An Architecture for Pluggable Disambiguation Mechanism for RDC based Voice Applications

Tanveer Faruquie, Pankaj Kankar, Nitendra Rajput, Abhishek Verma

IBM India Research Lab
Block 1, IIT Campus, Hauz Khas, New Delhi, India
{ftanveer,kpankaj,rnitendra,abhverma}@in.ibm.com

Abstract

Building speech-based conversational systems involves the development of several speech specific control mechanisms such as validation, confirmation, disambiguation in addition to the actual application call-flow. We present an architecture for pluggable disambiguation mechanisms for speech based conversational systems. The architecture provides a mechanism to decouple the disambiguation from the voice application. Several disambiguation strategies can be designed to disambiguate a user input. These strategies can be applied to the user input in a seamless manner. The disambiguated value from one component can be passed on to another component for further disambiguation. We implement the architecture by using it in the Reusable Dialog Component framework. Several illustrative examples are presented to highlight the effectiveness of having a pluggable disambiguation mechanism for voice applications.

1. Introduction

Speech based conversational system work on the basis of recognizing and understanding the user input. The system gets data through the user’s input, and the system also navigates based on the user input. However, there can be several reasons due to which the system is not able to understand what the user intended to say. In such cases, the conversational system needs to clarify the user input. This mechanism of clarification is known as disambiguation.

Disambiguation is an extremely important step in any user interface design. The design of a good disambiguation mechanism results in improving the usability of the system as a whole. Researchers have worked extensively on improving the disambiguation strategies in web-based interactive systems and also on speech based telephony systems. The disambiguation mechanisms in the voice domain are still application dependent [1, 2]. Mankoff, et.al. in [3] present various correction strategies, known as mediators, which are enabled on a particular toolkit. In [4], the authors present an architecture to handle ambiguity at the input event level.

In speech based conversational systems, there can be several instances wherein the system would need to disambiguate [5]. The ambiguity may arise due to multiple choices in speech recognition, understanding or target identification. The design of a disambiguation system should be such that it supports all the different modalities of ambiguity resolution.

Most conversational systems design the disambiguation in an ad-hoc fashion, such that the disambiguation is specific to the application and hence needs to be redesigned for every application [6]. In this paper, we propose an architecture for modeling domain independent ambiguity resolution. The central idea is to decouple the disambiguation from the voice application specifics. By providing disambiguation as a separate component in the voice application, not only does the application development becomes simpler, several interesting disambiguation mechanisms can be provided to the application. The architecture allows for piping the input of one disambiguator to another in a seamless manner. We illustrate the effectiveness of the disambiguation architecture in a Reusable Dialog Component framework [7] setting. As has been shown in [7], the RDCs provide a framework for rapid voice application development. We further enhance this framework by providing an architecture for pluggable disambiguation strategies.

We describe the programming model setting in Section 2. Several disambiguations in the RDC framework have been presented in Section 3. The disambiguation architecture is defined in Section 4. Section 5 illustrates the framework through examples. Conclusions and the discussion on future scope of work is presented in Section 6.

2. Voice Programming Model

The voice application development environment has graduated from the executable programs that used to be linked directly to the underlying speech engines towards a standards-compliant VoiceXML platform. This allows application development that is independent of the speech technologies API for a particular speech engine. Moreover, the server-side deployment of voice applications has facilitated the generation of dynamic VoiceXML applications, that can better handle the dynamic nature of interaction and hence make the application powerful.

Voice applications execute in a temporal manner, and hence require special mechanisms to handle error correction, confirmation, validation and disambiguation. These speech specific nuances of a voice application can be handled within prepackaged dialog components. This eases the overall task of application development and its use by non-speech specialists.

In this paper, we present the disambiguation architecture within the Reusable Dialog Component framework [7]. RDCs use a Finite State Machine (FSM) to model the various interaction states attained by the component.

The FSM of an RDC is shown in Figure 1. The FSM starts in either Dormant or Input state depending upon whether it is inside a container or hosted directly within the JSP page. In Dormant state the atom does nothing. When the atom is in Input state, it generates VoiceXML markup for accepting input from the user. After receiving input, it is checked for ambiguity, canonicalized and validated against constraints. If the input is valid, then the FSM transitions to Confirmation or Done depending upon whether the confirm attribute of the atomic RDC
Ambiguities arising in a dialog are handled by the RDC framework independent of the domain. The framework uses easily configurable automatic or explicit confirmation strategies. These strategies choose candidates considering one or a combination of the scores of the recognition results of the user utterance, the raw data, the context of the input, the dialog state and the validation constraints specified by the developer. The strategies can be integrated easily to generate a disambiguous dialog flow to the user.

Ambiguities can arise from different sources. At the highest level they arise because of the ambiguity in users mind about their expectation from the system. Such ambiguities are handled by providing flexible dialog navigation so that a user can jump back to the point where he wants to change the input. The framework supports navigational command such as go-back with target specification. The application developer has the option of enabling or disabling this feature at the RDC level. For every RDC, a target RDC can be specified, which is the one that is best suited to disambiguate the current dialog path.

Target ambiguities are caused by ill-formed user phrases which lack complete semantic content. Utterances such as ‘next Friday’, ‘at five Oclock’ are examples where the exact value is open to interpretation. Such kinds of ambiguities are handled in the data-model of the atomic RDC. The data-model disambiguation is programmed to disambiguate such utterance either using implicit disambiguation strategy or using explicit confirmations. An example of implicit disambiguation strategy is implemented in the tree based RDC, as will be shown in Section 5.2. The explicit confirmation strategy is implemented in the date and time RDC. In this case, the system explicitly asks the user to choose between ambiguous candidates. For example, a user utterance at five Oclock will be disambiguated by asking Five am or Five pm?

The ambiguities at lowest level are due to the recognition errors by the ASR. Such ambiguities should be handled early-on to avoid the propagation of errors caused by them. These can be resolved by using one or more of the disambiguators/mediators available specifically for the purpose. These disambiguators work on the confidence scores returned through the N-best list of the voice browser. A simple naive disambiguator returns the top entry of the N-best score. The threshold disambiguator takes two parameter $S$ and $R$ to prune the entries of the N-best list. It only consider those candidates whose score $s$ is above the threshold $S$ i.e. $s \geq S$ and the score difference $\delta s$ with the next candidate in the list is more than $R$. Mediators can also take the constraints specified in the RDC to score the individual candidates of the list by assigning a relevance score. Hence for a candidate $c$, $r_i(c)$ is the relevance score for the $i^{th}$ constraint specified in the RDC. $r_i(c)$ is defined as

$$r_i(c) = \begin{cases} 1 & \text{if } c \text{ satisfies the } i^{th} \text{ constraint} \\ 0 & \text{otherwise} \end{cases}$$

The final score, $r(c)$ of each candidate in the list is

$$r(c) = \sum_i r_i(c)$$

The candidate with highest relevance score is selected by the naive disambiguator, or, the threshold disambiguator can be used again to prune the list. An example is presented in Section 5.1.

4. Disambiguation Architecture

We define an extension to the RDC framework such that the recognition ambiguity in user input is resolved before being passed to the RDC. The extension allows an application developer to define new mediation strategies as well as reuse the existing ones that can be configured through an xml file.

![Figure 1: An RDC FSM.](image)

In case of recognition ambiguity, the proposed architecture can be plugged into the RDC framework at the point of user input. Figure 2(a) represents the input state of the RDC finite state machine at a finer granularity level without the disambiguation extension, whereas Figure 2(b) represents the same with the disambiguation extension. As shown in Figure 2(a), the input state comprises of prompting the user for input, collecting the user input and passing the collected input to the RDC. In this setting, the RDC is responsible for resolving the recognition ambiguity among the collected values. The proposed extension takes care of this problem through a pluggable, configurable disambiguation framework. As shown in Figure 2(b), the disambiguation takes place once the input has been collected from the user. Once the ambiguity is resolved, the outcome is passed on to the RDC.

We define an interface MediationStrategy as shown below.

```java
public interface MediationStrategy {
    // Set the configuration xml and ambiguous input set
    public void initialize(String config, Vector inputs);
    // Try and disambiguate the input set and return the outcome
    Set disambFinish = true, if disambiguation was completed
    // = false, if the disambiguation is not over
```
public Vector disambiguate();

// Same as disambiguate(), except that this gets executed on
// returning from user interaction.
public Vector continuedDisambiguation(Vector inputs);

The ambiguous input set could be resolved either with or without user interaction. When the ambiguity is resolved without user interaction, it is called automatic mediation. We have implemented the MediationStrategy interface to define a few automatic mediation strategies such as:

- Picking the input with the highest confidence score,
- Picking the inputs with confidence score above a threshold value.

The application developer can implement his own automatic mediation strategy by defining the disambiguate() function as required, and setting the disamFinished variable to true.

We implement the MediationStrategy interface to define the following two mediators that interact with the user:

**ChoiceMediator**: It presents the user with a set of choices and prompts him to choose one of them. The mediator can be configured on the prompt and option type to be presented to the user. For e.g., Press 1 for alpha, 2 for beta, etc. Say alpha for alpha, beta, etc. Such a mediator could be used in case there are utterances with approximately the same confidence score.

**RepetitionMediator**: The mediator prompts the user to repeat his input. It can be configured for the prompt to be presented to the user. For e.g., I was not able to understand what your said. Could you please repeat the input. Such a mediator could be used in case all user utterances are extremely low on confidence.

Both of the above mediators can also be configured on speech specific parameters like timeouts, barge-in, etc. For any custom mediation strategy involving user interaction, the application developer must define both disambiguate() and continueDisambiguation() functions.

Any combination of these mediators can be serialized one after another such that the output of one mediator acts as the input to the other. In order to define the flow between various mediators, the application developer can pass an xml file to the respective RDC using the attribute disamConfig. We have defined a Java class Disambiguate that reads this xml file and manages the flow between the defined mediators. The Java classes for various mediators are defined as named entities within the xml file and the flow between these entities is defined using the xml constructs as shown in the example below. The disambiguation process stops once a unique value has been obtained, or the defined flow has been executed. We have also provided support for a naive disambiguation mechanism, wherein the input with the highest score is passed on to the RDC. If the application developer wishes to use such a mechanism, the tag attribute useNaiveDisam should be set to true. Moreover, if the application developer wishes to bypass the provided disambiguation framework and let the RDC handle the recognition ambiguity itself, no value should be specified for the attribute disamConfig.

For any other type of ambiguity, including target ambiguity, we have a disambiguation state within the RDC finite state machine. It is important to note that the target ambiguity should be handled by the RDC internally since such ambiguity is very specific to the RDC itself. For e.g., a phrase like next Saturday is very specific to the date RDC. Also, target ambiguity can be handled only on resolution of the recognition ambiguity so that we have a single value to operate upon.

5. **Voice applications using disambiguation**

In this section, we illustrate the advantages and working mechanisms of the disambiguation feature in a voice application through a disambiguation in a tree RDC.

5.1. **Example of recognition disambiguation**

The recognition disambiguation in this example has been configured to have the following two mediators:

**Automatic Mediator**: It takes two parameters $S$ and $R$ to prune the input set. It considers only those candidates whose score $s$ is above the threshold $S$ i.e. $s \geq S$ and the score difference $\delta s$ with the next candidate in the list is more than $R$.

**Choice Mediator**: It prompts the user to select from a list of candidate answers. The prompt to be presented to the user is Please say 1 for option1, 2 for option2, or 3 for option3. The user can reply either through DTMF or spoken input.

The automatic mediator works on the input set and prunes it down depending on the two parameter values. If the pruned set contains just one element, then the disambiguation is completed and disambiguated value is passed to the RDC. Otherwise, the pruned set is passed to the choice mediator that prompts the user for correct option. Once the user specifies the choice, the selected value is passed to the RDC.

The mechanism gets plugged in the framework as shown:

```xml
<rdc:id=date disamConf=confi g/dateDisam.xml/>
```

The configuration files for flow across mediators and the choice mediator are shown below:

```xml
<config/dateDisam.xml:
  <initial name=sam/>
  <rule from=sam to=cm/>
  <entity name=sam class=disambiguate.PruneMediator />
  <entity name=cm class=disambiguate.ChoiceMediator confi g=confi g/cm.xml/>
  <config/cm.xml;
    <prompt>
      Please say 1 for option1, 2 for option2, or 3 for option3.
      Please enter your choice.
    </prompt>
  </config/cm.xml>
```

5.2. **Tree RDC disambiguation mechanism: Case study**

An example illustration of the target ambiguity being resolved internally within the disambiguation state is the Tree RDC. The tree RDC is responsible for traversing through a tree of nodes, depending upon user input, and for reaching a target node. Each node within the tree has compulsory attributes as shown in Figure 3, and any number of user-defined attributes. The tree to be traversed is provided to the RDC using an xml file.

Target disambiguation plays a significant role since the user input might lead to multiple target nodes. We provide configurable disambiguation mechanism, wherein the application developer can define the sequence of differentiators to be applied to disambiguate the target nodes using the tree RDC tag.
attribute disamMethod. There are three special differentiators, viz. type, context, and name. type is type of the tree node, context is ancestors of the tree node, and name is different possible names of the tree node as defined by the node attributes name, alt1, alt2, etc. The application developer can also use the node attributes, both compulsory as well as user-defined, as differentiators.

Figure 3: The tree node and its attributes

![Tree node with attributes](image)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>/name of node</td>
</tr>
<tr>
<td>alt1</td>
<td>/alternate name of node</td>
</tr>
<tr>
<td>altN</td>
<td>/alternate name of node</td>
</tr>
<tr>
<td>type</td>
<td>/type of node</td>
</tr>
<tr>
<td>apt-code</td>
<td>/user defined attributes</td>
</tr>
<tr>
<td>loc-name</td>
<td>/user defined attributes</td>
</tr>
</tbody>
</table>

Figure 4: A sample tree for people working in different companies

![Sample tree](image)

If the tree RDC detects target ambiguity, it moves to the disambiguation state and applies the differentiators one after another, in the order as specified by the application developer, on the ambiguous nodes until it is able to identify the differentiator that can uniquely identify every node. After this, the tree RDC presents the user with choices based on the selected differentiator to determine the user's intent.

**Scenario 1**

C: What name?
U: Robin
C: I know two people with the name Robin. Are you looking for the male Robin or the female Robin?
U: Female (Node D selected)

**Scenario 2**

C: What name?
U: Bobby
C: Which Bobby, the one who works for ABC or XYZ?
U: XYZ (Node G selected)

Figure 5: Sample disambiguation dialogs

Consider the tree of Figure 4. It represents a list of people working in a city categorized by organization and department. The target nodes are the people, while organization and department are intermediate nodes. The application developer has specified the differentiator sequence as "\texttt{sex, name, context}; \texttt{sex} being a user-defined attribute. Scenario 1 in Figure 5 shows that the user utterance is \textit{Robin}. Since there are two target nodes that match this user utterance, viz. node A and node F, the tree RDC will apply the differentiators on these two nodes starting with user-defined attribute \texttt{sex}. It realises that the nodes can be differentiated on sex since one is male and the other is female. So, it will ask the user to choose between male or female. Scenario 2 in Figure 5 shows that the user utterance is \textit{Bobby}. The two target nodes that match the utterance are both male and can not be disambiguated on alternate names either. But the two can be differentiated on the basis of context since one works for ABC and the other works for XYZ. So, the tree RDC will prompt the user as shown in Figure 5.

The above examples demonstrate the effectiveness of a pluggable and configurable disambiguation mechanism. The application developer has a choice to develop a particular disambiguation mechanism. The architecture allows for plugging in external disambiguation components into the application.

6. Conclusion and Future Work

This paper contributes an architecture for a pluggable and configurable disambiguation mechanism within the RDC framework. The need for disambiguation in voice applications is highlighted and different types of ambiguities in a conversational system are described. The implications of the presented architecture in voice applications have been described and a case study of a tree structured RDC is presented to illustrate the effectiveness of the proposed disambiguation mechanism.

Providing disambiguation as a separate component in the architecture is a step further in the direction of isolating voice application development from speech specific nuances. The disambiguation framework can be further enhanced to incorporate ambiguities arising from within the application. The richness of disambiguation architecture will be significantly improved as more disambiguation strategies get designed and are available to plug into voice applications as independent components. The importance of a disambiguation mechanism is going to grow as more complex voice applications are developed, demanding more understanding from a voice system.

7. References


