# Deriving Vocal Interfaces from Logical Descriptions in Multi-device Authoring Environments

Fabio Paternò and Christian Sisti

CNR-ISTI, HIIS Laboratory, Via Moruzzi 1, 56124 Pisa, Italy
{Fabio.Paterno,Christian.Sisti}@isti.cnr.it

**Abstract.** Model-based approaches for interactive Web applications have neglected vocal interaction. However, ubiquitous multi-device environments call for better support for such modality. In this paper we present a language for logical descriptions of vocal interfaces along with a transformation for deriving corresponding implementations and show an example application. Such results have been integrated into a multi-device authoring environment.

**Keywords:** Model-based user interface design, Vocal interfaces, XML-based user interface languages.

## 1 Introduction

The convergence of telecommunications and the Web is now bringing the benefits of Web technology to the telephone, enabling Web developers to create applications that can be accessed via any telephone, and allowing people to interact with these applications via speech and keypads [16].

In this context the W3C has developed a suite called Speech Interface Framework in which one of the main contribution is VoiceXML 2.0, a language designed for developing vocal interfaces with support for audio dialogues, vocal and DTMF recognition, recording and telephony feature. For this reason we are witnessing the spread of Voice Browsers, which offer Web Based services from any phone.

In a number of contexts of use vocal interaction is important: when the visual channel is busy (e.g. while car driving), for disabled people (e.g. the vision-impaired), or while users are moving. Some possible scenarios are in accessing business (e.g. booking services, airline information, etc.), public (e.g. weather information, news) or personal information (e.g. appointment calendar, telephone list). The vocal features make it suitable to support quick access to information and to interact in a way more similar to that used for communication among humans. Thus, in modern technological ubiquitous settings, the need for supporting vocal interaction is acquiring increasing importance, and the associated technology has considerably improved in terms of efficiency and accuracy, even if some limitations still apply (e.g. performance in noisy environments).

Model-based approaches for interactive systems are characterized by the use of logical languages, which identify and classify user interface elements and ways to compose them according to their effects. They have stimulated interest in particular with the advent of multi-device user interfaces [7] because they allow designers to better manage the complexity of user interfaces that have to adapt to varying interaction resources. They allow designers to concentrate on logical decisions without having to deal with a plethora of low-level implementation details. However, most model-based approaches have mainly addressed issues related to desktop/mobile adaptation and have neglected other modalities such as vocal interaction. Thus, there is a need for solutions able to facilitate the development of multiple versions of an interactive application, including the vocal one. For this purpose, the ideal solution is to identify a core set of interaction concepts independent of the modality and then refine it for each possible modality in order to account for its specific aspects. For example, the vocal modality in general is linear, not persistent, quicker and more natural in some operations. This implies the need for continuous feedback, and the rendering of short prompts or option lists in order to limit memory efforts.

Stanciulescu [17] indicated some criticisms of the model-based approach in user interface design. Our framework has been developed taking into account such comments and aiming to minimize the negative aspects. One of the highlighted points is the *high threshold*, the designer needs to learn a new language before starting new interface development. To this end, we have created a graphical-based tool to assist designers in developing multiple versions depending on the target platform, which share a common set of abstract concepts, thus reducing the learning effort when they start a new version.

Another problem in model-based approaches may be the *low ceiling*, the user interfaces that can be generated have various limitations due to the excessively abstract design. We address this criticism by proposing a model-based solution using two levels of abstraction, in which the lower level (that refines the higher one) permits good control over the resulting interface without having to know the details of the implementation language. The *unpredictability* of some final results is avoided by furnishing complete documentation of the transformation rules.

On the other hand, model-based approaches offer some advantages in terms of *methodology* (user -centred approach), *reusability* (the various concrete languages are refinements of the same abstract language) and *consistency* (between early design phase and final result).

In the paper after discussing some related work we introduce the proposed approach and present the logical language for vocal interaction, then we describe the transformation from the logical language to VoiceXML, and we show an example application in the museum domain. Lastly, some conclusions are drawn along with indications for future work.

## 2 Related Work

The problem of designing multi-device interfaces, including vocal ones, has been addresses in some previous work but still needs more general and better engineered solutions. Damask [6] includes the concept of layers to support the development of cross-device (desktop, smartphone, voice) user interfaces. Thus, the designers can specify user interface elements that should belong to all the user interface versions and elements that should be used only with one device type. However, this approach can be useful in developing desktop and mobile versions but does not provide particularly useful support when considering the vocal version, which requires user interface structures profoundly different from the graphical versions. XFormsMM [4] is an attempt to extend XForms in order to derive both graphical and vocal interfaces. In this case the basic idea is to specify the abstract controls with XForms elements and then use aural and visual CSS for vocal and graphical rendering, respectively. The problem in this case is that aural CSS have limited possibilities in terms of vocal interaction and the solution proposed requires a specific ad hoc environment in order to work. For this purpose we propose a more general solution able to derive implementations in the W3C standard Voice XML. Obrenovic et al. [9] have investigated the use of conceptual models expressed in UML in order to then derive graphical, form-based interfaces for desktop or mobile devices or vocal ones. UML is a software engineering standard mainly developed for designing the internal software of application functionalities. Thus, it seems unsuitable to capture the specific characteristics of user interfaces and their software. In [11] there is a proposal to derive multimodal user interfaces using attribute graph grammars, which have a well-defined semantics but limitations in terms of performance. The possibility of deriving vocal interfaces was addressed in [1] but using hardcoded solutions for the transformation and logical descriptions that were unable to describe typical Web2.0 interactions and access to Web services.

A different approach to multimodal user interface development has been proposed in [5], which aims to provide a workbench for prototyping them using off-the-shelf heterogeneous components. In that approach, model-based descriptions are not used and it is necessary to have an available set of previously defined components able to communicate through low-level interfaces, thus making it possible for a graphical editor to easily compose them.

To summarise, we can say that the few research proposals that have also considered vocal interaction have not been able to obtain a general solution in terms of logical descriptions and provide limited support in terms of generation of the corresponding user interface implementations. For example, in [1] the transformations were hard-coded in the Java implementation, while in [11] the transformations were specified using attributed graph grammars, whose semantics is formally defined but have considerable performance limitations.

## 3 The Proposed Approach to Vocal Interaction

In this paper we present a general logical language for vocal interaction, which is included in an overall environment able to support development of multi-device user interfaces. The associated authoring environment includes a transformation tool able to derive VoiceXML implementations from the logical specifications and satisfies the requirements for multimodal interface generation discussed in previous work [8], such as modality independence, support for specifying hierarchical grouping, etc.

One of our goals is to propose a framework that allows designers to generate vocal interfaces that take into account the challenges and principles for conversational interfaces design identified in [15]. One of these challenges is to *make the interaction feel conversational*. Our approach, in this sense, allows designers to set a number of synthesizing properties (e.g. prosody, volume, tone, speed, etc). Moreover, we support the barge-in technique that allows users to interrupt the speech synthesizer by using their voice. Another point is the problem of recognizing the start/stop of subdialogues corresponding to the grouping. We propose different solutions to communicate this information to the user, such as inserting simple delimiting sounds or prompting short meaningful phrases (e.g. "Main menu").

Another challenge is error recognition: "*One can never be completely sure that the recognizer has understood completely*" [15]. The errors are divided into three categories: rejection errors, occur when the recognizer does not match the user input with any expected utterance; substitution errors, when the platform erroneously recognizes a wrong but legal input; insertion error, if the recognizer accepts a noise as a valid input. Our framework allows different mechanisms to avoid these errors. In the case of rejection errors, a simple solution is to answer with a "*not understand*" feedback. This approach could stimulate user frustration and so, as suggested in [15], we provide designers with the possibility of implementing the *tapered prompting* technique. In this way the error messages that the user receives become more explicit as the number of errors increase.

To reduce the occurrence of the substitution errors, we allow the designers to verify the utterance when necessary. Lastly, the problem of insertion errors is attenuated by including in the generated vocal interfaces two vocal commands to start and stop the dialogue with the platform. In this way, the user can temporarily interrupt the interaction and restart it at his convenience.

## 4 A Logical Language for Vocal Interaction

MARIA is a recent model-based language, which allows designers to specify abstract and concrete user interface languages according to the CAMELEON Reference framework [2] (Fig. 1 shows an instance of the framework). This language represents a step forward in this area because it provides abstractions also for describing modern Web 2.0 dynamic user interfaces and Web service accesses. In its first version it provides an abstract language independent of the interaction modalities and concrete languages for graphical desktop and mobile platforms [10]. In general, concrete languages are dependent on the typical interaction resources of the target platform but independent of the implementation languages. In this paper we present a concrete language for vocal interfaces, which has been designed within the MARIA framework.

In MARIA an abstract user interface is composed of one or multiple presentations, a data model, and a set of external functions. Each presentation contains: a number of user interface elements (*interactors*) and interactor compositions (indicating how to group or relate a set of interactors); a dialogue model, describing the dynamic behaviour of such elements and connections, indicating when a change of presentation should occur. The interactors are first classified in abstract terms: edit, selection, output and control. Each interactor can be associated with a number of event handlers, which can change properties of other interactors or activate external functions.

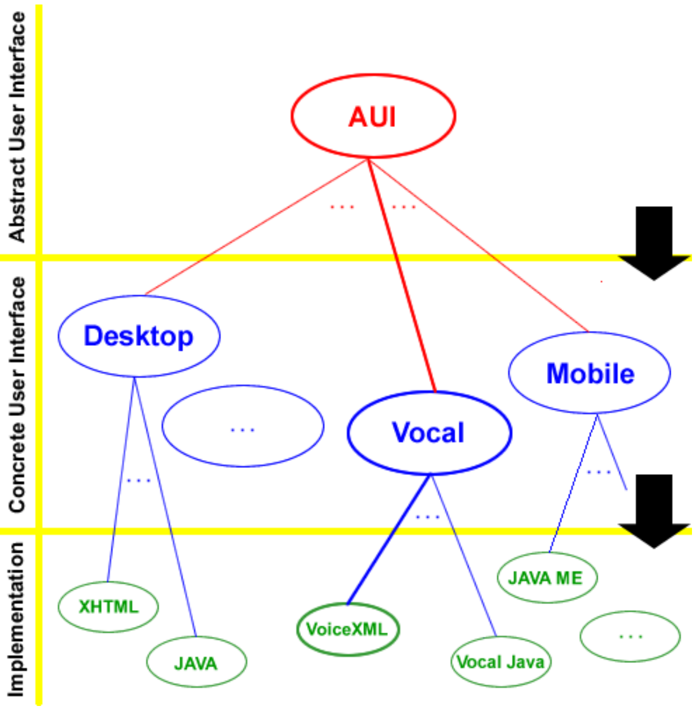


**Fig. 1.** Possible abstraction levels

While in graphical interfaces the concept of presentation can be easily defined as a set of user interface elements perceivable at a given time (e.g. a page in the Web context), in the case of vocal interfaces we consider a presentation as a set of communications between the vocal device and the user that can be considered as a logical unit, e.g. a dialogue supporting the collection of information regarding a user.

In defining the vocal language we have refined the abstract vocabulary for this platform. This mainly means that we have defined vocal refinements for the elements specified in the abstract language: interactors (user interface elements), the associated events and their compositions.

The refinement involves defining some elements that enable setting some presentation properties. In particular, we can define the default properties of the synthesized voice (e.g. volume, tone), the speech recognizer (e.g. sensitivity, accuracy level) and the DTMF (Dual-Tone Multi-Frequency) recognizer (e.g. terminating DTMF char).

Only-output interactors simply provide output to the user; the abstract interface classifies them into text, description, feedback and alarm output. Refinement of the text element is composed of two new elements: *speech* and *pre-recorded message*. Speech defines text that the vocal platform must synthesize or the path where the platform can find the text resources. It is furthermore possible to set a number of voice properties, such as emphasis, pitch, rate, and volume as well as age and gender of the synthesized voice. Moreover, we have introduced control of behaviour in the event of unexpected user input: by suitably setting the element named *barge in*, we

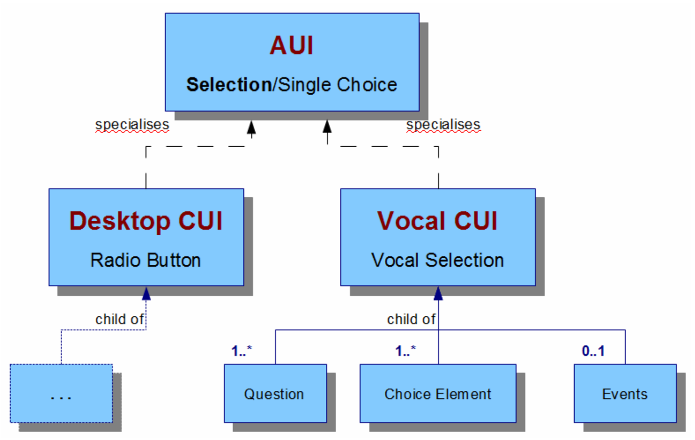


**Fig. 2.** Specialization example

can decide if the user can stop the synthesis or if the application should ignore the event and continue. As mentioned above, the other element that refines abstract text is pre-recorded message: it defines the path of pre-defined audio resources that must be played. We support the case of missing resources by defining an *alternative content* that can be synthesized when this case occurs. Besides speech and pre-recorded message, the other only-output element is *sound.* This element permits defining the path of a non-vocal audio resource that must be played by the platform. It is also possible to insert a textual description of the sound that could be used as additional information regarding the sound content.

Selection interactors permit performing a selection between a set of elements; the abstract interface distinguishes between *single* and *multiple choice*. In order to support such interactions in vocal context we have introduced the interactor *vocal selection* (see Fig.2). This element defines the question(s) to direct to the user and the set of possible user input that the platform can accept. In particular, it is possible to define textual input (word or sentences) or DTMF input. Depending on the type of selection one or multiple elements can be selected. Semantic differences between this two interactors are made at abstract user interfaces level: for the single choice it is possible to set only one selected element instead, in the case of the multiple choice, more than one user input can be set as selected elements.

The control interactor at the abstract level can be distinguished in activator, to activate a functionality, and navigator, to manage navigation between the presentations. The activator is refined in the vocal language into: *command*, in order to execute a script, *submits*, to send a set of data to a server, and *goto* to perform a call to a script that triggers an immediate redirection. While the navigator is refined into: *goto*, for automatic change of presentation; *link*, for user-triggered change of presentation, and *menu* for supporting the possibility of multiple target presentations.

Edit interactors gather more complex user input. We refined three abstract elements: text edit, numerical edit and object edit. Text edit, from the graphical web context point of view, can be regarded as an editable textual field. In the vocal context

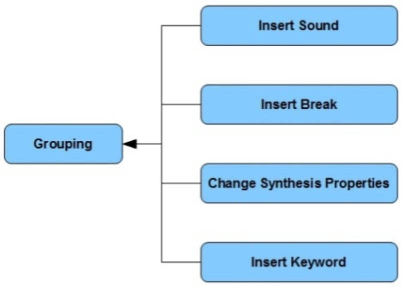


**Fig. 3.** Grouping refinement

we refined this concept with the *vocal textual input* element, which permits setting a vocal request and specifying the path of an external grammar for the platform recognition of the user input. Numerical edit is an interactor to collect numbers, which are refined into *numerical* input. As in the textual input it is possible to define a request and an external grammar. Moreover, it is also possible to set a predefined grammar, such as date, digits, phone or currency. Finally, we have refined the object edit interactor into a *record* element, which allows specifying a request and storing the user input as an audio resources. It is possible to define a number of attributes relative to the recording, such as *beep* to emit a sound just before recording, *maxtime* to set the maximum duration of the recording, and *finalsilence*, to set the interval of silence that indicates the end of vocal input. Record elements can be used for example when the user input cannot be recognised by a grammar (e.g. a sound).

In the logical language one of the elements that permit the composition of the interactors is *grouping*. From the visual point of view we could refine this concept, for example, into a table element. Group vocal interactor is more complex. We propose four solutions to permit the user to identify the beginning and the end of a grouping (see Fig. 3). Inserting a sound at the beginning and at the end for this purpose can be a good non-invasive solution. Another solution can be inserting a pause, which must be neither too short (useless) nor too long (slow system feedback). Moreover, it is possible to change the synthesis properties such as volume and voice gender. The last possibility is to insert keywords that explicitly define the start and the end of the grouping.

Another substantial difference of vocal interfaces is in the event model. While in the case of graphical interfaces the events are related mainly to mouse and keyboard activities, in vocal interfaces we have to consider different types of events: *noinput* (the user has to enter a vocal input but nothing is provided within a defined amount of time), *nomatch*, the input provided does not match any possible acceptable input, and *help*, when the user asks for support (in any platform specific way) in order to continue the session. All of them have two attributes: message, indicating what message should be rendered when the event occurs, and re-prompt, to indicate whether or not to synthesize the last communication again.

In order to facilitate authoring and editing of logical specifications a graphical editor has been designed and implemented (Fig. 4). It allows editing the various

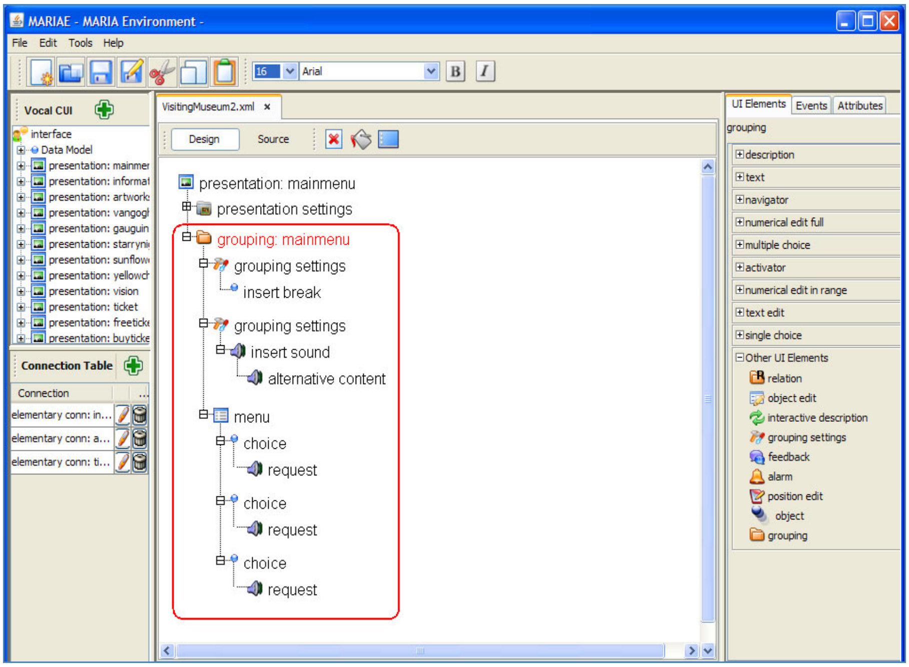


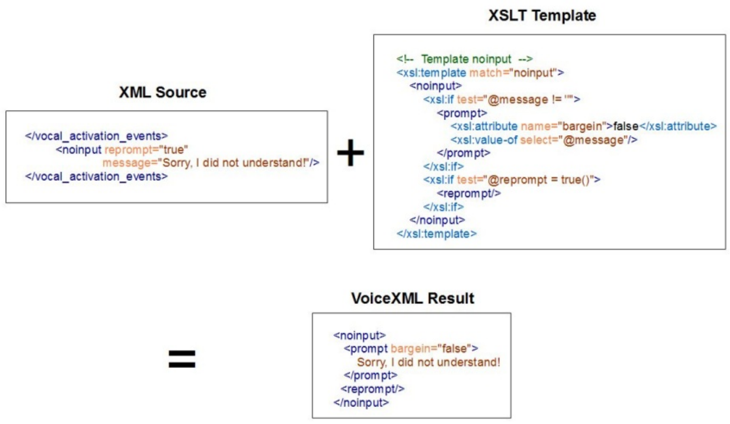
**Fig. 4.** The graphical editor for the vocal logical language

presentations in the central area. The user interface elements are created by drag-and-drop from the lists in the right frame, which show the elements that can be inserted according to the language specification. In the middle tab in the right frame it is also possible to specify the associated events and attributes. In the left side there is an interactive nested tree view of the presentations and the associated elements. The output of the tool is a logical description of the interface formalized in XML.

## 5 Transforming Logical Description into VoiceXML Implementations

The current standard for voice browsing implementation is VoiceXML [13]. Thus, this was the first target implementation language from the vocal logical description. Since VoiceXML is also an XML-based language, XSL Transformations (XSLT) [14] seemed the most appropriate technology for implementing such transformation. This language provides a number of constructs for creating mappings among elements of two XML-based languages. However, such mappings are not trivial to create because both languages have a structure that provides constraints about where to locate an element. For example, in VoiceXML a vocal output is implemented differently depending on whether it occurs in a form or in a menu. This has been solved using the “XSLT modes”, an XSLT technique to identify which template to use in the transformation when this element occurs. More generally, this mechanism allows the transformation to change template to apply in the mapping depending on the current context.

For the sake of clarity we show a simple example of application of an XSLT template in Figure 5. The XML source code is a simple excerpt of our logical language in which we describe the noinput event. Every time that the *XSLT Engine* finds a match with a noinput source tag the suitable template is called. In this case the template adds the suitable VoiceXML <noinput> tag and then tests if it was set up a message to synthesize. If true a <prompt> VoiceXML tag is added. Note that the attribute *bargein* of the VoiceXML code is forced to be false to prevent interruption of the system communication by any user input. Each presentation is associated with a VoiceXML document. The presentation element has a list of possible default settings. Some of them can be defined once for the entire presentation (e.g. speech recognizer properties); in this case they are opportunely mapped into a <property> VoiceXML. In other cases the properties can be set up multiple times in the presentation (e.g. voice synthesis properties). The policy for defining such properties follows the structure of the specification (bottom-up): first it checks whether local properties have been defined, if not the properties of the surrounding grouping operator, if any, are applied, and lastly, if even these are missing, the properties general to the entire presentation are used.



```
</vocal_activation_events>
        <noinput reprompt="true"
                 message="Sorry, I did not understand!"/>
</vocal_activation_events>
```

```
<!-- Template noinput -->
<xsl:template match="noinput">
  <noinput>
    <xsl:if test="@message != ''">
      <prompt>
        <xsl:attribute name="bargein">false</xsl:attribute>
        <xsl:value-of select="@message"/>
      </prompt>
    </xsl:if>
    <xsl:if test="@reprompt = true()">
      <reprompt/>
    </xsl:if>
  </noinput>
</xsl:template>
```

```
<noinput>
  <prompt bargein="false">
    Sorry, I did not understand!
  </prompt>
  <reprompt/>
</noinput>
```

**Fig. 5.** A small example of XSLT Transformation

SGRS[1] grammars format has been used to specify possible inputs, since that it is what the VoiceXML specification supports. The grammars define the set of possible input that the vocal platform is able to recognize. In the generated implementations we use predefined VoiceXML grammars, when possible, such as date, number and currency in the *numerical edit* element mapping.

In some cases we generate *inline grammars*, which are grammars defined in the transformation process and directly inserted in the VoiceXML code. This is the case of the *link* element that can be activated both by vocal or DTMF command thanks to two expressly created grammars (using the parameters defined in the logical description).

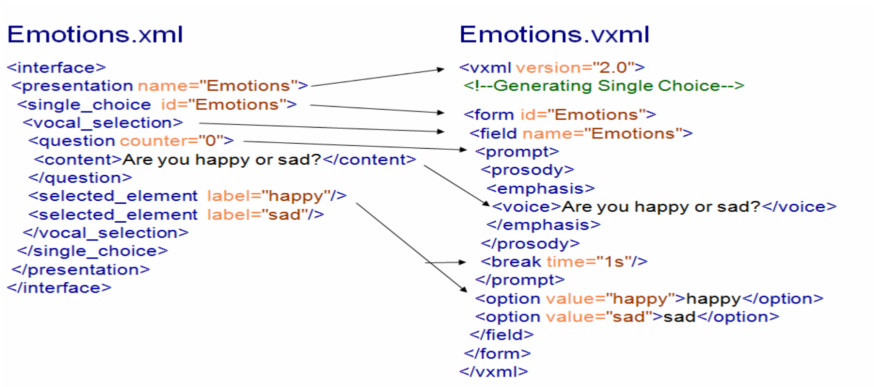
[1] Speech Recognition Grammar Specification Version 1.0, `http://www.w3.org/TR/speech-grammar/`

Another solution is used in the mapping of the *single vocal selection*: in this case, we know 'a priori' the list of possible inputs, and we can use one VoiceXML <*option*> element for each input. This is equivalent to using a grammar able to recognize the entire list of user input. Figure 5 shows an example of a logical description with a single choice and how its elements are mapped onto VoiceXML elements.

The case of the *multiple vocal selection,* instead, introduces a problem: we do not know how many choices (in the predefined list) the user will make. In our solution the platform asks all the choices one-by-one and the user must answer yes/no to accept/reject each one. This solution may seem verbose but in practise there may be two situations: few choices, in this case the verbosity is negligible; many choices, in this case the verbosity is an advantage because it will reduce the mnemonic effort of the user.

As mentioned in previous sections, *textual edit* is similar to the visual concept of editable text box. It is very hard (if not impossible) to have a grammar able to recognize every kind of user input. We prefer to leave it up to the application developer to implement an *external-grammar* (or to find a pre-built one) that satisfies the possible input (case by case). For example, suppose that the platform asks the name of the user, the developer should build a grammar containing a dictionary of names to provide for recognition.

In the visual context we have some mechanisms to force the input of numbers in a certain range (e.g. a spin box); from the vocal point of view we resolve this problem by carrying out a check, before accepting the user input, using the conditional VoiceXML tags. If the number specified by the user is out of range, we refuse the input and re-prompt the request.


Emotions.xml

```
<interface>
 <presentation name="Emotions">
  <single_choice id="Emotions">
   <vocal_selection>
    <question counter="0">
     <content>Are you happy or sad?</content>
    </question>
    <selected_element label="happy"/>
    <selected_element label="sad"/>
   </vocal_selection>
  </single_choice>
 </presentation>
</interface>
```

Emotions.vxml

```
<vxml version="2.0">
 <!--Generating Single Choice-->
 <form id="Emotions">
  <field name="Emotions">
   <prompt>
    <prosody>
     <emphasis>
      <voice>Are you happy or sad?</voice>
     </emphasis>
    </prosody>
    <break time="1s"/>
   </prompt>
   <option value="happy">happy</option>
   <option value="sad">sad</option>
  </field>
 </form>
</vxml>
```

**Fig. 6.** Example of Logical-to-Implementation transformation

The control interactor command is mapped into a VoiceXML variable that contains the results of the execution of a script that must be defined at presentation level (which corresponds to a VoiceXML document in the implementation). The others control interactors: submit, goto, link, and menu are mapped into corresponding VoiceXML elements. VoiceXML links must be declared externally to the dialogue constructs and thus they are globally activable. Each possible menu choice is transformed into a <choice> VoiceXML element. In this way the grammar for recognizing the user input is automatically provided by the VoiceXML browser.

In the case of textual input the developer has the possibility to specify an appropriate grammar containing the rules for the acceptable inputs. Numerical input can be recognised by predefined grammars. It is also possible to generate in the code the check whether the input satisfies a given range.

## 6 Application

The VoiceXML code generated by the above transformation has been tested with Tellme.Studio Voice Browser [12] (suggested by W3C) and has passed the validation test integrated in it. The applications have then been used through VoIP access to the vocal server.

Figure 7 shows the structure of some parts of an example application created with our environment. It consists of a virtual museum application. The rectangles represent the presentations. Each of them is a dialogue or a set of dialogues that can be logically grouped together. In general, each presentation supports three basic possibilities: to be executed again, to go to the main menu, and to go to the previous presentation.

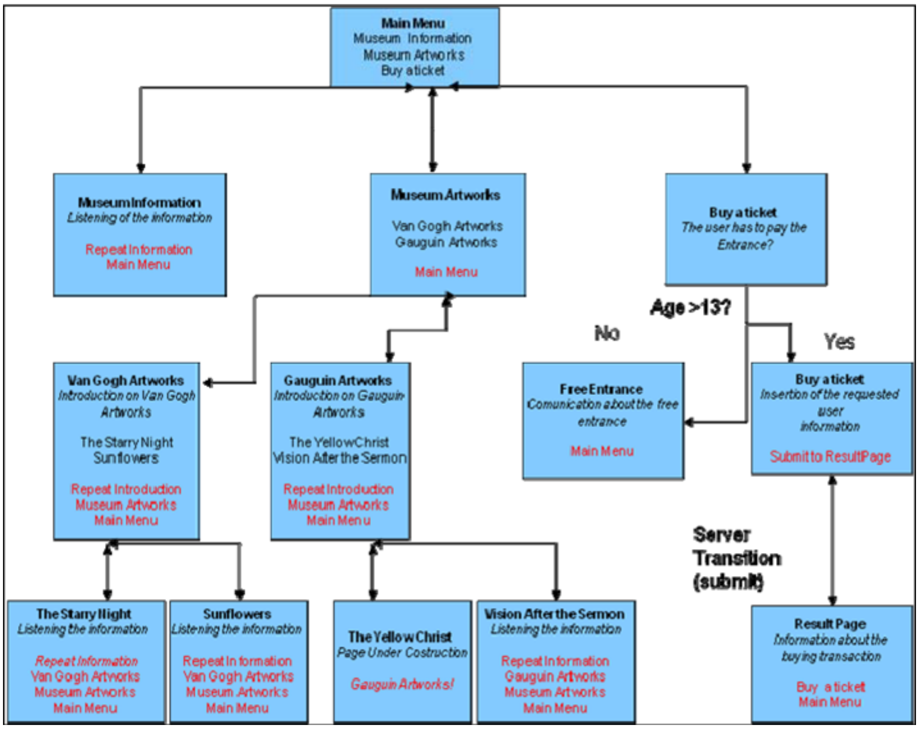


**Fig. 7.** An Example Application

The main menu allows the user to choose among three options: accessing more detailed information regarding the museum, performing a virtual visit, and buying a museum ticket. In the first case the user can just listen to the detailed information and then return to the main menu. In the second case the user can select an artist and then receive a related general description and then the possibility of accessing the associated artworks. When buying a ticket there are two possibilities depending on the age. In the case of under 13 the application activates a presentation communicating that the entrance is free otherwise the user has to provide personal information and perform the payment. More in details the system ask: *user name*, described logically with a text edit element with associated an external grammar that contain a dictionary of names; *user surname*, described in the same way of the previous one; *credit card type*, described by a single choice with three possible options (Visa, Mastercard, PostePay); *credit card number*, described by a numerical edit with associated a predefined grammar (number); *favoured museum's rooms*, described by a multiple choice with a number of possible options and *further notice*, described by a record element to permit at the user to leave a personal comment.

In the table below we show an example of resulting dialogue in order to better understand the interaction user-platform that we have obtained. We focus in this example only on the module related to the inserting of the user information.

| | **Input / Output** | **Logical Element** |
|---|---|---|
| **System:** | To buy a ticket you have to tell me some information. | Only Output |
| **System:** | What is your name? | Text Edit |
| **User:** | . . . **(do not respond)** | |
| **System:** | Sorry, I did't hear you. What is your name? | No input event |
| **User:** | Henry **(with noise)** | |
| **System:** | Sorry, I do not understand | No match event |
| **User:** | Henry | |
| **System:** | What is your surname? | Text Edit |
| **User:** | Smith | |
| **System:** | Kind of credit card between: Visa, Mastercard and PostePay? | Single Choice |
| **User:** | PostePay | |
| **System:** | Credit card number? | Numerical Edit |
| **User:** | 123456789# | |
| **System:** | We are interest to know which room you prefer to visit. Are you interested in Van Gogh's room? | Multiple Choice |
| **User:** | Yes | |
| **System:** | Are you interested also in Leonardo's room? | |

| | | |
|---|---|---|
| **User:** | Yes | |
| **System:** | After the *beep* leave a comment about this services | Record |
| **System:** | Beep | |
| **User:** | . . . **(some second silent length)** | |
| **System:** | Thanks. Do you want confirm you reservation? | Only Output |
| **User:** | Yes | |
| **System:** | | Submit |
| **System:** | Your request has been registered. | Only Output |

## 7 Conclusions and Future Work

This work introduces a novel logical language for vocal interfaces and the associated environment, which allows designers to easily compose vocal interfaces and derive VoiceXML implementations. This has been integrated in an environment for multi-device interface design and development, thus facilitating the implementation of multiple versions adapted to the various target modalities because of the use of a common abstract vocabulary, which is then refined according to the target platforms. This avoids requiring developers to learn a plethora of details of the many possible implementation languages

This result has been validated through the development of a number of vocal applications (one of them is briefly described in the paper), which have been rendered through publicly available voice browsers.

Future work will be dedicated to empirical tests in order to better assess how the development process is facilitated with this approach, especially when multi-device interfaces should be developed (e.g. desktop, mobile and vocal versions of the same application). We also plan to develop an automatic system able to support graphical-to-vocal adaptation and a new logical language for multimodal interfaces able to exploit the language presented in this paper for the vocal part.

## Acknowledgments

We gratefully acknowledge support from the EU ServFace Project (http://www.servface.eu).

## References

1. Berti, S., Paternò, F.: Model-Based Design of Speech Interfaces. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 231–244. Springer, Heidelberg (2003)
2. Calvary, G., Coutaz, J., Bouillon, L., Florins, M., Limbourg, O., Marucci, L., Paternò, F., Santoro, C., Souchon, N., Thevenin, D., Vanderdonckt, J.: The CAMELEON reference framework. CAMELEON project, Deliverable 1.1 (2002)

3. Edwards, A., Pitt, I.: Design of Speech-Based devices. Springer, Heidelberg (2007)
4. Honkala, M., Pohja, M.: Multimodal interaction with XForms. In: Proceedings ICWE, pp. 201–208 (2006)
5. Lawson, J., Al-Akkad, A., Vanderdonckt, J., Macq, B.: An open source workbench for prototyping multimodal interactions based on off-the-shelf heterogeneous components. In: Proceedings ACM EICS, pp. 245–254 (2009)
6. Lin, J., Landay, J.A.: Employing Patterns and Layers for Early-Stage Design and Prototyping of Cross-Device User Interfaces. In: Proc. CHI, pp. 1313–1322 (2008)
7. Myers, B.A., Hudson, S.E., Pausch, R.: Past, Present and Future of User Interface Software tools. ACM Trans. Comput. Hum. Interact. 7, 3–28 (2000)
8. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., Pignol, M.: Generating remote control interfaces for complex appliances. In: Proceedings ACM UIST'02, pp. 161–170 (2002)
9. Obrenovic, Z., Starcevic, D., Selic, B.: A Model-Driven Approach to Content Repurposing. IEEE Mutimedia, 62–71 (Januray-March 2004)
10. Paternò, F., Santoro, C., Spano, L.D.: MARIA: A Universal Language for Service-Oriented Applications in Ubiquitous Environment. ACM Transactions on Computer-Human Interaction 16(4), 1–30 (2009)
11. Stanciulescu, A., Limbourg, Q., Vanderdonckt, J., Michotte, B., Montero, F.: A Transformational Approach for Multimodal Web User Interfaces based on UsiXML. In: Proc. ICMI, pp. 259–266 (2005)
12. Tellme.Studio Voice Browser, `https://studio.tellme.com/`
13. Voice extensible markup language (VoiceXML) version 2.0, `http://www.w3.org/TR/2009/REC-voicexml20-20090303/7`
14. XSL Transformations (XSLT) Version 2.0, `http://www.w3.org/TR/xslt20/`
15. Yankelovich, N., Levow, G., Marx, M.: Designing SpeechActs: Issues in Speech User Interfaces. In: CHI 1995, pp. 369–376 (1995)
16. Voice Browser Activity (W3C), `http://www.w3.org/Voice/`
17. Stanciulescu, A.: A Methodology for Developing Multimodal User Interfaces of Information Systems. Ph.D Thesis, University of Louvain (2008)