Automatic Generation of Speech Interface for GUI Tools/Applications using Accessibility Framework

Naveen Kumar, M Sasikumar
CDAC Mumbai
{naveenk, sasi}@cdacmumbai.in

Abstract

A number of Automatic Speech Recognition Engines exist today. Speech Recognition can be used to implement speech based interface for applications, by giving commands from menus for these GUI Applications. Though quality of speech recognition engines today are not enough for open domain, for application interface they can work well. This provides an effective solution for accessibility for those who are unable to read/write any language. The manner in which Speech Engine is integrated with any application is often through making some code modification in an application Specific way. Our idea is that, instead of embedding Recognition logic inside the application, a separate application handles the responsibility of recognizing and firing the spoken menu command to the GUI application. This third application makes use of available accessibility framework to access the widget hierarchy (Menus) of the GUI application. The GUI application must conform to the accessibility framework.

1 Introduction

Humans are the only species on this planet who have the capability to store and process information in an objective way. A written text in a notebook is a simple example of this objectivity. The advent of computers have further enhanced this human capability. We are now able to store and process information of immense magnitude than was possible earlier. With the increasing use of computers new disciplines of study have emerged which concentrate on making our interactions with computers more natural and intuitive. This discipline, commonly known as *Human Computer Interaction*, concentrates on all aspects of our interaction with a computer system more commonly a desktop system, in order to make it more usable and accessible.

One aspect of accessibility is concerned with controlling a *desktop* computer, in a natural manner, through speech. There is a natural enthusiasm in being able to correspond to the desktop through speech. This form of interaction is more familiar to most of the people, and the only option for those who are illiterate.

Desktop interaction through speech input/output is also useful for another category of people, who suffer from forms of motor disabilities. Such disabilities hamper them from using keyboard and mouse or any other pointing device effectively. Another category of users who can benefit from such a system are the people who suffer from vision related problems.

Although speech recognition looks to be a promising aspect of *HCI*, we are still limited by the limitations imposed by the current s*peech recognition engines*. These limitations can be attributed to the immense complexity of the s*peech recognition* problem itself. This difficulty in solving the s*peech recognition* problem stems from the inherent vagueness and imprecision of the *Natural Language* and the way people speak. This is also affected by the noise inherent in the natural surrounding.

Although completely solving *speech recognition* problem is an open problem, good recognition rates have been achieved for limited vocabulary size. This rate goes as far as 97%[1].

One question arises at this stage. Can we make use of limited vocabulary set for all interactions with *desktop*?

One can make use of such a limited vocabulary to control and simulate the actions of mouse or any other pointing device. There are problems with this approach. Firstly, it will require constant absorption of visual attention to see if

the mouse is moving in right direction or correct clicks are performed at right locations. Secondly, a blind user will find it difficult to use such a system, because he does not know where to locate a mouse. Alternatively such a blind user can be provided with *text-to-speech* feedback mechanism which will tell him the location of mouse.

Then, *how do we speech enable an application or a desktop*? The answer to this question is governed by two prerequisites.

1. We know what to speak.
2. We can access and generate events on all GUI components and windows.
3. We can recognise speech well for small vocabulary set.

We need access to the application or desktop as another *input method*. But using speech as a conventional *input method,* is difficult owing to the fact that there is no fixed *context* to speak. We simply do not know what to say to *do something*. Our *desktop* isn't smart enough to resolve the *context* and generate an *event* on the application. Besides we cannot trust our speech engine for that kind of open interaction.

But there is a catch here. We are able to see and read (or get feedback from system for) the user interface elements (also known as *widgets*). If these *widgets* have a string associated with them as an identifier, they can be used to generate *speech vocabulary*. So we do have a *context* for generating *speech vocabulary*.

We can create a *speech vocabulary* out of what is visible to user (e.g menu bar, toolbar etc). The application window, dialog or widget in focus is the current *context* for creating such a vocabulary.

This leads to another problem. *How do we access these application or widgets from the desktop?* Traditional way to solve this is to embed *speech recognition* code, using speech API, inside the application itself and look for specific utterances for each screen, menu, toolbar or any other context. This means that every application will handle *speech input* in its own way. Surely this is a problem because if multiple concurrent applications (which are using speech as input mechanism) are running simultaneously, not only they will have to be synchronized for *speech input* but also the speech library API may not be used in a coherent way. This might lead speech library to an inconsistent state. Also speech code and synchronization related aspects will have be repeated for each application.

Another way to solve this problem is that instead of embedding *speech recognition* logic inside the source code of an application, we use a library or framework which is allowed to access all the *widgets* and *applications* that are available on the *desktop*. This framework will help us to find, locate and generate events on currently visible *widgets*. This *widgets* will act as *speech context* for us. This library will help us create an auxiliary *Assistive Technology* (AT) application which will handle and process *speech input* for all *applications* on *desktop* and *desktop* itself.

The rest of the paper describes this second approach to speech enable a GNOME desktop making use of the limited vocabulary accuracy available, and our experience with this.

2 Approach

Our speech work has been generally centered around *Linux desktop* and *open source* environment. Therefore, the approach that we have taken, to speech enable a *desktop*, has been greatly influenced by the *Linux desktop* environments more specifically GNOME.

Our conceptual understanding of architecture is shown in Figure 1.

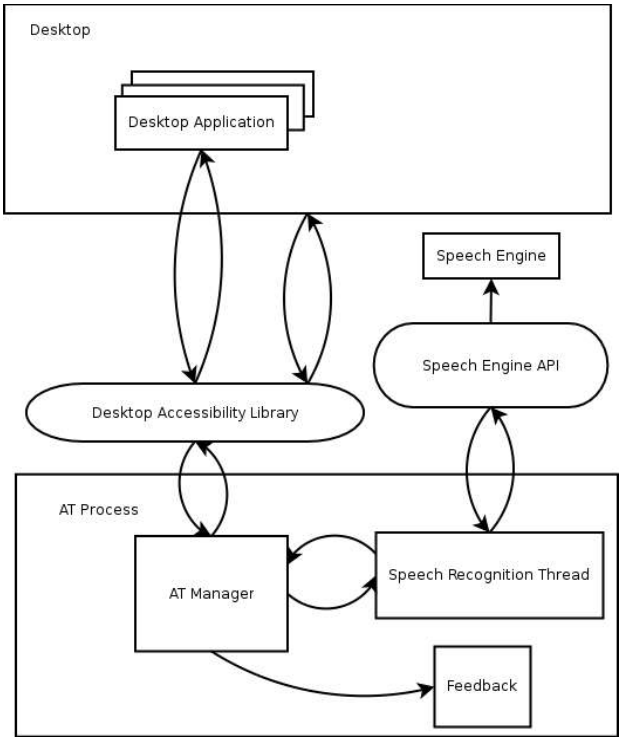

Figure 1 Conceptual Architecture

There are six important aspects to the approach/architecture proposed by us. These are:

1. Desktop Accessibility Library[4]
2. Speech Engine and API
3. Desktop and widget usability
4. Vocabulary building and Context
5. Dialog Management
6. Feedback

These are discussed individually below.

2.1 Desktop Accessibility Library

All GNOME desktop *applications/widgets* achieve *accessibility* by implementing toolkit independent *accessibility interface* known as *Accessibility Toolkit* (ATK). ATK describes a set of interfaces and API that must be implemented by all those GUI components and widgets which want themselves be accessible by different *Assistive Technology* (AT).

The GTK implementation of these interfaces are available in a module called GNOME *Accessibility Implementation Library* (GAIL). GAIL is dynamically loaded by GTK applications at runtime. Again the different ATs access these running applications through a toolkit independent Service Provider Interface library API called AT-SPI. The running applications export their *accessibility support* to AT-SPI via the relevant bridge.

If an application implements *accessibility interface* ATK, the AT-SPI library can be used to poke/dig the entire *widget hierarchy* of that application. AT-SPI provides primitives which allows an AT to perform actions on the *widgets*. AT-SPI also allows ATs to register for *accessibility* related *events* and perform actions on these events through their *listeners*.

Now as described in section 1 (Introduction), if *widgets* have a *string identifier* associated with them, our *context* creation task becomes easier. ATK also allows us to describe *roles* and *states* for individual *widgets*. The *roles* are a kind of string enumeration which describe what role a particular widget plays in an application. Some examples of roles are *application*, *panel*, *scroll bar*, *menu bar*, *menu*, *list*, *list item* etc. The *states* are also a kind of string enumeration which describe the *in-process* current state of the *widget*.

2.2 Speech Engine and API

A *speech engine* provides us with means and API to perform various forms of *speech recognition* task based on a trained vocabulary. We use CMU *Sphinx-4*[3] *speech recognition engine*. *Sphinx-4* is a state-of-the-art *speech recognition system* written entirely in the Java™ programming language. This *speech recognition system* is available as a *Live mode* system. *Live mode* is referred to a system which is ready to be used *off the shelf* for *speech recognition* purpose. The *Live mode* API can directly be used to integrate its *speech recognition* capabilities with another application. Some of the default *acoustic models* (e.g HUB4, WSJ20K, etc.) available with this system are quite large and *speaker independent* to a great extent.

Availability of such a large *speaker independent* acoustic model ensures that for most of the words required for the vocabulary of the GNOME desktop, no such acoustic training would be required. Still if such a training be required for languages other than english, it can be done by

using the acoustic training facilities available with Sphinx. But such acoustic training cannot be done by naive users and specialized speech researchers should do it on behalf of their community. Therefore we find that *speech localization* can be an issue here.

2.3 Desktop and Widget Usability

AT accesses the desktop using AT-SPI library. Once handle to the *desktop* is *accessible*, all applications and *widgets* are accessible to us. But for *speech recognition* purpose, not all *widgets* on an application on GNOME desktop is of interest to us. While invoking tasks through speech, there are a limited number of activities that we can perform on an *application widget*. For *widgets* which do not show all of the information at one time (e.g list, menu), we can perform tasks which expand their information or which allow us to navigate through such information. Some widgets are only *clickable*, while there are other *widgets* which allow us to edit text in them. Some just act as *containers* for other *widgets*. Therefore these are of no particular interest to us because they are either used for formatting purposes or to display static information(e.g labels). Therefore in our current approach we do not create any *context vocabulary* for these *widgets*.

For example, *widgets* of type panel are not useful to us. We can directly access *widgets* below it which act as visible container for other *widgets*. Another good example is a *ToolBar*, identifiable by role name "*tool bar"*. It is a container *widget* and can be used to access other useful *widgets* such as *cut*, *copy*, *paste*, *undo* buttons.

Figure 2 ToolBar Menu

In some standard applications the *ToolBar widget* do not have these buttons as direct descendants. Instead these buttons are contained in a container *Panel*, identifiable by role name "*panel"*. Now if we construct a vocabulary for *ToolBar context* these panels are not visibly identifiable and hence are rendered useless for our purpose. So we construct our vocabulary for the current *context* by going one level down the current hierarchy. The consequence of this is that our application must have the knowledge of all usable *widgets*.

Widgets which are of importance to *speech recognition* task specified here according to their role names are *application*, *menu bar*, *menu*, *menu item*, *radio menu item*, *check menu item*, *dialog*, *list*, *list item*, *page tab list*, *page tab*, *text*, *password text*, *push button*, *radio button*, *spin button*, *toggle button*, *combo box*, *tool bar*, *tree*, *tree table*, *editbar and entry*.

2.4 Vocabulary and Context

A vocabulary is for a c*ontext*. Vocabularies are phrases that can be invoked through speech, for recognition, by our AT at a particular instant of application's execution. In a particular *context*, our AT can execute some action on a particular *widget*, which may or may not cause a change in *context*. In most cases, our *context* is the direct manifestation of a c*ontainer widget* (exception *desktop context*), whose vocabulary points to *child widget* which itself can be *container* and hence another *context*. This *child context* can be accessed by invoking, that *speech phrase*, in parent *context*, which corresponds to it and causes parent to transfer the control to a *callback*.

These actions can range from simple click of the *widget* to making a call to a *callback* function, specialized to handle actions in case of a *context* switch. These *callbacks* must be registered and accessible to AT.

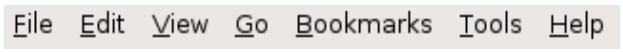

Figure 3 Menu Bar

For example at any point of time of execution suppose our AT holds a *MenuBar context*, identifiable by the role name *menu bar*, the vocabulary available to the current *context* are *File*, *Edit*, *View*, *Go*, *Bookmarks*, *Tools*, *Help*. Now an invocation of *File* from the vocabulary will cause the file menu to expand (Figure 3) and the *context* is transferred from *MenuBar context* to *Menu context* identifiable by the role name *menu*.

Figure 4 File Menu

The *Menu context* is a direct manifestation of *File Menu* (Figure-4) and it's children *widgets*. The current Vocabulary of *Menu context* are the child *widget* names which are *New Window*,...,*Quit*. Each of these are "*menu item" widgets* which are not *components*. Hence none of these are *sub-context* of this particular *Menu context*.

Apart from usual vocabulary of children in a *context* the present *context* must also hold a *context* to it's parent. This is a mechanism to return to the parent *context*.

Another example of specific *context* handling is when a widget is encountered with a *state EDITABLE* the callback function calls a *Predictive Text Interface* which is used to edit/write text. This *Predictive Text Interface* is itself to be controlled by *speech interface*.

It is also possible that for certain specific application or *widget contexts* we do not want a *dynamic vocabulary* to be generated. Such specifics can be handled in *callbacks* for these *contexts*, to provide a static *speech vocabulary*.

2.5 Dialog Management

Dialog Management is one of the most difficult aspects of this AT. Down the hierarchy of *speech contexts*, a certain invocation of a certain phrase causes a *dialog* to appear. This new *dialog* becomes the child of the Application (*Application context*) itself. Now if we follow the usual path of return, we have to trace our path back through *parent contexts* to *Application context* and then access the *dialog* from there. Of course there is an additional task of building new vocabulary for the *Application context*, owing to sudden appearance of this *dialog* child.

There is an alternative to create and register a *callback* for an *Object:children-changed* event on *widget* with Accessible role *application*. This *callback* will cause a suspension of all *context* and deallocation of all memory resources held up by the *application widget hierarchy*. The *callback* will reset the *context* of *speech vocabulary* to *Application Context* from where *Dialog context* can be easily accessed by another speech invocation.

One disadvantage of the current procedure is that the previous *vocabulary context* of the application is lost and user has to re-acquire that.

2.6 Feedback

The feedback mechanism provides the user with information of *in-context* vocabulary. This can be in the form of a simple application window showing possible *speech vocabulary* or in the form of some other *modality* like *text-to-speech*. One advantage this *Feedback* provides us is that user is aware of what is to be said at an instant to do certain action. This *Feedback* module will also act as the *point of presence* window for our AT. In layman terms users will know speech recognition framework is *in-process* and executing.

3 Architectural Design

Following things are clear in relation to our proposed approach, that:

1. We do not trust *speech recognition engine* with large vocabulary set.
2. AT-SPI API allows the auxiliary AT to probe *desktop* for applications, and *widget hierarchy* of a particular application. It also allows us to perform intended actions on these *widgets*.
3. Not all *widgets* are useful for us for invoking commands and controlling applications through speech.
3. A particular category of *widget* must be treated in a specific manner, for them to be helpful and effective in controlling the applications through speech invocations.

Following these *rules of thumb*, and our improved knowledge we modify our conceptual diagram shown in *Figure 1* as in *Figure 5* to include the specifics for GNOME desktop and Sphinx-4 *speech recognition engine*.

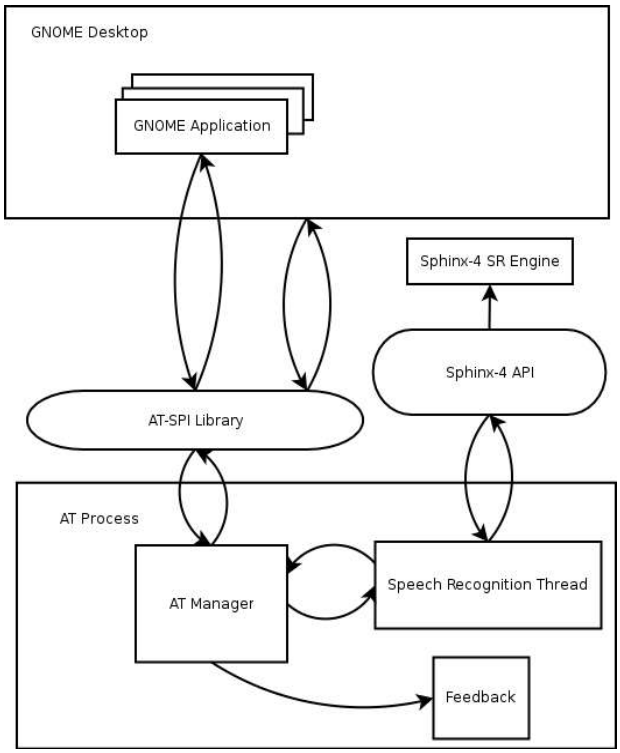

Figure 5 Architecture

Let us describe the major components in this diagram.

3.1 AT Process

AT Process performs the responsibility of initializing the AT Manager so that *desktop* is *accessible* to the AT.

3.2 AT Manager

AT Manager performs following responsibilities:

1. Initialize AT-SPI library to access GNOME desktop.
2. Initialize *speech thread* which recognizes speech invocations.
3. Register *callbacks* for events on particular *widgets*.
4. Listen to *speech events*.
5. Generate *events* on GNOME desktop or GNOME desktop application based on *speech events*.
6. Create and modify *context grammar*, which specifies *speech* vocabularies, using *Accessible* interfaces, based on new *context*.
7. Update *Feedback* module, of current *context* and *grammar*.

3.3 Speech Recognition Thread

S*peech Recognition Thread* performs following major responsibilities:

1. Listen and recognise *speech* invocations based on a *context grammar*.
2. Load new *speech grammar* based on changed *context*.
3. Notify *Listeners* of the *speech event*.

3.4 Feedback

Feedback updates users with current *context grammars* and exceptional conditions.

4 Our Work

Our work in speech enabling some *open source* applications formed the basis of this paper. We used *Sphinx-4 speech recognition engine* to speech enable applications.

There were portability and interface issues between Java™ based *Sphinx-4* API and applications written in C/C++. We solved this issue by using JNI to interface between them.

Initially we followed the traditional approach of speech enabling applications by modifying their source. We experimented with *pine email client* by modifying it's freely available source code, and were able to create a proof of concept system.

But this experience taught us what not to do and we adopted a more dynamic approach by using desktop accessibility API to speech enable applications. We were able to create another proof of concept system by implementing a part of second approach to provide speech interface to GTK compiled Mozilla Firefox browser. This paper generalises this work.

5 Conclusion

In this paper we presented ideas and architectures in speech enabling applications. Some of the ideas that we presented in this paper can also be used in other types of *Assistive Technology* solutions. These ideas can be adopted to various other devices and platforms to give a viable *accessible* solutions to solve different aspects of *accessibility*. As speech engines with better performance become available, the current architecture can be extended to locate the *speech contexts* in various other ways. We need not be dependent on what an application has to offer. We hope these ideas will help to bridge the digital divide in some way.

6 References

1. Alan Dix, Janet Finlay, Gregory D. Abowd, Russel Beale, "*Human-Computer Interaction*", Pearson Education, Third Edition.
2. Stephen Cook, "*Speech Recognition HOWTO*", http://www.faqs.org/docs/Linux-HOWTO/Speech-Recognition-HOWTO.html
3. Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, Joe Woelfel, "*Sphinx-4: A Flexible Open Source Framework for Speech Recognition*", http://cmusphinx.sourceforge.net/sphinx4/#whitepaper
4. *GNOME Accessibility Developer Information*, © 2003-2004,2007 by The GNOME Project, http://developer.gnome.org/projects/gap/, retrieved on November 26, 2007.