

LET'S GO: Improving Spoken Dialog Systems for the Elderly and Non-natives

Antoine Raux, Brian Langner, Alan W Black, Maxine Eskenazi

*Language Technologies Institute
Carnegie Mellon University*

{antoine, blangner, awb, max}@cs.cmu.edu

Abstract

With the recent improvements in speech technology, it is now possible to build spoken dialog systems that basically work. However such systems are designed and tailored for the general population. When users come from less general sections of the population such as the elderly and non-native speakers of English, the accuracy of dialog systems degrades.

This paper describes Let's Go, a dialog system specifically designed to allow dialog experiments to be carried out on the elderly and non-native speakers in order to better tune such systems for these important populations. Let's Go is designed to provide Pittsburgh area bus information. The basic system is described and our initial experiments are outlined.

1. Background

Although many recent dialog systems have shown that we can achieve effective spoken interaction with a computer, they tend to target the "average" portions of the population, those whose speech and hearing fall within the norm of the whole population. This has taught us much about how the dialog must change when people interact with a computer instead of a human. We have developed system architectures capable of finding and presenting useful information for the average user. But these systems cannot be used by everyone. People who are, for some reason, considered to be outliers in the general population cannot yet access the information these systems provide. The objective of the Let's Go project is to create a basic dialog system that we can use to test how to extend system access to extreme populations. The portions of the population that we have chosen as representatives of the extreme are the elderly and non-native speakers of English.

As we age, perception is lessened, attention is narrowed and memory is limited. This makes it extremely difficult to listen to and use the information a dialog system furnishes. When we speak a foreign language, we often have not mastered all of its sounds or its grammatical constructions. This makes it difficult for a dialog system to understand what we want. Our populations therefore complement each other since the elderly provide an extreme in the use of speech output and non-natives do the same for speech input.

Our specific interest in creating a dialog system for such populations came first from an observation that elderly visitors had apparently a much harder time using our spoken dialog systems than younger users. To investigate this, we devised a simple experiment with elderly visitors to CMU's Homecoming testing their comprehension of natural and synthetic speech over the telephone under a number of conditions. The results, [1], show a drop in comprehension as age increases.

In this paper we will describe the basic dialog system that we have created to serve as our testbed. Let's Go has its roots

in the CMU Communicator [2] system architecture. From that starting point, and with our experimental goals in mind, we have made modifications to the basic architecture making it easier to change necessary parts of the system, as we adapt to the new populations, such as making the parser more tolerant to grammatical errors. With the system now in place we are experimenting with ways to: enhance the speech output so that the elderly can understand it better; detect what a non-native speaker meant to say and offer hints of how to say it better.

Our system provides bus schedule information for the city of Pittsburgh. We are working with the Port Authority Transit System (PAT) to use their bus schedules and recordings of actual calls to their help desk to build our system.

2. Architecture

In order to be able to develop and test techniques in improving spoken dialog systems, a baseline system was built. CMU has significant experience in building spoken dialog systems. Since we want to have significant control of the dialog system, completely off-the-shelf systems like VoiceXML would be too restrictive. We therefore chose to build on the RavenClaw [3] system. This system is in turn built upon the MIT Galaxy architecture [4] and uses the CMU Sphinx speech recognizer [5] and the Festival Speech Synthesis System [6].

2.1. Telephone connection

Let's Go is connected to a phone line via a Gentner board that can support any telephony system that is supported by the Galaxy architecture.

2.2. Recognizer

We use the CMU Sphinx II speech recognizer with gender-specific telephone quality acoustic models from the Communicator system [2]. The data used for training consists of the CMU Communicator data collected over the last 4 years. We automatically split this data into male and female speech and trained separate models. Both models are then run in parallel and the best is selected. Like others, we have found this improves recognition accuracy.

We do have access to recordings from the PAT help-line although the content is often more general than just bus schedules, and the data has acoustic artifacts from the archiving compression used and therefore does not reflect the acoustic conditions of the telephone speech we expect. Thus, at present, using existing telephone band-width models is appropriate, but as we collect data, we will retrain our system.

2.3. Parsing and Language Modeling

Sphinx II uses a statistical language model (n-grams) for recognition. The result of the recognition is then parsed by Phoenix, a robust parser based on an extended Context Free Grammar allowing the system to skip unknown words and perform partial parsing [7].

Ideally, we would like to train the statistical language model on a corpus of transcribed dialogs corresponding to our particular task. Since the project started relatively recently and it took time to obtain proper permission to record calls to the Port Authority, we have just begun to receive specific data for our task and have not yet had time to preprocess it. The only Port Authority data we have used in the system so far is the set of official names of the bus stops, as stored in the schedule database.

Our approach to language modeling was to first write a grammar for our parser, then generate an artificial corpus of text from the parsing grammar and third, train a statistical language model on the artificial corpus. We wrote the grammar based on a combination of our own intuition and a small scale Wizard-of-Oz experiment we ran. The grammar rules used to identify bus stops were generated automatically from the schedule database.

In order to make the parsing grammar robust enough to parse fairly ungrammatical, yet understandable sentences, it was kept as general as possible. When used for speech generation, however, a very general grammar produces a very large amount of not only ungrammatical, but unnatural sentences. We therefore modified the grammar to make it suitable for speech generation and enhanced it by weighting the rules according to our observations of how frequent they are in natural language. We also adjusted the weight of the bus stop names according to how frequently they are likely to be present in user requests, again based on our own observations. Using the modified grammar, we generated a 200,000-sentence corpus which is large enough to cover most of the bus stop and time expressions in the domain. We trained a 3-gram model on the corpus using the CMU-Cambridge Statistical Language Modeling Kit [8].

Although the resulting language model is not as good as one built from real data, it allows us to obtain a usable prototype with which we can now collect and transcribe dialogs that take place in the experiments with extreme populations, while we await preprocessed real training data.

We are approaching the language modeling and dialog management with one of the main goals of the project in mind — detecting incorrect lexical and grammatical structures in non-native speech and offering correction. The language model on the one hand needs to be general enough to accept sentence structures and use of expressions that are not quite correct. For example, asking for “the coming bus” instead of “the next bus”, or “when the bus is coming” instead of “when is the next bus coming” should be acceptable to our system. However, the phrase “when done bus come here” would be difficult to accept. By accepting the former examples, we then want to give the user subtle correction help so that the next time he/she uses the word or expression it is correct. But this is not a language learning system. Some of the users are calling just before they run out the door to catch the bus. We therefore have at the most two short sentences for the correction. We are starting to build utterances where we take the incorrect response, “the coming bus”, for example, and respond with “You want the next bus?”, with higher pitch and intensity on the correct word, “next”. This advances the dialog while giving corrective information at the same time.

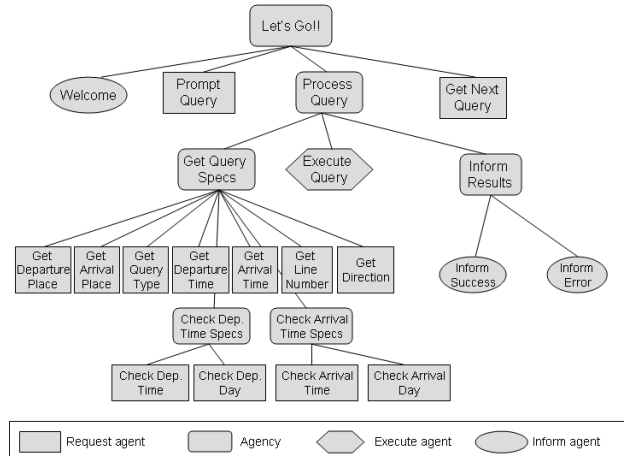


Figure 1: Tree Structure of the Dialog Agents

2.4. Dialog Manager

Dialog management in the Let’s Go system is based on Raven-Claw [3], a generic framework for dialog management. Raven-Claw features a task-independent dialog engine that carries out the dialog according to a task specification. To build a dialog system for a new task, a dialog specification is written as a tree structure where each node represents a “dialog agent”. Figure 1 is a simplified representation of the tree structure. Each leaf agent is in charge of one of four elementary tasks:

- Inform: send output to the user
- Request: request information from the user
- Expect: expect information without explicit request
- Execute: performs non-dialog tasks such as calling the database module

Non-terminal agents (a.k.a. agencies) group other agents and control their execution, capturing the higher level temporal and logical structure of the dialog.

Each call to the system starts with a welcome message that prompts the user to make a request. The system waits for the user’s answer and grabs concepts such as question type (e.g. “When is the next bus to X?”, “How can I go from X to Y?”) or departure and arrival times and places. If it has enough information to be able to submit a query to the database, it does so, presents the results to the user, and prompts for a new query. If more information is needed to make a complete query (e.g. the user gave the destination but did not specify a place of departure), the system explicitly prompts the user to provide the missing information (e.g. “Where are you leaving from?”).

Some agents (not represented in Figure 1) handle the explicit confirmation of recognized concepts. In the current system, each time a new piece of information is obtained from the user, the system repeats what it understood so that the user can detect and verbally correct misrecognitions as they occur.

2.5. Backend Manager

The backend of the system is primarily a database of bus schedules and routing information provided by the Port Authority. The Port Authority system consists of 15,218 stops (although some stops have multiple names). There are 2423 routes (including variations of routes according to time of day or week).

Although we have the database tables that the Port Authority uses internally, we had to make several alterations in how the data is stored to make it possible to find the information we need to provide to the user because the original structure was not suited for retrieving time information. Also, the Port Authority data often contains abbreviations for stops — sometimes several different abbreviations for the same stop. These abbreviations were converted to a consistent form that is more suitable for speech input and output. To more easily match stop names, we chose to fully expand all abbreviations, since that allows us to say that “5th Ave”, “5th Ave.”, and “Fifth Avenue” all reference the same street, but maintains its distinction with “5th Street”.

Stops can be specified in one of three ways: the nearest intersection to the stop (“Forbes [Avenue] at Murray [Avenue]”), a neighborhood (“Oakland”), or a landmark or other point of interest (“Pittsburgh International Airport”, “Waterworks Mall”). Of these, only the first are stored in the database. Thus we map neighborhoods and landmarks to intersections. However, both neighborhoods and landmarks can refer to several different stops. We need to be able to denote stops as departure or arrival points, and also identify which stop in a list is the one being referred to, given the context of the rest of the query. For example, not all stops within a neighborhood are on every route that goes through that neighborhood. At this point, this is handled by a “Stop Matcher” module that creates a mapping between a neighborhood or landmark and the appropriate stop. It will currently only return a single stop, although it will eventually need to return multiple stops.

The backend receives query information from the dialog manager, which consists of the type of query, information identifying the departure and arrival points, possibly a specific route, and time information. The departure and arrival information is passed through the Stop Matcher module to identify which stops to search for in the database. After executing one or more lookups to retrieve information from the database, the module then responds to the dialog manager with a result, which either contains the answer to the user’s query, or a failure code identifying a problem (for example, asking for a time for a bus going between two stops that are not connected by any bus route).

2.6. Language Generation

For language generation, we are using Rosetta, which is a language generation toolkit originally designed for the CMU Communicator. Rosetta is capable of generating utterances from templates, filling in slots with information received from the dialog manager. It can also randomly select from a list of templates for a given response. The generated utterances are then sent to the TTS module (in this case, Festival) for synthesis.

Rosetta identifies different kinds of actions that require utterances to be generated; these actions are their own self-contained modules that have mappings between different concepts and the templates that generate utterances for those concepts. This system uses three different modules for language generation: one to provide information to the user, one to request information from the user, and one that confirms information the user has given the system. Within these modules, there are a variety of templates that generate utterances. For example, the Request module has a “query.departure_place” template which generates the utterance “Where are you leaving from?”, which requests the corresponding concept from the user. The Inform module has a “current_time” template that randomly generates “The time is now [current time].” or “It is currently [current time].”

2.7. Synthesis

Since one major part of this project is to investigate the best output voice quality, we want to have significant control of the synthetic voice output. In our initial design we opted for the easiest solution that would give us a working system. Our very first version simply used a diphone synthesizer. The quality of the latter is basically inadequate for anyone but the dedicated to understand, and particularly not suitable for our target groups, the elderly and non-natives, with limited abilities in English.

Once we had a basic system running with a relatively stable language generation system, we built a limited-domain synthesizer using the techniques described in [9]. That is, we built a specific synthesized voice that is explicitly designed for the type of output we required. To do this we programmatically constructed all the phrases and templates that the language generation system could output. We then filled in the bus stop names, bus numbers, times etc, generating a list of sentences, (around 12,500). We then synthesized these to phoneme strings and greedily selected utterances with the best diphone coverage. This generated a list of 202 utterances. Then we removed this from the completed list and greedily selected a second set. This was done three times, generating a diphone-rich prompt set (for our domain) of 600 prompts. These were recorded and a voice was automatically built using the FestVox [10] build process including labeling phonetic data with a Sphinx acoustic model trained from this data.

The result is a high quality voice, that works well for the sentences generated by our system. It does not, however, deal with all bus stop names at present (or at least not consistently well). We are currently working with a subset of the 15,000. A better name-specific selection technique would potentially offer more consistent coverage.

3. Discussion

We deliberately choose the constrained domain of bus information for our domain, so that we can better control the low level aspects of information presentation rather than be distracted with the higher level problems in dialog control. However, although apparently simple at first, there are still a number of issues in the bus information domain that must be addressed.

Naming a bus stop appears simple at first but more complex as you look closer at the system. Users may employ a wide range of phrases to refer to bus stops, and we wish to make the system deal with this. Although there are problems in abbreviation expansion, and typos in the core data, the idea of a canonical name for a bus stop is fairly well defined. “Murray at Forward” refers to the stop on Murray Avenue where it meets Forward Avenue. However there may be multiple stops at a crossroads that have different names. “South Craig at Forbes” is not the same bus stop as “Forbes at South Craig”, though a human user is likely to conflate these as “Forbes and Craig”.

Our initial system offers a number of rule-derived alternative names for stops but this will be insufficient in a general system. Thus we are investigating methods to give more general matches of user-given names to bus stops, which must also be efficient. For example “Forbes” (a major thoroughfare in Pittsburgh) has many stops and is used by many routes, thus “Forbes” alone seems insufficient to identify a stop. But the 54C routes only have one stop on Forbes thus it is possible to refer to that stop in the context of the 54C: “from East Carson and 22nd to Forbes”

SYSTEM: Welcome to the CMU Let's Go! Bus Information System. How may I help you?

USER: I'd like to go to Forbes and Murray.

SYSTEM: To Forbes and Murray. Where are you leaving from?

USER: Carnegie Mellon.

SYSTEM: From Carnegie Mellon. There are five buses running between Carnegie Mellon and Forbes and Murray: the 501, the 59U, the 61A, the 61B, and the 61C. What else can I do for you?

USER: When is the next one?

SYSTEM: The next 61C going to Forbes and Murray will leave Carnegie Mellon at 5:13 PM.

Figure 2: Example of a Let's Go dialog

4. Evaluation

So far, we have only carried out empirical evaluations of the system.

An initial experiment was to try to elicit how users might speak to a bus information system. The idea was to see how they would formulate their queries in specific situations. We designed 5 scenarios for which the user needed to get some specific information on a bus (e.g. line number between a start point and an end point or time of the next bus at a given stop). We set up a dedicated phone line in our office and asked people in the Language Technologies Institute to pick one or two scenarios and call us. We did not try to emulate human-machine conversations and rather acted as if we were operators from the Port Authority. In all, we recorded 28 phone calls from 17 different callers (7 native and 10 non-native speakers of English). This data was used to manually extend the initial set of grammar rules for parsing and refine our dialog model.

The information gathered from this experiment was used in designing the input language for the system.

Since our initial telephone-based system has only recently become operable, we have not as yet carried out any formal tests. However we have made the following observations.

The system works well for simple requests. When some information is missing, it is able to request it explicitly from the user. Hence, the dialog can be very short when the user expresses a complete query in one sentence (e.g. "When is the next bus leaving X going to Y?"). It can be longer and more system-directed if part of the request is missing or not recognized (see Figure 2 for an example of such a dialog). Systematic explicit confirmation from the system can be annoying for some users but we found that, given the current number of speech recognition errors, it is important for the user to monitor the understanding of the system.

As said above, speech recognition is acceptable but far from perfect. We think that this is mainly due to the limitations of the "artificial" language model. As we get more experience with the system and collect data from a wider range of users, we are adjusting the generative grammar and thus improving the LM's quality. Ultimately, we will collect enough real data to train a model directly on it.

Our baseline synthesizer was the standard diphone synthesizer in Festival which is not sufficient (particularly over the telephone), hence our move to a domain synthesizer. Although building a domain synthesizer is more work, it is clear that a better output voice is necessary before we can make the system

available to a wider populations.

Naming bus stops is a non-trivial problem and we are looking at general techniques to be able to match what our users may say when referring to stops

5. Conclusions

We have described the Let's Go spoken dialog system. A telephone-based mixed-initiative spoken dialog system for Pittsburgh area bus information. The individual components are described highlighting the specific issues in constructing such a system with general dialog tools. Let's Go is specifically designed to improve dialog systems for the elderly and non-native speakers of English, two important populations who have difficulty in using standard spoken dialog systems.

The Let's Go project is currently setting up specific experiments with our target populations to better understand their limitations in accessing information through telephone-based spoken dialog systems.

6. Acknowledgements

This work was funded in part by NSF grant 0229715 "LET'S GO: improved speech interfaces for the general public". The opinions expressed in this paper do not necessarily reflect those of NSF. We would like to thank Maureen Bertocci, and Megan Huff from Pittsburgh Port Authority Transport for their help in this work.

7. References

- [1] M. Eskenazi and A. Black, "A study on speech over the telephone and aging," in *Eurospeech01*, Aalborg, Denmark, 2001.
- [2] A. Rudnicky, C. Bennett, A. Black, A. Chotimongkol, K. Lenzo, A. Oh, and R. Singh, "Task and domain specific modelling in the Carnegie Mellon Communicator system," in *ICSLP2000*, Beijing, China., 2000, vol. II, pp. 130–133.
- [3] D. Bohus and A. Rudnicky, "RavenClaw: Dialog management using hierarchical task decomposition and an expectation agenda," submitted to *Eurospeech03*, 2003.
- [4] S. Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue, "Galaxy-ii: A reference architecture for conversational system development," in *ICSLP98*, Sydney, Australia., 1998.
- [5] X. Huang, F. Alleva, H.-W. Hon, K.-F. Hwang, M.-Y. Lee, and R. Rosenfeld, "The SPHINX-II speech recognition system: an overview," *Computer Speech and Language*, vol. 7(2), pp. 137–148, 1992.
- [6] A. Black, P. Taylor, and R. Caley, "The Festival speech synthesis system," <http://festvox.org/festival>, 1998.
- [7] W. Ward and S. Issar, "Recent improvements in the CMU spoken language understanding system.," in *Proceedings of the ARPA Human Language Technology Workshop*, March 1994, pp. 213–216.
- [8] P. Clarkson and R. Rosenfeld, "Statistical language modeling using the cmu," in *Eurospeech97*, Rhodes, Greece, 1997.
- [9] A. Black and K. Lenzo, "Limited domain synthesis," in *ICSLP2000*, Beijing, China., 2000, vol. II, pp. 411–414.
- [10] A. Black and K. Lenzo, "Building voices in the Festival speech synthesis system," <http://festvox.org/bsv/>, 2000.