Avatar 2.0

Telecooperation

TK Project Student Lab

Saroj Sharma, Tulasi Ram Valleru, Debashis C.Ray, Ravula Pranay, Raja Sekhar Pula Venkata

Supervisor: Stephan Radeck-Arneth, Msc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Telekooperation
Prof. Dr. Max Mühlhäuser

Index

1. OVERVIEW .... FEHLER! TEXTMARKE NICHT DEFINIERT.

1.1 AVATAR 2.0 .... **FEHLER! TEXTMARKE NICHT DEFINIERT.**

1.2 SOFTWARES USED TO RELAIZE THIS SYSTEM .... **FEHLER! TEXTMARKE NICHT DEFINIERT.**

2. PROJECT GOAL **3**

3. ARCHITECTURE **4**

3.1 OVERALL ARCHITECTURE 4

3.2 PROJECT ARCITECTURE 4

4. PROJECT COMPONENTS **5**

4.1 JVOICEXML SERVER 5

4.2 MARYTTS SERVER 5

4.3 VOICE SYNTHESIZER 6

4.4 SPHINX 6

4.5 VOICEXML 6

4.6 BML 6

5. PROJECT CONFIGURATION **6**

5.1 JVOICEXML CONFIGURATION 6

5.2 TALKINGHEAD INTEGRATION 7

6. DESIGN DECISIONS AND FINDINGS **8**

6.1 UNIDIRECTIONAL 8

6.2 BIDIRECTIONAL 9

6.3 SUBDIALOG 10

6.4 FINDINGS 11

7. IMPLEMENTED SCENARIO **11**

7.1 WITH ANIMATION 11

7.2 WITHOUT ANIMATION 12

8. CONCLUSION **18**

9. FUTURE SCOPE **18**

10. REFERENCES **19**

11. APPENDIX **19**

1. Overview

1.1 Avatar 2.0

The scope of this document is explaining the configuration required for Avatar 2.0. Each component used for the project are explained in detail. This document also contains the design decisions taken for the approach and the reason for the decisions. Important tags are discussed in brief. It also contains the issues faced and solutions to overcome these issues.

1.2 Software used to Realize the System

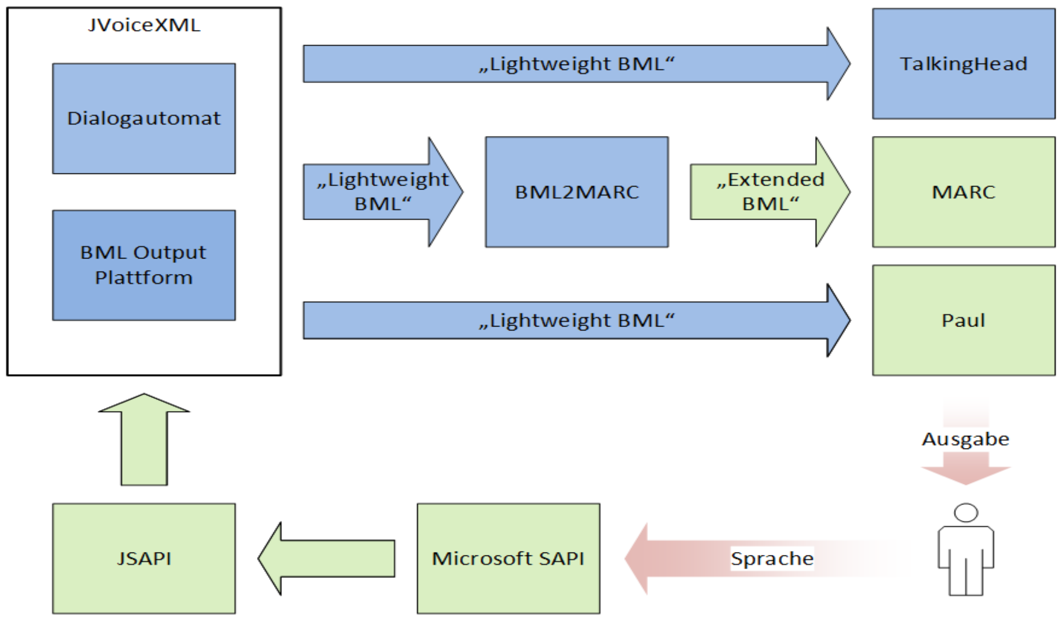
The project deals with the possibility of creating a system which creates not only a useful conversation with the user but also engages the user with animations that represent the machine conversation. With the system, a designer can design and implement real world scenarios in a document using a format which is open like VoiceXML and BML and can run these documents in the system. Now the system can engage in a conversation with the user for those particular scenarios. The system not only take voice feedback and reply user with the relevant answer but, it engages the user visually by producing relevant animations for the conversation.

2. Project Goal

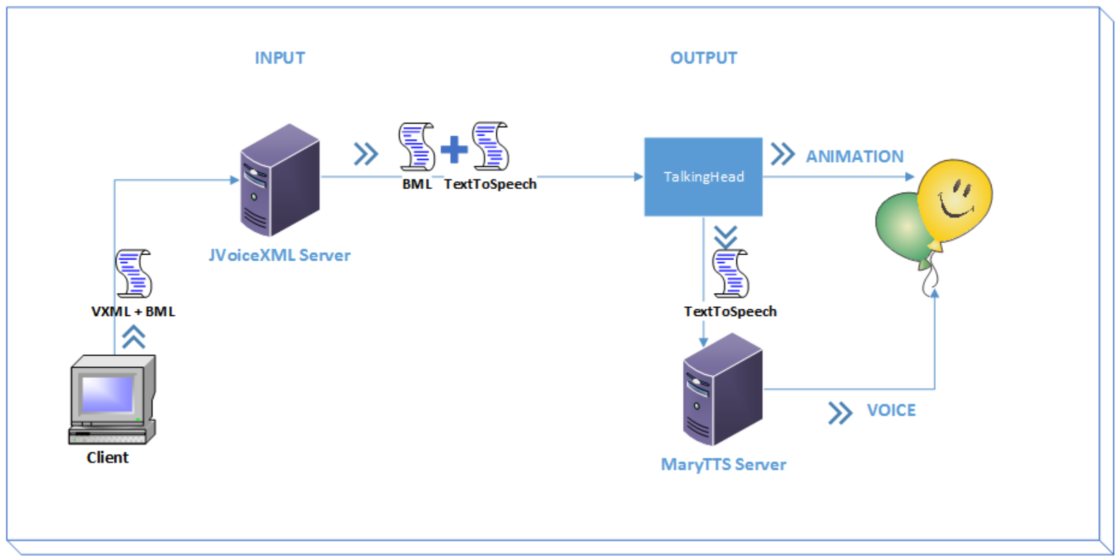
- Develop control dialog flows in VoiceXML based on given Scenarios.
- Scale the scenario across multiple VoiceXML documents in a modular fashion to implement the concept of reusability.
- Identify the scope of the VoiceXML documents for scaling the functionality of the avatar.

3. Architecture

3.1 Overall Architecture



3.2 Project Architecture



With Animation:

The VXML+BML file is sent to JVoiceXML server. JVoiceXML server interprets the VoiceXML tags into Text-to-Speech. The Text-to-Speech and BML is sent to TalkingHead which sends the Text-to-Speech to MaryTTS and also generates the animation. The output of this will be the synchronized voice and the animation.

Without Animation:
VoiceXML files are sent to JVoiceXML which interprets and gives the output.

Achieving Reusability

The reusability is achieved by making the splitting the dialogs into different VoiceXML documents. All these documents are related with respect to the flow based on the scenario. We have used submit and 'goto' tags to link the documents. These tags are explained in detailed in the later part of this document. Form id's are used for each dialog within the document and are used for moving from one form to another. Only one grammar file is used for the entire scenario so that to reduce the complexity of having the more grammars which might lead to the confusion state for the machine to understand the user input. But, the grammar file has to be called in each of the vxml document where ever is necessary.

Example, Emergency.vxml, a simple VoiceXML document which doesn't required any user input is implemented to call the Emergency form id in the Start.vxml which is to exit from any point of the scenario. The control jumps to Emergency.vxml from any of the vxml document if there is no input from the user, treating there might be an emergency and should exit the scenario.

Benefits to VoiceXML document Developer

- ➔ Can reuse the dialogs by calling them.
- ➔ Emergency scenario is fully reusable
- ➔ Can reuse the grammar tags for matching the user input

Transparency to the End User

- ➔ User will have the impression of a single document
- ➔ Control flow between the documents is smooth

4. Project Components

4.1 JVoiceXML Server

JVoiceXML is a VoiceXML implementation written in the JAVA programming language. It offers a library for easy VoiceXML document creation and a VoiceXML interpreter to process VoiceXML documents using JAVA standard APIs such as JSAPI and JTAPI.

4.2 MaryTTS Server

MaryTTS is an open-source, multilingual Text-to-Speech Synthesis platform written in Java. It was originally developed as a collaborative project of DFKI's Language Technology Lab and the Institute of Phonetics at Saarland University. It is now maintained by the Multimodal Speech Processing Group in the Cluster of Excellence MMCI and DFKI.

4.3 Voice Synthesizer

Speech synthesis is the artificial production of human speech. A computer system used for this purpose is called a speech synthesizer, and can be implemented in software or hardware products. A text-to-speech (TTS) system converts normal language text into speech; other systems render symbolic linguistic representations like phonetic transcriptions into speech.

4.4 Sphinx

Sphinx4 is a pure Java speech recognition library. It provides a quick and easy API to convert the speech recordings into text with the help CMUSphinx acoustic models. It can be used on servers and in desktop applications. Beside speech recognition Sphinx4 helps to identify speakers, adapt models, and align existing transcription to audio for time stamping and more.

4.5 VoiceXML

VoiceXML (VXML) is a digital document standard for specifying interactive media and voice dialogs between humans and computers. It is used for developing audio and voice response applications, such as banking systems and automated customer service portals.

4.6 BML

The communicative behavior markup language, or BML, is an XML based language that can be embedded in a larger XML message or document simply by starting a block and filling it with behaviors that need to be realized by an agent. The behaviors are listed one after another, at the same level in the XML hierarchy, with no significance given to their order. Generally the behaviors are single elements that contain no text or other elements, but this is not required. Behavior parameters, some of which are general and some of which are behavior specific, are specified as attribute values of the behavior element.

5. Project Configuration

5.1 JVoiceXML Configuration

The detailed explanation for configuring the JVoiceXML is given in JVoiceXML User Guide. For the current project we have used sphinx as a voice recognizer.

Following are the only additional configurations required for this project:

Copy files from org.JVoiceXML/config-props to org.JVoiceXML/personal-props and Enable BML and JSAPI20 in personal-props.

5.2 TalkingHead Integration

Animations are implemented by calling the sequence of smiley images which are under smiley images folder of Avatar project. Each animation is named with an id which contains the sequence of the images which results is a particular gesture. This is implemented in TalkingHeadConfig.xml which is in TalkingHead folder. To use TalkingHead for animations, follow the below steps

Change the .project file in ***avatar/org.JVoiceXML.implementation.bml*** with the following lines.

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
   <name>org.jvoicexml.implementation.bml</name>
   <comment></comment>
   <projects>
   </projects>
   <buildSpec>
         <buildCommand>
               <name>org.eclipse.jdt.core.javabuilder</name>
               <arguments>
               </arguments>
         </buildCommand>
         <buildCommand>

   <name>net.sf.eclipsecs.core.CheckstyleBuilder</name>
               <arguments>
               </arguments>
         </buildCommand>
   </buildSpec>
   <natures>
         <nature>org.eclipse.jdt.core.javanature</nature>

   <nature>net.sf.eclipsecs.core.CheckstyleNature</nature>
   </natures>
</projectDescription>
```

Import the org.JVoiceXML.implementation.bml folder into JVoiceXML project. Run the build.xml file of org.JVoiceXML.implementation.bml by which the jar file of this interface is copied into org.JVoiceXML/dist of JVoiceXML project.

Add the following line in the “platforms.xml”, which activates the BML-interface for JVoiceXML.

<platform name="org.JVoiceXML.implementation.bml"/>

Copy bml-implemantation.xml file into config folder of JVoiceXML project. Change the host and port to the following.

<beans:property name="host" value="127.0.0.1" />
<beans:property name="port" value="14010" />

Import libraries, org.jbml and TalkingHead folders into JVoiceXML project.

Change the below lines in TalkingHeadConfig.xml which will be in TalkingHead folder.

<input host="127.0.0.1" port="14010" buffer="65535" />
<output host="127.0.0.1" port="14011" buffer="65535" />
<xsddef filename="bml.xsd" />
<tts host="localhost" port="59125" locale="en-GB" voice="cmu-slt-hsmm" />

Import org.JVoiceXML.demo.dialogbml into JVoiceXML project.

Execution:

1. Run MaryTTS server
2. Run TalkingHead.java file from TalkingHead/src folder
3. Run JVoiceXML Server
4. Run the build from org.JVoiceXML.demo.dialogbml

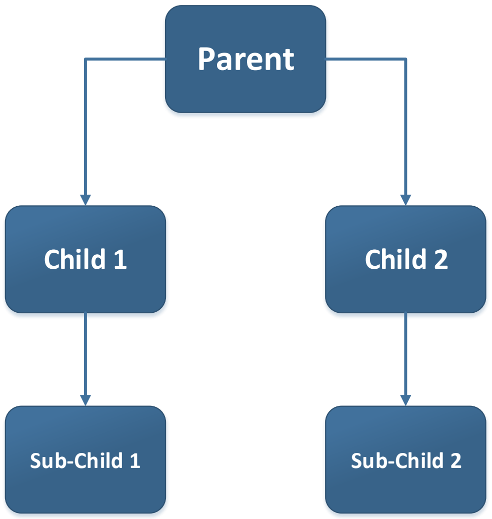
6. Design Decisions and Findings

To achieve flexible implementation of real work scenarios and to reduce redundancy (implementing same set of scenarios where ever encountered), we chose to divide a big scenario into subset of VXML documents representing a unique scenario which can be referred by other vxml documents. Thus an entire scenario can be described using modules with unique functionality and linking them representing the path. We achieved modularity using various elements provided by voice xml.

6.1. GOTO tag

<goto> element provides a unidirectional approach which can be used to transition application execution to another element or document. The <goto> element can transition to a specific form within the current form, dialog in the current document, or to a separate document. Additionally, it can transition execution to a specific form within a separate document.

Link two VoiceXML documents. Traverse from one vxml document to another but the flow is unidirectional.



Where does it satisfy?
This element can be used when the flow should be transferred to another document but doesn't need to return to the calling document.

Example:
We use this pattern normally during emergency or where ever we intent interpreter to exit.

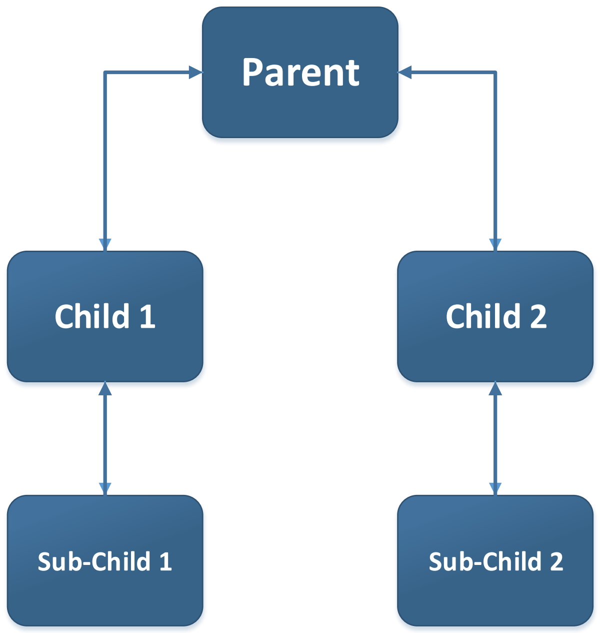
Can it be used to achieve full modularity?
Modularity concept cannot be realized only using GOTO tag as most of the functions need to return to the caller after finishing their task and this functionality won't be realized.

Solution 1:
The complex way to complete the desired functionality is by using variables and some java script to return to the document.
Solution 2:
Make use of Submit tag which provides bidirectional flow

6.2. Submit tag

The <submit> tag performs a GET or POST that will trigger a page transition. This means that document level variables will be lost once the new page is fetched and parsed. To preserve some of the current execution context, the <submit> tag can send document level variables as POST or GET variables to your application server script. This application server script gets parsed by your application server, which can then process variables sent to it before generating a VoiceXML document to be parsed by the VoiceXML platform to interact with the caller.

Link two VoiceXML documents. Traverse from one vxml document to another and the flow is bi-directional.



What problems it intend to solve?
Flow can be always returned to the initiator from the called document. This will serve the purpose of reusability by linking the documents.

Example case:
The documents which perform subset of tasks can always be called and can return to the state of the initiator document.

Where doesn't it satisfy the needs and how we overcome the need?
Many-to-one: Many voice xml documents calling a single document is not possible and so in some cases we use GOTO tag (only if the flow is bidirectional).

6.3. Subdialog

The <subdialog> tag is similar to a function call executed by a GET or POST. It allows you to execute a new VoiceXML document in a new context, but once the <subdialog> is complete, control will be returned to the parent document at the same location that the subdialog was called. The way this process works is that the <subdialog> tag sends document level variables to your application server script. Once the application server script gets parsed by the application server and generates a VoiceXML document, this VoiceXML document becomes a “Subdialog” of your original VoiceXML document. Below is a diagram that shows how multiple Subdialogs' can be used before returning context back to the original VoiceXML document.

Advantage:
Powerful. It can be called from any document. It can also do the job GOTO and Submit are intended to do.

Disadvantage:
Complex to control.

"While the second document is being executed, the calling dialog is suspended, awaiting the return of information. The second document provides the results of its user interactions using a <return> element, and the resulting values are accessed through the variable defined by the name attribute on the <subdialog> element."

Need to pass lot of parameters.

Will it make life easy for VoiceXML designer?

It is difficult to control the subdialog element for the first time users.

6.4. Findings

Submit and goto which realize modularity are confined to only traverse the links in the same document but cannot traverse to another document when using with MaryTTS but working with FreeTTS. Although every component needs to be independent,   the VoiceXML interpreter didn't achieve full compatibility with MaryTTS and fixing the issue will hand over the full flexibility to VoiceXML designer.

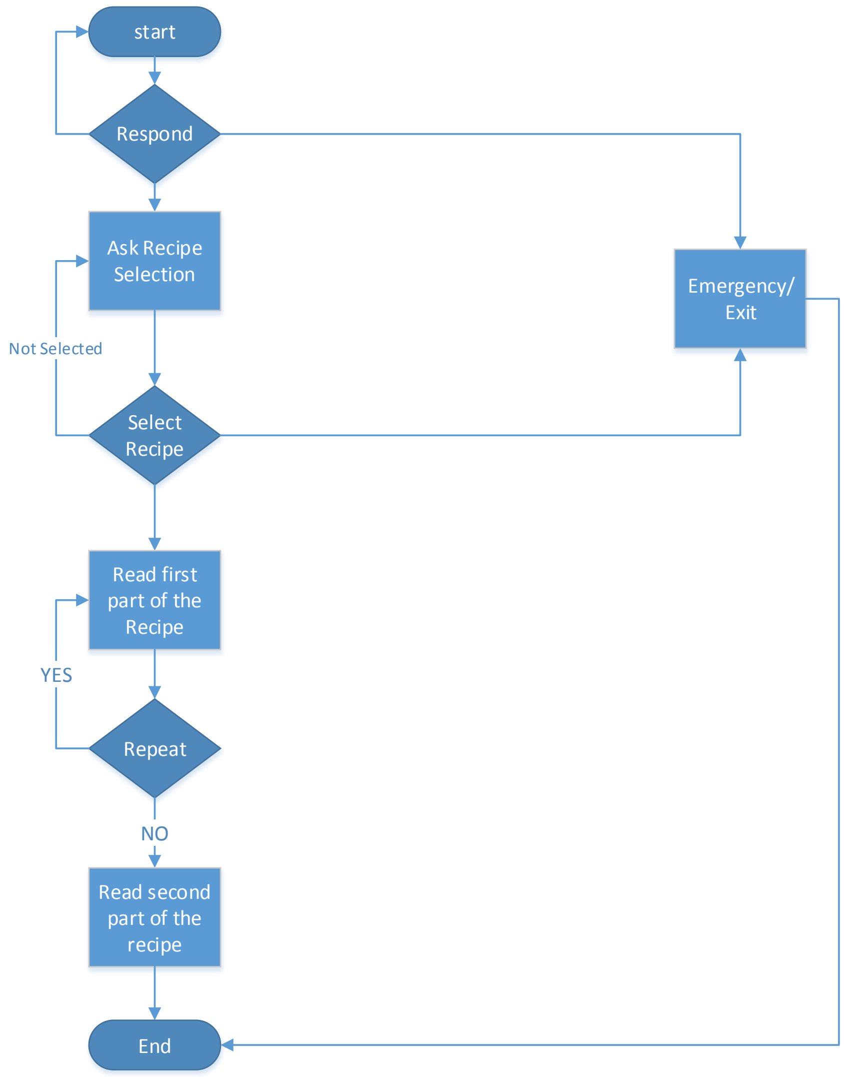
7. Implemented Scenario

7.1. With Animation

This is similar to the cooking scenario with animation. This scenario has only one VoiceXML document in which the flow is implemented.

Step 1: User is prompted with the welcome message.
Step 2: User says the word “Hungry”
Step 3: Machine prompts the list of the recipes available
Step 4: User selects one of the recipe by saying one, two or three.

If user says one, then the control jumps to the form id “potato”. Similarly, form Id chicken for two and Fish fry for three. Recipe is divided into two steps, so that machine can repeat the previous step if user doesn’t understand.



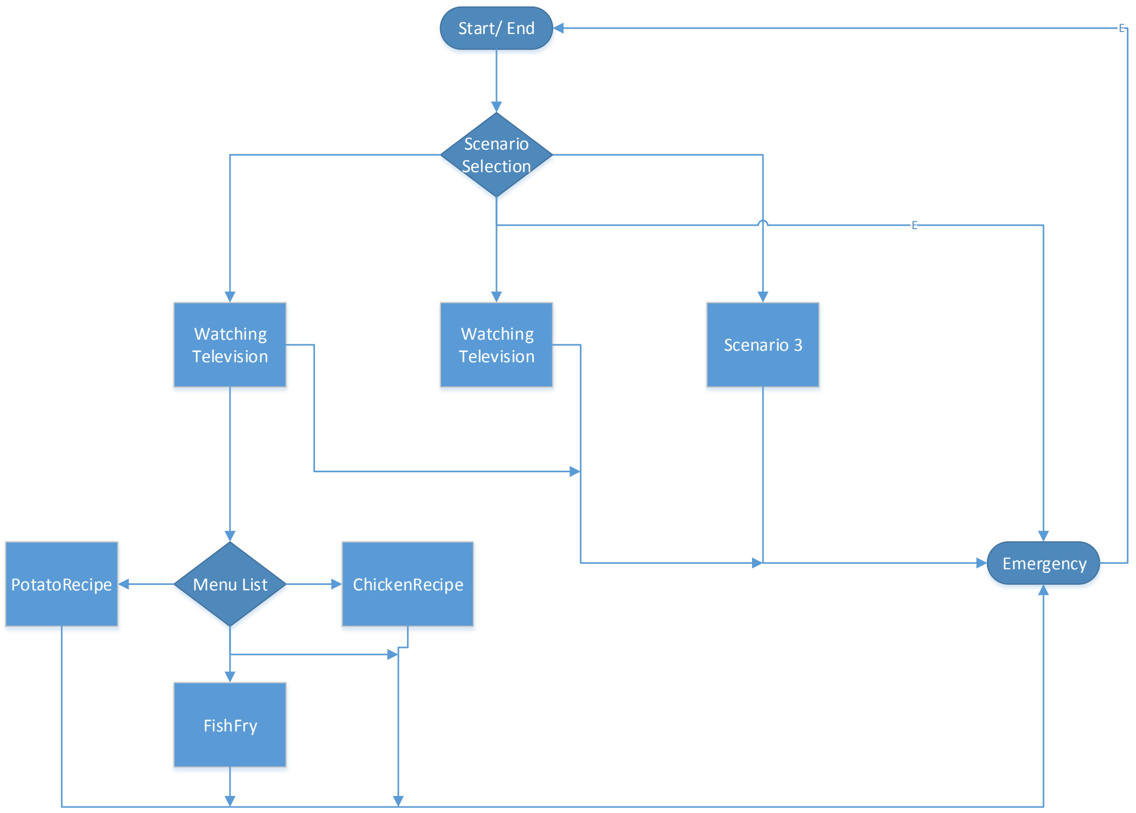
7.2. Without Animation

This is a complex scenario with multiple VoiceXML files which are called with respect to the user's input.

Start.vxml

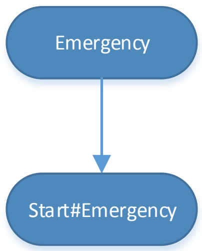
This document has the welcome message and is the starting point of the conversation. This document also contains the exit which is called from Emergency.vxml. There are many ways to end the flow but end from this document only happens when the control is in Emergency.vxml. These two tasks where implemented using form ids, start and end. There is no grammar file linked to this document as it doesn't expect any user input. The control immediately jumps to Scenario decision after the welcome message and exits when end form is called.

Overall Scenario Implemented:



Emergency.vxml

This is to exit any time when there is no input from the user. Emergency.vxml file is called when there is no input from the user for a certain period. Each vxml document has a link to emergency.vxml, which transports the control to start.vxml#Emergency. Instead of submit goto tag is used in all the vxml documents to reach the control to Emergency.vxml because with submit only one document can be linked to another i.e., linking more than one document to a document is not possible using the submit tag. By using goto the control can be moved to Emergency.vxml from more than one document also.

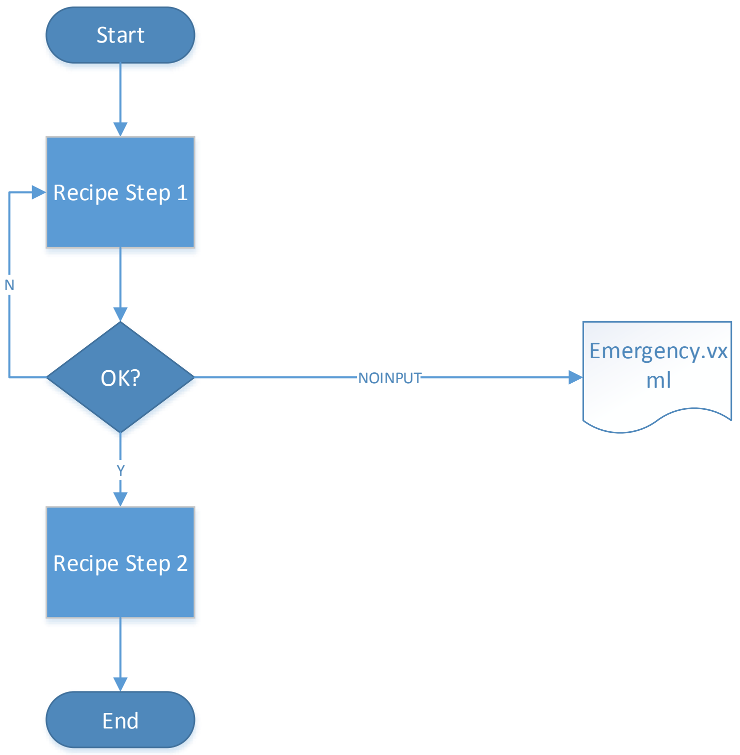


Menu.vxml

This document contains three items which machine reads out to the user. Grammar file is linked to this file as it expects the user input for calling the respective document. When user says other than one, two or three then the machine prompts "Please select the valid input" and the control flows again. Emergency.vxml can be called from this when there is no input from the user.

PotatoRecipe.vxml / ChickenRecipe.vxml / FishFry.vxml

These vxml documents contains the recipe which will be prompted by the machine. User interaction in between the recipe is made possible so that the previous step can be repeated again for better understanding. This concept is implemented by using if else conditions. If there is no input from the user then the control jumps to Emergency.vxml immediately.



Machine: What you like to have 1. Potato Soap 2. Chicken Curry 3. Fish Fry. Please select an option to have the recipe

User: One

Machine: In a large stock pot combine potatoes, onions, celery, bouillon cubes and enough water to cover all ingredients. Do you got this.

User: Yes

Machine: Bring to a boil and simmer on medium heat until potatoes are within 15 minutes of being finished. Add half and half, bacon, cream of mushroom soup and stir until creamy. Add cheese and stir until completely melted. Simmer on low until potatoes are done. Enjoy potato soup.

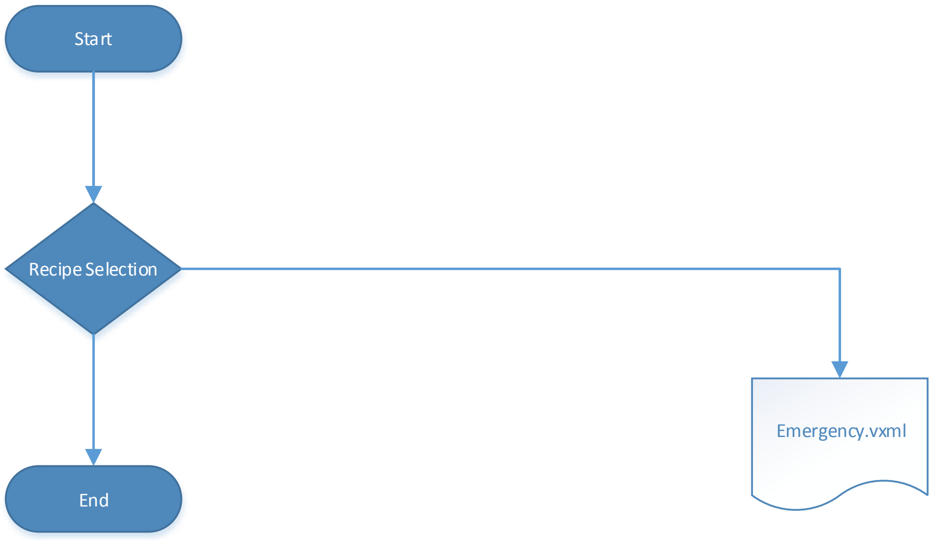
The above flowchart has 2 sub-scenarios implemented

1. Cooking
2. Watching Television

The flow starts with a Hello Message from the Machine. User can select any of the above mentioned scenarios from this point.

1. Cooking

This scenario will start when user says the word "Cooking" immediately after the welcome message is prompted by the machine. He will be prompted with list of recipes available. This is written in Menu.vxml, which also contains the code to take the decision on to navigate to the particular recipe. It is also possible that user can jump from this scenario to the Television scenario from Menu. For this user has to say the word "Television" when the machine prompts the list of recipes.

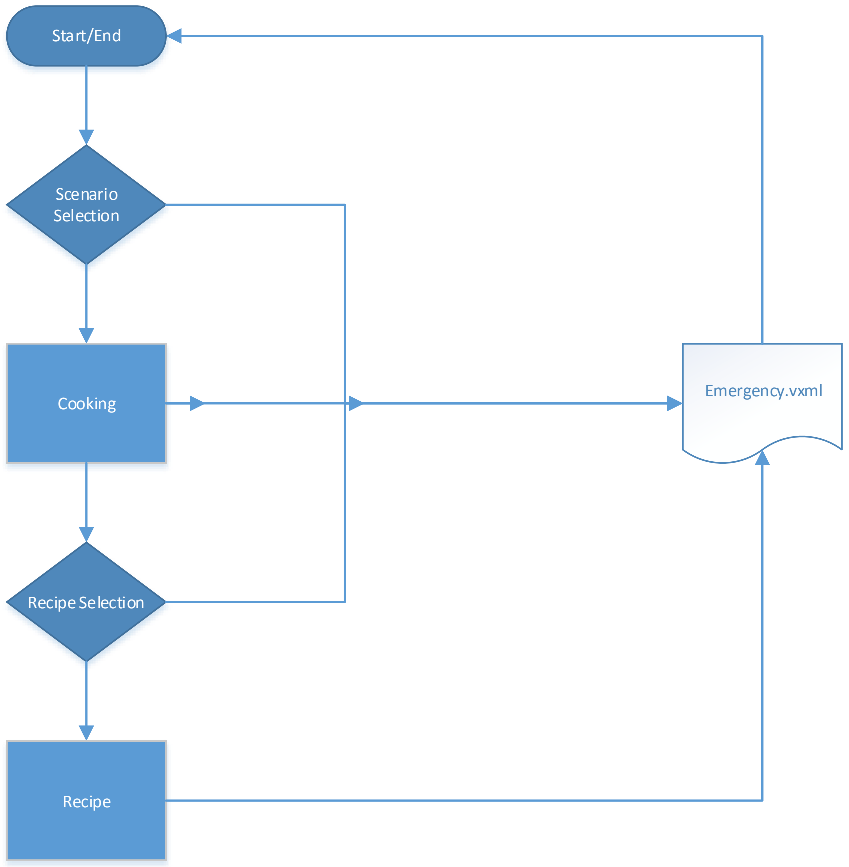


Selecting a recipe

When the user is prompted with the list of recipes, he can select the recipe by saying one, two or three. If users says one then the control jumps to PotatoRecipe.vxml, two to ChickenRecipe and three to FishFry.

Control flow within the recipe.

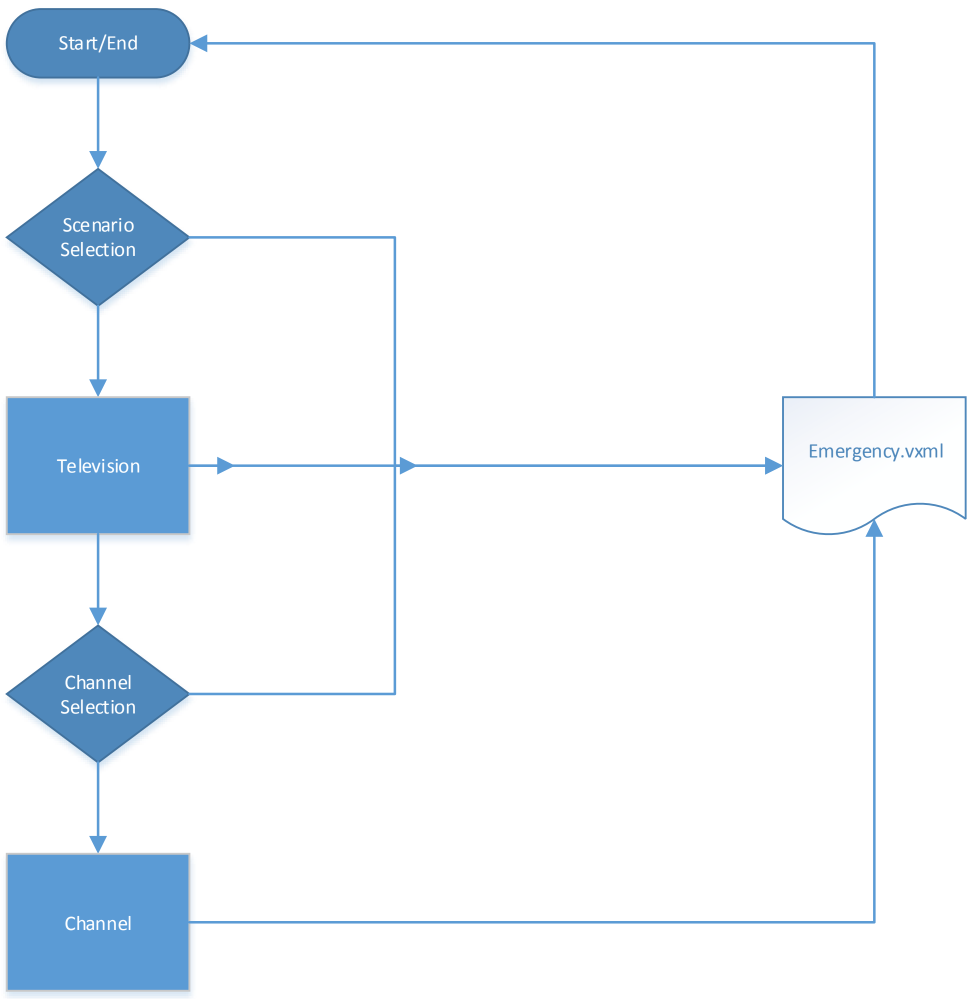
Each recipe is divided into 2 steps, so that user can ask for repeating the steps again which makes easier for understanding the recipe. This is implemented for all the three recipes.



2. Television

This scenario will start when user says the word “Television” immediately after the welcome message is prompted by the machine. Similar to Menu.vxml, this scenario has Channel.vxml, in which three options were implemented. Sprots.vxml will be called when user says One, Movies.vxml will be called for Two and TV Series will be called for the option Three.

The options for selecting for the options are mentioned the same to implement the concept of reusability of the Grammar.

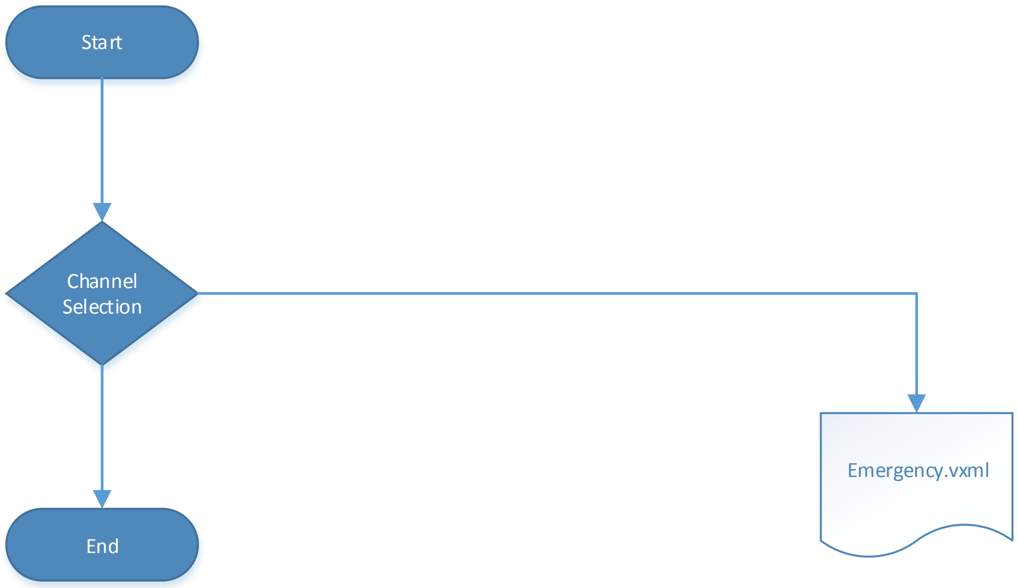


Channel.vxml

The control moves to this document from ScenarioSelection.vxml, when user says the word "TV". Machine will prompt three channel Sports, Movies and TV series. Then it waits for the user input, if the input is one then the control moves to Sports.vxml. Similarly, control moves to Movies.vxml for two and to TVSeries.vxml for three. If there is no input from the user then Emergency.vxml will be called. Grammar file is required for this document as it expects the user input for selecting one of the three options prompted.

Sports.vxml / Movies.vxml / TVSeries.vxml

Control moves to one of these documents when user says one, two or three immediately after list of channels are read out. For one the control moves to Sports.vxml, for two Movies.vxml and for three TVSeries.vxml. Grammar file is not required for these documents as there is no user input expected because these files only have a sample text which will be read out by the machine.



8. Conclusion

With the current implementation in the project, it is proved that dialogs can be made as modular with different VoiceXML documents and it is also possible to reuse the dialogs. In future, more complex scenarios can be implemented and also implementing this with animation. Unfortunately the current JVoiceXML interpreter may have some compatibility issues with MaryTTS speech engine and using this particular speech engine during our testing created undesirable behavior with some tags like Submit and GOTO. This behavior leads us to choose other text-to-speech engine FreeTTS for now.

9. Future Scope

Unlike VoiceXML 2.0/2.1, the focus in VoiceXML 3.0 is almost exclusively on the user interface portions of the language. By choice, very little work has gone into the development of data storage and manipulation or control flow capabilities. In short, VoiceXML 3.0 has been designed from the ground up as a presentation language, according to the definition presented in the Data Flow Presentation Framework.

10. References

1. http://mary.dfki.de/index.html
2. http://en.wikipedia.org/wiki/Speech_synthesis
3. http://cmusphinx.sourceforge.net/wiki/tutorialsphinx4
4. http://en.wikipedia.org/wiki/VoiceXML
5. http://en.wikipedia.org/wiki/Speech_Recognition_Grammar_Specification
6. http://www.techfak.uni-bielefeld.de/~skopp/download/BML.pdf
7. http://voice.cs.vt.edu/docs/prm_files/dataexchange.html
8. http://JVoiceXML.sourceforge.net/
9. http://www.w3.org/TR/voicexml30/

11.Appendix

SRGS

Speech Recognition Grammar Specification (SRGS) is a W3C standard for how speech recognition grammars are specified. A speech recognition grammar is a set of word patterns, and tells a speech recognition system what to expect a human to say. For instance, if you call an auto attendant application, it will prompt you for the name of a person (with the expectation that your call will be transferred to that person's phone). It will then start up a speech recognizer, giving it a speech recognition grammar. This grammar contains the names of the people in the auto attendant's directory and a collection of sentence patterns that are the typical responses from callers to the prompt.

VoiceXML Elements:

<form>	A Dialog for presenting information and collecting data.
<exit>	Exit a session
<goto>	Go to another dialog in the same or different document
<grammar>	Specify a speech recognition or DTMF grammar
<if>	Simple conditional logic
<noinput>	Catch a noinput event
<nomatch>	Catch a nomatch event
<prompt>	Queue speech synthesis and audio output to the user
<reprompt>	Play a field prompt when a field is re-visited after an event
<return>	Return from a subdialog
<submit>	Submit values to a document server
<vxml>	Top-level element in each VoiceXML document