SOUNDBITE: A SPEECH SYNTHESISER FOR MALTESE

K. Bugeja, G. Mangion, M. Borg, C. Vella and C. Gafà

Department of Research and Development, Crimsonwing,
Lignum House, Aldo Moro Road, Marsa, Malta
{kbugeja, gmangion, mborg, cvella, cgafa}@research.crimsonwing.com

ABSTRACT

This paper presents Soundbite, a third-generation, real-time, text-to-speech synthesiser for the Maltese language. We describe the system's software architecture and briefly cover the data structures and modules comprising the system. We emphasise the architecture's adaptability to different module implementations and conclude with a performance evaluation of the synthesiser.

***Index Terms*—** speech synthesis, maltese, text-to-speech, TTS, software architecture

1. INTRODUCTION

In this paper we describe the software architecture behind Soundbite, a real-time text-to-speech (TTS) synthesiser for the Maltese language. We discuss the issues and design decisions made to satisfy the core requirements of the synthesiser.

The primary requirement was to synthesise Maltese speech with a high level of naturalness and intelligibility with respect to actual spoken Maltese, limited to the mode of neutral discourse. The system was further required to synthesise speech within time intervals suitable for real-time applications. To this effect, we discuss techniques, algorithms and data structures adopted to ensure the required performance. Finally, the system was required to support enriched text input, enabling the synthesiser to apply volume, tonal and temporal effects to the resulting output, and generate event notifications where required. We thus discuss how these elements were factored in the design of the synthesiser.

The architecture of Soundbite follows in the steps of established generic TTS systems such as Festival [1], Mary [2] and DIXI [3]. The core feature of these systems is a multi-level data structure in which linguistic information is added and processed by the modules comprising the TTS pipeline. We adopted a similar formalism within Soundbite, consisting of several co-indexed data streams including lexical and control token sequences, phrasing information, part-of-speech (POS) and semiotic tags, phone and prosody specification, event markers, diphone specification and ultimately the audio signal.

2. ARCHITECTURE

Soundbite is a third-generation concatenative speech synthesis engine based on a two-stage pipeline. Automatic unit selection methods are used to query efficiently a speech database during runtime and determine the best choice of units for generating utterances for respective input text. Diphones are preferred over other unit types as they provide better phonetic coverage for a given database size [4, 5].

Figure 1 illustrates the two-stage pipeline by which text is converted to speech. In the first stage (front-end phase), orthographic text input is processed and a phonetic specification is generated. In the second stage (back-end phase), this specification is used to search for matching units in a database. These units are then concatenated into a final speech utterance.

2.1. Structures

At its core, Soundbite uses a multi-level data structure [6]. This comprises a hierarchy of linear data items, or streams, which are augmented and enriched at every step of the transformation, from text to utterance. At each step, previous streams are preserved, as later stages might still require them. We term this data structure, the *stream-map*. The primary reason for favouring multi-

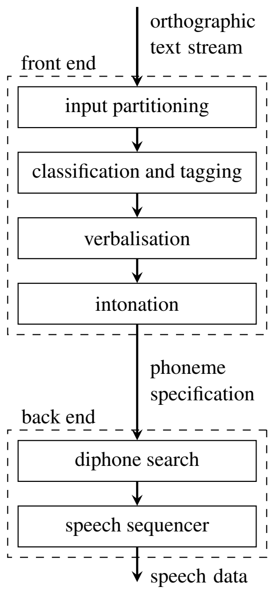

Fig. 1. Soundbite architectural overview

level data structures over other formalisms like string re-writing is essentially the lack of flexibility afforded by the latter [1]. Such methods transform and embellish an orthographic textual representation at every stage in the TTS pipeline, until it has been finally converted into an utterance. Besides being unwieldy, these methods transform data in such a way as to make it unavailable to later stages in the pipeline.

2.2. Phoneme Specification

The Maltese alphabet contains 30 letters, some of which are expressed as digraphs. Maltese operates with a system of 24 consonantal phonemes and 11 vocalic sounds, in addition to 7 diphthongal segments, each composed of one of the eleven vocalic sounds together with an [ɪ] or [ʊ] [7]. A phoneme specification is a sequence of such phonemes that is eventually used to look up diphones from a database, used to form the final utterance. The phoneme specification is derived from input text.

2.3. Grapheme to Phoneme

The front-end phase transforms its input through a number of stages. The first stage cleans the input by applying a filter, discarding ambiguous and illegal symbols, and partitions it into meaningful units like words. Further to partitioning, disambiguation and expansion stages resolve ambiguities like homographic words and expansion of acronyms, dates and numbers. This is carried out through the help of a semiotic classification process which uses natural language processing to help in the tagging. In contrast to languages like English, Maltese is close to being a homographic language. The correspondence between written symbols (graphemes) and sounds (phonemes) is almost one-to-one. This strong correlation between graphemes and phonemes allowed the use of a set of rules for converting input from graphemes to phonemes [4, 8]. Notwithstanding, Maltese text is usually peppered with words that do not follow these standard rules of grapheme-to-phoneme conversion, the reason being that names, surnames and words in other languages which are commonly used may have been rendered as they appear in the original language, rather that in transliteration. Accordingly, we employ a lexicon which helps in language disambiguation as well as grapheme-to-phoneme conversion of these irregular cases. We also employ the use of the lexicon to determine stress and syllable length when these cannot be immediately computed using standard rules. Finally, the phoneme specification is augmented with breaks and intonation information for the current speaker, before being fed to the back-end phase for rendering.

2.4. Diphone Search and Sequencing

In the back-end phase, the phoneme specification is converted to a diphone stream and used to perform unit look-up in a database of utterances. The diphone sequence that scores best, given factors like specification similarity and unit join costs, is returned. At this stage, the back-end also performs event sequencing; triggers like phoneme, word, phrase and sentence boundary events are encoded with the output, while the diphone units are concatenated to return the desired utterance.

3. IMPLEMENTATION

3.1. Abstraction

An important design goal for Soundbite was that of providing the abstraction and flexibility required for testing different algorithms without effecting changes to the core architecture. The front-end and back-end phases contain micro-pipelines, where each stage is exchangeable and various algorithms can be plugged in. This modular approach allows for an incremental development strategy, where basic modules are eventually replaced with more sophisticated versions, or different algorithms altogether.

3.2. Input Partitioning

Input to Soundbite is made up of data and control streams. The data streams consist of orthographic text, while the control streams define annotations like bookmarks and other special markers, and output modulators like volume and tonal effects, amongst others. Soundbite also supports asynchronous input in the form of abort and effect signals. Adapters are used to transform input data from other formats, like SAPI [9], into something Soundbite can understand and process.

The input streams may span multiple pages of text. To improve response times, input is segmented using a moving window, whose size is configurable. The windowed input is further processed and partitioned into phrases. Any phrases estimated to straddle the current window boundary are pushed to be processed in the next windowing iteration. The partitioned phrases are then processed sequentially by the front-end phase and sent to the back-end for consumption.

3.3. The Stream-map

The stream-map is a multi-level data structure which captures input transformations and embellishments across the TTS pipeline. It provides a useful abstraction for access to raw data in underlying structures. Accordingly, it allows us to access typed-data at each level of the hierarchy using both random-access and iterator patterns. The stream hierarchy is managed transparently; related streams (e.g. parent-child) store only differences, in order to minimise memory footprint. Furthermore, streams are encoded as sparse linear data items in a map data structure based on the Standard Template Library map [10].

3.4. Semiotic Analysis and NLP

The text classification problem [11,12] is addressed via a three-stage process, namely, *Open Classification*, *Natural Language Processing* and *Closed Classification*. Initially, during the *Open Classification* stage [13], every input token is analysed and associated to one or more semiotic classes. This is accomplished through the use of pattern matching. Every semiotic class has an associated regular expression, which captures the possible relationship of a lexeme to a class. Lexemes support Unicode characters and are therefore converted a priori to an 8-bit encoding that preserves graphemes specific to Maltese. The stream-map is embellished with a new stream containing the potential semiotic classes for each token.

During the next stages, we make use of a domain-specific scripting-language, an $if - then$ rule-based analyser and decoder, which is also supplied with constructs for accessing and modifying information in the stream-map. Rules exploit pattern-matching and the stream-map to simplify authoring; they provide an abstraction which allows one to focus on reasoning about Maltese grammar and semiotic classes rather then getting bogged down in implementation details. For each rule, the *antecedent* is comprised of a series of expressions. Starting with the current token, the first expression in the *antecedent* is evaluated; if a match is recorded, the second expression is applied to the subsequent token. A rule is matched, when the conjunction of the evaluation of all expressions is true, in which case, the *consequent* fires, performing an action on the stream-map such as adding, removing or updating information. An advantage of using a domain-specific language is that the rule authors need only be acquainted with the problem domain and how to express conditions and actions. The system automatically handles details such as performing boundary checks against the current position and consecutive tokens. Rules are formulated in a language similar to English. For example, the rule below, states that if the current token contains a word in Maltese and is tagged as a *part-of-speech* matching *article*, but the following token is not a hyphen, then the rule fires leading to the removal of the *article* tag.

$$\langle \mathbf{Lang}(maltese)\ \mathbf{POS}(article)\rangle$$
$$!\mathbf{Semiotic}(hyphen)\ \rightarrow\ \mathbf{DelPOS}(article)$$

This approach proved so powerful and flexible that we were able to simultaneously tackle both language disambiguation and classification, and code-switching. After the token stream has been classified, tokens are then sent to the verbalisers [13] corresponding to their semiotic type.

3.5. Front-end, back-end concurrency

A design goal of our speech synthesiser is that of being able to work with modest hardware, but exploit computational resources when available. Accordingly, after identifying the back-end phase as the most computationally expensive process in all of the TTS pipeline, we sought to increase the concurrency of our system in order to improve its response time. We cast the front-end and back-end phases as producer and consumer processes, the front-end producing phoneme specification streams, and the back-end consuming these specifications. The consumer processes execute asynchronously of the producer and one another. The system scales in the number of consumers, depending on the capabilities of the host machine. Increasing the number of consumers may introduce race conditions, whereby completing jobs out of sequence, consumers may reverse the order of phrases in the output. This has been handled through the introduction of a *Sequencer*, an entity which assigns a unique sequence identifier, in the form of a *slot*, to each job scheduled for consumption, ensuring that notwithstanding which job finishes first, the output is always orderly.

The output can be consumed in many different ways through the application of the *Output Device* abstraction. Output devices may be implemented as encoders and file writers (e.g. MP3 file output), null devices, or even to redirect output to a suitable system audio device. The Sequencer and Output Devices form a subject-observer system. Output Devices register with the Sequencer such that whenever contiguous slots are available for consumption, they are notified by the Sequencer and supplied with the respective slot information.

4. RESULTS AND EVALUATION

We have evaluated the response time of Soundbite with different input text sizes of 100×2^k tokens, where $0 \leq k \leq 9$. The evaluation was performed on a machine with an Intel i5 2.67GHz quad-core CPU and the results are shown in figure 2 above. The response time is influenced by the window size used in partitioning the input text. For this test, we chose the value of 1024 input tokens for the window size to amplify the partitioning effect. These results validate the use of the window-based input text partitioning technique in Soundbite, in that there is only a minimal increase in response time beyond the 1024-token mark when the input text in effect doubles in size for each new measurement. The response time seems to increase in linear proportion to the input size up to the window size specified for the test. Beyond this threshold, the response time still increases in linear proportion but at a much lower rate. We speculate that inputs that fit within one window are processed from beginning to end, taking substantial processing time as indicated by the steep incline of the plot. Beyond this, we attribute the additional time, indicated by the shallow incline of the plot, to the pre-processing performed on the complete input prior to processing the first window.

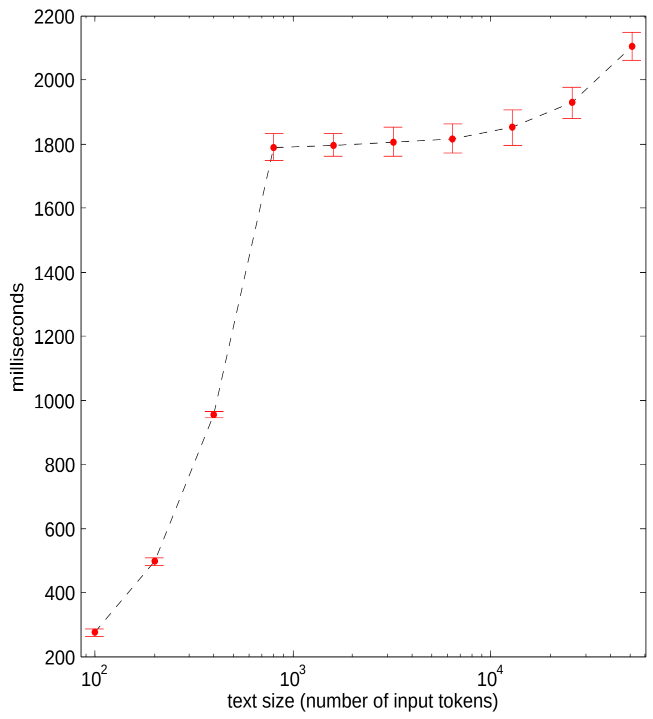

Fig. 2. Average response times (msec.) for different input text size. Error bars show 95% confidence intervals.

5. CONCLUSION

We have described the challenges encountered and the solutions adopted for the design and implementation of Soundbite. While the system meets the intended requirements, it is ripe for further work, particularly in the areas of natural language processing, prosody modelling and unit selection techniques. Beyond this, we also hope that Soundbite's open and modular architecture will encourage further work in the field of Maltese speech synthesis.

6. REFERENCES

[1] Paul Taylor and Alan W Black and Richard Caley, "The architecture of the festival speech synthesis system," in *In The Third ESCA Workshop in Speech Synthesis*, pp., 147–151.

[2] Marc Schröder, "The german text-to-speech synthesis system mary: A tool for research, development and teaching," in *International Journal of Speech Technology*, pp., 365–377.

[3] Paulo, Sérgio and Oliveira, Luís C. and Mendes, Carlos and Figueira, Luís and Cassaca, Renato and Viana, Céu and Moniz, Helena, "Dixi — a generic text-to-speech system for european portuguese," in *Proceedings of the 8th international conference on Computational Processing of the Portuguese Language*, Berlin, Heidelberg, 2008, pp., 91–100, Springer-Verlag.

[4] M. Borg and K. Bugeja and C. Vella and G. Mangion and C. Gafà, "Preparation of a free-running text corpus for maltese concatenative speech synthesis," in *In : 3rd Int. Conf. on Maltese Linguistics, Valletta, Malta*, 2011.

[5] Bozkurt, Baris and Ozturk, Ozlem and Dutoit, Thierry, "Text design for TTS speech corpus building using a modified greedy selection," pp., 277–280, 2003.

[6] Paul Taylor and Alan W. Black and Richard Caley, "Heterogeneous relation graphs as a mechanism for representing linguistic information," *Speech Communications*, vol. 33, pp. 153–174, 2001.

[7] Borg, Albert and Azzopardi-Alexander, Marie, *Maltese*, Routledge, London & New York, 1997.

[8] Micallef, Paul, *A Text To Speech System for Maltese*, Ph.D. thesis, University of Surrey, 1998.

[9] Microsoft Corporation, "Microsoft speech api," November 2012.

[10] Alexander Stepanov and Meng Lee, "The standard template library," Tech. Rep., WG21/N0482, ISO Programming Language C++ Project, 1994.

[11] Sproat, R. and Hirschberg, J. and Yarowsky, D., "A corpus-based synthesizer," in *Proceedings of the International Conference on Spoken Language Processing*, vol., pp., 563–566.

[12] Sproat, R., "English noun-phrase accent prediction for text-to-speech," *Computer Speech & Language*, vol. 8, no. 2, pp. 79–94, 1994.

[13] Taylor, P., *Text-to-Speech Synthesis*, Cambridge University Press, 2009.