A Widget-Based Approach for Creating Voice Applications

Andreas Hartl
Telecooperation Group
Darmstadt University of Technology
Alexanderstrasse 6
Darmstadt, Germany
andreas@tk.informatik.tu-darmstadt.de

ABSTRACT

Voice based applications nowadays are difficult to author compared to conventional ones; yet there is an increasing need for such applications in mobile environment. A reason for this difficulty is that application developers must tackle with relatively low level programming interfaces for voice based applications. This paper introduces *"audio widgets"* – a notion of higher level building blocks for such applications that ease modularizing and re-use.

1. INTRODUCTION

Voice based applications are an important building block of future ubiquitous and mobile computing environments. The reason is that such applications may be used in situations where hands and eyes must be free to use – e.g. while driving a car – or that the device is too small to support display and/or keyboard. Our research group is currently developing such a device designed to be small enough to fit into a headset [8].

While the concepts of interfaces based on windows, icons, menus and pointers (*WIMP-interfaces*) are widely known and adopted by programmers, there is no such generally adopted higher-level paradigm for voice based user interfaces. Instead, developers have to deal with various elements. For handling input, developers must deal with the grammars supplying models to understand the users' utterances. In most cases, context-free grammars (CFGs) are used for this purpose which consist of hand-crafted rules derived from the developer's inherent knowledge of the language or from corpus-based linguistic knowledge. Especially n-gram based grammars may be a powerful alternative in cases when the word order of the input is irrelevant. The output of voice based applications usually consists of either pre-recorded sentences or of textual data converted into audio via a text-to-speech engine. For example whether there should be acoustical clues for the users or whether the output should be structured, developers must implement such solutions for themselves.

One of the downsides of these approaches for voice based applications is that re-usability of grammars is very limited. Developers are likely to re-implement most of the grammar if they encounter similar problems in different applications. Another pitfall is internationalization which may require large part of the user interface handling code to be rebuilt if other languages do not resemble the original language closely.

This paper presents an approach that implements the notion of widgets from GUIs to audio based user interfaces. Whereas the term widget originally is targeted towards window gadgets, we conceive it to be much more general, as a high level building block of user interfaces. The idea of "audio widgets" therefore is a generalization of the idea of graphical widgets; in both cases widgets are higher level building blocks that abstract the core functionality of the interface (drawing pixels or creating grammars).

2. EXISTING SOLUTIONS

2.1 Context Free Grammars

Most solutions commercially available support context free grammars. Rules for such grammars usually are authored in a BNF-like notation and may include references to other rules, text literals and variable names as well as designators for optional and repeatable elements.

When authoring a rule one can use the following techniques: sequences for elements that occur after another, alternatives, optional elements and iterations of elements. Often also some form of recursion is allowed.

The Java Speech API Grammar Format [10] (JSGF) is a platform and vendor independent textual representation of CFGs. It targets only voice input as the application may use an arbitrary Java API for generating the output. The Java Speech API provides means for loading and deleting grammars into a speech recognizer and the ability to create a grammar at runtime. JSGF grammars provide a set of rules which may be activated and deactivated once loaded. Each rule must have a unique name in order to reference it. In order to prevent naming conflicts, rules may be put into packages which provide separate namespaces. A rule may contain literal text, or reference to other rules. When using references right recursions are allowed. Programmers of the JSGF define the grammars in their application and apply

them as necessary. They receive events once a particular grammar matches; programmers are free to interpret and handle these events as needed.

While JSGF extends the Java platform by additional means of input and output, VoiceXML [12] has a broader view towards voice based applications, it also targets output and navigation. VoiceXML applications are similar to web based applications: they are created by a server-side scripting technology and then interpreted by a *Voice Browser* which combines voice recognition and text-to-speech engines. VoiceXML applications consist of several forms that contain audio output and may define fields for entering data. VoiceXML provides several pre-defined fields for often used input elements (e.g. free form text, numbers, selections), developers can also use grammars to create their own fields. For navigation, VoiceXML forms can use URLs to link to others; also there is a special construct for menus that automatically creates the grammar necessary to select an entry. Currently VoiceXML only supports CFGs to an extent similar to JSGF and a grammar format for tone dial phones, although there are attempts to allow other grammars as well [11].

2.2 N-Grams

There are several types of statistics based grammars, the most widely used are N-Grams. N-Grams use a matrix that describes the probability of any word being entered based on an arbitrarily sized vector of previously recognized words. Compared to CFGs, n-gram based grammars are rarely used in commercial products.

N-grams are difficult to author as they usually are created using a sample text that is analyzed for its probabilities. This means that the text must be representative of future inputs. If an n-gram does not prove to be useful, one must alter the sample text and re-analyze it which leads to long development times when compared to CFGs. Yet there are some methods that ease creating n-grams either by using intermediate representations of the sample text, e.g. CFGs [4] or by using local n-grams which are merged together [9].

3. AUDIO WIDGETS

The idea of audio widgets was developed as part of our group's MUNDO project [5]. One of the goals of this project is to deliver a basic device that can operate hands- and eyes-free so that its user can focus on the task she is doing. Additionally, the device should work both in a disconnected and an online environment, with the ability to enhance its built in speech recognition by a server-based one if it is online.

We found that this was difficult to achieve using an approach based only on grammar descriptions for several reasons:

- Application designers must make a tradeoff between responsiveness and robustness of the system (the less rules the better) and ease of use (the more rules the better). In case of a dynamic environment which may change from online to disconnected mode and back again, the rule sets for both modes may differ strongly because of the limited capabilities of the built in recognition engine.

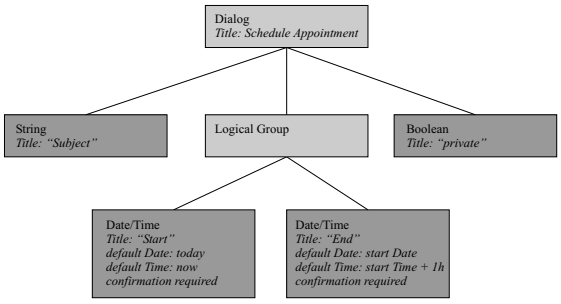

Figure 1: The logical tree of a dialog for a time scheduling application

- If a recognition engine has special features (e.g. n-gram grammars), developers must explicitly write code using these features. Therefore, an upgraded engine may have little to no impact on existing programs.
- MUNDO will allow users to associate other devices to the basic device. Such other devices may feature additional output capabilities such as displays. If developers wanted to use such multimodal environments with grammar based applications, they would have to write an additional graphical user interface by hand.

In order to circumvent these issues, we use audio widgets as an higher level representation of the developer's intent. When using widgets, one must not change the application but only the widget instance to adapt to changes of the underlying recognition engine. Additionally application developers do not have learn how to author different grammars as this task is delegated to the widget developer.

3.1 Architecture

MUNDO differentiates between the *logical* and the *physical* user interface. Application developers specify the logical content using abstract widget items that form a tree. The tree contains the data types to be input and output as well as meta-data describing e.g. the title of the element, its priority and recommended presentations of the widget.

Figure 1 shows a tree for the logical user interface for scheduling appointments. The tree's leaves specify logical widgets that define the data type to be entered and the meta-data necessary to identify the widget. The title may be rendered into prompts, can be used to disambiguate input data (*"Set start date to tomorrow"*), etc. Other meta-data entries define default elements and whether the content should be entered explicitly or whether the user just needs to acknowledge precalculated data. The logical tree also serves as a means for structuring widgets into related elements. In the above figure, the date/time entries are more closely related to each other, so they are contained inside a logical group. The software that transforms the logical into the physical tree uses the grouping as a hint that it should place the widgets in the same hierarchy level.

At runtime we transform the tree for the logical user interface into a tree of concrete widgets based on the knowledge of the device. Such a transformation may map one or more

logical widget to one physical widget and it may flatten or raise the hierarchy of the tree (c.f. [1, 2]).

Such a layer of indirection has been proven useful in the past [6] – a device is not required to support all logical widgets (as many of them may be mapped to one physical widget) thus saving space which is a crucial point for embedded devices. The varying hierarchy allows easy access to all information in cases when the environment permits it – e.g. if the server side speech recognition can resolve ambiguities – and fast access to crucial information in limited environments at the cost of users having to traverse additional hierarchy levels for less important information. Also, this architecture can be easily adapted to support additional modalities. For example, in order to render a logical widget onto a GUI, the GUI's own widgets can be used as physical widgets.

Finally, the runtime environment creates the concrete representation of the widgets specific for the environment out of the physical widget. For voice based applications, input widgets are converted either into CFGs of different grammar formats or into other grammar specifications available within the environment; output widgets typically are transformed into wave audio using text-to-speech (with parameters set by the widget) or prerecorded audio. Physical widgets are transformed into the appropriate representation as near to the device as possible – in most cases this will be done on the device itself.

3.2 Developing Widgets

Disconnecting the design of grammars from the design of audio widgets also means, that there is a need for a set of standard widgets. Also, it should be possible to create new widgets as special needs for inputting and outputting data arise. The design of physical and logical widgets should be comparable, even it the latter ones deal with higher level abstractions. For logical widgets, developers must provide a mapping to the appropriate physical widgets, for physical ones a mapping to grammars and parameters of the speech synthesis.

Although developers can develop entirely new widgets just by creating new classes, it is advisable to subclass already existing ones. In most cases, the development of new widgets will result in adding additional constraints to already existing ones. By constraining widgets, one can easily model affordances into the user interface while utilizing the power of the general purpose widgets: they are fine tuned to the execution environment and abstract from the grammar used – in a mobile environment a widget can consist of a CFG minimizing the recognizer's complexity, in server based environments it may use a more powerful recognition engine without further modifications.

A likely candidate for subclassing is the *select one out of many list* which presents its users a set of mutually exclusive choices. In the general case, this list should support arbitrarily long lists which means that, when prompting the user, it only gives away its title. To create a logical widget for entering booleans, a developer can subclass the list and make the following modifications:

- Create a prompt that indicates that this is a question with the answers *"yes"* and *"no"*
- Fill the list with standard items that are possible answers. This may include items such as *"yeah"* and *"nope"*
- Provide a mapping from the list item selected by the user to the appropriate boolean value

By using a list instead of creating an entirely new widget, this new boolean widget benefits from various implementations of the existing widget.

3.3 Prototype

We have developed a prototype that implements a dialer functionality for a Voice over IP application using audio widgets. Currently, it uses only two types of widgets: a descriptive text that is rendered by a text-to-speech synthesizer and the "1 out of many list" mentioned above. The descriptive text is used in the prototype whenever the application wants to create a prompt for the user, the list contains the names of all members in our research group.

The prototype is written in Java and uses IBM's JSGF implementation based on ViaVoice. As a text-to-speech engine, it uses the default ViaVoice implementation without any modifications. For input recognition, it converts widgets into context free grammars. Currently there exists a 1:1 mapping from logical to physical widgets, although we have a graphical representation of logical widgets as well for debugging purposes. We are now working on a second widget-set for restricted environments such as the hands free device mentioned above.

The grammar of the list widget is created in such a way that users may say only a part of any list entry to select it, e.g. the first name. This may lead to ambiguities which the widget can deal with in a limited way. If a user's utterance affects several items, the widget creates a new grammar that allows only selecting one of the ambiguous items and prompts the user to resolve the ambiguity.

For comparison we have designed a similar user interface using VoiceXML. We have experienced that VoiceXML is efficient if the full content of a list item is necessary to select it; in such a case a `<menu>` element is sufficient. Depending on the voice browser, the `<menu>` element may even support selecting an item by DTMF tones. Yet, if one wants to allow ambiguities such as the widget-based application, the developer must write grammars similar to those automatically created by the widget. In that case the VoiceXML solution is notably bulky compared to the widget based prototype.

3.4 Future Work

As a next step, we plan to research basic building blocks of voice based applications and develop widgets after them. The work already done in this area for graphical user interfaces [3] may ease this task. Currently we have identified the following often used types of input widgets: free form input that pass the recognized input to the application without further modification, an audio input widget that simply

records data, a list that lets users select one entry out of many, a list that allows multiple selections, a boolean input field for yes/no questions, an input field for numbers and a widget for selecting the date and/or the time.

Especially the date/time widget must deal with various types of input data: Not only absolute input like *"December, 24th, 2004"* but also with relative input *"tomorrow after the meeting"*. In order to correctly recognize the utterances, audio widgets must therefore have better knowledge of the environment they are deployed in than graphical ones. An easy way for achieving this is to give widgets the possibility to let widgets access the application's logical tree at runtime.

For the basics of outputting data, we are going to use audio output widgets that play back audio files and labels which are rendered by text-to-speech. Additionally, we are planing to deploy a widget for structured audio output [7] which allows browsing through larger amounts of text. We have yet to investigate if developers can create applications featuring ambient audio using any output widget and simply use it as a background or if a specialized ambient audio widget will be necessary.

As usual for user interface based research, it also will be necessary to make extensive usability tests (c.f. [13]). We are currently building a small set of audio based devices which support basic personal digital assistant functionality; the dialer prototype being one of its functionalities. Other applications will be accessing the personal calendar and scheduling new appointments, creating personal notes and listening to streaming audio. We plan to equip students with these devices and let them try to accomplish predesigned tasks; with a field test following where students should use the devices to ease their daily life at the university.

4. CONCLUSION

We have introduced an approach that introduces the concept of widgets to audio based applications. This creates a layer of abstraction between the application and the type of grammar used by the recognition engine thus enhancing portability and easing the process of developing the application.

We introduced trees of logical widgets which represent the data to be input and output along with meta- data augmenting the logical widgets. We described how these logical trees are transformed to trees of physical widgets based on the capabilities of the user's devices; the physical widgets then are transformed into the lower level representation required by the device.

We showed an example of how widget developers may use already existing widgets and subclass them in order to introduce additional constraints to the interface. We also showed a prototype application based on basic audio widgets and compared it to an application created using VoiceXML.

Finally, we have identified the main challenges of further developing audio widgets which are finding out the core widgets and evaluating user interfaces based on audio widgets.

5. REFERENCES

[1] J. Eisenstein and A. R. Puerta. Adaptation in automated user-interface design. In *Intelligent User Interfaces*, pages 74–81, 2000.

[2] J. Eisenstein, J. Vanderdonckt, and A. Puerta. Applying Model-Based Techniques to the Development of UIs of Mobile Computers. In *Proc. of Intelligent User Interfaces*, pages 69–76, Jan 2001.

[3] H.-W. Gellersen. *Methodische Entwicklung flexibler interaktiver Software*. PhD thesis, Karlsruhe, 1995.

[4] J. Gillett and W. Ward. A language model combining trigrams and stochastic context-free grammars. In *5-th International Conference on Spoken Language Processing*, pages 2319–2322, 1998.

[5] A. Hartl, E. Aitenbichler, G. Austaller, A. Heinemann, T. Limberger, E. Braun, and M. Mühlhäuser. Engineering Multimedia-Aware Personalized Ubiquitous Services. In *IEEE Fourth International Symposium on Multimedia Software Engineering*, pages 344–351, 2002.

[6] A. Hartl, G. Austaller, G. Kappel, C. Lechleitner, M. Mühlhäuser, S. Reich, and R. Rudisch. Gulliver – A Development Environment for WAP Based Applications. In *The Ninth International World Wide Web Conference.Amsterdam, NL*, 2000.

[7] F. James. *Representing Structured Information in Audio Interfaces: A Framework for Selecting Audio Marking Techniques to Represent Document Structures*. PhD thesis, Stanford, 1998.

[8] M. Mühlhäuser and E. Aitenbichler. The Talking Assistant Headset: A Novel Terminal for Ubiquitous Computing. In *Microsoft Summer Research Workshop, Cambridge*, Sep 2002.

[9] A. Nasr, Y. Estève, F. Bechet, T. Spriet, and R. de Mori. A Language Model Combining N-grams and Stochastic Finite State Automata. In *Eurospeech'99*, volume 5, pages 2175–2178, 1999.

[10] Sun Microsystems. Java Speech Grammar Format Version 1.0. http://java.sun.com/products/java-media/speech/forDevelopers/JSGF.pdf, Oct 1998.

[11] W3C. Stochastic Language Models (N-Gram) Specification (W3C Working Draft). http://www.w3.org/TR/2001/WD-ngram-spec-20010103/, Jan 2001.

[12] W3C. Voice Extensible Markup Language (VoiceXML) Version 2.0. http://www.w3.org/TR/2003/CR-voicexml20-20030220/, Feb 2003.

[13] N. Yankelovich, G.-A. Levow, and M. Marx. Designing SpeechActs: Issues in Speech User Interfaces. In *CHI*, pages 369–376, 1995.