Institut für Logik, Komplexität und Deduktionssysteme
der Universität Karlsruhe
Lehrstuhl Prof. Dr. A. Waibel

Multilinguale Spracherkennung durch Kommunikation monolingualer Systeme

Entwurf, Implementierung und Visualisierung

Diplomarbeit
von

Stefan Raschke

Betreuerin: Dipl. Inform. Tanja Schultz

1. Oktober 1999 - 31. März 2000

Ich bestätige hiermit, die vorliegende Arbeit selbstständig angefertigt und alle verwendeten Hilfsmittel vollständig angegeben zu haben.

Karlsruhe, den 31. März 2000

Stefan Raschke

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Kommunikation anhand von gesprochener Sprache stellt die wichtigste und natürlichste Methode der Verständigung für den Menschen dar. Jeder Mensch beherrscht seine Muttersprache und eventuell zusätzliche Sprachen, auch wenn er des Lesens und Schreibens dieser Sprachen nicht mächtig ist.

Trotz dieser Kenntnisse geht die Kommunikation über eine gemeinsame Sprache aber im Umgang mit Angehörigen anderer Völker oft verloren. Dies führt in einem Bestimmungsland mit unbekannter Sprache oft zu Mißverständnissen, wenn nicht sprach- und ortskundige Personen hinzugezogen werden.

Geschäftsleute und Politiker benötigen Dolmetscher, um bei Verhandlungen und Konferenzen die Informationen in oft zahlreiche Sprachen zu übersetzen. Unterhaltungssendungen benötigen Dolmetscher, um die Konversation mit den fremdsprachigen Gästen in die lokale Sprache zu übersetzen.

Dem ausländischen oder ortsfremden Besucher einer Stadt oder Sehenswürdigkeit wird mit Hilfe eines sprachgesteuerten Informationssystems ein mächtiges und intuitiv bedienbares Hilfsmittel an die Hand gegeben. Einfache Systeme an Flughäfen oder Bahnhöfen geben automatisch Auskunft zu Fahrplänen oder verkaufen Fahrkarten. Technisch aufwendigere, portable oder fahrzeugbasierte Systeme führen den Nutzer direkt und interaktiv ans Ziel, geben Wegbeschreibungen und -alternativen aus oder führen Touristen durch die Sehenswürdigkeiten ihres Reiseziels.

Die Sprachsteuerung solcher Systeme ermöglicht ihre Bedienung, ohne die Aufmerksamkeit zu stark von der primären Aktivität, sei es die Navigation im Straßenverkehr oder das Betrachten der gesuchten Sehenswürdigkeiten, abzulenken.

Offensichtlich sind diese Systeme aber nur praktisch anwendbar, wenn sie dem Anwender mehr als eine Sprache zur Verfügung stellen. Sowohl die Ein- als auch die Ausgabe muß so viele Sprachen wie nötig oder besser wie möglich zur Verfügung stellen, um den Bedürfnissen der internationalen Nutzerschaft gerecht zu werden.

Kehren wir zurück zu den Systemen, die nicht direkt verwendet werden, sondern im Hintergrund ihre nützliche Aufgabe verrichten. Rechnerbasierte Übersetzung stellt eine nützliche Ersetzung des Dolmetschens durch einen Menschen dar. Gerade bei ausschließlich rechnerbasierten Anwendungen, wie einer Videokonferenz, bietet sich diese Methode an.

Alle oben angesprochenen Systeme sind Beispiele für eine maschinelle Verarbeitung von gesprochener Sprache, im wesentlichen der Spracherkennung, der Sprachübersetzung und der Sprachausgabe, bzw. der Sprachsynthese. Die Spracherkennung ist hier die Grundlage, die allen Systemen gemein ist. Diese Systemkomponente muß vielen verschiedenen Anforderungen genügen. Ihre Erkennungsleistung muß unabhängig von der Aussprache des Anwenders sein. Die Verarbeitung der Spracheingabe muß schnell und im Hinblick auf eine portable Anwendung resourcenfreundlich sein. Die Erkennung muß mehrere Sprachen als Eingabe annehmen können, oder fähig sein, die Sprache zu identifizieren. Dies ist nötig, um sie zurückweisen zu können, falls ein entsprechendes Spracherkennungssystem nicht vorhanden ist. In dieser Arbeit werden vorhandene Theorien und Realisierungen betrachtet und durch den in Abschnitt 4 vorgestellten Ansatz erweitert.

1.2 Zielsetzung und Überblick

Diese Arbeit wurde im Rahmen des GlobalPhone Projektes erstellt. Dieses Projekt beinhaltet unter anderem monolinguale und multilinguale Spracherkenner für viele weitverbreitete Sprachen. Das Ziel der Arbeit war der Entwurf und die Implementierung eines Systems, welches automatisch eine von mehreren Sprachen als Eingabe bearbeitet. Als Basis dienen hierzu die vorhandenen monolingualen Spracherkenner.

Der hier verfolgte Ansatz unterscheidet sich von den herkömmlichen multilingualen Spracherkennern, da er weder eine vorgeschaltete Identifizierung der Sprache betreibt, noch ein explizit multilinguales System verwendet.

Durch eine Erweiterung des Suchalgorithmus der monolingualen Erkenner um eine Unterbrechungsmöglichkeit, an der bei Bedarf wieder angesetzt werden kann, wird eine Kommunikation- und Vergleichsmöglichkeit zwischen den Erkennern aufgebaut. Der erweiterte Suchalgorithmus überprüft und vergleicht während der Decodierung der Eingabe die Qualität der Hypothesen aller Systeme. Die Spracherkenner mit schlechter Bewertung werden nach und nach aus der Suche ausgegliedert. Die jeweils bis zum Vergleichspunkt bewältigten

Teilabschnitte der Suche werden weiterverwendet und sind somit nicht verloren. Die Hypothese des endgültig besten Systems wird dann als Ausgabe gewählt.

Weiterhin beinhaltet diese Arbeit eine JAVA Applikation ErDe , über die eine weitgehende grafische Kontrollmöglichkeit der Spracherkenner mit ihren zahlreichen Parametern geschaffen wurde. Hierfür wurde unter anderem die Kommunikation mit der Entwicklungsumgebung, die den Spracherkennungssystemen unterliegt, realisiert. So wurde eine Ausgabemöglichkeit der Verschriftung in einem der Eingabesprache gerecht werdenden Zeichensatz ermöglicht.

Um eine Vergleichsmöglichkeit mit anderen Erkenner zu haben, wurden die Spracherkenner des GLOBALPHONE Projektes integriert, was zur Vereinfachung von Testläufen unter Demonstrationsbedingungen führt. Bisher wurden neun monolinguale Erkenner, ein Sprachenidentifizierungssystem und ein multilingualer Erkenner mit spezieller Akustik in ErDe integriert.

2 Grundlagen der multilingualen Sprachverarbeitung

2.1 Einführung

Der Entwickler eines multilingualen Spracherkenners hat die anspruchsvolle Aufgabe, die Arbeit, die sonst in mehreren einzelnen Systemen erledigt wird, in einem universellen Erkenner zu realisieren. Ein paralleles Ablaufen von sprachabhängigen Systemen liefert zwar zumeist bessere Ergebnisse, ist aber eine sehr ressourcenunfreudliche Lösungsvariante des Problems.

Die folgenden Betrachtungen können als Verallgemeinerung der monolingualen Spracherkennung verstanden werden, da diese Ansätze die gleichen technischen Grundlagen haben.

Die in diesem Kapitel beschriebenen Grundlagen orientieren sich an den Strukturen der im GlobalPhone Projekt (Abschnitt 3.2) erstellten und in dieser Arbeit verwendeten Systeme. Die folgenden Ausführungen verstehen sich in keinem Falle als allgemeine oder gar vollständige Beschreibung der maschinellen Spracherkennung. Weiterführende und ausführlichere Einblicke in die Spracherkennung bieten [Schu95, Youn96].

2.2 Multilinguale Spracherkennung

Nach den grundlegenden Betrachtungen zur Theorie der Spracherkennung wird die Vorverarbeitung des Audiomaterials beleuchtet. Auf die Beschreibung der Wahl des Phonempools folgen Ausführungen zur akustischen Modellierung. Die Erklärungen zu den Sprachmodellen führen direkt zur Beschreibung der Decodierung. Abgerundet wird dieses Kapitel durch Ansätze zur Optimierung derselben.

2.2.1 Vorüberlegungen

Das eigentliche Problem der Spracherkennung läßt sich folgendermaßen zusammenfassen. Zu einem gegebenen akustischen Signal A muß das verwendete System die wahrscheinlichste Wortfolge W finden. Das Ziel ist es also, die bedingte Wahrscheinlichkeit $P(W|A)$ zu maximieren. $P(W|A)$ selber läßt sich aber nicht direkt berechnen. Der Umweg über den *Satz von Bayes* (Gleichung 1) mit Hilfe der Wahrscheinlichkeiten $P(A)$, $P(W)$ und $P(A|W)$ führt indirekt zu dem gewünschten Ergebnis.

Betrachtet man die drei Wahrscheinlichkeiten, kommt man zu dem Schluß, daß $P(A)$ zur Maximierung der Gleichung 1 nicht nötig ist. $P(A)$ stellt die Wahrscheinlichkeit für das beobachtete Signal A dar. Sobald die Eingabe des Sprachsignals beendet ist, sei es durch Aufnahme über ein Mikrofon oder durch Einlesen einer Datei, spielt diese Wahrscheinlichkeit für die weiteren Betrachtungen keine Rolle mehr. Um eine Maximierung von $P(W|A)$ zu erreichen, genügt es also, den Zähler der rechten Seite zu betrachten.

$$P(W|A) = \frac{P(A|W) \cdot P(W)}{P(A)} \tag{1}$$

Der Ausdruck $P(A|W)$ stellt die Wahrscheinlichkeit dar, das Signal A, gegeben die Wortfolge W, zu beobachten. In einem Spracherkennungssystem wird diese Wahrscheinlichkeit durch die akustische Modellierung (Abschnitt 2.2.3) angenähert. Die Wahrscheinlichkeit $P(W)$ bezieht sich auf die Wortfolge. Unabhängig vom eigentlichen Signal A wird hier die Wahrscheinlichkeit angegeben, mit der W beobachtet wird. Ein im Vorfeld erstelles Sprachmodell übernimmt die Aufgabe, dem Erkenner diese Wahrscheinlichkeit zu liefern (Abschnitt 2.2.4). Bevor mit der eigentlichen Erkennung begonnen werden kann, muß die Spracheingabe geeignet vorverarbeitet werden, um ein geeignetes Signal A zu bekommen.

2.2.2 Vorverarbeitung

Ein Sprachsignal enthält eine Fülle von Informationen, die für die Spracherkennung nicht notwendig sind. Zum Beispiel sind Informationen über den jeweiligen Sprecher im Signal enthalten, die zu seiner Identifizierung genutzt werden können, aber im Zusammenhang mit der reinen Spracherkennung überflüssig sind. Genauso lassen Mikrofon und Kanal ihre Spuren auf dem Signal zurück. Entfernt man diese bei der Vorverarbeitung, läßt sich das Signal kompakter darstellen. Aber auch das eigentliche Sprachsignal enthält Informationen, die entfernt werden können, um den Spracherkenner nur mit relevanten Daten zu versorgen.

Zu Beginn wird das gegebene kontinuierliche Signal tiefpaßgefiltert und durch einen A/D-Wandler in ein diskretes transformiert. Das analoge Sprachsignal enthält oberhalb einer Frequenz von 8 kHz nur noch wenige für die Spracherkennung notwendigen Informationen. Selbst wenn nur ein Frequenzband von 300 Hz bis 3,4 kHz (Telefon) verwendet wird, ist die Sprache noch verständlich. Nach *Shannons Abtasttheorem* ist es somit ausreichend, die Abtastung mit einer Frequenz von 16 kHz vorzunehmen, ohne nennenswerte Verluste beim Informationsgehalt zu verursachen.

Für die weitere Verarbeitung wird angenommen, das Sprachsignal sei innerhalb eines ausreichend kleinen Intervalls konstant. Dieses Intervall wird meist zu 10 ms gewählt. Um Informationen über vorhergehende und nachfolgende Intervalle zu erhalten, werden überlappende Fenster in 10 ms Abständen über das Signal geschoben. Um innerhalb dieser *frames* eine geeignete parametrische Repräsentation der enthaltenen Sprachinformation zu bekommen, wird das Kurzzeitspektrum berechnet. Eine Fouriertransformation liefert geeignete Spektralkoeffizienten. Durch eine Cepstralanalyse werden die Anregungskomponenten des Sprachsignals herausgerechnet und die Anzahl der Koeffizienten weiter reduziert.

Alternativ lassen sich auch Filterbänke verwenden. Es kommen u. a. Dreieck- und Trapezfilter mit unterschiedlichen Mittenfrequenzen und Bandbreiten zum Einsatz. Verwendet die Vorverarbeitung die gehörorientierte *Melskalierung* bei der Bestimmung der Filterabstände, so spricht man von *mel-Spektrumkoeffizienten*. Interessant sind dann die Kurzzeitenergien der gefilterten Signalkomponenten. Durch eine Transformation in *mel-Cepstrumparameter* lassen sich auch hier die Anregungskomponenten eliminieren.

Eine *Lineare Diskriminanzanalyse* (LDA) führt schließlich zu einer Reduktion der Dimensionalität der Eingabe. Außerdem führt diese Transformation des Merkmalraumes zu einer Vereinfachung der Klassifikation. Der Klassenzusammenhalt wird verstärkt und die Unterscheidbarkeit der einzelnen Klassen erhöht sich. Speziell im Zusammenhang mit der akustischen Modellierung durch Gaußsche Mischverteilungen (Abschnitt 2.2.3) erzielt eine LDA gute Ergebnisse ([ZhWe97, ZWFW97]).

Das Ergebnis der Merkmalgewinnung ist ein Vektor mit 8-50 Koeffizienten pro 10 ms Sprache. Im Weiteren sei A eine Folge von solchen Vektoren.

2.2.3 Akustische Modellierung

Das akustische Modell eines Spracherkennungssystems dient, wie in Abschnitt 2.2.1 beschrieben, zur Approximation von $P(A|W)$. Die Grundidee ist, die in Abschnitt 2.2.2 erhaltenen Merkmalvektoren mit Sprachuntereinheiten in Verbindung zu bringen. Der erste Schritt ist also die Aufteilung der zu erkennenden Sprachen in geeignete Untereinheiten (z. B. Phoneme), die dann in der Akustik modelliert werden können.

Um eine, für alle zu erkennenden Sprachen geeignete Modellierung zu erhalten, müssen Gemeinsamkeiten der Laute ausgenutzt werden. Die Abhängigkeiten der verschiedenen Sprachen untereinander können aus den Konventionen der *International Phonetic Association* (IPA) [IPA89], die eine Klassifizierung von internationalen Phonemen vorgenommen hat, oder anhand der computertauglichen (ASCII) Notation *Worldbet*, die James L. Hieronymus basierend auf den IPA-Konventionen erstellt hat, vorgenommen werden.

Die Phoneme, die durch ein gemeinsames Symbol dargestellt werden, können dann einer Kategorie zugeteilt werden. So erhält man eine Menge von Phonemen, die von mehr als einer Sprache geteilt werden und eine Menge, deren Elemente nur von einer Sprache genutzt werden. Das Verwenden einer gemeinsamen Akustik für alle Sprachen hat den Vorteil, daß das resultierende System weniger zu optimierende Parameter hat. So sinkt die Komplexität gegenüber einer Anzahl monolingualer Systeme. Die Modellierung wird robuster, da für die einzelnen Phoneme mehr Daten zur Verfügung stehen. Auf dieser multilingualen Basis wird dann der Erkenner aufgebaut.

Die dem Erkenner bekannten Wörter aller Sprachen, also diejenigen, die in den Trainingsmengen vorhanden sind, werden im Aussprachewörterbuch auf ihre Phonemketten abgebildet. So hat das System eine Referenz zwischen Worten und Phonemketten. Die Elemente des obigen multilingualen Phonemsets werden, da sich die Artikulation innerhalb einer solchen Einheit ändern kann, für die Modellierung in drei Segmente (*begin, middle, end*) unterteilt.

Die meisten modernen Systeme zur Erkennung kontinuierlicher Sprache verwenden *Hidden Markov Models* (HMMs), um die Modellierung mit den aus der Vorverarbeitung erhaltenen Merkmalvektoren zu verknüpfen. HMMs sind Markovketten mit Zuständen, Übergängen zwischen den Zuständen und einer Ausgabe pro Zustand, bei denen die genaue Zustandsfolge unbekannt und nur die Folge von Ausgaben sichtbar ist. Jeder Zustand produziert mit einer variablen Wahrscheinlichkeit einen der beobachteten Merkmalsvektoren. Für jeden der drei Teile des Phonems gibt es einen Zustand. Die einzelnen Zustände besitzen Schleifen, um eine unterschiedliche Ausprachedauer modellieren zu können. Der Endzustand eines Phonems verbindet sich mit dem Anfangszustand des folgenden Phonems. So lassen sich aus Phonemen Wörter und aus Wörtern Sätze bilden.

Um die Wahrscheinlichkeit bestimmen zu können mit der ein Zustand einen beobachteten Merkmalsvektor ausgibt, betrachtet man die Wahrscheinlichkeitsdichte $f(x|s)$ (Gleichung 2). Diese wird mit Hilfe von M Gaußverteilungen mit den Mittelwertsvektoren μ_{sm} und den Kovarianzmatrizen Σ_{sm} näherungsweise bestimmt. Durch die Verwendung von Mischverteilungen umgeht man die Unimodalität der Normalverteilung und kann jede beliebige Dichtefunktion annähern. Σ_{sm} und μ_{sm} werden auch als *Codebuch* eines Modells bezeichnet. Die Parameter c_{sm} nennt man *Gewichte*.

$$f(x|s) = \sum_{m=1}^{M} c_{sm} N(x|\mu_{sm}, \Sigma_{sm}) \tag{2}$$

Die Codebücher und Gewichte der Markovmodelle werden anhand der Trainingsdaten eingestellt. Dies geschieht mit dem sogenannten Viterbi-Training. Dieser Algorithmus verbessert iterativ die Parameter der Modelle, bis der Suchraum ausreichend gut dargestellt ist. Eine genaue und verständliche Einführung dieses Algorithmus findet sich in [Schu95].

Markovmodelle mit der in Gleichung 2 dargestellten Ausgabedichte nennt man kontinuierlich (CDHMM). Werden die Unterschiede zwischen den verschiedenen Modellen nur anhand der Gewichte dargestellt und dabei nur ein Codebuch verwendet, spricht man von semikontinuierlichen HMMs (SCHMM). Besitzen die HMMs diskrete Ausgabeverteilungen, spricht man entsprechend von diskreten Markovmodellen (DHMM). Die Verwendung von DHMMs erfordert einen Vektorquantisierer, um die Merkmalvektoren auf das verwendete Ausgabealphabet abzubilden. Da dieses Verfahren aber mit Informationsverlust verbunden ist, finden in der Spracherkennung hauptsächlich kontinuierliche Modelle Anwendung.

Da sich die Laute innerhalb einer Sprache mit ihrem Kontext verändern können, verwenden heutig Erkennern häufig eine kontextabhängige Modellierung. Diese Phoneme bezeichnet man als *Polyphone*. Je nach Kontextgröße spricht man dann zum Beispiel von *Triphonen* oder *Quintphonen*, die dann einen Kontext von einem bzw. zwei Phonemen haben.

Da für eine ausreichende Modellierung aller möglichen Polyphone aller Sprachen im allgemeinen, trotz der oben beschriebenen Zusammenfassung zu einem multilingualen Phonempool, nicht genügend Trainingsmaterial zur Verfügung steht, werden mehrere Polyphone durch die gleichen Codebücher modelliert. Würde man zum Beispiel 30 Phoneme als Triphone modellieren, müßten $30^3 = 27000$ Codebücher trainiert werden. Eine geeignete Ballung von Polyphonen wird mit Hilfe eines Entscheidungsbaumes erreicht. Die gestellten Fragen zum rechten oder linken Kontext des Polyphons können entweder mit "yes", "no" oder "don't know" beantwortet werden. Der Algorithmus endet, wenn eine vorbestimmte Anzahl an Ballungsräumen unterschritten ist. An jedem Knoten des Entscheidungsbaums wird diejenige Frage gewählt, die den maximalen Informationsgewinn zur Folge hat. Die Fragen beziehen sich sowohl auf phonetische als auch auf sprachenspezifische Inhalte. Für die Polyphonballungen werden dann Codebücher berechnet.

2.2.4 Sprachmodelle

Die Aufgabe der Sprachmodelle (englisch *Language Model* oder kurz: *LM*) ist es, die Wahrscheinlichkeit von Wortfolgen zu bestimmen. In Gleichung 1 entspricht dies dem Term $P(W)$. Die sogenannten *n-gramm LM* geben, basierend auf einem Trainingskorpus, die Wahrscheinlichkeit wieder, daß ein Wort gegeben seine $n-1$ Vorgänger, beobachtet wird. Um für jedes beobachtbare n-gramm eine verwertbare Abschätzung zu bekommen, benötigt man sehr große Datenmengen. Ein Korpus mit m Wörtern ermöglicht immerhin m^n n-gramme.

Verwendet ein Spracherkenner ein Sprachmodell, das aus *Trigrammen* ($n = 3$) besteht, dann nimmt die Trigrammwahrscheinlichkeit $P(W)$ einer Wortfolge W die in Gleichung 3 dargestellte Form an.

$$P(W) = \prod_{i=1}^{N} P(w_i|w_{i-1}, w_{i-2}) \tag{3}$$

Um Trigramme modellieren zu können, die in den Trainingsdaten zu selten bzw. gar nicht vorkommen und somit keine Abschätzung existiert, wird ein

Back-off Algorithmus verwendet (Gleichung 4). Wird ein Trigramm nicht gefunden, kann mit dessen Hilfe ein *Bigramm* $P(w_k|w_{k-1})$ verwendet werden. Fehlt auch dieses Bigramm, wird die gesuchte Wahrscheinlichkeit durch ein *Unigramm* $P(w_k)$ dargestellt. Um die Größenordnung der Wahrscheinlichkeiten zu erhalten, muß die Bigramm- bzw. die Unigrammwahrscheinlichkeit mit einer geeigneten Funktion normalisiert werden.

$$\hat{P}(w_k|w_{k-1}, w_{k-2}) = B(w_{k-1}, w_{k-2})P(w_k|w_{k-1}) \tag{4}$$

2.2.5 Decodierung

Dieser Abschnitt beschreibt die eigentliche Berechnung bzw. Annährung der in Gleichung 1 aufgeführten Wahrscheinlichkeit und führt die in den Abschnitten 2.2.3 und 2.2.4 gezeigten Mechanismen zusammen.

Jedes Wort im Wörterbuch eines Spracherkennungssystems wird durch eine Markovkette repräsentiert. Grundsätzlich arbeitet ein Erkenner so, daß er die Wahrscheinlichkeiten vergleicht, mit der die verschiedenen Markovketten die beobachtete Vektorfolge erzeugen. Die wahrscheinlichste Zustandsfolge bezeichnet dann die auszugebende Phonemfolge. Da es sich aber um Hidden Markov Models handelt, sind nur die erzeugten Merkmalvektoren nach außen sichtbar. Um die wahrscheinlichste Zustandsfolge für die Vektorfolge eines eingegebenen Wortes zu finden, verwendet man den *Viterbi*-Algorithmus.

Der Viterbi Algorithmus liefert diejenige Zustandsfolge, welche die Gleichung 5 maximiert. Der Algorithmus sucht für jeden Zustand den besten Vorgänger und speichert einen Verweis. Erreicht der Algorithmus den letzten Merkmalsvektor, kann die ganze Zustandskette zurückverfolgt und somit ausgegeben werden.

$$P(x_1 \ldots x_i|s_j, t_i) = P(x_i|s_j)max(P(x_1 \ldots x_{i-1}|s_j, t_{i-1}), P(x_1 \ldots x_{i-1}|s_{j-1}, t_{i-1})) \tag{5}$$

Um diesen Ansatz auf kontinuierliche Sprache auszudehnen, muß der Viterbi Algorithmus verallgemeinert werden. Da sich an jedem möglichen Wortende die Suche weiterverzweigt, müssen diese Wortübergänge markiert werden. Beim Erstellen der wahrscheinlichsten Zustandsfolge können dann mit Hilfe dieser Informationen die Einzelworte zu Sätzen verknüpft werden.

An dieser Stelle des Algorithmus kann auch Information aus den LM einfließen, da jetzt das ganze Wort bekannt ist. An Wortübergängen wird auf das wahrscheinlichste Vorgängerwort verwiesen und die Stelle des Übergangs in das aktuelle Wort vermerkt.

Damit fließen in der Berechnung die Informationen des Sprachmodells ($P(W)$) und des akustischen Modells ($P(A|W)$) zusammen und liefern eine Approximation für die Maximierung von Gleichung 1.

Aktuelle Spracherkenner verwenden, um die Fehlerrate zu minimieren, mehrere Suchdurchgänge bei der Decodierung. Um den Suchraum einer *linearen Suche* einzuschränken, wird meist eine *Baumsuche* durchgeführt. Die Baumsuche nutzt einen Wald von Allophonbäumen und weniger aufwendige Algorithmen. So existiert für jedes mögliche Wortanfangsphonem ein Baum, der alle möglichen Wörter realisiert. Die Vorteile sind eine Einsparung bei der Menge der Wortanfänge von mehreren Größenordnungen gegenüber einer linearen Suche. Der Nachteil ist, daß keine oder nur geringe Informationen des LM (siehe Abschnitt 2.2.6) einfließen können, da das Folgewort nicht bekannt ist. Die lineare Suche verwendet dann ein, durch die Baumsuche auf die wahrscheinlichsten Wörter eingeschränktes, Vokabular und aufwendigere Algorithmen [Wosz98].

2.2.6 Ansätze zur Optimierung der Decodierung

Die im Abschnitt 2.2.5 beschriebene Decodierung ist das "Herzstück" eines Erkenners. Bis zu 90% der gesamten Rechenzeit werden hier verwendet. In dieser Phase des Erkennungsvorgangs beeinflussen die eingesetzten Algorithmen sowohl die Fehlerrate als auch die benötigte Rechenzeit. Da ein Anwender eines Spracherkennungssystems weder minutenlang auf ein Ergebnis warten will, noch eine große Toleranz gegenüber einer hohen Fehlerrate mitbringt, sieht sich der Entwickler einem großen Dilemma gegenüber.

Die Algorithmen, die eine niedrigere Fehlerrate liefern, erhöhen im allgemeinen den Aufwand, während Methoden, die den Erkennungsvorgang beschleunigen, eine negative Auswirkung auf die Qualität der Ausgabe haben. Im folgenden werden kurz Methoden und Algorithmen vorgestellt, die also entweder Aufwand oder Fehlerrate verringern.

Lookaheads
Mit *Unigramm Lookaheads* und *Delayed Bigramms* [WoFi96] werden bereits während der eigentlichen Decodierung Informationen des Sprachmodells in die

Baumsuche eingebracht. Normalerweise ist es nicht möglich, ein Sprachenmodell anzuwenden, da das momentane Wort nur als Bruchstück vorliegt. Die einzige Information, die hier einfließen kann, ist die Unigrammwahrscheinlichkeit für alle möglichen Worte. Sobald die Suche in einem Blatt angekommen ist, werden Bigramm Sprachenmodelle angewandt, um das wahrscheinlichste Vorgängerwort zu bestimmen.

Phonem Lookaheads [Wosz98] schränken den Suchraum der Baumsuche ein. Viele mögliche Hypothesen bekommen bereits zu Beginn der Suche eine so schlechte Bewertung, daß es sehr unwahrscheinlich ist, daß sie verwertbare Ergebnisse liefern. Um diese Teilhypothesen bereits vor den aufwendigen Algorithmen der eigentlichen Suche und damit schon vor dem Pruning auszuschließen, verwendet man diese Methode.

Die Hypothesen werden einige Frames lang mit einer vereinfachten Akustik getestet. Diese Akustik verwendet kontextunabhängige HMMs und wenige Gaußverteilungen pro codebuch. Der resultierende Wert für die Teilhypothese wird zum bisherigen Score gezählt und alle Hypothesen, die einen festgelegten Wert unterschreiten, werden in der Baumsuche weiterverfolgt. Für die Baumsuche bedeutet dies weniger Knoten und weniger zu berechnende Ausgabewahrscheinlichkeiten.

Ausgabewahrscheinlichkeiten
Die Menge der verwendeten Vektoren einzuschränken, ist die naheliegenste Möglichkeit, den Aufwand bei der Berechnung der Ausgabeverteilungen zu reduzieren.

Bei der Berechnung des Score wird der Abstand zur nächsten Gaußverteilung verwendet. Die Berechnung kann abgebrochen werden, wenn der Abstand der ersten paar Koeffizienten bereits das bisher beste Resultat überschreitet. Dies erreicht man, wenn man eine *Ordnung der Verteilungen* [Wosz98] einführt. Bei jeder Berechnung wird derjenige Vektor an die erste Stelle der Bearbeitungsliste gestellt, der bisher das beste Resultat erzielt hat. Dies bricht die Berechnung bei den meisten Vektoren frühzeitig ab, da das beste Resultat bereits bei Beginn der Berechnung erreicht wird.

Eine weitere Methode ist eine Verschlankung der Codebücher durch *Entfernen von Vektoren* [Wosz98]. Werden Vektoren zur Abstandsberechnung um ein Vielfaches seltener als der am häufigsten verwendetet Vektor herangezogen, kann er entfernt werden.

Um festzustellen, ob ein Vektor der beste Kandidat bei der Abstandsberechnung ist, genügt es, eine Teilmenge der Koeffizienten zu betrachten.Dies erreicht man durch eine *Reduktion der Dimensionalität* [Wosz98]. Die eigentliche Berechnung muß aber mit allen Koeffizienten stattfinden, um die Größenordnung und Vergleichbarkeit der Bewertung zu erhalten.

Die Idee hinter *Bucket Box Intersection (BBI)* [FrRo96] ist, mit Hilfe eines vorberechneten Binärbaumes schnell eine Teilmenge von Vektoren zu finden, die dann in die Abstandsberechnung einfließen. Die folgenden Erläuterungen des Algorithmus orientieren sich an der zweidimensionalen Fassung (Abbildung 1). Um jeden Ellipsoid, der durch eine Gaußverteilung definiert ist, wird ein Rechteck gelegt. Um den Baum aufzubauen, wird der Raum nach und nach durch Hyperebenen geteilt. Die Ebenen werden so gewählt, daß in jedem resultierenden Teilraum die gleiche Anzahl von Rechtecken liegt. Dabei sollen so wenig Rechtecke wie möglich aufgeteilt werden. Der Algorithmus, der den Baum erstellt, wird gestoppt, sobald eine festgelegte Anzahl von Vektoren pro Blatt erreicht ist. Bei der Berechnung der Bewertung wird anhand des Baums der entsprechende Teilraum gesucht. Die sich in diesem Blatt (*Bucket*) befindenden Vektoren werden zur Berechnung herangezogen.

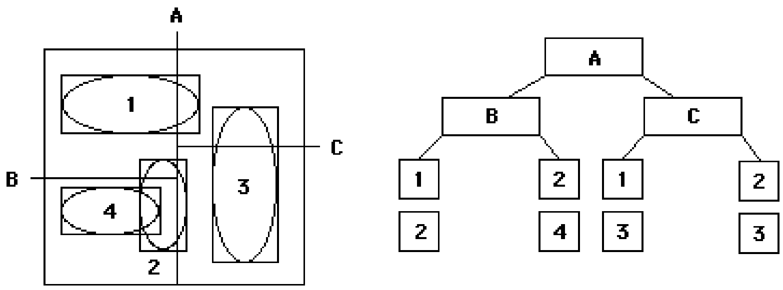


Abbildung 1: Bucket Box Intersection

MLLR

Die *Maximum Likelihood Linear Regression* (MLLR) [LeWo95] dient zur beschränkten Transformation eines sprecherunabhängigen Systems in ein sprecher- bzw. umgebungsabhängiges System. Diese Methode wird angewandt, um Kanal- und Umgebungseinflüße auszuschließen. MLLR basiert auf einem trainierten, sprecherunabhängigen System, dessen akustisches Modell mit einer geringen Datenmenge an einen neuen Sprecher angepaßt wird.

Die Adaption basiert auf der linearen Transformation der Mittelwerte der einzelnen Gaussverteilungen. Die nötigen Transformationsmatrizen erhält man, indem man die Daten des neuen Sprechers und die alten Daten auf stochastische Abhängigkeiten untersucht (*forward-backward* - Algorithmus).

Je weniger Adaptionsdaten vorhanden sind, desto geringer ist die Abdeckung der bestehenden Modelle. Um trotzdem eine geeignete Transformation zu erhalten, werden die Matrizen auf mehrere, verbundene Verteilungen angewandt. Bei sehr wenig Datenmaterial kann auch eine globale Transformation stattfinden, bei der die aus wenigen Trainingsdaten erhaltene Transformationsmatrix auf alle Modelle angewandt wird.

VTLN
Mit Hilfe der *Vocal Tract Length Normalization* (VTLN) [ZWFW97, ZhWe97] wird der Einfluß der unterschiedlich beschaffenen Vokaltrakte der Sprecher auf die Spracherkennung reduziert. Die Länge des Vokaltraktes spielt dabei die größte Rolle. Um eine Beeinflussung des Erkennerergebnisses weitgehend auszuschließen, wird das Spektrum jeder Eingabe mit einem *Warpfaktor* auf einen *Standardsprecher* abgebildet.

Diesen Warpfaktor erhält man, indem man die decodierte Eingabe mit einer Reihe von Warpfaktoren abbildet und dann mit Hilfe eines Maximum Likelihood Kriteriums bewertet.Um den Rechenaufwand gering zu halten, kann die Berechnung auf die stimmhaften Phoneme der Spracheingabe beschränkt werden. Der Warpfaktor, der das beste Ergebnis liefert, wird dann für die Abbildung verwendet.

2.3 Sprachenidentifizierung

Ein System zur automatischen Sprachenidentifizierung wird oft als vorverarbeitende Komponente verwendet. Mit ihrer Hilfe wird entschieden, an welches weiterverarbeitende System die Eingabe geleitet wird. Sei es ein entsprechendes Spracherkennungssystem oder ein Mensch aus Fleisch und Blut, der zum Beispiel telefonische Anfragen entgegennimmt. Auch in dieser Arbeit wird ein Identifizierungssystem, wie es in Abschnitt 2.3.1 beschrieben wird, zur Entscheidung über die Weiterleitung an einen entsprechenden Spracherkenner verwendet.

2.3.1 Phonemerkenner

Da jeder Sprache ein individueller Phonempool zugeordnet ist, können diese zur Identifizierung herangezogen werden. Bei einem Phonemerkenner werden die markierten Phoneme aller Sprachen in einer Akustik zusammengefaßt und der Erkenner trainiert. Beim Erkennungsvorgang werden dann die erkannten Phoneme nach den Sprachen sortiert und die Sprache mit den meisten erkannten Phonemen als Ergebnis ausgegeben. Es ist im allgemeinen nicht nötig, die ganze Satzlänge zur Identifikation zu verwenden, was den Vorgang beschleunigt. Ein solches System wurde auch bei dieser Arbeit verwendet. Dieses System wird in Abschnitt 3.2.2 näher erläutert.

2.3.2 Weitere Ansätze zur Sprachenidentifizierung

Für die Lösung des Problems der automatischen Sprachenidentifizierung gibt es weitere Ansätze, die aber im Verlaufe dieser Arbeit keine Anwendung fanden. Der Vollständigkeit wegen finden sie im folgenden kurz Erwähnung. Eine genauere Erläuterung dieser Ansätze kann in [ZiBe99] nachgeschlagen werden.

Die älteste Methode der Sprachenidentifizierung ist der Vergleich von *Sprachcharakeristiken im Spektralbereich*. Bei dieser Methode werden Beispiele von charakteristischen Lauten jeder Sprache gespeichert. Die Eingabe wird dann abschnittsweise mit den gespeicherten Beispielen verglichen. Die Sprache, deren gespeicherte Beispiele am häufigsten mit den Abschnitten übereingestimmt haben, wird als Ergebnis ausgegeben.

Basierend auf der menschlichen Fähigkeit, Sprachen anhand von Rhythmus, Tonlage, Dauer und Lautstärke zu identifizieren, wurden entsprechende maschinelle Systeme entworfen. Außer der Tatsache, daß diese Systeme in verrauschter Umgebung besser arbeiten als spektralbereich basierte, spielen sie in modernen Systemen keine Rolle.

Ein *wortbasierter Sprachenidentifizierer* liegt im Aufwand zwischen den Phonemerkennern und dem unten beschriebenen Ansatz mit Hilfe vollständiger Spracherkennungssysteme. Jede Eingabe wird mit allen sprachabhängigen Phonemerkennern bearbeitet. Für die erkannten Phonemfolgen werden mit Hilfe von Wörterbüchern Worte hypothetisiert. Es werden also zuerst Phoneme, Worte und dann die Sprache erkannt.

Der aufwendigste Ansatz ist die Verwendung von *vollständigen Spracherkennern* für jede mögliche Sprache. Jeder Erkenner liefert eine Hypothese mit Bewertung ab. Die dem Erkenner mit der besten Hypothese entsprechende Sprache wird ausgegeben.

3 Arbeitsumgebung

3.1 Janus

Janus ist ein Sprachübersetzungssystem, das in den *Interactive Systems Laboratories* der *Universität Karlsruhe* und an der *Carnegie Mellon University* in Pittsburgh entwickelt wurde. Das Übersetzungssystem besteht aus mehreren Komponenten. Für die Durchführung der vorliegenden Arbeit wurde die Spracherkennugskomponente dieser Entwicklungsumgebung verwendet. Im weiteren bezeichnet "Janus RTk" diese Komponente ([ScWW98]).

Das Janus RTk dient als Forschungsumgebung auf dem Gebiet der Spracherkennung. Das Hauptaugenmerk liegt deshalb auf leichter Erweiterbarkeit, hoher Flexibilität und unkomplizierter Handhabung. Dies wird durch eine objektorientierte Implementierung in *C* und durch einen integrierten *Tcl/TK* Interpreter erreicht. In seiner jetzigen Form existiert dieses System seit 1995, wurde aber ständig erweitert und verbessert.

Innerhalb des Systems sind vielfältige Architekturen für die unterschiedlichen Komponenten eines Spracherkenners realisiert, um die erwähnte Flexibilität bei der Erstellung eines Erkenners zu erreichen. Dies erlaubt es dem Anwender, variantenreiche Systeme zu erstellen. Das Janus RTk bietet z.B. alle Arten von HMMs zum Aufbau der Akustik oder eine Kombination mit neuronalen Ansätzen zur Realisierung der Ausgabewahrscheinlichkeiten.

Um einen Spracherkenner aufzubauen, verwendet der Entwickler sogenannte Objekte. Diese repräsentieren die einzelnen Komponenten des Erkenners und werden innerhalb des Janus RTk erzeugt. Objekte können z. B. Aussprachewörterbücher, Sprachmodelle oder HMMs sein. Die Objekte bauen aufeinander auf oder interagieren. Auf diese Weise lassen sich vielfältige Konstrukte realisieren.

Alle benötigten Objekte können direkt interaktiv erstellt und deren Methoden aufgerufen werden. Die Tcl/TK Schnittstelle erlaubt es, einen Erkenner schnell und effizient anhand von vordefinierten Skripten zu erstellen. Sie ermöglicht auch die grafische Ausgabe von Datenstrukturen und Sprachsignalen.

Werden an den Erkenner weitere Ansprüche gestellt, so erlaubt es der objektorientierte Ansatz der Implementierung des Janus RTk, dem Anwender neue

Module ohne größeren Aufwand hinzuzufügen. So kann die Funktionalität des Systems kontinuierlich erweitert werden.

3.2 GlobalPhone

Im Rahmen des Forschungsprojekts **GlobalPhone** an der Universität Karlsruhe [ScWa98, ScWa99(2)] werden unterschiedliche Vorgehensweisen bei der *Large Vocabulary Continous Speech Recognition* (LVCSR) für verschiedene Sprachen analysiert. Das Hauptziel des **GlobalPhone** - Projektes ist der Aufbau eines multilingualen Spracherkenners, der so viele Sprachen wie möglich in einem System vereint. Weitere Aufgaben sind Erstellung und Optimierung einzelner monolingualer Erkenner und eines Systems zur Identifizierung von Sprachen. Eine Beschreibung der einzelnen Systeme des GlobalPhone Projektes erfolgt in Abschnitt 3.2.2.

Um obige Aufgaben erfüllen zu können, wurde eine multilinguale Datenbasis (Abschnitt 3.2.1) erstellt. In den Tabellen 1 und 2 ([ScWa99(2)]) ist die Menge der zur Verfügung stehenden Trainingsdaten und ihre Abdeckung der Weltbevölkerung aufgelistet. Die Daten, die für des Training der Spracherkenner verwendet werden, entsprechen 80% der gesamten Datenbasis.

3.2.1 Datenbasis

Die Datenbasis des **GlobalPhone** - Projektes beinhaltet bisher 15 verschiedene Sprachen. Sie beinhaltet zu jeder einzelnen Sprache Daten von ungefähr 100 verschiedenen Sprechern. Jeder Sprecher hat um die 20 Minuten wirtschaftliche und politische Texte aus Zeitungen vorgelesen, die sowohl nationale als auch internationale Themen behandeln (vergleichbar mit dem Wall Street Journal). Die Sprecher und Sprecherinnen decken alle Altersstufen ab und sprechen verschiedene Dialekte ihrer Sprache. Die Aufnahmen wurden ausnahmslos in den Heimatländern der Sprecher von Angehörigen der entsprechenden Nationalitäten durchgeführt. Dies geschah, um Veränderungen der Aussprache durch Auslandsaufenthalte und dem Verwenden von sprachfremden Wörtern weitgehend entgegenzuwirken. Zu jedem Sprecher wurden persönliche Daten wie Geschlecht, Dialekt und Beruf aufgenommen. Umgebungsinformationen wurden genauso erfaßt wie Details über den jeweiligen technischen Aufbau.

Sprache		# Sprecher	# Spracheinheiten	# Stunden
CH-Mandarin	CH	112	219.000	26,7
Deutsch	DE	71	132.000	16,7
Englisch (WSJ)	EN	83	129.000	15
Französisch (Bref)	FR	74	123.000	13,9
Japanisch	JA	108	212.000	22,9
Koreanisch	KO	80	301.000	16,4
Kroatisch	KR	72	89.000	12
Portugiesisch	PO	80	130.000	16,5
Spanisch	SP	82	138.000	17,6
Schwedisch	SE	79	144.000	17,4
Türkisch	TU	79	87.000	13,2

Tabelle 1: Datenbasis

Rang	Sprache	Länder	Sprecher (in Mio)
1.	CH-Mandarin	China	907
2.	Englisch	USA, UK, Kanada, Australien	456
4.	Spanisch	Lateinamerika, Spanien	362
5.	Russisch	Russland, unabhängige Staaten	293
6.	Arabisch	Nord Afrika, Mittlerer Osten	208
8.	Portugiesisch	Brasilien, Portugal, Angola	177
10.	Japanisch	Japan	126
11.	Französisch	Frankreich, Kanada, Afrika, Schweiz	123
12.	Deutsch	Deutschland, Österreich, Schweiz	119
15.	Koreanisch	Korea, China	73
25.	Türkisch	Türkei	57
44.	Kroatisch	Balkan	20

Tabelle 2: Abdeckung

Sprache	Fehlerrate	Vokabular	PP
CH-Mandarin	14,5	45.000	207
Deutsch	11,8	61.000	200
Englisch	14,0	64.000	150
Französisch	18,0	30.000	240
Japanisch	10,0	22.000	230
Koreanisch	14,5	64.000	137
Kroatisch	20,0	15.000	280
Spanisch	20,0	15.000	245
Türkisch	16,9	15.000	280

Tabelle 3: Fehlerraten und Vokabulargrößen

Die Aufnahmen wurden mit einem tragbaren DAT-Rekorder und einem Sennheiser Mikrofon durchgeführt. Diese Vorgehensweise ergab digitale Daten in Stereoqualität mit einer Samplingrate von 48kHz. Für die weitere Verarbeitung auf UNIX Workstations wurde die Qualität auf 16kHz mit 16bit Auflösung reduziert.

Für alle Aufnahmen wurden Transkriptionen benötigt. Um den Aufwand dabei so gering wie möglich zu halten, wurden Texte verwendet, die bereits in elektronischer Form vorlagen. Alle Texte mußten aber nachbearbeitet werden. Da die Sprecher gebeten wurden, nach jedem Satz eine Pause zu machen, konnte die Aufteilung der Aufzeichnungen in einzelne Sätze weitgehend automatisiert werden. Diese Sätze wurden dann anhand der vorhandenen Texte geprüft. Artefakte der gesprochenen Sprache wie Stottern, abgebrochene Wörter, Räuspern, falsche Aussprache und sonstige kleinere Abweichungen wurden korrigiert bzw. eingefügt.

3.2.2 Die GlobalPhone - Spracherkennungssysteme

Im Rahmen dieser Arbeit wurden neun monolinguale Spracherkenner des GlobalPhone - Projektes verwendet. Die entsprechenden Sprachen sind Türkisch, Deutsch, Französisch, Englisch, Koreanisch, Japanisch, Kroatisch, Chinesisch und Spanisch. Im folgenden wird der Aufbau dieser Systeme kurz erläutert. Die Fehlerraten und Vokabulargrößen sind in Tabelle 3 ([ScWa99(2)]) dargestellt. Als multilingualer Erkenner diente ein System mit einer gemeinsamen Akustik für die neun Sprachen. Zur Sprachenidentifizierung wurde der

Phonemerkenner des GlobalPhone - Projektes verwendet, der in seinem Phonempool obige Sprachen vereint.

Die Datenbasis wird für die Trainings- und Testphasen aufgeteilt. Basierend auf dem Trainingsset (Tabelle 1) werden die akustischen Modelle (Abschnitt 2.2.3) und die Sprachmodelle der Erkenner trainiert. Da der Umfang der Datenbasis für die Erstellung geeigneter Trigramm Sprachmodelle zu gering ist, sind für viele Wörter des Testsets die nötigen N-gramme nicht vorhanden. Für all diese Wörter werden dem Sprachmodell zusätzlich Unigramme mit geringen Wahrscheinlichkeiten beigefügt. Dies führt zu *Out-Of-Vocabulary*-Raten (OOV) von 0%. Somit ist sichergestellt, daß die gemessenen Leistungen auf die Qualität der akustischen Modelle zurückzuführen sind und nicht durch die in den Sprachmodellen fehlenden Wörter bedingt werden.

Alle Spracherkennungssysteme wurden mit Hilfe des Janus RTk erstellt. Die folgenden Absätze beschreiben kurz die verwendeten Systeme. Eine genauere Erläuterung der Erkennerkomponenten findet sich in Abschnitt 2. Der in dieser Arbeit erstellte multilinguale Erkenner basiert auf der Kommunikation der einzelnen monolingualen Systeme des GlobalPhone - Projektes. Die folgenden Daten wurden für diesen Ansatz nicht verändert. Weitere Beschreibungen und zahlreichere Daten der Systeme finden sich in [ScWa99(2)].

Monolinguale Erkenner
Die Vorverarbeitung basiert auf 13 Mel-Cepstral Koeffizienten. Im Zeitbereich werden u.a. die Informationen aus der Bestimmung der Nulldurchgangsrate verwendet. Nach Eliminierung der Anregungskomponenten werden die Eingabevektoren mit einer LDA auf 32 Dimensionen reduziert.

Die akustischen Modelle verwenden kontinuierliche HMMs mit drei Zuständen. Die Anzahl der Polyphonemodelle wurde auf 3000 festgelegt. Es kommen Tri- und Quintphone zum Einsatz. Die Codebücher werden durch 32 Gaußsche Mischverteilungen modelliert.

Die Decodierungsphase besteht aus mehreren Durchläufen, die einzeln angewählt werden können. Grundsätzlich wird eine Baumsuche (*tree-pass*) verwendet, die den Suchraum für die folgende lineare Suche (*flat-pass*) einschränkt. Jeder Suchdurchgang liefert eine Hypothese. Zusätzlich zu diesen beiden Suchen kann noch ein tree-pass mit *Vokaltraktlängennormalisierung* (VTLN) oder ein tree-pass mit *Maximum Likelihood Adaption* (MLLR) stattfinden (Abschnitt 2.2.6). Abschließend kann noch ein *Lattice-Rescoring* stattfinden. Alle Systeme verwenden zur Beschleunigung des Erkennungsvorgangs *Phonem Lookaheads* und *Bucket Box Intersection* (Abschnitt 2.2.6).

Multilingualer Erkenner
Das GlobalPhone - Projekt verfolgt drei verschiedene Ansätze bei der Realisierung eines multilingualen Spracherkenners. Die Systeme arbeiten mit unterschiedlicher, kontextabhängig modellierter Akustik [ScWa99(2)].

Die Akustik des Systems *ML-sep* ist genauso aufgebaut wie die Akustik der entsprechenden monolingualen Erkenner. Es werden alle Phoneme der einzelnen Sprachen verwendet, die resultierenden Modelle werden aber in einem Modul zusammengefaßt.

Die beiden anderen Systeme arbeiten mit einem globalen Phonemset, welches die lautlichen Gemeinsamkeiten der Sprachen berücksichtigt und eine mehrfache Modellierung in der Akustik verhindert. Das dafür notwendige multilinguale Phonemset erhält man wie in Abschnitt 2.2.3 beschrieben. Auf diese Weise wurden 12 Sprachen der Datenbasis zusammengefaßt und ergaben 162 verschiedene Phonemkategorien, wobei 83 Kategorien von mehr als einer Sprache geteilt wurden und 79 nur einer Sprache entsprachen.

Die beiden auf diesem globalen Phonemset aufbauenden Systeme unterscheiden sich wie folgt. Das System *ML-mix* verwendet für alle Sprachen eine Akustik mit den nötigen Modellen aus dem Phonemset, ohne Informationen über die entsprechende Sprache zu bewahren. Die Akustik des *ML-tag*-Systems markiert alle Phoneme entsprechende ihrer Sprachzugehörigkeit. Um eine sinnvolle, kontextabhängige Akustik zu erhalten, muß hier das Fragenset, das beim Beschneiden der Modelzahl zur Anwendung kommt, um Fragen zur sprachlichen Abhängigkeit erweitert werden.

Die Leistung dieser multilingualen Systeme ist, bezogen auf *ML-sep*, um 0,3% bis 2,4% schlechter. Die *ML-tag*-Systeme erreichen bessere Leistungen als die *ML-mix*-Systeme. Erhöht man die Anzahl der Polyphonmodelle in der Akustik (z. B. 1500 pro beteiligte Sprache), so läßt sich die Leistung der *ML-tag*-Systeme noch steigern. Das *ML-tag*-System wurde als multilingualer Spracherkenner in das in Abschnitt 5 beschriebene Demosystem eingebunden.

Phonemerkenner
Die Basis für den hier verwendeten Sprachenidentifizierer bildet der Phonemerkenner des GlobalPhone Projektes. Der Phonempool des Phonemerkenners vereinigt die Phoneme der monolingualen Spracherkenner für jede der neun

Sprachen. Jedes Phonem hat eine Markierung, die seine Sprachzugehörigkeit angibt. Nach dem eigentlichen Erkennungsvorgang werden die erkannten Phoneme nach ihren Markierungen sortiert und die Sprache, deren Markierungen am häufigsten vorkommen, wird als Ergebnis ausgegeben.

Das Sprachenmodell des Phonemerkenners verwendet n-gramme, die nur Übergänge von Phonemen innerhalb einer Sprache erlauben. Dies reduziert "Ausrutscher" in andere Sprachen mit verwechselbar ähnlich klingenden Lauten. Für eine Sprachänderung ist damit alleine das akustische Modell verantwortlich. Ist diese Vorgehensweise nicht erwünscht, kann ein Sprachenmodell mit einer Gleichverteilung verwendet werden, was einer Phonemerkennung ohne Sprachenmodell entspricht.

4 Kommunikation monolingualer Systeme

4.1 Problemstellung

Bevor ein Entwicklungsteam die multilinguale Spracherkennung angeht, hat es im allgemeinen schon zahlreiche monolinguale Spracherkenner erstellt. Dies geschieht, um verschiedene Architekturen zu realisieren, um mit den Eigenarten der unterschiedlichen Sprachen vertraut zu werden und um eine Ausgangsbasis zur Erstellung eines multilingualen Systems zur Hand zu haben.

Da die üblichen multilingualen Erkenner aber meistens auf einem gemeinsamen Phonempool basieren, müssen die neuen Systeme im allgemeinen von Grund auf neu trainiert werden, um gute Ergebnisse zu erhalten. Die Ergebnisse, die mit den bestehenden monolingualen Systemen erreicht wurden, gehen bei dieser Vorgehensweise verloren.

Ansätze zur Erkennung mehrerer Sprachen, die bestehende monolinguale Systeme weiterverwenden, sind im allgemeinen mit einem hohen Aufwand an Speicherplatz und Rechenzeit verbunden. Eine Möglichkeit ist, alle verfügbaren Spracherkenner parallel zu starten und mit der gleichen Eingabe zu versehen. Die gelieferten Ausgaben müssen dann verglichen und bewertet werden. Da alle Systeme die Eingabe bearbeiten, ist der Aufwand bei dieser Methode der Spracherkennung um ein entsprechend Vielfaches höher als bei der monolingualen Erkennung.

Eine andere Möglichkeit ist, der Erkennung ein Sprachenidentifizierungssystem vorzuschalten, mit dessen Hilfe der entsprechende Spracherkenner ausgewählt wird. Dies setzt zusätzlich zu den monolingualen Erkennern voraus, daß ein Identifizierungssystem vorhanden ist, das alle nötigen Sprachen bewältigen kann. Der Aufwand an Rechenzeit sinkt aber auf einen Bruchteil der Rechenzeit, die im ersten Beispiel benötigt wurde.

Das im Rahmen dieser Arbeit entworfene System verbindet diese beiden Ansätze. Das Ziel war, die vorhandenen monolingualen Spracherkenner des GlobalPhone - Projektes (Abschnitt 3.2.2) für ein multilinguales System weiterzuverwenden und trotzdem obige Nachteile soweit wie möglich zu umgehen. Es sollte also kein Sprachidentifizierungssystem nötig und nicht alle Erkenner durchgehend mit der Decodierung der Eingabe beschäftigt sein.

Zu Beginn des Erkennungsvorgangs werden alle Systeme gestartet. Die Eingabe wird aber nicht von jedem Erkenner bis zum Ende bearbeitet, sondern die Systeme werden alle an vorgegebenen Stellen des Vorganges gestoppt. Die bis dahin erreichten Ergebnisse bestimmen, welche Systeme weiterlaufen und welche für den weiteren Verlauf der Suche eingefroren werden. Die Erkenner mit den schlechtesten Ergebnissen werden so nach und nach ausgeschaltet, bis am Ende nur noch das beste System die Eingabe bearbeitet und eine Hypothese liefert. So wird der Rechenaufwand nach und nach reduziert, bis er dem eines monolingualen Erkenners entspricht.

Um die Erkenner zu einem geeigneten Zeitpunkt anhalten zu können, war es vor allem nötig, die Qualität ihrer Ausgaben an beliebigen Positionen der Eingabe vergleichen zu können. Um dies zu erreichen, wurde eine Kommunikation zwischen den verschiedenen Systemen realisiert.

Eine weitere Problematik lag darin, die Erkennung an den richtigen Abbruchstellen wieder nahtlos fortzusetzen, sobald sich ein System qualifiziert hat. So verwenden die Erkenner ihre bereits bearbeiteten Teilergebnisse weiter. Die investierte Rechenzeit geht für diese Systeme also nicht verloren.

Ein zusätzliches Problem ist die Vergleichbarkeit der von den Erkennern ausgegebenen Bewertungen. Hier war es sehr hilfreich, daß die Erkenner alle die gleiche Struktur haben (vergl. Abschnitt 3.2.2). Die Bewertungen hatten somit weitgehend vergleichbare Größenordnungen. Dieses Problem wird in Abschnitt 4.3.2 ausführlich erläutert.

Bevor in Abschnitt 4.3 genauer auf den Entwurf von `Metasearch` eingegangen wird, folgt mit Abschnitt 4.2 ein kurzer Einblick in das Konzept des JANUS RTk für Entwickler. Hier soll und kann aber weder eine erschöpfende Erläuterung des geschriebenen Codes noch eine genaue Anleitung für die Modulentwicklung innerhalb des JRTk gegeben werden.

4.2 JANUS RTk als Entwicklungsumgebung

Das JANUS RTk ist eine objektorientierte, in *C* implementierte Entwicklungsumgebung. Die hier realisierbaren, unterschiedlichsten Spracherkennungssysteme werden durch aufeinander aufbauende und miteinander kommunizierende Objekte erzeugt. Die Handhabung dieser Objekte erfolgt über eine *tcl* -

Schnittstelle (vergleiche auch Abschnitt 3.1). Um die Entwicklung neuer Module zu vereinfachen und um eine einheitliche Bedienung zu erreichen, muß der Entwurf folgenden Richtlinien folgen.

Ein Grundgedanke dieser objektorientierten Struktur ist eine nach außen versteckte Implementierung, die nur von wenigen Methoden gebrochen wird. Diese Methoden bilden dann das Interface des neuen Moduls und sind hier über die *tcl* - Schnittstelle des JRTk aufrufbar. Zusätzlich zu diesen, vom Entwickler festgelegten Methoden, sollte jedes Modul einheitliche, von der Janus - Struktur vorgeschlagene Methoden implementierten, welche die Grundfunktionen und die Einbettung in das Gesamtsystem realisieren.

Die vorgegebenen Methoden sind `configure`, `puts` und `destroy`. Sie dienen dazu, die Parameter des Objektes zu manipulieren, Parameter und Inhalte auszugeben und das Objekt und den belegten Speicherplatz wieder freizugeben. Weiterhin muß jedes Modul einen geeigneten Initialisierungscode bereithalten, der beim Erzeugen eines Objekts ausgeführt wird. Über vorgegebene Strukturen ist auch die Kommunikation zwischen sowie das Zusammenfassen von einzelnen Objekten geregelt.

4.3 Entwurf von *Metasearch*

4.3.1 Übersicht

Der Grundgedanke des hier verfolgten Ansatzes, gegebene Spracherkenner weiterzuverwenden, wurde auf die Implementierungsidee ausgeweitet. Die in Abschnitt 2.2.5 erläuterte Decodierung wird in Janus durch sogenannte *Suchobjekte* (`search`) realisiert. Diese stellen alle möglichen Decodierungsarten wie Baumsuche (`treeForward`) oder lineare Suche (`flatForward`) zur Verfügung. Weiterhin speichern sie alle Einstellungen und definieren den verwendeten Suchraum. So wird in einem Suchobjekt zum Beispiel das durch eine Baumsuche eingeschränkte Vokabular abgelegt, um es gegebenenfalls für eine folgende lineare Suche zu verwenden. Die einzelnen Systeme bestehen zwar aus zahlreichen Objekten, welche die verschiedenen Elemente eines Spracherkenners realisieren, alle Informationen und Ergebnisse fließen aber in diesen Suchobjekten zusammen. Durch eine Kontrolle der Suchobjekte erreicht man eine Steuerung des gesamten Spracherkenners.

Um dies zu ermöglichen, wurde ein übergeordnetes Objekt `metaSearch` entwickelt. Diese Metasuche erlaubt es, einzelne Suchobjekte zusammenzufassen und zu steuern, ohne die eigentliche Implementierung der Suchobjekte zu verändern.

Die Aufgabe der Metasuche ist es, alle Suchobjekte mit der gleichen Eingabe zu starten, die Ausgaben in bestimmten Intervallen zu vergleichen und ungeeignete Suchen nach und nach abzuschalten. Um dies zu realisieren, reicht es aus, die initiale Baumsuche zu betrachten. Da die einzelnen Suchobjekte aber immer noch individuell kontrollierbar sind, ist es nach Durchlaufen der Metasuche möglich, weitere Suchmechanismen, wie zum Beispiel einen `flatForward`, anzuknüpfen.

4.3.2 Lösungen

Das `MetaSearch` Objekt vereinigt alle Parameter, Datenstrukturen und Methoden, deren Implementierung nötig waren, um die in Abschnitt 4.3.1 gezeigte Problemstellung zu lösen. Zu den Ausführungen in diesem und im folgenden Abschnitt findet sich eine Teilauflistung des Codes als Referenz in Anhang B.

Um die Verwaltung der einzelnen Suchobjekte zu realisieren, ist innerhalb des `MetaSearch` Moduls ein `SearchCtrl` Objekt definiert. Für jedes hinzugefügte Suchobjekt wird ein solches Objekt erzeugt. Um den Speicheraufwand so niedrig wie möglich zu halten, werden nur Zeiger auf die Suchobjekte gespeichert. Die einzelnen Suchen können wie gewohnt erstellt und auch einzeln verwendet werden. Es ist keinerlei Anpassung beim Initialisieren der Spracherkenner nötig, um die Metasuche zu verwenden. Die `SearchCtrl` Objekte werden mit Hilfe einer Listenstruktur im `MetaSearch` Objekt verwaltet.

Nach dem Start der Metasuche werden für alle Suchobjekte Baumsuchen gestartet. Die Decodierung wird nach der im Parameter `step` angegebenen Schrittweite in Frames angehalten. Zu diesem Zeitpunkt erfolgt ein Vergleich der jeweils erreichten Bewertung der Suchen. Die Bewertungen, die unterhalb eines Toleranzbereiches bezüglich des bisher besten Ergebnisses liegen, qualifizieren die entsprechenden Suchen für den nächsten Durchgang. Die Größe des Toleranzbereichs ergibt sich aus der Addition des Parameters `thold` zu dem bisher besten Ergebnis.

Dieser Vorgang wiederholt sich, bis die Bewertung eines Suchergebnisses so gut ist, daß alle anderen Ergebnisse über dem Toleranzbereich liegen. Danach wird die Baumsuche an der Abbruchstelle ohne weitere Unterbrechungen fortgesetzt.

Die Decodierung im JANUS RTk verwendet ein globales Suchobjekt für den Decodierungsvorgang. Die Informationen eines individuellen Suchobjekts werden in diese globale Struktur kopiert und der Vorgang gestartet. Normalerweise wird in dieser Struktur nur mit einer Suche gearbeitet. Da bei der Metasuche aber mehrere Suchobjekte im Wechsel bearbeitet werden, war es nötig, wichtige Variablen in den `SearchCtrl` Objekten zwischenzuspeichern und die Informationen aus der globalen Suche zurück in die individuellen Objekte zu schreiben. Jeder Wechsel eines Suchobjektes besteht also aus einem Rettungsvorgang des abgearbeiteten und einem Initialisierungsvorgang des zu bearbeitenden Suchobjektes.

Um die Vergleichbarkeit der einzelnen Bewertungsausgaben der Suchobjekte zu verbessern, war es nötig, eine *Normalisierung* einzuführen. Diese geschieht mit mehreren Quotienten, welche die Ausgaben der entsprechenden Suchen so optimieren, daß sie für die jeweilige Eingabesprache den niedrigsten Wert ausgeben. Während der Optimierung dieser Quotienten hat es sich gezeigt, daß deren Werte innerhalb einer Suche stark schwanken. Deshalb wurden auf den Testdaten (Anhang C.1) Quotienten für fünf verschiedene Framezahlen erstellt. Es sind Werte für 20, 40, 50, 80 und 100 Frames vorhanden. Ergebnisse bei anderen Framezahlen werden mit einem angenährten Quotienten normalisiert.

4.3.3 Methoden

Dieser und der folgende Abschnitt liefern einen Einblick in den Funktionsumfang der Metasuche. Hier ist an keine erschöpfende Auflistung der Codes gedacht, sondern an eine Einführung zur leichteren Handhabung der Schnittstellen des Moduls. Aufrufbare Methoden können, wie im JANUS RTk üblich, mit dem Parameter *-help* ausgegeben werden (Abbildung 2). Verwendet man *-help* als Parameter der einzelnen Methoden, wird eine entsprechend ausführlichere Hilfe ausgegeben.

puts
Diese Methode ist eine der Standardmethoden, die von jedem Modul implementiert werden und so allen erzeugten Objekten gemeinsam sind. Hier gibt diese Methode Auskunft über die hinzugefügten Suchobjekte und deren Er-

gebnisse des letzten Laufs von `Metasearch`. Zusätzlich werden alle Parameter der `Metasearch` aufgelistet (Abbildung 4).

best
Um detailliertere Auskunft über die beste Suche des letzten Laufs der Metasuche zu bekommen, wurde die Methode *best* implementiert. Sie liefert genaue Auskunft über Framezahl, Bewertung, Hypothese und verwendetet Parameter.

include/exclude
Diese `Metasearch` eigenen Methoden dienen zum Hinzufügen bzw. zum Entfernen von Suchobjekten.

reset
Um alle Suchen wieder in den Grundzustand zu versetzen, wurde `reset` entwickelt. Diese Methode betrifft nur die in den `SearchCtrl` Objekten abgespeicherte Information und beeinflußt keinen der Parameter von `Metasearch`. Diese Methode wird automatisch bei jedem Aufruf von `go` ausgeführt.

go
Diese Methode startet eine Metasuche mit allen hinzugefügten Suchobjekten. Beim Start der Metasuche muß ein Parameter übergeben werden, der alle Informationen über die Eingabe enthält. Dieser Parameter ist ein `Array`, wie es auch beim Aufruf einer Baumsuche innerhalb eines einzelnen Suchobjekts verwendet wird. Die Ausgabe liefert entsprechend den Einstellungen detaillierte Informationen über den Verlauf der Metasuche (Abbildung 3).

configure
Dies ist eine weitere Standardmethode, die es erlaubt, alle in Abschnitt 4.3.4 aufgelisteten Parameter einzustellen.

4.3.4 Parameter

Die folgenden Erläuterungen konzentrieren sich auf die einzelnen Parameter der Metasuche, ihre Bedeutung und was ihre Manipulation bewirkt. Die Ergebnisse der Metasuche mit unterschiedlichen Einstellungen der Parameter `step` und `thold` finden sich in Abschnitt 4.4.

step
Mit dem Parameter `step` wird die Schrittweite der einzelnen Suchen geregelt. Er gibt die Abbruchstellen bei der Decodierung an, an denen der Scorevergleich

```
DESCRIPTION

A 'MetaSearch' object runs and prunes several Searches.


METHODS

puts                displays all included Search objects
include             includes Search object(s)
exclude             excludes Search object(s)
best                display best of last MetaSearch go
reset               resets all MetaSearch params
go                  do a MetaSearch with all included Search objects

% 
```

Abbildung 2: Methoden der Metasuche

```
INFO    metaSearch.c(0882) ============================
INFO    metaSearch.c(0883) MetaSearch pass: 1
INFO    metaSearch.c(0884) Still active   : 2
INFO    metaSearch.c(0732) ----------------------------
INFO    metaSearch.c(0733) Active Search  : 'searchTU'
|----+----|----+----|----+----|----+----[40]
||||||||||||||||||||||||||||||||||||||||
INFO    metaSearch.c(0788) Score: 2141.663574
INFO    metaSearch.c(0929) normS: 40.967323
INFO    metaSearch.c(0732) ----------------------------
INFO    metaSearch.c(0733) Active Search  : 'searchSP'
|----+----|----+----|----+----|----+----[40]
||||||||||||||||||||||||||||||||||||||||
INFO    metaSearch.c(0788) Score: 2041.442383
INFO    metaSearch.c(0929) normS: 39.053848
INFO    metaSearch.c(0968) searchTU
INFO    metaSearch.c(0969) normS: 40.967323
INFO    metaSearch.c(0970) normT: 1.912874
INFO    metaSearch.c(0976) Deactivated searchTU.
INFO    metaSearch.c(0968) searchSP
INFO    metaSearch.c(0969) normS: 39.053848
INFO    metaSearch.c(0970) normT: 1.913052
INFO    metaSearch.c(1001) ============================
INFO    metaSearch.c(1002) MetaSearch pass: 2
INFO    metaSearch.c(1003) Final Search   : 'searchSP'!
INFO    metaSearch.c(0732) ----------------------------
INFO    metaSearch.c(0733) Active Search  : 'searchSP'
( riqueza
( presencia(2) de la
( presencia(2) de la bromas
( presencia(2) de la van a otro crack
( presencia(2) de la van a chocar con son
( presencia(2) de la van a chocar con son $ PNUD
( presencia(2) de la van a chocar con son $ pregunto otro
( presencia(2) de la van a chocar con son $ pregunto piloto )

INFO    metaSearch.c(0788) Score: 15299.757812
Timer: 166.185501
step  : 40, thold : 100, passN : 2
frame : 303, score : 15299.757812
hypo  :( presencia(2) de la van a chocar con son $ pregunto piloto
Search: searchSP
```

Abbildung 3: Ausgabe von `go`

vorgenommen wird. Seine Wahl hat den größten Einfluß auf die Qualität der Leistung der Metasuche bei der Sprachenidentifizierung. Wird er zu niedrig gewählt, werden geeignete Suchen zu früh gestrichen oder es werden zu Beginn kaum Suchen deaktiviert, da alle ähnliche Bewertungen haben. Außerdem findet ein häufiger Wechsel von Suchobjekten statt, der mit einem hohen Speicheraufwand verbunden ist, da gegebenenfalls nicht speicherresidente Komponenten eingelagert werden müssen.

thold
Dieser Wert beeinflußt den Toleranzbereich bei den Scorevergleichen. Wird er zu groß gewählt, werden Suchen mit zu schlechter Bewertung fortgesetzt. Fällt seine Wahl zu niedrig aus, werden vielversprechende Suchen zu schnell ausgeschaltet.

verb
Dieser Parameter reguliert, wie ausführlich die Informationsausgabe während der Metasuche ist. Der Wert '0' entspricht der Ausgabe mit geringstem Informationsinhalt.

norm
Wird `norm` auf '0' gesetzt, findet keine Normierung statt. Die Ausgaben der Erkenner werden direkt verwendet. So hängt die Vergleichbarkeit nur von der Struktur der Erkenner ab.

part
Auf den Wert '1' gesetzt, bewirkt dieser Paramter die Ausgabe von Teilhypothesen während der abschließenden Baumsuche.

partN
Die Ausgabe von Teilhypothesen erfolgt immer nach einer festgelegten Framezahl. Diese Zahl kann hier eingestellt werden.

4.4 Ergebnisse

4.4.1 Testszenario

Der erste Abschnitt der `Metasearch` kann als System zur Sprachenidentifizierung betrachtet werden. Um eine Aussage über die Güte der Identifizierungsleistung machen zu können, war es nötig, ein Vergleichssystem heranzuziehen. Mit der Verwendung des GlobalPhone Phonemerkenners aus Abschnitt 3.2.2 standen sich zwei unterschiedliche Ansätze gegenüber. Die Tabellen 4 und 5 vergleichen die erzielten Ergebnisse unter zahlreichen Variationen der Paramter `step` und `thold`. Die Testläufe fanden auf den in Anhang C.2 gelisteten Daten statt.

Tabelle 4 stellt die Testergebnisse unter Verwendung eines Wertes von '50' für `thold` dar, während bei den Ergebnissen in Tabelle 5 ein Wert von '100' eingestellt war. Um einen Blick für das Gesamtpotential dieser Variante der Sprachenidentifizierung zu bekommen, wurde bei einem Durchlauf der Suchen jeweils die gesamte Eingabe verwendet.

Diese bestmöglichen Ergebnisse finden sich in der ersten Spalte der Tabellen wieder und dienen als obere Grenze, die die maximal erreichbare Erkennungsleisung darstellt. In der zweiten Spalte sind die Resultate aus den Identifizierungsläufen des Phonemerkenners eingetragen. In den folgenden Spalten stehen dann die Ergebnisse bei unterschiedlichen Einstellungen für `step`. Die letzte Zeile der Tabellen zeigt die für die entsprechenden Parameter mittlere Anzahl von Durchläufen der Metasuche bis eine Entscheidung erzielt wurde.

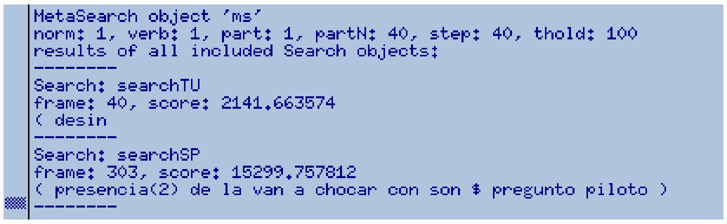
4.4.2 Betrachtung der Ergebnisse

Die starken Schwankungen bei der Erkennerleistung von Framezahl zu Framezahl innerhalb der gleichen Systeme, sind auf den nicht linearen Verlauf der Bewertungswerte und auf die Probleme bei der Anpassung der Normierungsquotienten zurückzuführen.

Leider war es nicht möglich, die Ausgaben direkt zu vergleichen, da die Werte für die Bewertungen nicht im gleichen Rahmen lagen. Hier lieferte speziell der kroatische Erkenner immer bessere Werte als alle anderen Systeme.

Betrachtet man die Ergebnisse in Abhängigkeit von der Länge der jeweils verwendeten Abschnitte der Eingabe, erkennt man, daß diese die Leistung am stärksten beeinflußt. Je länger das verwendete Teilstück ist, desto besser die Ergebnisse. Dies wird auch durch die Tatsache unterstrichen, daß das System, welches die Eingaben ganz bearbeitet hat, die beste Identifizierungsleistung erreicht.

Zusätzlich zur Steigerung der Leistung reduziert sich auch der zeitliche Aufwand, da der Wert für `N` mit steigendem `step` abnimmt. Die Verdoppelung von `thold` von '50' auf '100' bewirkt eine weiter Verbesserung der betrachteten Werte.



```
MetaSearch object 'ms'
norm: 1, verb: 1, part: 1, partN: 40, step: 40, thold: 100
results of all included Search objects:
--------
Search: searchTU
frame: 40, score: 2141.663574
( desin
--------
Search: searchSP
frame: 303, score: 15299.757812
( presencia(2) de la van a chocar con son $ pregunto piloto )
--------
```

Abbildung 4: Ausgabe von `puts`

step			20	40	50	80	100
Türkisch	82,5	70	75	65	55	50	72,5
Spanisch	65	60	52,5	60	45	65	57,5
Kroatisch	100	30	50	72,5	90	100	97,5
Deutsch	92,5	70	70	67,5	45	72,5	97,5
Chinesisch	100	92,5	82,5	92,5	97,5	85	92,5
total	88	64,5	66	71,5	66,5	74,5	83,5
N			3,125	2,495	2,325	2,19	2,145

Tabelle 4: LID - Metasearch für thold = 50

step			20	40	50	80	100
Türkisch	82,5	70	55	57,5	65	52,5	77,5
Spanisch	65	60	60	62,5	65	70	60
Kroatisch	100	30	87,5	95	95	100	100
Deutsch	92,5	70	87,5	85	60	47,5	97,5
Chinesisch	100	92,5	92,5	92,5	95	40	92,5
total	88	64,5	76,5	78,5	76	62	85,5
N			4,065	3,04	2,735	2,28	2,29

Tabelle 5: LID - Metasearch für thold = 100

5 Visualisierung

5.1 Java

Alles Wissen über und alle Referenz zu Java in dieser Arbeit sind ausnahmslos dem SUN Java Tutorial [JTut00], der SUN API-Spezifikation [JDoc00] und, als tragbarem Nachschlagewerk, der Schnellübersicht zu Java 1.2 [Stey98] entnommen. Besonders die beiden online-Referenzen seien jedem Java Programmierer als Nachschlagewerk und als Tutorium nahegelegt, da sie ständig den aktuellen Stand des Java *Development Kits* (JDK) wiederspiegeln. Zum einfachen Gebrauch lassen sich die HTML-Seiten auch als Paket herunterladen und lokal verwenden.

5.1.1 Sprache

"Java ist eine einfache, objektorientierte, dezentrale, interpretierte, stabil laufende, sichere, architekturneutrale, portierbare und dynamische Sprache, die Hochgeschwindigkeitsanwendungen und Multithreading unterstützt." [JTut00]. Ob er sich dieser hohen Meinung anschließen will, bleibt jedem Leser selbst überlassen. Im Rahmen dieser Diplomarbeit, bei der JDK 1.2.2 zum Einsatz kam, hat die Programmiersprache gute Dienste geleistet.

5.1.2 GUI

Mit der Klassenbibliothek *Swing* stellt Java ein mächtiges Werkzeug zur Verfügung, welches das Erstellen einer grafischen Nutzeroberfläche sehr vereinfacht. Klassen, die Bedienelemente wie Schieberegler, modale Dialoge, Listen und Menüleisten realisieren, werden ebenso zur Verfügung gestellt, wie Klassen zur einfachen Handhabung der Aktionen, die diese Elemente steuern oder von ihnen ausgelöst werden. Ab der Version 1.2 ist die bisher optionale Bibliothek *Swing*, im JDK integriert. Sie erweitert bzw. ersetzt die *AWT* Klassen, die diese Funktionen bisher erfüllten.

5.1.3 Unicode

Ein weiterer Grund für die Wahl von JAVA als Programmiersprache bei dieser Arbeit war, daß JAVA Unicode zur Darstellung von Zeichen verwendet. Das Unicode Konsortium wurde 1991 gegründet und hat sich zum Ziel gesetzt, den Unicode Standard weiterzuentwickeln und zu verbreiten. Der standardisierte Unicode ist eine identische Teilmenge des ISO/IEC 10646-1:1993. Auf den Internetseiten des Unicode Konsortiums [UniC00] finden sich ständig aktualisierte Tabellen aller Unicodesequenzen mit ausführlichen Beschreibungen und Darstellungen der entsprechenden Zeichen.

Die mit 16-bit kodierten Zeichen ermöglichen es, weit verbreitete Sprachen wie Türkisch oder Arabisch in den entsprechenden Schriften auszugeben, ohne daß ein Umweg über spezielle Grafikausgaben gegangen oder auf eine romanisierte Darstellung zurückgegriffen werden muß.

5.2 Funktionalität

Die hier vorgestellte JAVA Applikation realisiert eine `Erkennerdemo` für die Systeme des GLOBALPHONE - Projektes. Sie visualisiert die Ausgabe und sorgt über eine graphische Benutzerschnittstelle für eine einfache Manipulierbarkeit der Parameter und Eingabemodalitäten. Die folgenden Abschnitte beschreiben die Funktionen der einzelnen Programmkomponenten.

5.2.1 Systemmodus

Das Programm startet mit der in in Abbildung 5 dargestellten Konfiguration. Über das Menü `Systemmodus` (Abbildung 6) werden die verschiedenen Erkenner ausgewählt. Die verwendeten Erkenner sind die in Abschnitt 3.2.2 beschriebenen mono- und multilingualen Systeme des GLOBALPHONE Projektes bzw. das in Abschnitt 4 beschriebene, im Rahmen dieser Arbeit realisierte System.

Identifizierung (ID)
Wird dieser Systemmodus gewählt, verwendet JANUS den Phonemerkenner des GLOBALPHONE - Projektes. Dieser identifiziert die bei der Eingabe verwendete Sprache.

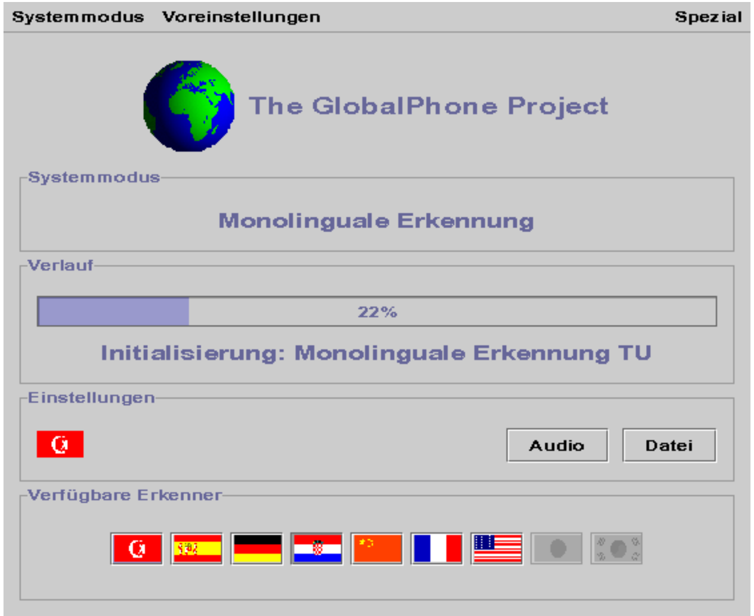


Abbildung 5: Hauptfenster

Abbildung 6: Systemmodus

Abbildung 7: Voreinstellungen

Spracherkennung (SE)
Ist dieser Modus gewählt, bestimmt der Nutzer explizit einen der verfügbaren Erkenner. Dies geschieht über die Knöpfe mit den Flaggen des entsprechenden Landes (Abbildung 5). Die Eingabe wird dann an den für diese Sprache trainierten Erkenner weitergeleitet. Die ausgegebene Hypothese des Erkenners wird dann im Ergebnisfenster dargestellt.

ID + SE: Monolingual
Dieser Modus ist eine Kombination der ersten beiden Modi. Der Nutzer wählt hier keinen Erkenner aus. Die Sprache der Eingabe wird zuerst identifiziert und dann an den entsprechenden Erkenner weitergeleitet. Die Ausgabe besteht aus der erkannten Sprache bzw. einer Meldung, daß und warum die Sprache nicht erkannt wurde. Ist für die erkannte Sprache ein initialisierter Erkenner vorhanden, wird automatisch die Hypothese ausgegeben.

ID + SE: Multilingual
Im Vergleich zum vorhergehenden Modus wird bei dieser Einstellung keine Sprachidentifizierung betrieben. Die Eingabe wird direkt an den multilingualen Erkenner des GlobalPhone - Projektes weitergeleitet.

ID + SE: Metasearch
Wird dieser Modus ausgewählt, kommt das in Abschnitt 4 beschriebene System zum Einsatz. Anhand der voreingestellten Parameter wird dann die Metasuche gestartet. Während der Identifizierungsphase der Metasuche werden die noch aktiven Suchen durch Flaggen dargestellt. Bei der Erkennungsphase erfolgt eine Ausgabe von Teilhypothesen im Hauptfenster.

Abbildung 8: Aufnahme

5.2.2 Ausgabe

Die Ausgabe geschieht über die Fenster `Log` (Abbildung 9) und `Ergebnisse` (Abbildung 10). Beide Fenster sind im Menü `Voreinstellungen` (Abbildung 7) ein- und ausblendbar.

Log-Fenster
In dieses Fenster (Abbildung 9) wird die Fehlerausgabe von JANUS dargestellt. Die Ausgabe entspricht genau den Informationen die auch ErDe von JANUS erhält. Die Darstellung dieser Informationen dient hauptsächlich der Kontrolle und Fehlerdiagnose. Diese Ausgabe wird zusätzlich in eine Datei geschrieben, da sonst die Informationen, die im Log-Fenster dargestellt sind, nach dem Beenden von ErDe verloren wären.

Ergebnisfenster
Das `Ergebnisfenster` (Abbildung 10) ist aufgeteilt in zwei Bereiche. Die Ausgabe dieses Fensters wird vorher in Unicodesequenzen übersetzt, um die der Sprache entsprechende Schrift darzustellen. Um von den Schriften, die auf dem zugrundeliegenden System installiert sind, unabhängig zu sein, wird ein *True-type*-Font verwendet, der mit dem JDK mitgeliefert wird. Die Referenz wird nur verwendet, wenn die Erkennung einer Datei erfolgt und eine Anbindung an die entsprechende Datenbasis vorhanden ist. In einer Demoumgebung wird dies selten der Fall sein. Der untere Teil des Ergebnisfensters wird für die Darstellung der Hypothesen verwendet. Hier erfolgt die Ausgabe der erkannten Sprache bzw. der vom Erkenner ausgegebenen Ergebnissätze. Erkannte Fehler werden ebenfalls hier dargestellt.

5.2.3 Eingabe

Mikrofon
Die Spracheingabe über ein Mikrofon erfolgt mit Hilfe eines mit ErDe gekoppelten Tcl/TK-Skripts (Abbildung 8, Abschnitt 5.3.2). Die Aufnahme beginnt, sobald der `Record`-Knopf gedrückt wird. Solange der Knopf gedrückt bleibt, wird der Aufnahmevorgang fortgesetzt. Wird er freigegeben und damit die Aufnahme beendet, beginnt die gewählte Erkennung.

Datei
Die Erkennung einer Datei ist nur möglich, wenn ErDe die GLOBALPHONE Datenbasis erreichen kann. Nach Betätigung des `Datei`-Knopfes im Hauptfenster wird der Nutzer aufgefordert, eine Identifikation der zu erkennenden Referenz einzugeben. Dabei muß genau auf die in der Datenbasis verwendeten Bezeichner geachtet werden. Möchte der Nutzer den zwölften Satz des 32. kroatischen Sprechers aus der Datenbasis einlesen, muß er zum Beispiel `KR032_12` eingeben.

Parameter-Dialog
Über den `Parameter`-Dialog (Abbildung 11) im Menü `Voreinstellungen` lassen sich einige der wichtigsten Parameter der Spracherkenner einstellen. Eine kurze Beschreibung der Parameter findet sich in Tabelle 6.

Parameter	Beschreibung
thold	Schwellenwert für `Metasearch`
step	`frame`-Zahl für `Metasearch`
beamSize	globaler Beam für aller Suchen
segmentLength	Eingabelänge bei der Sprachidentifizierung
topN	0 = alle Codebücher werden verwendet
doFlat	1 = zusätzliche lineare Suche
doVtln	1= Vokaltraktnormalisierung
doMLLR	1 = Maximum Likelihood Linear Regression
doLattice	1 = zusätzliches Latticerescoring.
doBBI	1 = *Bucket Box Intersection*
doLookahead	1 = lookaheads

Tabelle 6: Parameter

5.3 Entwurf

Im folgenden wird davon ausgegangen, daß der Leser mit den Grundzügen des objektorientierten Programmierens und mit der Programmiersprache JAVA weitgehend vertraut ist. Auf grundlegende Ausführungen und Erklärungen elementarer Begriffe wird hier verzichtet. Natürlich werden alle für das Verständnis notwendigen Ansätze und Vorgehensweisen erläutert.

5.3.1 Kommunikation mit dem JRTk

Das erste und wichtigste Problem war die Sicherstellung der Kommunikation mit dem JRTk. Da bei der Implementierung von einer Demoumgebung ausgegangen wurde, bei der eine Anbindung an ein Netzwerk nicht in jedem Falle gegeben ist, war es nicht notwendig, die Möglichkeit vorzusehen, den Erkenner auf einem anderem Rechner als ErDe auszuführen. Dies ermöglicht es, den üblichen Ansatz der Programmkommunikation über Sockets fallen zu lassen.

Stattdessen wird die Kommunikation mit dem JRTk, wie auch mit den anderen Programmen zur Audioeingabe und -ausgabe (Abschnitt 5.3.2) über die umgeleiteten Eingabe-, Ausgabe- und Fehlerströme realisiert.

Die `Runtime` - Klasse von JAVA bietet Methoden, die das Ausführen und Kontrollieren von Prozessen ermöglichen. Die dort spezifizierte Funktionalität bietet, zusammen mit den Methoden der `Prozeß` - Klasse ein Instrument zur Kommunikation mit externen Programmen. Nachdem ein Prozeß gestartet wurde, erhält man eine Instanz eines Prozesses zurück, die Informationen liefert und die Steuerung realisiert. Die `Prozeß` - Klasse stellt unter anderem die erwähnten Methoden zur Verfügung, die es erlauben, direkt in die Standardeingabe des Prozesses zu schreiben, seine Standardausgabe und seinen Fehlerausgang zu lesen.

Beim Aufbau der Kommunikation mit dem JRTk trat folgendes Problem auf. Das JRTk arbeitet nicht direkt mit der Standardeingabe. Es analysiert die Eingaben zuerst mit einem Kommandozeilen-Parser. Dieser trennt Eingabe in tcl-, janus- und betriebssystemspezifische Befehle und reicht diese an die entsprechenden Codestücke weiter. Außerdem werden hier unzulässige Eingaben abgefangen. Dies führt zu einer Blockade der Standardeingabe gegenüber der oben beschriebenen Vorgehensweise. Um diesen Ansatz weiterverfolgen zu können, war es nötig diesen Parser im Quellcode zu deaktivieren.

Die Tcl-Skripten, die den Aufbau der GLOBALPHONE - Systeme und den Ablauf des Erkennungsvorganges realisieren, waren bereits vorhanden. Sie wurden inhaltlich direkt übernommen, aber im Format leicht angepaßt.

5.3.2 Audioeingabe und -ausgabe

Da die von JAVA zur Verfügung gestellten Methoden zur Audioeingabe und -ausgabe die von JANUS verwendeten Formate nicht abdecken, mußte hier auf externe Funktionalität zurückgegriffen werden.

Das Tcl/TK-Skript `record.tcl` realisiert nicht nur die Aufnahme (Abschnitt 5.2.3), sondern stellt auch Möglichkeiten zur Manipulation der Parameter (Gain, ...) zur Verfügung. Die Verbindung zu ErDe wird durch die gleiche Vorgehensweise ermöglicht, die auch die Kommunikation mit JANUS realisiert (Abschnitt 5.3.1).

Um die Ausgabe zu realisieren, wurde das Pearl-Skript `playMail.pl` verwendet. Es konvertiert die in der GLOBALPHONE - Datenbasis gespeicherte Dateien und gibt sie wieder. Einige Ausgabeparameter können in den Audiovoreinstellungen (Abbildung 7) verändert werden. Es faßt alle nötigen Dienstprogramme des UNIX-Betriebssystems zusammen. Dieses Skript wird, im Gegensatz zu `record.tcl` nur bei Bedarf aufgerufen.

5.3.3 Internationalisierung

Um die Portierbarkeit von Code in andere Sprachen zu vereinfachen stellt JAVA *Internationalization* (*I18N*) zur Verfügung. Bei dieser Vorgehensweise werden alle Beschriftungen von Komponenten der grafischen Oberfläche und sonstige Ausgaben des Programms mit Bezeichnern belegt. Diese Bezeichner werden in zusätzlichen Dateien abgespeichert und dort mit dem eigentlichen Ausgaben identifiziert. Im Programm verwendet man diese Bezeichner anstelle der auszugebenden Texte. Wenn man die Ausgaben des Programmes in eine andere Sprache übersetzen will, genügt es, diese einzelne Datei zu übersetzen, der Programmcode bleibt unangetastet. Diese sogenannten `Properties` - Dateien von ErDe liegen in deutschen und englischen Fassungen vor.

5.4 Klassen

Die umfangreiche Klassenbibliothek der Programmiersprache JAVA liefert eine solche Vielfalt an Funktionalität, daß es im Rahmen dieser Arbeit nur bedingt nötig war, die bestehenden Klassen zu erweitern.

Die Aufgabe der JAVA Applikation ErDe ist die Steuerung des Spracherkenners und die Visualisierung der Ausgabe. Ein besonderes Augenmerk wurde dabei auf die Kontrolle der Parameter, die Erweiterbarkeit um neue Erkenner und die Darstellung der Ausgabe in den entsprechenden Schriften gelegt.

Um ErDe nach dem Starten eines Erkennungsvorganges nicht einfrieren zu lassen, wird die Kontrolle der Erkenner durch die Übergabe an einen eigenen `Thread` von der Oberfläche entkoppelt. Auch die Kontrolle der Audioeingabe über `record.tcl` erfolgt durch einen eigenen Thread.

Im folgenden werden kurz die einzelnen Klassen und deren Funktionalität eingeführt (Tabelle 7). Eine ausführlichere Auflistung der Klassen, ihrer Methoden

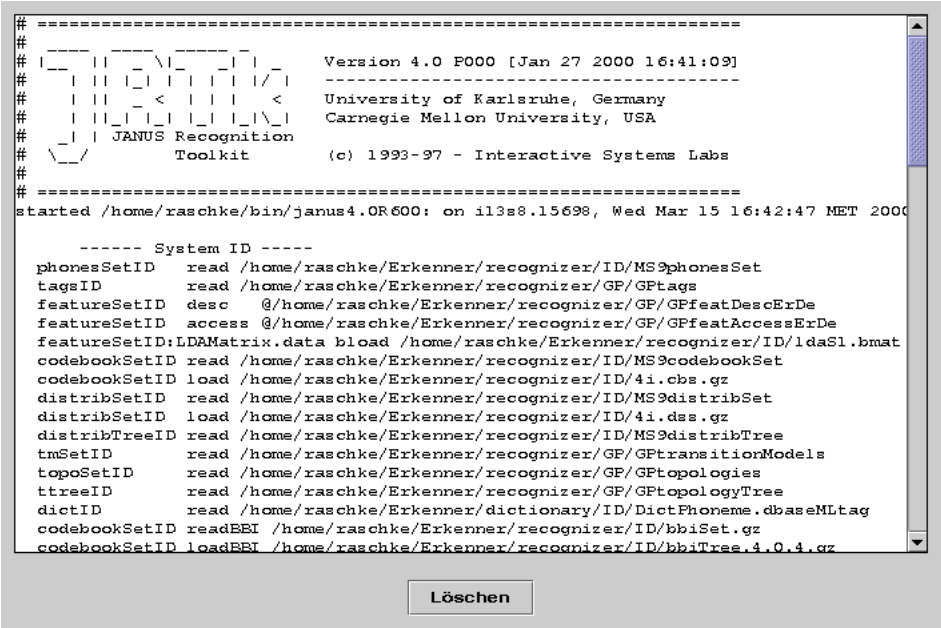


Abbildung 9: Log-Fenster für das JANUS RTk

Klasse	Funktionalität
ErDe	Hauptprogramm und globale Variablen
ErDeFenster	Hauptfenster
ErDeKnopf	Umschalter für unterschiedliche Erkenner
ErDeMenu	Menüleiste
ErDeAudio	Audiovoreinstellung
ErDeLog	Logfenster für die umgeleitete Janusausgabe
ErDeErg	Darstellung der Erkennerausgabe
ErDeJanus	Methoden für die Steuerung des JRTk
ErDeThread	Nebenläufigkeit für die Methoden aus ErDeJanus
ErDeUnicode	Darstellung der Ausgabe in Unicode
ErDeParameter	Konfiguration der Erkennerparameter

Tabelle 7: Klassenübersicht

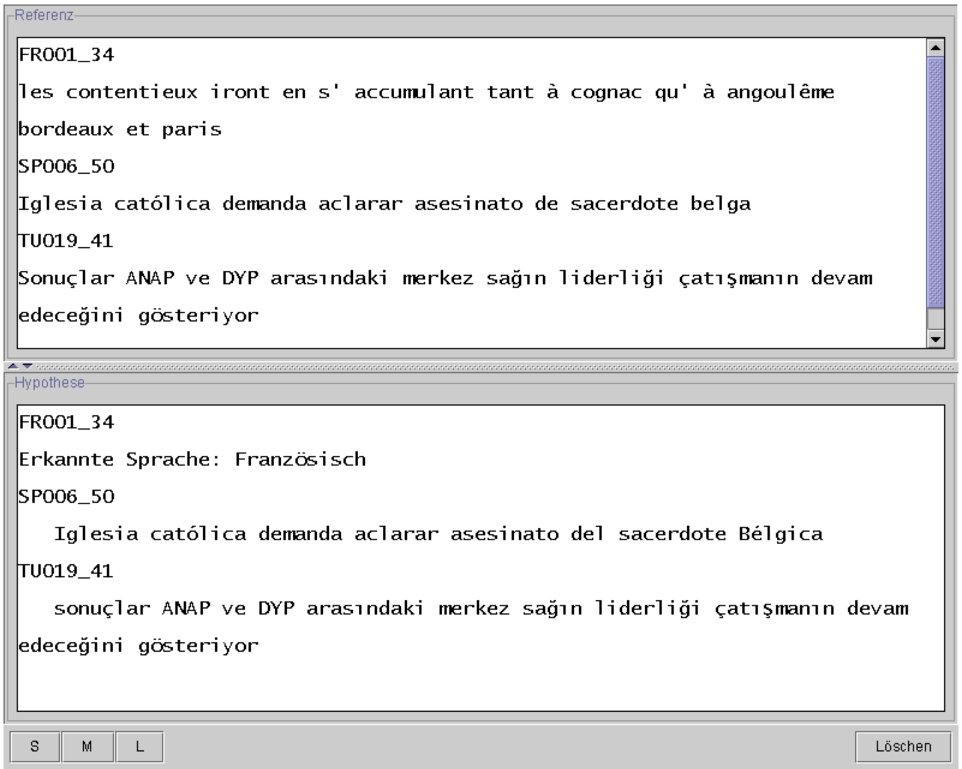


Abbildung 10: Ergebnisausgabe

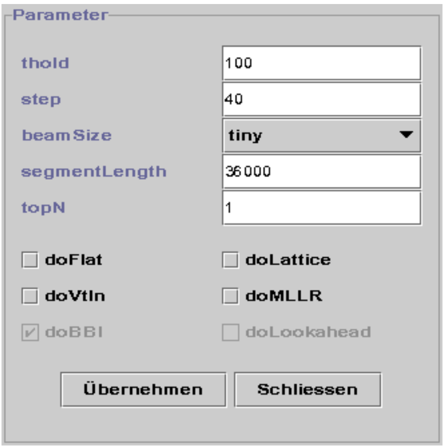


Abbildung 11: Parameter-Dialog

und Felder findet sich im Anhang A. Die Übersicht in Abbildung 12 zeigt die Zusammenhänge der einzelnen Klassen. Die Details der Implementierung werden hier nicht besprochen. Der Interessierte wende sich bitte an den Quellcode der entsprechenden Klasse.

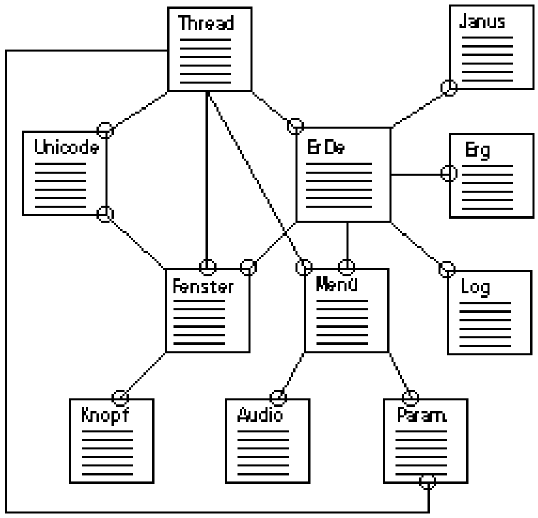


Abbildung 12: Zusammenhänge

5.4.1 ErDe

In dieser Klasse sind alle globalen Variablen und Konstanten deklariert. Das sich hier befindende Hauptprogramm hat folgende Funktionalität.

Das `File-encoding` wird hier gesetzt, um die entsprechenden Schriften zur Darstellung der Ausgabe zur Verfügung zu haben. Zusätzlich wird die `Properties`-Datei geladen, welche die Beschriftungen in der gewählten Sprache enthält. Die meisten der globalen Variablen und Konstanten werden hier initialisiert. Das Programm erzeugt eine Instanz der `Runtime`-Klasse und startet JANUS . Ein eigener Thread sorgt für die Initialisierung der Erkenner. Ab hier wird die Programmkontrolle dann an eine Instanz von *ErDeFenster* weitergegeben.

Dem Programm können Kommandozeilenargumente übergeben werden. Mit einer Integer Zahl (0-10) wird die Anzahl der zur Verfügung gestellten Erkenner gewählt. Wird '0' gewählt, wird nur der Phonemerkenner zur Sprachidentifizierung initialisiert. Wählt man '1', startet zusätzlich der multilinguale Erkenner. Bei '10' werden alle Erkenner in den JRTk geladen. Zwei zweistellige Strings steuern, welche `Properties`-Datei (Abschnitt 5.3.3) geladen wird. Momentan stehen Deutsch ('de DE') und US-amerikanisches Englisch ('en US') als Beschriftungssprache zur Verfügung.

5.4.2 ErDeFenster

Diese Klasse deklariert und definiert die meisten Elemente der Oberfläche. Alle Komponenten der Oberfläche, die in separaten Klassen implementiert sind, wie die Menüleiste, die Ein- und Ausgabefenster und Wahlknöpfe für die unterschiedlichen monolingualen Erkenner, werden hier zusammengeführt.

Das Erkennerumschalten, das Einlesen der Referenzen aus der GlobalPhone Datenbasis, das Starten der Threads für die Erkennung der gespeicherten Eingabe und das Aufräumen bei Programmende sind hier realisiert.

5.4.3 ErDeJanus

Diese Klasse beherbergt die gesamte Kommunikation mit Janus . Hier befinden sich alle Methoden, die zum Initialisieren und Starten der Erkennung in den verschiedenen Systemmodi benötigt werden. Der Bytestrom der Erkennerausgabe wird hier in Zeichenfolgen umgewandelt. Die Ergebnisse werden an die entsprechenden Ausgabefenster weitergeleitet, um das Logfenster auf dem neusten Stand zu halten. Auch befinden sich hier die Methoden, die den Verlauf der Erkennung kontrollieren, um ihn im Hauptfenster mit einem Verlaufsbalken und kurzen Informationen über den Fortschritt darzustellen.

5.4.4 ErDeThread

Die `run`-Methode dieser Klasse sorgt für die Nebenläufigkeit von vielen in ErDe realisierten Funktionen. Hier werden vor allem die Methoden von *ErDeJanus* aufgerufen, um die Spracherkenner weitgehend von den Funktionen der Oberfläche zu entkoppeln. Einige Funktionen, allen voran die detaillierte

Darstellung des Erkennungsverlaufs, stehen so auch während des Erkennungsvorganges zur Verfügung. Um eine Kollision zwischen Funktionsaufrufen, die eine direkte Manipulation des JRTk notwendig machen, zu vermeiden, werden diese mit einer globalen, booleschen Variablen abgesichert, die den Zugriff auf das Toolkit sperrt, bis der aktuelle Vorgang beendet ist.

In `ErDeThread` findet auch die Verbindung zwischen Sprachidentifikation und Spracherkennung statt, wenn der dritte Systemmodus gewählt wurde. Eine weitere wichtige Funktion ist die Steuerung der Spracherkennung nach einer Mikrofonaufnahme. Hierfür ist ein Thread zuständig, der direkt mit dem Aufnahmeskript kommuniziert und ständig dessen Fehlerausgabe kontrolliert.

5.4.5 ErDeUnicode

Diese Klasse stellt eine Datenstruktur zur Verfügung, um die aus der Datenbasis eingelesenen Sätze oder die vom Erkenner gelieferten Ergebnisse zu verwalten. Die Zeichenketten können hier gespeichert, ausgelesen und in Unicodesequenzen umgewandelt werden. Die Erkenner verwenden intern romanisierte Fassungen der einzelnen Schriften. Für jede Sprache existiert hier eine Methode, um die romanisierten Sonderzeichen wieder in ihre ursprüngliche, sprachentypische Form umzuwandeln.

5.4.6 ErDeMenu

In der Klasse **ErDeMenu** ist die von ErDe verwendete Menüleiste implementiert. Die Menübefehle dienen zum Einstellen des Systemmodus und zum Aufrufen der Ein- und Ausgabefenster (Abbildungen 6 und 7).

5.4.7 ErDeLog

Diese Klasse dient zur Ausgabe des Fehlerstroms von JANUS . Hier wurde eine einfache `JTextArea` mit Scrollmöglichkeiten und Eingabemethoden versehen. Zusätzlich wird hier die Ausgabe direkt in eine Datei geschrieben.

5.4.8 ErDeErg

Diese Klasse hat eine ähnliche Funktionalität wie *ErDeLog*, nur wurden hier, um bessere Vergleichsmöglichkeiten zu haben, die Ausgabefelder für die Hypothese und die Referenz in einem Fenster zusammengefaßt.

5.4.9 ErDeKnopf

Die Instanzen dieser Klasse dienen dazu, die Spracherkenner umzuschalten. Da jeder Knopf die gleiche Funktionalität hat, nur anders initialisiert werden muß, sind sie hier als eigene Klasse implementiert.

5.4.10 ErDeParameter

Diese Klasse stellt Funktionalität zur Darstellung und Manipulation der von den Erkennern verwendeten Parametern (Tabelle 6) zur Verfügung. Die Parameter sind in einem globalen Feld gespeichert. Da einige Änderungen der Parameter es nötig machen, die Initialisierung der Erkenner anzupassen, wird sofort ein Thread gestartet, der dies bewirkt.

5.4.11 ErDeAudio

Die Methoden dieser Klasse dienen zur Manipulation verschiedener Audioparameter wie Lautstärke, Port und Kodierung.

5.4.12 ErDeFahnenMast

Diese Klasse stellt einen speziellen `JLabel` zur Verfügung. Dieser wird genutzt, um den Fortschritt der Identifizierungsphase der *Metasearch* darzustellen. Das Objekt wird mit einer Reihe von Bilddateien initialisiert. Diese können dann in bestimmter Reihenfolge oder direkt dargestellt werden.

6 Zusammenfassung

Um multilinguale Spracherkennung zu betreiben, wurde bisher entweder ein neues System mit multilingualer Akustik entwickelt, oder die bestehenden monolingualen Systeme parallel ausgeführt.

Die erste Variante setzt einen Neuentwurf des Systems voraus, welches dann trainiert werden muß. Das parallele Ausführen von monolingualen Systemen ist nur mit hohem Rechenaufwand möglich und führt zu langen Antwortzeiten.

Die Multlinguale Spracherkennung durch Kommunikation monolingualer Systeme ist zwischen diesen beiden Ansätzen anzusiedeln. Sie bedarf keiner Neuentwicklung der Erkenner und durch das Ausschalten nicht benötigter Erkenner geht sie wirtschaftlicher mit der Rechenzeit um.

Um den, bereits in monolinguale Erkenner investierten Aufwand, in ein multilinguales System hinüberzuretten, werden die einzelnen Erkenner innerhalb eines Kontrollsystems zusammengefaßt. Alle Eigenschaften und Ergebnisse bleiben dabei erhalten. Die Anzahl der so zusammengefaßten Systeme wird nur durch die Menge des im verwendeten Rechners installierten Speichers beschränkt. Der Bedarf an Hauptspeicher des Gesamtsystems entspricht der Summe des Einzelbedarfs der Erkenner. Dieser Ansatz ermöglicht kein Zusammenfassen von Erkennerkomponenten, um den Speicherbedarf zu reduzieren.

Da es nötig ist, die einzelnen Erkenner zur Laufzeit zu vergleichen, wird ihr Ablauf an vorgegebenen Punkten gestoppt. Für jeden Erkenner wird eine Bewertung berechnet. Anhand dieses Vergleiches wird bestimmt, welcher Erkenner bis zum nächsten Halt weiterlaufen darf. Anhand dieser impliziten Spracheidentifikation, wird festgelegt, welcher Spracherkenner am Ende die gesamte Rechenzeit bekommt und die Endergebnisse liefert.

Da aber die Bewertungen, die die Erkenner liefern, nicht direkt vergleichbar sind, war es nötig, eine Normalisierung der Ausgabe einzuführen. Der hier verwendete, prototypische Ansatz gibt noch viel Raum für Verbesserungen. Ihm Rahmen dieser Arbeit war es nicht möglich, aufwendigere Methoden zu testen.

Zweifelsohne steckt in der Verbesserung der Normalisierung noch viel Potential zur Reduktion der Fehlerrate bei der Identifizierung der Sprachen. Bei der Einstellung der Paramter anhand des für die Entwicklung reservierten Testsets hat sich gezeigt, daß die Parameter eng verzahnt sind. Jede Verbesserung bei der Identifizierungsleistung für eine Sprache führte fast zwangsläufig zu einer Verschlechterung bei einer anderen Sprache. Sollte diese Methode der multilingualen Spracherkennung auch weiterhin verwendet werden ist es unabdingbar, effektivere Normalisierungen einzusetzen.

Der, für die multilinguale Spracherkennung am besten geeignete Ansatz, ist aber das Verwenden einer multilingualen Akustik. Nur hier können Erkennerkomponenten gemeinsam genutzt werden, um Aufwand zu sparen. So entstehen Systeme, die auch in schwächeren Rechnern zur multilingualen Spracherkennung genutzt werden können und nicht die Leistung einer Workstation benötigen.

A ErDe : Klassen, Methoden und Felder

A.1 ErDe.java

public class *ErDe*

static boolean *bereit* = false
static final int l*aenderZahl* = 11
static final String *janusName* =
"/home/raschke/bin/janus4.0R600"
static int *aktErk*
protected static ResourceBundle *beschriftung*
protected static Locale *momentLoc*
protected static Runtime *macher*
protected static Process *recPro*
protected static BufferedReader *errRec*
static protected BufferedWriter *schreiben*
protected static JPanel *panSprache*
static JPanel *panHaupt*
static protected JPanel *panID*
static protected JPanel *panMeta*
protected static JProgressBar *fortschritt*
protected static JLabel *labModus*
protected static JLabel *labUttId*
static protected JLabel *labSprachen*
static protected JLabel *labFortschritt*
static protected JLabel *labStatus*
static protected JLabel *labID*
static protected JLabel[] *labMeta* = new JLabel[laenderZahl]
protected static final ImageIcon[] *bilderFeld* =
new ImageIcon[laenderZahl + 1]
static protected final String[] *kurzFeld* =
new String[laenderZahl]
protected static final ImageIcon[] *flaggenFeld* =
new ImageIcon[laenderZahl]
protected static final ImageIcon[] *durchFeld* =
new ImageIcon[laenderZahl]
static protected final String[] *sprachenFeld* =

new String[laenderZahl+1]
static protected final String[] *pfadFeld* =
new String[laenderZahl]
static protected final ImageIcon *euro* =
new ImageIcon("images/euro.gif")
static protected ErDeFenster *mutter*
static protected ErDeJanus *janus*
static protected ErDeMenu *menuLeiste*
static protected ErDeLog *log*
static protected ErDeErg *refHyp*
static protected ErDeThread *warten*
static protected int[] *einstellungen*
static protected int[] *param*
static protected String *beamSize*
static protected String *port[]*
static protected int *systemModus*
satic protected int *sprache*
static protected String *audioPfad*
static protected String *kurz*
static protected String *uttId*
static protected String *dateiName* = " "
static protected String *transPfad* = " "

protected static void *beenden*()

public static void *main*(String[] args)

A.2 ErDeFenster.java

public class *ErDeFenster* extends JFrame implements ActionListener
protected JButton *modusKnopf*
protected ErDeUnicode *hypothese* = new ErDeUnicode()
protected ErDeUnicode *referenz* = new ErDeUnicode()
protected Process *abPro*

protected *ErDeFenster*()

public void actionPerformed(ActionEvent ae)

A.3 ErDeJanus.java

public class *ErDeJanus*

static protected Process *janPro*
protected String *janusString*
protected BufferedWriter *outJan*
protected BufferedReader *inJan*
static protected BufferedReader *errJan*

public *ErDeJanus*(String name)

public void *initial*()
public void *initialMono*()
public void *initialMulti*()
public void *initialMeta*()
public void *initialSprachID*()
public int *sprachIDMikro*()
public int *sprachIDDatei*(String uttId)
public String *erkennungMonoMikro*(int spr)
public String *erkennungMonoDatei*(String uttId, int spr)
public String *erkennungMultiMikro*(int spr)
public String *erkennungMultiDatei*(String uttId, int spr)
public String[] *erkennungMetaMikro*()
public String[] *erkennungMetaDatei*(String uttId)
public String *param*()
public String *config*()
public void *eingabe*(String ein)
public String *fehler*()
public void *wartenInit*()
public void *wartenID*()
public void *wartenMetaEins*()
public void *wartenMetaZwei*()
public void *warten*()

public void entleeren()
public void *ende*()

A.4 ErDeThread.java

public class *ErDeThread* extends Thread

public *ErDeThread*(String name, String satz, int sprache)

public void *run*()

ErDeUnicode *hypo* = new ErDeUnicode()

A.5 ErDeUnicode.java

public class *ErDeUnicode*

public *ErDeUnicode*(String inhalt)
public *ErDeUnicode*()

public void *setzen*(String inhalt)
public String *auslesen*()
public String *original*()
public String *suchen*()
public String *kodieren*(String private)
sprache String *deutsch*(String eingabe)
private String *tuerkisch*(String eingabe)
private String *franzoesisch*(String arbeit)
private String *chinesisch*(String arbeit)
private String *spanisch*(String arbeit)
private String *english*(String arbeit)
private String *kroatisch*(String arbeit)
private String *koreanisch*(String arbeit)
private String *japanisch*(String arbeit)
private static String *ersetzen*(String aString, String find,
String replace, boolean global)

A.6 ErDeMenu.java

public class *ErDeMenu* extends JMenu

implements ActionListener, ItemListener

protected static ButtonGroup *radioGruppeSprache* =
new ButtonGroup()
protected static JMenuItem *menuAudio* = new JMenuItem()
protected static JMenuItem *menuParam* = new JMenuItem()

public *ErDeMenu*()

public void *actionPerformed*(ActionEvent ae)

public void *itemStateChanged*(ItemEvent ie)

A.7 ErDeLog.java

public class *ErDeLog* extends JFrame implemends ActionListener
protected JTextArea *textLog* = new JTextArea()
protected JScrollPane *scrollLog* = new JScrollPane(*textLog*)
protected String *knopfName*

public *ErDeLog*(String titel, String inhalt, String knopfName)

public void *init*()
public void *hinzu*(String ausgabe)
public void actionPerformed(ActionEvent ae)

A.8 ErDeErg.java

public class *ErDeErg* extends JFrame implements ActionListener
protected JTextArea *textErste*
protected JTextArea *textZweite*

public *ErDeErg*(String titel, String erste, String zweite,
String knopfName)

protected void *hinzuErste*(String erste)
protected void *hinzuZweite*(String zweite)
protected void *init*()
public void *actionPerformed*(ActionEvent ae)

A.9 ErDeKnopf.java

public class *ErDeKnopf* extends JButton implements ActionListener

public *ErDeKnopf*(int wahl, int aus)

public void *actionPerformed*(ActionEvent e)

A.10 ErDeParameter.java

public class *ErDeParameter* extends JDialog implements ActionListener
protected JCheckBox[] *parameterBox* = new JCheckBox[6]
protected JTextField *textLength*
protected JTextField *textTopN*
protected JTextField *textStep*
protected JTextField *textTHold*
protected JComboBox *boxBeam*

public *ErDeParameter*(int[] initial, String beam)

public void *actionPerformed*(ActionEvent ae)

A.11 ErDeAudio.java

public class *ErDeAudio* extends JDialog implements ActionListener
String *eingabe* = null

JLabel *ausgabe*
protected JSlider *sliLaut*
protected JRadioButton *laut*
protected JRadioButton *kopf*
protected JCheckBox *shorten*
protected JCheckBox *swab*
protected int[] *einstellungen = new int[4]*

public int[] *audioUebergabe()*

public *ErDeAudio*(int[] initial)

public void *actionPerformed*(ActionEvent ae)

A.12 ErDeFahnenMast.java

public class *ErDeFahnenMast* extends JLabel
String *nation*
final int $MAX = 5$
final int $MIN = 0$
final ImageIcon[] *mastenFeld* = new ImageIcon[MAX+1]
int *hoehe*

public *ErDeFahnenMast*(String nation)

public void *tiefer*()
public void *hoeher*()
public void *max*()
public void *min*()
public void *hoehe*(int neueHoehe)

B MetaSearch : Neuerungen und Änderungen im JANUS RTK-Code

B.1 metaSearch.h

```
typedef struct {
        char* name;
        int frame;
        float score;
        int active;
        HypoList *hypoP;
        Search *sP;
} SearchCtrl;

extern int searchCtrlInit(SearchCtrl* sc, ClientData cd);
extern int searchCtrlDeinit(SearchCtrl* sc);

typedef struct LIST(SearchCtrl) SearchList;

typedef struct {
        char* name;
        int useN;
        int count;
        int step;
        int thold;
        int verb;
        int norm;
        int part;
        int partN;
        float best;
        int passN;
        int frameN;
        float* normA;
        SearchCtrl *finalP;
        SearchList list;
} MetaSearch;
```

extern int *MetaSearch_Init*(void);
extern int *metaSearchInit*(MetaSearch* ms, ClientData cd);
extern MetaSearch **metaSearchCreate*(char* name);
extern int *metaSearchDeinit*(MetaSearch* ms);
extern int *metaSearchFree*(MetaSearch* ms);
extern int *metaSearchLinkN*(MetaSearch* ms);
extern int *metaSearchInclude*(MetaSearch* msP, Search* searchP);
extern int *metaSearchExclude*(MetaSearch *msP, char* name);
extern int *metaSearchGo*(MetaSearch *msP, char* evalS);
extern int *metaSearchPuts*(MetaSearch* msP);

B.2 metaSearch.c

int *searchCtrlInit*(SearchCtrl* sc, ClientData cd)
int *searchCtrlDeinit*(SearchCtrl *sc)
static int *searchCtrlPutsItf*(ClientData cd, int argc, char *argv[])
int *searchCtrlFree*(SearchCtrl *scP)
static int *searchCtrlFreeItf* (ClientData cd)
int *metaSearchInit*(MetaSearch* ms, ClientData cd)
MetaSearch **metaSearchCreate*(char* name)
static ClientData *metaSearchCreateItf*(ClientData cd, int argc,
    char *argv[])
int *metaSearchLinkN*(MetaSearch* ms)
int *metaSearchDeinit*(MetaSearch* ms)
int *metaSearchFree*(MetaSearch* ms)
static int *metaSearchFreeItf* (ClientData cd)
static int *metaSearchConfigureItf*(ClientData cd, char *var, char *val)
static ClientData *metaSearchAccessItf* (ClientData cd, char *name,
    TypeInfo **ti)
int *metaSearchPutsCtrl*(SearchCtrl *scP)
int *metaSearchPuts*(MetaSearch* msP)
static int *metaSearchPutsItf* (ClientData cd, int argc, char *argv[])
int *metaSearchBest*(MetaSearch *msP)
static int *metaSearchBestItf*(ClientData cd, int argc, char *argv[])
int *metaSearchReset*(MetaSearch *msP)

```
static int metaSearchResetItf ( ClientData cd, int argc, char *argv[] )
int metaSearchInclude( MetaSearch* msP, Search *searchP)
static int metaSearchIncludeItf ( ClientData cd, int argc, char *argv[] )
int metaSearchExclude(MetaSearch *msP, char* name)
static int metaSearchExcludeItf (ClientData cd, int argc, char *argv[])
int metaSearchGoInit(MetaSearch *msP, char* evalS)
int metaSearchGoDeinit(MetaSearch *msP)
int metaSearchGoSingle(SearchCtrl *scP, char* evalS, int step,
                int z, int verb)
int metaSearchGo(MetaSearch *msP, char* evalS)
static int metaSearchGoItf (ClientData cd, int argc, char *argv[])

static Method searchCtrlMethod[] = {
        { "puts", searchCtrlPutsItf, "prints information about SearchCtrl" },
        { NULL, NULL, NULL }
};

TypeInfo searchCtrlInfo = { "SearchCtrl", 0, 0,
        searchCtrlMethod, NULL, searchCtrlFreeItf, NULL, NULL,
        itfTypeCntlDefaultNoLink,"SearchCtrl" };

Method metaSearchMethod[] = {
        { "puts", metaSearchPutsItf, "displays all included Search objects"},
        { "include", metaSearchIncludeItf, "includes Search object(s)"},
        { "exclude", metaSearchExcludeItf, "excludes Search object(s)"},
        { "best", metaSearchBestItf, "display best of last MetaSearch go"},
        { "reset", metaSearchResetItf, "resets all MetaSearch params"},
        { "go", metaSearchGoItf, "do a MetaSearch with all included
        Search objects"},
        { NULL, NULL, NULL}
};

TypeInfo metaSearchInfo = { "MetaSearch", 0, 0, metaSearchMethod,
        metaSearchCreateItf, metaSearchFreeItf, metaSearchConfigureItf,
        metaSearchAccessItf, NULL,
        "A 'MetaSearch' object runs and prunes several Searches." };

static int metaSearchInitialized = 0;
```

int *MetaSearch_Init* (void)

B.3 treefwd.h

extern void *dpsInitTFwdOncePerUtterance*(short frameN);
extern void *dpsTFwdOneFrame*(short frameX);
extern void *dpsDeinitTFwdOncePerUtterance*(short frameX);

B.4 itfMain.c

#define DISABLE_READLINE

#ifndef DISABLE_READLINE
#include "itfReadLine.h"
#endif

C Testdaten

Anhang C.1 listet die Daten auf, mit denen die Parameter für die Normalisierung der Bewertung optimiert wurden. In Anhang C.2 finden sich die Daten, mit denen die Ergebnisse in Abschnitt 4.4 erreicht wurden. Die Tabellen listen die Indizes der Sprecher auf.

C.1 Entwicklung

Türkisch	Spanisch	Deutsch	Kroatisch	Chinesisch
TU002_46	SP001_3	DE018_74	KR037_18	CH080_15
TU002_52	SP001_6	DE018_75	KR037_19	CH080_16
TU002_54	SP001_9	DE018_76	KR040_7	CH080_17
TU002_58	SP001_12	DE018_77	KR040_8	CH080_18
TU002_64	SP001_18	DE018_79	KR040_9	CH080_19
TU003_52	SP002_4	DE020_167	KR040_33	CH081_10
TU003_58	SP002_7	DE020_168	KR040_35	CH081_11
TU003_62	SP002_10	DE020_171	KR039_8	CH081_12
TU003_68	SP002_13	DE020_173	KR039_9	CH081_13
TU003_70	SP002_16	DE020_174	KR039_15	CH081_14
TU014_49	SP003_32	DE020_186	KR039_28	CH082_12
TU014_51	SP003_35	DE021_189	KR046_1	CH082_13
TU014_53	SP003_38	DE021_190	KR046_2	CH082_14
TU014_55	SP003_41	DE021_192	KR046_3	CH082_15
TU014_56	SP003_44	DE021_193	KR046_4	CH082_18
TU019_44	SP006_85	DE021_194	KR046_5	CH084_10
TU019_46	SP006_88	DE021_196	KR046_6	CH084_11
TU019_47	SP006_91	DE021_197	KR046_7	CH084_12
TU019_48	SP006_94	DE021_198	KR046_8	CH084_14
TU019_49	SP006_97	DE021_200	KR046_9	CH084_15

C.2 Test

Türkisch				
TU001_41	TU001_43	TU001_45	TU001_49	TU001_51
TU001_53	TU001_57	TU001_59	TU001_61	TU002_9
TU002_11	TU002_13	TU002_21	TU002_23	TU003_70
TU005_80	TU005_81	TU005_82	TU006_36	TU006_38
TU006_40	TU006_42	TU006_44	TU008_4	TU008_5
TU013_1	TU013_3	TU013_5	TU013_7	TU013_9
TU013_11	TU014_19	TU014_43	TU014_45	TU014_47
TU016_58	TU016_59	TU016_60	TU019_48	TU019_49

Spanisch				
SP001_15	SP001_51	SP001_57	SP002_1	SP002_46
SP003_5	SP003_17	SP003_23	SP003_47	SP004_1
SP004_34	SP004_58	SP004_7	SP005_15	SP005_45
SP006_4	SP006_7	SP006_19	SP006_31	SP006_43
SP006_67	SP006_94	SP006_10	SP006_13	SP006_22
SP006_100	SP007_9	SP007_69	SP007_21	SP007_39
SP007_45	SP007_48	SP007_51	SP007_60	SP007_63
SP007_66	SP007_75	SP008_22	SP008_10	SP008_11

Deutsch				
DE018_1	DE018_4	DE018_5	DE018_7	DE018_8
DE018_10	DE018_11	DE018_16	DE018_20	DE018_110
DE018_114	DE018_116	DE018_117	DE020_61	DE020_64
DE020_66	DE020_67	DE020_68	DE020_70	DE020_71
DE020_197	DE020_199	DE020_200	DE021_2	DE021_5
DE021_8	DE021_9	DE021_10	DE021_14	DE021_15
DE021_17	DE021_22	DE021_23	DE021_24	DE021_25
DE021_26	DE021_32	DE021_36	DE021_38	DE021_40

Kroatisch				
KR037_2	KR037_3	KR037_4	KR037_5	KR037_6
KR037_7	KR037_9	KR037_32	KR037_34	KR038_2
KR038_3	KR038_4	KR038_5	KR038_6	KR038_7
KR038_8	KR038_9	KR038_10	KR038_17	KR038_18
KR038_34	KR038_38	KR039_2	KR039_3	KR039_4
KR039_5	KR039_6	KR039_7	KR039_34	KR039_35
KR039_63	KR039_64	KR039_74	KR039_77	KR039_83
KR040_2	KR040_3	KR040_4	KR040_5	KR040_6

Chinesisch				
CH080_10	CH080_11	CH080_12	CH080_13	CH081_17
CH081_19	CH082_10	CH082_11	CH082_16	CH082_17
CH082_19	CH083_10	CH083_11	CH083_13	CH083_14
CH083_15	CH084_13	CH084_16	CH084_17	CH084_19
CH085_10	CH085_11	CH085_12	CH085_18	CH085_19
CH086_10	CH086_11	CH086_12	CH086_13	CH087_12
CH087_14	CH087_15	CH087_16	CH088_13	CH088_15
CH088_19	CH089_10	CH089_12	CH089_13	CH089_14

Literatur

[FrRo96] J. Fritsch, I. Rogina: *The Bucket Box Intersection (BBI) Algorithm for fast approximative evaluation of Diagonal Mixture Gaussians*, ICASSP, Atlanta, 1996

[IPA89] The IPA 1989 Kiel Convention. In: Journal of the International Phonetic Association (19), Seiten 67 - 82, 1989

[JDoc00] Sun Microsystems: *The API-Specification*, online edition, http://www.sun.java.com, USA, März 2000.

[JTut00] Sun Microsystems: *The Java Tutorial*, online edition, http://www.sun.java.com/docs/books/tutorial/index.html, USA, November 1999.

[LeWo95] C. J. Leggetter, P. C. Woodland: *Maximum likelihood linear regression for speaker adaptation of continuous density hidden Markov models* in Computer Speech and Language, Volume 9, Nummer 2, Seiten 171-185, 1995

[Schu95] Ernst Günter Schukat-Talamazzini: *Automatische Spracherkennung*, Vieweg Verlagsgesellschaft, Braunschweig/Wiesbaden, 1995

[ScWa99(1)] T. Schultz, A. Waibel: *Experiments towards a Multi-language LVCSR Interface*, ICMI, Hong Kong, 1999

[ScWa99(2)] T. Schultz, A. Waibel: *Language Adaptive LVCSR through Polyphone Decision Tree Specialization*, MIST, Leusden, Niederlande, 1999

[ScWa98] T. Schultz, A. Waibel: *Language Independant and Language Adaptive Speech Recognition*, ICSLP, Sydney,1998

[ScWa97] T. Schultz, A. Waibel: *Fast Bootstrapping of LVCSR Systems with Multilingual Phonem Sets*, Eurospeech, Rhodes, 1997

[ScWW98] T. Schultz, M. Westphal, A. Waibel: *The* GLOBALPHONE *Project: Multilingual LVCSR with* JANUS *3*, SQEL, Pilsen, 1998

[Stey98] R. Steyer: *Schnellübersicht Java 1.2*, Markt und Technik, München, 1998

[UniC00] Unicode Consortium: *Unicode 2.1*, online-edition, http://www.unicode.org/index.html, USA, November 1999

[WoFi96] M. Woszczyna, M. Finke: *Minimizing Search Errors Due to Delayed Bigramms in Real-Time Speech Recognition Systems*, ICASSP, Atlanta, 1996

[Wosz98] M. Woszczyna: *Fast Speaker Independent Large Vocabulary Continous Speech Recognition*, Dissertation, Universität Karlsruhe, 1999.

[Youn96] S. Young: *Large Vocabulary Continous Speech Recognition: a Review*, Cambridge, 1996.

[ZhWe97] P. Zhan, M. Westphal: *Speaker Normalization based on Frequency Warping*, ICASSP, München, 1997

[ZWFW97] P. Zhan, M. Westphal, M. Finke, A. Waibel: *Speaker normalization and speaker adaption - A combination for conversational speech recognition*, Eurospeech, Rhodes, 1997

[ZiBe99] M. A. Zissman, Kay M. Berkling: *Automatic Language Identifikation*, MIST, Leusden, Niederlande, 1999