DIALOGUE MANAGEMENT IN THE TALK'N'TRAVEL SYSTEM

David Stallard

BBN Technologies, Verizon

ABSTRACT

A central problem for mixed-initiative dialogue management is coping with user utterances that fall outside of the expected sequence of the dialogue. Independent initiative by the user may require a complete revision of the future course of the dialogue, even when the system is engaged in activities of its own, such as querying a database, etc. This paper presents an event-driven, goal-based dialogue manager component we have developed to cope with these challenges. The dialog manager is explicitly architected for asynchronous input and flexible control, and uses a tree-ordered rule language we have developed that also provides for close coupling with discourse processing. The dialogue manager is implemented as part of Talk'n'Travel, a simulated air travel reservation dialogue system we have developed under the US DARPA Communicator dialogue research program, whose purpose and scope we also briefly summarize.

1. INTRODUCTION

We discuss the dialogue management component of the Talk'n'Travel system. Talk'n'Travel is a research prototype system sponsored under the Communicator program [2] of the US Defense Advanced Research Projects Agency (DARPA). A general description is in [1]; some other systems in the program are [3], [7], and [9]. The common task of this program is a mixed-initiative dialogue over the telephone, in which the user plans a multi-city trip by air, including flight, hotel, and rental car reservations, all in conversational spoken English.

A block diagram of Talk'n'Travel is shown in Figure 1. The system consists of the Byblos statistical speech recognizer, the GEM robust language understander, the dialog management component, a language generation module, and a COTS text-to-speech system. A remote flight reservations web site provides live flight schedule data, with a central hub component providing communication between the various modules.

A key goal of this work is finding a compact and general way to represent dialogue control schemas, and especially to handle user utterances that dramatically change the dialogue state and disrupt the planned dialogue flow. If the user changes his mind, for example, or corrects the system's misunderstanding, the task may be completely restructured, rendering complex ongoing sub-dialogues irrelevant, and/or resurrecting previously completed ones. Suppose, for example, that the system and the user have previously agreed on a flight from Boston to Denver for September 20th, and the following dialogue then ensues:

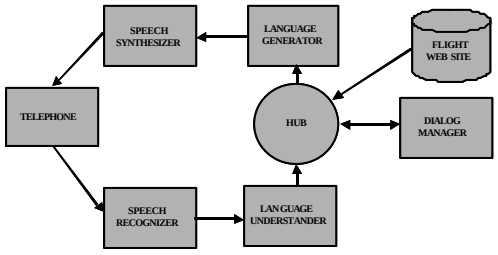
S: Do you want a hotel in Denver?
U: Yes, I do.
S: Please stand by while I look up the hotels on the web.
S: [pause for 5 sec]
S: OK, got them.. How about Marriot for $130 a night?
U: That's too expensive.
S: Cheaper than $130. How about Holiday Inn for $80?
U: Forget it, I don't want a hotel.
S: You don't want a hotel. How about a rental car?
U: No, I think I want to leave on the 22nd instead.

In this dialogue, the entire hotel reservation sub-dialogue has to be aborted, including negotiations, database queries, etc. What is more, the previous flight reservation sub-dialogue has to be partially resurrected as well, because the user has changed the date, requiring a new query to the web site, plus re-negotiation of the flight booking.

An additional problem arises from the potential asynchrony of user utterances. Suppose the user chose to say "Forget it. I don't want a hotel" during the pause in the dialogue while the the system was querying the web site for hotel availability. Even though the system was not engaging him in dialogue at that point, the user would naturally expect the system to understand him, abort the query and hotel booking sub-task, and return to the rest of the task.

Clearly, it is prohibitive to explicitly code for these various possibilities at every point in the dialogue control scheme. The challenge we explore in this paper is how to accommodate these disruptive events in a highly general and flexible way, without complicating the dialogue control scheme unmanageably.

In the remainder of the paper, we discuss our strategies for solving these problems, which include an event-driven dialogue



manager and a domain-independent tree-ordered rule dialogue control language, and present tentative evaluation results.

2. EVENT-DRIVEN INTERACTION

At any given time, a dialogue system is in one of three high-level states:

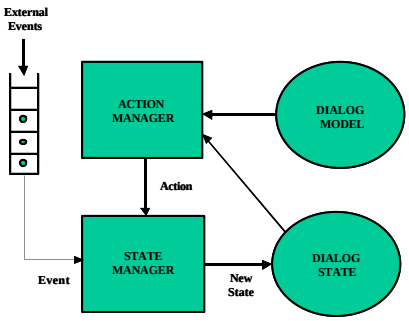
1. Speaking to the user
2. Waiting for the user to speak
3. Doing something else (e.g., querying a database)

Usually, barge-in is thought of as the ability of the user to interrupt the system while it is speaking, so that he can be heard by the system in state 1 as well as state 2. Many dialogue systems do not have a state 3 (or are in it for only a neglible period of time), because they move from prompt to prompt and do not have high-latency computations such as remote queries into their dialogue. But if a system does have a state 3, it is surely not reasonable for it to be deaf to the user during that period. Rather, the user should be able to barge in during those intervals as well.

Most dialogue frameworks provide a synchronous "prompt" function, like the PROMPT tag of VoiceXML [10], that plays an audio prompt to the user, and returns when either the user replies or a timeout period expires. If the user barges in over the prompt, this function will simply return early. This strategy will cope with states 1 and 2, but not with state 3, since the prompt function will not be executing during that period.

In order to overcome this limitation, the Talk'n'Travel dialogue manager adops a completely different approach that is event-driven and asynchronous by design. Figure 2 shows a block diagram of the dialogue manager, which consists of an event queue, an action manager, and a state manager, together with their associated knowledge bases. The event queue holds events received from the speech recognizer, speech synthesizer, and database query modules, which run in their own separate threads. The events received may be meaning representations of the user utterance, results from an external web query, and so forth.

The action manager determines the next action to execute, based on the most recent event, the dialog control model, and the current dialogue state. The system executes this action, and then checks the event queue for any new event. If an event is found, the state manager uses it to update the dialogue state accordingly. For example, if the event corresponds to a user utterance, the state manager will use the meaning representation of the utterance to update the constraints in the dialogue state. It does not matter for this purpose whether the event was received in response to the system's prompt, or whether it was spontaneous.



System actions such as prompts or database queries work by delegating the body of their computation to separate threads, and therefore complete very quickly. The event queue is checked each time an action is executed, so it is checked very frequently, and events are handled very soon after they arrive on the queue.

3. REPRESENTING DIALOGUE STATE

We view the dialogue as a cooperative attempt between user and system to find values for a set of descriptive entities $E_1,\ldots,E_n$, subject to a set of constraints on those entities $C_1,\ldots,C_m$. In the air travel problem, the E_i are air, hotel, or rental car bookings, and the constraints C_j are restrictions that the user imposes, such as "from Boston", "after nine AM", etc. For these purposes, the dialogue state at any given time is simply the complete set of such constraints that the user has expressed up to that time.

The constraints are represented by expressions called "path constraints", and are written in the form:

$$P\ R\ t_1,\ldots,t_n$$

where P is a path representing a functional composition of one or more attributes, R is a relation of n+1 argument places, and the $t_1,\ldots,t_n$ are zero or more argument terms. The interpretation of the path constraint is that the relation R holds between the value of the composite function P and the arguments $t_1,\ldots,t_n$. The following is an example.:

LEGS.2.FLIGHT.DEPART GT 9:00AM

This represents the constraint that the departure time of the flight of the second leg of the trip be after 9:00 AM. that the attributes making up the path are separated by periods. The dialogue state could contain this path constraint if the user had at some point during the dialogue said "I want to return sometime after nine in the morning".

A pair of distinguished relations, SOME and NONE, is used to represent, respectively, whether the user wants or does not want whatever object the path describes. These relations are unary, meaning that they do not take arguments on their right. The following is an example, indicating that the user does not want a hotel in his destination city:

LEGS.1.HOTEL NONE

The language understanding component of Talk'n'Travel outputs sets of path constraints as its meaning representation. For example, for "I want to fly to Denver on Monday" it would produce:

FLIGHT.DEST EQ DENVER
FLIGHT.DATE.DOW EQ MONDAY

The state manager combines these constraints with the existing dialogue state to produce a new dialogue state. A number of steps are involved. One is resolving ambiguity, such as figuring out what leg of the trip that user is talking about, and which "Monday" he is referring to. Another is merging the new constraints with the existing ones, and determing which if any of the existing constraints they should replace. The end result at each dialogue turn is (hopefully) a consistent and coherent picture of what the user wants from the system.

4. THE DIALOGUE CONTROL MODEL

The dialogue control model has the form of a tree, expressing a goal/sub-goal structure. The leaves of the tree correspond to the system's actions, such as prompting the user, querying the database, etc., and the interior nodes control and order the execution of these actions. Each action has an underlying goal it seeks to achieve, such as finding out what city the user wants to fly to, or what day he wants to leave on. Actions also have relevancy and executability conditions. Each of these conditions is represented by individual predicates attached to the action.

An example action is the following, which prompts the user for the departure date of a flight on the second leg of his trip:

```
(ELICIT LEGS.2.FLIGHT  DATE
        isRelevant: (isWanted LEGS.2.FLIGHT)
        isAchieved (filled LEGS.2.FLIGHT.DATE)
        isExecutable: true
        utterance: "What day are you leaving?")
```

This action will be *relevant* if the path LEGS.2.FLIGHT is wanted (that is, if the user is not on a one-way trip), and *achieved* if the system knows what date the user wants this flight to be on. The action is by default always executable. Execution of this action will result in the user being prompted with the argument utterance ("What day are you leaving?").

The interior nodes of the tree include control structure nodes, such as WHILE and WHENEVER, and an iteration node, FORALL. Other interior nodes represent plan invocations designed to achieve a complex sub-goal. The complete dialogue model tree is traversed from left to right, depth-first, subject to the control structure nodes, until it finds an action which is relevant, whose underlying goal is not yet achieved, and which is executable.

Once an action has been selected for execution, it may be executed repeatedly until either it achieves its goal, or the dialogue state changes so that the action becomes no longer relevant or no longer executable.

The following represents a (highly simplified) plan for finding out if the user wants a hotel, and booking him one if he does:

```
(defplan getHotel ($hotel)
  (askIfWanted $hotel utterance: "Do you want a hotel?")
  (while (isWanted $hotel)
    (startFetch $hotel)
    (whenever (isFetching $hotel)
      (speak "Please stand by, while I look up the hotels.")
      (waitForRetrieval $hotel
                        utterance: "Still fetching.")
      (speak "OK, got them"))
    (offer $hotel
          utterance: "Do you want [current $hotel]?")))
```

The 'askIfWanted' action is achieved when the system knows that the argument path is either wanted or unwanted – that is, that the user either wants or does not want a hotel. If this information is already known because the user volunteered it previously on his own initiative, the action will not be invoked. Otherwise, it will be invoked and prompt the user with the argument utterance "Do you want a hotel?".

If the user does want a hotel, the WHILE conditional node that follows will allow traversal of the sub-tree below it, giving rise to a sub-dialogue involving fetching the hotel data from the web site, uttering periodic pacifier utterances if the fetch turns out to be lengthy, and finally offering the hotels sequentially for the user to choose from.

The path of nodes from the root of the dialogue model tree is called the frontier path, and is held on the agenda stack. The frontier path gradually moves to the right as the dialogue progresses. A separate pass through the tree, the action restoration pass, scans the completed portion of the tree to the left of the frontier path. If an action in this portion of the tree is found to be suitable for execution in the current dialogue state, it is restored and pushed onto the agenda stack. This may happen if the dialogue state changes so that the action's achievement condition is no longer true, or if its relevancy or executability condition has become true when it was not before.

The WHILE control structure node enforces a relevancy condition for a whole sub-tree. While a WHILE node is executing, the system checks whether its condition is still true each time the dialogue state changes. If at any time the condition ceases to be true, the entire sub-tree below the WHILE node is terminated. To take the hotel reservation example of the introduction, if the user says "Forget it. I don't want a hotel", the whole hotel sub-dialogue is aborted, regardless of whether it is querying the hotel database or negotiating a reservation with the user.

The WHENEVER node plays a complementary role. During the restoration pass of the completed portion of the dialogue tree, if a WHENEVER node is found whose condition is true, that node and the whole sub-tree below it are restored and pushed onto the stack. This facility allows the system to restore previously completed sub-dialogues that have to be done over because of a change in dialogue state. Partial restoration of sub-dialogues is also possible. When the sub-dialogue is restored, if there are actions in the sub-dialogue whose achievement conditions are still met, they will not be executed. Thus, if the user decides to change the date of a booked flight, only the the data fetching and reservation negotiation pieces will be re-executed.

The WHENEVER node is also used to deal with general situations that may occur at any time during the dialogue, such as handling a conflict between the user's constraints, or explicitly confirming a user utterance that the system is not sure about. These special nodes are placed as the leftmost children of the root node, so that they will always be checked first in a restoration pass and given priority.

5. INITIAL EVALUATION

Talk'n'Travel participated in a common evaluation under the Communicator program. The dialogue manager component used embodied an earlier version of these ideas that included the dialogue control scheme presented here but not the event-driven control or barge-in.

The evaluation was conducted by the National Institute of Standards and Technology (NIST) in June and July of 2000, and included systems fielded by 9 different groups (ATT, BBN, CMU, Lucent, MIT, MITRE, SRI, and University of Colorado). A pool of approximately 80 subjects was recruited from around the United States. The only requirements were that the subjects be native speakers of American English and have Internet access. Only wireline or home cordless phones were allowed.

The subjects were given a set of travel planning scenarios to attempt. Each subject called each system once and attempted to work through a single scenario; the design of the experiment attempted to balance the distributions of scenarios and users across the systems.

Following each scenario attempt, subjects filled out a questionnaire to determine whether subjects thought they had completed their task, how satisfied they were with using the system, and so forth. The overall form of this evaluation was thus similar to that conducted under the ARISE program [4]. The table shows the results for Talk'n'Travel:

Talk'n'Travel's score of 80.5% was the highest of all participating systems, and ranked second on overall user satisfaction.

6. CONCLUSION

This paper described our strategy for event-driven dialogue management, and our tree-ordered rule language for dialogue control. The language is flexible, general, and extensible. We feel that the work described here can lead to more sophisticated and capable dialogue management systems in the future.

7. REFERENCE

[1] Stallard, D. (2000) Talk'n'Travel: A Conversational System for Air Travel Planning. In

[2] MITRE (1999) DARPA Communicator homepage http://fofoca.mitre.org/

[3] Ward W., and Pellom, B. (1999) The CU Communicator System. In *1999 IEEE Workshop on Automatic Speech Recognition and Understanding*, Keystone, Colorado.

[4] Den Os, E, Boves, L., Lamel, L, and Baggia, P. (1999) Overview of the ARISE Project. *Proceedings of Eurospeech, 1999*, Vol 4, pp. 1527-1530.

[5] Miller S. (1998) The Generative Extraction Model.

	Task Comp%	Q1	Q2	Q3	Q4	Q5
BBN	80.5%	2.23	2.09	2.10	2.36	2.84
Mean	62.0%	2.88	2.23	2.54	2.95	3.36

Scale: 1 = strongly agree, 5 = strongly disagree

Q1 It was easy to get the information I wanted
Q2 I found it easy to understand what the system said
Q3 I knew what I could do or say at each point in the dialog
Q4 The system worked the way I expected it to
Q5 I would use this system regularly to get travel information

Unpublished manuscript.

[6] Constantinides P., Hansma S., Tchou C. and Rudnicky, A. (1999) A schema-based approach to dialog control. *Proceedings of ICSLP*, Paper 637.

[7] Rudnicky A., Thayer, E., Constantinides P., Tchou C., Shern, R., Lenzo K., Xu W., Oh A. (1999) Creating

natural dialogs in the Carnegie Mellon Communicator system. *Proceedings of Eurospeech, 1999*, Vol 4, pp. 1531-1534

[8] Rudnicky A., and Xu W. (1999) An agenda-based dialog management architecture for soken language systems. In *1999 IEEE Workshop on Automatic Speech Recognition and Understanding*, Keystone, Colorado.

[9] Seneff S., and Polifroni, J. (2000) Dialogue Management in the Mercury Flight Reservation System. *ANLP Conversational Systems Workshop.*

[10] Boyer,L,et al (2000) Voice extensible Markup Language Version 1.0. Web: http://www.w3.org/TR/2000/NOTE-voicexml-20000505/

8. ACKNOWLEDGEMENTS

This work was sponsored by DARPA and monitored by SPAWAR Systems Center under Contract No. N66001-99-D-8615. The author would like to thank Daniel Kiecza and Francis Kubala for their help with telephony and speech recognition and their colloboration on event-driven communication.