ETUDE, A RECURSIVE DIALOG MANAGER WITH EMBEDDED USER INTERFACE PATTERNS

Roberto Pieraccini, Sasha Caskey, Krishna Dayanidhi, Bob Carpenter, Michael Phillips

SpeechWorks International, 17 State Street, New York, NY 10004, USA

{roberto, scaskey, krishna.dayanidhi, bob.carpenter, phillips}@speechworks.com

ABSTRACT

In this paper we describe ETUDE, a dialog manager that supports recursive descriptions of the dialog flow in spoken dialog applications. We also introduce the notion of user interface patterns, i.e. those dialog patterns that are frequently used in applications. We then describe how these patterns can be built into the dialog manager engine in order to facilitate the design and development of complex applications.

1. INTRODUCTION

Most of the enterprise telephony spoken dialog systems deployed today are based on the directed dialog paradigm [6], in which the flow of the conversation is highly structured with carefully designed prompts to solicit a response from the user that falls within the defined grammar of that dialog turn. In general a directed dialog can be represented by a finite state controller whose states correspond to the system actions (e.g. prompting, recognizing, accessing external databases, etc.). Traditionally, developers of enterprise dialog systems developed the logic of directed dialog call flow using tools provided by the telephony platform (e.g. Intervoice Brite, http://www.brite.com) and reminiscent of the IVR development systems, possibly with the support of native languages such as C++ and VB.

As the complexity of the systems evolves in both the number of dialog states of the controller and in the degree of mixed initiative, the cost of design, development and maintenance increases. One source of complexity in a directed dialog system is the introduction of general UI patterns that support mixed initiative. Examples of patterns that may appear at any step of the dialog are commands such as *repeat, back-up, startover,* as well as the commands for navigating between different branches of the application. In order to increase the degree of mixed-initiative and allow efficient interaction with the system, especially by expert users, users may provide extra information beyond what was requested in the prompt. Similarly allowing for digressions at some steps of the dialog, either for clarification or to complete subtasks, would enhance the overall usability of the system.

With the competing objectives of reducing design and development costs and also and allowing more flexible interactions, it seems necessary to completely or partially automate the design and the implementation of the dialog strategy. Furthermore, the designer must be afforded the freedom to specify the user interface with a fine degree of control. Such automation can be achieved by introduction of a dialog manager with built-in behavior patterns that can be understood, tuned and deployed by dialog system designers and developers. The challenge is to find the right compromise between built-in behavior of the dialog manager and the flexibility required by the designers.

Among several sophisticated dialog manager schema [1] [2], finite state automata and recursive transition networks [3] [4] have been successfully used in dialog system as ways of both describing and controlling the dialog flow. We describe here ETUDE, an implementation of a recursive transition network controller for dialog system that addresses the issues described above. ETUDE can be summarized as follows. A dialog flow is specified as a directed graph whose nodes represent actions (e.g. prompts, recognition, database access, etc.) that the dialog system invokes to interact with the external environment (e.g. the caller, the backend, etc.) and whose transitions are associated to conditions on the session variables. One of the distinguishing characteristics of ETUDE is that it permits recursion in the sense that a single node may be expanded as a whole dialog itself. In the rest of the paper, we describe how ETUDE implements UI patterns such as backing up, entering a subdialog and jumping out of the current dialog and taking up another one. ETUDE's dialog execution strategy directly supports state persistence, which is especially useful for stateless architectures such as VoiceXML.

In the rest of this paper we will describe the dialog flow abstraction and the implementation of the ETUDE dialog manager.

2. THE DIALOG FLOW ABSTRACTION

The state of an individual dialog session is represented by a *frame*, which maps *keys* consisting of strings to values, which can be strings, numbers, Booleans, sequences of values, or frames. A *dialog* is a pair $\mathrm{D} = \langle \mathbf{N}, N_S \rangle$, where $\mathbf{N} = \{N_1, N_2, \ldots, N_M\}$ is a set of nodes, and $N_S \in \mathbf{N}$ is the start or *initial node*. A *node* is a pair $N = \langle \mathbf{T}, A \rangle$ where $\mathbf{T} = \langle T_1, T_2, \ldots T_Q \rangle$ is a sequence of transitions, and A is an *action*. A *transition* is a pair $T = \langle N_E, C \rangle$, where $N_E \in \mathbf{N}$ is the *destination* node of the transition and C is a condition. A *condition* is a function mapping frames to Boolean values. An *action* is an arbitrary function mapping a frame to a frame. The execution of a dialog on a frame is defined according to the following pseudo-code:

```
Frame execute(Dialog d, Frame f) {
   for (Node n = d.initialNode; n != null; ) {
      f = n.action(f );
```

```
    Transitions ts = n.transitions;
    n = null;
    for (k = 0;k < ts.length && n != null; ++k)
       if(ts[k].condition(f))
             n=ts[k].destination;
  return f;
}
```

where *d*.initialNode is the initial node of dialog *d*; *n*.action is the action associated with node *n*; *n*.transitions is the sequence of transitions associated with node *n*; *ts*.length is the length of *ts*; *ts*[*k*] is the *k*+1st transition of *ts*; *ts*[*k*].condition is the condition associated with the *k*+1st transition of *ts*; and *ts*[*k*].destination is the destination node associated with the *k*+1st transition of *ts*.

Note that the evaluation function of a dialog has the same form as an action. In general, ETUDE supports recursion by allowing the action of a given node in a dialog to be given by another dialog. This helps structure complex dialogs into sub-dialogs.

3. GOTO AND GOSUB SHORTCUTS

In a directed dialog application the dialog manager strictly controls the course of the conversation and there is minimal built-in support for caller initiative. Directed dialog is an effective conversational strategy for new users, who appreciate the guidance provided by the system and it allows them to quickly form a mental map of the service. However a strictly directed dialog strategy can get in the way of expert and repeat callers who are seeking for a more efficient interaction. Certain applications require a higher degree of initiative. The concept of shortcuts tries to address both strategies by allowing designers to overlay a set of shortcuts over the directed dialog graph. We identified two kinds of shortcuts, namely GOTO and GOSUB shortcuts.

GOTO shortcuts permit transitions from an origin node within one dialog to a destination node that is outside the dialog. In practice, a GOTO shortcut acts as a transition, the only difference being in that the destination node may be outside the current dialog. Once a GOTO shortcut is executed, the dialog proceeds from the destination node without returning to the original node. GOTO shortcuts have to be defined and implemented taking into consideration the recursive nature of the dialog execution. To pick out a node uniquely, a path must be specified to the node through the sub-dialog hierarchy. For example, with:

$$D_0.\text{nodes} = \{\ldots, N_i, \ldots, N_j, \ldots\}$$
$$N_i = \langle T_i, D_1 \rangle$$
$$N_j = \langle T_j, D_2 \rangle$$
$$D_1.\text{nodes} = \{\ldots, N_k, \ldots\}$$
$$D_2.\text{nodes} = \{\ldots, N_l, \ldots\}$$

If we want to establish a GOTO shortcut from node N_l of dialog D_2 (when a certain condition C_l verifies) to node N_k of dialog D_1 when dialog D_1 is invoked as the action of N_i (dialog D_1 could be invoked as an action of other nodes as well). We also assume that dialog D_0 is the main dialog and that the action associated to the origin node N_l is a terminal (i.e. it is not a dialog) collection action (i.e. it is an action directly connected to speech recognition collection events). When the system is executing node N_l, the execution stack can be represented as $N_j.N_l$, meaning that the execution environment is currently executing the function associated to node N_j which in turn is invoking the function associated to node N_l. The destination node of the desired GOTO shortcut can be identified by the execution stack $N_i.N_k$. The algorithm for the implementation of a GOTO shortcut has to perform the following two operations: a) Pop nodes out of the execution stack until the outer execution layer is reached b) Push nodes into the execution stack until the defined destination node stack is reached.

An example of the use of GOTO shortcuts is global navigation commands. For instance, consider the following transaction (this example follows the dialogs shown in Fig. 1)

S: Would you like to get an account balance or make a transfer?
U: Make a transfer.
S: From which account would you like to transfer, checking or savings?
U: Savings.
S: How much would you like to transfer from savings?
U: Uh. Go to account balance.
S: Account balance. For which account would you like a balance, checking or savings?

GOSUB shortcuts are used to implement local digressions in the dialog, but differ from sub-dialogs in that they return for re-execution of the invoking node. An example of a GOSUB shortcut can be exemplified by the following dialog.

S: Would you like to get an account balance or make a transfer?
U: Make a transfer.
S: From which account would you like to transfer, checking or savings?
U: Savings, please.
S: How much would you like to transfer from savings?
U: Hmm. How much money do I have in my savings?
S: Account Balance. The balance of your savings account is 2,356 dollars and 37 cents.
S: How much would you like to transfer from savings?

In this case, in contrast to the GOTO shortcut example, the system executes the account balance sub-dialog and then returns to the calling node, re-executing the interrupted collection action. In order to give more flexibility to the designer for a fine-tuning of the prompts (e.g. in the example, the re-prompting for the transfer amount differs from the original prompt), the node execution function can detect whether a return from a GOSUB shortcut is in effect.

4. USER INTERFACE PATTERNS

There are certain recurrent interface patterns that appear, or are likely to appear, in many different dialog systems. Some of them can be considered universal patterns. Examples of those are back-up, start-over, repeat, main-menu. Their meaning is obvious in most dialog contexts, and they start to assume the quality of universal navigation commands. There is a strong analogy between these *UI universals* and the universal commands we expect to find in any properly designed desktop application, such as the File and Edit menus, the Undo command, Help, etc. Often, when some of the commands do not make sense in some part of the application they are still there in a disabled form (e.g. grayed out). Similarly, as spoken dialog applications become more and more pervasive and ubiquitous, and more and more users become accustomed to them, it will become natural to expect certain commands to be *always* available, such as back-up (which is analogous to the undo command in desktop applications), help, etc.[1]

There is another class of UI patterns that recur in many applications, but only in certain situations or in certain parts of some applications. The universal quality of these UI patterns is not in their presence at any point of the dialog, but in their use. For instance, let's consider list navigation. Depending on the kind of functionality of the list (e.g. selection, editing, etc.), the navigation follows certain predetermined patterns (e.g. Say *next, previous or that one*). In the desktop analogy, these patterns can be associated, for instance, with the procedures for opening and saving files, which are the same from application to application.

However, all these patterns may differ from application to application and from implementation to implementation. A universal consistency across applications and across implementations is desirable for several reasons. One of the main reasons is that consistency of UI patterns can help users learn how to use spoken dialog systems independently of the application[5], thus increasing the overall transaction completion rates, the caller's population acceptance of the spoken dialog technology, and the overall user satisfaction.

One way to guarantee and encourage consistency of the UI patterns across applications and implementations consists in embedding the underlying logic in the dialog manager engine. Considering also that the implementation of some of these patterns may be quite complex, a dialog manager engine that includes the most common UI patterns can be highly beneficial also to the reduction of the design/development cost of complex applications.

5. UI UNIVERSALS IMPLEMENTATION ISSUES

UI universals are defined as properties of collection dialog nodes (i.e. a dialog nodes that are associated with a collection action). If a dialog node allows a certain UI universal command, then the associated command word (e.g. *back-up)* and its syno-

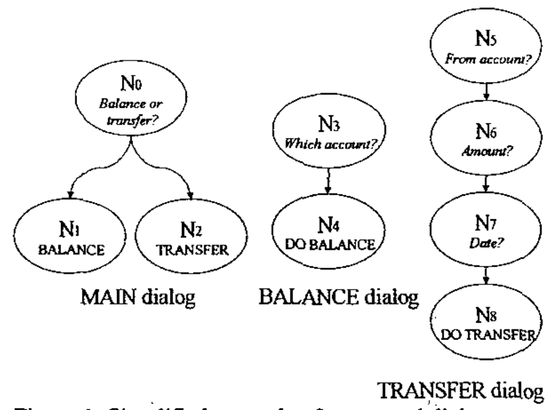

Figure 1: Simplified example of structured dialog.

nyms must be included in the grammar used by the collection action of that node. The ETUDE dialog manager does that automatically during the initialization phase. In order to keep the consistency of the interface, different levels of activation of universal commands can be specified for each collection node. In our implementation we have three levels of activation: *enabled*, corresponding to full functionality of the command, *acknowledged*, the command is recognized, but a prompt is played warning that the command is not active – analogous to the graying-out of features in desktop applications, and *disabled*, the command is not recognized.

Let's analyze in more details two UI universals: back-up and repeat.

The back-up command implements the undo feature for voice-based systems as in the following example:

S: This is the banking application. Do you want account balance or to make a transfer?
U: Make a transfer.
S: Which account do you want to transfer from? Checking or savings?
U: Savings.
S: What amount do you want to transfer from your savings?
U: Five hundred dollars.
S: Do you want to transfer five hundred dollars from savings to checking?
U: back-up
S: What amount do you want to transfer from your savings account?
U: back-up
S: Which account do you want to transfer from? Checking or savings?

In order to define the correct operation for back-up, it is necessary to define not only which nodes would accept the back-up commands, but also which node to back-up to. Of course the node to back-up to cannot be determined statically for any given collection node, since it depends on how the dialog evolved up to that point. Moreover, once a back-up is performed, the frame,

[1] W we did observe instances of users saying *Main Menu* in applications where a main menu was not even defined or announced.

i.e. the set of all session variables, must be reverted to the previous configuration (undo function). The back-up command is based on the concept of *back-up anchor*. A node that is defined as a back-up anchor is a node to return to when, successively in the dialog, the caller issues a backup command.

A back-up stack is kept in the session frame. Once a back-up anchor node is visited during the course of a dialog session, a new stack element is created including a copy of the current frame and a reference to the backup anchor node. The element is then pushed into the stack.

Once the user issues a back-up command during a dialog session, then the element at the top of the back-up stack is retrieved with a pop operation. The dialog manager then performs a transition to the back-up anchor node and restores the frame.

A back-up node may not be in the same dialog as the node visited when the back-up command was issued. Hence the transition to the back-up anchor has to be performed through a GOTO shortcut.

A mechanism similar to back-up can be implemented for the commands start-over and main-menu. In that case there is no need to keep a stack, but only one anchor and the related frame are kept for each session.

Similarly to the back-up command, the repeat command needs a repeat anchor to be defined. Once a node which is defined as a repeat anchor is visited, a pointer to that node is kept at a particular location in the current frame. When the user invokes the repeat command while a node activated for repeat is being visited, all the output nodes (i.e. the nodes associated with an output or prompting action) between the repeat anchor and the current node are executed by the dialog manager.

Examples of other common UI universals which logic can be included in the execution algorithm of a dialog are those corresponding to commands such as *operator, change-language,*and global navigation commands. Global navigation consists in having the initial nodes of branches of sub dialogs announce themselves with a special prompt (e.g. <earcon> *Account Balance*) and allowing the user to issue commands such as "*Go to account balance*") at any point in the dialog.

6. SUMMARY

We presented in this paper the concept of a dialog manager that supports a recursive definition of the dialog flow. The dialog flow abstraction is presented in detail. In addition we described the concept of UI patterns, i.e. those patterns that typically appear in most dialog systems. Some of these patterns can be defined as universals, meaning that their presence is expected at any point in the dialog and would improve the usability of the systems. We described how the logic of some of the UI universals can be embedded into the dialog manager engine, both helping encourage the introduction of the UI patterns across different application and reducing the cost of developing complex applications.

The authors wish to thank Stephen Springer (SpeechWorks International) for his help with the definition of the UI patterns and Anibal Jodorcovsky (currently with Intelerad Medical Systems) for his help with the initial implementation of ETUDE.

7. REFERENCES

[1] P. Constantinides, S. Hansma,, C. Tchou,, A.Rudnicky, "A schema-based approach to dialog control," in Proceedings of ICSLP. 1998, Paper 637.

[2] D. Stallard, "Talk'n'Travel: A Conversational System for Air Travel Planning," in Proceedings of the Association for Computational Linguistics 6th Applied Natural Language Processing Conference (ANLP 2000), Seattle, Washington, April 29 – May 4, 2000, pp. 68-75

[3] R. Pieraccini, E. Levin, W. Eckert, "AMICA: the AT&T Mixed Initiative Conversational Architecture," in Proceedings of EUROSPEECH 97, Rhodes, Greece, Sept. 1997.

[4] S. Seneff, J. Polifroni, "Dialogue Management in the Mercury Flight Reservation System," presented at Satellite Dialogue Workshop, ANLP-NAACL, Seattle, April 2000

[5] S.Shriver, A. Toth, X. Zhu, A. Rudnicky, R. Rosenfeld. "A Unified Design for Human-Machine Voice Interaction," in Proceedings of *CHI 2001*.

[6] E. Barnard, A. Halberstadt, C. Kotelly, M. Phillips, "A Consistent Approach To Designing Spoken-dialog Systems," in Proceedings of the Automatic Speech Recognition and Understanding Workshop, Keystone, Colorado, December 1999.