Introducing VIPS: A voice-interactive processing system for document management

by Alan W. Biermann, Kermit C. Gilbert, and Linda S. Fineman

Duke University
Durham, North Carolina

ABSTRACT

The voice-interactive processing system enables a user to display office-related data on a screen and manipulate it through a combination of voice and touch commands. The system responds immediately to each request, updating the screen so that the correctness of each action can be verified. If an undesired result is achieved, the user may back up and restate the command in more exacting language.

The processor is a general system interface designed to handle various domains including text manipulation, file handling, calendar management, message passing, and desk calculation. Examples of its behavior in the text manipulation domain are given.
BACKGROUND

The voice-interactive processing system (VIPS) is a voice-driven natural-language processor designed to perform in several areas of application within the general framework of office automation. VIPS is aimed at the naive or casual computer user, such as an upper level manager, and its major goal is to improve the accessibility of office automation systems for such individuals.

The architecture of VIPS embodies our second thoughts on natural-language processing. It benefits from our earlier experience in building a natural language system for matrix calculations. A major design goal has been to concentrate the domain-dependent aspects of the system as much as possible, so that transitions from one application area to the next will not necessitate major changes to the code. The agenda for bringing office systems tasks within the scope of VIPS begins with text manipulation, continues to file management, and may eventually cover calendar management, message passing, and desk calculation. The text manipulation function is operative and is being tested currently. The modifications necessary to refit that system for managing a tree-structured file system are understood. Calendar management seems to pose no new problems, but that application and the remaining ones have not yet been examined closely.

SYSTEM FEATURES

In the text domain, VIPS allows the user to retrieve from a file a preexisting document, to enter a new document from a keyboard, to edit the document, and to store all or any part of it in a file. Commands take the form of English imperative sentences, optionally augmented by typed input or by touch input to a display screen. Spoken input is recognized by a commercial connected or discrete speech recognition machine. Recently we have obtained the most reliable performance from a Votan V-5000 discrete recognizer. Touch input is captured by a Carroll Touch Technology touch screen mounted on a large-screen color monitor.

Commands are uttered as a sequence of discrete words, using a vocabulary of about 100 words. The speech recognizer produces best and second best guesses for each word spoken and these are passed to the error-correcting parser, which attempts to identify an acceptable command utterance. Audio feedback indicates to the user those occasions when the recognizer is unable to make any guesses about the input. An utterance ends with the token, "over," as in "delete this word over." Other control words in the vocabulary are "correction," which indicates that the user wants to alter the current unfinished command, and "goodbye," which is the graceful way out of VIPS.

The touch input capability permits the use of very succinct commands, such as "put this sentence after that sentence," where "this" and "that" are instantiated by temporally appropriate touches to the display.

Keyboard input is used to instantiate string variables mentioned in a spoken command. These string variables allow the user to identify to the system proper names, such as names of files or segments of the document, which could not otherwise be referenced with the limited voice vocabulary. For example, the command, "insert x1 after each x2 in paragraph two," would require typed input for x1 and x2, where x1 is a string to be inserted and x2 is a string to be searched for in paragraph two. The user is prompted for this input.

The user brings a document into VIPS by issuing the command, "retrieve x1" and typing the name of the file (i.e., instantiating x1) containing the text of interest. Alternatively, a user may say, "enter x1" and respond to the prompt by typing a document directly to VIPS. A document is written back to a given file by the command, "store the document in x1," where x1 is the name of some (possibly new) file. Selected parts of a document can be written to a file. For example, "store the title of each section in x1" could be used to create a table of contents.

When a file is retrieved, a screenful of formatted text is displayed, starting at the beginning of the document. Each subsequent command that alters the document causes the updating of the display. To step through the document, the user says "goto paragraph two" or "goto the first subsection," etc.

Editing commands insert or delete text, move it from one place to another, or cause one string to replace another string. The system can be focused on a text object or a class of objects by the "consider" command. If VIPS fails to execute the user's intent, that fact becomes apparent when the display is updated. The previous state of the text, in that case, can be restored by issuing the "backup" command.

SYSTEM COMPONENTS

VIPS consists of four PASCAL modules and is designed to run on an IBM Personal Computer interfacing with an IBM 5520 office automation system. The four modules are composed of an ATN-style parser, which time-stamps and merges touch and voice input, prompts for and captures typed input (if necessary), and produces a parse tree that identifies all the constituents of the command utterance. Second is a translator, which accepts a parse tree, an array of touch coordinates.
(if any), and a package of typed input (if any). It produces a "bubble structure," described in detail later, which guides the execution of the semantics module. Next is the semantics module, which receives the bubble structure and any typed input, maintains the text-data world and the context of the user-VIPS dialogue, and effects the user's command by interpreting the bubble structure. The fourth module is the formatter, which receives the text-data world (or a portion of it) and formats it for display and printing after the execution of each command utterance, then returns to the semantics module a map of the updated display for the purpose of resolving subsequent touch inputs.

DESIGN CONSIDERATIONS

Text-Data World

The objects of discourse in our text domain form a logical containment hierarchy of characters, words, sentences, paragraphs, subsections, sections, and documents. Titles and other such entities are defined. A left- and right-bracket character pair is defined for each object class in the domain. When a document is first brought into the VIPS text-data world, a transducer identifies the objects in it according to a few, simple rules. This identification is preserved by enclosing the object in the appropriate left and right brackets and entering this marked-up object into a linear array. Single text characters are not bracketed as a matter of course, to conserve memory, but can be under certain circumstances.

In the text domain it is necessary to transduce a document once to establish the containment hierarchy. Violations of the hierarchy are possible and can handled by the system. For example, it might be reasonable to put a paragraph, perhaps some quoted material, inside a sentence. VIPS allows this, but it is the user's responsibility to decide whether or not such an act makes sense. Our goal has been to exclude knowledge about text from VIPS as much as possible, relying on the user's knowledge, the immediate updating of the display, and the back-up capability to keep the text-data world well formed.

Context Mechanism

The context for interpreting commands in VIPS is developed and preserved by a stack of "focus" lists. Each list consists of a set of pointers to a set of objects of some class, for example, pointers to some words or paragraphs. All the objects pointed to by a given list are actually contained by the objects pointed to by the list immediately below in the stack. The focus stack determines the order in which objects are searched for in the text-data world during the execution of a command.

An example will clarify the context mechanism. The command "retrieve xl," will cause a pointer to the retrieved document to be pushed onto the stack. A subsequent command to "consider the last two sentences" will cause the search for sentences within the object pointed to by the focus list at the top of the stack. In this case, the system looks for the "last two sentences" in the document. If two or more sentences are found, a pointer to each of the last two of them will be entered into a list and the list will be pushed onto the focus stack. The stack would then become as shown.

2. sent(last-1), sent(last)
1. document

If acceptable objects are not found, the focus stack is removed (or popped) and another search for appropriate objects ensues based on the list now at the top of the stack. In this example, after being popped once, the stack would be empty, and, thus, the search would terminate unsuccessfully.

Assuming that "the last two sentences" were found, if the next command is "capitalize the first character in those sentences," then a search of the focus stack will be made for a list of pointers to sentences. In this case, a list referencing two sentences will be found at the top of the stack and the first character of each of the sentences will be capitalized. Pointers to those two characters will then be added to the stack to yield

3. char, char
2. sent(last-1), sent(last)
1. document

If a command is given referencing objects not found on the top of the stack, the stack is popped until appropriate objects are found.

If touch inputs have been associated with the command utterance, then "those sentences" will be searched for in that part of the text-data world described by the display map generated at the end of the execution of the previous command. This is equivalent to searching the display itself. This type of search takes precedence over any use of the focus stack. When a successful search of this type is completed, the focus stack will have at its top a list of pointers to the touched sentences. Directly below that list will be a list pointing to objects that actually contain the sentences that were touched. The list below the touched sentences list frequently would point only to the document. However, if the touched sentences were in a paragraph, say, that was represented by a list already on the stack, then the pointer to that paragraph would remain on the stack immediately below the list pointing to "those sentences." Thus, touch processing wipes out only as much pre-existing focus as necessary to maintain the principle that an actual (narrowing) containment hierarchy is represented by the focus stack.

Before leaving this topic, it should be noted that no effort is made to develop a complete path of narrowing containment relationships on the focus stack. Only an actual path is desired. A stack with a pointer to single character at the top and a pointer to the entire document just below is often a sufficient representation of context in our scheme.

EXECUTION OF A COMMAND

Semantics execution will be illustrated for the utterance "print the title of each subsection in section two." The parse of the sentence will indicate that the verb, "print," has one operand, "title," and that the operand has one postnominal modifier,
"of each subsection," and that the modifier is postnominally modified by "in section two." The roles of the quantifier, "each," and the ordinal, "two" (i.e. "the second"), are also identified in the parse. The parse is translated into a network of nodes we have named the "bubble structure." By interpreting this structure, the semantics module achieves the intended result of the user's spoken command.

In the case of the example utterance, the following bubble structure guides semantic processing:

```
VERB(print)
  CONTAINEROF(section)
  TYPEGEN(section)
  APPLY(second)
  APPLY(the)
  TYPEGEN(subsection)
  APPLY(each)
  TYPEGEN(title)
  APPLY(the)
  COLLECT
EXECUTE
```

The initial command notes the imperative verb, "print," and sets it aside for later execution. The indented sequence of instructions (CONTAINEROF to COLLECT) finds the set of objects referenced by the noun phrase, "the title of each subsection in section two." A list of pointers to this set of objects is handed to the imperative verb for dereferencing in the final EXECUTE bubble. Noun group resolution involves finding, passing, testing, and collecting objects from the text-data world. In fact, it is always the pointer to an object that follows a data-driven control flow with objects precipitating down through the bubbles and collecting at the final COLLECT bubble. Typically to the preceding example.

The interpretation of the indented sequence of bubbles follows a data-driven control flow with objects precipitating down through the bubbles and collecting at the final COLLECT instruction. Two of the instructions, CONTAINEROF and TYPEGEN, are object generators. The APPLY instructions are filters that either delete objects or pass them along as they arrive.

The task of "CONTAINEROF(x)" is to find some object that has recently been mentioned in the dialogue (possibly implicitly) and which can contain an x. For example, a person might say "consider the title of the paper," and then say, "capitalize each word." In interpreting the second utterance CONTAINEROF(word) appears in the bubble structure. The CONTAINEROF function uses the focus stack, described previously, to help find the meaning of "word." Since the title has just been mentioned, and is on the focus stack, and since it does, in fact, contain words, we have (conceptually):

```
CONTAINEROF(word) = title of the paper.
```

The command, then, results in only the words in the title being capitalized even though there may be many other words in the environment.

Continuing with the earlier example, the bubble, CONTAINEROF(section), will find an object that does contain at least one section, say, the document. This object, the document, is passed to the second bubble, TYPEGEN(section). This bubble has the task of generating all possible objects of the type, section, from the object it received, that is, from the document. The first section is passed down through the lower bubbles in the structure, then the second section, and so forth. This continues until either all sections in the container have been generated or until the TYPEGEN bubble is turned off.

The APPLY bubbles filter objects passed to them. APPLY(second) will absorb the first object that arrives and pass the second one, and then turn off the generator above it. APPLY(the) is largely a clear passage for all objects except that it does check that the correct number are passed; exactly one for a singular definite noun group, for example.

When a section arrives at the TYPEGEN(subsection) bubble, processing similar to that described above, finds subsections and passes them down to the APPLY(each) bubble, which in turn, passes each one down to the TYPEGEN(title) bubble. Here titles within subsections are generated and passed through the APPLY(the) bubble to the COLLECT bubble where the set of titles is accumulated. Finally, the EXECUTE bubble is handed the imperative, "print," and the set of titles and prints the items in the set.

This model of semantics is broadly applicable to office automation domains such as file manipulation, calendar management, and desk calculations that have hierarchical organization similar to the text example given here. For example, in the calendar domain, the sentence "list the first appointment in each day of the second week" would be processed identically to the preceding example.

HUMAN FACTORS

At the time of writing, the VIPS system is not ready for human-factors testing, but we expect it to outperform its predecessor, VNLC, on most dimensions. In problem-solving sessions, users speak to VNLC at the rate of about one word per second, and they utter several sentences per minute. Error rates from the speech equipment have been high—on the order of 10%—but system error correction has reduced this rate significantly. The VNLC system executes about 75% of user commands immediately and correctly with most errors caused by voice misrecognition.

RELATED WORK

A number of projects have developed natural language database interfaces, but few have built task-oriented processors of the kind we describe here. There also have been many projects over the past two decades in speech technology, where the goal has been to learn how to build voice recognition equipment. Our project seeks not to develop a voice recognizer but to use existing recognizers efficiently with a well-designed error-correcting natural-language processor.

ACKNOWLEDGMENT

This work has been supported by the IBM Corporation GSD Agreement No. 260880.
REFERENCES


