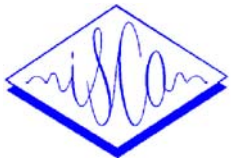

INTERSPEECH 2004 - ICSLP
8th International Conference on Spoken Language Processing
ICC Jeju, Jeju Island, Korea
October 4-8, 2004

Florence: a Dialogue Manager Framework for Spoken Dialogue Systems

Giuseppe Di Fabbrizio *Charles Lewis*

AT&T Labs – Research
180 Park Ave – Florham Park, NJ 07932 - USA
{pino,clewis@research.att.com}

Abstract

Recent advances in speech and language technology have made spoken dialogue systems mainstream in many industries. They allow customers to engage in natural speech interactions with machines instead of being compelled to navigate menus of options with touch tones inputs. VoiceXML was a major milestone for the process of using automated speech applications to expose business portals to ubiquitous telephone access. By the introduction of a uniform and universally accepted client-server browser model, the VoiceXML programming model greatly simplified previously dominant proprietary computer telephony interfaces. However, natural language spoken dialogue systems entail more complex interactions with the user which, depending upon the application domain, may require computational models that are difficult to express directly in VoiceXML. This paper describes Florence, a dialogue manager with a more general approach that uses an extensible and flexible framework to combine interchangeable and interoperable dialogue strategies as appropriate to the task. Florence's declarative XML-based language facilitates the development of natural language applications and allows the dialogue author to encapsulate and reuse different algorithms between applications. Moreover, it addresses large-scale natural language issues related to enterprise backend access, logging, distributed deployment, and fail-over support. These issues must be addressed in a modern, industrial-strength application server environment.

1. Introduction

VoiceXML[1] was a major milestone for the process of using automated speech applications to expose business portals to ubiquitous telephone access. In a few years of existence, it has promoted a standard development environment, made speech applications more portable and reusable across different vendors, and created a new breed of voice applications solutions allowing market players to adopt flexible business models ranging from fully packaged solutions to general hosting models. It opened the proprietary Interactive Voice Response (IVR) market, a market previously dominated by large vendors and closed system solutions. On the other hand, VoiceXML only supports form-based and menu-driven dialogues with a limited level of initiative for the user. Similarly, the Speech Application Language Tags (SALT) [2] Forum proposal and the XHTML+Voice Profile [3] W3C recommendation follow the same VoiceXML dialogue management paradigm, although they introduce the concept of multimodal input/output access in addition to traditional telephone audio,.

Natural language services, however, tend to be designed as mixed-initiative systems, where the initiative for driving a dialogue forward can be taken by the user, the system, or a combination of the two. Furthermore, complex task-oriented systems [4][5][6] where speech acts analysis [7] is used to support computational theories of communication in collaborative problem solving scenarios, require deeper dialogue models.

This paper describes Florence, a dialogue manager (DM) that uses an extensible and flexible framework to combine interchangeable and interoperable dialogue strategies as appropriate to the task. The goal of the Florence project was to create a best-of-breed toolkit capable of implementing the rich variety of voice applications that are developed at AT&T. Where previous toolkits have focused on call routing [14] or plan-based interactions, Florence was expected to handle a variety of possible dialogue strategies.

This paper will describe the design and architecture of the Florence toolkit for dialogue development and runtime engine. Florence is distinguished from other dialogue management tools by its capacity to easily integrate new dialogue strategies and algorithms, the simplicity of the implementation language, and the support it provides for a number of commonly used dialogue patterns.

The support provided for the development process itself will also be described, including provisions that have been made for debugging and testing during this process. Deployment of the DM into full voice application architecture will also be outlined. Finally, we will take a look at how Florence may be adapted for new dialogue strategies, and even new roles in agent-based systems. The goal of the Florence kit has been to promote the implementation of new ideas, while supporting existing ones as simply as possible.

2. System Architecture

Figure 1 shows a high level logical architecture of the AT&T Spoken Dialogue Systems. It adopts the standard 3-tier web architecture where there is a clear separation between the client (i.e., VoiceXML browser), integrated in the telephony server, the application server and the enterprise information system or database backend. Once a phone call is established through the telephony server, either via traditional Public Switched Telephony Network (PSTN) or Voice over IP (VoIP), an initial fetch request is sent to the application server. The controller servlet responds to the request based on a predefined routing configuration and dispatches it to the pre-allocated dialogue manager resources.

The DM generates the initial VoiceXML page that prompts the caller either with a pre-recorded or synthesized greeting message. At the same time, it activates the top level Automatic Speech Recognizer (ASR) grammar. The caller's speech is then translated into text and sent back to the controller and then the DM, which is responsible for querying

the Spoken Language Understanding (SLU) module for a semantic representation of the utterance. Based on the SLU reply and the implemented dialogue strategy, the DM engages in a mixed initiative dialogue to drive the user through a specified call flow.

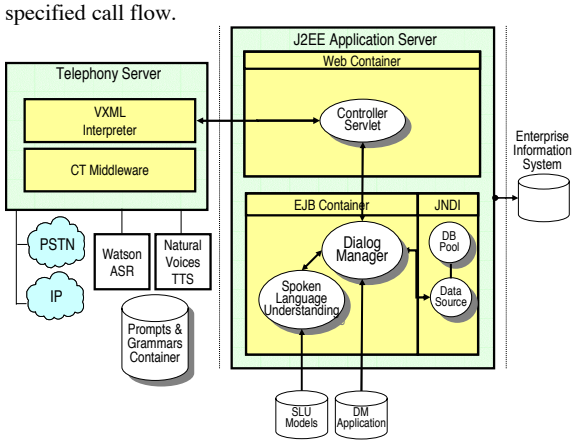

Figure 1. AT&T Spoken Dialogue System Architecture

The DM iterates the previously described steps until the call reaches a final state (e.g., in the case of a routing application, the call is transferred to a live agent, an IVR or the caller hangs up). The DM is also responsible for accessing the enterprise backend with either an HTTP interface or JDBC technology. To keep the dialogue state information during the duration of the HTTP session, the architecture components are implemented as statefull session Enterprise Java Beans (EJBs). This solution delegates the state management to the Java enterprise application server, which avoids complex session information bookkeeping traditionally done by exchanging session cookies. The beans communicate with each other with standard HTTP protocols and XML data structures. The EJB infrastructure also assures scalability, reliability and fail-over support in case one on the application servers goes down. Each component automatically provides multiple levels of logging as well as optional logging destinations (i.e., flat file, database, Java messages, etc.) that can be changed at runtime without modifying the application binary.

2.1. DM Input/Output

Communication with the DM is accomplished through XML documents. XML encompasses a number of existing standards for both the parsing of the user input, and the formatting of the output for the voice platform. The ASR output string is the input to the SLU module that classifies the user's utterance and translates into an XML semantic representation. Either the Natural Language Semantic Markup Language [11] (NLSML) or the Extensible MultiModal Annotation (EMMA) [12] are suitable formats. These markup languages are intended for use by systems that provide semantic interpretations for a variety of inputs, including, but not necessarily limited to, speech and natural language text input. Our implementation is based on a simplified version of NLSML. To achieve the classification task, the SLU uses an extended version of a boosting-style classification algorithm [8] trained on an application dependent corpus. The generated NLSML document is sent to the DM module which updates the local context and the dialogue history. Figure 2 shows an example of SLU classification result. The selection of an XML format for input means that control over exactly how meaning is extracted from the input can be left to the application. An XPath parser is the tool of choice for this [10].

As long as the information the DM needs to make decisions about the dialogue is contained in the input in a consistent pattern, XPath can be used to systematically extract it into variables for decision making. Logical predicates, described by XPath expressions, allow the DM to rank classes and assign a contextual interpretation to the input based on the current context content. Figure 3 illustrates how Florence can extracts values such as the 1st best call-type and the confidence score using XPath expressions. The local context stores the state variables and keeps track of the history for dialogue backtracking and repairs.

```
<result>
 <interpretation>
  <instance>I have a question about my bill and my phone number is
<phone-number-us><area-code> nine oh eight </area-code><local> two
seven three one two one seven </local></phone-number-us></instance>
  <classes>
   <class name="Billing_Services" score="0.980712" offset="1"/>
   <class name="Home_Number" score="0.964713" offset="2"/>
  </classes>
 </interpretation>
</result>
```

Figure 2. SLU Output Example

```
<set var="callType" select="//classes/class[position()=1]/@name" />
<set var="confidence" select="//classes/class[position()=1]/@score"/>
```

Figure 3. XPath Expressions Example

On the other end of the process, templates are used to provide consistently formatted XML output to the target platform. The process that fills in the templates permits variables from the decision making process, as well as information about the deployment environment, to be incorporated into the output.

Section 3.4 discusses how this approach can be used for any XML-formatted standard, for both the expected input and generated output. Until then, we will discuss the system in terms of input in the NLSML format and output to VXML.

3. Florence Architecture

This section will describe what happens in the DM at runtime from the time it receives the input to when it generates new output (Figure 4). The Florence runtime engine maintains state, so it's important that the same engine instance is used for each step in the dialogue through the whole session.

The first key concept is the *dialogue stack*. This is a data structure that tracks the order in which dialogs have been invoked, and allows control to be returned to a calling dialogue when a subdialog has been completed.

Objects called *flow controllers (FC)* track information about a specific dialogue on the stack. This information includes local variables and the point in a dialogue where a subdialog is invoked if this occurs.

Local and *global context* are used to maintain variable information. Each flow controller has a local context, and the application as a whole has a global context. Variable information can be passed between local and global contexts, and between the local contexts of a dialogue and its subdialog.

Actions are how the system communicates with the caller. Each action includes VoiceXML snippets and, optionally, a template.

At runtime, these primary components operate together to

process the user input to create the formatted output based on the application. The application defines a series of flow controllers that control this process.

Multiple instances of a dialogue (as represented by an FC) may exist simultaneously on the stack. When dialogue creates a subdialog, control of the dialogue is surrendered to it. The application configuration file indicates the top level dialogue that will be used. A flow controller for this dialogue is created and put on the stack before any input is received.

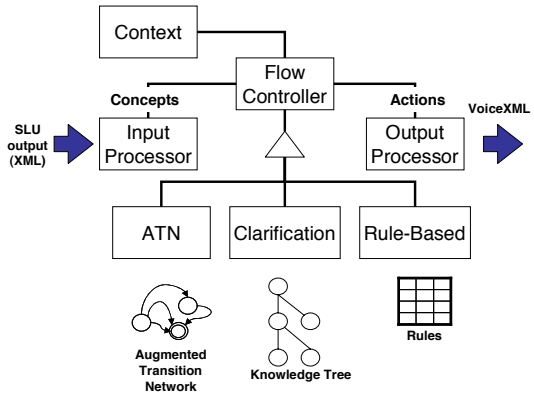

Figure 4. Florence Architecture

When Florence receives new input, it is put into the local context of each flow controller on the stack. The flow controller with control of the dialogue will then be allowed to add actions to the output and change variables in the context. It may also create a new subdialog or return control to a previous dialog. When the top-level dialogue returns control, the application is complete. A flow controller may also declare the turn to be complete. When this happens, the actions that have been fired are incorporated into a template, which Florence returns as output. The dialogue is then paused until new user input is received.

This is a general outline; the specifics of what occurs inside of a flow controller have intentionally been left vague. In a later section we will discuss how this allows various dialogue strategies to be incorporated. The most commonly implemented dialogue strategy algorithm uses Augmented Transition Networks [13] (ATNs) to represent the dialogue flow. The ATN operates on the current input and the local context to control the interaction flow. An ATN structure consists of states, transitions, conditions, and a set of variables or registers. States can execute operations on context variables and optionally invoke recursively other networks or subdialogs, return control to a previous dialogue, or declare the turn to be complete. A single transition from the state is chosen when the state has completed execution, based on the conditions in the outgoing transitions. The actions named in the transition are fired, and control passes to the next state. This basic strategy is sufficient for call routing and other simple dialogs. Another strategy that is available in Florence is the Clarification FC. This strategy uses a hierarchical category tree to help the user to specify a topic. This tree includes conditions that describe the categories and topics, and prompts for eliciting more specific information. It permits the user to take the initiative by volunteering information beyond what the system requests, and is able to resolve inconsistencies and vagueness in this input.

The isolation of dialogue logic written by application authors into modular subdialogs permits the re-use of simple dialogues for repeated tasks, such as disambiguation and confirmation strategies. This isolation also permits the creation of standard dialogs for more complex tasks such as collecting user account information and navigating database results. Reusable modules like this can be leveraged to provide rapid application development, code re-use, and the other benefits of component development.

3.1. FXML

There are two types of files in a Florence application: the dialogue strategy file(s), and the application configuration file. Both types are in the Florence XML syntax, FXML.

A Florence application's configuration file provides key information, such as what the top-level dialogue of the application is, what output processor is used, what SLU engine is used, what types of debugging information and log messages will be captured, and so forth. The structure and content of an application configuration data is generally as follows:

```
<fxml>
 <configuration>
  <dialogfile>HelloWorldSD.fxml</dialogfile>
  <slu/><output/>
 </configuration>
</fxml>
```

Figure 5. Florence Configuration File Example

The <fxml> element tag, which is the parent for all other FXML tags used, establishes this as a Florence application file consistent with the FXML schema. The configuration tag establishes the file as an application configuration data file type and contains child elements used to define specific configuration data.

Similarly, the dialogue data is also wrapped in a <fxml> element, followed by the specific FC tag (e.g., <atn>, <clar>, etc.):

```
<fxml>
<atn name="HelloFlorenceSD" start="StartState" default="DefaultState">
 <local/>
 <subdialogs/>
  <!-- Actions -->
  <actiondefs><actiondef name="Greetings" text="Hello, I'm Florence. I have
nothing else to say."/></actiondefs>
 <states><!-- States -->
  <state name="DefaultState/>
  <state name="EndState"/>
  <state name="StartState" pause="true">
   <transitions>
    <transition name="Start" from="StartState" to="EndState">
     <actions><action>Greetings</action></actions>
    </transition>
   </transitions>
  </state>
 </states>
 </atn>
</fxml>
```

Figure 6. Florence "Hello World" Dialogue

The example in Figure 6 is the venerable "Hello World" applications in Florence where an ATN flow controller is used. The elements of a dialogue are primarily the local context, a list of subdialogs that are used, a list of actions that are used in the FC to create output, and the information for the specific flow controller, in this case states and transitions.

This division allows dialogs to be re-used in applications that have different input/output configurations. Even in cases where re-use does not occur, it is very useful in practice to keep multiple configuration files for different environment

setups, such as a command line setup for debugging purposes.

3.2. Native Dialogue Features

Regardless of the type of Flow Controller used to handle the dialogue logic, there a number of features of Florence that will always be available. First and foremost, there are Context Shifts.

The structure of Florence permits a Context Shift to be defined anywhere in the application, and for it to be effective even if the conditions of the shift aren't met until several subdialogs have been called. This makes it possible for an author to include re-usable subdialogs without having to modify them for global shifts that must available, such as a context shift to a help system or operator.

All Flow Controllers also have access to a history of past inputs. For some applications, this precludes the need for extensive record keeping of past user inputs. There are also other features that are necessary for a useful dialogue development environment: a global context is available to expose data to all dialogs and a cleanup dialogue can be defined in case the application is responsible for actions after the dialogue has been completed.

When the DM author wishes to uses any of these features, the control structure is already in place so they can focus on the parameters that are specific to their application. Context shifts, for example, require parameters to describe certain key phrases (such as "quit", "start over", or a complete change of topic) or conditions will trigger an application specific response.

3.3. Development and Deployment Features

XML is used extensively in Florence. It is used for input, to store variable values, to retrieve database results, and to maintain a history. It would be extremely difficult to use and manipulate this information if there was not an XML-specific manner of doing so. Fortunately, there is. Florence applications can use the XPath standard for this purpose.

Florence DM applications can be created and debugged in a local desktop development environment before they are deployed on the J2EE application server. The Florence Toolkit includes a local copy of the XML schema, a local command line tool, and a local SLU server specifically for this purpose. Ultimately, however, DM applications always need to be tested in the deployment environment with the other architectural components.

3.4. Extensibility

Although VoiceXML is the most typical output, an application author can also configure the application to provide output in any XML-based language by replacing the output template and updating the actions in the application with elements to fill the template correctly. This easily extends to multimedia XML formats such as SMIL, SALT or XHTML+Voice. When plain text output is sufficient (as might be the case during application development/debugging), Florence's own simple output processor can be used in lieu of any output template.

The input format can also be changed to another XML compatible format, as long as the information required by the application is present in the new format. This also requires minor changes to the application: XPath expressions which referred to the old format must be updated. As with the input XML, multimedia XML formats can also be used as output.

Also, as mentioned earlier, the Florence framework can be extended with new types of Flow Controllers. The basic logic of the ATN controller can be replaced with a different algorithm for interacting with the user, such as a rule-based algorithm. The basic Florence algorithm described in section 3 will still be executed, but the logic used inside of the Flow Controller will be selected by the application from the available flow controllers. This allows dialogue authors to write sub-dialogs with different algorithms, depending on the nature of the task and use them interchangeably. The author could, for example, have a top-level dialogue based on the ATN model call a subdialog implemented with a rule-based controller.

Florence provides applications with the capacity to call outside systems for data or calculations. Not only is this critical for applications which need to access a database, but it also allows the Florence application to call an outside application to do computation, if necessary.

4. Summary

Florence is a flexible and extensible dialogue manager framework that combines best practices in dialogue management research with support for the industrial-strength needs of commercial conversational services. Florence also provides developers with features to support application authoring, testing, and re-use. The platform-neutral approach and the capacity to support new algorithms make Florence an ideal environment for dialogue research and for large deployments of natural language spoken dialogue systems.

5. References

[1] "Voice Extensible Markup Language (VoiceXML) Version 2.0", W3C Recommendation 16 March 2004, http://www.w3.org/ TR/2004/REC-voicexml20-20040316/

[2] SALT Forum, http://www.saltforum.org/

[3] "XHTML+Voice Profile 1.0", W3C Note 21 December 2001, http://www.w3.org/TR/xhtml+voice/

[4] G. Ferguson and J. Allen, "TRIPS: An Intelligent Integrated Problem-Solving Assistant", in Proc. of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI, 26-30 July 1998, pp. 567-573.

[5] Rich, C.; Sidner, C.L., "COLLAGEN: A Collaboration Manager for Software Interface Agents", An International Journal: User Modeling and User-Adapted Interaction, Vol. 8, Issue 3/4, pages 315-350, 1998

[6] A. Rudnicky and W. Xu "An agenda-based dialog management architecture for spoken language systems", IEEE ASRU Workshop, 1999, p I-337

[7] John R. Searle "Speech Acts: An Essay in the Philosophy of Language", Cambridge University Press, 1969.

[8] Robert E. Schapire and Yoram Singer, "BoosTexter: A Boosting-based System for Text Categorization", Machine *Learning*, vol. 39, no. 213, pp.135-168, 2000.

[9] W3C Multimodal Interaction Framework, W3C NOTE 06 May 2003, http://www.w3.org/TR/mmi-framework.

[10] "XML Path Language (XPath), Version 1.0", W3C Recommendation 16 Nov 1999, http://www.w3.org /TR/xpath

[11] "Natural Language Semantics Markup Language for the Speech Interface Framework", W3C Working Draft 20 Nov 2000, http://www.w3.org/TR/2000/WD-nl-spec-20001120

[12] "EMMA: Extensible MultiModal Annotation Markup Language", W3C Working Draft 18 December 2003, http://www.w3.org/TR/ 2003/ WD-emma-20031218.

[13] D. Bobrow and B. Fraser, "An augmented state transition network analysis procedure", In Proceedings of the IJCAI, pages 557-567, Washington, D.C, May, 1969.

[14] A. Abella and A. Gorin, "Construct Algebra: Analytical Dialog Management", Proc. ACL, Washington D.C., June 1999.