Creating a low cost VoiceXML Gateway to replace IVR systems for rapid deployment of voice applications.

Adam King, Alfredo Terzoli and Peter Clayton
Department of Computer Science
Tel: 046 6038291 Fax: 046 6361915
Rhodes University Grahamstown, 6140
g00k4406@campus.ru.ac.za A.Terzoli@ru.ac.za P.Clayton@ru.ac.za

***Abstract*— This paper reports on the creation of a low cost VoiceXML gateway which can be used to replace traditional Interactive Voice Response (IVR) platforms.**
The gateway is created by integrating a VoiceXML interpreter, OpenVXI and a PBX, Asterisk, producing a Linux based, open source, system which is both a PBX and a VoiceXML browser.
Reasons for choosing the components for the gateway and then the integration of these components are discussed.
VoiceXML applications can be used to replace IVR systems, which are then rendered by the gateway.[1]

***Index Terms*— Asterisk, VoiceXML, IVR, VoIP**

I. INTRODUCTION

Many organisations make use of IVRs to provide voice interfaces to their users. These systems can be used to reduce user interaction with human call center agents, used to interact with voice services (such as voice mail systems) or used to bring new and inventive voice services to customers.

Currently there is no uniform way of deploying IVR platforms and implementing IVR systems. IVR applications are tightly coupled to their underlying platforms. This can lead to problems such as different IVR systems not being able to interact with each other and IVR applications not being portable. Installing and developing IVR systems can also be expensive.

This paper proposes that VoiceXML be used as a uniform means of creating IVRs. VoiceXML is a mature XML based standard which can be used to describe voice applications. Like HTML, VoiceXML pages have to be rendered by browsers.

The use of VoiceXML would allow the decoupling of voice applications from their underlying platforms, shielding developers from the low level aspects of the specific platform, and increasing portability of voice applications. However VoiceXML gateways are costly, which may dissuade many organisations. There is therefore a need for low cost VoiceXML gateways.

This paper will firstly investigate the relationship between VoiceXML and IVRs, detailing the advantages VoiceXML can bring to the process of creating IVRs. It will then discuss the components need to create a VoiceXML gateway, giving a brief description of the components selected for this particular gateway. Once the components have been discussed the paper will look at the task of integrating the components to create a gateway. Finally the complete architecture will be summarised, detailing a typical call.

[1]This work was undertaken in the Distributed Multimedia Centre of Excellence at Rhodes University, with financial support from Telkom SA, Business Connexion, Comverse, Verso Technologies, Tellabs, StorTech THRIP, and the National Research Foundation.

II. VOICEXML AND IVRS

VoiceXML provides sufficient functionality to completely replace existing IVR systems, and has added advantages in the area of voice application development.

VoiceXML applications residing on the corporate web server have the same access to corporate databases as the web server itself, and VoiceXML developers can take advantage of existing web application development tools [1], [2]. Developers are also able to make use of familiar, and mature, web technologies, for example extending standard web security methods to create secure voice applications [2]. VoiceXML also abstracts away from the low-level, platform specific, details. Due to this abstraction, VoiceXML applications are portable across web servers and gateways [1]. These factors help drive down development costs while increasing the longevity and scope of the VoiceXML applications. Companies can change platforms without affecting existing development.

In contrast there seems to be no standard means of interaction between IVR systems from different vendors, and enterprises may deploy proprietary systems which are not compatible with their existing web technologies. VoiceXML can be used as a means of standardising the creation of voice applications [3].

While creating VoiceXML applications may be an attractive option, they have to be rendered by a VoiceXML browser. Browsers can either be purchased, or a hosting solution can be employed.

According to Ruiz *et al* VoiceXML gateways are very expensive, due to the specialised hardware required [4]. Eisenzopf claims that the cost of VoiceXML gateways begins at $10 000, while Thompson *et al* put this figure significantly higher, at $100 000 [3], [5]. This figure, in reality, is an estimation of the total startup costs, including costs such

as line rental, purchasing and installing the equipment and development costs.

Another option is to have the VoiceXML applications hosted by Voice hosting Service Providers (VSPs), as described by [5]. In this case there are no startup costs when using a hosted solution, only the cost of hiring the lines from the VSP. Maintenance of the gateway now becomes the responsibility of the VSP. This means that an enterprise using the hosted solution need only maintain their VoiceXML pages, and not an IVR platform as well, lowering maintenance costs as well as sharing ownership costs.

While hosting seems to be a viable option for smaller enterprises, there are a number disadvantages, especially in the South African context. Firstly, the cost of the hosting is still high, because the gateways employed for hosting are expensive. Secondly, VSPs are predominantly found abroad, limiting access to the application for many South Africans. In addition to this hosted solutions may not allow easy customisation, such as synthesising speech in different accents and different languages or recognising speech in different languages.

These high startup costs and unattractive hosting options will hinder the adoption of VoiceXML as a standard means of creating IVRs, particularly amongst the many small to medium enterprises which can be found in the South African business landscape. There is therefore a need for a low cost VoiceXML gateway. Such a gateway will bring the benefits of using VoiceXML based voice applications to organisations which would otherwise not be able to to afford it, increasing the uptake of VoiceXML.

III. Gateway Components

Creating a VoiceXML gateway from scratch is possible but impractical. On the other hand following a component-orientated approach can significantly reduce the scope of the project, making the construction of a gateway feasible [6].

Gateway components have to be identified and then sourced from the open source community. Using open source systems allows systems to be integrated with more flexibility, and of course ensures lower costs. The VoiceXML 2.0 specification [7] describes the components needed to construct a fully compliant VoiceXML 2.0 platform. These are:

a) Document acquisition: The gateway must have a means of fetching documents from a web server, for the interpreter to parse. This fetching mechanism must be able to fetch documents using HTTP.

b) Audio output: VoiceXML 2.0 allows for both speech synthesis and the play back of audio files. Any implementation of a gateway must be able to synthesise text, and fetch and play audio files. The gateway discussed in this paper supports MP3, GSM and WAV audio formats, amongst others.

c) Audio input: A fully compliant VoiceXML 2.0 gateway must be able to simultaneously recognise both character input, via DTMF, and spoken input. The gateway implemented in this paper is only intended to be used as a replacement for IVRs, and so speech recognition is not required. The gateway must also be able to record user input.

d) Transfer: The gateway must be able to transfer calls. Also implied here is the need for a telephony platform, to handle calls to the gateway.

It follows that to implement a gateway a developer will need the following components:

- A VoiceXML interpreter, which can fetch and interpret VoiceXML 2.0 documents,
- A Text To Speech (TTS) system, to provide synthesised speech, and a means of playing back audio files,
- An Automatic Speech Recognition (ASR) system, for speech recognition, a DTMF recogniser, and a means of recording audio input, and finally
- A telephony platform to handle calls, including the transfer of existing calls.

A. Components

This gateway uses OpenVXI as the interpreter, Festival provides synthesised text and the telephony platform is Asterisk. Asterisk will also be responsible for playing audio files (section IV-D), DTMF recognition (sections IV-D and IV-E) and recording audio.

1) OpenVXI: OpenVXI is an open framework where developers can add their own components to create a VoiceXML browser [8], [9]. To this purpose OpenVXI exposes three APIs to the developer, the `tel` API, the `prompt` API and the `rec` API. Creating the gateway involves fleshing out the APIs, to allow OpenVXI to interact with the third party components. The reader should note that an interface represents the realisation of an API.

The `tel` API is responsible for setting up and tearing down calls, monitoring the calls to detect hang ups and providing the functionality to transfer calls. The `prompt` API represents a one way interaction between the gateway and the user. It is responsible for playing back audio files and synthesised text. The `prompt` API also handles the fetching and caching of pre-recorded audio. The `rec` API is responsible for evaluating user input against any active grammars, and recording utterances. (It is not responsible for converting DTMF and voice signals into symbols; this task falls to the third party components.)

The interpreter is fully compliant with the VoiceXML 2.0 specification, meaning that it can parse VoiceXML 2.0 documents, its functionality, however, is dependent on the amount of fleshing out that has been done to the APIs. OpenVXI runs on a Linux platform and is written in C and C++. It uses SpiderMonkey as its JavaScript engine and Xerces as the XML parser, which are open source projects available on the Linux platform. OpenVXI has been chosen as the VoiceXML interpreter.

publicVoiceXML was also considered as an open source VoiceXML browser. It claims to provide built-in DTMF and telephony support and makes provisions for adding TTS systems. Initially this project seemed ideal. Unfortunately it does not seem very active, and was therefore discarded.

2) Festival: The Festival speech synthesis system, developed at CMU, is a Linux based open source framework, written in C++, for creating TTS systems [10]. Festival can

be used to create different voices in different languages. [11] investigates creating new voices for the Festival system, concluding that creating new Festival voices is a relatively simple procedure. To this effect the Speech Technology and Research (STAR) Group[2] are developing new Festival languages in some of the indigenous South African languages. This is a clear advantage in the South African context, allowing voice applications to 'speak' in more than one language, bringing these applications to a larger number of South Africans.

The quality of the voices depends on the the training data used when creating the voice [10]. However the standard, free, voices which come with Festival produce clear results, and the majority of the output is understood, although the voices do sound robotic.

3) Asterisk: Asterisk is an open source telecommunications environment [12]. It can be used as a TDM and/or packet switched voice and video PBX and an IVR platform. As a Voice over IP gateway Asterisk supports IAX, SIP, MGCP and H.323, at the same time it can be used as an interface to the PSTN. Asterisk's aim is "to interface any piece of telephony hardware or software with any telephony application, seamlessly and consistently" [12]. It runs on a Linux platform and is written in C.

One goal of this project is to integrate OpenVXI and Asterisk in a way that does not change Asterisk's source code, so that the gateway can be added to any existing instances of Asterisk simply and seamlessly.

One major benefit of using Asterisk as the telephony interface is the fact that it supports so many telecommunications technologies. This means that the VoiceXML gateway will be exposed to all the technologies supported by Asterisk, allowing a lot of flexibility with regards to interacting with the voice applications.

Once all the components have been selected they have to be integrated to create the gateway. The integration of these components is discussed in the following section.

IV. Creating the Gateway

This section will look at the actual construction of the gateway. The construction involves the integration of the components described in section III-A into a working system.

Firstly, we need a means of integrating OpenVXI into Asterisk. typical call is initiated by the client, and is routed to Asterisk. Asterisk then decides what to do with the call using dialplan logic. Should the call need to be directed to a VoiceXML service Asterisk routes the call to the interpreter.

A. Dialplan Applications

Once Asterisk knows that the call has to be passed to the interpreter it needs a means of interacting with the gateway. This must be done as seamlessly as possible. There are two options available here. Using an AGI script to run the interpreter, or creating a dialplan application.

An Asterisk Gateway Application (AGI) "executes an Asterisk Gateway Interface compliant program on a channel. AGI allows Asterisk to launch external programs written in any language to control a telephony channel, play audio, read DTMF digits, etc. by communicating with the AGI protocol on stdin and stdout" [13]. While this may seem to be the easiest means of integrating the interpreter into Asterisk, it is not the most elegant.

For this reason it was decided to create a dialplan application. Dialplan applications can be called directly from the dialplan and, like AGI applications, are available to any channel. This means that the VoiceXML service will be available to any channel Asterisk can support. Dialplan applications are relatively simple to create in Asterisk. A C application is written and put in the `asterisk/apps` directory. This file must follow a basic template. It must be named `app_<app name>.c`, in this case `app_voicexml.c`. A similar naming convention is also used when naming functions and variables which are then called by Asterisk at runtime. For instance, execution in an application will begin in the `<app name>_exec` function. Finally, a target must be created for the application in the `asterisk/apps` makefile to ensure that the module is created. This will require the Asterisk source and a re-build, which creates the shared object and moves it into `/usr/libs/asterisk/modules`. These modules are then loaded by the Asterisk module loader at start up. Creating these applications in this manner provides a means to add functionality to Asterisk without changing the source code, an important constraint to this project. Acquiring the source and re-building Asterisk is relatively simple, and so creating new dialplan applications is a simple means of generically extending functionality in many different Asterisk implementations.

B. Adding the interpreter

Once we have created a basic application we can begin to add VoiceXML functionality. This means initialising and activating the interpreter. OpenVXI is linked dynamically at build time to the `voicexml` application. Control is then passed to OpenVXI, which can begin interpretation of the VoiceXML application. The interpreter can now start parsing a page. OpenVXI needs to be able to perform semantic actions on the channel, and so needs to be passed a pointer to the particular channel. Therefore another dynamic link has to be created, linking Asterisk to OpenVXI and allowing a pointer to the channel to be passed to the interpreter.

Once OpenVXI and Asterisk are linked, the two processes can interact with each other. Before OpenVXI is initialised, Asterisk has set up the call and, once OpenVXI has exited, Asterisk is responsible for tearing down the call. This functionality does not have to be implemented again in the `tel` API. When control is passed to the interpreter, the channel, representing a correctly set up call, is passed as well. Once the interpreter is done with the call (a hang-up has been received or the VoiceXML application has exited), control is passed back to the dialplan application, at which point Asterisk tears down the call.

Once Asterisk and the OpenVXI are able to interact with each other, the APIs have to be given the functionality to

[2]http://www.star.za.net/

render the VoiceXML. Specifically, the APIs have to be able to perform actions, described by the VoiceXML, on the channel. A pointer to the channel is passed from the dialplan application to the OpenVXI platform, and from there to the prompt API, along with pointers to the `tel` and `rec` APIs, resulting in a somewhat prompt-centric architecture.

Once OpenVXI has control of the channel, it begins interpreting the page, constructs a prompt, and sends it to the `prompt` interface to be played. During the playing of the prompts Asterisk is responsible for collecting DTMF tones, translating these tones into characters, and passing the result to the `prompt` interface. The `prompt` interface is then responsible for setting the value of the received input in the recognition interface. The `rec` interface then evaluates this input against the current grammar when the prompt is done playing. (Depending on the prompt's barge-in property the playing of the prompt can be stopped as soon as input is received, or only once it is done.) Input received during a prompt which requires no input is ignored by the `rec` interface, but is still set by the `prompt` interface.

In addition to setting received input, the `prompt` interface also has to alert the `tel` interface when it receives a hang-up signal from the channel, so that the line status can be updated. The interpreter frequently checks the status of the line, and once it detects that the line has been hung up it will stop queuing prompts, release resources, and exit. This returns control to the `voicexml` dialplan application, which will also exit, and Asterisk can commence tearing down the channel.

C. Queuing Prompts

OpenVXI describes prompts using the Speech Synthesis Markup Language (SSML) specification [14]. SSML is used as a standard method to define input for text synthesisers, ensuring that the input adheres to a pre-defined set of rules, thus improving performance. It is also a way to provide additional linguistic information to the synthesisers [15].

The interpreter creates SSML documents from the VoiceXML page and passes them to the `prompt` interface, which is responsible for rendering the prompts. A simple SSML parser has been written to extract the key/value pairs which are passed as arguments in the `<prompt>` tag. The SSML represents both types of prompts, synthesised speech and pre-recorded audio. Besides extracting the parameters the parser has to differentiate between these two types and act accordingly. A `prompt` data structure has been created which is used to represent the results of the parsing. A queue of these `prompt` structures has been implemented.

Typically, the SSML document representing the initial prompt is generated, and the prompt interface will attempt to play this immediately. The interpreter will only attempt to queue a prompt once it is sure, barring a hangup, that prompt will be played. If the next prompt to be queued is dependent on user input, the interpreter will only queue the prompt after the user input has been received. Additionally, if a prompt is to be played twice it will be queued twice.

This behavior simplifies the matter of choosing a data structure to represent the queue. Prompts are only queued if they are definitely going to be played, so the prompt at the head of the queue can always be played. If a prompt is to be played twice it will be queued twice, so the prompt at the head of the queue, once played, can be discarded.

D. Playing Prompts

The `prompt` API defines `VXIpromptPlay`, which is responsible for playing the prompts. As said, there are two types of prompts which can be played, pre-recorded audio and synthesised text. Playback of pre-recorded audio is made relatively simple by using existing Asterisk functionality, which can play audio on a channel in a number of formats. This is provided by the `ast_app_getdata` function, which plays a file and returns any DTMF input received during playback.

Synthesising text requires the integration of Festival into the system. Festival provides a number of APIs to allow other applications to interact with it. These are, Scheme API, Shell API, C/C++ API, Java and JSAPI, C only API and the Client/Server API [10]. The Client/Server API has been used in this project.

There are a number of reasons for this. Firstly, this API provides sufficient functionality. While the Scheme API may provide more fine grained control, such control is unnecessary. Secondly, once the server is running there are no time penalties associated with initialisation, and so prompts can be generated with minimal delay. Some of the APIs, in particular the Shell API and Scheme API, will incur time penalties while initialising. Thirdly, the Client/Server API provides the option of load balancing, and different servers can be configured differently. Configuration files are used when setting up Festival servers. This means that different servers can synthesise text using different voices and different languages. Finally, excellent examples of using the Client/Server API can be found from both the Festival documentation [10] and from Asterisk itself.

Once the Festival server is configured and running, a client application, in this case the `prompt` API, can connect to the server, send a string, and the server will send back the waveform representing the string. This waveform is received as an array of characters. Playing this waveform on the channel is done by piping the waveform to the channel. The channel reads from the pipe and the `prompt` interface writes the waveform to the pipe. The Asterisk function `send_waveform_to_fd` is used to write to the pipe.

While the prompt is being played back, each frame received from the channel is monitored. There are ten distinct frame types, including DTMF frames, voice frames, video frames and `NULL` frames. `NULL` frames are generated when the line is hung up. When the `prompt` interface detects a `NULL` frame it can stop playing the prompt and notify the telephony interface. When we receive a DTMF signal we can treat it as input, and notify the recognition interface.

E. DTMF input

Once DTMF input is received, it has to be sent to the `rec` interface, using a `VXIMap`. The `VXIMap` data structure is a container for key/value pairs which are of type `VXIchar *`. The `VXIchar` type represents the locale char type, and is

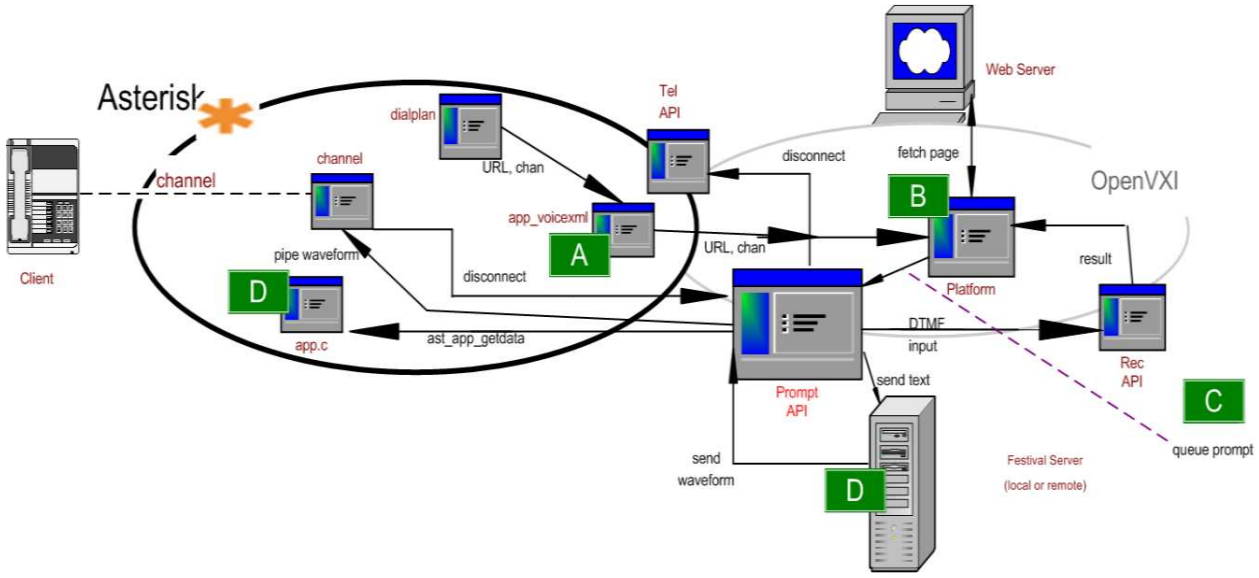

Fig. 1. The complete system, including interactions
A is described in section IV-A
B is described in section IV-B
C is described in section IV-C
D is described by sections IV-D and IV-E

used for portability. The `VXIMaps` are used by OpenVXI to pass data across interface boundaries, in this case between the `prompt` and `rec` interfaces. The `VXIMap` must specify the type of input, speech or DTMF, as well as the value of the input. The VoiceXML specification states that input can either be speech only, or DTMF only, or a combination of the two. The `rec` interface is then responsible for matching the input against the active grammar, or grammars, and acting accordingly. Traditional IVRs only accept DTMF user input, so in this case speech input is ignored.

In keeping with the 'prompt centric' architecture all inputs, including hangups, are detected during the playback of a prompt. The input collected by the `prompt` interface is sent to the `tel` and `rec` interfaces when appropriate.

V. Complete Architecture

Figure 1 shows the complete architecture, illustrating the interaction between the components once they have been integrated. It describes the process of handling a typical call which requests a VoiceXML service, rendering the VoiceXML page(s) while interacting with the user, and finally ending the call. This process, described in detail in section IV, is described here.

Once the call is set up the dialplan application, `app_voicexml`, is executed, and is passed a pointer to the channel and the URL of the VoiceXML page.

`app_voicexml` is then responsible for initialising the OpenVXI platform. Once that is done it passes control, as well as pointers to the channel and URL, to OpenVXI. OpenVXI has a built in internet interface, which is responsible for fetching the VoiceXML document(s).

When the document(s) has been returned from the web server the interpreter begins parsing and sending prompts to the `prompt` interface to be queued.

The prompt interface is responsible for queuing and then playing the prompts. Before the prompts are queued they have to be parsed by the SSML parser. Prompts which need to be synthesised are sent to the Festival server, and the resulting waveform is piped to the channel. Pre-recorded audio prompts are played using `ast_app_getdata`, which is provided by the `app` module.

While prompts are being played the `prompt` interface is responsible for collecting input, which is sent to the `rec` interface, and detecting and then alerting the `tel` interface of hangups. The `rec` interface attempts to match the received input against any active grammars, and sets the result for the interpreter to act upon appropriately.

VI. Conclusion

Using VoiceXML to create IVRs is a better solution than creating dedicated IVR platforms and applications. VoiceXML 2.0 provides more than enough functionality to allow VoiceXML applications to completely replace traditional IVR systems, and should the gateway be completely compliant, VoiceXML has the means to extend and improve IVR systems as we know them today.

While using VoiceXML as a voice application development language is attractive, the costs and inconveniences associated with VoiceXML gateways can prove a hindrance. However it is possible to create a low cost VoiceXML gateway which has enough functionality to completely replace traditional IVR systems. This architecture will be used to replace a mark reading IVR at Rhodes University.

This paper shows that open source components can be used to build a VoiceXML gateway which which has sufficient functionality to replace traditional IVR systems. Integrating

OpenVXI and Asterisk produces a hybrid VoiceXML gateway and PBX. The reader should note that as the VoiceXML 2.0 specification requires speech recognition the gateway being discussed is not compliant. However, speech recognition will be added to the system, creating a more natural means of interacting with the applications, and so improving usability. This functionality will be provided by integrating the Sphinx4 speech recognition system to the gateway discussed in this paper.

The main components of this gateway, OpenVXI, Asterisk and Festival are all mature and active projects, ensuring the gateway's longevity.

REFERENCES

[1] C. Bajorek, "VoiceXML - Taking IVR to the Next Level," 2000. [Online]. Available: http://www.cconvergence.com/article/CTM20000927S0003

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

[3] Jonathan Eisenzopf, "Is VoiceXML right for your customer service strategy?" 2002. [Online]. Available: http://www.ddj.com/dept/architect/184413221

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

[5] David L. Thomson and John M. Hibel, "The Business of Voice Hosting with VoiceXML," Lucent Speech Solutions, Tech. Rep., 2000, Avaliable at http://members.tripod.com/ David_Thomson/papers/Host4.doc.

[6] O. Nierstrasz, S. Gibbs, and D. Tsichritzis, "Component-Oriented Software Development," *"Communications of the ACM*, vol. 35, no. 9, pp. 160–165, 1992. [Online]. Available: citeseer.ist.psu.edu/nierstrasz92componentoriented.html

[7] W3C, "Voice Extensible Markup Language (VoiceXML) Version 2.0," 2004. [Online]. Available: http://www.w3.org/TR/voicexml20

[8] C.-H. Hsu, M.-R. Hsu, C.-Y. Yang, and S.-C. Chang, "On the construction of a VoiceXML Voice Browser," 2002, In ISCSLP 2002, paper 25.

[9] Jerry Carter, Brian Eberman, David Goddeau, Darren Meyer, "Building VoiceXML Browsers with OpenVXI," *In proceedings of the International WWW Conference*, May 2002. [Online]. Available: http://www2002.org/CDROM/refereed/260

[10] A. W. Black, P. A. Taylor, and R. Caley, "The Festival Speech Synthesis System: System documentation," Human Communciation Research Centre, University of Edinburgh, University of Edinburgh, UK, Tech. Rep., 2002, avaliable at http://festvox.org/docs/manual-1.4.3.

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

[12] M. Spencer, M. Allison, and C. Rhodes, "The Asterisk Handbook. Vol. 2." 2003.

[13] Various Authors, "Asterisk Help System for Asterisk version 1.0.6," 2005.

[14] W3C, "Speech Synthesis Markup Language (SSML) Version 1.0," 2004. [Online]. Available: http://www.w3.org/TR/speech-synthesis

[15] A. Isard, "SSML: A Markup Language for Speech Synthesis," 1995. [Online]. Available: citeseer.ist.psu.edu/isard95ssml.html

A King has completed an Honours degree in Computer Science at Rhodes University. He is currently reading towards a Masters degree at the same institute and is a Telkom bursar

P Clayton is the Director of Research, Computer Science Department, Rhodes University. His research interests include distributed multimedia, distributed and parallel processing and distance learning technology.

A Terzoli is the Project Director for the Centre of Excellence (CoE) in Distributed Multimedia at Rhodes University and the Project Director for the CoE in E-Commerce and E-Learning at the University of Fort Hare. His research interests include real-time multimedia over packet networks and ICTs in rural development.