M.A.T.
A toolkit for development of TTS technology
Björn Adolfsson

Supervisor: Antonio Bonafonte
Approved: 1st of July 2005
Examiner: Inger Karlsson

....................
(signature)
PFC en tecnología de habla

M.A.T.
A toolkit for development of TTS technology
Björn Adolsson

Tribunal:
Josep Salavedra Moli
Antonio Bonafonte
Cávez
Fransisco Masana Nadal

Tutor:
Antonio Bonafonte
Cávez

Barcelona 29th June 2005
PFC en tecnología de habla
(Master Thesis in Speech Technology)
Departament de Teoria del Senyal i Comunicacion
Universitat Politècnica de Catalunya
C.Jordi Girona 1-3
08034 Barcelona Spain
Examenarbete i Talteknologi
M.A.T.
A toolkit for development of TTS technology
Björn Adolfsson

Godkänt: Examinator: Handledare:
Inger Karlsson Antonio Bonafonte Cávez

Stockholm 29th June 2005

Examenarbete i Talteknologi
(Master Thesis in Speech Technology)
Department of Speech, Music and Hearing
Royal Institute of Technology
SE-100 44 Stockholm
Abstract

The aim of this thesis is to analyse and extend an existing interface to a text-to-speech (TTS) system in order to improve and simplify research activities at Universidad Politécnica de Catalunya, Barcelona. An analysis of the concatenation based TTS system at TALP, UPC, has been made in order to identify its structure and to isolate speech processing modules as they construct the base units in development of automatic TTS systems. The old interface has been replaced with two interactive dynamic interfaces under Linux/UNIX: one for console-use on systems without the X-window manager and a second with support for graphic libraries. Smart user facilities such as command/file completion and variable-resolving has been implemented. Furthermore, the original system has been extended with two speech processing modules which convert the results from an TTS execution to TeX Info and HTML document (version 2.0 and 3.2), allowing results to be exported to standardised document types.
Resumen

El objetivo de este proyecto final de carrera es analizar y extender una interfaz existente de un sistema de conversión de texto a voz (TTS) para mejorar y simplificar las actividades de investigación en la Universidad Politécnica de Catalunya, Barcelona. Se ha llevado a cabo un análisis del sistema TTS basado en concatenación desarrollado en el TALP (UPC), con el objetivo de identificar su estructura y de aislar los módulos de procesado de voz, ya que constituyen las unidades básicas en el desarrollo de sistemas TTS automáticos. Se ha sustituido la interfaz antigua por dos interfaces dinámicas interactivas bajo Linux/UNIX: una para su uso mediante consola en sistemas sin X-Window Manager y la otra con soporte de librerías gráficas. Se han implementado facilidades inteligentes como la expansión de comandos/ficheros y la resolución de variables. Además, el sistema original se ha extendido con dos módulos de procesado de voz que convierten los resultados de una ejecución de TTS en documentos TEX Info y HTML (versión 2.0 y 3.2), permitiendo que los resultados sean exportados a documentos de tipo estándar.
Sammanfattning

Acknowledgement

First I would like to express my sincere gratitude toward Associate Professor Antonio Bonafonte Cávez, not only for making this work possible and for his feedback and guidance during the thesis, but also for his support on everyday issues that arise in a metropolitan city. I wish also to give my thanks to PhD David Sündermann at Siemens Inc and PhD Pablo Agüero for their support and explanations on a variety of topics from basic logic and programming trouble shooting to advanced TTS technology.

Furthermore, I would like to thank my examiners Antonio Bonafonte Cávez, Josep Salavedra Moli, Fransisco Masana Nadal and Inger Karlsson for helpful comments on my work. Finally, my best regards to all the people at TALP, Universidad Politécnica de Catalunya Barcelona, and to my friends Eva, David, Gianni and Sandro for making my stay in Barcelona a wonderful experience.

Björn Adolfsson
Stockholm April 2005
List of Figures

2.1 General structure of a TTS system .............................................. 4
2.2 Prosody generation in a TTS system ........................................... 7
2.3 A formant based rule synthesis scheme ....................................... 9
3.1 The process of the UPC TTS conversion ..................................... 13
3.2 An intuitive multilayer structure .............................................. 14
3.3 Multilayer class hierarchy ....................................................... 15
3.4 SABLE tags in the *Sable text* module .................................... 16
3.5 A synthesis selection example .................................................. 21
3.6 Operators available for the multilayer in Mloperator ..................... 22
3.7 Some graphic libraries for the X-window system ............................ 24
4.1 The flowchart of the interface ................................................. 30
4.2 The menu bar of the GUI ....................................................... 34
4.3 The File menu options .......................................................... 34
4.4 The File menu items .............................................................. 34
4.5 The Text-To-Speech menu options .......................................... 36
4.6 The Module window ............................................................. 36
4.7 The pop-up menu in the module browser .................................. 37
4.8 The parameters associated the *mloperator* module ..................... 37
4.9 The window to change parameter values ................................... 38
4.10 The Preferences window with the general-tab active ................... 39
4.11 The Preferences window with the module-tab active ................... 40
4.12 The Preferences window with the wave-tab active ..................... 40
4.13 The Help menu ............................................................... 41
5.1 General class relations for the Mi_Cli ..................................... 43
5.2 General use-cases in the Mi_Cli interface ................................ 45
6.1 General use-cases in the Mi_Gui interface ................................ 48
7.1 Module class hierarchy ......................................................... 52
C.1 The window displayed when opening an old program ..................... 69
C.2 The window displayed when saving the existing program with a new name 69
C.3 The window displayed when adding a module .................................. 70
C.4 The textfile window .......................................................... 70
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CART</td>
<td>Classification And Regression Trees</td>
</tr>
<tr>
<td>FLTK</td>
<td>Fast Light ToolKit (Fulltick)</td>
</tr>
<tr>
<td>GNU LGPL</td>
<td>GNU Lesser General Public Licence</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model</td>
</tr>
<tr>
<td>IPA</td>
<td>International Phonetic Alphabet</td>
</tr>
<tr>
<td>JSML</td>
<td>Java Synthesis Markup Language</td>
</tr>
<tr>
<td>LPC</td>
<td>Linear Predictive Coding</td>
</tr>
<tr>
<td>M.A.T.</td>
<td>Multilayer Analysis Toolkit</td>
</tr>
<tr>
<td>MBR-PSOLA</td>
<td>Multi-Band Re-synthesised PSOLA</td>
</tr>
<tr>
<td>MFCC</td>
<td>Mel Frequency Cepstral Coefficient</td>
</tr>
<tr>
<td>MLCLI</td>
<td>MultiLayer Command-Line Interface</td>
</tr>
<tr>
<td>MLGUI</td>
<td>MultiLayer Graphical User Interface</td>
</tr>
<tr>
<td>MLTTS</td>
<td>Multilayer TTS system</td>
</tr>
<tr>
<td>MSD-HMM</td>
<td>MultiSpace Probability Distribution HMM</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>POS</td>
<td>Part Of Speech</td>
</tr>
<tr>
<td>PSOLA</td>
<td>Pitch-Synchronous Overlap and Add</td>
</tr>
<tr>
<td>SAMPA</td>
<td>Speech Assessment Methods Phonetic Alphabet</td>
</tr>
<tr>
<td>SPM</td>
<td>Speech Processing Module</td>
</tr>
<tr>
<td>SSML</td>
<td>Speech Synthesis Markup Language</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>STML</td>
<td>Spoken Text Markup Language</td>
</tr>
<tr>
<td>STDLIB</td>
<td>The C++ Standard Library</td>
</tr>
<tr>
<td>TAM</td>
<td>Text Analysis Module</td>
</tr>
<tr>
<td>TCL/TK</td>
<td>Tickle Tee-Kay</td>
</tr>
<tr>
<td>TD-PSOLA</td>
<td>Time-Domain PSOLA</td>
</tr>
<tr>
<td>TTS</td>
<td>Text-To-Speech</td>
</tr>
<tr>
<td>URL</td>
<td>Unified Resource Locater</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
# Contents

1 Introduction ........................................... 2  
   1.1 Thesis description ..................................... 2  
   1.2 Objectives ........................................... 2  
   1.3 Thesis outline ........................................ 3  

2 Text-to-speech systems ............................... 4  
   2.1 Text analysis module .................................. 4  
      2.1.1 Documental structure detection .................. 4  
      2.1.2 Normalisation ..................................... 5  
      2.1.3 Linguistic analysis ................................ 5  
      2.1.4 Pronunciation ..................................... 6  
   2.2 Prosody generation .................................... 6  
      2.2.1 Pauses ........................................... 6  
      2.2.2 Duration .......................................... 7  
      2.2.3 Pitch ............................................. 7  
      2.2.4 Intensity .......................................... 8  
   2.3 Speech synthesis ...................................... 8  
      2.3.1 Formant synthesis .................................. 8  
      2.3.2 Articulatory synthesis .............................. 9  
      2.3.3 Statistic based synthesis ......................... 9  
      2.3.4 Concatenative synthesis ............................ 10  
      2.3.5 Other voice-synthesis-models .................... 11  

3 Case study: TTS at UPC ............................... 12  
   3.1 The analysis technique at UPC ........................ 12  
      3.1.1 The multilayer structure ......................... 13  
   3.2 Text Analysis Module ................................ 14  
      3.2.1 Text analysis ..................................... 15  
      3.2.2 Normalisation ..................................... 16  
      3.2.3 Phonetic transcription .............................. 17  
      3.2.4 Linguistic analysis ................................ 18  
   3.3 Prosody ............................................... 18  
      3.3.1 Pauses ........................................... 18  
      3.3.2 Duration .......................................... 19  
      3.3.3 F0-models ......................................... 19  
      3.3.4 Intensity .......................................... 20  
   3.4 Synthesis .............................................. 20  
      3.4.1 Other modules ..................................... 22  
   3.5 Development languages .............................. 23
Chapter 1

Introduction

Through the last decade the development of TTS technology has expanded rapidly. New theories have risen especially in the field of concatenating unit-selection-based TTS systems. Thus, a need to improve and update analysis tools has surged. In this thesis, a useful improvement of a toolkit that aid researchers at the Universidad Politénica de Catalunya (UPC) in their investigation of TTS technology, will be proposed. The toolkit is designed for the concatenative TTS system developed at UPC.

1.1 Thesis description

At Universidad Politénica de Catalunya (UPC) a state-of-the-art TTS toolkit has been developed. The system is modular including modules for speech analysis, prosody modelling and sound generation. However, the system lacks some properties that would make it more convenient to support research activities.

First, the program is not interactive. Therefore, if the researcher wants to try different models, he/she needs to restart the program with this new setup. The main problem is that a restart of the system is highly costly due to the initialisation of large amounts of data. It would therefore be better if the loading and the execution would be separated. Making it iterative allows performing new experiments very fast, easily and to take new results into account.

Second, the system can be extended but it requires recompiling and relinking. It would be much more convenient to include plug-in functionalities so that only new modules are relinked. Furthermore, this would naturally reduce the executable size because only the explicitly needed modules would be loaded.

Third and last, the system writes the result of the processing of each module into a text-file. However, it is difficult to analyse the results, as there are no tools to monitor this complex data. For instance, if a synthetic sentence does not sound natural, detailed analysis has to be done to determine if the problem is due to modelling of text, concatenation of segments or manipulation of concatenated segments.

1.2 Objectives

The main objective of this thesis is to solve the problems with the UPC-TTS-system. The tasks have been broken down into four parts.

- A case study of the UPC TTS system has been made in order to customise the interactive interface by taking the demands from researchers into account.
• The existing toolkit has been extended. Two interactive interfaces which allow the user to select speech processing modules, reload modules, give help, select text files to process, create new projects, save projects, open projects etc. have been developed. They include smart user facilities such as auto-completion capability of commands, filenames and options.

• The speech processing modules have been organised as dynamic loadable plug-ins, so that new modules can be developed and added to the toolkit easily.

• Two different attempts to visualise the data emerging from an execution of the TTS system have been made. Also, ideas on how to construct a third data visualisation have been proposed.

1.3 Thesis outline

This report starts with a brief introduction to the basics of TTS theory. In the third chapter, the results from a case study of the TTS system developed at the UPC are presented, as they form the base for the solution of the problems. This part also includes information on development environment and implementation ideas which were present in the old interface. This will give some hints to possible extensions and further development. In Chapter 4 an user guide to the new interfaces is given; it also introduces the interfaces’ programmatic structure. Chapter 5 and 6 give a more detailed explanation of the two underlying interfaces of the M.A.T.- Multilayer Analysis Toolkit interface. These chapters are based on the structure of the multilayer introduced and discussed in Chapter 3. Chapter 7 attends to the problem of dynamic plug-in-portability and explains the two new modules developed to monitor the data from a TTS execution. Finally, the last chapter contains conclusions.
Chapter 2

Text-to-speech systems

A text-to-speech system converts written text into speech. This chapter explains the basic theories in a TTS system. In general, TTS systems consists of three general modules, see

![Diagram of TTS system]

Figure 2.1: General structure of a TTS system

Figure 2.1. First, the Text Analysis Module (TAM) which converts the plain text into text associated with its language units\(^1\). Second, the prosody generation tries to generate the melody of the written text. Third the speech synthesis takes all the analysed text with the generated prosodic properties and converts it into speech. All these parts will be discussed briefly below.

2.1 Text analysis module

The TAM is responsible for indicating all knowledge about the text that is of prosodic\(^2\) nature. This module determines attributes which support the prosody modules in the generation of the correct prosodic representation. The TAM can be broken down in many different parts. I have chosen to split it into four steps: Structure detection, Normalisation, Linguistic analysis and Phonetic transcription as these divisions have been found frequently in literature Huang, X. et al (2001); Bonafonte, A., Escudero, D. & Riera, M. (2004).

2.1.1 Documental structure detection

During the structure detection, the document syntax is identified and split into sections by a marker system. These markers or tags, split the document into different units. Each unit can then explicitly be analysed by other modules. The representation of this structure is preferably expressed with a widely known syntax as this facilitate comparison between systems and identification of text analysis steps. Extensible markup language (XML) or the speech synthesis markup language (SSML) are recommendable markup languages as they are common and use a clear syntax. The usage of XML or SSML results in an output from

\(^1\)The units that describe a language, such as phonemes, syllables, words, letters etc.

\(^2\)The melody of the speech.
the TAM which can be used by other applications. Markup languages aid the system with the specification of document sections (paragraphs, syllable etc.) and speech settings (who is speaking and in what mode). Some elements of document structure e.g. the punctuation marks (semicolon, commas, etc.) have direct implications on the prosody generation as they indicate sections of silence. A system may of course use its unique tagging system, but a common syntax is of global benefit and make system results portable.

A system must also be able to convert unmarked text, without loosing naturalness and prosody features. However, this is not a straight-forward process. For example; the determination of the language used in a short text may be difficult to predict, as it may not contain sufficient number of words to predict the language. Further problems are the punctuation marks, as they are used both in abbreviations and in structural detection. A demonstration is given in the example below.

Mr. Heaney, the teacher, wrote: 1.0 + 1.0 = 2.0.

As we can see this obviously creates problems. But, bearing in mind that a automatic TTS system rather should be focused on the main bulk of the text instead of solving special cases, we might accept these errors. Using rules to solve frequently encountered trouble-sequences is one of many ways to get around the problem.

### 2.1.2 Normalisation

Input text often includes sequences of characters with a specific significance to human readers such as abbreviations, acronyms, mathematical formulae, charts etc. Since these sequences mostly are uttered in a different manner from how they are written, there exists a need for an interpreter that converts the special sequence of characters to a standardised/normailised text. For example: Mr. is normally pronounced mister rather than [M]/[fr]. Thus, the normalised text will contain the sequence to word translation that describes how to interpret and pronounce the sequence. Also the inverse normalisation may be of interest in certain conversion systems, i.e. converting text into specialised abbreviations, acronym etc. The normalisation faces many complicated problems which are due to the specific language rules. These problems are normally solved by associating the sequences to special stored pronunciation objects, converting them to known sequences (rules) or by statistical analysis.

Abbreviation ambiguities do exist and is a complicated problem within the borders of normalisation. Since sequences like St. can have different meanings (Street, Saint etc.), the contextual information hidden in a text influence our choice of pronunciation. One way to deal with this problem can be by statistical means and document classification Huang, X. et al (2001).

### 2.1.3 Linguistic analysis

Linguistic analysis or syntactic and semantic parsing is a parser which analyse all known text according to the linguistic rules that apply to the appropriate language. One particular problem is the ambiguity of words. For example, the English word *mind* may signify the noun as well as the verb. Fortunately in the case of *mind* the pronunciation is the same and the prosody will be generated correctly. The verb *read* however, does not possess that property and the ambiguity is a fact. The ideal parser has to be able to convert and resolve these ambiguous words. This is normally not a straight-forward process. Context
information has to be analysed before the parser can guess or determine the proper pronunciation of the word. Using the part of speech (POS) tagging labels is one of many tactics to solve ambiguities. Other techniques and information are discussed in Huang, X. et al (2001) Chapter 14.

2.1.4 Pronunciation

Once we have the tagged text, we must determine the pronunciation of the text. Thus a translation from letters to pronunciation symbols is needed. This is normally made by a grapheme\(^3\) to phoneme\(^4\) conversion. This transcription can be made by using a large lexicon with transcriptions for each word in combination with morphological rules. When an unknown word is encountered, language specific rules on how to transcribe a word with phonemes (letter to sound) can be used.

It is also of interest to determine the syllables in the actual text as they contain information on the stress pattern (stressed syllable) of words. The syllables of each word are, as well as the phonetic transcription, normally stored in the dictionary and the syllable that carries stress is marked. If the words are not in the lexicon, rules or statistics can determine the syllable and which syllable(s) is/are ought to carry stress (have intonation) according to language rules. The last step before the generation of prosody is the assignment of accents on the sentence level. This marking normally takes into account information concerning if the words carry the main information of an utterance (content words) or are function words (articles, conjugations etc.) as content words tend to be accented more often than function words.

2.2 Prosody generation

The prosody model is a mixture of physical and phonetic effects that speakers use to generate the speech rhythm; that is speech attitudes, assumptions, attention etc. The written phrase can be pronounced in many different ways by a speaker, depending on his/hers emotional and attentional states. Theses states are due to circumstances of the text such as its type (email, news etc.) and the environmental effects (speaker modes etc.). For example; when reading a story to a child the prosody is clearly different from when the same person reads the news even though there may be many similar phrases within the two texts.

The prosody generation deals with these problems by assignment of segmental duration, pitch contour, intensity of the phonemes and pauses between words. The assignment of breaks and sentence accents is sometimes called symbolic prosody. Figure 2.2 below, contains a basic scheme of the prosody generation. Some systems use parametrised algorithms to model the prosody while others use real speaker measured data via table look-up methods Huang, X. et al (2001).

2.2.1 Pauses

In a long sentence, speakers normally pause a number of times. These pauses have traditionally been thought to correlate with syntactic structure but might more properly be thought of as markers of information structure Huang, X. et al (2001). There are many places to pause in a long sentence, but a few where it is critical not to pause. The main goal

\(^3\)Greek: graph is graphic, -eme is symbol

\(^4\)Greek: phonema which means sound symbol
of a TTS system is to avoid placing pauses anywhere that might lead to misinterpretation or breakdown of understanding. Normally, the pauses are added to the text as complementary information for the synthesis process in a tag-like form. For example <PAUSE DURATION=10ms> or <break>.

There exist two different types of pauses: human respiration pauses and linguistic pauses. The relation between these pauses has to be taken into account in a TTS system, as speakers tend to place respiratory pauses at linguistic pauses. However, pauses can not simply be inserted at certain points in a phrase to make good prosody. The insertion of pauses influence other aspects of prosody as well.

2.2.2 Duration

The duration characteristics normally are investigated at several levels since it is interesting to observe parameters such as speaking rate (durations of words) and phoneme duration. The duration contains not only phonetic information, but as well the speaker state for example; anxiety, stress, calmness etc. Many heuristic methods have been proposed to change the durations, but since the relationships between durations and levels of speech (word level, paragraph level etc.) are complex, an integral method has not yet been developed. One manner to determine the duration is by rules, introduced by Klatt, D.H. (1987). However, there are many more and among the today commonly used methods are the methods which automatically infer durations from databases, for example CART Breiman, L. et al (1984); Hirschberg, J., Prieto, P. (1996).

2.2.3 Pitch

The pitch of a sound refers to the human perception of the variation in the fundamental frequency of a sound over time. This is normally described, in terms of a TTS system, by the pitch contour (fundamental frequency over time, F0). Pitch generation is commonly divided into two levels, first the symbolic prosody (2.2.1) and then the generation of the pitch contour. The pitch contour is modelled by at least four different features: Pitch range, gradient prominence, declination and micro-prosody Huang, X. et al (2001).

- Pitch range refers to the limits within which all accent and tones must be realised. Pitch range variation that is correlated with emotion and attention is called paralinguistic.
• Gradient prominence is the relative strength of a given accent position with respect to its neighbours and the current pitch range value.

• Declination is the long-term downward trend of pitch across a typical reading style within a sentence.

• Micro-prosody is the part of the contour which may be a pure phonetic aspect of the sound. It may also signal speaker state, such as the variation of cycle lengths in pitch-period (jitter) and the variation in energy values of cycles (shimmer) attributes according to Huang, X. et al (2001). For example a more closed vowel, /i/, has a higher F0 than a nose open vowel, /a/.

The pitch contour can also be modelled in different manners, as by data measured from real speakers and table lookup methods as well by parametric methods. One of many parametric models is the superposition model. It is usually composed of super-positioned contour-components which added together construct the model of the F0-curve.

2.2.4 Intensity

The intensity is one of the prosody parameters which has not been given much attention, but is today an interesting field of research especially within the area of expressive speech. Some pronunciation of words with special significance (for example specific names, identifiers etc.) may successfully improve the clarity of a phrase by a simple increment or decrement of its intensity.

2.3 Speech synthesis

When the phonetic characters have been determined with their proper prosody, the last stage before finishing the TTS-conversion is the voice generation or speech synthesis. This module takes the phonetic information and converts it into a waveform. The most common synthesis methods are normally classified within four classes: formant synthesis, articulatory synthesis, statistical synthesis and concatenation synthesis.

2.3.1 Formant synthesis

The formant synthesis is a signal-processing approach to generate the output speech. The information that humans require to distinguish between vowels can be represented purely quantitatively by the frequency-contents of voiced sounds Klatt, D.H. (1987). These characteristic spectra, formants, are formed by resonances in the vocal tract.

Formant synthesis does not use any human speech samples at runtime. Instead, the output speech is created using an acoustic model of the human speech production Klatt, D.H. (1987) and two types of input signals: noise to model unvoiced sound and impulse trains to model voiced sound. Parameters such as fundamental frequency, voicing, noise levels etc. are varied over time to create a waveform of artificial speech. Thus, each synthesised phoneme or pronunciation symbol is related to a specific description of the human speech production, i.e. the source filter with coefficients. Many formant synthesis based systems generate artificial, robotic-sounding speech, and the output would seldom be mistaken for the speech from a real human. However, maximum naturalness is not always the goal of a speech synthesis system, and formant synthesis systems have some advantages over concatenate systems. One of the major is that formant synthesisers are often smaller
Figure 2.3: A formant based rule synthesis scheme.

programs, because they do not have a database of speech samples. Thus, they can be used in embedded computing situations where memory space and processor power often are scarce.

2.3.2 Articulatory synthesis

Articulatory synthesis is based on computational models of the human vocal organs and the articulation processes occurring there, including teeth, lips etc. It can be compared with formant synthesis but it is more human-like and versatile, thus more complex. The description of the mechanical motions of the articulators and the pressure in the lungs and the tracts are normally included in the model. This is theoretically the most accurate method but it suffers from two major problems: measured data is difficult to collect and models are computationally complicated Lemmetty, S. (1999). These models are currently not sufficiently advanced to be used in speech synthesis systems but interesting for researchers.

2.3.3 Statistic based synthesis

Statistical synthesis is today based on hidden Markov models (HMM) Jelinek F. (1976); Jelinek F. (1998) and has its origin in the speech recognition theories, i.e the conversion of a waveform into text. The implementations of HMMs in the inverse form, TTS, have not yet reach its culmination and is a fairly new approach. Most of HMM based TTS systems use wave concatenation as the base. Complete HMM based synthesisers have also been constructed with the use of multi-space probability distribution HMM (MSD-HMM) Masuko, T. (2002). In those synthesisers speech parameter sequences are generated from HMM directly based on maximum likelihood criterion, where each model (HMM) typically corresponds to a certain triphoneme or phoneme. By considering relationships between static and dynamic parameters, smooth spectral sequences are generated according to the statistics of the static and dynamic parameters modelled by HMMs through a special source generator. This includes models of the fundamental frequencies (F0) patterns. The benefit of HMMs compared to traditional concatenation TTS systems is that since there is no need for a large database of prerecorded sound segments, automatic TTS systems can be implemented on machines where storage is limited.

\footnote{A source and a filter controlled by the exact parameters generated by the HMMs}
2.3.4 Concatenative synthesis

The most common synthesisers today are those that concatenate recorded waveforms into speech. Concatenative synthesis is based on the concatenation of segments of recorded speech. Generally, concatenation synthesis gives natural sounding synthesised speech if the concatenated parts are well chosen. However, natural variation in speech and automated techniques for segmenting the waveforms sometimes result in audible glitches in the output, detracting from the naturalness. There are three main subtypes of concatenation synthesis: unit selection, diphone and domain-specific synthesis.

Unit selection synthesis

Unit selection synthesis uses large speech databases. During database creation, each recorded utterance is segmented into some or all of the following: individual phones, syllables, morphemes, words, phrases, and sentences. The division into segments can be done using a number of techniques: clustering Berkhin, P. (2002), modified speech-recognisers, by hand (rules), visual representations such as the waveform and spectrogram. An index of the units in the speech database is then created based on the segmentation and acoustic parameters like the fundamental frequency. At runtime, the desired target utterance is created by determining the best chain of candidate units from the database (unit selection). The output from the best unit selection systems is often indistinguishable from real human voices, especially in contexts for which the TTS system has been specialised (travel information systems, etc.). However, maximum naturalness often requires unit selection speech databases to be very large, in some systems ranging into gigabytes of recorded data and numbering into the dozens of hours of recorded speech.

Diphone synthesis

Diphone synthesis, which is highly related to unit selection synthesis, uses a minimal speech database containing all the diphones (two half phones modelling the transition between phonemes) occurring in a given language. The number of diphones depends on the phonotactics of the language. Spanish and Catalan has about 800 diphones. In diphone synthesis, only one example of each diphone is contained in the speech database. At runtime, the target prosody of a sentence is superimposed on these minimal units by means of digital signal processing techniques such as LPC, PSOLA, TD-PSOLA or MBR-PSOLA. The quality of the resulting speech is generally not as good as that from unit selection, but more natural-sounding than the output of formant synthesisers. Diphone synthesis suffers more from the glitches of concatenation synthesis and the distortion of the prosodic changes created by the synthesis methods. However, systems based on diphone synthesis can be constructed with a small database compared to other synthesis systems.

Domain-specific synthesis

Domain-specific synthesis concatenates pre-recorded words and phrases to create complete utterances. It is used in applications where the variety of texts which the system will output is limited to a particular domain, like weather reports and central talking clocks. This technology is very simple to implement, and has been in commercial use for a long time. The naturalness of these systems can potentially be very high because the variety

---

7For example: Fröken Ur in Sweden
of sentence types is limited and closely matches the prosody and intonation of the original recordings. However, because these systems are limited by the words and phrases in their database, they are not general-purpose and can only synthesise the combinations of words and phrases they have been pre-programmed to do.

2.3.5 Other voice-synthesis-models

The above mentioned synthesis technologies are the most common in today’s TTS systems. However, there do exist other systems (hybrid-synthesis systems) which join aspects of formant and concatenate synthesis to minimise the acoustic glitches. See reference Lemmetty, S. (1999) for a complete list of speech synthesis technology.
Chapter 3

Case study: TTS at UPC

In order to improve the structure of the existing TTS system an analysis of its implementation has been made. This case study first introduces the data structure of the UPC TTS system, i.e. the multilayer, which is the fundamental concept of the UPC TTS system. Second, the three different stages: TAM, prosody and synthesis, previously introduced in Chapter 2 will be analysed and explained. Each stage consists of a set of speech processing modules, which are presented and described shortly. These two parts described the state of former UPC TTS system. Third and last the existing implementation, including choices of programming languages for its improvement is discussed.

The UPC TTS system is based on concatenation of natural speech segments for two different languages: Spanish and Catalan (the system is used by a company with several other languages included, see 3.2.1). Both UPC languages are derived from Latin. Thus, they possess similarities in their linguistic structure and orthography, but their pronunciation is distinct. These advantages and problems are taken into account in the speech processing modules developed in the old UPC TTS system. For further information about the system, please contact the responsible personnel at the TALP research centre, http://www.talp.upc.es/.

3.1 The analysis technique at UPC

One of the crucial events in the development of a TTS system is the representation of all the analysis information created during the TTS process. Since no designed TTS system creates the perfect human utterance one has to change and manipulate the speech-processing parameters\(^1\) before acceptable speech can be produced. The flow of the data and the technique of analysing TTS conversion is shown in Figure 3.1. An input text is processed by a list of speech processing modules and the result of each process step is stored in a storage container. If the synthesised speech has low quality the speech processing parameters stored in the container are analysed and adjusted. Thus, it is of great interest to visualise analysis parameters and their relationships. To do this, a main parameter container which is flexible and controls all types of parameters and links that are found in the speech description are in use at UPC.

The UPC system uses a main container called multilayer (explained below, 3.1.1) which is processed by the different speech processing modules. Some modules are stand-alone, whilst others are dependent on previously executed modules. The structure of a single module is discussed in Chapter 7. Worth noticing at this level is the sequence of analysis:

\(^1\)F0, filter coefficients etc.
If there are Speech Processing Modules

Process and add info to multilayer

True False

Listen to speech signal

Sounds ok?

True False

Create a new module list

Change values of parameters

Show multilayer

Figure 3.1: The process of the UPC TTS conversion

First, an empty container (multilayer) is created and an input file is fed into the system. Second, modules are processed and depending on the input text speech information is created inside the container. Third and last the information stored in the layer is analysed and recorded speech segments are concatenated to synthesised speech. Depending on if the speech was acceptable, some modules may be re-processed with other settings to improve the result. The modules in the UPC TTS system are given an explanation and a relation to the theories presented in Chapter 2, in the section 3.2, 3.3 and 3.4 below. The information is taken from interviews with Pablo Agüero and Antonio Bonafonte, responsible personnel for the module system.

3.1.1 The multilayer structure

One intuitive structure for a TTS conversion system is to have all linguistic levels represented as layers where each layer contains information about every unit or parameter stored in the layer. An example is shown in Figure 3.2.

The UPC TTS system’s smallest unit, mlValue, has been chosen to consist of one of three standard types: string, float or integer. All containers and the structure of the UPC TTS are introduced below.

mlValue

The class mlValue is the base unit for the representation of data in the UPC-TTS system. It is simply a container which can contain either a string, float, or an integer. They are normally used as identifiers in layers. The CHUNK-layer would contain strings and the other layers might contain floats or integers.
mlKeyValue

The mlKeyValue-class is a map container which associates an identification key, of the type string, with a mlValue container. Thus one may associate speech-processing parameters with values, for example DURATION with the value 50ms.

mlNode

The nodes are the most important containers since this is where the actual analysis information is stored. The node representation is what normally relates speech analysis information with other information within the analysis. For example a certain unit in the layer CHUNK, that is a sequence of text, may be related to a certain syllable in the layer SYLLABLE. Therefore, the mlNode class does not only contain values. It also contains all links between the actual node and the nodes related to it. This is solved by adding information about the container of the node, i.e. the layer, and by using pointers to relate to other nodes. The nodes can be seen as an instance of a mlKeyValue with a linked list to other mlKeyValues.

mlLayer

The layers are containers of nodes, mlNodes. The layers can have links to other layers. The links are not only direct links, but can also be through other layers (indirect links).

mlMLayer

The multilayer class (mlMLayer) is the main container of all the information needed to convert a given input text to a certain output speech. It is to be considered as a simple container of speech-processing-layers, where each layer corresponds to either linguistic units or other speech related units that simplifies the human interpretation of a text’s structure. From the implementational point of view, the multilayer contains different links (C++ iterators) to the information contained inside the each layer.

This system gives great flexibility to the intended users as many of the speech parameters and variables can be modelled with the multilayer structure. In Figure 3.3 the multilayer-class relation is shown with Unified Modelling Language (UML) scheme UML (2004). From now on all data structure and relations are shown with UML version 1.0. Only the most important attributes are shown in the class boxes.

3.2 Text Analysis Module

The text analysis module at UPC is divided into three different levels: plain text analysis,
normalisation and phonetic transcription. Each one is referred to as a stand-alone module or a composite of such modules.

3.2.1 Text analysis

Since a text can be of many different types and be divided in many different ways: parts, chapters, paragraphs etc. this module should be given special attention. Every clue extracted from the structure of the text contains important information for the prosody and the synthesis analysis. At UPC the information is coded with SSML\(^2\) and SABLE\(^3\) tags, identified by the Sable text module.

SABLE text

The Sable text module analyses the text and identifies SABLE tags markers. The SABLE markup tags consist of the markers in table 3.4. For each identified tag a layer is created. SSML syntax, which is similar to SABLE tags can also be identified. The UPC-system is also capable of separating none/partly marked simple structures through small heuristic models specially adapted to Spanish and to Catalan.

Set Language

The Set language module sets the language of the text if the SABLE-tag with the same name was specified, otherwise a default value is chosen. Then files with information on where to find class rules, expand rules etc. are looked for as these rules are language specific. The UPC-TTS contains several voices for a set of languages: Mexican, Portuguese, Brazilian, French, Spanish and Catalan.

Set Speaker

The Set speaker module sets the speaker and his/hers characteristics (speaker rate etc.). In the UPC-TTS system there are several different speakers (about 10) with typically 1-3

\(^2\)http://www.w3.org/TR/speech-synthesis/
\(^3\)http://www.bell-labs.com/project/tts/sable.html
<table>
<thead>
<tr>
<th>TAG ID</th>
<th>Attributes</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;EMPH&gt;</td>
<td>LEVEL</td>
<td>Sets the emphasis of the text</td>
</tr>
<tr>
<td>&lt;BREAK&gt;</td>
<td>LEVEL, MSEC, TYPE</td>
<td>Sets a prosodic break</td>
</tr>
<tr>
<td>&lt;PITCH&gt;</td>
<td>BASE, MIDDLE, RANGE</td>
<td>Sets pitch properties</td>
</tr>
<tr>
<td>&lt;RATE&gt;</td>
<td>SPEED</td>
<td>Sets the speech rate of text</td>
</tr>
<tr>
<td>&lt;VOLUME&gt;</td>
<td>LEVEL</td>
<td>Set the volume</td>
</tr>
<tr>
<td>&lt;AUDIO&gt;</td>
<td>SRC, MODE, LEVEL</td>
<td>Load and play an audio URL</td>
</tr>
<tr>
<td>&lt;MARKER&gt;</td>
<td>MARK</td>
<td>Serves as a anchor</td>
</tr>
<tr>
<td>&lt;SABLE&gt;</td>
<td></td>
<td>Identifies the document as SABLE-doc</td>
</tr>
<tr>
<td>&lt;SAYS&gt;</td>
<td>MODE, MODETYPE</td>
<td>Defines the way the text is to be said</td>
</tr>
<tr>
<td>&lt;LANGUAGE&gt;</td>
<td>ID, CODE</td>
<td>Sets the language</td>
</tr>
<tr>
<td>&lt;SPEAKER&gt;</td>
<td>GENDER, AGE, NAME</td>
<td>Sets the speaker</td>
</tr>
</tbody>
</table>

Figure 3.4: SABLE tags in the *Sable text* module

hours of recorded speech for each speaker.

### 3.2.2 Normalisation

The normalisation at UPC is made by rules as it is a fast and effective way of solving abbreviations, dates etc. in Spanish and Catalan. The normalisation can solve acronyms, abbreviations, Arabic and Roman numerals, time expressions and dates. There are multiple content of ambiguities within these contexts. Since there are many different rules in the UPC-TTS, only two short demonstrations are given here. In all cases where the normalisation cannot be completed, the basic rule is to treat the sequence as individual characters.

Example date:
01/01/2004, 01.01.2004, 01/1/2004, 01/Enero/2004

If these sequences are encountered in the text, the parser will try to convert the first number to a day, the second token (number or string) to a month and the third number to a year. If the parser does not succeed, due to unknown dates etc. the numbers are treated as sequence of numbers and strings as words. In our case the translation for the dates in the sequence would be: *UNO DE UNO DE DOS MIL CUATRO* except the last one, which would be rendered as *UNO DE ENERO DE DOS MIL CUATRO*.

Example special names:
WindowsXP and Forum2004

These special words are treated as a separate word until the uppercase characters/digits are encountered. The characters are taken as a sequence of pure characters. The semantic translation would be:

```xml
<word>WINDOWS</word> <char>XP</char> and
<word>FORUM</word> <card>2004</card>.
```

From the above mentioned examples we can conclude that character pronunciation is
to be used when there are semantic unclarieties about sequences of tokens. This is not ideal since it does not produce a correct conversion but at least it does not confuse nor does it change the interpretation of the text.

The two modules which implement these rules are WORDCLASS and WORDEXPAND.

WORDCLASS
The Wordclass module classifies the characters and creates the TOKEN layer. It identifies sequences with known pronunciation, for example using the rules above. It is done by passing a mask through the text and when a match is found, it is marked by the SAYAS tag and then expanded by the Wordexpand module. The SAYAS tag defines the way the text is to be read, for example: <SAYAS DATE>. The rules must also be tried in a specific order.

WORDEXPAND
The Wordexpand module expands the tokens into words and expands the information in SAYAS to normalised text, i.e. it converts 1 to UNO.

3.2.3 Phonetic transcription
The Phonetic transcription re-writes each word according to its pronunciation, using a phonetic alphabet. The most common phonetic alphabet is the International Phonetic Alphabet\(^4\) (IPA). In Europe TTS systems normally use the SAMPA\(^5\) alphabet. The process of transcription is based on a dictionary with words and their phonetic transcription and a set of methods (knowledge based rules, data mining (CART) etc.) that solves the problems of transcription of unknown words. In Catalan some ambiguities occur that rules cannot solve and therefore a dictionary is needed. For the vowel e, if written without any orthographic accent, is impossible to determine the correct phoneme (either /E/ or /e/) in a stressed syllable Bonafonte, A., Escudero, D. & Riera, M. (2004). In Spanish the TTS systems use some hundred different rules that covers the mayor part, but not all, of the existing words Bonafonte, A., Escudero, D. & Riera, M. (2004). If a word has no explicit phonetic transcription, the transcriptions can be approximated by linear letter-to-phoneme binary question trees. The functionality is the following: At each branch in the tree is checked asked if the letter has a certain attribute. At the end of the tree a phonetic symbol with the most probable pronunciation of the original letter is encountered.

Some languages, as for example Finnish, use conjugations that are added to the end of words. In these cases morphological analysis can be of great advantage since it reduces the dictionary size and makes the TTS system faster. However, morphological rules are complex. In the case of Spanish the morphological approach is not worth the cost, since the same result can be obtained by doubling the size of the dictionary Bonafonte, A., Escudero, D. & Riera, M. (2004). At UPC the transcription is made by this larger dictionary for both Catalan and Spanish. However, since there are many homographs\(^6\) (orthographically identical words) in Catalan and Spanish, linguistic analysis has to extend the interpretation-information of the text. There are two modules that handle the transcription from orthography to phonetic: PHONETIC and PHONOTACTIC. Furthermore there are two modules that treat phonetic conversion and relations: PH2DP and PHONLOGIC.

\(^4\)http://www2.arts.gla.ac.uk/IPA/ipa.html
\(^5\)http://www.phon.ucl.ac.uk/home/sampa/home.htm
\(^6\)from Greek: homo - grapho ; homo means same, graph means written
PHONETIC
The Phonic module uses a lexicon database that converts each word into a phonetic sequence. It also determines stressed syllables and sentence accents for words not present in the dictionary by rules.

PHONOTACTIC
The Phonotactic module changes the phonetic transcription in case of co-articulation. For example: la andaluza (the Andalusian girl) will be corrected to landaluza.

PHONOLIGIC
The Phonologic module finds information on which word and which triphoneme a demi-phoneme Mariño, J.B. et al (1997) belongs to. It also includes information on the position within the syllable, the associated diphone, which level of stress is applied and the position relative to the next phrase break (pause). This information is later used by the selection module to select the best unit.

3.2.4 Linguistic analysis
The linguistic part of the TTS-conversion at UPC is based on POS indicating labels. The module that carries out the POS tagging is called TAGGER.

TAGGER
The Tagger module identifies each word as linguistic type by POS-tag. This is done by a look up in a word database where each word corresponds to a code which explains the type of the word. If there are ambiguities these are solved by context-dependent rules and statistic data. For example Compró un coche rojo, porque el rojo es más juvenil. (I bought a red car because red is more youthful) contains the same word rojo but the first one is an adjective and the second one a noun. Thus a rule may be; since rojo statistically occurs often as adjective it is adjective by default but when it is preceded by an article and followed by a verb it is a noun.

3.3 Prosody
The prosody model at UPC is processed in four steps. First, pauses are analysed. Second, each phoneme is assigned a duration. Third, the F0-curve is modelled. Last, the intensity is modelled based on the segment type. The modules chosen to perform these stages are selected manually by the user.

3.3.1 Pauses
In an arbitrary text pauses are indicated with different type of separators such as commas, semicolons etc. In western-world languages the punctuations normally are the pause-indicators. At UPC the pauses can be assigned by three different modules, with three different theoretical approaches.

Pause Rule
The Pause rule module introduces pauses according to rules. For instance, a pause can only be introduced after a content word and before a function word. It takes into account
the maximum length between pauses.

**Pause CART**

The *Pause cart* module uses a CART Breiman, L. et al (1984) tree to decide if there is a pause after a word. The features used include POS-tags from the three previous and following words, and the distance to a punctuation mark in words and syllables Hirschberg, J., Prieto, P. (1996).

**Pause FST**

The *Pause FST* module is based on a finite state transducer (FST). The source language is the position of words and the target language is True, False. As in the previous cases the transducer is inferred from training data. See Bonafonte, A., Agüero, P. (2004) for further information.

**3.3.2 Duration**

The UPC duration analysis is based on three different models: a rule based duration selection algorithm, a data exploration model which uses linguistic information (POS-tag) and a sum of products.

**Duration**

The *Duration* module models the sequential duration using a model called sum-of-products Febrer, A., Padrell, J. & Bonafonte, A. (1998). The duration is computed taking into account the phoneme, if it is prepausal, if it is a stressed syllable etc. The parameters are estimated from data.

**Duration Kb**

The *Duration Kb* module uses a rule-based model with explicit linguistic information to calculate the duration (Knowledge Based). It establishes the duration of each phoneme using phonetic rules. For instance, there are different values for each vowel depending on stress, if it is prepausal and phonetic class of the following phoneme.

**Duration CART**

The *Duration cart* module uses a data exploration algorithm (a regression tree) to estimate the duration. The features are the same as in the previous modules (stress, identity of phoneme, properties of next phoneme etc.)

**3.3.3 F0-models**

The variation in the fundamental frequency can be estimated by many methods. At UPC this intonation curve can be modelled by four different models: line segments, superpositional Bezier-curve segments, Fujisaki model and templates.

**Line**

The *F0-line* module produces the F0-contour based on two components, one for the phrase and one for the stress. The phrase component is a piece-wise straight line. The accent is a hat contour that is added at the stressed syllable.

**Bezier Superposition**
In the *Fo-line* module, the F0-contour is modelled by two components: the phrase and the stress. These components are in this module, *Bezier superposition*, represented by a vector of bezier curve parameters. The parameters of each curve is predicted using regression trees based on features as sentence mode, number of phrases, position of phrase in sentence, number of syllable in the stress group, position of stressed syllable etc. Agüero, P. & Bonafonte, A. (2004).

**FUJISAKI**

The *Fujisaki* module Fujisaki, H. & Sudo, H. (1971) generates the F0 contour in the logarithmic frequency domain superposing the baseline value \(\ln(F_0)\) and the output of two critically damped linear filters. The first system account for the global phrase component and the second for the local accent component. The phrase command defines the amplitude and the time of the delta pulses that enter to the phrase pulse. The accent commands are defined by the amplitude, position and duration of pulses that enter to the accent filter. These inputs can be defined by rules; however the UPC system finds the values using CART trees. The algorithm is explained in Agüero, P., Wimmer, K. & Bonafonte, A. (2004). The features are the same as the ones introduced in the previous modules.

**TEMPLATE Mbl**

The *Template Mbl* module copies the F0 contour from example contours in the database (Memory based learning). For each accent group, a similar accent group (with similar prosodic features, i.e. duration etc.) is searched for in the database. The F0 is then adapted to match the duration and the stress position by means of morphologic techniques (interpolations, smoothing etc.).

### 3.3.4 Intensity

In the UPC TTS system the intensity of the base unit is modelled according to two different theories.

**Energy**

The *Energy* module calculates the energy (intensity) using simple rules. For each phoneme the mean energy is calculated from a database. Then, if the syllable is stressed the energy of the phoneme is increased by a given amount. Furthermore, at the end of a sentence the energy is decreased.

**Energy Model**

The *Energy model* module calculates the energy of the phoneme by regression trees, similar to what was done in the *Duration cart*. Furthermore, for the most frequent diphones the energy in the juncture is tabulated.

### 3.4 Synthesis

The synthesis is based on a database with natural sound segments, split into half-phones (diphones), which are concatenated according to the information given by the TAM and the prosody modules. The speech fragments are chosen according to a selection algorithm that minimises the frequency discontinuities between the fragments. This implies that the durations of the word/phrase are not exactly the same as for the ideal phrase since a recorded fragment may be longer or shorter. The selection algorithm which is a pre-module
to the synthesis, elects the phonetic units so that the synthesis needs to do the least concatenations possible. The complete algorithm is described in Febrer, A. (2000). In the system, these steps are represented by PH2D, PHRASE SELECTION, SELECTION and SYNTHESIS.

PH2DP

The phoneme to demiphone module (Ph2dp) splits each phone with its determined characteristics to two demiphones. A demiphone can be seen as half a phone. A phone is a realisation of a phoneme. The databases at UPC store demiphones.

PHRASE SELECTION

The Phrase selection module looks for stored phrases that are to be used as they were recorded. This should be done before any prosody modelling, since a minimisation of the number of synthesised segments is preferred.

SELECTION

The Selection module determines which segments are to be concatenated. This algorithm is based upon the minimisation of two cost functions, the target cost $C_{Target}$ and the concatenate cost $C_{concatenation}$.

- The target cost is calculated taking into account fundamental frequency, energy, duration, triphone, diphone and word. The objective is to find a recorded element that matches the unit as closely as possible, thus minimising the cost.

- The concatenation cost is the cost of bringing two recorded segments together. Thus, it depends on transition, spectral values (Mel Frequency Cepstral Coefficients, MFCC), continuity in F0, inclination of F0 and energy. The objective is to maximise the number of units taken from the same recorded phrase, as this aids naturalness.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>PHL</th>
<th>Target PH</th>
<th>Instance PH1</th>
<th>Instance PH2</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0end:</td>
<td>80Hz</td>
<td>50Hz</td>
<td>60Hz</td>
<td>70Hz</td>
</tr>
<tr>
<td>F0begin:</td>
<td>105Hz</td>
<td>80Hz</td>
<td>87Hz</td>
<td>93Hz</td>
</tr>
<tr>
<td>Duration:</td>
<td>67ms</td>
<td>95ms</td>
<td>107ms</td>
<td>97ms</td>
</tr>
</tbody>
</table>

Figure 3.5: A synthesis selection example.

For example the left phoneme PHL defined in Figure 3.5 is to be connected with a second phoneme PH1 or PH2, see same figure. From our previous analysis steps we know that the target phoneme should have some properties shown in Target PH. However, the phonemes which are stored in our database (PH1,PH2) do not possess these values. As can be seen from table 3.5 the best frequency option is PH1 since it minimises the frequency separation compared to the Target PH, but at the cost of a increased phone duration. However, a minimisation of a single segment is not sufficient as the cost of the whole sequence needs to be minimised. Thus local mismatches may occur to the benefit of a low total cost. After the calculation of the cost the chosen segments are related to a file where their waveform is stored and a label which indicates their start and end times is added to the segment so that the synthesis knows from where to take the segments.
Synthesis

The Synthesis module concatenates the chosen segments by means of signal processing methods. The selection module indicated which segments were to be concatenated and at what point in time. Thus, the synthesis algorithm chooses a low energy zone between marked pitch zones to minimise distortion and then applies the Time Domain Pitch Synchronous Overlap and Add (TD-PSOLA) to concatenate the corresponding segments. This also allows the user to modify the pitch and the duration of the sequence.

3.4.1 Other modules

There are some modules implemented in the system which are not of direct speech processing character, but useful in many other ways. These modules are explained below. They are normally used when a specific module is changed or when a database is created as the input file may not always be a text file. It could also be a segment marked sound file that is to be converted to a multilayer.

Text2Speech

The Text2speech module contains all the above discussed modules necessary to make a TTS-conversion. The configuration of this module makes the UPC-TTS produce an optimised TTS-configuration.

MLoperator

The Mloperator module is used to change values inside a layer or to manipulate standard module parameters. The defined operators are shown in Figure 3.6. An example is: LAYER = "PHONE" OP = "D += 30; F0 -= 40; SEG ; LANGUAGE = es"

This sequence of operators, separated by semicolons, work on the layer PHONE. This operation will add 30 ms to D (Duration), diminish the fundamental frequency by 40 Hz, eliminate the chosen SEG and set the language to es (español).

| Operator | Semantic                   | Values   |
|----------|----------------------------|----------!|
| =        | Set to new value           | int, float, string |
| +=       | Add to existing value      | int, float, string |
| *=       | Multiply with the existing value | int, float, string |
| -=       | Remove from existing value | int, float |
| /=       | Divide with the existing value | int, float |
| -        | Delete the key             | All      |
| ->       | Rename the key             | All      |

Figure 3.6: Operators available for the multilayer in Mloperator.

IoMLayer

The IoMlayer module is the input and output multilayer module, which loads or writes multilayer from or to files. This module can be loaded with several special parameters which changes file extension and file names etc. The parameters can be found in the iomlayer.cpp file at UPC TALP-research centre.
Mar2ML

The Mar2ml module converts an annotated speech file into a multilayer. A segment marked file is a marked sound file with information about the text that it represents, such as syllable, words, letters, accents etc. By using this module as the first module of an analysis, one can create an multilayer with information taken from a sound file. If succeeded by the following modules one can create marked database segments, which later can be used in the real TTS conversion.

This type of input can also be used when a researcher wants to test a certain module’s functionality, by simply letting the real signal, which by this module has been converted to a multilayer, be processed by the test module and thereafter see if the synthesiser choses the same elements or if the module has affected its characteristics. By using a real speech file as input to the system, module errors can be identified.

Epochs2F0

The Epochs2f0 module adds the fundamental frequency (F0) to the analysed layer from an instant pitch detection marked file. A pitch marked file contains markers for every physical glottal closure that can be measured in a sound file. From these time intervals the fundamental frequency can be calculated and information added to the multilayer.

Wav2Energy

The Wav2energy (wave to energy) module calculates the energy in a sound file and adds the information to the multilayer.

Fea2ML

The Fea2ml (features to multilayer) module is used when new TTS technologies or new modules are developed or implemented to the system. Additional information may have to be add in order to make them work. This module is meant to add this extra information, i.e. features, to the multilayer. For example, spectral information may be interesting to add when distinguishing between vowels.

3.5 Development languages

Since the old interface to the UPC-TTS system has been written mainly in C++ and C, the most logical language option to use was either a high level language such as TCL/TK or JAVA or a more medium-level language as C++. The levels of programming refers to how near to the hardware the programming language is. In low level languages instructions are written thinking directly of the inter-performance with the hardware while high level languages more abstract or conceptual code is written.

The facilities of implementing graphic programming were also taken into account, as most of today’s programs and interfaces include graphic elements. Whilst there are parts of the written system that are still to be extended and improved and since the researchers also were familiar with the C++ programming language, C++ seemed to be the proper language. The low execution time of C++ was also considered as a an advantage. In general, the problem with lower level programming is the amount of code needed to produce the same output as with a high level language. The benefit is control over processes, speed and the ability to take advantage of the hardware architecture. It is also worth noticing that the scripting languages (TCL/TK) has the advantage of very easily being portable to other operating systems.
3.5.1 Requirements

The operating system in the project was Linux (Red Hat 7.3). The standard installed compiler (GCC/G++ 2.96) was used. The proposed requirements for the thesis were portability to other operating systems, and an easy to use interface. Two interfaces: a graphic interface and a basic command line interface, were proposed. Since there were more than one user-interface, two libraries or languages have been chosen: C++ with the Standard Template Library for the command line interface and Fast Light Toolkit (FLTK) for the graphic part of the user interface. The C++ was chosen as mentioned in the introduction due to its speed and the researcher’s familiarity with it. The choice of graphic library is discussed in the next subsection.

Graphic libraries

On the market there exist a variety of different graphic libraries. One encounters an ocean of packages which can easily be implemented on a Linux system Jeng Jou, H. (2004). Many

<table>
<thead>
<tr>
<th>Name</th>
<th>Library size</th>
<th>Portability</th>
<th>Distributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trolltech QT</td>
<td>7644kB</td>
<td>Win32, MacOS, Embedded devices</td>
<td><a href="http://www.trolltech.com">www.trolltech.com</a></td>
</tr>
<tr>
<td>FOX Toolkit</td>
<td>2967kB</td>
<td>Win32, MacOS</td>
<td><a href="http://www.fox-toolkit.org">www.fox-toolkit.org</a></td>
</tr>
<tr>
<td>GTK+/gtkmm</td>
<td>1410kB</td>
<td>Win32, Direct-CB, BeOS</td>
<td><a href="http://www.gtk.org">www.gtk.org</a></td>
</tr>
<tr>
<td>wxWindows</td>
<td>5939kB</td>
<td>Win32, MacOS, OS/2, Embedded devices</td>
<td><a href="http://www.wxwindows.org">www.wxwindows.org</a></td>
</tr>
<tr>
<td>FLTK</td>
<td>724kB</td>
<td>Win32, MacOS</td>
<td><a href="http://www.fltk.org">www.fltk.org</a></td>
</tr>
</tbody>
</table>

Figure 3.7: Some graphic libraries for the X-window system

factors influence the choice of library: licence, speed, simplicity etc. From the point of view of a university, free licenced libraries are always preferred. Normally graphic libraries for the X window system are completely free, but there are exceptions. The libraries under investigation, see Figure 3.7 were all under the GNU LGPL (GNU Lesser General Public Licence) licence when used for a non-commercial purpose. Trolltech had their own licence that is similar to the GNU LGPL under non-commercial circumstances. All the libraries in the table support Linux and UNIX (X-window system). As we can see from table 3.7 there are no larger differences between the compatibilities of the graphical libraries. Thus, other factors affect the choice.

The choice of graphic library, for this project, is made taking into account the size of the library and the executional size of the executable file as this property affects execution speed which may be of interest in text-to-speech conversion. In the table above the smallest

---

\(^7\)All libraries are dynamic libraries and all can be downloaded from their homepages. The versions are the following: Qt 3.03, FOX 1.0.053, GTK 1.2, wxGTK 2.4 and FLTK 1.1. FOX was compiled without 3D support.

\(^8\)From the Free Software Foundation; http://www.gnu.org.
library is the Fast Light Toolkit (FLTK, Fulltick). This library compiles the hello world program on Linux 586 in 114 Kb with a static library, according to the FLTK homepage. The execution file has few execution lines and is therefore fast. For more information on X-window graphical libraries, see Plozer, L. (2003) or the homepages of the libraries.

3.5.2 C++

C++ is a general-purpose programming language extended from C, on a medium level of abstraction with a bias toward system programming. It supports data abstraction, object oriented programming (OOP) and generic programming. It is rather complex and complete. Therefore only the most common characteristics are presented:

- OOP; OOP is the possibility to orientate the programming to objects. It allows the programmer to design applications from a communicative point of view, between objects rather than on a structured sequence of code. In addition, it allows the re-use of code in a more logical and productive way.

- Portability; C++ can practically compile in almost any type of computer and operating system, without almost any changes. C++ is one of the most used and ported to different platforms programming language, thus instruction and help can be found in many places.

- Brevity; Code written in C++ is very short in comparison to other languages, since the use of special characters is preferred before key words, thus saving effort.

- Modular programming; An application’s body in C++ can be made up of several source code files that are compiled separately and then linked together, saving time since it is not needed to recompile the complete application when making a single change, but only the file that contains it. In addition, this characteristic allows to link C++ code with code produced in other languages like Assembler or C.

- C compatibility; Any code written in C can easily be included in a C++ program without any changes.

The C++ language comes with a library of utility functions, as a large part of programming involves mundane tasks such as managing collections of objects, manipulating text strings, searching, sorting and performing formatted input and output. The international C++ standard draft defines a library of general-purpose functions and object descriptions (classes) that simplify such tasks. The standard classes also provide means of data interchange between different third-party libraries. At the core of the C++ Standard Library (STDLIB) is the STL, which is a set of container classes, algorithms and related components designed to be compound in many ways. It is also extensible to programmers. From here on the word class refers to the programmatic meaning of a C++ class, that is an object description.

**Important C++ features**

There are some special features and syntax in the C++ language which may not be of common knowledge and which have been used within the context of the development of the M.A.T. interface. At this point, these properties may seem confusing but their differences are vital for the dynamic loading and for the design of this project. They are described below and can be found in Stroustrup, B. (2000).
• Abstract class: An abstract class is a class from which an instance, i.e. an object, cannot be created. It works as a template for a base class. For example a Vehicle class could be an abstract class to a Bus and a Car class, since busses and cars are vehicles but with certain differences. Therefore a Bus or a Car is not just a Vehicle, they are extended vehicles. Abstract classes merely possess structure and a common base for the inherited classes.

• Function levels: Functions are sections of code that are to be executed in sequence. When classes possess functions, for example the Car class can have a move function, they are associated to the class by different properties. The C++ syntax allows three different function properties for definition purpose:

Member functions are functions defined inside the class scope, that is within the brackets of the class. These functions must be called from an object which is a realisation of the class. They have the encapsulation properties to access all members of that class, i.e. all information. For example: car1.move() where car1 is an realisation of the Car class.

Static member functions must be defined inside the scope of a class but can be called without an object. For example: Car::move(); where the function move can be executed with only the class defined and without an object.

Friend functions are defined outside the scope and can be called without an object and possess the properties of a member function, thus plain C functions can be used to control C++ class contents. For example if the class Car possess the declaration friend move(); another function named move in another file may completely define and use the data stored in the class which has been defined somewhere else (other file).

• Map: A map is a container defined in the STDLIB. It associates (maps) a key with a single value (multi-mapping also exists but is not interesting in this project). For example: the string key mymodule can be associated with a pointer to a memory address where a function is loaded. By using the key, the associated value can be restored.

3.5.3 FLTK

The Fast Light Toolkit is a cross-platform C++ GUI toolkit for UNIX, Microsoft Windows, and MacOS X. It is an improvement of the old Forms GUI for C. FLTK provides modern GUI functionality and supports 3D graphics via OpenGL and its built in GLUT emulation. It contains C++ sub-classing theory and thus incorporates C++ code without compromises. It is considered as a low-level GUI because it is written atop core libraries (Xlib,WIN32 or Carbon). Further features can be found in Sweet, M. et al (2001).

The structure of FLTK

The basic objects in the FLTK is the Fl_Widget and the Fl_Group. These two classes are the hierarchal parents of all the subordinated classes. The Fl_Widget controls single objects while the Fl_Group is the basic container of Fl_Widgets, such as a window or a line browser.

---

9In C++ a function is called member function
10There are other properties such as public, protected and private but they all concern data encapsulation and protection in hierarchal structures.
FLTK uses method name overloading to make short names for get/set methods. A set method is always of the form void name(type) and a get is always type name(void) const. The set method does not call the redraw function which redraws the widget).

Event handling

All FLTK applications are based on a simple event processing model. User actions such as mouse movement, button clicks and keyboard activity generate events that are sent to an application. The application may then ignore the event or respond to the user, typically by redrawing a button in down position when the button is pressed. The events are passed downward in the hierarchic tree and each handle method of each widget returns a value indicating the used event. Events are identified by integer argument passed to the virtual Fl_Widget::handle(int) method. All built in events have a defined name, such as the event FL_RELEASE represents the integer value for when a mouse button is released. The propagation of an event is what a widget indicates, by returning 0 or 1 from the handle method. 0 means that the code inside the widget will not be executed. Thus the event can be send elsewhere. This eliminates the need for event masks or tables, and is probably the main reason FLTK is smaller than other toolkits. Normally, the parent objects are responsible for passing all events to their children.

FLTK also supports idle, timer and file pseudo-events that cause a function to be called when they occur.

- Idle functions are called when no user input is present and no timers or files need to be handled, i.e. when the application is not doing anything. These callbacks are often used to update 3D displays etc.

- Timer functions are called after a specific amount of time has expired. They can be used to pop up a progress dialogue after a certain amount of time or to do other things that need to happen at more or less regular intervals.

- File functions are called when data is ready to be read or written, or when an error condition occurs in a file. They are most often used to monitor network connections for data-driven displays.

FLTK applications must periodically check or wait for events or use the Fl::run() method to enter a standard event processing loop. This function does not stop and return to its caller until all of the windows under FLTK control are closed by the user or by the program.

How to incorporate FLTK programs

FLTK uses a special naming system for all of its components: All public symbols in FLTK start with the characters 'F' and 'L'. The built in functions are either Fl::name() or fl_name(). Its class and type names are capitalised: Fl_Foo. All the constants and enumerations are completely uppercase: FL_FOO. The header files which declares the objects start with <FL/...>.

To write a FLTK program the header file FL/Fl.h has to be included. It defines the starting processes of all FLTK symbols. Then, depending on whether the OpenGL support is needed, one or more of the following libraries have to be included:

- libfltk: The basic widgets and groups in the FLTK GUI.

- libfltk_gl: Support for OpenGL, i.e a 3D graphic rendation system.
• libfltk_forms: The conversion classes to the Forms library.

• libfltk_images: Contains image classes, system icon support and a help dialogue widget.
Chapter 4

User guide to the Multilayer Analysis Toolkit, M.A.T.

This chapter explains the usage of the two interfaces developed in this project to solve the problems with UPC TTS system. However, this discussion is focused on how to use the program and its functions rather than on answering programming questions. It is only to be considered as an introduction to the solutions, i.e. the two following chapters. Questions about the underlying structure are discussed in Chapter 5 and Chapter 6. Here, the overview of the user functions and the user’s possibilities to manipulate the multilayer structure are presented in two sections. An example is given in Appendix B. In the first section the Command-Line interface is introduced with its commands and user functions. Then the Graphic user interface is explained as it is can be based on and run in parallel with the Command-Line interface. The schematics of the processing technique is the same as the one previously introduced in Figure 3.1.

4.1 The Command-Line interface

The Command-Line interface is based on a set of commands\(^1\). The user writes a command (a text string on the command line) and if it is valid, it is executed. Each command executes an order, which either changes, iterates or manipulates components of the multilayer program class (ML_Program). The flowchart is given in Figure 4.1.

This section explains the available commands in the Command-Line interface. The text can also be found in the file of each command. The help files are explained in section 5.1.2. An updated list can be accessed by pressing TAB + TAB inside the program.

Add

Add input files to the input file container and modules to the module container. The modules are not loaded, but they are checked for existence and for valid symbols (in the dynamic symbol-table). One can also add a module to a specific row in the module list. If path is not specified then the value in MODDIR or INDIR is taken as file location.

Example:  
```
add module /home/guest/mod/module1.so
add module /home/guest/mod/module1.so 10
add textfile textfile.txt
```

\(^1\)Explained in section 5
Create a multilayer

Get a user command

If user command exists

Process multilayer and perform tasks

Was exit called?

Error msg

Start

End

Figure 4.1: The flowchart of the interface
Edit

Edit a file or parameter-file with the program defined by the variable EDIT. The standard value for the EDIT-variable is DEFAULT. When default is used, the program tries to launch EMACS to read the context of the file. Edit can also be used to modify the parameters.

Example: edit my_interesting_file.txt
         edit parameters

Help

Give additional information concerning the commands by displaying the help file associated with the command. The help files are looked for in the directory specified by the help path variable. If not specified then the actual directory is searched. The help files are named using the following syntax: f_commandname.hlp

List

List files, variables, modules and already written mat programs. Prints a short message on the screen i.e. which files, modules and variables are in the system and their state. It is also possible to show the context of a multilayer program (M.A.T). When viewing a M.A.T program the contents of the EDIT-variable is used to display the file.

Example: list files
         list variables
         list modules
         list program mymlprogram

Load

Load an existing M.A.T program or modules to the M.A.T-interface. When a program is loaded, the program running is not saved. Modules are loaded into memory dynamically to the heap.

Example: load program myprogg
         load module module1
         load modules

Mlayer2

Convert the multilayer into different file formats. The file format is specified as the option. These converter functions have also been written as dynamically loadable modules. Each converter function normally has a default file-output if none is specified. For information concerning the modules, such as which parameters that can be used, their code files should be checked. The special information on how to use an Info file can be found in reference Info (2004).

Example:
mlayer2 html2.0 my2_0.html ---> my2_0.html contains HTML 2.0 tags
mlayer2 html3.2 ---> constructs per default mlayer_3_2.html
mlayer2 texinfo my.info ---> info file ready to be processed by the GNU makeinfo tool
New
Create a new M.A.T program. Erases the running program without saving it.

Example: new program /home/mymprogg

Quit
Terminate program.

Example: quit

Remove
Remove files or modules from program. It simply removes the module from the program container. Files are treated as modules, i.e. they are simply removed. Since files are only passed as arguments to the SPM-modules, they are never loaded.

Example: remove file myfile

Run
Run a complete program or a specific module. This will only execute the modules which are loaded in memory, independently of which modules are present in the list of modules.

Example: run
        run module mymod

Save
Write the actual program to a file with the program name. If specified with the "as" option, the mlprogram is written to the character sequence after that option, in a forced overwrite mode.

Example: save
        save as /home/mypromm

Set
Set environmental variables. These variables are stored in a global container in the M.A.T. interface. At loading time the variables are read from the program file first as key from string and then as value from string. For the standard variables only certain values are valid. If the value is unknown then functions will proceed as with a default value. Users may add new variables in the program file and then externally refer to them by the C++
extern call.

Example: set <variable> <value>
Show

Show layer info. In the non-graphic mode, the layer is simply printed to the standard output (normally screen) to a short overview of the contents of the multilayer. In the graphic mode and when the show button is pressed, the different layers are shown according to the variables declared in M.A.T. interface. For instance HTML ON will give a layer converted into HTML 2.0 code. More type conversions of the representations will be available in future versions.

Example: show

Step

Step through a sequence of modules. For each inputfile a module or a sequence of modules are executed. If a specific module is to be processed its corresponding identification number must be written after step, see below. It is also possible to process specified sequences of modules with the ‘-’ operator. After each module execution a question is asked if the user wants to continue executing the next module.

Example: step
step 1-4
step 1,3,4

System

Make a system call to the background shell where the program was executed. External programs may be launched within the M.A.T. structure. However, some text list programs that use scrolling facilities may not scroll properly. In those cases the M.A.T. program process should be paused and the system call should be made in the shell. In Linux the M.A.T. process can be temporarily stopped by the macro CTRL-Z and the re-started by the fg %job command.

Example: system call thing_to_call
! thing_to_call

Unload

Unload specific module or all modules from the free store. This does not remove them from the program list shown by the list modules command.

Example: unload modules
unload module module1

4.2 The Graphic User Interface

The Graphic user interface is the front face to the multilayer structure that the user specifies with the GRAPH_ON option at start-up. It is the second interface implemented to simplify the usage of the UPC TTS. Its main objective is to simplify the use of the multilayer structure by means of clear and simple graphic objects, widgets. The simplification is realised by a graphic menu, see Figure 4.2, which contains the most common commands and functions from the MI_Program class. These menu-functions resolve the proposed use-cases described in 6.1.2. The functions and their usage are specified below.

33
The first menu item is the **File** menu, the second the **TTS** menu and the last the **Help** menu. Below, each specification includes a screenshot of the actual function. The menu items are accessible via mouse clicks, shortcut key sequences or by typing the underline letter in their label text.

### 4.2.1 The File Menu

![File menu options](image)

The **File** menu contains four different sub-menu items: **New**, **Open**, **Save** and **Save as**, see Figure 4.3.

**The New item**

The New item creates a new program. It creates an empty file with a user-chosen name and adds the default variables (explained in 6.1) to the file. Creating a new program automatically revokes the program that is running, and deletes it from memory. If the
running program is not saved, the settings changed by the user will be lost. When the mouse button is released over the sub-menu item New the menu in Figure 4.4 is shown on the screen. This window contains seven different widgets:

- The Show field displays the different filters that can be used to view files and directories. Normally this filter sequence is set to asterisk (showing all files and directories), but can be manipulated by the programmer to any sequence of characters. A custom filter can also be created by choosing the Custom filter option inside the filter list (press the down-arrow).

- The Favourites are to be used as storage for frequently used search paths or files, from which the user wish to launch, create or save the program. They can be manipulated in the same way as favourites in an Internet browser.

- The button showing a folder with a star, in the upper right corner, creates a new folder in the current directory.

- The Display window shows all the files filtered by the predetermined filter. Fields are pure text objects and are automatically chosen with a double click.

- The Preview window, marked by a ? in Figure 4.4, gives, if the preview box has been ticked, a first view of the chosen file. This view treats the file as a pure text file, and does not do any interpretation of any standardised code such as XML, HTML etc.

- The File name field contains the absolute file path to the actual object. The field supports file completion.

- The two buttons Ok and Close confirms, respectively cancels the creation of the new program. However, the Ok button contains a return button symbol, which indicates that by pressing the return key on the keyboard the button will be executed. This is a standard feature implemented throughout the M.A.T.-interface.

The other sub-menu items, that is Open, Save and Save as all open similar windows with similar functions. Therefore their screenshots have been placed in the appendix C Figures C.1 and C.2. The save function does not show a window since it only overwrites and updates the program file.

4.2.2 TTS menu

The Text-to-speech menu, Figure 4.5, contains all functions that are related to the text-to-speech conversion. These items are: Run, Modules, Textfiles, Edit parameters, Show layer, Wave analysis and Parameters. Each one is explained below.

Run

The Run item executes the run-command found in the Command-Line interface. Thus, it processes all loaded modules on each input file. For example, if the run command was to be executed with the module configuration in Figure 4.6, i.e. three modules, the first and second modules would process all textfiles. The third module would not be processed since it is not loaded, (it does not have a small circle infront of the Sable text text).
Figure 4.5: The Text-To-Speech menu options

Figure 4.6: The Module window
Modules

The Module window shows all the existing modules which have been added to the program and their status. The window contains a browser of modules, check boxes, status flags and operator buttons.

The browser is a check box browser which displays each module as a line. There is also, on each line, a check box which indicates if this module is to be processed by the operator buttons. Checked values will be processed. The loading is indicated by a flag (a green circle) in front of the name of the module. If the flag is visible then the module is loaded.

The operator buttons are load/load all and unload/unload all modules, check/unchck all modules. The buttons are combinations of the check all and load/unload operator buttons. The Add button allows the user to add an external module from a file chooser window similar to the window found in Figure 4.4 (The exact display can be found in appendix C.3). The module is inserted before the latest clicked line in the browser. One has to bear in mind that a module is loaded into memory only one time. The remove button eliminates the last clicked module from the module list. If it was loaded, it is unloaded and then removed. The Module browser also contains a pop-up menu with special functions.

![Module Browser](image)

Figure 4.7: The pop-up menu in the module browser

It pops up when the user right clicks with the mouse over a module line in the browser. The menu is shown in Figure 4.7. This pop-up menu contains three items: Parameters, Properties and Run. The Properties are not implemented in this version.

![Module Parameters](image)

Figure 4.8: The parameters associated the mloperator module
• The Run option, see Figure 4.7, processes the module that corresponds to the clicked line on all input files.

• The Parameter item displays a window, Figure 4.8, with all parameters related to the chosen module. These module parameters can be changed via a click with the left mouse button on the desired parameter. The change window is shown in Figure 4.9. It contains two fields: the first with the old parameter value and the second with the parameter value which is to be used as new module parameter. The two fields are present since they support drag and drop which simplifies the recovery of values, if the user accidentally erases the new parameter value.

![Parameter Window](image)

Figure 4.9: The window to change parameter values

Text files

The Textfile window is, as the module window in Figure 4.6, a list of text lines with the names of the textfiles included in the project. These textfiles can be added and removed in the same manner as the modules introduced above, by the two operator buttons at the bottom of the window. The view button launches the editor defined in the EDIT variable on the marked textfile. This allows the user to modify the textfiles in a simple manner in case of orthographic or semantic errors.

Edit parameters

The Edit parameters item edits the parameters associated with each module. First, a file is created with all parameters from all modules included in the project. Second, the editor defined in the EDIT variable is launched on the created parameter file, thus allowing the user to manipulate all parameters in a text editor form. Afterwards, when the editor has been closed, the parameter-file is read and the associated parameters are updated with their corresponding module. This parameter-file has a predetermined name and overwrites any existing file with the same name, without asking.

Show layer

The Show layer item displays the context of a multilayer, normally after the execution of a sequence of modules. The type of analysis tool is set in the Preference window (see below). At the moment there are three different options to choose from:
• Two HTML-conversions, one with a built-in viewer and a second with an external viewer defined by the variable WEB. The results will then be shown as ordinary Internet files with possibilities to move between related links in the multilayer.

• The multilayer can also be converted to a plain text file, thus maintaining the possibility to view the results as they were shown in the old interface.

Wave analysis

The Wave analyser item executes the command stored in the WAVECOM variable. In this variable a user may add a program that reads a predetermined wave file and then launches a graphic analysis tool to allow the user to manipulate the wave characteristics of the output file. However, any program or command could be defined as its value. The variable has not been given a default value since many different wave file analysers are used in TTS technology.

Preferences

![Preferences window](image)

Figure 4.10: The Preferences window with the general-tab active.

The Preferences item contains all settings available in the graphic interface. There are three different tabs: General, Module and Wave-preferences.

• The General preferences, shown in Figure 4.10, contain fields for manipulation of input directory, output directory, module directory and check boxes which determine the different manner to visualise the multilayer data, section 4.2.2.

• The Module preferences, Figure 4.11, contain the settings path which is the path to the file that determines how modules should be started when they are loaded into memory.

• The Wave preferences, Figure 4.12, contain the wave analyser program with its startup syntax.

39
Figure 4.11: The Preferences window with the module-tab active.

Figure 4.12: The Preferences window with the wave-tab active.
4.2.3 Help menu

The Help menu, Figure 4.13, contains help tools which can be used to find information about implemented commands and answers to software related questions. There are three menu items: Manual, Command Help and About.

![Help menu](image)

Figure 4.13: The Help menu.

Manual

This Manual item contains the complete user manual for the M.A.T.-interface. This manual is Chapter 4 of the thesis report, in HTML format, thus allowing the user to move between different sections and parts of the manual.

Command Help

The Command help item displays a complete list of all the commands added to the interface. These commands are shown with an internal HTML-viewer included in the FLTK libraries which allows the user to easily browse the commands. The information about each command is collected from the information file associated with the command, i.e. the f-commandname file.

About

This window contains information about developers and where updated information can be found.
Chapter 5

The implementation of the Command-Line Interface

As mentioned in section 1.1, the original system lacked a satisfactory way to write commands and to reload program applications. To simplify the usage of the UPC-TTS system an interactive command line interface has been implemented. This section will give information about the Command-Line interface. Command line interfaces can be found in many terminal-systems; for example Bash-shell, Csh etc. The developed system contains command and option completion, related to the multilayer structure under Linux/UNIX. This functionality is based on the GNU-readline libraries and components Fox, B. (2001). Variable resolution has also been implemented. Shell-defined environmental variables, indicated at the command line prompt by the $ sign followed by the variable name in uppercase, are replaced by their internal significance. Direct connections to the background shell has also been added.

The interface has been written as a C++ class, Multilayer Command-Line Interface (Ml_Cli), and can therefore be extended and implemented in other codes. How to extend an instance of the Ml_Cli class with new commands are discussed below in the section 5.2.

5.1 Structure

The structure of Ml_Cli is basically a prompt in a terminal window with a pre-analysed input. Depending on the input, different commands are executed.

Generally, a command (in any command line system) is a program which is placed in a certain directory (such as /bin on UNIX systems) or with its search path added to a common known global system variable such as PATH on Windows systems. The program parses an input stream (reads the argument) and executes according to the code within the program. When the operating system receives a call from the associated input-stream (normally keyboard) it searches through the known directories specified by system variables until it finds the command file. This is normally a linear search in the same sense as searching (iterating) through a vector or searching for keys in a map structure. Thus a simple way of storing commands is in a plain vector.

Each command generally consists of basic help information, a function that actually solves problems (worker-function) and a more complete help file that describes its function. These three attributes have been implemented in the Multilayer Command class (Ml_Command). The class structure and its hierarchical relations with their most impor-
tant attributes and operations are displayed in Figure 5.1.

The Mi_Cli has some internal environment variables which are loaded at the start. The standard variables are the following:

- **HELP:** This variable contains the directory where all the help files are stored.
- **EDIT:** The Edit variable contains the program-call used to view files, when the module parameters are analysed in both graphical and none graphical mode. If this value is set to *DEFAULT*, Emacs is called.
- **LOG:** This flag tells the program to write all the debugging information to a text log file. When this is set to ON, log is written.
- **VERBOISE:** The verbose flag increases the output from the program to the standard output (normally the screen), with debugging information. When this is set to ON, text is verbose. In any other case the verbose mode is ignored.
- **MAKEINFO:** This variable contains the program that converts a texinfo file to an info file. With the DEFAULT value, the GNU makeinfo tools are used.

The standard variables should normally be given values by the user. If not, the program uses default values which then make references to objects according to the background shell’s interpretation of that string.

![Diagram of class relations for Mi_Cli](image)

Figure 5.1: General class relations for the Mi_Cli

### 5.1.1 GNU-readline implementation

The filename completion, command completion and option completion have been implemented under Linux and UNIX to facilitate the usage of the interfaces. This functionality
is based on the GNU-readline library and its extensions including the history link. The readline input library works according to key-bindings in the file defined by the inputrc-variable and is normally set to the ~/.inputrc file. To change the behaviour of the key bindings in the M.A.T.-interface the section MAT has to be added in the local rc-file. Then key bindings can be added according to the GNU Readline manual Fox, B. (2001). For example, this will change the completion function from TAB to the CTRL-z key:

```plaintext
# These are comments
# inside the local inputrc-file

# include global standard settings.
\$include /etc/inputrc

# load a standard behaviour
set editing-mode emacs

\$if MAT
"\C-z": complete
\$endif
```

Most common is the use of the standard key binding sequences, i.e. using the TAB key as completion key. The completion in the M.A.T. interface is first made by linear iteration of the command-vector. Secondly, iteration on the choosen commands option vector is made, if the first word has been completed. If a command does not contain options filename completion is automatically performed. A single blank space is considered as separation between words, thus separating command from option. As we have seen in Figure 5.1 the Ml_Command class has a one to many relation with the Ml_Option class. This indicates that a single command can have many options. Each option has some properties, whereof two are presented in the figure: name and func. The func is the unique working function that is called to perform the execution of the program. By changing only this pointer’s address, i.e. associating another function to the pointer, new functionalities and code can be implemented.

### 5.1.2 The Help-file system

A help system has been implemented to simplify the learning process for a new user. It is based on text files normally found in the directory /PATH/help or the directory defined by the help variable. Each command when initialised is given a reference to a help file (a normal text file with information) named f_commandname.hlp. When the user calls for help on a command, the help file associated with that command is shown on the screen. These help files should all be placed in the same directory and the variable help-path should be set to that absolute directory path.

### 5.1.3 Use cases

The interaction between the user (researcher) and the TTS systems data structure (operations, objects, instances etc.), that is use-cases, is displayed in Figure 5.2 for a quick overview of the necessary commands and basic extend functions. Since there are two parts of the interface; one part that is expandable by the programmer and a second part which is intended for the researcher, two actors has been added in the figure. As observed in the figure, most of the use-cases are of command type, i.e. they are implemented as commands.
Figure 5.2: General use-cases in the Mi_Cli interface
A complete reference list of the commands can be found among the \textit{private static} functions for the \texttt{MI\_Cli}-class in the programming manual on the CD or by a double completion at runtime (\texttt{TAB + TAB}).

5.2 How to extend

The \texttt{MI\_Cli}-class can be extended in many ways. However, one has to recognise that the class is made to work with the multilayer-structure only. Since the \texttt{MI\_Command}-class is not dependent on a class dependent \textit{working function}, one may easily add a new command without having to change the \texttt{MI\_Cli}-class. This will probably be the most common way to extend. For example, to add a new command called \texttt{com} with two options to the \texttt{MI\_Cli} class, the following lines should be copied:

\begin{verbatim}
//the vector with the associated options and their work functions
vector<MI\_Option> opt;

opt.insert(opt.end(),*(new MI\_Option(new string("first-option"),
&option1)));
opt.insert(opt.end(),*(new MI\_Option(new string("second-option"),
&option2)));
MI\_Command com1;
com1=new MI\_Command((new string("com"),new string("thehelpfile"),opt));

//we add it to the MI\_Cli-class by
myMI\_Cli\_instance->mi\_add\_command(com1);
\end{verbatim}

The memory allocated by the new command will automatically be freed when the \texttt{MI\_Cli}-class destructor is called.

One may use the environmental variables defined in the map \textit{variables}, as well as defined by the proper shell, when is in need of reading and manipulating interface states and in/output file locations etc. The \textit{variables} map is a global map container with two string types. It can be included and accessed in any file by this code:

\begin{verbatim}
#include <map>
extern map<string,string>variables;
\end{verbatim}
Chapter 6

The implementation of the Graphic User Interface

The graphic user interface relies on a reference to a Ml Cli instance and the FLTK library’s built-in functionality. The interface contains some basic functions to manipulate the multilayer, configure speech processing modules, make basic outputs of layers etc. all implemented in a graphic environment with widgets. Below, its structure and interface variables are explained.

6.1 Structure of the Multilayer Graphic User Interface

The Multilayer Graphic User Interface (Ml Gui) class possesses the same structure as the Ml Cli-class, that is it wraps an instance of an Ml Program object with an interface. However, it does depend on the Ml Cli-class since some existing functions inside the Ml Cli-class are called through the Ml Gui class. The reason for this nested structure is to avoid repetition of code. The dependency on the Command-Line interface is not a negative feature since the combination of the two interfaces sometimes simplifies the usability. Furthermore, they are launched as two independent threads that allow the user to execute commands in parallel.

The Graphic User interface also possesses standard variables that are different from those introduced in section 5.1.

- **HTML**: The HTML variable indicates if the multilayer shall be shown in HTML in the graphical mode, when the show command has been executed. To show the text **ON** shall be written after the HTML-value.

- **TEXT**: The TEXT variable does the same as the HTML variable but the multilayer is converted to a normal textfile and displayed without any link properties.

- **WEB**: The WEB variable indicates if an external Internet browser is to be used as the viewer of the HTML documents (version 3.2). It should be set to **ON** if this is the intention.

- **WEBREADER**: The web variable determines the program that is to be launched when the WEB variable has the value **ON**. This could be any type of program.

There are three variables that are not yet used by the interface. These variables are TEXTEDIT, Tree and XML. The Tree variable creates a tree structure from the contents
of a multilayer if it is set to ON. However, the tree structure has not been fully implemented and does not work in this version. The XML variable has not been developed since there were problems in the conversion from the mlayer to XML tags and the TEXTEDIT was used to let the user decide if the parameters of all modules were to be manipulated by an internal text editor or by EMACS.

### 6.1.1 Help system and the manual

The Help system under the graphic environment is based on the same file system as the Command-Line interface, introduced in 5.1.2. The information gathered by the Command-Line interface is converted to HTML code, since the FLTK graphic library has a built-in HTML viewer and this allows the user to click through the help information in an Internet browser fashion.

The user manual from this thesis report (Chapter 4) has also been included within the graphical user interface. The manual has been implemented using the FLTK HTML viewer and the conversion of a pdf-document to HTML.

### 6.1.2 Use cases

![Diagram](image.png)

**Figure 6.1: General use-cases in the M1_Gui interface**

The use-cases for the M1_Gui are rather similar to the use-cases presented in 5.2, but are fewer since the usage of the M1_Gui class implies the usage of the Command-Line
interface. The most important use cases are given in Figure 6.1.
Chapter 7

Dynamically loadable modules

The speech processing modules that existed at UPC at the start of the project, had to be compiled with the main program, as mentioned in the introduction. Thus, a change of the module configuration implied a complete new linking of the entire program. This of course becomes unacceptable when the main program reaches a certain size or when many researches develop different modules and compile them with the main program as incompatibilities may occur between different versions of the main program on different computers. To solve this problem dynamic compilation, linking and identification of modules have been included as new system features. This chapter also explains how the two new modules, ml2html and ml2texinfo, have been constructed and in what manner their functionality support the user.

To understand how to build these module plugins some basic understanding of how compilers work is needed. Therefore a brief introduction is given to the important compilation steps that are used with these plugins. Then the structure of the modules are introduced.

7.1 Compilation theory

This section will briefly introduce the important compiling concepts that are significant in the development of plugins (dynamic loadable code) in the C language environment. The compilation process is composed of four major stages: preprocessing, compilation, assembly and linking.

Preprocessing

The first stage of the compilation process is to use the preprocessor to expand macros and include header files into the proper code files. The resulting file is an extended version of the original file with the same syntax.

Compilation

The second stage is the compilation process which is the conversion from the high-level language (C++, Java etc.) source code to the assembly language, for a specific processor. Assembly language normally contains fewer and simpler instructions than the original language. Thus this stage can be treated as a transcription, from high-level to low-level language.
Assembly

The purpose of the assembler is to convert assembly language into machine code and to generated an object file. An object file is basically a file containing machine language instructions and data in a form which the linker can use to create an executable program. Each routine or data item defined in an object file has a corresponding symbol name by which it is referenced. A symbol generated for a routine or data definition can either be a local definition or a global definition. Any reference to a symbol outside the object file is known as an external reference. When there are calls to external functions in the assembly source file, the assembler leaves the addresses of the external functions undefined, to be filled in later by the linker. Sometimes the compiler program creates symbol tables within the object files. They are tables that store the location (or offset for field independent code) of functions or variables by name. Normally they include a compiler flag that indicates the status of the symbol, for example the GCC uses U to indicate undefined symbol.

Linking

The final stage of compilation is the linking of object files to create an executable. In practise, an executable file requires many external functions from system and program libraries. Consequently, the compiler programs (GCC, KCC, Borland etc.) internally possess complicated commands which link system functions. However, the underlying concept is to construct a complete sequence of machine code by either copying the linked object files into the same compiled file or identify the undefined symbols in such a manner that the loader program associated with the compiler can identify the object file at run-time and make a reference to that code by means of dynamic loading. The first approach is called static linking and the second dynamic linking. The static linking includes the referenced object file within the executable file. This increments the size of the executable and implies the two files to be linked together. On the contrary, dynamic linking extends the information in the executable so that the referenced object file can be linked during run-time by a loader program. This diminishes code size of the program and allows multiple use of a single library (shared libraries) without code copying. However, these shared libraries have to be defined among system variables or system directories so that they can be found.

7.2 Data structure of the Modules

The objective has been to make the modules as stand-alone as possible, without any references\(^1\) in the main program. By wrapping the original speech processing module with an extended module structure containing several new instances such as file type indicator, path indicator etc. the dependency on the main program diminishes. The extended module class is called \textit{Ml Module}.

Since loader programs (the programs that place the code in the memory) normally return pointers to the dynamically loaded object and to internal identifiers, each \textit{Ml Module} object has been given two pointers. One pointer identifies the loaded object to the loader program (known as a handle), such as a void pointer (\texttt{void*}) for the \texttt{ld} loader under Linux. A second pointer that identifies the address of the symbol that the program is interested in. Since the address to the object has to be resolved by an identifier known to the pro-

\(^1\)With references I mean having explicit cases of modules named in the main program code. For example: \texttt{if (module=="MYLIB")}...
gram, each module must contain the same identification symbol. However, this cannot be an ordinary member function (3.5.2) since the C++ coding system wraps class members with class identification symbols. Therefore each module class has an external C-defined identification symbol, \textit{launcher}, within its code file. Thus, the program knows that it has to look for a certain symbol (the \textit{launcher}-function) and that the returned void pointer corresponds to a certain structure. In our case the launcher-function is a function that returns a pointer to a downcasted C++ \textit{new}-allocated object of the abstract class \textit{mlModule}. Each function of interest, i.e. the call to process the multilayer, is declared virtual to force the implementation of these functions in each speech processing module. To get hold of the new module pointer a hard cast is made since the C++ standard prohibits some type conversions. The void pointer is converted to an unsigned long integer (normally 32bit) and then converted into a pointer to function (the launcher) symbol.

7.2.1 The \textit{Ml} Module class

The most important attributes of the Module class (\textit{Ml Module}) are shown in Figure 7.1. The \textit{Ml Module} class contains functions for loading/unloading a library into/out of the memory. It also makes the pointer conversion from the void pointer discussed in 7.2 returned by the \textit{dlopen} program Youngdale, E. (2000).

There are several criteria which the modules have to fulfil before they can be loaded. First the path to the dynamic loadable code must be added to the loader path. Under Linux, with the \textit{dlopen} loader this is normally specified by the \textit{LD_LIBRARY_PATH} variable. Second, the created module class must contain an external C-declared function named launcher that returns a new pointer to the class. This is fulfilled by the following code:

\begin{verbatim}
// at the end of myModule.cpp
extern "C" {mlModule* launcher(mlKeyValue& set){return new myModule(set);}}
\end{verbatim}

Another aspect of loading dynamic code or classes into memory is the self-registering objects introduced by Beveridge, J. (1998). Self-registering objects is an easy and comfortable theory of letting the module designers do the their proper identification by adding a module identifier to a global container. However, the drawback of this technique is the required global map container (factory) which in fact is a global variable. Using global

\textsuperscript{2}Symbols refer to functions or variables described as addresses in the memory as discussed in 7.1

\textsuperscript{3}Hard cast is a casting that cannot be controlled by compiler at compilation.
variables always has a slight drawback, since it complicates maintenance and consistency of the program. Due to the map container used by these factories multiple modules with identical keys cannot be loaded. In the M.A.T. interface a factory has not been implemented, but it would have been an appealing option. One problem in the M.A.T. interface was finding an identification key for each module since they were not self-registering and thus the program user had to identify the modules. However, when the user assigns a name, by writing a filename to load, that input is used as the identification key.

The sequential importance of module loading was also taken into account as the modules are to be loaded in the same sequential order as they are chosen. Thus, a map cannot be the container of \texttt{ML\_Module} objects since an iteration on a map returns the modules in an alphabetical order. Thus, a normal vector has been implemented.

### 7.2.2 The mlModule class

The \texttt{mlModule} class is the abstract base class for speech processing modules connected to the multilayer structure. As explained in 3.5.2 the abstract classes possess a structure to generate inherited classes. Hence, each instance of a speech processing module inherits the structure of the \texttt{mlModule} class.

Each speech processing module is initialised at creation time with a set of settings (type mlKeyValue) to indicate special parameters. The settings are entered by the user and stored in a file which is specified by the variable \texttt{SETTINGSPATH} within the interfaces. All settings are passed to each module at startup, and the module only reads the ones that are relevant. The content of the module is normally a set of C or C++ functions which add, manipulate or remove information from the current multilayer.

At runtime, when the code has been added to the memory, parameters (type mlKeyValue) are passed as arguments to the module just before execution. These parameters can be manipulated by the user, in order to change the behaviour of each module or to change input files etc., thus potentially changing the values inside the multilayer.

Furthermore, as some modules need the same parameters as previous modules, all parameters introduced in the system are passed to each executed module. Thus, the parameter list will be extend each time a new module defines new parameters. This is not ideal, since each module only reads the parameters that it needs. Thus, simplicity can be gained by eliminating the superfluous parameters. However, the development of such solutions is not included in this thesis.

### 7.3 Linux implementation of the module plugins

On a Linux machine with the GCC compiler, as in the environment at UPC, the dynamic loading is made by the C-written loader \texttt{dlopen}. There are three different flags to indicate the state of loading:

- \texttt{RTDL\_LAZY} resolves undefined symbols from the dynamic library as code is executed.
- \texttt{RTDL\_NOW} resolves all symbols before \texttt{dlopen} returns.
- \texttt{RTDL\_GLOBAL} takes the external symbols defined in the library and makes them available to subsequently loaded libraries.
The RTDL_LAZY is first used when the user tries to add a module to the program. The module is checked to control if it is a valid loadable library. Later, when the module is loaded into memory, the RTDL_NOW flag is passed as argument.

The compiler flags must also be taken into account, both during the compilation and during linking as without some flags, the dynamic loading cannot be accomplished. With GCC 2.96 compiling a dynamic module library would need the following syntax:

```sh
g++ -c -Iall_include_dir -fPIC mymodule.cpp -o mymodule.o
g++ -Llib_dirs mymodule.o -llibs -shared -Wl,-t,-E,-soname,mymodule.so,-rpath, /home/mylib/ -o mymodule.so
```

- `-c`: Do not make the linking.
- `-I`: Specifies the name of the directory where the included files (.h-files) are located. This can be specified multiple times.
- `-L`: Specifies the name of the directory where the library files (.so, .dll etc.) are located. As the `-I` option this can be specified several times to include different search paths for the linker.
- `-l`: This option specifies a specific library within all the directories added by the `-L` option. Each library file is normally called `lib*.so` where the asterisks should be taken as name and extension of the library.
- `-fPIC`: Compiles position independent code.
- `-o`: The next word is taken as the name of the output file generated by the compiler.
- `-shared`: Make a shared object.
- `-Wl`: Pass the following comma-separated options directly to linker.
- `-t`: Trace the filenames processed by the linker.
- `-E`: Export dynamic (ld-linker option). Adds all symbols to the dynamic table.
- `-soname`: Tell the linker to internally name the shared file by the following comma separated text. When a library is loaded the linker associates the library with the symbol `soname` which is indicated inside the library as one of its first elements.
- `-rpath`: Put the path (/home/mylib) of the shared object into the local variable that specifies where to find dynamic libraries. On an ELF system this is `LD_RUN_PATH`. On other Linux system this is normally `LD_LIBRARY_PATH`.

In Appendix D.1 a make script (for the GNU-maketool) is presented. It converts all the files containing the word `launcher` with the extension `.cpp` in the directory of the script, to a shared object file, using the same name as the code file. The extension is `.so`.

### 7.4 How to extend

How to create and add a new module is explained by the following example. It shows how to create a module, called `Mi2File`, which writes the multilayer to a specific file named defined in `OUTFILE`, i.e. `layer.txt`.
// inside the M2File.cpp file

if (defined ML2FILE)
    define ML2FILE

#include "mlmodule.h"
#include <iostream>

define OUTFILE layer.txt
using namespace std;

class M2File: public mlModule
{
public:
    M2File(): mlModule(){}
    M2File(mlKeyValue &x): mlModule(x) {}
private:
    int process(mlMLayer &ml, mlLayer::iterator begin, mlLayer::iterator end, const mlKeyValue *options, mlKeyValue * parameters) const;
    ofstream f;
    f.open(OUTFILE);
    if (f.good())
        f<<ml;
    else return -1;
    f.close();
    return 0;
};

extern "C" {mlModule* launcher(mlKeyValue & set){return new M2File(set);}}
#endif

The user only has to compile the code with the syntax given in 7.3 (for Linux) and then simply add the module to the program in the same way as any other module.

### 7.5 Data monitoring modules

A further problem with the UPC TTS system was the difficulty to analyse the data information from a TTS conversion if the user had observed errors, see 1.2. To solve this problem two new modules have been implemented, ml2html and ml2texinfo.

#### 7.5.1 Multilayer to HTML module, ml2html

This module converts the multilayer to a HTML file that can be read by any browser or web tool. The reason for this conversion is that in the original program the multilayer could only be converted to a text file, which did not possess any properties for moving between nodes (linking). Thus, as HTML code possessed this property, and the fact that HTML code version 2.0 could be displayed within the FLTK library, it was implemented. However, due to the simplicity of HTML 2.0 code, colours could not be implemented. Therefore, an optional conversion to HTML code version 3.2, where the colour tag exists,
is available since there existed a need to indicate if two following nodes had one or more mlValues with equal value (a specific demand from one of the researchers).

The implementation of the HTML conversion is a straight syntax conversion from each node in the multilayer to a header in the HTML code. As HTML code constructs its links by names, each node is given a unique name (the layer name and the number of the node) to identify that node. Thus each node can be referred to or linked to by any other node via its name. This allows the construction of a internally linked HTML file, which the user can use to see the values associated with the node by a click on its links.

7.5.2 Multilayer to TeX-Info module, ml2texinfo

The module ml2texinfo converts the multilayer to a Tex Info file, which is a type-setting file with link support that can be used in non graphic environments, for example client-server relations. However, the info program used to view this file must be of version 4.1 or newer.

The Tex info files are constructed with a long chain of nodes, which in our case are the nodes of all layers connected in a sequence. The last node of a layer is connected to the first node of the following layer etc. By using the node identifications system introduced in the HTML module (section 7.5.1), each node can be linked to the following node and the former node. This allows the user to navigate through all nodes by simple clicks on predetermined keys (P)revious, (N)ext and (U)p. As each node in the Tex Info files can contain text information and links in an non-sequencial order, it was possible to maintain the structure of the HTML code in a manner that each value and each link of a multilayer-node can be shown within the same window.
Chapter 8

Conclusions

This thesis has described the evolution of two new user interfaces for the UPC TTS system based on the C++ and FLTK programming languages. The interfaces have been developed using experiences and suggestions for improvements gained in a case study of the UPC TTS system.

The case study surveyed the functionality of the old TTS system. Following that, different data structures, such as the multilayer, mlayer, mlnode etc. were introduced in order to facilitate the development of the interfaces. Their usage was interpreted in a natural and logic manner as their structure possessed properties that aids the user’s interpretation of the speech analysis. The general text-to-speech conversion steps in the concatenative TTS system at UPC were all analysed and related to different processing steps (speech processing modules). These speech processing modules have been converted to a dynamic code, making them independent from the main program and thus removing them from the main compiling code; i.e. they were introduced as plugins. To be able to implement these new module structures, the modules from the old system have been rewritten and extended with a launcher function that dynamically creates module objects by down-casting an abstract module class, without using the module name inside the code. As a result of the independence of the modules, a reduction of system code size and a simplified maintenance structure have been obtained.

Furthermore, the first interface was specially designed for the non graphical mode when users work on a computer that does not support the X-window manager. This normally occurs under server client conditions as when a researcher wants to work on a server from his/her house. For this usage the GNU readline library was implemented with its command completion and its history links. By implementing this library a dependency to Linux/UNIX is created as this library has not yet been developed and implemented in Windows. However, this can be solved by implementing a Linux loader for Windows and run Linux under Windows. The readline command completion has also been extended to complete the options of a command, which further facilitates the use of the interface. After a TTS conversion the results can be converted to a normal text file, HTML documents or the TeX Info system as this allows the results to be monitored in a simple manner.

Finally, the second interface supports usage with the X-window manager and the FLTK library. This graphic library contains different browsers to view modules, input files, results, adjust preferences etc. Since the library supports a built in HTML viewer the results from a TTS conversion can be converted into an HTML file and viewed within the program. Variables can also be defined to link to other external viewer programs.

The benefits of these improvements are: First, the added interactivity allows the user to choose, process and manipulate speech processing modules in a simple manner. Second,
the structure of the entire TTS system is easier to maintain and to develope. Third and last, the data information created in a conversation from text to speech is clearer and more straightforward for an inexperienced user.

Apart from the implementations above, ideas have emerged how to make this program even more useful. However, these ideas have not been realised since there were not enough time. The ideas were the following:

- Make a Man Machine Interaction (MMI) questionary to see what functionalities the users lack.
- Construct a module that converts the multilayer into a graphic tree.
- Include a client-server relation between the main program and the users.
- Create drag and drop support for the Module window.
Bibliography


Wagner, T. & Towsley, D. (1995) *Getting Started with POSIX Threads* Department of Computer Science, University of Massachusetts at Amherst, July 19
Appendix A

A basic startup program

This appendix contains a basic program example, which specifies all the variables used in the M.A.T. interface. First the code is presented then each indicator is explained.

This program uses the speech processing modules MLOperator and sable_text and is written as a plain textfile.

#TXTFILES
/home/bjorn/pfc/prj/Tts/src/cli/in/ejemplo1.txt
#END
/home/bjorn/pfc/prj/Tts/src/cli/in/ejemplo2.txt

#MODULES
/home/bjorn/pfc/lib/debug/mod/mloperator.so LAYER="CHUNK" OP="TEST1 = testing"
/home/bjorn/pfc/lib/debug/mod/sable_text.so
#END

#INDIR
/home/bjorn/pfc/prj/Tts/src/cli/in/
#END

#OUTDIR
/home/bjorn/pfc/prj/Tts/src/cli/out/
#END

#MODDIR
/home/bjorn/pfc/lib/debug/mod/
#END

#SETTINGSPATH
/home/bjorn/pfc/prj/Tts/src/cli/settings.txt
#END

#PROGRAMPATH
#END

#WAVECOM
Appendix B

An example of a synthesis problem

This section contains a small example of how to use the program in real life. The example is valid for both interfaces, but it is explained using only the Command-Line Interface with the simplification of building a startup program in a text file since it is faster to start that way, rather than using the commands inside the interfaces. The example is a re-synthesis of a phrase: Juan llegó a Barcelona hoy. (Juan arrived at Barcelona today) when the first rendering sounds unnatural.

B.1 Initialisation

First we must open a plain textfile (we call it ex1) and define some interface variables. The variables of interest in this case are: #TEXTFILES, #MODULES, and #SETTINGSPATH. See Appendix A on how to define these variables and their description.

- Between the markers #TEXTFILES and its #END we place the file containing the phrase Juan .... At this point we can give some information to the system for example by using some SABLE tags (<LANGUAGE ID=es><SPEAKER NAME="marta">Juan...<SPEAKER/></LANGUAGE>).

- Between the markers #MODULES and its #END we place all our speech processing modules of interest. In our first approximation we add the following modules:

```
"/mod/mloperator.so
"/mod/sable_text.so
"/mod/set_language.so
"/mod/speaker.so
"/mod/wnorm.so
"/mod/phrase_selection.so
"/mod/pause.so
"/mod/phonetic.so
"/mod/duration.so
"/mod/energy.so
"/mod/f0_contour.so
"/mod/ph2dp.so
"/mod/phonologic.so
"/mod/selection.so
"/mod/synthesis.so
```
The modules have all been explained in Chapter 3. Since no parameters are passed to the modules some default values are chosen.

- Between the markers #SETTINGSPATH and its #END we add our absolute path to the settings file which contain all the vital parameters for the modules (<settings.txt>). A complex file may look something like this: (responsible personnel at UPC maintain these files and can explain the settings exact meaning.)

```plaintext
#----------------------------------------
# Speaker and language settings
#----------------------------------------
LANGUAGE_DIR = "lang"
SPEAKER_DIR = "spk"
TRN_DIR = "./trn"
LOAD_CONFIGURATION = "defaults.txt"
SDB_SUFFIX = "16.sdb, .sdb"
#----------------------------------------
# Miscellaneous
#----------------------------------------
PHRASING = 1
PHONE = "phone.txt"
SPEECH_PERCENT = 0.85
NORMALIZE_E = 1
MAX_GAIN = 6
SDB_IN_MEMORY = 0
DIR_ROOT = "./"
#----------------------------------------
# SABLE Settings
#----------------------------------------
SABLE_CONTAINERS = "EMPH, LANGUAGE, DIV, PITCH, RATE,
    SPEAKER, VOLUME, NOTSAY"
SABLE_EVENTS = "MARKER, AUDIO, BREAK"
SHORT_LINE_LENGTH = 50
LONG_LINE_LENGTH = 100
DEFAULT_RATE = 170.0
RATE_fastest = 200.0
RATE_fast = 185.0
RATE_medium = 170.0
RATE_slow = 155.0
RATE_slowest = 140.0
DEFAULT_VOLUME = 1.0
VOLUME_loudest = 3.0
VOLUME_loud = 2.0
VOLUME_medium = 1.0
VOLUME_quiet = 0.5
#----------------------------------------
# Pauses
#----------------------------------------
PHRASING = 0
#----------------------------------------
```

65
# Mar2ml Epochs2Fo i wav2energy
#-----------------------------------
MAR_DIR = "mar"
PIT_DIR = "mar"
WAV_DIR = "mar"
#-----------------------------------
# Fujisaki
#-----------------------------------
TREE_DIR = "./tree"
FUJISAKI_GA = "ga.tree"
FUJISAKI IG = "ge.tree"
#-----------------------------------
# Debug
#-----------------------------------
SHOW_SELECTION = 0
SHOW_PRUNE = 0
SHOW_CLUSTER = 0
SHOW_PAUSE = 0

Now we are ready to start the program by `mat exe1` and to move on to the next stage. With `mat -help` the actual start-up options are shown.

### B.2 Execution

When the program starts there will be a few messages that indicate if all modules are correct and if the settings file was read in a proper manner. Sometimes, if the modules have been compiled in an incorrect manner, there will be problems with the symbol resolving part. One can then try a recompilation of that module with all its functions included statically.

To start the analysis we execute the two following commands and the modules are processed on the input file.

```
load modulesun
```

We can at this stage view the contents of the multilayer by either the `show` command or any of the `mlayer2...` commands if it is needed.

### B.3 Validation and re-execution

To listen to our speech signal (out.wav) we launch for example Wavesurfer Sjölander (2004) by the following line:

```
! wavesurfer out.wav &
```

I presume that the output wave file is called `out.wav` and is known to the system by either some settings or parameters passed to the synthesis module. When listening we may detect some sequence that sounds unnatural. Supposing that the unnaturalness is due to the F0 curve, we may change our module (F0_contour) to another module (F0_Fujisaki) by:

```
unload module 11
add module "/mod/fujisaki.so" 11
load module 11
```
If the unnaturalness is due to a badly written module we can write `unload module 11` and change the code of the module, recompile it and let the new improved module replace the old one and then do a `load module 11`. We are now ready to re-execute the modules and listen to our re-synthesised utterance.

```plaintext	run
! wavesurfer out.wav
```

If the synthesis is still not good enough we can repeat the process until a satisfactory result is achieved. Another option to improve the speech synthesis may be to edit the multilayer by `edit mlayer` and then process only certain modules like the synthesis modules, or the prosody modules.
Appendix C

Notifications

C.1 Further module information
When constructing a module with multiple modules with the same name, only one module is added into memory, i.e. the last one in the program file. In those cases when information about modules is shown to the user the information is collected from the module that is inside the heap. That is, if we have two modules with the same name `mymod1` and our module list is:

- `mymod1`
- `mymod2`
- `mymod3`
- `mymod1`

a loading of the list into memory will result in 4 objects of the type `Ml_Module` but the second `mymod1`-module will not have a dynamic library pointer (dlib). This is due to iterations in C++ and how the modules are identified.

C.2 Screenshots
This section contains the screenshots that were not suitable to show in the user manual.
Figure C.1: The window displayed when opening an old program.

Figure C.2: The window displayed when saving the existing program with a new name.
Figure C.3: The window displayed when adding a module.

Figure C.4: The text file window.
Appendix D

Scripts

D.1 Conversion of code files to dynamic libraries

This script is based on a script developed at UPC that defines compiler flags and looks for a certain directory structure. The script below looks for a file Make_aux in a directory two levels below the current. More about the Make_aux script is explained in Bonafonte, A. (2004).

SHELL = /bin/sh
# If ROOT_DIR is not defined try to guess it here
ifndef ROOT_DIR
export ROOT_DIR := $(shell pwd | sed \'s/prj/.*\'/\'\s{}/\'\s{}/\')
endif

export ROOT_SOURCE := $(ROOT_DIR)/prj
include $(ROOT_SOURCE)/Make_aux

#-----------------------------------------------
# This part of the makefile only is used when executed from the target directory
# Make_aux remakes it from there (and set/unset THIS_IS_A_SRC_DIR)
#-----------------------------------------------

ifndef THIS_IS_A_SRC_DIR
# Directories
MOD_DIR := $(ROOT_LIB)/mod
OBJ_MOD_DIR := $(patsubst $(ROOT_SOURCE)/%,$(ROOT_OBJ)/%debug,$(THIS_DIR))
INCLUDE_DIR1 := $(ROOT_DIR)/prj/Tts/src/inc
INCLUDE_DIR2 := $(ROOT_DIR)/inc

# Compilation
CXX=g++
CPPFLAGS=-I$(INCLUDE_DIR1) -I$(INCLUDE_DIR2) -fPIC

# TO DO: All the .cpp files, ... or choose the ones you need
CPPSRC1 := \$(notdir \$(wildcard \$(SOURCE_DIR)/*.cpp))

# First will make the out file then the so-file
all: OFILES SOFILES

#----------------------------------------
DEP1 := \$(CPPSRC1:.cpp=.d)
OBJ1 := \$(DEP1:.d=.o)
MODFILES := \$(shell grep -H 'launcher' *)

SOFILES1 := \$(sort \$(MODFILES:.o=.so))
#SOFILES1 := \$(sort \$(OBJ1:.o=.so))#Remove duplicated names

#Static libraries
#LDFLAGS := \$(ROOT_DIR)/lib/debug/libml.so \$(ROOT_DIR)/lib/debug/libutils.so
LDYNLIBS := \$(ROOT_DIR)/lib/debug/libml.so \$(ROOT_DIR)/lib/debug/libutils.so
SOFLAGS := -shared -fPIC -t -Wl,-E#,--symbolic
OFILES: \$(OBJ1) \$(DEP1)
\@{}-test -d \$(\@{}D) || (echo "{}Create intermediate mod
dir: \$(\@{}D)");
mkdir -p \$(\@{}D)
include \$(DEP1)

define MAKESOFILE
\$(shell \$(CXX) \$(LDFLAGS) \$(LDYNLIBS) \$(SOFLAGS),-soname,\$(SOFILE),-o
\$(SOFILE)
\$(OBJ_MOD_DIR)/\$(subst .so,.o,\$(SOFILE)))
mv -vf \$(SOFILE) \$(MOD_DIR)/
endef

SOFILES: \$(OBJ1)
\@{}-test -d \$(MOD_DIR) || (echo "{}Create mod dir: \$(MOD_DIR)"{}; mkdir
-p \$(MOD_DIR) )
\@{}echo 'Making Dynamic loadable files'{}
\$(foreach SOFILE,\$(SOFILES1),\$(MAKESOFILE))

#----------------------------------------
clean:
- rm -f \$(SOURCE_DIR)/*~{}
c- rm -f \$(DEP1) \$(OBJ1)
#----------------------------------------
clean_all:
- rm -f \$(SOURCE_DIR)/*~{}
c- rm -f \$(DEP1) \$(OBJ1)
c- rm -f \$(MOD_DIR)/*~{}
endif

72