Sonderforschungsbereich 378
Ressourcenadaptive kognitive Prozesse

KI-Labor am Lehrstuhl für Informatik IV

Leitung: Prof. Dr. W. Wahlster

REAL

Universität des Saarlandes
FB 14 Informatik IV
Postfach 151150
D-66041 Saarbrücken
Germany

Tel. 0681 / 302-2363

Memo Nr. 68

Das REAL Speech Interface

Christian A. Müller

Januar 1999

Sonderforschungsbereich 378
Ressourcenadaptive
Kognitive Prozesse

ISSN 0944-7822

68

Das REAL Speech Interface

Christian A. Müller
SFB 378 – Projekt REAL
Fachbereich Informatik
Universität des Saarlandes
D–66123 Saarbrücken
cmueller@cs.uni-sb.de

19. Januar 1999

Inhaltsverzeichnis

Zusammenfassung

In der vorliegenden Arbeit wird ein System zur Verarbeitung gesprochener Sprache vorgestellt, das im Rahmen des Projektes REAL (REssourcenAdaptierende Lokalisation) (Sonderforschungsbereich 378) am Fachbreich Informatik der Universität des Saarlandes entwickelt wurde. Es besteht aus voneinander unabhängigen Sprachein- bzw. ausgabekomponenten und stellt eine Sprach-Schnittstelle des natürlichsprachlichen Dialogsystems RONTRA (RObot´s Natural language TRAnslator) dar. Die Spracheingabekomponente wurde auf der Basis des *IBM VoiceType Development Toolkits* entwickelt, das im Rahmen des *Solution Developer Program* von der Firma IBM für die Entwicklung sprachgesteuerter Anwendungen zur Verfügung gestellt wird. Bei der Sprachausgabekomponente handelt es sich um ein konkatenatives Synthesesystem auf Wortebene, d.h. die Ausgabe wir aus einzelnen Wörtern zusammengesetzt, die zuvor per Mikrophon aufgezeichnet wurden.

1 Einführung

Die Ein- und Ausgabe technischer Systeme wie z.B. autonome Roboter ist für den Benutzer in der Regel weder intuitiv, noch leicht verständlich. Natürlichsprachliche Schnittstellen dienen als Übersetzer zwischen den engen Eingabekanälen des Systems und den breitbandigen menschlichen Kommunikationskanälen [KW89]. Ein Nachteil, den sie mit sich bringen, ist der hohe Schreibaufwand beim Eingeben der natürlichsprachlichen Befehle. Gesprochene Sprache ist für den Menschen die natürlichste Art zu kommunizieren, der Aufwand beim Sprechen ist gegenüber dem Schreiben vergleichsweise gering.

Seit langem wird an der Entwicklung von Spracherkennungs- und -generierungssystemen gearbeitet. Was die Spracheingabe betrifft, geht es in erster Linie um Faktoren wie Sprecher- und Domänenunabhängigkeit, Robustheit, und natürlich die Trefferquote bei der Erkennung. Sprachausgabesysteme werden vor allen Dingen gemessen an der Ähnlichkeit der synthetischen Sprache mit der eines Menschen. Bei der Spracheingabekomponente, die in der vorliegenden Arbeit vorgestellt wird, handelt es sich um ein robustes System mit hoher Trefferquote, das auf eine ganz spezifische Domäne zurechtgeschnitten ist. Die Ausgabekomponente ist ebenfalls fest mit der Anwendung verbunden, für die sie entwickelt wurde und generiert verständliche und natürlich klingende Äußerungen.

Beschrieben wird das System RESI (REal Speech Interface), das als Adaption für gesprochene Sprache für das natürlichsprachliche Dialogsystem RONTRA (RObot's Natural language TRAnslator) entwickelt wurde [Sto98]. Diese Systeme entstanden im Rahmen des Projektes REAL (REssourcenAdaptierende Lokalisation) (Sonderforschungsbereich 378) am Fachbereich Informatik der Universität des Saarlandes. REAL beschäftigt sich mit Raumbeschreibungen, die unter Ressourcenbeschränkungen gegeben werden. RONTRA ist ein System zur Analyse und Umsetzung natürlichsprachlicher Instruktionen in Roboterbefehle. Es umfaßt außerdem die Generierung von Erläuterungen während der Ausführung der Befehle durch den Roboter, die Generierung von Erklärungen beim Auftreten eines Fehlers, und den Austausch von Umgebungsinformationen zwischen Mensch und Roboter.

2 Spracheingabe

Die Spracheingabekomponente unterstützt die Eingabe von fließend gesprochenen Sätzen, die Verifizierung des erkannten Strings, und die Übertragung des Ergebnisses an das REAL Grundsystem.

Entwickelt wurde das System auf der Basis des *IBM VoiceType Development Toolkits*, das im Rahmen des *Solution Developer Program* von der Firma IBM für die Entwicklung sprachgesteuerter Anwendungen zur Verfügung gestellt wird.[1] Die Funktionsweise des *IBM VoiceType Development Toolkit* und die Vorgehensweise bei der Entwicklung entsprechender Anwendungen wird in Abschnitt 2.1 beschrieben. Abschnitt 2.2 beschreibt die Entwicklung der RESI Spracheingabekomponente als *Command and Control*-Anwendung. In Abschnitt 2.3 wird die Bedienung der aktuellen Implementierung der Spracheingabekomponente erläutert.

2.1 Das IBM VoiceType Development Toolkit

Das *IBM VoiceType System* unterstützt zwei verschiedene Arten der Spracherkennung. Sogenannte *Command and Control*-Anwendungen können mit gesprochenen Befehlen gesteuert werden. Im Gegensatz dazu gibt es die Möglichkeit der Entwicklung von *Dictation*-Anwendungen, bei denen der Benutzer unter Verwendung einer großen Anzahl von Wörtern Texte mit dem Mikrophon erfassen kann. Aufgrund der Größe des Lexikons laufen *Dictation*-Anwendungen allerdings lediglich im *isolated speech* Modus, d.h. der Sprecher muß zwischen den einzelnen Wörtern eine Pause machen.[2] *Command and Control*-Anwendungen laufen im *continuous speech* Modus, d.h. die Eingabe kann in einem natürlichen Tempo gesprochen werden, ohne daß Pausen zwischen den Wörtern gemacht werden müssen.

Ein wichtiger Faktor bei der Spracherkennung ist die Sprecherabhängigkeit. Bei einem sprecherabhängigen System ist es erforderlich, daß der Benutzer das System explizit auf seine individuelle Stimme anpaßt. Moderne Spracherkennungsysteme können ohne großen Trainingsaufwand von verschiedenen Sprechern genutzt werden. Zukunftsweisende Forschungssysteme wie Verbmobil [Wah97] sind sprecheradaptiv, d.h. können sich an den individuellen Sprecher anpassen. *VoiceType* ist ein sprecherunabhängiges System, bietet jedoch eine optionale *enrollment* Funktion (siehe Abschnitt 2.1.1), mit der sich ein Sprechertraining durchführen läßt. Dadurch kann die Erkennungsleistung insbesondere im Diktiermodus entscheidend verbessert werden.

2.1.1 Die *Speech Recognition Engine*

Das Herzstück des Spracherkennungssystems ist die *Speech Recognition Engine*. Sie erkennt die Sprachdaten, übersetzt sie in einen Text-String, und überträgt diesen an

[1] siehe http://www.developer.ibm.com/

[2] Mit der aktuellen Version des IBM Spracherkennungssystems *ViaVoice* ist auch im *Dictation*-Modus Eingabe im *continuous speech* Modus möglich

die Anwendung. Die Anwendung hat über ein API (*application programming interface*) Zugriff auf die *Speech Recognition Engine*. Beim *IBM VoiceType Development Toolkit* heißt diese Schnittstelle *Speech Manager API* oder kurz SMAPI (siehe Abbildung 1) . Mithilfe der SMAPI-Schnittstelle kann die Spracheingabefunktion von verschiedenen Anwendungen genutzt werden.

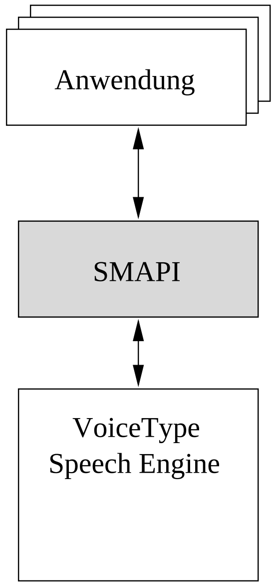

Abbildung 1: Die Anwendung kommuniziert über das *Speech Manager Application Programming Interface*(SMAPI) mit der *Speech Recognition Engine* .

Die *Speech Recognition Engine* verwendet folgende Ressourcen zur Verarbeitung gesprochener Sprache:

- Lexikon
- Wortnutzungsmodelle
- Aussprachevarianten
- Ein Stimm-Modell

Das Lexikon, die Wortnutzungsmodelle, und die Aussprachevarianten bilden die sogenannte Domäne der Anwendung. Diese Daten werden von der *Speech Recognition Engine* gebraucht, um die gesprochene Eingabe für die Anwendung zu dekodieren. Das *VoiceType Development Toolkit* enthält eine „allgemeine Bürodomäne“, die zwischen 20.000 und 30.000 Wörter umfaßt. Andere Domänen wie z.B. „Radiologie“, „Journalismus“, „Notfall-Medizin“ sind ebenfalls erhältlich. Diese können für die Entwicklung von *Dictation*-Anwendungen verwendet werden. Für *command and control*-Anwendungen stellt das *VoiceType Development Toolkit* verschiedene Tools bereit, mit denen der Entwickler anwendungsspezifische Domänen erstellen kann.

Ein Lexikon besteht aus einer Liste von Wörtern oder Phrasen, mit denen die Eingabedaten der *Speech Recognition Engine* verglichen werden. Meist gehört zum Lexikon ein Wortnutzungsmodell, das statistische Informationen bereitstellt, mit denen die Wahrscheinlichkeit bestimmter Wortfolgen gewichtet werden kann. In *VoiceType* gibt es drei verschiedene Lexikonarten, die von der *Speech Recognition Engine* unterschiedlich behandelt werden.

- Ein Befehlslexikon (engl. *command vocabulary*) wird verwendet, um Wörter und Phrasen aus einer vordefinierten Liste zu erkennen, die dynamisch zur Laufzeit erstellt wird. Befehlslexika unterstützen den *continuous speech* Modus und können z.B. zur Implementierung von einfachen sprachgesteuerten Menüs verwendet werden. Die Dekodierung der Wörter basiert rein auf deren Klang. Eine Gewichtung der Wahrscheinlichkeit durch ein Wortnutzungsmodell erfolgt nicht: jedes Wort und jede Phrase in der Liste hat bei der Auswahl die gleiche Wahrscheinlichkeit.

- Ein Grammatiklexikon (engl. *grammar vocabulary*) wird verwendet, um Wörter und Phrasen zu erkennen, die in einer kompilierten Sprachgrammatik enthalten sind. Die Sprachgrammatik wird bei der Entwicklung der Anwendung erstellt. Auch Grammatiklexika arbeiten mit *continuous speech*. Sie sind zu bevorzugen, wenn komplexere Sprachkommandos erkannt werden sollen.

- Ein Diktierlexikon (engl. *dictation Vocabulary*) wird zur Erkennung von freiem Text verwendet, der im *isolated speech* Modus eingegeben werden muß. Das Wortnutzungsmodell besteht dabei aus einer Datenbank mit Wortfolgen, die in der entsprechenden Sprache häufig vorkommen. Durch die Repräsentation von Wörtern im Kontext (vorangehendes und nachfolgendes Wort) ermöglicht das Wortnutzungsmodell eine Unterscheidung akustisch gleicher Wörter.

Aussprachevarianten eines Wortes sind seine möglichen phonetischen Repräsentationen. Bei manchen Wörtern ist es sinnvoll, mehrere Aussprachevarianten zu festzulegen, damit die Erkennungsleistung möglichst unabhängig vom Sprecher ist. Das *VoiceType Development Toolkit* enthält einen *Dictionary Builder*, mit dem ein lexikonspezifisches Wörterbuch erstellt werden kann, das alle Aussprachevarianten der Lexikoneinträge enthält.

VoiceType bietet die Möglichkeit, optional ein persönliches Stimm-Modell aufzubauen. Beim sogenannten *enrollment* nimmt der Benutzer eine bestimmte Anzahl von Mustersätzen auf, auf deren Basis das Stimm-Modell des Benutzers aufgebaut wird. Es enthält Benutzereigenschaften wie z.B. den Akzent und akustische Variablen wie Hintergrundgeräusche und den Mikrophontyp.

2.1.2 Architektur des *VoiceType* Spracherkennungssystems

Abbildung 2 stellt die Architektur des *VoiceType* Spracherkennungssystems dar. Sie zeigt die Hauptkomponenten der *Speech Recognition Engine* und die Interaktion mit einer Anwendung über SMAPI.

Der Akustik-Prozessor Der Akustik-Prozessor wandelt die rohen Klangdaten in ein internes Format um. Er besteht aus zwei Komponenten, dem *Signalprozessor* und dem *Etikettierer*.

Der Signalprozessor verarbeitet die Mikrophonsignale die mit 11KHz Samplerate aufgenommen werden. Da diese Signale aus Daten und Nebengeräuschen bestehen, muß der Signalprozessor an die aktuelle akustische Umgebung angepaßt werden, um den Einfluß der Nebengeräusche möglichst zu reduzieren. Die Ausgabe des Signalprozessors ist eine Menge von Merkmalen, die eine Hundertstel Sekunde der Eingabefolge repräsentieren. Die Merkmale sind zweidimensional (Energiemenge

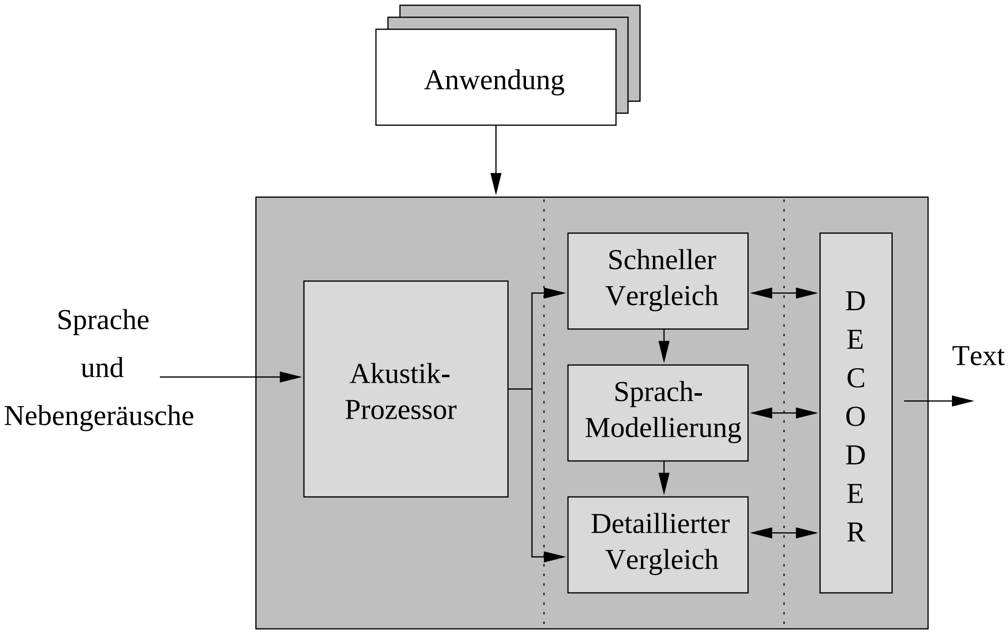

Abbildung 2: Die Hauptkomponenten der *Speech Recognition Engine* und die Interaktion mit einer Anwendung über SMAPI

pro Frequenzbereich). Die Summe der Merkmale stellt einen Vektor dar, der die Parameter der Eingabesequenz repräsentiert (siehe Abbildung 3).

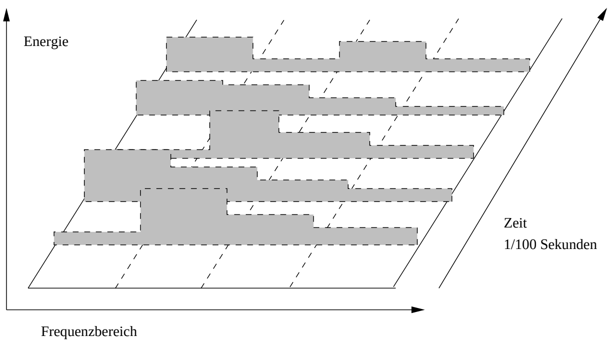

Abbildung 3: Der Merkmalsvektor repräsentiert die Parameter der Eingabesequenz

Der Etikettierer konvertiert die Ausgabe des Signalprozessors in eine Folge sogenannter Etiketten. Diese werden mit Prototypen verglichen und den verschiedenen Klangkategorien zugewiesen, die in etwa den Phonemen der gesprochenen Sprache entsprechen. Falls der Benutzer die *enrollment* Funktion genutzt hat, werden statt der Standardprototypen die individuellen Prototypen aus dem *enrollment script* verwendet.

Das Word Matching Die nächste Phase des Spracherkennungsprozesses wird von der *word matching* Komponente übernommen. Sie identifiziert die in Frage kommenden Wörter und verteilt die Wahrscheinlichkeiten auf der Basis der akustischen Analyse und des Kontextes. Dieser Vorgang besteht aus drei Schritten. Beim *fast* *match* wird ein approximativer Vergleich der Eingabe mit allen Wörtern im Vokabular durchgeführt und eine Menge von möglichen Kandidaten gebildet. Bei der Sprachmodellierung werden die Wahrscheinlichkeiten der Wortfolgen analysiert, unabhängig von ihrer akustischen Form. Dadurch kann die Größe der Kandidatenmenge reduziert werden. Bei Befehlslexika ist das Sprachmodell uniform, d.h. jedes Wort im Lexikon hat die gleiche Wahrscheinlichkeit. Bei einem grammatikbasierten Lexikon wird das Sprachmodell durch eine reguläre Grammatik.[3] Beim dritten Schritt, dem detaillierten Vergleich, wird ein genauerer akustischer Vergleich mit der reduzierten Kandidatenmenge durchgeführt. Dieser Vorgang ist ressourcenintensiver als der schnelle Vergleich, da das verwendete akustische Modell eine feinere Granularität hat.

Der Decoder Die letzte Komponente der *Speech Recognition Engine* ist der *decoder*. Er berechnet die Wahrscheinlichkeit von Wortfolgen (partielle und vollständige Sätze) auf der Basis der Bewertungen von Akustik- und Sprachmodellierung und identifiziert den wahrscheinlichsten vollständigen Satz. Dieser Satz wird an die Anwendung zurückgegeben.

2.1.3 Verwendete *VoiceType* Konfiguration

In den vergangenen Abschnitten wurden die verschiedenen Modi des *VoiceType Development Toolkits* vorgestellt, die zur Entwicklung einer Anwendung zur Verfügung stehen. Für das RESI-System wurde die Variante des **grammatikbasierten Lexikons** gewählt. Normale Befehlslexika sind für die Implementierung von Anwendungen gedacht, die einfache gesprochene Kommandos erkennen können. Zur Verarbeitung von komplexen Kommandos, wie sie für RONTRA benötigt werden, sind sie jedoch nicht geeignet. Diktierlexika mit vordefinierten Domänen wiederum sind für die Erkennung von freiem Text ausgelegt und bieten für diese spezifische Anwendung keine zufriedenstellende Erkennungsleistung.

Die optionale *enrollment* Funktion wurde nicht verwendet, da das System nicht auf einen Benutzer zugeschnitten werden sollte. Stattdessen wurde versucht, die angestrebte Erkennungsleistung durch einen geeigneten Entwurf der zugrundeliegenden Sprachgrammatik zu erzielen. Genaueres darüber folgt im Abschnitt 2.2.2.

2.2 Entwicklung der RESI Spracheingabekomponente als *Command and Control*-Anwendung

In Abschnitt 2.1 wurden die verschiedenen Möglichkeiten beschrieben, mit dem *VoiceType Development Toolkit* ein Spracherkennungssystem zu entwickeln. Wir werden nun sehen, wie bei der Implementierung des RESI-Systems vorgegangen wurde. Im einzelnen werden folgende Entwicklungsschritte beschrieben:

- Analyse der Anwendungsdomäne
- Entwicklung der Sprachgrammatik und des Lexikons
- Kompilierung der Sprachgrammatik

[3] siehe [HU88] für eine Einführung in die Sprachenklassifizierung nach Chomsky und die Funktionsweise Endlicher Automaten

- Aufbau des Aussprachewörterbuchs
- Test der Sprachgrammatik und des Lexikons
- Entwicklung der Anwendungsschnittstelle

2.2.1 Analyse der Anwendungsdomäne

Zu Beginn der Entwicklung eines Spracheingabe-Systems steht die Analyse der Anwendungsdomäne. Untersucht werden unter anderem die Größe des Lexikons, und die zu erwartende Vielfalt der grammatischen Konstruktionen. Implementierung des System stehen für Die Domäne KAMRO und CHEW zur Verfügung. KAMRO (Karlsruher Autonomer Mobiler ROboter) ist ein mobiler Zweiarm-Roboter, der zur Durchführung von Serviceaufgaben entwickelt wurde (siehe Abbildung 4). Innerhalb des CHEW-Szenarios (CHemical Exchange Warehouse) [Fei97] wurde eine Robotersimulation entwickelt, der Chemikalien transportiert, mit denen umzugehen für einen Menschen zu gefährlich ist.
Die Eingabekomponente der natürlichsprachlichen Schnittstelle für diese Roboter übernimmt folgende Aufgaben [LLH+94], [Sto98]

- Auftragsspezifikation: der Benutzer kann natürlichsprachliche Befehle auf beliebigem Abstraktionsniveau formulieren, die die Bewegungen des Roboters steuern. Die Befehle können nicht nur in der Imperativform gegeben werden (Sätze 1 a,b), sondern auch in der Deklarativform (Sätze 1 c,d). Die Objektbezeichnungen können durch beschreibende Ausdrücke ersetzt werden, wodurch die Konstruktion unter Umständlich erheblich komplexer wird (Sätze 1 e,f).
- Rückfragen: der Benutzer kann vom Roboter Ausführungserläuterungen oder Erklärungen zur Fehlerbehebung erhalten, indem er z.B. eine entsprechende Äußerung formuliert (Sätze 1 g,h,i).
- Austausch von Umgebungsinformationen: Roboter und Benutzer können sich gegenseitig unterstützen indem sie Informationen austauschen. Der Mensch kann das System z.B. nach der Position von Objekten fragen (Satz 1 j).

(1) a. Baue das cranfield-assembly-benchmark sofort zusammen.
  b. Bewege deinen rechten Manipulator 15 Zentimeter nach vorne.
  c. Das Pendel soll in den Schaft eingepaßt werden.
  d. Du sollst das Zwischenstück in die rechte Seitenplatte stecken.
  e. Drehe die Seitenplatte, die sich vor dem Loch des Schafts befindet, um 85 Grad nach rechts.
  f. Nimm den hinteren linken Bolzen und stecke ihn in den vorderen Bolzenbehälter.
  g. Erkläre deine ganzen Aktionen.
  h. Sage mir die Ursache des Fehlers.
  i. Wodurch kam der Fehler zustande?
  j. Wo befindet sich der Schaft ?

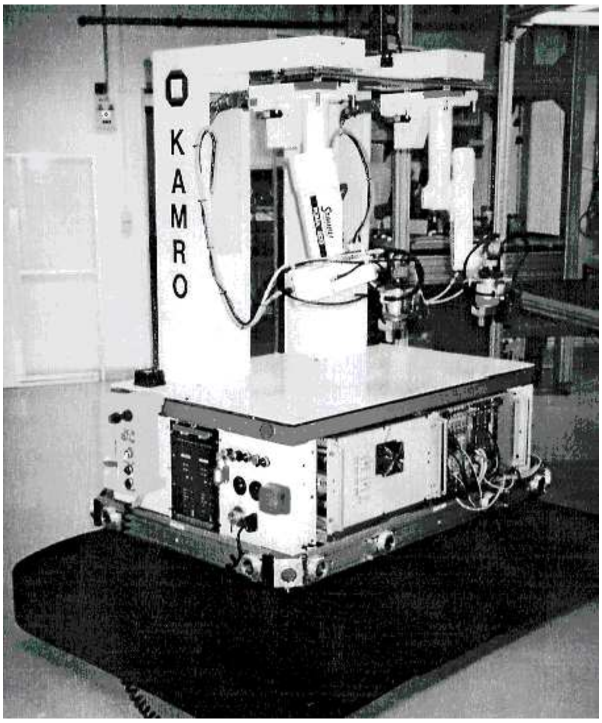

Abbildung 4: KAMRO: Karlsruher Autonomer Mobiler ROboter

2.2.2 Entwicklung der Sprachgrammatik und des Lexikons

Die Sprachgrammatik stellt ein Sprachmodell (siehe Abschnitt 2.1.2) für die *Speech Recognition Engine* dar. Sie enthält eine Beschreibung der Syntax der erlaubten Phrasen und Sätze. Der Vorteil der Verwendung eines solchen Sprachmodells ist die Verbesserung der Erkennungsleistung und die Reduzierung der benötigten Rechnerkapazität. Die Vielfalt der möglichen Eingaben wird jedoch auf die Phrasen und Sätze beschränkt, die von der Sprachgrammatik erkannt werden. Um ein optimales Verhältnis zwischen Erkennensleistung und Eingabefreiheit zu erreichen, sollte die Sprachgrammatik möglichst genau die Menge der Sätze der entsprechenden Domäne erkennen. In Fall der aktuellen Implementierung des RESI-Systems bedeutet dies die Menge der KAMRO Befehle und Rückfragen.

Zur Erstellung der Sprachgrammatik wird die Sprache SRCL (*Speech Recognition Control Language*) verwendet. SRCL ist eine Logiksprache, die wiederverwendbare Muster und substituierbare Parameter unterstützt. Wenn z.B. das Symbol ⟨ZIFFER⟩ für die Repräsentation der Ziffern von 0 bis 9 verwendet wird, kann eine dreistellige Zahl durch ⟨ZIFFER⟩⟨ZIFFER⟩⟨ZIFFER⟩ definiert werden.

SRCL entspricht einer an die Spracherkennung angepaßten Bakus-Naur Form.[4] Die allgemeine Form einer SRCL Grammatik ist eine Produktion, die aus vier Teilen besteht:

- Linke Seite (⟨ Regelkopf ⟩): eindeutige Bezeichnung der Regel
- Zuweisungsoperator (=)
- Rechte Seite (⟨ Sätze und Phrasen ⟩): gültige Sätze und Phrasen.

[4] für eine umfassende Erläuterung der SRCL Syntax siehe [IBM97]

- Produktionenbegrenzer (Punkt)

SRCL Grammatiken sind reguläre Grammatiken, d.h. ihre Mächtigkeit ist die von endlichen Automaten. Abbildung 5 zeigt eine vereinfachte Darstellung der RESI Grammatik in Automatenform. Die Übergänge, die in Großbuchstaben etikettiert sind, repräsentieren komplexe, wiederverwendbare Elemente der Grammatik. Beispielhaft wird das Element „OBJEKT IM AKKUSATIV“ detailliert dargestellt. Die hervorgehobenen Zustände und Übergänge zeigen die Verarbeitung des Satzes „Bewege den Stift 25 cm nach vorne“. Ein Satz gilt als erkannt, wenn der Automat am Ende der Eingabekette einen Zielzustand (Kreis mit doppelter Linie) erreicht hat. Die Grammatik beschreibt genau Menge der Eingabesätze der zugrundeliegenden natürlichsprachlichen Schnittstelle. Sie wird in Anhang A) vollständig dargestellt.

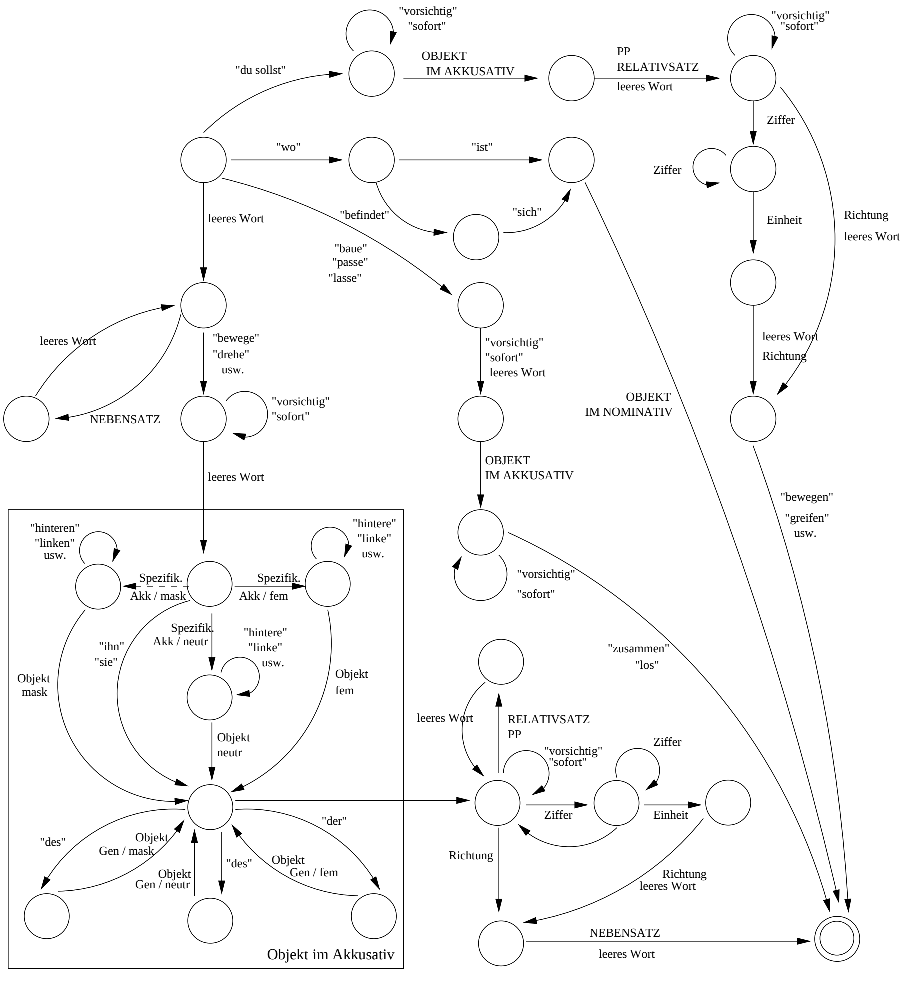

Abbildung 5: Eine vereinfachte Darstellung der RESI Grammatik in Automatenform

Die Sprachgrammatik kann auch als gerichteter azyklischer Graph (engl. directed acyclic graph, DAG) dargestellt werden (siehe Abbildung 6). Die Wurzel des DAGs ist die obligatorische Regel ⟨ root ⟩, die Menge der Blätter bildet das Lexikon. Für jedes Wort im Lexikon muß mindestens eine Aussprachevariante mit dem *dictionary builder* aufgenommen werden (siehe Abschnitt 2.2.4).

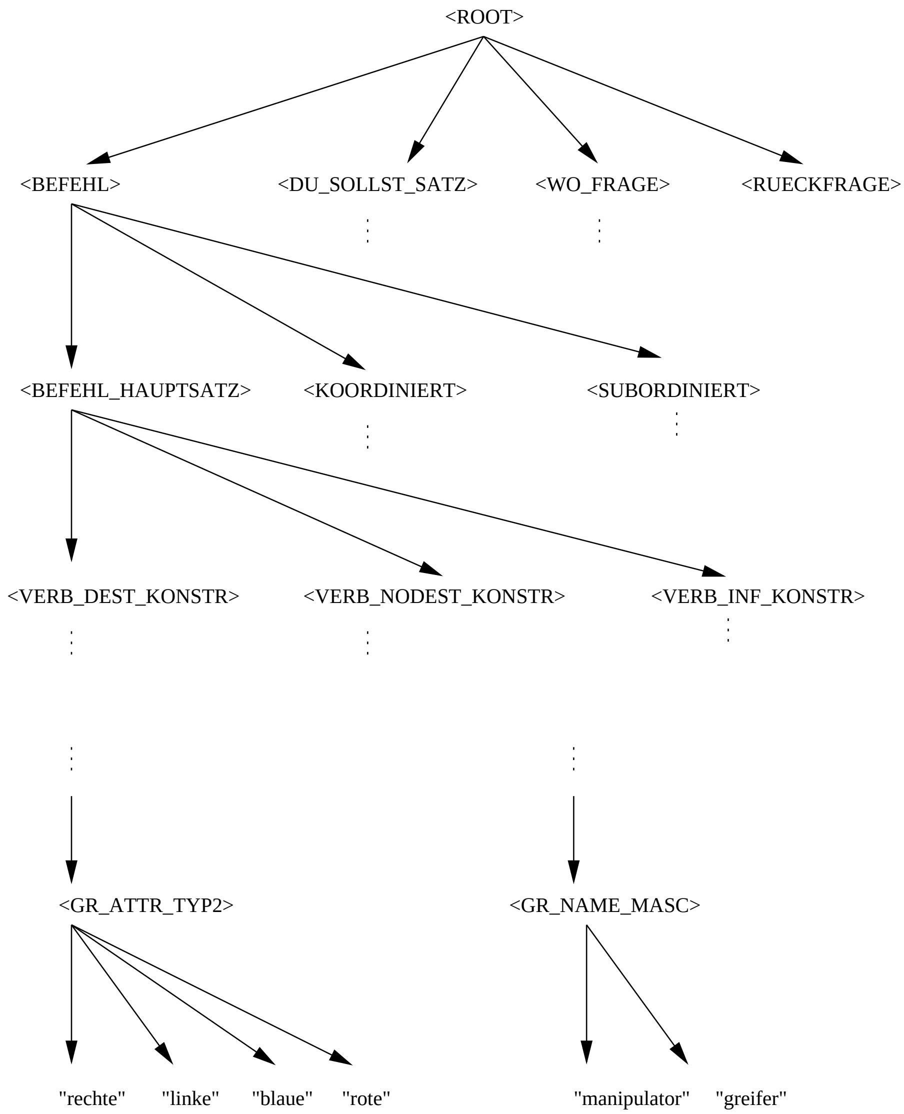

Abbildung 6: Bei der DAG-Repräsentation der Sprachgrammatik bilden die Blätter das Lexikon.

2.2.3 Kompilierung der Sprachgrammatik

Der *grammar compiler* konvertiert die Sprachgrammatik, die in SRCL definiert wurde, in eine binäre Datei mit der Erweiterung „FSG“ (finite state grammar). Diese wird zur Laufzeit von der *Speech Recognition Engine* verwendet, um die Wörter und Phrasen zu bestimmen, die der Benutzer zum aktuellen Zeitpunkt eingeben kann.

Der *grammar compiler* unterstützt drei Optionen. Die *probabilistic computation* Option schaltet die probabilistische Berechnung an bzw. aus. Im Default-Modus ist probabilistische Berechnung abgeschaltet, d.h. alle Wörter sind in einem bestimmten Zustand gleich wahrscheinlich. Der Unterschied in der Erkennungsleistung zwischen an- und abgeschalteter probabilistischer Berechnung hängt stark von der Grammatik ab. Ein Beispiel:

(2) a. ⟨Zahl⟩ = „Punkt“ ⟨Ziffer⟩|⟨Ziffer⟩

b. ⟨Ziffer⟩ = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8| 9

Das erste Wort, das gesprochen wird, ist „Punkt“ oder eine Ziffer. Mit angeschalteter probabilistischer Berechnung haben alle initialen Wörter die Wahrscheinlichkeit 1/11. Bei nicht-probabilistischer Berechnung hat „Punkt“ die Wahrscheinlichkeit von 0,5, während die Ziffern jeweils eine Wahrscheinlichkeit von 0,05 haben.

Die *mumble words* Option des *grammar compilers* steuert die Behandlung von nichtsprachlichen Äußerungen („Gemurmel“). Wenn die *mumble words* Option gesetzt ist, ignoriert die *Speech Recognition Engine* jegliche Daten, die nicht in der Sprachgrammatik definiert sind.

Mit der *silence words* Option können Sprechpausen behandelt werden. Wenn diese Option gesetzt ist, ignoriert die *Speech Recognition Engine* alle Pausen während der Eingabe.

Die Syntax des *grammar compilers* wird in [IBM97, Kapitel 7] genauer beschrieben.

2.2.4 Aufbau des Ausspracheworterbuchs

Wenn die Grammatik entwickelt und kompiliert ist, muß ein Wörterbuch erstellt werden, das für alle Wörter des Lexikons mindestens eine Aussprachevariante enthält. Dieses Wörterbuch dient der *Speech Recognition Engine* zur Identifikation der Wörter aus der Eingabesequenz. Das *VoiceType Development Toolkit* enthält einen *dictionary builder* (siehe Abbildung 7), mit dem sich das Aussprachewörterbuch automatisch generieren läßt. Der *dictionary builder* extrahiert die Wörter aus dem Lexikon und sorgt dafür, daß für jedes Wort mindestens eine Aussprachevariante vorliegt. Die genaue Funktionsweise des *dictionary builder* wird in [IBM97, Kapitel 8] erklärt.

2.2.5 Testen der Sprachgrammatik und des Lexikons

Solange die Spracheingabekomponente des RESI-Systems noch nicht implementiert war, wurde die Sprachgrammatik mit dem *grammar test tool* des *VoiceType Development Toolkits* getestet (siehe Abbildung 8). Das *grammar test tool* stellt eine Verbindung zur *Speech Recognition Engine* her, in dem es eine Anwendung simuliert,

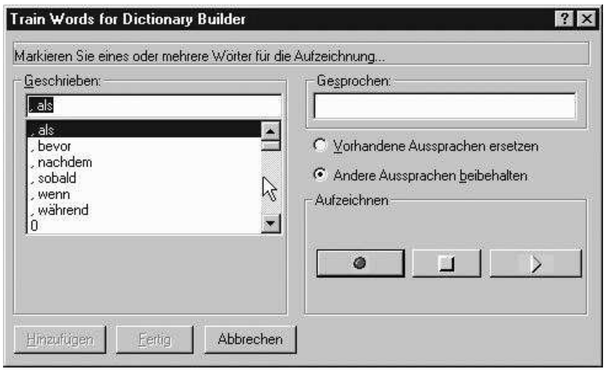

Abbildung 7: Der Dictionary Builder

und bietet so dem Entwickler der Grammatik die Möglichkeit, das Sprachmodell zu verbessern. Das *grammar test tool* wird in [IBM97, Kaptitel 9] beschrieben.

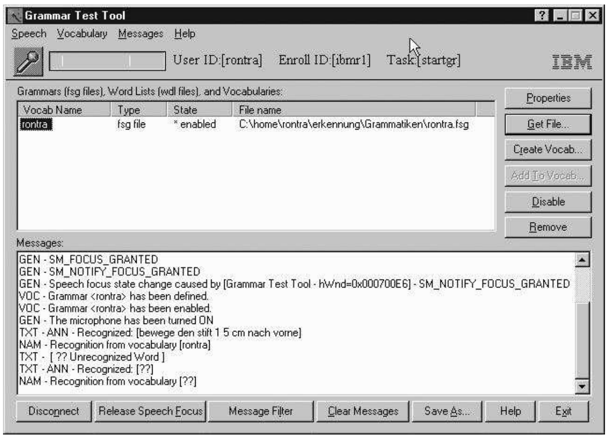

Abbildung 8: Das Grammar Test Tool

2.2.6 Entwicklung der Anwendungsschnittstelle

Die Spracheingabekomponente des RESI-Systems wurde unter Verwendung der SMAPI Schnittstelle (siehe Abschnitt 2.1.1) in Visual C++ implementiert.[5] Als Grundlage diente dabei der Quellcode des mitgelieferten Beispielprogramms „reco“, von dem die Prozeduren zur Aktivierung und Steuerung der *Speech Recognition Engine* übernommen wurden. Die Oberfläche des Programms und die Verarbeitung des erkannten Strings wurden an die gegebenen Anforderungen angepaßt.

In der aktuellen Version des Systems wird der erkannte String zunächst evaluiert und dann über eine TCP/IP Verbindung an RONTRA übermittelt. In Anhang B finden sich einige kommentierte Auszüge aus dem Quellcode. Es werden speziell einige Passagen aufgeführt, die die Anbindung an die *Speech Recognition Engine* und

[5] Die Implementierung erfolgte mit dem Microsoft Developer Studio, da die zugrundeliegende Version des *VoiceType Development Toolkits* lediglich C++ Anwendungen unter dem Betriebssystem „Microsoft Windows NT“ unterstützte. Von Seiten der Firma IBM ist jedoch eine Java Schnittstelle geplant.

die Verarbeitung des Strings realisieren. Die Entwicklung einer Anwendungsschnittstelle mit SMAPI wird in [IBM97, Kapitel 10] beschrieben.

2.3 Bedienung der Spracheingabekomponente

Die Spracheingabekomponente (siehe Abbildung 9) nimmt gesprochene Roboter-Kommandos und -Rückfragen entgegen, wandelt sie in einen Text-String um, und sendet den Text über eine TCP/IP Verbindung an RONTRA. Hauptfunktion ist dabei die Verbindung des Eingabegerätes (Mikrophon) mit der *VoiceType Speech Recognition Engine* (siehe Abschnitt 2.1.1) und der Sprachgrammatik (siehe Abschnitt 2.2.2). Nach dem Starten des Programmes muß zunächst die Sprachgrammatik ausgewählt werden.

Nach dem Auswählen und Laden der Grammatik wird die *Speech Recognition Engine* mit „Start“ aktiviert. Der Vorgang der Erkennung kann jederzeit mit dem „Stop“ Button unterbrochen werden. Die Eingabe sollte zügig und deutlich gesprochen werden, wobei sich der Sprecher an die gültigen Sätze der Grammatik halten muß. Sind diese Bedingungen gegeben, läßt sich auch ohne Anpassung des Systems an den Sprecher eine recht hohe Trefferquote erzielen. Wird ein ungültiger oder unvollständiger Satz gesprochen, so wird das Ergebnis mit einem Fragezeichen markiert.

Nach der Eingabe eines Satzes, wird das Ergebnis mit dem Button „Eingabe verstanden ?“ evaluiert und über eine TCP/IP Verbindung an das RONTRA System übermittelt. Danach kann das System mit „Start“ erneut aktiviert werden.

Abbildung 9: Die Bedienungsoberfläche der RESI Spracheingabekomponente

3 Sprachausgabe

Die Sprachausgabekomponente wandelt die Dialogbeiträge des Roboters, die von RONTRA in einen natürlichsprachlichen Ausgabestring übersetzt wurden, in gesprochene Sprache um, und gibt sie über einen Lautsprecher aus. In Abschnitt 3.1 wird kurz auf die verschiedenen Techniken der Sprachsynthese eingegangen und begründet, warum die RESI Sprachausgabekomponente als Konkatenationssynthese auf Wortebene implementiert wurde. Der Aufbau des Systems wird in Abschnitt 3.2 beschrieben.

3.1 Techniken der Sprachsynthese

Nach dem derzeitigen Stand der Forschung unterscheidet man drei grundlegende Arten der Sprachsynthese: die Formant- oder Regelsynthese, die Artikulatorische Synthese, und die Konkatenationssynthese. Die Formant- oder Regelsynthese basiert auf der Überlegung, daß man zur Wahrnehmung eines Lautes mit einigen Formanten auskommt. Ein Formant wird durch die Parameter-Frequenz, Amplitude und Bandbreite repräsentiert, die in mitunter sehr komplexen Regeln kodiert werden. Die Technik zeichnet sich dadurch aus, daß ohne großen Aufwand verschiedene Stimmen produziert werden können und die Ausgangssignale wenig Speicherplatz benötigen. Die Qualität der erzeugten Laute läßt jedoch besonders im Fall von Plosiven und Frikativen zu wünschen übrig.

Bei der Artikulatorischen Synthese werden die akustischen Eigenschaften des menschlichen Vokaltraktes durch eine Kombination von Röhren modelliert. Das Ergebnis klingt oft unnatürlich.

Natürlich klingende Ausgaben lassen sich mit einer Konkatenationsynthese erreichen. Dabei werden natürlich produzierte Äußerungsteile aufgezeichnet, digitalisiert und in elementare sprachliche Einheiten zerlegt. Das synthetisierte Signal besteht aus einer Kombination der elementaren Bausteine in variierender Reihenfolge. Die Ausgabe ist allerdings auf die zuvor aufgenommene Stimme beschränkt. Problematisch kann darüberhinaus der Speicherplatzbedarf sein, der von der Art der Bausteine (Wörter, Silben, Halbsilben, Phoneme, Diphone) und der Größe des Lexikons abhängt.

Für die Implementierung der Sprachausgabekomponente des RESI-Systems wurde eine Konkatenationsynthese auf Wortebene gewählt, da diese Technik natürlich klingende Ergebnisse ermöglichte und die Größe des Lexikons die Wahl von Wörtern als Bausteine zuließ. Ein Nachteil dieser Vorgehensweise ist darin zu sehen, daß eine Anpassung des Systems an andere Domänen eine ständige Erweiterung der Klangdatenbasis erfordert. Bei einer Konkatenationsynthese auf Basis kleinerer Bausteine ist dies nicht der Fall. Ein solches System wäre dann zu einem gewissen Grad domänenunabhängig.

3.2 Aufbau der Sprachausgabekomponente

Die Implementierung der Sprachausgabekomponente erfolgte als Java-Applikation. Die Hauptfunktionen der Anwendung sind: Aufbau einer TCP/IP Verbindung mit dem Grundsystem, Verwaltung einer Pipeline mit Sprachaufträgen, Zerlegung der

Äußerungen in die Grundbausteine (Wörter), und Aufrufen der entsprechenden Klangdateien. Die Funktionen werden im einzelnen in den folgenden Abschnitten beschrieben.

3.2.1 Verbindung mit dem Grundsystem

Die Ausgabeanwendung erhält Äußerungsaufträge vom Grundsystem über eine TCP/-IP Netzwerkverbindung, die zu Beginn einer Sitzung aktiviert wird. Dazu müssen zunächst der Name des Servers und der Übertragungsport festgelegt werden. Ist die Verbindung aktiviert, wartet das System auf Äußerungsaufträge. Zu Testzwecken können diese auch manuell eingegeben werden (siehe Abbildung 10) . Implementiert wurde die Verbindung unter Verwendung der Java SOCKET Klasse. Der kommentierte Quellcode dieser Funktion findet sich in Anhang C.1.

Abbildung 10: Die RESI Sprachausgabekomponente

3.2.2 Die Auftragspipeline

Die Ausgabegeschwindigkeit der Äußerung kann durch Reduzierung der Pausen zwischen den Wörtern und Sätzen zu einem gewissen Grad variiert werden. Dennoch kommt es vor, daß während der Versprachlichung einer Äußerung bereits neue Aufträge vom Grundsystem eintreffen. Die Äußerungsaufträge werden deshalb durch eine *Pipeline* [6] geführt und der Reihe nach verarbeitet (siehe Anhang C.2). Das kann in verschiedenen Dialogsituationen zu Beeinträchtigungen führen, da die Reaktion des Benutzers auf eine Äußerung möglicherweise erst erfolgen kann, wenn sich der Zustand des Systems bereits wieder geändert hat. Dieses Problem ist jedoch nicht spezifisch, sondern ergibt sich bei allen Dialogsystemen, die mit gesprochener Sprache arbeiten, da die Sprechgeschwindigkeit nicht beliebig erhöht werden kann.

[6]Mit *Pipeline* wird hier ein Stapel bezeichnet, der nach dem *FiFo*-Prinzip arbeitet (First in - First out)

3.2.3 Tokenisierung und Versprachlichung

Kern der Sprachausgabekomponente ist die Zerlegung einer Äußerung (in den meisten Fällen ein Satz) in Wörter und die Verkettung der entsprechenden Grundbausteine. Für die Beispieldomäne KAMRO wurde zunächst ein Lexikon zusammengestellt und per Mikrophon aufgezeichnet. Die Klangdaten liegen im *Sun Audio Format* (.au) vor und dienen als Basis für die Konkatenation.

Eine prosodische Verarbeitung ist in Konkatenationssystemen auf Wortebene nur schwer möglich. Die einzelnen Wörter müßten in unterschiedlichen intonationellen und tonalen Varianten aufgezeichnet werden, was zu einem Anwachsen der Klangdatenbasis in beträchtlichem Ausmaß führen würde. Um den Aufwand der Erstellung und Speicherplatzbedarf der Klangdatenbasis möglichst gering zu halten, wurden lediglich zwei tonale Variationen der Wörter aufgezeichnet: eine neutrale Variante für Wörter innerhalb der Äußerungseinheit und eine abfallende Variante für Wörter am Ende.

Der Ablauf der Tokenisierung und des Abspielens der Klangdateien kann aus dem kommentierten Quellcode-Abschnitt in Anhang C.3 entnommen werden.

4 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein System zur Verarbeitung gesprochener Sprache vorgestellt, das aus zwei unabhängigen Komponenten besteht, nämlich einer Spracheingabe- und einer Sprachausgabekomponente. Die Spracheingabekomponente, die mithilfe des *VoiceType Development Toolkits* entwickelt wurde (siehe Abschnitt 2.1), erfüllt in vollem Umfang die Erwartungen, die daran geknüpft wurden. Sie ermöglicht die gesprochene Eingabe von komplexen Befehlen, die zuvor in einer Grammatik spezifiziert wurden. Das System arbeitet dabei sprecherunabhängig und stabil; die Befehle können im sogenannten *Continuous Speech* - Modus eingegeben werden, das heißt satz- und nicht wortweise. Es ist darüberhinaus an verschiedene Domänen anpaßbar, indem die zugrundeliegende Sprachgrammatik ersetzt bzw. erweitert wird.

Der phonologische Anspruch der aktuellen Implementation der Sprachausgabekomponente ist eher gering. Zwar ist die Ausgabe verständlich und klingt natürlich, was aus der verwendeten Methode der Wortkonkatenation resultiert (siehe Abschnitt 3.1), jedoch werden kaum prosodische und intonationelle Informationen verarbeitet. Dazu wäre es notwendig, die syntaktische Struktur des zu äußernden Satzes zu parsen. Allerdings ist es ineffizient, aus einer konzeptuellen Repräsentation zunächst einen Satz für die textuelle Ausgabe zu generieren, um ihn anschließend für die gesprochene Ausgabe wieder zu parsen. Es bietet sich an, auf *Concept-to-Speech*-Verfahren zurückzugreifen, bei denen die gesprochene Ausgabe direkt aus der konzeptuellen Repräsentation generiert wird. Dasselbe Konzept könnte somit gleichzeitig als Grundlage für die textuelle und gesprochene Ausgabe dienen.

A Sprachgrammatik

```
<ROOT> =                  <BEFEHL> | <DU_SOLLST_SATZ> | <WO_FRAGE> | <>.
```

```
<BEFEHL> =                <BEFEHL_HAUPTSATZ> | <KOORDINIERT> | <SUBORDINIERT>.

<KOORDINIERT> =           <BEFEHL_HAUPTSATZ> und <BEFEHL_HAUPTSATZ>.

<SUBORDINIERT> =          <TEMP_NEBENS_VOR>? <BEFEHL_HAUPTSATZ>
                          <TEMP_NEBENS_HINTER>?.

<WO_FRAGE> =              wo <VERB_WO> <OBJECT_NOM>.

<BEFEHL_HAUPTSATZ> =      <VERB_DEST_KONSTR> | <VERB_NODEST_KONSTR>
                          | <VERB_INF_KONSTR>.

<TEMP_NEBENS_VOR> =       <TEMP_KONJ> du <OBJ_GR_ACC> <NAEHERE_BEST>? <MODIFIER>?
                          <DEST>? <VERB_IM_NEBENS_KOMMA>.

<TEMP_NEBENS_HINTER> =    <TEMP_KONJ_KOMMA> du <OBJ_GR_ACC> <NAEHERE_BEST>?
                          <MODIFIER>? <DEST>? <VERB_IM_NEBENS>.

<DU_SOLLST_SATZ>=         "du sollst" <MODIFIER>* <OBJ_GR_ACC> <NAEHERE_BEST>?
                          <MODIFIER>* <DEST>? <VERB_INFINIT>.


<VERB_DEST_KONSTR> =      <VERB_DEST> <MODIFIER>* <GREIFER_ACC> <MODIFIER>*
                           <DEST>? | <VERB_DEST> <MODIFIER>* <OBJECT_ACC>
                          <NAEHERE_BEST>?  <MODIFIER>* <DEST>?.


<VERB_NODEST_KONSTR> =    <VERB_NODEST> <MODIFIER>* <GREIFER_ACC> <MODIFIER>*
                          | <VERB_NODEST> <MODIFIER>* <OBJECT_ACC>
                          <NAEHERE_BEST>? <MODIFIER>*.

<VERB_INF_KONSTR> =       <VERB_FERNABH> <MODIFIER>* <GREIFER_ACC> <MODIFIER>*
                          <VER_INF_ZUSATZ>
                          | <VERB_FERNABH> <MODIFIER>* <OBJECT_ACC>
                          <NAEHERE_BEST>? <MODIFIER>* <VER_INF_ZUSATZ>.


<OBJ_GR_DAT> =            <OBJECT_DAT> | <GREIFER_DAT>.

<OBJ_GR_ACC> =            <OBJECT_ACC> | <GREIFER_ACC>.


<GREIFER_ACC> =           <GR_ACC_MASC> | <GR_ACC_FEM>.

<GREIFER_DAT> =           <GR_DAT_MASC> | <GR_DAT_FEM>.

<GR_ACC_MASC> =           <DET_ACC_MASC_GREIF> <GR_ATTR_TYP1>? <GR_NAME_MASC>.

<GR_DAT_MASC> =           <DET_DAT_MASC_GREIF> <GR_ATTR_TYP1>? <GR_NAME_MASC>.

<GR_ACC_FEM> =            <DET_ACC_FEM_GREIF> <GR_ATTR_TYP2>? <GR_NAME_FEM>.

<GR_DAT_FEM> =            <DET_DAT_FEM_GREIF> <GR_ATTR_TYP1>? <GR_NAME_FEM>.
```

```
<GR_ATTR_TYP1> =          rechten | linken | blauen | roten.

<GR_ATTR_TYP2> =          rechte | linke  | blaue  | rote.

<GR_NAME_MASC> =          manipulator | greifer.

<GR_NAME_FEM> =           hand.


<OBJECT_NOM> =            <OBJECT_NOM_MASC> <OBJ_GEN>? | <OBJECT_NOM_FEM>
                           <OBJ_GEN>? | <OBJECT_NOM_NEUTR> <OBJ_GEN>?.

<OBJECT_ACC> =            <OBJECT_ACC_MASC> <OBJ_GEN>? | <OBJECT_ACC_FEM>
                          <OBJ_GEN>? | <OBJECT_ACC_NEUTR> <OBJ_GEN>?.

<OBJECT_DAT> =            <OBJECT_DAT_MASC> <OBJ_GEN>? | <OBJECT_DAT_FEM>
                          <OBJ_GEN>? | <OBJECT_DAT_NEUTR> <OBJ_GEN>?.

<OBJECT_NOM_MASC> =       <DET_NOM_MASC_OBJ> <LOK_TYP2>* <OBJ_NAME_MASC> |
                          <OBJ_PRON_NOM>.

<OBJECT_ACC_MASC> =       <DET_ACC_MASC_OBJ> <LOK_TYP1>* <OBJ_NAME_MASC> |
                          <OBJ_PRON_ACC>.

<OBJECT_DAT_MASC> =       <DET_DAT_MASC_OBJ> <LOK_TYP1>* <OBJ_NAME_MASC> |
                          <OBJ_PRON_ACC>.

<OBJECT_NOM_FEM> =        <DET_NOM_FEM_OBJ> <LOK_TYP2>* <OBJ_NAME_FEM>
                          | <OBJ_PRON_NOM>.

<OBJECT_ACC_FEM> =        <DET_ACC_FEM_OBJ> <LOK_TYP2>* <OBJ_NAME_FEM>
                          | <OBJ_PRON_ACC>.

<OBJECT_DAT_FEM> =        <DET_DAT_FEM_OBJ> <LOK_TYP1>* <OBJ_NAME_FEM>.

<OBJECT_NOM_NEUTR> =      <DET_NOM_NEUTR_OBJ> <LOK_TYP2>* <OBJ_NAME_NEUTR>.

<OBJECT_ACC_NEUTR> =      <DET_ACC_NEUTR_OBJ> <LOK_TYP2>* <OBJ_NAME_NEUTR>.

<OBJECT_DAT_NEUTR> =      <DET_DAT_NEUTR_OBJ> <LOK_TYP1>* <OBJ_NAME_NEUTR>.

<OBJ_NAME_MASC> =         fehler | schwinger | hebel | schaft  | bolzen |
                          stift | bolzenbehälter .

<OBJ_NAME_FEM> =          fehlerursache | seitenplatte .

<OBJ_NAME_NEUTR> =        cranfield-assembly-benchmark |cab | pendel |
                          zwischenstück | loch.

<OBJ_PRON_NOM> =          er | sie | es.

<OBJ_PRON_ACC> =          ihn | sie | es.

<OBJ_GEN> =               des <LOK_TYP1>* <OBJ_NAME_GEN_MASC>
                          | der <LOK_TYP1>* <OBJ_NAME_GEN_FEM>
                          | des <LOK_TYP1>* <OBJ_NAME_GEN_NEUTR>.
```

```
<OBJ_NAME_GEN_MASC> =   schwingers | hebels | schafts  | bolzens | stifts |
                        bolzenbehälters .

<OBJ_NAME_GEN_FEM> =    fehlerursache | seitenplatte .

<OBJ_NAME_GEN_NEUTR> =  cranfield-assembly-benchmarks |cabs | pendels
                        | zwischenstücks.


<DET_ACC_MASC_GREIF> =  den | deinen.

<DET_DAT_MASC_GREIF> =  dem | deinem.


<DET_ACC_FEM_GREIF> =   die | deine.

<DET_DAT_FEM_GREIF> =   der | deiner.


<DET_NOM_MASC_OBJ> =    der | ein.

<DET_ACC_MASC_OBJ> =    den | einen.

<DET_DAT_MASC_OBJ> =    dem | einem.


<DET_NOM_FEM_OBJ> =     die | eine.

<DET_ACC_FEM_OBJ> =     die | eine.

<DET_DAT_FEM_OBJ> =     der | einer.


<DET_NOM_NEUTR_OBJ> =   das | ein.

<DET_ACC_NEUTR_OBJ> =   das | ein.

<DET_DAT_NEUTR_OBJ> =   dem | einem.



<LOK_TYP1> =            rechten | linken | oberen | unteren | vorderen
                        | hinteren | vordersten | hintersten | entfernten.

<LOK_TYP2>  =           rechte | linke | obere | vordere | hintere  // untere
                        | vorderste | hinterste | entfernte.


<VERB_WO> =             befindet sich | ist.

<VERB_DEST> =           bewege | hebe | senke | drehe | stecke | befördere
                        | positioniere | postiere | setze | stelle | kippe
                        | führe | wende | öffne | schliesse | füge | lege .
                        | bring.
```

```
<VERB_NODEST> =           nenne | greife | hole | löse | nimm | packe | suche
                          | wähle.

<VERB_FERNABH> =          baue | passe | lasse |füge | nimm.

<VER_INF_ZUSATZ> =        zusammen | ein | los | auf.

<VERB_REL_SATZ> =         liegt | steht | befindet.

<VERB_2PERS> =            nennst | bewegst | hebst | senkst | drehst
                          | steckst | beförderst | legst
                          | positionierst | postierst | stellst | kippst
                          | bringst | führst | wendest | öffnest | schliesst |
                          zusammenbaust | einpast | losläst |einfügst | nimmst
                          | greifst | holst | löst | packst | suchst | wählst
                          | fügst |  aufhebst .

<VERB_2PERS_KOMMA> =      nennst,| bewegst, | hebst, | senkst, | drehst,
                          | beförderst, | legst, | bringst, | steckst, | nimmst,
                          | positionierst, | postierst, | stellst, | kippst,
                          | führst, | wendest, | öffnest, | schliesst, | fügst,
                          | zusammenbaust, | einpast, | losläst, |einfügst,
                          | aufhebst, | greifst, | holst, | löst, | packst,
                          | wählst, | suchst,  .

<VERB_INFINIT> =          nennen | beheben | bewegen | heben | senken | drehen |
                          greifen | stecken | befördern | legen | bringen
                          | positionieren | postieren | stellen | kippen
                          | führen | wenden | öffnen | schliessen | fügen |
                          zusammenbauen | einpassen | loslassen |einfügen |
                          aufheben | greifen | holen | lösen | packen
                          | wählen | aufnehmen | nehmen | suchen .

<VERB_PARTIZIP> =         genannt | getan | bewegt | gehoben | gesenkt | gedreht
                          | gesteckt | befördert | gelegt
                          | positioniert | postiert | gestellt | gekippt
                          | geführt | gewendet | geöffnet | geschlossen | gefügt |
                          zusammengebaut | eingepasst | losgelassen |eingefügt
                          | genommen | aufgehoben | gegriffen | geholt | gelöst
                          | gepackt | gesucht | gewählt | aufgenommen | gebracht .

<VERB_IM_NEBENS> =        <VERB_2PERS> | <VERB_PARTIZIP> hast.

<VERB_IM_NEBENS_KOMMA>= <VERB_2PERS_KOMMA> | <VERB_PARTIZIP> hast,.

<VERB_TEMP_BEST> =        auftrat.


<DEST> =                  <DEST_TYP1> | <DEST_TYP2> | <DEST_TYP3>.

<DEST_TYP1> =             <WERT> | <WERT> <NACH_DEST> | <NACH_DEST>.

<DEST_TYP2> =             <P_DEST> <OBJECT_ACC> | zwischen <OBJECT_ACC> und
                          <OBJECT_ACC>.

<DEST_TYP3> =             um | um <WERT> | um <WERT> <NACH_DEST>.
```

```
<NACH_DEST> =             nach <EXPL_DEST>.

<P_DEST> =                in | vor | hinter | auf | neben.

<EXPL_DEST> =             vorne | hinten | links | rechts | oben | unten.

<WERT> =                  <ZIFFER>+ <EINHEIT>.

<ZIFFER> =                0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

<EINHEIT> =               mm | cm | m | grad.


<NAEHERE_BEST> =          <PP> | <REL_SATZ>.

<PP> =                    <P>  <OBJ_GR_DAT> |  zwischen <OBJ_GR_DAT> und
                          <OBJ_GR_DAT>.

<P> =                     an | bei | in | nahe bei? | vor | hinter | rechts von
                          | links von | neben | mit.

<REL_SATZ> =              <REL_PRON> sich? <PP>? <VERB_REL_SATZ>.

<REL_PRON> =              der | die | das.



<MODIFIER> =              <PRIORITY> | <CAREFUL> | <TIME>.

<PRIORITY> =              sofort | jetzt.

<CAREFUL> =               vorsichtig.

<TIME> =                  zuvor | danach.

<TEMP_BEST_PRON> =        ", als" | ", wenn".

<TEMP_KONJ> =             nachdem | während | bevor | sobald.

<TEMP_KONJ_KOMMA> =       ", nachdem" | ", während" | ", bevor" | ", sobald".

<INTERR_PRON> =           wie | was | wer | wo | welcher.

<RUECKFRAGE> =            <PRES_KONSTR> | <PAST_KONSTR> | <FUTUR_KONSTR>.

<PRES_KONSTR> =           <INTERR_PRON> lautet dein plan.

<FUTUR_KONSTR> =          <INTERR_PRON> wirst du <OBJECT_ACC>? <VERB_INFINIT>
                          <TEMP_BESTIMMUNG>?.
```

```
<PAST_KONSTR> =         <INTERR_PRON> hast du <OBJECT_ACC>? <VERB_PARTIZIP>
                        <TEMP_BESTIMMUNG>?.

<TEMP_BESTIMMUNG> =     <TEMP_BEST_PRON> <OBJECT_NOM> <VERB_TEMP_BEST>.
```

B Auszüge aus dem Quellcode der Spracheingabekomponente

B.1 Verbindung mit der *Speech Recognition Engine*

```
//---------------------------------------------------------------------------
// StartRecoEngine()
//---------------------------------------------------------------------------
// Diese Funktion initialisiert die Verbindung der Anwendung zur Speech
// Recognition Engine
//---------------------------------------------------------------------------
long CRecoDlg::StartRecoEngine()
{
        long rc;
        char *szVersion;

        rc = SmApiVersionCheck(SM_API_VERSION_STRING, &szVersion);

        if (rc == SM_RC_OK)
        {
                rc = SmOpen(0, NULL);

                if (rc == SM_RC_OK)
                {
                        int nNumArgs = 0;
                        SmArg smArgs[30];

                        SmSetArg(smArgs[nNumArgs], SmNapplicationName, "reco");
                        nNumArgs++;
                        SmSetArg(smArgs[nNumArgs], SmNrecognize, TRUE);
                        nNumArgs++;
                        SmSetArg(smArgs[nNumArgs], SmNnavigator, FALSE);
                        nNumArgs++;
                        SmSetArg(smArgs[nNumArgs], SmNwindowHandle, m_hWnd);
                        nNumArgs++;
                        SmSetArg(smArgs[nNumArgs], SmNconnectionId,
                        MY_CONNECTION_ID);
                        nNumArgs++;

                        rc = SmConnect(nNumArgs, smArgs, SmAsynchronous);
                }
        }

        return rc;
}

//---------------------------------------------------------------------------
// StopRecoEngine()
```

```
//----------------------------------------------------------------------
// Diese Funktion schliesst die Verbindung zur Speech Recognition Engine.
//----------------------------------------------------------------------
long CRecoDlg::StopRecoEngine()
{
        SM_MSG reply;
        long  rc = 0;

        rc = SmDisconnect(0, NULL, &reply);

        if ( rc == SM_RC_OK )
        {
                m_bEngineReady = FALSE;
                rc = SmClose();
        }

        return rc;
}
```

B.2 Der Erkennungsvorgang

```
//----------------------------------------------------------------------
// StartRecognizing()
//----------------------------------------------------------------------
// Initiiert den Erkennungsprozess.
//----------------------------------------------------------------------
int CRecoDlg::StartRecognizing()
{
        SM_MSG reply;
        int rc;

        rc = SmRequestFocus(&reply);

        switch (rc)
        {
                case SM_RC_FOCUS_GRANTED:
                        // Already have focus, go ahead and start listening.
                        rc = BeginListening();

                        if (rc == SM_RC_OK)
                        {
                                m_bRecognizing = TRUE;
                        }

                        break;

                case SM_RC_OK:
                        // Will begin listening after SM_FOCUS_GRANTED
                        arrives.
                        m_bRecognizing = TRUE;
                        break;

                default:
                        Display("Error on SmRequestFocus: %s, rc=%d",
```

```
                        SmReturnRcDescription(rc), rc);
        }

        return rc;
}

//---------------------------------------------------------------------------
// StopRecognizing()
//---------------------------------------------------------------------------
// Beendet den Erkennungsprozess.
//---------------------------------------------------------------------------
int CRecoDlg::StopRecognizing()
{
        SM_MSG reply;
        int rc;

        rc = SmMicOff(&reply);

        if (rc == SM_RC_OK || rc == SM_RC_MIC_ALREADY_OFF)
        {
                m_bRecognizing = FALSE;
        }
        else
        {
                Display("Error on SmMicOff: %s, rc=%d",
                SmReturnRcDescription(rc), rc);
        }

        return rc;
}

//---------------------------------------------------------------------------
//  BeginListening()
//---------------------------------------------------------------------------
//  Startet den Erkennungsvorgang.
//---------------------------------------------------------------------------
int CRecoDlg::BeginListening()
{
        SM_MSG reply;

        int rc = SmMicOn(&reply);

        if (rc == SM_RC_OK || rc == SM_RC_MIC_ALREADY_ON)
        {
                rc = SmRecognizeNextWord(&reply);

                if (rc != SM_RC_OK)
                {
                        Display("Error on SmRecognizeNextWord: %s, rc=%d",
                        SmReturnRcDescription(rc), rc);
                }
        }
        else
        {
                Display("Error on SmMicOn: %s, rc=%d",
                SmReturnRcDescription(rc), rc);
```

```
        }

        return rc;
}
```

B.3 Behandlung des erkannten Strings

```
//---------------------------------------------------------------------------
// ProcessRecognizedPhrase()
//---------------------------------------------------------------------------
// Behandelt den erkannten String.
//---------------------------------------------------------------------------
int CRecoDlg::ProcessRecognizedPhrase(SM_MSG reply)
{
        SM_WORD* pWords;
        ULONG nNumWords, nFlags;
        SM_MSG replySync;

        int rc = SmGetFirmWords(reply, &nNumWords, &pWords);

        if (rc == SM_RC_OK)
        {
                rc = SmGetPhraseState(reply, &nFlags);

                if (rc == SM_RC_OK)
                {
                        CString szRecoText = _T("");

                        if (nFlags & SM_PHRASE_REJECTED)
                        {
                                szRecoText += "??";
                        }

                        ULONG nCounter = nNumWords;

                        while (nCounter > 0)
                        {

                                szRecoText += " ";
                                szRecoText +=(LPCTSTR)pWords->spelling ;

                                pWords++;
                                nCounter--;
                        }

                                                StopRecognizing();

                            m_buttonSendto.EnableWindow(TRUE);
                            m_buttonStartReco.SetWindowText("&Start");

                        Display("Erkannt: <%s>", (LPCTSTR)szRecoText);
                        m_phrase = szRecoText;
                        UpdateData(FALSE);

                        rc = SmRecognizeNextWord(&replySync);
```

```
                if (rc != SM_RC_OK)
                {
            Display("Fehler bei SmRecognizeNextWord: %s, rc=%d",
            SmReturnRcDescription(rc), rc);
                }
        }
        else
        {
                Display("Fehler bei SmGetPhraseState: %s, rc=%d",
                SmReturnRcDescription(rc), rc);
        }
    }
    else
    {
        Display("Error on SmGetFirmWords: %s, rc=%d",
        SmReturnRcDescription(rc), rc);
    }



    return rc;
}
```

B.4 Netzwerkfunktionen

```
//----------------------------------------------------------------------
// Netzwerkfunktionen
//----------------------------------------------------------------------

bool CRecoDlg::ConnectTo() {
        struct sockaddr_in server;
        struct hostent *host;
        char *server_name = "barney.cs.uni-sb.de";
        unsigned short server_port = m_serverport;

        va_list vaArgList;

        if (!hasWinsock) return FALSE;

        Display("Serverinformationen werden ermittelt:");

        va_start(vaArgList, m_servername);
    vsprintf(server_name, m_servername, vaArgList);

        Display(server_name);

        host = gethostbyname(server_name);

        if (host == (struct hostent *) 0) {
                Display("Fehler beim Ermitteln
                der Serverinformationen -> Keine Servererbindung möglich");

                return FALSE;
        } else {
                Display("Servername: %s", host->h_name);
```

```
                Display("Serveraddr: %d", *(unsigned long *) host->h_addr);

                server.sin_family      = AF_INET;
                server.sin_port        = htons( server_port);
                server.sin_addr.s_addr = *((unsigned long *) host->h_addr);

        }

        if (!hasSocket) {
                if ((soc = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                {
                        Display("FEHLER: %d",WSAGetLastError());
                        Display("Fehler beim Erstellen eines Sockets!");

                        return FALSE;
                } else {
                  Display("Socket konnte erfolgreich erstellt werden!");

                  hasSocket = TRUE;
                }
        }

        if (!hasServerConnection && hasSocket && hasWinsock) {
        if (connect(soc, (struct sockaddr *)&server, sizeof(server)) < 0) {
                      Display("FEHLER: %d",WSAGetLastError());
                      Display("Verbindung mit Server schlug fehl!");

                        return FALSE;
                } else {
                        Display("Serververbindung errichtet!");

                        hasServerConnection = TRUE;

                        return TRUE;
                }
        }

        return TRUE;
}


bool CRecoDlg::Disconnect() {
        if (hasServerConnection) {

                shutdown(soc, 0x01);
                closesocket(soc);

                hasSocket = FALSE;
                hasServerConnection = FALSE;

                return TRUE;
        } else {
                return FALSE;
        }
}
```

```
CString CRecoDlg::preprocessMessage(CString input) {
        CString output;

        if (input.GetAt(0) == ' ' && input.GetAt(1) != 'w'
                && input.GetAt(2) != 'o') {
                output = input.Right((input.GetLength() - 1));
                output = "BEFEHL:" + output + "!%%%";
                Display("BEFEHL: %s", output);
        }

        if (input.GetAt(0) == ' ' && input.GetAt(1) == 'w'
                && input.GetAt(2) == 'o') {
                output = input.Right((input.GetLength() - 1));
                output = "FRAGE:" + output + "?%%%";
                Display(output);
        }

        if (input.GetAt(0) == ' ' && input.GetAt(1) == 'w' &&
                input.GetAt(2) == 'i' && input.GetAt(3) == 'e') {
                output = input.Right((input.GetLength() - 1));
                output = "FRAGE:" + output + "?%%%";
                Display(output);
        }

        if (input.GetAt(0) == ' ' && input.GetAt(1) == 'w' &&
                input.GetAt(2) == 'a' && input.GetAt(3) == 's') {
                output = input.Right((input.GetLength() - 1));
                output = "FRAGE:" + output + "?%%%";
                Display(output);
        }


        if (input.GetAt(0) == '?' && input.GetAt(1) == '?') {
                output = input.Right((input.GetLength() - 3));
                output = "FEHLER:" + output + "%%%";
                Display(output);
        }

        return output;
}


void CRecoDlg::sendtoConnection(const char* pszFormat, ...) {
        char szLine[1024];
        va_list vaArgList;

        va_start(vaArgList, pszFormat);
        vsprintf(szLine, pszFormat, vaArgList);
        va_end(vaArgList);

        if (hasServerConnection) {
        Display("Folgender Text wird gesendet: %s", szLine);
        send(soc, szLine, sizeof(szLine), MSG_OOB);
        }
}
```

C Auszüge aus dem Quellcode der Sprachausgabekomponente

C.1 Verbindung mit dem Grundsystem

```
class Connection extends Thread {
  // Die Klasse läuft als eigener thread
  public final static int DEFAULT_PORT = 6789;
  // vereinbarter Port
  protected int port;
  protected String host;
  protected Socket client;
  protected DataInputStream in;


  public Connection (int port, String host) {
    if (port == 0) port = DEFAULT_PORT;
    this.port = port;
    this.host = host;

    try {
      client = new Socket(host,port);
      // RESI ist der Client der Verbindung
      in = new DataInputStream(client.getInputStream());
    }
    catch (IOException e) {
      RESI.applet.addItem("Error: " + e);
    }
    RESI.applet.addItem("RESI: requesting data from port " + port);
    // mit addItem wird ein Text in das Tracefenster geschrieben
    RESI.applet.Verbindung.setBackground(Color.yellow);
    // wenn die Verbindung steht, leuchtet das Anzeigefeld gelb
    this.start();
  }
  public void run() {
  // hier beginnt der Teil der auf Daten vom Server wartet.

    String line;
    StringBuffer revline;
    int len;
    try {
      for(;;) {
        line = in.readLine();    // eine Zeile wird eingelesen
        if (line == null) break;

        Socket_Interface.pushStack(line);
        // die Zeile wird in die Auftragspipeline geschickt.
      }
    }
    catch (IOException e) { ; }
    finally { try {client.close();} catch (IOException e2) {;} }
  }
}
```

C.2 Die Auftragspipeline

```
public static void pushStack(String Satz) {
// In der Funktion pushStack() werden die Zeilen, die über die
   Socket-Verbindung übertragen werden, in die Auftragspipeline
   geschickt.


  int g = Integer.parseInt(RESI.applet.GroesseStack.getText());

  RESI.applet.Stack[Position] = Satz;
  // Der Stack ist ein String-Array. Die hereinkommende
     Äußerungseinheit wird auf eine neue Position im Feld
     geschrieben.
  RESI.applet.gesprochen[Position] = false;
  // Bei Sätzen , die noch nicht versprachlicht wurden, wird der
  "gesprochen"-Schalter auf "false" gesetzt.
  RESI.applet.addItemStack("Rontra: " + RESI.applet.Stack[Position]);
  if (Position < g) Position++;
  // der Positionsmarker wird inkrementiert.
  else RESI.applet.addItem("buffer overflow!");




}

public void run() {

  String h = RESI.applet.EingabeHost.getText();
  int s = Integer.parseInt(RESI.applet.EingabePort.getText());
  int w = Integer.parseInt(RESI.applet.Wartezeit_Satz.getText());
  MyConnection = new Connection(s,h);
  // eine neue Verbindung wird aufgebaut.

  // Folgendende Schleife durchläuft ständig die Auftragspipeline
     Wenn eine Äußerungseinheit gefunden wird, die noch nicht
     versprachlicht wurde, wird sie an die Funktion "process"
     weitergegeben. Anschliessen wird der Schalter "gesprochen"
     auf "true" gesetzt.

  while(true) {
    if (Position!=0) {
      for (int i=0; i<Position; i++) {
        if(!RESI.applet.gesprochen[i]) {
          RESI.applet.Eingabe.setText("");
          RESI.applet.Eingabe.setText(RESI.applet.Stack[i]);
          RESI.applet.gesprochen[i] = true;
          RESI.applet.process();
          try {
            w = Integer.parseInt(RESI.applet.Wartezeit_Satz.getText());
            Thread.sleep(w);
          } catch (InterruptedException e) {
            RESI.applet.addItem("Error :" + e);
          }
```

```
          }
        }
      }
    }
  }
}
```

C.3 Tokenisierung und Versprachlichung

```
void process() {  // Tokenisierung und Auswahl der Klangdateien

    Satz = preprocess(Eingabe.getText());
    // Die Äußerung wird zunächst durch einen Preprocessor geschickt,
       der alle Interpunktionen und sonstigen Zeichen entfernt, die
       nicht verarbeitet werden können. Dafür ist die Funktion
       preprocess() zuständig, die hier nicht näher beschrieben wird.

    Teile= new StringTokenizer(Satz);
    // Die Zerlegung erfolgt mit der StringTokenizer Klasse

    Integer Counter = new Integer (Teile.countTokens());
    Counter2 = (Teile.countTokens());
    if (Counter.intValue() ==  0) addItem ("No Input!");
    if (Counter.intValue() ==  1) addItem ("Just 1 Word!");
    if (Counter.intValue() > 1) addItem (Counter.toString() + " Words...");
    // Die Anzahl der Wörter der Äußerungseinheit wird ausgegeben.
    Delete.enable();
    try {
      while (Teile.hasMoreTokens()) {
        Wort = (Teile.nextToken());  // Tokenisierung der Äußerungseinheit

        if (Counter2 > 1) {
          Wort = Soundverzeichnis.getText() + "/" + Wort + "˜.au";
          // an das Wort wird die Erweiterung .au angehängt. Die Namen
          der Soundfiles entsprechen dem "Inhalt". Das Zeichen ˜ ist
          das Symbol für alle Wörter, die innerhalb einer Äußerung vorkommen.
          Sie haben einen anderen tonalen Verlauf als Wörter am Ende.

        } else Wort = Soundverzeichnis.getText() + "/" + Wort + "_.au";
          // Das Zeichen _ ist das Symbol für alle Wörter, die am Ende
          einer Äußerung vorkommen (siehe oben).

        long Bytes = getSize(Wort);
        if (Bytes == 0) {
          addItemNG(Wort);
          addItem("Missing File: "+ Wort);
          Speichern.enable();
        // Wenn keine entsprechende Klangdatei vorhanden ist, wird dies
           in einem Tracefenster vermerkt. Die Daten können später
           abgespeichert werden. Somit können fehlende Klangdateien ergänzt
           werden.
        } else  {
          SoundAusgabe.play(Wort);
        // Ist die Klangdatei vorhanden, wird sie ausgegenen.
```

```
          Dazu wird sie an die Funktion  SoundAusgabe.play()
          weitergegeben, die hier nicht
          näher beschrieben ist.
         try {
           Teiler = 100 / Integer.parseInt(Geschwindigkeit.getText());
           Thread.sleep(Bytes/Teiler);
         // Die Sprechgeschwindigkeit kann in der Konfigurationsdatei
            angegeben werden.
         } catch (InterruptedException ev) {
           addItem("Error : " + ev);
         }
       }
       Counter2--;
     }
   } catch (NoSuchElementException event) {
     addItem("Error: " + event);
   }
 }
```

Literatur

[Fei97] FEIBEL, H.: *Ein Konfigurationssystem für Planungsaufgaben in sicherheitskritischen Anwendungen*. Doktorarbeit, Technische Fakultät , Universität des Saarlandes, 1997.

[HU88] HOPCROFT, JOHN E. und JEFFREY D. ULLMAN: *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Internationale Computer-Bibliothek. Addison-Wesley, Bonn, 1988.

[IBM97] IBM CORPORATION: *IBM VoiceType Development Toolkit Programers Guide*, 1997.

[KW89] KOBSA, ALFRED und WAHLSTER, WOLFGANG (Herausgeber): *User Models in Dialog Systems*. Symbolic Computation. Springer, Berlin;Heidelberg;New York, 1989.

[LLH+94] LÜTH, T. C., TH. LÄNGLE, G. HERZOG, E. STOPP und U. REMBOLD: *KANTRA: Human-Machine Interaction for Intelligent Robots Using Natural Language*. Technical Report 104, SFB 314, Project R1, Institute of Real-Time Computer Systems and Robotics IPR, University of Karlsruhe, and Project N2 (VITRA), Universität des Saarlandes, 1994.

[Sto98] STOPP, EVA: *Natürlichsprachliche Dialoge mit mobilen Robotern auf der Basis einer ressourcenadaptiven Referenzsemantik räumlicher Umgebungen*. Doktorarbeit, Technische Fakultät, Universität des Saarlandes, 1998.

[Wah97] WAHLSTER, PROF. DR. WOLFGANG: *VERBMOBIL Erkennung, Analyse, Transfer, Generierung und Synthese von Spontansprache*. Technischer Bericht198, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, 1997.