Vox et praeterea nihil

Kai Tverrå

June 3, 2004

Department of Informatics
University of Bergen
PB. 7800
N-5020 BERGEN

Acknowledgements

I would like to thank my supervisor, Professor Marc Bezem, for the help and encouragement he has given me during the process of writing this thesis, and for always asking the right questions.

A special thanks goes to Ole-Morten and Nils-Anders, for proofreading and for all the technical expertise they provided and patience they showed giving it.

Last, but not least, I would like to thank Kitty, for all her support and encouragement, for being a bright light when all seemed dark, and for simply being there when I needed her the most.

Contents

List of Figures

List of Tables

1 Introduction

For a long time, HTML and graphical browsing has been the way to surf the World Wide Web. But now, with an increasing effort put into voice recognition and speech synthesis, voice browsers bring the World Wide Web to the telephones of the world, not as a competitor, but as a supplement to the services and possibilities offered by graphical browsers. To ensure this expansion The World Wide Web Consortium[1] has created several working groups, that are concerning themselves with different aspects of voice browsing, and there is an ongoing process to create an open standard for creating voice applications. This is VoiceXML — an XML[2] hybrid — which is currently to be found in version 2.0.

VoiceXML applications at its most basic level are easy to develop, much like HTML, and can be deployed immediately, since the technology to support them is already in place. This is in stark contrast to the proprietary interactive voice response (IVR) platforms that have been dominant up until now, which require expensive equipment running proprietary formats, and which need to be operated by specialists.

There are two main types of spoken language dialog systems; transaction based and information provision systems. Transaction based systems let the user conduct some kind of transaction, like buying or selling stocks, whereas information provision systems provide some sort of information on request, like weather information. This thesis will look into this new standard, and make a test application to see what needs to be taken into consideration, and what is necessary to make it a good application, especially from a user's point of view. The focal point of this paper will be the design process. The test application will be an automatic directory service, i.e. an information provision system, where a caller interacts with an information system through a voice-to-text interface, and the system responds by means of text-to-speech.

Section 1 will be an introduction to voice-browsers and the VoiceXML standard, which then will be dealt with in greater detail in section 2. In section 3 a presentation and an analysis of the area of interest will be made, which will be reflected over in section 4, where issues regarding the design of the system will be treated. The resulting application will be presented in section 5. A more in-depth presentation of grammars and different grammar formats will be given in section 6. Section 7 concludes the paper by summing up and reflecting over the experiences made.

[1] http://www.w3.org
[2] http://www.w3.org/XML

1.1 What is VoiceXML?

VoiceXML is, in short, the HTML of voice-browsing, an open standard markup language for voice-based interaction between man and machine. VoiceXML is based on W3C's Extensible Markup Language (XML), and is designed to give the programmer extensive control over the flow of the dialog, which shows that VoiceXML is more than just "voice HTML", since HTML lacks this kind of control feature. Whereas HTML assumes a graphical web-browser, with display, keyboard and mouse, VoiceXML assumes a voice-browser, with audio and keypad input — or Dual Tone Multi Frequency (DTMF) input as it is also known — and audio output. The voice-browser, or voice interpreter, relies on automated speech recognition (ASR) for the input, and text-to-speech (TTS), speech synthesis, and recordings for the audio output. The user is essentially interacting with the system by listening to prompts and recordings, and directs the flow by means of spoken input.

Lately, there has been a shift from TTS to waveform concatenation, i.e. speech generated from libraries of prerecorded waveforms to create a more lifelike and seamless output. This shift of focus is mainly because tests show that users judge voice applications on the basis of this very output, and the judgement is passed swiftly, so there is a higher demand and expectancy in this field than previously. Growing use, and therefore a greater range of potential users, is also a reason. This has led to an increasing effort put into making the quality of the output better, instead of solely focusing on recognising the input. This may, on the other hand, lead the user to perceive the computer as more lifelike, and thus put undue confidence in the application's ability to comprehend and infer meaning, a process called anthropomorphism, which we will come back to later.

Voice applications do not neccessarily need speech technology, but may even be implemented using the keypad and prerecorded waveforms. This will, of course, not require the same level of hardware and expertise, and might as a result be more cost-efficient, but the applicability will in all probability be somewhat limited. For instance, when ordering a taxi by telephone, an automatic system could look up the address of each incoming call, and offer the caller to send a taxi to this address, thereby bypassing the queue to the manual operator. If the caller wants this, he can confirm by pressing the keypad, or he can wait for an operator to answer his call. For most customers, this will be what is needed, and the system, in its simplicity, is enough to lessen the workload on the manual operator severely. This will also benefit those customers with the need for a manual operator, since this may shorten the wait for a manual operator.

1.2 Why voice-browsers?

Voice-browser systems are handsfree, making them usable where handheld devices would be awkward and perhaps impossible to use, e.g. in complete darkness, or when operating in an environment that requires free use of both hands, like driving. They are also low cost, and therefore a viable option to expensive human operators. Voice-applications are also online twentyfour hours a day, all year, without any additional cost.

They also extend the availability to groups that are excluded today — partly or completely — from graphical web browsers, like the hard of seeing, the blind, or even illiterates. And as telephones are far more common than computers, voicebrowsing by telephone makes the Internet accessible to a great number of people who are thus far not connected through computers, thereby increasing the mass of potential customers, or recipients of information, enormously, without incurring any additional cost through hardware upgrades for the customers. The application is available from a hightec mobile telephone in New York, or a payphone in Ouagadougou.

However, all that glitters is not gold. In noisy environments, voice-browsers will be difficult to use, if not completely useless, and poor audio-quality in the transmission could make it hard to navigate through the applications. Also, limitations on speech recognition technology may make them poor choices sometimes, e.g. if non-native speakers are trying to make reservations for plane tickets on an automated system. The former may prove problematic to remedy, but the latter can — in some cases — be helped through making the application *speaker-dependent*. Other shortcomings of interest might be that a voice-browser obviously does not support graphics, and they may not be adequate in situations where privacy is needed, for instance when logging into a system with user name and password.

Speaker dependence means to which degree the system requires the knowledge of a speaker's voice characteristics to successfully process speech. The speech recognition engine can be taught how the individual user pronounces sounds, words and phrases, and can accordingly be trained to the individual user's voice. If the speech recognition system has to work on a large vocabulary, this is a clear advantage to ensure recognition, and avoid possible pitfalls. But in the test application this is not feasible, because, firstly, one cannot expect the users to spend the time and money needed to train the system to their voice, and secondly, the administration of such a system would be hard, if not impossible, simply because of the number of users.

Another way to minimise the latter problem is to make the application multimodal, that is, to allow for alternate ways of navigation according to where or how the application is to be used, or who is using it. Of course, one may then lose some of the advantages mentioned previously, but one gains a higher chance of success. This means, in the case of non-native speakers trying to order tickets, that providing touchi tone alternatives for destinations and times, could be a possible circumvention of recognition problems. For a person who is able to fully make use of all the available methods of input, voice enabled interaction will only increase the usability, e.g. by making her able to edit a document by voice, mouse or keyboard.

2 VoiceXML

A VoiceXML application is built up from one or more VoiceXML documents that have the same application root document, and each document contains various VoiceXML instructions for the application. The root document is loaded whenever one of the application's documents is loaded, and it remains loaded as long as the application is active. The information in the root document is available to all documents in the application.

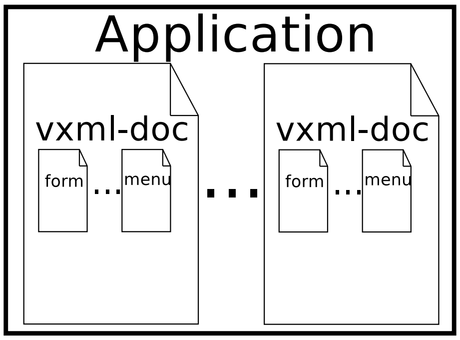

Figure 1: Application build-up

VoiceXML documents define applications as a set of dialog states, and the user is, at any time, either in a state or being transitioned to a state. Every dialog can be broken up into discrete dialog elements called *forms* or *menus*, with every form or menu having a name, and being responsible for executing some part of the dialog, and choosing which dialog state to transition to next. For example, a menu presents the user with a set of choices, and based on the choice made, the user is transitioned to another state. This goes on until there are no more states to transition to, or the user explicitly chooses to exit the application. A form defines an interaction that collects information from the user, and then, based on this information, makes the transition to a new state. Essentially, a menu is a form with only one piece of information to gather, and will therefore not be treated separately, but implicitly together with the forms.

So, every VoiceXML application or document constitutes a conversational finite state machine, moving the user from one state to the next, each transition decided by the dialog element the user is in at the time. These transitions are specified using Uniform Resource Identifiers (URI), which can point to another form in the same document, another document, or to a document in a completely different application. If the URI does not refer to a document, the current document is assumed, or if no dialog item in the document is specified, in which case the first dialog in the current document which has not yet been visited, is assumed. Execution is terminated when a dialog does not specify a successor, and all dialog elements in the current document have been visited, or if an explicit exit command ends the dialog. All these transitions are handled and controlled by the *Form Interpretation Algorithm* (FIA).

2.1 Form Interpretation Algorithm

The Form Interpretation Algorithm (FIA) determines in which order the different elements of a form are to be executed. Every form has one or more form-items, which all have three common — but optional — attributes. They may be named with the *name* attribute, given an initial value with the *expr* attribute, and a guard condition may be explicitely specified with the *cond* attribute. Form items are either *control items*, e.g. the <block> and <initial> elements, *field items*, e.g.<field>, <transfer>, or subdialog elements.

The control elements are used to process data, or to initialise variables. The <block> element, for instance, may execute an action, but does not gather user input, while the field elements are used to prompt the user for input. The input given must be in accordance with the active grammar set, which defines the input allowed. If no input is given, or the input does not match with a grammar, an event handler will be activated to solve the problem. All the field elements assign the user input to a local variable, whose name matches that of the *name* attribute of the field item. Grammars will presented in more detail in sections 4.3 and 6.

When the recogniser responds, the FIA searches for application-defined executable code, contained within the <filled> elements. This element has two optional attributes; *namelist* and *mode*, where *namelist* is a space-delimited list of form items (informal variables) to which this <filled>-element applies. The *mode* attribute is either defined as *all* or *any*, which refers to how many form items in the namelist needs to be filled for the action to be carried out. *All* would, as the name states, need all the items, while *any* will execute if the last user input matched any of the items in the *namelist*.

The FIA can be divided into four phases: the *initialisation* phase, where all variables and counters are reset, or set to predefined vaules, the *selection* phase, that determines which form item to visit next, the *collection* phase, which attempts to collect information from the selected item, and the *processing* phase, which either transitions to a location specified, generates an event, or executes the action specified by the <filled>-item. A menu can be viewed as a form containing a single field whose grammar and <filled> action are constructed from the <choice> elements.

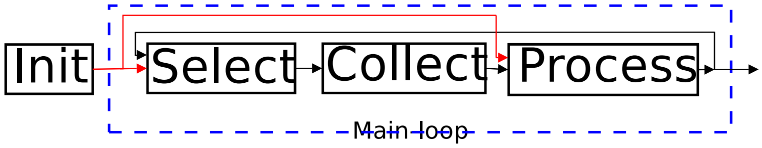

Figure 2: The phases of the Form Interpretation Algorithm

Init: initialise all formal and informal variables, either to explicitely set values by using the "expr" or "src" attributes, or to be undefined. Prompt-counters, that increments each time a prompt is played, which can be used for tapered prompting, are set to 1. If this form's grammar is visible outside the form, and the user entered by a match elsewhere, the main loop is entered at the processing phase, since there is input to process already.

Select: choose a form item to visit. If the last FIA main loop iteration ended with a <goto nextitem>, the target form item is chosen, otherwise a form item with an unsatisfied guard condition is chosen. If none exists, an implicit exit is executed

Collect: The selected form item is visited. Prompts for the form item are queued, the grammars for the form item are activated, and the form item is executed. If a <field> item is selected, the user input is collected, or if a <block> is chosen, then the block's form item is set to a defined value, thus ensuring it will not be executed again in this run of the FIA, and the executable context of the block is run.

Process: If an event was thrown in the previous phase, this will be handled first, e.g. <no match> or <no input> events. Next, the interpreter identifies the context of the grammar that was matched. If the match was made in a grammar other than the local form item grammar, FIA exits and passes execution to a new form-item, else the execution will proceed in the current form item. Finally, if the <filled> condition is satisfied, the action defined therein is executed, else a new iteration of the main loop will be made.

So, the VoiceXML interpreter is at all times in one of two states: *waiting* for input or *transitioning* between form items in response to an input. For a more detailed reading on FIA, read appendix C in the VoiceXML standard [W3Cb]. In table 1 one can see an example of a caller interacting with the main menu, choosing the residential lookup. In table 2, which is an example of a complete run of the test application, one can see how the Form Interpretation Algorithm interprets this input. The different phases are shown according to the pseudocode in appendix C in the VoiceXML standard, and the VoiceXML code that is used can be found in Appendix B.1.

Application:	Welcome to Foo Automatic Listing Service (prompt1) Please choose your service: Residential, business, reverse lookup (prompt2)
Caller:	Residential
	Caller transferred to residential ...

Table 1: A run of the main menu

2.2 Architectural Model

A VoiceXML-application is, as was previously stated, a collection of VoiceXML documents, which in turn may contain one or more dialogs in the shape of forms or menus.

A document-server (e.g. a web server) processes requests from a client application — the VoiceXML interpreter — through the VoiceXML interpretation context. In reply to this, the web server produces VoiceXML documents, which in turn are processed by the VoiceXML interpreter. The documents need not all be situated on the same server, but will be accessible through URIs whereever they may be situated. This is illustrated in figure 3.

The implementation platform is controlled by the VoiceXML interpreter and by the VoiceXML interpreter context. The implementation platform generates events in response to user actions and system events, and some of these events are acted upon by the VoiceXML interpreter itself, while others are acted upon by the VoiceXML interpreter context. For example, if the user

	Entering FIA for: <form id='intro'>
Init:	Initialise the block item at line 13 (block1)
Select:	Guard condition for block1 is true
	Select item block1
Collect:	Set (informal) block form item variable block1 to true
	Execute content for block1 (prompt1 is queued)
Process:	Nothing to do, continue main loop
Select:	Last iteration of FIA ended with a <goto next> item, therefore its attribute is selected (**#main**)
	Entering FIA for: <menu id='main'>
Init:	Initialise item main
Select:	Guard condition for main is true
	Select item main
Collect:	Calculating queued prompts, current prompt counter is 1
	Adding prompt2 to queued prompt list
	Retrieving active grammars
	Executing field item **#main**
	Play prompt1: "Welcome to Foo Automatic Listing Service"
	Play prompt2: "Please choose your service: ..."
	Collect input from user (user chooses 'residential' either by ASR or DTMF)
Process:	Utterance is matched against the active grammar set (in this case the three choices in the menu, with corresponding dtmf-values)
	Transition according to the *next*-attribute to residential.vxml
	Entering FIA for residential.vxml...

Table 2: FIA interpretation of the test run made in table 1

gives a certain input, the VoiceXML interpreter context may transition the user to another place in the dialog, or to a new dialog, like saying "main" will bring the user back to the main menu in the test application.

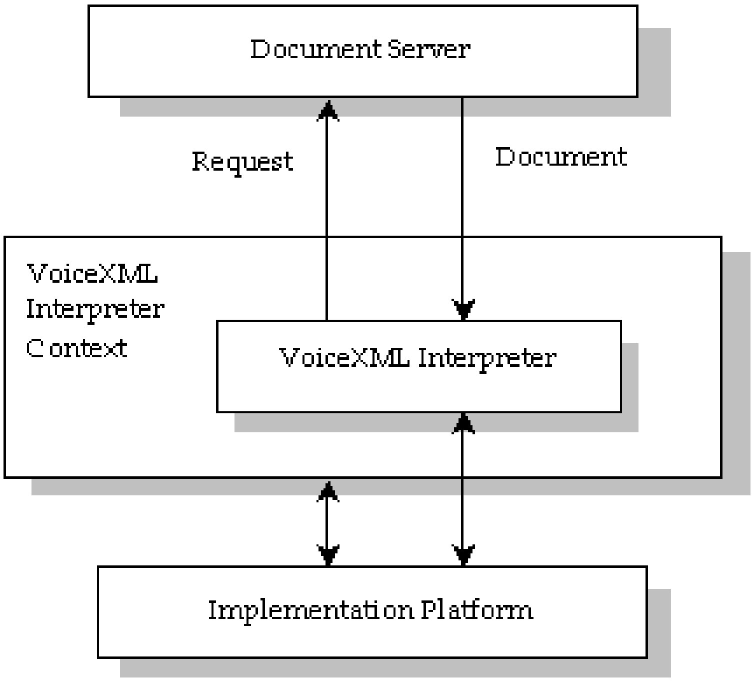

Figure 3: Architectural Model (from the VoiceXML standard [W3Cb])

2.3 Test application

A test application was made for a commercial directory service, where the callers can consult an automated telephone directory service by voice queries. The application uses BeVocal's online deployment environment, PHP and PostgreSQL. PHP — or PHP Hypertext Preprocessor — is a simple, but powerful open source scripting language, used to serve dynamic web content. The test application uses PHP version 4.3.4. PostgreSQL is an open source database, used to store the sets of test data, and PostgreSQL version 7.4.1 is used here. In building the test application, general principles of usability and programming was emphasised. The test application will be dealt with in greater detail in section 5.

The VoiceXML code in this paper is written in accordance with the VoiceXML standard version 2.0 [W3Cb] and tested using BeVocal's online programming environment[3], taking care to avoid using proprietary tags and options where possible.

A subset of VoiceXML has been defined (see Appendix A) to make this paper more self-contained and to make the reading of this paper easier. It also gives the reader the most commonly used VoiceXML elements, and may serve as a springboard into VoiceXML proper.

3 Directory Service

The test application in this paper is an automated commercial directory service. A directory service delivers the listed information related to phone records, such as phone number, or information on who owns a particular phone number, like name or address. This is normally done by operators who specialises in querying a database, and who have been trained to extract information from the database, as well as from the callers.

3.1 Methodology

To get a clearer understanding of what a directory service offers, and, in particular, how the interaction between the service provider representative — the operator — and the service user — the caller — goes, a study of such a service provider has been conducted.

Several operators were interviewed, the routines on how a new operator is trained, and what guidelines they follow, were scrutinised. Some time was also spent listening to operators answering calls.

Talks with the technical staff were also carried out, to get a clearer understanding and overview of the underlying technology on which the service is built.

3.2 Operators

Human operators play a key role in call-centers supplying directory services, since they are the "interface" the customers meet. Even though efficiency and reliability in finding the "correct information" is important, the behaviour and professionalism of the operator is also part of the assessment the customer makes of the service.

Most directory service providers can meet the two first requirements, and the gap between the providers is in this respect miniscule. Therefore the

[3] `http://cafe.bevocal.com`

latter two play an increasingly important part. To ensure that these non-measurable goals are met, new operators go through rigorous training.

Although template dialogs have been created, they can be seen as nothing more than a guideline in the real world. Obviously, many calls follow a specific pattern, making them suitable for a template, but many calls are also truly original. The experienced operator knows how to handle these, and gently navigates the caller through.

Most operators query the database as soon as they have some input, and narrow or widen the search by deleting or adding more information depending on the response from the database.

3.3 Analysis

A directory service offers the possibility to get listed telephone numbers by giving information connected to a certain person or company. There are three main categories of calls made: residential, business, and reverse lookup. The first two are what traditionally is understood by directory service, one wants to find a phonenumber of a person, a restaurant et cetera, but reverse lookup, i.e. retrieving information associated with a phonenumber – such as name or address – is becoming increasingly popular. Reverse lookup is in some countries prohibited by law, to ensure people protection of their privacy, whereas other countries lay this choice with the individual person, in giving them a choice to individually stating what information should be made available, and under which circumstances. For example, one can allow for one's phone number to appear in a manual phonebook, but not a digitised phonebook or directory service available through the Internet.

Through interviewing the operators, reading the work manuals, and training programs, an analysis of the directory service was made. This was done to get a clear understanding of how a typical conversation between an operator and a caller is, and also to see how an operator respond to deviations from this norm. Based on these findings an analysis was made, and a straightforward, uncomplicated call was constructed, following the general principles on dialog design as described on pp. 143–182 in [BM01] and [Shn98].

This analysis and basic template then served as a point of departure for the VoiceXML dialogs. On a basic level all calls can be said to be broken into three phases:

1. Greeting
2. Dialog/Input phase
3. Output

Operator:	Welcome to Foobar Directory Service, how may I help?
Customer:	The number of Hans Hanssen in Fooville please.
Operator:	The number is 22 34 56 78
	Call terminated ...

Table 3: A simple conversation

3.3.1 Phase 1: Greeting

A short, explanatory welcome message to start the session, and to inform the caller what services are available. It might also be prudent to notify the user that help is available and how to get it, if it is needed. The welcome message should make it clear that this is an automated service.

3.3.2 Phase 2: Dialog/Input phase

In this phase the caller gives the information she thinks neccessary for the operator to conduct the search. The example shown in table 3 is an ideal situation, and a large part of the calls handled is of this variety. But, unfortunately, not all, and one problem often encountered by operators is that a query is too narrow, i.e. the information provided for the search is too confined so that a match in the information system cannot be found. The operators handle this case by selectively removing parts of the information given by the customer[4]. This is exemplified in table 4. A usable query normally contains the last name combined with one other searchword, as a rule first name or city.

Operator:	Welcome to Foobar Directory Service, how may I help?
Customer:	The number of Hans Wilhelm Hanssen, Barstreet 33c, Fooville.
	[**Operator selectively deletes parts of input**]
Operator:	The number is 22 34 56 78
	Call terminated ...

Table 4: Example of a too narrow search

Another problem is when the search criteria are too wide, and the query generates too many possibilities. The operators then ask for additional infor-

[4]Parts that may be deleted in this case are: middle name, housenumber, and even street name

mation — in most cases the city where the subscriber lives — thus narrowing the search, as exemplified in table 5.

Operator:	Welcome to Foobar Directory Service, how may I help?
Customer:	The number of Hans Hanssen please.
	[**Operator executes query,**
	but gets too much response from the system]
Operator:	Where does Mr. Hanssen live?
Customer:	In Oslo
	[**Operator executes new query**]
Operator:	The number is 22 34 56 78
	Call terminated ...

Table 5: Example of a too wide search

The operator does not in all cases conduct a new query, but scans manually through the hits the last query generated, in case the result is already there. If the result of the query is very large, on the other hand, then refining the query would greatly reduce the number of possible hits. This is still simplified, but will serve as point of departure for an automated system, since it covers a major part of the calls handled by a directory service.

While executing the query, the operator repeats the information. This is to confirm to the caller that the request has been understood, and to avoid silence, leaving the caller to wonder what is happening.

In cases where the caller only has scarce information, a skilled operator knows how to collect pertinent pieces of information to get the "correct" result. T his may be to imply that the number may be listed under a different name (e.g. spouse), or, as mentioned earlier, omitting information that may be erroneous.

3.3.3 Phase 3: Output

In this phase the result of the query is presented to the caller. If there are multiple hits, the operator may try to group them according to a viable criteria, like city, or the caller is given the opportunity to have them all. In many cases, the caller is asked to provide more information to narrow it down. If additional services are available, these may now be offered, else the call is terminated with a "Thank you for calling" or something of this sort.

As seen in table 6 the boundaries between phase two and three can be a bit blurry. Most calls follow a path similar to this, and some may even be more

Phase	Agent	Utterance
1:	Operator:	Welcome to Foobar Directory Service, how may I help?
2:	Caller:	I am looking for A. Smith
	Operator:	<enters "A. Smith" into the IS>
		You are looking for A. Smith's telephonenumber?
		<being a quite common name, the IS gives too much feedback for the operator too process, thus making further inquiries neccessary>
	Operator:	Do you know A. Smith's address?
	Caller:	Yes, it is "Nowherelane"
3:	Operator:	There are two A. Smiths in Nowherelane, do you want both numbers?
	Caller:	Yes, please
	Operator:	The numbers are 555-55858 and 555-55859
	Operator:	Is there anything else I can do for you?
	Caller:	No thanks
	Operator:	Thank you for calling, have a nice day
		Call terminated . . .

Table 6: Phases of a call

compact. Some queries are so common, e.g. the helpdesk of a large company, that the operator knows the answer by heart, being able to reply almost instantaneously. A manual operator is also capable of inferring the correct phone number out of a set of numbers, if for example the caller is asking for a company which are listed with a lot of phone numbers, the operator may infer which number is the one to the switchboard. For an automatic service to be able to offer this kind of service, each record must be tagged to reflect if it is a switchboard or if it is an "ordinary" phone number. Generally though, a call takes thirty to forty seconds, something that must be taken into consideration when creating prompts.

4 Design Issues

Making a directory service available for automatic voice queries presented a host of design questions, ranging from optimal database design, voice interfaces and voice grammar choices, prompt design and creating data sources for dynamic inputs.

4.1 Dialog Design

The analysis of the calls made it clear that a directory service was twofold: acquiring information to provide a phone number, or provide a phone number and get the associated information. But another distinction also seemed prudent, namely business versus residential lookups. This latter distinction has to do with the database structure and creating separate queries for residential and business, since business usually contains more information. By separating them at an early stage, it may be easier to apply changes to these if the need arises.

This means that a caller will first be presented with three choices: residential, business, or reverse lookup. A help option will also be added, but will – if the dialogs are carefully crafted – hopefully prove to be superflous. Making the prompts clear, short, and offering few options will make this possible. By making it a menu with three possible choices — apart from the application-wide hotwords like "main", and event-triggers like "repeat" — minimal effort is required of the user.

Choosing one of these options will transfer the caller to a new dialog, where information will be collected, analogous to form-filling in HTML and XML. The dialogs will again need to be concise, and unambigous.

When the information is gathered, it will be submitted to another service that will execute the lookup in the database, and which will create an answer or require additional information from the caller. More on this later.

4.1.1 Design Principles

Design principles that need to be heeded are:

1. *Minimise the cognitive load for the users*, i.e. do not ask the user to remember too much. A general rule of thumb is that people can remember "seven-plus or minus-two chunks" of information. It is therefore advisable to keep menu choices to a minimum, and keep information brief. Having the key information as close as possible to the expected input will also help.

2. *Balance efficiency and clarity.* In other words; do not sacrifice the clarity of prompts and feedback to make them as short as possible. Short is not neccessarily sweet, and longer prompts may not be more clarifying.

3. *Ensure high accuracy.* This may be done by clearly stating that help is available and how the user can get it. Using tapered prompts will also ensure this. Tapered prompting means to have different prompts if the prompt needs to be repeated, e.g. if the system did not understand or recognise the input, a more elaborate prompt is played. By gradually expanding the prompt, giving more information and exemplifying the expected input, the caller is coached to give the correct input. If this does not help, the user should be transferred to a human operator.

4. *Recover from errors gracefully.* Use positive feedback or ask anew if something is wrong. Do not let the users detect this unless it is unavoidable, but then communicate errors quickly and do not pass out blame. It would also be a good idea to have the different prompts reflect what went wrong, for instance if the system could not understand the input, the system could prompt with "Sorry, did not get that. Please repeat ...", or if the system did not recognise the input, i.e. the input was outside the allowed set of utterances, the prompt could be "Sorry, did not recognise the input, please ...".

4.1.2 Design Styles

It is important to remember that many people are not comfortable, nor at ease with, speaking and dealing with machines. Nevertheless, it is important to make it absolutely clear that they are — in fact — *not* dealing with a person, but an automated service.

Making the interaction as brief as possible may be a way to deal with this. Attempting to make the application more human — anthropomorphication — would in most cases only make matters worse, because it gives the caller false expectations of what the system can do. The system does not understand more by appearing more human. To enable the system to infer meaning, it must be pre-programmed for the eventuality that a particular incident will occur. A fair number of situations might be thought of, but only "ordinary" extraordinaries, true one-in-a-million occurences will inevitably cause the system to not understand the input. It will also be a matter of cost. How much time and money is to be invested into something that is very unlikely to occur? By making it obvious that this is an *automated* system, these implications may be avoided, and it might be cheaper in the long run, to transfer extraordinary cases to a human operator, possibly charging the customer more for this service.

Creating an application that only heeds the needs of the technophobic may also be a mistake. Providing touch tone shortcuts, and the possibility to use "barge-in", i.e. to cut prompts short by interrupting them, would increase usability for the more technosavvy. This would of course increase the speed of the searches done by intermediary and expert users, whilst novice users would be able to use the slightly slower, but more comprehensible, full dialogs and voice interaction.

4.1.3 Application-directed vs mixed-initiative

There are two ways of guiding users through automated services; *application-directed* and *mixed-initiative*. By letting the application direct what is to happen and when, one may reduce the risk of errors, but the users may find this too confining and controlling, making them unwilling to use the application.

System:	Do you want residential, business or reverse lookup?
User:	Residential
System:	The name of the person you are looking for?

Table 7: Application-directed call-flow

In mixed-initiative, the caller will be more in charge, but the system will have to infer the information from the input the caller gives, making it error-prone.

A combination of both will be preferable, where for instance the user is first given the opportunity to interact using mixed-initiative, but if something

User:	I want the number of A. Smith living in London
System:	You want the number of A. Smith, living in London, correct?
User:	Yes
System:	The number is: ...
	<or get more information>

Table 8: Mixed-initiative call-flow

goes amiss, the application-directed approach can be used, to safely steer the user through. In some error-prone sections of the applications, letting the application control the interaction will minimise the chance of errors, thus giving the user a more reliable service. In the implementation of the test application, the application-directed approach will be used.

Making an application mixed-initative would give the users a feeling of being much more in control of the dialog, but probably only feasable in smaller applications. In large applications this would be very expensive, if at all possible. One would have to put severe limits on the input and limit the application's field of context. For example making an automatic switchboard for a company might be possible, depending on the number of employees, whereas creating a mixed-initiative automatic phone directory would in all probability fail, simply because of the number of possible input values it would have to contain. To correctly infer what information in the dialog the system was to interpret as a valid input value would be difficult, because of ambiguity: is the input valid, or simply "garbage"? "John Hopkins, please" would be easy to interpret, but "Give me the phone number of the head of John Hopkins, please. The hospital, not the university, I mean" would perhaps be possible, but hard.

One way of giving the users more control of the dialog, whilst using the application-directed approach, would be to give them the possibility to navigate through the menus, by enabling "key-words" like "back" and "main menu". The increase in programming cost would be low, but the increase in usability would be substantial, without making the application too error-prone.

4.2 Prompt Design

Prompts indicate that it is time for user input, and can be seen as turn-taking cues, that is, the system prompts the user for some piece of information, and waits for the user to give an input that has a match in the active grammar set. Their purpose may therefore be said to be twofold: they prompt the user to give an input, and may convey to the user what input is expected at this point in the dialog. The main menu in the test application is a good example of this, the user is asked to choose from one of the three alternatives, i.e. the expected input for this dialog is "residential", "business", or "reverse lookup". This grammar set does not express all the allowed input, but only the input that is relevant for this dialog.

It is essential that prompting is swift and efficient, but not at the expense of clarity. Short prompts with few options are therefore preferable to long prompts with many options. Preceding prompts with instructions, and only repeating the prompts may ensure this. For example, by dividing the welcome message from the menu prompts, the design makes sure that the user only needs to hear the welcome message once, even though the input given to the menu item is not recognised, or if the user makes a new query. This saves the user time, and focuses on the expected input. By having key information immediatly before the user is expected to give the input, it is made easier for the user to understand what is the expected input.

By avoiding the pronoun "I" whereever possible, one may accentuate that it is indeed an automated system, and that one is talking to a non-person. In some cases, it is not possible to leave out the pronoun without sacrificing what people perceive as "normal" conversation, and making the prompts sound robotic. This is something that must be evaluated in each independent case.

Even though TTS and waveform concatenation has improved the quality of synthesised speech, it still cannot compare to real speech. This means that using professionally recorded voice prompts, with voice pitch and intonation reflecting the context it is to be used in, will make the application more fluent. Mixing prerecorded prompts and synthesised speech in the same prompt should be avoided, even though the voice is the same. This is to keep the prompts as fluent as possible, and also to avoid too many comparisons between real speech and synthesised speech. In some instances this may result in having to choose a solution with more synthesised speech than strictly neccessary. For example "There was 1 hit. Kit Walker, Fastlane 83, 1000 Fooville with phone number 555-83389". If the names, address and city are generated by means of TTS, and the other parts are prerecorded playbacks, the mix will be unfortunate, but making the whole output with generated speech may not be an ideal situation.

Choosing what voice is to be used, is also of importance, both in TTS and the prerecorded messages. Studies show that people tend to perceive male voices as more authorative and more intelligible than female voices. On the other hand, most users expect an operator to be female, which should also be taken into consideration. If the application is application-directed, one may soften the approach by using a female voice. It is also possible to use both a female and a male voice, where for instance the male voice can be used for the help-features, whereas the other prompts are executed with a female voice, but this should be used with care, so as not to make the application seem haphazard and disorganised.

4.3 Grammar design

Everywhere the application is to have some sort of input from the user, a set of possible values to be accepted needs to be declared. This is done by using a grammar. A grammar is a set of rules or grammar classes that defines the set of expressions that are to be accepted at a given place in the dialog. Grammars may be trivial lists of possible words, or complex sets of phrases, and grammars may be incorporated into the application code — inline grammars — or they can be externally available grammar files. The inline grammars are typically quite small and uncomplicated, like the grammar in the link-element of the root document that only contains two possible values, namely *main* and *menu*, whereas an external grammar is usually used when the grammar is somewhat larger and non-trivial, like all possible surnames in a company. Most VoiceXML gateways also support compiling the grammar files, thus only having to do this once, instead of at run-time, minimising the load on the voice-portal and making the application faster.

By having the grammar file external, it is also possible to let several applications use it, eliminating the need to maintain several large grammars, that are identical, or close to it. Some grammars are so common, that they have been incorporated into the VoiceXML interpreter, for example boolean values, that should accept different forms of affirmatives, ranging from “yes” to the rather informal “yep”. One problem with built-in grammars is that they may be made for specific regions, e.g. the built-in *phone*-grammar, which has ten digits, in contrast with Norway, which has only eight.

Another benefit of having the grammar external, is that no changes to the VoiceXML code would be needed. Instead, upgrading the grammar file would simply be a question of altering a grammar file, compiling it and copying it to the correct position, thereby replacing the old grammar file, thus minimising the application’s downtime. This is of great value if it is likely that the grammar will change quite frequently, like e.g. the name-grammars in the test application. It would also be possible to switch the grammar format used, as long as the voice browser supports the grammar format. Grammars will be dealt with in greater detail in section 6.

5 The Application

The test application was built according to the VoiceXML standard version 2.0 [W3Cb], and tested using BeVocal's online development platform It is a fairly basic, bare-boned application, but it should serve well as a prototype, and it should also be sufficient in exemplifying general concepts. The application is a simple automated directory service, based on 10.000 generated test data for residential and 200 generated test data for business. All information about one person or business is called a record. The test data reside in a PostgreSQL database, which the VoiceXML application queries indirectly through a PHP-script. Based on the results of this query, the script generates a VoiceXML-document, which it returns to the voice-interpreter.

5.1 Scenario

The scenario in table 5.1 describes how the interaction between the caller and the test application system flows, and also tries to shed light on some of the internal mechanisms in the system.

In addition to the gateway phone number, every application also has an extension number, which identifies it. The voice gateway translates the extension to the corresponding URL, and requests through HTTP the VoiceXML document specified from the web server that hosts the files. This means that any web server can host the VoiceXML files, since all voice recognition and speech synthesis is done by the Voice Gateway. An application can have its files on one server, or spread on several web servers. When the web server returns the corresponding VoiceXML document, it is run through the FIA, which then generates the prompts, and handles the input. This is described in detail in section 2.1. If there is no, or too many, records found, the exceptions must be handled by the PHP. More on this is sectionapp:ext.

To summarize, the Voice Gateway routes the incoming call to the correct VoiceXML dialog, runs the dialog by means of FIA, handles input by voice recognition and DTMF, and plays output to the caller using speech synthesis and prerecorded waveforms, and then transitions to the next dialog.

5.2 Flowchart

The flowchart shown in figure 5 is somewhat simplified, as it only refers "user assistance" and it does not show in detail how exceptions like no or too many records found will be handled. The flowchart was made to only show the flow of the VoiceXML dialog in general, to give a feeling of how the application is built.

1	Caller places call to VoiceXML gateway
2	The gateway translates the extension to a corresponding URL
3	The gateway (client) places a HTTP-request to the URL spefified
4	The target webserver responds with a VoiceXML document containing a VoiceXML dialog
5	The gateway interprets the VoiceXML document, and interacts accordingly by means of ASR and speech synthesis. The collected input is then submitted by HTTP to the URI designated by the VoiceXML
6	The webserver containing the target URI processes the input, and responds accordingly

Table 9: Step by step scenario walkthrough

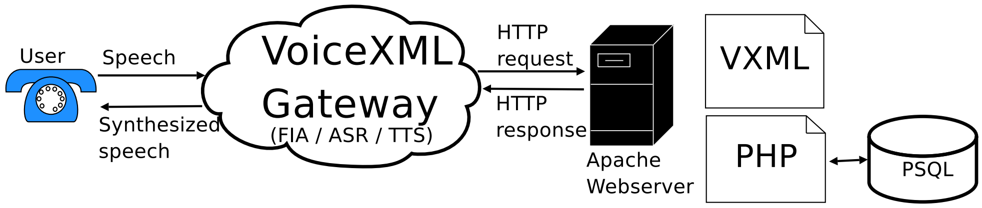

Figure 4: VoiceXML conversation flow

5.3 Possible extensions

As stated, the application is simplfied. In this section, several possible extensions and problems will be considered, and — in some cases — an outline of a solution will be given.

In section 3 and 4 it was mentioned that the users input was repeated, both to show that the input was understood, to indicate that it was being processed, and to offer an opportunity to confirm that this was indeed the correct input. As the test application was built on a text based system, recognition was not an issue until the last testing. The tests showed that asking confirmation for all input was awkward and irritating for the users. One way to solve this problem would be to take advantage of the confidence scores

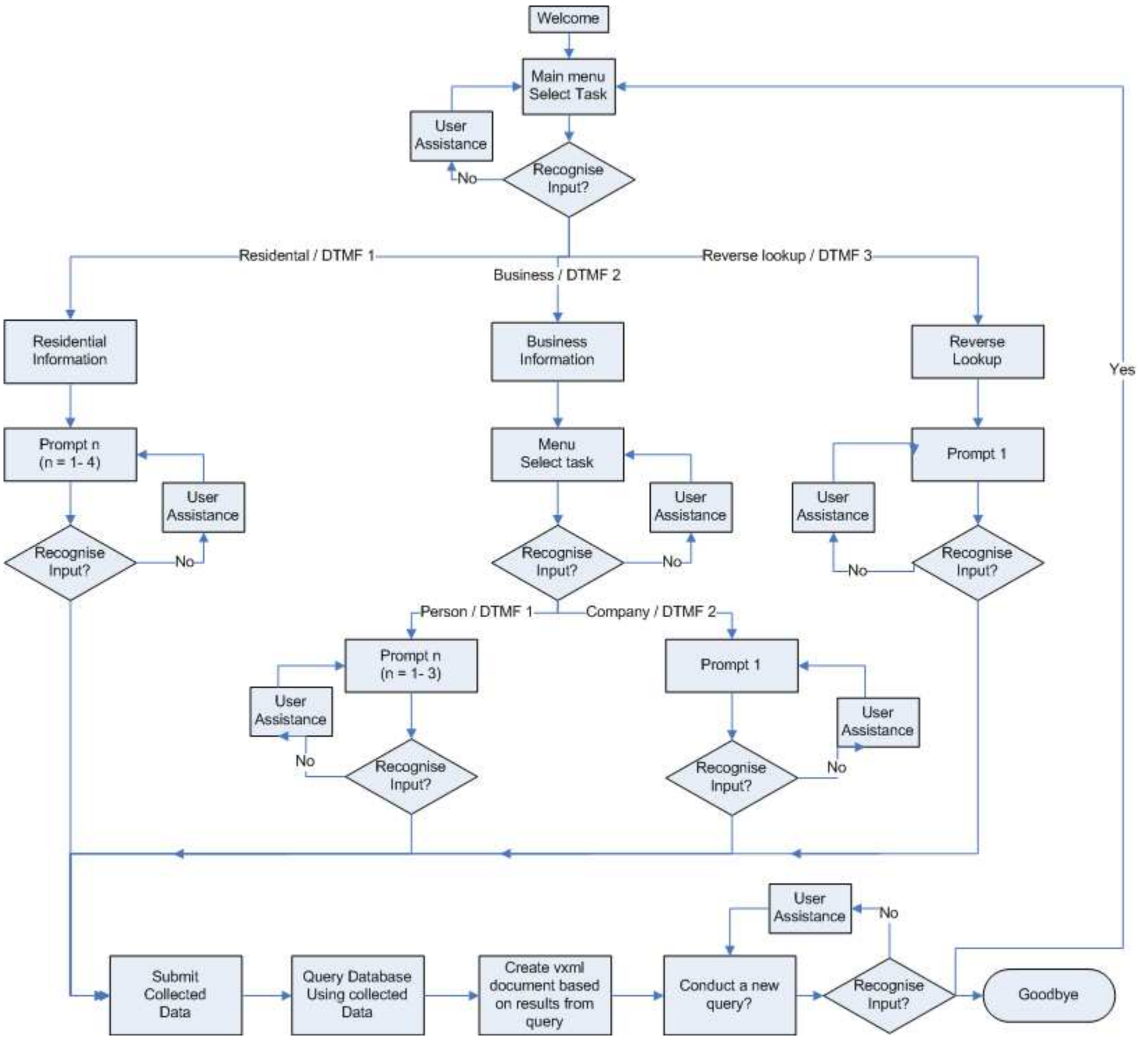

Figure 5: Flow of VoiceXML dialogs in the application

given by the ASR. When the system recognises input, it is put in a list and a confidence score is given. Confidence scores are a number ranging from 0.0 to 1.0, where 0.5 indicates a 35% confidence that this is the correct result. The confidence level to be accepted is an important consideration to make, setting it too low would result in getting more false recognitions, whereas setting it too high would increase the occurrence of no-match events. Instead of setting the confidence level higher, one could make the system ask for confirmation only if the confirmation score was too low, thus avoiding too many no-match errors, and also unnecessary confirmations when the confidence score was high. This would make the application more user friendly.

Another problem that still needs to be addressed is what to do when a query generates zero or too many possible hits. Both these problems would have to be handled by the underlying system. In the case of the test application this would mean PHP. If there where zero records found, one could eliminate parts of the input — e.g. street name, first name or even city — and see if this would generate some hits. If too many records were found, the system could be programmed to ask for additional information, based on the information that is given. An interesting question in this case is how many records are too many to give to the caller? This would be dependent on how it is presented, obviously, just giving 25 records in any order would be too many, but 25 records sorted by city and offering the caller the possibility to state which of the possible cities it is would perhaps be satisfactory. Enabling the caller to navigate in the results, and breaking off system output, would also make it possible to present more records to the caller.

In the flowchart in figure 5 no explanation is given to the term "user assistance", which would be needed everywhere the system fails to understand, either because of a recognition-error, which would mean a failure by ASR to recognise the input, or an error due to not understanding the input, meaning the input was outside the grammar. It should be reflected in the user assistance what is the reason that the prompt is played again, and it would be preferrable not to play the same prompt over and over again. This can be handled by setting a maximum number of times any prompt is to be replayed, and transferring the caller to a manual operator if this occurs.

5.4 Technology and tools used

After a brief trial with Motorola and TellMe, BeVocal's developer platform was used to develop the VoiceXML part of the application. The former two did not at the time (in their free-to-use online or desktop platforms) support the new standard, whereas BeVocal does.

A web server – Apache/1.3.28 (Debian GNU/Linux) – was set up to host the VoiceXML files. The web server also supports PHP, which is used for communicating with the database – PostgreSQL 7.3.4 – containing the test data.

All implicit and explicit variables that are used in VoiceXML are ECMAScript objects, but ECMAScript only provides light-weight general computing capability to VoiceXML code, therefore it was decided to use PHP for communication with the database and creating dynamic VoiceXML documents.

5.5 The VoiceXML Platform

As mentioned, BeVocal's online development platform was used to develop the VoiceXML-application. This platform simulates calls to the application using graphical means — VocalScripter — obviously without requiring ASR or TTS, making it easier to focus on the design and logic of the application. The platform retrieves the files from any URL specified, and also offers to compile external grammars, but only support GSL-formatted files today.

5.6 Webserver

By separating the application logic, which runs on a standard webserver, from the voice dialogs, which are running on a telephony server, the availability of voice applications are greatly increased. In particular since it enables developers to build phone services without having to buy or run the equipment needed for ASR and TTS. This can be done by *Voice Service Providers* (VSP), that are analogous to todays ISPs. A local Apache Webserver was set up, containing all the vxml and php files, which is then available from the developer platform. It is also possible to call the application through BeVocal, but this requires the caller to have an user account and a pin-code.

5.7 Database

A PostgreSQL database was set up and 10.000 residential records and 200 business records generated and inserted into the database. The database was chosen to have a flat, simple structure, meaning that all pertinent data was contained within one table. This was done to emphasise speed in searches. The table has ten fields, not all strictly neccessary in the prototype, but some fields may come in handy if the scale of the database dictates the queries to be more specific.

1. **id** A unique key for all records
2. **user_type** The user type, differentiating between residential and businesses (possible values R or B).
3. **first_name** The subscriber's first and middle name (if any)
4. **last_name** The subscriber's last name
5. **company_name** The company name, if the phone is registered to a business.

6. **street_name** The subscriber's address
7. **house_no** The housenumber
8. **zip** Zip code
9. **city** The location of the subscriber
10. **phone_no** The phone number registered to the subscriber

Also, if a subscriber has more than one phone number, each is treated as an individual record by the database. Again, this is done for efficiency, and to minimise the load on the database. It may be possible, if deemed neccessary, to emulate one-to-many relations by comparing the results and then group them if they have similarities. This increases the load on the database, but will offer greater functionality.

6 Grammars

As previously stated, grammars define sets of rules that declare what values are to be allowed within the application. Here, grammars will be introduced and presented in more detail, with examples from the test application, usually the same example in the three formats, but occasionally only in GSL and ABNF. Some comparison of the formats will be attempted, but not at great depth, since that would be beyond the scope of this paper.

6.1 Grammar formats

At the present, there are several grammar formats available for creating the grammar files needed in voice applications. The World Wide Web Consortium is now working on a new standard grammar format, the Speech Recognition Grammar Specification Version 1.0 (SRGS) [W3Ca], which was recently submitted for recommendation. New grammars for voice applications should be made to conform with this standard, thus avoiding the proprietary formats, like Nuance's Grammar Specification Language (GSL) and Java Speech Grammar Format (JSGF), that are in widespread use today. SRGS is in fact built on JSGF, and therefore JSGF will not be handled separately here, but indirectly through SRGS, even though they are not identical, but a step further in evolution.

In voice enabled browsers to come, SRGS must be supported, whereas GSL and JSGF may be supported. The grammar compiler on BeVocal's online development platform only supports GSL today, and therefore this has been chosen as grammar format in the test application.

Operator	Name	Usage	Meaning
()	Concatenation	(A B C ... D)	A and B and C ... and D
[]	Disjunction	[A B C ... D]	A or B or C ... or D
?	Optional	?A	A is optional
+	Positive closure	+A	1 or more repetitions of A
∗	Kleene closure	*A	0 or more repetitions of A

Table 10: Syntax of the right-hand side of GSL-rules

6.1.1 Speech Recognition Grammar Specification (SRGS)

The World Wide Web Consortium's (W3C) ongoing project in creating a standard for the development of grammars to be used in voice-based applications, SRGS, can be built on two different forms, e.g. *Augmented Backus-Naur Form* (ABNF) and *XML Form*. These forms are semantically mappable, thereby making it possible to build automatic transformations between the forms. This means the forms can be used interchangably, as they both either accept a given input or they both reject it, and they will also parse any input string they have accepted identically.

6.1.2 Grammar Specification Language (GSL)

This is the proprietary format from Nuance, one of the leading companies in the field of voice recognition. The syntax of the right-hand side of the GSL rules are depicted in table 10, where A. . . D can be terminals, non-terminals, or the expressions themselves. The syntax and other aspects will be dealt with in greater detail in the following sections.

6.1.3 Input modes

There are two ways of communicating with a voice-browser, voice and DTMF. The SRGS grammar formats are restricted to either recognising DTMF or speech, not both, but GSL has no such restriction. This means that GSL can have a mix of both DTMF and speech, enabling it to have "type-ahead" shortcuts which would be impossible with ABNF and XML. For example, in the testapplication, it would be possible to type the phonenumber and also type '2' and '1' to take the user directly to the person lookup in the business category, which would be a nice feature for novice and expert users.

non-terminal	=>	right-hand side
right-hand side	=>	expression as described in 6.1.2 for GSL, or the ABNF equivalents

Table 11: GSL and ABNF rules

6.2 Grammar build-up

A grammar is built up of one or more rules which specifies the input values that are to be accepted. Each rule can consist of two parts; an optional rulename that is unique within the grammar and which identifies it for use in other rules, and an obligatory rule expansion part that defines the possible values that can be uttered.

As inline grammars ABNF and XML must have a top level rule — root-rule — that is an explicit starting point of the grammar, but if the grammars are external, this is optional. GSL, on the other hand, always takes the first publicly scoped[5] rule to be the root. It is also possible to specify which rule to use in particular, to directly reference the rule needed, if this rule has public scope. This is not the case when using precompiled grammars, which must have a root-rule. If no rule is specified when referencing the grammar, the root-rule is assumed.

6.2.1 Rule syntax

The syntax of the rule names differ between the various grammar formats. It is to be noted that names starting with a capital letter in GSL is a reference to another rule, whereas ABNF would use a "$rulename" to indicate a rule reference, and XML would use <ruleref uri="rulename"/ >. So, "Example" would mean a rule reference in GSL, and "example" would be a terminal, while in ABNF both would denote the same, e.g. a terminal. In XML this would be handled by the use of the appropriate tags.

The way the rules are declared also differ slightly. Whereas XML handles the declarations by means of tags and their attributes, GSL and ABNF rules are built as depicted in table 11.

In the examples below, the grammar rule *lang* gives three alternatives; *prolog, standard ml* or *voicexml*. On the right-hand side of GSL and ABNF the name of the rule is given, e.g. "Lang" for GSL, and "$lang" for ABNF. In XML this is done by an attribute *id*. These are the names used when referencing the rules.

[5]See section 6.2.2 for more on this.

GSL:	Lang = [prolog (standard ml) voicexml]
ABNF:	$lang = prolog	(standard ml)	voicexml
XML:	<rule id="lang"> <one-of> <prolog> <(standard ml)> <voicexml> < /one-of> < /rule>

6.2.2 Scope of rules

Each rule that is defined has a scope, either *public* or *private*. If nothing is specified, the scope is set to private by default. A rule with public scope is also visible outside its grammar, as opposed to a grammar with private scope, that is confined to be visible inside the grammar in which it has been defined. This means, a public scoped rule may be referenced explicitly in the ruledefinitions of other grammars, but a private-scoped rule cannot, being only accessible from within the same grammar in which the rule is declared. The syntax of how scope is declared is shown below.

ABNF:	private/public $rulename = $choice_1 \| \ldots \| choice_n$
GSL:	Rulename:public = [$choice_1 \ldots choice_n$]
XML:	<rule id="rulename" scope=private/public> <one-of> <item> $choice_1$ < /item> <item> $\cdots$ < /item> <item> $choice_n$ < /item> < /one-of> < /rule>

In GSL there is no special syntax for marking the scope of a rule as private. If no rule in the grammar is marked public, then *all* rules in the grammar are implicitly public. But if one or more rules are marked public, then all rules *not* marked public have private scope.

6.2.3 Recursion

Both GSL and SRGS[6] permit rules to directly or indirectly reference themselves, thus giving them the expressive power of *context-free grammars* (CFG).

[6]SRGS is here short for both grammar forms (ABNF and XML)

But, one should note that it is not required for the form grammar processor to support recursive grammars, and that it is *simple right recursion* and *embedded recursion* that is supported. *Simple left recursion* is made illegal, to ensure that the interpreter does not enter an infinite loop. In this section only ABNF and GSL examples are shown.

This means, that while

ABNF: $Digits = $Digit | ($Digit $Digits)
$Digit = 0|1|...|9
GSL: Digits [Digit (Digit Digits)]
Digit [0 1 2 ... 9]

is allowed, this is not

ABNF: $Digits = $Digit | ($Digits $Digit)
GSL: Digits [Digit (Digits Digit)]

It is also possible for rules to indirectly reference themselves, by referencing a rule that has as one of its subcomponents a reference to the originating rule itself.

ABNF: $NounPhrase = $Noun | ($Noun $PrepositionalPhrase);
$PrepositionalPhrase = $Preposition $NounPhrase;
GSL: NounPhrase (Noun ?PrepositionalPhrase)
PrepositionalPhrase (Preposition NounPhrase)

One should also take care to avoid left-recursion when indirectly referring to rules, because the left-recursion is less obvious in these cases, as exemplified below.

ABNF: $a = ($b $c);
$b = ($c b b);
$c = ($a c);
GSL: A (B C)
B (C b b)
C (A c)

6.2.4 Special rules: NULL, VOID, GARBAGE and RESISTOR

The grammar formats also contains some special rules. Two which they all have in common: *VOID* and *NULL*, and apart from that, ABNF and XML have one other they share *GARBAGE*, and GSL has one called *RESISTOR*.

NULL: Defines a rule that matches if the user is silent, for example using the following grammar rule Lang = ([NULL standard] ml) to match "standard ml" or just "ml". This may in many cases also be solved using optional constructs, i.e. Lang = (?standard ml).

ABNF: $lang= ($NULL ml) | (standard ml)
GSL: Lang = [NULL standard] ml
XML: <rule id="lang"/ >
<one-of>
<item><ruleref special="NULL"/ > ml< /item>
<item> standard ml < /item>
< /one-of>
< /rule>

VOID: Defines a rule that does not match anything that may be spoken, making the sequence in which it is put unspeakable.
ABNF: $VOID
GSL: VOID
XML: <ruleref special="VOID"/ >

The NULL and VOID rules can be used to decide if a rule is active or not, without having to completely change the grammar, which would be awkward to maintain. For example, assume an application that offers weather reports. In summer it also offers watertemperature and opening hours for water resorts, and in winter it answers questions about amount of snow at different skiresorts. These alternatives should not be available at the same time, and the VOID and NULL rules can be used to create a grammar where these rules are easily turned on and off.

GARBAGE: Defines a rule that matches anything until the next rule match, the next terminal or the end of the input. This is not available in GSL, only in ABNF and XML.
ABNF: $GARBAGE
XML: <ruleref special="GARBAGE"/ >

RESISTOR: Changes the probability that a rule can be spoken. This is only available with GSL.

6.2.5 Rule repetition

Sometimes it is necessary to allow for an expression to be repeated any or a particular number of times. This is solved quite differently in SRGS and GSL. As depicted in section 6.1.2, GSL has three operators that handle repetitions. In SRGS this is done by means of giving a number n to say how many repetitions are needed, with $n \in \mathbf{N} \cup \{0\}$, or a range $n - m$, with $m, n \in \mathbf{N} \cup \{0\}, n \leq m$, is specified.

Format	Syntax	Meaning	Example
ABNF	<0-1>	Optional	$answer = yes please<0-1>;
XML	repeat="0-1"		yes <ruleref id="please" repeat="0-1"/ >
GSL	?		Answer (yes ?please)
ABNF	$< n >$	n repetitions	$pin = $dig<4>;
XML	repeat="n"	$n = 4$	<ruleref id="dig" repeat="4"/ >
GSL	none		Pin (Dig Dig Dig Dig)
ABNF	$< n - m >$	repeat between n to m times	$topping = $top<2-6>;
XML	repeat="$n - m$"		<ruleref id="top" repeat="2-6"/ >
GSL	none		Topping (Top Top ?Top ?Top ?Top ?Top)
ABNF	$< n- >$	repeat zero or more times	
XML	repeat="$n-$"		
GSL	*		
GSL	+	one or more	

6.3 Grammars in the application and possible extensions

A grammar that can be used in a real-life directory service will be costly to build and to maintain, since it must contain all possible values, e.g., all the possible names in the directory service. In this paper a small set of data was created and the grammars made to suit them.

Even though the grammars of the test application are quite trivial, they are external. This was done deliberately, for readability and ease of updating the grammars. It would have been possible to extend the grammars quite dramatically, but the intention of the test application was to show how easy it can be to develop VoiceXML-applications, and to make it as generic as possible. As it stands now, the test application can be used as a framework for many kinds of automated directory service.

One extension to the grammars that would be advisable would be to group the input where this is possible. That is, group elements that have the same semantic content, like "doctor, general practioner, GP", "twenty, 20", and the aforementioned boolean epxressions "yes, yep, yes please" and so on, and return one representation of these, e.g. "doctor", "20" and "yes".

This is called *slot-filling commands* in GSL, or *tags* in SRGS, and is an interpretation of the semantic content that was recognised. If this is used, the voice interpreter will return the value that is defined in the slot or tag.

ABNF:	$color = blue | marine {color="blue"};
XML:	<item>blue <tag>color="blue"< /tag>< /item>
	<item>marine <tag>color="blue"< /tag>< /item>
GSL:	Color [[blue marine] {color blue}]

It is also possible to use weight on the elements, i.e. indicating the likelyhood of an element occurring. For instance, when dealing with cities, a big city is more likely to occur than a small one. By logging the number of times the specific elements occur, one can also create weights that reflect probability. This can be reflected in the grammars as indicated below.

ABNF:	/0.25/(Baton Rouge) | /3.14/(Washington DC);
XML:	<item weight="0.25">Baton Rouge< /item>
	<item weight="3.14">Washington DC< /item>
GSL:	[(Baton Rouge) 0.25 (Washington DC) 3.14]

Another thing that might make a automatic directory service more usable is the possibility to mark the way certain elements are to be pronounced, either in external lexicon documents or changing the language that are associated with a rule expansion, i.e terminal, rule reference, or a combination of both. The prior is not supported by BeVocal today, and the latter only supports US or British English.

It would be a daunting task to try to have all the names of people and streets to be correctly pronounced, but this should nevertheless be strived for. Foreign names, or even foreign-sounding names, are a challenge. The SRGS formats allow for tagging names with a language marker that specifies how it is to be understood, so that recognition is more certain. But even if this is possible, it would mean that someone would have to tag all these words appropriately. For this to be feasible, the number of elements to be tagged in this manner would have to be fairly small, and not likely to change too often. The SRGS formats support this kind of tagging, whereas GSL does not.

7 Conclusions

As this is a fairly new field, a field which is of growing interest and which is rapidly expanding, it is also "work in progress". This applies both to VoiceXML and to SRGS, which the World Wide Web Consortium makes an effort to keep up to speed with what users and the industry want at any given time. This is reflected in the regular updates that are made to the standards.

Testing VoiceXML has shown that the specifications sometimes actually exceed what the voice service providers (VSP) offer today. This is particularly the case with regard to SRGS, which for example support both external pronunciation lexicons and language tagging of the grammar rules. BeVocal, on the other hand, does only support limited language support and no pronunciation lexicons yet.

VoiceXML is easy to use, like its counterpart HTML, and it requires little training, and hardly any hardware, to develop an application. All that is needed is a webserver to serve the vxml-documents and a VSP to handle interpretation. So anyone can make their own voice-application. This paper has tried to shed light on some important aspects one needs to keep in mind when creating voice-enabled applications, in a hope to help future VoiceXML-developers.

Much of the work put into creating a VoiceXML application, goes into creating the underlying system which produces the dynamic VoiceXML documents, and handles the logic of the application. VoiceXML is more or less only a way of presenting the data, much like its graphical equivalent HTML. Therefore good design is paramount in creating userfriendly applications.

Having a text based environment for testing made it easier to focus on the dialog flow and prompt design, because even though the quality of ASR and TTS has improved lately, it still leaves a lot to hope for. And before this is better, many users will avoid using automated services, even if it is saving them time and money.

For those expecting voice-interaction with computer systems like in *Star Trek* or even in *2001: A Space Odyssey* in the near future, well, they will have to wait longer. It is also important to keep in mind that voice-recognition systems are just that, they recognise input and generate output, “Vox et praeterea nihil” — a voice and nothing more. To some extent they may also try to infer semantic meaning in the given input, but this is still only feasable within a given domain with a restricted context, largely due to the fact that this is language specific. This would imply that languages spoken by relatively few people would not have semantic interpreters developed for their language, since this is not interesting from an economical point of view. The only hope here would be linguistic enthusiasts from the open-source communities.

A VoiceXML subset

These are the elements actually used in the test application, with a short explanation. It is important to note that this is *not* an exhaustive list of VoiceXML, but only a subset. Most elements have more attributes, legal parents, and legal children, but they have been removed for easier reading. For the complete syntax of VoiceXML, see section 1.4 in the VoiceXML Standard Version 2.0 [W3Cb].

To make it easier to see how the code is built up, an EBNF[7] representation of the elements used in the test application has been made. {} indicates zero or more repetitions, [] denotes an optional construct, | divides different choices, and () groups the elements contained within.

<table>
<tr><td>VXML</td><td>::=</td><td><vxml ATTR> [MENU LINK] {FORM} < /vxml></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>FORM</td><td>::=</td><td><form ATTR> [BLOCK FILLED] {FIELD}< /form></td></tr>
<tr><td>MENU</td><td>::=</td><td><menu ATTR> [PROMPT HELP] {(CATCH|CHOICE)} < /menu></td></tr>
<tr><td>LINK</td><td>::=</td><td><link ATTR> GRAMMAR< /link></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>FIELD</td><td>::=</td><td><field ATTR> GRAMMAR PROMPT< /field></td></tr>
<tr><td>FILLED</td><td>::=</td><td><filled> {VAR} [IF] ACTION < /filled></td></tr>
<tr><td>BLOCK</td><td>::=</td><td><block>TEXT GOTO< /block></td></tr>
<tr><td>CHOICE</td><td>::=</td><td><choice ATTR> TEXT< /choice></td></tr>
<tr><td>HELP</td><td>::=</td><td><help>TEXT ACTION TEXT< /help></td></tr>
<tr><td>CATCH</td><td>::=</td><td><catch ATTR> {(ACTION|TEXT)}< /catch></td></tr>
<tr><td>GRAMMAR</td><td>::=</td><td>(<grammar ATTR>[TEXT]< /grammar>)|
(<grammar src= "TEXT"/ >)</td></tr>
<tr><td>IF</td><td>::=</td><td><if ATTR> ACTION [ELSE]< /if></td></tr>
<tr><td>ELSE</td><td>::=</td><td><else/ > TEXT ACTION</td></tr>
<tr><td>ACTION</td><td>::=</td><td>GOTO | SUBMIT | <exit/ > | <reprompt/ > | <enumerate/ ></td></tr>
<tr><td>GOTO</td><td>::=</td><td><goto next= "TEXT"/ ></td></tr>
<tr><td>SUBMIT</td><td>::=</td><td><submit next= "TEXT" namelist= "TEXT"/ ></td></tr>
<tr><td>VAR</td><td>::=</td><td><var name= "TEXT" expr= "TEXT"/ ></td></tr>
<tr><td>ATTR</td><td>::=</td><td>See below for possible attributes</td></tr>
</table>

<block>

A form item that contains executable code that will be executed if the block's form item is undefined and the cond attribute evalutes to true. Blocks are typically only executed once per form invocation.

[7] It is important to note that EBNF and ABNF are *not* the same. Look in Appendix E for more on this

Attributes

name	The name of the form item variable that determines if this block is to be executed or not. This will only happen if this variable is undefined. Defaults to an inaccessible internal variable.
expr	The initial value of the form item variable. By default this is undefined. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
cond	A boolean condition that must evaluate to true for the element to entered.

Legal parents

<form>

Legal children

<enumerate>, <exit>, <goto>, <if>, <prompt>, <submit>, <var>

<catch>

Containts the markup to execute when the specified event is thrown

Attributes

event	The event, or events, that will trigger the catch. A list of events may be specified, indicating that this element catches all the events named in the list. In case of multiple events, a separate event counter is maintained for each event.

Legal parents

$<$field$>$, $<$form$>$, $<$menu$>$, $<$vxml$>$

Legal children

<enumerate>, <exit>, <goto>, <if>, <prompt>, <reprompt>, <submit>

<choice>

Defines an item in a menu selection, forming an implicit grammar for that the menu.

Attributes

dtmf	The touchtone sequence for this choice. It is equivalent to a simple DTMF ¡grammar¿.
next	The URI this choice will transition to. Only one must be specified otherwise an error event is triggered.

Legal parents

<menu>

Legal children

<enumerate>, <grammar>

<if>

These elements are used for conditional logic, and they are processed if the specified condition evaluates to true.

Attributes

cond	A boolean test, that needs to evaluate to *true* for the section to be executed. If an <else> element is present, this will execute in all other instances, whereas <elseif> has a condition of its own to fulfill.

Legal parents

<block>, <catch>, <filled>, <help>, <if>

Legal children

<else>, <elseif>, <enumerate>, <exit>, <goto>, <if>, <prompt>, <reprompt>, <submit>, <var>,

<enumerate>

Vocalises the choices of a menu in the sequence they are given.

Legal parents

<block>, <catch>, <choice>, <enumerate>, <field>, <filled>, <help>, <if>, <menu>, <prompt>

Legal children

<enumerate>

<exit>

Ends a session, terminating all loaded documents.

<field>

A field specifies an input item to be gathered from the user.

Attributes

name	The form item variable in the dialog scope that will hold the result. The name must be unique among form items in the form.
expr	The initial value for the field variable, *undefined* by default If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
cond	An expression that must evaluate to true after conversion to boolean in order for the form item to be visited, or if the attribute is not specified.
type	The type of field, i.e., the name of a builtin grammar. Platform support for builtin grammar types is optional.

Legal parents

<block>

Legal children

<catch>, <enumerate>, <exit>, <filled>, <grammar>, <help>, <link>

<filled>

This element specifies an action to perform when some combination of input items are filled.

Attributes

namelist	A list of input that will trigger the action. This attribute may not be used if the <filled> element is inside a <field>

Legal parents

`<field>`, `<form>`

Legal children

<clear>, <enumerate>, <exit>, <goto>, <if>, <prompt>, <reprompt>, <submit>

<form>

Form items are the key component of VoiceXML documents, that represnts a single dialog.

Attributes

id	The name of the form, a unique identifier
scope	The default scope of the form's grammars

Legal parents

<vxml>

Legal children

<block>, <catch>, <field>, <filled>, <grammar>, <help>, <link>, <var>

<goto>

Transitions to a new dialog, either in the current document, or an external file. Can contain only one of *next*, *expr*, or *nextitem*.

Attributes

next	The URI of the dialog to transition to
expr	An ECMAScript expression that generates the URI
nexitem	The name of the next form item to visit in the current form

Legal parents

<block> <catch>, <filled>, <help>, <if>

Legal children

—

<grammar>

Provides a grammar that specifies the set of input that may be given.

Attributes

src	The URI specifying where to find an external grammar
scope	Defines the area where the grammar is active, either *document* or *dialog*.
type	The MIME type of the grammar. If none is given, the interpreter will try to determine this automatically.

Legal parents

<choice>, <field>, <form>, <link>

Legal children

—

<help>

Provides markup to execute when an utterance in the standard *help* grammar is matched. This element is a shorthand for <catch event="help">.

Attributes

count	How many times this event must occur before this element is entered. Default is 1.
cond	A boolean condition that must be satified for the element to be entered.

Legal parents

Legal children

<enumerate> , <exit> , <goto> , <if> , <prompt> , <reprompt>

<link>

Specify a transition common to all dialogs in the link's scope.

Attributes

next	The URI of the dialog to transition to
expr	An ECMAScript expression to dynamically generate the URI

Legal parents

`<field>`, `<form>`, `<vxml>`

Legal children

<grammar>

<menu>

A dialog for choosing amongst alternative destinations, may be seen as a form with a single field element, where the choices form the dialog's grammar..

Attributes

id	The name uniquely identifying the menu
scope	Defines the area where the grammar is active, either *dialog* (default) or *document*.
dtmf	If this is set to true, an implicit DTMF is made, based on the position of the choices.

Legal parents

<vxml>

Legal children

<catch>, <choice>, <enumerate>, <help>, <prompt>

<prompt>

Queue speech synthesis and audio output to the user

Attributes

bargein	Whether a user can interrupt a prompt. Default if true.
cond	An ECMAScript expression that must be satisfied for the prompt to be spoken.
count	The minimum of times this prompt for this item must already have been spoken before this particular prompt is used.

Legal parents

<block>, <catch>, <field>, <filled>, <help>, <if >, <menu>

Legal children

<enumerate> , <value>

<submit>

Submits the information collected to the document server.

Attributes

next	The URI to which the query is sent
expr	An ECMAScript expression that generates the URI to visit
namelist	The list of variables to send. All field item variables are sent by default, unless otherwise specified by this attribute

Legal parents

<block>, <catch>, <filled>, <help>, <if>,

Legal children

—

<value>

Inserts the value of an ECMAScript expression in a prompt.

Attributes

expr	The expression to evaluate

Legal parents

<block>, <catch>, <choice>, <enumerate>, <field>, <filled>, <help>, <if>, <menu>, <prompt>

Legal children

—

<var>

Declares a variable.

Attributes

name 5B	The name of the variable that will hold the result, and the scope of the variable is determined from the position in the document at which the element is declared.

Legal parents

<block>, <catch>, <filled>, <form>, <help>, <if>, <vxml>

Legal children

—

<vxml>

Top-level element in each compliant VoiceXML document.

Attributes

version	The version of the VoiceXML of this document. This field is required.
xmlns	The designated namespace for VoiceXML (required). The namespace for VoiceXML is defined to be `http://www.w3.org/2001/vxml`.
application	For multiple-document applications, this is the URI of root document.

B VoiceXML

B.1 Root document

```
<?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">
 <!-- Enables the user to return to the menu (#main) at any time -->
 <link next="kai.vxml#main">
  <grammar>[main menu]</grammar>
 </link>

 <!-- Welcome message -->
 <form id="intro">
  <block>
   <!-- Can also play a prerecorded message here -->
   Welcome to Foo Automatic Listing Service.
   <goto next="#main"/>
  </block>
 </form>

 <!-- If caller wants to make multiple queries, this is the reentry point -->
 <menu id="main">

  <prompt>
   Please choose your service: <enumerate/>
  </prompt>

  <choice dtmf="1" next = "residential.vxml">residential </choice>
  <choice dtmf="2" next = "business.vxml">business</choice>
  <choice dtmf="3" next = "phone.vxml">reverse lookup</choice>

  <help>Please choose <enumerate/>, or say repeat to repeat your choices</help>
  <catch event="repeat nomatch noinput"> <reprompt/> </catch>
 </menu>

 <!-- The generated vxml points here -->
 <form id="again">
  <field name="cont" type="boolean">
   <prompt>
    Do you want to make a new lookup?
```

```
   </prompt>
  </field>

  <filled>
   <if cond="cont">
    <goto next="#main"/>
   <else/>
    Thank you for calling, have a nice day
    <exit />
   </if>
  </filled>
 </form>
</vxml>
```

B.2 Residential lookup

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

<!-- Collects the information needed for searching the database
     $Id: residential.vxml,v 1.8 2004/03/16 13:50:31 kait Exp $  -->

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

 <form id="residential">
  <block>
   Please provide the information neccessary to complete the search
  </block>

  <field name="first">
   <grammar src="compiled:grammar/ogc-935678538"/>
   <prompt>
    The first name of the person you are looking for
   </prompt>
  </field>

  <field name="last">
   <grammar src="compiled:grammar/ogc-935678538"/>
   <prompt>
    The last name of the person you are looking for
   </prompt>
  </field>

  <field name="str">
   <grammar src="compiled:grammar/ogc-138932948"/>
   <prompt>
    Street
   </prompt>
  </field>

  <field name="city">
   <grammar src="compiled:grammar/ogc-138932948"/>
   <prompt>
    City
   </prompt>
  </field>
```

```

  <filled>
   <var name="type" expr="'R'"/>
   <submit next="../php/db2.php" namelist="type first last str city"/>
  </filled>
 </form>
</vxml>
```

B.3 Business lookup

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

<!-- Collects the information needed for searching the database
     $Id: business.vxml,v 1.4 2004/04/30 13:04:58 kait Exp $  -->

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

 <form id="intro">
  <block>
   Please provide the information neccessary to complete the search
   <goto next="#compmenu"/>
  </block>
 </form>

 <menu id="compmenu">
  <prompt>
    Do you want to search by person or company
  </prompt>
  <choice dtmf="1" next="#person">person</choice>
  <choice dtmf="2" next="#company">company</choice>
 </menu>

 <form id="person">
  <field name="first">
   <grammar src="compiled:grammar/ogc--935678538"/>
   <prompt>
    The first name of the person you are looking for
   </prompt>
  </field>

  <field name="last">
   <grammar src="compiled:grammar/ogc--935678538"/>
   <prompt>
    The last name of the person you are looking for
   </prompt>
  </field>

  <field name="city">
   <grammar src="compiled:grammar/ogc-138932948"/>
   <prompt>
```

```
    City
   </prompt>
  </field>

  <filled>
    <var name="type" expr="'B'"/>
    <submit next="../php/db2.php" namelist="first last city type"/>
  </filled>
 </form>

 <form id="company">
  <field name="company">
   <grammar src="compiled:grammar/ogc--935678538"/>
   <prompt>
    Company name
   </prompt>
  </field>

 <field name="city">
   <grammar src="compiled:grammar/ogc-138932948"/>
   <prompt>
    City
   </prompt>
  </field>

  <filled>
    <var name="type" expr="'B'"/>
    <submit next="../php/db2.php" namelist="company city type"/>
  </filled>
 </form>
</vxml>
```

B.4 Reverse lookup

```
<?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" xml:lang="no" xmlns="http://www.w3.org/2001/vxml">
<!-- type="phone" does not work and must most probably be interchanged
     with a new grammar fitted for this applications specific needs -->

<form id="rev">
 <field name="phone_no">
  <prompt>
  <grammar src="compiled:grammar/ogc-1791348435"/>
   Please provide phone number.
  </prompt>

  <filled>
   <submit next="../php/db2.php" namelist="phone_no"/>
  </filled>
 </field>

</form>
</vxml>
```

B.5 VoiceXML document generated by PHP

This is the resulting VoiceXML document if inserting these parameters into a residential query: first name "kit", last name "walker", street "fastlane", and city "fooville".

```
<?xml version="1.0"?>
 <!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
    "http://cafe.bevocal.com/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0"
      application="http://patrician.rexta.net/~kait/vxml/kai.vxml"
      xmlns="http://www.w3.org/2001/vxml"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:bevocal="http://www.bevocal.com/">

 <form>
  <block>
   There where 1 hit
  </block>

  <block>
   Kit Walker, Fastlane 83, 1000 Fooville, with phone number 555-83389
  </block>

  <block>
   <goto next="http://patrician.rexta.net/~kait/vxml/kai.vxml#again"/>
  </block>
 </form>
</vxml>
```

C PHP

PHP (now a recursive acronym for “PHP Hypertext Preprocessor”, but originally “Personal Home Page Tools”), is a widely used open-source programming language used primarily for server-side applications.

C.1 Database query

The PHP code contains several *print*-lines. These are neccessary to create the VoiceXML document that are sent back as response to the query that results from the interaction between the user and the application. To make the code prettier, it is possible to create a PHP-module that one can include into the code, as is done in the case of the database-connection.

```
<?php
 print ("<?xml version=\"1.0\"?>\n");
 print ("  <!DOCTYPE vxml PUBLIC \"-//BeVocal Inc//VoiceXML 2.0//EN\"\n");
 print ("  \"http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd\">\n");
 print ("\n");

 print ("<vxml version=\"2.0\"\n");
 print ("  application=\"http://patrician.rexta.net/~kait/vxml/kai.vxml\"\n");
 print ("  xmlns=\"http://www.w3.org/2001/vxml\"\n");
 print ("  xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"\n");
 print ("  xmlns:bevocal=\"http://www.bevocal.com/\">\n");

 include '/home/kait/hf/code/db/connect_pg.php';

 $dbh = @connect_pg("fooListing");
 $stm = createStm();
 $sth = pg_query( $dbh, $stm)
              or die ("Statementhandle error: " . pg_last_error($dbh));
 $hits= pg_num_rows( $sth );

 print ("  <form>\n");
 createReply ( $hits, $sth, $_GET["type"] );
 print ("  </form>\n</vxml>");

 function createStm () {
   $first = $_GET["first"]; #First name
   $last  = $_GET["last"];  #Last name
   $str   = $_GET["str"];   #Street
   $h_no  = $_GET["h_no"];  #House number
```

```
  $zip   = $_GET["zip"];   #zip-code
  $city  = $_GET["city"];  #City
  $phone = $_GET["phone"]; #Phone number
  $type  = $_GET["type"];  #Type of record business, residential or reverse
  $comp  = $_GET["company"]; #Company name

  #Build query
  if ( $type == "R" ) {
   $query = "SELECT first_name,last_name,street_name,house_no,zip,city,phone_no
           FROM simple WHERE ";
  } elseif ( $type == "B" ) {
   $query = "SELECT first_name,last_name,company_name,phone_no FROM simple WHERE ";
  }

  $query_elements = array();

  # builds a query from existing elements
  if($first) { $query_elements[] = "first_name ILIKE '$first'"; }
  if($last) { $query_elements[]  = "last_name ILIKE '$last'"; }
  if($str) { $query_elements[]   = "street_name ILIKE '$str'"; }
  if($h_no) { $query_elements[]  = "house_no ILIKE '$h_no'"; }
  if($zip) { $query_elements[]   = "zip='$zip'"; }
  if($city) { $query_elements[]  = "city ILIKE '$city'"; }
  if($phone) {
    $f = substr($phone,0,3);
    $l = substr($phone,3,5);
    $query_elements[] = "phone_no='$f-$l'";
  }
  if($type) { $query_elements[]  = "user_type ILIKE '$type'"; }
  if($comp) { $query_elements[]  = "company_name ILIKE '$comp'"; }

  $tmp = implode(" and ",$query_elements);
  $query = $query.$tmp;
  return $query;
}

function createReply ( $hits, $sth, $type ) {
  # To display the result
  if ($hits > 1 ) {
    writeBlock ( "There were $hits hits" );
  } elseif ($hits == 1) {
    writeBlock ( "There was $hits hit" );
  } else {
```

```
    writeBlock ( "No hits" );
  }

  if ( $hits > 5 ) {
    writeBlock ("Too many hits to read them all");
  } else {
     if ($type == "R" ) {
    while ($row= pg_fetch_array( $sth, NULL, PGSQL_NUM )){
             $name = "$row[0] $row[1]";
             $addr = "$row[2] $row[3], $row[4] $row[5]";
             $phon = "$row[6]";

             writeBlock ( "$name , $addr with phone number $phon" );
             $no++;
        }
     } else {
        while ($row= pg_fetch_array( $sth, NULL, PGSQL_NUM )){
         $name = "$row[0] $row[1]";
     $comp = "$row[2]";
         $phon = "$row[3]";

         writeBlock ( "$name , working for $comp with phone number $phon" );
         $no++;
        }
    }
  }
  writeBlock (" <goto next=\"http://patrician.rexta.net/~kait/vxml/kai.vxml#again\"/
}

 function writeBlock( $msg ){
   print ("    <block>\n");
   print ("      $msg\n");
   print ("    </block>\n");
 }
?>
```

D Grammars

D.1 Name.gram

```
;GSL2.0
NAME
[
 [(?FName LName)]
 [(FName ?LName)]
 [CName]
]

FName
[
 [ jack jill ron bob barbara kit eddie cindy hermione harry
   anthony nemo rob evil ronald bill sue drew peggy]
]

LName
[
 [ smith potter jones foo walker scotch tape owen meany
   kenievel johnson robson keegan smith baggins holmes]
]

CName
[
 [ foorox foosoft easyfoo acme (acme pizza) fooburger mcfoo fooking ]
]
```

D.2 Addr.gram

```
;GSL2.0
ADDR
[
 [(STREET ?CITY)]
 [(?STREET CITY)]
]

CITY
[
 [ fooville foocreek hogwarts barville foobarcity barking mad ]
]

STREET
[
 [ bourbonstreet highstreet fastlane (tranquil gardens) (grimauld place)
   (company lane) foostreet ]
]
```

E Abbreviations

This list of abbreviations and explanations is by no means complete, but it will hopefully be of some help trying to navigate through this paper.

ABNF Augmented Backus-Naur Form extends the basic Backus-Naur Form, and is documented in RFC 2234.

ASR: Automatic Speech Recognition

BNF: Backus-Naur Form is a metasyntax for formally describing formal languages, i.e. to express context-free grammars.

DTMF: Dual Tone Multi Frequency, the standard set of tones produced by the keys on a telephone handset

EBNF: Extended Backus-Naur Form is any variation on the basic Backus-Naur Form (BNF) notation used to describe the syntax of languages with the help of the following constructs: "[...]" for optional items, "*"-suffix to denote *Kleene closure* (zero or more repetitions of an element), "+"-suffix for one or more repetitions and curly brackets enclosing a list of alternatives. Super- or subscripts can be used to indicate a range of repetitions, i.e. between n and m occurences o f an element. EBNF is defined in ISO 14977.

ECMAScript: A standard script format defined by the European Computer Manufacturers Association. It is described in ECMA-262, which can be found here: `http://www.ecma-international.org/publications/standards/ECMA-262.HTM`

FIA: Form Interpretation Algorithm, described in detail in section 2.1.

GSL: Grammar Specification Language, Nuance's proprietary format for defining grammars to be used in voice-applications.

IS: Information System, typically a database system.

IVR: Interactive Voice Response systems are computerised systems that allows a person to select an option from a voice menu and otherwise interact with a computer system by voice, usually by means of a telephone.

Multi-modal: Combined simultanous text and speech input in a system

PHP: is now a recursive acronym for "PHP Hypertext Preprocessor", but originally "Personal Home Page Tools". This is a widely used open-source scripting language used primarily for server-side applications

PostgreSQL: is an free object-relational database server.

SRGS: Speech Recognition Grammar Specification

TTS: Text To speech

URI: Uniform Resource Identifier, a unifying syntax for the expression of names and addresses of objects on the network as used in the World Wide Web. Details can be found in RFC 2396

URL: Uniform Resource Locator is a standardised address for some resource, for example a document or image, on the Internet. The current form are detailed in RFC 2396.

References

[BM01] Bruce Balentine and David P. Morgan. *How to Build a Speech Recognition Application*. EIG Press, 2001.

[DD03] Korry Douglas and Susan Douglas. *PostgreSQL - A comprehensive guide to building, programming, and administering PostgreSQL databases*. Sams Publishing, 2003.

[Lou97] Kenneth C. Louden. *Compiler Construction - Principles and Practice*. PWS Publishing Company, 1997.

[Set96] Ravi Sethi. *Programming Languages - Concepts and Constructs*. Addison-Wesley, 1996.

[Shn98] Ben Shneiderman. *Designing the User Interface - Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman, 1998.

[ST03] David Sklar and Adam Trachtenberg. *PHP Cookbook*. O'Reilly, 2003.

[W3Ca] World Wide Web Consortium W3C. Speech Recognition Grammar Specification Version 1.0. Website. `http://www.w3.org/TR/speech-grammar/`.

[W3Cb] World Wide Web Consortium W3C. VoiceXML Standard Version 2.0. Website. `http://www.w3.org/TR/2004/PR-voicexml20-20040203/`.