Multilingual Data Configurable Text-to-Speech System for Embedded Devices

Kimmo Pärssinen & Marko Moberg

Nokia Technology Platforms
Tampere, Finland
{kimmo.parssinen; marko.moberg}@nokia.com

Abstract

In this paper a low footprint multilingual text-to-speech (ML-TTS) framework is presented. The system is a part of a speaker independent name dialing system that has been introduced in Nokia Series 60 mobile phones. In the ML-TTS systems that are based on the Klatt88 engine there usually exist sets of language specific rules that are used to modify the speech synthesis parameters. Usually, the size of the program code due to the language specific rules becomes large when the number of languages increases. In addition, adding TTS support for a new language is not so easy when the TTS rules are implemented as program code. The development work would require the modifications of the source code, which is always prone to errors and time consuming. The paper presents a novel scheme that both alleviates the memory problems and also makes the language development easier compared to the typical existing solutions. In this framework the language dependent TTS rules are implemented as a scripting language that is stored in text files, one file per each language. The files are converted into a binary form and the rules therefore are implemented as data. With the approach, only the data of the active language needs to be kept in memory and typically the size of a single data file remains small. During synthesis an interpreter is used to process the rules and modify the synthesis parameters accordingly. Moreover, adding TTS support for a new language involves writing the new set of language specific rules and ideally no modifications to the TTS engine code are needed. In addition to the language specific rules, all language dependent information, such as the prosodic model, is stored into the binary file i.e. the language package. Also due to the introduction of the language packages, the TTS engine can be configured to any desired set of languages simply by preparing and providing the associated language packages.

1. Introduction

Development of a TTS system is an interdisciplinary effort. It requires knowledge about human speech production and languages being developed. The actual implementation of a fully functional system, on the other hand, requires good software skills. It is often difficult to find a single person to master all the areas of TTS development. Instead, the experts of various fields must work together. Especially the development of multiple languages tends to require linguistic knowledge that can only be acquired by consulting the experts who are familiar with the given language. Therefore it is necessary to be able to separate the language creation process from the actual TTS engine development. This becomes even more evident when the TTS engine needs to support multiple languages at the same time.

In the low-end ML-TTS systems based on the Klatt88 engine, there are a number of language specific rules that operate on the synthesis parameters [1][2][3]. These language specific rules are usually implemented as program code. However, such an approach has several drawbacks. Firstly, the size of the program code increases with the number of supported languages. Therefore, the size of the random access memory (RAM) that is needed for running a ML-TTS engine becomes large. Secondly, the rules need to be compiled and linked together with the rest of the software before the synthesis output can be obtained and evaluated. Thirdly, the language development requires a good knowledge of the programming language and offers many unrestricted ways of implementing the rules. The excessive “freedom” may result in a complex and developer specific implementation of the language rules.

Some of these shortcomings can be alleviated with dynamically linked libraries (DLLs). In such implementation, the language specific TTS rules are implemented as one DLL per language. This approach allows the loading of languages to the RAM memory just when they are needed instead of keeping all the languages of a ML-TTS system in the memory at once. The problem is, however, that the DLLs would still need to be stored on ROM or FLASH memory. The total size of the DLLs would still be large while the size of the program code executed in the random access memory (RAM) would be reduced. The use of DLLs may also separate the development of the synthesizer engine software and language rules. Modification of language specific rule DLLs does not require recompilation and linking of the main synthesis engine.

The DLL approach does not solve the issues with the expressiveness of the programming language and can not produce platform independent sets of language rules. The memory footprint is also larger than necessary due to some code that is needed in all the DLLs. Some restrictions in linking and in memory configuration may also apply since the DLLs contain executable code.

This paper presents an alternative, multilingual low footprint TTS system architecture that separates the synthesis engine and language rules into separate modules. Languages are described completely in language specific data files making the actual synthesis engine language independent. Similar kind of approaches to TTS system design are for example the SVOX TTS system described in [4], ETI Eloquence and its delta language in [5] and [6], and Festival synthesis system presented in [7] and [8]. Our approach aims to offer true multilingualism with lower footprint utilizing the

properties of short utterances in limited domain synthesis with unlimited vocabulary.

2. TTS System Architecture

The TTS system consists of a completely language independent engine and a language specific data that is loaded into memory during synthesis. The top-level system diagram showing the main functional modules is presented in Figure 1. The text-to-phoneme (TTP) module will not be described in detail in this paper.

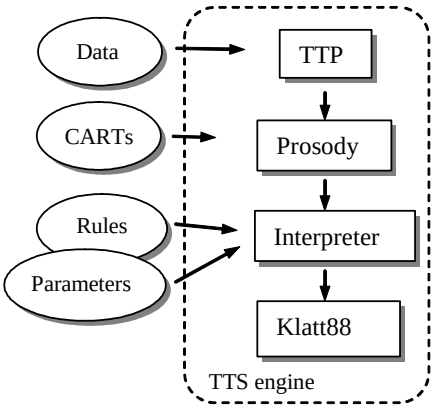

Figure 1: Block diagram of the TTS system.

The synthesizer engine includes the following data configurable and language independent modules: The TTP module, a prosody module, a rule processing module and an actual formant synthesizer (Klatt88). The language data consists of e.g. lookup-tables for TTP, prosody parameters for prosody creation, and phoneme parameters and language specific rules to modify them.

The prosody module uses CARTs (Classification And Regression Trees), which have been encoded into a binary format [9]. The trees have been automatically generated and do not require continuous tuning [7][8]. This paper concentrates mainly on the interpreter module and on the rules that relate to it.

3. Language Specific Rules

3.1. Speech synthesis control data description

In the final system the control data is stored into a language dependent and general data packages that are loaded into the system memory during speech synthesis. The language dependent data consists of the following data types:

1. CARTs to predict syllable boundaries, syllable accent, phoneme or segment durations and CARTs to predict the pitch in the beginning, in the middle and at the end of the syllable. Instead of CARTs, tonal languages use pitch templates for each given tone.

2. Klatt88 parameter table to determine Klatt parameters (e.g. formants, bandwidths, gains, spectral tilt) as described in [2] and [10], and phoneme attributes (type, manner, and place for consonants, rounding, frontness, height and length for vowels) for each phoneme of a given language [11].

3. Phoneme sequence rules to perform context dependent phoneme modifications (remove, replace, add), adding aspiration and short pauses into the phoneme sequence to be synthesized. The durations of the phonemes can also be altered.

4. Phoneme class and context dependent rules to modify the Klatt88 parameter values and implement co-articulation. These are split into the following rules: phoneme type rules, phoneme manner rules, previous context dependent rules, next context dependent rules to modify the formant synthesizer parameters. The modification is carried out using specific values and ramps as shown in Figure 2.

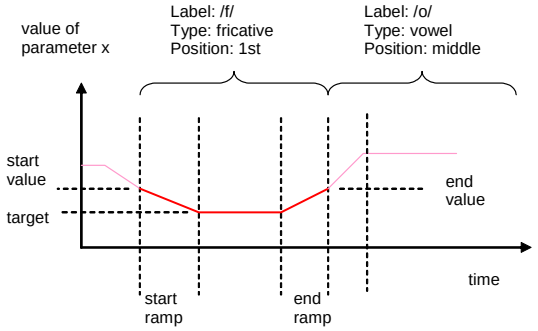

Figure 2: Visualization of ramps and values which can be modified by the rules.

The figure presents the three distinctive sections of a single Klatt parameter trajectory of a phoneme /f/: 1) start ramp, 2) steady-state target, and 3) end ramp. The ramps and the values can be defined and modified in a similar manner for each Klatt parameter.

The general data consists of phoneme manner and type rules that are in common with all the languages. It is also possible to store phoneme and language mapping rules into this data package.

3.2. Syntax of the Rules

The phoneme sequence and phoneme level rules are first written using a specific scripting language and then parsed into a tree like data structure. A part of the simple informal top down BNF type of grammar for the rules is given in Figure 3 [12]. The rule description language consists of three different types of rule *nodes* (conditions). The nodes can contain *questions* and *target* data. The node types are IF, ELIF and ELSE and their functionality and properties are more or less similar to those found in e.g. C. IF and ELIF nodes always contain a question/condition that is tested. The questions can be targeted to the phoneme between indices [-3,3] using *relatives* PPP, PP, PREV, CUR, NEXT, NN, and NNN, where the current phoneme has the index zero.

If the condition is found to be true possible target functions and operations specified in the target data field are

called to modify the phoneme sequence and/or phoneme's Klatt parameters. ELSE node differs from IF and ELIF so that it only contains target data and in this case the target data is not optional.

The *logicals* (AND, OR) allow the testing of several conditions in one question node. The node can have multiple occurrences of both ANDs and ORs thus providing a powerful way of expressing complex questions.

```
RULE ::= [NODE]+

NODE ::= NODETYPE:
QUESTION[:QUESTION:LOGICAL]*

            [NODEDATA]*

            [NODE]*

NODETYPE ::= "IF"|"ELIF"|"ELSE"

QUESTION  ::=  RELATION"|"RELATIVE"|"
FEATURE"|"VALUE

NODEDATA ::=  TARGETS : RELATIVE
[OPERANDS | NUMBER] [OPERANDS |
NUMBER]*

RELATION  ::=  "IS"|"NEQUAL"|"GREATER"|
"LESS"

RELATIVE    ::=    "PPP"|"PP"|"PREV"|
"CUR"|"NEXT"|"NN"|"NNN"

FEATURE  ::=  "LABEL"|"TYPE"|"MANNER"
|"PLACE"|"LENGTH"|"HEIGHT"|
"FRONTNESS"|
"POSITION_IN_THE_SEQUENCE"|
"KLATT_PARAMETER"|…

VALUE ::=  PHONEMECLASSES|STRESS|
BOOLEAN|KLATTIND|NUMBER|PHONEMELABEL|…

TARGETS ::= "START_RAMP"|"END_RAMP"|
"START_VAL"|"END_VAL"|
"MODIFY_DURATION"|"SET_DURATION"|
"PHONEME_CHANGE"|"ADD_ASPIRATION"|…

OPERANDS ::= "Target function specific
values and identifiers"

LOGICALS ::= "AND"|"OR"
```

Figure 3: Syntax of the synthesis rule language.

An example illustrating how the rules are used can be found in Figure 4. The first IF node checks whether the manner of the current phoneme is plosive. In case the condition matches, the second nested IF node is evaluated. If the current phoneme label is "b", the actions are taken to modify the ramps and values (see also Figure 2) using the specified target functions with given arguments.

The rule framework does not set any restrictions on the number of matching conditions. If the conditions are not mutually exclusive there might be several rules affecting the same set of parameters. In such cases, the succeeding settings overwrite the preceding ones. This property must be taken into account when designing the rules. For example, the general rules should be applied before the more specific ones.

```
IF:
{
 (
 IS|CUR|MANNER|PLOSIVE
 )
 IF:
 {
  (
  IS|CUR|LABEL|"b"
  START_VAL_ABS:CUR
  AV 30
  START_RAMP:CUR
  AV 40
  END_VAL:CUR
  AV 80 F1 80 F2 80 F3 80
  )
 }
}
```

Figure 4: Extract from the Finnish phoneme manner based rules

3.3. Coding and Applying the Rules

Given that the language dependent TTS rules are stored in a text file for a single language, the text file needs to be converted into a binary data file that can be interpreted with the computer program. The idea is that each variable, condition, operation, and operand described in the grammar of the rules is enumerated and converted into the numerical representation according to the enumeration list. During synthesis the binary file is loaded and the synthesizer's data structures are initialized according to the data package. An interpreter performs the operations specified in the rules. The interpreter scans trough a phoneme sequence and checks if the condition of a rule matches. If the condition matches, "true" is pushed to stack that holds the values of the conditions. Otherwise "false" is pushed to the stack. When a logical operator is met in the condition field of the rule, the stack is evaluated and the final value is stored again on top of the stack. Finally, when the conditions have been met, the correct modifications to the Klatt88 parameters are done by calling the enumerated target functions with given parameters that are stored in the node data of the rule.

3.4. Configuration of the TTS System

The TTS system presented can be configured for the set of target languages by preparing the TTS rules and other language dependent information and compiling those into the binary language package. Each package needs to be listed in a configuration file that determines which languages are supported. The configuration is composed of the general, language independent data package, and the language dependent data packages. The general data package stores the TTS rules that are in common with all the languages. The language dependent data packages include all the language dependent information. When the language configuration changes the configuration file needs to be updated accordingly. Ideally, there is no need to change the

implementation of the TTS engine when the set of TTS languages changes.

4. Development Process

The development of a TTS language requires certain steps, which can be partially automated. The training of the prosody model is an automated process that is usually carried out once. The definition of phoneme parameters and language specific transition rules require manual work and several rounds of iteration.

CARTs were used to encode and predict the location of the syllable boundary, syllable stress and, in case of the CART based intonation also syllable pitch. The trees were automatically trained using a variety of different level features extracted from the annotated training set. For tonal languages, pitch templates were used to model the intonation. These templates were hand tuned by a native speaker. The parameter driven Klatt88 synthesis engine uses a set of values to generate each frame of synthetic speech. The current implementation has 39 modifiable parameters associated with each different speech sound i.e. phoneme. The phonemes usually differ between languages so that there needs to be separate phoneme sets for each given language. The back end of the synthesis system uses those phoneme parameters and modifies them according to some context dependent rules that were described in section 3. Since the development process of the rules is highly iterative by nature a graphical user interface was created to speed up the rule development. It provides an easy access for creating, converting and testing the synthesis rules in practice. The system is described in more detail in [13].

The system also provides a module to create mappings between languages or, in practice, between different set of phonemes. This mapping module provides the rules for presenting the sounds of the source language with the given set of phonemes of the target language. Since the sounds of the source language are presented using the phonemes of the target language through applying a specific set of mapping rules, the languages should be relatively similar both in the prosodic and the phonetic sense. The goal of the phoneme mapping is to use the phonemes of the target language to create a sequence of phonemes that would sound as close to the original source language as possible. This method is described in [14] and referred to as language morphing.

5. Results

The memory requirements of the language specific parts of the DLL and the rule based approach are presented in Table 1. The same functionality i.e. the same set of language rules was implemented using both the DLLs and the rules script language. The DLL sizes were obtained using a GNU compiler with size optimizations. *Strip* was used to remove their symbol tables to make the binaries more compact. The size of the rule interpreter is approximately 50 kB, which need to be taken into account when comparing the figures presented in the table. For example, language specific processing in a monolingual system for Finnish would require 45 kB as DLL implementation and 63.3 kB (13.3 kB + 50 kB) as rules. However, a multilingual system supporting Finnish, UK English, German and French would take 196.6 kB as DLL but only 108.8 kB as rules including the interpreter.

Table 1: List of TTS languages and their sizes.

Language	DLL size (kB)	Rule size (kB)
Brazilian Portuguese	60.9	14.3
Canadian French	55.2	16.6
Cantonese	42.0	14.3
Czech	27.1	11.7
Finnish	45.0	13.3
French	48.7	14.1
German	49.9	15.4
Greek	48.8	11.8
Italian	57.8	12.0
Mandarin	49.1	14.6
Portuguese	48.0	13.3
Russian	35.6	17.4
Spanish	60.3	13.6
Swedish	34.3	16.4
Taiwanese	49.1	14.6
UK English	53.0	16.0
US English	53.9	16.6
US Spanish	60.7	13.9

The average sizes of a language specific DLL and a language specific rule file are approximately 49 kB and 15 kB, respectively. The memory reduction of a language specific data/program code was 69% in the average.

6. Conclusion

With the structure and implementation of the TTS engine described in this paper the TTS system can be configured to any desired set of languages ideally without modifications to the engine code. The configuration takes place by preparing the language specific rules and compiling those together with the rest of the language dependent information into the language specific packages. The language dependent packages are listed in the configuration file that is parsed by the engine code. By changing the configuration file, the TTS engine can be configured for any set of target languages. Adding new TTS languages is also simple. A language package is compiled from the rule files of the new language and the configuration file is updated accordingly.

In addition to the easier configuration and simplified language development, the advantages include reduced RAM and ROM/FLASH memory footprints. Since there is no longer language dependent code, the size of the executable is significantly reduced. It was also shown that the TTS rules can be represented in a more compact form using the new implementation. Therefore both the size of the executable code and the size of the language data are reduced. The amount of saved memory in the rule based system increases with the number of supported languages. Another benefit is that when the languages are described as data they are platform independent thus making it easier to support multiple hardware configurations and operating systems.

7. References

[1] Keller, E. (ed.), *Fundamentals of speech synthesis and speech recognition*, John Wiley & Sons Ltd, West Sussex, England, 1994, pp. 23-40.

[2] Klatt, D. H., "Software for a cascade/parallel formant synthesizer", *J. Acoust. Soc. Amer*., *67(3) , pp. 971-995, 1980.*

[3] Allen, J., Hunnicutt, S. M. and Klatt, D., *From text to speech: The MITalk system*, Cambridge University Press, Cambridge, 1987.

[4] SVOX Ag, SVOX Technical White Paper, http://www.svox.com, 2003, pp. 4-5.

[5] Hertz, S. R., "The delta programming language: An integrated approach to non-linear phonology, phonetics and speech synthesis", *Papers in Laboratory Phonology I*, Cambridge University Press, 1990.

[6] Hertz, S. R., Younes, R. J. and Zinovieva N., "Language-Universal and Language-Specific components in the Multi-Language ETI-Eloquence Text-To-Speech System", *In Proceedings of XIV International Congress of Phonetic Sciences, Vol 3*, pp. 2283-2286, San Francisco, 1999.

[7] Taylor, P., Black, A. and Caley, R., The Architecture of the Festival Speech Synthesis System, In *Proceedings of the 3rd ESCA Workshop on Speech Synthesis, Jenolan Caves, Australia*, pp. 147-151, 1998.

[8] Black, A., Taylor, P. and Caley, R., *The festival speech synthesis system, system documentation*, Centre for Speech Technology Research, University of Edinburgh, 1999.

[9] Breiman, L., Friedman, J. H., Olshen, R. A. and Stone, C. J., *Classification and Regression Trees*, Wadsworth Inc., Belmont, CA, USA, 1984.

[10] Klatt, D. H. and Klatt, L. C. "Analysis, synthesis, and perception of voice quality variations among female and male talkers", *J. Acoust. Soc. Amer*., *87(2), pp. 820-857, 1990.*

[11] Ball, M. J. and Rahilly J., *Phonetics, the science of speech*. Oxford University Press Inc., New York, USA, 1999, pp. 40-59, 85-91.

[12] "Augmented BNF for Syntax Specifications: ABNF", http://www.ietf.org/rfc/rfc2234.txt

[13] Moberg, M. and Pärssinen, K., "Integrated Development Environment for Multilingual Data Configurable Synthesis System", In *Proceedings of International Conference of Speech and Computer 2005*, Patras, Greece, pp. 155-158, 2005.

[14] Moberg, M., Pärssinen, K. and Iso-Sipilä, J., "Cross-Lingual Phoneme mapping for Multilingual Synthesis Systems", In *Proceedings of ICSLP 2004, Jeju, Korea*, 2004.