

A Fine Grained Component Architecture for Speech Application Development

Andrew Hunt and Willie Walker

A Fine Grained Component Architecture for Speech Application Development

Andrew Hunt and Willie Walker

SMLI TR-2000-86

June 2000

Abstract:

This paper introduces a new architecture for the design and development of speech user interfaces. In this design, the speech input and output capabilities are implemented using the smallest reasonable components and combining these components into sophisticated interfaces by hierarchical composition, cross-referencing, customization and extension. To define legal spoken input, a speech user interface component comprises any number of input expressions each of which concisely defines syntactic, semantic and activation constraints plus support for composition by cross-reference to other input expressions. Similarly, components contain output expressions with equivalent compositional capabilities for dynamic generation of spoken output. The component design follows the Model-View-Controller architecture. The architecture has been applied to the development of multi-modal desktop applications and speech-only dialog systems. The benefits of this fine-grained component-based architecture include simplification of speech application development, enhancement of speech user interface consistency by automatic generation of common patterns, and improved maintainability and readability of speech software. In analyzing the architecture, we explore some of the fundamental differences between the requirements to support graphical interfaces and speech interfaces.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:

william.walker@east.sun.com
andrew.hunt@speechworks.com

© 2000 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS

Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>. Reports in this series are also available online on the World Wide Web at <http://www.sun.com/research/>.

A Fine Grained Component Architecture for Speech Application Development

Andrew Hunt and Willie Walker
Sun Microsystems Laboratories, Burlington MA, USA

1. Introduction

Research towards the goal of spoken interactions between humans and computers falls into three broad areas. Firstly, computers must have the ability to process incoming speech and to produce speech output. This is the domain of speech recognition and speech synthesis research, fields which have been active for several decades. Secondly, we need to understand the human perspective of spoken interaction with computers. Research on human-computer interaction by speech is more recent but has brought about a better understanding of how and when to use speech and how to design around some of the limitations of the underlying speech technologies ([1], [2]). This field has drawn from other areas of user interface design and extensively from study of spoken language in linguistics and psychology. Finally, we need to understand the structures and principles of the computer systems that use speech recognition and synthesis for the construction of usable speech applications. It is in this third domain that we see limitations in both the research understanding and commercial practice of speech system design. In this paper we present a novel architecture for the design and development of speech user interfaces that embodies new ideas about how to construct speech interfaces. We reflect on both the pragmatic issues of building systems with this architecture and on some of the principles of building speech systems.

1.1 Background

Speech engines are those parts of the system that process incoming or outgoing speech signals. Specifically, a *speech recognizer* is an engine that converts incoming speech to text and a *speech synthesizer* is an engine that converts text to speech output. Speaker verifiers and classifiers form a third class of speech engine but are not used in this work.

While existing commercial and research speech engines are sufficiently powerful to implement sophisticated speech user interfaces, the engines still have weaknesses that limit deployment in many domains and cause applications to fall short of user expectations. Current speech recognizers are flexible, inexpensive and widely available but are not robust and accurate across application domains. Current speech synthesizers are understandable but produce mechanical-sounding speech that reduces user acceptance of synthetic speech output. In both cases it is very likely that ongoing improvements in speech engines will facilitate more sophisticated and usable speech interfaces.

The field of speech user interface design has been studying effective use of spoken language as a means of facilitating human interaction with computers. For computer input, the user interface design addresses what a user can say and when it can be said. For computer output, the design specifies computer prompts, feedback to users, and other spoken and non-speech output. In

combining input and output, user interface designs also consider dialog flow, timing, help, disambiguation, recovery for errors and more. While it is likely that there is much still to be learned about good speech user interface design, the existing body of knowledge is sufficient to build usable conversational systems.

Our observation is that the current systems for implementing the advanced speech user interfaces are too complex and require a high degree of expertise. Our research is motivated by a belief that a better understanding of how to construct speech applications will both simplify speech interface development and facilitate advances in speech user interface design.

1.2 Related Work

Much of the research and deployment of speech systems has focused on dialog systems in which voice is the only means of communication. Common evaluation paradigms such as the Air Travel Information System task [3] and, more recently, the DARPA Communicator project [4] have focussed the research community on specific tasks that have pushed the bounds of automated conversational interactions. For example, the MIT JUPITER system [5], on which the Communicator architecture is partially based, combines multiple capabilities to implement a telephone weather system: recognition and understanding of noisy natural conversational speech, processing of that speech to database queries, and conversion of query responses to natural language spoken output. The JUPITER system and others participating in the Communicator project are capable of holding long conversations (exceeding 50 turns), recovering from communication errors, and handling complex dialog phenomena such as ellipsis, ambiguity and mixed-initiative dialog.

These research systems are not intended for widespread deployment and the internal mechanisms for processing speech and dialog require tremendous expertise to understand and maintain. However, the experience of the advanced systems has been distilled into commercial systems and emerging standards that have less sophisticated conversational capabilities but are simpler and typically faster to develop. The SpeechObjects system [6] and the DialogModules system [7] are reusable software components that implement sub-dialogs such as the collection of phone numbers, credit card information and so on. The VoiceXML specification [8] and its predecessors such as VoxML and SpeechML take a complementary approach of authoring dialog systems in a textual markup language.

1.3 Objectives

Following the observation that current systems for implementing advanced speech user interfaces are too complex, our research objective was to investigate means for simplifying the development of speech applications. Based on current object-oriented design practice, we explored component-based designs with the intention of identifying and using the smallest, reasonable speech components and understanding the desirable and undesirable properties of these components. For instance, we wanted to produce an effective design pattern for speech components that would permit components to operate independently but also enable composition of components to produce sophisticated speech interfaces. From a deployment perspective, our goal was to provide speech application developers with an effective toolkit with which they may build effective end-user

applications. There were additional constraints to address the pragmatics of supporting deployed systems.

- ◆ *Simplify development of speech user interfaces.* The design should reduce development time, reduce cost, reduce requirements for developer training and expertise, enhance maintainability, and shield developers from unnecessary complexity.
- ◆ *Promote good speech user interfaces.* The design should facilitate rapid prototyping of speech interfaces, support iterative redesign of the interface based on user trials, and allow user interface designers without programming skills to build and modify those parts of the systems that directly impact the user experience. The design should embody good speech user interface principles including promoting the consistency of spoken interfaces, plus anticipation and recognition of what users are likely to say.
- ◆ *Embody common patterns.* Common sets of speech patterns (e.g., “set the *property* to *value*”), components for common objects (e.g., numbers, lists), common operations upon the components (e.g., turning components on and off as focus changes) and other recurring patterns should be built into the design to promote reusability, consistency and speed of development.
- ◆ *Support customization and extensibility.* Developers should be able to extend and customize the common patterns to meet specific application needs.
- ◆ *Support multi-modal and speech-only interfaces.* The architecture should be appropriate for the design and deployment of speech-only interfaces (e.g., telephone-based dialog systems) and multi-modal interfaces (e.g., desktop and small-device interfaces that combine speech and graphical interfaces).

Section 2 describes the fine-grained speech component architecture in detail. Section 3 analyses the architecture against the objectives and discusses aspects of the architecture including its use for key application classes, experiences from our implementation, some significant differences between graphical and speech user interfaces, and some of the tools that we have used with the architecture to facilitate the design and analysis of applications. In Section 4 we present our conclusions.

2. Architecture

2.1 Speech User Interface Components

In graphical environments, a component-based approach to building applications has proved continuously successful. The core graphical components are abstract entities that are rendered visually and which permit manipulation by users with a mouse and keyboard. For example, a button object is a virtual proxy for a physical button. The Model-View-Controller (MVC) design pattern [9] is commonly used to implement the graphical components. The Model represents the underlying application data, the View presents the data to the user, and the Controller allows a user to manipulate the Model.

In our architecture, we apply a component-based, MVC design pattern to building *speech user interface components* (for brevity we use the term *speech component*). As in graphical applications, the model continues to represent the application data and functionality. The view, which broadly speaking is the flow of information from the computer to the user, is implemented with speech output provided by speech synthesis technology (a view does not need to be visual). The controller, which handles the flow of information from the user to the computer, is implemented with speech input provided by speech recognition technology.

We have sought to identify and implement small, reusable speech components. For instance, the speech component for an integer allows a user to give commands to modify the integer value and to hear the integer being read back. If the integer represents a user’s age, then one reasonable input would be “*I am thirty years old.*” and one output might be “*Records show you are thirty years old.*”

In following sections, we describe the speech input and speech output systems implemented by *expressions* that define what the user may say to a component (for input) and what the computer may say to the user (for output).

Following that, we introduce the use of collections of speech components that work together in a hierarchy to permit sophisticated interfaces to be built by linking expressions of multiple speech components together. To implement linking, each speech component is required to have an identifier (we use a string) that is at least unique amongst its siblings in the component hierarchy — we refer to this as the *speech component name*.

2.1.1 Pre-built, Customized and Extended Components

As with most speech component systems, our system includes pre-built, generic, speech components. This is most conveniently represented by classes and inheritance in an object-oriented system (we have used the Java™ Programming Language [10]). The root class is a generic speech component class. Table 1 lists some of the classes that inherit from the root class.

Name	Description	Example
Action	Perform a specified action when directed.	Send email command.
Choice	Select amongst an enumerated set of options.	Turning on or off bold text in a text editor.
Value	Input/output for a named value. Value is sub-classed for booleans, integers, and floats.	Interest rate in a mortgage application.
List	Selection amongst a set of many components.	Airline preference selection in a travel reservation system.

Table 1: A set of basic building-block speech components

Given the tremendous flexibility of human language and the possibility for significant differences in language use across applications, it is necessary to be able to modify pre-built components, develop new components and so on. An implementation in an object-oriented language can provide this flexibility by permitting customization and inheritance (extension) of the built-in classes. Customization of any speech component is possible by direct modification of any or all of

the input expressions including the automatically generated expressions. Also, as we will explore in later sections, the basic speech components can be combined into compound components and into hierarchies to produce more advanced interfaces. Table 2 lists some examples of application-specific speech components with an explanation of how they might be implemented.

Example	Description	Implementation
Age	Interface to a user's age.	Integer component named "age."
Mortgage	Represents the term, interest rate, loan amount and payment plan for a mortgage.	Compound component including floating point values for the interest rate, loan amount and payment amount, plus a choice to select between weekly and monthly repayments.
Name list	Select one or more individuals from a list of people.	Compound component constructed from a component for each name and a list component that allows selection.
Font chooser	Select font characteristics including typeface, bolding, italics, size and so on.	Compound component constructed from a choice for typefaces, an integer for size, and switch for bold and italics.

Table 2: Examples of application-specific speech components

2.2 Speech Input

The basic building block of the speech input system is the *input expression*. An input expression defines a phrase or phrases that a user may speak. These may be full utterances (complete spoken sentences) or may be any continuous segment within an utterance.

Each input expression is associated with a speech component. Any speech component may have any number of input expressions (zero, one or many) and each expression represents something that can be said about that component. Each input expression of a speech component must have a name unique to the component. By combining the unique component names and unique input expression names, it is possible to unambiguously reference any expression of any component.

In the following sections we explain the core functionality and data that must be captured by any input expression:

1. What can the user say?
2. What does the user mean?
3. When can the user say it?

2.2.1 What Can the User Say?

In each input expression, we represent what the user can say by a *grammar fragment*. The grammar fragment should be a legal expansion of a rule definition (also known as a non-terminal) that the speech recognizer can understand. For our work, and in the examples below, we use the format defined in the Java™ Speech API Grammar Format [11].

An input expression may represent a complete speakable utterance about the speech component or may represent part of an utterance. As an example, consider a speech component for an interest rate for which the model is a floating point number representing the current rate as a percentage. We can define input expressions that capture various utterances to set the rate: “*set the interest rate to 7 percent*,” “*make the rate 3.5 percent*.” We might also define an input expression for utterances that request a relative setting: “*increase the rate by a percent*.”

An input expression can also represent a part of an utterance. A common pattern is to define an input expression representing the various ways in which an application’s data value can be referred. For the interest rate component we could define a list of aliases as:

```
<alias> = [the] (rate | interest rate | annual interest rate);
```

The syntax is that angle brackets represent the name of an expression, a vertical bar indicates a set of alternative phrases, square brackets indicates optionality, and parentheses performs grouping. This expression allows phrases such as “*the rate*”, “*rate*”, or “*the annual interest rate*.”

The expression for aliases is not intended to allow a user to say just “rate.” This expression is instead intended to be used as a building block for more complex input expressions. Similarly, a common pattern is to define an input expression for the legal values which, for the interest rate, would include sub-utterances such as “*5 percent*” or “*10.3 percent*.” Together, these two expressions can build an expression that supports the user saying “*make the rate 3.5 percent*” and the other examples above.

```
<assign> = make <alias> <legalValue>;
```

A less obvious input expression might be one that combines the aliases and legal values as follows:

```
<phrase> = <alias> <legalValue>;
```

The syntax of this expression is that the expression named `phrase` comprises a sequence that first contains a reference to the expression named `alias` followed by the expression named `legalValue`. It will match phrases such as “*the rate 10 percent*.” This expression becomes useful when we get to compound components that support commands such as “*make the rate 10 percent and the term 15 years*” by compounding an interest rate component and a term component.

It is particularly useful to allow the definition of what the user can say to change as the application’s data or state changes (assuming the underlying recognizer supports such changes). For example, a list of available printers might be dynamically created according to system resources and might change as a user request narrows the list (e.g., by selecting only color printers).

2.2.2 What Does the User Mean?

Understanding user input involves the conversion of a string of natural language into a format that can be interpreted and executed by an application. Many existing natural language processing systems typically perform this conversion by producing a set of *feature-value pairs* or some similar representation (also referred to as a slot filling system). For example, the Swiftus natural language processor of the SpeechActs system developed previously in Sun Microsystems Laboratories [12] is the predecessor of the system described below and used Lisp code to interpret speech input as feature-value pairs.

Given an input of “*set the interest rate to 7 percent,*” the semantic output might be the two follows features with values:

```
value = 7; action = set interest rate;
```

From this data it is obvious how to write application code to perform the requested action. However, with our focus on efficient application development, we desired a more direct process so the interpretation of actions results directly in function calls on the data models. For the interest rate example we would expect a call of the form:

```
appObject.setInterestRate(7.0);
```

The system we developed for this purpose embeds scripting tags into the grammars of input expressions. These tags interpret the speech input and perform the actions requested by the speech input and so are known as *Action Tags*. A full explanation of this system is beyond the scope of this paper so we will describe only the capabilities necessary to understand how the speech component architecture operates.

A tag is a string associated with some part of the regular expression defined in an input expression. For the Java Speech API Grammar Format, a tag is an arbitrary string enclosed by curly braces as shown in the following example (see [11] for details).

```
<city> = (New York | the big apple) {cityCode = "NYC";};
```

The Action Tags system allows the tag to contain programming code and defines how that code is executed. In our current implementation, the tags contain fragments of ECMAScript [13] which is a standardized form of JavaScript™ [14]. The Action Tags system permits each input expression to have a return value which is an ECMAScript object. The most effective approach is for the scripted tags to transform the spoken input into a programmatic form that captures the meaning of the words that match the input expression. To perform this transformation, the scripted tag has access to the words detected by the recognizer that match the input expression and to the objects returned by sub-expressions that are referenced in this expression. For example, an input expression for a date would typically return an object that contains the day, month and year spoken by the user.

We have found it particularly useful to use the LiveConnect [14] capability of some ECMAScript implementations. LiveConnect allows direct reference to application objects from within tags and also allows the scripted tags to create application objects dynamically. Thus it is possible to define an input expression that contains an action tag that references an application object (typically a model object) and directly performs function calls that carry out the actions requested in the spoken input that matches the input expression.

For example, the mechanism to set the interest rate is implemented as follows. The input expression for legal values is defined to return a value that is a floating point number (not shown). We define an input expression that performs the set as follows:

```
<setValue> = <legalValue> {rateModel.setRate($legalValue);};
```

The string between curly braces is the tag containing a fragment of ECMAScript code. The reference to `rateModel` is a reference to the application object for the model underlying the interest rate speech component. The syntax of `$legalValue` is an ECMAScript variable that references the return value of the `<legalValue>` expression. Finally, the command calls the `setRate` function on the `rateModel` with the value spoken in the `legalValue`.

Of course the method described here is not the only way to translate spoken words into actions. We argue, however, that this method is simpler and more direct than most systems and that it enhances the overall architecture by making code more readable and maintainable.

As evidence for this claim we suggest that the *co-location* of syntax and semantics is important. Because we define what can be said (the syntax captured by the regular expression) and what it means (the semantic interpretation embodied in scripts in tags) in one place, there is an increased chance that they will be consistent and can be maintained consistently. Moreover, unlike many other systems, the extraction and execution of the semantics are co-located, a feature that we have found to substantially reduce the amount of code and the number of human coding errors.

2.2.3 When Can the User Say it?

As the state and data of an application change, the set of utterances that the user might say can change. Thus, we need to continually update the set of utterances that the recognizer is listening for. We capture this by *activating* and *deactivating* input expressions. More specifically, we define the concept of an *enabling condition* which is a boolean value that indicates whether a specific input expression should be currently active.

There may be input expressions that are always active, for example, “quit.” There are also input expressions that are never active, such as the sub-utterance expressions explored in Section 2.2.1.

The most interesting enabling conditions are those that are dynamic. Some of the common conditions we have encountered are focus-related. In a dialog system, support for deictic focus is necessary to capture a sequence of commands like “Set the interest rate to 5 percent. No, make that 6 percent.” The exact same sequence may occur in a desktop application that combines speech with a graphical user interface. However, in a multi-modal system with graphics and speech, deictic focus is also tied to keyboard focus. For instance, a user may click on a visual interest rate field and then say “Make that 6 percent.” (In Section 3, we explore the similarities and differences in GUI and SUI design in more detail.)

Deictic focus is explicitly built into the architecture since it is a shared resource. It is implemented as an enabling condition that is true when a component has deictic focus and which automatically returns to false when another component grabs focus. Typically only one entity is able to own deictic focus at any time. One exception is for spoken input that refers to more than one entity. For example, following the command “set the interest rate to 5 percent and make weekly

payments of \$200,” it might be reasonable to say “*make that 6 percent*” or “*make that \$300.*” Shared deictic focus is supported by granting deictic focus to all components that request focus on a single input utterance. We return to this issue in Section 3 since this reveals an interesting distinction between the focus models of speech user interfaces and graphical user interfaces.

Other forms of dynamic activation we encountered include tracking window activation in a graphical application and tracking arbitrary properties of objects (e.g., only allow the interest to be set once a user name has been entered). It is also useful to construct enabling conditions as boolean operations composed of other enabling conditions.

2.2.4 Automatically Generated Expressions

There are many groups of expressions and many sets of components that exhibit similar spoken forms. We introduce the idea of *generated expressions* as a way of capturing spoken patterns. A generated expression is one that is automatically created by the system — not by the application developer. As with all input expressions, an automatically generated expression must be constructed with its grammar fragment, with the action tags for semantic interpretation, and with an appropriate enabling condition.

The most common occurrence of generated expression is in parametrically-generated speech components. For instance, we see common spoken structures in named values (e.g., “*interest rate is 5 percent,*” “*employee id is 1234,*” “*departure date is June 23*”). In our existing implementation nearly all speech components for named values are implemented by providing just the rule definitions for aliases and legal values. A set of complementary input expressions is automatically generated to capture a sophisticated set of commands relating to the value. Table 3 lists some illustrative examples of generated expressions that we build around those two rule definitions (our current implementation provides 13 automatically generated expressions).

Name	Meaning	Example Phrase
setValue	Wraps the legal values and adds action tag to perform a set call on the model.	<i>5 percent</i>
assignCommand	Complete sentence that performs assignment of value.	<i>make the rate 5 percent</i>
relativeAssign	Modify numeric values by a relative amount.	<i>increase the rate by 2 percent</i>
deicticAssign	Assignment when component has deictic focus.	<i>make that 7 percent</i>
requestFocus	Explicit request to obtain deictic focus.	<i>go to the interest rate</i>
speakValue	Command to have the value read.	<i>what’s the rate?</i>

Table 3: Examples of generated input expressions for a named value component

There are two motivations for the use of generated expressions. First, there is a demonstrable reduction in the amount of work to be done by the developer resulting in saved time and reduced errors. Second, there is increased consistency in the user interface because the patterns are repeated

in multiple components across the entire speech user interface. A desirable effect is that a user who learns a particular pattern of issuing commands to one component can reliably apply that same pattern elsewhere. Moreover, if user testing reveals that the patterns need to be added to or modified to capture additional patterns spoken by users, a modification to the generated expressions is made once and can be automatically distributed across the entire speech interface, even at runtime.

2.2.5 Cross-referencing Expressions

In previous sections, we have alluded to the ability to build input expressions that include references to other input expressions. For this mechanism to work we require two conditions. First, each expression must have a unique name. This is enforced by ensuring that each speech component has a unique name and that each input expression within the component has a unique name. In the examples here, a fully-qualified input expression name is a concatenation of the component and expression name. For example, `<mortgage.rate.legalValue>` references the legal value expression of the interest rate component in the mortgage application. For simplicity, references to expressions defined in the same speech component are typically the expression name only, for example, `<legalValue>`.

There are no explicit constraints on cross-references within or across expressions or even across component hierarchies (explained below). There may, however, be constraints applied by the speech recognizer or natural language systems. For example, recursion may be prohibited.

2.3 Output

The design of the speech output capabilities mirrors the speech input system in most respects. Any speech component may have any number of associated *output expressions*. Output expressions may contain cross-references to other output expressions either within or across speech components. Output expressions may be automatically generated, may be customized, and so on.

There are differences between input and output expressions and these differences reflect an asymmetry in speech user interfaces. Because the computer can only speak a single output expression at a time, output expressions do not require an enabling condition but do require a mechanism that triggers an output expression to be spoken. By comparison, when it is the user's turn to talk there may be multiple (possibly hundreds) input expressions active because there is a wide range of things that the user may say, and there is not a method to force a user to speak.

The semantic interpretation is also asymmetric. Output expressions are typically constructed by the developer or are embodied in a generated pattern that was human-authored. The computer does not need to understand the content of the output expression. Thus, unlike input expressions, output expressions have no semantic interpretation capabilities.

Two useful features of output expressions are that they may be dynamically generated and that they may contain text markup to control the output rendering by the speech synthesizer. Dynamic generation is the ability to generate the text to be spoken only at the instant that the request to speak is made. For example, in a speaking clock application there may be an expression for current time in which the text to be spoken is generated by getting the clock time and converting it to an appropriate

format at the time that the request to speak is made. Without a dynamic generation capability, the application could be burdened with updating the output expression for the clock every second or minute. By comparison, current speech recognition systems do not provide an ability to modify grammars during recognition of a utterance so input expressions need to be static leading up to the time that a user starts speaking.

The text markup capability allows the generated output text to include directives to the speech synthesizer that influence the way text should be spoken. The type of control depends upon the capabilities of the underlying speech synthesizer. In our work and in the following example we use the Java Speech API Markup Language [15]. This allows, for example, marking of emphasis, control of pronunciation, control of pitch, speaking rate and other prosodic features, and marking of special text constructs such as dates and times.

In the following output expression, named `stock`, we reference the `currentTime` expression of the clock component and the `currentValue` expression on the stock component (both these referenced expressions are likely to be dynamically generated). This example also shows the use of the `<emp>` element which is the element of the Java Speech API Markup Language that requests emphasis of the words between the `<emp>` and `</emp>` tags (standard XML syntax). Note that for output expression we use double angle brackets to indicate expression names to avoid ambiguity with the single angle bracket syntax of XML.

```
<<stock>> = At <<clock.currentTime> the price was  
           <emp> <<currentValue>> </emp>;
```

When an application requests that the `stock` expression be spoken, the expression is expanded. It might generate text for the speech synthesizer such as the following.

```
At <sayas class="time">2:45pm</sayas> the price was <emp> $78 </emp>.
```

2.4 Component Hierarchies and Compounding

Individual speech components can support considerable input and output expressive power because they permit multiple expressions and cross-referencing of those expressions (both for input and output). When cross-referencing is used across components, the expressive power is increased further. For example, for a font selection speech component (with similar functionality to font selection in typical document systems) we may permit the input command “*use a 10 point bold Helvetica font.*” The basic components might be an integer component for point size, a boolean value for both bolding and italics and a choice list for typeface. Each of these components may have an active input expression that controls the component alone (e.g., “*select Courier font*”). Each component should also have an input expression that can be used in compound statements (e.g., from the compound command above, “*10 point,*” “*bold,*” “*Helvetica*”).

We construct an input expression to combine appropriate input expressions of the point size, bolding and typeface components. In most designs we have found it logical to have a container component that is the parent of all the font selection speech components and that contains the input expressions that combine those components. There is, however, no architectural requirement for this

since cross-references can go between any pair of components without restriction by the tree hierarchy. For the font example, the parent may have an input expression such as:

```
<command> = use [a] [<size.value>] [<bold.value>] [<face.value>] font;
```

There are several advantages to combining expressions in this way. First, any change in the referred expressions is automatically captured by the compound expression and thus consistency within the user interface is preserved while development effort is reduced. Second, expressions tend to be simpler and thus easy to write and evolve. Third, for input expressions, the semantic interpretation responsibility is delegated from the parent to the child and thus processing input is also broken into smaller, simpler actions. Finally, the compounding mechanism is identical for both input expressions and output expressions making the design style easier to learn.

2.4.1 *Designing Hierarchies*

The design of component hierarchies is under the control of the application developer since the architecture imposes no specific constraints. We have found three considerations important in designing hierarchies: joint activation of components, reusability and clarity. Each of these considerations reflects the common-sense approach of putting related components together in the hierarchy.

Our implementation provides a default enabling condition for a speech component which tracks the activation of its parent component. With this design, activation and deactivation of a root component turns the speech input of the entire sub-tree on and off (except components that override this behavior). One common use of this behavior is in joint speech-graphical user interfaces. When a window has focus, we want all the associated speech components to be active. If the speech components associated with the window share a common root and if the enabling condition for that root component tracks the activation of the window, then the whole tree of speech components can be enabled and disabled automatically as window focus is gained and lost.

Reusability of speech component hierarchies is encouraged when the component hierarchy reflects the design of the application's model objects. Often this also leads to cross-references between objects that are closer in the component tree which tends to make the designs easier to understand and maintain. So, although cross-references can occur between input expressions on any pair of components, good programming practice tends towards localization of references.

Finally, although it is not a requirement of the architecture, our system ties the use of speech recognizers and synthesizers to the component tree. The speech engines are resources maintained by the root component and are inherited down the tree except where a component and its sub-tree choose to override the selection. In a multi-lingual application we might override the speech engines at specific points in the component tree to create a sub-tree for each language supported by the application.

2.5 **Rendering**

In this section we describe the mechanism by which the data contained in the speech components is transformed into data and controls for the speech recognizer and speech synthesizer. Although we

describe a system based on a single speech recognizer and a single speech synthesizer, the architecture can support multiple input and output engines (e.g. for multiple languages) by selectively rendering components to different engines.

2.5.1 Controller

The controller is the module of the architecture responsible for intermediating between the set of speech components created by the application and the speech recognizer. We use the term “controller” from the Model-View-Controller architecture. It’s specific responsibilities are to:

- ◆ Extract information from the speech components and their input expressions and set up the speech recognizer to listen for the appropriate input speech.
- ◆ Update the recognizer’s data as the components and input expressions are dynamically changed. Changes that must be tracked include changes in enabling conditions or the grammars of input expressions, changes in component hierarchies and the creation and deletion of components and expressions.
- ◆ Process speech recognition results by processing the action tags to determine the meaning and actions requested by the user. These typically lead to changes in the model.

The transformation process is dependent upon the underlying speech recognizer’s design and capabilities. For this architecture to be effective, we require the following minimum capabilities. The recognizer must be capable of performing recognition of speech input against dynamically created and possibly changing grammars (changes in grammars may be restricted to times between input utterances). It must be possible to activate for recognition any combination of rules (also called non-terminals) defined in these grammars and to change the set of active rules dynamically (again it is reasonable to restrict changes to between input utterances). The recognizer must be capable of generating recognition results that convey the string of detected words and rejecting the sequence if it is not an exact match to an active grammar.

The following method for mapping speech components onto a speech recognizer is straightforward and is implementable with most commercial and research recognizers. Each speech component is associated with a grammar in the speech recognizer. Each input expression of the component creates a rule in the component’s grammar. The definition of the rule is the regular expression defined for the input expression. The rule is activated and deactivated according to the enabling condition of the input expression. When the recognizer detects incoming speech that matches the rule, it is passed to the speech component that owns the matched input expression. The action tags are executed in the context of the component and any other components referenced by the matched input expression.

As was mentioned in Section 2.2.2, action tags in input expressions co-locate the syntax, semantic interpretation and semantic execution. The controller design facilitates this co-location by transferring the spoken input back to the speech components for processing and by allowing that processing to descend down the tree of referring input expressions. From a development viewpoint we have also found this provides significant reduction in the amount of supporting code.

The mapping of speech components and input expressions to grammars and rules, respectively, does not need to be one-to-one, as is the case for the system described above. One important case that we have found for modifying this mapping is when optimizations of the grammars can improve recognition performance (especially accuracy). A side-effect of using generated expressions is that an application can easily generate many similar expressions (e.g., “*change the font to Helvetica,*” “*change the font size to 10 point,*” and so on). Having many branches in active grammars with the same word patterns confuses many speech recognizers. We have implemented a controller that compresses out these redundancies to improve recognizer accuracy without affecting what the user can say, without placing any burden on the developer to be aware of such details of the speech recognition performance, and without affecting the processing of recognition results with action tags.

Other possible functions for the controller which we have considered but not tested are performing automatic tests for ambiguities in grammars, monitoring for complex grammars that will degrade recognition performance (e.g., by runtime calculation of perplexity), and providing a checkpoint for debugging and system monitoring.

2.5.2 *View*

Once again, the speech output functionality mirrors that of the speech input system and is again simpler. The viewer is the module of the architecture that is responsible for intermediating between the application’s set of speech components and the speech synthesizer. Its specific responsibilities are as follows:

- ◆ Upon a call to speak an output expression, the viewer recursively expands the expression to produce a text string that may contain text markup.
- ◆ The viewer passes the text string to the speech synthesizer.
- ◆ Depending upon application requirements the viewer may track the progress of the spoken output of the expression (e.g., wait until output is complete).

2.5.3 *Separation of the User Interface Data and Engines*

The controller and viewer operate to convert the user interface data contained by speech components and their input / output expression to a form that can be processed by the speech engines. This separation is particularly advantageous to application developers that lack any specialized knowledge of speech engines since much of the complexity of the systems can be hidden from them.

The separation also permits specialized capabilities such as dividing the computational resources across machines. The speech components may reside with the application data on one machine while the speech engines reside elsewhere on a network. One application of this client-server configuration is the deployment of speech interfaces on computing devices that lack sufficient resources to perform speech recognition or synthesis.

Finally, if the grammar format of input expressions or the markup of output expressions is incompatible with those of the recognizer and synthesizer, the controller and viewer may perform automatic conversions.

2.6 Dynamic Definitions

As an application's data and state change it is often necessary to update the user interface. As mentioned previously, any change in state or data may require a change in the definition of the input expressions (the grammar, the semantics or the enabling condition) or the definition of the output expressions. Moreover, these changes may also cause expressions to be created or deleted, speech components to be created or deleted and even the hierarchy of components to be modified.

It is the responsibility of the viewer and the controller to reflect these changes. For instance, when the controller is informed that any part of the speech input data is changed, it must reflect those changes into the recognizer. Because changes tend to occur in collections, we have found it important to apply a full set of changes simultaneously rather than one at a time to avoid the recognizer having an inconsistent set of grammar definitions.

3. Speech vs. Graphics

If there is a single lesson to be learned from developing an architecture for speech applications it is that writing a speech user interface is not the same as writing a graphical user interface. From this it follows that there is no way by which we could trivially or automatically extend existing graphical applications to support a compelling speech interface. In this section we explore some similarities of speech and graphics as user interfaces, but focus primarily on some of the fundamental and applied differences in designing and implementing applications with these different modes of interaction and how this is reflected in our architecture.

3.1 Similarities of Speech and Graphics

It is worth noting briefly some of the similarities of speech and graphical systems. First, most advantages of the MVC pattern apply to the design of both graphical and speech interfaces. The separation of the core application logic of the model from the interface implementation in the view and controller represent separation of function and form. This permits separate evolution of the core logic and the presentation to the user. This also permits the same application logic to be presented through more than one user interface, either through multiple simultaneous views/controllers or through different views/controllers at different times. As a result, we have found that existing graphical applications designed with separated models and user interface software have been the easiest to extend to speech interfaces.

Second, there are similarities in the user interface design process that need to be supported in the architecture. For example, rapid development and iterative enhancement of interfaces through user testing are solid practices in both graphical and speech interface design.

Third, although the nature of component-based designs may differ for speech and graphical applications, the use of common object designs wherever practical is useful to developers since it reduces the learning curve.

3.2 Fundamental Differences between Speech and Graphics

The differences between graphical and speech user interfaces derive, in large part, from the fundamental difference between spoken and visual communication human-to-human and human-to-computer. These fundamental differences affect the user interface design of speech vs. graphical interfaces, impact the implementation of applications with speech or graphical interfaces, and affect our design of an architecture intended to effectively supports both these activities.

Previous research in our group involved the deployment of conventional desktop applications, including email and calendars, as conversational speech-only agents [1]. This work confronted the differences between the graphical and speech versions of the applications from a user interface perspective.

- ◆ *Visibility*: Graphical interfaces are visible to the user and most functionality is apparent to the user by visual inspection or by manipulation (e.g., exploring menus). In contrast, Speech is invisible which makes it challenging to communicate the functional boundaries of an application to the user [1]. Users may experience more difficulty determining what actions they can perform and how to invoke those actions. Moreover, if speech input is misrecognized and the misrecognition is not apparent to the user's mental model of the application, the boundaries may become especially unclear. From a developer viewpoint, the invisibility of speech systems makes them harder to develop and debug and motivates the use of visual monitors (see Section 4.2).
- ◆ *Transience*: Speech input and output are transient: once you hear it or say it, it's gone. By contrast, graphical interfaces may be persistent and good design practice leaves information visible to a user until it is no longer required. Thus speech interfaces make different demands upon the users' attention and memory ([1], [2]).
- ◆ *Bandwidth Asymmetry*: Speech input is typically much faster than typed input whereas speech output can be much slower than reading graphical output, particularly in circumstances that permit visual scanning [1].
- ◆ *Temporality*: Keyboard and mouse events are discrete, nearly instantaneous events in which meaning and intent may be conveyed by a single event or may be conveyed by a sequence of discrete events. Speech input is neither instantaneous or discrete since an utterance may take many seconds to be spoken and consists of continuous data that is transformed to a word sequence by the speech recognizer. Although the final speech recognition result is effectively an instantaneous event it may be delayed by a noticeable time from when the user stops speaking.
- ◆ *Concurrency*: Speech-only communication tends to be both single-channel and serial. Most people find it difficult to listen and speak simultaneously or to actively listen to more than one voice at a time. Even though most listeners are capable of selecting one voice amongst competing voices (referred to as the "Cocktail Party" effect), the same is not true for multiple simultaneous channels of speech synthesis output from a computer. By contrast, visual output allows concurrent

displays of multiple channels of data which may be processed together by user or amongst which the user's attention may shift. Furthermore, many aspects of visual output are designed to respond in real-time to user input (e.g., text scrolls as a user manipulates a scroll bar) so input and output may be concurrent.

3.3 Semantic Differences between Speech and Graphics

The differences between speech and graphical user interfaces cited in the previous section deal with timing and presentation issues. There are also differences in the ways in which information is packaged and interpreted. These differences are most apparent in the input to an application. In this section we explore how the traditional input by keyboard and mouse differs from input by speech.

The basic communicative function for a user interface is to support a user who has a particular thought or intention and who wishes to convey that thought or intention to an application. We will assume for now that the thought or action makes sense to the application. The *surface form* is the input actually received by the computer: mouse movements, mouse clicks, keyboard presses, or audio input containing speech made up of a sequence of words.

As the previous section explained, these input forms differ temporally, especially in bandwidth. The way in which high level meaning is contained differs as well. A keyboard input for "B" may have no particularly meaning. However, when followed by "oston" and when typed in the context of a text field which an application developer associates with the destination city for a flight reservation there is an emergent meaning. Thus, it becomes the responsibility of the input capabilities of graphical toolkit and an application's usage of the toolkit to provide *context* to keyboard and mouse input so that an input event sequence is converted into meaningful application behavior.

The use of action tags in input expressions of speech components serves the same purpose of defining the context in which a word sequence from a recognizer is converted into meaningful and appropriate application behavior (see Section 2.2.2). Because of the differences in the two forms of input it is no surprise that the software that supports interpretation of the input is very different between graphical and speech interfaces.

Speech input can efficiently convey *compound semantics*: more than one piece of information or command in one input event. For example, "*fly from Boston to London next Thursday, preferably in the morning.*" To convey the same data by keyboard and mouse input would require multiple input events which together have the same meaning. There are examples of compound semantics in graphical systems (e.g., a mouse press indicating both "x" and "y" coordinates and an "OK" button on a dialog box that causes a set of values to be applied and the dialog box to close) but good interface design practice tends to guard against "overloading" the functionality of visual components.

One common form of compound semantics in speech involves commands that require both selection and action. For example, in "*forward the last three messages from Bob to Nicole*" there is an advanced selection of three messages, an action in the form of forwarding the messages, and a property in the form of the recipient. Even a simple command such as "*set the interest rate to 8%*" involves both selection of the interest rate field and entry of a value.

Dialog context is a form of context for which there is no direct counterpart in graphical interfaces. Anaphoric, deictic and elliptic references are examples of spoken forms that require a context to be maintained in which the references can be resolved. This is an area in which the flexibility of natural communication can become a liability for computer systems that are not sufficiently well-designed to interpret user input. That said, it is not an area in which we can shed light except to say that it stands as a complex research issue.

Keyboard input is targeted to specific components by window focus and by keyboard focus within an application (except for capabilities such as hot keys). Mouse input is targeted at specific components by the location of the pointer. Being invisible, speech is not also targeted at a specific component so it is not always clear what context should interpret particular spoken input. In the worst case, which is not necessarily uncommon, there is an ambiguity because more than one component may be listening for a specific command. The commonly-cited advantage to this flexibility of speech input is, however, that a user can quickly jump between topics or access functionality that may be buried in graphical menus.

There are some areas in which speech input is substantially lacking by comparison to keyboard and mouse input because of differences in the communication of semantics. For example, the role of the mouse as a pointing device makes input of spatial data very effective (e.g., setting the location of a scroll bar, choosing a color in a continuous palette, or drawing arcs). By contrast, speech is very inefficient for conveying spatial information (e.g., “*up, up, up more, up, right a bit, OK!*”).

3.4 Applied Differences between Speech and Graphics

The basic differences between speech and graphical systems that we have outlined in the preceding sections directly or indirectly impacted the design of a software architecture to support speech application development. The following are some of our findings including several distinctions between our architecture and existing software systems for building graphical user interfaces.

- ◆ *No “auto-speech-enabling”*: We often encounter developers with a desire to automatically speech-enable existing applications, for instance, by scanning graphical components for data to build the speech interface. This approach is often described as “*speech buttons*” since it would be desirable, for example, to click a button by speaking its label. Although this may be highly desirable, there is insufficient information in existing configurations of graphical components to obtain natural speech interfaces. For instance, buttons may display iconic images or abbreviated text, neither of which would be normally spoken by users. Furthermore, user studies have found that spoken and graphical vocabularies differ — users do not always say what they see [1]. It is worth noting that problems from transition between user interface paradigms are not new. The transition from text-based to graphical user interfaces required re-engineering of applications and efforts at automated conversions were found lacking.
- ◆ *Differing adherence to the MVC pattern*: Most developers currently design applications for graphical interfaces alone and this constraint does not force separation of the data model and user interface code. In prototype applications we have found it strongly desirable to separate the data model from the speech component system. This separation becomes particularly important in multi-modal applications since a single model may have multiple views and controllers which

need to interoperate.

- ◆ *Different type hierarchies*: The component inheritance hierarchy for GUIs tends to reflect either the visual representation or the form of manipulation of objects. For example, buttons and checkboxes often inherit from the same class since they are both manipulated in the same way, namely by a mouse click. The speech type hierarchy tends to reflect the meaning of the underlying data. For example, all integer components may inherit from the same objects even if they might be represented visually as a text field, a slider or a list.
- ◆ *Complexity asymmetry*: Broadly speaking, developers spend more time developing the visual design of GUI applications than the input capabilities (for mouse and keyboard). The reverse tends to be true in speech since setting up the grammars and activation of input expressions is typically a more complex process than defining the output expressions.
- ◆ *Undoing compound semantics*: Because a single utterance may contain more than one command or result in more than one action, mechanisms to undo commands must be capable of undoing all the actions of each command.
- ◆ *Focus*: We have found that speech focus, modelled as deictic focus, seems reasonably correlated with window and keyboard focus in the graphical interface. In both cases, the focus attempts to capture the user's current attention, which in dialog terms may be the current topic. However, the similarities and differences deserve considerable research. For example, if the user speaks the following two commands, "Set the height to 6 meters and the length to 8 meters. No, make that 7 meters." Should the application change the length or the height?
- ◆ *Misunderstandings*: In graphical interfaces, when the user types a key on the keyboard or moves the mouse, there is no mistaking what the user has done. With speech, the speech recognizer can misunderstand what the user has said. For example, the recognizer may think the user said "wreck a nice beach" instead of "recognize speech." When a speech recognition error occurs, the error is not always detected by the user and it is not always easy to design a user interface that presents feedback to the user to detect those errors without overburdening the user with feedback or distracting from the interface. Thus speech applications tend to adopt different strategies for recovering from mistakes.
- ◆ *Different hierarchies*: The primary function of graphical containers that implement hierarchies of components is to control screen real-estate and to determine the layout of components within that space. Hierarchies of speech components do not have a strong functional role beyond assisting developers in presenting and understanding code and with concurrent activation.
- ◆ *Different compounds*: The only form of compounding in speech components is cross-referencing between expressions. There is no form in speech that is analogous to compounds in graphical interfaces: for example, using an icon in a menu.

4. Discussion

4.1 Implementation Overview

In Section 2.1.1 and elsewhere in Section 2, we described some of the basic components of the speech component system. In this section we describe some additional details of our implementation of the architecture.

Because we developed the architecture in the Java Programming Language, it was our intention to make speech components that were compatible with existing technologies and APIs for the Java™ platform. Some of the basic design decision (which need not be followed in alternative implementations) include:

- ◆ *Event delegation model*: The event delegation (or event-listener) model is used extensively in applications written using the Java programming language. As such, the design pattern is familiar to developers and widely supported in development environments (e.g., JavaBeans™ tools). In our implementation all asynchronous communication between models and speech components, amongst speech components and between speech components and their viewers and controllers, uses the event delegation model.
- ◆ *Swing and AWT support*: To support common patterns in implementation of multi-modal applications we optionally allow any AWT component (and by extension any Swing component) to be tracked by a speech component. This allows us to conveniently tie the deictic focus for speech to the keyboard focus for the tracked component or to activate and deactivate the speech component as the graphical component gains or loses window focus.
- ◆ *Java Speech API support*: We use the Java Speech API [16] to access speech recognition and speech synthesis resources. We have also used the Java Speech API Grammar Format [11] for writing input expression grammars and the Java Speech API Markup Language [15] for writing output expressions.
- ◆ *Client-server model*: As was pointed out in Section 2.5, the architecture makes no assumptions about the location of the speech resources used to process speech input and output. Thus client-server speech recognition and synthesis is supportable without application redesign. We have designed a controller and viewer to use remote speech engines. We anticipate that this separation of user interface logic and I/O resources will be useful in large-scale telephony service deployments and for providing speech interfaces to small devices that lack the computing resources to perform speech recognition or synthesis.
- ◆ *Tools*: In the next section we briefly describe a visual development tool that permits software developers and user interface designers to view and manipulate speech components of a live application. This facilitates both debugging and rapid application development.
- ◆ *Accessibility*: Following our observation that automatically speech-enabling a graphical application will not produce an elegant user interface, we have not attempted to build an accessibility tool from our speech components (what is known as a *Type 2* accessible application, [17]). Instead our objective has been to enable a developer to directly author the speech user interface for an appli-

cation. If the application supports both graphics and speech then it may be directly accessible by users through different modalities (what is known as a *Type 1* accessible application). For example, users with visual impairments may be able to use the spoken interface while users with hearing impairments may be able use the graphical interface. However, as we pointed out in Section 3.2 and Section 3.3, it is not possible to guarantee that all forms of access to a multi-modal application are equally effective.

4.2 Visualizing Speech Components

The invisibility of speech interfaces (see Section 3.2) provides an analogous problem to developers since they are unable to see the interface they have written and determine quickly whether it is the intended design. Moreover, because speech applications can often allow thousands or even many millions of different input utterances at some point in time, it is usually not possible for the developer to determine or say all possible input utterances to determine whether the application is operating to specification. To address these problems we developed graphical tools that display the current state of the speech component hierarchy and allow the developer to introspect the system and even modify the system at runtime. These developer tools have become practical because of the break-down of the user interface software into many components that are easily navigable through the component hierarchy and which are each small enough to display and understand.

This visual tool is also very useful to speech user interface designers because it allows them to directly modify the speech input and output interface designs. By allowing runtime changes to the interface, the visual tools facilitate rapid iteration of the design. For instance, in response to problems encountered by trial users the designer can immediately modify and test a speech interface.

4.3 Future Work

Despite every effort to simplify the development of speech interfaces we do not claim that we have made the process simple, only simpler than with previous methods. Software developers and user interface designers still have to learn the idiosyncrasies of spoken communication and in most cases developers and designers must learn some of the differences between speech and graphical systems.

There are some specific technical issues that we have not yet addressed. We are hopeful that the architecture can be extended effectively to address many, but not all, of these issues.

- ◆ *Disambiguation*: There are two forms of input ambiguity possible in this architecture. There may be ambiguity within a component or input expression. For example, the command “*send a message to Bob,*” the name may come from a list component in which there is more than one Bob. In most cases such ambiguities are easily detectable and we are confident that a framework can be developed for resolving ambiguities at the component level. Ambiguity across components is more complex. The ambiguity of focus given in Section 3.4 is one example. If a developer is able to anticipate such ambiguities there may be some possibility that they could provide their own disambiguator (e.g., by checking with the user). If, however, a cross-component ambiguity is not anticipated we do not see yet how to automatically resolve the ambiguity with the user.
- ◆ *Undo*: We have not implemented an undo mechanism on any of our prototype applications but we

expect that the architecture can work with existing undo techniques, especially if the facility is supported by the models or wrappers for the models. One interesting undo feature required by speech is that it must be possible to undo multiple actions triggered by compound semantics in speech input.

- ◆ *Out-of-grammar errors*: No matter how well a speech user interface is designed and how flexible its input grammars are, it is not possible to constrain what a user says except by ignoring everything we don't understand. The current design of input expressions with explicit grammars for what the user can say is not robust when presented with unexpected input. Statistical techniques for speech recognition are able to cope with out-of-vocabulary and out-of-grammar input but present greater complexity to the semantic interpretation of speech input. It would be a particularly interesting challenge to try to modify the design of input expressions to support statistical grammars and robust parsing.
- ◆ *Help*: With the invisibility of speech interfaces it is particularly useful to be able to inform the user at any time what they can say. The speech component architecture presents two opportunities which we have only partially explored. The application designer may assume responsibility for building a help system, for example, by providing a list of acceptable commands in any application state. As the size of the application grows and as the set of acceptable commands is changed dynamically it could be more effective to have a partially or completely automated system. Since the enabling conditions define when particular input expressions can be spoken and because input expressions define what the user can say, it could be practical to automatically generate and update a list of legal commands for a user.
- ◆ *Dictation*: We have focussed on a speech input capability that can be used in speech-only and in multi-modal applications. Since dictation is currently only effective when visual feedback is available we have not attempted to develop a component for dictation input.
- ◆ *Internationalization*: The separation of the model from the view and controller and the co-location of syntax and semantics for input expressions both facilitate the authoring of internationalized applications. We have not, however, explored how to implement user interfaces that are functionally different or presented in a different ways across languages. For example, dialogs have a different flow in different languages often because of cultural differences like politeness at the start or end of a conversation.
- ◆ *Speech engines performance*: We are bound by the performance of speech recognition and speech synthesis technologies. Limitations in the accuracy and robustness of speech recognizers negatively impact the design of speech interfaces by requiring constraints in the input expressions that limit what the users can say. Limitations in the quality of speech synthesizers constrain the use of speech output and compound the inherent bandwidth limitations and transience of speech output (see Section 3.2). However, it is also the case that ongoing improvements in the speech engines can be translated into more sophisticated user interfaces.

5. Conclusions

We have presented a new approach to building applications with speech user interfaces by using fine-grained speech components. The architecture simplifies the development of speech applications and requires less experience to construct and evolve speech interfaces than previous systems. The architecture is intended to support good speech user interface design principles.

The system for breaking down the interface into speech components each with a multiplicity of input expressions and output expressions allows each component to be simple and to operate independently. The input expressions capture the basic requirements of what a user can say, what it means and when they can say it in one object. The output expressions encapsulate what the user can say. References are permitted across input expressions and across output expressions and this compounding of expressions permits complex interfaces to be constructed from simple components. Furthermore, for many classes of speech components there are common patterns for input and output expressions that can be automatically generated by templates which both reduces the workload for developers and increases the consistency of the user interface.

In Sun-internal deployments, we found significant reductions in the amount of code required to implement speech interfaces along with greater simplicity of that code and even greater functionality. The architecture is simpler to explain and learn and, with the use of visual tools, applications are easier to develop and debug. The visual tools also permit rapid iteration and enhancement of the user interface by allowing the user interface designer to modify the input and output expressions at runtime in response to problems encountered by users.

Through developing a speech component architecture and implementing applications with both graphical and speech user interfaces we have explored some of the basic differences between speech and graphical systems. Understanding these differences will be important for the majority of developers who have prior experience in developing graphical applications but who have not previously worked with speech interfaces.

Acknowledgements

The authors wish to thank their colleagues at Sun Microsystems Laboratories for excellent comments on drafts of this paper and for many insightful discussions. Thanks go particularly to Bo Begole, Nicole Yankelovich and Bob Kuhns.

References

- [1] Yankelovich, N., “*How do Users Know What to Say?*,” ACM Interactions, Vol. III, Number 6, 1996.
- [2] Schmandt, C., “*Voice Communication with Computers: Conversational Systems*,” Van Nostrand Reinhold, New York, 1994.
- [3] Hemphill, C., Godfrey, J., and Doddington, G., “*The ATIS Spoken Language Systems pilot corpus*,” Proc. DARPA Speech and Natural Language Workshop, Hidden Valley, PA, 1990.
- [4] Goldschen, A., and Loehr, D., “*The Role of the DARPA Communicator Architecture as a Human Computer Interface for Distributed Simulations*,” Simulation Interoperability Workshop, Orlando, Florida, March 14-19, 1999
- [5] Zue, V., et al., “*JUPITER: A Telephone-Based Conversational Interface for Weather Information*,” IEEE Transactions on Speech and Audio Processing, Vol. 8, No. 1, 2000.
- [6] Nuance Communications, “*Nuance SpeechObjects and V-Commerce Applications*,” 1999.
<http://www.nuance.com/>
- [7] SpeechWorks, “*DialogModules*,” 1999.
<http://www.speechworks.com/>
- [8] VoiceXML Forum, “*Voice Extensible Markup Language: VoiceXML*,” 1999.
<http://www.voicexml.com/>
- [9] Krasner, G.E., and Pope, S. T., “*A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*,” Journal of Object-Oriented Programming, Vol. 1, No. 3, pp26-49, 1988.
- [10] Gosling, J., Joy, B., and Steele, G., “*The Java Language Specification*,” Addison-Wesley, 1996.
- [11] Sun Microsystems Inc., “*Java Speech API Grammar Format*,” 1998.
<http://java.sun.com/products/java-media/speech/>
- [12] Martin, P., Crabbe, F., Adams, S., Baatz, E., and Yankelovich, N., “*SpeechActs: A Spoken Language Framework*,” IEEE Computer, Vol. 29, Number 7, July 1996.
- [13] ECMA, “*Standard ECMA-262: ECMAScript Language Specification*,” 1998.
<http://www.ecma.ch/>
- [14] Flanagan, D., “*JavaScript: The Definitive Guide*,” O’Reilly, 1998.
- [15] Sun Microsystems Inc., “*Java Speech API Markup Language*,” 1997.
<http://java.sun.com/products/java-media/speech/>
- [16] Sun Microsystems Inc., “*Java Speech API Programmer’s Guide*,” 1998.
<http://java.sun.com/products/java-media/speech/>
- [17] Chisholm, W., Illingworth, C., Novak, M., and Vanderheiden, G., “*Java Accessibility Preliminary Examination*,” Trace R&D Center, University of Wisconsin-Madison, 1997.
http://trace.wisc.edu/docs/java_access_rpt/report.htm

About the Authors

Andrew Hunt is a prior Principal Investigator for the Speech Group in Sun Microsystems Laboratories. While in the Labs, his research activities included the development of software architectures that support efficient development of speech applications. Andrew was the technical lead for the Java™ Speech API, including editing the specifications for the API and for the associated specifications of the Java Speech API Grammar Format and the Java Speech API Markup Language. Andrew holds a PhD from the University of Sydney, Australia where he researched speech recognition and prosodic modelling. He has developed advanced speech synthesis techniques both as an Associate Researcher at the Advanced Telecommunications Research Laboratories in Kyoto, Japan, and at the University of New South Wales, Australia.

Willie Walker is the current Principal Investigator for the Speech Group in Sun Microsystems Laboratories. His current research activities include the work described in this paper as well as integrating speech technology into desktop, telephony, and wireless systems. Prior to joining Sun, Willie worked for Digital Equipment Corporation on the X Window System and Motif. His work on X/Motif included the XKB keyboard extension as well as investigating making the X Window System more accessible to people with disabilities. While there, he also helped found DACX, the Disability Action Committee for X. Since joining Sun in 1997, Willie worked on the Java Foundation Classes. His major contributions to that effort include the design and implementation of the Java Accessibility API and the Multiplexing Look and Feel for Swing.