BUILDING VOICEXML-BASED APPLICATIONS

Christina Bennett, Ariadna Font Llitjós, Stefanie Shriver, Alexander Rudnicky and Alan W Black

Language Technologies Institute
Carnegie Mellon University, Pittsburgh, PA 15213, USA
{cbennett,aria,sshriver,air,awb}@cs.cmu.edu

ABSTRACT

The Language Technologies Institute (LTI) at Carnegie Mellon University has, for the past several years, conducted a lab course in building spoken-language dialog systems. In the most recent versions of the course, we have used (commercial) web-based development environments to build systems. This paper describes our experiences and discusses the characteristics of applications that are developed within this framework.

1. INTRODUCTION

Spoken-language system development can be a time-consuming process. One reason is that developers, if starting from a blank slate, need to develop a large number of components, drawing together complex and fundamentally different technologies. Accordingly, a great deal of emphasis has been placed on the development of toolkits and environments that hide complexity and allow developers to rapidly prototype and deploy speech-based applications. Many such environments have been developed. The CSLU toolkit [1] is a notable example of a toolkit developed for teaching purposes. Several commercial systems, often derived from earlier IVR development environments, have also come into use [2] [3], as have web-based authoring tools using VoiceXML [4], [5], [6].

The essence of a useful development environment lies in its ability to hide some layers of a spoken language system and to present the user with an abstraction suitable for building an application. We can in general identify three layers of technology in such systems: 1) the base computation layer, consisting of engines that interface with the environment and provide core services such as recognition and synthesis; 2) the language engineering layer, consisting of acoustic, language and lexical models for the recognition engine, grammars for the understanding component, and generation and synthesis data for the output component; and 3) a dialog structure and application layer for defining the behavior of the system.

Most development environments focus on the dialog structure component of the process and offer users tools for constructing application *call flow*. Although in reality the implementation of a well-designed spoken language system will necessarily touch all layers, the dialog flow layer is what we most naturally understand as the core of a "dialog" system.

We have been teaching a speech system design course for a number of years, using layer 1 components based on Sphinx and Phoenix, and a variety of layer 2 tools (see, e.g., http://www.speech.cs.cmu.edu/tools).

We were nevertheless interested in finding a way to maximize the time students could spend on the interaction design aspects of system building. Accordingly we investigated the capabilities of VoiceXML, a dialog markup language that has found widespread acceptance in the field [7], [8].

Overall the course focuses on the process of developing a telephone-based spoken-language application, with one application being the focus each time the course is given; in most cases the instructors defined the application domain and provided an initial backend, though students are also encouraged to propose their own domains. The course covers: task analysis, dialog and interface design, representation of domain knowledge, integration with a backend, and usability testing of the resulting artifact. Weekly meetings feature occasional lectures as well as detailed discussion of ongoing work.

In this paper we describe two of the projects that were undertaken by students in the course and use these to assess the style of development that markup languages encourage as well as their pedagogical suitability. The first project is the Pittsburgh Busline, a telephone-based system that provides schedule information about buses traveling in and out of Pittsburgh's university neighborhood. The second project is the NBA Update Line, which provides callers with real-time information about NBA basketball games. The Busline systems were primarily developed using an early implementation of VoiceXML 1.0. The NBA Update Line was developed using VoiceXML 2.0. (Respecting their terms of use, we do not further identify the development environments). Note that the focus of the course was not to teach the use of VoiceXML per se but rather to use it as a tool for implementing dialog designs.

2. THE PITTSBURGH BUSLINE

2.1. Domain

Bus schedule information is a relatively simple domain, requiring (in the specification given to the class) only three pieces of information from the user: the departure location, the bus route or routes that the user is interested in, and the direction or destination of travel. Our Busline systems were designed to access a backend website that scrapes the transit schedules from the Allegheny County Port Authority site [9].

2.2. Two Busline applications

The two Pittsburgh Busline systems were developed concurrently and independently (see [9]). Despite domain simplicity, the independent development of the systems caused them to have several functional differences. The individual developers saw the domain from different viewpoints, and established different priorities for each system.

For example, each system uses a different order in which to solicit information from the user. System *A* asks first for location, then direction, and finally route, whereas system *B* asks for location, route, and then direction. While this may seem to be a superficial difference, but it stems from fundamental differences in design decisions. Figures 1 and 2 show the call flow for the two systems.

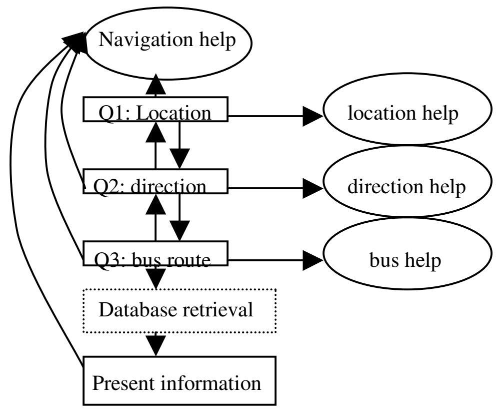

Figure 1: Call flow diagram for Busline *A*.

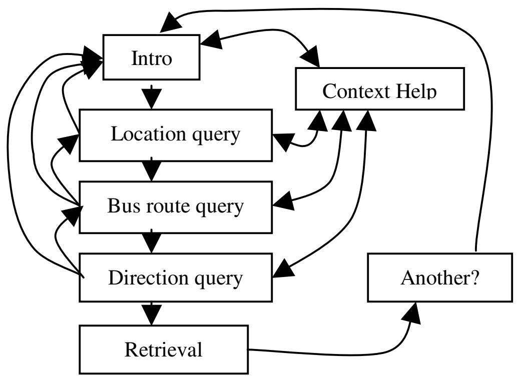

Figure 2: Basic call flow diagram for Busline *B*.

System *A* development focused on novices, aiming for a helpful, informative system. This system has extensive help information so that the user needs minimal domain knowledge to get information (see ⊗ in Fig. 3). This was partly in response to informal feedback from users during the early developmental stages. Specifically, once the system has the location and direction, it can provide the novice user with all the possible bus routes that travel in the direction specified. At the same time, it was also designed to take the least possible amount of time to complete the task, so it allows expert users to just say the information (even before prompting for it), and to quickly flow through the dialog (see * in Fig. 3). This proved particularly useful for this task, since most people calling the system had an immediate need for the information (for example, hoping to catch the very next bus). See Fig. 3 for a sample dialog.

System *B*, on the other hand, was developed with expert users in mind. Since the focus of this system was to provide information as quickly and easily as possible to someone familiar with the domain (i.e. a frequent bus rider), it was designed to eventually make use of a user profile database. The system expects the user to be familiar with the city bus system, and thus be able to provide the correct bus number(s) for the desired destination. As a result, system *B* has less help information but allows the user to specify multiple bus routes in a single query, instead of cycling through the dialog multiple times. The system also allows the user barge-in; an example is marked by * in Fig. 4, which shows a short basic dialog (i.e. without any requests for help, undo, etc.).

Welcome to Busline! (User barges-in) [*If you'd like some tips on how to use the system, you can say tips or press six at any time. I you need prompt specific help, just say more or press star.*]
 * I'm at CMU
Leaving from Carnegie Mellon.Now, if you are heading downtown, please say downtown, and (barge-in) [*if you are heading away from downtown, please say outbound*]
 I'd like to go to the airport
Heading downtown .What busses would you like to have the schedule for? (2 seconds pause). *If you need information about what buses to take, say more or press star.*
 * more
⊗ *If you want to know the schedule for the 61 buses, say 61 or press six then one. If you're on Forbes and Craig, you can take the 61s as well as the 500 to go downtown. You could also take the 71s, the 500, and the 59 U, if you go to Fifth and Morewood. If you want to go to the airport, say or press 28.*
 back
What busses would you like to have the schedule for?
 I'd like the schedule for the 28.
[database retrieval]
You can interrupt me at any time by saying repeat, to start telling you the times again, or by saying quit, to hang up.
The time is now 4:54 pm. In the next few hours, a 28X will reach Forbes and Morewood at the following times: 5:03pm 5:23pm 5:43pm (...)
 thanks
Do you want me to hang up?
 yep
Thanks for using Busline. Goodbye!

Figure 3: Sample dialog using Busline *A*.

Welcome to Busline! If you'd like to hear some pointers on using the system, you can say "pointers" at any time. What is your name?
 This is Alex
Hello Alex! What is your departure location?
 Squirrel Hill
Leaving from Squirrel Hill. What buses do you need (User barges-in) [*the schedule for? You can list as many as you'd like.*]
 * 61 and 501
Which direction are you headed, inbound or outbound?
 Inbound
Heading inbound. [database retrieval]
[The following are in the default TTS voice.]
The current time is 11:14 a.m. The next 501 will arrive at 11:20 a.m. The next 61 B will arrive at 11:23 a.m. The next 61 A will arrive at 11:32 a.m. The next 61 C will arrive at 11:36 a.m.
[Pre-recorded agent voice now returns.]
Would you like to continue?
 No thanks
Thanks for using Busline! Goodbye!

Figure 4: Sample Busline *B* dialog.

In both Busline applications, a dialog manager (VoiceXML code) is responsible for filling all three slots from the user, then building a query with those values, and finally initiating a backend retrieval [9] through a CGI invocation.

Both systems allow the user to correct an utterance that was misunderstood, start over from the beginning, request help, have the last question repeated, and make multiple queries in the same call. Both systems also used recorded prompts instead of the platform's built-in TTS system, except for dynamically generated query responses, which could not be prerecorded.

3. NBA UPDATE LINE

The target user of this system is a fairly knowledgeable basketball fan; the system must therefore be able to provide detailed information and as such deals with a greater range of information than the Busline projects. In essence, this application provides a telephone interface meant to replicate existing sports information web pages as closely as possible. Users can ask the system about scores around the league for current and past games, about statistics for players and teams in individual games, for lists of games happening on the current day, and for lists of players in each game (see [9]). The system was extended to include score updates for current NCAA basketball games. Fig. 5 shows a sample interaction.

Welcome to the NBA Game Update Line. What would you like to know?

Go to the Golden State game.

Golden State, 55. San Antonio, 71. 4:24 left in the 3rd quarter. The last play was Antawn Jamison made layup. What would you like to know about this game?

Who's in foul trouble?

Dampier and Arenas are in foul trouble for Golden State. No one is in foul trouble for the Warriors. What else would you like to know about this game?

Go to the Sonics game.

*Sorry, Seattle is not playing today. Would you like to hear about their last game?**

Yeah.

The last Seattle game was on March 7th. Seattle beat Charlotte at home, 92 to 89. What would you like to know about this game?

Who had the most points for the Hornets?

Mashburn led the Hornets in points, with 27.

Start over

Going back to the main NBA Update menu. What would you like to know?

Help

To hear all the current scores, say "all scores." To hear yesterday's scores, say "yesterday's scores." To hear information about a specific game happening today, say the name of an NBA team...

College scores

Here are all the current top-25 NCAA scores: Boston College 62, #10 Pittsburgh 76, Final score. Washington, 64, #13 Oregon 86, final score….

Figure 5: Sample NBA Update Line interaction.

The NBA Line uses VoiceXML to handle speech input and "special-event" dialog management, such as no-input events, misrecognitions, and help and quit requests. All other dialog management and backend information processing is handled via CGI scripts which hold state information and which scrape a variety of real-time-updated web pages to find the appropriate information.

The NBA information domain differs from the Busline domain in that users are likely to have multiple queries during a single call (since the information is constantly being updated while games are in progress, they might even ask the same question several times). To help reduce complexity, the application is divided into two main states, as shown in Fig. 6. From the general state, the user can find out about all the scores for the current or prior day. To access statistics about specific games, the user must request to change to a game. Once in a game state, the user can go back to the general state, or jump to a different game state.

In this domain, the greatest advantage of having separate states for each game was that it allowed for the dynamic creation of grammars for each game. We wanted to allow users to ask about statistics for specific players, but this created a serious ambiguity problem: "how many points did Williams have?" could refer to any of the 14 current NBA players named Williams. Creating grammars and a game state for each game reduced this problem (but did not completely eliminate it: Toronto still has two Williams' on their roster) and also addressed another issue inherent in the sports domain: players are frequently traded during the season, so implementing fixed team roster grammars is an ineffective approach.

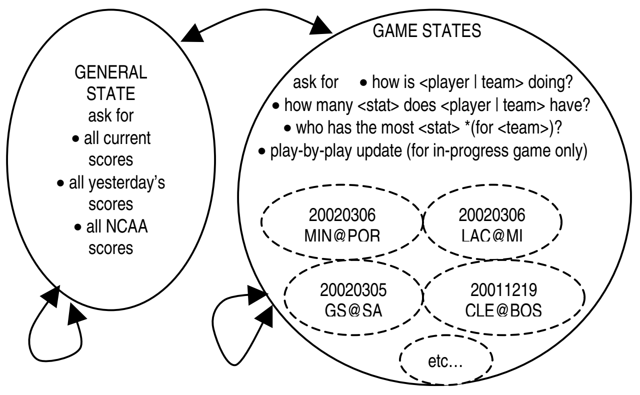
GENERAL STATE
ask for
• all current scores
• all yesterday's scores
• all NCAA scores
GAME STATES
ask for • how is <player | team> doing?
• how many <stat> does <player | team> have?
• who has the most <stat> *(for <team>)?
• play-by-play update (for in-progress game only)
20020306 MIN@POR
20020306 LAC@MI
20020305 GS@SA
20011219 CLE@BOS
etc...

Figure 6: NBA Update Line state diagram.

Beyond the state constraints however, the NBA Line can essentially be considered a user-initiated system (a system-initiated exception is shown at * in Fig. 5, where the user asks about a Sonics' game that is not in progress). Although the system always finishes its output with a prompt to the user, this prompt really only serves to remind the user to speak and to help the user remember which state they are in ("what would you like to know?" for the general-state; "what (else) would you like to know about this game?" for the game-state).

System-initiated dialog is also used to provide game updates that the user has not necessarily asked about. When the dialog is in the game state, the VoiceXML <timeout> length is reduced, and rather than triggering an error, the resulting <noinput> event sends a request to the backend to retrieve and report the last play of the game.

4. COMPARING THE SYSTEMS

Since the Busline domain supports a single task, it seemed natural to implement the systems using a system-initiative policy, prompting the user for each of the required slots. Even had it been possible to create a more open-ended interaction, it seems that for this particular type of task, having the computer drive the dialog worked quite well. It could also be argued that this is the most efficient way to fill the three slots to retrieve the information: as with most simple information retrieval tasks, a significant determiner of success is the user's ability to remember what the system needs to know. Transferring this responsibility to the system increases its usability.

The NBA Update Line, on the other hand, supports several different but related tasks. At any point in time the user needs to have the option of switching between different tasks, thus supporting mixed-initiative becomes more important.

Both systems rely on barge-in to give the user control over system output. While barge-in may be construed as a facet of mixed-initiative, it is probably more accurately thought of as a way for the developer to simplify the problem of selecting information for output. That is, the system can offer a superset of what might actually be needed and the user can interrupt output when sufficient information has been relayed.

5. DISCUSSION

Although the VoiceXML specifications [7] note that mixed-initiative dialogs are possible with VoiceXML, in our experience, implementation involved adding a number of dummy slots to the grammars and processing code to handle "missing" information. Additionally it seems that VoiceXML (in the implementations we made use of) does not support grammars that provide partial parses, a feature that is exploited in many state-of-the-art dialog systems [10], [11].

It is worth noting that the various web-based VoiceXML development environments can be quite different. Even when optimized for the same version of VoiceXML, the code still may not be directly portable due to proprietary additions or unsupported features. While we appreciate the reasons this might come to be, we nevertheless consider it unfortunate.

5.1. Suitability

Although VoiceXML applications can become rather complicated as they grow in size, it turns out to be quite easy to develop a simple interactive system with the desired dialog flow and basic functionality. Even novices can develop prototype applications in a few days. (None of the developers of the applications discussed in this paper had any prior experience with VoiceXML.)

VoiceXML applications have the advantage of allowing developers (in this case students) to abstract away from low-level implementation details and instead concentrate on the specific domain and on dialog design. This proved to be particularly useful for our purposes, since during a 15-week course there is simply not enough time to build a full dialog system from scratch (including configuring components for speech recognition, language modeling, grammars/parsing, dialog management, backend functionality, and text-to-speech) that exhibits a satisfactory level of functionality.

On the other hand, given enough time and energy, one can build almost arbitrarily complex applications. Though development time was limited to one semester, the NBA Update Line system provides access to a wide variety of information. The addition of NCAA scores to the system was a successful, late-semester experiment in seeing how well the NBA Line dialog structure could be adapted to new, yet similar domains (i.e. other sports/leagues). In fact this was quite simple, requiring only an adaptation to the grammar and a minimal change to the backend CGI script.

Furthermore, VoiceXML allows enough flexibility that even given the same assignment (e.g. create a bus schedule information system), students are likely to create very different applications based on their perceptions about the domain and the target user. The resulting systems provide the opportunity for comparative user studies and discussions on the relative effectiveness of different dialog strategies.

6. CONCLUSIONS / FUTURE WORK

VoiceXML enables effective exploration of dialog system design. Commercial VoiceXML development environments offer a relatively easy entry point that allows diverse dialog systems to be built. Dialog design problems such as providing consistent help, dealing with users with varied experience, etc., can be addressed easily without concerning oneself with lower-level mechanics, such as synthesis and recognition.

However, we are aware of the limitations of such frameworks, and we are investigating alternatives. For instance, a more comprehensive logging functionality would help us better analyze problems within dialogs: currently, when building and testing our grammars, we are unable to find out what a user actually said, necessary information for extending language coverage. Such limitations are not inherent in the VoiceXML framework per se and some environments may now provide such a feature. But from a pedagogical perspective, maintaining our own platform, such as one based on OpenVXI [12], would offer that desired control, though with an increase in infrastructure cost. Properly integrated with our existing tools, this would allow us to support teaching activities at different levels of spoken language system architecture, all in the context of a complete and open working system.

This research was sponsored in part by the SPAWAR, 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.

7. REFERENCES

[1] OGI Tookit: http://cslu.cse.ogi.edu/toolkit/
[2] Periphonics: http://nortelnetworks.com/products/04/oscar/
[3] Unisys: http://www.unisys.com/comm/
[4] BeVocal: http://cafe.bevocal.com
[5] Hey Anita: http://freespeech.heyanita.com
[6] Tellme: http://studio.tellme.com
[7] VoiceXML: http://www.w3.org/TR/VoiceXML20
[8] SALT: http://www.saltforum.org/
[9] http://www.speech.cs.cmu.edu/BusLine/icslp02
[10] Strik, H., Russel, A., van den Heuvel, H., Cucchiarini, C. and Boves, L. "A spoken dialogue system for public transport information," in H. Strik, N. Oostdijk, C. Cucchiarini, & P.A. Coppen (eds.) *Proc. of Dept of Language and Speech*, **19**: 129-142, Nijmegen, The Netherlands, 1996.
[11] Rudnicky, A. I., Thayer, E., Constantinides, P., Tchou, C., Shern, R., Lenzo, K., Xu, W., Oh, A., "Creating Natural Dialogs in the Carnegie Mellon University Communicator System", *Proc. of Eurospeech 1999*, **4**: 1531-1534, 1999.
[12] http://www.speech.cs.cmu.edu/openvxi/index.html