ihren Namen. Eine vollkommen restriktive Vorgabe, beide Modellierungen exakt gleich zu gestalten, wäre aus den oben angeführten Gründen wiederum nicht wünschenswert.

5.2.4 Domänenmodell

Die Typenhierarchie des Domänenmodells könnte dahingehend erweitert werden, daß alle Objekte, d.h. hier im Wesentlichen die Geräte mit ihrer vollen Funktionalität attributiert und damit dann auch die möglichen Aktionen entsprechend erweitert werden. Zum Beispiel beschreibt die vorliegende Spezifikation für das Objekt OBJ_PLAYER noch nicht alle Funktionen, die ein Videorecorder tatsächlich zur Verfügung stellt. Dabei sollte auch noch die Überlegung miteinbezogen werden, die Konzepte wirklich so speziell wie nur irgend möglich zu machen, um letztlich mehr Weltwissen bereits im Dialog halten zu können. Auf der anderen Seite wäre eine Entsprechung der Klassenhierarchie der Applikation zum Domänenmodell ebenfalls wünschenswert. Bei Veränderungen der Struktur des Domänenmodells ist allerdings mit Vorsicht vorzugehen, da kleine Änderungen an anderer Stelle (z.B. in der Grammatik) ebenfalls Auswirkungen haben.

Eine das System deutlich vereinfachende Änderung wäre die Aufhebung der Fehlmodellierung, die durch die auf Seite 24 beschriebene falsche Annahme entstanden ist. Hier wäre eine Lösung, in die Aktionskonzepte zur Veränderung von Eigenschaften noch die zu verändernde Eigenschaft als Merkmal hineinzunehmen, beispielsweise in der folgenden Form:

```
desc act_changeProperty inherits act_withOneObject {
    prp_degree     : DEGREE;
    act_increase   : INCREASE;
    prp_changeable : PRP_CHANGE;
};
```

In den Grammatikregeln kann damit eine Konvertierung der Eingabe `brighter` auf `{ INCREASE|INC true PRP_CHANGE|NAME "brightness" }` erreicht werden. So kann ein über ACT_CHANGEPROPERTY identifizierbares Dialogziel CHANGEPROPERTY eine entsprechende Funktion der Applikation mit dem zusätzlichen Parameter *property* aufrufen.

5.2.5 Dialogziele

Eine deutlich übersichtlichere Struktur der Dialogziele ließe sich durch eine Zusammenfassung der Dialogziele zur Veränderung verschiedener Eigenschaften erreichen. Dazu ist die im vorigen Abschnitt erwähnte Änderung der Aktionskonzepte notwendig. Es ist allerdings sicherlich nicht sinnvoll, alle Ziele zu einem einzigen allgemeinen Veränderungsziel zu vereinigen, sondern es erscheint hilfreich, eine Unterteilung in Ziele, die die Position oder Ausrichtung eines Gerätes, und solche, die eine abstrakte Eigenschaft (Helligkeit, Lautstärke) verändern, vorzunehmen. Der Vorteil zusammengefaßter Ziele liegt in der besseren Lesbarkeit und damit Wartbarkeit des Dialogsystems. Je mehr einzelne Einträge und Formulierungen zu berücksichtigen sind, desto anfälliger für Fehler wird das System. Eine vollständige Zusammenfassung verschiedener Ziele zu einem einzigen macht allerdings die Angabe von passenden, allgemeingültigen Templates schwierig, da sich häufig keine neutrale Satzkonstruktion für eine Klärungsfrage finden läßt. Hier erscheint also ein Weg „in der Mitte" sinnvoll, bei dem wie bereits vorgeschlagen, Dialogziele thematisch zusammengefaßt werden.

Entsprechend der Änderung der Dialogziele müßte dann die Schnittstelle der Java-Applikation umgestaltet werden, sowie natürlich auch die Grammatik und die Klärungsfragen (siehe auch die folgenden Abschnitte).

5.2.6 Grammatik

Grundsätzlich müßte die Grammatik noch mit Hilfe einer Datensammlung erweitert werden, um alternative Formulierungen zu berücksichtigen. Zudem müßte eine Anpassung der Konvertierung in den Grammatikregeln erfolgen, falls vorher die Dialogziele umgestaltet worden sein sollten. Die bei der oben vorgeschlagenen Änderung der Ziele mit großer Wahrscheinlichkeit nötige Mehrfachkonvertierung von Merkmalen für ein Symbol der Grammatik würde folgendermaßen (o.ä.) definiert werden können:

```
act_changeProperty = ...'brighter'
                     { INCREASE|INC true PROPERTY "brightness" }
                     ...
```

Generell sollte in die Grammatik noch wesentlich mehr freier Dialog aufgenommen werden, der nur am Rande mit der Erfüllung der Dialogziele zu tun haben muß. Dies wurde bereits in den Vorschlägen zur Vorgehensweise erläutert, wenn es aber nur um eine Erweiterung des Systems geht, können auch hier noch deutlich mehr Elemente eingebracht werden. Die Änderungen hier würden sich dann natürlich gegebenenfalls auf die Typenhierarchie und die Templates auswirken. Die Typenhierarchie wäre insofern betroffen, als unter Umständen neue Konzepte eingeführt werden müßten, die unabhängig von den Dialogzielen für die Templates und Klärungsfragen verwendet werden können. Alleine hier wird deutlich, wie verwoben die einzelnen Teile miteinander sind und was die Hinzunahme eines einzigen Satzes, der vielleicht zunächst nicht berücksichtigt wurde, für Auswirkungen haben könnte.

5.2.7 Klärungsfragen

Bei den Templates für die Klärungsfragen wäre es wünschenswert, noch die Möglichkeit, auf eine Frage mit „ja" oder „nein" zu antworten hinzuzunehmen. Grundsätzlich sollte dies möglich sein, ließ sich jedoch in der vorliegenden „ariadne"–Version noch nicht durchführen. Weiterhin sollte die Option aufgenommen werden, bei Auflistungen eine Indexangabe als Antwort zu akzeptieren, was ebenfalls bisher nicht durchführbar war, in einer noch in der Entwicklung befindlichen Version von „ariadne" aber möglich sein wird.

Die oben angeführten Veränderungen in Dialogzielen und Grammatik würden natürlich auch in den Templates Veränderungen notwendig machen. So müßten etliche Templates zu einem (oder entsprechend dem in Abschnitt 5.2.5 gemachten Vorschlag zu zwei) Dialogziel(en) zusammengefaßt und die Fragen entsprechend generalisiert werden.

Literaturverzeichnis

[1] M. Denecke
An Integrated Development Environment for Spoken Dialogue
Proceedings of the Workshop on Toolsets in NLP, Coling 2000, Luxembourg.

[2] M. Denecke
Object-oriented techniques in grammar and ontology specification
Proceedings of the Workshop on Multilingual Speech Communication (MSC-2000), pp 59-64, Kyoto, Japan, October 2000.

[3] M. Denecke
Informational Characterization of Dialogue States
International Conference on Speech and Language Processing, Beijing, China, October. 2000.

[4] M. Denecke, A. Waibel
Dialogue Strategies Guiding Users to their Communicative Goals
Proceedings of Eurospeech 97.