Putting People First: Specifying Proper Names in Speech Interfaces

Matt Marx *Chris Schmandt*
marx@media.mit.edu geek@media.mit.edu
+1 617 253 9848 +1 617 253 5156
Speech Research Group
MIT Media Laboratory
20 Ames St.
Cambridge, MA 02139

ABSTRACT
Communication is about people, not machines. But as firms and families alike spread out geographically, we rely increasingly on telecommunications tools to keep us "connected." The challenge of such systems is to enable conversation between individuals without computational infrastructure getting in the way. This paper compares two speech-based communication systems, Phoneshell and Chatter, in how they deal with the keys to communication: proper names. Chatter, a conversational system using speech-recognition, improves upon the hierarchical nature of the touch-tone based Phoneshell by maintaining context and enabling use of anaphora. Proper names can present particular problems for speech recognizers, so an interface algorithm for reliable name specification by spelling is offered. Since individual letter recognition is non-robust, Chatter implicitly disambiguates strings of letters based on context. We hypothesize that the right interface can make faulty speech recognition as usable as TouchTones – even more so.

KEYWORDS: speech recognition, error-repair, user interface, conversational systems.

INTRODUCTION
A familiar puzzle goes like this: "If a tree falls in a forest but no one hears it, does it make a sound?" Philosophical considerations aside, the point is that the person makes the event meaningful. What makes communication meaningful is the interaction of two people who share ideas in order to maintain a common reality. To recast that puzzle in the context of communications, "If a message is sent but no one receives it, does it mean anything?"

Names as protocols
We establish contact in face-to-face communication by saying someone's name. Names are, in effect, the addressing protocol of conversation. When technology facilitates communication, however, we must employ different addressing protocols such as phone numbers or email addresses which may bear little or no resemblance to the person's name. Since these data are only pointers, remembering them is often difficult. Looking up someone's phone number or email address can be tedious: hence the proliferation of "quick dial" buttons on telephones and "alias" features on email programs. We want to refer to people by their names, not by their numbers or handles: I don't say "call 6172539848" but "give Matt a call"; likewise, I'll say "I got email from Don" rather than "I just received a message from 1194822.345@compuserve.com."

There is perhaps a deeper reason that these telecommunications pointers can be so bothersome. Emmon Bach points out [1] that names are *rigid designators* -- that their referents do not and cannot change. A name is part of an identity, part of who we are. Telephone numbers, by contrast, are reassigned as customers relocate; thus they are "flexible designators" at best (again, to use Bach's terminology). An email address, too, may have a limited life span. Focusing on the number or address instead of the name, as is inevitable with telephone or email usage, complicates communication.

Speech: a control channel for communications
As personal communication services become ubiquitous, so do their accompanying computational tools. The proliferation of laptop computers and Personal Digital Assistants (PDAs) indicate a desire for on-demand personal information. A parallel phenomenon is telephone access to everything from bank account balances to movie schedules via TouchTones.

Both telephones and PDAs suffer from impoverished interfaces, however. The shrinking screens and chiclet keyboards of PDAs limit the use of eyes and hands; any fingernail-biter who has worn a calculator wristwatch can attest to the difficulty of jabbing at tiny keys. Even worse is the standard telephone, with no display and only twelve keys.

Speech offers rich interaction with minimal space requirements. We speak faster than we can type, and the

words we speak can describe an infinity of concepts. Speech has been shown to be an interactive and expressive medium due to the various linguistic and auditory cues available. [2] There is little learning curve since we are all acquainted with conversation beginning with our earliest years. Further, the hardware required to facilitate voice control is minimal. A remote device captures audio for processing at a workstation elsewhere; as computing power becomes more compact, voice-controlled PDAs may shrink to the size of a microphone.

If speech is a powerful interaction modality for personal communication services, and if proper names are so crucial to communication, then it follows that speech user interfaces must deal with names efficiently. The remainder of the paper concerns the question of handling proper names in telephone-based speech interfaces for communication systems. Phoneshell demonstrates name specification in conventional touch-tone-based interactive voice response (IVR) systems, but its necessarily hierarchical interface interferes with the ability to deal with multiple applications. Improvements offered by the speech-recognition-based system Chatter are discussed along with the consequences of error-prone recognition. Finally, the problem of faulty proper name recognition is compensated for by an algorithm for reliably spelling names letter-by-letter (despite poor letter recognition).

NAMES IN A TOUCHTONE INTERFACE: PHONESHELL

Phoneshell is a touch-tone interface to personal communication services including email, voice mail, rolodex, calendar, news, and phone dialing. [6] During a single phone call, the user can listen to a message (email or voice) inviting him to lunch, check and update his calendar with the new appointment, and call a third party to invite her along. The continued use of Phoneshell -- it handles upwards of 25 calls a day from the dozen members of the Media Lab speech group -- demonstrates the utility of speech interfaces for personal communication systems.

Keypad-based name specification

Since the user must likely deal with more names than there are keys on the telephone, and since waiting to hear the desired name a long list is tedious, the only sensible way to specify a name using TouchTones is to spell it out. But touch-tone spelling is tricky due to the mapping of letters to the telephone keypad. Three letters are assigned to each of the keys 2-9 as is printed on most telephone keypads; the missing ‘q’ and ‘z’ are assigned to 8 and 9 respectively. Thus the system must disambiguate the keys typed by the user. The user enters a number of keys and Phoneshell attempts to ferret out the intended name; if there are multiple matches, it asks the user to choose between them. In the example below, we want to specify the name Edwards, which on the keypad is spelled 3-3-8-2-7-3-7.

Significant is the fact that not all six letters have to be entered for the system to figure out which name the user intended. By intelligently whittling down the list to only the relevant choices, the user is spared unnecessary work in typing keys which are not needed in order to uniquely specify a name.

The success of keypad disambiguation depends on several factors including the size of the database and the length of

NAMES	KEYPAD	LETTERS
Bennett Davis Dennison Deveraux Edwards Ellison Franklin Nelson Smith Smithson Thomas	3	d, e, f
Davis Dennison Deveraux Edwards Ellison Franklin	3	d, e, f
Dennison Deveraux Edwards	8	w, x, y
Deveraux Edwards	2	a, b, c
Edwards		

Figure 1: implicit disambiguation of successive keys narrow the list of possible names to one.

the names in that database. A study of disambiguation [3] showed the chances of different names mapping to the same touch tone keys to be no worse than the likelihood of finding multiple individuals with identical last names; in either case, user effort is required to identify the desired name from the two or three duplicates/matches. From our experience with Phoneshell, most names in a rolodex of approximately 100 people can be specified with four keys or less. The process of specifying a name in Phoneshell is reliable and rather expedient.

Hierarchical interaction model

Any touch-tone application with more commands than available keys must be segmented. Like most IVR systems, Phoneshell presents the user with a menu, each option leading to another menu, and so on until the actual command can be invoked. As each keypress drops the user into a lower level of the hierarchy, an escape key ‘#’ is provided to let the user step one level back up the hierarchy. This hierarchical structure forces the user to navigate out of one set of choices and then into another set with a series of keypresses.

Although Phoneshell supports multiple applications, it maintains no context between them. In this respect it is

like running multiple applications in a window system and mousing across windows. Yet the visual paradigm of the window system is absent from the speech-only interface, placing a greater cognitive burden on the user to keep track of state. Losing one's place is of far greater concern.

Now, it is conceivable that Phoneshell could keep track of some context between applications, remembering the last name specified. But how would one recall that name? Designating a special "use the most recently specified name" key is not an option since in every Phoneshell application, each key already has a function assigned to it.

KEY	ACTION
5	enter email reader
2	read message from Jeff
#	exit message
#	exit email reader
2	enter rolodex
jef*	find Jeff
9	enter call mode
2	select home number
#	cancel call (line busy)
#	exit rolodex
1	enter voice mail
2	send message
jef*	send message to Jeff
<record message>	
3	confirm sending

Figure 2: in a hierarchical touch-tone interface, the name "Jeff" must be specified twice, once for each of two consecutive tasks.

The anaphoric reference could only be done in the same way as normal specification -- that is, by specifying out an anaphoric "word". For instances, one could type "11*" -- which doesn't spell anything normally, and even if 'q' and 'z' are assigned to the '1' key spells only QQ, QZ, ZQ or ZZ, very unlikely names -- could recall the last name the same way that "him" or "her" recalls the last person in the discourse. Yet a special code like this is unnatural and may lead to more confusion than it alleviates. Further, the hierarchical structure renders context-keeping unnatural: a user, consciously exiting one application and entering another, may simply not expect the system to have kept track of the last name specified.

Since Phoneshell does not maintain context between tasks, a name must be re-specified for each new task. For instance, to log in, read email from Jeff, attempt to call him at home, and then send him a voice message requires several steps.

The user has to specify Jeff's name twice in order to perform two tasks relating to him. The navigation and re-specification necessitated by the hierarchical structure and the lack of memory require the user to invest a significant amount of effort in the mechanics of the application. Focusing attention on such "application overhead" draws the user away from the task at hand: communicating with people. The user wants to deal with Jeff, not a suite of applications.

NAMES IN A CONVERSATIONAL INTERFACE: CHATTER

Chatter improves upon Phoneshell by refocusing the user's attention on people. Although the functionality of Chatter parallels that of Phoneshell, offering access to voice mail, email, rolodex and phone dialing, its interface is vastly different. Using real-time speaker-independent speech recognition, Chatter carries on a conversation with the user to accomplish the desired tasks.

Keeping track of context

Any tolerable conversationalist has an idea of who and what is being talked about. Chatter keeps track of the conversation by means of a discourse model, an implementation of the Grosz/Sidner discourse theory. [4] The basic notion is that any utterance has a *structure*, a *purpose*, and a *state*, and that each of these need to be captured in order to facilitate conversation. For instance, the utterance "send Jeff a voice message" has the *structure* of [VERB PERSON DET ADJ NOUN], the *purpose* of starting a sequence of recording a voice message and then delivering it, and the *state* of making Jeff the topic of discourse.

Chatter captures each of these three aspects in a set of dialog segments, which are C++ classes with methods for deducing the purpose described by the structure and then executing it, and with variables to keep state. In the above example, for example, the utterance is processed by a ComposeSegment. The structure is interpreted by the segment's parsing method as a command to record a voice message, so the method for recording a message is executed. At the same time, when the person is specified, Chatter sets a state variable in a PersonSegment to indicate that Jeff is the most recent person specified. [5]

Segments exist for logging in to the system, reading messages, composing messages, placing calls, accessing the rolodex, and hanging up. At a given time, one or more of the segments may be available to process the user's utterance; the user can issue a desired command without having to navigate menus or explicitly switch applications. In the middle of reading the day's email messages, for instance, the user can ask, "Where's Chris? What's his work phone number?", call him and then return to reading the email messages where he left off. There is no distinction between applications in Chatter; instead, the different functions the user can perform are tasks which can be performed or interrupted and returned to later. The user need not exert effort to keep track of his place in the dialog; Chatter does that automatically.

Anaphora

The discourse model keeps track of names in the conversation. Since Chatter keeps state, it knows who is

being talked about. In conversation, we use a pronoun or *anaphor* to refer to a person who has already come up in conversation: "I saw Jennifer at the party over the weekend and talked to *her* there, and then I met *her* at work on Monday, where I gave *her* the paper I promised." This is more concise than "I saw Jennifer at the party over the weekend and talked to Jennifer there, and then I met Jennifer at work on Monday, where I gave Jennifer the paper I promised." Anaphoric reference helps to speed conversation and make it less tiresome. Imagine speaking an entire day without using anaphors!

Anaphors minimize the problem of name specification. Once a person has been specified, a PersonSegment is initiated and its state variable set to that person's name. Any anaphors which follow are then resolved to that person until a new person is specified, at which time a new segment is introduced and the old one deleted.

A name can be specified *explicitly* or *implicitly*. Explicit specification occurs when the user introduces a person to the conversation, usually by giving a command containing someone's name: "Send Jeff a voice message." A new PersonSegment is introduced and its state variable set to 'Jeff'. Implicit specification occurs when a person is introduced into the conversation by the system, as when the user receives either email or voice mail. (The assumption is that the user will want to respond, and if not, another name will be specified anyway.) In order to introduce a new segment for the message sender, the address of the email or phone number from which the voice mail came must be matched against the rolodex to find the appropriate name.

Let's reexamine the interaction described for Phoneshell in Chatter's conversational interface. In addition to the

Chatter: Hello. Who's this?
User: It's Matt.
C: Hi Matt! You have some messages. Hear them?
U: Sure.
C: Message one is from Jeff about class tomorrow. Read it?
U: Yes. <computer reads message>
C: What do you want to do with this message?
U: Call him at home.
C: Calling Jeff at home. <no answer>
U: Send him voice mail.
C: Record your voice mail message for Jeff. <message is recorded>
C: Go ahead and deliver it?
U: Yes.

Figure 3: the name "Jeff" need not be explicitly specified in a conversational interface as it is inferred from the context of receiving a message and then later referenced by anaphora.

conveniences afforded by speaking rather than using TouchTones, the process of name specification is simplified by implicit inference and anaphoric reference.

As the dialog shows, the name "Jeff" is implicitly specified by Chatter when his email arrived. After that point, the anaphor "him" is twice resolved to "Jeff". Thus the problem of redundant explicit specification in Phoneshell is solved by supporting anaphora in Chatter's discourse model. The interaction scheme is neither hierarchical or modal. The user can receive an email message from someone, check their phone number, dial it, and if there is no answer then send a voice message -- all without "changing applications" or navigating menus.

WHY NAME RECOGNITION IS HARD

In the last section we focused on the utility of anaphora and implicit specification in minimizing explicit name specification. Yet explicit specification is a large part of the interaction, and explicit specification involves name recognition. Proper names present a particular problem for speech recognizers.

Speaker-dependent recognition: the training problem

The user has to train a speaker-dependent recognizer on each word by saying it several times so that the recognizer can build an acoustic model to use for pattern-matching. Given an utterance, it checks to see which model makes the best match and returns that word as having been recognized. Thus the recognizer works only for the person who trained it and only on the words which were trained.

A long list of words, such as the names contained in a rolodex, might require substantial training. Even if the user is willing to perform the initial round of training, the job is never completely done: as new people are added to the rolodex, their names must be trained as well. The user experience includes not just usage but set-up and maintenance as well, and if those overhead costs are high the user may abandon the system.

Speaker-independent: the transcription problem

Speaker-independent recognizers, by contrast, are not tuned for a specific voice. Instead of building models of words, they use precomputed models of phonemes, the smallest lexically significant elements of speech. (E.g., the word "chat" contains three phonemes: 'ch', 'a', and 't'.) The phoneme models are based on data from a large number of speakers, and the words to be recognized are entered as strings of phonemes in the lexicon.

Because speaker-independent recognition is difficult, the space of possibilities is constrained with a *grammar*, a syntactic representation of well-formed utterances. The grammar specifies which words to listen for and how they may be sequenced. For instance, the grammar defined below specifies a list of names that the recognizer should listen for. The count of possible options for the recognizer to match the utterance against at any time is called the measure of *perplexity*. The grammar below has perplexity of 10 since there are ten names.

Given an utterance, the recognizer charts a path through the space of possibilities based on the probability of one

```
Start(name).
name ---> Bennett
name ---> Davis
name ---> Dennison
name ---> Deveraux
name ---> Edwards
name ---> Franklin
name ---> Nelson
name ---> Smith
name ---> Smithson
name ---> Thomas
```

Figure 4: A grammar for name recognition. Each name in the grammar is represented as a series of phonemes in the lexicon (see figure 5).

phoneme following another. The "most likely" path, which will correspond to one of the utterances defined in the grammar, is then returned as having been recognized.

A speaker-independent recognizer cannot function without an accurate lexicon, for if the strings of phonemes for some word do not correspond to its actual pronunciation, the recognizer will most surely fail. Most proper names, however, are not included in the lexicon. It has been estimated [7] that there are over 1.5 million surnames in the United States, with approximately 1/3 of those unique.
It is possible to augment the dictionary with user-defined pronunciations, but this presupposes a working knowledge of phonetics as well as the notational conventions of the recognizer. Even if one has the requisite skills, the process is tedious. Furthermore, as in the case of training a speaker-dependent recognizer, whenever a name is added to the rolodex it must be added to the lexicon, as well.

The alternative is to automatically generate pronunciations for proper names not found in the dictionary, but this is difficult. Although language-specific rules exist for converting words into phonetic representations, names come from a variety of languages, each with its unique set of pronunciation rules. Identifying the linguistic roots of a particular name is nontrivial, though an algorithm is offered by Vitale [8]. Even if the nationality can be determined, some international names have been Anglicized and are thus pronounced differently than in their mother country. Hence an automatically generated phonemic representation of a name is likely incorrect. With the dictionary full of incorrect representations, the recognizer is doomed. For example, the first syllable of the name Sidner "side-ner" will likely be incorrectly transcribed as "sid" instead of "side", so when the recognizer attempts to match the phonemes in the dictionary for Sidner against the user's pronunciation, it will may confuse it with another name.

Continuous spelling

Given the inevitability of faulty name recognition, a natural fallback mechanism is to spell out the name. We define a grammar in which each name is represented as the series of letters of which it consists. In spelling out the name, the recognizer treats the series of letters like a string of words.

```
Start(name).
name ---> B E N N E T T
name ---> D A V I S
name ---> D E N N I S O N
name ---> D E V E R A U X
name ---> E D W A R D S
name ---> F R A N K L I N
name ---> N E L S O N
name ---> S M I T H
name ---> S M I T H S O N
name ---> T H O M A S
```

Figure 5: A grammar for spelling names. Each word is represented as a series of letters, each of which has a phonetic representation of its own. (e.g., the letter 's' consists of the phonemes /eh/ and /s/).

Continuous spelling works rather well, but it is not foolproof. Since the user speaks all the letters in a continuous string without any space between letters, the recognizer does not know how many letters to listen for. And since each letter is itself represented by a series of phonemes, the coarticulation of these phonemes may result in confusion as to how many letters were spoken. If, for instance, my rolodex contained by people with the last name "Marx" and "Marks", the recognizer may confuse the spelling. The series of phonemes for "M A R X" is /eh m ey ar eh k s/ and for "M A R K S" is /eh m ey ar k ey eh s/. Especially when spoken quickly, these phoneme strings are potentially confusable.

Another problem is the rate at which people spell names. If we could agree to spell at an identical, constant rate, spelling out a name might be a viable alternative. Yet people spell at different rates depending on a number of factors. If you ask me to spell my own last name, I'll very quickly say "M-A-R-X". Ask me to spell the name of the former Soviet leader, and it takes a little more time: "G-O-R.....B-A-*CHEV*". But a pause after the R can be fatal; as many recognizers automatically endpoint speech (as opposed to using a push-to-talk interface), the recognizer will assume that the user was finished spelling if the pause after the 'R' is long enough.. The lack of length constraints often means failure for continuous spelling.

Letter-by-letter spelling

Another alternative is spelling the name one letter at a time, processing one letter and then prompting for the next one. Letter-by-letter spelling avoids two shortcomings of continuous spelling: 1) since each letter is processed individually, the recognizer can use the count of how many letters have been spoken as an additional constraint in matching names, and 2) since each letter is prompted for and recorded separately, the rate at which people spell names is irrelevant.

Obviously it takes longer to spell a name letter-by-letter than to spell it continuously, but then again it takes longer to spell a name continuously than to speak it as a word. The tradeoff here is speed versus accuracy. If letter-by-letter spelling is more reliable than continuous spelling, it can provide a useful fallback mechanism when continuous spelling fails.

Reliable letter-by-letter spelling is nontrivial because recognition of individual letters is poor. Anyone who has spelled their name and address over the phone is familiar with techniques used to disambiguate similar-sounding letters: "Is that 'b' as is 'boy' or 'p' as in 'Paul'?" Although the perplexity is only 26, many of the letters in the alphabet are easily mistaken for each other. The members of the "e-set" are dominated by a high front vowel: /b/, /c/, /d/, /e/, /g/, /p/, /t/, /v/, /z/. Sets for other vowels exist as well, including the "a-set" (/a/, /h/, /j/, /k/), the "i-set" (/i/, /y/) and the "u-set" (/u/, /q/). Letters can also be confused for characteristics other than vowels; for instance, /s/, and /x/ are all dominated by a voiceless fricative: "ssss". One cannot rely on the recognizer to correctly identify all letters correctly.

There are at least two approaches to overcoming faulty letter recognition. One is to replace the letters with words beginning with those letters -- words that are sufficiently acoustically distinct so as not to be confused by a recognizer. This is the military-style "alpha" "bravo" "charlie" method of spelling. We reject this as too user-hostile.

Our approach is to figure out what letter the user *might have meant* by using considering the mistakes the recognizer is likely to make as well as what letters are relevant for the set of names in the application. To this end, we construct a confusability matrix for a recognizer and then work backwards to discern which letter the user must have meant in order for the string of letters to match a name in the list. This requires no extra knowledge on the part of the user; indeed, the process of disambiguation should be implicit and transparent. Details of the relevant algorithm are given in the next section.

SPECIFYING NAMES

Since the user will need to explicitly specify names at many points in the discourse, Chatter needs a reliable mechanism for specifying a name from a large number of choices. The following algorithm gives a three-step process for reliably specifying a name.

1. Name recognition. If the user says a name, as when trying to specify someone to look up in the rolodex, the recognizer will do its best to match the utterance against a name that it knows. Since the cost of dealing with the wrong person may be very high (as in sending a voice message to the wrong person) the system parrots back the name for explicit yes/no verification.

2. Continuous spelling. Failing name recognition, the user may try to spell out a name all at once, as in "P-A-T S-M-I-T-H". The system then pieces together the various letters into a name and repeats it for verification.

3. Letter-by-letter spelling. Failing continuous spelling, Chatter prompts for letters one at a time. When enough letters have been gathered to uniquely specify a name, Chatter returns the name.

The previous section discussed the unreliability of name recognition and continuous spelling, and thus the need for letter-by-letter spelling. Gathering letters one by one to uniquely specify a name is a process of carving impossibilities away from the list, similar to Phoneshell's implicit disambiguation of telephone keypresses. Chatter performs a similar process of disambiguation, but the disambiguation is based on how the individual recognizer misinterprets letters rather on the universal three-letters-to-one-number mapping of the telephone keypad. The following letter-by-letter spelling algorithm uses several techniques to guarantee that the correct name can be specified even when individual letter recognition is unreliable.

Subsetting letters

The previous section discussed the problems of letter recognition. The perplexity of the alphabet is 26, and so if we can reduce that perplexity our chances of success will increase. Now, given a list of names we need only listen for the relevant letters. We list the first letter of each name,

names	subset	recognized
Bennett **D**avis **D**ennison **D**everaux **E**dwards **E**llison **F**ranklin **N**elson **S**mith **S**mithson **T**homas	b, d, e, f, n, s, t	e
E**d**wards E**l**lison	d, l	d
Edwards		

Figure 6: Chatter only listens for the relevant letters, (marked in boldface), simplifying recognition.

remove redundancies, and then subset to that list of letters instead of the whole alphabet. The size of this list depends on the number of names and their distribution among the letters of the alphabet. The chances of recognizing the correct letter are improved since the recognizer has fewer letters to confuse. We repeat the process for the second

letter, third letter, and so on. The further along we get, the higher the likelihood that fewer choices will remain.

The lower perplexity which results from subsetting to only the relevant letters increases recognition performance. Thus the recognizer will never return a letter which could not show up at that letter position in the name.

Second-guessing the recognizer

Even if we subset to a small number of letters, the recognizer may still identify letters incorrectly. We construct a *confusability matrix* for the recognizer based on many trials of each letter. Any letter that is returned mistakenly for another is considered part of that letter's *confusability set*. Since we are working with a speaker-

a<-- *ah*	**n**<-- *anrs*
b<-- *abdepvz*	**o**<-- *lo*
c<-- *ctz*	**p**<-- *cdepvz*
d<-- *cdvz*	**q**<-- *qu*
e<-- *e*	**r**<-- *iry*
f<-- *fx*	**s**<-- *fs*
g<-- *gt*	**t**<-- *dept*
h<-- *h*	**u**<-- *qu*
i<-- *iy*	**v**<-- *bdepvz*
j<-- *adgjktz*	**w**<-- *fmnw*
k<-- *adjkq*	**x**<-- *sx*
l<-- *l*	**y**<-- *y*
m<-- *mn*	**z**<-- *defnstvxz*

Figure 7: Confusability matrix for Dagger, a speaker-independent, continuous-speech recognizer. A letter **returned by the recognizer** might have been mistaken for a letter *spoken by the user.*

independent recognizer, we want a speaker-independent confusability matrix; ideally the confusability matrix should be constructed using the same subjects whose utterances were used to build the phoneme models.

From the table, if the recognizer returns the letter /d/, chances are the user said one of the following: /c/, /d/, /v/, or /z/. It is clear that some letters like /l/ are (nearly) always correct: if the recognizer returns that letter then certainly that's the one the user said. The arrows point to the left in order to reflect the chronological ordering: that is, if the user says the letter /c/, /d/, /v/, or /z/, then the recognizer could return the letter /d/.

Now, for each letter returned by the recognizer, we consult that letter's confusability set. We then consider any letter in that set to be a valid match for that letter, and thus use all of those letters in the confusability set to whittle down the list of possible names.

Shown below is an example of how Chatter converges on a name given imperfect letter recognition by consulting the confusability matrix. The task is the same as for Phoneshell: finding the name "Edwards" among a list of ten names. The columns "SUB" and "RET" list the letters subset to and the letter returned by the recognizer, respectively. The column "CONFUSABLE" represents the confusability set for the letter returned by the recognizer.

names	sub	ret	confusable
Bennett **D**avis **D**ennison **D**everaux **E**dwards **E**llison **F**ranklin **N**elson **S**mith **S**mithson **T**homas	b, d, e, f, n, s, t	t	d, e, p, t
D**a**vis D**e**nnison D**e**veraux E**d**wards E**l**lison T**h**omas	a, e, d, l, h	d	c, d, v, z
Edwards			

Figure 8: Letters spoken by the user, though misrecognized, are implicitly resolved to match the desired name.

The algorithm converges on the name Edwards with only two letters, only one of which was recognized correctly. Indeed, the recognizer will converge on the correct name even if the wrong letter is returned every time (assuming the confusability matrix is accurate).

Adding the constraint of length

What happens, though, when the user has said all the letters, but more than one name matches that which was specified? There are two cases. In the first, Chatter converges on two or more names *of the same length* -- were one shorter it would have been eliminated previously -- which match the letters in the confusability matrix for each of the letters returned. The system stops asking for new letters, parrots back each of the possibilities, and asks the user to choose between them as in Phoneshell.

In the second case, the desired name is a prefix of another name in the last. So even though the user has said all of the letters in the name, the recognizer continues to ask for more letters since it thinks there are more matches to be had. Chatter allows the user to terminate the letter-specification procedure by saying "that's all". At this point, Chatter evaluates its list of potential matches. If there is a name with as exactly the number of letters that have been collected so far, then it returns that one as the correct choice. Thus the user is allowed to constrain the list by signaling that the maximum number of letters has already been entered -- a shortcoming of continuous spelling. Since Smithson is also in the list, the system finds them both

legitimate matches until the user ends with "that's all." Then it returns Smith as the only appropriate match given

names	sub	ret	confusable
Bennett **D**avis **D**ennison **D**everaux **E**dwards **E**llison **F**ranklin **N**elson **S**mith **S**mithson **T**homas	b, d, e, f, n, s, t	s	f, s
S**m**ith S**m**ithson	m	m	m, n
Sm**i**th Sm**i**thson	i	i	i, r
Smi**t**h Smi**t**hson	t	t	d, e, p, t
Smit**h** Smit**h**son	h	h	a, h
Smith Smithson	"that's all"		
Smith			

Figure 9: Multiple names that match are handled by saying "that's all", which gives the added constraint of length. Chatter infers that the shorter name was intended.

the constraint of length. This added constraint makes letter-by-letter recognition more reliable than continuous spelling.

EVALUATION

Both Phoneshell and Chatter use implicit disambiguation of user input in spelling names. With TouchTones, each letter is confusable with the two others which share its key, whereas with speech recognition, the confusability set for each letter can be different. In the confusability matrix computed for the TI recognizer, the average confusability set-per-letter is 3.3, suggesting that implicit disambiguation of spoken letters should be almost as robust as that of TouchTones.

Yet the question "are names best handled with TouchTones or speech recognition" is not answered by the simple metric of how quickly names can be spelled. One must consider the entire interaction involved in dealing with names.

In Phoneshell, specifying a name always involves spelling it out with TouchTones, so specification takes a constant amount of time. Using speech recognition, Chatter might get the right name the first time, which certainly takes less time than spelling it out with TouchTones. If it fails, however, then spelling it out will add up to more time than it would have taken to spell it with TouchTones. It is thus difficult to rank the methods by the amount of time taken. Name recognition is faster if it works but takes longer if it fails and the name has to be spelled out; hence, it is unclear which method is optimal for the initial specification of a name.

Re-specifying a name, as in calling the person after you've read email from them, is easier with Chatter since anaphora can be used. Saying "him" or "her" (which should be recognized robustly) is faster than spelling out a name each time with TouchTones. Actions which involve several steps for a single person are likely more efficient with Chatter.

The familiarity of speech is significant, as well. People have been spelling words verbally for most of their lives, whereas entering letters on a telephone keypad is an unfamiliar process. For users of rotary-dial telephones, who continue to constitute a significant percentage of the population (especially in Europe), TouchTones are not even an option. Admittedly, talking to a speech recognizer is not as natural as talking to a person, but the process does not require an entirely new protocol.

CONCLUSIONS

Both TouchTones and speech recognition can deliver interfaces which enable users to manipulate names in speech interfaces to communication systems. The impoverished 12-key input channel of touch-tone-based applications such as Phoneshell forces the user to spell names in order to specify them, and the necessarily hierarchical structure make it difficult to handle re-specification of names. The conversational interface of Chatter, made possible by speech recognition, breaks down barriers between applications, making direct specification potentially easier (just saying the name) and re-specification simpler still.

Unlike touch-tone recognition, which is robust, speech recognition is poor, especially of proper names which may not have professionally-tooled pronunciations to work with. Thus a system for spelling names with speech recognition, implicitly disambiguating the recognized letters to figure out which name the user was trying to spell, is necessary to preserve the integrity of the interface. Armed with this reliable fallback method, the Chatter interface can deliver on its promised benefits. And as speech synthesis becomes more realistic and recognition more reliable, intelligently structured discourse like Chatter may occasionally leave users wondering whether a computer or a human is on the other end of the line.

Computers and their addressing protocols are like stagehands which no one wishes to see in the spotlight. Phoneshell took a strong step towards this goal by offering an interface that shielded the user from the addressing

protocols of telephone and computer communication. Chatter improves further by hiding much of the computational infrastructure, allowing user to focus on people rather than on applications.

ACKNOWLEDGMENTS

The various elements of the Chatter infrastructure as well as most of the interactional details were designed and implemented by Eric Ly. Charles Hemphill provided almost daily assistance with Texas Instruments' Dagger recognizer. Special thanks to Raja Rajasekharan at Texas Instruments for making it possible for us to use Dagger. This work was sponsored by Sun Microsystems Laboratories.

REFERENCES

1. Bach, E. *Informal Lectures on Formal Semantics,* State University of New York Press, p. 97.

2. Chalfonte, B., Fish, R., and Kraut, R. "Expressive Richness: A comparison of Speech and Text as Media for Revision" *in proceedings of* ACM conference on Computer-Human Interaction, May 1991.

3. Davis, J. "Let Your Fingers Do the Spelling: Implicit disambiguation of words spelled with the telephone keypad" *in proceedings of* the American Voice Input/Output Society, 1990.

4. Grosz, B., and Sidner, C. "Attention, Intentions, and the Structure of Discourse." *Computational Linguistics,* 12(3):175-203, 1986.

5. Ly, E. "Chatter: A Conversational Learning Speech Interface" *in proceedings of* AAAI Spring Symposium on Intelligent Multi-Media Multi-Modal Systems, March 1994.

6. Schmandt, C. "Phoneshell: the Telephone as Computer Terminal" *in proceedings of* ACM Multimedia Conference, August 1993.

7. Spiegel, M. "Pronouncing Surnames Automatically", In *Proceedings of the 1985 Conference.* San Jose, CA: American Voice I/O Society, September 1985.

8. Vitale, T. "An Algorithm for High Accuracy Name Pronunciation by Parametric Speech Synthesizer", in *Journal of Computational Linguistics*, 17(1), pp. 257-276.