Proceedings of the Human Language Technology Conference (HLT-2002), San Diego, CA, USA, March.

DialogXML: Extending VoiceXML for Dynamic Dialog Management

Eric Nyberg, Teruko Mitamura,
Paul Placeway and Michael Duggan
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{ehn,teruko,pwp,md5i}@cs.cmu.edu

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

Abstract

The characteristics of VoiceXML make it an attractive choice for the implementation of embedded dialog systems on networked hardware. Nevertheless, VoiceXML lacks some important features found in existing proprietary approaches. It is difficult to build a complex, multi-dialog system directly in VoiceXML. This paper describes DialogXML, an extension to VoiceXML that supports a more implicitly declarative language for dialog scenarios, and ScenarioXML, a straightforward combination of DialogXML with the template-filling mechanism of Java Server Pages. The paper describes an initial prototype system, which extends the OpenVXI VoiceXML interpreter for DialogXML, implements a web server for ScenarioXML, and utilizes the KANTOO natural language analyzer for understanding user inputs.

1. Introduction

Depending on the nature of the task, a dialog system might utilize different dialog representations of varying degrees of complexity. The simplest approach involves the use of a hierarchy of menus or state transition graphs, which restrict the user to a fixed set of pre-defined dialog turns. More sophisticated systems support mixed initiative, where the precise number and sequence of dialog turns are not predefined. Some tasks (e.g. navigation or route planning) require dynamic creation of dialog turns at run time, based on access to a remote database or information service [7, 2, 3, 11, 9].

The proposed W3C standard for VoiceXML [13] provides a representation for flexible form-filling in a dialog scenario. A single VoiceXML element can contain several form elements, which correspond to dialog states. A form can invoke multiple grammars in order to recognize the user's input. The system's response depends on which context variables have been set as a result of input matching. Through the use of conditional expressions and goto statements, the VoiceXML interpreter may transition to a different form in the current VoiceXML structure, or to a new VoiceXML structure. Transitions within the current structure are local to the VoiceXML interpreter; transitions from one VoiceXML structure to another are triggered by URLs which are retrieved via HTTP/CGI (see Figures 6 and 7).

The characteristics of VoiceXML make it an attractive choice for the implementation of embedded dialog systems on networked hardware. The VoiceXML interpreter may rely on remote server(s) to provide the sequence of dialog scenarios to be presented to the user. Nevertheless, VoiceXML lacks some important features found in existing proprietary approaches, such as the HDDL description language supported by Philips SpeechMania [2, 3]. While the form-filling action of the VoiceXML interpreter does not impose a particular dialog ordering upon the user, and could be thought of as a simple agenda or plan-based dialog manager, it is difficult to build a complex, multi-dialog system directly in VoiceXML. If an application requires several related dialogs, and flexible switching between dialogs and subdialogs, all possible transitions from a given state must be coded exhaustively by hand. This is less preferable than the more declarative approach taken in HDDL, where dialogs are defined as separate scenarios and the system automatically handles transitions between dialogs.

Our goal is to extend VoiceXML to make it more practical for large-scale dialog systems. We envision a mobile application environment (e.g. a mobile information system) where an embedded speech recognizer and VoiceXML interpreter are connected to remote servers that support a variety of information-seeking tasks (car navigation, restaurant information, voice-activated control, etc.). While we prefer the simplicity of VoiceXML for run-time processing, a more supportive representation is required for creation of the dialog scenarios. It should be possible for different developers to write dialog scenarios independently – a given developer should not need to know the names of the other dialogs, their states, etc. in advance in order to write his or her own dialogs. It should also be possible to integrate the VoiceXML interpreter with more sophisticated natural language analyzers, to take advantage of the broader coverage and deeper linguistic analysis provided by unification grammar-based systems [6].

This paper describes DialogXML, an extension to VoiceXML that supports a more implicitly declarative language for dialog scenarios. We also introduce ScenarioXML, a straightforward combination of DialogXML with the template-filling mechanism of Java Server Pages (JSP) [12]. ScenarioXML provides an easy way to dynamically generate content (e.g. navigation directions accessed from a remote database). Dialog scenarios in ScenarioXML are also JSP pages. Dynamic content creation and compilation of DialogXML to VoiceXML are handled automatically at run time through the invocation of JSP filters by the web server. We also describe the integration of the OpenVXI VoiceXML interpreter with the unification grammar and pattern matching components of the KANTOO natural language analyzer [6] (see Figure 1).

We have constructed an initial prototype system that handles text input. The next phase of our research will address the integration of the dialog management system with speech recognition and understanding. Therefore this paper does not address an important set of issues related to robustness at the speech recognition / NLP interface. The proposed architecture is independent of and neutral with respect to speech recognition; any approach which can be integrated via the `grammar` mechanism in VoiceXML can be utilized.

In Section 2, we introduce DialogXML and ScenarioXML. In Section 3, we present the details of the architecture and the current implementation. In Section 4, we summarize the current status of the system and discuss our planned research activities for the future.

2. Dialog XML and Scenario XML

The use of VoiceXML with a suitable interpreter has generated substantial interest as an effective way to develop simple command and control systems [13]. However, in order to use an existing VoiceXML interpreter in more complex environments with dynamic context and context switching, some enhancements to the VoiceXML representation and interpretation process are desirable.

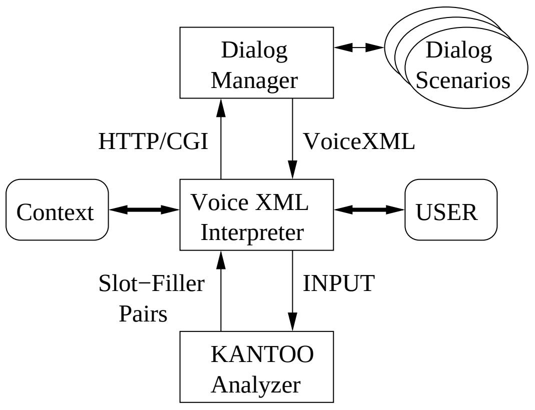

Figure 1: **Modular Architecture**

Most VoiceXML dialog systems use simple context-free language models for user utterances. In order to support full natural language parsing (e.g., with a unification grammar), the VoiceXML interpreter must be extended to handle `grammar` elements that contain references to more sophisticated grammar modules.

Natural dialog has a notion of state, and dialog designers often think in terms of state-transition networks. VoiceXML supports simple form and action pairs, but does not explicitly model states and transitions among states. Although it is possible to author and maintain a set of large-scale dialogs directly using VoiceXML, it would be tedious and time-consuming to do so.

2.1. Dialog XML

To address these issues, we have formulated a new level of representation, called Dialog XML (DXML), which clearly expresses the notion of states and transitions. Furthermore, each element in Dialog XML has an unambiguous mapping to a corresponding set of VoiceXML elements, so that Dialog XML can be automatically compiled to VoiceXML. An example Dialog XML structure is shown in Figure 5. The corresponding (compiled) VoiceXML structure is shown in Figures 6 and 7.

In Dialog XML, transitions between states in a dialog are clearly represented by `arc` elements which contain `grammar` and destination (`dest`) elements. This makes the structure of the resulting dialog much easier to follow visually. Note that the corresponding VoiceXML structure does not directly link or encapsulate the grammar and destination together. In VoiceXML 1.0, the `form` element allows its `grammar` and `filled` tags to appear anywhere. By enforcing a more dialog-centric structure, Dialog XML provides a more convenient notation for the dialog designer.

Transitions in Dialog XML include `dest` and `push`. The `dest` element is directly mapped to the `filled` element in Voice XML. However, the `push` element cannot be directly mapped to a corresponding VoiceXML element. Instead, each `push` is mapped into a specially constructed `form` element which invokes a subdialog and returns to the current state. For example, the single, concise `push` in the second arc in state `s1` in Figure 5 corresponds to the (much more complicated) third `form` element in Figure 6.

2.2. Scenario XML

An independent consideration is how the system will handle dynamic content and linking between dialogs. In a system with a wide variety of possible interactions or information exchanges, it becomes necessary to work with templates rather than static, hard-wired states and transitions. Since our goal is to use VoiceXML interpretation at run-time, some higher-level representation must be provided to represent dynamic content in a more abstract manner.

To address this need, Dialog XML is combined with a template-filling system (Java Server Pages [12]), deriving a higher-level representation we refer to as Scenario XML. Java Server Pages are used to express dynamic content (see Figure 4). In the example, the arcs which follow the `prompt` element are filled in by invoking a filter (`common-arcs.jspf`) that will derive the possible set of transitions by examining the available set of dialogs for triggering grammars. The four arcs generated by this call in the current system follow the `prompt` in Figure 5. The process of gathering the triggering conditions for a set of dialogs into a set of arcs is visually represented n Figure 2.

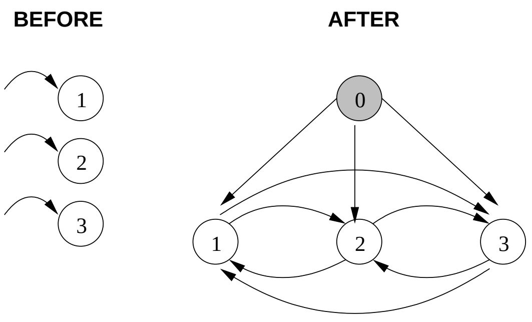

Figure 2: **Arc Compilation**

The Scenario XML example illustrates two important characteristics of our approach: a) when dialogs share a considerable amount of content and/or structure, this can be written once in a shared template; b) dialogs can be constructed independently of one another, since the system will insert global transitions to other dialogs automatically. This is important because requiring the human developer to maintain a correct and consistent treatment of global transitions across a large set of dialogs with many states is cumbersome and error-prone.

3. Architecture and Data Flow

The architecture shown in Figure 1 illustrates how the Dialog Manager component interacts with the VoiceXML Interpreter and KANTOO Analyzer. The system is started by calling the VoiceXML Interpreter on the JSP page that represents the top-level dialog scenario (`main.jsp`, see Figure 4). The JSP URL is fetched from the Dialog Manager (web browser), which invokes the appropriate JSP calls to fill in the dynamic information. The Dialog Manager also invokes a JSP filter which converts the higher-order DialogXML constructs into their low-level VoiceXML forms[1]. The resulting VoiceXML page is passed back to the VoiceXML interpreter, which the prints the prompt and initiates the dialog with the user.

Once the user has entered some input, the system will test all of the `grammar` elements in the current `form`; these correspond to the arcs in a particular dialog state in the original DialogXML. In order to integrate the use of full NL analysis, we extended the `grammar` interpretation mechanism in VoiceXML by defining a new grammar type, `kantoo/patrick`. This involves creation of a new type-to-class mapping, and coding of a new class that implements the abstract `grammar` interface (these are straightforward extensions, which are directly supported by the VoiceXML implementation).

The KANTOO Analyzer processes the user input to produce an interlingua expression (semantic frame). The individual `grammar` elements in VoiceXML refer to pattern-matching rules which are unified with the interlingua, possibly setting slot-filler values in a context variable as a side-effect (see Figure 3). The main advantage of this approach is that multiple variations of the same utterance can be recognized by a single semantic pattern rule, a useful improvement over simpler context-free or regular-expression grammars.

In the standard VoiceXML interpreter, subdialogs can only influence the outcome of a parent dialog through their return value. Since the parsing of a complex NL input might provide several important pieces of information about the context (current location, destination, etc.), it was necessary to extend the context-handling capabilities of VoiceXML. The new grammar type can return a set of slot-filler context variables to the VoiceXML interpreter, which merges this information with the existing dialog context.

Another feature of standard VoiceXML is that the context of a particular dialog does not persist when a new dialog is invoked. Since we require the ability to maintain context across two sequential dialogs (e.g., if the user asks about parking at a location following a navigation scenario), it was necessary to modify the VoiceXML interpreter to maintain context across dialogs unless an explicit reset is encountered.

```
;; This PATRICK rule file handles the
;; top-level switch to the directions module.
;; It accepts an optional location and
;; destination.
(node DIRECTIONS :rules
  ((*TRY*
    ((*OR* ((%(ir patient CONCEPT)
              =c *O-DIRECTION)
            (%dir <= %(ir patient)))
           ((%(ir object CONCEPT)
              =c *O-DIRECTION)
            (%dir <= %(ir object)))
           )
     ;; This is our convention for
     ;; reporting which pattern matched
     (%(x "directions_request") = "accept")
     (*TRY* ((%pp <= %(dir q-modifier))
             (%(pp CONCEPT)
               =c "*Q-SOURCE_FROM")
             (%(x location)
               = %(pp object CONCEPT))))
     (*TRY* ((%pp <= %(dir q-modifier))
             (%(pp CONCEPT) =c "*Q-GOAL_TO")
             (%(x destination)
               = %(pp object CONCEPT)))))))
  )
```

Figure 3: **Sample Context Rule**

While implementing these features did involve modifications to the VoiceXML Interpreter itself (and should be considered a proposed extension to the 1.0 standard), the modifications were minor, involving less than 20 lines of source code. As a result, the VoiceXML interpreter becomes a "stateful" rather than "stateless" client in the architecture.

4. Current Status and Future Work

The first prototype of this system has been implemented using the Speechworks VXML Interpreter [10], Tomcat web server [5] and KANTOO Analyzer [6]. The Dialog Manager (Tomcat) and KANTOO run on a single server machine, which is accessed by the VXML Interpreter via HTTP/CGI (Tomcat) and TCP/IP (KANTOO). A set of simple navigational dialogs have been implemented in English, which feature information about directions and parking that are dynamically

[1]A full description of the DXML to VoiceXML compiler is beyond the scope of this document.

accessed from a separate database. The compilation of ScenarioXML and DialogXML to VoiceXML is carried out in real time, and requires at most only a few seconds per invocation.

Our next steps include: a) extending the system to Japanese; b) adding other information-seeking tasks (e.g. restaurant information) and control tasks (car systems interface); c) integration with real-time speech recognition. We believe that DialogXML and the proposed extensions to VoiceXML will provide a useful open-source system that is much more suitable for dialog management in large-scale practical applications than VoiceXML alone.

```
<DialogXML>
<dialog name="main">
  <state name="s0">
    <arc>
      <dest state="s1"/>
    </arc>
  </state>
  <state name="s1">
    <action><prompt>
      How can I help you?
    </prompt></action>
<%@ include
  file="dxml-templates/common-arcs.jspf" %>
    <arc>
      <dest state="s0"/>
    </arc>
  </state>
</dialog>
</DialogXML>
```

Figure 4: **Sample ScenarioXML**

```
<DialogXML>
<dialog name="main">
  <state name="s0">
    <arc>
      <dest state="s1"/>
    </arc>
  </state>
  <state name="s1">
    <action><prompt>
      How can I help you?
    </prompt></action>
    <arc>
      <grammar src="directions_help.pat"
               type="kantoo/patrick"/>
      <push dialog="directions-help.jsp" />
    </arc>
    <arc>
      <grammar src="directions_request.pat"
               type="kantoo/patrick"/>
      <push dialog="directions-request.jsp"/>
    </arc>
    <arc>
      <grammar src="parking_request.pat"
               type="kantoo/patrick"/>
      <push dialog="parking-request.jsp"/>
    </arc>
    <arc>
     <grammar src="top_level_no_context.pat"
              type="kantoo/patrick"/>
     <dest dialog="main.jsp"/>
    </arc>
    <arc>
      <dest state="s0"/>
    </arc>
  </state>
</dialog>
</DialogXML>
```

Figure 5: **Sample DialogXML**

```
<vxml version="1.0">
  <form id="main.s0">
      <block>
        <if cond="delete DMEnv.topic"/>
        <goto next="#main.s1"/>
      </block>
  </form>
  <form id="main.s1">
    <grammar src="top_level_no_context.pat"
             type="kantoo/patrick"/>
    <grammar src="park_req.pat"
             type="kantoo/patrick"/>
    <grammar src="dir_req.pat"
             type="kantoo/patrick"/>
    <grammar src="dir_help.pat"
             type="kantoo/patrick"/>
    <block>
      <prompt>How can I help you?</prompt>
    </block>
    <block cond="DMEnv.top_level_no_context
                 != undefined">
      <goto next="main.jsp"/>
    </block>
    <field expr="DMEnv.top_level_no_context"
           name="top_level_no_context"/>
    <block cond="DMEnv.park_req
                 != undefined">
      <if cond="delete DMEnv.park_req"/>
      <goto next="#main.s1.arc2"/>
    </block>
    <field expr="DMEnv.park_req"
           name="park_req"/>
    <block cond="DMEnv.dir_req
                 != undefined">
      <if cond="delete DMEnv.dir_req"/>
      <goto next="#main.s1.arc1"/>
    </block>
    <field expr="DMEnv.dir_req"
           name="dir_req"/>
    <block cond="DMEnv.dir_help
                 != undefined">
      <if cond="delete DMEnv.dir_help"/>
      <goto next="#main.s1.arc0"/>
    </block>
    <field expr="DMEnv.dir_help"
           name="dir_help"/>
    <filled namelist="dir_help">
      <if cond="delete DMEnv.dir_help"/>
      <goto next="#main.s1.arc0"/>
    </filled>
    <filled namelist="dir_req">
      <if cond="delete DMEnv.dir_req"/>
      <goto next="#main.s1.arc1"/>
    </filled>
    <filled namelist="park_req">
      <if cond="delete DMEnv.park_req"/>
      <goto next="#main.s1.arc2"/>
    </filled>
    <filled namelist="top_level_no_context">
      <goto next="main.jsp"/>
    </filled>
    <block>
      <goto next="#main.s0"/>
    </block>
  </form>
```

Figure 6: **Sample VoiceXML (Part I)**

```
  <form id="main.s1.arc0">
      <subdialog name="retval"
                 src="directions-help.jsp">
          <filled>
              <if cond="retval.retval">
                  <goto next="#main.s1"/>
                  <else/>
                  <goto next="#main.s1"/>
              </if>
          </filled>
      </subdialog>
  </form>
  <form id="main.s1.arc1">
      <subdialog name="retval"
                 src="directions-request.jsp">
          <filled>
              <if cond="retval.retval">
                  <goto next="#main.s1"/>
                  <else/>
                  <goto next="#main.s1"/>
              </if>
          </filled>
      </subdialog>
  </form>
  <form id="main.s1.arc2">
      <subdialog name="retval"
                 src="parking-request.jsp">
          <filled>
              <if cond="retval.retval">
                  <goto next="#main.s1"/>
                  <else/>
                  <goto next="#main.s1"/>
              </if>
          </filled>
      </subdialog>
  </form>
</vxml>
```

Figure 7: **Sample VoiceXML (Part II)**

Acknowledgements

The authors thank Denso Corporation and Hitachi Ltd. for their support of this research.

References

[1] Araki, M., K. Ueda, T. Nishimoto and Y. Y. Niimi (2001). "Dialogue scenario generation from XML-based database", Proceedings of the 1st NLP and XML Workshop, held at the Sixth Natural Language Processing Pacific Rim Symposium, Nov. 27-30, Tokyo, Japan, `http://www.afnlp.org/nlprs2001`

[2] Aust, Harald and Olaf Schröer (1998). "An Overview of the Philips Dialog System", Proceedings of the DARPA Broadcast News Transcription and Understanding Workshop February 8-11, 1998, Lansdowne, Virginia.

[3] Baum, M., G. Erbach, M. Kommenda, G. Niklfeld and E. Puig-Waldmüller (2001). "Speech and Multimodal Dialogue Systems for Telephony Applications Based on a Speech Database of Austrian German", ÖGAI Journal 20/1, pp.29-34, 2001. `http://www.ftw.at`

[4] Belvin, R., *et al.*, "Development of the HRL Route Navigation Dialogue System", *Proc. HLT 2001*, pp. 74–78.

[5] Jakarta Tomcat, `http://jakarta.apache.org/tomcat`.

[6] Nyberg, E. and T. Mitamura (2000). "The KANTOO Machine Translation Environment", *Proceedings of AMTA 2000.*

[7] Price, P. (1990). "Evaluation of Spoken Language Systems : The ATIS domain", *Proc. 3rd DARPA Workshop on Speech and Natural Language.*

[8] Ramakrishnan, N., *et al.*, "Mixed-Initiative Interaction = Mixed Computation", *Proc. ACM SIGPLAN Workshop PEPM'02*, January 2002

[9] Seneff, S., *et al.*, "Organization, Communication, and Control in the Galaxy-II Conversational System", *Proc. Eurospeech 99.*

[10] SpeechWorks OpenVXI 2.0.1, `http://www.speech.cs.cmu.edu/openvxi`

[11] David Stallard, "Talk-n-Travel: A Conversational System for Air Travel Planning," in Proceedings of the Association for Computational Linguistics 6th Applied Natural Language Processing Conference (ANLP 2000), Seattle, Washington, April 29 - May 4, 2000, pp. 68-75.

[12] Sun Microsystems (2002). "Java Server Pages", `http://java.sun.com/products/jsp`.

[13] VXML 1.0, 03/07/00, `http://www.w3.org`