Accepted American Voice Input/Output Society (AVIOS 99), San Jose, CA. 5/26/99
pp267-276

Hearing What Speech Recognition Can't:
A Language to Understand Your Users' Behavior

Christine A. Halverson, Daniel B. Horn*, Clare Marie-Karat, and John Karat

IBM T.J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532 USA
+1 914 784 7751
halve, ckarat, jkarat@us.ibm.com

*University of Michigan
Collaboratory for Research on Electronic Work
701 Tappan Street, Room C2420
Ann Arbor, MI 48109-1234
danhorn@umich.edu

Accepted American Voice Input/Output Society (AVIOS 99), San Jose, CA. 5/26/99
pp267-276

Abstract

Speech input still lags behind keyboard and mouse as an input modality for computers. While advances in automatic speech recognition (ASR) technology are significant, the integration of speech input into applications and their ultimate usability could be greatly improved. In this paper, we present our approach to discovering usability issues in ASR systems.

Our understanding of these issues is a result of our evaluation study of three desktop speech systems. We performed an activity focused study where all sessions were videotaped and an in-depth analysis completed. During analysis, we developed a coding scheme that is particularly useful for examining desktop speech dictation systems. Like a language, codes are abstractions that allow you to describe user behaviors. We argue the focus on a realistic speech recognition activity, coupled with in-depth examination of use patterns as abstracted through our coding, helped expose usability problems that would otherwise be hidden. By example we demonstrate how this analysis helped us understand user problems with speech applications, and how this understanding can help aid design recommendations.

1. Introduction

Despite significant accomplishments during the last three decades, speech input has still not gained widespread acceptance as a primary input modality for computers. One reason why user acceptance lags behind technical accomplishment is the usability of desktop speech software. While we like to think that speech is a natural form of communication (Clark & Brennan, 1991; Oviatt, 1995) it is misleading to think that this means that it is easy to build interfaces that will provide a natural interaction between human and machine (Yankelovich, et al, 1995).

One of the problems with evaluating speech systems is determining where the problems lie. Speech systems introduce an additional layer between the user and her understanding of the application because of the step of interpreting the acoustic signal as text. This is in contrast to the direct manipulation nature of keyboard interfaces. This added indirection causes problems for both the user and the developer.

For example, when a user presses a sequence of keys on a keyboard she is certain of the result. If she types Q U I T followed by enter and the wrong thing happens, she (and we as observers) can be sure that the error is in the application. In contrast, input to an ASR system may result in errors that are not as easy to evaluate. In the example above, imagine the user had said “quit” and nothing happened. If she is interacting with a dictation application, then perhaps the word “quit” appeared in the dictation window. Or perhaps, if there is a place for feedback from the speech engine it shows “cute”, or another incorrect word, or a phrase like “misrecognized command”. If there is no feedback, or if it takes a while before she realizes her command has not been acted upon, then she may have no idea what is going on. Alternatively, it may appear that the application is unresponsive. The developer is left similarly puzzled.

To better understand the usability problems with speech, we conducted a study to evaluate user performance and satisfaction during completion of text creation tasks using three commercially available continuous speech recognition systems (Karat et. al., in press). Direct user testing was undertaken because it uncovers problems that heuristics or usability

walkthroughs will not (Bell et. al., 1991; Doubleday et. al., 1997). Further, a detailed analysis uncovers important details of users' interactions that may be missed using only discount usability methods.

Our approach was to have users perform a complete activity based on one expected use of the software – in this case composing an email reply. In this way, we are more likely to discover problems that would only arise in the context of use. During test sessions we coded users actions and behaviors as much as possible. However, we discovered that too much was happening to even see and record, much less understand. To better understand use patterns and problems we performed a subsequent in-depth analysis of test session videotapes.

A key part of the analysis was developing a coding scheme – an abstraction that makes it easier to see patterns and relationships in the data – that is particularly useful for analysis of speech software. First, we provide a brief description of the study and its pertinent features. Then we present an explanation of coding and an introduction to our scheme and its structure. Next, we illustrate the utility of the coding scheme by explaining an example coded log. We follow this with a discussion of advantages and drawbacks of our approach and coding scheme.

2. Our Approach

2.1. Studying dictation systems for text creation

Our coding scheme arose from a competitive evaluation of three commercially available ASR systems that shipped as products in 1998. These systems – IBM ViaVoice 98 Executive, Dragon Naturally Speaking Preferred 2.0, and L&H Voice Xpress Plus – all share some important features. First, they all recognize *continuous* speech as opposed to forcing the user to speak "discretely" with pauses between words. Second, all are *modeless systems*. Each application has integrated command recognition into the dictation so that the user does not need to explicitly identify an utterance as text or command. To do this the user must learn a command grammar (a list of specific command phrases), and sometimes a keyword that is uttered to indicate that what follows is a command. In general, commands must be spoken together as a phrase, without pausing between words, in order to be recognized. Otherwise, the words are treated as text.

We were primarily interested in whether speech input was substantially different from keyboard and mouse input (K/M) for text creation tasks. For this reason, we focused on two kinds of text creation: transcribing existing text and composing new text. For the text composition task we used a realistic email task. The study compared speech and K/M as input modalities for both kinds of tasks across all three speech recognition products. (The study itself was designed for statistical comparisons between the three systems. We balanced for age, gender, modality, and kind of text creation and accounted for potential order effects. Details are available in Karat et. al., in press.

We videotaped all sessions, and our analysis included coding of all of the pertinent actions carried out by subjects in the study. We paid particular attention to the interplay of text entry and correction segments during a task, as well as strategies used to make corrections (see Halverson et. al., in press). Our coding scheme, comprised of around 100 codes, reflects both the focus on text creation (i.e. dictation applications) and the importance of error correction for speech applications.

2.2. What is a coding scheme?

Codes are simply names that abstract and represent important observed behaviors. In general, these abstractions reduce the data, which makes it easier to see patterns and relationships. There is a long history in many sciences of reducing data in just this sort of way. For example, taxonomies in biology arrange hundreds of species into groupings and categories, which help differentiate commonalties and differences much more quickly. Social and behavioral sciences have successfully adapted these ideas to the observation of human and animal behavior for about a century. Where biological taxonomies categorize based on all known observations of animals, behavioral science categorizes based on all observed behaviors. In this case, the observed behaviors are restricted to the scope of the test.

The coding scheme we developed does several things for us. First, it reduces the extent of the data and makes it easier to review. With the abstraction imposed by a coding scheme, it is easier to compare and group instances into like categories than when referring to the original videotape or a detailed transcript. Second, the abstraction into codes and the accompanying data reduction helps to order the data. This can result in seeing patterns that are not otherwise obvious. It can also help group, collate, and quantify data in order to understand the extent of particular problems. In addition, codes allow one to conduct automated searches of logs, identifying similar actions or problems.

Third, the process of coding forces you to decide whether a particular occurrence fits one pattern or another. Making this kind of distinction teaches the analyst a lot about where problems arise. (This also presents the danger that you won't *see* phenomena that are not covered by your codes. We'll discuss this more below.) In the case of speech, this is particularly important because of the confusion between problems in the engine versus problems in the application. In addition, this deeper knowledge can help prioritize the severity of specific problems. Finally, the process of applying the codes and extending them to accommodate different features gives added insight into redesign effort should be directed.

2.3. Coding scheme overview

While our coding scheme arose from the data in our study, its overall structure reflects important distinctions necessary in thinking about the usability of speech based systems. At the highest level user behavior is distinguished in three useful ways: basic user actions, user problems, and system problems. We present user actions first, then user and system categories.

2.3.1. User Actions

The term—*user actions*—refers to what users do with the application and hardware in order to accomplish their tasks. What counts as an action is connected with the available hardware, input modalities, and application supported actions. For example, all three systems we studied allow interaction by K/M as well as voice input via microphone. Microphone use is not a common skill, and our prior experience suggested that we should follow mike use carefully (Lai and Vergo, 1995). Thus we tracked mike operation: on, off, and the start of dictation.

Our focus on the interplay between text creation and correction meant we were interested in all actions that related to error detection and correction. For K/M entry, correction can be made by backspacing and retyping, by selecting the incorrect text and retyping, or by dialog

techniques generally available in word processing systems such as Find/Replace or Spell Checking. There are some parallels for error correction in ASR systems. By monitoring the recognized text, users can correct misrecognitions with a speech command equivalent of "backspacing" (current systems generally have several variations of a command that remove the most recently recognized text – such as SCRATCH or UNDO). There are ways of selecting text (generally by saying the command SELECT and the string to be located), after which redictating will replace the selected text with newly recognized text. Additionally, correction dialogs provide users with a means of selecting a different choice from a list of possible alternatives or entering a correction by spelling it.

To handle these different methods we came up with action codes for both K/M and voice input which delineate the separate parts of error correction—getting to the word and operating on it—and are comparable for both modalities. In addition, although we did not use a word processing program that allowed operations like Find/Replace, we did need codes to describe use of the correction dialogs (a feature in all three applications). We also needed commands to handle common actions such as formatting text, again for both modalities.

2.3.2. User and System Categories

While there are 28 codes for the basic actions, there are many more needed to account for the user and system behaviors observed. Here again, the codes are indicative of the behavior observed. We distinguish user categories from system categories based on where the problem is observed. Most (if not all) usability problems are ultimately problems in the design or understandability of the system; however it is still useful to separate them out. We use the term "System" to include application software and hardware.

In speech systems there is an additional distinction between the speech engine (and any hardware dedicated to that purpose) and the application that uses speech as input. Separating engine problems from application problems is difficult but essential. Users are often completely unaware of the distinction, but understanding where a problem occurs is vital for knowing how to fix it.

USER CATEGORIES	**SYSTEM CATEGORIES**
Microphone	**Issues**
Microphone Control	Microphone movement
	Extraneous sounds
Dictation & Command	**Issues**
Original Dictation	Misrecognitions
Command and Control Issues	
Faulty Command Models	
Correction	**Issues**
Problems using the correction box	Bugs in correction box
Cognitive Issues	**& Indicators**
General Indicators	
Patterns of use	
Other	**Issues**
Interface effects	Inconsistent system behaviors
Use of Help	General issues
Enrollment	

Table 1. Broad categories and functional divisions within user and system problem categories.

Table 1 summarizes the main divisions that cover the over 70 codes in these two categories. For dictation applications there are five main groupings of issues (and codes), as

follows: microphone related; dictation and command; correction features; cognitive issues; and other general issues. These groups span hardware, software, and fundamentally human problems. Within each group there are further divisions that group codes into functional areas. These functional divisions help pinpoint problem areas. Within each segment are the codes themselves. The scope of these codes and how they are applied will become more apparent in the following example.

3. Case Study

Users, like experts, have difficulty articulating what they do and why things work or don't. Nevertheless, if they can not always speak to us about where the problems lie, they can show us if we look closely. We present one episode from our logs, which shows the power of coding in two distinct ways: identifying exactly what the error is, and how this identification helps to formulate a design response.

3.1. Understanding where the error really is

The process of applying (or creating) a coding scheme forces you to examine why you choose to categorize a behavior in one fashion or another. This has the added effect of helping you understand where to fix users' problems: in the speech engine, in the application, in the underlying command structure, or through user training—to name a few. Our initial approach focused on problems with features—broad categories such as command languages, or the switch to modeless commands and the accompanying problems. As we reviewed the video, we quickly realized the depth and breadth of problems associated with commands. Being able to detail issues more explicitly helped us understand how to fix them. We'll use an example to make this clear.

Our user, Beth[1], is trying to correct an error using the correction dialog. Overall, she is attempting to select a word and open the correction dialog. This dialog box presents her with a list of alternate words. With one command, she can choose one of the words listed and it will replace the selected word in the text. A separate command hides the window associated with the correction dialog. She uses several commands that she had previously used with success, but in this case, the result is not the same.

Figure 1 shows a transcript of what she says juxtaposed with what she expects to happen and what does happen. She successfully selects the word to be corrected and opens the correction dialog. However, repeated attempts to make the correction seem to elicit no response from the application, and result in mounting user frustration. What's going on here?

[1] Not her real name.

	Beth says	What she expects	What she sees
	SELECT doors	Word "doors" will be highlighted	"doors" is highlighted
	CORRECT THIS	Correction dialog should open and display a list of alternates	Dialog opens and displays list
a	PICK ONE HIDE CORRECTION WINDOW	First word on list is chosen, substituted for highlighted word and dialog box goes away	Nothing happens
	CANCEL	The dialog box goes away	Nothing happens
a	CANCEL HIDE CORRECTION WINDOW	The dialog box goes away	Nothing happens

Figure 1. Example of problems using a correction dialog box.

Everything Beth says is a command, so we can assume that is where the problem lies. However, there are several competing options. One possibility is that the command is misrecognized. Another possibility is that she is saying the wrong, or incorrectly saying the right command. Perhaps, since there is no response, the command is not being *heard* at all – possibly because of a hardware problem. How do we distinguish between these?

We start by ruling out the most obvious—misrecognitions—which would be a system error. We would expect the command to produce the effect of selecting the first word in the alternates list, and it doesn't. This suggests it is a misrecognition (**MisRec**) of a command (**Cmd**) as dictation (**Dict**)[2]. However, we see no response from the application. Because it is happening inside a correction dialog, we can't observe the misrecognition appear in the dictation window. In addition, because of the way the correction dialog works, the misrecognition (if it exists) is blocked from appearing at all. For now that seems to end the possibility of a system error.

Switching focus from system to user categories, perhaps she is not using the correct command. Lines 3a, 3b, and 5b in figure 1 are all valid commands, however, lines 4 and 5a are not. How these commands are invalid provides us important information about users' expectations and understanding of the program. Therefore, it is important to distinguish among the different possibilities. We might code these invalid commands as a user problem of using the incorrect (**Incor**) command (**Cmd**), in this case leaving off the required deictic "that". If this is the case we would code line 4 as **USABILITY Incor Cmd Deic miss**. This would put our focus on the user. If there are lots of problems of this kind we might take a closer look and decide that the command structure itself was problematic. We need to be careful because command problems are often more subtle than just misremembering a command.

Using the code explained in the previous paragraph implies that the command CANCEL THAT works inside a correction dialog. While CANCEL THAT does work in this program, unfortunately it is not a valid command within the correction dialog. Thus, we must adjust the coding to reflect that the user has a poor (**Faulty**) mental model about the command (**Cmd**). This means that we take this as evidence that the user does not understand how CANCEL THAT works as a command, as well as where it works. Because this particular confusion (trying to use CANCEL THAT inside a correction dialog) does not often occur, there is no dedicated code. Therefore, we code it as **USABILITY Faulty Cmd Other**.

[2] If this were the right code (**MisRec Cmd Dict**) it would be important for us to understand more about the misrecognition. So the coding scheme delineates if the observed dictation is the same words as spoken (**right words**) or different (**wrong words**). This assumes that we can see the result of the misrecognition.

The importance of the distinction between incorrect commands and faulty commands relates to why users have trouble. For example, repeated evidence of forgetting the deictic or pointing word of the command might suggest changing the command. Continued misuse of a command in a single area where it is not valid, like this example, suggests inconsistency in how the command operates overall. This would likely require deeper changes to the application's structure—for example making the command work in the correction dialog.

This still leaves us trying to understand lines 3a and b, and 5b, since these are all valid commands. One possibility is how she says the commands. Remember that these are *modeless* systems where command phrases must be spoken as a phrase to be recognized as a command. When the user pauses between words, it results in the speech engine interpreting the command as dictation (**USABILITY Pause**). We distinguish this from a misrecognition because it is the user who is at fault here, not the speech engine. We can not easily see if this is the case because she is not working in the dictation window.

Another possibility for misrecognition is if she does not pause, even between the two commands. If we compare the cadence of how the commands are spoken we can tell that this seems to be the case. Reviewing the tape we now understand that Lines 3a and b are run together into one command, as are lines 5a and b. This profoundly changes our view of what is happening, in addition to requiring a change in the coding. Figure 2 presents this revised view with the code **USABILITY No Pause** now assigned to lines 3 and 6. (What was lines 5a and b is now coded as 2 codes in 5 and 6, recognizing the problem with CANCEL THAT, as well as this problem.)

	User Says	CODE		Comment
		Umbrella	**Major / minor**	
	SELECT doors		VOICE select	
	CORRECT THIS		cor-open	
	PICK-ONE-HIDE-CORRECTION-WINDOW	USABILITY	No Pause	Strings two commands together without any pause
	CANCEL	USABILITY	Faulty Cmd Other	wrong command - nothing happens
	CANCEL-HIDE-CORRECTION-WINDOW	USABILITY	Faulty Cmd Other	wrong command
		USABILITY	No Pause	-and without pause takes second command as garbage, tries to put into entry window of correction dialog as dictation

Figure 2. Coding of example from figure 1.

Understanding the root of her error is important for two reasons. First, in order to fix the problem, we must understand what to fix. Codes which identify how she says the command (**USABILITY No Pause**) emphasize a completely different aspect than the codes which focused on user misunderstanding of the commands. In this case our attention is directed to how to train the user—not only about how to dictate but also how to say, and how not to say, the commands.

We chose this example because it was concise and provided a way to show how to work through a variety of coding options. It is clearly not an exhaustive demonstration of all the

codes. The example illustrates how we choose which code to apply in a particular circumstance. Additional context is important, whether derived from the user's behavior or the application.

This example is also illustrative of how coding specificity is important to understand the range of problems associated with a particular application area—in this case commands. The example highlights some of the problems of a modeless system. In considering alternative codings for Beth's problem, we also raised the issue of user misunderstanding of how and where commands work, as well as not remembering the command at all. Finally, we have illustrated how the specificity of the coding reveals different options for developers.

4. Discussion

We have presented an example that illustrates how our coding scheme helped us understand subtle (and not so subtle) problems in desktop speech applications. As a process, determining the codes helped clarify where problems occurred—distinguishing between application, engine, and user. We have also shown how it is important to clarify exactly what is happening in order to understand what is an appropriate response.

Our analysis of our product (ViaVoice) in the competitive evaluation study (Halverson et. al., in press; Karat et. al., in press) resulted in two dozen specific recommendations. Recommendations were split between application specific and engine specific. Of these, six immediately made it into the product and another five are included in the product upgrade due at the end of the year. A large part of this success is due the care taken with this analysis.

However, there are several negative issues of which to be aware. Detailed analyses of this sort are both time and labor intensive. Overall, our coding scheme consists of approximately 100 codes. In our competitive evaluation (Karat, et. al., in press) we coded over 6500 individual events for 12 subjects covering both speech and K/M text creation tasks. This included a coding of all of the pertinent actions carried out by subjects in the study. We coded misrecognitions of text and commands and attempts to recover from them, along with a range of usability and system problems. Particular attention was paid to the interplay of text entry and correction segments during a task, as well as strategies used to make corrections. Because of the extensive time required to do this, we completed the detailed analysis for only 12 of the 24 subjects we studied.

One reason it is so time intensive is that we are applying the codes post-test by reviewing the video tape. One argument is to streamline the coding scheme so that coding can be done "real-time" as the test is conducted. In our experience, this would be extremely difficult for two reasons. The pace of problems is often too fast to code and document completely. In addition, many problems appear the same on the surface, and it is only by reviewing a session segment repeatedly that the analyst catches the subtle details and bits of information that eventually distinguish between one code and another.

A second problem is that having a predefined coding scheme may blind you to behavior or issues that do not fit your scheme. We would caution that you should always realize that data might crop up that does not fit the categories you have determined. These are opportunities to better understand what is going on in your application and users' interactions with it.

5. Conclusion

Improving the usability of ASR systems is the challenge facing us. We embarked on a quest to discover the usability issues associated with desktop dictation systems, armed with our usability evaluation skills. We feel our activity based approach, coupled with a fine-grained analysis made possible by our coding scheme, exposed usability issues that might not have otherwise been seen. To demonstrate this, we presented the basic structure of our coding scheme and discussed its application through an example from our data.

Our coding taxonomy captures user and system issues unique to the integration of speech with a functional application, such as word processing. It could be used as a starting point for coding other experiments involving speech user interfaces, saving some analysis time. While studying users directly is preferable, it may not always be feasible. Our coding scheme may also provide a more detailed start for a speech-specific usability walkthrough.

Finally, we feel the process of creating the coding scheme, and applying it, helped alter our perception of the issues in desktop speech systems. By categorizing and naming things, one develops a richer understanding of the material. Specialization leads to development of specific nomenclature. If standard taxonomic approaches (ours or others) are adopted, we will have a richer base from which to understand and discuss human-computer interaction.

References

Bell, B., Rieman, J., & Lewis, C. (1991). Usability Testing of a Graphical Programming System: Things we missed in a programming walkthrough. In *Proceedings of CHI '91*, ACM Press, pp 7-12.

Clark, H. H. & Brennan, S. E. (1991). Grounding in communication. In J. Levine, L. B. Resnick, and S. D. Behrand (Eds.), Shared Cognition: Thinking as Social Practice. APA Books, Washington.

Doubleday, A., Ryan, M., Springett, M., & Sutcliffe, A. (1997) A comparison of usability techniques for evaluating design. In *Proceedings of DIS '97* (Amsterdam, NL). ACM Press, pp 101-110.

Halverson, C., Horn, D., Karat, C-M. . & Karat, J. (in press) The Beauty of Errors: Patterns of error correction in desktop speech systems. In *Proceedings of INTERACT '99* (Edinburgh,Scotland, August 1999).

Karat, C-M, Halverson, C., Horn, D. & Karat, J. (in press) Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *Proceedings of CHI '99* (Pittsburgh, PA, May 1999), ACM Press.

Lai, J. & Vergo, J. (1997). MedSpeak: Report Creation with Continuous Speech Recognition, in *Proceedings of CHI '97* (Atlanta GA, March 1997), ACM Press, 431 - 438.

Oviatt, S. (1995). Predicting spoken disfluencies during human-computer interaction. Computer Speech and Language, *9*, 19-35.

Yankelovich, N., Levow, G. A., & Marx, M. (1995). Designing SpeechActs: Issues in speech user interfaces, in *Proceedings of CHI '95* (Denver CO, May 1995), ACM Press, 369-376