Making Mac Listen:
A Voice Recognition Toolkit for Macintosh Applications

Alma Whitten and Robert McCartney
Department of Computer Science and Engineering
University of Connecticut, U-155
Storrs, CT 06269-3155
E-mail: alma@cse.uconn.edu, robert@cse.uconn.edu

Abstract:

Commercial products now exist for the Macintosh which can perform recognition of discrete utterances for a set of pre-trained words. The question arises of how this capability might be integrated into and used within an application. In particular, how we might integrate such capabilities into an application without radical redesign, while maintaining its original non-voice capabilities and appearance to the user.

We have developed and implemented a toolkit in Macintosh Common Lisp which can be used with any voice recognition product capable of generating an AppleEvent with a recognized utterance as a string parameter. The toolkit is a package consisting of centralized processing code and a set of specialized versions of standard MCL user-interface objects, such as windows, buttons and other dialog items. Integrating the toolkit into an application allows the user to refer to any on-screen object by a sufficient subset of its text label, causing the object to respond as if it had been mouse clicked. All such processing is transparent to the application designer, who need merely substitute the provided object types for the standard versions and include the processing code. A level of

voice recognition capability is thus provided which is dynamically responsive to the state of the screen display, which requires no pretraining beyond the association of a sufficient vocabulary of discrete words with AppleEvents, and which allows the user to mix voice input with mouse and keyboard at any time.

In this paper, we compare our approach to the alternative of predefining an action for each utterance to be recognized. We discuss the algorithms, specialized user interface objects, and data structures used to maintain information about the screen contents and to support incremental recognition of on-screen objects. We discuss the options we provide for error recovery and for conservative vs. liberal recognition strategies, and give an example of how voice capabilities were added to an existing application.

Since the user interface objects involved are standard for the Macintosh, the current implementation may also serve as a prototype for translation into other programming languages such as C. The hardware/software product currently being used to recognize individual words is the Voice Navigator II SW from Articulate Systems, Inc.

1 Introduction

When attempting to make use of a voice recognition product which can perform adequate recognition of discrete utterances for reasonably large vocabularies, a first approach is to assign recognizable words to keyboard and/or mouse action macros. This approach, which is likely to be the responsibility of the end user, is useful but has several drawbacks: the number of trainable responses is limited to the number of trained words; effectable responses are limited to those which can be realistically anticipated and pre-trained; and no use is made of the semantic nature of the heard words beyond that which might be inherent in the user's macro assignment.

We have investigated and implemented a second approach, which targets the application designer rather than the end user. The approach begins with the decision to utilize the semantics available from

context, and does so by restricting the interpretation of recognized words to references to objects, such as buttons and menu items, which are currently displayed on-screen. Reference is presumed to be primarily by text label, but references by name of type may also be made. When a reference to an individual on-screen object is identified, that object is triggered as if it had been mouse clicked.

From the user's perspective, an application which utilizes the **Voice Toolkit** will respond to speech input by visibly marking those objects on-screen which are associated with the entire sequence of words heard so far, until a single object has been identified. Figure 1 shows an example application immediately after processing speech input of the word "button"; the question mark display indicates that further input is required before a single object can be identified, and those objects which are associated with the word "button" either through their types or through their text labels have been marked using italics. Once the identification has

Figure 1 - User's perspective

been made, that object is activated as if it had been mouse clicked, and the speech processing state is reset. If an object is activated by mouse click or other input method at any point, the speech processing state is also reset and all marking of objects is removed; this allows the user to change input methods freely.

In order to make use of our implementation, the application designer includes the Voice Toolkit code and uses the versions of the standard user interface objects which we provide. These objects are completely compatible with the original versions, but include additional capabilities which allow for speech recognition response as described above. If the resulting application is run on a system that includes speech recognition abilities, it will respond to speech input; if the system does not include such abilities, the application will behave as it would have originally.

Additionally, there are a variety of options which can be set by the application designer using a function provided for that purpose. These options control several aspects of the response behavior, and can be used to tailor that behavior as appropriate for a given application. The designer may choose to provide the user with access to these options as well.

2 Use of Available Vocabulary

Given a vocabulary of n recognizable words, the approach of defining a response macro for each word clearly yields only n system responses. This ratio may be improved upon if the speech processer is designed to provide for hierarchical responses. By hierarchical we mean processing such that a single trained word causes different system responses depending on the context in which it is heard; an example of this would be the processing of the word "page" within the pulldown menu activated by the word "insert", as opposed to the processing of the same word within the pulldown menu activated by hearing "format". This would allow for n^k responses with k equal to the number of levels of hierarchy which the speech processer can handle. However, each response must still be individually defined, so although the ratio of speech training to usable reaction is improved, the ratio of response training to usable reaction remains one to one.
The approach we present herein requires that each vocabulary word be matched to the reaction of generating a specialized AppleEvent with itself as a string parameter. All response beyond that point is handled by the Voice Toolkit within the application itself. Vocabulary words received as speech input are matched to a system reaction using the context of the types and text labels of the currently on-screen objects. Using this approach, the initial training of n vocabulary words to produce n corresponding AppleEvents allows the system to react by activating any on-screen object which is associated with a sufficient subset of the vocabulary to uniquely identify it within the current screen display. Within a given screen context, this approach allows for $n!/(n-j)!$ usable system reactions, where j is the realistic bound on the word length of a text label. Taken over a variety of screen contexts, the vocabulary size no longer places any limitation on the number of usable reactions to speech input.

3 Provided Options

3.1   Recognition Strategies

The Voice Toolkit allows the application designer to modify the speech recognition behavior to the level of conservatism most appropriate to the application. At the most conservative level, one designated word must be spoken to alert the system at the start of each sequence of speech input, and, once the sequence has been processed to identify a reference to a particular on-screen object, another designated word must be spoken to confirm the presented identification before the object will be triggered.

At a slightly more liberal level, individual objects may be set so that, when one of those particular objects is identified for reference, confirmation of the identification is not required. In an application which in general requires conservatism, this may be appropriate for check-boxes and radio buttons, for which triggering is unlikely to have any irrevocable consequences.

If this degree of conservatism is unnecessary, the confirmation step may be done away with entirely. Likewise, the alerting step may also be done away with so that speech input will always be accepted for processing; this may be done in combination with any of the choices of confirmation strategy.

The designation of the words to be used in the above strategies is also under control of the application designer; for details refer to the sections on public functions and on object reference processing.

3.2   Error Recovery

The Voice Toolkit provides an optional method for attempting to recover from a mis-identified word within a sequence of speech input. When this option is enabled and there is no on-screen object which is associated with all of the current sequence of speech input, the system utilizes its previous experience of successful recoveries to construct a set of guesses as to which is the intended object. These guesses are then presented to the user singly in order of expected probability until either the user validates a guess as correct or the set of guesses runs out. This process is described in detail in the section on guessing and learning behavior later in this paper.

4 Implementation Overview

The Voice Toolkit is a package consisting of the **Voice Handler**, which is responsible for receiving and processing the heard words according to the option parameter settings and the data it maintains about the objects currently displayed on-screen, and the specialized voice-aware object versions, which communicate with and respond to the Voice Handler as needed.

4.1   Voice Handler

4.1.1   Interface to Voice Recognition Product

The Voice Handler defines a special AppleEvent handler which allows it to accept recognized words for processing as string parameters to AppleEvents of type *aevt* and id *hear*. It can therefore be used with any voice recognition product which is capable of generating such AppleEvents. More detailed information on how to set this up is given in the appendix to this paper.

4.1.2   Data Structures

The Voice Handler maintains several categories of data structures. The first group consists of the parameters which reflect the user's or designer's choices from the options which are provided to control the behavior of the Voice Handler's interaction with the user. These are described as follows.

`*mark-method*`   Specifies the method which is used to indicate to the user which on-screen items are currently candidates for reference according to the words heard so far.

Currently the options available are *:ITALIC*, *:BOLD*, or any of the standard MCL color values such as **blue-color** or**red-color**. The default value is *:ITALIC*.

`*start-word*`   The recognizable word which is used to alert the Voice Handler for a sequence of speech input to reference an object. If the Voice Handler is already in an alert state, this word will be handled like any other. If this parameter is set to *nil*, the Voice Handler will always be in an alert state. The default value is `"LISTEN"`.

`*fire-word*`   When the Voice Handler has identified a sequence of speech input as a reference to a particular on-screen object, speaking this recognizable word confirms the reference and causes the Voice Handler to trigger that object as if it had been mouse clicked. If the particular reference has not yet been identified, this word is handled like any other. If this parameter is set to *nil*, the Voice Handler will trigger the referenced object as soon as it can make the identification, without waiting for user confirmation. After triggering the referenced object, the Voice Handler resets to either an unalert state or a refreshed alert state, depending on the value of `*start-word*` as described above. The default value of `*fire-word*` is *"GO"*.

`*cancel-word*`   This word is used to reset the Voice Handler, canceling processing of the current sequence of speech input. The reset takes place as in the description given for `*fire-word*` above. This word will always be processed in this context; it can never be used to reference an object. The default value is *"FORGET IT"*, to be processed as one word.

`*guessing*`   This is a boolean value which enables or disables the Voice Handler's ability to offer guesses to correct for misheard words.

`*next-guess*`   If the Voice Handler's guessing ability is enabled, then at the point in processing a sequence of speech input where a word has been misheard and the Voice Handler is offering guesses as to the object to be referenced, this word is used to tell the Voice Handler to offer its next guess. At all other times this word is processed like any other. The default value is `*"NEXT"*.

The second group is made up of variables which are used to record and maintain the current state of matching a speech reference to an on-screen object. These are:

`*attention*`   This variable is a boolean value which determines whether or not the Voice Handler is currently alerted for a sequence of speech input.

`*wordlist*`   This is a list of the words recognized so far in the current sequence of speech input.

`*marked*`   This is a list of the on-screen objects which are currently candidates for reference according to the current sequence of speech input, and which have been marked for the user's notice accordingly using the method specified by the `*mark-method*` parameter as previously described.

`*fixes*`   When a word has been misheard and the Voice Handler is exercising its guessing ability, this variable holds a list of the current guesses, ordered by decreasing probability according to the Voice Handler's previous experience.

Third, the following group of parameters holds information about the objects currently displayed on-screen:

`*screen*`   This variable identifies the window which currently has input focus, unless the window is not a Voice Window, in which case it is *nil*.

`*wordtable*`   This is a hash table which, given a heard word as a key, provides a list of the objects currently on-screen which contain that word in their text labels or in their object typenames.

Fourth, the following variable is used for the learning and guessing option:

`*twintable*`   A hash table which maps an assumed misheard word to the accumulated data from previous successful assumptions of mishearings of that word. Lastly, this variable is used to suppress the Voice Handler's communication with the user until it has been confirmed that the system is set up to handle voice input:

`*voice-system*`   Initialized to *nil*, this variable is set to *t* as soon as the application receives a speech related AppleEvent.

4.1.3   Object Information Management

The data structures which provide information about the voice-type objects currently displayed on-screen, `*screen*` and `*wordtable*`, are updated in response to notification provided by the voice-type objects themselves. This updating takes place whenever a window gains or loses input focus, or when there is a change to the subviews of the window which currently has input focus. If a notification to update occurs during the processing of a sequence of speech input, the Voice Handler's processing state is reset.

4.1.4   Object Reference Processing

The Voice Handler allows the user to trigger a user interface object by means of a sequence of speech input rather than by a mouse click. As each heard word is processed, on-screen objects which are associated with the entire current speech input sequence are marked for the user's notice. When enough words have been heard to identify one particular object, then that object is activated as if it had been mouse clicked, and the Voice Handler is reset.

Processing a sequence of speech input begins with the placing of the Voice Handler in an alerted, *waiting-for-input* state. If `*start-word*` is *nil,* this is done automatically whenever the Voice Handler is reset; otherwise reset places the Voice Handler into the *unalert* state until it receives input of the `*start-word*`.

While in the *waiting-for-input* state, the Voice Handler maintains a set S of those on-screen objects which have been marked using the specified `*mark-method*` to identify them to the user as current candidates for reference. Each time the Voice Handler is placed into the *waiting-for-input* state, the set S begins as empty. When the first regular speech input word w_i is received, the Voice Handler accesses the `*wordtable*` and retrieves the set R_i of currently on-screen objects associated with that

word. These objects are then marked using the specified *mark-method*, and S is set equal to R_i . As each successive speech input word w_j is received, its set R_j of associated objects is retrieved, and S is set equal to the intersection of S and R_j, unmarking the eliminated objects accordingly. The Voice Handler remains in this state and processes input in this fashion until either the *cancel-word* is heard, in which case the Voice Handler is reset, or it becomes true that S no longer contains more than one object.

At this point, if S contains exactly one object, then that object is presumed to be the object of reference, and the Voice Handler enters its *success* state. If the value of *fire-word* is *nil* or if the object is set as volatile, the object is activated as if it had been mouse clicked, and the Voice Handler is reset. If the object is not volatile and *fire-word* is not *nil*, then the Voice Handler waits for user validation in the form of input of the *fire-word*, or user rejection by input of the *cancel-word*.

On the other hand, if S contains no objects, the behavior of the Voice Handler depends on whether guessing is enabled. If guessing is not enabled, the Voice Handler enters the *failure* state and waits for user input of the *cancel-word* in order to reset. If guessing is enabled, the Voice Handler enters the *guessing* state and generates a sorted list of plausible guesses, which process is described in detail later in this paper. The generated guesses are presented to the user one at a time by marking the object associated with the guess and waiting for user reaction. When presented with a guess, the user has the following options: input the *fire-word* to validate the guess as correct and process as from the *success* state; input the *cancel-word* to reset the Voice Handler and start over; input *next-guess* to be presented with the next guess on the list; or input another regular word, which will cause a new, more precise list of guesses to be generated and presented. When the Voice Handler has no more guesses to present, it goes into the *failure* state as described above.

Should any object be activated by another method such as a mouse click or keypress during the process described above, the Voice Handler will be reset.

4.2   Voice-Type Objects

The specialized object versions fall into three categories of responsibility. First are those top level objects which are not themselves referenced by speech input, but instead act as the manager for the voice-type objects they contain; in the current implementation only Voice Windows are top level voice-type objects. They are responsible for notifying the Voice Handler whenever there is a change to the contents of the screen display.

Next are the basic voice-type objects, such as Voice Buttons and Voice Check-Boxes, which do not contain other voice-type objects; these respond to commands from the Voice Handler in order to process the speech input sequence. Finally, there are special cases which do not interact directly with the Voice Handler at all, represented in our implementation by Voice Sequences, which are contained in Voice Windows but act only to manage the Voice Slots they contain.

4.2.1   Top Level Voice-Type Objects

Top level voice-type objects notify the Voice Handler when they receive input focus, and then cause all of their voice-type subviews to identify themselves to the Voice Handler so that the *wordtable* can be updated. The only other point at which they interact directly with the Voice Handler is when they close or hide.

4.2.2   Basic Voice-Type Objects

The Voice Handler interacts primarily with the basic voice-type objects, which include Voice Buttons, Voice Radio Buttons, Voice Check-Boxes and Voice Slots. A basic voice-type object must provide the following methods:

identify(object)   This method is used by higher level voice-type objects to cause the basic voice-type objects they contain to identify themselves to the Voice Handler as currently displayed on-screen.

text(object)   Returns a text string containing all words to be associated with the object.

mark(object)   Modifies the text display of the object on-screen according to the current value of *mark-method*, to inform the user that the object is a candidate for reference according to the current sequence of speech input.

`unmark(object)` Returns the text display of the object on-screen to its original mode.

`select(object)` Triggers the action that the object would perform in response to a mouse click.

In addition to providing the above functions, basic voice-type objects are required to inform the Voice Handler whenever they are triggered by any method, in order that the Voice Handler may reset itself.

Basic voice-type objects contain an additional attribute which identifies them as being *volatile* or not. Objects which are volatile will be triggered by the Voice Handler as soon as the reference is identified, without waiting for user confirmation, regardless of whether `*fire-word*` is *nil*. This attribute may be set by the application designer using the keyword `:volatile t` within `make-dialog-item`. Voice Slots inherit this attribute from the Voice Sequence which created them. Voice Radio Buttons and Voice Check-Boxes are volatile by default; all of the other basic voice-type objects are not.

4.2.3   Hybrid Voice-Type Objects

A hybrid voice-type object such as a Voice Sequence never interacts directly with the Voice Handler. Instead, its responsibility is to spawn a Voice Slot for each of its individual parts, and to pass along any `identify` command it receives to each of its Voice Slots. The standard methods for the Voice Sequence all return values as if the Voice Slots were not present, and the Voice Slots interact with the Voice Handler as individual basic voice-type objects.

5 External Aspects

5.1   Public Functions

Other than the specialized object types and corresponding versions of the standard methods on those objects, the Voice Toolkit provides only two functions to be used by the application designer. The first of these functions, `set-voice-handler`, is used to set the behavioral option choices at application start-up or at any time thereafter. To accomplish this, the function may be invoked with any of the following keyword arguments:

`:alert-on` Sets the value of `*start-word*` to the input string.

`:cancel-on` Sets the value of `*cancel-word*` to the input string.

`:accept-on` Sets the value of `*fire-word*` to the input string.

`:next-guess-on` Sets the value of `*next-guess*` to the input string.

`:guessing-p` Sets the value of `*guessing*` to *t* or *nil*.

`:mark-method` Sets the value of `*mark-method*` to the method specified.

The arguments provided with the above keywords are checked for validity, and any invalid argument produces an immediate error. This validity checking includes the fact that `*fire-word*` may not be set to *nil* while guessing is enabled and vice versa, since the user must be able to validate a presented guess.

The second function, `close-voice-handler`, which has no arguments, is used to shut down the Voice Handler. It is only necessary to call this function if the guessing and learning option has been enabled and it is desired to save what has been learned during the current use of the application.

5.2   Interaction with User

The tools with which the Voice Handler communicates with the application user are the `*mark-method*` and the **Voice Flag**. The `*mark-method*` is used to visibly indicate to the user each on-screen object which is associated with all of the words in the current speech input sequence, allowing the user both to visually determine which spoken word would next be most appropriate with greater ease, and also to verify that an error in the speech recognition has not occurred. The Voice Flag is a small floating window which indicates the current processing state of the Voice Handler by the display of a variety of corresponding symbols, and appears whenever a Voice Window currently has input focus. The five possible processing states described in the

section on object reference processing are indicated by the five symbols shown in Figure 2.

Figure 2 - Voice Flag States

The five Voice Flag displays shown in Figure 2 are explained as follows:

blank   Corresponds to unalert state, waiting to hear `*start-word*`.

blue question mark   Indicates waiting-for-input state. Expects to hear either another regular input word, or `*cancel-word*`.

red circled question mark   Indicates guessing state: the sequence of speech input received does not match any on-screen object, the guessing option is enabled, and the currently marked on-screen object represents the Voice Handler's offered guess. Expects to hear `*fire-word*`, `*next-guess*` or `*cancel-word*`, will also accept additional regular input words in order to narrow down guesses.

yellow smiley face   The success state, meaning that the currently marked on-screen object has been identified as the object referenced by the current sequence of speech input, and `*fire-word*` is not nil. Expects to hear `*fire-word*` or `*cancel-word*`.

green unhappy face   This is the failure state. The sequence of speech input received does not match any on-screen object, and either the guessing option is disabled or the Voice Handler has run out of guesses to offer. Waiting to hear `*cancel-word*`.

6 Guessing and Learning

6.1   Behavior

When the guessing option is enabled by calling set-voice-handler with keyword `:guessing-p t`, the Voice Handler will attempt to recover from misheard words by guessing at plausible corrections for one of the words in its speech input sequence. If it should be the case that more than one word in the speech input sequence was misheard, then the Voice Handler will not be able to recover and the user will need to use the `*cancel-word*` and start over.

Given a speech input sequence of n words which cannot be matched against any object currently on-screen, the Voice Handler takes one of the heard words, assumes that it was mis-heard and that the remaining $n-1$ words were heard correctly, and retrieves the list of objects which might be referenced by the $n-1$ assumed correct words. This is done for each of the n heard words in turn, appending the retrieved lists of objects to create a list of potential guesses. This list necessarily contains no duplicates, since in order for it to do so there would have to be an on-screen object which was a candidate for reference by two different $n-1$ size subsets of an n size set of words, and, since the union of those subsets would be same as the set itself, the object would be a valid candidate for reference by the entire speech input sequence and the Voice Handler would not be guessing.

The generated list of guesses is then sorted in decreasing order of weight according to the Voice Handler's previous experience. The weight of a guess is calculated as follows: first, the set X of words which might be the correct identifications of the assumed misheard word w is isolated by taking the set of words associated with the guessed object and removing those words which are accounted for by the words in the speech input sequence which were assumed to be heard correctly. Then, w is used as an index into the `*twintable*` to retrieve a set Y of words and associated data which represents the experience gained from previous user-validated guesses for which w was the word assumed mis-heard. Finally, the guess weight is generated by summing and normalizing the weights of those words in Y which are also in X , where the weights are derived from the data representing previous experience.

The sorted list of guesses is then presented to the user, one guess at a time as described in the earlier section on object reference processing. If the user validates a presented guess by speaking the `*fire-word*` then, for the word w and sets X and Y of words previously mentioned, the accumulated experience for mis-hearings of w as represented in Y is updated by increasing the weight and count associated with each word in X by, respectively, $1/m$ where m is the size of X , and incrementing by one. Words in X which were not previously present in Y are added with weight and count initialized to zero before being included in the update.

As an example of this process, consider a situation where the user wishes to reference the item labeled "Tech Report" in Figure 1, and has spoken first "report", and then "tech" to distinguish between the two reports. Assume that the system has identified the word "report" correctly but has misheard "tech" as "check". The Voice Handler then retrieves the set of on-screen objects which can be referenced by "report", which consists of the objects labeled "Tech Report" and "expense report", and the set of on-screen objects which can be referenced by "check", which is the empty set. The set of possible guesses therefore now contains the objects labeled "Tech Report" and "expense report".

This set of guesses now needs to be ordered. For both of the generated guesses, "check" is the word assumed misheard. The `*twin-table*` is accessed and all information about previous mishearings of "check" is retrieved. The weight of the guess "Tech Report" is set equal to the weight accumulated by previous experiences of mishearing "check" for "tech", and the weight of the guess "expense report" is likewise set according to previous experience of
mishearing "check" for "expense". Presumably over time the former will have accumulated more weight, and the guess of "Tech Report" will be presented to the user first. If the user validates the guess by speaking the `*fire-word*`, the experience weight for mishearings of "check" for "tech" will be incremented for future reference.

6.2   Representation and Storage

The accumulated error recovery experience is stored in the `*twintable*` such that $word$ is keyed to a list of tuples of the form $<word_i , weight_i , count_i>$, in which $word_i$ has been a candidate as the correct identification of the speech input misheard as word and $weight_i$ and $count_i$ are as described in the previous section on behavior. The mishearing of $word_i$ as $word_j$ and the mishearing of $word_j$ as $word_i$ are treated as completely separate events; no assumption is made about the relatedness of their probabilities.

This information is stored in a file named *Experience File* , and is loaded into the `*twintable*` at whatever point the

guessing option is enabled. If the experience file is not present, the accumulation of experience begins again from scratch, and an experience file is created when the new experience is saved at application shut-down. This allows the user to clear the accumulated experience simply by removing the experience file, or to maintain different sets of experience for different users by providing the appropriate experience file for each.

6.3   Additional Capabilities

An interesting byproduct of the Voice Handler's ability to use experience to identify and deal with homonyms arises from the fact that the Voice Handler's accumulated experience deals only with whether the word that was misheard could be corrected by substituting a different word, at a level completely ignorant of the shape or sound of the words involved. Because of this, the Voice Handler, given appropriate user reinforcement of its guesses, is equally able to learn to guess at corrections involving synonyms of heard words.

As an example of this, our testbed implementation exhibits a tendency to mishear the word "cut" as "put". After a period of use with guessing enabled, the Voice Handler begins to assign a high weight to guesses which involve "cut" as a candidate for the correct identification of the word misheard as "put". Now suppose an input word is correctly identified as "slice". If the speech input sequence contains other words which are associated with the object associated with "cut", then that object will be presented to the user as a guess. If the user validates the guess, then "cut" will be recorded as a possible correct identification of "slice", and, given continued user reinforcement, will continue to accumulate weight as such.

7 Efficiency Issues

Integration and use of the Voice Toolkit adds processing cost to an application at three times: at application load and shut-down; when a new Voice Window receives input focus; and when voice input is being processed. As far as the first is concerned, the code involved is not lengthy, and the time required to load it should not be a noticeable addition to any but very small applications. Should the learning/guessing option be enabled, the initial loading and final saving of the experience data are each theoretically $O(n^2)$ for a vocabulary of n recognizable words, since at most each of the n words could have previous experience of being matched against the remaining $n-1$ words, creating a table of size $n(n-1)$.

When a Voice Window receives input focus, the processing involved is that of identifying the currently displayed objects to the Voice Handler. The controlling factors here are the number j of currently displayed objects and the number k of words associated with them. It is clear that both of these factors have fairly small realistic bounds, enforced by limited screen display area, and since the updating of the `*wordtable*` involves k hash table accesses each with traversal of a list of length at most j , this can effectively be considered constant time. For a reasonably crowded Voice Window running on a Quadra 700, there is no noticable delay time. The remaining processing is all directly related to the handling of speech input; should no speech input be received, the only extra processing which will take place is the reset notification to the Voice Handler whenever a voice-type item is mouse clicked.

Handling of speech input involves the following processing for each heard word: the hash table access to retrieve the set of objects associated with that word; the set intersection of the retrieved set with the set of currently marked objects; and the unmarking of the objects no longer present in the resulting set. If the word is the first in the sequence, then processing instead involves only the hash table access and the marking of each object present in the retrieved set. The highest complexity aspect of this processing is the set intersection, which is $O(n^2)$ in the set size, but the set size can not be larger than the number of objects which can be realistically displayed in limited screen space. Also, the marking and unmarking processes, which require graphics output, effectively overshadow the other

aspects of the processing time.

8 Example Application

The application used as a testbed in the development of the Voice Toolkit was COOKIE [2], a case-based planner which operates in the domain of meal planning and preparation. COOKIE's user interface interacts with the user through a series of windows containing a variety of predefined menus and buttons in order to determine a set of goals for a desired dinner and select a case which satisfies those goals. When this is accomplished, COOKIE turns control over to DEFARGE [3], its execution monitor, which leads the user through the step-by-step preparation of the meal in real time.

DEFARGE's user interface contains displays of the actions to be performed immediately, the actions which are waiting for the result of some test to be positive, and the tests themselves. The user interacts with DEFARGE by clicking on the actions as they are performed and the tests as the corresponding results become positive. Tests and actions are displayed as text descriptions, as shown in Figure 3.

As an application, COOKIE is well suited for utilization of voice input, since its use involves primarily interaction with dialog-items, with very little keyboard input required. However, the first approach of defining mouse action macros as voice responses, as we described in our introduction, is of little use in this instance. COOKIE's preliminary stage involves a large and varied number of buttons and scrolling menus; predefining a command for each possible choice would be extremely tedious. Even worse, DEFARGE's scrolling menus have dynamically changing contents according to the recipe being executed, so it is not possible to predefine voice responses for the menu items.

COOKIE and DEFARGE both use their own subclasses of several of the standard user interface objects; with use of the Voice Toolkit these become subclasses of the voice-type versions. At this time the trained vocabulary contains about 80 words. As COOKIE's development has not yet moved into a phase involving steady realistic use, it remains to be seen which of the Voice Handler option configurations will be most appropriate.

Figure 3 - Defarge waiting for user validation of italicized item

Validity as a Prototype

9.1   Other Languages and Platforms

The approach presented in this paper should be equally usable
in any object-oriented, window-based, event-handling user
interface. Since this describes the X Windows system for UNIX
and the Windows system for DOS as well as the Macintosh user
interface, the validity of the approach should be broad.

Transferring the functionality of the Voice Toolkit to another
language and/or platform would involve the following:

* Defining a special event type with a string parameter, and a
corresponding event handler for that type.

* Creating a package of Voice Handler code to maintain the
data structures and interact with the user as described.

* Creating specialized subclasses of the user interface objects
which respond to the Voice Handler while maintaining
compatibility with the original versions.

9.2   Aspects Specific to MCL

The implementation of AppleEvent handlers within MCL
enforces two functionality constraints which

may or may not be present in other languages. First, AppleEvent handlers are required to return normally; any attempt to throw past them is blocked at run-time. This means that a call to `return-from-modal-dialog` made as the result of activation of an object by voice will be blocked, unless that activation is performed after the return from the AppleEvent handler.

Second, AppleEvent handlers will not process newly arriving AppleEvents until the current call to the AppleEvent handler has returned. Should the activation of an object by voice result in a call to `modal-dialog`, within the AppleEvent handler, then AppleEvents generated by further voice input will queue up until the modal dialog and AppleEvent handler return.

Solving both of these problems requires that the activation of the object be performed after the AppleEvent handler has returned. This is accomplished by making use of the MCL system variable `*eventhook*` which contains a list of functions of no arguments. Each time MCL receives an event (not an AppleEvent), each function in the `*eventhook*` list is called until one returns a non-*nil* value. When the Voice Handler wishes to activate an object, it inserts the function which is to be called at the beginning of the `*eventhook*` list. The AppleEvent handler is then free to return, regardless of the behavior of the function to be called, and the function will be called at least as soon as the next clock tick event arrives. Since the MCL handling of regular events does not involve the problematic qualities of the AppleEvent handler, this approach removes both of the problems described.

10   Further Research Issues

10.1   Other User Interface Objects

Common user interface objects not implemented in the current Voice Toolkit include table-dialog-items as well as the menu-items and pull-down-menus associated with the menubar. The table-dialog-items were not included because they are not provided as a direct implementation with MCL 2.0, and therefore an application designer who has used them will have developed an individualized version, without knowledge of which it would be risky to attempt to maintain compatibility for a voice-type version.

Implementing voice-type versions of the pull-down-menus and menu-items was investigated; however, although methods seem to exist which provide access to the functions those objects use to respond to mouse clicks, they are undocumented within the commercial release of MCL 2.0. More complete information about those methods would be necessary before implementation of the corresponding voice-types could proceed. It may be noted, however, that Voice Menu Items would fall into the category of basic voice-type objects, and Voice Pull-Down

Menus should be hybrid voice-type objects.

10.2   Summarizing Experience

Regarding the storing of experience data, two issues might be addressed. First, as the implementation currently stands, the count of occasions on which a word may have been the correct identification for a mis-heard word has no upper bound in unbounded time. It may be the case that, in realistic use, this will never become a problem, but it might be desirable to enforce some large arbitrary upper bound at which the Voice Handler no longer bothers to collect further experience.

Second, rather than allow entries in the `*twintable*` to grow to large sizes which contain several heavily weighted values and many minimally weighted incidental values, each of which will be considered whenever a guess is generated, one might consider a mechanism to identify and prune incidental values from the table.

11 Related Work

Research in speech recognition has so far tended to focus on the mechanics of matching sound patterns, rather than on the utilization of the resulting ability. A search for relevant publications yielded only the work of the Speech Research Group within the Media Laboratory at MIT, which has developed systems called Xspeak and Xspeak II to provide a speech interface to X Windows [4]. The capabilities of these systems are very similar to that of the Voice Control software provided with the Voice Navigator II, and fall into the category we have described as a first approach.

Xspeak allows the user to navigate among XWindows by speech input, and requires the specific training of a name to be associated with a particular window. Xspeak II extends to allow the user to interact with the contents of the window as well. Using Xspeak II, a user can activate objects such as buttons through speech input; however, the activation is performed by specifying a location within the current window and actually producing a mouse event at that location in order to cause the application to respond. This retains the positional reference approach which is intuitively appropriate when a pointing device is in use; rather than the more language oriented approach we have chosen to use as more appropriate to speech input.

12 Conclusions

The use of an approach for speech input processing which is based on direct text label association rather than on macro assignment and object screen position is more natural to the medium of voice recognition and is feasible in terms of computational overhead and system response time. Making use of the object-oriented nature of window-based user interfaces allows this approach to be implemented fairly simply and integrated into applications with a minimum of required modification. Furthermore, allowing programmers to make use of existing speech recognition products at a level which is not specific to the particular device in use is more consistent with the way in which other input devices such as mouse and keyboard are used, and therefore likely to be more practical and useful.

Acknowledgements

References

[1] Articulate Systems, Inc., Cambridge, MA. *Voice Navigator Owner's Guide* , rev. b edition, 1990.

[2] Robert McCartney. Reasoning directly from cases in a case-based planner. In *Proceedings of the 12th annual conference of the Cognitive Science Society* , pages 101-108, Cambridge, MA, July 1990.

[3] Robert McCartney and Karl R. Wurst. DEFARGE: a real-time execution monitor for a case-based planner. In *Proceedings of the DARPA Workshop on Case-Based Reasoning* , pages 233-244, Washington, DC, 1991.

[4] Chris Schmandt, Mark S. Ackerman, and Debby Hindus. Augmenting a window system with speech input. In *IEEE COMPUTER* , pages 50-56, August 1990.

Appendix

A   Directions for Application Designer

The Voice Toolkit will be provided for inclusion on the MacHack '93 Proceedings Disk, and can also be had by contacting the authors at their E-mail addresses.

IMPORTANT: in the changes described below, be aware of the following:

1. Types may be changed to voice types or not as the designer sees fit; however, voice-type dialog items must be in voice-type windows if they are to respond correctly.

2. Voice-type objects are compatible with their standard versions for inheritance purposes. Specialized versions will inherit all voice capabilities and cooperate with the Voice Handler accordingly.

3. When changing types, change only direct references such as in calls to `make-instance`, `defclass`, `make-dialog-item`, and `type-of`. Do

not change function names; `window-show` does NOT become `voice-window-show`.

Make the following changes to integrate the Voice Toolkit into an application:

* Load the Voice Toolkit within the application.

* Change references to type `window` to `voice-window`.

* Change references to type `button-dialog-item` to `voice-button`.

* Change references to type `radio-button-dialog-item` to `voice-radio-button`.

* Change references to type `check-box-dialog-item` to `voice-check-box`.

* Change references to type `sequence-dialog-item` to `voice-sequence`.

* Change calls to `return-from-modal-dialog` which are made for Voice Windows to `voice-return-from-modal-dialog`.

* Add calls to `set-voice-handler` to tailor behavior as appropriate to the application.

* If the guessing option will be enabled, a call to `close-voice-handler` should be added at application shut-down so that gained experience will be saved.

B   Speech Recognition Product Set-Up

B.1   General Requirements

In order to properly interface a speech recognition product to the Voice Handler, the speech recognizer must be set up so that, for each word which is to be recognizable, speech input of that word causes an AppleEvent to be generated with application signature `CCL2` (Macintosh Common Lisp 2.0), type `aevt`, id `hear`, and the recognizable word as a string parameter. This assumes that the application will be running under MCL 2.0, in the case of a different MCL or a standalone application, the application signature should be modified accordingly.

B.2   Directions for Voice Navigator II or SW

Within the Language Maker utility [1], each word must be associated with a Voice Extension command. This is accomplished by choosing Voice Extension from the provided menu, then modifying the arguments within the parentheses as shown:

`@VXTN(GEVT,CCL2,aevt,hear ---- TEXT word)`

The arguments are explained as follows:

GEVT   Specifies the Generate AppleEvent Voice Extension.

CCL2   The application signature of MCL 2.0.

aevt   The type of the generated AppleEvent.

hear   The id of the generated AppleEvent.

----   A "direct object" involved in parameter passing.

TEXT   The type specification for the parameter.

word   The text string of the recognizable word.