ROBUST DIALOG MANAGEMENT ARCHITECTURE USING VOICEXML FOR CAR TELEMATICS SYSTEMS

Yasunari Obuchi[1]*, Eric Nyberg, Teruko Mitamura, Michael Duggan*[2]*, and Scott Judy*
Language Technologies Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA
{obuchi, ehn, teruko, md5i, scottj}@cs.cmu.edu

Nobuo Hataoka
Central Research Laboratory, Hitachi Ltd., Kokubunji, Tokyo 185-8601, Japan
hataoka@crl.hitachi.co.jp

Abstract

This paper describes a dialog management architecture for car telematics systems. The system handles the user's spontaneous utterances and the variable communication conditions between the in-car client and the server. The communication is based on VoiceXML/HTTP, and the development of the server-side application is based on layered extensions of VoiceXML, called DialogXML and ScenarioXML. These extensions provide support for state-and-transition type programming, enable access to dynamic external databases, and share commonly used dialogs via templates. The client system includes a set of small grammars and lexicons for various tasks; only relevant grammars and lexicons are activated under the control of the dialog manager. The server-side applications are integrated via an abstract interface, and the client system may include compact versions of the same applications. The VoiceXML interpreter can switch between applications on both sides intelligently. This helps to reduce bandwidth utilization, and allows the system to continue even if the communication channel is lost.

1. Introduction

Spoken dialog management in car telematics is a challenging target for the speech and language research. The various challenges include efficient creation of dialogs, accurate analysis of the user's utterance, and managing the cooperation between the client and the server. Because of the strict limitation on computational resources and communication bandwidth, the client and the server systems need to divide their tasks in an appropriate manner. VoiceXML[1] provides a basis for design of the system architecture, in which the server system sends the minimal information necessary to guide the dialog, and the client system sends the minimal information necessary to describe the user's response.

In [2], Carpenter, et al. proposed a framework for server-side dialog management. Since VoiceXML does not support intrinsic functions for states and transitions, their framework assumes that the dialog manager[3] controls the entire flow of the dialog, and sends small segments of VoiceXML to the client. However, in mobile applications such as car telematics systems, the communication channel is narrow and unstable. Therefore, we prefer to send a VoiceXML document including several turns of the dialog at once. In previous work, we have proposed extensions to VoiceXML, called DialogXML and ScenarioXML[4]. DialogXML realizes higher level control of the dialog flow using states and transitions, and ScenarioXML realizes systematic ways to access external databases. We adopted those extensions in the dialog manager. The ScenarioXML written by the developer is compiled into VoiceXML documents corresponding to a large portion of the dialog. In the server system, as in [2] and [4], Java Server Pages (JSP)[5] are used by the dialog manager to create VoiceXML documents dynamically, so the application can easily be connected to external databases.

Another challenge of the in-vehicle dialog system is the analysis of the user's utterance. In a system that has rich computational and/or communication resources, such as a telephony gateway, a large-vocabulary continuous speech recognition (LVCSR) system (e.g., SPHINX[6]) and a large scale natural language processing (NLP) system (e.g., KANTOO[7]) can be integrated. However, in a client system with limited resources, analysis should be simplified. Our system uses a simple speech recognizer with a regular grammar, and a set of small grammars and lexicons play the role of the NLP system. A (grammar, lexicon) pair defines a task, and the dialog manager can activate one or more tasks by enumerating the names of those pairs. Such grammars and lexicons can be created by hand, or derived from corpora that include sentences from the specified task. In this scheme, the developer can make a robust dialog system easily.

The third challenge addressed by our system is task switching between the client and the server systems. Sometimes the in-vehicle client loses the connection to the server and keeps working as a stand-alone system. In such a case, the client system continues the dialog in a reduced way and provides limited information to the user. Then, after the connection is recovered, the client and the server negotiate to establish a new context, and start the next task.

2. System Architecture

Figure 1 shows the architecture of the car telematics system described in this paper. In the client system, VoiceXML Interpreter interacts with the user via automatic speech recognition (ASR) and text-to-speech (TTS) interfaces. We use the VoiceXML Interpreter developed by Hitachi CRL[8]; it

1 Currently at Central Research Laboratory, Hitachi, Ltd.
2 Currently at Software Engineering Institute, CMU

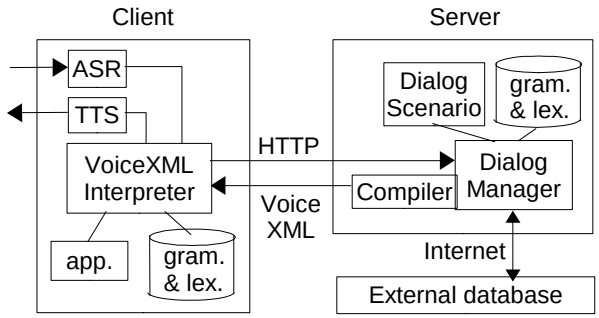

Fig. 1. *System architecture*

supports most of the functions defined in VoiceXML 2.0[1], and also includes an additional input/output channel to allow asynchronous communication with an internal application such as a GPS navigator. Therefore, although the system usually communicates by sending HTTP requests to the Dialog Manager (DM) and receiving VoiceXML documents in return, it can also work in standalone mode by utilizing internal applications. For example, an asynchronous signal from a GPS module can interrupt an ongoing dialog (e.g., a query about restaurant information) when the vehicle approaches an intersection where a turn is required. Grammars and lexicons are stored in the client and the server, and the DM specifies the location of the grammar and the lexicon to use, simply by including a specific URI in the VoiceXML document. If a server-side grammar and lexicon are needed, they are sent with the VoiceXML document. On the server side, the DM plays a central role and controls the dialog according to pre-defined dialog scenarios. In order to create a VoiceXML dialog including dynamic data (such as specific navigation directions), the DM communicates with external database through the Internet. The external databases provide various kinds of static and dynamic information, such as traffic and parking conditions, nearby restaurant names, and the weather forecast. The VoiceXML compiler is a part of the DM, and compiles the ScenarioXML into the VoiceXML format that is understandable by the VoiceXML Interpreter. Grammars and lexicons are stored in the server system, and the DM sends them with the VoiceXML output if the client side grammar and lexicon are not enough for the current dialog.

3. Extensions of VoiceXML

In [4], layered extensions of VoiceXML are proposed to realize easy development of VoiceXML applications with dynamic contents. First, DialogXML was proposed to make it possible for the developer to write any dialog flow using states and transitions. Although it is possible to write an equivalent form-filling and if-then style VoiceXML document by hand, state and transition style is much easier to understand for developers. After writing a DialogXML document, the DialogXML compiler translates it into VoiceXML, which is usually much longer and difficult to read than the original DialogXML. Second, ScenarioXML was proposed to handle dynamic contents collected from an external database. Since most of the information is dynamic in real life applications, it is necessary to

```
<javaloopstates namebase="s" array="Route" final="sx" index="i">
 <action><prompt>
  <javaval expr="(String)Route.get(i)"/>
 </prompt></action>
 <arc>
  <grammar src="next.gram" type="application/x-hgf" fieldlist="next"/>
  <gotoloopnext/>
 </arc>
</javaloopstate>
```

Fig. 2. *Example of ScenarioXML: loop and access to external database*

```
<jumplist>
 <arc name="help">
  <grammar src="help.gram" type="application/x-hgf" fieldlist="help"/>
  <destination dialog="help.xml"/>
 </arc>
</jumplist>
```

Fig. 3. *Example of ScenarioXML: common arc*

```
<state name="s1">
  <action><prompt>
    Go straight on Fifth Avenue.
  </prompt></action>
  <arc>
    <grammar src="next.gram" type="application/x-hgf" fieldlist="go"/>
    <dest state="s2"/>
  </arc>
  <arc>
    <grammar src="help.gram" type="application/x-hgf" fieldlist="help"/>
    <push dialog="help.xml"/>
  </arc>
</state>
```

Fig. 4. *Example of DialogXML*

generate the DialogXML output with up-to-date information included at run-time. In our system, JSP technologies are used for that purpose. The higher-level control of the JSP engine is useful when writing dialog scenarios, since dialog template can be combined with Java calls that insert dynamic content. A ScenarioXML compiler was developed to translate ScenarioXML documents into DialogXML.

Figure 2 shows an example of ScenarioXML; a loop of similar states is described in a higher level programming style, and a Java function with an incrementing argument is called to access to the external database. In this example, each function call gets the next instruction of the route guidance. After providing the instruction, execution moves to the next state and gets another instruction. Figure 3 shows another example of ScenarioXML. Since there are some typical patterns that could be used anywhere in the dialog, those patterns are described as "common arcs." In this example, insertion of the "help" dialog can be added anywhere easily using this ScenarioXML component as a common arc.

Figure 4 shows a segment of DialogXML generated from the examples of Fig. 2 and Fig. 3 by the ScenarioXML compiler. A state has an action and a set of arcs. In this example, the action and the first arc were generated from the main ScenarioXML from Fig. 2, and the second arc was added as a common arc from Fig. 3. Since route guidance tasks consist of several steps

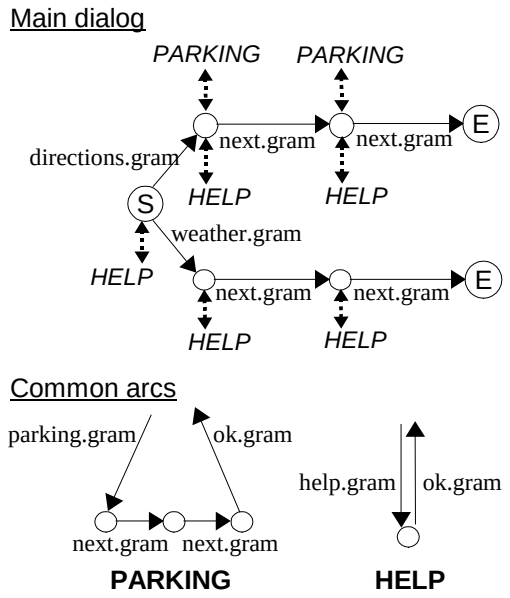

Fig. 5. *Transitions and grammars*

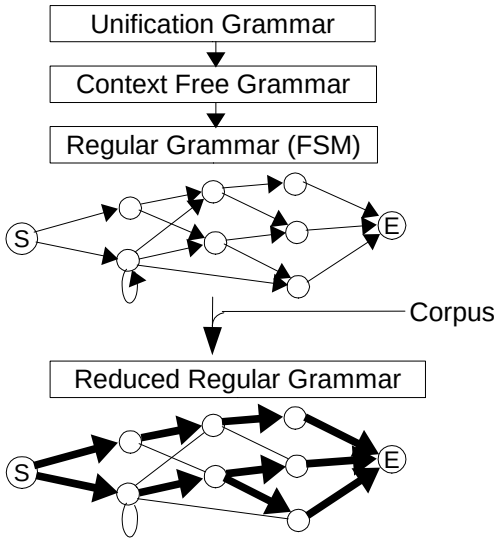

Fig. 6. *Grammar Compilation Using Corpus*

(directions from an origin to the destination), they will include several states that are similar to this example. Finally, the DialogXML document is compiled again to generate the VoiceXML that can be interpreted by the VoiceXML Interpreter.

A more complicated example is shown in Fig. 5. There are two flows of the main dialog and two types of common arcs. In this figure, every arc is related to a specific grammar. It means that the control of the dialog flow is tightly related with the grammar selection. This principle is described in detail in the next section.

4. Grammars and Lexicons

In VoiceXML, a grammar specifies the coverage of the user's utterance. It includes both the set of allowed words and the structure of the sentence in terms of parts-of-speech. To avoid confusion, we refer to the former as a "lexicon", the latter as a "grammar" and a pairing of a lexicon and a grammar as a "<grammar>." A <grammar> operates on an input to capture a set of attribute-value pairs from the user's utterance. One can claim that the widest coverage of the user's utterance could be achieved by using a LVCSR module and a statistical language model. In such a case, an NLP module must be used to extract information about a specific attribute-value pair. However, it is not reasonable to implement such modules in the client system because the computational resources are limited in the vehicle. Therefore, we use a simple speech recognizer with a regular grammar and a small lexicon, defined as a <grammar>, in the ASR part of the client system.

Another important role of a <grammar> is the control of the dialog flow. In Fig. 5, it is shown that each arc has a <grammar> specified by a filename with the ".gram" extension. Since the VoiceXML specification allows us to include multiple <grammar>s in a form, we can easily split the flow of the dialog by checking which <grammar> covers the user's utterance. If we have the table of <grammar> names and the table of attribute-value pairs allowed in each <grammar>, the developer can write ScenarioXML documents simply by referring to the appropriate <grammar>s.

In speech systems, it is also important to keep <grammar>s small because the larger the perplexity, the more recognition errors will occur. Therefore, building <grammar>s requires balance between two competing constraints: keeping the <grammar> small for recognition accuracy, and expanding to achieve sufficient coverage. A skilled programmer may be able to write such <grammar>s by hand, but it would also be useful to have a system that can create such <grammar>s automatically.

Figure 6 describes a procedure for automatic <grammar> creation using a corpus[9]. We also refer to it as "grammar compilation" because a basic grammar is written by hand, and then compiled into a form that is harder for humans to read but more suitable for the specific task. First we create a unification grammar (UG)[10], that is written in a human readable manner. The UG is then compiled into a context free grammar (CFG) by expanding all constraints. For a single UG rule, a set of CFG rules is created where each CFG rule corresponds to a single set of legal feature-value assignments on the UG rule right-hand side. Then the CFG is compiled to a regular grammar (RG) by introducing the upper limit of the number of recursions[11]. The derived RG can be expressed by a finite state machine (FSM) as in the figure. Then the FSM is used to parse the sentences in the corpus. After parsing all sentences, only the nodes and arcs that were activated by any sentence remain, and other nodes and arcs are deleted. By this procedure, we can have a reduced regular grammar that covers all sentences in the corpus and is smaller than the original grammar.

On the other hand, there is no automatic lexicon compilation scheme so far. If we use only the words from the original corpus that were recognized by arcs in the grammar, the generalization ability would be very weak. The sentence "How can I get to Tokyo?" would not be covered even if the corpus includes "How can I get to Kyoto?" However, if we generalize arcs to recognize

any words matching the appropriate part of speech, the generalization would be too strong, and poorer speech recognition performance would result. Currently we are using semantic word recognition categories (e.g., LOCATION) that are created partly by hand for each of our tasks. Automatic lexicon compilation using a corpus would be a future task of this project.

5. Switching External/Internal Applications

In car telematics systems, communication between the server and the client is usually unstable. The system must be robust against sudden channel disconnection. The VoiceXML specification includes an "error.badfetch" event that signals an error in fetching the requested document. Therefore, our VoiceXML documents have event handlers for "error.badfetch" that switches dialog control to local, compact application inside the client. For example, if the dialog is about traffic guidance, the internal application will know the route from the current position to the destination, but will not have access to any dynamic information such as the current traffic conditions. If the user asks about traffic conditions when the channel is lost, the system would reply "I'm sorry. Currently I can't access that information." Then the local dialog manager will switch into a wait state, and poll the server periodically in an attempt to reconnect, until it is forced to proceed to the next dialog by the user's command.

It is also possible to store the complete DM application in the client if the application does not require any dynamic information. For example, voice control of the vehicle air-conditioner can be done within the client. By using client-side applications for such small tasks, we can reduce bandwidth utilization between the client and the server.

As described in Section 2, client-side applications can use the "back door" of the VoiceXML Interpreter to communicate with it asynchronously. This mechanism would be convenient if we want the system to interrupt the dialog as soon as it detects the network re-connection. However, it is also possible to realize client-side applications simply by accessing static VoiceXML documents stored in the client.

6. Current Status

We have developed an initial system that is integrated with the ScenarioXML and DialogXML compilers. The control of the dialog flow using grammars and lexicons described in Section 4 was also implemented, although the automatic grammar compilation using a corpus is currently being tested in a separate prototype system. The task switching described in Section 5 is also being tested separately. The initial system will be modified to communicate with the internal GPS module, as described in Section 2, by the date of the workshop.

7. Conclusions

In this paper, we described a dialog management architecture for car telematics systems. The system consists of the client and the server, and designed to minimize the communication bandwidth utilization between them. In the server system, the Dialog Manager controls the dialog in accordance with the pre-defined scenarios written in ScenarioXML. The developer can write state-and-transition style dialogs using various templates, which support integration of dynamic information from external databases. The analysis of each user utterance within a dialog is achieved by a pre-selected grammar and lexicon, so that the developer has only to select appropriate sets of grammars and lexicons in each dialog. These grammars can be written by hand, but it is also possible to construct them automatically using sample dialogs for each task. Finally, we described how the system switches control between the server and the client according to the status of the communication channel. The system is robust in the presence of sudden network disconnections, and bandwidth utilization can be reduced by the use of client-side applications for small, static tasks.

Acknowledgements

The authors would like to thank Ichiro Akahori and Masahiko Tateishi of DENSO Research Laboratories for their support in collecting dialog corpora.

References

[1] W3C, "Voice Extensible Markup Language (VoiceXML) Version 2.0 Working Draft," http://www.w3c.org/TR/voicexml20/

[2] B. Carpenter, S. Caskey, K. Dayanidhi, C. Drouin, and R. Pieraccini, "A Portable, Server-Side Dialog Framework for VoiceXML," *Proc. of ICSLP 2002*

[3] R. Pieraccini, S. Caskey, K. Dayanidhi, B. Carpenter, and M. Phillips, "ETUDE: A Recursive Dialog Manager with Embedded User Interface Patterns," *Proc. of ASRU 2001*

[4] E. Nyberg, T. Mitamura, P. Placeway, and M. Duggan, "DialogXML: Extending VoiceXML for dynamic dialog management," *Proc. of HLT 2002*

[5] Sun Microsystems, "JavaServer Pages," http://java.sun.com/products/jsp/

[6] X. Huang, F. Alleva, H. Hon, M. Hwang, K. Lee, and R. Rosenfeld, "The SPHINX-II Speech Recognition System: An Overview," *Computer Speech and Language*," vol.2, pp.137-148, 1993

[7] E. Nyberg, and T. Mitamura, "The KANTOO Machine Translation Environment," *Proc. of AMTA 2000*

[8] T. Kujirai, H. Takahashi, A. Amano, and N. Hataoka, "Development of VoiceXML Interpreter and Continuous Words Recognition Engine - Development of Speech Recognition Technologies for Voice Portal," (in Japanese) *IPSJ SIGNotes*, SLP-33-12, 2000

[9] M. Tateishi, I. Akahori, S. Judy, Y. Obuchi, T. Mitamura, and E. Nyberg, "A Spoken Dialog Corpus for Car Telematics Services," *Proc. of Workshop on DSP in Vehicular and Mobile Systems*, 2003 (to appear)

[10] S. M. Shieber, H. Uszkoreit, J. Robinson, and M. Tyson, "The formalism and Implementation of PATR-II," SRI International, Menlo Park, California, 1983.

[11] A. Black, "Finite State Machines from Feature Grammars," *Proc. of Int. Workshop on Parsing Technologies*, 1989