MULTI-PLATFORM IMPLEMENTATION OF SPEECH
APIS

A Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science in the
Graduate School of The Ohio State University

By

Karen Manukyan, B.S., M.S.

* * * * *

The Ohio State University

2008

Master’s Examination Committee:

Dr. Eitan Gurari, Adviser

Dr. Rajiv Ramnath

Approved by

__________________________
Adviser
Graduate Program in
Computer Science and
Engineering
ABSTRACT

We live in an era of rapidly developing information technologies. Computers and the internet make finding information easier than ever before. Unfortunately not everyone can fully benefit from this: a vast amount of information is available in only textual form, making it difficult or even impossible for visually impaired and dyslexic persons to access it. Special tools for vocalizing textual information can be of great assistance to them, and a valuable tool for sighted people as well.

The goal of this project is to design and implement speech APIs that are easy to extend and use with a variety of speech engines on different operating systems. JSAPI (Java Speech API) serves as a base for the project. Proposed by Sun Microsystems, JSAPI consists of a set of well defined interfaces for Java classes and the speech markup language JSML. No implementation of JSAPI is provided by Sun. The few available implementations of JSAPI, firstly, do not have the full set of features that may be needed for speech enabled applications and, secondly, are limited to a specific speech engine and operating system.

This project takes significant steps towards solving these problems.
Dedicated to my fiancée, parents and brother
ACKNOWLEDGMENTS

I want to express my gratitude to Dr. Eitan Gurari for his support in the course of the project. Not only is he an excellent adviser, but also a wonderful person.

I want thank my fiancée Katie for helping me to proofread the thesis and, of course, for moral support.

I also want to thank Dr. Rajiv Ramnath and my friends, James Dinan, Issam Safa, Oleksiy Busaryev, Oleg Mishchenko and Sheetal Ghadse, for coming to the thesis defense, the last step towards my graduation.
VITA

July 30, 1980 .......................... Born - Odessa, Ukraine

September 1997 - June 2001 ............... B.S. Computer Science,
                                         Odessa National Polytechnic University, Odessa, Ukraine.

September 2001 - June 2002 ............... M.S. Computer Science,
                                         Odessa National Polytechnic University, Odessa, Ukraine.

June 2005 - June 2008 ..................... Graduate Research Associate,
                                         The Ohio State University.

FIELDS OF STUDY

Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Component</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Thesis statement</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Results</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Organization of the thesis</td>
<td>3</td>
</tr>
<tr>
<td>2. Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Speech Markup Languages</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 SAPI TTS Markup Language</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 JSML</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3 SSML</td>
<td>8</td>
</tr>
<tr>
<td>2.1.4 STML</td>
<td>9</td>
</tr>
<tr>
<td>2.1.5 SABLE</td>
<td>11</td>
</tr>
<tr>
<td>2.1.6 Aural CSS</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Speech Engines</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Microsoft Speech Engine</td>
<td>15</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>High-level architecture.</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Adapter elements and Java – C++ interface.</td>
<td>36</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Speech synthesis is a valuable feature for many computer applications. For some of them, such as accessibility tools and applications for the visually impaired, it is required. While plain text synthesis may be sufficient for some applications, others may require more sophisticated speech synthesis mechanisms, such as setting various vocal properties of the synthesized speech and allowing the user to interact with and control the synthesis and browsing processes.

1.1 Motivation

The tex4ht tool is available for translating \LaTeX{} files into different hypertext formats. The hypertext code captures the logical structures of the documents, and can be provided with information on how to render the data visually and auditorially. Various powerful browsers for visual representation of hypertext documents exist. Some of the browsers have support (natively or with the help of plug-ins [15]) for highly structural material, such as mathematical formulas, that cannot be effectively presented in a plain text format. However, not many advanced tools are offered for voice browsing.
Most of the available voice tools are intended for vocalizing plain text and HTML and are not sufficiently sophisticated to handle highly structural content [13, 14, 24]. Some of the more advanced screen readers, for instance JAWS [10], can use other tools to provide support for certain kinds of highly structured content.

Among the few speech tools that work with highly structural material are MathPlayer [11] and Emacspeak [4]. MathPlayer can not be easily used to add MathML support to any application, as it was designed as Internet Explorer plug-in. JAWS, for instance, can use MathPlayer’s functionality only indirectly through Internet Explorer. Thus, MathML support in JAWS is limited to web-pages loaded in Internet Explorer with the MathPlayer plugin installed. Not only is MathPlayer limited to Internet Explorer and Windows OS, but its support of markup languages is also limited to MathML only.

Analogously, Emacspeak can be used only within Emacs, which provides the control and navigation mechanism, but requires external browsers to serve the markup and, feature-wise, is not flexible enough for tailoring to deal with highly structured code.

Voice browsers can greatly assist visually impaired people, especially those who need access to information that is too complex to be presented in plain text format. The following are desirable properties in voice browsers:

- **Interactivity**: users should be able to control and navigate the process of speech synthesis.

- **Understanding the structure of documents**: the voice browsers should understand the underlying structure of the hypertext documents and should be able to provide efficient navigation capabilities to the users.
• *Adjustable speech properties*: Various voice properties, such as: volume, rate, pitch, etc. should be adjustable to provide for optimal user experience.

1.2 Thesis statement

The importance of advanced voice browsers has been illustrated in the sections above. Powerful and easy to use speech API is an essential component for facilitating the development of sophisticated voice browsers. A speech API implementation that can easily be extended to work with different speech synthesizers on a variety of operating systems will simplify the creation of cross-platform highly configurable voice browsers. Unfortunately, no such speech API implementations exist.

The main focus of this thesis is the design and development of a multi-platform Java Speech API implementation which can be used with different speech engines. The thesis also reviews existing speech synthesis software components and markup languages that can be used for the implementation of Java Speech API.

1.3 Results

The result of this project is an open source library, implementation of the speech synthesis part of the Java Speech API, for Linux and Windows operating systems. This library can be used by any application that needs speech synthesis capabilities. The library can also be easily extended for use with speech engines other than those already supported.

1.4 Organization of the thesis

The thesis is organized as follows: Chapter 2 overviews existing speech markup languages, speech synthesizers and speech APIs. Chapter 3 presents the main design
and implementation decisions made in the process of working on the project. Chapter 4 discusses potential avenues for future development of the project. Appendix A provides detailed instructions on how to build and use the library on Windows and Linux operating systems.
This chapter provides a review of some of the existing speech markup languages, speech engines and speech APIs. Short examples are provided for each of the markup languages. Another brief example is given for Java Speech API (JSAPI).

2.1 Speech Markup Languages

Cascading Style Sheets (CSS) are used to describe visual representation of hypertext documents. Similarly, the aural version of CSS [3] and special speech markup languages [28], such as Speech Synthesis Markup Language (SSML) [20], can be used to specify the voice properties of documents.

Several speech markup languages exist. Some of them were developed for specific speech synthesizers, while others were developed as standards to be supported by various synthesizers.

The rest of this section reviews and provides examples of existing speech markup languages.

2.1.1 SAPI TTS Markup Language

SAPI TTS Markup Language is a proprietary XML markup language developed for Microsoft TTS synthesizer. The following tags are defined in SAPI TTS:
- SAPI: restricts the scope of speech parameters' values, i.e. the closing SAPI tag resets the values of all parameters to the ones.
- LANG: specifies the language of the document.
- BOOKMARK: requests that the synthesizer send a notification when the the tag is reached.
- EMPH: emphasizes the contained text.
- PARTOFSP: specifies the part of speech (noun, verb, etc.) of the enclosed word.
- PITCH: sets pitch for the enclosed text.
- PRON: specifies pronunciation for the contained text.
- RATE: sets rate for the enclosed text.
- SILENCE: produces a pause in the synthesized speech.
- SPELL: specifies that the enclosed text should be spelled out, rather than synthesized in a regular way.
- VOICE: sets various properties (age, gender) of the synthesized voice.
- VOLUME: sets volume for the enclosed text.

SAPI TTS markup language example:

```xml
<SAPI>
Hello World<VOLUME LEVEL="0.5"><RATE SPEED="1">Volume is half the max, rate is increased by 10%</RATE></VOLUME>
</SAPI>
```
Detailed description of SAPI TTS markup language can be found in [17].

2.1.2 JSML

JSML is an XML markup language designed by Sun Microsystems for their Java Speech API (JSAPI). JSML was not developed to be used with a specific speech engine. It is intended to be used by a variety of engines on different platforms.

Tags defined in JSML include the following:

- PROS: sets volume, rate and pitch for the contained text.
- SAYAS: specifies the way the contained text should be said: as a number, as a date, spelled out letter by letter, etc.
- EMP: emphasizes the contained text.
- BREAK: produces a break (pause) in the synthesized speech.
- MARKER: requests that the synthesizer send a notification when the the tag is reached.
- ENGINE: passes engine specific data to the engine.
- PARA: defines contained text as a paragraph.
- SENT: defines contained text as a sentence.
JSML example:

<JSML>
Hello World<PROS VOL="0.5" RATE="+10%">Volume is half the
max, rate is increased by 10%</PROS><MARKER MARK="Marker reached"/>
Pause for a short time<BREAK SIZE="small"/>Emphasize the last
<EMP LEVEL="strong">word</EMP>
</JSML>

For additional information about JSML refer to [1].

2.1.3 SSML

SSML, designed by W3C’s Voice Browser Working Group, is an XML markup
language based on JSML. SSML was designed as a standard for assisting applications
in controlling speech synthesis.

Among SSML tags are the following:

- speak: root tag.

- say-as: specifies the way the contained text should be said: as a number, as a
date, spelled out letter by letter, etc.

- phoneme: specifies a phonemic pronunciation for the contained text.

- voice: sets voice gender, age and “name”.

- emphasis: emphasizes the contained text.

- break: produces a break (pause) in the synthesized speech.
• prosody: sets volume, rate and pitch for the contained text.

• audio: adds audio from an audio file to the synthesized speech.

• mark: requests that the synthesizer send a notification when the tag is reached.

SSML example:

```xml
<?xml version="1.0"?>
<speak version="1.0" xmlns="http://www.w3.org/2001/10/synthesis"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3.org/2001/10/synthesis
    http://www.w3.org/TR/speech-synthesis/synthesis.xsd"
    xml:lang="en-US">
Hello World<prosody volume="50" rate="+10\%">Volume is half the max, rate is increased by 10\%</prosody><mark name="Marker reached"/>
Pause for a short time<break strength="weak"/>Emphasize the last <emphasis level="strong">word</emphasis>
</speak>
```

More information about SSML can be found in [20].

2.1.4 STML

STML, an SGML based markup language, was developed by Bell Labs and Edinburgh University.

The following tags are defined in STML:

• LANGUAGE: defines the language of the contained text.
• GENRE: defines the genre of the contained text.

• SPEAKER: defines the speaker (sex, age, etc.) for the contained text.

• INTONAT: defines a pitch range for the enclosed text.

• BOUND: defines an intonational boundary for the contained text.

• EMPH: emphasizes the contained text.

• RATE: defines rate for the enclosed text.

• PHONETIC: defines enclosed text as a phonetic transcription.

• LITERAL: specifies that the contained text should be synthesized literally (spelled out, for instance), rather than synthesized in a regular way.

• DEFINE: defines pronunciation for the words in the contained text.

• SRC: adds audio from an audio file to the synthesized speech.

• OMITTED: specifies that the contained text should not be synthesized.

STML example:

<SPEAKER DESCRIPTION=female>
Female speaker has been set
<LITERAL mode=spell> this text is spelled out </LITERAL>.
<OMITTED VERBOSE=yes>
This text is not synthesized, but a listener
is notified that the text is omitted
</OMITTED>
For more information about STML refer to [21, 28].

### 2.1.5 SABLE

SABLE, an XML speech markup language, was developed by Bell Labs. The goal of SABLE was to combine existing speech markup languages: an early version of SSML, JSML and STML.

The following tags are defined in SABLE:

- **SABLE**: markup root tag.
- **EMPH**: emphasizes the contained text.
- **BREAK**: produces a break (pause) in the synthesized speech.
- **PITCH**: defines pitch for the enclosed text.
- **RATE**: defines rate for the enclosed text.
- **VOLUME**: defines volume for the enclosed text.
- **AUDIO**: adds audio from an audio file to the synthesized speech.
- **ENGINE**: passes engine specific data to the engine.
- **MARKER**: requests that the synthesizer send a notification when the tag is reached.
• PRON: defines a custom pronunciation for the contained text.

• SAYAS: specifies the way the contained text should be said: as a number, as a date, spelled out letter by letter, etc.

• LANGUAGE: defines the language of the contained text.

• SPEAKER: defines speaker’s properties for the contained text (sex, age, etc.).

• DIV: defines the contained text as a sentence or a paragraph.

SABLE example:

<SABLE>
Hello World<VOLUME LEVEL="0.5"><RATE SPEED="+10%">Volume is half the max, rate is increased by 10%</RATE></VOLUME>
<MARKER MARK="Marker reached"/>
Pause for a short time<BREAK LEVEL="small"/>Emphasize the last <EMPH LEVEL="strong">word</EMPH>
</SABLE>

A detailed description of SABLE can be found in [16, 27].

2.1.6 Aural CSS

Aural CSS defines additional, aural properties (voice, volume, rate, pitch, etc.) for HTML (XML, in general) elements. The same way as “regular” CSS properties define the way HTML elements are rendered by browsers, aural properties define the way HTML elements are vocalized by browsers. Unfortunately, modern browsers do
not widely support Aural CSS [2]. One of the projects aimed to deliver Aural CSS support to the users is FireVox [6], a Firefox extension.

Example of Aural CSS:

```css
<style type="text/css">
  .low_pitch_voice {
    voice-pitch: x-low;
  }

  .high_pitch_voice {
    voice-pitch: high;
  }

  .slowly_pronounced {
    voice-rate: x-slow;
  }

  .medium_volume_level {
    voice-volume: medium;
  }
</style>
```

For a detailed description of aural CSS refer to [3].
2.2 Speech Engines

Special graphical software components are used in visual browsers to render the contents of hypertext pages on users’ computer screens. Analogously, software components, speech synthesizers that can produce a vocal representation of a hypertext page are needed for voice browsers.

Even though many different speech synthesizers exist, not all of them can be efficiently used for sophisticated speech browsers. To satisfy the requirements of this project, speech engines must have the following features:

- A rich set of various events corresponding to the different states of speech synthesis, such as the starting of word synthesis, the reaching of a marker, etc.
- Support for sophisticated speech markup language.
- An API that can be used to control the speech synthesis process.

Speech events are crucial for ensuring the interactivity of the speech browsers: it is impossible to implement intelligent navigation through the synthesized text and synthesis control without getting information about the synthesis process from the speech engine.

Defining the logical structure and voice properties of the text requires a sophisticated speech markup language.

Speech API is necessary to control the speech synthesis process: start, pause, resume, cancel speech synthesis process, etc.

The following is a review of some of the existing speech synthesizers.
2.2.1 Microsoft Speech Engine

Microsoft speech engine is integrated into the Windows operating system. The speech engine provides a COM-based API, called SAPI, for manipulating the engine. In addition to the API, SAPI TTS markup language can be used to set various vocal properties of the synthesized text. The newest version of the Microsoft speech engine also supports SSML.

The following events are generated by the Microsoft speech synthesizer and can be retrieved through the API:

- SPEI_START_INPUT_STREAM: indicates the beginning of synthesis.
- SPEI_END_INPUT_STREAM: indicates the end of synthesis.
- SPEI_VOICE_CHANGE: indicates the change of voice used for synthesis.
- SPEI_TTS_BOOKMARK: indicates that a bookmark element was reached in the synthesized text.
- SPEI_WORD_BOUNDARY: indicates that the engine is starting to synthesize a word in the text.
- SPEI_PHONEME: indicates that a phoneme was synthesized by the engine.
- SPEI_SENTENCE_BOUNDARY: indicates that the engine is starting to synthesize a sentence in the text.
- SPEI_VISEME: indicates that the engine encountered a viseme in the text.
- SPEI_TTS_AUDIO_LEVEL: indicates a change in the audio level.
Additional data corresponding to an event are also returned by the engine with each of the events. For instance, word position (index) and length are returned for an SPEI.TTS.BOOKMARK event, and a corresponding text message (the value of MARK attribute of BOOKMARK tag) is returned for an SPEI.TTS.BOOKMARK event.

Thus, the Microsoft Speech engine has the following advantages:

- It supports speech markup language (SAPI TTS markup language). The newest version of the engine also supports SSML.
- It provides an API for manipulating the engine.
- It comes with Windows operating system at no additional cost. Speech SDK can be downloaded separately for free.
- It generates a wide variety of events corresponding to the states of the synthesis process.
- It produces high quality speech.
- It is well-supported. New versions are released regularly.

And the following disadvantages:

- Works only in the Windows operating system.
- Is not open source.

For more information about the Microsoft speech engine refer to [12].
2.2.2 Festival

Festival is an open source speech synthesizer for Unix systems. Festival provides Scheme and C/C++ APIs. SABLE [16, 27] speech markup language can be used to define vocal properties of text synthesized by Festival. The most recent version of Festival was released in July 2004.

Festival’s API does not provide functionality that would allow for easily retrieving synthesis events from the Festival engine.

Advantages of the Festival speech engine:

- Supports SABLE markup language.
- Provides an API for manipulating the engine.
- Is free and open-source.
- Produces high quality speech.

Disadvantages of the Festival speech engine:

- Works only in Unix systems.
- Does not generate speech synthesis events.
- No new releases since July 2004.

A detailed description of the Festival project can be found at [23].

2.2.3 Espeak

Espeak is an open source speech synthesizer for Linux and Windows platforms. Espeak supports a subset of SSML and provides a C API. Various events can be
received from the engine through the API, allowing the client application to react to them. New versions of Espeak with added features and better SSML support are released regularly.

During the work on the project, I discovered a bug in Espeak. I contacted the author of Espeak with a description of it. The bug was fixed in the next version of Espeak. Thus, the high level of Espeak’s support was proved.

Espeak’s API provides functionality for receiving synthesis events from Espeak engine. The engine generates the following events:

- espeakEVENT_WORD: indicates that the engine is starting to synthesize a word in the text.
- espeakEVENT_SENTENCE: indicates that the engine is starting to synthesize a sentence in the text.
- espeakEVENT_MARK: indicates that a bookmark element was reached in the synthesized text.
- espeakEVENT_PLAY: indicates that an audio element was reached in the synthesized text.
- espeakEVENT_END: indicates the end of sentence or clause.
- espeakEVENT_MSG_TERMINATED: indicates the end of synthesis process.
- espeakEVENT_PHONEME: indicates that a phoneme was synthesized by the engine.
Additional information corresponding to each of the events (position in the text, value of the mark attribute, etc.) is also generated and returned by Espeak synthesizer.

To summarize, the advantages of Espeak are as follows:

- Supports SSML.
- Provides an API for manipulating the engine.
- Generates a wide variety of events corresponding to the states of synthesis process.
- Produces high quality speech.
- Is well-supported. New versions and bugfixes are released regularly.
- Works in Both Unix/Linux and Windows operating systems.
- Is free and open-source.

Espeak’s disadvantages are as follows:

- SSML was not fully supported, as of April 2008.
- API does not provide functions for pausing/resuming the engine.

More information about Espeak project is available at [5].

2.2.4 MacinTalk synthesizer

MacinTalk synthesizer is integrated into Mac OS X. MacinTalk uses its own non-XML markup to set various speech properties.
MacinTalk API provides functionality that allows users to receive the following events notifications:

- The synthesizer is about to synthesize a word.
- The synthesizer is about to synthesize a phoneme.
- End of synthesis.
- End of text processing (the synthesis is not necessarily finished yet).
- The synthesizer encountered a sync speech command in the text.

Thus, MacinTalk speech synthesizer has the following advantages:

- Is integrated with Mac OS X.
- Produces high quality sound.
- Is well-supported.
- Generates a wide variety of events corresponding to the states of synthesis process.

And also the following disadvantages:

- Does not support any of the speech markup languages described in section 2.1.
- Is limited to Mac OS.

For more information about MacinTalk refer to [19].
2.3 Speech APIs

Speech APIs should be distinguished from speech engines’ APIs. While speech engines’ APIs are designed for specific speech engines, speech APIs are designed to be used by different speech engines and applications.

The speech APIs reviewed in this thesis are JSAPI (Java Speech API) and Speech Dispatcher.

2.3.1 JSAPI

JSAPI specification was developed by Sun Microsystems. It was designed as a cross platform speech API that can work with a wide variety of speech engines and can be easily used by applications. However, no implementation of JSAPI was provided by Sun. Only a few third-party implementations of JSAPI are available [22, 7].

JSAPI specification defines a number of Java classes and interfaces for speech synthesis and recognition. This thesis reviews the speech synthesis related part of JSAPI.

JSAPI classes reside in `javax.speech` package and its subpackages. The synthesis related classes and interfaces are located in the `javax.speech.synthesis` package.

The main access point to JSAPI’s synthesis functionality is the `Synthesizer` interface. It provides functions for synthesizing both JSML and plain text. Three overloaded `speak` functions can be used for synthesizing JSML strings from different sources, such as `URL` class (Internet URL or local file), `String` class and `Speakable` class. Plain text can be synthesized with the help of the `speakPlainText` function.
of the `Synthesizer` interface. Additionally, the interface provides functions for setting speakable listeners (`SpeakableListener` interface), cancelling speakables scheduled for synthesis, etc. `Synthesizer` interface inherits additional functions from the `Engine` interface: functions for allocating (`allocate`), deallocating (`deallocate`), pausing (`pause`), resuming (`resume`) the synthesizer, and several other functions.

`SpeakableListener` is an interface for the classes that are intended to receive and process events from synthesizers. `SpeakableListener` defines a set of functions, each of which corresponds to one of the events that can be fired by a synthesizer during the speech synthesis process. The following event handling functions are included in the `SpeakableListener` interface:

- `markerReached`: indicates that a marker was reached in the speakable.
- `speakableCancelled`: indicates that the speakable’s synthesis was cancelled.
- `speakableEnded`: indicates that the speakable’s synthesis process has finished.
- `speakablePaused`: indicates that the speakable’s synthesis process has paused.
- `speakableResumed`: indicates that the speakable’s synthesis process has resumed.
- `speakableStarted`: indicates that the speakable’s synthesis process has started.
- `topOfQueue`: indicates that the speakable is on the top of synthesizer’s queue.
- `wordStarted`: indicates that a word in the speakable is about to be synthesized.

Another important JSAPI class is `Central`. It is used to create synthesizers and recognizers. Many other classes and interfaces are defined in JSAPI. Refer to [9] for the complete list and description of the classes and interfaces.
The example below illustrates the basic usage of the classes and interfaces discussed earlier in the section:

```java
import javax.speech.*;
import javax.speech.synthesis.*;
import java.util.Locale;

public class JSAPI_Demo {
    public static void main(String args[]) {
        try {
            // Create a synthesizer for English
            Synthesizer synth =
                Central.createSynthesizer(
                    new SynthesizerModeDesc(Locale.ENGLISH));

            // Get the synthesizer ready to speak
            synth.allocate();
            synth.resume();

            // Synthesize JSML text
            synth.speak("<JSML>Hello World</JSML>", null);

            // Wait until speaking is done
            synth.waitEngineState(Synthesizer.QUEUE_EMPTY);
        }
    }
}
```
// Clean up
    synth.deallocate();
} catch (Exception e) {
    e.printStackTrace();
}

For more information about JSAPI refer to [8, 9].

2.3.2 Speech Dispatcher

Another project that is worth noting is Speech Dispatcher. The goal of the project is to develop a common API for the existing speech engines and, therefore, make it easier for the speech applications developers to develop applications that can use a variety of speech engines.

The speech markup language used by Speech Dispatcher is SSML.

As of May 2008 Speech Dispatcher is available for Unix/Linux systems only and supports the following speech engines:

- Festival
- Flite
- Espeak
- Cicero
- IBM TTS
Epos

DecTalk software

Speech Dispatcher employs client-server architecture: Speech Dispatcher’s main module runs in a server mode in the system, and the speech applications open a connection, which is used to issue commands to the server and receive events from it. Client applications can request to receive the following events:

- SPD_BEGIN: start of the synthesis process.
- SPD_END: end of the synthesis process.
- SPD_INDEX_MARKS: marker encountered during the synthesis process.
- SPD_CANCEL: synthesis process cancelled.
- SPD_PAUSE: synthesis process paused.
- SPD_RESUME: synthesis process resumed.

Speech Dispatcher provides C/C++, Python and Guile APIs. Unlike Java Speech API, Speech Dispatcher’s C/C++ API is not object oriented: it consists of a number of functions for managing the server connection, speech synthesis, events notifications, etc. Events are delivered to the client applications with the help of function callbacks.

The following example, similar to the one from section 2.3.1 in the sense that it also says the “Hello World” phrase, demonstrates the basics of Speech Dispatcher and differences between Java Speech API and Speech Dispatcher’s C/C++ API:

```c
#include <libspeechd.h>
```
#include <semaphore.h>

// Semaphore, which prevents the program thread from
// terminating while the synthesis is still in process
sem_t semaphore;

// Callback function, which is called when
// synthesis process is finished
void synthesisEndCallback(size_t msg_id,
   size_t client_id,
   SPDNotificationType type) {

    // Speech synthesis is done, unlock the
    // semaphore to terminate the program thread
    sem_post(&semaphore);
}

int main(int argc, char *argv[]) {

    SPDConnection *connection;

    // Initialize the semaphore
    sem_init(&semaphore, 0, 0);

    // Open Speech Dispatcher connection, use threaded mode
    connection = spd_open("client_name", "connection_name",}
"user_name", SPD_MODE_THREADED);

// Set data format to SSML rather than plain text
spd_set_data_mode(connection, SPD_DATA_SSML);

// Set synthesisEndCallback function as callback for
// synthesis end event
connection->callback_end = synthesisEndCallback;

// Enable end of synthesis event notification
spd_set_notification_on(connection, SPD_END);

// Say "Hello World"
spd_say(connection, SPD_MESSAGE, "<SSML>Hello World</SSML>");

// Wait on the semaphore until the synthesis is over
// and the event handler unlocks the semaphore
sem_wait(&semaphore);

return 0;
}

More information about the Speech Dispatcher project is available at [18].
2.4 JSAPI Implementations

I uncovered only two nearly complete implementations of JSAPI in the process of working on the thesis. This section reviews these implementations.

2.4.1 Talking Java

The first implementations of JSAPI, called Talking Java, is available from Cloud-Garden [22]. Talking Java works only with Microsoft SAPI compatible engines, and is available for Windows operating system only. Talking Java is not open source, and is only available under commercial license. The latest version was released in 2004.

2.4.2 FreeTTS

Another implementation, called FreeTTS [7], is a free and open source partial implementation of JSAPI, written completely in Java. It uses Flite speech engine. Flite is a lightweight engine derived from the Festival speech engine. The latest version of FreeTTS was released in 2005.

2.4.3 Summary

Neither of the reviewed JSAPI implementations can be used with multiple speech engines, and neither of them has had any recent releases.
CHAPTER 3

PROJECT DESIGN AND IMPLEMENTATION

The previous chapter of this thesis gave an overview of existing speech APIs and markup languages. This chapter discusses the process of selecting a speech API to be implemented and a markup language to be supported by the software library. It also addresses design and implementation decisions made while working on the project. Special attention is paid to designing the library in a way that will allow for a multi-platform implementation that can be used with different speech engines.

3.1 Speech API and Markup Language Selection

The speech library developed in this project had to provide a speech API. Either the speech API could be developed from scratch or an existing API could be adopted. We chose to implement Java Speech API [9], designed by Sun Microsystems, for this project for the following reasons:

1. JSAPI specification completely satisfies the requirements of the project.

2. Software developed by my adviser, Dr. Eitan Gurari, is oriented at JSAPI and is currently using JSAPI implementation called Talking Java [22]. The library developed within this project is intended to be used with Dr. Eitan’s software,
therefore the library needs to provide the JSAPI interface to make the process of transition from Talking Java to the library as smooth as possible.

We chose JSML as a speech markup language for the project because, firstly, Dr. Gurari’s project was oriented at JSML and secondly, JSML is the language used in JSAPI specification. As can be seen from Chapter 2, almost all of the reviewed XML-based markup languages are rather similar. Hence, adding support for another XML-based markup language to the project should be relatively easy.

3.2 Speech Engines Selection

The main criteria in selecting speech engines for the project were the following:

1. Support for an XML-based markup language.

2. Rich set of events corresponding to the states of speech synthesis.

3. Frequently released bugfixes and new versions.

Support for XML languages makes it much easier to implement JSML support: the problem is basically reduced to translating JSML-specific tags and attributes’ values to the tags and attributes’ values of the markup language supported by the engine.

Receiving events from the speech engine is necessary for the JSAPI implementation because JSAPI specification defines a set of events, such as the beginning of the word, marker, etc.

The advantage of using a speech engine with frequently released bugfixes and new versions is obvious.
Based on the criteria listed above, the following speech engines were selected for the project: Microsoft speech engine for Windows OS and Espeak for Linux OS.

3.3 Challenges

Various challenges arose during the course of the project. Most of the challenges were due to the major differences between the speech engines used in the project implementation. This section of the thesis overviews the nature of the challenges and solutions to them. Some of the solutions are described in more detail in sections 3.4 and 3.5.

3.3.1 Multi-Platform Implementation

Developing multi-platform software is an involved task. The software must provide the same user and application interfaces on all of the supported platforms. Also, the software must produce the same results on the different platforms. The main difficulty in fulfilling the abovementioned requirements is that the software may depend on modules and libraries specific to the platforms. In other words, on one of the platforms the software may use one set of modules and libraries, while on another platform it may use a completely different set modules and libraries to implement the same functionality.

The library developed in this project, for instance, uses the Microsoft Speech Engine on Microsoft Windows OS. At the same time it uses Espeak speech engine on Linux OS. While these two engines are very different, the library is expected to produce the same results with either of them.

Sections 3.4 and 3.5 discuss the way the problem was solved in this project.
3.3.2 Inter-Language Communication

This project incorporates code written in two different programming languages: Java and C++. Code generated by the Java compiler is not compatible on a binary level with code generated by any of the C++ compilers. Nevertheless, modules written in Java need to communicate with modules written in C/C++.

An approach which uses Java native methods [25] to solve the problem is discussed in section 3.5.

3.3.3 Differences in Speech Engines’ APIs and Markup Languages

The APIs of the speech engines used in the project are quite different. Firstly, the signatures of the APIs’ functions are specific to each of the engines. Secondly, the sets of the APIs’ functions are not the same. For instance, Microsoft Speech Engine’s API includes functions for pausing and resuming the synthesis process, but Espeak’s API does not have similar functions in its API. Thus, other functions from Espeak’s API must be used to emulate the pause/resume functionality. Similarly, not all the functionality available through Espeak’s API is directly available through Microsoft Speech Engine’s API.

Speech markup languages supported by the engines also differ from one another: the Microsoft Speech Engine supports SAPI TTS Markup Language, while Espeak supports SSML. Moreover, the library needs to support yet another speech markup language: JSML.

A solution that uses special wrappers to hide the differences in engines’ APIs and markup languages is discussed in sections 3.4 and 3.5.
3.3.4 Differences in Speech Engines’ Events Systems

Events are fired by speech engines to indicate various stages of the speech synthesis process. Events are crucial for building interactive applications. The library must produce the same events for a given input with all of the supported speech engines. But the sets of events supported by different speech engines are distinct and may not include the events needed for the library.

This project approaches the problem by using wrappers around the speech engines. These wrappers adapt the events specific to each of the engines to a common set of events. Thus, the differences in engines’ events systems are hidden from the library by the wrappers and the library works with a unified set of events.

3.4 Architecture Overview

As was pointed out in section 3.3.1, one of the main requirements of the project was the ability to use the library on different operating systems and with different speech engines. To satisfy the requirement the library was split into platform/engine independent and platform/engine dependent layers. These two layers communicate with each other through an established interface. High-level architecture of the library in the form of the abovementioned components is presented in Figure 3.1. The figure shows both the components developed in the scope of the project and the third-party components that were used in the project.

As can be seen in Figure 3.1, not all the components of the library are multi-platform, but it is still required that the library can be used on different platforms. The multi-platform feature of the library is achieved by providing separate implementations of platform/engine dependent modules for each of the supported engines.
These modules serve as engines’ adapters (drivers): they translate the interface function calls to the engine-specific function calls, JSML to the engine specific markup languages and engine specific events to JSAPI events. They are similar to the “Adapter” design pattern [26].

3.5 Implementation Overview

The platform independent part of the library was implemented in the Java programming language while the platform dependent parts were written in the C++ programming language. Java native methods mechanism [25] was used to enable communication between Java and C++ modules.
Figure 3.2 displays more details of the functionality of the adapter modules and of the interface between the Java and C++ modules.

The functionality of the native methods and adapter modules can be illustrated with the help of the following example:

Consider a situation where a speech application that uses the library needs to synthesize the “Hello World” phrase with the word “World” spelled out. It corresponds to the `<JSML>Hello <SAYAS CLASS="literal">World</SAYAS></JSML>` JSML string. The application, therefore, executes the following code:

```java
synth.speak(
    "<JSML>Hello <SAYAS CLASS="literal">World</SAYAS></JSML>", null);
```

Now, a corresponding method from the native methods interface needs to be called with the JSML string converted to that of a markup language supported by the speech synthesizer currently used. Conversion of the strings (markup language tags, attributes and attributes’ values) is done in the JSAPI module with the help of the corresponding adapter module. Every adapter module provides a mapping between JSML tags’ and attributes’ names and the speech engine’s markup language’s tags’ and attributes’ names. Adapter modules also provide functions that convert JSML attributes’ values to the speech engine’s markup language attributes’ values.

On Windows platform, using the Microsoft Speech Engine adapter, the JSML string is converted into the `<SAPI>Hello<SPELL>World</SPELL></SAPI>` SAPI string. Thus, the SAYAS tag is converted into the SPELL tag, and the CLASS attribute and its value are converted into empty strings because the SPELL tag in SAPI markup language does not need an attribute.
Figure 3.2: Adapter elements and Java – C++ interface.
On Linux OS, using the Espeak Engine adapter, the same string is converted into the `<SSML>Hello <say-as interpret-as="characters">World</say-as></SSML>` SSML string. Thus, the `SAYAS` tag is converted into the `say-as` tag, and the `CLASS` attribute and its value, `literal`, are converted into the `interpret-as` attribute and the `characters` attribute value.

After the string is presented in the engine’s markup language, the `speakText` native function is called to pass the string to the speech engine through the engine adapter. This call invokes the C++ implementation of the native method. The native method’s implementation, first of all, converts the Java string to the wide-character multibyte string, which can be used in C++ code, and then calls the `ss_speakText_nb` function from the speech engine adapter’s interface. The call to the `ss_speakText_nb` function invokes the function’s implementation from the currently used engine’s adapter. That leads to the call to the corresponding function from the engine’s native API. On Windows platform, the `Speak` function from the Microsoft Speech Engine’s `ISpVoice` COM interface is called. On Linux OS, the `espeak_Synth` function from Espeak’s API is called.

The native functions layer is responsible for allocation and deallocation of string objects that represent the Java `String` class contents in C/C++ code. Many of the native functions’ implementations, when called, allocate the string objects and, therefore, must deallocate them before returning from the call to avoid resource leaks. Deallocating these objects manually is not a trivial task because many of the functions, first of all, have several exit points and, second of all, may stop execution and return due to various exceptions. Thus, it is almost impossible to deallocate the string objects manually at every possible exit point of every function. A special
resource management class [29], \texttt{CStringCharsHolder}, was implemented to manage the string objects. An automatic object of the resource management class is created and initialized with a string object each time it is created. The destructor of the class takes care of deallocating the managed string object. Each time a function returns, all of its automatic objects, including the objects of the resource management class, are destroyed. Thus, no matter how and when a function terminates execution and returns, the resource management class objects created in it are automatically destroyed: their destructors are called and the managed string objects are properly deallocated.

Since the library is designed to work with a number of different speech engines’ adapters (and corresponding speech engines), it is necessary to have a way to specify which speech engine to use. Implementation of the library allows the users to specify which speech engine adapter to use by listing the name of a speech engine adapter in the \texttt{speech_engine} file. The file lists one or more speech engine adapters (one per line). When the library loads, it reads the \texttt{speech_engine} file from the current directory and tries to load the speech engine adapters from the file one by one. As soon as loading of one of the speech engine adapters succeeds, the rest of the lines in the file are ignored, i.e. the library does not try to load any other speech engine adapters.

The following example illustrates the process of loading a speech engine adapter in more detail.

Consider a \texttt{speech_engine} file which consists of the following two lines:

\begin{verbatim}
espeak_engine
ms_engine
\end{verbatim}
Speech application starts and initiates loading of the JSAPI library. The JSAPI library reads the `speech_engine` file’s first line and tries to load the corresponding speech engine adapter. On Linux OS, it will try to load the `libspeech_engine.so` dynamic library. On Windows, it will try to load the `speech_engine.dll` dynamic library. If the dynamic library is found and loaded successfully, the second line of the file is ignored. But if the dynamic library cannot be loaded, the JSAPI library reads the second line of the file and tries to load either the `libms_engine.so` or the `ms_engine.dll` dynamic library, depending on which operating system it is running on. If the dynamic library can be loaded, the JSAPI library loading process finishes successfully and the JSAPI library can be used by the speech application. But if the dynamic library cannot be loaded, the JSAPI library throws an exception which indicates that the process of initializing the JSAPI library failed.

Events corresponding to the states where a synthesizer has reached a marker or has started synthesizing a word are very important for the voice browsers. These events allow the browsers to identify the current position in the synthesized text and in the hierarchical structure of the documents: speech markup markers can be used to define hypertext documents’ structure. Both of the speech engines used in this project support these two kinds of events, making it easy to implement support for the events in the library implementation. The speech adapters simply pass the events to the library when corresponding events are retrieved from the engines.
3.6 Conclusion

Design of the library allows for multi-platform implementation by sharing the multi-platform Java module across the platforms and providing separate implementations of the platform-dependent modules (speech engine adapters) for each of the supported platforms. The library does not include an implementation of a speech engine that would natively support JSML. Instead, it implements software components that “adapt” existing speech engines to the needs of the library. Speech engines used by the library do not have to provide JSML support, though it is important that the engines support a speech markup language sophisticated enough to represent JSML strings. Speech engine adapters enable the use of speech engines with the library by translating JSML and the library’s API calls to the speech engines’ native markup languages and API calls.

Current implementation of the library provides speech engine adapters for the Microsoft Speech Engine (Microsoft Windows OS) and the Espeak speech engine (Linux OS).

The architecture of the library makes it relatively simple to add support for new speech engines by implementing corresponding speech engines’ adapters.

Also, a special mechanism is provided that allows the user to choose a speech engine adapter to be used with the library.
CHAPTER 4

FUTURE WORK

There are certain areas that can be improved and several new features that can be added to the project. This chapter gives detail of possible directions of future work on the project.

4.1 More Complete JSAPI Support

Currently the project does not provide a full implementation of JSAPI. The following features need to be implemented for complete JSAPI support:

- Speech recognition: corresponding JSAPI classes and speech engine adapters’ functionality need to be implemented.

- Support for languages other than English should be added.

To enable support for other languages, first of all, speech engines that support multiple languages must be used. Second of all, support for the locales must be added to the library implementation. Locales can be used to set the current language and instruct the underlying speech engine to produce speech for the language currently set.
As of May 2008, Espeak supports several languages, among them: English, French, Spanish, Afrikaans. Support for some of the languages is still in development. The Microsoft speech synthesizer currently supports English and Chinese.

4.2 Other Speech Engines and Platforms Support

Support for only two speech engines has been implemented at this point in time: Microsoft Speech Engine and Espeak.

The Microsoft Speech Engine is currently used on Windows OS for speech synthesis. It can also be used for speech recognition.

The library uses the Espeak speech engine on Linux OS for speech synthesis. Espeak does not provide speech recognition functionality, therefore a separate speech recognition engine must be used to implement speech recognition support in the Linux version of the library.

Adding support for other speech engines and other platforms, such as Mac OS and, possibly, platforms for mobile devices should also be considered.

4.3 Other Speech Markup Languages Support

Only JSML is currently supported by the library. Adding support for other speech markup languages would be useful for users who prefer markup languages other than JSML.
4.4 Transforming source code to the literate programming style

Transforming the source code of the library to the literate style will help to create comprehensive documentation for the project. It will greatly simplify the future development of the project.
APPENDIX A

COMPILATION AND INSTALLATION

This appendix serves as a guide for the building and installing of the library on Windows and Linux operating systems. It lists all development tools and dependencies needed to build and use the library.

First of all, you will need to download the source code of the library from the following location:

www.cse.ohio-state.edu/manukyan/jsapi-project.zip

After downloading the code, unzip the downloaded file.

A.1 Windows

A.1.1 Building the Library

The library consists of two main components: JSAPI classes implementation (written in Java) and Microsoft Speech Engine adapter (written in C++).

You will need to install the following tools on your system to compile the Java module of the library:

- Apache Ant build system

- Sun Java Development Kit version 6 or above
You also must make sure that the PATH environment variable contains the folders with the tools’ binaries.

To build the Java module of the library, execute the build.bat file in the project’s jsapi_impl directory. The build.bat invokes Apache Ant and passes the project’s build file as an argument to it.

The step produces the jsapi_impl.jar file in either the project’s lib or lib.dbg folder, depending on whether or not the library contains debug information.

To build the Microsoft Speech Engine adapter, you must have Microsoft Visual C++ 8.0 installed on your system. The adapter can be built in either of two ways:

- Launch Microsoft Visual C++, open the project file (ms_engine.vcproj file in the projects’ engine_interface\ms_engine folder) and build the project.

- Execute the build.bat file in the project’s engine_interface\ms_engine folder.

Either of the two options produces the ms_engine.dll file in either the project’s lib or lib.dbg folder, depending on whether or not debug information is added to the compiled file. If you are using the second option for compiling the adapter, you must make sure that the Visual C++ binaries folder is included in the PATH environment variable.

**A.1.2 Installing the Library**

To install the library for a speech application, perform the following steps:

1. Copy the files ms_engine.dll and jsapi_impl.jar to the working directory of the application that uses the library.
2. Create a file named `speech_engine` in the same directory. Add a line with the name of the speech engine adapter to the file: `ms_engine`. The JSAPI library reads the file to determine which speech engine adapter to load.

A.2 Linux

A.2.1 Building the Library

The following tools are necessary for building the Java module of the library:

- Apache Ant build system

- Sun Java Development Kit version 6 or above

To build the module, execute the `build.sh` script in the project’s `jsapi_impl` directory. It launches Apache Ant, which uses the Java compiler to compile the code and create the `jsapi_impl.jar` file in either the project’s `lib` or `lib_dbg` folder. One of these two folders is chosen based on whether or not the library is built with debug information.

The following tools and libraries are needed to build the Espeak engine adapter:

- GNU C/C++ Compiler

- GNU build tools (make, etc.)

- Espeak library with development (C header) files

- Boost Thread library

To build the adapter, open a terminal, change the current directory to the project’s directory `engine_interface/espeak_engine` and execute the `make` command in the
terminal. This step will produce a shared library `libespeak_engine.so` in either the project’s `lib` or `lib_dbg` folder.

### A.2.2 Installing the Library

The library can be installed for a speech application by following these steps:

1. Copy the files `libespeak_engine.so` and `jsapi_impl.jar` to the working directory of the speech application that uses the library.

2. Create a file named `speech_engine` in the same directory. Add a line with the name of the speech engine adapter to the file: `espeak_engine`. 
BIBLIOGRAPHY


