An Approach to VoiceXML Application Modeling

Xin Ni [1] *Meng Ye* [2] *Lianhong Cai* [3]

[1,3]Tsinghua University, Beijing, China
[2]IBM China Research Lab
nx01@mails.tsinghua.edu.cn, yemeng@cn.ibm.com, clh-dcs@tsinghua.edu.cn

Abstract

In this paper an approach for VoiceXML web application modeling is presented with which a voice application can be modeled as a call flow diagram that can be further transferred to be deployable Java code by an automated code generator implemented in the same study. This paper focuses on the modeling work while the brief discussion on code generation is given as well.

1. Introduction

Using speech technology to improve the communication efficiency and naturalness between human and machine is a well-accepted idea. However, until the occurrence of some industry standards such as VoiceXML [1] and SALT [5], the proprietary programming models and APIs from different speech technology vendors set a high bar in terms of skill and knowledge to the application developers. Furthermore, the developed applications lack portability among different speech platforms. With VoiceXML or SALT, voice web applications can be developed with script languages that are neutral to different speech technologies. A general architecture of VoiceXML applications in this manner is shown in Figure 1 in which the voice browser acts as a translator between web servers and telephone users. VoiceXML documents are fetched from the web server and interpreted by voice browser. Although this is an impressive progress, further simplifications are expected – Can we have some sort of visual prototyping environments for this purpose resembling those ones in the market for web application developing?

Our study is aimed at building a visual authoring tool, rendering the evolutionary prototyping approach in [2], for voice application developing. To accomplish the task, two problems should be addressed: 1) a set of visual descriptor for voice application modeling and the corresponding construction rules; 2) a mechanism for automatic code generation based on the visual model created. The first problem can be addressed by the VoiceXML modeling through which the visual descriptor set is derived from the tags of VoiceXML with semantic abstraction. The following criteria should be fulfilled for this: keeping the amount of the descriptors and the rules as small as possible meanwhile ensuring that most VoiceXML based applications could be described clearly enough. As for the second problem, code generation can be achieved by template-based techniques. With both of them the procedure of voice application developing can be broken into two steps: 1) Modeling the application with the visual descriptors and the construction rules; 2) Generating the deployable code automatically based on the callflow diagram, the result of the modeling step.

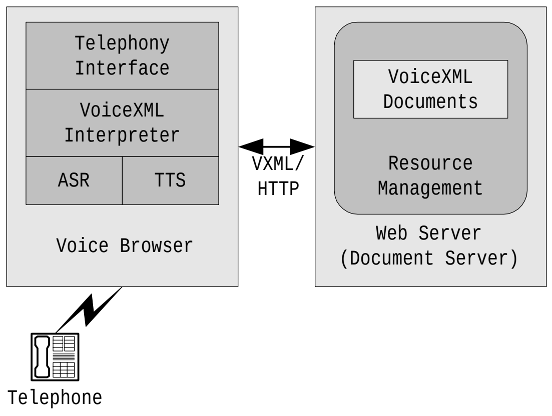

Figure 1. The Architecture of VoiceXML Application

In the two sections following, both the modeling VoiceXML work and the visual descriptors represented in functional blocks are discussed in detail. Moreover, section 4 presents the test and practice on the given model with a brief discussion on the code generation.

2. Modeling VoiceXML

The main task of modeling VoiceXML is to extract a set of functional units and their semantic rules from the VoiceXML language specification, which are both supposed to be represented in visual descriptors. In our work, the modeling requirements are limited to the management of dialogs and their resources. However the programming interfaces for other requirements (such as. specific variable manipulation and some scripting) are considered as well. Some underlying modeling criteria are as following:

1) On the functional (semantic) level, the model should comply with the VoiceXML specification.
2) The functional units should be concise individually and small in total amount.
3) The construction semantic rules should be simple and clear.

Among them, the first one is dominant. For if there are certain voice applications were not be prototyped in our framework, our study would be limited in its utility. The methods we adopted to achieve the criteria above are to be discussed in the following subsections.

2.1. Function Collection and Refinement

A straightforward way to ensure the functional integrity is to collect functions for each VoiceXML tag from the language specification, i.e. collecting each tag's grammar phenomena and extracting the corresponding semantics. For example, for the <prompt> tag, all its functionality components such as text content, audio URI, variable reference, TTS markup and other properties (e.g. bargein, count, and etc) will be collected. However, this method will only derive out another description of VoiceXML without any simplification.

To address this issue, two kinds of refinement work are conducted in our study. Firstly, the simple refinement is used to move some common properties existing in many tags to the <property> tag to reduce the functionality overlap among the original tags. For example, the "bargin" and "timeout" properties of the <prompt> tag can be specified and fulfilled by the <property> tag. Secondly, for those functions that may be fulfilled in different ways according to the VoiceXML specification, the complex refinement is conducted to fix the way they are implemented to facilitate the automatic code generation, through functional combination and replacement. For example, instead of achieving the conditional selection of prompts by using the "count" and "cond" properties in the <prompt> tag, we can achieve the same by the combining of a case unit (see 3.7) and a basic prompt element (see 3.2). The main goal of these refinement works is to achieve simple and pure functional elements that meet the criterion 1. The first criterion is met because we only select a complete subset, in terms of the coverage on all VoiceXML functionalities, from all the possibilities supported by the language specification.

2.2. Function Agglomeration

Because the functional elements generated with above methods are often functional fragments that cannot function independently, agglomeration on their functionality should be conducted to generate further abstracted function modules to simplify the visual modeling of voice applications in terms of reducing the amount of the visual descriptors and the complexity of the construction rules. For example, properties, grammars, prompts and events can be combined into one "input" block (see section 3 for detail).

The agglomeration work is carried out with careful study on typical VoiceXML applications. For example, it is popular in practice to utilize the event handling mechanism of VoiceXML applications to deal with user's commands in different scopes. A simple usage is to specify introductions for the global "help" event in outline while specifying detail help messages for the local inputs. Following this way, the event throwers and event catchers must deal with different scopes.

Another principle is that the functional granularity of each unit is determined by balancing the usability and the flexibility. For example, in the previous refinement works, the function of the <menu> tag is replaced by the combination of a prompt and a case unit. However, the menu selection is a rather commonly used unit that would cause much inconvenience if absent. So the menu unit in our solution yields much more high-level convenience than the "prompt-plus-case" approach does.

Through agglomeration work, functional units are linked only by means of "next jump" or "event handling", and their reference correlation is restricted to variable reference. Consequently the functional units are high-level, in small amount and loosely coupled. Thus criteria 2 and 3 could be both met in this extent.

2.3. Assembling Test

The performance of the VoiceXML modeling in our study is tested by using the generated functional units to a variety of typical voice applications such as voice portal, email access and directory dialer. If there is anything cannot be described clearly in the test, those procedures mentioned in section 2.1 and 2.2 will be conducted again for a better result. Section 4 gives an overview on test and practice on the model.

3. Block Description

Through the VoiceXML modeling ten fundamental blocks are constructed and are to be discussed in this section. Each block has its functions for dialog management. Carrying out basic tasks, they could be further combined as composite blocks for more complicated tasks.

3.1. Start and Exit

In our framework, an application is restricted to be a composition of various functional modules with a single entry point and a single exit point. In the “exit” block, an exit message can be specified and will be played when application exits. In the “start” block, several meta settings can be defined like the application general settings (e.g. application name, resource path of audio and subdialog), the global properties and events, the resource identifiers, and possible embedded code for initialization. The paths of audio and subdialog tell the program where to find the corresponding resources on the web server. And the global properties and events are active throughout the runtime execution of the application unless overridden by local ones (defined in a non-start block). Denoted by arrowhead lines in diagram, the events are classified into two types: system and custom. Two system events, i.e. "help" and "error", can be specified in the start block. A custom event acts as the <link> tag in VoiceXML, which triggers a flow jump if the user's voice input matches its grammar. Resource identifiers are actually Java beans containing external resources that can be converted to text content. In order to make the framework more extensible, programming interfaces are defined and ready for developer to embed extra Java code to fulfill further tasks such as the manipulation on backend database. And in the “start” block, some initializing tasks such as connecting to third-party application servers may be required and implemented by developers.

3.2. Prompt

The prompt block handles the output of synthesized speech or prerecorded audio. In this block, the prompt contents are composed of many segments that will be played in order. Prompt segments can be of 4 types: text, audio, block and resource. The text content, the audio URI, the block label and the resource identifier are required respectively for these 4 types. The block type is designed for the variable references of form items (e.g. input and record reference). The external resource (e.g. records in database) is accessed through the resource identifiers and will be transformed to pure text content when needed at runtime.

3.3. Input

This block catches user’s voice input according to its grammars. It includes prompt messages, grammars, local properties and local events. The grammars can be any one of the following 4 types or their combination: builtin, external, inline and resource. All builtin grammars defined in VoiceXML specification are supported. The external grammar can be fetched through a specified URI. With the inline type, the grammar can be defined at the design time while with the resource type it can be obtained from external resources at runtime. Three system events can be defined locally in this block: "help", "noinput" and "nomatch". Local custom events are also supported.

3.4. Record

This block is for collection of user utterances. It consists of prompt messages, recording properties (e.g. beep prompt, maximum time for recording and DTMF termination) and a handler for the "noinput" system event. Other events are not included due to the runtime characteristic of recording. The collected utterance will finally be submitted to the remote web server for further manipulation.

3.5. Subdialog

The subdialog is the way defined in VoiceXML specification for both leveraging the reusable dialog components [4] from the 3rd parties such as the IBM's reusable components [3] to achieve software reusability, and increasing the modularity of VoiceXML applications. It is kept exactly in our framework for same purposes.

3.6. Menu

The menu block in our framework has the same functionality as the <menu> tag of VoiceXML. However it is implemented by combining the <field> tag and the <option> tag. This makes the user's voice input available for further reference by other blocks.

3.7. Action, Case and Loop

These 3 blocks have no direct relationship with any VoiceXML tag. They are introduced into the prototyping framework with the consideration of flow control, flexibility and extensibility. Loop construction and selection construction are commonly adopted in all modern programming languages for flow control. In VoiceXML specification, these two kinds of construction are weakly supported given the fact that the web-programming model is indeed an MVC model. In the VoiceXML application case, the VoiceXML code of most voice applications, with the exception of the very simple ones containing only static VoiceXML pages, falls in the View part of the MVC model, which runs on the client side (the Voice Browser in Figure 1), and the logic of the Controller part runs on the server side. Hence only weak support on flow control is needed from the VoiceXML perspective in a static MVC model. However, in practice, the VoiceXML code of most applications is generated dynamically at runtime by the code of the Controller part. In other words, if we view a complete voice application as a web application and VoiceXML code is the result of the logic of the Controller part, full support on flow control becomes a necessary. On the other hand, the Action block is designed to provide a hook for embedding extra code.

4. Practice

In order to facilitate the code generation stage, the application specification, which is an XML description of the callflow diagram, is adopted to keep the results of the application-modeling task. The application specification could be automatically generated according to the visual diagram by fetching the properties of each visual descriptor in the diagram with a well defined DTD. In the scenario practice shown in figure 2, a sample of such specification is given and the corresponding Java servlet code is generated by parsing it. As shown in the example, each descriptor is described in a separate tag element. The template-based code generation is leveraged in our study. There will be one or more templates for each descriptor depending on the complexity of its functionality. Two stages are involved, as shown in figure 3. In stage 1, code templates are created according to the specifications of the functional units, while in stage 2, the result templates are tested through assembling. In real code generation, in addition to generating individual VoiceXML pages for each block, the code generator should accumulate as many blocks as possible in a single VoiceXML page to improve the communication efficiency of the voice browser and the web browser and thus reduce the load of the server side.

From the example fragment we can find that the resource identifiers yield many dynamic characteristics to the target application. For example, in the "Service" menu, the content of the identifier "other" will not be determined until at runtime (may be "traffic" or something else).

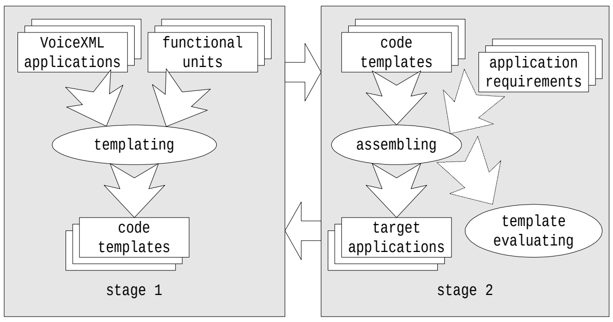

Figure 3. Code Templating

5. Conclusions and Future Work

In this paper an approach for VoiceXML application modeling is presented. Future work includes a full-featured GUI environment for diagram design and the construction of the composite functional block to simplify modeling work of complex applications.

6. References

[1] VoiceXML Specification http://www.voicexml.org http://www.w3.org/TR/voicexml20
[2] Fabrice Kordon and Luqi, "An Introduction to Rapid System Prototyping" IEEE Transactions on Software Engineering, vol. 28, no. 9, Sep. 2002
[3] Stephane H. Maes, "A VoiceXML Framework for Reusable Dialog Components" Symposium on Applications and the Internet (SAINT) 2002.
[4] Reusable Dialog Requirements for Voice Markup Language, http://www.w3.org/TR/reusable-dialog-reqs

[5] Speech Application Language Tags (SALT) Specification http://www.saltforum.org

```
<input name="Yourname">
  <message>
    <segment type="text">what's your name?</segment>
  </message>
  <grammars>
    <grammar sort="grammar" type="external">
      gram/names.gram
    </grammar>
  </grammars>
  <jump target="Welcome"/>
</input>

<prompt name="Welcome">
  <message>
    <segment type="text">hi</segment>
    <segment type="block">Yourname</segment>
    <segment type="text">welcome to</segment>
    <segment type="resource">placename</segment>
    <segment type="audio">
      <src>audio/introduction</src>
      <instead>This is a nice place.</instead>
    </segment>
  </message>
  <jump target="Service"/>
</prompt>

<menu name="Service">
  <message>
    <segment type="text">
      what can I help you, hotel or
    </segment>
    <segment type="resource">other</segment>
  </message>
  <options>
    <option dtmf="1">
      <fixedgrammar type="inline">
        hotel
      </fixedgrammar>
      <jump target="Hotel"/>
    </option>
    <option dtmf="2">
      <fixedgrammar type="resource">
        other
      </fixedgrammar>
      <jump target="Other"/>
    </option>
  </options>
</menu>
```

Figure 2. Fragment of Application Specification Example