A Tool to Support Speech and Non-Speech Audio Feedback Generation in Audio Interfaces

Lisa J. Stifelman
Speech Research Group
MIT Media Laboratory
20 Ames Street, Cambridge, MA 02139
Tel: 1-617-253-8026
E-mail: lisa@media.mit.edu

ABSTRACT
Development of new auditory interfaces requires the integration of text-to-speech synthesis, digitized audio, and non-speech audio output. This paper describes a tool for specifying speech and non-speech audio feedback and its use in the development of a speech interface, Conversational VoiceNotes. Auditory feedback is specified as a context-free grammar, where the basic elements in the grammar can be either words or non-speech sounds. The feedback specification method described here provides the ability to vary the feedback based on the current state of the system, and is flexible enough to allow different feedback for different input modalities (e.g., speech, mouse, buttons). The declarative specification is easily modifiable, supporting an iterative design process.

KEYWORDS
Speech user interfaces, auditory feedback, text-to-speech synthesis, non-speech audio, hand-held computers, speech recognition.

INTRODUCTION

As computers continue to decrease in size, speech and sound will become a primary means of communication between the human and computer. It will be increasingly common for people to speak to their computers, VCRs, and wrist watches, and for them to speak back. While speech and sound are becoming more prevalent components for interaction, better tools are needed for designers and developers of audio interfaces. Speech recognition technology has moved from speaker-dependent isolated word recognition to speaker-independent continuous speech, increasing the complexity of both the spoken input and output. Most continuous speech recognition systems (e.g., Plaintalk [16], Hark [1], Dagger [13]) provide a mechanism for a developer to specify what a user can say to an application.[1] An easily modifiable, declarative mechanism is also needed for specifying speech and non-speech audio feedback (i.e., auditory icons [9] or earcons [4]).

Natural language generation research has tended to focus on producing coherent multisentential text [14], and detailed multisentential explanations and descriptions [22, 23], rather than the kind of terse interactive dialogue needed for today's speech systems. In addition, sophisticated language generation tools are not generally accessible by interface designers and developers.[2] The goal of the work described here was to simplify the feedback generation component of developing audio user interfaces and allow rapid iteration of designs.

This paper describes a tool for specifying speech and non-speech audio feedback and its use in the development of a speech interface, Conversational VoiceNotes. Auditory feedback is specified as a context-free grammar, where the basic elements in the grammar can be either words or non-speech audio sounds.

FROM VOICENOTES TO *CONVERSATIONAL* VOICENOTES

VoiceNotes [28, 29] explored a speech interface for a hand-held notetaking device. VoiceNotes allows a user to capture and randomly access spontaneous thoughts, ideas, or things-to-do in contexts where writing would be inconvenient (e.g., while driving or walking down the street). This research explored the idea of a speech-driven, hand-held computer with a microphone, speaker, and only a few buttons; no keyboard or screen. VoiceNotes demonstrated the utility of stored speech, overcoming the time consuming nature of listening to speech by providing the user with random access, dynamic speed control, and customizable feedback.

Conversational VoiceNotes expands the navigational model of the original system, providing the user with multiple ways of organizing and accessing notes in the speech database. Conversational VoiceNotes uses the PlainTalk

UIST 95 Pittsburgh PA USA
© 1995 ACM 0-89791-709-x/95/11..$3.50

[1] These systems utilize a grammar to constrain the perplexity of input, and enable efficient searching within the speech recognition engine.

[2] One barrier is that most interface and application development environments use 'C' while AI and natural language researchers work in LISP.

[16] speaker-independent continuous speech recognition and text-to-speech synthesis systems. Using continuous speech input, users can describe or add actions to their notes. Currently implemented descriptors are importance, day of the week, and a time to be reminded. Descriptors can be added to notes as a preamble to recording (e.g., "Take an important note") or postamble (e.g., "It's important"). These descriptions can then be used to navigate through the speech database (e.g., "Play all my important notes"), allowing random access to small subsets of notes. In addition to scaling the interface, allowing the user to manage a larger number of notes, the goal is to provide added flexibility to support different user preferences or mental models.

CRITERIA FOR DESIGN

In developing Conversational VoiceNotes, a tool was needed to support easier integration of speech and non-speech audio feedback. Conversational VoiceNotes uses digitized speech for the user's notes, text-to-speech synthesis for responding to the user's voice commands, and non-speech audio for navigational cues and describing notes. In early prototypes of the system, each of these types of output were embedded in various places in the code. This made it difficult to make modifications to the design, integrate the different forms of feedback, and ensure consistency throughout the system. It was therefore important to have a mechanism for describing speech and sound feedback as a whole, in a form that would be easily modifiable.

Another important consideration is the ability to dynamically vary the type and amount of feedback [7, 29]. Conversational VoiceNotes varies the audio feedback depending on factors such as the user's preferred output modality, detail level, and the input modality employed. Brennan [6, 8] emphasizes a collaborative view of interaction in which the computer must adapt its responses to its conversational partner and the current context of the dialogue. In addition, studies show that users prefer dynamic to static or repetitive feedback. Yankelovich [32] found that progressively modifying the feedback for speech recognition rejection errors[3] was preferred over a single repetitive message. A study of educational games for children found that programs with a number of different messages for the same situation were preferred to those using the same feedback over and over again [30].

Since audio output can be used independently of speech input, the generation method should not assume the existence of speech recognition. The feedback specification method described here can apply to any mode of input (e.g., mouse, keyboard, physical buttons, speech), and is flexible enough to allow different feedback for different modalities.

[3]Rejection errors occur when a speech recognizer is unable to report any of the words spoken by the user [26].

SPECIFYING AUDITORY FEEDBACK

Early in the process of developing Conversational VoiceNotes it became clear that a tool was need to support feedback generation, specifically for providing:

- synthetic speech feedback for responding to speech database queries
- digitized speech for playing a user's voice notes
- non-speech audio for navigational cues
- combinations of the above types of feedback
- multiple levels of feedback (i.e., differing amounts of detail)
- selection between different types of feedback (synthetic speech vs. non-speech audio).

The following sections present the basic design of the feedback generation tool. Note that the code examples have been simplified and modified for the paper.

Feedback Grammar

Auditory feedback is specified as a context-free grammar. The format of the grammar is based on one used by Plaintalk [2, 31] for continuous speech recognition input. The goal was to make the two formats as compatible as possible, so that they might eventually be used in conjunction with one another. The advantage of a hierarchical grammar is the ability to reuse components and avoid repetition. The specification is modular—for example, once the grammar rules for handling dates and times have been written, they can be used in a variety of applications.

Using Variables in Feedback Rules

One problem with many speech recognition grammars is that they do not provide a mechanism for sharing data (i.e., variables) with the application. The programmer is often forced to hard code information into the grammar. This is extremely limiting, since the grammar cannot be synchronized with the application data. For example, the following definitions hard code the names of people, duplicating information already contained inside a user's personal information manager or a group information database:

```
SENDMESSAGE:    "Recording a note for" PERSONNAME;
PERSONNAME:     "Chris" | "Barry" | "Eric";
```

The feedback grammar described here, allows the use of application variables inside nonterminal[4] definitions. The grammar can access data within the application which in turn can access any database. In the example below, %personName accesses the value of the personName variable in the application:

[4]A terminal is an atomic unit in the grammar that cannot be broken down any further (e.g., a word string). A nonterminal is a collection of terminals and may also contain other nonterminals (see Figure 1).

```
%define SendMessage
  "Recording a note for" %personName
%end
```

Note that in this grammar specification, %define is used to begin a nonterminal definition and %end to complete it. Angle braces (e.g., <nonterminal-name>) are used whenever a nonterminal is referenced [2].

Dynamic Generation Based on Application State

The feedback grammar allows the application developer to specify the conditions under which different types of feedback will be generated. If there is more than one rule inside a definition, then more information is needed to determine which rule to use to generate the feedback. A conditional statement can be specified for each rule (indicated using a semicolon after the rule) as shown in the following example:

```
%define Modifier        /* conditional statements */
  "high priority"       ; %priority==HIGH
  "low priority"        ; %priority==LOW
  ""
%end

%define VoiceObj
  "reminder"            ; %reminderVar==TRUE
  "note"
%end

%define RecordResponse
  "Recording a" <Modifier> <VoiceObj> "for" <WeekDay>
%end
```

The conditional statement:

```
%reminderVar==TRUE
```

is equivalent to the 'C' programming construct:

```
if (reminderVar == TRUE) {
  VoiceObj = "reminder";
}
else {
  VoiceObj = "note";
}
```

If no conditions are specified, the feedback generator will use the first rule in the list. In the example above, if reminderVar is true the result will be "Recording a reminder for ..." If reminderVar is false, then the second rule ("note") will match by default since there are no associated conditions and the result will be "Recording a ... note for ...".

The definition of Modifier shows an example of a *null* rule, which is useful for keeping the specification concise. In this definition, the last rule ("") will be executed if the priority is not set to high or low. Without the null rule, two <RecordResponse> rules would be needed—one with the <Modifier> component, and one without it.

The program also supports a logical AND operator, so multiple conditions can be placed on the right hand side of a rule. For example, the feedback grammar condition:

```
%reminderVar==TRUE %event==PLAY
```

is equivalent to the 'C' construct:

```
if (reminderVar == TRUE && event == PLAY)
```

Currently, AND is the only logical operator implemented, however, OR can be accomplished using multiple rules.

Sharing Constant Definitions

In addition to sharing variables between the application program and grammar, constant definitions can also be shared. The 'C' programming constructs #define and #include can be used inside the grammar, thereby avoiding duplicating definitions already specified in 'C' header files (i.e., .h files), or worse, hard-coding values into the grammar. Notice the condition %priority==HIGH in the definition for <Modifier>. The definition of HIGH is contained in a 'C' header file that is shared by the application and the grammar.

Integrating Sound into the Grammar

In addition to word strings, sound references can also be used in the feedback grammar. Note that "sound" refers to digitized speech or non-speech audio. A simple example is a prompt for recording a voice message:

```
%define RecordMessage
  "Record your message at the beep" SOUND:beep
%end
```

All sound terminals in the grammar are specified using SOUND: followed by a name or variable. In the above example, the sound named beep is used.

The feedback generation program passes an object list containing word strings and sound references back to the application (see Implementation section). In Conversational VoiceNotes, audio output is then handled by text-to-speech and sound libraries.[5] The feedback tool does not generate audio output itself since the application handles all audio output and is set up to allow interruption by the user. All sounds are currently stored in AIFF sound files, however, there is nothing to preclude the use of synthesized [10] rather than sampled sounds.

In the definitions below, the sound name of the user's voice note (or voice reminder) is contained in the variable %currentNote.

[5]The text-to-speech and sound libraries are built on top of the Macintosh Toolbox to provide a higher level, easier to use, programmatic interface to speech and sound.

```
%define ReminderPlayBack
  "Remember to" SOUND:%currentNote
%end

%define MoveResponse
  "Move" SOUND:%currentNote "to where?"
       ; %needCategory==TRUE
  SOUND:%currentNote "moved to" <Category>
%end
```

The first definition states that when a reminder is played, the system precedes it with the phrase "Remember to" before playing the user's voice note. The MoveResponse output depends on a single condition—whether the user has specified the new category. If the user requests "Move this note to my grocery list", the system would respond by playing the note being moved (e.g, *Buy apples*) followed by the phrase "moved to groceries".

FEEDBACK GENERATION IN CONVERSATIONAL VOICENOTES

The following sections discuss the use of the feedback generation tool by Conversational VoiceNotes to support a variety of interface design considerations.

Feedback Levels

Conversational VoiceNotes uses two types of output—speech and non-speech, two types of input—speech and buttons, and two levels of detail (terse or verbose). The audio feedback is varied depending on the variables listed below:

- user preferred output modality
- user preferred detail level
- input modality employed
- time elapsed since the last user command

Conversational VoiceNotes selects the type of feedback based on the conditional statements in the grammar. The RecordNoteResponse definition below states that sound output is used in response to recording a note when the user employs button input or has requested terse feedback.[6] Button input is associated with terse or sound output more often than speech input. In a user test of the original VoiceNotes interface [29], users did not expect speech feedback in response to pressing the record button on the device and often spoke over it.

```
%define RecordNoteResponse
  SOUND:beep          ; %input==BUTTON_INPUT
  SOUND:beep          ; %fbLevel==TERSE_FEEDBACK
  "Recording a note" SOUND:beep
%end
```

The StopResponse definition states that no explicit feedback is provided for stopping audio playback (except silence) if button input is used or terse feedback is preferred.

```
%define StopResponse
  ""                  ; %input==BUTTON_INPUT
  ""                  ; %fbLevel==TERSE_FEEDBACK
  "Stopped"
%end
```

A concern with the interface is that users will not remember their place in the speech database and put a note in an unintended place. Therefore, if the time since the user's last command has exceeded a threshold (i.e., timeElapsed == TRUE), then the system reminds the user of the current location. For example, when a new note is recorded, the system reports the name of the category where the note has been placed:

```
%define NewNoteResponse
  "Note added to" <Category> ; %timeElapsed==TRUE
  SOUND:thunk[7]             ; %output==SOUND_OP
  "Note added to" <Category> ; %fbLevel==VERBOSE_FB
  "New note added"
%end
```

Speech and Non-Speech Audio Feedback

In Conversational VoiceNotes, non-speech audio is used for navigational cues and to provide information about notes. For example, whenever an "important" note is played, it is preceded by an auditory icon that sounds like a trumpet. To reinforce this cue, if a note is marked as important (e.g., by saying "It's important") and sound output is selected, the trumpet sound is also played:

```
%define DescribePriority
  SOUND:trumpet SOUND:%currentNote
       ; %output==SOUND_OP
  "Note marked as important" SOUND:%currentNote
       ; %output==SPEECH_OP
%end
```

Rules can also combine sound output and text to be spoken. In the example below, if the user preference is set to SOUND_OP, when a category of notes is selected, an "opening" auditory icon (named opencat) is played followed by text-to-speech synthesis of the category name.

```
%define SelectResponse
  SOUND:opencat <Category>  ; %output==SOUND_OP
  "Moving into" <Category>  ; %output==SPEECH_OP
%end
```

Error Correction

Due to the error-prone nature of current speech recognition technology, some experimentation was also done with a mechanism for error correction and feedback. In particular, *substitution errors*[8] often occur when users enter dates and times as in the following example dialogue:

[6]The addition of an OR operator would remove the duplicate rule in the RecordNoteResponse definition.

[7]The thunk sound uses the analogy of the note being dropped into a container and hitting the bottom.

[8]Substitution errors occur when all or part of the utterance spoken by the user is recognized as a different utterance (e.g., the user says 3 o'clock and the recognizer reports 2 o'clock) [26].

User: "Play today's notes"
System: Notes for today ... "work on the CHI position paper", "meet with Jordan"
User: "Remind me about this"
System: At what time do you want to be reminded?
User: "At **3 o'clock**"
System: Note marked as reminder for today at **2 o'clock** ...
User: "No I said 3 o'clock"
System: Oops sorry, note marked as reminder for today at 3 o'clock "meet with Jordan"

```
%define CorrectionRule
    "Oops sorry, " <Response> ; %correction==TRUE
    <Response>
%end
```

Whenever the user speaks a day or time for a reminder, the system adds an editing expression[9] to the speech recognition grammar, allowing the user to say "No I said" followed by a new day and time. The system's response to such an editing expression is extremely important for maintaining mutual belief between the system and user. By responding "Oops sorry" Conversational VoiceNotes acknowledges the user's intention to correct an error, keeping the dialogue on track.

Notice in the above dialogue that the correction response repeats all of the information for the reminder. While an elliptical[10] response such as "Note marked for 3 o'clock" is briefer, inclusion of all reminder attributes implicitly informs the user what the system has assumed to be correct [12, 27, 32] allowing further repair if needed.

IMPLEMENTATION

Feedback State Frame

The feedback generator is designed to work in conjunction with a 'C' program. The program shares variables with the feedback grammar using a feedback state frame (see example below). Variables in the feedback state frame can be used inside nonterminal definitions or conditional statements in the grammar. For every variable to be used in the grammar, the programmer specifies the text name and a pointer to the application variable.

```
KVPair FbStateFrame[] = {
  {"event",          &Input.event},
  {"object",         &Input.obj},
  {"argument",       &Input.arg},
  {"currentPosition", &AppState.currentPos},
  {"currentNote",    &AppState.currentNote},
  {"inputType",      &FeedbackInfo.inType},
  {"outputType",     &FeedbackInfo.outType},
  {"feedbackLevel",  &FeedbackInfo.fbLevel},
};
```

[9] An editing expression (e.g., er, rather, no, I mean) is used to signal a correction to the listener [18].

[10] Ellipsis is the omission of one or more words that are expected to be implicitly understood based on the context.

In the example, variables from an "Input" structure that contain the user's voice command are included in the feedback state frame. In addition, there is information about the current state of the application—the current note being played and its position in the speech database. Lastly, there is information that is critical for use in the feedback generation—the input modality used, the output modality preferred by the user, and the level of detail.

Parsing the Grammar

The feedback grammar is lexically analyzed and parsed using variants of Lex and Yacc [15, 17, 21]. Using these tools, a separate tree is created for each nonterminal definition (%define) in the grammar. The nonterminal is the root node of the tree, with a subnode for every rule in the definition. The rule nodes have a subnode for each token contained in the rule (see Figure 1). Figure 2 shows a tree for the nonterminal definition GetReminderInfo. For each rule node, the program creates a list of conditions.

Tokens	Example
nonterminal references	<Date>
word strings	"on Friday"
word string variables	%personName
sound names	SOUND:whoosh
sound variables	SOUND:%currentNote

Figure 1: Tokens that can be used in rules.

```
%define GetReminderInfo
    "On what day" <Reminded> ; %day==TRUE
    "At what time" <Reminded> ; %time==TRUE
    "On what day and time" <Reminded>
%end
```

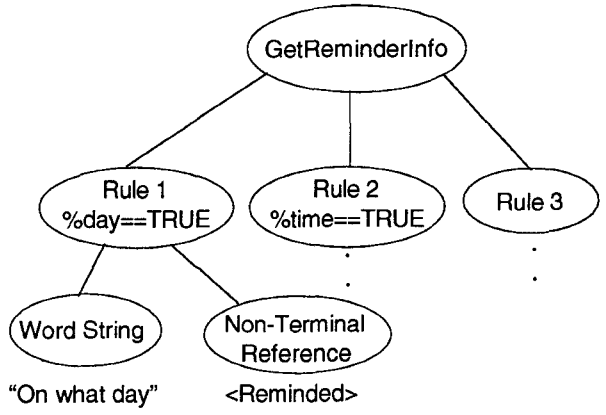

Figure 2: Tree for a nonterminal definition.

In a second stage of processing following parsing, all of the nonterminal references are resolved—a link is created between each reference and the tree for that nonterminal. In the example above, the <Reminded> reference would get linked to the Reminded tree. In the resulting structure, nonterminal definitions may have more than one parent

node, since they can be referenced in multiple places throughout the grammar. This tree structure (technically a directed acyclic graph, or DAG), identifies common subexpressions (i.e., nonterminals) to eliminate duplication [3].

Checking for Errors in the Grammar

During the second stage processing, as nonterminal references are resolved, two kinds of error checking are performed. First, an error occurs if a nonterminal reference has not been defined in the grammar. The program continues to attempt to resolve the remainder of the references, and outputs an error report listing all the unresolved references. Second, each nonterminal definition node contains a flag indicating whether or not it has been referenced anywhere in the feedback grammar. A warning is given in the error report for every unreferenced nonterminal. When developing the Conversational VoiceNotes grammar, this error report was an invaluable resource for debugging the grammar.

Generation Solving Function

The feedback generation solving function uses a depth-first search through the tree of nonterminals starting at the root node.[11] Each rule is checked in the order listed in the grammar until one succeeds. The following is a very simple example grammar to show the process of feedback generation, from grammar specification to final result:

```
%define Root
  <QuestionResponse>  ; %needInfo==TRUE
  <AnswerResponse>
%end

%define QuestionResponse
  "Call who?"              ; %person==NULL
  "Call" %person "where?"
%end

%define AnswerResponse
  "Calling" %person "at" <Place>
%end

%define Place
  "home"                   ; %location==PERSONAL_LOC
  "work"                   ; %location==BUSINESS_LOC
%end
```

Given this example grammar, the algorithm would first evaluate the <QuestionResponse> rule; if this rule failed, the <AnswerResponse> rule would be evaluated next. A rule succeeds if all of its conditional statements evaluate to TRUE. If no conditions are specified (as in the <AnswerResponse> rule), a rule is automatically accepted.

If a rule contains a nonterminal reference, then the algorithm checks the rules for this reference, and this continues until a leaf node is reached. The following example shows how the result "Call who?" could get generated:

```
Given:    needInfo = TRUE
          person = NULL

Result:   "Call who?"
```

The algorithm starts at the root node, first evaluating the <QuestionResponse> rule. Since %needInfo is TRUE, then the algorithm traverses to the definition of <QuestionResponse> and checks these rules. Starting at the first rule ("Call who?"), the algorithm evaluates the condition %person == NULL. Since this is true and the rule does not contain any more nonterminal references, a leaf node has been reached successfully. In order to generate the final result, each node in the tree maintains a list of tokens that gets propagated from the leaf up to the root node of the tree. In this example, the token "Call who?" gets copied from the QuestionResponse node to the Root node and the algorithm completes successfully.

In the sample generation illustrated above, each rule condition that was checked completed successfully. However, if one of the rules had failed (i.e., a rule condition evaluates to FALSE), the algorithm would pop up a level to test the next rule for that nonterminal. The next example shows how the result "Calling Barry at home" could get generated:

```
Given:    needInfo = FALSE
          person = "Barry"
          location = PERSONAL_LOC

Result:   "Calling Barry at home"
```

Again, the algorithm starts at the root node, evaluating the <QuestionResponse> rule. Since %needInfo is FALSE, the algorithm then goes on to the next rule, <AnswerResponse>. This rule has no conditions, so it is accepted automatically. Next, the algorithm traverses to the definition of <AnswerResponse> and checks the first rule, "Calling" %person "at" <Place>. This rule also has no conditions, so the algorithm traverses another level to the definition of <Place> and checks its first rule "home". The variable location is set to PERSONAL_LOC, and there are no more nonterminal references to check, so a leaf node has been reached successfully.

As in the previous example, the final result is propagated from the leaf to the root node of the tree. First, the token "home" is copied from the Place node to the AnswerResponse node. It is added onto the end of the list of tokens located at this node—"Calling", "Barry", and "at". Note that the %person token is resolved to "Barry" using the key-value pairs specified in the feedback state frame. Lastly, a list of 4 tokens ("Calling", "Barry", "at", and "home") is copied from the AnswerResponse node to the

[11] The root node of the grammar is specified in the first %define in the feedback specification.

Root node of the tree. The grammar generator returns this list of tokens to the application.

RELATED WORK

Many conversational speech interfaces are being developed using the latest continuous speech recognition and text-to-speech synthesis technologies (e.g., [19, 32, 33]). While ATIS (Air Travel Information Service) research [25] has focused on effective speech recognition performance in specialized domains, better tools are needed for designers to develop new user interfaces employing speech technology in a variety of domains. In addition, as Zue states "current research in spoken language systems has focused on the input side" alone, rather than on both speech input and generation [34].

Researchers at Sun Microsystems are currently working on SpeechActs, a general environment for developing speech-based applications [20]. This architecture consists of a natural language component called SWIFTUS, a grammar compiler for speech recognition, and a discourse manager. The goal is to support the use of a variety of speech recognition systems. These tools have been used in the development of several applications for accessing information (e.g., email, calendar) over the telephone using spoken input [32]. The SpeechActs architecture is an important step toward easier development of speech-based interfaces. Without a development tool like SWIFTUS, a speech application developer is forced to write a separate grammar for the speech recognizer and natural language components of the system. In addition, these specifications would not be portable for use with another speech recognition system. SpeechActs has focused on speech input specification, but would be a good environment for integration with an audio output tool like the one described in this paper.

Brennan and Hulteen have developed a general model for adaptive feedback [7]. As a testbed for their model they developed a conversational telephone agent that adapts its feedback depending on the state of the interaction between the user and computer. The feedback model has seven possible states for responding to the user. For example, if the agent is in the "attending" state but has not heard the words spoken by the user, it might respond "What was that again?" A number of factors are taken into account such as the frequency of user interruptions to correct the system in the recent dialogue history, the frequency of misrecognitions (e.g., rejection errors) made by the system, and the level of noise in the user's environment. This work represents an important step toward the development of conversational speech interfaces that are able to repair errors in communication and adapt to the current context of the discourse. However, tools like the feedback grammar described in this paper are needed to help designers implement this kind of feedback model.

ISSUES FOR FUTURE WORK

Throughout this paper, examples have been given showing the generation capabilities of the auditory feedback tool. However, a number of limitations were also experienced. Given the declarative specification of the grammar, the feedback can be changed quickly, without recompiling the application. However, a context-free grammar was found to be too limited in its generative power. The grammar becomes long and unwieldy as additional rules are needed to handle cases such as singular versus plural forms. The sections below describe other limitations encountered when implementing the feedback for Conversational VoiceNotes and how these issues might be addressed in future work.

Auditory Feature Specification

In the original VoiceNotes interface, the speed of the audio feedback is varied. For example, when deleting a note, the system responds "Deleting ..." and then plays back the contents of the note being deleted at a rate 1.5 times the user's speed setting. The current feedback grammar does not provide support for specifying playback speed. A general mechanism for specifying features (e.g., speed, duration) of tokens in the grammar is needed for cases like this one. This could also be useful for allowing parameterization of auditory icons (e.g., not just specifying a "thunk" sound but how big and what type of object) [10].

The grammar also does not provide a syntax for specifying that sounds be mixed or spatialized. The problem becomes one of auditory streaming [5]—sounds must be broken into auditory streams according to their semantic groupings. In Conversational VoiceNotes, notes and descriptive audio icons should form one auditory stream, and navigational sounds another. There are many potential cues that can be employed to cause such streaming (e.g., spatial location, pitch differences, time synchronization). Additional support is needed for this in the grammar. For example, a designer needs to be able to specify that two sounds should begin or end at the same time or be normalized in length. Ultimately, there needs to be a way to group segments of auditory output, creating multiple auditory streams.

Adaptive Feedback

In addition to coding information such as playback speed in the grammar, there needs to be a mechanism to account for dynamic changes based on discourse context. The global and local focus of the discourse [11] will impact the use of pronouns, ellipsis, and prosodic cues such as accent. One difficulty is although something may be pronominalized in human-human conversation, this may not be appropriate in the same instances for conversation with an error-prone speech recognizer (see example below).

The decision of when to pronominalize will depend not only on discourse structure but on factors such as the user's experience level, the number of speech recognition errors that have occurred in the present interaction, and even the noise level of the listening environment. Brennan and

Hulteen's conversational telephone agent adapts its feedback based on factors like these [7].

Another important consideration is how critical is the action that is about to be performed. For example, deleting a voice note has a destructive consequence whereas playing it back does not. Brennan and Hulteen use "positive evidence" [8] to notify users of an intended action before it is completed. The following are two example dialogues from their system:

User:	"Call Lisa."
System:	"Ok, I'll call Lisa."
User:	"Call Lisa."
System:	"Ok, I'll call Lewis."
User:	"No! I said call Lisa."

In the first case, given only discourse focus information, a pronoun could have been used—"OK, I'll call her." However, the system explicitly repeats the command as part of the feedback in order to give positive evidence of the name recognized by the system. Given this evidence, the user has the opportunity to interrupt if an error has occurred. The error-prone nature of speech recognition as well as the added cognitive load of listening to synthetic speech [24] must be taken into account when designing auditory feedback.

Integration with Speech Input

Due to the device dependent nature of current continuous speech recognition grammars, the feedback grammar tool has not yet been integrated with a particular speech input format. The input grammar for Conversational VoiceNotes is specified using the format for the Plaintalk speech recognition system and is not currently integrated with the feedback grammar. In order for this tool to be most valuable to designers using both speech input and output, it is critical to allow the input and output grammars to share a lexicon, nonterminals, and phonetic pronunciation rules.

CONCLUSION

This paper describes a tool to support generation of speech and non-speech auditory feedback and provides examples of its use in an application. In particular, Conversational VoiceNotes uses the grammar to specify conditions under which different types and amount of feedback should be generated. Once a feedback grammar has been created, it is simple to modify, supporting an iterative design process.

The paper outlines a number of important issues for future work in this area. Researchers exploring uses of speech and non-speech audio have primarily studied the two in isolation. Given the growing need for auditory feedback in the user interface, it is necessary to focus on how speech and sound can be used together effectively. The work presented in this paper is an important step in this direction.

ACKNOWLEDGMENTS

Thanks to Barry Arons for helping with the design and coding of the feedback compiler and providing valuable comments on drafts of this paper. Jordan Slott helped with debugging. Eric Hulteen, Barry Arons, and Chris Schmandt gave valuable input in the design of Conversational VoiceNotes. Kai-Fu Lee and Eric Hulteen provided Plaintalk, and Bob Strong, Matt Pallakoff, and Ted Kopulos provided technical support.

This work was sponsored by Apple® Computer, Inc. and Sun Microsystems.

REFERENCES

1. HARK Prototyper User's Guide. BBN Systems and Technologies: A Division of Bolt Beranek and Newman Inc., 1993.
2. Speech Rules. Chapter 8 in *Macintosh Quadra 840AV and Macintosh Centris 660AV Computers*. Apple Computer. Inc. DeveloperPress, 1993.
3. A.V. Aho, R. Sethi and J.D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1988.
4. M.M. Blattner, D.A. Sumikawa and R.M. Greenberg. Earcons and Icons: Their Structure and Common Design Principles. *Human-Computer Interaction*, 4(1):11-44, 1989.
5. A.S. Bregman. Auditory Scene Analysis: The Perceptual Organization of Sound. Cambridge, The MIT Press, 1990.
6. S.E. Brennan. Conversation with and through computers. *User modeling and user-adapted interaction*, 1(1):67–86, 1991.
7. S.E. Brennan and E.A. Hulteen. Interaction and Feedback in a Spoken-Language System: A Theoretical Framework. *Knowledge-Based Systems*, (March)1995.
8. H.H. Clark and S.E. Brennan. Grounding in communication. In J. Levine, L.B. Resnick and S.D. Teasley, editor, *Perspectives on socially shared cognition*, pages 127-149. APA, 1991.
9. W.W. Gaver. The SonicFinder: An interface that uses auditory icons. *Human-Computer Interaction*, 4(1):67-94, 1989.
10. W.W. Gaver. Synthesizing Auditory Icons. *In Proceedings of INTERCHI '93*, pages 228-235. ACM, 1993.
11. B. Grosz and C. Sidner. Attention, Intentions, and the Structure of Discourse. *Computational Linguistics*, 12(3):175–204, 1986.
12. P.J. Hayes and D.R. Reddy. Steps toward graceful interaction in spoken and written man-machine

communication. *International Journal of Man-Machine Studies*, 19:231-284, 1983.

13. C.T. Hemphill. Dagger User's Guide and Reference Manual. Texas Instruments Incorporated, 1993.
14. E.H. Hovy. Planning Coherent Multisentential Text. *In Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, pages 163–169, 1988.
15. S.C. Johnson. YACC: Yet Another Compiler Compiler. University of California, Berkeley, CSRG, 1986.
16. K. Lee. Towards Conversational Computers: An Apple Perspective. *In Proceedings of EuroSpeech.* Berlin, Germany, 1993.
17. M.E. Lesk and Schmidt. Lex—A Lexical Analyzer Generator. University of California, Berkeley, CSRG, 1986.
18. W.J.M. Levelt. Speaking: From Intention to Articulation. The MIT Press, 1989.
19. E. Ly and C. Schmandt. Chatter: A Conversational Learning Speech Interface. *In Proceedings of AAAI Spring Symposium on Intelligent Multi-Media Multi-Modal Systems*, 1994.
20. P. Martin and A. Kehler. SpeechActs: A Testbed for Continuous Speech Applications. *In Proceedings of AAAI '94 Workshop on the Integration of Natural Language and Speech Processing, 12th National Conference on AI*, 1994.
21. T. Mason and D. Brown. LEX & YACC. O'Reilly & Associates, Inc., 1991.
22. K.R. McKeown. Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text. Cambridge University Press, 1985.
23. J.D. Moore and C.L. Paris. Planning Text for Advisory Dialogues. *In Proceedings of the 27th Annual Meeting of the Association of Computational Linguistics*, pages 203–211. ACL, 1989.
24. D.B. Pisoni, H.C. Nusbaum and B.G. Greene. Perception of Synthetic Speech Generated By Rule. *Proceedings of the IEEE*, 73(11):1665-1676, 1985.
25. P. Price. Evaluation of Spoken Language Systems: The ATIS Domain. *In Proceedings of DARPA Speech and Natural Language Workshop*, pages 91–95, 1990.
26. C. Schmandt. Conversational Computing Systems. New York, Van Nostrand Reinhold, 1993.
27. C. Schmandt and B. Arons. A Robust Parser and Dialog Generator for a Conversational Office System. *In Proceedings of AVIOS*, 1986.
28. L.J. Stifelman. VoiceNotes: An Application for a Voice-Controlled Hand-Held Computer. Master's Thesis. Massachusetts Institute of Technology, 1992.
29. L.J. Stifelman, B. Arons, C. Schmandt and E.A. Hulteen. VoiceNotes: A Speech Interface for a Hand-Held Voice Notetaker. *In Proceedings of INTERCHI '93*, pages 179-186. ACM SIGCHI, 1993.
30. E. Strommen. "Be Quiet, You Monster!": Speech as an Element of Software for Preschool. Presented at the Annual Meeting of the American Educational Research Association. Children's Television Workshop, 1991.
31. B. Strong. Casper: Speech Interface for the Macintosh. *In Proceedings of EuroSpeech.* Berlin, Germany, 1993.
32. N. Yankelovich, G. Levow and M. Marx. Designing SpeechActs: Issues in Speech User Interfaces. *In Proceedings of CHI '95.* ACM SIGCHI, 1995.
33. V. Zue, J. Glass, D. Goodine, H. Leung, M. Phillips, J. Polifroni and S. Seneff. The Voyager Speech Understanding System: Preliminary Development and Evaluation. *In Proceedings of IEEE 1990 International Conference on Acoustics, Speech, and Signal Processing*, 1990.
34. V.W. Zue. Human Computer Interactions Using Language Based Technology. *Presented at the 1994 International Symposium on Speech, Image Processing, and Neural Networks*, 1994.