Natural Language Generation in the IBM Flight Information System

Scott Axelrod
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
axelrod@us.ibm.com

Abstract

The IBM flight information system is a speech based conversational system which allows users to create multi-leg airline travel itineraries based on live flight availability information. We discuss here the components of the system relating to natural language generation. The deep generation component of the system decides what to say based on information shared with the user about constraints on flights. The surface generation component generates text based on a simple error catching mechanism and a few basic "generation constructs".

1 Introduction

In this paper we report on our experience in developing the natural language generation component of the IBM flight information system which engages the user in a dialog in English over the phone in order to find an itinerary for airline travel which best meets the user's requirements. Although the focus here is on the airline travel domain, our techniques should apply to many domains where the system helps the user find some data obeying suitable constraints. We consider here both the problem of *deep generation* (deciding what information to say) and *surface generation* (deciding how to say the information in spoken language).

2 Architecture

The system has a central hub that passes a frame of information around a loop of servers. It runs compatibly with the evolving architectural standard of the DARPA Communicator project[1] participated in by several sites including MIT (Seneff et al., 1998), CMU (Rudnicky et al., 1999), University of Colorado (Ward and Pellom, 1999), and AT&T (Biatov et al., 2000). The system obtains flight data by connecting to the SABRE computer reservation system used by many travel agents. Dialog management is based on the form-based dialog manager described in (Papineni et al., 1999). The form based dialog manager generates messages to the user under the circumstance when the user has not yet specified enough constraints to allow for a SABRE query. After that point, the generation components discussed in this paper apply. The deep generation module passes its decision about what to say to the surface generation module in the form of a high level list of what kinds of things to say. The details are passed in the contents of the frame of information which is shared between the servers. Both components of the generation module place information in the frame to provide feedback to the servers implementing speech recognition and natural language understanding.

The generation components discussed here have been implemented in the tcl programming language. The advantage of this over a custom scripting language is that it leverages both the power and flexibility of a fully programmable scripting language which is platform independent. The choice of tcl was one of convenience since it interoperated best with other components of our dialog system.

[1] For a description of the Communicator program and documentation of the architecture see http://www.darpa.mil/ito/research/com/index.html and http://fofoca.mitre.org.

3 Example Conversations

To give the reader a flavor of our system, particularly in regards to natural language generation, we begin by presenting three excerpts from *actual* telephone dialogs with novice users. The user input is the output from automatic speech recognition. The system output does not currently include prosody information; however, punctuation does get translated into pauses.

Excerpt 1: The first excerpt begins after the user has entered their identification code.

1 `SYSTEM: Hello Bob, Please tell me about the first leg of your trip. For example, you can say, I want a flight to chicago leaving next tuesday.`

2 `BOB: I WOULD LIKE TO SEE ALL FLIGHTS FROM NEW YORK TO NATIONAL AIRPORT IN WASHINGTON D.C. NEXT THURSDAY`

3 `SYSTEM: what time would you like to leave on january twenty seventh from`

new york to washington national

4 BOB: TEN A.M.

5 SYSTEM: i see a few flights from new york to washington national which depart about ten A M on january twenty seventh. do you prefer to depart from newark, J F K, or la guardia.

6 BOB: LA GUARDIA

7 SYSTEM: there are several flights from la guardia. you can fly on delta departing at nine thirty A M or ten thirty A M, or on U S air at ten A M. please specify an airline or time.

8 BOB: U.S. AIR

9 SYSTEM: U S-air flight 63 57 departs la guardia at ten A M on Thursday, january twenty seventh and arrives in washington national at eleven seventeen A M. Is this flight okay?

10 BOB: YES

Excerpt 2: The second excerpt regards a flight from New York to Melbourne, Australia. The excerpt is the system response after the user has restricted his New York departure to La Guardia airport, similarly to (Excerpt 1, Line 6):

SYSTEM: i see a couple of flights from la guardia. both of them are on american and make connections in chicago o'hare and los angeles. the first flight departs at nine thirty A M and arrives at eight twenty five A M. the second flight departs at nine thirty A M and arrives at eight twenty five A M.

Note: The two flights differ only in the second of three segments of the trip, although the user would only discover this rather unusual coincidence upon asking for details about the individual flights. The difficulty in this case arises because the mechanism for generation for a very short list of flights has not yet been subsumed as part of the aggregation mechanism (described below) which would have automatically determined that the two flights should be grouped.

Excerpt 3:

1 SYSTEM: I will book this leg. Do you want to continue with another leg? Please say yes or no.

2 JANE: YES I NEED TO CONTINUE BUT I NEED TO HAVE FLIGHT COMING BACK

3 SYSTEM: Okay, continuing with leg two. flying from phoenix to newark on what date?

4 JANE: OKAY THAT WOULD BE FINE JUNE THIRD

5 SYSTEM: what time would you like to leave on june third?

6 JANE: OF THE TWO O'CLOCK IN THE AFTERNOON

7 SYSTEM: america west flight 7 74 departs phoenix at one thirty six P M on Saturday, june third and arrives in las vegas at two thirty eight P M, connecting to continental flight 12 82 departing at three fifteen P M and arriving in newark at ten forty one P M. Is this flight okay?

8 JANE: NO IS THERE A DIRECT FLIGHT FROM PHOENIX TO NEWARK I ON JUNE THIRD ANYTIME TO THIRD

9 SYSTEM: pardon me, what time would you like to leave?

10 JANE: TIME TEN O'CLOCK IN THE MORNING

11 SYSTEM: i can't find any nonstop flights from phoenix to newark on america west which depart about ten A M on june third. you may want to try changing the departure time to seven twenty five A M, eight fifty nine A M, twelve seventeen P M, or three oh five P M.

4 Deep Generation

In this section we will focus on some tasks handled by our deep generation module, emphasizing the role of *constraint functions* as a unifying theme. Although the notion can be developed more broadly in various ways, we will consider a constraint function here to be a function on a single data record consisting of information about a particular flight from one location to another, possibly making connections. Examples of a constraint function are: **timeDepVal**, giving the departure time of the flight; **timeArrClass**, giving the class of the arrival time (before six A.M., between six A.M. and noon, etc); and **connVal** giving the list of connection cities. A *constraint* on a data record is the condition that some given constraint function has a given value.

In a typical turn a user may modify the list of constraints imposed on the flights under discussion[2]. How the system interprets the user input, searches for flights satisfying the constraints, and decides what to say about them are all affected by the shared conversational context between system and

[2]For brevity, we focus in this section on system response to user input whose content consists solely of constraints modifications. Processing of other kinds of input such as questions (e.g. "when does the nine A.M. flight arrive?") is handled similarly.

user. Specifically, we have found the following most useful to keep track of:

1. the constraints the user has imposed on the flights;
2. what information about the user input constraints the system has repeated back to the user;
3. the flights the system has conveyed information about to the user; and
4. the constraints on flights that the system has discovered and whether those constraints have been conveyed to the user or can be deduced by the user.

In this section we focus on two particular cases that need to be handled by any dialog system in which the user and system negotiate to find a suitable record from a source of data: the under-constrained case and the over-constrained case.

4.1 Grouping of Information

In this section we discuss how the system decides what to say in the under-constrained case when there are many flights satisfying the user request. Examples of the system response in this case can be found in (Excerpt 1, Turn 5), (Excerpt 1, Turn 7), and Excerpt 2. The following example occurred when a user requested a departure after 10:00 A.M., after having previously imposed the constraints of flying from Chicago to Miami on March third. The system responded as follows:

```
(1) there are several flights which depart
    after ten A M.
(2) all of them leave from chicago o'hare
    and arrive in the afternoon.
(3) do you prefer to fly on american or
    united.
```

Part (1) of the system response summarizes the most salient constraints of the user input using the summary script of section 5[3]. Part (2) is a specification of the significant information common to all flights. In part (3), the system has decided which under-specified constraint is most likely relevant to the user, grouped the flights according to the values of the constraints, and prompted the user by specifying the possible values of the constraint.

The significant common information in part (2) and the most relevant grouping in part (3) are computed by what we call *the aggregation algorithm*[4]. The principal domain dependent data needed by the algorithm consists of utility functions for each constraint telling how high a priority it is to go into detail about that constraint. The output is a tree structure which represents the hierarchy of constraint information that is deemed most useful to convey to the user.

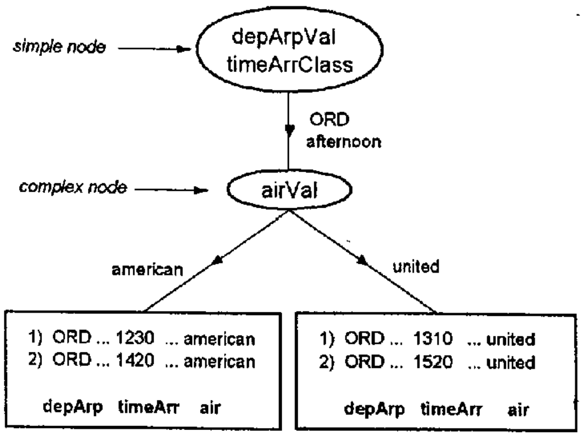

Figure 1: Example of an Aggregation

More specifically, the inputs to the aggregation algorithm consist of a flat list of data records (e.g. a table of flights) together with a list of *aggregation specifications*. An aggregation specification is a triple consisting of: (1) a constraint function by which data may be grouped, (2) a sort function which orders the groups according to their constraint value, and (3) a utility function to determine how useful this grouping is (which may depend both on conversational context as well as when in the algorithm the grouping is attempted). The utility functions also have the ability to return codes that control the search for the best tree. For example, a utility function can declare itself to be the highest priority, thus pruning the search. The output is a tree with non-terminal nodes labeled by lists of constraint functions, edges labeled by values taken by the constraint functions labeling the node above, and terminal vertices labeled by a list of the data records satisfying the constraints specified by the labelings of all its ancestor nodes and edges.

For the example discussed above, the output of the aggregation algorithm is depicted in Figure 1. The top node and the edge below it indicate that all the flights leave from Chicago O'Hare in the afternoon (i.e. the constraint **depArpVal** takes on

[3] Some readers may have noticed that, in (Excerpt 1, Turn 5), the system unnecessarily reviewed constraints that have recently been reviewed. This is because the generation mechanism used before enough constraints have been satisfied to query the data base has not yet been fully unified with the mechanism discussed in this paper.

[4] The term "aggregation" is sometimes used within the generation community referring to a process of combining groups of linguistically similar phrases. One might say the aggregation here is occurring on a *semantic* level, i.e. the internal representations of the flights are being grouped.

the SABRE code "ORD" for Chicago O'Hare and the constraint **timeArrClass** takes on the value "morning"). We call this node a *simple node* because there is only one edge emanating from it. By contrast, the node below is a *complex node* since the constraint function at that node **airVal** can take on more than one value (either "american" or "united"). The box on the lower left contains those input flights which are on American Airlines and depart from Chicago O'Hare in the morning, and similarly for the box on the lower right.

For our application we have found it best to use the same kind of utility function for all constraints. When only this type of utility function is used the behavior of the aggregation algorithm is quite simple and always produces a tree similar to the one in Figure 1, namely one with two nodes: a simple one above a complex one. Corresponding to the notion of simple node and complex node, we call a constraint function (chosen from the aggregation specification) *simple* if it yields the same value when applied to all of the data records and *complex* otherwise. The simplified aggregation algorithm effectively proceeds as follows:

(1) For each simple constraint function (whose value is not known to the user based on the conversational history) apply a significance test. Place those constraints functions that pass the test (if there are any) in the top node of the tree.

(2) Pick the complex constraint function of maximum positive utility and place that in the node below the top. If all utilities are negative, the node remains empty.

As an example, when **depArpVal** is a simple constraint it is deemed significant if it is *not* the only airport serving the departure location the user requested. In our example, since Chicago is served by both O'Hare and Midway airports, the fact that all flights land in O'Hare is deemed significant to tell the user. As our airline travel system develops we expect to have available more expert knowledge about the airline travel domain. For example, the significance test for **depArpVal** may be modified in the future if the system has a way of knowing that Chicago O'Hare is the airport the user would naturally expect in many circumstances.

4.2 Relaxation

In this section, we consider the over-constrained case in which no suitable flights can be found that satisfy the user request. One example of the system response in such a case occurs in (Excerpt 3, line 11). Another example is the following:

```
(1) there don't seem to be any nonstop
    flights from san francisco to newark
    new jersey on united which serve
    breakfast and depart after nine A M
    on february tenth.
(2) you may want to try changing your
    choice of meal, the airline to
    continental, or the departure time
    to seven oh five A M or eight twenty A
    M.
```

In part (1), the system first reviews detailed information about what it believes the current user request is. This is particularly useful to help alert the user to any previous conversational error. In part (2), the system suggests possible relaxations that may be of interest to the user. A relaxation here is just a change of a single constraint in the user request which would allow flights to be found. For example, the system response (2) above indicates that there are flights on united which satisfy all of the other user constraints listed in (1) above.

5 Surface Generation

There are many approaches to generating text from an underlying semantic representation. Simple templates are adequate for many purposes, but result in a combinatorial explosion in the number of templates required to produce output for all possible circumstances. There are also several powerful generation packages available. One package in particular that we found it insightful to experiment with was FUF(Elhadad, 1989), which is short for "Functional Unification Framework"(Elhadad and Robin, 1992). FUF comes available with a reusable grammar of English(Elhadad and Robin, 1996). Although we found the sophisticated linguistic framework of FUF/SURGE difficult to adapt to our needs, we have found it helpful to include analogues of some elements of that framework in the approach we now describe.

After our initial experiments, we decided to "evolve" a surface generation module starting with the straight forward model of template filling and procedure calls provided by the programming language tcl. To overcome the problem of combinatorial explosion in program size, our surface generation makes use of an exception catching mechanism which allows sub-phrases within a complicated phrase to be "turned on" if the semantic input required for them is present. This can be done recursively. This approach has a side benefit of being very robust because detailed error catching is built in. Even if the script writer makes an unintentional error in part of a script (and no alternatives for generating the information in the erroneous part are available) only that part will fail to be generated.

Our system makes available to the developer several useful domain independent *constructs*. In addition to these basic constructs, our surface generation

```
[opt-s {[DoStops $stops]}] [opt-s {$rtow}] [Noun flight]
[opt-s {from [DoArp $locFr]} ] [opt-s {to [DoArp $locTo]} ]
[opt-s {on [DoAir $air]} ]
[opt-s { which
          [NonEmptyConjunction [list
            [opt-s {[Verb cost] [DoPriceRange $price]}]
            [opt-s {[Verb have] flight number $fltNum}]
            [opt-s {[Verb serve] $meal}]
            [opt-s {[subst $::Script(VPDep)]} ]
            [opt-s {[subst $::Script(VPConnect)]} ]
            [opt-s {[subst $::Script(VPArr)]} ]     ]] } ]
```

Figure 2: Fragment from summarization script (generating text after vertical bar in examples in Table 1).

has a morphology module (giving the correct form of a word based on number, tense, etc.) and a library of routines for generating simple phrases. To give the reader a flavor of our approach, we discuss the example of the script which generates phrases such as those in Table 1.

1. There are | several flights.
2. I can't find any | roundtrip flights from New York to Chicago.
3. There don't seem to be any | nonstop flights which serve breakfast and make a connection in Dallas.
4. There is only one | flight on American which departs between six p m and nine p m on February second and arrives in the morning on February third.
5. I see quite a few | flights which cost less than \$1000 and arrive in the morning.

Table 1: Sample output from summarization script. (The vertical bar has been added to demarcate the separation between parts generated by separate subscripts.)

Phrases such as the ones above are generated by surface generation when it is asked by deep generation to summarize some of the constraints on what kind of flight the user is looking for and the approximate number of flights found. The script fragment in Figure 2 generates phrases like the ones after the vertical bar in the above examples. Variables such as **locFr**, **dateDep**, and **air** correspond to user specified constraints on departure location, departure date, airline, and so on. Only those variables will be set which deep generation has decided should be summarized. Since there are thirteen variables referred to in the short script below and the (even shorter) subscripts it refers to, they are capable of generating 2^{13} different kinds of phrases expressing the desired content. It is perhaps a fortunate property of the airline travel domain we are restricting to that this approach allows fairly simple scripts to be used in circumstances where an inordinate number of templates would have been required.

We offer a few words of explanation of the script in Figure 2. First, the "morphology" procedure **Verb** provides the appropriate morphological form of a verb (depending on the current setting of number, tense, etc.). The procedure **subst** is used for expanding the subscripts referred to. The procedures **DoAir**, **DoArp DoPriceRange**, and **DoStops** are from the "phrase library". They generate appropriate phrases associated with an airline, an airport, a price range, or whether or not a flight is nonstop. One may think of these as rules for converting the semantic information, previously determined by deep generation and stored in variables such as **air** and **price**, into a surface realization. For example, "`[DoAir $air]`" returns "`American`" and "`[DoPrice $Price]`" returns "`less than $1000`".

The construct **opt-s** (short for optional substitution) includes the text generated by expanding its argument if that expansion is successful, or else catches and ignores any errors if the expansion was not successful. The construct **NonEmptyConjunction** is used to adjoin a list of phrases. (The separators between phrases are optional arguments.) If the input list is empty, however, an error is generated. In such a case (e.g. examples 1 and 2 above), the error is caught by the enclosing **opt-s**, so the entire "which" clause is omitted.

Another example of a construct is **SayOnce**. This is used when generating a list of phrases, so that a particular script fragment will only be expanded and included the first time it is encountered. For example, **SayOnce** has been used to omit the second occurrence of the word "departing" in (Excerpt 1, Turn 7). Similarly, in the following response to a user query about the arrival times of the flights under discussion, the second occurrence of the word "flights" has been omitted by a simple application of SayOnce:

```
i see at least 3 flights which arrive
between two P M and six P M, and 4
which arrive between six P M and ten P
M.
```

6 Conclusion

In developing our deep and surface generation modules we have followed a strategy of starting with a simple approach and adding basic building blocks as they are warranted, for example the generation constructs described in section 5 and the utility functions of sections 4.1. This strategy has helped us develop generation modules which are flexible, robust, and interact well with the other components of our system. Also, the tools presented here tend to reduce the growth in code size with complexity (as measured by the number of possible constraints).

We are optimistic that these methods can be applied to other domains, although certainly additional features would have to be added. For instance, in Excerpt 2, we gave an example of a shortcoming of our system that arose when we summarized details about a very short list of flights. This problem could be fixed either by subsuming the case of a very short list of flights into the general aggregation mechanism or by adding an additional mechanism to handle this separate case better. Since the problem seemed insignificant enough in the airline travel domain we have not yet resolved it, but we expect that experience with other domains will dictate the best approach.

We consider it to be an advantage of this approach that it is not tied to a particular linguistic framework and affords rather straight forward development. This certainly seems appropriate for our application so far, where the summary script of Figure 2 represents the typical level of complexity of the scripts we have had to develop. It is possible that this could become a limiting factor as the complexity, scope, and variety of domains increases. However, we expect other limitations to become more pressing. For example, we plan to investigate additional building blocks which will be useful as we begin to delve into issues such as improving our help messages or adding emphasis to particular parts of the information we want to convey, either via prosody or more finely crafted text.

Acknowledgements

The author would like to thank Mark Epstein, Adwait Ratnaparhki, Salim Roukos, Kishore Papineni, and Todd Ward for helpful discussions. This work was supported in part by DARPA contract # MDA972-97-C-0012.

References

K. Biatov, E. Bocchieri, G. Di Frabbrizio, C. Kahm, E. Levin, S. Narayanan, A. Pokrovsky, P. Ruscitti, M. Rahim, and L. Walker. 2000. Spoken dialog systems: Some case studies from AT&T. In *Presentation at DARPA Communicator Workshop*, Charleston, SC, Jan. 2000. See http://www.dsic-web.net:8501/pub/comm_2000jan/ATT-Narayanan.pdf for presentation and http://www.dsic-web.net/ito/meetings/communicator_jan00/agenda.html for conference agenda.

Michael Elhadad and Jacques Robin. 1992. Controlling content realization with functional unification grammars. In *Aspects of Automated Natural Language Generation*, Lecture Notes in Artificial Intelligence, 587, pages 89–104. Springer, Berlin.

Michael Elhadad and Jacques Robin. 1996. An overview of SURGE: A re-usable comprehensive syntactic realization component. In *Proceedings of the 8th International Workshop on Natural Language Generation*, Beer Sheva, Israel.

Michael Elhadad. 1989. FUF: The universal unifier user manual. Technical report, Department of Computer Science, Columbia University. URL = http://www.cs.bgu.ac.il/surge/index.htm.

K. A. Papineni, S. Roukos, and R. T. Ward. 1999. Free-flow dialog management using forms. In *Proceedings of Eurospeech-99*, pages 1411–1414, Sept. 1999.

A.I. Rudnicky, E. Thayer, P. Constantinides, C. Tchou, R. Shern, K. Lenzo, W. Xu, and A. Oh. 1999. Creating natural dialogs in the Carnegie Mellon Communicator system. In *Proceedings of Eurospeech-1999*, pages 931–934, Budapest, Hungary, Sept. 1999.

S. Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue. 1998. Galaxy-II: A reference architecture for conversational system development. In *Proceedings of ICSLP-1998*, pages 1153–1156, Sydney, Australia, Nov. 30–Dec. 4, 1998.

Wayne Ward and Bryan Pellom. 1999. The CU Communicator system. In *1999 IEEE Workshop on Automatic Speech Recognition and Understanding*, Keystone Colorado, Dec. 1999.