TALK'N'TRAVEL: A CONVERSATIONAL SYSTEM FOR AIR TRAVEL PLANNING

David Stallard
BBN Technologies, GTE
70 Fawcett St.
Cambridge, MA, USA, 02238
Stallard@bbn.com

Abstract

We describe Talk'n'Travel, a spoken dialogue language system for making air travel plans over the telephone. Talk'n'Travel is a fully conversational, mixed-initiative system that allows the user to specify the constraints on his travel plan in arbitrary order, ask questions, etc., in general spoken English. The system operates according to a plan-based agenda mechanism, rather than a finite state network, and attempts to negotiate with the user when not all of his constraints can be met.

Introduction

This paper describes Talk'n'Travel, a spoken language dialogue system for making complex air travel plans over the telephone. Talk'n'Travel is a research prototype system sponsored under the DARPA Communicator program (MITRE, 1999). Some other systems in the program are Ward and Pellom (1999), Seneff and Polifroni (2000) and Rudnicky et al (1999). 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 all flights, hotels, and rental cars, all in conversational English over the telephone.

The Communicator common task presents special challenges. It is a complex task with many subtasks, including the booking of each flight, hotel, and car reservation. Because the number of legs of the trip may be arbitrary, the number of such subtasks is not known in advance. Furthermore, the user has complete freedom to say anything at any time. His utterances can affect just the current subtask, or multiple subtasks at once ("I want to go from Denver to Chicago and then to San Diego"). He can go back and change the specifications for completed subtasks. And there are important constraints, such as temporal relationships between flights, that must be maintained for the solution to the whole task to be coherent.

In order to meet this challenge, we have sought to develop dialogue techniques for Talk'n'Travel that go beyond the rigid system-directed style of familiar IVR systems. Talk'n'Travel is instead a mixed initiative system that allows the user to specify constraints on his travel plan in arbitrary order. At any point in the dialogue, the user can supply information other than what the system is currently prompting for, change his mind about information he has previously given and even ask questions himself. The system also tries to be helpful, eliciting constraints from the user when necessary. Furthermore, if at any point the constraints the user has specified cannot all be met, the system steps in and offers a relaxation of them in an attempt to negotiate a partial solution with the user.

The next section gives a brief overview of the system. Relevant components are discussed in subsequent sections.

1 System Overview

The system consists of the following modules: speech recognizer, language understander, dialogue manager, state manager, language generator, and speech synthesizer. The modules

interact with each other via the central hub module of the Communicator Common Architecture.

The speech recognizer is the Byblos system (Nguyen, 1995). It uses an acoustic model trained from the Macrophone telephone corpus, and a bigram/trigram language model trained from ~40K utterances derived from various sources, including data collected under the previous ATIS program (Dahl et al, 1994).

The speech synthesizer is Lucent's commercial system. Synthesizer and recognizer both interface to the telephone via Dialogics telephony board. The database is currently a frozen snapshot of actual flights between 40 different US cities (we are currently engaged in interfacing to a commercial air travel website). The various language components are written in Java. The complete system runs on Windows NT, and is compliant with the DARPA Communicator Common architecture.

The present paper is concerned with the dialogue and discourse management, language generation and language understanding components. In the remainder of the paper, we present more detailed discussion of these components, beginning with the language understander in Section 2. Section 3 discusses the discourse and dialogue components, and Section 4, the language generator.

2 Language Understanding

2.1 Meaning Representation

Semantic frames have proven useful as a meaning representation for many applications. Their simplicity and useful computational properties have often been seen as more important than their limitations in expressive power, especially in simpler domains.

Even in such domains, however, frames still have some shortcomings. While most naturally representing equalities between slot and filler, frames have a harder time with inequalities, such as 'the departure time is *before* 10 AM', or 'the airline is *not* Delta'. These require the slot-filler to be some sort of predicate, interval, or set object, at a cost to simplicity uniformity. Other problematic cases include n-ary relations ('3 miles from Denver'), and disjunctions of properties on different slots.

In our Talk'n'Travel work, we have developed a meaning representation formalism called path constraints, which overcomes these problems, while retaining the computational advantages that made frames attractive in the first place. A path constraint is an expression of the form :

(<path> <relation> <arguments>*)

The path is a compositional chain of one or more attributes, and relations are 1-place or higher predicates, whose first argument is implicitly the path. The relation is followed by zero or more other arguments. In the simplest case, path constraints can be thought of as flattenings of a tree of frames. The following represents the constraint that the departure time of the first leg of the itinerary is the city Boston :

LEGS.0.ORIG_CITY EQ BOSTON

Because this syntax generalizes to any relation, however, the constraint "departing before 10 AM" can be represented in a syntactically equivalent way:

LEGS.0.DEPART_TIME LT 1000

Because the number of arguments is arbitrary, it is equally straightforward to represent a one-place property like "x is nonstop" and a three place predicate like "x is 10 miles from Denver".

Like frames, path constraints have a fixed format that is indexed in a computationally useful way, and are simpler than logical forms. Unlike frames, however, path constraints can be combined in arbitrary conjunctions, disjunctions, and negations, even across different paths. Path constraint meaning representations are also flat lists of constraints rather than trees, making matching rules, etc, easier to write for them.

2.2 The GEM Understanding System

Language understanding in Talk'n'Travel is carried out using a system called GEM (for Generative Extraction Model). GEM (Miller, 1998) is a probabilistic semantic grammar that is an outgrowth of the work on the HUM system (Miller, 1996), but uses hand-specified knowledge in addition to probability. The hand-specified knowledge is quite simple, and is expressed by a two-level semantic dictionary. In the first level, the entries map alternative word strings to a single word class. For example, the following entry maps several alternative forms to the word class DEPART:

Leave, depart, get out of => DEPART

In the second level, entries map sequences of word classes to constraints:

Name: DepartCity1
Head: DEPART
Classes: [DEPART FROM CITY]
Meaning: (DEST_CITY EQ <CITY>)

The "head" feature allows the entry to pass one of its constituent word classes up to a higher level pattern, allowing the given pattern to be a constituent of others.

The dictionary entries generate a probabilistic recursive transition network (PRTN), whose specific structure is determined by dictionary entries. Paths through this network correspond one-to-one with parse trees, so that given a path, there is exactly one corresponding tree. The probabilities for the arcs in this network can be estimated from training data using the EM (Expectation-Maximization) procedure.

GEM also includes a noise state to which arbitrary input between patterns can be mapped, making the system quite robust to ill-formed input. There is no separate phase for handling ungrammatical input, nor any distinction between grammatical and ungrammatical input.

3 Discourse and Dialogue Processing

A key feature of the Communicator task is that the user can say anything at any time, adding or changing information at will. He may add new subtasks (e.g. trip legs) or modifying existing ones. A conventional dialogue state network approach would be therefore infeasible, as the network would be almost unboundedly large and complex.

A signifigant additional problem is that changes need not be monotonic. In particular, when changing his mind, or correcting the system's misinterpretations, the user may delete subtask structures altogether, as in the subdialog:

S: What day are you returning to Chicago?
U: No, I don't want a return flight.

Because they take information away rather than add it, scenarios like this one make it problematic to view discourse processing as producing a contextualized, or "thick frame", version of the user's utterance. In our system, therefore, we have chosen a somewhat different approach.

The discourse processor, called the state manager, computes the most likely new task state, based on the user's input and the current task state. It also computes a discourse event, representing its interpretation of what happened in the conversation as a result of the user's utterance.

The dialogue manager is a separate module, as has no state managing responsibilities at all. Rather, it simply computes the next action to take, based on its current goal agenda, the discourse event returned by the state manager, and the new state. This design has the advantage of making the dialogue manager considerably simpler. The discourse event also becomes available to convey to the user as confirmation.

We discuss these two modules in more detail below.

3.1 State Manager

The state manager is responsible for computing and maintaining the current task state. The task state is simply the set of path constraints which currently constrain the user's itinerary. Also included in the task state are the history of user and system utterances, and the current subtask and object in focus, if any.

The state manager takes the N-best list of recognition hypotheses as input. It invokes the understanding module on a hypothesis to obtain a semantic interpretation. The semantic interpretation so obtained is subjected to the following steps:

1. Resolve ellipses if any
2. Match input meaning to subtask(s)
3. Expand local ambiguities
4. Apply inference and coherency rules
5. Compute database satisfiers
6. Relax constraints if neccesary
7. Determine the most likely alternative and compute the discourse event

At any of these steps, zero or more alternative new states can result, and are fed to the next step. If zero states result at any step, the new meaning representation is rejected, and another one requested from the understander. If no more hypotheses are available, the entire utterance is rejected, and a DONT_UNDERSTAND event is returned to the dialogue manager.

Step 1 resolves ellipses. Ellipses include both short responses like "Boston" and yes/no responses. In this step, a complete meaning representation such as '(ORIG_CITY EQ BOSTON)' is generated based on the system's prompt and the input meaning. The hypothesis is rejected if this cannot be done.

Step 2 matches the input meaning to one or more of the subtasks of the problem. For the Communicator problem, the subtasks are legs of the user's itinerary, and matching is done based on cities mentioned in the input meaning. The default is the subtask currently in focus in the dialogue.

A match to a subtask is represented by adding the prefix for the subtask to the path of the constraint. For example, "I want to arrive in Denver by 4 PM" and then continue on to Chicago would be :

LEGS.0.DEST_CITY EQ DENVER
LEGS.0.ARRIVE_TIME LE 1600
LEGS.1.ORIG_CITY EQ DENVER
LEGS.1.DEST_CITY EQ CHICAGO

In Step 3, local ambiguities are expanded into their different possibilities. These include partially specified times such as "2 o'clock ".

Step 4 applies inference and coherency rules. These rules will vary from application to application. They are written in the path constraint formalism, augmented with variables that can range over attributes and other values. The following is an example, representing the constraint a flight leg cannot be scheduled to depart until after the preceding flight arrives:

LEGS.$N.ARRIVE
LT
LEGS.$N+1.DEPART

States that violate coherency constraints are discarded.

Step 5 computes the set of objects in the database that satisfy the constraints on the current subtask. This set will be empty when the constraints are not all satisfiable, in which case the relaxation of Step 6 is invoked. This relaxation is a best-first search for the satisfiable subset of the constraints that are deemed closest to what the user originally wanted. Alternative relaxations are scored according to a sum of penalty scores for each relaxed constraint, derived from earlier work by Stallard (1995). The penalty score is the sum of two terms: one for the relative importance of the attribute concerned (e.g. relaxations of DEPART_DATE are penalised more than relaxations of AIRLINE) and the other for the nearness of the satisfiers to the original constraint (relevant for number-like attributes like departure time).

The latter allows the system to give credit to solutions that are near fits to the user's goals, even if they relax strongly desired constraints. For example, suppose the user has expressed a desire to fly on Delta and arrive by 3 PM, while the system is only able to find a flight on Delta that arrives at 3:15 PM. In this case, this flight, which meets one constraint and almost meets the other, may well satisfy the user more than a flight on a different airline that happens to meet the time constraint exactly.

In the final step, the alternative new states are rank-ordered according to a pragmatic score, and the highest-scoring alternative is chosen. The pragmatic score is computed based on a number of factors, including the plausibility of disambiguated times and whether or not the state interpreted the user as responding to the system prompt.

The appropriate discourse event is then deterministically computed and returned. There are several types of discourse event. The most common is UPDATE, which specifies the constraints that have been added, removed, or relaxed. Another type is REPEAT, which is generated when the user has simply repeated constraints the system already knows. Other types include QUESTION, TIMEOUT, and DONT_UNDERSTAND.

3.1 Dialogue Manager

Upon receiving the new discourse event from the state manager, the dialogue manager determines what next action to take. Actions can be external, such as speaking to the user or asking him a question, or internal, such as querying the database or other elements of the system state. The current action is determined by consulting a stack-based agenda of goals and actions.

The agenda stack is in turn determined by an application-dependent library of plans. Plans are tree structures whose root is the name of the goal the plan is designed to solve, and whose leaves are either other goal names or actions. An example of a plan is the following:

```
CompleteItinerary =>
   (Prompt "How can I help you?")
   (forall legs $n
        GetRouteInfo
        GetSpecificFlight
        GetHotelAndCar
        GetNextLeg))
```

This is a plan for achieving the goal CompleteItinerary. It begins with a open-ended prompt and then iterates over values of the variable $N for which constraints on the prefix LEGS.$N exist, working on high-level subgoals, such as getting the route and booking a flight, for each leg. The last goal determines whether there is another leg to the itinerary, in which case the itera

The system begins the interaction with the high-level goal START on its stack. At each step, the system examines the top of its goal stack and either executes it if it is an action suitable for execution, or replaces it on the stack with its plan steps if it is a goal.

Actions are objects with success and relevancy predicates and an execute method, somewhat similar to the "handlers" of Rudnicky and Xu (1999). An action has an underlying goal, such as finding out the user's constraints on some attribute. The action's success predicate will return true if this underlying goal has been achieved, and its relevancy predicate will return true if it is still relevant to the current situation. Before carrying out an action, the dialogue manager first checks to see if its success predicate returns false and its relevancy predicate returns true. If either condition is not met, the action is popped off the stack and disposed of without being executed. Otherwise, the action's execute method is invoked.

The system includes a set of actions that are built in, and may be parameterized for each each domain. For example, the action type ELICIT is parameterized by an attribute A, a path prefix P, and verbalization string S. Its success predicate returns true if the path 'P.A' is constrained in the current state. Its execute method generates a meaning frame that is passed to the language

generator, ultimately prompting the user with a question such as "What city are you flying to?"

Once an action's execute method is invoked, it remains on the stack for the next cycle, where it is tested again for success and relevancy. In this case, if the success condition is met – that is, if the user did indeed reply with a specification of his destination city – the action is popped off the stack. If the system did not receive this information, either because the user made a stipulation about some different attribute, asked a question, or simply was not understood, the action remains on the stack to be executed again. Of course, the user may have already specified the destination city in a previous utterance. In this case, the action is already satisfied, and is not executed. In this way, the user has flexibility in how he actually carries out the dialogue.

In certain situations, other goals and actions may be pushed onto the stack, temporarily interrupting the execution of the current plan. For example, the user himself may ask a question. In this case, an action to answer the question is created, and pushed onto the stack. The dialogue manager then executes this action to answer the user's question before continuing on with the plan. Or the state manager may generate a clarification question, which the dialogue manager seeks to have the user answer.

Actions can also have a set of conditional branchings that are tested after the action is executed. If present, these determine the next action to execute or goal to work on. For example, the action that asks the user "Do you want a return flight to X?" specifies the branch to be taken when the user replies in the negative. This branch includes an action that asks the user "Is Y your final destination?", an action that is executed if the user did not specify an additional destination along with his negative reply.

Unlike the approach taken by Ward and Pellom (1999), which seeks to avoid scripting entirely by driving the dialogue off the current status of the itinerary, the Talk'n'Travel dialogue manager thus seeks to allow partially scripted dialogue where appropriate to the situation.

4 Language Generation

The language generator takes a meaning frame from the dialogue manager, and generates a text string in English for it. It uses a set of pattern-based rules that map constraints into alternative syntactic realisations. For example, the following rule allows a constraint on departure time to be realized as "leave at 3 PM" or "3 PM flight":

```
LEG.$N.DEPART_TIME EQ $X
  ⇨ [leave at $X],
    [nom-comp $X]
```

Different realization rules can be selected for depending upon whether the constraint is to be realized as an assertion or as a description. The generation algorithm assembles the selected realizations for each constraint into a simplified syntax tree, selecting appropriate inflections of verb and noun heads as it does so. Terminal values in constraints are realized as type-specific nominals, such as "3 PM" or "Delta".

A crucial feature of the generation process is that it adds to each prompt a paraphrase of the most recent discourse event, corresponding to what the system thinks the user just said. This helps keep the conversation grounded in terms of mutual understanding between the participants.

5 Example Scenario

The following is an example dialog with the system:

S: Welcome to Talk'n'Travel. How can I help you?
U: I need a flight from Boston to Denver tomorrow.
S: On Thursday, November 18, flying from Boston to Denver. What time do you want to leave?
U: I want to leave before 10 AM.
S: Leaving before 10 AM. Do you have an airline preference?
U: Continental.

S: Continental. I'm sorry, there are no flights on Continental leaving before 10 AM. I can get you a flight on Continental leaving at 10:10 AM instead. Would that be OK?
U: Yes.
S: OK, I've got you on Continental flight 429, departing Boston at 10:10 AM, arriving at Denver at 3:44 PM. Do you want to book a return flight from Denver to Boston?
U: Yes, on Monday.
S: Alright, we'll add a flight on Monday, November 22, from Boston to Denver to your itinerary. I have a Continental flight at 7 AM. Is that OK?
U. Anything later?
S: Leaving after 7 AM. There are no flights on Continental after 7 AM. I can get you a flight after 7 AM on Frontier, Northwest, and United instead. Would that be OK?
U: How about United.
S: United. I have a United flight at 8:01 AM. Is that flight OK?
U: Yes.
S: OK, I've got you on United flight 1762, departing Denver at 8:01 AM, arriving at Boston at 1:42 PM.

6 Current Status and Conclusions

The Talk'n'Travel system described here was successfully demonstrated at the DARPA Communicator Compare and Contrast Workshop in June 1999. We are currently collecting data with test subjects and are using the results to improve the system's performance in all areas, in preparation for the forthcoming common evaluation of Communicator systems in June 2000.

To determine the performance of the system, we ran an informal experiment in which 11 different subjects called into the system and attempted to use it to solve a travel problem. None of the subjects were system developers. Each subject had a single session in which he was given a three-city trip to plan, including dates of travel, constraints on departure and arrival times, airline preferences.

8 of the subjects were successful. Of successful sessions, the average duration was 387 seconds, with a minimum of 272 and a maximum of 578. The average number of user utterances was 25, with a minimum of 18 and a maximum of 37. The word error rate of the recognizer was 11.8%.

The primary cause of failure to complete the scenario, as well as excessive time spent on completing it, was corruption of the discourse state due to recognition or interpretation errors. While the system informs the user of the change in state after every utterance, the user was not always successful in correcting it when it made errors, and sometimes the user did not even notice when the system had made an error. If the user is not attentive at the time, or happens not to understand what the synthesizer said, there is no implicit way for him to find out afterwards what the system thinks his constraints are.

While preliminary, these results point to two directions for future work. One is that the system needs to be better able to recognize and deal with problem situations in which the dialogue is not advancing. The other is that the system needs to be more communicative about its current understanding of the user's goals, even at points in the dialogue at which it might be assumed that user and system were in agreement.

Acknowledgements

This work was sponsored by DARPA and monitored by SPAWAR Systems Center under Contract No. N66001-99-D-8615.

The author wishes to thank Scott Miller for the use of his GEM system.

References

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

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

Miller S. (1998) The Generative Extraction Model. Unpublished manuscript.

Dahl D., Bates M., Brown M., Fisher, W. Hunicke-Smith K., Pallet D., Pao C., Rudnicky A., and Shriberg E. (1994) Expanding the scope of the ATIS task. In *Proceedings of the ARPA Spoken Language Technology Workshop*, Plainsboro, NJ., pp 3-8.

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

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

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.

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

Nguyen L., Anastasakos T., Kubala F., LaPre C., Makhoul J., Schwartz R., Yuan N., Zavaliagkos G., and Zhao Y. (1995) The 1994 BBN/BYBLOS Speech Recognition System, In *Proc of ARPA Spoken Language Systems Technology Workshop*, Austin, Texas, pp. 77-81.

Stallard D. (1995) The Initial Implementation of the BBN ATIS4 Dialog System, In *Proc of ARPA Spoken Language Systems Technology Workshop*, Austin, Texas, pp. 208-211.

Miller S. and Stallard D. (1996) A Fully Statistical Approach to Natural Language Interfaces, In *Proc of the 34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, California.