Chatter: A Conversational Learning Speech Interface

Eric Ly[1] *and Chris Schmandt*
Speech Research Group, MIT Media Laboratory
20 Ames Street, Cambridge MA 02139
ly@media.mit.edu, geek@media.mit.edu

ABSTRACT
The proliferation of communication devices has brought into being a host of media, including the use of voice over telephones, text in e-mail and images in faxes. As people become more mobile, they will need portable devices to access information in a timely way, whether they are in the office or on the road. The telephone is a convenient and ubiquitous device which people already know how to use, and it offers a familiar interface. But, traditional touchtone-based applications are hard to use.

Chatter is a speech-only application operated over the telephone for accessing information about members of a work group, including messages, locations of people, addresses and phone numbers. It accepts instructions from the user through a speech recognizer and generates output with a speech synthesizer. Chatter maintains a discourse model, making exchanges more natural and efficient. It develops a model of the user over time, which it then uses to suggest courses of action and to alert the user to potentially interesting information.

KEYWORDS
Interface agents, audio input and output, time management, interaction styles, discourse, language parsing and understanding, speech recognition and synthesis.

MOTIVATIONS

Today, people have an increasing need to get timely information in a variety of settings. The greater mobility of working people has created the need to stay in touch with others remotely, spurred by the advent of the telephone and, at the same time, encouraging the development of portable computers, pagers and fax machines. Besides voice, people now also communicate using text messages and images. Meanwhile, workstations have been evolved to handle different forms of media, with which they can now begin to deal effectively. Text, voice and images can now be stored cheaply on the desktop, where it has also become common for keeping personal information. As important as it is to access information on the desktop, however, there is an increased need to access the *same* information away from the desktop.

Though not immediately obvious, the telephone is an effective remote interface for accessing such information for several reasons. People are already accustomed to using it, and they are available everywhere. Furthermore, as computational and storage requirements increase, telephones as interfaces will not encounter the same scaling problems like portable computers, since they can be connected to arbitrarily powerful back-ends.

Roots of this project grew from an earlier project called Phoneshell. Phoneshell is a telephone interface which lets users access text and audio information stored on a workstation [8, 9]. It contains a suite of applications available through a system of menus, including voice mail among a local group of users, rolodex, calendar, dial-by-name, e-mail, and fax and pager management applications. With the exception of playing and recording audio messages, Phoneshell uses speech synthesis for output and the telephone's keypad for input. It has been in actual use by the group for several years and proves to be a vitals means of communication.

Yet, its user interface leaves more to be desired. While direct, the use of menus for options and the keypad for command selection is relatively impoverished. Unlike graphical interfaces, speech is a time-consuming, serial medium. Simply listening to a menu of commands takes time. Phoneshell also groups related commands into hierarchical menus, and the difficulties encountered with such a command structure are well-known. Given that the application is also completely command-driven and feature-rich, a user was often lost in a sea of choices.

The aim of Chatter is to present these choices in a more

1. The author is currently affiliated with Stanford University and can also be reached at ly@cs.stanford.edu.

Presented at the AAAI '94 Spring Symposium on Multi-Media Multi-Modal Systems, Stanford, CA, March 1994.

effective way. The telephone domain can accommodate a more "conversational" approach to interaction, since people already tend to communicate in this way with others.

Chatter is an attempt to solve these difficulties in Phoneshell's user interface. Since people are more accustomed to speaking than pressing buttons, Chatter uses a speech recognizer to accept commands instead of buttons. Natural conversation also employs powerful context to make communication more expressive and efficient, so context is a significant consideration in speech interfaces. Until now, most interfaces have also been one-sided; if the conversational experience is to be more natural, then interfaces must become more proactive. They should learn about their users and make suggestions to assist at appropriate times, helping them to focus on the important aspects of the program. Chatter uses a combination of discourse theory and machine learning to make such a conversational interface.

USER INTERFACE

The primary aims of the design of the Chatter user interface are two-fold: (1) the interaction should be as efficient as possible without incurring significant opportunity for miscommunication; (2) the interface should be proactive in making appropriate suggestions. Designing a good conversation requires making the computer's feedback brief and interruptible to conserve time and reduce the amount of information the user must remember [6, 13]. Studies of communication over telephones also indicate that speakers alternate turns frequently, and utterances are typically very short [7, 15]. If an interface can also make correct suggestions often, users only have to agree to them, rather than issuing complete commands. Because speech recognition will never be perfect, getting users to choose among fewer utterances will result in better recognition rates.

Functionality

Rather than having applications, Chatter contains the abilities to help the user complete a number of *tasks*:

- **E-mail and voice mail.** The application allows the user to play text or voice messages [11] and multimedia messages with text and voice attachments. Text messages are converted to speech using a speech synthesizer, and voice messages come from the voice mail system or from voice attachments in multimedia mail. Users can also send or reply using voice messages, which are formatted as voice mail or multimedia mail. Commands for saving messages to files for later viewing, deletion or forward are also present.
- **Phone dialing.** Chatter uses telephone numbers found in the user's rolodex to place calls for the user, a natural function for telephones. When the user makes a request to call, Chatter initiates a conference call, dials the number and patches the third party into the call.
- **Rolodex access.** The application also allows the user to access the information in his rolodex, which stores information about people's home, work and e-mail addresses, home, work and fax numbers, and additional remarks. While the rolodex is maintained at the workstation, its data can be obtained remotely.
- **Activity information.** To facilitate real-time communication, Chatter can also inform the caller of the whereabouts of others in the work group. Information about the location of users is collected from active badges users wear [12].

Sample Dialog

The following session demonstrates many of the ideas of the Chatter interface. It is annotated with noteworthy aspects of the interaction. Text not parsed by the recognizer are in italics:

Dialog	Annotation
C:Hi. Who's this? H:It's Eric. C:Hi Eric! You have two messages. Hear them? H:OK.	The user logs on. Chatter makes a suggestion about messages to read, and the user accepts the suggestion. What the user does first is remembered by the interface so it can be suggested later.
C:Message 1 is a voice message from Lisa. *Hi Eric, it's Lisa. I have the references you asked about. I'm at home today and may not get to Lab. If you need them in a hurry, call me. Bye!* End of message.	The interface plays Lisa's recorded voice mail message.
H:Where's Lisa? C:Lisa was last seen about ten minutes ago on Phoneshell. Call her?	The user interrupts the current topic of dialog to ask the location of Lisa. Since Chatter has location information about users, it can give information about when Lisa was last seen. Chatter infers from past situations that the user is probably interested in calling her, so it makes the suggestion.
H:OK. Call her at home. C:Calling her at home. *Conversation with Lisa takes place. Eric decides to go to Lisa's house to go pick up papers.*	The user affirms the suggestion and adds additional information to the command about calling her at home. Chatter uses the number stored in the user's rolodex for dialing.

C:Anything else about Lisa? H:Yes, what's her home address?	After the call finishes, it reminds the user they've been talking about Lisa and asks if he would like to know more about her.
C:Her home address: 572 Main Street #118, Cambridge, MA 02139. H:OK.	Chatter echoes the important information back in the next utterance because speech recognizers often commit mistakes. The address information is given by the user's rolodex.
C:Message 2 is from Don Jackson about "a phoneshell bug". Read it? H:OK. C:Eric, this message seems to be break phoneshell. Can you fix it? Thanks. Original message from SPOOL@northeastern.edu follows: ...	Chatter returns to reading messages for the user. Unlike the previous message, Chatter first asks if this message is to read because it is potentially long.
C:End of message. Send a reply? H:No, forward it to Jordan.	The interface has noticed that the user usually replies to Don's messages, so it makes a suggestion. The user rejects, and the new situation is recorded for future reference.
C:Record your voice message to forward to Jordan. H:*Jordan, it's Eric. Can you see if you can fix this bug? Don is having some trouble with it. Thanks.* C:Sending your message.	Chatter knows that Jordan can receive voice messages, so it decides that it would be easiest for the user to record a voice message.
H:Call. C:Call Don? H:Yes. C:Calling Don at work.	The user says *call*, an ambiguous command because no person is specified. Because of context provided by the message being read, the agent makes a guess as to who it may be.
No answer from Don. H:Cancel. Send a voice message. C:Send a voice message to Don? H:Yes. C:OK. Record your message. H:*Hi Don, it's Eric. I forwarded your bug to Jordan, so it should be fixed soon.*	Don is not at his phone, so the user exits from the call and decides to send a message.
C:Sending your message. No more messages to read. OK. What now? H:Good-bye. C:See you later.	All messages are read, so Chatter has run out of topics to talk about. It asks the user what he would like to do, and the user signs off.

The above dialog shows how the user can operate in a fairly natural dialog with the interface. Not unexpectedly, spoken dialog is shown to be a meaningful medium for human-computer interaction because it is more expressive than commands. It implies that speech interfaces have the potential to be even more expressive than even visual interfaces if they can use powerful context.

Architecture

Because Chatter is meant to operate on real-world data and events, its information is collected by several subsystems, which run asynchronously of the interface. They are described in greater detail in [4, 5].

Of particular importance is the collection of speech and audio data from the telephone. Input of audio is provided through a workstation telephone interface. A Sun Sparcstation handles all call control via an ISDN software and hardware interface. Telephone audio is captured on the workstation through an audio server process [1]. A software-only speaker-independent, HMM-based speech recognizer from Texas Instruments [14], operating at close to real-time, receives audio from the audio server and performs recognition on the input stream. A DECtalk device is used to convert text to speech for output to telephone audio.

MODELING DISCOURSE

Chatter maintains a discourse model of the human-computer interaction. It is derived from the Grosz/Sidner discourse theory, which was first proposed for task-oriented discourse in which participants are communicating to perform tasks [2]. According to the model, discourse can be analyzed into three interrelated components: a *linguistic structure*, an *intentional structure* and an *attentional state*. The linguistic structure is the external decomposition of the linear sequence of utterances into nested discourse segments, which serve as convenient elements for analysis. A particular discourse segment has an associated intention—the reason a speaker communicates it in the first place, say, a necessary step in a chain of instructions. These intentions are captured in the intentional structure. The attentional state is a dynamic internal representation of the objects, properties and relations salient at each point in the discourse. This information represents the objects introduced into the discourse that presumed to be known by the participants; it forms the basis for the use and resolution of anaphora.

The three components interrelate because a change in one usually results in a change of the other two. Since the structures are nested, the current state of the discourse can be represented as a data structures on a stack, known as the *focus space stack*. The discourse theory

also models interruptions, which are vital to interactive speech applications. Interruptions are discussed in more detail in a later section. The following sections discuss the major issues in designing a discourse model for Chatter.

Implementation

Implementing a real dialog system involves using domain-independent (the discourse theory) and domain-dependent information (the task model). Chatter implements a general framework for building conversational systems, on top of which dialog specific to the given task domain is built. One advantage of this approach is that the basic framework is general enough that it can be re-outfitted for other task domains.

The implementation organizes discourse around a set of data structures representing segments. Since the three components of discourse interrelate, it is convenient to construct data structures representing segments of discourse and associated focus spaces. The basic approach is to divide the interaction into a set of *dialog segments*, each of which is represented by a data structure that maintains state of the segment's dialog. Each dialog segment roughly corresponds to a major task of the application. For instance, a segment exists for reading messages, another one for sending and replying to messages, and so on. The segments also have computational capabilities for resolving pronouns, generating user feedback and subsetting vocabulary for the recognizer.

In this framework, the entity known as the focus space stack will be called the *dialog stack*. Elements on this stack are the previously-named dialog segments. Dialog segments are slightly different from the theoretic discourse segments because they combine both linguistic and intentional structures into the same entity. The state of the dialog stack represents both the linguistic nesting (for purposes of anaphoric resolution, say) of the dialog as well as its intentional state (goals of the current tasks).

Dialog segments are implemented as C++ classes, all of which inherit from a base **DialogSegment** class. The dialog system currently has eight dialog segment classes.

Segment	Function
BaseSegment	Simply asks the user what he would like to do.
CallSegment	Manages the task of calling a person.
ComposeSegment	Allows the user to compose an e-mail or voice mail message.
ExitSegment	Becomes active at the end to terminate the conversation with the user.
GreetSegment	Initiates a Chatter session with the user; helps establish the identity of the user.
HoldSegment	Handles the dialog for putting Chatter on hold.
PersonSegment	Responds to user's questions about a person's information, such as his addresses and phone numbers.
ReadSegment	Handles the reading of electronic and voice mail messages.

Speech Frame Dispatch Algorithm

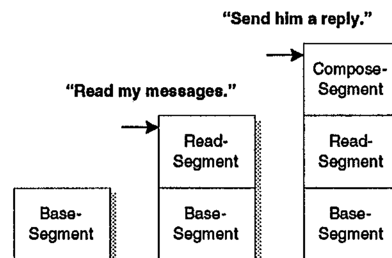
The dialog stack is initially empty. User utterances are analyzed into semantic speech frames (described in more detail in [5]), and segments are pushed on or off the stack as they are needed to process the user's utterances. When the user gives an utterance, it is converted into an event, which is sent to the appropriate segment based on the following algorithm.

1. Send the speech frame to the top segment on the stack and see whether it can process the frame. If so, the segment performs the appropriate execution and returns a value indicating that it has processed the frame and the algorithm is finished.

2. If the top segment cannot process the frame, then try the frame with the next lower segment on the stack. Continue to do so until the a segment has been found which can process the frame.

3. If no segment on the stack can process the frame, then see whether one of the inactive segment classes can process the frame. Each segment class implements a method which determines whether the segment can respond to a given speech frame. The first segment class which is found to respond to the frame is instantiated and pushed onto the top of the stack.

As a simple execution example, consider the following scenario where the user speaks *read my messages*, hears the first message and says *send him a reply.* Initially, the dialog stack contains only **BaseSegment**.



After receiving a speech frame for the first utterance, the application realizes that none of the active segments can process it. It finds that the **ReadSegment** class can process the frame, so it instantiates a new segment and pushes it onto the stack. At the second utterance, the system realizes that neither **ReadSegment** nor **BaseSegment** can respond to the second frame, so it finds that the **ComposeSegment** class can respond, instantiates one, and pushes it onto the stack to handle the frame.

Segment Termination

Determining when a segment should complete and be popped off the dialog stack depends on whether the purposes of the segment have been fulfilled. With task-oriented discourse, the answer depends on whether the task at hand has completed. However, there are several subtle issues which are a challenge to solve: some tasks have "logical" end point. For instance, it is safe to say that the **ReadSegment** does not terminate until after all messages have been read (or it has been told not to present more messages), so the interface can be said to "drive" the completion of this segment. Likewise, a **ComposeSegment** does not terminate until a message is delivered. For other segments, such end points may be less clear. In the case of a **PersonSegment**, the segment may complete as soon as it receives a speech frame which it cannot process, assuming the user has changed the topic.

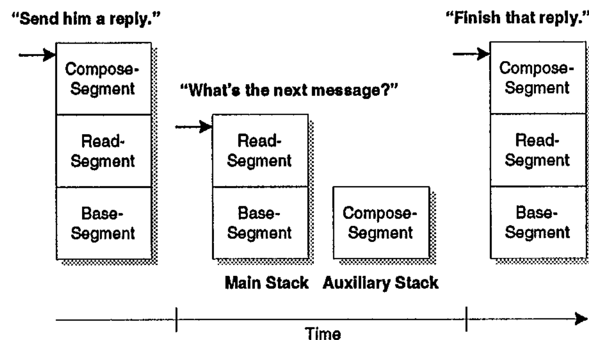
For those tasks having natural breakpoints, the interface may terminate a segment when such points arise. For the ambiguous case, deciding when to terminate is an issue of arbitration between the user and interface. Either the user or interface can terminate a segment, so the problem reduces to whether the user or interface is *driving* the dialog. Fortunately, some tasks are amenable to being more user-driven while others are more interface-driven. Consequently, the user is driving its completion by asking the questions. (When the arbitration is more arbitrary, some learning mechanism can be used to discover the user's habits and discover plausible stopping points. This is an area of future work.)

In the implementation of **DialogSegments**, a segment decides when it should self-terminate. The decision to terminate is usually a result of processing the latest speech frame or some timer expiring. A **DialogSegment** issues a *terminate* message to itself, and the dialog mechanism will remove it from the dialog stack at the end of the event cycle.

Even for segments with natural termination points, determining when to terminate is not as straightforward as noted. Often, leaving a segment active on the stack after the task has completed is necessary because the user's next request may force an otherwise terminable segment to remain active. For example, a **ComposeSegment** invoked for replying to a message may be considered terminable when its message is delivered. Yet, the user may want to continue the topic by saying something like *send the reply to Atty*, forcing the segment to remain active because *the reply* needs to be resolved. (In general, the user may actually want to continue or refer to focus that is long past in the discourse. In this case, a more complete history of the discourse needs to be memorized and algorithms for calculating references. This is also a topic for future work.)

Interruptions

When handling questions and tasks, segments may possibly be suspended so that a sub-task can be performed. Interruptions are disconnected flows of interaction. In this dialog framework, they are detected when a active segment responding to a speech frame is not at the top of the stack. In this case, all segments above it are considered to be interrupted. An auxiliary stack is created, and these segments are temporarily stored on the auxiliary stack. When the interrupted segments are reintroduced, they are returned to the main stack. Continuing with the example in the last section, suppose that while the user is sending a reply, he asks *what's the next message?* The **ComposeSegment** he was using for the reply is interrupted because the lower **ReadSegment** processes the command, so the **ComposeSegment** is placed on an auxiliary stack. When he reintroduces the topic by saying *finish that reply*, the **ComposeSegment** is replaced on the main stack and the auxiliary stack is deleted.



At the present time, it is not known whether more than one auxiliary stack is needed, since it seems reasonable that an interruption should be interruptible. The system currently allows multiple interruptions by maintaining a series of auxiliary stacks. However, in actual practice people may only rarely interrupt an interruption.

Reintroducing Interrupted Segments

The ability to interrupt segments creates the need for

mechanisms to reintroduce interrupted or unfinished segments. Reintroducing a topic can occur in two ways. First, the user may want to return to a segment that was previously interrupted. The segment to which he is interested in returning is below the top of the stack or in one of the auxiliary stacks. For instance, the user is in the process of replying to a message using a **ComposeSegment**. He then asks for the sender's phone number, invoking a **PersonSegment**. Then he says, *ok, send the reply to him.* The phrase *the reply* effectively ends the **PersonSegment** segment and reintroduces the previous **ComposeSegment** by referring to the definite reply.

In the second case, the application may want to reintroduce a segment, which occurs when a segment has completed and is popped off the discourse stack, leaving an old active segment on top of the stack. One example of this situation is one where the user has just finished sending a message with a **ComposeSegment**. He was reading mail previously with a **ReadSegment**, so the interface may ask him whether he would like to continue reading messages. This question serves as the reintroduction of the unfinished task of reading mail by the interface.

To accommodate segment reintroduction, the implementation is modified in two ways: first, the algorithm in 3.2 is changed so that any interrupted segment is allowed the chance to process a speech frame before the dialog system instantiates a new segment. Second, the vocabulary for reintroducing a segment must remain active in the recognizer. This vocabulary set may be more restrictive than the one used when the segment is on top of the main stack. Some segments have explicit reintroductions while others have implicit ones. For example, an interrupted **ComposeSegment** can be reintroduced by *finish that message* or *finish that reply.* For the **ReadSegment**, reintroducing an interrupted segment is implicit: the same command for reading another message while in the mail reading context can also be used to reintroduce the task, such as *what's the next message?*

Repairing Errors

A major problem in using speech recognizers is its accuracy; no recognizer will ever be perfect so interaction techniques must be developed to accommodate the presence of errors. Chatter provides several mechanisms for error repair due to recognition errors or changes in user intention. First, because it is impossible to tell when any utterance is actually correctly recognized, any information given is explicitly or implicitly restated or "echoed" to the user in the subsequent response. In the case that an utterance is incorrectly recognized, the user has the opportunity to correct the accepted information. At any point, the information given in the last step can be changed, whether it is as the user intended. The resulting interaction is still efficient because the information to be echoed is usually short or can be summarized.

Speech recognizers usually exhibit three types of recognition errors: insertion errors, rejection errors and substitution errors. Chatter handles each in different ways:

- **Insertion errors.** To prevent the interface from accepting commands which were not spoken, the interface currently has *stop listening* and *pay attention* commands, described above briefly. Upon hearing *stop listening*, the interface suspends itself until a *pay attention* is heard. These commands allow the user to "turn off" the interface to stop extraneous insertion errors. If an insertion error has already occurred, the user can issue a *cancel* command to cancel the current task.

- **Rejection errors.** Such errors cannot be handled satisfactorily at the moment because the speech recognizer does not report the fact that something was spoken but yet unrecognizable. It is expected the recognizer's programmatic interface will be extended so that such errors are reported to the application, allowing the it to query the user again.

- **Substitution errors.** Such errors are the most challenging of the three to correct; they occur when the recognizer mistakes what the user has spoken. Consider the dialog:

  H: Send e-mail to Barry.
  Computer heard send e-mail to Eric.
  C: Record your e-mail message for Eric.
  H: Send e-mail to *Barry.*
  ...

  Barry is misrecognized for *Eric*, so the user repeats the entire original utterance hoping to clarify his request. Assuming the recognizer correctly recognizes at this point, the problem is that the system does not know whether the user has interrupted the task with a new one or whether he is trying to make a repair (intonation information is not available).

  To avoid the problem of resolving the discrepancy, the following convention for repairing information has been adopted. Whenever a piece of information is misrecognized, the user can repair the incorrect information but preceding a correction by *no*. In the above example, the mistake can be corrected by *no, Barry.* This correction is then verified by the response *send it to Barry instead*?

More serious errors or misunderstandings can be cancelled by a *scratch that* command, which allows the system to completely forget all information that was newly introduced in the most recent utterance. This command is distinct from a *cancel* or *never mind* command, which has the effect of aborting the active segment.

Dialog Generation

The dialog system uses a relatively simple but powerful scheme to generate responses and queries based on template-matching. Each dialog segment class maintains a set of templates for generating responses. A template is a complete utterance with embedded unbounded named variables. These variables represent information to be echoed to the user or information to be queried. As a user's utterance is analyzed, some of these variables and their associated values are asserted into a database. When the utterance is completely analyzed and the time arrives to generate a response, the system searches for a template in which exactly the same variables have been asserted into the database, unifies the template, and speaks the response.

LEARNING ABOUT THE USER

As programs and the information they access become increasing complex, interfaces must also become more proactive in alerting the user to the capabilities of the program at appropriate times during interactions. Similarly, since user interactions are often consistently idiosyncratic, they are amenable to many levels of automation by the interface. Interfaces should develop models of their users through observation and make suggestions about appropriate actions to take.

In Chatter, a machine learning approach is used to predict information and actions which a user is likely to want to use. This section describes the machine learning approach taken to user modeling based on memory-based reasoning (MBR) algorithms described in depth in [3, 10, 5]. The approach has also been used in an e-mail agent and a calendar scheduling agent, described in [3].

MBR generally works by representing situations as vectors of features. These features are computed and stored in a database of situations over time. Part of the recorded situations are the actions that the user took in them, so these actions are the source for making suggestions to the user.

Chatter makes suggestions by looking in the MBR database for a set of situations which are similar to the current situation. These previous situations have associated action values recording the user's past actions. Based on these values, a confidence level between 0 and 1 is computed based on the similarity of the action values. If the confidence is above some threshold *T*, then the interface makes the most frequently occurring action as a suggestion to the user.

These suggestions must be fitted into the context of the discourse so that they arise at appropriate times; if the user is trying to perform a task and the interface continues to make irrelevant suggestions, such a customizable interface becomes an annoyance rather than a convenience. A later section addresses these issues in detail.

Unlike other rule induction schemes in which rules are inferred based on regularities in the data and then used, MBR works directly off the database. No explicit rules are pre-computed. This approach was chosen because it is not only simple but also particularly well-suited to handling idiosyncratic behavior users exhibit.

Suggesting Information

In this task domain, the information of interest are e-mail and voice mail messages. The interface can become proactive by bringing up interesting information to the user. Presumably, if the user has nothing to talk about, the interface can always strike a conversation with a new topic—in this instance some message to read. A list of messages sorted from "most interesting" to "least interesting" can be constructed by determining the following features of messages:

Type	Features
E-mail	Subject
	Is it a reply message?
	Sender
	Recipient
	Is the mail is directly address to the user?
	Mail domain of sender
	Quantized length (5 values)
	Interest flag
Voice mail	Sender string
	Does the call have an associated caller id?
	Quantized length (5 values)
	Interest flag

These features are computed automatically by the program whenever new messages arrive [5]. The choice of features descends from informal observations of rules that users in the group used for filtering their mail with a regular-expression based mail filtering program. The body of messages are not currently analyzed.

The *quantized length* feature is an integer value representing a description of the length of the message. The actual length is not used because it is relatively meaning-

less in the MBR algorithm. The length property may be an important feature in the speech domain because there is a considerable cost to listening to lengthy messages. The *interest flag* is not actually a feature of the message but of the situation during which it was read. The flag is true when the user has attempted to contact the sender by mail or phone through the interface. This event is considered important because presumably, the user is attempting to communicate with the person, and any messages from that person may have to do with communicating with him.

Once the features of messages are represented, the interest level the user gives them must also be represented as part of the situation. This table presents the four possible levels and how user actions are translated into one of the actions:

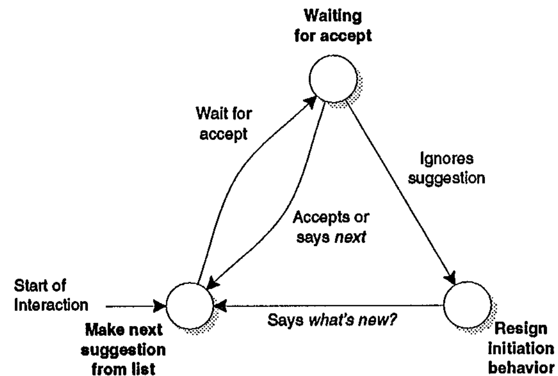
Interest level	How it gets this label
Very interested	User listens to message on first pass or interest flag is on.
Interested	User listens to all or part of message on a subsequent pass.
Not interested	User does not listen to message body at all.
Ignore	Upon hearing sender and subject, user deletes message without listening to message.

When Chatter presents messages, it does not read message bodies by default on the assumption that most messages will be either uninteresting or too lengthy. The interface only recites the sender and subject of each message. In the table, first pass means that the user chooses to listen to a message when it is presented to him the first time. Subsequent pass means he reads the message when re-scanning the message list.

Statistics are collected at the end of an interactive session with Chatter, and the sort for new messages will occur at two levels. The list is ordered using the four interest levels. When a new message arrives, its interest level is predicted using MBR, which places the message into an interest category. The confidence level of the message is then used to rank the message inside the category. This process is used to generate a sorted list of messages.

Introducing Message Suggestions

Once message suggestions are collected into a buffer, they must wait until there is an appropriate point in the dialog for changing the topic. Messages and other information types can arrive asynchronously during an interactive session but should be held until the current task is completed. More importantly, there is also the issue of whether the user or agent is controlling the initiative and how it can be arbitrated. Chatter takes the straightforward approach by implementing the following state machine:



The application maintains a list of interesting topics. When the user first begins a dialog, Chatter attempts to take the initiative by making suggestions about interesting messages. This process continues as long as the user is interested by responding positively to the suggestions. When the user ignores a suggestion, Chatter goes into a state where it does not suggest new information of interest until the user asks it to.

Suggesting Actions

Chatter also uses MBR to learn the "paths" that users take in their interactions with the application. This approach is inspired by other user interfaces which take the "guided" approach, in which at the end of every step or command, the interface suggests the next step. The intuition is that performing one task might suggest another task. For example, when someone reads a message, the next natural step may be to reply to it.

Chatter takes a different tack on the problem. In other user interfaces, the "next" step is often at the whim of the application's designer, whose guesses, though reasoned, may not correspond to what users' actually do. Users may often use an application in unexpected yet useful ways, and a learning interface can provide the mechanism for building an emergent task structure.

Initially, the user begins with an application with a somewhat unstructured task set. As the user uses the application more and more, the transitions from task to task are remembered. Connections between tasks are established, and the interaction becomes more guided, except that these links are customized to the individual. The interactive experience becomes more efficient because the agent can suggest an appropriate course of action to which the user needs only to answer *yes*, or

give another choice to proceed. Suggestions may be given by the segment currently at the top of the stack.

So far, these ideas have been used in Chatter most deeply in the interactions relating to reading messages. After a message is read, one or more of the following actions can be taken on the message:

Action	How it gets this label
Reply	User replies to message.
Forward	User forwards message to another user. (In a forward, the recipient is also recorded.)
Save	User saves message into mail folder.
Delete	User deletes mail after reading it.
None	User reads next or previous message.

As an action is taken on an individual message, it, along with features of the message, are recorded in the MBR database for use in predicting future actions. The actions above are not all mutually exclusive; if more than one action is taken on a message, then the system assumes that they are all likely candidate actions and asserts records for each action taken. When the agent has a high confidence on a suggestion, it asks whether the user would like to perform the task. If the agent's suggestion is rejected, the new choice is added to the memory.

Other segments have simpler preference modeling, as their functionality is more limited. The features used in representing situations for segments are given below:

Segment	Features	Predicted Actions
BaseSegment	Next segment	Most likely invoked segment?
CallSegment	Called party's name	Call work or home number?
Compose-Segment	Recipient's name	Use what delivery means?
	Action after mail is sent	Action after mail is sent?
ExitSegment	None	None
GreetSegment	None	None
HoldSegment	None	None
PersonSegment	Information requested (e.g. phone number, address)	Information requested next?
	Action(s) taken on person	Next action?
ReadSegment	E-mail and voice mail features given above	Interest in message?
	Action(s) taken on message	Action on message?

For example, the **BaseSegment**, which is the active top segment at the beginning of an interaction, tries to learn which task the user will most likely perform first after the user has logged in. It alleviates the user from having to tell the interface what to do first over time.

General Algorithm for Integrating Learning

An efficient interface means that the user does not have to say a lot to get the point across. While learning is an important element for speeding up interactions, it is only one component toward a more efficient interaction. Information needed to complete a task may be obtained from many sources, since the desktop is so rich with information. In performing tasks which require obtaining information, the interface should generally be implemented using the following interaction algorithm:

1. See if the information is already provided as part of the instructed command. For example, if the user says *send a message to Don*, then the recipient is specified as part of the utterance.
2. Whenever possible, look in databases for information. The Chatter domain provides some opportunities for retrieving information about users from pre-programmed databases. Some arguments for commands may be filled in by information from the database based on already provided values. For example, a database exists for users of voice mail subscribers. If the user says *send a message to Barry*, and Barry is voice mail subscriber, then suggest sending Barry voice mail.
3. Determine whether MBR can guess at some of the unknown features of the situation. A partially instantiated situation provides useful context, which can be used to predict some of the unknowns of a new situation. If so, then suggest the most-likely feature value. The user has a chance to accept or reject it.
4. Engage in a dialog with the user to query for the necessary information. If the interface cannot find the information anywhere, it can only ask the user to supply it.

An information-rich environment frees the user from having to specify commands fully every time.

CONCLUSIONS

The ubiquity of telephones today makes it possible to access many kinds of information on the graphical workstation with speech, allowing users to stay in touch with their information in a timely way. People have developed rich, extensive conventions for communicating in

speech, yet many computer speech systems today still analyze utterances without context, not taking advantage of the rich use of context found in conversation. Chatter models conversation about a given task domain, resulting in a natural and efficient interaction style.

The interaction is guided or agent-oriented rather than directly manipulated. The interface brings to the user's attention interesting messages and offers to automate actions. It also asks questions when it needs more information to complete a task.

To model the user, Chatter relies on a machine learning approach to collect relevant features about situations for each user and, in future situations, makes suggestions based on the assumption that the user will make similar choices.

ACKNOWLEDGMENTS

We would like to thank Charles Hemphill of Texas Instruments for adapting their recognizer to work with the server presented in this paper. Our appreciation to Raja Rajasekaran of Texas Instruments for making it possible for us to use the recognizer. Thanks to Candy Sidner and Pattie Maes for supervising this thesis project. Matthew Marx is assisting in the implementation of this project.

This work was sponsored by Sun Microsystems, Inc.

REFERENCES

1. B. Arons. Tools for Building Asynchronous Servers to Support Speech and Audio Applications. *Proceedings of the* ACM Symposium on User Interface Software and Technology, 1992.

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

3. R. Kozierok and P. Maes. A Learning Interface Agent for Scheduling Meetings. *In Proceedings of* the ACM-SIGCHI International Workshop on Intelligent User Interfaces, 1993.

4. E. Ly, C. Schmandt and B. Arons. Speech Recognition Architectures for Multimedia Environments. *In proceedings of* American Voice Input/Output Society, 1993.

5. E. Ly. Chatter: a conversational telephone agent. MIT Masters Thesis, Media Arts and Sciences Program, 1993.

6. A. Rudnicky and A. Hauptmann. Models for Evaluating Interaction Protocols in Speech Recognition. *In proceedings of* CHI '91, pp. 285–291. ACM, 1991.

7. D. Rutter. Communicating by Telephone. Pergamon Press, 1987.

8. C. Schmandt. Caltalk: A Multimedia Calendar. *In proceedings of* American Voice Input/Output Society, 1990.

9. C. Schmandt. Phoneshell: The Telephone as Computer Terminal. *In Proceedings of* ACM Multimedia '93 Conference, 1993.

10. C. Stanfill and D. Waltz. Toward Memory-Based Reasoning. *Communications of the ACM*, 29(12): 1213–1228, 1986.

11. L. Stifelman. Not Just Another Voice Mail System. *In proceedings of* American Voice Input/Output Society, pp. 21–26, 1991.

12. R. Want and A. Hopper. Active Badges and Personal Interactive Computing Objects. *IEEE Transactions on Consumer Electronics*, 38(1):10–20, 1992.

13. J. Waterworth. Interaction with Machines by Voice: A Telecommunications Persepctive. *Behaviour and Information Technology*, 3(2):163–177, 1984.

14. B. Wheatley, J. Tadlock and C. Hemphill. Automatic Efficiency Improvements for Telecommunications Application Grammars. Texas Instruments technical report.

15. C. Wilson and E. Williams. Watergate Words: A Naturalistic Study of Media and Communications. *Communications Research*, 4(2):169–178, 1977.