Make your own notes.
NEVER underline or
write in a book.

RHODES UNIVERSITY
LIBRARY
Cl. No. TR 07-19
BRN

Constructing a Low-Cost, Open-Source, VoiceXML gateway

Submitted in fulfilment
of the requirements of the degree
Masters in Computer Science
of Rhodes University

Adam King

March 22, 2007

Abstract

Voice-enabled applications, applications that interact with a user via an audio channel, are used extensively today. Their use is growing as speech related technologies improve, as speech is one of the most natural methods of interaction. They can provide customer support as IVRs, can be used as an assistive technology, or can become an aural interface to the Internet. Given that the telephone is used extensively throughout the globe, the number of potential users of voice-enabled applications is very high.

VoiceXML is a popular, open, high-level, standard means of creating voice-enabled applications which was designed to bring the benefits of web based development to services. While VoiceXML is an ideal language for creating these applications, VoiceXML gateways, the hardware and software responsible for interpreting VoiceXML applications and interfacing with the PSTN, are still expensive and so there is a need for a low-cost gateway.

Asterisk, and open-source, TDM/VoIP telephony platform, can be used as a low-cost PSTN interface. This thesis investigates adding a VoiceXML service to Asterisk, creating a low-cost VoiceXML prototype gateway which is able to render voice-enabled applications.

Following the Component-Based Software Engineering (CBSE) paradigm, the VoiceXML gateway is divided into a set of components which are sourced from the open-source community, and integrated to create the gateway. The browser requires a VoiceXML interpreter (OpenVXI), a Text-To-Speech engine (Festival) and a speech recognition engine (Sphinx 4).

The integration of the components results in a low-cost, open-source VoiceXML gateway. System tests show that the integration of the components was successful, and that the system can handle concurrent calls. A fully compliant version of the gateway can be used in the real world to render voice-enabled applications at a low cost.

Acknowledgements

Most importantly, to my family. Your support, love and sacrifices will never go unnoticed, and have got me where I am today.

To Alf, whose influence and hard work is evident on every page of this thesis. Thanks for all the years of guidance, help and advice.

To Peter, who has provided valuable input and helped shape this project into what it is today. Thank you for the time and effort spent in making my masters studies possible.

To the Computer Science department and associated staff here at Rhodes. It has been pleasure being part of such a great team.

To my peers, and friends, for all the support and friendship, and for providing a valuable place to bounce ideas off.

Finally, to Grahamstown, Rhodes and the Rat, without which I wouldn't be who I am today.

Underlining marks and highlighting ruin books for other readers.

Contents

List of Figures

List of Tables

Glossary of terms

API	Application Programming Interface.
ASR	Automatic Speech Recognition.
ATDM	Automated Telephone Disease Management.
BRI	Basic Rate Interface.
CBSE	Component-Based Software Engineering.
CLI	Command Line Interface.
COM	Component Object Model.
CORBA	Common Object Request Broker Architecture.
COTS	Commercial Off The Shelf.
DTMF	Dual Tone Multi Frequency.
EMMA	Extensible MultiModal Annotation.
FIFO	First In First Out.
FXO	Foreign Exchange Office.
FXS	Foreign Exchange Station.
HMM	Hidden Markov Model.
HTML	HypterText Markup Language.
IAX	Inter-Asterisk eXchange.
IDE	Integrated Development Environment.
IPC	Inter Process Communication.

ISDN	Integrated Services Digital Network.
IVR	Interactive Voice Response.
JNI	Java Native Interface.
JVM	Java Virtual Machine.
LAN	Local Area Network.
MGCP	Media Gateway Control Protocol.
PBX	Private Branch eXchange.
PCM	Pulse-Code Modulation
PDA	Personal Digital Assistant.
PML	Phone Markup Language.
PRI	Primary Rate Interface.
PSTN	Public Switched Telephone Network.
SIP	Session Initiation Protocol.
SRGS	Speech Recognition Grammar Specification.
SSML	Speech Synthesis Markup Language.
TDM	Time Division Multiplex.
TTS	Text To Speech.
UDP	User Datagram Protocol.
URL	Uniform Resource Locator.
VOFR	Voice Over Frame Relay.
VoiceXML	Voice eXtensible Markup Language.
VoIP	Voice over Internet Protocol.
VSP	Voice hosting Service Provider.
WER	Word Error Rate.
WSJ	Wall Street Journal.

XML eXtensible Markup Language.

XP eXtreme Programming.

Chapter 1

Introduction

Voice-enabled services have been around for a while. Due to the improvement in speech-related technologies, they are becoming increasingly popular. The potential to reach vast numbers of users is very large. Telephones are a familiar sight in many homes and businesses worldwide, and provide a relatively inexpensive, reliable means of communication. It follows that a voice-enabled application which is able to interface with the PSTN has the potential to reach many users cheaply and reliably. In addition to this, many people are used to interacting with the telephone, and it is therefore a very familiar means of communication. While full natural language processing is not achievable at present, well designed speech user interfaces can result in satisfied users.

The most common form of voice-enabled applications are Interactive Voice Response (IVR) systems, which traditionally have been created using low-level, platform-specific languages. Newer examples of voice-enabled applications include multimodal applications and information interfaces, both of which require a bridging of the PSTN and IP networks, and applications to assist the visually impaired or the elderly. VoiceXML (Voice eXtensible Markup Language) has been proposed for creating voice-enabled applications. VoiceXML is an open, high-level, XML-based language aimed at creating audio dialogs. Using VoiceXML brings all the advantages of web-based development to the development of voice-enabled applications.

VoiceXML has sufficient functionality to replace, and extend, traditional IVR systems in a standard, portable manner, as well as to create any other voice-enabled applications. These applications, described by VoiceXML, require rendering on a VoiceXML browser, in a relationship very similar to the one that exists between web applications and web browsers. Because of the number of potential PSTN users, these browsers should be capable of interacting with the PSTN. A browser with these capabilities can be referred to as a gateway. These gateways must have specialised hardware to interface with the PSTN, which is very expensive. Consequently, VoiceXML gateway prices can begin at $10 000 USD [37]. Hosting can be considered as a means of reducing costs. Hosted solutions do not require the outright purchase of a gateway; instead lines are hired on other gateways provided by entities known as Voice hosting Service Providers (VSPs) [21]. While this is beneficial, hosting costs are still high, because the VSPs need to recoup the high gateway costs.

There is, therefore, a need for a low-cost VoiceXML gateway, and the purpose of this thesis is to investigate the construction of such VoiceXML gateway. Asterisk, an open-source telephony platform, provides an inexpensive PSTN interface through PC-based hardware, and so gives us the opportunity to create a low-cost gateway. In this thesis we will be investigating the addition of a VoiceXML service to the Asterisk system.

1.1 Aim

Defining a set of goals provides a clear benchmark against which the success of the thesis can be evaluated. As said, the central thrust of the thesis revolves around the construction of a VoiceXML gateway; more specifically, whether or not it is possible to create a VoiceXML gateway which is both inexpensive and effective.

While it may be possible for an individual to create a browser from scratch, it is perhaps an unreasonable undertaking. We will therefore investigate the feasibility of constructing a usable VoiceXML gateway using the Component-Based Software Engineering methodology (Section 1.2.1). Due to time constraints the implemented gateway will not be fully compliant, but will instead be a proof-of-concept system.

Keeping the cost of the browser as low as possible is a major priority, and so other components/systems used in the creation of the browser need to be at least free and preferably, where possible, open-source.

Therefore the research question of the thesis is:

> Is it possible to create a low-cost, open-source VoiceXML gateway using Asterisk as the telephony platform and integrating open-source components to provide the required functionality?

We will answer this question through the actual construction and testing of a such a gateway.

1.2 Methodologies and environment

This thesis will subscribe, to a certain degree and where appropriate, to certain methodologies. The task of creating a VoiceXML gateway can be approached as the integration of two different disciplines, namely Software Engineering and Systems Development. The Software Engineering aspect governs how the system is structured – essentially the evolution of the system's architecture – while actual implementation of the system is a Systems Development problem.

The two methodologies used for the thesis, Component-Based Software Engineering and Extreme Programming will be discussed here.

1.2.1 Component-based software engineering

The concept of Component-Based Software Engineering (CBSE) is not new and was originally proposed by McIlroy at a NATO Software Engineering conference in 1968 [57]. He suggested a change in the software industry whereby a new 'sub-industry' dealing in components would enable the mass production of software, similar to the production of hardware. This new industry would create independent components. In McIlroy's proposal these components were routines of a high quality, which could be used in many different contexts. Software systems would then be created by integrating those routines which best implement the required portions of functionality. In addition to diminishing development times, this process would also result in the creation of software with improved reliability and performance, as the components are written, once, by experts, and then distributed across many different applications [57].

The modern definition of CBSE does not appear to have changed dramatically. It can still be thought of as a building block-based style of development [66], with the additional restriction of having to fall within the constraints of a clearly defined architecture [13]. There is evidence of this industry being active today. Some successful examples of companies selling components are componentsource.com, ilog.com and flashline.com [86]. The total value of components sold in 1998 was $440 million USD.

While improving development time, reliability and performance, CBSE also leads to ease of maintenance [18, 13] and improved flexibility [10].

The definition of a component has evolved somewhat from what was originally proposed as a routine. Cox and Song [18] refer to a component as an independent software artifact which has a clearly defined interface to provide a specific functionality. Cai *et al.* [13] state that a component has three features - it is an independent and replaceable part of a system which provides a specific functionality, forms part of an architecture and communicates with other components via interfaces. Grundy *et al.* [32] define a component as being an extension of an object. Events are used to advertise the component's behavior, and the component is responsible for making public information about its data, methods and events, allowing interaction from other components. Conversely, Szyperski distinguishes between a component and an object [86]. He claims that a component is an independent, executable piece of software which has no externally visible state and which may be created by a third party. Objects, on the other hand, are not independent and executable, are instantiated and have a unique identity.

While these definitions may differ, there are some common themes. A component is an independent, executable piece of software which provides a defined functionality within a well defined architecture, and is replaceable. Components within a system interact through well defined interfaces.

Components can be sourced by two methods, building components from scratch or obtaining pre-built ones. Building components from scratch forfeits any development time gains, but still can provide flexibility, performance benefits, ease of maintenance and the possibility of structuring development teams. In addition to these benefits, purpose built components will best fit the system, not requiring the configuration effort third party components would [86]. Since one of the main motivating factors for using CBSE in this thesis is that its use can significantly reduce development time and simplify the development process, we are interested only in obtaining third party components. Such components are known as Commercial Off The Shelf (COTS) components. When using COTS components the Software Engineering task becomes one of "application assembly" [10]. This means, however, that one is reliant on the creator of the COTS component for long term support [10]. To avoid such vendor lock-in when creating COTS based components systems, it is desirable to create an open system, as described by Oberndorf [67]. An open components based system is one which allows simple interchanging of components. Essentially, it strives for flexibility. This is achieved by using well documented and standard interfaces represented by APIs, protocols or file standards. Open in this context refers to the interfaces being open.

We can now define a component as it will be used for the purposes of this thesis:

Component: An independent, replaceable, executable piece of third-party software with well defined interfaces used to provide a specific functionality within a larger system.

Following a COTS based approach may complicate the issue of component selection somewhat, as there is no guarantee that the components will run on the same platforms or be written in the same languages. While careful component selection may help, it can also lead to a possible sacrifice in quality if the ideal component is not compatible with the system's chosen platform or language. There are, however, a number of CBSE enablers available such as Common Object Request Broker

Architecture (CORBA), Component Object Model (COM) and the subsequent .net framework, and JavaBeans [13, 66], which facilitate component interaction. In addition to these formal methods, techniques such as simple client-server architectures can be used to integrate components.

CBSE is therefore an excellent methodology to facilitate the creation of a VoiceXML gateway. This piecemeal approach can drastically reduce the time and effort required to develop the gateway. In addition, it will result in improved stability and performance. The creation of an open system means that the components of the gateway will be 'plug and play-able'. This allows other developers with a different set of requirements, such as not having a low-cost pre-requisite, to add components of their choice.

Thesis Implications What are the implications of the CBSE approach for our project? Firstly, the VoiceXML gateway requirements have to be determined and analyzed to identify the required components. Next the components themselves must be sourced. Besides having to be supported, be sustained and have open interfaces, the components also have to adhere to the requirements prescribed by the thesis. Namely, they have to be pre-built, COTS, systems which have open standards, are robust and free, and must be able to perform adequately in a real world environment. Succinctly, a COTS based Open-Source system with open standards.

Once the components have been identified and sourced, a simple architecture can be created displaying the component's interaction. Finally, the components can be integrated according to a development methodology.

1.2.2 Agile and extreme programming

Attention to Systems Development methodologies has been shown to be beneficial [26] to development projects still, traditional methods such as the Waterfall method, Unified Process and the Spiral Model have been criticized in part for not being responsive enough to change, being over complex and for treating people as predictable entities [53, 2]. Sommerville [81] discusses the need for Agile methodologies. Traditional 'plan-based' software development methodologies emphasized careful and meticulous planning, formal documentation and a tightly controlled development process. A large effort is therefore required before the actual development process can begin. The rigidity and necessity of documentation was due to the largeness of the projects and the fact that development teams were often disparate and geographically separated. The planning process represents a large overhead, which can overwhelm smaller projects. In addition, because of the pre-development effort, traditional methods do not cater for changes in requirements very well. When change occurs it occurs at a high cost, as the specifications and designs have to be altered to reflect the changes.

Agile methods, as described by Sommerville [81], are considered an alternative to these 'plan-driven', heavyweight processes. These Agile methods all subscribe to the Agile Manifesto [40].

Focusing on the actual creation of software as opposed to design and documentation, Agile methods are capable of reducing both the overhead and the cost of change. These methods are all iterative, with quick development cycles aimed at frequently delivering working software to the customer. Abrahamsson [1] states that a methodology can be considered Agile if it is incremental, cooperative, straightforward and adaptive.

While there are many methods subscribing to the Agile process, there are a number of principles common to each. Users are heavily involved in the development processes, providing and prioritizing new requirements, and testing functionality. Delivery of software is incremental, with each delivery containing new functionality. Emphasis is placed on the people (and their skills) involved in the development process as opposed to developers following a rigid development process. Agile projects do 'embrace change'. In fact, they do expect change and proceed accordingly. Finally, these projects must attempt to maintain simplicity, both in software and the development process.

The perceived lack of structure and discipline involved with Agile methods leads to the criticism that they promote a 'hacking' style of development [48], as discussed by DeMarco and Boehm [22]. Agile methods are also not suited to large teams and projects [53]. Discipline, however, can be incorporated into a project by infusing some plan-driven elements, balancing both Agile and plan-driven aspects in a single project [8]. Agile processes are not recommended for large and/or critical systems, where a thorough understanding of the systems requirements is required [48, 81].

Additionally, Sommerville warns that not all the Agile principles are feasible for all situations [81]. In particular, customer involvement depends on the customer's ability to spend time with the developers, developers may not get along, prioritizing changes may be difficult, and maintaining simplicity may prove to be difficult.

Fortunately, this thesis does not involve a large team and is not safety critical, and so Agile Methods can be employed.

There are a number of Agile methods, including:

- eXtreme Programming
- Scrum
- Crystal
- Adaptive Software Development

- Dynamic Systems Development Method
- Feature Driven Development

Of these, eXtreme Programming (XP) can be considered the most popular [1], and those elements of XP which are deemed advantageous will be employed by this thesis. In fact, field studies [26] suggest organizations are consciously customizing methodologies in a process called methodology tailoring. XP itself recognizes that it does not need to be employed verbatim [1, 6].

Kent Beck described XP in 1999 [6], which he developed while working on a project at Daimler-Chrysler. The XP life cycle comprises of development episodes, which are the "smallest units of scheduling" [6]. Units of functionality, known as user stories, are written on cards. Pairs of programmers implement each user story following a Test Driven Development method [31]. While opinion regarding pair programming is uncertain and divided [34, 63, 94], it cannot be employed for the purposes of this thesis as there is only a single developer.

Once the code passes all the tests, it is integrated into the system and the full suite of tests are run. A refactoring process then takes place, to ensure that the entire system is as simple as it can be and there there is no redundant code. Keeping the design and code as simple as possible gives the system its agility, meaning that the cost of change does not grow exponentially over time, as it does with plan-driven projects. This simplicity also reduces the possibility of introducing errors into the system. The result is that important decisions can be delayed for as long as possible, increasing the chance of them being correct.

The XP development process has four control variables, according to Beck. External forces - including management - and the development team use these variables to control the project. Best practice is to have three variables set by external forces and the fourth set by the developers. The variables are:

Cost	Increasing the cost of the project can reduce development time and improve quality and scope, but Beck warns that too much money can be detrimental.
Time	More time leads to improved quality and broader scope.
Quality	Beck cautions against using quality as a control variable, but sacrifices in quality can reduce costs and time.
Scope	Reducing the scope increases quality while reducing time and cost. Scope is the best control variable to use and should be set by the development team.

For the purposes of this thesis, cost, time and quality are all governed by external forces, and so scope becomes the only available control variable. This form of scope-driven development dictates the development process. The most important functionality has to be implemented first, because should there be a forced change in scope then less important functionality is dropped.

This whole development process, according to Beck, comprises four basic activities, has at its core four values and involves a set of principles and activities. The four basic activities are coding, testing, listening and designing. The four core values are communication, simplicity, feedback and courage. XP's fundamental principles include rapid feedback, simplicity and incremental change. The list of practices which facilitate XP is relatively long, however those that have relevance to this thesis are small releases, simple design, continuous testing, refactoring and continuous integration.

Thesis Implications Employing methodology tailoring we can extract those aspects of XP which are of benefit, bringing structure and discipline to the development process. While formal customers do not exist to provide user stories and test the system, members of the research group can be used to fill their role. In addition to the research group, the author performs both the role of developer and user. Disassembling the system into units of functionality required by the CBSE approach provides clearly defined, clearly separated episodes of development. Integrating a new component provides additional functionality and hence represents a user story. Scope-driven development determines the order of integration.

The integration of each component will represent a complete development episode. Tests are created before any implementation is started, and once the tests are passed integration and refactoring take place. Creating the tests can be as simple as creating VoiceXML applications. Unfortunately, however, these tests can not always be automated. Testing voice recognition, for instance, requires human involvement.

Keeping in line with the XP paradigm, as well as open CBSE practices, the design of the system will be kept as simple as possible, minimizing the cost of change, reducing the potential for errors and emphasizing 'plug and play-ability'.

1.2.3 Environment

Once the question of how to proceed has been answered, the question of the development environment can be addressed. This involves the selection of platforms, applications and IDE's or any other similar tools.

Firstly, the question of which platform to use is determined by the selection of Asterisk as the primary telephony platform. Since Asterisk is Linux based, we will be developing specifically for the Linux platform. Initially, therefore, we need a Linux machine, Asterisk and a C compiler, as Asterisk is written in C. The development machine runs Gentoo Linux with the 2.6.11.10 kernel, gcc 3.3.4 and Asterisk version 1.0.6. gcc is an open-source C and C++ compiler, amongst other languages. Java was also used in the development process. In particular Sun's JDK1.5 and its compiler, javac, are used.

For the purpose of creating and editing source code, the text-based editor, Vim, has been used. This provided a number of advantages. Firstly, it allows simple, remote, platform independent editing of the code under development, and secondly Vim provides syntax highlighting for, amongst others, C, C++ and Java, which was found to be a great aid in the development process.

So, armed with a Linux platform, a text editor, some compilers, a version of Asterisk and a methodology, the task of creating a VoiceXML gateway can get underway.

1.3 Document overview

Chapter 2: This chapter presents a review of work related to this project and background information. The concepts of voice-enabled applications, VoiceXML and Asterisk are defined and explained.

Chapter 3: This chapter explores the selection of components and the process undertaken before implementation could commence. The collection of components is important, as correctly selected components can greatly ease the task of implementation.

Chapter 4: The task of implementation is broken into two chapters. This chapter deals with extending Asterisk's functionality, using the components selected in Chapter 3, to include a VoiceXML interpreter and audio output capabilities.

Chapter 5: This chapter covers the second half of the implementation effort - adding audio input functionality to the system.

Chapter 6: In this chapter a summary of the completed work is presented and related to the project's aims. Future work is also discussed.

Chapter 2

Related work

This chapter presents a review of work related to this project in order to establish a clear understanding of all the relevant concepts.

Firstly we will show the potential of voice-enabled applications and their popularity in telephony, justifying the need for a low-cost VoiceXML gateway.

Secondly, since we are proposing to use VoiceXML to create these applications and are attempting to build a gateway to execute them, a thorough understanding of VoiceXML, a VoiceXML application and how VoiceXML is interpreted is required. Besides clarifying concepts and improving understanding, this process will also allow us to draw up a set of requirements and divide a gateway into a set of components.

Finally, we will describe the Asterisk system. While Asterisk is a component of the system, it has been selected at the outset, and provides the base upon which a VoiceXML service will be built. So before we look into component selection we need to discuss the Asterisk system. In particular, we will describe the use of Asterisk as a PSTN interface and the available methods of extending Asterisk's functionality.

2.1 Voice-enabled applications and IVRs

The notion of using voice as a mode of human computer-interaction has been around for a number of years. The first full English speech synthesis system was developed in Japan in 1968 [77], and the Hidden Markov Model (HMM), used by many speech recognition systems today, was initially proposed in the late 1960s [75]. In 1971 IBM developed its first speech recognition system, which was used by engineers to talk to a computer [35]. Applications which incorporate these speech technologies are called voice-enabled applications:

Voice-enabled Application: Any application which is accessed through an audio channel and uses voice in the form of voice recognition, text synthesis or pre-recorded audio as a form of input and/or output.

Voice-enabled applications are not limited to interaction between a human and a computer, though. Perhaps the most common example of a voice-enabled application is the traditional Interactive Voice Response (IVR) system, which has been described as a system where users interact with a pre-recorded script using DTMF input [17]. Today IVRs are evolving to include features such as voice recognition. In fact, voice recognition has been shown to improve both performance [85] and customer satisfaction [3] in an IVR system.

IVRs have been applied to many different areas, from call centers to the health care industry and data collection. They are able to provide continuous, automatic access to information at a low cost per call for the user [17].

In the corporate environment IVRs are used by companies as an extra service option for users [84] and as a means of creating a business advantage by providing better access to information for customers [3]. IVRs are also used in call centers to reduce costs, by reducing agent-user interaction time. It is estimated that more than 70% of all business interactions are conducted via call centers, and that agents represent about 70% of a call center's costs [11]. Hence, the correct use of an IVR in the call center environment can result in large savings.

The use of IVRs in the health industry has been researched extensively. IVRs can be used to focus on health related behavior, informing and advising patients, aiding diagnosis, monitoring disease or counseling [38]. Using IVRs as reminder services for patients has been shown to be effective [19]. Studies have also shown that Automated Telephone Disease Management (ATDM) can be an effective form of health care [72]. In a survey amongst low income diabetes sufferers where ATDM was used as part of their care, 85% of respondents were satisfied with their calls, and 76% said they would choose to use ATDM in the future [71].

Another important application of IVRs is as an aid in data collection surveys. Computer aided data collection is now the major means of survey research in the USA [87]. Using IVR systems to conduct interviews over the telephone is an efficient, low-cost means of gathering data. Surveys conducted over the phone using an IVR system have been found to be preferred to web-based questionnaires [29], and produce more honest input [58, 87]. While the process of recording audio prompts can be long and expensive, it has been shown that speech synthesis can be used as a replacement without

Revenue Model	Description	Viability
Advertising	Revenue is generated from advertisers on voice portals	Low
Fee based	Revenue is generated from monthly or yearly fees charged for a service	High
Premium rate	Revenue is generated from the amount charged for a user to make a call	Medium (suitable for specialist services or small markets)
Referrals	Revenue is generated from companies referred by voice portals	Low
Selling over the phone	Revenue is generated from orders placed over the phone and savings in customer service agents charges	High - (Dependent on integration with existing ecommerce systems and voice verification systems)
Development & outsourcing	Revenue is generated by developing and hosting an enterprise voice enabled application for a third party	High
Content provision	Revenue is generated by selling on aggregated or specially prepared content to voice portal providers	Medium - (Dependent on the continuing success of the voice portal market)

Table 2.1: Vcommerce revenue models and their chances of success as described by Duggan [23]

affecting the quality of the answers [58], driving survey costs down even further. It has been found, however, that more people are likely not to complete IVR-based surveys [58, 29, 87].

While IVRs have been shown to be versatile, caution must be exercised when implementing an IVR system, as poor user interface design can lead to customer dissatisfaction. Poorly designed interfaces can be a hindrance and a source of frustration, causing users to opt for a live agent at the first opportunity. The design of an IVR dialog is therefore critical to the success of any IVR-based system [84, 93, 39].

In addition to enhancing IVRs, the field of voice-enabled applications is expanding as speech related technologies improve [16]. The increasing popularity of mobile devices such as PDAs and smart phones is one of the drivers of this process. The interfaces for these devices are not always adequate or convenient [92, 64], screens are small and input devices are inadequate. As users become increasingly mobile, they require that their information also becomes mobile and the most natural and unobtrusive means of interacting with information is by using voice. Multi-modal applications, applications which have both visual and aural interfaces [42], are becoming a popular means of information interaction on these mobile devices. Other examples of voice-enabled applications include audio interfaces to web sites, databases and personal computers.

The concept of a voice enabled web provides the opportunity to generate revenue. While speech technology remains immature and continues to evolve, these revenue opportunities will as well [23]. However, a number of models have already been already been used, successfully and unsuccessfully, as shown in Table 2.1.

Aside from bringing the web to telephones, voice services are being created to allow telephone users remote access to their personal computers. These services are aimed at providing users unlimited access to personal desktop applications such as e-mail and calendars [70, 78].

A final example of the use of voice-enabled applications is as an assistive technology for both older adults and those with visual or motor disabilities [39], or when interacting directly with a device is impossible, such as when driving a car [64]. Voice-enabled applications can be used to aid interaction between elderly people and computers and as an efficient reminder service [96]. Similarly, voice-enabled applications provide visually impaired people with a mode of interaction with a computer.

The use of VoiceXML, designed to create audio dialogs, can enhance voice-enabled applications, as it separates the application logic from the interface itself. VoiceXML is becoming one of the major IVR languages [88], is being used as a bridge between the web and the PSTN [80, 62], and to create assistive services [61]. The following section describes, in more detail, the VoiceXML language.

2.2 VoiceXML

This section describes the origins of VoiceXML, looks at the specification, at how VoiceXML applications are created and interpreted and at the advantages associated with VoiceXML development.

2.2.1 Origins and definitions

VoiceXML is an open, XML-based standard which can be used to describe speech enabled applications [9, 5]. It provides a means of creating audio dialogs utilizing speech synthesis, pre-recorded audio, DTMF recognition and speech recognition [56]. While it has also been described as HTML for the telephone [5, 59], it is important to note that it is not only restricted to the telephone, as any audio channel will suffice. The first specification, VoiceXML 1.0, was released in 2000. The primary goal of VoiceXML is to bring advantages associated with web based development to IVRs [56]. The current version of the standard is 2.0 [56], which was released in March 2004. Due to the popularity of the 2.0, standard the W3 released VoiceXML 2.1 [68], which defined a set of features (see Appendix C) not initially included in 2.0 but commonly found in commercial 2.0 implementations [68].

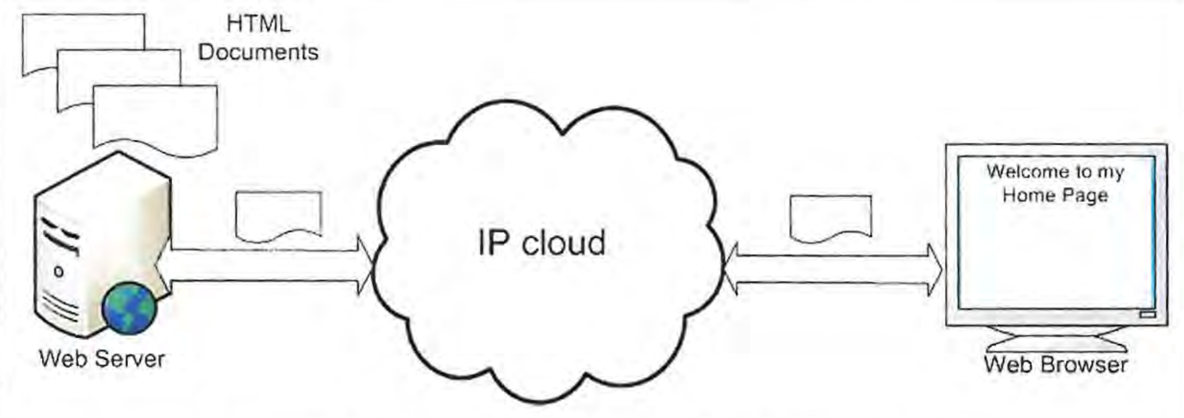


Figure 2.1: A web browser requesting and displaying a web page

The VoiceXML architecture is very similar to that of web based applications: a VoiceXML interpreter is analogous to a web browser, although the interaction with the user is aural as opposed to visual [20]. We can therefore begin by briefly looking at a simple web transaction, a web browser requesting and displaying a web page, shown in Figure 2.1.

Web applications reside on a web server. When a user requests to view a page in a browser, the browser is responsible for fetching the page from the web server and rendering it. The user is then able to interact with the web page via the browser. The HTML document contains no platform specific information, and so any HTML compliant browser on any platform is able to process the application. This abstraction brings with it a number of advantages. The HTML application is portable across many platforms. It is the browser's responsibility to ensure that the application is rendered correctly, not the application's. The application developer is only concerned with application logic at a high level, easing the development and maintenance effort. Finally, because HTML is popular and standard, many tools have been created to assist in the development process.

These features of the web-based model were non-existent in the voice application arena. Languages and applications were low level and specific. This led, in 1995, to attempts to ease the task of creating voice-enabled applications. These attempts are at the origin of VoiceXML, as discussed in the VoiceXML specification [56]. In 1995 AT&T designed an XML-based language, named Phone Markup Language (PML), aimed at simplifying their speech recognition applications. The PML idea was picked up by Motorola and Lucent. By 1998, at a W3C-hosted voice browser conference, AT&T and Lucent had separate implementations of PML and there existed a number of other voice browser languages created by other organizations. These organizations then decided to work together to create a voice browser language, and so AT&T, Lucent, Motorola and IBM created the VoiceXML Forum. The result was VoiceXML, which has been accepted as a W3C standard.

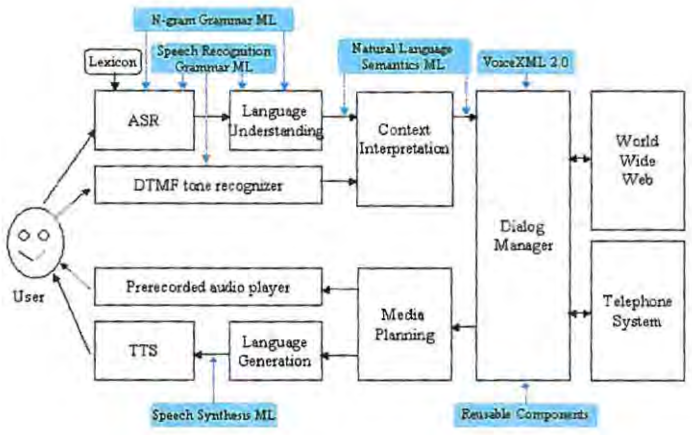


Figure 2.2: The W3C Speech Interface Framework [90]

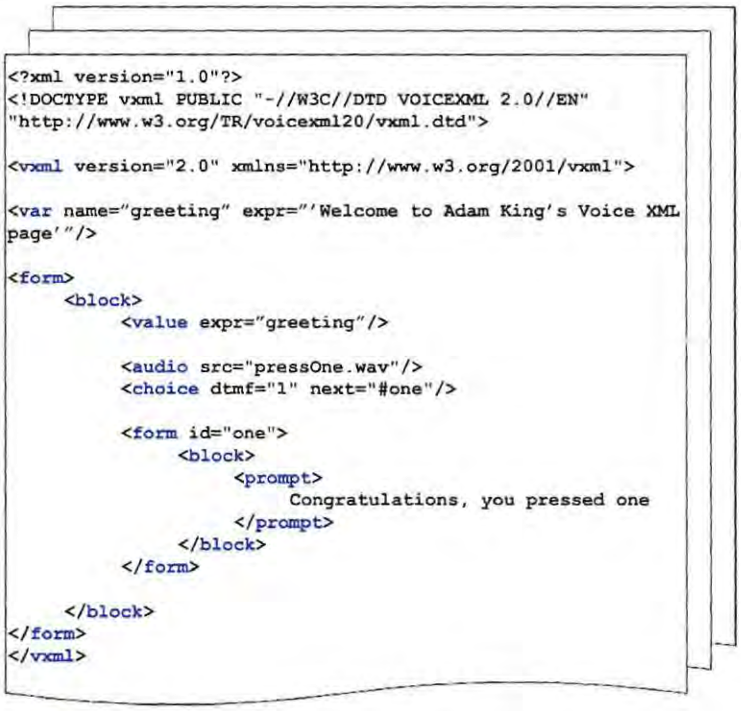
Subsequently, the W3C's Voice Browser Working Group defined a number of standards aimed at supporting speech applications. These fall under the Speech Interface Framework [90], which is a set of open standards that can be used to create speech and natural language applications [45]. Speech applications created using the Speech Interface Framework are rendered by a voice browser. A voice browser is hardware and software capable of interpreting voice markup languages so that it can generate voice outputs and interpret voice inputs [90]. Beasley *et al.* describe a voice browser as an application residing in the network that is able render VoiceXML applications using speech, telephony and Internet components [5].

The Speech Interface Framework can be seen in Figure 2.2. The rounded blue boxes represent data conforming to one of the standard Speech Interface Framework specifications. The arrows show the flow of inputs and outputs, and the white boxes represent the components of a typical speech application.

One can see that VoiceXML is the standard means of defining the Dialog Manager. It is also apparent that there are four modes of I/O: Automatic Speech Recognition (ASR), DTMF tones, Pre-recorded audio, and Text To Speech (TTS) synthesis.

The Speech Recognition Grammar Specification (SRGS) [12] is used to specify the grammars used to determine user input. Specifically, VoiceXML grammars represent the set of words and/or DTMF inputs the application 'understands'.

Speech Synthesis Markup Language (SSML) [91] is used as a standard means of defining input for text synthesizers, ensuring that the input adheres to a pre-defined set of rules, thus allowing good



```
<?xml version="1.0"?>
<!DOCTYPE vxml PUBLIC "-//W3C//DTD VOICEXML 2.0//EN"
"http://www.w3.org/TR/voicexml20/vxml.dtd">

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml">

<var name="greeting" expr="'Welcome to Adam King's Voice XML
page'"/>

<form>
     <block>
          <value expr="greeting"/>

          <audio src="pressOne.wav"/>
          <choice dtmf="1" next="#one"/>

          <form id="one">
               <block>
                    <prompt>
                         Congratulations, you pressed one
                    </prompt>
               </block>
          </form>

     </block>
</form>
</vxml>
```

Figure 2.3: A simple 'Press One' VoiceXML application

performance. SSML is also be used to provide additional linguistic information, such as emphasis, to synthesizers [36].

A simple VoiceXML 2.0 example is the 'press one' IVR , shown in Figure 2.3. This simply synthetically welcomes a user before prompting the user to press the DTMF key corresponding to the digit '1'. If the user presses it, he is congratulated. While the application is very simple, and has no real purpose, it is an example of the "press one for sales, press two for inquiries" structure that is so common in IVRs. The tags in blue are VoiceXML tags, a full list of which can be seen in Appendix A, and the black tags are XML tags. The IVR includes examples of speech synthesis, pre-recorded audio and DTMF input. The speech synthesis is represented by a SSML document describing the prompt, which is passed to a TTS engine. A simple grammar is used to determine what action to take should the user press one. It is important to note the difference between recognizing a DTMF tone and matching input against a grammar. Recognizing a DTMF tone simply involves the representation of a DTMF tone as a token, which is then matched against a grammar. Grammars deal exclusively with tokens, and are not responsible for converting audio signals into tokens.

The application begins by synthesizing the text "`Welcome to Adam King's Voice XML page`", which is expressed as a variable. Once the text has been synthesized, an audio file , `pressOne.wav`, is played. This file asks the user to press one, and if the user presses one the text "`Congratulations, you pressed one`" is synthesized, and the application exits. If the user does not press one, the application exits once the audio file is finished playing.

We can see that any VoiceXML browser wanting to render this IVR by implementing the Speech Interface Framework has be be able to interpret the specified markup languages and provide semantic action (playing pre-recorded audio, synthesizing speech, 'understanding' utterances) accordingly.

We can now define a VoiceXML browser for the purposes of this thesis:

VoiceXML Browser: Hardware and software residing in the network that is capable of rendering VoiceXML applications which interact with a user in a standardized manner via synthesized speech, pre-recorded audio, DTMF and speech recognition over an audio channel.

The concept of a VoiceXML gateway can also be defined now, as it is a specialised form of VoiceXML browser:

VoiceXML Gateway: A VoiceXML browser that has at least one of its audio channels interfaced to the PSTN.

Now that the terms VoiceXML, VoiceXML browser and VoiceXML gateway are understood, we can investigate how VoiceXML works.

2.2.2 VoiceXML in detail

Before a VoiceXML gateway can be constructed, a complete understanding of the specification is required. This process defines the requirements of the gateway, which are used to identify units of functionality, and so help to disassemble a gateway into its components. This whole section is an overview of the VoiceXML specification and is based on [56]. The VoiceXML architectural model, the gateway requirements and the application and interpretation processes are all discussed.

2.2.2.1 Architectural model

The architectural model described by the specification can be seen in Figure 2.4. There are four components: the document server, the interpreter, the interpreter context and the implementation platform. The interpreter, interpreter context and the implementation platform represent a VoiceXML gateway as it is defined for the purposes of this thesis. The document server can regarded simply as a web server. The interpreter deals solely with the VoiceXML language itself, essentially VoiceXML

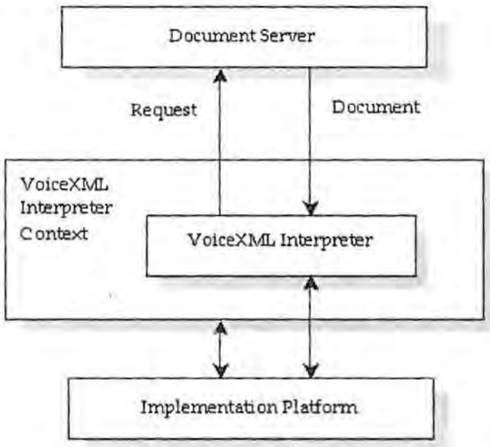


Figure 2.4: The VoiceXML architectural model, as described by the VoiceXML 2.0 specification [56]

elements, while the interpreter context deals with those events which are not part of the VoiceXML language, such as an incoming telephone call. Any action, receiving of input or playing audio files, is the responsibility of the implementation platform.

Communication using events is described by the VoiceXML event model, which is explained by Turner [88]. The event model is hierarchical, with four different levels to handle events: the platform level, the application level, the form level and the field level. An application is made up of forms, which in turn consist of fields. Therefore, field event handlers are the most specific and platform event handlers the most general. When events occur, control is transferred to the corresponding event handler.

Using the IVR seen in Figure 2.3 as an example, we can see how the architecture components interact. The gateway receives a request from the user who wishes to listen and interact with the IVR. This request event is thrown by the interpretation platform and is received by the the interpreter context, as the request event is not part of the VoiceXML language. The interpreter context then requests the relevant VoiceXML document from the web server. The implementation platform is responsible for fetching the file from the server using HTTP [25]. Once the document has been fetched, it is handled by the interpreter. The interpreter generates the events to synthesize text as well as a request to fetch and play back the audio file, `pressOne.wav`. It is the responsibility of the implementation platform to honour these requests. Once the audio prompt has been played, the implementation platform passes any input it receives to the interpreter. Should the implementation platform not receive any input within a pre-defined time, it will pass a time-out event to the interpreter context.

2.2.2.2 Implementation platform requirements

As one can see from the previous scenario, the implementation platform is responsible for providing the voice gateway's functionality. The platform's requirements are listed by the specification as:

1. Document acquisition
2. Audio output
3. Audio input
4. Transfer

The implementation platform must retrieve documents requested by either the interpreter or the interpreter context using HTTP.

It is also responsible for providing audio output support. This involves synthesizing and outputting TTS and playing audio files identified by a URI. The platform must at least support both mu-law and A-law raw and mu-law and A-law WAV formats, although other formats may also be supported.

With regards to audio input, the platform is responsible for detecting and reporting both spoken and DTMF input as well reporting input detection events. Grammar support for inputs must be provided via support for SRGS, including multiple and dynamic grammar matching. N-best recognition, one or more possible recognition results with their associated confidence scores [24], must also be supported. Actual recognition, in other words converting the audio into symbols, is the responsibility of the platform. Additionally, the platform is required to record audio in any of the supported formats.

The final requirement of the implementation platform, as required by the specification, is that it must provide the functionality to transfer connections, for example, transferring a call.

Since we are creating a VoiceXML gateway, the further requirement of PSTN interfacing capabilities can be added.

2.2.2.3 Applications

A VoiceXML application is a set of one or more VoiceXML documents which contain one or more *dialogs*. Dialogs are either *menus*, which present the user with a set of options, or *forms*, which, similar to HTML forms, provide information and can collect values. Users transition between dialogs and can only ever be within a dialog. The first document of a VoiceXML application is known as the root document. When the gateway begins executing the application, the root document is loaded into the interpreter and, unless otherwise stated, execution begins at the first dialog. The root document remains loaded for the duration of the user's interaction with the application, which effectively means that any variables defined in the root document are global, and apply for any child documents. Each dialog can have grammars associated with it, and these grammars are activated when the dialog is entered. An active grammar is one which is actively waiting for input. This includes any grammars associated with the particular input item, including those grammars specified by a *link* (See Appendix A) element, the grammars, and linked grammars, of the form and document, and those grammars and linked grammars of the root document.

The example seen in Figure 2.3 is a simple VoiceXML application consisting of a single document containing a single form dialog, which in turn contains a single subdialog, in this case another form. Subdialogs are used to represent a new interaction within a dialog, and return to the calling dialog. Execution of the application begins at the first, and only, form element in the document, as no specific dialog is specified. The active grammar will, if the user inputs a 1, transfer the user to the subdialog, which, once the prompt is played, will return to the parent dialog. At this point, or if the user did not press 1, the application exits, as no other dialogs are specified for transition.

This IVR is a small example of a machine directed application, which is an application where the only active grammar is the one belonging to the dialog the user is in. The alternative, when the list of active grammars includes those linked to the input item or those active in the root document, is known as a mixed-initiative application.

If, during a mixed initiative application, input is received during the execution of dialog A which matches an active grammar in dialog B, execution is transferred to dialog B as if the input was received while dialog B was executing. This functionality can be used to add power and flexibility to the application. Help functions of varying scope can be created, for instance, and the navigation of an applications can be greatly improved. Traversing many layered menus in a traditional IVR means entering each menu and selecting the next menu, while in a mixed initiative application one can simply select the exact menu.

Now that the architecture has been described and an application has been defined we can look at the actual interpretation of a VoiceXML document.

2.2.2.4 Form Interpretation Algorithm (FIA)

The interpretation process is governed by the Form Interpretation Algorithm (FIA), which is described by the VoiceXML specification [56]. The FIA is responsible for driving input and output between the user and a dialog [20]. Any activity related to the form is controlled by the FIA, including initialising the form, prompting the user and controlling the grammars. Essentially, the FIA is responsible for selecting a form item from the form to interpret, interpreting it while collecting inputs and events, and performing the required action. There are two types of form items, input items and control items. Input items gather input from the user and control items contain a block of VoiceXML code to be executed. There are five input elements:

1. `<field>`
2. `<record>`
3. `<transfer>`
4. `<object>`
5. `<subdialog>`

The two control elements are:

1. `<block>`
2. `<initial>`

The FIA has four phases: an INITIALIZATION phase, after which it enters a loop which contains the SELECT phase, the COLLECT phase and the PROCESS phase.

The INITIALIZATION phase occurs when a form is entered and is responsible for initialising prompt counters and variables. Each prompt has a counter which counts the number of times a prompt has been played since the form began executing. Any variable that is within scope is initialized. If the *expr* attribute of a variable is specified, the variable is initialized to that value, otherwise the variable is initialized to undefined. (The *var* element requires a *name* attribute and an optional expr attribute specifies the value of the variable. See Appendix A).

Once the INITIALIZATION phase is complete, the main loop is entered. If the user has entered the current form because a grammar belonging to the current form has been matched during another form's execution, the loop begins at the PROCESS phase, as there is already input on which to act, otherwise the loop begins at the SELECT phase. The loop repeatedly selects a form item and interprets it, running until the FIA interprets a *transfer* of control or *exit* statement or when there are no more form items left to interpret, at which point the application exits.

The SELECT phase is responsible for selecting the next form item to interpret. This will either be the item specified by a *goto* element, or the next available item in the form.

When the next form item has been selected, the COLLECT phase is responsible for collecting input and events for the PROCESS phase and queuing prompts. The COLLECT phase queues only those prompts that are to be played before the next COLLECT phase is entered, and queues them in order of their execution. The execution of a prompt results in audio being played to the channel. The prompt queue will always contain only those prompts which can expected to be played (barring, say, a hangup) and in the correct order of execution. The PROCESS phase deals with any input collected in the COLLECT phase.

The FIA will execute the next iteration of the loop unless a transfer or exit statement has been interpreted.

Using the example in Figure 2.3 we can observe the FIA at work.

When the document is loaded, the FIA begins with the INITIALIZATION phase. This will reset the prompt counters and initialise the `greeting` variable to '`Welcome to Adam King's Voice XML page`'.

Once INITIALIZATION has been completed, the FIA enters the main loop at the SELECT phase. Since there is no goto element specified from a previous iteration (this is the first iteration) the SELECT phase selects the first available form item, in this case the first block item. Once the form item has been selected, the SELECT phase is complete and the FIA continues with the COLLECT phase.

The COLLECT phase first queues any prompts in the selected form item's scope. In this example it will be the initialized `greeting` variable, which is to be synthesized, and the audio file `pressOne.wav`. When the prompts are queued, their prompt counters are also incremented. The COLLECT phase then activates any grammars and waits for input or a time out. "Activating the grammar" in this example means that the application will match any DTMF 1 input. The final task of the COLLECT phase is to execute the form item. Since the selected form item is a block element, this means executing the block's content, in this case rendering the two prompts.

The FIA will only proceed with the PROCESS phase if the COLLECT phase managed to collect any input. If no input has been received during the COLLECT phase the loop iterates, the SELECT phase will not able to find another form item and so the application exits. Should there be input, however, the PROCESS phase can process the input. If the input is DTMF 1 the choice element is considered filled and execution transitions to the `one` form, where the FIA will select the block element, queue and play the prompt and exit. If input has been received but does not match an active grammar a `nomatch` event is thrown.

2.2.2.5 Summary

We now have a complete insight into VoiceXML documents and the process of rendering them. This allows us to define requirements for the functionality in a gateway, and help the task of separating a VoiceXML gateway into its components. The three architectural components of a gateway are the interpreter, the interpreter context and the implementation platform. These interact using standards defined in the Speech Interface Framework to render VoiceXML applications. Creating a gateway requires at least a PSTN interface.

A VoiceXML application consists of one or more VoiceXML documents which consist of one or more dialogs. The interpretation of these dialogs is done by the FIA, which reacts to inputs or events and generates requests for outputs. The task of capturing inputs, generating events and honoring the FIA's requests for output falls on the implementation platform. This, for example, is what actually plays audio on a channel, or receives input from a channel.

We can now define the task of creating a gateway in terms of a number of smaller tasks. Firstly, a VoiceXML interpreter is required. This is an implementation of the FIA. Secondly, an implementation platform which interacts with the interpreter is needed. The platform must be able to fetch documents from a web server, listen for voice and/or DTMF inputs, synthesize speech, play audio files of a certain format on an audio channel and transfer connections. Finally, these inputs and outputs must be rendered on a PSTN interface.

2.2.3 Advantages of VoiceXML

As we have seen, VoiceXML is very similar to HTML, both in terms of appearance (both are 'tagged' languages [59]) and in the fact that both follow the client-server model [41].

The VoiceXML equivalent transaction to the one seen in Figure 2.1 can be seen in Figure 2.5. We can see that the server side of the transaction remains unchanged. Those web servers which are able to serve XML content are able to host the VoiceXML applications, and serve requests to the VoiceXML gateway, which may or may not represent a gateway to the PSTN. A user can use any audio device to interact with the VoiceXML application via an audio channel. This audio channel includes telephony networks such as the PSTN, GSM networks and VoIP protocols.

The similarity in architectures results in similar advantages. VoiceXML applications are portable, high-level applications which have many tools associated with ease of development. Because VoiceXML is a high-level language it is relatively easy to learn, potentially lowering development and maintenance costs.

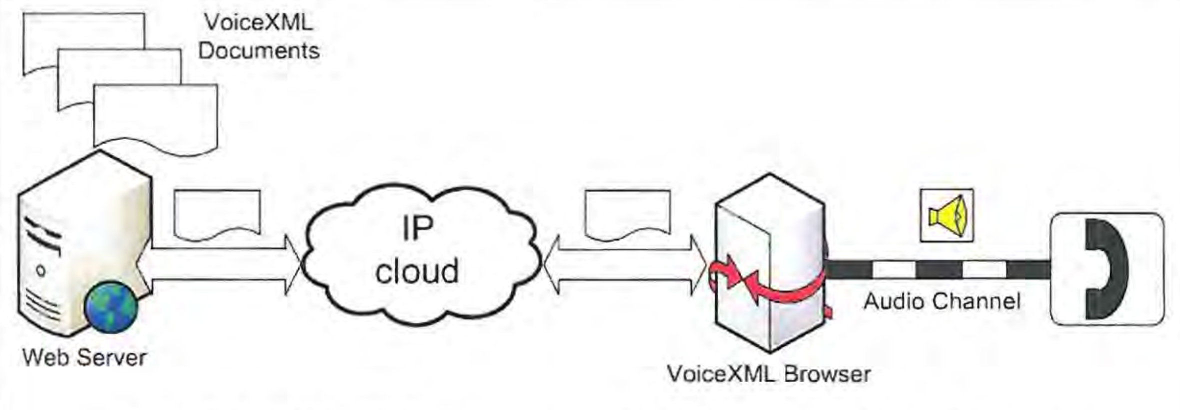


Figure 2.5: A VoiceXML gateway requesting and rendering a VoiceXML page

VoiceXML provides the functionality to completely replace the traditional development of IVRs, bringing the advantages of the VoiceXML model to the IVR domain. It can replace the low-level, platform-specific, proprietary applications which have been traditionally been used to create IVRs. Since it is an open standard, VoiceXML can be expected to work on any compliant platform, meaning that organizations can change platforms without having to change any existing voice applications [4]. Additionally, a VoiceXML application residing on a corporate web server has the same access to corporate databases as the web site, which can be accessed using familiar, mature web technologies [9].

These advantages are also enjoyed when using VoiceXML to create other voice-enabled applications. Ease of development and portability means that creating assistive applications becomes a simple task. The fact that VoiceXML gateways bridge the IP and PSTN networks means that VoiceXML is an ideal language for the creation of audio information interfaces.

2.2.4 Summary

Being an open, high-level web-based standard means that VoiceXML enjoys many advantages over the traditional means of IVR creation and is therefore an ideal means of replacing IVR systems as well as creating other voice-enabled applications. VoiceXML gateways are entities residing in the network which interpret these applications. While a browser in itself may be inexpensive, a connection to the PSTN is essential, and the resulting combination is currently very expensive.

A VoiceXML gateway can be divided into three parts, the interpreter itself, the interpreter context and the implementation platform, which is responsible for any audio on the channel, amongst other things. The implementation platform requires a text-synthesis system, a speech-recognition engine and the ability to fetch documents from a web server. The interpreter uses the Form Interpretation Algorithm to interpret VoiceXML documents.

The final section of this chapter discusses the Asterisk system, into which a VoiceXML service will be integrated.

2.3 Asterisk

Before we can proceed there are a number of things we need to know about Asterisk. Firstly, we need to understand what it is and what is it capable of. Secondly, how it can provide an inexpensive PSTN interface, and finally, how it is possible to extend Asterisk to include extra functionality. Specifically, how does one begin going about adding VoiceXML functionality to Asterisk. This section addresses these issues, and provides a little discussion on VoIP, which will also be supported by the VoiceXML gateway.

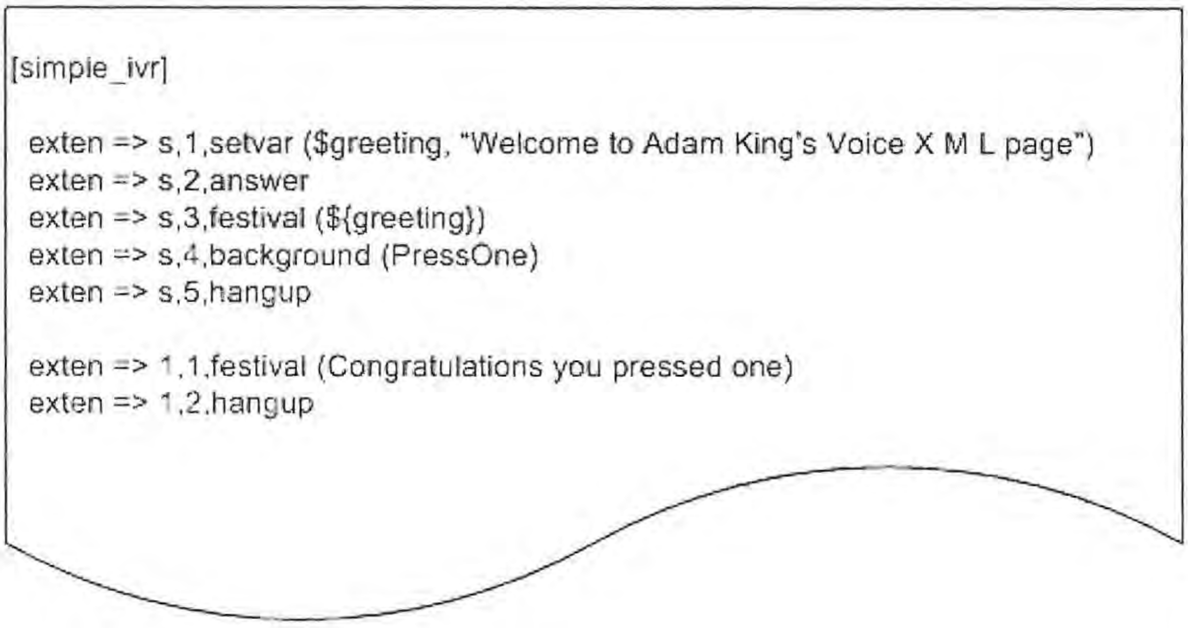
As stated previously, Asterisk is a pre-selected component of our system, and so it is where we begin our investigation into the creation of a VoiceXML gateway, before moving on to the selection of the other required components in the next chapter.

2.3.1 What is Asterisk?

The Asterisk handbook [82] describes Asterisk and its uses. Asterisk is an open-source suite of telecommunications software which aims to support every possible telephony technology [69, 82]. The applications for Asterisk are quite broad. It can be used as a conference server, a Private Branch eXchange (PBX), a VoIP gateway or an IVR server, for instance. Since we are using it as our telephony platform, of particular interest to this thesis are its PSTN interfaces and its VoIP capabilities. This section will also explore the use of Asterisk as an IVR server, as this represents a possible alternative to VoiceXML.

Once installed on a PC running Linux, Asterisk does not require any other hardware, besides a network card. Using Asterisk in this way means, of course, that there is no PSTN interface, but that is easily fixed via inexpensive PC-based PSTN hardware. This means that for a bit more than the price of a desktop PC one gets a VoIP gateway, an IVR server, and, most importantly for our purposes, a gateway to the PSTN.

The VoIP protocols currently supported by Asterisk are Session Initiation Protocol (SIP), Inter-Asterisk eXchange (IAX), Media Gateway Control Protocol (MGCP) and H.323. Voice Over Frame



```
[simple_ivr]

  exten => s,1,setvar ($greeting, "Welcome to Adam King's Voice X M L page")
  exten => s,2,answer
  exten => s,3,festival (${greeting})
  exten => s,4,background (PressOne)
  exten => s,5,hangup

  exten => 1,1,festival (Congratulations you pressed one)
  exten => 1,2,hangup
```

Figure 2.6: A simple 'Press one' dial plan IVR

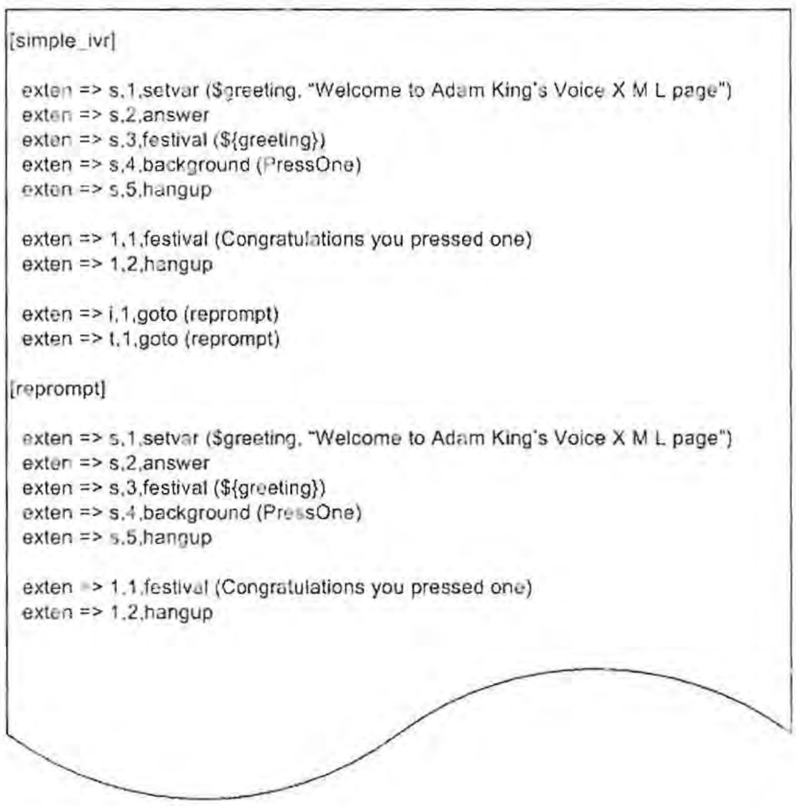
Relay (VOFR) is also supported. The latest Asterisk beta provides some support for the jingle protocol, which is used by gtalk. These capabilities, which are provided out of the box, mean that once the VoiceXML gateway has been integrated into Asterisk, it will have support for these VoIP technologies. The gateway will have both PSTN and VoIP capabilities. Since these technologies, in particular SIP, are becoming increasingly common, this is a major advantage.

According to the Asterisk handbook, the most important part of Asterisk is its dial plan. The Asterisk dial plan is responsible for routing all calls through a variety of applications and on to their final destination. The dial plan consists of one or more extension contexts [69]. These contexts can be used to implement a host of features including security, routing, auto-attendant and multilevel menus amongst others. The extension contexts can be used to create IVRs.

An IVR equivalent to the VoiceXML application shown in Figure 2.3 and created in the Asterisk dial plan can be seen in Figure 2.6. An extension context is made up of one or more steps, referred to as priorities. A priority is always of the form

```
exten => <exten>,<priority>,<application>,[(<args>)]
```

where *exten* is the extension the user has dialed (and it is used to match DTMF input), *priority* represents the order of execution, *application* is the application to execute at any given priority, and *args* are the arguments, if any, passed to the application. The functionality of grammars is created



```
[simple_ivr]

  exten => s,1,setvar ($greeting, "Welcome to Adam King's Voice X M L page")
  exten => s,2,answer
  exten => s,3,festival (${greeting})
  exten => s,4,background (PressOne)
  exten => s,5,hangup

  exten => 1,1,festival (Congratulations you pressed one)
  exten => 1,2,hangup

  exten => i,1,goto (reprompt)
  exten => t,1,goto (reprompt)

[reprompt]

  exten => s,1,setvar ($greeting, "Welcome to Adam King's Voice X M L page")
  exten => s,2,answer
  exten => s,3,festival (${greeting})
  exten => s,4,background (PressOne)
  exten => s,5,hangup

  exten => 1,1,festival (Congratulations you pressed one)
  exten => 1,2,hangup
```

Figure 2.7: A simple 'Press one' IVR system with re-prompting capabilities

using <exten>, which allows pattern matching. There is also a special set of extensions. Among them, *s* represents the start extension, *t* is the timeout extension and *i* is the invalid extension. In this example the user enters at the s extension and the `greeting` variable is initialized, the call is answered and then the application `festival` is used to synthesis `greeting`. The `festival` application uses the Festival TTS to synthesis text. The `background` application plays the `PressOne.wav` file while waiting for DTMF input. If 1 is received, the congratulatory prompt is synthesized and the call is hung up, otherwise the call is simply hung up.

While this IVR has the same basic functionality as the one created using VoiceXML, the two applications differ, functionality wise, in how they handle abnormal events, timeouts and incorrect (out of grammar) input. While VoiceXML has a default behavior, which can be overridden, given a *noinput* or *nomatch* event, similar behavior has to be explicitly built into the dial plan. In the event of an input error, VoiceXML re-prompts the user. To create similar handlers in Asterisk, the use of the special extensions t and i are required. A revised dial plan IVR can be seen in Figure 2.7, which shows the simplest way to re-prompt. The lack of a prompt counter results in some extra complexity to re-prompt once, let alone the three times VoiceXML defaults to.

While it is possible to create IVRs in the Asterisk dial plan, it is not as good as VoiceXML for creating large voice-enabled applications, for a number of reasons. While the dial plan notation may be relatively simple to learn it is not as familiar as the tagged notation of VoiceXML. More importantly, portability of the IVR applications is not guaranteed. While the syntax may remain constant, porting a dial plan may conflict with another Asterisk application's routing plan or the applications on the two machines may not be consistent, for instance. The dial plan IVR has to reside on the local machine, and to take an Asterisk dial plan to another platform is out of the question. Speech recognition can not be assumed - speech recognition formally supported by Asterisk is available, but is not free and requires a license. Finally, all those advantages gained by VoiceXML by mirroring web based development are lost using the dial plan.

2.3.2 The PSTN interface

This section describes the process involved in adding a PSTN interface to Asterisk. It notes and describes the different options and their associated prices before briefly describing the installation and configuration process.

The Asterisk telephony interfaces are divided into three categories, Zaptel hardware, non-Zaptel hardware and packet voice. Of these, the Zaptel and non-Zaptel hardware can be used to interface with the PSTN. The difference between Zaptel and non-Zaptel hardware is that Zaptel hardware supports pseudo-TDM switching while non-Zaptel hardware provides proper TDM switching. Pseudo-TDM hardware provides the same functionality, and almost the same performance, as TDM hardware but at a considerably lower cost [54]. The hardware can provide connectivity to the PSTN via a T1 or E1 Primary Rate Interface (PRI), an ISDN Basic Rate Interface (BRI), or a simple Plain Old Telephone System (POTS) interface. Any of these interfaces can be used to create a gateway to the PSTN, depending on requirements.

The first task in creating a PSTN gateway is to purchase the hardware, usually in the form of a PCI card. The different cards fit different requirements, including number of available channels per card, location and signaling type, and cost requirements. South Africa falls in the E1 area, and the available options are PRI, BRI or POTS interfaces. Table 2.2 shows the capacity and current cost of these interfaces. The FXO card is an analog interface, requiring a normal telephone line, while PRI and BRI are ISDN interfaces. The number of channels is the number of simultaneous voice calls the line can handle. We can see that, excluding the cost of a PC, a simple, double channel PSTN gateway can be created for as little as R360. (This figure ignores the cost of line rental as it is unavoidable: it will be incurred by any gateway.) So, an open-source, Asterisk-based VoiceXML gateway can be created for the cost of a PC plus the cost of a card. A relatively large 30 channel VoiceXML gateway for instance will cost in the region of R10 000, accounting for a generous R8 000 for an entry-level

Device	Number of Voice Channels	Approximate Cost
Single span PRI	30 (E1)	R4800
Single span BRI	2	R360
Single port FXO	1	R1200

Table 2.2: Possible PSTN interfaces in South Africa and their current approximate prices (October 2006) [60]

PC (October 2006), which is significantly lower than the $10 000 USD to $100 000 USD quoted in Chapter 1.

Once the hardware has been purchased, setting it up and configuring Asterisk is a simple task. Interfaces in Asterisk are implemented as channels, and a channel, the *zap* channel, represents the Zaptel and non-Zaptel hardware. One simply needs to install the hardware, load its drivers and configure the system. Configuration is simple, and involves editing two well documented files, `zaptel.conf` and `zapata.conf`. The `zaptel.conf` file simply defines the spans and groups associated with an ISDN interface. A span usually maps to a physical interface (an ISDN port for example), which can be divided into one or more groups. `zapata.conf` is used for any Asterisk specific configuration, such as the signaling type, the context an incoming call is placed into, or the group associated with the interface. Example configuration files for a BRI, single span interface can be seen in Appendix C.

As we have seen, creating a PSTN gateway using Asterisk is both cost-effective and simple. We can now look at the question of extending Asterisk's functionality.

2.3.3 Extending Asterisk

Because Asterisk aims at including support for any telephony technology, it has been designed to allow simple extension [82]. This section investigates these methods, and selects the most appropriate for our task. Some important restrictions govern the selection process. Firstly, extending Asterisk's functionality must not change any source code. In other words, the functionality must be added as transparently as possible. This will maximize the number of existing Asterisk implementations the VoiceXML service can be added to, and, at the same time, ensure that the integration is as seamless as possible. Secondly, this functionality must be added in such a way as to be available to any channel. This means that the VoiceXML service will be available to any technology Asterisk supports, or any technology which will be supported in the future.

Hitchcock [33] thoroughly explores the issue of extending Asterisk. There are four methods of adding to Asterisk's functionality. These are the Dialplan, the External Interfaces, the Asterisk Gateway Interface and Application Modules.

The IVR example above is an example of using the Dial Plan to extend Asterisk. This method can be disregarded as, while it may be able to mimic a VoiceXML application, it is insufficient and does not provide enough functionality to completely implement a VoiceXML service.

The second option, External Interfaces, are interfaces which allow other applications to connect, make queries and issue commands. There are two interfaces provided, the Manager API and the Asterisk Command Line Interface (CLI). The Manager API interacts with other applications via a TCP port. The CLI presents a prompt at which users, or applications, can enter commands. Once again, these two methods do not provide sufficient functionality to implement a fully fledged VoiceXML service.

The Asterisk Gateway Interface (AGI) is a dial plan application, similar to `hangup`, `answer` or `festival` which were seen in Figures 2.6 and 2.7. This application executes any external executable within the AGI environment. The external application has access to a limited set of commands, and Asterisk and the executable interact using standard in and standard out. There are almost no restrictions on an AGI application as far as implementation is concerned, the only requirements being that it is an executable and that it can communicate with standard in and standard out. While an AGI application does not alter the Asterisk source and is visible to all Asterisk channels, the control of the hosting Asterisk is limited.

This leaves the Application Module. An Asterisk application is a module which conforms to a naming convention (see Section 4.1.2) which is compiled against, and linked to, the Asterisk libraries. This option presents by far the most powerful method of extending Asterisk because it is linked to the Asterisk libraries. Applications of this form are used in the dial plan itself, are available to all channels and do not alter the source. This method, therefore, is ideal for our purposes. It has some limitations, however. The applications must be written in C and must conform to the Asterisk C API.

These constraints, fortunately, do not represent a major obstacle, and so the way to add a VoiceXML service to Asterisk is to create an application module. This method conforms to our requirements and presents sufficient functionality to implement a fully fledged VoiceXML gateway. Unlike an AGI application, however, using this method means that the application must be written in C. The process of extending Asterisk using application modules is fully explained in Section 2.3.3.

2.3.4 Summary

Asterisk is open-source, multi-protocol software system that is easily configurable and extensible. While it may serve as an IVR server, it does not afford IVR developers the same advantages as VoiceXML does. By adding suitable hardware, Asterisk can become an inexpensive gateway to the PSTN.

Creating a VoiceXML gateway as an application module and then adding relatively inexpensive hardware allows us to create a low-cost VoiceXML gateway with TDM and VoIP interfaces.

2.4 Summary

This chapter has outlined some of the uses of voice-enabled applications and IVRs, showing that they represent a large and important field. We have also seen how VoiceXML works and how it can be used to improve and ease IVR development. While VoiceXML is ideal for creating voice-enabled applications, VoiceXML gateways are expensive. This chapter has shown that it might be possible to create a low cost VoiceXML gateway using Asterisk.

Now that we have justified the need for a low-cost gateway, and identified a means of creating such a device, we can discuss the actual creation. Following the CBSE paradigm, we first need to identify the various functionality units and obtain COTS systems which can be used as components. These components must have clearly defined interfaces, in keeping with our requirement for an open system, and preferably be open-source, to keep costs to a minimum. Once the components are decided upon, a simple architecture can be defined, and the components can be integrated using XP as a development guide.

The following three chapters detail this process. Chapter 3 describes the process of selecting components and creating a simple architecture, and Chapters 4 and 5 cover the integration process.

Chapter 3

Component selection and system design

In the previous chapter we have seen how CBSE can decrease development time while increasing quality and maintainability, that there is a need for a low-cost VoiceXML gateway and that the telephony platform for such a system is available. We have also gained a deeper understanding of a VoiceXML gateway, its requirements and the interpretation process. We can now continue with the process of creating a VoiceXML gateway on top of the telephony platform.

This chapter separates the VoiceXML gateway architecture into separate units of functionality by examining the gateway requirements. Once these units have been identified, COTS systems can be selected to perform these functions. We will discuss the selected systems here, weighing them up against a list of requirements to determine their suitability.

We will conclude the chapter showing the architecture of the proposed gateway.

3.1 Requirements and restrictions

The component selection process requires that the different units of functionality have been correctly identified. Fortunately, the architecture of a VoiceXML gateway lends itself very nicely to a components-based approach, as it is very modular in nature. A VoiceXML gateway, as defined in Section 2.2.1 is:

> Hardware and software residing in the network that is capable of rendering VoiceXML applications which interact with a user in a standardized manner via synthesized speech, prerecorded audio, DTMF and Speech recognition over any audio channel.

Looking at the architecture of a VoiceXML gateway (Section 2.2.2.1) we can see that the rendering of the VoiceXML document is handled by two separate entities. Interpretation is done by the interpreter and input and output is handled by the implementation platform. Our first requirement, therefore, is an interpreter. The second is the implementation platform which is able to "interact with a user in a standardized manner via synthesized speech, pre-recorded audio, DTMF and Speech recognition". The implementation platform, therefore, needs a TTS engine, functionality to output an audio file, a DTMF recognizer, speech recognition engine and access to an audio channel. It has already been decided that Asterisk will be used to provide the audio channel, in the form of a PSTN interface. Finally, a gateway needs to be able to fetch documents from a web server, also a responsibility of the implementation platform. The succinct list of requirements, and so the required components, are as follows:

- VoiceXML 2.1 interpreter
- TTS engine
- Speech recognition engine
- Document fetcher

Viewing the components in relation to the Speech Interface Framework we see that the interpreter is essentially an implementation of the FIA, the TTS should be able to synthesize SSML documents, the speech recognition engine has to handle SRGS grammars and the document fetcher must be able to fetch documents using HTTP. An advantage of both the VoiceXML architecture and the Speech Interface Framework is that the selection of components is very straightforward. In addition to this, the functional units all have clear, well defined boundaries, and so are ideal for the creation of an open system, as described in Section 1.2.1.

The sourcing of the components is governed by a handful of restrictions. These come from many different sources: the thesis requirements themselves, the decision to build a component-based system and the selection of Asterisk as the telephony platform. As stated in Section 1.1, the gateway must be inexpensive and open-source. Section 1.2.1 states that the system is to be COTS-based and open. To make the system open requires clearly defined interfaces with which other components can interact [67]. These interfaces, which can either be APIs or a protocol, lend flexibility to the entire system. Finally, using Asterisk means that the components must be able to run on the Linux platform. In summary, the components must conform to the following requirements:

- Open-source
- Implemented by a third party
- Clearly defined API or protocol
- Well supported (longevity and maintainability)
- Independently executable
- Available on the Linux platform

When choosing open-source components care must be taken to ensure that the projects are well maintained and supported. The level of activity in the mailing lists and the activity of the project itself are indicators of the amount of maintenance and support a project has. These indicators must be kept in mind when choosing the components.

Now that it has been established what exactly is needed, we can discuss the selection of the components.

The following sections discuss the components which have been selected as well as the reasons for their selection. They are: OpenVXI (interpreter and document fetcher), Festival (TTS engine) and Sphinx 4 (speech recognition engine).

3.2 OpenVXI

OpenVXI is an open-source VoiceXML interpreter written in C and C++. It was originally started at Carnegie Mellon University before being taken over by ScanSoft and then finally by Vocalocity. It is essentially a framework for building VoiceXML gateways [24]. While the project itself is open-source, support is sold by Vocalocity, which means that documentation is rare and this can complicate implementation. The mailing lists, however, have been very active [24]. The version of OpenVXI used in this thesis is 3.3, which claims to be completely VoiceXML 2.1 compliant. An initial goal of the project was to make the interpreter as portable as possible. While this has been achieved, it can run on a host of different platforms including Linux, Windows and Solaris, it has been done at the cost of simplicity. Mechanisms introduced to ensure portability, such as adding extra layers of interaction or specialized data types to pass information across interface boundaries, increase the complexity of the system. The construction of a VoiceXML gateway using OpenVXI is discussed by Eberman *et al.* [24].

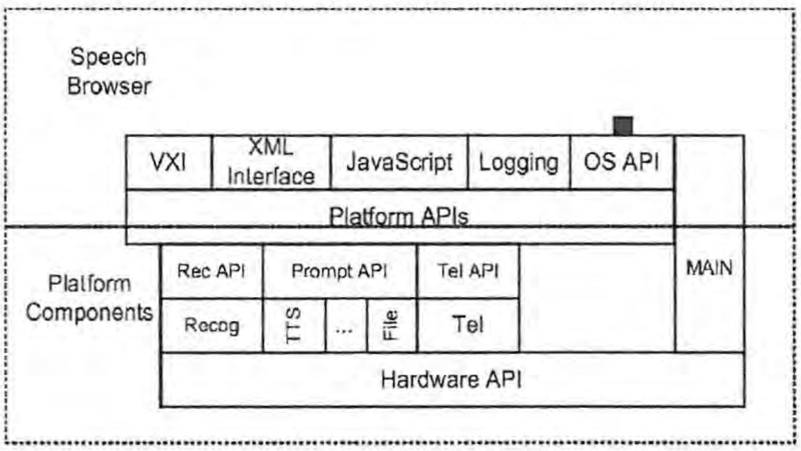


Figure 3.1: The OpenVXI architecture, as described by Eberman *et al.* [24]

The OpenVXI framework can be seen in Figure 3.1. One can see that the framework is separated into two distinct sections, the Speech Browser and the Platform Components, which is consistent with the VoiceXML architecture, where the Speech Browser is equivalent to the interpreter and interpreter context and the Platform Components with the implementation platform. The OpenVXI framework heavily influenced the selection of CBSE as a software engineering methodology, as it lends itself to a component-based approach.

The Platform Component APIs are just stubs, which must be fleshed out to give the gateway any form of audio capabilities. One can see that the remainder of our components, including Asterisk, integrate very nicely into this architecture.

3.2.1 Speech browser

The Speech Browser component of the OpenVXI architecture requires no further implementation on our part, and is independently executable, but will have no audio functionality until such time as the APIs are implemented. It itself comprises of a number of components: the VXI, an XML interface, a JavaScript component, a logging interface and an OS API.

The VXI (VoiceXML Interpreter) is the main control loop, the FIA, and can interpret VoiceXML 2.1. A SAX XML parser (OpenVXI uses Xerces) is used to both validate the VoiceXML. ECMAscript scripting functionality is required by the VoiceXML specification to provide some programmatic functionality to VoiceXML documents, and this is provided by OpenVXI's JavaScript interface. OpenVXI uses SpiderMonkey, Mozilla's JavaScript interpreter, for this purpose. The logging and OS interfaces are self explanatory.

In addition to these components, OpenVXI provides fully implemented Internet and caching functionality, which fetches and caches both VoiceXML documents and any associated files, such as grammar audio files. This caters for the Document Fetcher item on the list of requirements.

3.2.2 Platform components

The Platform Component section of OpenVXI is where implementation is required. Filling out the stubs gives the gateway audio functionality, including providing an audio channel. The stubs are the `rec` API, the `prompt` API and the `tel` API.

The `rec` API is responsible for handling any input, including voice recognition, DTMF recognition and recording. It is responsible for converting the input waveforms (speech or DTMF tones) into corresponding tokens and matching the tokens against one or more grammars. The interpreter is responsible for activating grammars, so the `rec` API requires a DTMF and speech recognizer, as well as recording functionality. The act of matching the tokens returned from the recognizers against the active grammars has been implemented. As per the specification, the speech recognition engine itself must be able to handle dynamic grammars and n-best results with confidence indicators. As we have seen in Section 2.3.1, Asterisk provides DTMF tone recognition, and so, component wise, all that is required to implement the `rec` API is a compliant speech recognition engine. Asterisk also provides recording functionality.

The `prompt` API is responsible for fetching and playing audio as well as synthesizing TTS prompts. Once again Asterisk can be used to provide some functionality, as it is able to play audio files (Section 2.3.1), so an SSML TTS is all that is required for the `prompt` API.

Finally, the `tel` interface interacts with the gateway's audio channel. This API has to provide any telephony functionality required by the specification. In our case this functionality (setting up and tearing down calls for instance) is provided by Asterisk. The `tel` API is where the gateway functionality of the browser is implemented. It is plain to see how using Asterisk to provide the `tel` API's functionality results in a VoiceXML gateway.

The fleshing out of these APIs gives the interpreter audio functionality, and results in the creation of the implementation platform. OpenVXI, therefore, is a good candidate for the purposes of this thesis, but it still worthwhile mentioning publicVoiceXML as an alternative.

publicVoiceXML [74] claimed to be a turn-key solution, providing DTMF recognition and TTS capabilities. This seemed ideal, but unfortunately this did not appear to be true. The project could not compile and seemed to have been abandoned at the time of component selection.

3.2.3 Summary

OpenVXI conforms to our list of requirements. It is open-source, implemented (and maintained) by a third party and independently executable. Component interaction is provided by an API. Since it is written in C and C++, interaction between OpenVXI and Asterisk is simple, and can easily be integrated into an Asterisk Application Module. It also appears to be well maintained and supported, as suggested by Eberman and by the fact that it supports VoiceXML 2.1. Finally, it runs on the Linux platform.

We can now discuss the two outstanding components required for the gateway, the TTS engine and the speech recognition system.

3.3 Festival

Festival is a Linux-based, open-source framework for creating multilingual TTS systems, written in C++ [7]. Different voices can be created in any language in a relatively simple and quick manner [55]. The advantages of this feature can best be understood if we look at voice-enabled applications from the South African perspective. A synthesized voice is impersonal and alien, which can make a user uneasy. Making the accent of the voice familiar can help this. Additionally, as South Africa has many official languages, different languages means a higher number of potential users. Research into creating Festival voices in indigenous languages is currently underway, for example, at the Speech Technology and Research (STAR) group [83]. For the purposes of this thesis we used the default, free, voices which are distributed with the Festival system. These produce clear, if very mechanical, sounding voices, which suffice for our purposes. While Festival claims to be capable of handling SSML documents, and so meets the Speech Interface Framework specification, we will only be passing text to the Festival engine to simplify matters.

The framework provides six interfaces which can be used to create a full speech-synthesis system, which are described by Black *et al.* [7]. There are a few restrictions to keep in mind when deciding on an API. Firstly, time delays associated with initialization must be kept to an absolute minimum. Since we are using the synthesized speech for real-time applications, it must be made available immediately as any time delays will result in a degradation in the usability of the system. Also, the API must provide sufficient functionality to allow concurrent access to the synthesis engine, without any resource-dependent time delays. These two conditions also apply to the speech recognition-engine discussed in the following section.

3.3.1 Festival interfaces

The six interfaces are implemented using either an API or a client-server approach. They present access to the engine in a variety of manners and languages, each of which provide a different combination of control and ease of use. The interfaces are the Scheme API, Shell API, C/C++ API, Java and JSAPI, C only API and the client-server API. These interfaces are explained by Black *et al.* [7].

The Scheme API allows full control of the Festival functionality using the Scheme programming language. The command

```
(SayText "Welcome to Adam King's Voice XML page")
```

will create the prompt required for the VoiceXML example discussed in Section 2.2. Commands are either read from files or from the command prompt. This API represents the most fine grained, as one has access to a full programming language. The Scheme API is also the most direct means of controlling Festival, as most other methods use Scheme at some point. Commands are interpreted by Festival in either *batch* mode or *live* mode. In batch mode Festival will exit as soon as it is done executing the Scheme commands, while in live mode it continuously reads Scheme commands from the command line.

Using the Scheme API in batch mode does not satisfy the system requirements, as there is an initialization time delay associated with Festival having to restart after every synthesis request. Using the Scheme API in the live mode only incurs the initialization penalty the first time it is started, but not on subsequent synthesis requests. Unfortunately, neither method allows concurrent access to the synthesis engine, which means that a new Festival process has to be created for each concurrent call, wasting resources and incurring at least one time penalty for every call. We therefore have to look at one of the other methods.

The Shell API provides simple access to Festival through the shell. Besides being limited, this method does not allow concurrent users and will result in an initialization time penalty for each call. This is because a new Festival instance has to be created for each call.

The C/C++ API allows developers to embed Festival in their own C/C++ applications. An application can link its source to the Festival libraries, which gives the application access to a number of external commands. This method is desirable, as OpenVXI, with which we would be interacting, is written in both C and C++. This method will also allow each process handling a call to have full access to the Festival system. However, it had to be discarded as it still has an initialization penalty.

The three final APIs all make use of the Festival server. The text to be synthesized is sent to the Festival server as Scheme commands and the server replies with the synthesized text. Since the client sends Scheme commands, a lot of the Scheme API functionality is retained. The client-server approach has numerous advantages. Clients do not have to link with other, Festival specific, code; the server can reside on a separate machine; and, most importantly, a client-server approach allows concurrent synthesis requests and, once the server is up and running, there are no initialization penalties. The C client-server interface is therefore ideal. Additionally, Asterisk uses this approach in implementing the `festival` application, and Festival provides an extensive example. Therefore a C++ client was created to provide half the functionality of the prompt API, i.e the acoustic rendering of text messages. The other half, the playing of pre-recorded audio, will be done directly by Asterisk.

3.3.2 Alternatives

The Cepstral speech synthesis system also runs on the Linux platform, and it is also multilingual. Comparing the Festival free voices to the demos available for Cepstral it appears, to this author's untrained ear, that the Cepstral output is of a better quality. Unfortunately, Cepstral is closed source, and using Cepstral is not free. Voices require licensing, and cost \$29.99 USD (October 2006) each; and a further license is required for concurrent execution, which starts at \$200 USD (October 2006) for four ports [15]. In addition to this, voices cannot be created by a third party, and so obtaining unsupported languages and accents may be problematic. However, this does raise the point that some of the open-source components may not be the most suitable in all scenarios, and should an organization be able to afford a Cepstral voice they may want to use it. This highlights an advantage of creating an open component-based system, as it would be relatively simple to replace Festival with Cepstral. While this may require some knowledge of the system and implementation skills, it is still better than having to pay for another system with expensive hardware. Additionally, the more open the system is, the easier the component swapping should be. In this case it should simply mean changing the `prompt` API to interface with the Cepstral system.

3.3.3 Summary

The Festival TTS engine conforms with our list of requirements. It is Linux-based and open-source, independently created and executable. Component interaction is done by a client-server architecture. The two additional advantages, the ability to create specific voices and the fact that Asterisk already uses it, makes it a very attractive option.

Test	S3.3 WER	S4 WER	S3.3 RT	S4 RT (1)	S4 RT (2)	Vocabulary Size	Language Model
TI46	1.217	0.168	0.14	.03	.02	11	isolated digits recognition
TIDIGITS	0.661	0.549	0.16	0.07	0.05	11	continuous digits
AN4	1.300	1.192	0.38	0.25	0.20	79	trigram
RM1	2.746	2.88	0.50	0.50	0.41	1,000	trigram
WSJ5K	7.323	6.97	1.36	1.22	0.96	5,000	trigram
HUB4	18.845	18.756	3.06	~4.4	3.95	60,000	trigram

Table 3.1: Comparison of Sphinx 3.3 and Sphinx 4 performances, as shown in [14]

3.4 Sphinx

The final component that is required is a speech recognition engine. This must be able to support SRGS, n-best results, confidence scores and dynamic grammars, as described in Section 2.2.2.2. The telephony environment in which we are wanting to use the engine adds two important constraints. Firstly, speaker independence is required. Secondly, the speech engine must come with 8Khz audio samples, as the PSTN samples audio at 8Khz. (While it may be possible to create our own samples, this is time consuming and certainly beyond the scope of building a proof-of-concept prototype.)

Sphinx is an open-source, speaker-independent, continuous-speech, large-vocabulary speech-recognition engine which was created at Carnegie Mellon University [47, 46]. It has been observed, by joining the mailing list, that the Sphinx project has a large and active community. There are three main versions: Sphinx 2, Sphinx 3 and Sphinx 4. Sphinx 3, a successor to Sphinx 2, is written in C++ and runs on the Linux platform. Sphinx 4 is written in Java. Both support n-best recognition lists with confidence scores and the SRGF specification [76, 14]. While it may seem preferable to use the Sphinx 3 system, as it is written in C++, Sphinx 4 outperforms Sphinx 3 both in terms of word error rate and real-time performance in a number of tests run on different acoustic models, as can be seen in Table 3.1. The Word Error Rate (WER) figure shows how many words were incorrectly recognized (as a percentage), the lower the figure the better the accuracy. RT stands for Real-Time and represents the time taken to decode the utterance. The Sphinx 4 tests were run on both single and dual processor architectures, hence the different real-time results. One can see that Shpinx 4 outperforms Sphinx 3 in most tests, except in the RM1 test, where it has a worse WER, and the HUB4 test, which has a worse RT figure. One can see that the 5000 word Wall Street Journal (WSJ) model tested better in Sphinx 4 than in Sphinx 3. An 8Khz version of this model is contained in the Sphinx 4 package and is what will be used for this project. We have therefore chosen the Sphinx 4 system to implement the `rec` API.

Sphinx 4 has been designed to be as modular and flexible as possible, supporting many kinds of grammars and acoustic models [44]. Creating applications using Sphinx is discussed in the Sphinx 4 help files [43]. Building such an application simply requires a Java front end, a configuration file and a grammar. The configuration file allows us to configure the engine to accept audio from the PSTN. A detailed description of how to create a speech-recognition application using Shpinx 4 will be given in Section 5.3.1.

3.4.1 Alternatives

A relatively new alternative is the inclusion of speech recognition functionality directly into the Asterisk system. The LumenVox speech engine has been added to Asterisk and provides out-the-box speech recognition. While this may seem ideal, this service is not free, as licensing fees are required. A single speech-recognition port costs $50 USD while 5 ports cost as much as $1 500 USD (October 2006) [52]. As with the Cepstral TTS engine, these costs conflict directly with the requirements set out in this thesis, and so LumenVox will not be discussed further.

3.4.2 Summary

Sphinx 4 satisfies most of the requirements for a component for the system under construction. It is open-source, independently created and available on the Linux platform. The one requirement which is not immediately obvious is the one of a clearly defined interface.

Integrating Sphinx into the system involves three distinct entities: the engine itself, the front end, and the OpenVXI `rec` interface. We have to ensure that the entities can all interact. The front-end simply uses the Sphinx API. An interface between OpenVXI's `rec` interface and the Sphinx front-end results in the integration of Sphinx into the system. This integration process is discussed in full in Chapter 5.

While the integration issue may have been simplified greatly by the use of Sphinx 3 (because different languages would not be an issue), we chose Sphinx 4 for two reasons. Firstly, and most importantly, Sphinx 4 outperforms Sphinx 3 both in terms of word error rate and real time performance. Secondly, integrating Sphinx 4, which is written in Java, into OpenVXI, which is written in C and C++, an interesting exercise in Inter Process Communication (IPC) on a Linux platform. Audio must be sent to Spinx for recognition and the resulting recognition must be set in the `rec` API.

In addition to meeting the project requirements Sphinx 4 is also flexible enough to conform to all the gateway related requirements. It is therefore ideal for our purposes. From this point on Sphinx 4 will simply be referred to as Sphinx.

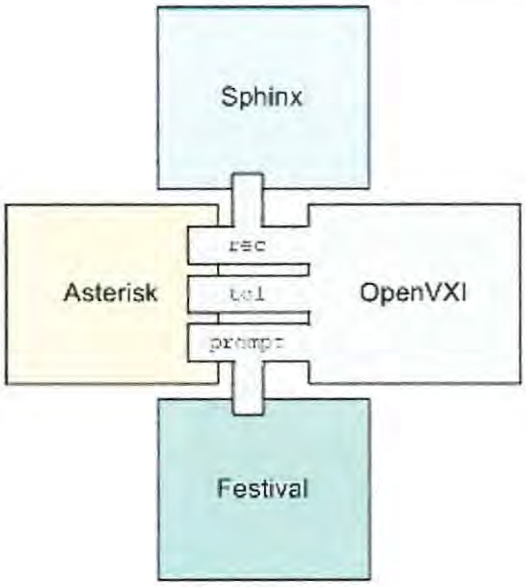


Figure 3.2: The VoiceXML gateway's architecture

3.5 System architecture

As discussed in Section 1.2.2, we wanted to keep the architecture of the gateway as simple as possible. Fortunately, due to the way in which VoiceXML has been designed (Section 2.2.2.1), this is not a complicated task, particularly given that the system has only four components (Asterisk, OpenVXI, Festival and Sphinx). OpenVXI's architecture is clearly influenced by the VoiceXML architecture, resulting in the distinction between the Speech Browser and the Platform Components. Since we are integrating the components into OpenVXI, this will have a large impact on the gateway's architecture. This architecture can be seen in Figure 3.2.

There are a few things to note about the architecture. The first is that despite the requirement of an open system, Asterisk and OpenVXI have become tightly coupled and heavily dependent on each other. Asterisk will be used to provide DTMF recognition and audio playback as well as telephony functionality, which means that it is used to flesh out all three of the OpenVXI interfaces. Architecturally, this means that the core of our system, which was Asterisk, has expanded to include OpenVXI, and removing either of these will result in an entirely new system.

On the other hand, both Festival and Sphinx are loosely coupled, keeping in line with our open system requirements. These systems can easily be replaced, as they are not bound to the system.

The simplicity of the architecture is apparent. The core of the system is the combination of the Asterisk and OpenVXI systems (in other words the audio channel to the PSTN and the interpreter).

RHODES UNIVERSITY LIBRARY

This core provides hooks for the other two components to interact with.

3.6 Summary

This chapter follows directly from the CBSE principles discussed in Section 1.2.1. We have divided the gateway up into units of functionality, after examining the VoiceXML specification. Given Asterisk as a pre-requisite, the remaining units of functionality have been found to be: an interpreter, a text-to-speech (TTS) system, a speech-recognition engine and a document fetcher. Components were selected to provide the required functionality.

The three components selected, in addition to Asterisk which was pre-selected, are: OpenVXI (interpreter and document fetcher), Festival (TTS system) and Sphinx (speech-recognition engine). We have seen how these components have met both functionality and thesis requirements, justifying their inclusion in the system.

The next two chapters discuss the integration process of the selected components.

Chapter 4

Building the gateway: Phase 1

This chapter discusses the first half of the implementation process, that is the integration of Asterisk, OpenVXI and Festival, resulting in a VoiceXML gateway which has only output capabilities.

The entire implementation process subscribes to the XP methodology, which was described in Section 1.2.2. Following the scope-driven development paradigm gives us both the order of implementation and the user stories for each stage in the development process. Firstly, the task of extending Asterisk, as described in Section 2.3.3, must be completed. To this we have to add VoiceXML interpretation capabilities, resulting in a system which can handle PSTN (and other) calls, fetch and interpret VoiceXML documents but can not interact with a user. This interaction is made possible by filling the OpenVXI APIs to provide input and output capabilities. Prompting capabilities, in terms of scope-driven development, are more important than handling input, and so we will examine the process of implementing OpenVXI's `prompt` API.

In keeping with the XP paradigm we employed continuous testing. At each step in the development process we discuss the testing process. Fortunately, creating these tests was usually a simple matter of creating a VoiceXML document to test whichever unit of functionality that has been implemented. Once the new functionality passed the tests, the code could be refactored and integrated into the main system.

As the various user stories were implemented we could see the system grow, finally culminating in a VoiceXML gateway.

4.1 Creating an Asterisk application

Our first, and most important, task in the development process is to extend Asterisk in a way that does not alter the source code, making it portable across different Asterisk installations, and allowing any channel, the Zap and VoIP channels in particular, to access the new functionality. For this we have chosen to create an Asterisk application module. Asterisk applications of this sort are called from the dial plan. This section simply describes the process of creating and adding an Asterisk application module. The actual functionality of the application is discussed later in the chapter. This process is generic, and anyone wishing to add an application module has to first complete this step. It must be noted that this process has changed slightly in later versions of Asterisk - the naming of some variables has changed.

Before creating the application, and in keeping with XP practices, we need to create the tests. Hence this section will also discuss and motivate the test. Finally, the application has to be named. We have simply decided to call it `voicexml`.

4.1.1 Creating the test

Creating a test for this is simple, as it involves a single dial plan command. Since we want to be able to pass the application arguments, in particular the VoiceXML URL, the test has to reflect this functionality:

```
exten => 999,1,voicexml(http://king.rucus.net/test.vxml)
```

Should a user dial 999, the `voicexml` application must be executed, and, to verify argument handling, the application must simply output the argument. Should the test pass it means that we have successfully created a new Asterisk dial plan application which is exposed to any channel supported to Asterisk.

Now that we have created the test we can proceed with the application module itself.

4.1.2 The `voicexml` application module

There are two steps involved in creating an Asterisk application module. The first is to create the application itself and the second is to ensure it gets built and loaded. While this is a relatively straightforward process, it does require some knowledge of the Asterisk system and some Linux fundamentals. We will discuss this process in the rest of this section.

All the Asterisk applications are written in C and follow a simple template. The voicexml skeleton application is shown in Appendix D. These C files all follow a naming convention. The C file must be named 'app_<name>.c' where *name* corresponds to the name of the application. In our case this will be app_voicexml.c. Each application is then required to provide a number of global variables and functions. The variables all point to strings which are used for the description of the application in places such as the Asterisk CLI. These follow a similar naming convention, with the application name being substituted into the name of the variable to create a unique identifier.

These variables are followed by a number of methods: voicexml_exec, unload_module, load_module, description, usecount and key. Of these voicexml_exec, unload_module and load_module are of interest. load_module and unload_module are responsible for loading and unloading the application respectively. voicexml_exec is where execution begins when the application is called, and where application specific functionality is added. In this user story it simply checks for arguments and output them, but eventually this is where OpenVXI and Asterisk will be integrated.

Handling the arguments is relatively simple. The voicexml_exec function accepts two arguments, a pointer to the channel structure associated with the call and a void pointer, called data, which points to any arguments passed to the application from the dial plan. Checking for arguments is a simple matter of evaluating the data variable.

The channel structure passed to voicexml_exec is what is used by the application to manipulate the channel. It has already been initialized at this point, and so the application itself is not concerned with any channel specific issues. This means that any application created in this manner is immediately available to any Asterisk channel, and so is available to any Asterisk supported telephony technology. In our case it means that the voicexml application will be available to the zap channel, and so the PSTN, creating a gateway. In addition to this, the application will support any channels that are added at a later date. This means that should Asterisk include a new technology, say a new VoIP protocol, it will have immediate access to the voicexml application, with no changes to the application required.

It is important to note that while the application has to conform to the template and naming conventions, it is still simply a C file, and so other methods and global variables can be added as usual. Once the template has been created, it needs to be built, linked and loaded against the Asterisk source. Once again, this is a fairly simple process, but once again is done in a manner which is specific to Asterisk.

All Asterisk applications can be found in the apps/ directory of the Asterisk source, and so our new application, app_voicexml.c, must be put there. To ensure that the application is built and linked, the apps/Makefile file has to be edited. The make command, which is used to build Asterisk, is discussed by Fowler [27]. Instructions on what to do with specific files are passed to make in the form of a Makefile. A Makefile consists of one or more targets. A target represents the file which is to be built. It depends on one or more prerequisite files, and has an associated action, which is responsible for the actual building. So a simple `Makefile` entry to build an object file, say `my_object_file.o`, from a C file, `my_c_file.c`, would look like this:

```
my_object_file.o : my_c_file.c
  #action - build the object file, which requires my_c_file.c
  gcc -c my_c_file.c -o my_object_file.o
```

So, we need to create a target to the Makefile to build `app_voicexml.c`. This can be done in two ways, explicitly and implicitly. While it will suffice to add our application implicitly at this time, we will discuss both methods here, as the implications of creating explicit targets will become apparent in the following section.

The majority of Asterisk applications are built implicitly. What this means is that they are all built using the same build commands. This generic approach eases the task of creating the `Makefile`, and will suffice for most cases as most applications do not require any special build arguments. Since our application does not have any abnormal build requirements at the moment (we are not linking to any extra libraries for instance) this method is sufficient for our purposes. The `apps/Makefile` file contains an `APP` variable, which is a list of all the targets which are to be built in this manner. Simply adding `app_voicexml.so` to this list will ensure that the application module will be built correctly, passing the standard arguments to the compiler.

The build process creates a shared object, which is dynamically linked to Asterisk, and loaded at run-time.

The install process copies the shared object files from the `apps/` directory to Asterisk's main module folder, `/usr/libs/asterisk/modules/`. These modules are loaded at run-time, and can be configured in the Asterisk module loader configuration file found at `/etc/asterisk/modules.conf`.

It is plain to see that extending Asterisk in this manner is very simple and powerful. Creating the application is a simple matter of fleshing out the skeleton and ensuring that it gets built by editing the appropriate `Makefile`. The creation of the module does not require the Asterisk source to be altered, and is done in such a way as to be available to any channel.

Finally, we can run the tests, which show that we have successfully created the application, and that we can pass arguments from the dial plan to the application. The test is simply run by dialing 999 from a telephone connected to the Asterisk system. If we log to the CLI from the voicexml `application` we see the following output when running the test:

```
NOTICE[30189]: app_voicexml.c:125 voicexml_exec:
     voicexml called successfully
NOTICE[30189]: app_voicexml.c:126 voicexml_exec:
     argument is: http://king.rucus.net/test.vxml
```

Since this is such a simple process, the task of refactoring and integrating can be ignored. Now that we can add functionality, we can look at integrating OpenVXI into the voicexml application.

4.2 Asterisk and OpenVXI

The the integration of Asterisk and OpenVXI is mainly an exercise in linking libraries, as we have to link OpenVXI and the `voicexml` application. Once that is done, we can add the interpretation functionality to the `voicexml` application, and pass the channel pointer to the OpenVXI platform. Once this is complete, we will have a VoiceXML gateway with no input or output capabilities. While this does not immediately create anything useful, it does result in the core of our system, and the architecture described in Figure 3.2 begins to take shape.

Linking Asterisk and OpenVXI in this way provides most of the functionality required by the `tel` API without actually needing to add anything to the API itself. By the time the interpreter is required in the `voicexml` application, the incoming call has been set up and answered, and so handling the incoming call is not part of the `tel` API's responsibility. Similarly, once interpretation of the VoiceXML application is complete, the `voicexml` application can simply exit and Asterisk will be responsible for tearing down the call. The remaining functionality required by the `tel` API is to provide support for the *transfer* and *disconnect* elements. Our proof-of-concept gateway will not support the transfer element, but providing support for the disconnect element will be discussed later.

In addition to this, the linking process will also give the OpenVXI APIs access to the Asterisk DTMF recognition and audio file handling functions, as described in Section 4.3 and Chapter 5.

Once again we can begin the process by designing a simple test.

4.2.1 Creating the test

There are a number of things which have to be tested. Firstly, since we are making use of OpenVXI's document fetching mechanisms, we need to be sure that the VoiceXML documents can be fetched.

Secondly, we need to test that the integration has been successful, and that the voicexml application is able to initialise and execute the interpreter, passing it the URL argument and the channel pointer. A means of testing this was not immediately apparent, as there was no means of input or output. It was decided to simply create a VoiceXML application which plays some prompts and then edit the prompt API to see if the interpreter makes the request to render the prompts. (As said, at the moment our gateway will not be able to render the prompts, but we can study the output to determine if the interpreter makes the requests to the API.) Of course, should this test be passed, we can assume that the document-fetching test also passed.

Once again, creating the tests is a simple task, and involves creating a VoiceXML application and uploading it to a web server. The VoiceXML application that has been created is a simplified version of the example application in Figure 2.3, and we expect a successful test to output the `greeting` and the `pressOne.wav` prompt requests.

4.2.2 Adding the VoiceXML interpreter to the voicexml application

To allow the `voicexml` application to interpret VoiceXML documents the `voicexml` application and OpenVXI have to link to each other, as both components need access to each other's libraries. `voicexml` needs to begin the interpretation process and OpenVXI needs to be able to control the Asterisk channel from its APIs.

To do this, we have to ensure that when each component builds it is linked to the correct libraries at the correct places. So we once again edit the respective `Makefile`, creating explicit rules (as described in the previous section) to ensure that the prerequisite files are linked correctly.

Applications can be linked either dynamically or statically. If an application is statically linked to a set of libraries, those library functions used by the application will be copied and built into the resulting executable. This allows the application greater portability, as the linked libraries are not required on the target machine. The downside of this is that more memory will be used.

When a library is dynamically linked, a reference to the library is inserted into the executable, and the actual linking only takes place at run-time. This means that the size of the application will be smaller, but the libraries are required on the target computer, hindering portability. Linking in this way means that the application and the library are independent, and many different applications can link to the same library, avoiding having the same functions repeated in many different applications [28]. An added, although somewhat risky, advantage is that changes to the library will affect any dynamically linked modules.

In keeping with our open system approach we will use dynamic linking. This means that, while OpenVXI and Asterisk are still tightly coupled, they are still separate entities. This eases the task of maintenance, as upgrading or installing either can be done independently.

Once the components are linked, we can begin to add functionality to the `voicexml` application, by filling out the `voicexml_exec` function, taking care to do any required housekeeping. The following pseudo code segment shows the application logic:

```
voicexml_exec (channel, data)
    /* deal with the parameters */
    if we have a URL
        /* set up and run the interpreter */
        initialise the interpreter, passing URL and channel
        allocate the interpreter resources
        interpret the VoiceXML
        /* once we get here we are done with the VoiceXML
         * application, for whatever reason, and can shut down
         */
        shut down the interpreter
        clean up memory
    else (there is no URL)
        show an error message and exit
```

One can see that the linking process is where the majority of the work has been done, and that once that is accomplished we can use the OpenVXI API to run the interpreter. All that remains is to pass the channel pointer through to the APIs, which has been achieved by adding an extra argument to the initialization function. It is passed to the interpretation engine when it is initialized by the `voicexml` application and then again when the engine initializes the APIs.

It is also apparent that neither the `voicexml` application itself nor the interpreter concern themselves with any call specific details, maintaining the channel portability amongst Asterisk channels, including the Zap channel, which gives our system and interface to the PSTN.

We can now apply the tests to ensure that the process has been completed successfully.

4.2.3 Running the test

As with the previous test dialing 999 runs the test, and we see the following output:

```
NOTICE[30189]: app_voicexml.c:125 voicexml_exec:
       voicexml called successfully
NOTICE[30189]: app_voicexml.c:126
       voicexml_exec: argument is:
          http://king.rucus.net/test.vxml
<?xml version='1.0'?>
<speak version='1.0'
       xmlns='http://www.w3.org/2001/10/synthesis' xml:base=''
       xml:lang='en'>
  Welcome to Adam King's Voice XML page
</speak>
<?xml version='1.0'?>
<speak version='1.0'
       xmlns='http://www.w3.org/2001/10/synthesis' xml:base=''
       xml:lang='en'>
  <audio pressOne.wav/>
</speak>
```

There are a number of things to note. Firstly, the VoiceXML doc has been fetched correctly. Secondly, the document has been processed and the interpreter has attempted to play the prompts, so we can assume that the linking process has been successful.

This output also shows that OpenVXI handles prompts internally as SSML documents, which is compliant with the VoiceXML standard. The interpreter encountered the two prompts in the VoiceXML application and created SSML documents for the prompts before sending them to be queued by the `prompt` interface. The interpreter queues prompts in the manner discussed in Section 2.2.2.4, and the implementation of this queue is discussed in the following section. Both types of prompts (text and audio files) are represented by SSML documents, which means that it is not possible to send the queued SSML documents directly to the Festival system, as it can not play back audio files. Therefore the SSML prompts have to be parsed at some stage to differentiate between audio files and text for synthesis. This process is also discussed in the following section.

The resulting architecture can be seen in Figure 4.1. The figure illustrates the tightly coupled relationship between Asterisk and OpenVXI, and how Asterisk is capable of interacting with all three OpenVXI APIs. Now that Asterisk and OpenVXI are linked, we can move on to giving the gateway voice by implementing the `prompt` API.

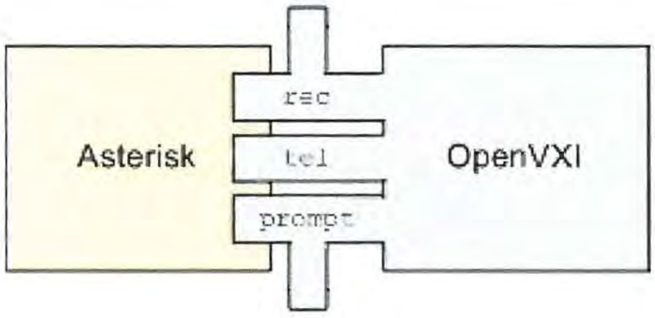


Figure 4.1: The core of the gateway after Asterisk and OpenVXI are integrated

4.3 Giving the gateway voice

This section describes the implementation of the `prompt` API, which will enable the gateway to produce output. This is not quite a simple as integrating Festival into the system. As noted in the previous section, an SSML parser will have to be created and a prompt queue implemented. Only once these mechanisms have been put in place can we look at actually playing the prompts on the channel.

This development episode will result in a gateway with output capabilities, and we can start creating some VoiceXML applications which have some use, mainly information services. The first example that comes to mind is the time server - "At the beep the time will be".

Since the tests created in the previous section used simple prompts to prove that the interpreter was functioning correctly, we do not have to create new tests. Running the tests and evaluating the output requires a user. One has to listen to the call to ensure that the gateway is producing output and that the output is of a satisfactory nature. Listening to the Festival output from the Shell API and comparing it with that of the gateway will give us an indication of the level of success of the integration process. Should the system produce audio of a worse quality than when using standard Shell API, we can assume that the integration was not completely successful. While a test of this sort would be regarded as subjective, it will only be used as an indicator as to whether the integration process has been successful. Should understandable speech be turned into garbled sound, we can assume that the integration was unsuccessful.

We will begin by describing how the APIs work, and then investigate the contents of the `prompt` API which determines the implementation requirements. We can then look at the SSML parsing routine, queue implementation and the playing of both types of prompts. Finally, the test will be run.

4.3.1 The prompt API

As stated in Section 3.2, a primary goal of the OpenVXI system is portability, which is reflected in the structure of the APIs. They are more complex than if portability was not required. This section discusses the APIs, using the `prompt` API as an example. It will also examine those functions of the API which need to be filled.

The Speech Browser component of the system has been implemented so that it is not in any way platform-specific, maintaining the goal of portability. The Platform Components, however, are platform dependent as they have to access hardware, sound cards and telephony interfaces for instance.

These have therefore been implemented as a set of APIs, which are used to implement any platform-specific requirements, and are responsible for asynchronous event handling.

The framework has been designed to allow components to be replaced either at run-time or initialization, as stated by Eberman *et al.* [24]. Each API is built on an abstract C interface, in this case the `VXIpromptInterface`. The `VXIpromptInterface` is essentially a branch table used to transfer control to the API, allowing the run-time replacement of components in a generic, portable manner. The APIs themselves contain another structure, `VXIpromptImpl` in this case, which has as its base the abstract interface, and includes pointers to any other interfaces the API may require, such as the logging or Internet components and, in particular, the channel pointer. When the resources are created for the API, the abstract interface's function pointers are set to point to the appropriate functions. In other words, each API is represented in the OpenVXI system simply by an abstract C interface containing function pointers. Anyone using the OpenVXI framework must create an interface which resolves these pointers and provides the required functionality. OpenVXI provides an example of how to do this, which we have used and extended. This implementation, represented by the `VXIpromptImpl` structure, provides the functions the base interface points to. To this structure we have added a pointer to the Asterisk channel representing the call, giving the various functions control of the channel.

Communication from the APIs to the interpreter is handled by return values. An extensive list of result codes has been provided by OpenVXI, which are returned to the interpreter. Each interface has a set of codes which are defined in the include file of the API. A result code of zero indicates success. Result codes smaller than zero indicate a severe error, which is likely to be the result of a platform error (such as an out-of-memory error) and may cause the termination of the call. Result codes larger than zero are warnings, and are usually application related errors, such as errors when fetching an audio file or restricted access to the TTS system. `VXIprompt_RESULT_SUCCESS` is an example of a successful `prompt` API return code.

The base interface, the `VXIpromptInterface`, gives us an indication of the functions which have to be provided:

- `VXIpromptBeginSession`
- `VXIpromptEndSession`
- `VXIpromptPlay`
- `VXIpromptPlayFiller`
- `VXIpromptPrefetch`

- `VXIpromptQueue`
- `VXIpromptWait`
- `VXIpromptStop`

Of these, the most important functionality is provided by `VXIpromptPlay` and `VXIpromptQueue`. We have not implemented the `VXIpromptPrefetch` and `VXIpromptPlayFiller` functions, as they do not represent important gateway functionality, and so could be disregarded in the construction of our proof-of-concept system.

`VXIpromptBeginSession` is used to set up a prompting session for a call, and in this case all that has to be done is to set the channel pointer to the pointer passed through from the `voicexml` application. `VXIpromptEndSession` is similarly used at the end of a session to do any housekeeping. In this case we ensure that the prompt queue is empty and the memory has been freed. We also remove any files that were used during the prompting session. `VXIpromptStop` is used to stop a playing prompt. This behavior is discussed when implementing the barge-in attribute in Section 4.3.4. The final two functions that have been implemented, `VXIpromptPlay` and `VXIpromptQueue`, are discussed in depth further on in this chapter. Before that can be done we have to discuss the SSML parser.

4.3.2 SSML parser

This section discusses the implementation of a simple SSML parser. This parser is only used to extract the most important information from the SSML document, and the rest of the information is disregarded. As we have seen in Section 4.2.3, OpenVXI handles prompts as SSML documents internally. While it would be ideal to pass the SSML directly to Festival for synthesis. we have chosen not to for two reasons. Firstly, the interpreter represents both audio and synthesized prompts as SSML documents. Asterisk, which does not support SSML, is used to render the audio prompts and so the audio prompts, at least, have to be represented in some other fashion, requiring the SSML to be parsed. Secondly, it is simpler to pass Festival plain text as opposed to an SSML document. While disregarding SSML specific functionality reduces the VoiceXML functionality somewhat (words will not be emphasized, for instance), it makes the text synthesis process simpler, and this is sufficient for the purposes of this thesis. It must be noted that the gateway still conforms to the standard in that it represents prompts as SSML documents and can handle these documents.

We therefore have to parse the SSML documents, extract the required information, and represent them in some manner appropriate to our system. Since each prompt is queued before it is played, we have two opportunities to parse the SSML - VXIpromptQueue and VXIpromptPlay. VXIpromptQueue takes a number of parameters, including a pointer to the SSML document itself and a list of properties which include some of the VoiceXML tags' attributes. It makes sense to parse before queuing the prompts, as we have access to the attributes.

The list of attributes passed to the VXIpromptQueue are passed as a VXIMap. The VXIMap data structure is a container for key/value pairs which are of type VXIchar *. The VXIchar type represents the local char type. All the primitive data types used by OpenVXI are declared as typdefs to allow portability across as many operating systems as possible. VXIMaps are used to pass data across interface boundaries, in this case from the interpreter to the prompt interface.

Before parsing can begin, a data structure must be chosen to represent a prompt. A prompt structure was created for this purpose. The attributes required for the structure can be derived from examining both the prompt and audio tags. The prompt structure contains type, bargeIn, fetchtimeout, fetchinit and payload attributes. The type attribute indicates whether the prompt is an audio file or text and bargeIn is a Boolean variable indicating whether or not user input can stop the prompt while it is playing, and is used for both types of prompts. fetchtimeout and fetchinit are audio attributes which control the fetching of an audio file. Finally, the payload attribute is used to store either the URL of the audio file or the text which is to be synthesized.

These account only for some of the more common attributes. The most notable exclusion is the *xml:lang* attribute of the prompt tag, which has been ignored by the parser because we only have one language available at one time. When the Festival server is started, it is configured to use a voice but this functionality has not been made available to the gateway.

Now that a data structure representing a prompt exists, the task of parsing the SSML can begin. The bargeIn property of a prompt is not part of the SSML, and is instead passed to VXIpromptQueue as a property in the VXIMap. An accessor, VXIMapGetProperty, has been provided to get values from a VXIMap by their name. We can therefore use this to set the bargeIn property of the prompt structure.

SSML tags, as can be seen in Section 4.2.3, consist of an *xml* tag followed by a *speak* tag containing a number of attributes, a payload and then finally a trailing speak tag [91]. It is assumed, since the XML document has been created by the interpreter, that it is well formed, and we can disregard the opening xml tag. While the speak tag contains the xml:lang attribute, this is ignored and so the speak tag is also ignored. The first part of the SSML document that is of interest to us is the payload. If the payload is not an audio tag it is assumed that it is text to be synthesized, and the payload attribute of the prompt structure is pointed to the text. If the payload is an audio tag it also has to be parsed and the fetchtimeout, fetchinit and src attributes have to be extracted. In the case of an audio tag, the payload attribute of the prompt structure is set to the value of src. Finally we can set the `type`. The `type` attribute is important because it is used to easily check which system, Asterisk or Festival, must be used to render the prompt.

Once the prompt has been parsed and its corresponding `prompt` data structure has been filled, it can be queued for playing.

4.3.3 Creating the prompt queue

The VoiceXML specification [56] requires a queue to hold those prompts which are to be played by the Implementation Platform. This section will briefly discuss the implementation of such a queue.

It is the responsibility of the FIA to queue the prompts, which it does during the COLLECT phase. As seen in Section 2.2.2.4, the FIA will only queue those prompts which will be played and in the correct order. A prompt which is in the queue will be played unless a disconnect element is encountered, in which case the queue is flushed, or if a hangup is detected. This holds true even if a transfer element is encountered - all prompts in the queue are played before a transfer takes place. Because of this, we know that when the interpreter requests a prompt to be queued it can be unconditionally added to the queue, and once the prompt has been played it can be discarded. Should the same prompt have to be played for a second time, it is the responsibility of the FIA to queue the prompt again.

This behavior greatly simplifies the queue data structure - it is simply a First In First Out (FIFO) queue. A FIFO queue can simply be implemented as a singularly linked list. To create this list a pointer to the next `prompt` is added to the `prompt` structure, and a global variable, `currPrompt`, is created and initialized to `NULL`. This always points to the head of the queue, and is only updated once a prompt has finished executing, not as it begins executing, because of the `bargein` behavior, which will be discussed later. The queue is global because a number of functions need to access the queue, including `VXIpromptPlay`, `VXIpromptQueue` and `VXIpromptStop`.

Adding and removing `prompts` from the queue is straightforward. Once the SSML has been parsed, a prompt is added to the queue according to the following algorithm:

```
create a new prompt
initialize the new prompt's next pointer to NULL
if this is the first member of the queue
        point currPrompt to this prompt
else
        add the prompt to the end of the queue
```

As said above, prompts can be removed from the queue at two stages, once they have been played and when the prompting session shuts down. Once a prompt has finished executing (or has been interrupted by the user) it must be removed from the queue and its memory freed. Removing a prompt from the `queue` is a simple matter of updating the global `currPrompt` to point to the next element in the queue. If `VXIpromptEndSession` is called and there are still elements in the queue it means that there has been an abrupt end to the call; either a disconnect tag was encountered or a hangup was received. In both cases the call has completed and the queue is no longer needed, and so it must be flushed and the memory freed.

The system can now parse, if in a rather rudimentary way, SSML documents and queue them appropriately. It must now be extended to play prompts of either type.

4.3.4 Playing the prompts

As we have seen, there are two types of prompts, audio prompts and synthesized text. This section will discuss the process of playing each type. Once this process has been completed the gateway will be able to produce output.

The FIA, during the COLLECT phase, is responsible for queuing and playing prompts. In terms of OpenVXI this means that the interpreter sends a message to the `prompt` API implementation to play the prompts in the queue, and specifically, in our case, `VXIpromptPlay` is called. Due to the work done by the FIA when queuing the prompts, playing the prompts is as simple as emptying the queue. The prompt at the head of the queue is always the next prompt to be played and the queue can be emptied, barring, say, a hangup. The algorithm to do this is very simple:

```
while there is something in the queue
        if it is an audio prompt
                fetch the file
                use Asterisk to play the file
        else
                use festival to synthesize the prompt
                use Asterisk to play the audio

        /* when we get here the prompt has finished
         * executing and the queue member will not
         * be needed again - discard
         */
        remove the prompt from the queue
        free memory

        return the result code
```

As said earlier,, the `prompt type` attribute is used to distinguish between the two types of prompts. Both types of prompts require some form of acquisition - a file must be fetched or audio must be generated. Once the audio has been obtained it can be rendered on the Asterisk defined channel, which has been passed to the API from the `voicexml` application.

During the execution of the prompts the interpreter is in the WAITING state [56] (the FIA is in the COLLECT phase), which is when the system waits for input. In other words, input is being received while prompts are being processed. This affects the design of the system, as it will be seen in the next section. Currently, however, we are not concerned with input, and we next investigate the rendering of each type of prompt next.

4.3.4.1 Audio

As we have seen, Asterisk can play audio files to the channel, and can therefore be used to execute audio prompts. The version of Asterisk used for this thesis, 1.0.6, comes with support for the GSM, wav and raw formats, amongst others. GSM is the predominant format used by many Asterisk applications but since the wav and raw formats are supported, the gateway complies with the VoiceXML standard. In addition to this, Asterisk provides generic file format support, which is defined in `file.h`. `ast_format_register` is a function which can be used to add support to Asterisk for a specific file format. We will use wav files for the tests.

Since audio prefetching is not currently supported, audio is only fetched once an audio prompt reaches the front of the prompt queue. This may cause delays in some scenarios, but we will ignore them in this version of the system. As already stated, playing the file requires two steps - fetching the audio file from a remote server and playing the file on the channel. It was decided that the `wget` interface would provide the simplest interface for file fetching. `wget` is a Linux utility for downloading files from the Internet and is described in the Linux Manual Pages [65]. `wget` can navigate through proxies, provides support for a number of timeout scenarios and can be configured to only attempt to fetch a file if it does not already exist on the host machine. Since the `prompt` element in the queue has already been parsed, its attributes can be passed straight to `wget`. If the fetch process gives an error (the fetch times out or the file does not exist, for instance) the execution of the prompt is abandoned and the result code `VXIprompt_RESULT_FETCH_ERROR` is returned, indicating to the interpreter that the file could not be fetched.

Should the file be retrieved successfully we can proceed with playing it on the channel. To achieve this we need to stream a file to the channel associated with the prompting session, wait for the file to stop streaming (or be interrupted) while accepting input, and finally stop the streaming.

Examining Asterisk's file handling capabilities in `file.h` we see that a number of file streaming functions are provided. Of interest to us at this point is `ast_streamfile`, `ast_waitstream` and `ast_stopstream`. `ast_streamfile` streams a file to a channel and takes as its arguments the channel to which the file must be streamed, the file name of the file and a language preference. It returns 0 on success or -1 on failure. If the stream completes naturally, 0 is returned. Finally the `ast_stopstream` simply stops a stream on a specified channel.

These functions can be used to stream the audio file to the channel and, because OpenVXI has been linked to the Asterisk libraries, the functions can be called from within the OpenVXI system. We just need to include the header file, `file.h`. The language argument passed to `ast_streamfile` is used by Asterisk to help navigate through its sound libraries, and does not apply to the gateway. We can therefore simply pass the channel and the file name to `ast_streamfile` and let Asterisk handle the rest. If an error occurs during streaming, `VXIprompt_RESULT_HW_ERROR` is returned to the interpreter. `ast_waitstream` is used to wait for input while the file is being played. Finally `ast_stopstream` is called to ensure that the streaming process ends cleanly.

While the streaming process itself is simple, it justifies a number of design decisions and highlights some of the advantages of the system. Firstly, and perhaps most importantly, it justifies the tight coupling between OpenVXI and Asterisk. Should this relationship not exist, another component would have to be obtained to give the gateway the functionality to play audio files, requiring an interface both to Asterisk and the `prompt` API, complicating both the implementation process and the system architecture.

Secondly, it provides a concrete example of why the channel is passed right through to the `prompt` API. While one may assume that since the channel structure represents the call, it should be controlled by the `tel` API, one can see that this is not the case. The `tel` API is used primarily as a means for OpenVXI to interface with the underlying telephony platform to set up and tear down calls. This is handled internally by Asterisk before the `voicexml` application is even called, and so the channel structure now represents the audio channel through which the gateway interacts with the user. Playing audio files on the channel is an example of this interaction.

Thirdly, the benefits of parsing the SSML before queuing the prompt can be seen, as this greatly simplifies the play back process.

Fourthly, this process provides a small example of the use and range of the OpenVXI error codes. Thanks to these codes, creating a robust mechanism to fetch and play audio files was easy. The system can recover, and discriminate between, fetching and playing errors.

Finally, the streaming process provides an opportunity to discuss an example of the advantage of dynamic linking. Should further file format support be added to Asterisk it will immediately become available to the gateway, without requiring even a re-build of OpenVXI.

The gateway is now able to fetch and play audio files for the user to listen to. The following section describes the process of synthesizing prompts and playing the resulting audio to the user.

4.3.4.2 Synthesized speech

Rendering a prompt which is to be synthesized is, in theory, very similar to rendering an audio prompt, and requires two basic steps: obtaining the prompt, in this case the synthesized text, and then streaming it to the channel. That, however, is where the similarities end. Obtaining the prompt means sending the text to a TTS engine, Festival, and receiving the result. Because we are not simply streaming audio files, the process of playing the prompts on the channel is very different.

The first task is to integrate the Festival system into the `prompt` API using the client-server approach discussed in Section 3.3.1. The client-server approach has been selected because it allows concurrent processes, in this case prompting sessions, to access the Festival TTS engine simultaneously while removing any initialization-related time penalties. Another advantage of this approach is that both Festival and Asterisk provide extensive examples of clients. In particular, Asterisk provides an example of streaming the Festival output to a channel, and this is discussed later in this section. While it may be possible to simply use the Asterisk `festival` application to synthesize the text, this is not ideal because it does not give sufficient control of the synthesis process to the `prompt` API. We did, however, rely heavily on the `app_festival.c` file for guidance.

Before we begin with the implementation, Festival had to be installed and configured. Since we will be following a similar approach to the Asterisk Festival client implementation we can install Festival as per Asterisk's requirements. This is discussed on the voip-info wiki [89].

Once Festival is installed, it must be configured before it can be used by Asterisk. There are three ways of doing this: modifying the Festival configuration file, patching the Festival source code or using an external Perl script. Since patching the source violates our requirement for an open system, this option can be ignored. Using an external Perl script is overly complicated. The simplest and most effective way of setting Festival to run with Asterisk is to change the configuration file. The following Scheme command, taken from the wiki, is placed in the `festival.conf` file.

```
(define
  (tts_textasterisk string mode)
    "(tts_textasterisk STRING MODE) Apply tts to STRING.
     This function is specifically designed for use in
     server mode so a single function call may synthesize
     the string. This function name may be added to the
     server safe functions."
  (utt.send.wave.client (utt.wave.resample (utt.wave.rescale
  (utt.synth (eval (list 'Utterance 'Text string))) 5) 8000)))
```

This simply defines the command which will be sent to the server by the Festival client. The full command is of the form `(tts_textasterisk string mode)` where `string` is the string to be synthesized. The final portion of the Scheme command ensures that the text sent back to the client is sampled at 8Khz, and so is suitable for a telephony environment. (This configuration file can also be used to change the voice, and so the language, of the system.)

With the server correctly configured we can proceed with the implementation. In broad terms, obtaining the synthesized text is a simple matter of connecting to the Festival server, sending it the properly formatted command, and receiving the result.

A TCP socket is used for the connection, and a socket is created. By default the Festival server runs on port 1314 and binds to `localhost`. This behavior can be changed in the configuration file and the server can be run on a remote machine, enabling load balancing or separate servers running on separate interfaces which can be used to provide different voices. This is a rudimentary way of adding multilingual support to the gateway. A file descriptor is used as a handle to the socket, and so we can treat the socket as a simple file. (Using Unix file abstraction in this manner provides a simple and powerful way to manage a network connection [73].) If the socket fails to open for whatever reason, `VXIprompt_RESULT_TTS_ERROR` is returned to the interpreter. If the socket is successfully opened we can send the command, and wait for the result, which is sent by the Festival server as a sampled waveform.

The waveform, sent by the server as a file, is read into a buffer by the client. In this way the audio can be treated as an array of characters. Once the waveform has been received, it can be sent to the channel. To do so the waveform must be separated into frames. A Unix pipe (another example of Unix file abstraction) is used to break the waveform up into chunks which will be used as data for a frame.

A Unix pipe is simply a special file with a read end and a write end. A read process will block when trying to read from an empty pipe, and a write process will block when trying to write to a full pipe [51]. Pipes are used extensively in Unix for Inter-Process Communication (IPC). Following the example in app_festival.c, a pipe is created and the process forks. The child process then writes the waveform (in chunks) to the pipe and the parent process reads from the pipe, creates a frame, and writes the frame to the channel, as can be seen in the following algorithm, which has been taken from the Asterisk festival application:

```
for ever
        /* give the last frame a second to play,
         * observing the channel for activity
         */
        wait for 1000 ms
        /* check the state of the channel */
        read a frame from the channel
        /* check the type of frame that has been read
         * most normal frames are classified as voice
         * frames, such as silence, other frames types
         * include video, DTMF, and control
         */
        if the frame is a voice frame
                read the waveform data from the pipe
                create a frame
                write the frame to the channel
        if this is the last of the waveform
                exit
```

There are a few things to note here. One would assume that waiting before sending each frame would result in patchy audio. This, however, is not the case. Because ast_write writes the frames to the channel in a non-blocking manner, we have to wait in between sending frames, otherwise the frames overlap and the audio is garbled. Checking what is on the channel before sending also produces cleaner sounding audio, as the frames are sent only when we can be sure that the user will hear them. For example, if a waveform is being played to a channel and a user pushes a key, the DTMF tone the user hears does not overlap the other audio: the waveform is paused, during execution, while the DTMF tone is played. Another reason for reading from the channel is so that we can detect a hangup. If a hangup is detected we can stop sending the waveform to the channel and exit gracefully. The result code VXIprompt_RESULT_FAILURE is returned if either a hangup is received or ast_write fails for some reason. Otherwise VXIprompt_RESULT_SUCCESS is returned.

This implementation episode signals the end of the user story, and so we can move to the testing stage. Before that it is worth noting that some of the aspects touched on in this section, Unix file abstraction and pipes, will be used later in an attempt to create IPC between the Java Sphinx process and the `voicexml` application, and are revisited in the following chapter.

4.3.5 Running the test

Once again we can verify results of the test by logging strategically in the code and looking at the CLI output.

```
NOTICE[30189]: app_voicexml.c:125 voicexml_exec:
       voicexml called successfully
NOTICE[30189]: app_voicexml.c:126
       voicexml_exec: argument is:
          http://king.rucus.net/test.vxml
saying: Welcome to Adam King's Voice XML page.  bargein = 1
playing audio file: pressOne.wav
  fetchtimeout=7000 fetchinit=2 bargein = 1
--22:00:18-- http://king.rucus.net/pressOne.wav
     => '/tmp/app_voicexml/SIP/1000-df29/pressOne.wav'
Resolving king.rucus.net... 146.231.115.12
Connecting to king.rucus.net[146.231.115.12]:80... connected.
HTTP request sent, awaiting response...
    200 OK Length: 18,810 [text/plain]
100%[====================================>] 18,810 --.--K/s
22:00:19 (15.72 MB/s) -
  '/tmp/app_voicexml/SIP/1000-df29/pressOne.wav'
    saved [18810/18810]
-- Playing '/tmp/app_voicexml/SIP/1000-df29/pressOne'
    (language 'en')
```

We can see that the SSML we saw logged in Section 4.2.3 has been parsed, the important information extracted, and the text synthesized. The argument passed from the dial plan to the `voicexml` application is the URL of the VoiceXML document to be parsed. The two prompts are queued, parsed and played. Firstly "Welcome to Adam King's Voice XML page" is synthesized, and then the audio file is played. The audio file to play is pressOne.`wav`, which is fetched from the web server and saved

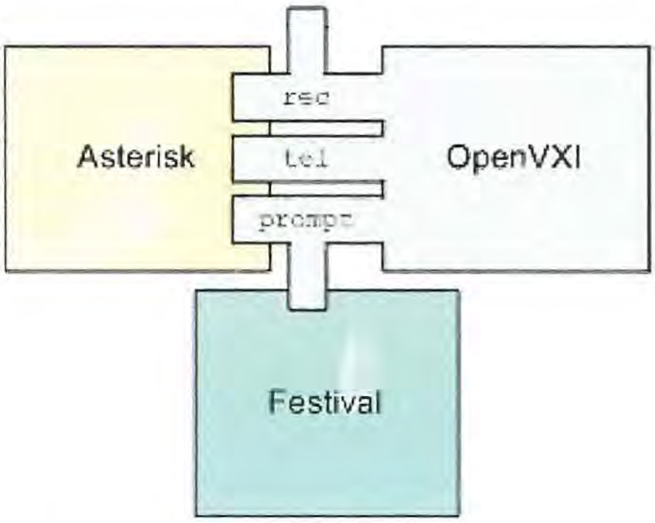


Figure 4.2: The new architecture reflecting the integration of Festival

to the session's unique work space, in this case the `/tmp/app_voicexml/SIP/1000-df29/` directory. The workspace is created using the unique channel name, in this case `SIP/1000-df29`, which allows us to clean up at the end of a session safely, without running the risk of removing files that have just been downloaded or are in use. Finally, Asterisk streams the file to the channel.

Comparing the synthesized text output from the gateway with Festival's normal output we do not hear any difference in quality, and so we can assume that the integration was successful.

It can be seen that the test was passed. The resulting architecture can be seen in Figure 4.2, with Festival interfacing with the `prompt` API.

4.4 Summary

This chapter described the implementation of the gateway's output functionality. This involved integrating Asterisk, OpenVXI and Festival. Asterisk and OpenVXI now form the core of the system.

To integrate Asterisk and OpenVXI an Asterisk application module, `voicexml`, was created, and the Asterisk and OpenVXI libraries were linked. This gives the application the capability to interpret VoiceXML documents.

An SSML parser was then written to handle prompt requests from the interpreter and a queue was created to hold the parsed prompts. Prompts are either audio files or text, which is to be synthesized. Asterisk was used to play the audio files. The client-server API of the Festival system was used to convert text into a waveform, which is then played on an audio channel. Testing the system showed that both types of prompts could be rendered.

The final element of the gateway - input - will be discussed in the next chapter.

Chapter 5

Building the gateway: Phase 2

This chapter discusses the second half of the implementation phase - adding input functionality to the system, essentially giving the gateway ears. In particular, we will discuss adding DTMF and speech recognition support. Integrating Sphinx into the system requires integrating a Java application with Asterisk, which is written in C. Two different approaches to this will be discussed, one which failed and one which was successful.

The integration step described in this chapter results in the final system, and so the final architecture will be described. A final test involving concurrent calls will then be conducted to test the complete functionality of the gateway.

Before implementation can begin, a test has to be created. Should the test be successful, we can claim to have created a low-cost, open-source, proof-of-concept VoiceXML gateway, as per the aims of this thesis.

5.1 Creating the Test

The test is the simple part of the process: all we need to do is create a simple VoiceXML application which requires both DTMF and speech input. Working from the original example (Figure 2.3) we will add a greeting, using a simple grammar. The VoiceXML required to do this is as follows:

```
<audio src="pressOne.wav"/>
<choice dtmf="1" next="#one"/>
<form id="one">
     <block>
          <prompt>
               Congratulations, you pressed one
          </prompt>
     </block>
</form>
<grammar type="text/gsl"> [ (Good Morning) (Hello) ] </grammar>
<audio src="test.gsm"/>
<filled>
     <prompt> Hello user </prompt>
</filled>
```

The VoiceXML code simply creates a greeting. If the user greets the VoiceXML application, the application answers with "Hello user". The grammar accepts either "Good Morning" or "Hello".

The test makes use of all the components introduced in Chapter 3. It does not test dynamic grammars and n-best recognition results because, while required by the VoiceXML specification and provided by Sphinx, they have not been implemented. The ability to change grammars would provide a means of using only the active VoiceXML grammars when listening for input, and so may improve the recognition performance of the system, as smaller grammars result in a lower word error rate. Unfortunately, due to time constraints, support for dynamic grammars has not been added.

5.2 Adding DTMF support

Input is collected during the execution of prompts, and therefore the output functions discussed in the previous section need to be extended to capture DTMF input as well. (The specification [56] states that input is only collected during the play back of prompts.) So, there are two areas which have to be altered, the streaming of the audio file and the streaming of the waveform from Festival. Another form of input which we can detect is a hangup, and that must also be dealt with.

Asterisk includes a DTMF recognizer, so we simply have to read from the channel while outputting both types of prompts and check for any DTMF frames or `NULL` frames, which indicate a hang up. We also have to implement the `bargein` attribute.

As the output process already reads from the channel, this is simple and our algorithm only changes slightly.

```
for ever
        /* wait for the last frame to play */
        wait for 1000 ms
        /* check the state of the channel */
        read a frame from the channel
        if the frame is null
                inform the tel API
                stop the prompt
                exit
        if the frame is a DTMF frame
                if the prompt's bargein property is true
                        set the result in the rec API
                        stop the prompt
                        exit
        if the frame is a voice frame
                read the waveform data from the pipe
                create a frame
                write the frame to the channel
        if this is the last of the waveform
                exit
```

Adding this behavior presents a number of design questions, which we will discuss here before adding support for DTMF input during audio prompt execution.

The `prompt` API needs access to both the `rec` and `tel` APIs. This is done by altering the `VXIpromptImpl` structure to include pointers to the other APIs and initializing the pointers when the prompt session begins. What this means is that all events are caught by the `prompt` API, which is then responsible for informing the other APIs, resulting in a somewhat prompt-centric architecture, and altering the design of the system slightly. Given the way VoiceXML behaves, however, the new design is acceptable. A VoiceXML application begins, from the user's point of view, with a prompt. Any input received before a prompt is played is disregarded, due to the behavior of the FIA. Input (and events) are detected during a prompt's execution, after which the next prompt is played. We can see that the nature of a VoiceXML application is itself very prompt-centric, and this has been reflected in the implementation.

The next design consideration is how to handle input with regards to the `bargein` attribute. If the prompt's `bargein` attribute is set to true, a user may interrupt a prompt with an utterance or a DTMF key. This interruption happens at the first indication of any input. It is the VoiceXML developer's responsibility to ensure that multiple inputs, such as a telephone number, are acquired correctly using grammars. The `hotword` attribute of the `<prompt>` attribute can be used to ensure that a prompt will only be interrupted once a grammar is matched, allowing for multiple inputs and barge-in during the execution of a prompt.

The specification describes how to handle input received during the execution of prompts with a `bargein` property of false [56]. Any DTMF input received in such a situation is not buffered and so can be ignored. This means that prompts which can not be interrupted are used only to render information and are not used to collect input. This behavior seems to match the behavior of traditional IVRs.

When interrupting the prompt it is important to set the the appropriate variable in the `rec` API to the recognized token before stopping the prompt. The FIA enters the PROCESS phase once the COLLECT phase is finished, and so, once the prompt finishes executing, the interpreter requests the `rec` API to attempt a recognition (match the input against an active grammar). There are two situations when the interpreter will make the request: when a prompt finishes executing naturally, and once a prompt has been interrupted. The input is therefore set before the prompt it interrupted, to ensure that the `rec` interface uses the input that caused the prompt to be interrupted.

The second area which needs to be changed when adding DTMF support is the playback of audio files. Initially `ast_streamfile` was used to play the file. This does not, however, provide any mechanism of collecting input. While this method would suffice for prompts with their bargein attribute set to false, it will does not allow any input collection when `bargeIn` is set to true. There are two options available to provide this. Firstly, a technique similar to the text synthesis process discussed earlier can be used, where we read from the channel while writing to it and evaluate each frame. The other option is to check Asterisk's functionality to see if a mechanism usable in this case is provided. A function called `ast_app_getdata` is available for Asterisk's IVR modality. This function plays an audio file to a channel and gets any DTMF input. As soon as the function gets some input the streaming is stopped. This behavior is precisely what is needed and so DTMF support can be added to the audio prompt very simply. If the prompt's bargein property is set to `false` we can ignore any DTMF input and continue as before. If, however, it is set to true, `ast_app_getdata` is used as opposed to `ast_streamfile`.

Now that the input has been obtained, it has to be sent to the `rec` interface, so that it can be evaluated against the active grammars at the request of the interpreter. A function, `VXIpromptHandleInput`, was created for this purpose. Since we are passing information across interface boundaries, a `VXIMap` was created. A `VXIMap` is a table, which in this case reduces to a key/value pair holding the type and corresponding value of input received. Care must be taken when creating the `VXIMap`, as memory management issues can arise. It is the responsibility of the `prompt` API to set aside memory for the structure, and the responsibility of the `rec` API to ensure that it is freed.

Catering for the input on the `rec` API side is simply a matter of assigning the received `VXIMap` to a global variable. The `rec` interface is responsible for managing all the active grammars. When the interpreter requests a match to be made the global map is used.

We now have a system which is able to completely replace traditional IVR platforms in terms of functionality. It can interact with a user using synthesized speech and audio files and receive user input via DTMF tones. The system, as it is at the moment, can be considered a low cost VoiceXML IVR platform, and could be used to replace many of the IVR systems in use today. The final component of the gateway, speech recognition, can potentially improve user satisfaction. The following section will discuss the integration of Sphinx into the system.

5.3 Adding Sphinx

As discussed in Section 3.4, it was decided to use the Sphinx 4 Java version as opposed to Sphinx 2 or 3, which are written in C++. This section looks at the creation of speech-recognition applications using Sphinx, and then goes on to examine the integration process. The integration process provides a discussion on Inter-Process Communication (IPC) between a Java process and a native process on a Linux platform. The first attempt at this, using the JNI, the Invocation Interface and the Unix file abstraction failed, but will be discussed here because of the lessons learned. We will then look at a client-server implementation, which worked and resulted in speech-recognition support for both the VoiceXML gateway and the Asterisk system.

5.3.1 Creating Sphinx applications

The creation of Spinx applications is discussed in the Sphinx help files [43]. Once the Sphinx system has been installed, a speech enabled application can be created by simply creating three files: a Java application, an XML configuration file and a grammar file.

The Java application is surprisingly simple to create. There are three main classes used in Sphinx applications: the `Recognizer` class, the `Result` class and the `ConfigurationManager` class. The application must load the configuration file and configure the different components using the

ConfigurationManager, capture input for the Recognizer and process the Result returned by the Recognizer.

The Sphinx engine has been designed to be as configurable and modular as possible. To achieve this the ConfigurationManager class uses the configuration file to configure the different components. The configuration file can be used to configure any components as well as to define properties. The components of interest are the front-end pipeline, the grammar and the acoustic model. The front-end pipeline is used to configure the engine to accept the input format, the grammar component is responsible for loading the grammar, and the acoustic model component defines which acoustic model is to be used. As already stated, we will be using the WSJ 8Khz acoustic model, and so the acoustic model component must be suitably configured.

Finally, the grammar file used to define the Result is created. It can be in various formats, including SRGF. Excerpts from a Sphinx application and the corresponding configuration file can be seen in Appendices E and F respectively.

Now that we know how to create a Sphinx application, we can concentrate on the integration, passing the audio from the channel to Sphinx.

5.3.2 JNI and pipes – a first attempt

The integration of the Sphinx Java application into the current system requires bidirectional interaction between Asterisk and Sphinx. All the audio has to be passed from Asterisk to the Sphinx process, and the result has to be passed to OpenVXI's rec API. To ensure that the Sphinx engine has the best chance of making the correct recognition, it is desirable to have the audio stream up and running by the time the VoiceXML interpretation starts. Altering the structure of the voicexml application discussed in Section 4.2.2 we get this:

```
voicexml_exec (channel, data)
    /* deal with the parameters */
    if we have a URL
        /* set up the speech recognition */
        start the sphinx process
        thread and send the audio

        /* set up and run the interpreter */
        initialise the interpreter, passing URL and channel
        allocate the interpreter resources
        interpret the VoiceXML
        /* once we get here we are done with the VoiceXML
         * application, for whatever reason, and can shut down
         */
        shut down the interpreter
        clean up memory
    else there is no URL
        show an error message and exit
```

We can see that the `voicexml` application needs to have two threads of execution. One is responsible for sending the audio to the Sphinx process and another to interpret the VoiceXML. Before any audio can be sent, however, the Sphinx Java process has to be started. To start the Java Virtual Machine (JVM) and run the application, and receive the result from the application, we decided to use the Invocation API and the Java Native Interface (JNI). The JNI can be used to allow native processes and Java applications to interact with each other, as described in [30, 49]. The JNI can be used to incorporate native libraries into Java applications. Java developers may find this necessary in real-time application where Java does not perform adequately, or where existing native libraries already perform the required functions. The Invocation API, on the other hand, can be used by native processes to gain direct control of the JVM. A native process is one which is written in either C or C++. The JNI and Invocation APIs seem ideal for this situation, as they provide the two-way interaction required.

Given that we have a possible means of interaction between the different processes, we can investigate a mechanism for passing the audio from the voicexml application to the Java process. There are a number of issues to keep in mind when designing and implementing such a mechanism. Firstly, audio is read from the channel in Asterisk specific frames. These need to be converted to a constant audio stream to be sent to Sphinx. Secondly, Asterisk is able to handle many different types of audio formats and codecs. While Sphinx is able to handle different types of audio, the type is set in the configuration file and changing the audio format during execution is not possible. Therefore we have to ensure that the audio being sent to the Sphinx process is in the format that matches the chosen acoustic model. Thirdly, Java does not have very good access to low level, platform specific devices, as it is dependent on the JVM. Sending the audio to a Unix abstract device (such as `/dev/null`) on the Linux platform will not work because the Java application will have difficulty in accessing the device in a generic manner.

It was therefore decided to stream the audio from Asterisk to a file which the Sphinx application would access independently. Because of the amount of housekeeping that would be required to ensure that the two independent processes access the file correctly, it was decided to use a pipe.

The pipe will block a write while there is nothing to read, and vice versa. Due to the Unix file abstraction system, the pipe can simply be treated by both processes as a normal file. The Asterisk function `ast_writefile` creates a stream for writing in a specified format. One can use this function to write a stream to a file in a consistent format, which can be read on the other side by the Java application, treating the pipe as a simple wav file. This scenario theoretically solved all our problems. The `ast_writefile` creates a stream of audio of a consistent format which is written to a file. The Java application on the other side can handle the pipe as a normal file. What is left to do is to start the Java process.

Starting the Java process requires another thread to run the JVM and a second pipe for communication between the two processes. Since the initialization of the Sphinx process takes a while, the `voicexml` application needs to wait before proceeding with the VoiceXML interpretation. The initialization penalty can not be avoided, and this means that there is a pause before the VoiceXML begins executing, which is not desirable. The `voicexml` application now looks like this:

```
voicexml_exec (channel, data)
    /* deal with the parameters */
    if we have a URL
        /* set up the speech recognition */
        create a pipe for the audio
        create a pipe for other communication
        create a thread to manage the JVM

        /* to synchronize the processes we can simply
         * read from the communication pipe - this read
         * will block until the Java application writes
         * to the pipe
         */
        read from the communication pipe
        start an audio thread

        /* set up and run the interpreter */
        initialise the interpreter, passing URL and channel
        allocate the interpreter resources
        interpret the VoiceXML
        /* once we get here we are done with the VoiceXML
         * application, for whatever reason, and can shut down
         */
```

```
          shut down the interpreter
          stop the threads
          remove the pipes
          clean up memory
      else there is no URL
          show an error message and exit

/* JVM thread */
start the JVM
start the Sphinx application - passing, as files, both pipes

/* audio thread */
read from the channel
write a stream to the pipe in wav format
```

The Sphinx application simply wrote to the communication pipe once initialized and then read audio from the audio pipe, treating it as a wav file. Finally, the JNI API was used to link the Sphinx application with the `rec` API and set the input. Adding native libraries to a Java application is relatively simple, as described by Schuermann [79]. The native method is declared, without a body, in the Java application and the library is loaded in a static block following the declaration, as follows:

```
private native void setResult(String result);
static { System.loadLibrary("VXIrec"); }
```

`javah`, a C header file generator, is then used to create a C header file from the Java application which is included in the native library. The Sphinx application can now theoretically call `setResult` to interface with the `rec` API.

Once the solution was implemented and tested it was found that not only did it not work, but it had managed to break the DTMF recognition. While the main symptom of failure was that the `ast_writefile` call continuously gave an error, there are also a number of issues which would ensure that the solution would not have worked. We will first discuss the `ast_writefile` issue before discussing the related issues.

Studying the Asterisk file writing process we saw that, after every frame was written, the header of the wav file was updated. Looking at the header structure of a wav file we saw that it included a `chunksize` field used to indicate the size of the wav file [95]. After each write Asterisk attempts to update this header, and this fails. Since a pipe is a file of size zero, the wav header does not exist, and has already been read by the Sphinx process. In fact, it is not possible to `lseek` on a pipe [51]. (`lseek` is used to change the position in the file where writing (or reading) occurs [50].) Since the process of writing to the wav file cannot update the file size header, because the wav file is actually a pipe, it gives an error. The Java program on the other end is being blocked by the behavior of the pipe, as it attempts to read while there is no corresponding write. While this could be changed, it involves altering the Asterisk source code to blindly write to the pipe and neglect updating the wav header. Since we are attempting to avoid altering the Asterisk source, the method of sending the audio to the pipe needed to be revised.

Had the streaming worked, or a suitable alternative been found, the solution still would have failed. The JNI call to a native library is not able to call a particular instance of that library, and so the Sphinx application would not interact with the instance of the rec API associated with the call, and the input would have been lost. Similarly, there is no way to handle concurrent calls, and there is no way to associate an executing instance of a library with a JNI call.

Finally, because there were two process concurrently reading from the channel, on average only half of the DTMF input would be received by the `prompt` API, as the other half is read by the process streaming to the audio file. While this is unacceptable, any solution to the Sphinx integration problem requires a separate thread to send all the audio to the Sphinx process and so this must be solved.

These errors caused a re-consideration of the problem, and a new solution was created based on the client-server architecture.

5.3.3 The client-server approach

Since the first solution was not feasible we had to investigate another approach. It was decided to use a client-server architecture whereby a client, the `voicexml` application, streams the audio to a Sphinx server. Before we begin discussing this approach it is worth reviewing some XP principles which come into play due to the change. Besides using the test based and scope-driven development approaches prescribed by XP, a conscious attempt has continuously been made to keep the code as simple as possible. The effect of this is to minimize the cost of change over time as well as to reduce the possibility of errors in the system. Combining the two separate processes that read from the channel into a single one represents a large change to the design of the system, as the DTMF input solution has to be reworked. Fortunately, because the code has been kept as simple as possible, making the alterations does not affect the progress of the project drastically.

The solution to the channel access problem is to only read from the channel at a single point - when sending the audio to the Sphinx application. Any DTMF input received is simply sent to the prompt API where it is dealt with, as before, by the `VXIpromptHandleInput` function.

It was decided to create first a simple Asterisk recognition application to see whether the client-server approach was viable, and therefore another application module, `app_recognize.c`, was created, in the same manner as `app_voicexml.c`. Should the new approach work, it would provide a speech recognition system for Asterisk itself, as well as for the gateway.

Fortunately, the `recognize` application was very similar to the `voicexml` application discussed in the previous section without, of course, the OpenVXI integration. Instead of writing to a pipe audio had to be written to a socket. (Due to Unix file abstraction, sockets and pipes are essentially equivalent, and can be treated in the same manner.) So, instead of opening a file stream we now opened a socket and connect to the server.

Once a connection to the server was made, the client had to synchronize with the Sphinx process, like before. The synchronization was necessary to ensure that audio was only sent once the server was ready to accept it and that the VoiceXML application did not begin execution before such a time. Once the server was ready, the client could send the audio.

As with the previous approach, the problem of audio formats had to be addressed. The client had to ensure that the same format was consistently sent to the server, and that the format was one which could be handled by the server. The native audio format used by Asterisk is PCM SLINEAR, which is 8Khz, signed, big endian audio with 160 bytes per sample, and any Asterisk supported audio should be able to be converted into this PCM SLINEAR format. Since we had already chosen an 8Khz acoustic model, we simply had to configure the front-end pipeline to accept this audio, as it can be seen in Appendix F.

The recognize application behaves as follows:

```
recognize_exec (channel, data)

    open a socket to the sphinx server
    create a thread to listen for results
    connect to the sphinx server
    wait for the sphinx server to initialize
    for ever
        if the audio is not PCM SLINEAR
            convert the audio
        send the audio to the socket

/* result thread */
for ever
```

```
listen to the socket
log any results to the CLI
```

Now that we have a client that can connect to a server and send correctly formatted audio, we can look at implementing the corresponding server. Since the responsibility of the server is to simply farm out Sphinx processes, it was called `SphinxFarm`.

5.3.4 SphinxFarm

`SphinxFarm` is a Java TCP server which farms out Java processes, in our case Sphinx applications, to any incoming connections. As seen previously, the main issue associated with integrating Sphinx and Asterisk is the transport of the audio from the C application to the Java application. The device used in this scenario, a socket, is again easily accessible by a Java application.

The choice of Java as an implementation language was made because we are farming out Java processes. It was also found that because of the way in which Java handles exceptions, a great deal of robustness could be easily built into the server.

TCP has been chosen because it encapsulates the notion of a connection, while UDP is connectionless. We need to ensure that the client is sending the audio to the correct Sphinx process, and that the Sphinx process in return is sending the results to the correct client, and providing this behavior using TCP is simpler than using UDP. If UDP was used we would have had to create some form of connection, either by using another TCP connection to negotiate a separate port for communication between the client and the thread associated with it, or by manipulating the UDP packets to provide some TCP functionality.

The final issue affecting the design of the `SphinxFarm` application is the issue of time penalties. `SphinxFarm` must attempt to minimize any Sphinx related time penalties - in particular the initialization delay. This has been done by pre-initializing the Sphinx processes at start up. The server initializes a number of Sphinx processes before opening a socket and waiting for connections.

The pre-initialization process works as follows. Each Sphinx process is run in a separate thread, and can be in one of three states - INITIALIZED, CONNECTED or WAITING. The INITIALIZED state refers to a thread containing a Sphinx process, which has been initialized but has not yet been run. (Specifically, its `run` method has not been called.) A thread in the CONNECTED state is one which is busy recognizing speech, and is connected to a client. The WAITING state refers to a thread which has already been run and has dealt with at least one client connection, but is currently idle. A WAITING thread, in other words, is one which has been in the CONNECTED state at least once, while an

INITIALIZED thread has never been. Because the number of connections is theoretically unbounded, a vector has been used to keep track of the threads. A load argument, passed to the server at start up, is used to determine the amount of Sphinx processes to pre-initialize.

Once initialization is complete, the server opens a socket and listens for connections. On an incoming connection the server first attempts to find a thread in the INITIALIZED or WAITING state before creating a new thread as follows:

```
for each thread in the vector
   synchronized
     if the thread is initialized and is not connected
        use this thread
/* there are no such threads */
create a new thread
```

Since both the server process and the client process itself can change the state variables, any access of the variables is done in a `synchronized` block statement. Once a suitable thread has been selected, the server passes the socket to the Shpinx process and executes the thread. Executing the thread involves either running it for the first time, in the case of an INITIALIZED thread, or waking a thread, in the case of a WAITING thread. To wake a thread a process simply calls the `notify` method.

On the Sphinx side, on initialization the Sphinx front-end must load and configure the components and allocate the resources. This is what takes time, and means that the value of the load argument should be carefully selected. Setting the value too low means that calls will, at first, take an initialization penalty, as not enough processes have been initialized. Setting the value too high will result in useless consumption of system resources. If the load value is never exceeded (the value is greater than the highest number of concurrent calls), there should never be any time penalties, and a client process will have immediate access to a voice recognition service. Initialization was timed on two separate machines, A and B. A had an Intel Pentium 4 CPU 3.00 GHz with 1 GB RAM and B had an Intel Xeon CPU 2.40 GHz CPU with 1 GB RAM. A is a desktop PC which was running many other processes at the time. It was observed that initialization on A took around 1 minute, while on B initialization took around 8 seconds, which illustrates that the initialization time can be substantial, and therefore it is well worth avoiding.

Once the process is initialized, it waits for the server to pass it a socket, at which point the socket's input stream is set as Sphinx's data source, and the process can be activated.

Upon activation, the process must indicate to the client application, using the socket, that it is ready to receive audio. This is done to synchronize with the client process, as in the event of the load value being exceeded the client has to wait for the new process to initialize before proceeding. After confirmation, the recognition process can continue as normal, obtaining results and sending them to the client process over the socket.

When a client disconnects, the connection is closed, the thread's state is updated to WAITING and the thread calls the `wait` method, which causes the thread to wait until the server calls `notify`.

The client-server approach has a number of advantages over the wav file approach. Firstly, and most importantly, it works, as will be shown in the following section. Secondly, being able to pre-initialize threads means that we can eliminate the initialization time penalty. Finally, being able to suspend threads and change their input streams means that initialized processes can be assigned to many different clients. The thread is immediately available after a client disconnects.

5.3.5 Testing the recognize application

This test has been designed with a number of issues in mind. Firstly, it must test the integration between the two systems. This involves checking that the audio is sent to Sphinx and that the client is getting recognized tokens back: this can be done by logging the result from the Sphinx application. Testing that the result of the recognition is returned to the Asterisk application is done by simply displaying text read from the socket to the CLI.

Secondly, the transitioning of the thread states must be tested. The following scenarios must be tested for:

- a single call connecting to an INITIALIZED process
- a single call connecting to a WAITING process
- the load value being exceeded
- concurrent calls

Finally, an indication of the word error rate is required. Comparing this error rate with the native error rate of Sphinx in identical conditions helps determine whether the audio transfer mechanism, the socket, has been implemented correctly. In what follows, `[n]` represents the thread's index in the vector. Since the Asterisk server and the `SphinxFarm` server are both running on the same machine, all the connections are from `127.0.0.1`.

Testing for a single call connecting to an initialized thread and then to a waiting thread:

```
[0] --> Initialising
Server initialization successful!
  Bound to: 127.0.0.1:1315
  Pre-loaded clients:     1

[0] --> Connected.
[0] --> Hang up - pausing thread
New connection from /127.0.0.1
[0] --> Connected.
```

Testing for the load value being exceeded. This tests simply did not initialize any processes, and so the first connection (at index 0) results in the load being exceeded. The following output was observed:

```
Server initialization successful!
  Bound to: 127.0.0.1:1315
  Pre-loaded clients: 0

New connection from /127.0.0.1
load exceeded - initializing
[0] --> Connected.
```

Finally, testing with concurrent calls:

```
[0] --> Initializing
[1] --> Initializing
Server initialization successful!
  Bound to: 127.0.0.1:1315
  Pre-loaded clients: 2

New connection from /127.0.0.1
[0] --> Connected.
New connection from /127.0.0.1
[1] --> Connected.
[1] --> Hang up - pausing thread
[0] --> Hang up - pausing thread
New connection from /127.0.0.1
[0] --> Connected.
```

	TEST A	TEST B	TEST C
First Utterance	100%	95%	75%
Second Utterance	90%	80%	65%
Full Utterance	90%	80%	65%

Table 5.1: Sample test results

As we can see from the tests, the mechanism for minimizing initialization time penalties is working correctly. There are no initialization time penalties when the number of clients is less than the given load value and when the load is exceeded a new process is created properly. We can now move on to assessing the recognition performance of the system.

Three very simple tests were run using a variety of speakers and grammars. The test subject for test A was a South African male and test B an Australian female. Tests A and B made use of the same grammar, while test C introduced a different grammar, the speaker being the same as in test A. The grammars were as follows:

```
grammar testAB;
public <greeting> =
  ( (Hello | Good Morning) ( Adam | Bradley | Will | Rita ) );
grammar testC;
public <direct> =
  ( call to | move to ) ( quick fox | Bradley | Adam );
```

The speaker's actual utterance was evaluated against the result produced by `SphinxFarm`. The test results can be seen in Table 5.1, and give some indication of the word error rate that can be expected. Unfortunately, the word error rates of between 10% and 35% that have been observed are very large. Since the value is expected to be around 7% (Table 3.1) this is a disappointing result.

The discrepancy between the different values can be due to a number of things, one of which may be the integration mechanism – incorrectly formatted audio being sent to the Sphinx process for example. In an attempt to rule out the integration, another test was conducted. Using the same telephone, we recorded a set of utterances, conforming to the grammars above, to file using Asterisk's recording functionality. A Sphinx front-end was created to recognize utterances from a file and the recorded files were passed to the Sphinx system. An example of the front end output was as follows:

```
Loading Recognizer...
Decoding
  /usr/src/sphinx4/src/demo/sphinx/wavfile/test.wav
```

```
WAVE (.wav) file, byte length: 62924,
  data format: PCM_SIGNED 8000.0 Hz,
  16 bit, mono, 2 bytes/frame, little-endian,
  frame length: 31440
RESULT: good morning rita
```

It was observed that saving the utterances as files before decoding them resulted in an error rate of approximately 30%, which falls within the range of the `SphinxFarm` test results. This result strongly indicates that the problem does not lie in the integration, but somewhere else instead.

While it is not been determined what is causing the poor recognition performance, the fact that the WSJ acoustic model comprises of American accented voices, while our accent is South African must account for at least some of the discrepancy. Creating our own acoustic model would clarify this problem. This was, however, outside the scope of the thesis.

Finally, since we can assume that the process itself is correct, we can integrate Sphinx into the `voicexml` application in a similar manner to what has been done in the `recognize` application. The `recognize` application also extends the Asterisk system independently of the gateway implementation.

5.4 Results

Running the test described in Section 5.1 we can observe the following logging excerpts from the CLI:

```
NOTICE[30189]: app_voicexml.c:125 voicexml_exec:
    voicexml called successfully
NOTICE[30189]: app_voicexml.c:126
    voicexml_exec: argument is: http://king.rucus.net/test.vxml
saying: Welcome to Adam King's Voice XML page.  bargein = 1
/* fetch and play pressOne */
NOTICE[11261]:
  app_voicexml.c:301 asr_stream: DTMF input received 1
setting dtmf input as 1
saying: Congratulations, you pressed one  bargein = 1
/* fetch and play test */
```

```
NOTICE[11331]:
  app_voicexml.c:286 asr_stream: Sphinx recognized: --> hello
setting speech input as hello
saying: Hello user  bargein = 1
```

Of course, if the user did not provide any input or the input was not correct (which it will be for speech input approximately 10-35% of the time) `nomatch` or `noinput` errors were thrown and the VoiceXML default behavior was observed, as can be seen in the following example:

```
saying: Welcome to Adam King's Voice XML page.
  bargein = 1
app_voicexml.c:301 asr_stream: DTMF input received 2
saying: Sorry, I didn't understand you.
  bargein = 1
saying: Welcome to Adam King's Voice XML page.
  bargein = 1
saying: Welcome to Adam King's Voice XML page.
  bargein = 1
saying: Sorry, I didn't hear you. bargein = 1
saying: Welcome to Adam King's Voice XML page.
  bargein = 1
saying: I didn't hear you that time either.
  bargein = 1
```

Looking at the tests we can see that the DTMF collection is reliable and, while perhaps not quite suitable for real world purposes, speech input can be received and handled correctly.

The final architecture can be seen in Figure 5.1. It can be seen that the architecture is slightly different from the architecture shown in Figure 3.2, as the initial architecture assumed that Sphinx would only be integrated into OpenVXI's `rec` interface. One can see, however, that this is not the case, and there is interaction between Asterisk and Sphinx. The audio is being passed from the voicexml module to the Sphinx process, which represents the integration of Sphinx and Asterisk as well.

5.5 Final testing

After creating a complete gateway we needed to test the system a little more thoroughly. To do this we used tests similar to those created for testing the various steps taken in the integration process

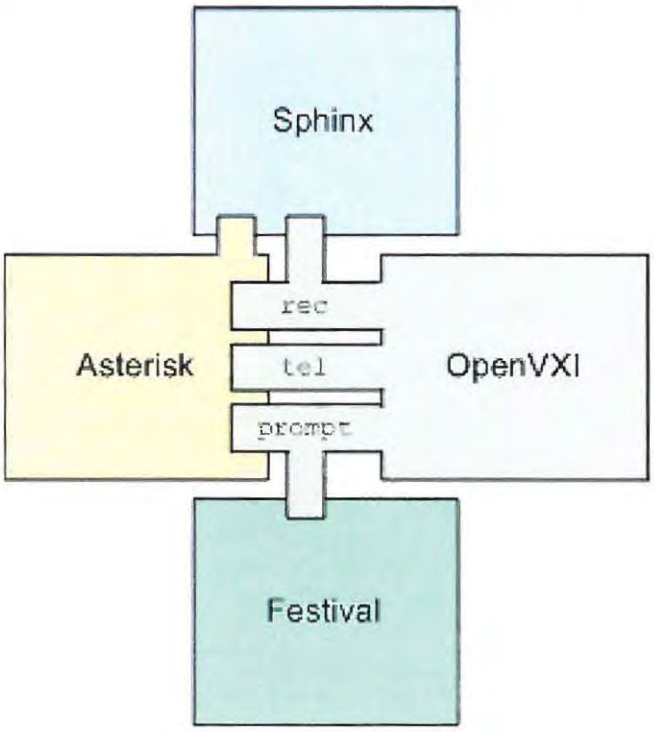


Figure 5.1: The final gateway architecture

and run them with concurrent calls. We also executed successive calls, to show that the system was stable. The unique channel name, provided by Asterisk, was added to the logging information to discriminate between the different calls.

Output from the CLI is as follows (to improve readability, the unique identifier representing the first call is shown in bold and italics, and the second call just in bold):

```
app_voicexml.c:121 voicexml_exec:
  SIP/1001-f1f1 - voicexml called successfully
app_voicexml.c:122 voicexml_exec:
  SIP/1001-f1f1 - argument is:
    http://frog.ict.ru.ac.za/helloWorld.xml
SIP/1001-f1f1 - saying:
  Press or say one, two, three or hash. bargein = 1
app_voicexml.c:121 voicexml_exec:
  SIP/1000-4282 - voicexml called successfully
app_voicexml.c:122 voicexml_exec:
  SIP/1000-4282 - argument is:
  http://frog.ict.ru.ac.za/helloWorld.xml
SIP/1000-4282 - saying:
  Press or say one, two, three or hash. bargein = 1
SIP/1000-4282 - setting dtmf input as 1
SIP/1000-4282 - saying:
  Congratulations, you pressed one  bargein = 1
SIP/1000-4282 - playing audio file: test.gsm
  fetchtimeout=7000 fetchinit=2 bargein = 1
http://frog.ict.ru.ac.za/test.gsm
     => '/tmp/app_voicexml/SIP/1000-4282/test.gsm'
  '/tmp/app_voicexml/SIP/1000-4282/test.gsm'
    saved [18810/18810]
-- Playing '/tmp/app_voicexml/SIP/1000-4282/test.gsm'
   (language 'en')

 SIP/1001-f1f1 - setting dtmf input as 2
SIP/1001-f1f1 - saying:
  Congratulations, you pressed one  bargein = 1
 SIP/1001-f1f1 - playing audio file: test.gsm
  fetchtimeout=7000 fetchinit=2 bargein = 1
```

```
http://frog.ict.ru.ac.za/test.gsm
      => '/tmp/app_voicexml/SIP/1001-f1f1/test.gsm'
   '/tmp/app_voicexml/SIP/1001-f1f1/test.gsm'
     saved [18810/18810]
-- Playing '/tmp/app_voicexml/SIP/1001-f1f1/test.gsm'
    (language 'en')
SIP/1000-4282 - app_voicexml.c:286 asr_stream:
   Sphinx recognized: --> hello
SIP/1000-4282 - setting speech input as hello
SIP/1000-4282 - saying: Hello user  bargein = 1
```

We can see two calls accessing the same VoiceXML document. Call A (`1001-f1f1`) and call B (`1000-4282`) happen simultaneously. B enters 1 before any input is received from A. B then goes on to play an audio file. During the playing of B's audio prompt A inputs 2, and the audio prompt is played to A. Finally, B says "hello" and the corresponding prompt is played. A does not enter any more input.

As can be seen, the system can handle concurrent calls, accepting input from each, and we can therefore conclude that the expected gateway functionality is finally available.

5.6 Summary

This chapter described extending the system to handle input. In particular we discussed adding DTMF support and integrating Sphinx, which is Java-based.

Adding DTMF functionality involved obtaining the token representing the DTMF tone from Asterisk and setting this value in the `rec` interface. Since Asterisk provides DTMF tone recognition, obtaining the corresponding token was a simple matter of reading from the channel, during the execution of a prompt which could be interrupted, and collecting any frames containing DTMF tokens. These tokens were passed to the `rec` interface using a `VXIMap`, and, if appropriate, the execution of the prompt was interrupted. Any input received during the execution of a prompt which can not be interrupted is ignored.

Linking Sphinx system required integrating a Java application into Asterisk, which is written in C. Initially the Java Native Interface (JNI) and Unix file abstraction were tried. The Sphinx process was started by the Invocation API. A wav file (represented by a Unix pipe) was then used to transfer the audio from the Asterisk application to the Sphinx process. The `voicexml` application wrote all the audio from the call to the pipe. This approach failed, because the writing process could not update the wav file header, and so we investigated another approach.

The second approach, based on a client-server model, was successful. We created `SphinxFarm`, a Java server which serves Sphinx processes. The `voicexml` application was extended to connect to an initialized Sphinx process and send audio over a socket. `SphinxFarm` is responsible for managing the processes, in such a way as to minimize any initialization time delays, associating a client request with an initialized Sphinx process.

While this approach has been found to be successful, a poor word error rate of between 65% and 90% was observed. This will have to be improved before this solution can be used in a real world application.

Chapter 6

Conclusion

The purpose of this chapter is to review and summarize the work that has been done in this thesis. The completed work will then be compared to the aims of the project as discussed in Chapter 1 to show that we have achieved our goal. We will also discuss future work and extensions to the project.

6.1 Completed work

The work that has been done can be broadly organized into four distinct categories: methodologies were chosen to aid the construction process, the concepts of voice-enabled applications, VoiceXML and the Asterisk system were explored, components were selected and discussed and, finally, these components were integrated to create the gateway.

Due to the time constraints involved with the project, building the gateway from scratch was unfeasible. We therefore needed to select a software engineering methodology which could reduce development time. As seen in Section 1.2.1, the CBSE approach is designed to do exactly that. A system is built by integrating a set of components. This approach drastically reduces development time while improving the flexibility and maintainability of the system. Next a methodology governing the implementation process was chosen. eXtreme Programming, discussed in Section 1.2.2, was used. Scope-driven development gave us the order of implementation, ensuring that the most important parts of the system were implemented first.

Once the methodologies were selected, voice-enabled applications, the VoiceXML standard and Asterisk were discussed in Chapter 2. We saw that voice-enabled applications are being used in call centers, in the health sector, for example to assist the elderly and impaired, and, in general, as an information interface. These applications can all be created using VoiceXML. The VoiceXML specification, first released in 2000, can be used to create voice-enabled applications. It was found that a VoiceXML gateway consists of a VoiceXML interpreter and an input and output component. Inputs can either be DTMF tones or speech fragments and outputs are either synthesized speech or audio files. We have seen how the VoiceXML interpreter interprets a VoiceXML document by implementing the FIA. We defined a VoiceXML gateway as a VoiceXML browser which is able to interface with the PSTN. Finally, Chapter 2 discussed the Asterisk telephony platform. It was shown how Asterisk has the potential to be used as the starting point to create a low-cost PSTN gateway, and how the Asterisk system can be extended using the Application Module API.

Next, the component selection process was discussed in Chapter 3. The main requirements for a VoiceXML gateway were found to be an interpreter, a TTS system and an ASR system. OpenVXI, Festival and Sphinx 4 were selected to provide these required units of functionality. OpenVXI is a framework for building VoiceXML gateways. It provides a set of APIs which have to be "filled-out" to create a functional gateway. The APIs represent the gateway's input, output and telephony capabilities. Festival was selected to provide text synthesis functionality and the Java based Sphinx 4 was selected as the speech recognition engine. Spinx 4 was selected over the C++ based Sphinx 3 because of its better performance.

Finally, Chapters 4 and 5 discussed the actual implementation of the system. Chapter 4 discussed the creation of the gateway's prompting functionality. This involved creating an Asterisk application, `voicexml`, linking the OpenVXI libraries and adding audio file playback and speech synthesis functionality. Asterisk was used to play audio files and the Festival system was integrated into OpenVXI's `prompt` API to provide text synthesis.

Chapter 5 discussed the second half of the implementation effort, adding input functionality. DTMF support was added using Asterisk's DTMF recognizer. Integrating the Sphinx system with Asterisk was more difficult and required two attempts. In the second, we made use of a client-server structure, where a server, `SphinxFarm`, served Sphinx processes to individual `voicexml` processes. This allowed the Java and C processes to interact using a socket, which was also used to pass the audio between the two processes.

Integrating all the components resulted in a proof-of-concept VoiceXML gateway. Revisiting the thesis aims, we can evaluate the system created by the implementation process to gauge whether or not the goals have been met.

6.2 Achievements

In Section 1.1 the question was posed as to whether it was possible to create a low cost, open source VoiceXML gateway using the Asterisk telephony platform. To this we added the request to create an open COTS-based system, to decrease the time required for implementation, and to maintain clear boundaries for each component.

This thesis has described the construction of a low-cost, open-source, proof-of-concept VoiceXML gateway. Asterisk can be used to create a PSTN gateway for as little as R360 (in addition to the cost of a PC), and so the main cause of high-cost VoiceXML gateways (the expensive PSTN hardware) has been avoided. Asterisk was then extended by adding open-source COTS systems to create the proof-of-concept gateway.

Tests have shown that system is capable of rendering prompts represented as SSML documents. It is able to fetch audio files from a remote web server and play them to a user as well as synthesizing text and outputting the resulting waveform.

Support for both DTMF and speech input has been provided. The speech-recognition component of the system has a high word error rate. Tests have shown, however, that this is not due to an implementation error, but is instead due to some other factor, possibly the lack of a comprehensive South African acoustic model.

We can therefore positively answer the question from Section 1.1: it is possible to create a low-cost open-source proof-of-concept VoiceXML gateway.

6.3 Future work and extensions

There are a number of areas where further work can be done.

Firstly, the gateway must be made fully standards compliant. The transfer and record tags have to be supported, and dynamic grammar handling and n-best recognition must be added to the `SphinxFarm` system. The functionality required for the transfer and record tags is provided by Asterisk, and since Asterisk is already linked to OpenVXI, supporting these tags should not require much effort. Dynamic grammars and n-best recognition are also supported by Sphinx, and they should be integrated into the gateway. Finally, the system must be extended to handle SSML documents. Since Festival can already handle SSML this simply involves updating the commands sent to the Festival server by the client.

The second area requiring work is the speech-recognition performance. The use of speech recognition in voice-enabled applications can improve customer satisfaction, and providing this feature adequately is important, besides being required to be compliant. The cause of the poor recognition rate must be identified and repaired.

Thirdly, the system must be tested for robustness and scalability. A large amount of telephone calls must be made to the system, simulating a real world scenario, which will indicate whether or not the system is robust and can scale appropriately.

Finally, being South African, it would be nice to include better language support – specifically for African languages. Adding voices to Festival is a relatively simple task, and to extend the system to be able, at least, to prompt in a multi-lingual fashion, is desirable. This can improve access to information for a larger section of the population.

6.4 Conclusion

VoiceXML is capable of creating voice-enabled applications. In addition to this, VoiceXML gateways can be used to bridge both IP and PSTN networks, from the point of view of information distribution and collection.

We have proved, through the building of a proof-of-concept prototype, that Asterisk can be extended to create a low-cost, open-source VoiceXML gateway, which can be used to render these VoiceXML-based voice-enabled applications. Before this system is used in the real world, it must be made fully standards compliant.

References

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile software development methods - Review and analysis. Technical Report 478, VTT PUBLICATIONS, 2002.

[2] Ali Khan. A Tale of Two Methodologies for Web Development: Heavyweight vs Agile. Masters Thesis, Department of Information Systems, The University of Melbourne, Australia, June 2004.

[3] Jon Anton. The past, present and future of customer access centers. *International Journal of Service Industry Management*, 11(2):120–130, 2000.

[4] Chris Bajorek. VoiceXML - Taking IVR to the Next Level, 2000. Avaliable at http://www.cconvergence.com/article/CTM20000927S0003 [Last Accessed: 23 November 2006].

[5] Rick Beasley, John O'Reilly, Kenneth Michael Farley, and Leon Henry Squire. *Voice Application Development with Voice XML*. Sams, Indianapolis, IN, USA, 2001.

[6] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[7] Alan W. Black, Paul A. Taylor, and Richard Caley. The Festival Speech Synthesis System: System documentation. Technical report, Human Communciation Research Centre, University of Edinburgh, University of Edinburgh, UK, 2002. Avaliable at http://festvox.org/docs/manual-1.4.3 [Last Accessed: 23 November 2006].

[8] Barry Boehm. Get Ready for Agile Methods, with Care. *Computer*, 35(1):64–69, 2002.

[9] Kristy Bradnum. VoiceXML: A Field Evaluation. Honour's thesis, Department of Computer Science, Rhodes University, Grahamstown, South Africa, November 2004.

[10] Alan W. Brown and Kurt C. Wallnau. The Current State of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[11] Lawrence Brown, Noah Gans, Avishai Mandelbaum, Anat Sakov, Haipeng Shen, Sergey Zeltyn, and Linda. Statistical Analysis of a Telephone Call Center:A Queueing-Science Perspective. *Journal of the American Statistical Association*, 100:36–50, 2005.

[12] Mike Brown, Dan Burnett, Emily Candell, Jerry Carter, Debbie Dahl, Debajit Ghosh, Andrew Hunt, Stefan Krause, Sol Lerner, Bruce Lucas, Jens Marschner, Scott McGlashan, Yves Normandin, Brad Porter, Dave Raggett, David Ramsthaler, Luc Van Tichelen, Kuansan Wang, and Laura Werner. Speech Recognition Grammar Specification Version 1.0, 2004. Avaliable at http://www.w3.org/TR/speech-grammar [Last Accessed: 23 November 2006].

[13] Xia Cai, Michael R. Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In *APSEC*, page 372, 2000.

[14] Carnegie Mellon University and Sun Microsystems and Mitsubishi Electric Research Laboratories. Sphinx-4. Technical report. Avaliable at http://cmusphinx.sourceforge.net/sphinx4 [Last Accessed: 23 November 2006].

[15] Cepstral LLC. Cepstral Text-to-Speech. Avaliable at http://www.cepstral.com/ [Last Accessed: 23 November 2006].

[16] Phillip R. Cohen and S. L. Oviatt. The Role of Voice Input for Human-Machine Communication. *Proceedings of the National Academy of Sciences*, 92(22):9921–9927, 1995.

[17] Ross Corkrey and Lynne Parkinson. Interactive voice response: Review of studies 1989-2000. *Behavior Research Methods, Instruments, & Computers*, 34(3):342–353, 2002.

[18] Philip T Cox and Baoming Song. A Formal Model for Component-Based Software. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 304, Washington, DC, USA, 2001. IEEE Computer Society.

[19] Albert G. Crawford, Vanja Sikirica, Neil Goldfarb, Richard G. Popiel, Minalkumar Patel, Cheng Wang, Jeffrey B. Chu, and David B. Nash. Interactive Voice Response Reminder Effects on Preventive Service Utilization. *American Journal of Medical Quality*, 20:329–336, 2005.

[20] Avnish Dass, Vikas Gupta, and Charul Shukla. *VoiceXML 2.0 Developers Guide Building Professional Voice-Enabled Applications with JSP, ASP, & ColdFusion*. McGraw-Hill/Osborne, Berkeley, California, USA, 2002.

[21] David L. Thomson and John M. Hibel. The Business of Voice Hosting with VoiceXML. Technical report, Lucent Speech Solutions, 2000. Avaliable at http://members.tripod.com/ David_Thomson/papers/Host4.doc [Last Accessed: 23 November 2006].

[22] Tom DeMarco and Barry Boehm. The Agile Methods Fray. *Computer*, 35(6):90–92, 2002.

[23] Bryan Duggan. Revenue Opportunities in the Voice Enabled Web, 2002. Dublin Institute of Technology, School of Computing Report.

[24] Brian Eberman, Jerry Carter, Darren Meyer, and David Goddeau. Building voiceXML browsers with openVXI. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 713–717, New York, NY, USA, 2002. ACM Press.

[25] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol: HTTP/1.1. Internet Engineering Task Force: RFC 2616, June 1999. Avaliable at http://www.ietf.org/rfc/rfc2616.txt [Last Accessed: 23 November 2006].

[26] Brian Fitzgerald. The use of systems development methodologies in practice: a field study. *Information Systems Journal*, 7(3):201–212, 1997.

[27] G. Fowler. A case for make. *Softw. Pract. Exper.*, 20(S1):35–46, 1990.

[28] Michael Franz. Dynamic Linking of Software Components. *Computer*, 30(3):74–81, 1997.

[29] Scott Fricker, Mirta Galesic, Roger Tourangeau, and Ting Yan. An Experimental Comparison of Web and Telephone Surveys. *Public Opinion Quarterly*, 69(3):370–392, 2005.

[30] Evgeniy Gabrilovich and Lev Finkelstein. JNI-C++ integration made easy. *C/C++ Users J.*, 19(1):10–21, 2001.

[31] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139, New York, NY, USA, 2003. ACM Press.

[32] John C. Grundy, Warwick B. Mugridge, and John G. Hosking. Constructing component-based software engineering environments: issues and experiences. *Information & Software Technology*, 42(2):103–114, 2000.

[33] J. Hitchcock. Decorating Asterisk: Experiments in Service Creation for a Multi-Protocol Telephony Environment Using Open Source Tools. Masters thesis, Department of Computer Science, Rhodes University, Grahamstown, South Africa, March 2006.

[34] Hanna Hulkko and Pekka Abrahamsson. A multiple case study on the impact of pair programming on product quality. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 495–504, 2005.

[35] IBM. Research History Highlights. Avaliable at http://www.research.ibm.com/about/past_history.shtml [Last Accessed: 23 November 2006].

[36] A. Isard. SSML: A Markup Language for Speech Synthesis, 1995. PhD thesis, University of Edinburgh.

[37] JA Quiane Ruiz and JR Manjarrez Sanchez. Design of a VoiceXML Gateway. *In proceedings of the Fourth Mexican International Conference on Computer Science*, 00:49–53, 2003.

[38] Julie A. Wright Robert H. Friedman Jeffrey P. Migneault, Ramesh Farzanfar. How to write health dialog for a talking computer. *Journal of Biomedical Informatics*, 39:468–481, 2006.

[39] Candace Kamm. User Interfaces for voice applications. pages 422–442, 1994.

[40] Kent Beck and Mike Beedle and Arie van Bennekum and Alistair Cockburn and Ward Cunningham and Martin Fowler and James Grenning and Jim Highsmith and Andrew Hunt and Ron Jeffries and Jon Kern and Brian Marick and Robert C. Martin and Steve Mellor and Ken Schwaber and Jeff Sutherland and Dave Thomas. Manifesto for Agile Software Development. Avaliable at http://www.agilemanifesto.org [Last Accessed: 23 November 2006].

[41] Irina Kondratova. Performance and Usability of VoiceXML Application. In *Proceedings of the Eighth World Multi-Conference on Systemics, Cybernetics and Informatics*.

[42] Irina Kondratova. Voice and multimodal technology for the mobile worker. *Special Issue Mobile Computing in Construction*, 9:345–353, 2004.

[43] Sun Microsystems Laboratories, Carnegie Mellon University, and Mitsubishi Electric Research Laboratories. Sphinx-4 Application Programmer's Guide. Technical report. Avaliable at http://cmusphinx.sourceforge.net/sphinx4/doc/ProgrammersGuide.html [Last Accessed: 23 November 2006].

[44] Paul Lamere, Philip Kwok, Evandro B. Gouvea, Bhiksha Raj, Rita Singh, William Walker, and Peter Wolf. The cmu sphinx-4 speech recognition system. Technical report. Available at http://research.sun.com/sunlabsday/docs.2004/sphinx4.pdf [Last Accessed: 23 November 2006].

[45] Jim A. Larson. VoiceXML 2.0 and the W3C speech interface framework. In *IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 5–8, 2001.

[46] Kai-Fu Lee, Hsiao-Wuen Hon, and Mei-Yuh Hwang. Recent progress in the SPHINX Speech Recognition system. In *HLT '89: Proceedings of the workshop on Speech and Natural Language*, pages 125–130, Morristown, NJ, USA, 1989. Association for Computational Linguistics.

[47] Kai-Fu Lee, Hsiao-Wuen Hon, and Mei-Yuh Hwang. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):35–45, January 1990. see also IEEE Transactions on Signal Processing.

[48] Linda Levine. Reflections on software agility and agile methods: challenges, dilemmas and the way ahead. Technical report, 2005.

[49] Sheng Liang. *The Java Native Interface Programmer's Guide and Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[50] Linux Programmer's Manual. Lseek - reposition read/write file offset, 2001. Avaliable at http://ibm5.ma.utexas.edu/cgi-bin/man-cgi?lseek+2 [Last Accessed: 23 November 2006].

[51] Linux Programmer's Manual. Pipe - overview of pipes and FIFOs, 2005. Avaliable at http://ibm5.ma.utexas.edu/cgi-bin/man-cgi?pipe+7 [Last Accessed: 23 November 2006].

[52] LumenVox. LumenVox - Innovators of Speech Recognition Technology and Deployment Tools. Avaliable at http://www.lumenvox.com/ [Last Accessed: 23 November 2006].

[53] M. A. Awad. A Comparison between Agile and Traditional Software Development Methodologies. Honours Thesis, School of Computer Science and Software Engineering, The University of Western Australia, 2005.

[54] Nasser K. Manesh. Asterisk: A Non-Technical Overview, 2004. Avaliable at http://www.millenigence.com/articles/asterisk-non-technical-review.pdf [Last Accessed: 23 November 2006].

[55] Matthew Hood. Creating a Voice for Festival Speech Synthesis System. Honour's thesis, Department of Computer Science, Rhodes University, Grahamstown, South Africa, November 2004.

[56] Scott McGlashan, Daniel C. Burnett, Jerry Carter, Peter Danielsen, Jim Ferrans, Andrew Hunt, Bruce Lucas, Brad Porter, Ken Rehor, and Steph Tryphonas. Voice Extensible Markup Language (VoiceXML) 2.0, 2004. Avaliable at http://www.w3.org/TR/voicexml20 [Last Accessed: 23 November 2006].

[57] M. Douglas McIlroy. Mass-Produced Software Components. In J. M. Buxton, Peter Naur, and Brian Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98, 1976.

[58] Eleanor Singer Mick P. Couper and Roger Tourangeau. Does voice matter? An interactive voice response (IVR) experiment. *Journal of Official Statistics*, 20(3):1–20, 2004.

[59] Mark Miller. *Voicexml: 10 Projects to Voice Enable Your Web Site*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[60] Miro Distribution. Miro Distribution - POWERFUL | COMMUNICATION | SOLUTIONS. Avaliable at http://www.miro.co.za/ [Last Accessed: 23 November 2006].

[61] Ramkishore Modukuri and Richard Morris. Voice based web services - An assistive technology for visually impaired persons. *Technology and Disability*, 6:195–200, 2004.

[62] Narayanan Annamalai. An Extensible Transcoder for HTML to VoiceXML Conversion. Masters Thesis, The University of Texas at Dallas, 2002.

[63] Jerzy Nawrocki and Adam Wojciechowski. Experimental Evaluation of Pair Programming. In *Proceedings of European Software Control and Metrics*, London, UK, April 2001.

[64] Georg Niklfeld, Robert Finan, and Michael Pucher. Component-based multimodal dialog interfaces for mobile knowledge creation. In *Proceedings of the workshop on Human Language Technology and Knowledge Management*, pages 1–8, Morristown, NJ, USA, 2001. Association for Computational Linguistics.

[65] Hrvoje Niksic. Wget - The non-interactive network downloader, 2006. Linux Programmer's Manual. Avaliable at http://gentoo-wiki.com/MAN_wget_1 [Last Accessed: 23 November 2006].

[66] Jim Q. Ning. Component-Based Software Engineering (CBSE). In *SAST '97: Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*, page 34, Washington, DC, USA, 1997. IEEE Computer Society.

[67] Patricia A. Oberndorf. Facilitating Component-Based Software Engineering: COTS and Open Systems. In *SAST '97: Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*, page 143, Washington, DC, USA, 1997. IEEE Computer Society.

[68] Matt Oshry, RJ Auburn, Paolo Baggia, Michael Bodell, David Burke, Daniel C. Burnett, Emily Candell, Jerry Carter, Scott McGlashan, Alex Lee, Brad Porter, and Ken Rehor. Voice Extensible Markup Language (VoiceXML) 2.1, 2006. Avaliable at http://www.w3.org/TR/voicexml21 [Last Accessed: 23 November 2006].

[69] J. Penton and A. Terzoli. Asterisk: A Converged TDM and Packet-based Communications System. In *Proceedings of SATNAC 2003 - Next Generation Networks*, September 2003.

[70] Manuel A. Perez-Quinones and Jochen Rode. You've Got Mail! Calendar, Weather and More: Customizable Phone Access to Personal Information. Technical report, Computer Science, Virginia Tech, 2004. Avaliable at http://eprints.cs.vt.edu/archive/00000698/01/chi05phonePIM.pdf [Last Accessed: 23 November 2006].

[71] John D. Piette. Satisfaction With Automated Telephone Disease Management Calls and Its Relationship to Their Use. *The Diabetes Educator*, 26(3):1003–1010, 2000.

[72] John D. Piette, Morris Weinberger, Frederic B. Kraemer, and Stephen J. McPhee. Impact of Automated Calls With Nurse Follow-Up on Diabetes Treatment Outcomes in a Department of Veterans Affairs Health Care System. *Diabetes Care*, 24:202–208, 2001.

[73] Padmanabhan Pillai and Kang G. Shin. Extending UNIX File Abstraction for General-Purpose Networking, January 2004. Avaliable at http://www.intel-research.net/Publications/Pittsburgh/101220041324_277.pdf [Last Accessed: 23 November 2006].

[74] publicVoiceXML. Official Home of publicVoiceXML. Avaliable at http://publicvoicexml.sourceforge.net [Last Accessed: 23 November 2006].

[75] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. pages 267–296, 1990.

[76] Mosur K. Ravishankar. Sphinx-3 s3.X Decoder (X=6). Technical report. Avaliable at http://cmusphinx.sourceforge.net/sphinx3/s3_description.html [Last Accessed: 23 November 2006].

[77] Sami Lemmetty. Review of Speech Synthesis Technology. Masters Thesis, Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology, 1999.

[78] Chris Schmandt. Phoneshell: the telephone as computer terminal. In *MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia*, pages 373–382, New York, NY, USA, 1993. ACM Press.

[79] Udo Schuermann. The "JNI HOW-TO", 2001. Avaliable at http://ringlord.com/publications/jni-howto/ [Last Accessed: 23 November 2006].

[80] Zhiyan Shao, Robert Capra, and Manuel A. Pérez-Quiñones. Annotations for HTML to VoiceXML Transcoding: Producing Voice WebPages with Usability in Mind. *CoRR*, cs.HC/0211037, 2002.

[81] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[82] Mark Spencer, Mack Allison, Christopher Rhodes, and The Asterisk Documentation Team. The Asterisk Handbook Version 2. Technical report. Avaliable at http://www.digium.com/handbook-draft.pdf [Last Accessed: 23 November 2006].

[83] STAR. The Speech Technology And Research (STAR) Group. Avaliable at http://www.star.za.net [Last Accessed: 23 November 2006].

[84] F. W. M. Stentiford and P. A. Popay. The Design and Evaluation of Dialogues for Interactive Voice Response Services. *BT Technology Journal*, 17(1):142–148, 1999.

[85] Bernhard Suhm, Dam McCarthy, and Pat Peterson. Maximizing Benefit while Minimizing Risk in Speech-Enabling Customer Care. Technical report, BBN Technologies. Avaliable at http://www.bbn.com/docs/whitepapers/max-benefits.pdf [Last Accessed: 23 November 2006].

[86] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[87] Roger Tourangeau, Darby Miller Steiger, and David Wilson. Self-administered questions by telephone: Evaluating interactive voice response. *Public Opinion Quarterly*, 66(2):265–278, 2002.

[88] Kenneth J. Turner. Analysing interactive voice services. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 45(5):665–685, 2004.

[89] voip info.org. Asterisk Festival Installation, 2006. Avaliable at http://www.voip-info.org/wiki/view/Asterisk+festival+installation [Last Accessed: 23 November 2006].

[90] W3C. Introduction and Overview of W3C Speech Interface Framework, 2000. Avaliable at http://www.w3.org/TR/voice-intro [Last Accessed: 23 November 2006].

[91] W3C. Speech Synthesis Markup Language (SSML) Version 1.0, 2004. Avaliable at http://www.w3.org/TR/speech-synthesis [Last Accessed: 23 November 2006].

[92] Merrill Warkentin. The Next Big Thing in eCommerce. *Decision Line*, 32(1):7–10, 2001.

[93] M.F. Weegels. Users' Conceptions of Voice-Operated Information Services. *International Journal of Speech Technology*, 3(2):75–82, 2000.

[94] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the Case for Pair Programming. *IEEE Softw.*, 17(4):19–25, 2000.

[95] Scott Wilson. WAVE PCM soundfile format, 2003. Avaliable at http://ccrma.stanford.edu/courses/422/projects/WaveFormat/ [Last Accessed: 23 November 2006].

[96] Mary Zajicek, Richard Wales, and Andrew Lee. Towards voiceXML dialogue design for older adults. In: Palanque, P., Johnson, P. and O'Neill, E., Editors, 2003. Design for SocietyProceedings of HCI 2003.

Appendix A

VoiceXML 2.0 elements

Element	Purpose
`<assign>`	Assign a variable a value
`<audio>`	Play an audio clip within a prompt
`<block>`	A container of (non-interactive) executable code
`<catch>`	Catch an event
`<choice>`	Define a menu item
`<clear>`	Clear one or more form item variables
`<disconnect>`	Disconnect a session
`<else>`	Used in <if> elements
`<elseif>`	Used in <if> elements
`<enumerate>`	Shorthand for enumerating the choices in a menu
`<error>`	Catch an error event
`<exit>`	Exit a session
`<field>`	Declares an input field in a form
`<filled>`	An action executed when fields are filled
`<form>`	A dialog for presenting information and collecting data
`<goto>`	Go to another dialog in the same or different document
`<grammar>`	Specify a speech recognition or DTMF grammar
`<help>`	Catch a help event
`<if>`	Simple conditional logic
`<initial>`	Declares initial logic upon entry into a (mixed initiative) form
`<link>`	Specify a transition common to all dialogs in the link's scope
`<log>`	Generate a debug message

Element	Purpose
`<menu>`	A dialog for choosing amongst alternative destinations
`<meta>`	Define a metadata item as a name/value pair
`<metadata>`	Define metadata information using a metadata schema
`<noinput>`	Catch a noinput event
`<nomatch>`	Catch a nomatch event
`<object>`	Interact with a custom extension
`<option>`	Specify an option in a <field>
`<param>`	Parameter in <object> or <subdialog>
`<prompt>`	Queue speech synthesis and audio output to the user
`<property>`	Control implementation platform settings
`<record>`	Record an audio sample
`<reprompt>`	Play a field prompt when a field is re-visited after an event
`<return>`	Return from a subdialog.
`<script>`	Specify a block of ECMAScript client-side scripting logic
`<subdialog>`	Invoke another dialog as a subdialog of the current one
`<submit>`	Submit values to a document server
`<throw>`	Throw an event
`<transfer>`	Transfer the caller to another destination
`<value>`	Insert the value of an expression in a prompt
`<var>`	Declare a variable
`<vxml>`	Top-level element in each VoiceXML document

Table A.1: VoiceXML elements in the VoiceXML 2.0 specification

Appendix B

VoiceXML 2.1 elements

Element	Purpose	Enhanced/New
<data>	Fetches arbitrary XML data from a document server	New
<disconnect>	Disconnects a session	Enhanced
<grammar>	References a speech recognition or DTMF grammar	Enhanced
<foreach>	Iterates through an ECMAScript array	New
<mark>	Declares a bookmark in a sequence of prompts	Enhanced
<property>	Controls platform settings	Enhanced
<script>	References a document containing client-side ECMAScript	Enhanced
<transfer>	Transfers the user to another destination	Enhanced

Table B.1: Elements added or enhanced in the VoiceXML 2.1 specification

Appendix C

Sample zaptel configuration files

Example zaptel configuration files. This sample configuration represents a single PRI interface connected either to the PSTN.

The `zaptel.conf` file. This file is not Asterisk specific and is used to configure the interface itself.

```
# use US indication tones
loadzone=us
defaultzone=us
# configure the span - a span usually corresponds
# to an interface format is
# span=(spannum),(timing),(LBO),(framing),(coding)[1]
# ccs and ami indicare an E1 PRI as opposed to a T1
span=1,1,0,ccs,ami
# confugure the bearer channels - these channels carry the voice
bchan=1-15
bchan=17-31
# configure the data channel
dchan=16
```

The `zapta.conf` file, which is Asterisk specific and is used to configure Asterisk's zap channel. This example shows only a small subset of the available configuration options.

[1]http://www.voip-info.org/wiki/index.php?page=Zaptel.conf+span+syntax

```
switchtype=euroisdn
; singalling type - specifies that we are on
; the customer premises side of a connection,
; essential if we are connecting to the PSTN
signalling=pri_cpe
pridialplan=unknown
echocancel=yes
; do not answer the channel immediately
immediate=no
; available channels can be divided into
; groups - different groups can be used
; for different purposes, there is
; one group in this example
group = 1
; this group enters the dial plan at the pstn context
context=pstn
; assign some channels to the group'
channel => 1-15
channel => 17-31
```

Appendix D

Application module skeleton

The `voicexml` application skeleton. Of interest here is the naming of the variables and functions.

```
// descriptions for the Asterisk CLI help system
static char *tdesc = "voicexml";
static char *app_voicexml = "voicexml";
static char *voicexml_synopsis = "VoiceXML Application";
static char *voicexml_descrip =
  "voicexml(url[|sphinxhost|[port]])\n"
  "  url:         VoiceXML url\n"
  "  sphinxhost: SphinxFarm host - default is 127.0.0.1\n"
  "  port:        Port to connect to the SphinxFarm on -
  "               default is 1315\n\n"
  "The voicexml app renders a
  "voicexml document with Festival \n"
  "for Voice Synthesis and Sphinx4 for voice recognition.";
STANDARD_LOCAL_USER;
LOCAL_USER_DECL;
static int voicexml_exec
   (struct ast_channel *chan, void *data) {
  // execution begins here
}
// Asterisk housekeeping
int unload_module(void) {
  int res;
  STANDARD_HANGUP_LOCALUSERS;
  res = ast_unregister_application(app_voicexml);
  return res;
}
int load_module(void) {
  int res;
  res = ast_register_application
      (app_voicexml, voicexml_exec,
       voicexml_synopsis, voicexml_descrip);
  return res;
}
char *description(void) {
  return tdesc;
}
int usecount(void) {
  int res;
  STANDARD_USECOUNT(res);
  return res;
}
char *key() {
  return ASTERISK_GPL_KEY;
}
```

Appendix E

A Sphinx 4 application

Exerpts of the Sphinx process used to recognize speech.

```
/* configure the components */
URL url = new File (conf).toURI (). toURL ();
ConfigurationManager cm = new ConfigurationManager(url);
recognizer = (Recognizer) cm.lookup("recognizer");
reader = (StreamDataSource) cm.lookup ("streamDataSource");
...

/* allocate the resource necessary for the recognizer */
recognizer.allocate();
...

/* grab the results */
while (true)
{
  Result result = null;
  String resultString = null;
  result = recognizer.recognize();
  if (result != null)
  {
    resultString = result.getBestResultNoFiller();
    System.out.println
    ("[" + id + "] --> " + resultString);
```

```
        setResult (resultString);
      }
      else
      {
        System.out.println
       ("[" + id + "] --> Hang up - pausing thread");
        try {
        conn.close ();
        synchronized (this)
        {
          this.connected = false;
          this.wait ();
        }
    }
    catch (InterruptedException e)
    ...
```

Appendix F

Sphinx configuration

Excerpts from the Sphinx configuration file

```
<!-- the dictionary configuration, set to WSJ 8Khz -->
<component name="dictionary"
     type="edu.cmu.sphinx.linguist.dictionary.FastDictionary">
  <property name="dictionaryPath"
    value="file:
/home/adam/sphinxfarm/
WSJ_8gau_13dCep_8kHz_31mel_200Hz_3500Hz/dict/cmudict.0.6d"/>
  <property name="fillerPath"
    value="file:
/home/adam/sphinxfarm/
WSJ_8gau_13dCep_8kHz_31mel_200Hz_3500Hz/dict/fillerdict"/>
  <property name="addSilEndingPronunciation" value="false"/>
  <property name="allowMissingWords" value="false"/>
  <property name="unitManager" value="unitManager"/>
</component>

<!-- the acoustic model configuration, set to WSJ 8Khz -->
<component name="wsj8k"
  type=
"edu.cmu.sphinx.model.acoustic.
WSJ_8gau_13dCep_8kHz_31mel_200Hz_3500Hz.Model">
    <property name="loader" value="wsjLoader"/>
    <property name="unitManager" value="unitManager"/>
  </component>
  <component name="wsjLoader"
    type=
"edu.cmu.sphinx.model.acoustic.
WSJ_8gau_13dCep_8kHz_31mel_200Hz_3500Hz.ModelLoader">
  <property name="logMath" value="logMath"/>
  <property name="unitManager" value="unitManager"/>
</component>

<!-- the front-end pipeline -->
<component name="streamDataSource"
  type="edu.cmu.sphinx.frontend.util.StreamDataSource">
  <property name="sampleRate" value="8000"/>
  <property name="bitsPerSample" value="16"/>
  <property name="bigEndianData" value="true"/>
  <property name="signedData" value="true"/>
  <property name="bytesPerRead" value="160"/>
</component>
```

RHODES UNIVERSITY LIBRARY