AN AGENDA-BASED DIALOG MANAGEMENT ARCHITECTURE FOR SPOKEN LANGUAGE SYSTEMS

A. Rudnicky, Xu W.

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

ABSTRACT

Dialog management can be seen as a solution to two specific problems: (1) providing a coherent overall structure to interaction that extends beyond the single turn, (2) correctly manage mixed-initiative interaction, allowing users to guide interaction as per their (not necessarily explicitly shared) goals while allowing the system to guide interaction towards successful completion. We propose a dialog management architecture based on the following elements: *handlers* that manage interaction focussed on tightly coupled sets of information, a *product* that reflects mutually agreed-upon information and an *agenda* that orders the topics relevant to task completion.

1. INTRODUCTION

Spoken language interaction can take many forms. Even fairly simple interaction can be very useful, for example in auto-attendant systems. For many other applications, however, more complex interactions seem necessary, either because users cannot always be expected to exactly specify what they want in a single utterance (e.g., schedule information) or because the task at hand requires some degree of exploration of complex alternatives (e.g., travel planning). Additionally, unpredictable complexity is introduced through error or misunderstanding. We are interested in managing interaction in the context of a goal-oriented task that extends over multiple turns.

Dialog management in the context of purposeful tasks must solve two problems: (1) Keep track of the overall interaction with a view to ensuring steady progress towards task completion. That is the system must have some idea of how much of the task has been completed and more importantly some idea of what is yet to be done, so that it can participate in the setting of intermediate goals and generally shepherd the interaction towards a successful completion of the task at hand. (2) Robustly handle deviations from the nominal progression towards problem solution. Deviations are varied: the user may ask for something that is not satisfiable (i. e., proposes a set of mutually-incompatible constraints), the user may misspeak (or, more likely, the system may misunderstand) a request and perhaps cause an unintended (and maybe unnoticed) deviation from the task. The user might also underspecify a request while the system requires that a single solution be chosen. Finally the user's conception of the task might deviate from the system's (and its developers) conception, requiring the system to alter the order in which it expects to perform the task. Ideally, a robust dialog management architecture can accommodate all of these circumstances within a single framework.

We have been exploring dialog management issues in the context of the CMU Communicator [3]. The Communicator handles a complex travel task, consisting of air travel, hotels and car reservations.

2. MODELING DIALOG

Existing approaches to dialog management are difficult to adapt to the current problem because they either impose a rigid structure on the interaction or because they are not capable of managing data structures beyond a certain level of complexity. Call-flow based systems (more generally, graph-based systems) handle the complexity of dialog management by explicitly enumerating all possible dialog states, as well as allowable transitions between states. This serves the purpose of partitioning the problem into a finite set of states, with which can be associated topic-specific elements (such as language and interactions with other system components such as a database interaction). Movement between states is predicated on the occurrence of specific events, either the user's spoken inputs or through (e.g.) a change in back-end state. It is the nature of these systems that the graphs are typically trees (where individual nodes correspond to the specification of certain information or the setting of constraints). Except for the simplest tasks, graph systems have a number of limitations. Unless the graph is carefully designed, users will find themselves unable to switch to a topic that is coded in a different sub-tree, without going through the common parent of the two. Often the only way to get there is through the root node of the dialog. Similarly it is not always possible to navigate an existing tree, in order, e.g., to correct information supplied in an earlier node.

Frame-based systems provide an alternate, more flexible approach. Here the problem is cast as form filling: a particular system action is tied to a form that specifies all relevant items of information for an action. Dialog management consists of monitoring the form for completion, setting elements as these are specified by the user and using the presence of empty slots as a trigger for questions to the user. Form-filling does away with the need to specify a particular order in which slots need to be filled and loosens the requirement for the system designed to correctly intuit the natural order in which information is supplied. This in any case is impossible for many tasks, as different users may have different, incompatible, problem-solving styles. While ideally suited for tasks that can be expressed in terms of filling a single form, form-filling can be combined with graph representations (typically ergodic) to support a set of (possibly) related activities, each of which can be cast into a form-filling format.

Both graph and frame systems share the property that the task usually has a fixed goal which can be achieved by having the user specify information (fill slots) on successive turns. Using a filled out form the system can perform some action, such as information retrieval. While this capability encompasses a large number of useful applications it does not necessarily extend to more complex tasks, for example ones where the goal is to create a complex data object, such as a plan (e.g. [1]).

In our own work we have been building a system that allows users to construct travel itineraries. This domain poses several problems: there is no "form" as such to fill out, since we do not know beforehand the exact type of trip an individual might take (though the building blocks of an itinerary are indeed fixed). The system benefits from being able to construct the itinerary dynamically; we denote these solution objects "products". Users also expect to be able to manipulate and inspect the itinerary under construction. By contrast, frame systems do not afford the user the ability to manipulate the form, past supplying fillers for slots. The exception is the selection of an item from a solution set. We do not abandon the concept of a form altogether: an itinerary is actually a hierarchical composition of forms, where the forms in this case correspond to tightly-bound slots (e.g., those corresponding to the constraints on a particular flight leg) and which can be treated as part of the same topic of conversation.

3. TASK STRUCTURE AND SCRIPTS

Intuitively (as well as evident from our empirical studies of human travel agents and clients) travel planning develops over time as a succession of episodes, each focused on a specific topic (such as a given flight leg, a hotel in a particular city, etc.). Users treat the task as a succession of topics, each of which ought to be discussed in full and closed, before moving on to the next topic. Topics can certainly be revisited, but doing so corresponds to an explicit conversational move on the part of the participants.

Consequently we implemented a dialog management strategy that takes advantage of this task structure ([3]). By analogy to what we observed in the human-human data we refer to it as a script-based dialog manager. Script in this context simply refers to an explicit sequencing of task-related topics, Each topic is expressed as a form-filling task, with conventional free-order input allowed for form slots and a slot-state driven mixed-initiative interaction (i.e., ask the user about any empty slot). The topic-specific form is actually composed of two parts: constraint slots (typically corresponding to elements of a query) and a solution slot (containing the result of an executed query).

The control strategy is also actually more complex: slots are pre-ordered based on their (domain-derived) ability to constrain the solution; this ordering provides a default sequence in which the system selects elements to ask the user about. Control is predicated on the state of a slot (whether constraint or solution). The state can either be "empty", in which case the system should ask the user for a value, filled with a single value, in which case it is "complete", or filled with multiple values. The last case is cause to engage the user in a clarification sub-dialog whose goal is to reduce multiple values to a single value, either by selecting an item in the solution set or by restating a constraint. Figure 1 shows the structure of the Flight Leg topic in the script-based system.

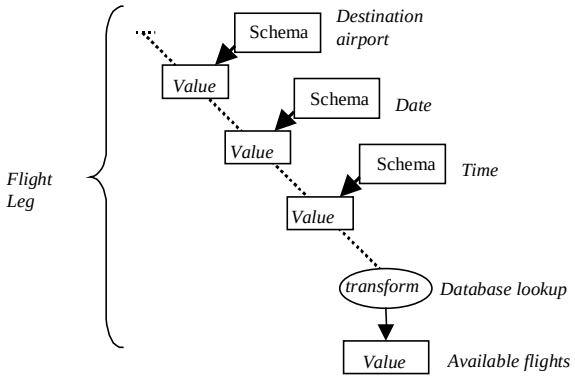

Figure 1 Task-based dialog control in a script-based system, as determined by the structure of a compound schema, with contributions from three simple schema.

4. AN AGENDA-BASED ARCHITECTURE

While capable of handling routine travel arrangements efficiently, the script-based approach has a number of perceived limitations: the script is very closely identified with the product data structure. Specifically, we used a fixed product structure that served as a form to fill out. While the entire form does not need to be filled out to create a valid itinerary, it nevertheless set limits on what the user can construct. Instead we wanted a form structure that could be dynamically constructed over the course of a session, with contributions from both the user and the system. The script-based approach also seemed to make navigation over the product difficult. While we implemented a simple undo and correction mechanism that allowed the user to revisit preceding product elements, users had difficulty using it correctly. While some of the difficulty could be traced to inadequate orientation support, the source was more likely the inability of the system to treat the product structure independent of the script.

We sought to address these problems by introducing two new data structures: an agenda to replace a fixed script and a dynamic product that could evolve over the course of a session. In the agenda-based system, the product is represented as a tree, which reflects the natural hierarchy, and order, of the information needed to complete the task. A dynamic product is simply one that can be modified over the course of a session, for example by adding legs to a trip as these are requested by the user rather than working from a fixed form. Operationally, this means providing a set of operators over tree structures and making these available to the user and to the system. In our case, we defined a library of sub-trees (say air travel legs or local arrangements) and a way to attach these to the product structure, triggered either by the setting of particular values in

the existing tree or through explicit requests on the part of the user ("and then I'd like to fly to Chicago").

Each node in the product tree corresponds to a handler, which encapsulates computation relevant to a single information item. All handlers have the same form: they specify a set of receptors corresponding to input nets, a transform to be applied to obtain a value and a specification of what the system might say to the user in relation to the information governed by the handler. Handlers correspond to the schema and compound schema of the script-based system (see Figure 1).

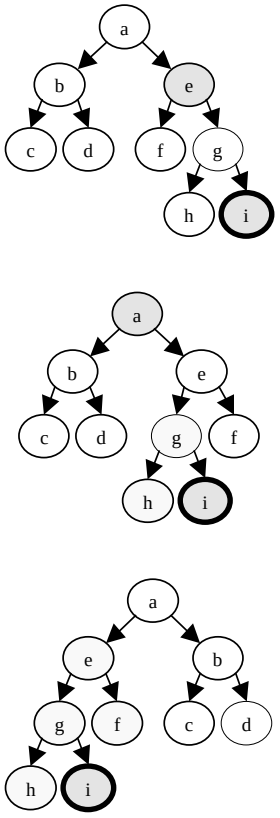

Figure 2 Rotating sub-trees to bring a topic into focus. Node i is the new focus.

The agenda is an ordered list of topics, represented by handlers that govern some single item or some collection of information, The agenda specifies the overall "plan" for carrying out a task. The system's priorities for action are captured by the agenda, an ordered list of handlers generated through traversal of the product structure. The handler on the top of the agenda has the highest priority and represents the focused topic. A handler can capture relevant input from the user and can generate prompts to the user. A single handler deals only with a mini dialog centering on a particular piece of information (e.g. departure date). The agenda is a generalization of a stack. It indicates both the current focus of interaction (i.e., the top-most handler) as well as all undealt-with business, and captures the order in which such business should be dealt with. (The system's high-level goal is to ensure that all values in the current product tree have valid settings.) As all items in the agenda are potentially activatable through what the user speaks, the user has corresponding control over the topic in focus. The agenda also contains generic handlers that sort to the bottom of the agenda. These can be used to consume any inputs that are not caught by product-derived handlers (for example, requests for help).

The order of the agenda is generated from the left-to-right, depth-first traversal of the product tree. When a user input comes in, the system calls each handler per their order in the agenda and each handler will try to interpret the user input. When a handler captures a single piece of information, the information is marked as consumed. This guarantees that a single information item can be consumed by only one handler.

After an input pass, if the user's input does not directly cause a particular handler to generate a question, then the system will go through an output pass during which each handler will have a chance to generate a prompt about itself (e.g. the handler for departure date can ask user for departure date).

The framework can determine the next step from the return code of a handler, it can choose to continue the current pass, exit input pass and switch to output pass, exit current pass and wait for input from user, etc. During a pass, a handler can also declare itself as the focus through its return code. In this case, it will be promoted to the top of the agenda. In order to preserve the context of a specific topic, we use a method called sub-tree promotion. In this method, a handler is first promoted to the left-most node among its sibling. Then its parent node is also promoted this way. Figure 2 gives an example of sub-tree promotion.

The system also handles the dependencies among the nodes of the product tree. A typical dependent relationship is between a parent node and a child node. Usually, the value of a parent node is dependent on its children. Each node maintains a list of its dependent nodes and it will notify its dependents about any changes of its value. The dependent node can then declare itself invalid and therefore a candidate topic for conversation.

The following dialog, generated using the system, shows a number of features: the ability to absorb an implicit change of topic on the part of the user (A1—A3), adding to an existing itinerary (A8—A10) and handling an explicit topic shift (U11).

A1:	Hello. You've reached the C M U Communicator. Tell me your full name
U2:	hi i'd like to fly to seattle tuesday morning
A3:	Traveling to Seattle on Tuesday, August eleventh in the morning. Your full name?
U4:	wei xu
A5:	Hello, Wei, I'm glad to hear from you again. You can now ask me for help.
A6:	OK. Let's see I have a United flight departing Pittsburgh at ten forty five a.m. arriving into Seattle at one fifty seven p.m. Is that OK?
U7:	Yes
A8:	Will you return to Pittsburgh from Seattle?
U9:	no i will go to Los Angeles
A10:	Traveling to Los Angeles. on what date
U11:	Sorry can we go back to talk about the previous leg again
A12:	For the first leg, you are on a United flight 1446 departing Pittsburgh at ten forty five A.M. tomorrow, arriving in Seattle at one fifty seven P.M. Do you want to take that?
U13:	actually i want to leave on wednesday
A14:	a flight on wednesday.
A15:	I have a flight departing Pittsburgh at ten forty five a.m., arrives Seattle at one fifty seven p.m. Is that OK?
U16:	Yes

5. SYSTEM IMPLEMENTATION

The Carnegie Mellon Communicator is telephone-based and is implemented as a modular distributed system, running across NT and Linux platforms. Communication between modules is handled through the DARPA Communicator architecture, based on the MIT Galaxy architecture. Currently the task is captured in an approximately 2500-word language based on corpora derived from human-human, wizard of oz and human-computer interaction in this domain. Domain information is obtained from various sources on the Web; the system is typically configured for about 250 destinations worldwide, with a concentration on North America. The system otherwise understands over 500 destinations worldwide.

A publicly accessible demonstration has been available since the summer of 1998, at 1-877-CMU-PLAN. Current information is available at http://www.speech.cs.cmu.edu/Communicator.

The system uses the Sphinx II ([2]) decoder in a real-time mode and supports barge-in. A top-1 hypothesis is produced by the decoder and parsed by Phoenix ([4]) using a semantic domain-specific grammar. The resulting parse is evaluated for coherence then passed to the dialog manager. The parse is treated as a set of concepts, or nets, and individual handlers in the agenda respond to these. Currently, either individual nets or net-subnet combinations are matched for. Once matched, the contents of the net are either consumed directly (i.e., to set a target value) or transformed through a call to a domain agent. Currently the system has three major domain agents, a travel backend, a date-time module and a user profile module. The transform result is stored as the target value or another action taken, as described previously.

Also associated with each handler are a set of calls to the language generation module (effective a domain agent). Calls correspond to fixed conditions within the handler and specify an output speech act plus the relevant set of concepts. We use a stochastic language generator trained on instances of actual human utterances (those of a travel agent in our case) as well as a history mechanism to generate output. A marked-up string is then passed to a TTS module. We are currently experimenting with a variety of TTS engines.

6. SUMMARY & CONCLUSIONS

The agenda-based approach addresses the problem dialog management in complex problem-solving tasks. It does so by treating the task at hand as one of cooperatively constructing a complex data structure, a product, and uses this structure to guide the task. The product consists of a tree of handlers, each handler encapsulates processing relevant to a particular schema. Handlers correspond to simple or compound schema, the latter acting essentially as multi-slot forms. A handler encapsulates knowledge necessary for interacting about a specific information slot, including specification of user and system language and of interactions with domain agents. Handlers that deal with compound schema coordinate tightly bound schema and correspond to specific identifiable topics of conversation. We define tightly bound as those schema that users expect to discuss interchangeably, without explicit shifts in conversational focus.

We believe that individual handlers can be authored independently of others at the same level of hierarchy, in turn we believe this will simplify the problem of developing dialog systems by managing the complexity of the process.

The agenda contains all topics relevant to the current task. The order of handlers on the agenda determines how user input will be will be attached to product nodes. Both the system and the user however have the ability to reorder items on the agenda, the system to foreground items that need to be discussed, the user to reflect their current priorities within the task.

The mechanisms described in this paper do not cover all necessary aspects of dialog management but do provide an overall control architecture. For example, clarification processes, which involve possibly extended interaction with respect to the state of a value slot, fit into the confines of a single handler. We believe that the agenda mechanism can be adapted easily to less-complex domains that might currently be implemented as a standard form-based system (for example a movie schedule service). We do not know as yet how well the technique will succeed for domains of complexity comparable to travel planning but with different task structure.

7. ACKNOWLEDGEMENTS

This research was sponsored by the Space and Naval Warfare Systems Center, San Diego, under Grant No. N66001-99-1-8905. The content of the information in this publication does not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

We would like to thank the other members of the DARPA Communicator project in the Carnegie Mellon Speech group without whose contributions this work would not have been possible: Eric Thayer, Ravi Mosur, Kevin Lenzo, Paul Constantinides, Rande Shern, Alice Oh, Rita Singh and others.

8. REFERENCES

[1] James F. Allen, Lenhart K. Schubert, George Ferguson, Peter Heeman, Chung Hee Hwang, Tsuneaki Kato, Marc Light, Nathaniel G. Martin, Bradford W. Miller, Massimo Poesio, and David R. Traum, ‘‘The TRAINS Project: A case study in building a conversational planning agent,’’ *Journal of Experimental and Theoretical AI*, 7(1995), 7-48.

[2] Bansal, D. and Ravishankar, M. New features for confidence annotation. In *Proceedings of the 5th International Conference on Spoken Language Processing (ICSLP)*, December 1998, Sydney, Australia

[3] Rudnicky, A., Thayer, E., Constantinides, P., Tchou, C., Shern, R., Lenzo, K., Xu W., Oh, A. Creating natural dialogs in the Carnegie Mellon Communicator system. Proceedings of Eurospeech, 1999, Paper r014.

[4] Ward, W. and Issar, S. Recent improvements in the CMU spoken language understanding system. In *Proceedings of the ARPA Human Language Technology Workshop*, March 1994, 213-216.