Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Technical Memo
TM-05-01

Portierung von Merkmalsextraktion auf die PocketPC-Plattform

Michael Feld

October 2005

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210
E-Mail: info@dfki.uni-kl.de

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341
E-Mail: info@dfki.de

WWW: http://www.dfki.de

Deutsches Forschungszentrum für Künstliche Intelligenz

DFKI GmbH

German Research Center for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest nonprofit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focusing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialization.

Based in Kaiserslautern and Saarbrücken, the German Research Center for Artificial Intelligence ranks among the important "Centers of Excellence" worldwide.

An important element of DFKI's mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI's six research departments are directed by internationally recognized research scientists:

- Image Understanding and Pattern Recognition (Director: Prof. Thomas Breuel)
- Knowledge Management (Director: Prof. A. Dengel)
- Intelligent Visualization and Simulation Systems (Director: Prof. H. Hagen)
- Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- Language Technology (Director: Prof. H. Uszkoreit)
- Intelligent User Interfaces (Director: Prof. W. Wahlster)

Furthermore, since 2002 the Institute for Information Systems (IWi) (Director: Prof. August-Wilhelm Scheer) is part of the DFKI.

In this series, DFKI publishes research reports, technical memos, documents (eg. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster
Director

Portierung von Merkmalsextraktion auf die PocketPC-Plattform

Michael Feld

DFKI-TM-05-01

This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITW-01 IN C02).

ISSN 0946-0071

Portierung von Merkmalsextraktion auf die PocketPC-Plattform

Michael Feld

29. Oktober 2005

Zusammenfassung

Diese Arbeit stellt eine Softwarelösung zur Extraktion von phonetischen Merkmalen, die als Grundlage für andere Applikationen dienen, aus einem digitalen Sprachsignal auf der PocketPC-Plattform vor. Die Entwicklung der Lösung wird schrittweise von der Festlegung der Kriterien über den Entwurf bis hin zur Implementierung beschrieben. Das Ergebnis ist eine fertige Bibliothek, welche die geforderten Merkmale bietet. Zuvor wird ein kurzer Überblick über das Projekt M3I, in dessen Rahmen diese Arbeit entstand, sowie die computergestützte Sprachanalyse im Allgemeinen gegeben. Darüber hinaus werden einige softwaretechnische Aspekte der Applikationsportierung angesprochen und im Hinblick auf die mobile Plattform konkretisiert.

1 Zweck der Merkmalsextraktion

Sprachmerkmalsextraktion auf akustischer Ebene ist der Ausgangspunkt für viele Algorithmen und Verfahren, welche zur wissenschaftlichen Sprachuntersuchung und -analyse sowie in kommerziellen Anwendungen eingesetzt werden. Diese Arbeit ist im Rahmen von M3I[1] entstanden, einem am DFKI[2] durchgeführten Projekt zur Entwicklung von Verfahren zur multimodalen Interaktion mit mobilen Geräten. Ziel des Teilprojekts AGENDER[3] ist es hierbei, den Benutzer eines mobilen Gerätes sowie den aktuellen Kontext zu klassifizieren, ohne den Benutzer zur Eingabe dieser Informationen aufzufordern. Als Quelle für diese Datenakquisition wurde die Spracheingabe des Benutzers bestimmt, welche nebenläufig im normalen Betrieb ausgewertet wird. Auf Basis einer solchen Sprachprobe ist es dann möglich, Rückschlüsse auf das Alter und Geschlecht des Benutzers zu ziehen, sowie über die Geräuschkulisse der Umgebung. Gegenwärtig werden für alle diese Eigenschaften jeweils zwei Kategorien unterschieden, z.B. jung und alt für die Altersklassifikation. Diese Information kann dann wiederum anderen Anwendungen zur Verfügung gestellt werden, wobei die möglichen Einsatzbereiche sehr vielfältig sind. Das Alter kann von Computeranwendungen beispielsweise generell gut zur Adaption der Lesbarkeit für ältere Menschen genutzt werden.
Die Architektur des vollständigen Systems gestaltet sich wie folgt: Das mobile Gerät repräsentiert aus Sicht von M3I den Client. Auf dem Client wird eine Applikation ausgeführt, die vom Endanwender gestartet wurde, z.B. ein digitaler Einkaufsassistent. Wenn diese Applikation die AGENDER-Funktionen in Anspruch nehmen möchte, so kann sie auf die Funktionen der M3I-Bibliothek zugreifen, welche als DLL[4] für den PocketPC implementiert wurde. Die Anwendung kann dann eingehende Sprachproben des Benutzers (z.B. für Sprachsteuerung der Anwendung) an die Bibliothek weiterleiten, und erfährt auf Nachfrage die aktuelle Einschätzung des Benutzers bzgl. Alter und Geschlecht. Bisher wurde die Verarbeitung der Sprachdaten ausschließlich auf einem externen Server durchgeführt. Dieser M3I-Server kann Verbindungsanforderungen mehrerer anonymer mobiler Clients annehmen, z.B. über ein Funknetzwerk (WLAN), und führt auf Anfrage hin die Klassifizierung durch, wobei hier

[1] A Mobile, Multi-modal and Modular Interface, `https://xantippe.cs.uni-sb.de/~cmueller/m3i/php/website2.php`

[2] Deutsches Forschungszentrum für Künstliche Intelligenz

[3] Age+Gender

[4] Dynamic Link Library, siehe auch Abschnitt 9

– je nach Volumen – ein Cluster zum Einsatz kommen kann. Eingehendere Informationen zur M3I-Architektur sind in [5] nachzulesen.
Ein zukünftiges Ziel ist es, diese Klassifizierung auch durchführen zu können, wenn die Verbindung zum Server eingeschränkt ist, oder sogar wenn gar kein Server vorhanden ist. Denkbar sind dann auch verschiedene Szenarien paralleler oder kooperativer Verarbeitung.
Die primären Klassifizierungsmethoden von AGENDER sind künstliche neuronale Netze, die jeweils für Alter und Geschlecht existieren. Neuronale Netze können unterschiedliche Strukturen aufweisen; typischerweise bestehen sie aus Knoten mit Formeln, welche jeweils ihre Eingabewerte unter einer bestimmten Gewichtung kombinieren und das Ergebnis mit einem vordefinierten Schwellenwert vergleichen und so zu einem binären Ergebnis kommen. Das Netz entsteht dadurch, dass mehrere solcher Knoten miteinander verbunden werden. Als Eingabewerte für diese Klassifizierer werden die akustischen Sprachmerkmale verwendet, die aus der Eingabedatei extrahiert werden. Dies führt in der servergestützten Architektur ein Prozess durch, der auf dem Server selbst oder einem lokalen Cluster läuft. Ist jedoch der Server nicht verfügbar, so würde diese Aufgabe dem Client zufallen.
AGENDER ist das für diese Arbeit ausschlaggebende Einsatzgebiet der Merkmalsextraktion, jedoch sind auch noch zahlreiche weitere Anwendungen denkbar, die eine Merkmalsextraktion auf dem PocketPC nutzen könnten, z.B. zur wissenschaftlichen Analyse von beliebigen Audiodaten oder um – analog zu AGENDER – weitere Informationen aus der Sprachprobe zu entnehmen. Für PC und Macintosh gibt es zu diesem Zweck die Softwaresuite PRAAT[5], die über Skripte in andere Anwendungen eingebunden werden kann.

2 Anforderungen

Die Software zur Merkmalsextraktion, die für diese Arbeit entwickelt wurde, soll neben der eigentlichen Werteberechnung noch einige Rahmenanforderungen erfüllen, um erfolgreich mit der bereits vorhandenen Architektur eingesetzt werden zu können.
Als erstes muss die Ausführung unter *Intel StrongARM* und kompatiblen Prozessoren gewährleistet sein, da diese die Mehrzahl der eingesetzten Geräte ausmachen. Als Entwicklungsgerät diente der *HP Jornada 568*, welcher mit einem *ARM SA1110* Prozessor ausgestattet ist. Von Seiten des Betriebssystems wird Kompatibilität mit *Windows CE 3.0* und höher (*PocketPC 2002 / Windows Mobile 2003*) vorausgesetzt, welche ebenfalls einen de-facto Standard für PocketPCs darstellen. Diese beiden Faktoren bewirken bereits eine starke Einschränkung in Bezug auf die Auswahl der Entwicklungstools. Einziger echter Kandidat sind die (kostenlos) von Microsoft angebotenen Werkzeuge *eMbedded Visual C++ 4.0* (kurz: *eVC++*) und *PocketPC 2002 SDK*. Dabei stellt *eVC++* eine integrierte Entwicklungsumgebung mit eigenem Compiler dar, die stark an das verwandte *Visual C++* erinnert, jedoch für mobile Geräte optimiert wurde. Für den Compiler müssen einige benutzerdefinierte Optionen angegeben werden, um die Kompilierung für *PocketPC 2002* zu ermöglichen; herstellerseitig wird nur *PocketPC 2003* und höher unterstützt. Das *PocketPC 2002 SDK* enthält plattformspezifische Dateien für *Windows CE 3.0*, wie z.B. Header-Dateien und Bibliotheken, welche zum Kompilieren benötigt werden, sowie umfangreiche Dokumentation und eine Emulator-Software. Obwohl die Entwicklungsumgebung an sich gut geeignet ist für die PocketPC-Entwicklung und auch keine gravierenden Schwächen aufweist, so ist das Projekt mit dieser Entscheidung aber festgelegt in Hinsicht auf den existierenden Quellcode, der direkt in das Programm übernommen werden kann. Die Alternative, PocketPCs mit *Linux* als Betriebssystem auszuwählen, hätte in Bezug auf die Entwicklungsmöglichkeiten sicher eine größere Flexibilität bedeutet, beispielsweise was die Auswahl des Compilers betrifft. Allerdings wäre dadurch die Integration mit bestehenden Anwendungen erschwert worden, denn gegenwärtig sind die große Mehrzahl der Softwarelösungen für *Windows CE* konzipiert.
Eine weitere Anforderung an die Software ist, dass sie möglichst universell in andere Programme integriert werden kann, d.h. sie soll die Extraktionsmethoden in Form einer Bibliothek zur Verfügung stellen. Für die exakte Implementierung gibt es dann noch mehrere Möglichkeiten, auf die in Abschnitt 9 näher eingegangen wird.

[5] `http://www.praat.org`

3 Sprachverarbeitung mit dem Computer

Die Merkmalsextraktion ist eine Form der Sprachverarbeitung. Diese erfolgt auf dem Computer stets digital, d.h. die Sprachinformation liegt in Form einer binären Audiodatei vor oder wird – sofern eine Aufnahme in Echtzeit verarbeitet wird – als solche temporär im Arbeitsspeicher angelegt. Diese binäre Repräsentation enthält häufig weniger Information als die ursprüngliche analoge Sprachprobe (siehe [7]), da bei der Digitalisierung Informationen verloren gehen können. Deshalb spielt die Qualität der Aufnahme eine große Rolle und wirkt sich auch auf die Ergebnisse der Merkmalsextraktion aus. So können Abweichungen in den berechneten Werten häufig auf verschiedene Formate der Audiodateien zurückgeführt werden. Aus diesem Grund kann es empfehlenswert sein, die Daten in einem zuvor festgelegten gemeinsamen Format abzuspeichern oder sie zumindest vor der Analyse in ein gemeinsames Format zu überführen (Konvertierung).
Die Kriterien, welche die Qualität der Audiodaten bestimmen, sind *Samplingrate*, *Bitrate* und *Anzahl der Kanäle*. In der Natur sind akustische Phänomene Schallwellen. Um sie in eine digitale Form zu bringen, müssen diese Wellen durch Abtastpunkte (*Samples*) approximiert werden (*Quantisierung*). Dabei können einfache Speicherverfahren oder aufwändige Komprimierungsalgorithmen wie [8] zum Einsatz kommen. Die Anzahl der aufgezeichneten Samples pro Sekunde stellt die Samplingrate dar, welche in Hertz angegeben wird (44.100 Hz entspricht CD-Qualität). Die Samplingrate bestimmt auch die maximale Frequenz, die aufgezeichnet werden kann, wobei nach dem in [7] vorgestellten *Sampling-Theorem* die Samplingrate mehr als das doppelte der höchsten im Signal auftretenden Frequenz (der *Nyquist-Frequenz*) betragen soll. Die menschliche Stimme deckt bei gesprochenen Äußerungen nach [2] und [3] etwa einen Bereich von 75 bis 500 Hz ab, so dass mit 16 kHz eine ausreichende Auflösung erreicht wird. Die Bitrate gibt an, mit welcher Genauigkeit ein einzelnes Sample gespeichert werden kann (z.B. 16 Bit bei CDs). Eine höhere Bitrate ergibt eine breitere Klangdynamik. Mehrere Kanäle werden verwendet, um Raumklang zu simulieren. Jeder Kanal kann eigene Audiodaten enthalten und die Kanäle können prinzipiell wie verschiedene Aufnahmen behandelt werden.
In Hinblick auf Qualität sollten die Samplingrate und Bitrate so hoch wie möglich gewählt werden, da dies genauere Ergebnisse liefert. In der Praxis gelten hierfür jedoch einige Beschränkungen:

1. Die Hardware (insb. das Mikrofon) und der Aufnahmetreiber müssen die geforderten Werte unterstützen. Gerade was die Bitrate angeht wird dies sehr oft durch die Hardware auf 16 Bit begrenzt. Auch sollte man bei mobilen Geräten davon ausgehen, keine besonders hochwertigen Mikrofone vorzufinden. Selbst die OEM[6]-Treiberimplementierung ist oftmals nur minimal vorhanden und fehleranfällig.

2. Je höher Samplingrate und Bitrate gewählt werden, desto größer ist der anfallende Speicherverbrauch. Wird die Sprachprobe nach der Verarbeitung wieder gelöscht, so ist das zwar nur ein unbedeutender Punkt, aber wenn die Sprachdateien archiviert werden sollen, so wird der Speicherverbrauch ein gewichtiges Argument sein. Nähere Untersuchungen zu diesem Thema sind in [7] zu finden.

3. Mit höherer Aufnahmequalität steigt auch die zu verarbeitende Datenmenge und damit die benötigte Rechenzeit. Dieser Punkt ist besonders wichtig bei Echtzeitanwendungen auf mobilen Geräten, da deren Leistungsfähigkeit immer noch sehr gering ist verglichen mit Desktop-PCs.

4. Wie zuvor erwähnt ist es sinnvoll, die Audiodaten vor der Merkmalsextraktion und Analyse in ein gemeinsames Format zu bringen, um den Vergleichswert der erhaltenen Werte zu erhöhen. Bei der Wahl dieses gemeinsamen Formates sollte daher das jeweilige Minimum der anzunehmenden Eingabeformate bzgl. Samplingrate und Bitrate für unterschiedliche Geräte und Applikationen gewählt werden, da Formate mit niedrigeren Raten nicht (oder nur unzulänglich) in ein Format mit größeren Raten konvertiert werden können.

[6]Original Equipment Manufacturer

Für die Sprachmerkmalsextraktion genügt ein einziger Audiokanal. Sollte die Aufnahme im Stereo-Format vorliegen, so können die Kanäle vor der Merkmalsextraktion gemischt werden, oder es wird ein Kanal ausgewählt.

4 Verwendete Sprachmerkmale

Aus einer digitalisierten Sprachprobe können mit Hilfe von Algorithmen eine ganze Reihe von Merkmalen extrahiert werden. Die Softwaresuite PRAAT wurde genau für diesen Zweck entwickelt, und die Online-Hilfe zum Programm gibt einen kleinen Einblick in die vielfältigen Möglichkeiten, Sprachsignale mittels Algorithmen zu untersuchen.
In dieser Arbeit soll jedoch nur auf die Merkmale eingegangen werden, welche für die spätere Klassifizierung von Alter und Geschlecht durch die neuronalen Netze relevant sind und deren Bedeutung in [6] näher beschrieben wird. Dabei handelt es sich um die folgenden drei Gruppen von Merkmalen:

Pitch

Das Merkmal *Pitch* beschreibt die Tonlage eines Sprachsignals. Diese ergibt sich aus der Periodendauer bzw. Frequenz der Schallwellen, welche in der Natur kontinuierlich sind, und kann daher nur schwer an diskreten Stellen erfasst werden. Zur Bestimmung der Tonlage an einem bestimmten Punkt muss immer auch ein Zeitfenster links und rechts dieses Punktes betrachtet werden, um eine Periodizität feststellen zu können. Pitch wird in Hertz, Mel oder Halbtönen angegeben. Aus einer endlichen Menge von Pitch-Werten für ein Sprachsignal kann durch parabolische Interpolation eine Funktion der Zeit erstellt werden, die als *Pitch-Kontur* des Signals bezeichnet wird. Für die Klassifizierung durch AGENDER sind folgende Eigenschaften der Funktion von Interesse:

- Höchste auftretende Frequenz (Maximum)
- Niedrigste auftretende Frequenz (Minimum)
- Durchschnittliche Tonlage
- Standardabweichung
- 50%-Quantil (Median)
- Absolute mittlere Steigung

Jitter

Durch *Jitter* können Abweichungen des Pitch-Wertes beschrieben werden, wobei jeweils unterschiedlich große Umgebungen betrachtet werden können. Voraussetzung ist, dass das Signal bereits in Perioden eingeteilt wurde. Der Jitter wird dann berechnet, indem der Betrag der Frequenzdifferenzen aller Paare von aufeinander folgenden Perioden bestimmt und aus diesen dann der Mittelwert gebildet wird. Dieser wird dann üblicherweise prozentual zur durchschnittlichen Frequenz des Signals angegeben. Folgende Variationen der Jitter-Funktion werden unterschieden:

- Lokaler *Jitter* (absolut)
- Lokaler *Jitter*: Jitter im Verhältnis zur durchschnittlichen Frequenz
- *Relative Average Perturbation (RAP)*: Hierbei wird die Abweichung einer Periode vom Mittelwert aus der Periode selbst und den beiden benachbarten Perioden betrachtet.
- *Five-point Period Perturbation Quotient (PPQ5)*: Differenz aus der Periode und dem Mittelwert aus ihr selbst und den vier nächsten Nachbarn
- *Difference of Differences of Periods (DDP)*: Wählt von der Abweichungen zweier aufeinander folgender Perioden nochmals den Betrag der Differenz.

Shimmer

Der Parameter *Shimmer* ist sehr eng mit dem Jitter verwandt. Es handelt sich um das Gegenstück zur Frequenz im Bereich der Amplitude. Hier werden Schwankungen der Amplitude innerhalb verschieden großer benachbarter Umgebungen betrachtet und üblicherweise auch wieder zur durchschnittlichen Amplitude des Audiosignals in Bezug gesetzt. Die einzelnen relevanten Funktionen sind:

- Lokaler *Shimmer*: Mittelwert aus dem Betrag der Differenz der Amplituden in je zwei aufeinander folgenden Perioden im Verhältnis zur durchschnittlichen Amplitude
- Absoluter lokaler *Shimmer*: Mittelwert aus dem Betrag des Zehnerlogarithmus der Differenz der Amplituden in je zwei aufeinander folgenden Perioden. Das Ergebnis wird mit 20 multipliziert.
- *Three-point Amplitude Perturbation Quotient (APQ3)*: Analog zum Jitter RAP wird als Subtrahend der Mittelwert aus der Periode und ihren beiden Nachbarn gebildet.
- Analog *APQ5* und *APQ11* mit den vier bzw. zehn nächsten Nachbarn
- *Difference of Differences of Periods (DDP)*: Wählt von der Abweichungen zweier aufeinander folgender Perioden den Betrag der Differenz der Amplituden.

Um die genannten Merkmale zu erhalten, geht der M3I-Server wie folgt vor: Zuerst wird ein eingehendes digitales Sprachsignal durch Up- bzw. Downsampling in ein fest definiertes Format gebracht. Die Ausgabedatei wird dann an ein PRAAT-Skript übergeben, welches automatisch alle oben aufgeführten Sprachmerkmale extrahiert. Im nächsten Abschnitt soll erläutert werden, wie dieser Schritt auf dem PocketPC nachvollzogen wurde.

5 Gründe für die Portierung von PRAAT

Die wichtigste Entscheidung, die zu Beginn des Projekts getroffen werden musste, war die Art und Weise, auf die die Extraktionsalgorithmen implementiert werden sollten. Hier kamen prinzipiell zwei Möglichkeiten in Frage: Verwendung eines bereits vorhandenen Werkzeugs oder vollständige Neuimplementierung aller benötigten Algorithmen. Im Falle dass auf ein bereits existierendes Programm oder frei verfügbaren Quellcode zurückgegriffen würde, müssten noch diverse Fragen zur Integration erörtert werden, und hierbei handelt es sich um einen gewichtigen Punkt, der über Erfolg oder Scheitern einer Methode entscheiden kann. Die typischen Integrationsverfahren sind im Einzelnen:

Einbindung einer Bibliothek: Im Optimalfall werden vorhandene Algorithmen über eine öffentliche Schnittstelle in Form einer Bibliothek (z.B. DLL) zur Verfügung gestellt. Wenn solch eine Bibliothek existiert und alle gewünschten Funktionen enthält, so kann diese meist direkt ohne großen Aufwand in das Rahmenprojekt integriert werden. Dies setzt natürlich voraus, dass die Bibliothek, da sie kompilierten, ausführbaren Code enthält, auf der Zielplattform lauffähig ist. Auch muss die Sprache des Rahmenprojekts (im Falle des M3I-Clients C++) Aufrufe aus DLLs unterstützen.

Einbindung einer Programmdatei: Steht für das gewünschte Programm keine Bibliothek zur Verfügung, sondern nur eine ausführbare Programmdatei, so kann diese möglicherweise ebenfalls genutzt werden. Dazu muss das Programm Parameter wie die zu verarbeitende Sprachdatei aus der Umgebung einlesen können (z.B. über die Kommandozeile, eine Umgebungsvariable oder eine Konfigurationsdatei), die Ergebnisse in programmatisch verwertbarer Form zurückliefern (z.B. über die Konsolenausgabe oder eine temporäre Datei) und den Vorgang vollautomatisch ohne GUI[7] und Benutzereingriffe durchführen können. Wie bei der Bibliothekseinbindung muss auch hier die Zielplattform unterstützt

[7]Graphical User Interface, grafische Benutzeroberfläche

werden, allerdings muss keine Schnittstelle zwischen den Programmiersprachen bestehen, da diese bereits durch den Aufruf der ausführbaren Datei über das Betriebssystem vorgegeben ist.

Diese Lösung hat immer gewisse Geschwindigkeitsnachteile gegenüber einer Bibliothek, da der Startvorgang des Programms zusätzliche Zeit im Betriebssystemkernel benötigt.

Portierung eines Programms: Wenn ein Programm für die benötigte Plattform nicht existiert, so kann es als Bibliothek oder ausführbare Datei nicht eingebunden werden. Falls der Quellcode des Programms erhältlich ist, so besteht jedoch die Möglichkeit einer Portierung auf die Zielplattform.

Hierbei muss vor allem der vorhandene Code an die API[8] des Zielbetriebssystems und -prozessors angepasst werden und evtl. compilerspezifisch verändert werden. Je nach Unterschiedlichkeit der Plattformen kann sich dies mehr oder weniger umfangreich gestalten. Beispielsweise gibt es einige Prozessoren, welche keine Fließkommaoperationen unterstützen, sodass diese dann bei einer Portierung ggf. komplett ersetzt werden müssten. Bei Sprachen, die in einer virtuellen Maschine ausgeführt werden (z.B. *Java*, *.NET*) sind im Allgemeinen keine Änderungen dieser Art notwendig, da sie speziell auf Plattformunabhängingkeit ausgelegt sind. Weiterhin müssen bei einer Portierung plattformspezifische Besonderheiten auf Ressourcenebene beachtet werden, die meist nicht durch den Compiler entdeckt werden. So hat der PocketPC beispielsweise eine kleinere Anzeige als ein Desktop-Computer, und alle Anwendungen sollten auf diesen Umstand hin optimiert werden.

PRAAT ist eine frei verfügbare Anwendung zur Sprachanalyse und -synthese, die 1992 von Paul Boersma und David Weenink vom *Department of Phonetics* an der Universität von Amsterdam geschrieben wurde und seitdem ständig weiterentwickelt wird. Sie beinhaltet umfangreiche Funktionen und Algorithmen zur Arbeit mit Sprachdateien, sowie Methoden zur Visualisierung der Ergebnisse. PRAAT erlaubt auch das Schreiben eigener Skriptdateien, um neue Funktionen zu definieren oder Sprachproben automatisiert bzw. stapelweise abzuarbeiten, was es für die Integration in andere Projekte qualifiziert.
Verweise in der Literatur und Recherchen im Internet zum Thema „computergestützte Sprachanalyse" belegen die weite Verbreitung von PRAAT. Die Tatsache, dass es – einschließlich Quellcode – frei verfügbar ist (veröffentlicht unter der GNU[9] Lizenz), unterstützt seine Popularität. Auf der anderen Seite ist PRAAT aber auch eine von relativ wenigen professionellen Lösungen, die in diesem Bereich angeboten werden. Die genannten Gründe sowie die Tatsache, dass es alle benötigten Algorithmen beherrscht, waren ausschlaggebend für den Einsatz im M3I-Server.
Da PRAAT schon einige Zeit erfolgreich im M3I-Server eingesetzt wurde, lag es nahe, das Phonetikwerkzeug auch auf dem PocketPC zu verwenden. Das war allerdings nicht ohne weiteres möglich, da PRAAT von Haus aus nur als Version für Windows auf dem Desktop-PC, Linux und Macintosh existiert. Außerdem beeinflusst die Integration über Skriptdateien auf dem mobilen Gerät die Geschwindigkeit wesentlich stärker und ist nicht zweckmäßig.
Eine Portierung von PRAAT auf den PocketPC und Umstrukturierung als Library war durch den Quellcode in C prinzipiell möglich. Eine erste Analyse des Quellcodes ergab, dass eine vollständige Portierung mit einem gewissen Aufwand verbunden war und daher auch weitere Alternativen in Betracht gezogen werden sollten. Da die Suche nach weiteren geeigneten Werkzeugen im Wesentlichen erfolglos verlaufen war, kam als Alternative nur eine Neuimplementierung der Algorithmen in Frage.
Die Idee einer eigenen Implementierung konnte sich aus mehreren Gründen nicht durchsetzen. Zum einen muss zum Erhalt der Sprachmerkmale Jitter und Shimmer eine Periodifizierung (Periodenerkennung) der Spracheingabe durchgeführt werden, ein nicht triviales Verfahren, das den Aufwand der Neuimplementierung schnell mit dem der Portierung aufwiegt. Für PRAAT spricht darüber hinaus die überragend gute Qualität, die auf jahrelanger Forschung

[8] Application Programming Interface
[9] `http://www.gnu.org/licenses/gpl.html`

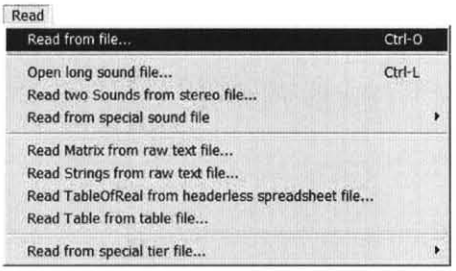

Abbildung 1: Menü *Read*

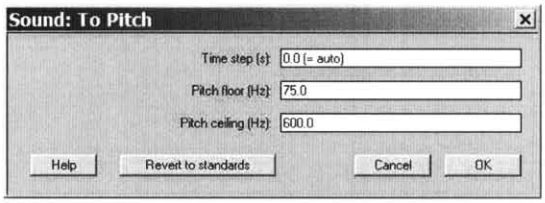

Abbildung 2: Parameter für *To Pitch...*

und Benutzerfeedback beruht. Gerade was die Leistung angeht sind die verwendeten Algorithmen stark optimiert, und einige mathematische Funktionen nutzen intern die performante Bibliothek GSL[10].
Es gibt noch einen weiteren Punkt, der wesentlich für PRAAT spricht, und das ist die Kompatibilität mit der restlichen M3I-Architektur. In der Hauptsache bezieht sich dies auf die errechneten Werte. Für die spätere Klassifizierung ist es sehr wichtig, dass die Werte kompatibel zu denjenigen sind, mit denen die Klassifizierer trainiert wurden, da es bei Abweichungen zu ungewollten Fehlklassifizierungen kommen kann. Während der Entwicklung von AGENDER wurde die Beobachtung gemacht, dass sich die Ergebnisse einer Merkmalsextraktion selbst zwischen einzelnen Versionen von PRAAT unterscheiden können. Dies ist kein fehlerhaftes Verhalten, da die zugrunde liegenden Algorithmen nicht genau definiert sind. Der Einsatz einer anderen Software auf dem PocketPC könnte also zu erheblichen Abweichungen in den Resultaten führen. Die Verwendung von PRAAT dagegen garantiert, dass Server und Client aus der gleichen Datei (bei Einsatz der selben PRAAT-Version) identische Werte extrahieren. Natürlich kommt die gewonnene Kompatibilität auch der Bedienung und Integration zugute, weil plattformübergreifend gleichnamige Funktionen aufgerufen werden.

6 Beispiel einer Merkmalsextraktion

In diesem Abschnitt soll der Ablauf einer typischen Merkmalsextraktion mit PRAAT vorgestellt und erläutert werden. Der Vorgang wird in diesem Beispiel mit der GUI-Version von PRAAT unter Windows durchgeführt; es werden aber auch immer die entsprechenden Funktionen im PRAAT-Quellcode gegenübergestellt, welche die gleiche Funktionalität bieten und die in der portierten Bibliothek aufgerufen werden können.
Die PRAAT-Umgebung arbeitet mit *Objekten*. Beim Start der Anwendung ist die Umgebung leer. Der erste Schritt besteht darin, eine Sounddatei mit der Sprachprobe zu öffnen. Dies geschieht mit Hilfe des Befehls *Read from file...* im Menü *Read* (Abb. 1). Im Programmcode entspricht dies der Funktion `Sound_readFromSoundFile()`, die als Argument den Dateinamen erhält und ein `Sound`-Objekt (eine `struct` in C) zurückliefert. PRAAT unterstützt mehrere Dateiformate, darunter die verbreiteten Formate *AIFF*[11] und *WAV*[12]. Nach dem Öffnen der Datei wurde der Umgebung ein neues `Sound`-Objekt hinzugefügt. Wird das Objekt ausgewählt, so wird eine Liste von möglichen Aktionen angezeigt, mit denen das Objekt analysiert, visualisiert und in andere Objekte transformiert werden kann. So kann mit *Play* die kurze Sprachprobe testweise abgespielt werden.
Zuerst soll nun die Tonlage (*Pitch*) und die damit verbundenen Werte bestimmt werden. Hierzu wird mit PRAAT ein `Pitch`-Objekt berechnet, welches eine Funktion in Abhängigkeit von der Zeit darstellt (Abb. 3). Da die resultierende Kurve generell guter Ausgangspunkt zur späteren Berechnung von periodischen Phänomenen ist, findet man die Funktion *To Pitch...* unter *Periodicy*. Wie bei vielen Funktionen stellt PRAAT hier mehrere in Frage kommende Algorithmen zur Auswahl, die sich in Umfang und Verfahren leicht unterscheiden. In diesem

[10] GNU Scientific Library, `http://www.gnu.org/software/gsl/`
[11] Audio Interchange File Format
[12] Waveform

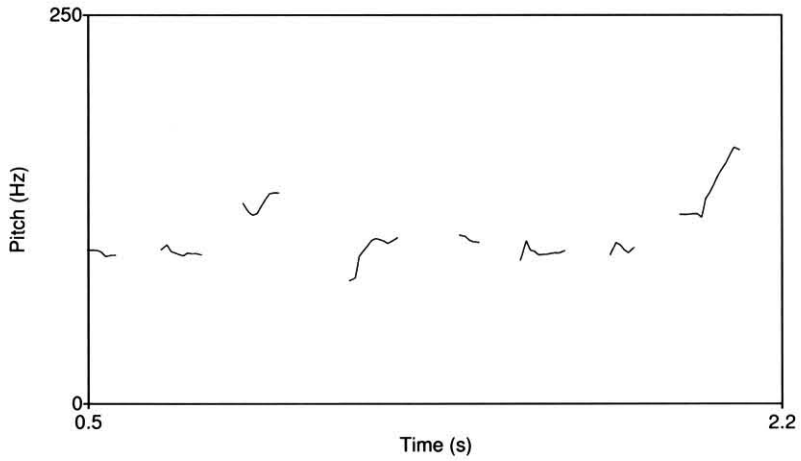

Abbildung 3: Pitchkontur eines Sprachsignals

Fall ist die Standardfunktion gut geeignet. Für die Berechnung können nun in einem gesonderten Dialogfeld noch weitere Parameter spezifiziert werden (Abb. 2). Hierbei handelt es sich um die Schrittweite (*Time Step*), welche die Genauigkeit des Resultats bestimmt, sowie den zu erkennenden Bereich. Dieser wird auf 75-500 Hz eingeschränkt, wo üblicherweise Stimmen zu finden sind. Pausen und einige Umgebungsgeräusche werden dadurch herausgefiltert. Im Quellcode wird ein `Pitch`-Objekt mit der Funktion `Sound_to_Pitch()` aus dem `Sound`-Objekt erzeugt, welche die gleichen Parameter nimmt wie das Dialogfeld.
Aus dem `Pitch`-Objekt können dann die einzelnen Werte für die Klassifizierer mittels einfacher arithmetischer und statistischer Funktionen wie Mittelwert, Maximum und Standardabweichung berechnet werden. Der Mittelwert beispielsweise wird über den Befehl *Get mean...* im Menü *Query* berechnet (Abb. 4). Als zusätzliche Option kann der Zeitbereich eingegrenzt und die Ausgabeeinheit geändert werden. Die portierte Library bietet für jeden dieser Werte ebenfalls eine eigene Bestimmungsfunktion an, z.B. `Pitch_getMean()` für den Mittelwert.
Für die Bestimmung von *Jitter* und *Shimmer* (siehe Abschnitt 4) muss das `Sound`-Objekt nun mit Hilfe des bereits erstellten `Pitch`-Objekts in ein `PointProcess`-Objekt transformiert werden. Der `PointProcess` enthält eine Reihe von Punkten auf der Zeitachse, die für akustische Wiederholungsintervalle stehen, und wird über einen Periodenerkennungsalgorithmus in der Nähe von Stellen mit großer Amplitude berechnet. Intervalle ohne Stimme werden dabei ignoriert. Um den PointProcess zu erstellen, müssen das `Sound`- und `Pitch`-Objekt gemeinsam markiert werden und dann der Befehl *To PointProcess (cc)* aufgerufen werden (Abb. 5). Die entsprechende Funktion der Bibliothek lautet `Sound_Pitch_to_PointProcess_cc()`. Abb. 6 stellt einen solchen PointProcess dar. Die vertikalen Linien grenzen jeweils die Perioden voneinander ab.
Der Jitter kann direkt aus dem PointProcess bestimmt werden. Dazu wird im Menü *Query* einer der Befehle zum Ausgeben der Jitter-Variationen ausgewählt, z.B. *Get jitter (local)...*. Als Parameter lassen sich der Zeitbereich einschränken sowie besonders kurze oder lange Perioden per Grenzwert aus der Betrachtung ausschließen (*Shortest period/longest period*). Die Funktion für den lokalen Jitter in der Library lautet `PointProcess_getJitter_local()`.
Um die Shimmer-Werte zu erhalten ist zunächst noch ein Umweg über das `AmplitudeTier`-Objekt erforderlich. Dieses Objekt enthält die Amplitudenwerte für alle Perioden des PointProcess. Man erhält es, indem `Sound` und `PointProcess` selektiert werden und dann *To AmplitudeTier (period)...* aufgerufen wird. Die anzugebenden Parameter entsprechen denen der Jitter-Berechnung, da sie ebenfalls auf einem `PointProcess`-Objekt arbeiten. Im Programmcode wird das `AmplitudeTier`-Objekt durch den Aufruf der Funktion `PointProcess_Sound_to_AmplitudeTier_period()` erzeugt. Ein Beispiel für ein `AmplitudeTier`-Objekt ist in Abb. 7 unter der Sounddarstellung abgebildet. Danach wird der Shimmer-Wert analog zu Pitch und Jitter über das Menü *Query* extrahiert oder über die

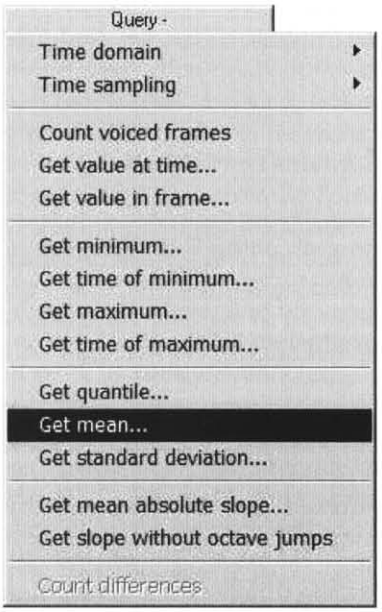

Abbildung 4: Menü *Query* für das `Pitch`-Objekt

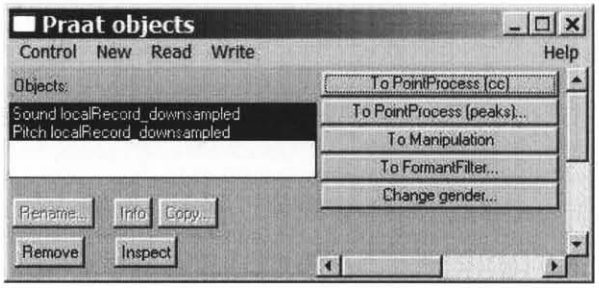

Abbildung 5: Umwandeln von `Sound`- und `Pitch`-Objekt in `PointProcess`

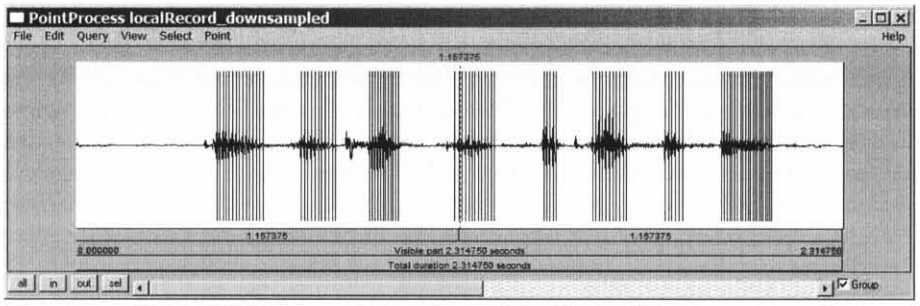

Abbildung 6: Visualisierung eines PointProcess

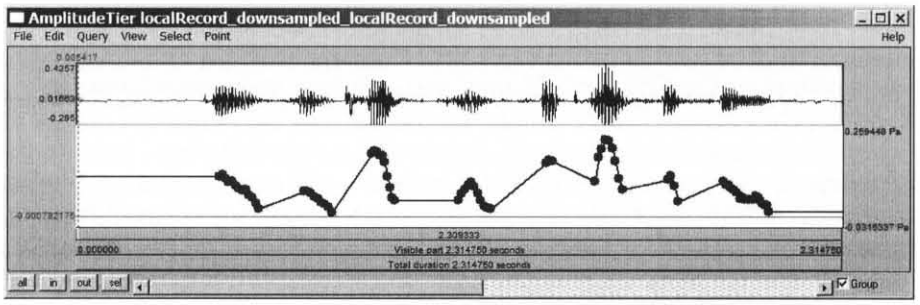

Abbildung 7: Sound und zugehörige Amplitude

Quellcode-Funktion `AmplitudeTier_getShimmer_local()` (für lokalen Shimmer).
Für einige der in diesem Abschnitt genannten Algorithmen stellt PRAAT noch weitere Alternativen zur Auswahl, die je nach Einsatzgebiet möglicherweise besser geeignet sind. Die grundlegende Vorgehensweise ändert sich dadurch aber nicht.

7 Portierung im Allgemeinen

Nachdem die Entscheidung getroffen wurde, eine Bibliothek auf PRAAT basierend zu implementieren, begann die konkrete Portierungsphase. Diese kann grob untergliedert werden in Analyse des Quellcodes, Projektportierung und Codeportierung.
In der Analysephase wird der vorhandene Quellcode eingesehen und untersucht. Es ist wichtig, den Aufbau des Programms zu verstehen, um später eine weitgehend automatisierte Konvertierung zu ermöglichen. Wenn beispielsweise beim Kompilieren Fehler auftreten, so ist in die Fehlermeldung im Allgemeinen wenig Kontextinformation einbezogen, und ohne ein gutes Verständnis der Projektstruktur ist es sehr schwierig, die nötigen Querverweise und damit die Ursachen zu finden und zu beheben. Zudem erhält man während der Analyse ein besseres Gefühl für den Programmierstil des ursprünglichen Autors. Auch das ist sehr hilfreich, denn dadurch können große Codefragmente schneller durchgesehen werden, weil die optische und semantische Auffassung zügiger vonstatten gehen kann. Hauptsächlich wird bei der Codeanalyse aber die Machbarkeit der Portierung geprüft. Es kann durchaus passieren, dass nachträglich einige Gründe gegen eine Portierung sprechen. Auch im Fall von PRAAT war dieser Fall für einige Zeit wahrscheinlich, da sich bei den Tests herausstellte, dass das C-basierte Objektmodell nicht mit dem Compiler von *eMbedded Visual C++* kompatibel war[13]. Wann immer während der Analyse ein potenzieller Kandidat für ein Hindernis bei der Portierung auftaucht, so ist eine genauere Untersuchung und Abschätzung der Kosten für die Lösung erforderlich, da das Projekt andernfalls Gefahr läuft, erst während einer späteren Phase abgebrochen zu werden. Die Kosteneinschätzung kann das gesamte Spektrum von „einfach" über „sehr aufwändig" bis hin zu einer Quasi-Neuimplementierung abdecken. Beispielsweise kann es sein, dass ein bestimmter Plattformaufruf (Betriebssystem-API-Funktion) unter der Zielplattform nicht existiert. Gibt es aber eine ähnliche Funktion, so muss im Wesentlichen nur der Aufruf an allen Vorkommen im Quellcode ersetzt werden. Handelt es sich aber um ein umfangreiches Subsystem des Betriebssystems (z.B. eine Schnittstelle für 3D-Grafik), so muss die Funktionalität „von Hand" nachgebildet werden, was sehr umfangreich ausfallen kann. Grundsätzlich können sich bei einer Portierung während jeder Phase unvorhergesehene Schwierigkeiten ergeben, aber aus ökonomischen Gründen ist die frühzeitige Feststellung generell förderlich.
Nach der Analyse folgt die Portierung auf Projektebene. Das *Projekt* stellt dabei die größte Code-Organisationseinheit dar, bestehend aus Modulen, Paketen, Programmen, Bibliotheken, Codedateien, Ressourcendateien usw. Inwieweit die Struktur des Quellprojekts beibehalten werden kann hängt von verschiedenen Faktoren ab, u.a. davon wie das Originalprojekt gegliedert war (separate Komponenten oder kompakter Verbund), welcher Grad an Komplexität vorliegt (Anzahl der Module) und wie kompatibel die Entwicklungswerkzeuge sind. Besonders wenn die Entwicklungswerkzeuge nicht kompatibel sind, muss hier möglicherweise einige Arbeit investiert werden. Bei großen Projekten findet man häufig komplexe Erstellungskonfigurationen (Build Configurations) vor, z.B. in Form von Skripten für externe Build-Tools, Makros der IDE[14] oder Makefiles. Ein weiteres Beispiel sind die Einstellungen für Compiler und Linker, die in das Zielprojekt übertragen werden müssen. Hier kann es mitunter vorkommen, dass die neuen Werkzeuge einige der Einstellungen gar nicht oder nur in einer nicht kompatiblen Weise unterstützen. Und schließlich sind auch noch Umgebungsvariablen und Verweise auf externe Include-Dateien zu beachten. Nach dieser Phase sollte das zu portierende Programm bereit für die Kompilierung sein (unabhängig davon, ob diese erfolgreich ist).
In der letzten Phase, die meist zugleich auch die längste ist, wird die Portierung des Codes durchgeführt. Wenn das Projekt grundlegende Änderungen erfordert, z.B. Entfernen von nicht

[13] Dieses Problem konnte jedoch nicht zuletzt durch die hilfreiche Mitwirkung des Autors glücklicherweise gelöst werden.
[14] Integrated Development Environment, Integrierte Entwicklungsumgebung

benötigter oder schwer portierbarer Funktionalität aus dem ursprünglichen Code, so werden diese zuerst durchgeführt, da es den Code für den weiteren Ablauf reduziert. Danach werden Probleme mit Sprachkonstrukten gelöst, die der Zielcompiler nicht unterstützt. Bei einer Portierung zwischen deutlich verschiedenen Sprachen (z.B. C++ nach *Java*) wird das Programm hier blockweise übersetzt, was sehr lange dauern kann. In manchen Fällen gibt es auch Tools, welche diesen Prozess teilweise automatisieren können, wenn die Sprachen zwar syntaktisch verschieden, aber von Umfang und Semantik ähnlich sind (z.B. C# nach *Visual Basic .NET*). Ansonsten aber kann man die Portierung üblicherweise mit Hilfe des Compilers vereinfachen, da er die unbekannten Konstrukte meldet. Als letztes müssen Verweise auf externe Funktionen vom Betriebssystem (Plattformaufrufe) und der Laufzeitumgebung (Runtime) berichtigt werden. Auch hier wirkt der Compiler unterstützend mit, da er alle nicht vorhandene Funktionen auflistet. Schwieriger ist es, Verschiedenheiten in der Semantik von Funktionen zu entdecken. Dies kann nur durch gute Kenntnis der Plattformen und ausführliches Testen geschehen.
Eine ausführliche Codeanalyse kann sich bei der Portierung bezahlt machen. Im Optimalfall sind die problematischen Stellen und die jeweiligen Lösungen bereits bekannt und müssen nur noch implementiert werden, was dann auf eine weitgehend mechanische Arbeit hinausläuft. In der Praxis ist es aber schwierig, diese Stellen ohne Hilfe des Compilers alle zu identifizieren. Weitere Informationen zu Konzepten der Portierung sind in [4] zu finden. In dieser Arbeit wird unter anderem auch der Frage nachgegangen, wie sich die Portierbarkeit einer Anwendung beurteilen lässt.

8 Portierung von PRAAT

8.1 Analyse des Quellcodes

Die dieser Arbeit zugrunde liegende Version von PRAAT ist 4.2.13 (im Laufe des Projekts wurde auf 4.2.16 aktualisiert). Das von der Website des Autors erhaltene Archiv mit dem Quellcode enthält bereits eine Ordnerstruktur, die in einzelne Module gegliedert ist. Beim Erstellen der ausführbaren Datei werden alle Module zu einer einzigen Datei kompiliert, welche dann die Anwendung darstellt, die auch separat auf der Website erhältlich ist. Das Archiv enthält die C-Quellcodedateien (getrennt nach `.h` für Deklarationen (*Header*) und `.c` für Implementierungen), ein sog. *Makefile* zum automatischen Erstellen des Programms mit dem unter Unix verbreiteten Hilfsprogramm *make*, plattformspezifische Umgebungsdefinitionen für das Erstellen, sowie einige Beispielskripte (einfache Textdateien mit Erweiterung `.praat`). PRAAT enthält auch eine Hypertext-Dokumentation, auf die vom Programm aus zugegriffen werden kann. Diese ist zwar in der vorliegenden Version noch nicht ganz vollständig, aber die vorhandenen Kapitel sind gut ausgearbeitet und sehr aufschlussreich. Die Dokumentation ist komplett in den Quellcode eingebettet.
Die einzelnen Module, von denen jedes in einem eigenen Verzeichnis liegt, sollen im folgenden kurz beschrieben werden:

- *main*: Enthält im Wesentlichen den Einsprungpunkt für die Anwendung, also die Funktion `main()`. PRAAT wird hier geladen und gestartet. Außerdem werden noch einige globale Präprozessorsymbole deklariert, z.B. wenn die Anwendung für die Konsole kompiliert wird.

- *sys*: Wie der Name schon andeutet ist dieses Modul sehr wichtig für die gesamte Anwendung. Es enthält unter anderem eine Abstraktion für objektorientierte Programmierung in C, viele elementare Datentypen wie Enumerationen, Formeln und komplexe Zahlen, die Grafikroutinen und die GUI-Bibliothek sowie einige Steuerelemente für die Dateneingabe (in PRAAT als *Editoren* bezeichnet), die Dokumentationsanzeige (d.h. einen Hypertext-Betrachter mit Suchfunktion) und eine Betriebssystem-Abstraktionsebene (genannt *melder*). Die objektorientierte Programmierung von PRAAT wird weiter unten in diesem Abschnitt noch aufgegriffen. Die GUI wird in Abschnitt 8.4 näher behandelt.

- *fon*: Dies ist das größte Modul. In ihm befinden sich die meisten Komponenten zur Audiorepräsentation und Sprachanalyse, beispielsweise das `Sound`-Objekt und die

Funktionen zum Einlesen von Sounddateien, die Objekte `Pitch`, `PointProcess` und `Amplitude` und einige weitere Komponenten, die aber für das AGENDER-Projekt nicht von konkreter Bedeutung sind.

- *dwsys* und *dwtools*: Diese beiden Module enthalten viele allgemeine mathematische Funktionen, z.B. für schnelle Fourier-Transformation[15], Tabellen, Matrizen, affine Transformationen, Eigenwerte, Polynome, Diskriminanten usw. Der Name der Module soll vermutlich auf ihren Autor (David Weenink) hinweisen.
- *gsl*: Enthält eine speziell angepasste Version der GNU Scientific Library (GSL). Die GSL besteht aus über 1000 numerischen und weiteren mathematischen Funktionen, z.B. aus dem Bereich Zufallszahlengeneratoren, komplexe Zahlen, Bessel-Algorithmen, Interpolation und numerische Integration.
- *artsynth*: Deckt die erweiterte Sprachsynthese (Articulatory Synthesis) ab, mit der die Bildung von Lauten simuliert werden kann. Dieser Bereich ist derzeit für AGENDER ebenfalls weniger relevant.
- *ffnet*: In diesem Modul ist eine Implementierung von Neuronalen Netzen mit Vorwärtspropagierung (*Feedforward neural networks*) enthalten.
- *lpc*: Die Abkürzung steht für *Linear Predictive Coding*. Mit Algorithmen der Funktionen in diesem Modul können auf Basis früherer Sprachproben Aussagen über folgende Samples gemacht werden.
- *ipa*: Dieses Modul enthält lediglich Schriften, die als C-Dateien codiert wurden, d.h. die Zeichen sind im Code enthalten und werden per Funktionsaufruf abgerufen.

Wie bereits erwähnt wurde ist der gesamte Code auf C basiert, d.h. er verwendet keine der Funktionen der neueren Sprache C++. Diese Entscheidung liegt vermutlich einerseits darin begründet, dass der Beginn der Entwicklung schon sehr lange zurückliegt, und zum anderen wohl auch in der etwas besseren Performance von C. Eine interessante Besonderheit des Codes ist die Einführung eines Programmiermodells für Objektorientierung, das allein auf C-*Strukturen* und *Makros* basiert. C besitzt im Gegensatz zu C++ zwar keine objektorientierten Konstrukte, aber der Autor sieht offenbar einen Vorteil in dieser Sichtweise und hat die fehlenden Konstrukte durch Makros nachgebildet. Dadurch kann er zwar eine ähnliche Syntax erreichen wie dies mit Klassen in C++ möglich ist, allerdings sind auch große Nachteile damit verbunden. Zum einen leidet die Lesbarkeit des Codes stark unter den Makros. Makros werden an einer Stelle im Code definiert, und zwar indem einem Platzhalter eine Zeichenfolge zugewiesen wird, die dann überall im Quellcode für den Platzhalter eingesetzt wird. Dabei werden auch einfache Argumente unterstützt. Makros erlauben zwar eine starke Verkürzung von Code, aber wenn sie übermäßig benutzt werden, fällt es für Außenstehende oft schwer, einen Codeblock zu verstehen. Außerdem handelt es sich bei Makros um Präprozessor-Konstrukte. Sie werden vor der eigentlichen Kompilierung angewendet, und zwar auf rein textueller Ebene. Dadurch ist es möglich, Makros zu definieren, die vor der Einsetzung *syntaktisch* nicht korrekt sind, wie die Klammersetzung in Listing 1 zeigt. Besonders problematisch ist es, wenn der Compiler einen Fehler meldet, der durch ein Makro verursacht wurde bzw. durch das Ersetzen eines Makroplatzhalters aufgetreten ist. Die Fehlermeldung wird sich auf den Code nach der Makroeinsetzung beziehen, die Entwicklungsumgebung zeigt jedoch nur die Quelldaten vor Makroeinsetzung (vgl. [9]).
Da PRAAT Makros so intensiv für sein Objektorientierungs-Modell nutzt, und da diese Makros oft sehr lang sind und in sich auch wieder Makros enthalten, so ist es wohl nachvollziehbar, dass die Portierung und das Debuggen durch diesen Umstand beträchtlich erschwert wurde.
Einige der Makros, mit denen in PRAAT Klassen und Methoden definiert werden können, sind in Listing 2 angegeben. Listing 3 enthält mehrere kleinere Hilfsmakros zur Arbeit mit Objektinstanzen, die in echten objektorientierten Sprachen als fest in die Sprache integrierte Schlüsselwörter zur Verfügung stehen, z.B. in C++ `new` zum Erstellen von Objekten, `delete` zum Löschen und `this` als Zeiger auf die aktuelle Instanz. In Listing 4 schließlich wird ein Beispiel für eine Klassendefinition mit diesen Makros gezeigt.

[15] Fast Fourier Transform, FFT

Listing 1 Beispiel für ein Makro

```
#define macro openFile(fileName

char* fileName = "localrecord.au";
file f = macro, read_only);
```

Listing 2 Makros zum Erstellen von Klassen und Methoden

```
#define class_create(klas,parentKlas) \
typedef struct struct##klas *klas; \
class_create_opaque (klas, parentKlas)

#define class_create_opaque(klas,parentKlas) \
typedef struct struct##klas##_Table *klas##_Table; \
struct struct##klas##_Table { \
void (* _initialize) (void *table); \
char *_className; \
parentKlas##_Table _parent; \
long _size; \
klas##_methods \
}; \
struct struct##klas { \
klas##_Table methods; \
klas##_members \
}; \
extern struct struct##klas##_Table theStruct##klas; \
extern klas##_Table class##klas;


#define class_methods(klas,parentKlas) \
static void _##klas##_initialize (void *table); \
struct struct##klas##_Table theStruct##klas = { \
_##klas##_initialize, #klas, \
& theStruct##parentKlas, \
sizeof (struct struct##klas) }; \
klas##_Table class##klas = & theStruct##klas; \
static void _##klas##_initialize (void *table) { \
klas##_Table us = table; \
if (! class##parentKlas -> destroy) \
class##parentKlas -> _initialize (class##parentKlas); \
class##parentKlas -> _initialize (us);
#define class_method(method)  us -> method = method;
#define class_method_local(klas,method)  us -> method = \
    class##klas##_##method;
#define class_methods_end  }
```

Listing 3 Hilfsmakros für Klassen

```
#define I  Any void_me
#define thou  Any void_thee
#define iam(klas)  klas me = (klas) void_me
#define thouart(klas)  klas thee = (klas) void_thee
#define my  me ->
#define thy  thee ->
#define his  him ->
#define our  my methods ->
#define your  thy methods ->
#define new(klas)  Thing_new (class##klas)
#define forget(thing) _Thing_forget ((Thing *) & (thing))
```

Listing 4 Beispielklasse `data` (Parentklasse `thing`); vereinfacht

```
#define Data_methods Thing_methods \
struct structData_Description *description; \
int (*copy) (Any data_from, Any data_to); \
int (*equal) (Any data1, Any data2); \
int (*writeAscii) (I, FILE *f); \
int (*readAscii) (I, FILE *f); \
int (*writeBinary) (I, FILE *f); \
int (*readBinary) (I, FILE *f); \
/* Messages for scripting. */ \
double (*getXmin) (I); \
double (*getXmax) (I); \
double (*getYmin) (I); \
double (*getYmax) (I); \
double (*getX) (I, long ix); \
double (*getY) (I, long iy); \
double (*getCell) (I); \
double (*getVector) (I, long icol); \
double (*getMatrix) (I, long irow, long icol); \
double (*getFunction0) (I); \
double (*getFunction1) (I, double x); \
double (*getFunction2) (I, double x, double y); \
double (*getRowIndex) (I, const char *rowLabel); \
double (*getColumnIndex) (I, const char *columnLabel);
class_create (Data, Thing)

class_methods (Data, Thing)
class_method (copy)
class_method (equal)
class_method (writeAscii)
class_method (readAscii)
class_method (writeBinary)
class_method (readBinary)
class_methods_end
```

8.2 Gesamtes Programm vs. einzelne Algorithmen

Da PRAAT als Ganzes sehr komplex ist und einige Komponenten enthält, die für die PocketPC-Version nicht benötigt werden (z.B. GUI, Skriptengine, Sprachsynthese und einige andere Algorithmen), bestand auch die Möglichkeit, anstatt des gesamten Programms nur die benötigten Algorithmen und eine minimale Menge an gemeinsam genutztem Code zu portieren. Wäre letzteres effizient zu realisieren gewesen, so hätte es durchaus eine Alternative dargestellt. Dies war allerdings nicht der Fall, und zwar hauptsächlich aus folgenden Gründen:

- Der gemeinsame Code ist sehr umfangreich (mathematische Routinen, GSL, OO[16]-Framework, *melder*)
- Zwischen den Modulen sind häufig Querverweise zu finden, d.h. ein Modul benutzt Header-Deklarationen eines anderen Moduls.
- Die objektorientierte Sichtweise verhindert, dass man einzelne Algorithmen leicht extrahieren kann; stattdessen sind sie in ein Objekt integriert und verwenden wiederum einige weitere Objekte, die dann auch wieder vollständig übernommen werden müssen. Dies gilt natürlich auch für die OO-Abstraktionsebene.
- GUI und Algorithmen sind nicht getrennt. Stattdessen enthält jedes Objekt auch die Funktionen, die für seine Visualisierung notwendig sind. Näheres hierzu in Abschnitt 8.4.
- Da der Code in C geschrieben ist, lässt er sich nur schlecht in ein C++-Projekt integrieren, da es zwischen den beiden Sprachen einige wesentliche Unterschiede gibt, die über den verschiedenen Funktionsumfang hinausgehen.

8.3 Anlegen des Projekts

Die eigentliche Portierung wurde durch das Anlegen der Projektdatei mit *eMbedded Visual C++* eingeleitet. Hierzu wurde ein neues *Windows CE*-Projekt angelegt, wodurch bereits einige Standardeinstellungen vorgegeben waren. In das neue Projekt wurden zuerst alle Quellcodedateien (`.h` und `.c`) von PRAAT Version 4.2.13 importiert. *eVC++* erkennt an der Erweiterung `.c`, dass es sich um C-Code handelt (im Gegensatz zu `.cpp` bzw. `.cxx` für C++-Code), und kompiliert diesen entsprechend. Die Version 4.0 von *eVC++* erstellt CE-Programme standardmäßig für *Windows CE 4.x*. Über eine Compileroption musste dies auf *Windows CE 3.0* abgeändert werden, damit die Bibliothek auch unter dem älteren Betriebssystem läuft (vgl. Abschnitt 2). Danach wurde noch die Compilervariable `CONSOLE_APPLICATION` gesetzt. Diese wird von PRAAT verwendet, um das Verhalten für Konsolenanwendungen zu optimieren, was bedeutet, dass einige Codeteile, die speziell nur für die grafische Anwendung erforderlich sind, dann nicht kompiliert werden müssen. Leider reicht die Variable alleine aber nicht aus, um sich der gesamten Grafikroutinen zu entledigen, so dass hier noch einiger zusätzlicher Aufwand erforderlich war. In der Projektdatei ist außerdem der Typ von Bibliothek festgelegt, der erstellt werden soll, nämlich *statisch* oder *dynamisch*. Das war zu diesem Zeitpunkt aber noch keine vorrangige Entscheidung und wurde erst später ausgemacht (siehe Abschnitt 9).

8.4 Entfernen der GUI-Komponenten

Die grafische Benutzeroberfläche von PRAAT wird durch einen von Paul Boersma entwickelten Emulator für MOTIF[17], einer Grafikerweiterung für das X-Window-System, bereitgestellt. In MOTIF basieren alle UI-Steuerelemente (z.B. ein Eingabefeld, genannt *TextEditor*) auf einem *Widget* und können verschiedene Stile besitzen. Der MOTIF-Emulator kann die GUI dann je nach Plattform für X (Linux), Windows oder OS/X (Macintosh) erzeugen. Es gibt aber auch noch einige weitere Komponenten, welche von betriebssystem-spezifischen Grafikfunktionen abhängen, z.B. die Hypertext-Ausgabe und die Druckfunktionen.

[16] object oriented, objektorientiert

[17] `http://www.opengroup.org/motif/`

Das Grafiksystem von Windows CE (Window Manager und GDI) ist zwar in großen Teilen kompatibel mit Windows für Desktoprechner, aber einige Funktionen wurden auf der mobilen Plattform weggelassen, um Ressourcen zu sparen. Da für die PocketPC-Bibliothek keine der grafischen Komponenten benötigt wird, wurde die Entscheidung getroffen, sie aus dem Code zu entfernen und die Referenzen zu beseitigen, da eine Portierung auf Windows CE wegen der Unterschiede im Grafiksystem in diesem Fall sehr aufwändig gewesen wäre.
Zuerst wurden die Quellcodedateien identifiziert, welche ausschließlich für das Grafiksystem zuständig sind. Die wichtigsten davon sind:

- `motif.c/.h`, `motifEmulator.c`, `Gui.h`, `GuiP.h`, `Ui.h`: MOTIF-Emulation
- `...Editor.c/.h`: Stellen die GUI-Steuerelemente (*Widgets*) dar (44 Dateien)
- `GuiWindow.c`: Implementierung der Fenster als Widgets
- `GuiText.c`: Ausgabe von grafischem Text
- `Graphics.c/.h`, `Graphics_....c`: Zeichenbereich und Zeichenfunktionen
- `Image.c/.h`: Zeichenbereich für Funktionen
- `praatP.h`: PRAAT-Menüumgebung und Objektaktionen
- `Manual.c/.h`, `ManPage.c/.h`, `ManPages.c/.h`: Dokumentation

Eine Eigenheit von PRAAT ist es, dass die meisten Objekte sowohl algorithmischen Code, als auch grafischen Code in ein und derselben Datei enthalten. Beispielsweise hat das Objekt `Sound` Methoden zum Auslesen des digitalen Signals aus einer Datei, zum Konvertieren in ein `Pitch`-Objekt, zum Abspielen über die Soundkarte, als auch zum Zeichnen in ein Fenster. Das hatte zur Folge, dass sehr viele Dateien Verweise auf die Grafikfunktionen enthielten, die zuvor entfernt wurden. In C/C++ werden solche Verweise meist durch Inklusionsanweisungen (`#include "<dateiname>"`) angedeutet. Diese Anweisungen führen beim Erstellen zu einem Fehler im Präprozessor, da dieser die entfernten Dateien nicht findet und somit nicht inkludieren kann. Es ist aber in C/C++ nicht notwendig, die von einer Codedatei benötigten Dateien immer explizit anzugeben; es genügt, wenn selbige Dateien von irgend einer anderen Datei im Projekt inkludiert werden, sofern die betroffenen Dateien in der richtigen Reihenfolge kompiliert werden. Aber auch in diesem Fall wird das Erstellen fehlschlagen, und zwar während des Kompilierens oder spätestens während des Linkens. Bei letzterem Vorgang werden alle nicht aufgelösten Funktionsverweise im Programm gesucht und gemeldet, allerdings kann im Gegensatz zur Fehlersuche durch den Präprozessor nicht herausgefunden werden, in welcher Datei sich die Funktion befand, d.h. welcher benötigte Verweis entfernt wurde.
Aus dem Code von PRAAT wurden folgende Inklusionsanweisungen entfernt, um die entfernten GUI-Komponenten zu reflektieren:

- `#include "Graphics.h"` in 41 Dateien, hauptsächlich Klassendefinitionen (z.B. `Categories.h`)
- `#include "Ui.h"` in `Formula.c`
- `#include "praatP.h"` in `Formula.c` und `Interpreter.c`
- `#include "Gui.h"` in `Interpreter.h` und `melder_audio.c`
- `#include "Image.h"` in `Manipulation.h`

Durch das Entfernen der Verweise wurden aber die impliziten Abhängigkeiten noch nicht beseitigt. Die meisten solcher Abhängigkeiten bestanden in Referenzen auf das `Graphics`-Object. So sieht beispielsweise die Funktionsdefinition zum Zeichnen eines `AmplitudeTier`-Objekts in `AmplitudeTier.c` folgendermaßen aus:

```
void AmplitudeTier_draw (AmplitudeTier me, Graphics g,
     double tmin, double tmax, double ymin, double ymax,
     int garnish)
{
    // ... Hier wird die AplitudeTier "me" in den
    //     Grafikkontext "g" gezeichnet ...
}
```

Diese Memberfunktion konnte nach dem Entfernen des Grafiksystems nicht mehr kompiliert werden, weil die benötigte Definition von `Graphics` nicht mehr verfügbar war. In diesem und in etwa hundert weiteren Fällen war zu überlegen, wie diese restlichen Abhängigkeiten entfernt werden konnten. In dem obigen Beispiel scheint es rein intuitiv möglich zu sein, die gesamte Methode `AmplitudeTier_draw()` (sowie die Deklaration in der Headerdatei) zu löschen, da in der Library eigentlich kein Bedarf zum Zeichnen eines `AplitudeTier`-Objekts (oder irgend eines anderen Objekts) besteht. Allerdings bedeutet dies, dass dann alle Stellen, an denen `AmplitudeTier_draw()` aufgerufen wird, auch wieder nachbearbeitet werden müssen usw. Ähnliche Methoden des Typus `..._draw()` oder `..._paint()` zum Zeichnen eines Objekts machten den Hauptanteil der Fälle aus, die korrigiert werden mussten. Zum Glück konnten sie in den meisten Fällen entfernt werden, ohne weitere Kompilierungsfehler herbeizuführen. Wenn Verweise existierten, dann größtenteils in den bereits entfernten Dateien oder anderen Zeichenmethoden von Objekten.
Statt die Prozeduren komplett zu entfernen wurden sie für dieses Projekt generell nur auskommentiert, um eine strukturelle Ähnlichkeit zum Ursprungsprojekt beizubehalten. Dies kann sich als Vorteil beim Zusammenführen erweisen, z.B. wenn später Änderungen einer neueren Version des Programms integriert werden sollen.
Ein Beispiel einer solchen Quellcodeänderung sieht wie folgt aus:
Auskommentierte Funktionsdeklaration in `AmplitudeTier.h`:

```
/*
void AmplitudeTier_draw (AmplitudeTier me, Graphics g,
     double tmin, double tmax, double ymin, double ymax,
     int garnish);
*/
```

Auskommentierte Funktionsdefinition in `AmplitudeTier.c`:

```
/*
void AmplitudeTier_draw (AmplitudeTier me, Graphics g,
     double tmin, double tmax, double ymin, double ymax,
     int garnish)
{
RealTier_draw (me, g, tmin, tmax, ymin, ymax, garnish,
               "Sound pressure (Pa)");
}
*/
```

Zu erwähnen ist hier noch, dass die Blockkommentar-Syntax `/* ... */` es häufig erforderlich machte, innerhalb größerer auskommentierter Methoden die Kommentarsyntax bereits vorhandener Kommentare in `//` zu ändern, da Kommentare in C/C++ nicht geschachtelt werden können.
Nicht immer war der Fall so einfach, dass eine ganze Methode auskommentiert werden konnte. Oftmals handelte es sich bei den Prozeduren, welche auf die Grafikfunktionen verwiesen, nicht um reine Zeichenfunktionen oder sie wurden noch von anderen Teilen des Projekts benötigt. In diesem Fall war die Analyse dann schwieriger und resultierte meistens darin, dass die grafik-spezifischen Anweisungen bzw. Blöcke auskommentiert wurden.
Die wichtigsten Fälle dieser Art waren:

- In der Datei `Formula.c` in Funktion `Formula_compile()` wurde

Listing 5 Entfernter Block mit *Widget*-Verweis

```
if (my editorClass) {
   praatP. editor = praat_findEditorFromString
                                    (my environmentName);
   if (praatP. editor == NULL)
      return Melder_error ("Editor \"%s\" does not exist.",
                     my environmentName);
} else {
  praatP. editor = NULL;
}
```

```
theInterpreter = interpreter ?
   interpreter : UiInterpreter_get ();
```

eresetzt durch

```
theInterpreter = interpreter;
```

- In der Datei `Interpreter.c` wurden zwei Änderungen vorgenommen:
  1. Die Funktion `Interpreter_createFromEnvironment()` wurde entfernt, da diese auf dem `Widget`-Objekt (*Editor*) basiert, welches nicht mehr existierte.
  2. Ein weiterer Block wurde in Funktion `Interpreter_run()` entfernt, ebenfalls mit Bezug auf GUI-Widgets (siehe Listing 5).
- Weitere Auskommentierungen in `Minimizers.c` in den Funktionen `classMinimizer_after()` und `Minimizer_minimize()`
- In `OTGrammar.c` in den Funktionen `OTGrammar_PairDistribution_learn()` und `OTGrammar_Distributions_learnFromPartialOutputs()`
- In `Sound_to_Harmonicity_GNE.c` in Funktion `Sound_to_Harmonicity_GNE()`.

Abschließend lässt sich sagen, dass das Entfernen der Grafikkomponenten aufgrund der Architektur von PRAAT relativ umständlich und zeitaufwändig war, verglichen mit einer Portierung aber auf jeden Fall die bessere Alternative darstellte.

8.5 Entfernung der Tonausgabe-Komponenten

Mit den Komponenten zur Wiedergabe von Sounds verhielt es sich ganz ähnlich wie mit der Grafik. Auch die Audiowiedergabefunktionen waren für den Einsatz als Bibliothek für AGENDER nicht relevant. Sie sind eher für experimentelle Umgebungen interessant als für Endanwendersysteme. Das alleine wäre natürlich noch kein Grund gewesen, die Komponenten zu entfernen, denn ein größerer Funktionsumfang wäre an sich kein Nachteil gewesen. Allerdings waren – wie bei dem Grafiksystem – auch hier die Kosten der Portierung zu hoch, weil Windows für Desktop-Rechner und Windows CE nicht ausreichend kompatibel sind, um mit nur geringen Änderungen auszukommen.
Das Entfernen des Audiosystems wurde wie folgt durchgeführt:

- In der Datei `ExperimentMFC.c` wurde die Funktion `ExperimentMFC_playStimulus()` auskommentiert, die in Listing 6 zu sehen ist.
- In der Datei `Longsound.c`:
  1. Es wurde ein Verweis auf `Melder_stopPlaying()` auskommentiert, da ja in der Bibliotheksanwendung keine Sounds abgespielt werden, die anzuhalten wären.

Listing 6 Auszug aus der Funktion `ExperimentMFC_playStimulus()`

```
void ExperimentMFC_playStimulus (ExperimentMFC me,
     long istim)
{
...
Sound_playPart (my playBuffer, 0.0,
   (initialSilenceSamples + carrierBeforeSamples
      + stimulusSamples + carrierAfterSamples)
   * my samplePeriod,0, NULL);
}
```

2. Außerdem wurde die Funktion `LongSound_playPart()` geleert (d.h. die Funktion ist noch vorhanden, aber sie enthält keinen Code mehr, so dass der Aufruf ohne Wirkung bleibt).

- In `Manipulation.c` wurden die Funktionen `Manipulation_playPart()` und `Manipulation_play()` ebenfalls geleert.
- In `melder_audio.c`:
  1. Alle Funktionen zum Abspielen von Audio wurden entfernt, z.B. `MelderPlay()`.
  2. Die Struktur `static struct MelderPlay` wurde entfernt.
- In `Pitch_to_Sound.c` wurden die beiden Funktionen `Pitch_play()` und `Pitch_hum()` geleert.

Außer den genannten Beispielen gab es weitere Auskommentierungen von Funktionen, die nicht mehr aufgerufen wurden, analog zur Grafik, z.B. in `PitchTier_to_Sound.c`, `PointProcess_and_Sound.c` und `Sound_audio.c`. Wie auch vorher bei der Grafik wurden im Falle des Entfernens von Prozeduren und Strukturen diese nur auskommentiert, nicht gelöscht.

8.6 Weitere Korrekturen

Mit dem Ausbau des Grafiksystems und des Soundsystems waren bereits zwei große Hürden überwunden. In diesem Abschnitt werden alle weiteren Schritte der Portierung beschrieben, die noch durchgeführt wurde, bevor das Projekt schließlich ohne Fehler kompiliert werden konnte.

- Die Datei `main_praat.c` mit der `Main()`-Funktion, die normalerweise das Hauptfenster und das PRAAT-Logo anzeigt, wurde entfernt. Eine Bibliothek besitzt im Gegensatz zu einer Anwendung keinen Einsprungpunkt. Dynamische Windows-Bibliotheken verfügen zwar über die Möglichkeit einer Initialisierungsfunktion, auf diese kann aber meist verzichtet werden, insbesondere im Fall von PRAAT.
- Einige weitere Kerndateien, die nur im Anwendungsmodus benötigt werden, wurden entfernt. Dazu zählen `praat.c`, `praat.h` und `praat_....c`. In diesem Zusammenhang wurde überdies aus `SVD.c` und `TextGrid_extensions.c` der Verweis auf `praat.h` entfernt.
- Die Texte für die Dokumentation (siehe Abschnitt 8.1) wurden aus allen Modulen entfernt (Dateien `manual_....c`).
- Die meisten Dateien des Synthesemoduls (*artsynth*) wurden für diese Arbeit zur Reduzierung der Komplexität entfernt, da dieses Modul nicht benötigt wurde.

- In der Datei `Manipulation.c` wurden die Funktionen `synthesize_pulses_hum()`, `synthesize_pitch_hum()` und `synthesize_pulses_pitch_hum()` so abgeändert, dass sie jetzt *„nicht unterstützt"* melden, da die benötigten Synthesefunktionen entfernt wurden.

  Beispiel:

```
static Sound synthesize_pulses_hum(Manipulation me) {
   return Melder_errorp ("NOT IMPLEMENTED");
   //return PointProcess_to_Sound_hum (my pulses);
}
```

- Die PRAAT-eigene OO-Abstraktionsebene verwendet ein Makro mit dem Namen `new` zum Erstellen von Objekten (vgl. Listing 3). Da `new` in C++ aber ein reserviertes Schlüsselwort ist (es steht für den Standard-Heapspeicherallokator), musste das Makro umbenannt werden. Als neuer Name wurde `praat_new` gewählt:

```
#define praat_new(klas)  Thing_new (class##klas)
```

  Beispiel:

```
Activation me = new (Activation);
```

  wurde geändert in

```
Activation me = praat_new (Activation);
```

  Diese Substitution wurde an insgesamt 204 Stellen durchgeführt.

- Die C-Funktion `time()` zur Bestimmung der Sekunden wird von der eVC++-Laufzeitumgebung auf dem PocketPC nicht unterstützt. Verwendungen dieser Funktion wurde in einen vergleichbaren API-Aufruf von Windows CE umgeändert, nämlich `GetTickCount()`, welches allerdings Millisekunden zurückliefert.

  Beispiel:

  In `Formula.c`, Funktion `Formula_run()`:

```
time_t today = time (NULL);
```

  wurde ersetzt durch

```
time_t today = (int)(GetTickCount()/1000);
```

  Analog wurde dies in den Dateien `Sequence.c`, `Thing.c` und `Sound_files.c` durchgeführt.

- Die C-Funktion `ctime()` zur Umwandlung einer Zeitangabe in eine Zeichenfolge wird ebenfalls auf dem PocketPC nicht unterstützt. Diese Funktion konnte an allen Vorkommen deaktiviert werden, da sie nur für die Benutzerausgabe benötigt wird.

- Eine weitere auf dem PocketPC nicht vorhandene C-Funktion ist `clock()`. Dieses gibt die von einem Prozess genutzte CPU-Zeit seit dessen Start an, z.B. zum Initialisieren eines Zufallszahlengenerators. In allen Fällen konnte `clock()` durch `GetTickCount()` (s.o.) ersetzt werden, auch wenn dies im Allgemeinen nicht kompatibel ist.

  Beispiel:

  In `melder.c`, Funktionen `Melder_stopwatch()` und `Melder_progress()` wurde

```
clock_t now = clock ();
```

ersetzt durch

```
clock_t now = GetTickCount();
```

und entsprechend auch in `Sound_audio.c`.

- Einige C-Funktionen des Dateisystems sind auf dem PocketPC nicht verfügbar. Dazu zählen die von PRAAT verwendeten Funktionen `rewind()` (Rücksetzen eines Dateizeigers auf den Dateianfang), `getcwd()` (Bestimmen des Arbeitsverzeichnisses) `chdir()` (Ändern des Arbeitsverzeichnisses) und `remove()` (Löschen einer Datei).

  Im Falle der Arbeitsverzeichnis-Funktionen war eine alternative Implementierung nicht nötig, da die Bibliothek sowieso ohne ein Arbeitsverzeichnis auskommt. Statt `rewind()` konnte ein Aufruf von `fseek()` dienen, welcher die Dateiposition verschieben kann, also auch auf 0. Und für `remove()` konnte wieder eine Funktion der Windows CE-API eingesetzt werden. Realisiert wurde dies in allen vier Fällen durch Makros, die an einer zentralen Stelle (in einer neuen Headerdatei) definiert und dann über eine Inklusionsanweisung eingebunden wurden:

  ```
  #define rewind(f) (void) fseek(f, 0L, SEEK_SET)
  #define chdir(x) (void)0
  #define getcwd(x,y) (void)0
  #define remove(x) DeleteFile(x)
  #define GetEnvironmentVariable(a,b,c) (0)
  #define GetWindowsDirectory(buf,size) buf="\\Windows"
  ```

- In der Datei `Formula.c` wurden einige Formeln in der Funktion `Formula_run()` deaktiviert, da externe Symbole nicht mehr verfügbar waren (entfernte Dateien, s.o.), und zwar `praat_selection()`, `praat_getNameOfSelected()` und `praat_getIdOfSelected()`.

- In `gsl_err__error.c`, Funktion `gsl_error` musste der Aufruf

  ```
  abort ();
  ```

  abgeändert werden in

  ```
  exit(gsl_errno);
  ```

  da `abort()` von der eVC++-Runtime nicht unterstützt wird. Analog wurde in `Melder.c`, Funktion `Melder_fatal()`

  ```
  abort ();
  ```

  ersetzt durch

  ```
  exit(1);
  ```

  Nach dem Aufruf von `exit()` wird das Programm geschlossen und der Parameter als Rückgabewert an den Aufrufer zurückgeliefert. Werte ungleich 0 werden dabei als Fehlercodes betrachtet.

- In `gsl_errno.h`, `melder_files.c`, `melder_sysenv.c`, `Sound_audio.c` und `LongSound_extensions.c` wurde jeweils der Verweis auf die betriebssystem-spezifische Datei `errno.h` entfernt. Diese Datei ist im *PocketPC SDK* nicht enthalten; die Deklarationen der Headerdatei sind stattdessen in anderen Dateien untergebracht.

- In der Datei `Sound_files.c` wurde ein Verweis auf `stdio.h` hinzugefügt:

```
#include <stdio.h>
```

Diese Datei deklariert einige grundlegende Ein- und Ausgabefunktionen. Diese – wie auch einige der folgenden hinzugefügten Headerdateien – waren zuvor über Inklusionsanweisungen in anderen Dateien referenziert, die aber im Zuge der Portierung entfernt worden waren.

- In einigen Dateien der GSL (`gsl_poly__balance.c`, `gsl_poly__companion.c` sowie `gsl_poly__qr.c`) wurde ein Verweis auf `stddef.h` hinzugefügt:

```
#include <stddef.h>
```

- In die Headerdatei `melder.h` wurde ein Verweis auf `assert.h` eingetragen:

```
#include <assert.h>
```

- In `gsl_poly__qr.c` wurden mehrere Verweise auf andere GSL-Dateien hinzugefügt, die ansonsten vom Compiler nicht gefunden worden wären. Im ursprünglichen PRAAT-Paket konnte das Programm auch ohne diese Verweise auskommen, da der verwendete Compiler (*gcc*) mit Hilfe des Makefiles so gesteuert wurde, dass er die Dateien bereits verarbeitet hatte.

```
#include "gsl_poly.h"
#include "gsl_math.h"
#include "gsl_errno.h"
```

- Entsprechend wurden auch in `gsl_sf__cheb_eval.c` und `gsl_sf__cheb_eval_mode.c` Verweise hinzugefügt, die vom Compiler benötigt wurden:

```
#include "gsl_sf_result.h"
#include "gsl_sf__chebyshev.h"
#include "gsl_math.h"
#include "gsl_errno.h"
#include "gsl_mode.h"
```

- Bei den folgenden beiden Makros, die Teil der GSL sind, trat ein Problem auf:

```
#define MAT(m,i,j,n) ((m)[(i)*(n) + (j)])
#define FMAT(m,i,j,n) ((m)[((i)-1)*(n) + ((j)-1)])
```

Eine Anwendung des Makros, wie z.B. in

```
MAT (m, 0, nc - 1, nc) *= g;
```

funktionierte so nicht und ergab einen Kompilierungsfehler (*error C2106: '*=' : left operand must be l-value*). Daher wurden diese beiden Makros an allen Stellen manuell expandiert, so dass die obige Zeile danach beispielsweise folgendermaßen aussah:

```
((m)[(0)*(nc) + (nc - 1)]) *= g;
```

Zur Anwendung kam dies in den Dateien `gsl_poly__balance.c`, `gsl_poly__companion.c`, `gsl_poly__qr.c` (jeweils für Makros `MAT` und `FMAT`).

- In `gsl_sf_airy.c`, `gsl_sf_bessel_zero.c`, `gsl_sf_cheb_eval.c`, `gsl_sf_erfc.c`, `gsl_sf_gamma.c`, `gsl_sf_trig.c`, `gsl_sf_zeta.c` und `gsl_sf_bessel.c` traten Probleme mit dem C-Schlüsselwort `inline` auf, welches zur Performanceoptimierung genutzt werden kann. Dieses Schlüsselwort wird anscheinend von *eVC++* nicht erkannt, denn der Compiler meldete an Stellen mit diesem Schlüsselwort Syntaxfehler. Daher wurden Funktionsdefinitionen wie die folgende

  ```
  inline static int airy_aie(const double x,
         gsl_mode_t mode, gsl_sf_result * result) {...}
  ```

  umgeändert in

  ```
  static int airy_aie(const double x,
         gsl_mode_t mode, gsl_sf_result * result) {...}
  ```

  d.h. das Schlüsselwort `inline` wurde entfernt. Dies kann gefahrlos erfolgen, denn es hat keine Auswirkungen auf die Funktionsweise des Codes.

- In `gsl_sf_bessel_j.c`, Funktion `gsl_sf_bessel_j2_e()` wurde der Verweis auf die Funktion `gsl_sf_bessel_Jnu_asymp_Olver_e()` entfernt, da diese im gesamten Projekt nicht existent war. Die Funktion wurde ersetzt durch den kompatiblen Aufruf von `gsl_sf_bessel_Jnu_asympx_e()`. Analog geschah dies für die Dateien `gsl_sf_bessel_Jn.c` und `gsl_sf_bessel_Jnu.c`. Außerdem wurde die Deklaration von `gsl_sf_bessel_Jnu_asymp_Olver_e()` in `gsl_sf_bessel_olver.h` entfernt.

- In `gsl_sf_bessel_y.c`, Funktion `gsl_sf_bessel_yl_e()`: Der Verweis auf die Funktion `gsl_sf_bessel_Ynu_asymp_Olver_e()` wurde entfernt, da nicht ebenfalls existent; sie wurde ersetzt durch den kompatiblen Aufruf von `gsl_sf_bessel_Ynu_asympx_e()` und analog in `gsl_sf_bessel_Yn.c` und `gsl_sf_bessel_Ynu.c`. Auch hier wurde die Deklaration in `gsl_sf_bessel_olver.h` entfernt.

- In `gsl_sf_bessel_zero.c` in der Funktion `gsl_sf_bessel_zero_Jnu_e()` wurde ein *„else"*-Fall auskommentiert, da die dort auftauchende Funktion `gsl_sf_bessel_Olver_zofmzeta()` nicht vorhanden war.

- In `gsl_sf_chebyshev.h` musste ein `#if ... #endif`-Block eingefügt werden, um eine Mehrfachinklusion der Headerdatei zu vermeiden:

  ```
  #ifndef __GSL_SF_CHEBYSHEV_H__
  #define __GSL_SF_CHEBYSHEV_H__
  ...
  #endif /* __GSL_SF_CHEBYSHEV_H__ */
  ```

- In `Interpreter.c`, Funktion `Melder_includeIncludeFiles()` wurde eine Assertion entfernt, da `assert()` nicht von der eVC++-Laufzeit unterstützt wird. Im Grunde genommen ist es sogar üblich, solche Anweisungen in den veröffentlichten Versionen von Programmen zu deaktivieren.

- In der gleichen Datei wurde

  ```
  newText = allocationMethod == 1 ?
     Melder_malloc (newLength + 1)
     : XtCalloc (1, newLength + 1);
  ```

abgeändert in

```
newText = allocationMethod == 1 ?
   Melder_malloc (newLength + 1)
    : (void)0;
```

da `XtCalloc()` kein Befehl von Windows CE ist, und analog

```
if (allocationMethod == 1) Melder_free (*text);
else XtFree (*text);
```

geändert in

```
if (allocationMethod == 1) Melder_free (*text);
```

wegen der Anweisung `XtFree()`.

- In `Interpreter.c` und `Interpreter.h` mussten eine Dummy-Funktion und die dazugehörigen Deklaration erstellt, da die eigentliche Funktion in einer bereits entfernten Datei war:

  ```
  int praat_executeCommand()
  ```

  Dies war unproblematisch, da die Funktion dem Ausführen von Skripten dient und dies nicht zu den Anforderungen an die Bibliothek gehört.

- In `melder.h` und `melder_audio.c`: Ein Variablenname `explicit` wurde entfernt, da es sich dabei um ein in C++ reserviertes Schlüsselwort handelt.

  ```
  int Melder_stopPlaying (int explicit);
  ```

  wurde geändert in

  ```
  int Melder_stopPlaying (int);
  ```

- Der Aufruf von `GetWindowsDirectory()` in der Funktion `Melder_getPrefDir()` in `melder_files.c` wurde entfernt und durch den hartcodierten Verzeichnisnamen des Windows-Ordners auf dem PocketPC (`\Windows`) ersetzt. Dieser Ordnername kann im Gegensatz zum Desktop-Betriebssystem vom Benutzer nicht geändert werden. Analog wurde die Funktion `sendpraat()` in `sendpraat.c` angepasst.

- Der Abruf von Umgebungsvariablen mit getenv() wird von eVC++ nicht unterstützt. Aus diesem Grund wurde der entsprechende Aufruf in `melder_sysenv.c` entfernt.

- Einige Include-Anweisungen mussten für den Compiler von `NUM.c` nach `NUM.h` und von `NUM2.c` nach `NUM2.h` verschoben werden, wobei es sich eigentlich auch um den üblichen Programmierstil für C/C++ handelt.

- In `NUM2.h` wurde die Anweisung

  ```
  #define isnan gsl_isnan
  ```

  aus der Datei `gsl_config.h` hinzugefügt.

- In `Table_extensions.c` und `TableOfReal_extensions.c`: Vordefinierte Datentabellen (z.B. *Peterson-Barney*) wurden entfernt, da sie für die Bibliothek nicht benötigt werden, und da dies die Kompilierung deutlich schneller macht.

Ein weiteres Problem trat bei einigen `struct`-Definitionen (benutzerdefinierte Datentypen) in PRAAT 4.2.13 auf. Diese waren zwar mit dem Compiler *gcc* kompatibel, jedoch aufgrund der Namensgebungen nicht mit *eVC++*. Dank der Mitwirkung des Autors konnte diese Hürde durch einen Umstieg auf Version 4.2.16 behoben werden, in der die `struct`-Definitionen universell kompatibler gemacht wurden.

9 Bereitstellung als Bibliothek

Auch wenn der Code von PRAAT nach Ausführung der im vorigen Abschnitt beschriebenen Schritte schon vollständig auf den PocketPC portiert worden war, so fehlte doch noch die Schnittstelle zur Anwendung, die PRAAT in Form einer Bibliothek aufrufen sollte. Der Code einer Bibliothek ist für das aufrufende Programm unsichtbar, bis auf die Deklarationen einiger *öffentlicher Funktionen*, welche die Schnittstelle ausmachen. Die Funktionen können dann von anderen Anwendungen aufgerufen werden, wenn die Schnittstelle bekannt und die Bibliotheksdatei verfügbar ist. Statt Funktionen kann die Schnittstelle auch andere Sprachelemente (z.B. Konstanten und Klassen) enthalten, aber in einigen Fällen ist es sinnvoll, zugunsten der Kompatibilität mit anderen Sprachen darauf zu verzichten.
Unter Windows werden zwei Arten von C/C++-Bibliotheken unterschieden: *Statische Bibliotheken* und *DLLs* (*Dynamic Link Libraries*). Bei jedem Projekt, in dem eine Bibliothek erstellt wird, müssen die beiden Möglichkeiten verglichen und die jeweiligen Argumente abgewogen werden.
Eine statische Bibliothek ist eine Datei mit der Endung `.lib`, welche nur aus Objektmodulen besteht, d.h. vorkompiliertem Code, die ohne Linken erstellt wurden und die auch nicht direkt aufgerufen werden können. Während des Erstellens einer Anwendung wird der Code der Bibliothek dann fest in diese eingebunden und somit Teil der ausführbaren Datei. Verweise auf die Funktionen in der Bibliothek werden vollständig zur Kompilierzeit aufgelöst. Vorteile:

- Etwas schneller während der Ausführung (kein Linken zur Laufzeit)
- Kompakt, keine Abhängigkeit von einer zusätzlichen Bibliotheksdatei
- Weniger Speicherplatzbedarf auf dem Datenträger (sofern nur eine Anwendung installiert ist, die diese Bibliothek benötigt)
- Keine Konflikte zwischen Anwendungen, da jede ihre eigene Bibliothek benutzt

Eine DLL dagegen besitzt weitgehend das gleiche Format wie eine ausführbare Datei[18] und wird nicht in die Programmdatei der Anwendung kopiert, sondern bleibt als eigene Datei bestehen. Eine Anwendung muss dafür sorgen, dass die benötigten DLL-Dateien immer verfügbar sind (z.B. im Anwendungsverzeichnis). Sie werden beim Start der Anwendung geladen, sobald sie benötigt werden. Zu diesem Zeitpunkt erfolgt auch das Linken, was normalerweise durch das Betriebssystem durchgeführt wird und etwas zusätzliche Zeit beansprucht. DLLs können auch im Systemverzeichnis gespeichert werden, so dass mehrere Anwendungen zur gleichen Zeit auf dieselbe DLL zugreifen können. Dadurch erhöht sich allerdings auch die Gefahr von Konflikten, z.B. beim Zugriff auf globale Variablen. Vorteile:

- Reduzierung des Arbeitsspeicherverbrauchs, da die DLL erst bei Bedarf geladen wird
- Weniger Bedarf an Speicher im RAM und auf der Festplatte, wenn mehrere Anwendungen die gleiche Bibliothek benutzen (Shared Library)
- Anwendung kann zur Laufzeit entscheiden, welche Bibliothek geladen werden soll
- DLL kann Einsprungpunkt für Initialisierungscode enthalten
- DLL-Datei kann binäre Ressourcen wie Bitmaps enthalten

Im Fall von PRAAT wurde die Entscheidung zugunsten der statischen Bibliothek getroffen. Es kann wohl davon ausgegangen werden, dass der praktische Einsatz der Bibliothek – wenn auch sehr interessant – sich vorerst auf M3I und evtl. einige wenige weitere Anwendungen beschränken wird, die aber sicher nicht auf demselben Gerät und zur gleichen Zeit eingesetzt werden, so dass der entscheidende Vorteil einer DLL nicht gegeben ist. Auf der anderen Seite sprechen aber der Geschwindigkeitsgewinn und die zusätzliche Kompaktheit in Hinblick auf den PocketPC für die statische Bibliothek.
Bei einer DLL müssen die Funktionen angegeben werden, welche die öffentliche Schnittstelle ausmachen, so dass eine Exporttabelle generiert werden kann. Für eine statische Bibliothek

[18] auch als PE (Portable Executable) bezeichnet

Gerätebezeichnung	Prozessor	Taktfrequenz	Plattform
HP Jornada 568	ARM SA1110	206 MHz	Windows CE 3.0
HP Ipaq H6300	TI OMAP 1510	200 MHz	Windows CE 4.2
HP Ipaq H5450	Intel PXA250	400 MHz	Windows CE 4.2
HP Ipaq HX4700	Intel PXA270	624 MHz	Windows CE 4.21

Tabelle 1: Testgeräte für die Benchmarks

ist das nicht notwendig, da hier immer alle globalen Konstrukte exportiert werden. In PRAAT sind – wie bei den meisten C-Programmen – alle Funktionen global. Eine kleine Anpassung ist aber dennoch notwendig: Da es sich bei der M3I-Client-Anwendung, welche die Bibliothek benutzt, um ein C++-Programm handelt, sind die Funktionsaufrufe nicht unmittelbar kompatibel. Das liegt daran, dass der C++-Compiler alle Funktionsnamen im ausgegebenen Objektcode „dekoriert" (mit Zusatzinformationen über die Datentypen annotiert) und der C-Compiler nicht. Dadurch können die Aufrufe beim Linken dann nicht mit den Funktionen verknüpft werden. Um dieses Problem zu lösen, muss dem C++-Compiler mitgeteilt werden, dass er für die betreffenden Methoden Namen in C-Syntax verwenden soll. Dies wird über die Option `extern "C"` erreicht, die vor jede öffentliche Funktionsdeklaration eingefügt werden muss. Damit aber die Option nicht auch für den Export beim Erstellen der Bibliothek angewendet wird, wurde für PRAAT das folgende Makro geschrieben:

```
#ifdef __cplusplus
#define XPORT  extern "C"
#else
#define XPORT
#endif
```

Eine öffentliche Funktion wird dann mit dem neuen Makro `XPORT` deklariert, z.B.:

```
XPORT Sound Sound_readFromSoundFile (MelderFile fs);
XPORT Pitch Sound_to_Pitch (Sound me, double timeStep,
            double minimumPitch, double maximumPitch);
```

Das Makro bewirkt, dass beim Kompilieren der Bibliothek mit dem C-Compiler das Wort `XPORT` durch Leerzeichen ersetzt, also ignoriert wird. Unter dem C++-Compiler, mit dem die Anwendung erstellt wird, ist aber die Konstante `__cplusplus` definiert, und damit wirkt das Makro hier anders: Es setzt für `XPORT` jeweils die benötigte Option `extern "C"` ein.
Die PRAAT-Bibliothek befindet sich in der Datei `PraatCE.lib`. Zum Erstellen einer Anwendung, welche die Funktionen zur Merkmalsextraktion nutzt, muss dieser der Pfad der Datei angegeben werden. Außerdem werden die Funktionsdeklarationen benötigt, die im Quellcode zu finden sind.

10 Benchmarks und Optimierungen

Neben der physikalischen Größe des Geräts ist der wichtigste Unterschied zwischen Desktop / Notebook und dem PocketPC die geringere Leistung, insbesondere was die CPU betrifft. Daher war es sehr interessant, mit der neuen Bibliothek einige Leistungsmessungen auf verschiedenen mobilen Geräten (siehe Tabelle 1) durchzuführen. Dazu wurde eine Testanwendung entwickelt, welche aus einer Audiodatei mit einer Sprachprobe eine Reihe von Merkmalen extrahiert, und zwar diejenigen, die auch für AGENDER benötigt werden (vgl. Abschnitt 4).
Die genauen Ergebnisse der ersten Messung sind Tabelle 2 zu entnehmen. Wie zu erkennen ist benötigte die Merkmalsextraktion trotz der geringen Dauer der Sprachprobe auf allen Geräten eine beträchtliche Zeit, auch auf den neueren Modellen. Da sich PRAAT jedoch mitunter durch die hohe Präzision seiner Ergebnisse auszeichnet und bis jetzt noch keinerlei Optimierungen in Hinsicht auf die Performance vorgenommen wurden, war ein solches Ergebnis auch nicht weiter verwunderlich.

Gerät	Benötigte Zeit
Jornada 568	82s
Ipaq H6300	106s
Ipaq H5450	59s
Ipaq HX4700	30s

Tabelle 2: Erste Messung mit Sounddatei 2,3s, 8000 Hz, 16 bit, mono

Funktion	Zeitanteil
Öffnen	0,31%
Pitch-Objekt erstellen	98,33%
PointProcess erstellen	0,86%
AmplitudeTier erstellen	0,32%
Werte berechnen	0,18%

Tabelle 3: Zeitanteil der Funktionen an der Gesamtzeit

Um diese Zeiten zu verbessern wurde nachfolgend eine detailliertere Timing-Analyse des Codes durchgeführt. Ziel dieser Untersuchung war es, die verschiedenen Schritte im Prozess der Merkmalsextraktion in Hinblick auf ihren Anteil an der Gesamtzeit zu untersuchen, um dann den Bereich, in dem eine Optimierung durchgeführt werden sollte, weiter einzuschränken. Die Messung förderte das in Tabelle 3 dargestellte Resultat zu Tage.
Wie sich unschwer feststellen lässt hat ein einzelner Prozess, nämlich das Berechnen der Pitch-Kontur aus dem Audiosignal, einen überproportional hohen Anteil an der Gesamtzeit – es macht fast die komplette Rechenzeit aus. Ausgehend von diesen Zahlen wurden die weiteren Optimierungsbemühungen im Folgenden auf diesen Teilbereich beschränkt.
Die fragliche PRAAT-Funktion, welche für den enormen Zeitkonsum verantwortlich ist, ist `Sound_to_Pitch()`, welche intern gleich `Sound_to_Pitch_ac()` aufruft. Hierbei wird eine Frequenzanalyse mit Hilfe der Autokorrelationsmethode aus [1] wie folgt durchgeführt: Für eine große Anzahl an Punkten innerhalb des Sprachsignals (standardmäßig im Abstand von je 0,01 Sekunden) wird ein kleiner Bereich einer festen Größe betrachtet, der um diesen Punkt zentriert ist. Die Größe des Bereichs hängt von der minimalen zu erkennenden Frequenz ab. Auf das Signal in diesem Bereich wird dann ein Hanning-Window angewendet. Dies ist eine Funktion, die den gleichen Definitionsbereich wie das Teilsignal besitzt und zu den Rändern hin gegen Null geht. Bei Multiplikation mit dem Audiosignal wird dieses an den Rändern abgeschwächt und im Zentrum hervorgehoben. Vom Ergebnis wird dann die normalisierte Autokorrelation berechnet, und diese wiederum durch die Autokorrelation der Hanning-Window-Funktion dividiert. Um die Pitch-Frequenz zu finden, müssen nun die lokalen Maxima gesucht werden, wobei das größte Maximum der beste Kandidat für die Frequenz am betrachteten Punkt ist. Da das Signal aber nicht als Funktion vorliegt, sondern vielmehr in Form von einzelnen Samples, können diese Maxima nur durch Interpolation der vorhandenen Werte bestimmt werden. Die Form der Kurve entspricht der einer $sin(x)/x$-Funktion, daher wird diese zur Berechnung benutzt. Die Tiefe der Interpolation bestimmt dabei die Genauigkeit des Ergebnisses. PRAAT verwendet zwei Durchläufe: Zuerst werden mit einer einfachen Interpolation die möglichen Kandidaten für das Maximum bestimmt. Von diesen Kandidaten werden dann nur die n größten Stellen beibehalten[19] und in einem zweiten Durchlauf die Maxima und Frequenzen mit einem genaueren Algorithmus und höherer Interpolationstiefe erneut berechnet. Dieser zweite Durchlauf stellt offenbar eine geeignete Möglichkeit dar, die Geschwindigkeit zu Lasten der Präzision zu erhöhen. Ein kompletter Verzicht auf den zweiten Durchlauf erbrachte eine Geschwindigkeitssteigerung um ungefähr Faktor neun, was weitere Messungen belegen (Diagramm 1 und 2). Der Verlust an Genauigkeit beträgt bei Pitch im Durchschnitt etwa 0,00125 Hz, bei Jitter und Shimmer weniger als 0,00001 Prozentpunkte.
Bei den neuen Messungen wurden auch verschiedene Audioformate gegenübergestellt, um deren Auswirkung auf die Verarbeitungszeit festzustellen. Es zeigt sich, dass die Samplingra-

[19] Die Konstante n kann nach Bedarf gewählt werden, z.B. 20. Wenn mehrere Maxima sehr ähnliche Werte aufweisen, werden noch weitere Filterverfahren angewendet, siehe [1].

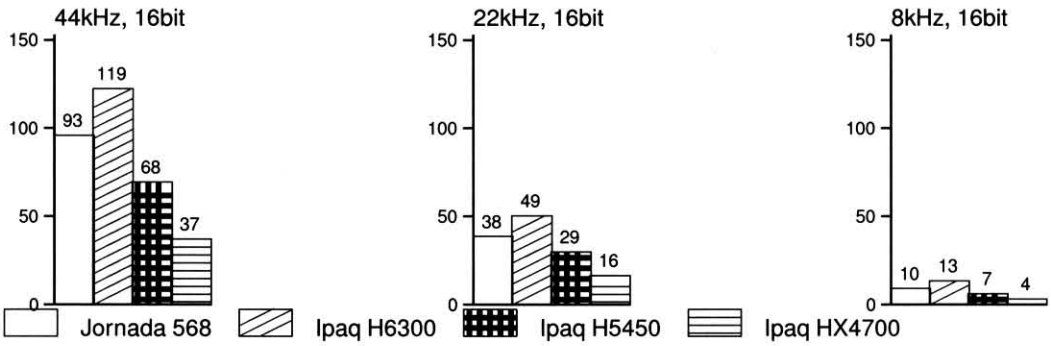

Diagramm 1: Benchmarks nach Optimierung, Test mit Sounddatei 2,3s mono

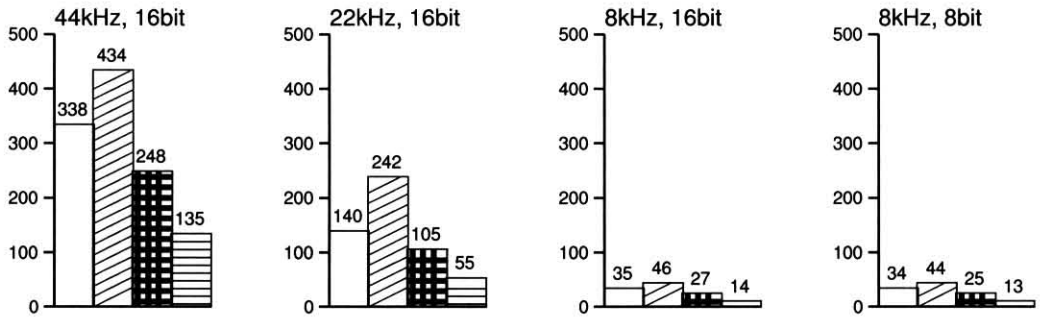

Diagramm 2: Benchmarks nach Optimierung, Test mit Sounddatei 8,4s mono

te einen sehr großen Einfluss auf diese Zeit hat, und dass daher auf dem PocketPC vorläufig nur mit Raten unter 22 kHz gearbeitet werden sollte. Die Bitrate dagegen wirkt sich nur in geringem Maße auf die Zeitdauer aus. Auch ist es sinnvoll, aus längeren Sprachaufnahmen kürzere Teilsegmente zu verwenden, da die Zeiten sonst ebenfalls sehr schnell ansteigen (überproportional zur Länge der Aufnahme) und unpraktikabel werden.
Zusätzliche Optimierungen wurden vorerst im Rahmen dieser Arbeit nicht durchgeführt, da sich keine weiteren Anhaltspunkte für effiziente Verbesserungen boten. Es muss auch immer bedacht werden, dass für die erfolgreiche Klassifizierung eine recht hohe Genauigkeit der Messwerte erforderlich ist, und dass durch Vereinfachungen der Berechnung nicht zu viel Präzision verloren gehen darf. Außerdem kann man aus den Zahlen erkennen, dass sich die Rechengeschwindigkeit mit neueren Geräten und damit besseren CPUs drastisch verbessert und der von Desktop-PCs annähert. Da die Verarbeitung der Sprache für die meisten Anwendungen von AGENDER nicht in Echtzeit erfolgen muss, sondern im Hintergrund erledigt werden kann, können Verarbeitungszeiten im Bereich von einigen Sekunden auch durchaus toleriert werden.

11 Ausblick

Die in dieser Arbeit entwickelte Bibliothek bietet die Möglichkeit, auf dem PocketPC verschiedene Verfahren zur Merkmalsextraktion direkt zu nutzen und aus anderen Anwendungen heraus zu erschließen. Durch die Portierung von PRAAT profitiert sie von dessen hoher Qualität und Leistung.
Die nächsten Schritte werden sein, die Merkmalsextraktion auf dem mobilen Gerät in die M3I-Architektur einzugliedern und ein Client-Konzept für den PocketPC zu entwerfen. Dieser Client soll dann u.a. eine Klassifizierung von Alter und Geschlecht des Benutzers nur anhand von Spracheingabe durchführen können, und zwar auch dann, wenn kein M3I-Server verfügbar ist oder die Verbindungsqualität schlecht ist.

Literatur

[1] BOERSMA, PAUL: *Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound*. IFA Proceedings, 17:97–110, 1993.

[2] KENT, RAY D. und CHARLES READ: *Acoustic Analysis of Speech*. Thomson Learning (Singular), Canada, 2002.

[3] LADEFOGED, PETER: *A Course in Phonetics*. Harcourt College Publishers, Orlando, 4. Auflage, 2001.

[4] MOONEY, JAMES D.: *Issues in the Specification and Measurement of Software Portability*. In: *15th International Conference on Software Engineering*, Baltimore, 1993.

[5] MÜLLER, CHRISTIAN und JÖRG BAUS: *A Client/Server Architecture for Resource Adaptive Speech Processing*. unveröffentlicht, 2003.

[6] MÜLLER, CHRISTIAN und FRANK WITTIG: *Speech as a Source for Ubiquitous User Modeling*. In: *In Proceedings of User Modelling 2003, Lecture Notes in Computer Science*. Springer, 2003.

[7] PLICHTA, BARTEK und MARK KORNBLUH: *Digitizing Speech Recordings for Archival Purposes*. Working Paper, 2001.

[8] TILMAN LIEBCHEN, MARCUS PURAT und PETER NOLL: *Lossless Transform Coding of Audio Signals*. In: *102nd AES Convention*, München, 1997.

[9] WILLINK, EDWARD D. und VYACHESLAV B. MUCHNICK: *An Object-Oriented Preprocessor fit for C++*. In: *IEEE Proceedings on Software*, 2000.

Portierung von Merkmalsextraktion auf die PocketPC-Plattform

Michael Feld

TM-05-01
Technical Memo