Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Speech Communication 48 (2006) 863–887

SPEECH COMMUNICATION

www.elsevier.com/locate/specom

An advanced platform to speed up the design of multilingual dialog applications for multiple modalities ☆

Luis Fernando D'Haro [a], Ricardo de Córdoba [a,*], Javier Ferreiros [a], Stefan W. Hamerich [b], Volker Schless [b], Basilis Kladis [c], Volker Schubert [b], Otilia Kocsis [c], Stefan Igel [d], José M. Pardo [a]

[a] *Grupo de Tecnología del Habla, Universidad Politécnica de Madrid, Ciudad Universitaria s/n, 28040 Madrid, Spain*
[b] *Harman/Becker Automotive Systems, Ulm, Germany*
[c] *Knowledge S.A. (LogicDIS group), Patras, Greece*
[d] *Forschungsinstitut für anwendungsorientierte, Wissensverarbeitung (FAW), Ulm, Germany*

Received 16 February 2005; received in revised form 30 September 2005; accepted 2 November 2005

Abstract

In this paper, we present a complete platform for the semiautomatic and simultaneous generation of human–machine dialog applications in two different and separate modalities (Voice and Web) and several languages to provide services oriented to obtaining or modifying the information from a database (data-centered). Given that one of the main objectives of the platform is to unify the application design process regardless of its modality or language and then to complete it with the specific details of each one, the design process begins with a general description of the application, the data model, the database access functions, and a generic finite state diagram consisting of the application flow. With this information, the actions to be carried out in each state of the dialog are defined. Then, the specific characteristics of each modality and language (grammars, prompts, presentation aspects, user levels, etc.) are specified in later assistants. Finally, the scripts that execute the application in the real-time system are automatically generated.

We describe each assistant in detail, emphasizing the methodologies followed to ease the design process, especially in its critical aspects. We also describe different strategies and characteristics that we have applied to provide portability, robustness, adaptability and high performance to the platform. We also address important issues in dialog applications such as mixed initiative and over-answering, confirmation handling or providing long lists of information to the user. Finally, the results obtained in a subjective evaluation with different designers and in the creation of two full applications that confirm the usability, flexibility and standardization of the platform, and provide new research directions.
© 2005 Elsevier B.V. All rights reserved.

☆ This work was partly supported by the European Commission's Information Society Technologies Programme under contract no. IST-2001-32343. The authors are solely responsible for the contents of this publication.

* Corresponding author. Tel.: +34 91 3367366x4209; fax: +34 91 3367323.
E-mail addresses: lfdharo@die.upm.es (L.F. D'Haro), cordoba@die.upm.es (R. de Córdoba), jfl@die.upm.es (J. Ferreiros), shamerich@harmanbecker.com (S.W. Hamerich), vschless@harmanbecker.com (V. Schless), bkladis@knowledge-speech.gr (B. Kladis), vschubert@harmanbecker.com (V. Schubert), okocsis@knowledge-speech.gr (O. Kocsis), sigel@faw.uni-ulm.de (S. Igel), pardo@die.upm.es (J.M. Pardo).

0167-6393/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.specom.2005.11.001

Keywords: Automatic dialog systems generation; Dialog management tools; Multiple modalities; Multilinguality; XML; VoiceXML

1. Introduction

1.1. Motivation

The growing interest from companies in using new information technologies as means to getting closer to the final users has led to the quick growth and improvement of automatic dialog systems. They are now able to provide services such as reservations (San Segundo et al., 2001; Lamel et al., 2000; Levin et al., 2000), customer care (Strik et al., 1997) or information retrieval (Zue et al., 2000; Seneff and Polifroni, 2000), 24 h a day 7 days a week. Besides, given the different characteristics and requisites of the final users, the service is expected to be available in several languages (Turunen et al., 2004; Meng et al., 2002; Uebler, 2001) and to provide quick assistance in real time using different modalities such as: voice, gestures, touch screens, Web pages, interactive maps, etc. (Almeida et al., 2002; Wahlster et al., 2001; Gustafson et al., 2000; Oviatt et al., 2000). Obviously, to build this kind of systems, a complete platform is necessary to design, debug, execute and maintain these services and, at the same time, they have to provide the maximum number of features to the designer and final user, a high level of portability, standardization and scalability to minimize design time and costs. With these objectives in mind, and based on the results and experience obtained in previous projects (Córdoba et al., 2001; Lehtinen et al., 2000; Ehrlich et al., 1997), we undertook the European Project GEMINI (Generic Environment for Multilingual Interactive Natural Interfaces) developed from 2002 to 2004 (Gemini Project Homepage, 2004). The result is a complete platform that consists of a set of tools and assistants that guide the designer from the first steps of the design until the execution of the service. Such platform, its assistants, the strategies to simplify the design process, and the solutions to the problems of handling multiple modalities and multilinguality in a unified design environment is what we describe here.

1.2. Alternative approaches

Nowadays, several companies and academic institutions work in the development of such tools or platforms. To begin with, we have studied several approaches to find out their characteristics, positive aspects and limitations, so that we could contribute new ideas to the field.

The following tools developed in an academic environment are significant: CSLU's RAD toolkit from Oregon University (McTear, 1999; Cole, 1999), which allows the development of multimodal system initiative dialogs (voice and images), using a representation based on state graphs (McTear, 1998), that are built using a toolbar with objects that represent the different functions, such as flow diagrams and actions in the dialog. It also provides an interface to execute Tcl/Tk scripts that increases the possibilities of interaction, data retrieval, analysis, grammar development, etc., in the modules and application development. GULAN (Gustafson et al., 1998) is a platform used to build a yellow-pages system using voice and interactive maps in Web that searches for several services in Stockholm. Dialog flow is defined using a tree representation whose nodes model the structure, focus and the actions that are going to be executed inside each dialog state. SpeechBuilder (Glass and Weinstein, 2001), developed at the Spoken Language Systems Group from MIT, lets the designer use modules that conform the Galaxy architecture (Polifroni and Seneff, 2000) as a real-time platform, which communicates with the application using the HTTP protocol and parameter passing using a CGI script. To make the design, there is a Web interface to define, using an action definition language based on examples, the relevant semantic concepts and actions that are allowed in the application.

Although all these tools are easy to use thanks to their visual interface and their real-time platform, their main problem is that in order to use or increase their features the designer has to know several programming languages, and they offer a low standardization as they are tied to a specific execution platform. Besides, they have serious limitations when trying to implement dialog strategies that take into account the user level or when simultaneously building the application in several languages.

Regarding commercial platforms (e.g., Nuance, IBM WebSphere, Microsoft Speech Application SDK, Audium Builder, UNISYS, etc.), in general they provide several high-level tools to build multimodal and multilingual dialog applications (focused mainly in voice access systems) using widespread standards such as VoiceXML and SALT (Wang, 2002), and a support technology that accelerates the development of any service. In addition, there are several Web portals, e.g., the ones from BeVocal Café, Tellme Studio, VoiceGenie, etc., that help in script development, provide out-house hosting service, and offer a real-time platform for companies that do not have the technology required. However, a large drawback is that the runtime platform depends on the underlying technology (speech recognizer, text to speech systems, dialog managers, etc.) so the behavior may vary between platforms and it is difficult to integrate proprietary modules. They also present difficulties in integrating new modalities or transferring the service between operating systems.

We should also mention the growing interest in systems that use markup languages based on XML to define input/output data exchanged between their different modules (Flippo et al., 2003; Katsurada et al., 2002; Wang, 2000). Examples of this kind of languages are VoiceXML, and, more recently, SALT, that have helped a lot in the definition of dialogs, while they offer flexibility and portability to provide complete voice services quickly (Komatani et al., 2003; Bennett et al., 2002). Considering this and several factors such as the possibility of including new modalities and improvements, the independence of the service execution platform, and the ease of debugging of tag based languages, we decided to create our own language (called GDialogXML) as a format for internal communication between the platform assistants and to use the standard VoiceXML and/or xHTML as output scripts for the runtime system.

Regarding systems that offer strategies to accelerate dialog design we should mention (Denecke, 2002) where a complete three-layer architecture for rapid prototyping of dialog applications is presented. In the first layer, language and domain-independent algorithms are provided to describe the dialog objectives, discourse history and the semantic representation of the speech recognizer output. In the second layer, the interaction mechanism between user and system is described (e.g., variables used by the recognizer, database access variables and methods, dialog states, etc.) Finally, the third layer contains the dialog controller that uses the information from the other two layers and interacts with the final user. This system is similar to our proposal in some aspects, as the handling of concepts to facilitate multilingual interaction, the use of special variables related to the system and dialog status, and the use of automatic templates for each dialog state. Nevertheless, as the author admits, the templates fail when not all the states of the dialog can be covered. In our system, we have tried to avoid this by using more flexible and general templates, although less automatic.

In (Polifroni et al., 2003) a rapid development environment for speech dialogs from online resources is described. The development process first extracts knowledge from various Web applications and composes a dynamic database from it. The design automation is mainly based on the contents of such database. This is one of the differences with our approach: our platform uses the database structure, not its contents, to extract knowledge for the design process. Because of this, the design is more domain-independent, as it is more feasible to find data structures that are similar between several services and, therefore, can be applied in several applications. Besides, there are databases whose content cannot be easily used for research purposes for security reasons as in banking databases. Another important difference is that the speech dialog applications generated by our platform will be implemented in VoiceXML, which allows the generated dialogs to be executed with any VoiceXML interpreter.

More recently, in (Pargellis et al., 2004) a complete platform to build voice applications is described. The dialog structure can be modified using a set of templates adapted to the final user of the system, as well as several resources and service features. As in our proposal, the platform automates the generation of the final script in VoiceXML, the grammars and prompts, and the application flow; nevertheless, their proposal differs in that the automation efforts, in a similar way as in (Polifroni et al., 2003), because it is more focused on the dynamic contents of the database than on its structure, so it could be more domain dependent.

In relation to multimodal systems, we should especially mention the work in (Johnston et al., 2002), where a multimodal architecture for a dialog system based on finite states is described. This architecture allows a synchronous multimodal input/output of data using speech and/or gestures/images with a pen on a PDA. The authors emphasize the methods followed to guarantee the multimodal interaction, and the features provided for each modality and for the design of new applications. Even though we do not provide this kind of interaction in our system right now, our future work is oriented towards the creation of a similar mechanism using a standard language as X + V, the use and generation of multimodal grammars and the adaptation of our platform to a more distributed architecture that guarantees the synchronization of the different modalities.

Regarding tools that can be used to debug and evaluate the application off-line, we should mention the SUEDE platform (Klemmer et al., 2000), developed in the Group for User Interface Research from Berkeley University, that offers a graphical interface to capture and analyze training data using the Wizard of Oz (WOZ) technique. The objective is to provide the designer with a way of testing the prompts and system behavior by studying the system mistakes arising from problems in real time or bad recognition of user input. In our case, we have tried to minimize some of these design problems using automatic and configurable templates for the treatment of common recognition errors.

Finally, we have found out that our platform follows a very similar approach to the Agenda system (now called RavenClaw) from CMU (Rudnicky and Xu, 1999; Bohus and Rudnicky, 2003), in that the designer can create a service using a hierarchical representation of the task and its subcomponents, facilitating maintenance and scalability, and that each state is described using a set of forms with information regarding its restrictions and optional slots.

1.3. Relevant definitions

Throughout this paper we are going to use some terms that do not necessarily have the same meaning as the ones used in common literature or that do not present a general accepted definition. To clarify them and avoid confusions we want to define them here from the perspective of our platform.

Designer and user: The term designer refers to the person that uses the platform to build the service, and user refers to the final client of the developed service.

Multiple modalities: The common usage of the term multimodality in dialog applications refers to the ability to support the communication with the user through several channels to obtain and provide him/her information (Nigay and Coutaz, 1993). The most widely used modalities are voice, gestures, mouse, images or writing, which can be combined simultaneously or otherwise during the dialog. However, instead of multimodality, we have focused on providing multiple modalities from the designer point of view, referring to the platform's ability to generate the service for two modalities in a unified and simultaneous way: Web and voice. Right now, these modalities work apart from each other instead of being combined (synchronized) in the real-time system.

Mixed initiative and over-answering: It is well known that the concept of mixed initiative includes over-answering, as mixed initiative is a generic term used to refer to a flexible interaction between the user and the system to get together to reach a common final solution (Allen et al., 1999). However, we preferred to differentiate them to maintain the consistence with the specifications and implementation of the VoiceXML standard (McGlashan et al., 2004). In this sense we will use the term *mixed initiative* to indicate the system's ability to ask simultaneously for two or more compulsory data from the user, and, if the user's answer is incomplete—or the recognizer fails—new subdialogs are started to obtain the missing data. With *over-answering*, we indicate the user's ability to provide additional data—not compulsory at that state—to the system.

Dialog, state and action: From the terminology established by the W3C for an event-driven model of dialog interaction (W3C, 1999), we can find the following definitions:

- *Dialog*: a model of interactive behavior underlying the interpretation of the markup language. The model consists of states, variables, events, event handlers, inputs and outputs
- *State*: the basic interactional unit defined in the markup language... A state can specify variables, event handlers, outputs and inputs.

In spite of the differences in these definitions, throughout the paper we will use both terms with very little difference, as they will refer, from the perspective of a finite state machine, to each interaction with the user—or a set of them—needed to fulfill a service task. Nevertheless, the term *dialog* will be more associated to the interaction with the user, whereas *state* will mostly refer to a set of interactions and other additional actions, such as a database access.

On the other hand, the term *action* refers to each procedure needed to complete a state or a dialog, for example: calls to other dialogs, arithmetic or string operations, programming constructs, variable assignments, etc.

Slot: This term refers to each piece of compulsory information that the system has to ask the user in order to offer the service.

1.4. Paper organization

The paper is organized as follows. In Section 2 we present the overall architecture of the platform, its internal communication format based on XML and a brief description of its scope and limitations. In Sections 3–5, a full description of each platform module is presented, emphasizing the techniques and strategies used to ease the dialog design of applications. The separation of the sections corresponds to the three levels in the architecture. In Section 6 we present the approaches followed to provide portability and standardization to the platform, and some aspects of the implementation and the runtime system. In Section 7 we show the results of a subjective evaluation of the platform made by several designers. After that, we present some future plans in Section 8 and finish with the conclusions in Section 9.

2. Platform structure

Fig. 1 shows the complete platform architecture, also called Application Generation Platform (AGP). All these modules are independent of each other; nevertheless, they were integrated into a common graphical interface (GUI) to guide the designer in the design step by step and, at the same time, let him go back and forth. The platform is divided into three main layers. The reason for this division is to separate clearly the aspects that are service specific (general characteristics of the application, database structure, database access), those corresponding to the high level dialog flow of the application (modality and language independence), and the specific details imposed by each modality and language. In this way, the designer is able to create several versions of the same service (for different modalities and languages) in a single step at the intermediate level.

In more detail, the assistants of the first layer are used to specify the overall aspects of the service (e.g., modalities and languages to be implemented, general default values for each modality, libraries, etc.); then, the database structure, not its contents, is described (classes, attributes, relations, etc.); and

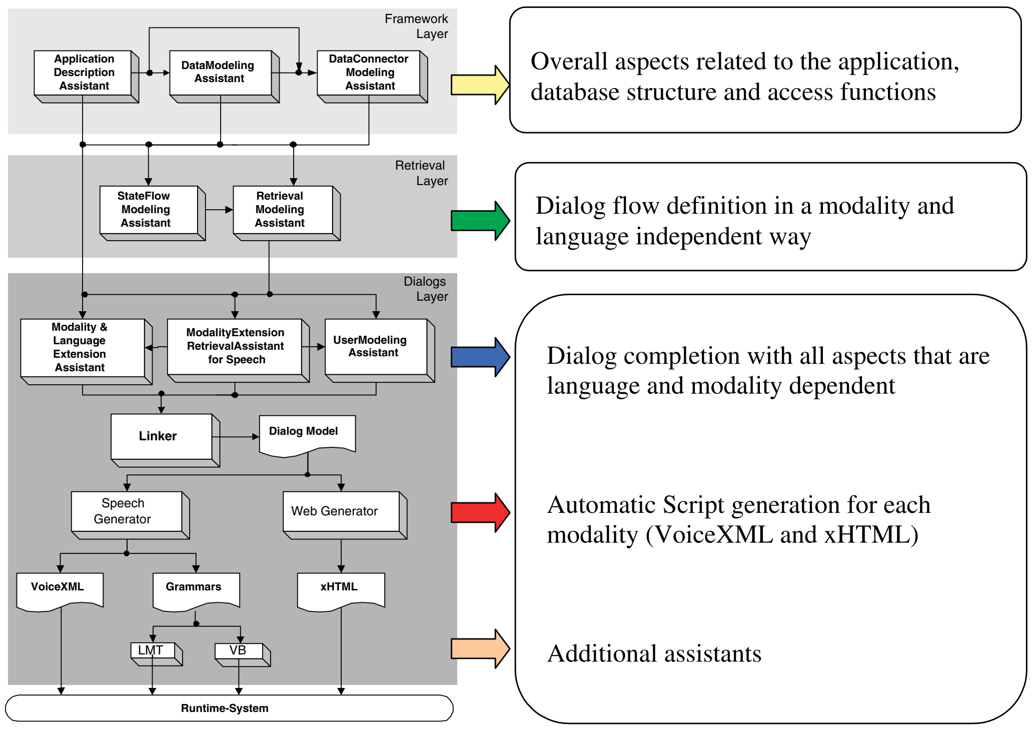

Framework Layer
Application Description Assistant
DataModeling Assistant
DataConnector Modeling Assistant
Retrieval Layer
StateFlow Modeling Assistant
Retrieval Modeling Assistant
Dialogs Layer
Modality & Language Extension Assistant
ModalityExtension RetrievalAssistant for Speech
UserModeling Assistant
Linker
Dialog Model
Speech Generator
Web Generator
VoiceXML
Grammars
xHTML
LMT
VB
Runtime-System
Overall aspects related to the application, database structure and access functions
Dialog flow definition in a modality and language independent way
Dialog completion with all aspects that are language and modality dependent
Automatic Script generation for each modality (VoiceXML and xHTML)
Additional assistants

Fig. 1. Platform architecture.

finally, the database access functions, needed for the real-time system, are defined (not their implementation).

In the second layer, the general flow of the application is modeled, including all the actions that form it (transitions and calls between dialogs, input/output information, calls to subdialogs, procedures, etc.) It is important to mention that in this layer no modality/language specific details are defined, such as prompts/grammars, recognition errors, design of the Web page, etc., as all these will be defined in the next layer. To be able to be modality and language independent, in this layer all the input/output data provided by/to the user are managed as *concepts*.

Finally, the third layer contains the assistants that complete the general flow specifying for each dialog the details that are modality and language dependent. Here, the prompts and grammars for each language, the appearance and contents of the Web pages, the error treatment for speech recognition mistakes or Internet access, the presentation of information on screen or using speech, etc., is defined. Furthermore, in this layer the final scripts of the service are generated, unifying all the information from the previous assistants.

As we describe in Section 2.2, all the assistants communicate between themselves using a common GDialogXML syntax. More details of the architecture can be found in (Hamerich et al., 2004a,b). In Sections 3–5, each layer and assistant of the platform are described. To clarify the design process and the interaction with the assistants, we will show some steps of an example dialog where a bank transfer between accounts is carried out, asking the user for the number of the source account, the destination account and the amount of money to be transferred.

2.1. Scope and limitations

The main objective of the platform is to allow the construction of dialog applications for multiple modalities and languages at the same time. The generated applications can be used to access services based on database queries/modification (e.g., banking, train reservations, share prices in real time, etc.) through a telephone or a Web browser. Considering the limitations imposed by the standards used in the scripts generated by the platform (see Section 6), it is limited in the current version to the execution of each modality on its own. In any case, we consider that the platform is well prepared for true multimodality. The only missing things right now are new code elements for synchronization in our XML syntax and a new code generator (e.g., for X + V).

For the speech modality, the platform generates a script using the VoiceXML 2.0 standard, so the main limitation is the impossibility of creating user initiative dialogs, but it allows a certain degree of mixed initiative. Regarding the Web modality, the platform generates pages made up with Web forms (including radio buttons, textboxes, combo boxes, etc.), and coded using the xHTML language, so they are accessible from a conventional Web browser. Besides, the platform allows the coding of multimedia contents (e.g., videos, recordings, images, etc.) as part of user output. Finally, because the output is coded in xHTML, an expert designer might use it as a base to add other more complex audiovisual resources, such as animations, interactive maps, etc., using specialized Web design tools.

In (Allen et al., 1999) four levels of mixing initiative are identified: unsolicited reporting, subdialog initiation, fixed subtask initiative, and negotiated mixed initiative. Unsolicited reporting allows an agent to inform others about critical information needed out of turn. Subdialog initiation allows the system to initiate a subdialog in certain situations, e.g., to ask for a clarification. In a fixed subtask initiative, the system keeps the initiative for a task, and it executes the task interacting with the user when necessary. In the negotiated mixed-initiative level, there is no fixed assignment of responsibilities or initiative, so agents can negotiate who takes the initiative and proceeds with the interaction based on it. On the other hand, (McTear, 2002) states that finite-state models are always fixed system-initiative, while frame-based systems may permit some degree of mixed initiative, but that they may also be fixed user-initiative. Finally, (Allen et al., 2001) survey five levels of systems in increasing complexity of software architecture: finite-state, frame-based, sets of contexts, plan based, and agent based models. Considering this perspective, our platform covers the first two levels of task complexity in Allen's classification, and supports, as a frame-based system, the lower levels of mixed-initiative interaction: unsolicited reporting and a few cases of subdialog initiation for the management of lists of objects (see Section 5.2.1).

2.2. GDialogXML

In order to ease communication inside the platform we have developed a new object oriented abstract language based on XML tags named Gemini Dialog XML or GDialogXML. Its main feature is its flexibility, which allows the modeling of all application data, the database access functions, the definition of all variables and actions needed in each dialog state, system prompts, grammars, user models, Web graphical interface, etc. Then, this information is used to carry out the conversion to the languages used for the final presentation of the service according to the modality (VoiceXML and/or xHTML).

Besides, the syntax allows the addition of new modalities, and the update to new versions of the script languages generated by the platform with little effort.

As (Schubert and Hamerich, 2005; Hamerich et al., 2003; Wang et al., 2003) describe in more detail, the GDialogXML syntax provides the means needed to model the following aspects: general concepts, data modeling, and dialogue modeling in a dependent and independent modality and language way. As general concepts, we can mention: variable and constant definition, variable assignments, file paths, arithmetic, boolean or string operations, control structures for loops and jumps, variable types (lists, objects, references to objects, atomic data), etc. For data modeling, we can specify the classes with attributes, which can have simple data types such as string, integer, boolean or complex types as embedded or referenced objects or lists, supporting inheritance from base classes, etc. Regarding dialog modeling, all dialog models consist of dialog modules that call each other. As the length of this paper does not allow us to include the complete specification of the syntax, the reader can find it in (Gemini Project Homepage, 2004). In any case, to clarify the input/output representation used in the assistants, we will include some fragments of the generated code in some assistants, giving suitable explanations of it.

3. Framework layer

This layer has three assistants that allow the overall specification of the service, the description of the database structure and the database access functions.

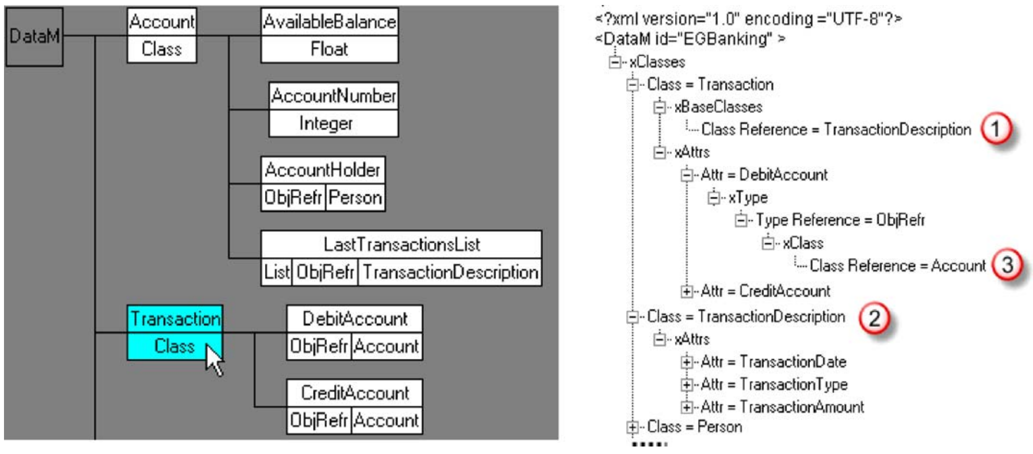
3.1. Application description assistant (ADA)

In this assistant, several overall aspects of the application, such as the number of modalities and languages, the location of some services such as the database access, database connection settings (total number of connection errors, timeouts), etc. are specified; for the speech modality, the timeout values for some events such as no input, default confidence levels for speech recognition, maximum number of repetitions/errors before transferring to the operator, etc.; for the Web modality, possible errors (e.g., page not found, non-authorized, timeouts, etc.). More information regarding the error handling capabilities of the AGP can be found in (Wang et al., 2003). Besides, the default overall strategy for dialogs is defined: system-driven or mixed initiative.

Finally, the designer specifies the libraries, which will be used to speed up the design process. Several types of libraries can be selected containing the definition of: data models, database access functions, list of prompts and grammars for each language, and dialogs from the general model of the application (see Section 4.2). The platform provides some generic libraries, such as prompts and grammars for confirmations, generic data models, etc., but its main potential is the possibility of saving most of the work done in the platform as libraries, including complete dialogs, so that after the creation of a few applications, the designer will have a complete set of libraries adapted to his liking and that can be reused in future applications. The platform allows the loading of libraries and provides the functionality to edit their code to adapt them to a new application.

3.2. Data model assistant (DMA)

This assistant defines the data structure (or data model) of the service specifying the classes, including inheritance, attributes and types that make up the database. It uses as input the location of libraries and files specified in the ADA. It is possible to define a class with attributes inherited from other classes. The attributes can be of several types: (a) atomic (e.g., strings, boolean, float, integer, date, time, etc.), (b) full embedded objects or pointers to existing classes, or (c) lists of atomic attributes or complex objects. A graphical view of a class and its attributes can be seen in Fig. 2 where, for the bank transfer example, the Transaction class has been defined, which is made up of two object type attributes from the class Account: the first one, DebitAccount, to specify the source account and the second one, CreditAccount, to specify the destination account. On the other hand, the class Account has several atomic type attributes (balance and account



```
<?xml version="1.0" encoding ="UTF-8"?>
<DataM id="EGBanking" >
  xClasses
    Class = Transaction
      xBaseClasses
        Class Reference = TransactionDescription
      xAttrs
        Attr = DebitAccount
          xType
            Type Reference = ObjRefr
              xClass
                Class Reference = Account
        Attr = CreditAccount
    Class = TransactionDescription
      xAttrs
        Attr = TransactionDate
        Attr = TransactionType
        Attr = TransactionAmount
    Class = Person
```

Fig. 2. Graphical details of a class and its attributes, and code fragment generated for the Transaction class.

number in the example) and other complex ones (account holder and last transactions list). We can also see the code generated for the Transaction class, together with a reference to its base class (Transaction inherits everything from the base class), called TransactionDescription (number 1) and the attributes that will be inherited (number 2). In number 3, the DebitAccount attribute is an object reference (ObjRefr) to the class Account, and the same applies to the CreditAccount attribute.

The design of the data model is accelerated in the assistant by the following features:

1. Re-utilization of libraries with models previously created, which can be copied totally or partially, or a new class can be created by mixing several original classes.
2. Automatic creation of a class when it is referenced as an attribute inside another one.
3. Definition of classes inheriting the attributes of a base class.

Finally, one advantage of this way of defining the data model is that it is not necessary to have the information contained in the database. This could be important if the real database cannot be accessed for security reasons (e.g., a bank database with confidential information regarding the clients). This was one reason to base the automation efforts on the structure of the database, not on its contents.

3.3. Data connector modeling assistant (DCMA)

This assistant allows the definition of the structure of the database access functions that are called from the runtime system. These functions are specified as interface definitions including their input and output parameters. This allows the use of database functions by dialog designers, without needing to know much about database programming at all. It uses the libraries specified in the ADA and the data model defined in the DMA.

As the runtime platform itself must be independent from backend systems and databases used in an application scenario, we leave the concrete implementation (in any suitable programming language: SQL, ORACLE, Informix, etc.) of the access functions to database or backend experts, meaning they will provide the functionality for the database functions, which have been created by the dialog experts. As the resulting model is independent from any implementation detail, it is not affected by changes in the system backend as long as the interface remains stable.

The main acceleration strategy in this assistant is the possibility of relating the input/output variables to attributes and classes from the data model, which were defined in the previous assistant. All this information is kept in the output model, which is going to be used automatically in future stages of the design process (see Section 4.2.2 item 2). In Fig. 3, the code generated by the assistant for the banking example is shown. In this case, the tag xArgumentVars (number 1) contains the information regarding the input parameters (the debit account number, the destination account and the amount to be transferred) and the tag xReturnValueVars (number 2) contains the return arguments (in this case, a boolean variable that indicates if the transfer is successful or not).

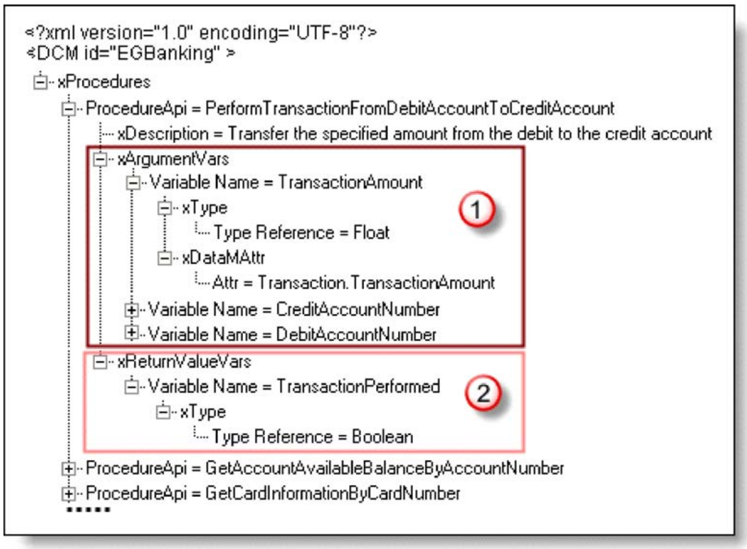

```
<?xml version="1.0" encoding="UTF-8"?>
<DCM id="EGBanking" >
 ⊟ xProcedures
    ⊟ ProcedureApi = PerformTransactionFromDebitAccountToCreditAccount
       xDescription = Transfer the specified amount from the debit to the credit account
       ⊟ xArgumentVars
          ⊟ Variable Name = TransactionAmount
             ⊟ xType
                  Type Reference = Float
             ⊟ xDataMAttr
                  Attr = Transaction.TransactionAmount
          ⊞ Variable Name = CreditAccountNumber
          ⊞ Variable Name = DebitAccountNumber
       ⊟ xReturnValueVars
          ⊟ Variable Name = TransactionPerformed
             ⊟ xType
                  Type Reference = Boolean
    ⊞ ProcedureApi = GetAccountAvailableBalanceByAccountNumber
    ⊞ ProcedureApi = GetCardInformationByCardNumber
    .....
```

Fig. 3. GDialogXML code generated by the DCMA for the bank transfer.

4. Retrievals layer

In this layer, the service flow is defined at a high level, i.e., in a language and modality independent way, so all it is done using concepts.

4.1. State flow modeling assistant (SFMA)

This assistant is very important because it drastically accelerates the design process, especially in the next assistant. As input, it uses the general strategy for the service that is specified in the ADA, and the data model specified in the DMA. Then, the designer has to specify the states that make up the dialog flow, the data (slots) that have to be filled by the user in each state and the transitions between the current state and the following one(s). Additionally, it is possible to specify which slots are optional (for over-answering) and which ones can be asked for by using mixed initiative. Here, only the flow structure is defined, not the conditions that determine the transitions between states, internal actions, nor other more detailed aspects because these are defined in the next assistant in a rule-based manner.

Fig. 4 shows the code generated by the SFMA for the example dialog; it includes information regarding the slots (field xInputFieldVars), dialog transitions (field xCalls), and generic information of the application, such as the name of the initial dialog. The figure also shows the definition of the state where the bank transfer data are collected. In this example, only the account names in that state have been selected, and the collection of the amount to be transferred has been left for the next state, called GetTransactionAmount. Besides, both slots are collected using mixed initiative (the tag "xIs-MixedInititative" is set to true).

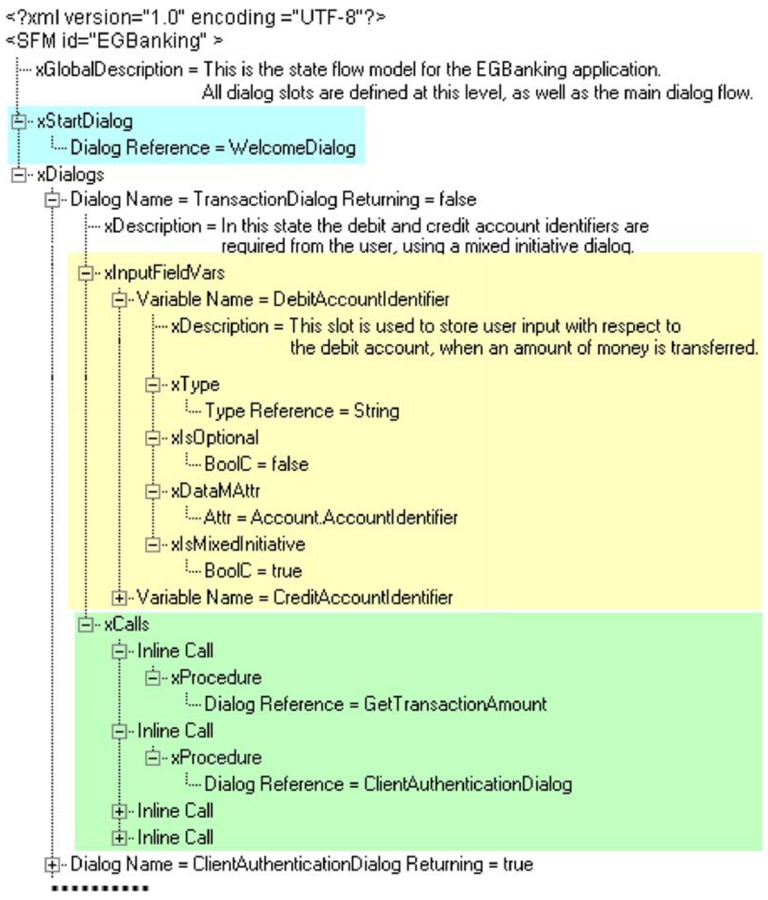
As a speed up strategy, the slots can be defined by making reference to atomic attributes of the classes defined in the data model, which is the most usual case, or as independent items. These references to the data model ease the work in the following assistant where action proposals for that specific dialog are automatically presented (see Section 4.2.2). Moreover, if a transition to an undefined state is specified, that state is created automatically.

4.2. Retrieval modeling assistant (RMA)

In this assistant, a detailed definition of each dialog in the application is made; so, it uses all the information from the previous assistants, generating the detailed definition of *all* the actions (e.g., loops, if-conditions, math or string operations, transitions between states, information regarding mixed initiative and over-answering, calls to dialogs to provide/obtain information to/from the user, etc.) to be done in each state defined in the SFMA, or in new states that can be defined later.

4.2.1. Capabilities and dialog types

Given the large amount of actions that can be carried out in each state, a large programming effort was necessary here, looking for its automation and



```
<?xml version="1.0" encoding ="UTF-8"?>
<SFM id="EGBanking" >
 xGlobalDescription = This is the state flow model for the EGBanking application.
                      All dialog slots are defined at this level, as well as the main dialog flow.
 xStartDialog
    Dialog Reference = WelcomeDialog
 xDialogs
    Dialog Name = TransactionDialog Returning = false
       xDescription = In this state the debit and credit account identifiers are
                      required from the user, using a mixed initiative dialog.
       xInputFieldVars
          Variable Name = DebitAccountIdentifier
             xDescription = This slot is used to store user input with respect to
                            the debit account, when an amount of money is transferred.
             xType
                Type Reference = String
             xIsOptional
                BoolC = false
             xDataMAttr
                Attr = Account.AccountIdentifier
             xIsMixedInitiative
                BoolC = true
          Variable Name = CreditAccountIdentifier
       xCalls
          Inline Call
             xProcedure
                Dialog Reference = GetTransactionAmount
          Inline Call
             xProcedure
                Dialog Reference = ClientAuthenticationDialog
          Inline Call
          Inline Call
    Dialog Name = ClientAuthenticationDialog Returning = true
    ..........
```

Fig. 4. GDialogXML code generated by the SFMA.

flexibility. Starting with the main window, it allows several editing and visualization capabilities such as a tree-structured flow diagram where each leaf and branch represents the states and possible transitions defined in the previous assistant. A color-coding convention shows whether a dialog has been edited or not, the dialog type, etc. In addition, it is possible to access information regarding actions and variables already defined for each leaf (dialog) in the flow. All the automatically generated dialogs (DGets and DSays), libraries and database access functions already defined can also be used and edited. Other available capabilities are the safe creation/deletion of dialogs, variables and constants, and the visualization of information from previous assistants.

The platform provides four basic dialogs types that cover the usual possibilities in programming: based on a loop, based on a sequence of actions (or subdialogs), a switch construct based on information input by the user or a switch construct based on the value of a variable. Besides, empty dialogs, with no action inside, can be created (used to specify the call to a dialog that will be defined completely afterwards) so that a top-down design of dialogs can be made; in this case, the dialog type is selected whenever the designer tries to edit the empty dialog. Another possibility is dialog cloning, useful when the dialog to be defined is very similar to an existing one.

The tool also provides the possibility of manually creating dialogs to obtain information from the user (called DGet), and dialogs to provide information to the user (called DSay), which will be described further in the next section.

4.2.2. Strategies to accelerate the design

The following useful strategies have been added to this assistant to accelerate dialog design (D'Haro et al., 2004):

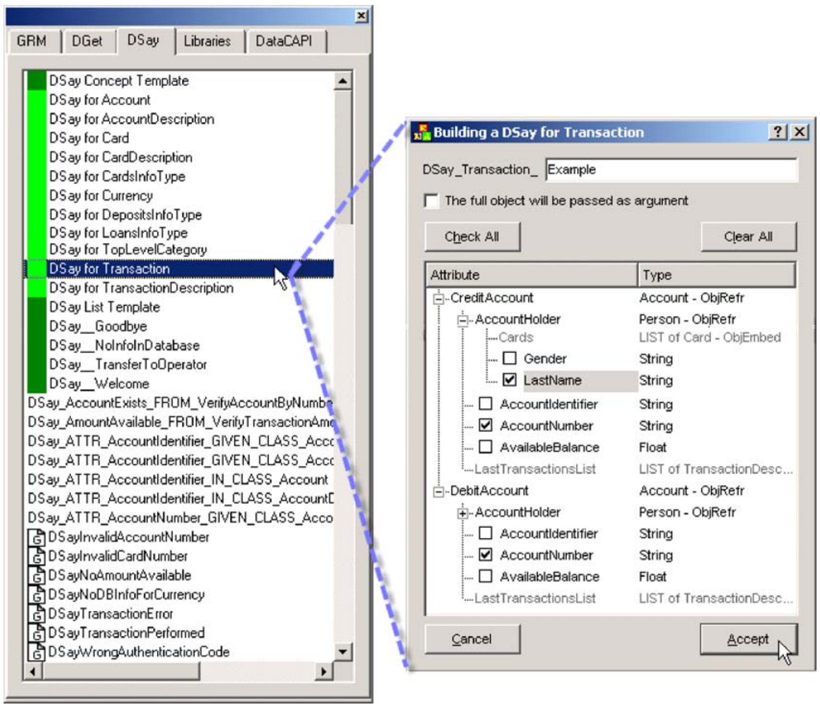
(1) *Automatic dialogs*: When the RMA is started, it analyses the information from the Data Model

looking for all attributes defined as atomic types to generate dialogs to obtain information from the user (DGet) and dialogs to provide information to the user (DSay). These dialogs include a tag used by the Modality Extension Assistants (see Sections 5.2 and 5.3) to find out whether the prompt/output concept to be presented to the user (for DSay), the grammar/input concept used by the recognizer/ Web generator and the confirmation strategies (for DGet) have to be specified.

If the attributes are complex or include object inheritance, the assistant is not able to generate automatic dialogs for them. However, the assistant provides configurable DSay dialogs using a template (see Fig. 5) that shows the class and its attributes, expanding the complex attributes (with inheritance and objects) and selecting any attribute that will form part of the prompt. Other DSay dialog templates are also available: generic DSay to provide concepts, configurable DSay to present variables from a dialog, DSay to present lists of objects, specific DSay to provide the value returned by the database access functions, and predefined DSay such as: Welcome, Goodbye, Transfer to operator, etc. Besides the DGet and DSay dialogs, dialogs from loaded libraries and database access functions can be used.

Fig. 5 shows all the dialogs mentioned above, with an example of the configurable template for the Transaction class called 'DSay for Transaction' where several attributes have been selected and will be provided to the user in the real-time system. The flexibility of this template lets the designer select attributes from the different child classes of the Transaction class (e.g., AccountNumber), complex attributes coming from inherited classes and contained in another class (e.g., LastName from class AccountHolder included in class CreditAccount). When dialog definition is over, it is added to the list of dialogs in the DSay tab, so that it can be used later on.

(2) *Automatic generation of action proposals in each state*: Every time a dialog defined in the SFMA (see Section 4.1) is edited, a popup window called "SFM proposals" is shown (see Fig. 6) in which the assistant includes all the actions that are considered relevant (and with many chances to be used) for that state. To decide which actions are relevant, all the information already defined in previous assistants, especially the SFMA, is analyzed using the following strategies for each of the four sections:



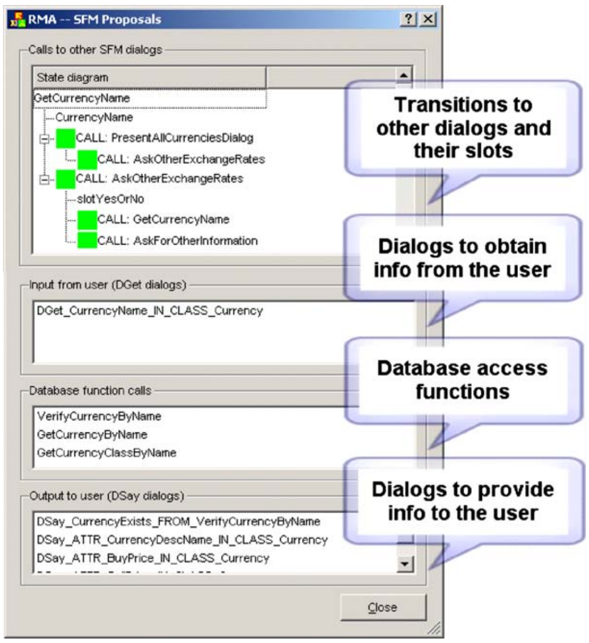
DSay Concept Template
DSay for Account
DSay for AccountDescription
DSay for Card
DSay for CardDescription
DSay for CardsInfoType
DSay for Currency
DSay for DepositsInfoType
DSay for LoansInfoType
DSay for TopLevelCategory
DSay for Transaction
DSay for TransactionDescription
DSay List Template
DSay__Goodbye
DSay__NoInfoInDatabase
DSay__TransferToOperator
DSay__Welcome
DSay_AccountExists_FROM_VerifyAccountByNumbe
DSay_AmountAvailable_FROM_VerifyTransactionAm
DSay_ATTR_AccountIdentifier_GIVEN_CLASS_Acc
DSay_ATTR_AccountIdentifier_GIVEN_CLASS_Acc
DSay_ATTR_AccountIdentifier_IN_CLASS_Account
DSay_ATTR_AccountIdentifier_IN_CLASS_AccountI
DSay_ATTR_AccountNumber_GIVEN_CLASS_Acco
DSayInvalidAccountNumber
DSayInvalidCardNumber
DSayNoAmountAvailable
DSayNoDBInfoForCurrency
DSayTransactionError
DSayTransactionPerformed
DSayWrongAuthenticationCode

DSay_Transaction_ Example

☐ The full object will be passed as argument

Attribute	Type
CreditAccount	Account - ObjRefr
AccountHolder	Person - ObjRefr
Cards	LIST of Card - ObjEmbed
☐ Gender	String
☑ LastName	String
☐ AccountIdentifier	String
☑ AccountNumber	String
☐ AvailableBalance	Float
LastTransactionsList	LIST of TransactionDesc...
DebitAccount	Account - ObjRefr
AccountHolder	Person - ObjRefr
☐ AccountIdentifier	String
☑ AccountNumber	String
☐ AvailableBalance	Float
LastTransactionsList	LIST of TransactionDesc...

Fig. 5. Auxiliary screen of the RMA and popup window for dialog configuration.


RMA -- SFM Proposals
Calls to other SFM dialogs
State diagram
Input from user (DGet dialogs)
Database function calls
Output to user (DSay dialogs)
Close
Transitions to other dialogs and their slots
Dialogs to obtain info from the user
Database access functions
Dialogs to provide info to the user

```
GetCurrencyName
  CurrencyName
  CALL: PresentAllCurrenciesDialog
    CALL: AskOtherExchangeRates
  CALL: AskOtherExchangeRates
    slotYesOrNo
    CALL: GetCurrencyName
    CALL: AskForOtherInformation
```

```
DGet_CurrencyName_IN_CLASS_Currency
```

```
VerifyCurrencyByName
GetCurrencyByName
GetCurrencyClassByName
```

```
DSay_CurrencyExists_FROM_VerifyCurrencyByName
DSay_ATTR_CurrencyDescName_IN_CLASS_Currency
DSay_ATTR_BuyPrice_IN_CLASS_Currency
```

Fig. 6. Example with automatic dialogs and database access function proposals.

- Slots asked in the current state, the transitions and the corresponding slots in those destination states. Direct information from the SFMA is used for that.
- State specific DGets: to select them, the system looks for the slots defined in the SFMA; if they are related to the Data Model, the system selects the corresponding dialogs automatically; if not, a more relaxed criterion is used, which is to look for a match in the name or attribute type.
- Database access functions: to filter the possible functions already defined in the DCMA, the system first considers functions with the same number and type of input parameters as the defined slots for the current dialog. The next criterion is as follows: if the parameter includes a reference to the data model (see Section 3.2), there should be a match in class and attribute between slot and parameter; if not, they should match in type. If no function passes these filters, a more relaxed filter is applied (e.g., similarity between names). If even with the relaxed filter, there is no function in this window, it would probably mean that there is no database access function suitable for that state and it should have been defined before (see Section 3.3). The assistant offers the possibility of creating those functions, and then reload the information in this window.
- State specific DSays: they are selected in a similar way to DGet dialogs, but we also include DSays specific to the values returned by the database functions selected in the previous step.

Fig. 6, shows the banking application example, where given the currency name the system provides its specific information (buy and sell price, general information, etc.), all the designer would need to do is to select the corresponding DGet in the window (DGet_ATTR_CurrencyName_IN_CLASS_Currency), then the database access function GetCurrencyByName and finally the DSays that provide the desired attributes from the currency. To finish the design, he/she would drop the call to the next state (e.g., AskOtherExchangeRates).

(3) *The passing of arguments between actions is automated*: This is a critical aspect of dialog applications design. Several actions and states have to be 'connected' as they use the information from the preceding dialog.

To automate this connection, the assistant detects the input/output variables required in each action and, using a popup window, it offers the most suitable already defined variable of a compatible type; if there is more than one variable of a compatible type, the assistant sorts them according to the name similarity between variable and dialog. If there is no compatible variable already defined in the system or the name proposed by the assistant is not desired, a new local or global variable can be created in the same window.

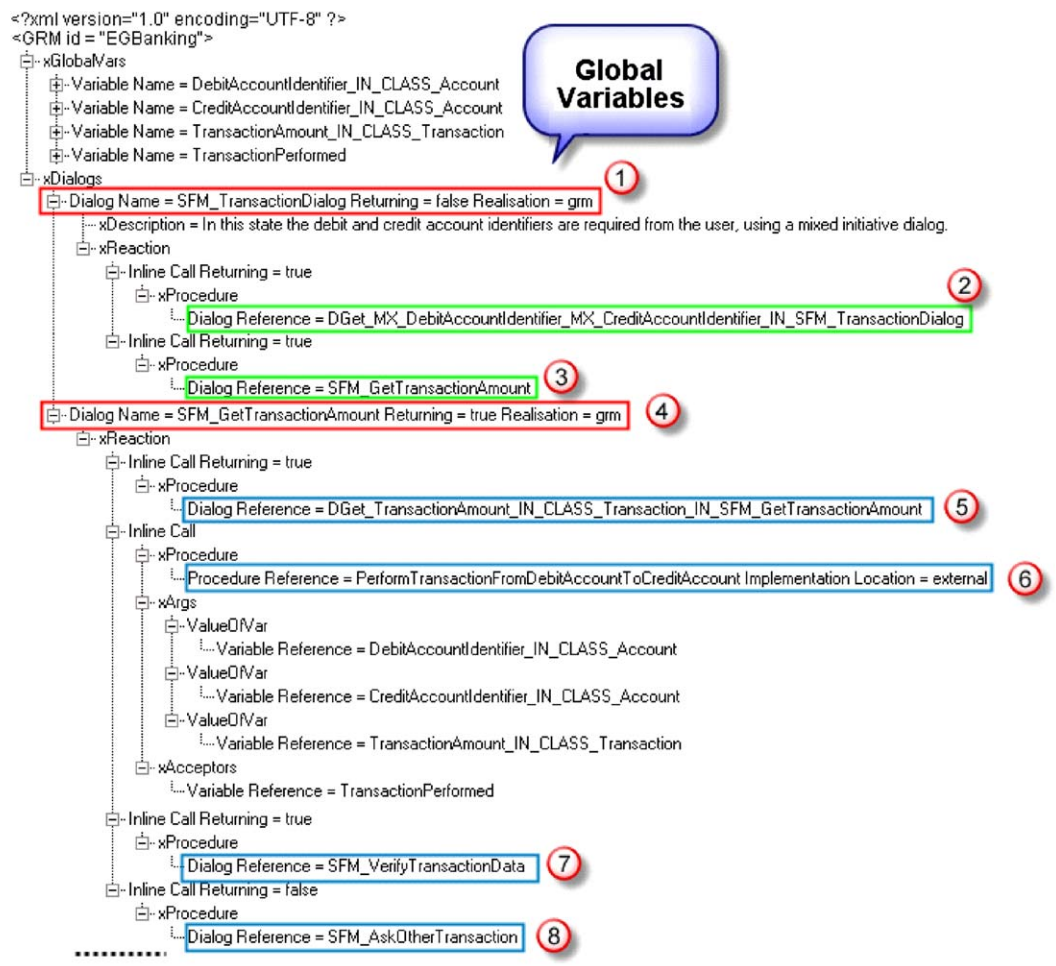
Moreover, if the designer makes a mistake or needs to edit the matching made in the previous steps, the assistant provides a window where all this matching can be edited.

4.2.3. Mixed initiative and over-answering

This capability allows the creation of complex dialogs where the system can ask for several slots at the same time or the user can answer with optional information. Even though these two functionalities might be considered as speech modality dependent or unnecessary for the Web modality, so they should not be handled at this stage, we preferred to include them here for two main reasons: first, because some other modality that can be included in the future might benefit from their functionality, and, second, we can make it possible that in a Web page the user does not have to fill in all required fields at the same time. In this case, the Web system would detect that a certain slot is missing and, instead of generating an error, it would ask for the missing data in a posterior form, in a similar way as VoiceXML handles mixed initiative with several slots.

To provide this functionality, the system offers a Mixed Initiative Template which the designer can drag and drop over the dialog that is being edited. The template shows available slots which can be selected (by default, the ones specified in the SFMA for the current dialog). Moreover, the template gives the possibility of adding optional slots to be used for over-answering at the same time. With this information, the system generates the necessary programming code (calls and automatic dialogs) in GDialogXML syntax that controls the mixed initiative handling: ask for several slots at the same time, handle the situation in which the user answers partially or only some of the slots are filled after the recognition, so the system has to ask again for unsolved slots, and handle the keeping of the optional slots when they are input by the user.

To admit over-answering, the procedure is very similar: when the designer drops any DGet (action to obtain data from the user) he/she is offered to select additional slots as over-answering from that specific state and slots from the following states in the flow (with a limit of two in the hierarchy). As default, the slots defined as optional in the SFMA are automatically converted into over-answering slots here. In the runtime system, the behavior is that before any DGet the system checks whether the data to be asked has been already obtained in a previous state in the flow (as would be the case with over-answering). To help in this checking, in the final script all slots are declared as global variables, so they can be accessed from any state.



```
<?xml version="1.0" encoding="UTF-8" ?>
<GRM id = "EGBanking">
⊟-xGlobalVars
   ⊞-Variable Name = DebitAccountIdentifier_IN_CLASS_Account
   ⊞-Variable Name = CreditAccountIdentifier_IN_CLASS_Account
   ⊞-Variable Name = TransactionAmount_IN_CLASS_Transaction
   ⊞-Variable Name = TransactionPerformed
⊟-xDialogs
   ⊟-Dialog Name = SFM_TransactionDialog Returning = false Realisation = grm
      -xDescription = In this state the debit and credit account identifiers are required from the user, using a mixed initiative dialog.
      ⊟-xReaction
         ⊟-Inline Call Returning = true
            ⊟-xProcedure
               └-Dialog Reference = DGet_MX_DebitAccountIdentifier_MX_CreditAccountIdentifier_IN_SFM_TransactionDialog
         ⊟-Inline Call Returning = true
            ⊟-xProcedure
               └-Dialog Reference = SFM_GetTransactionAmount
   ⊟-Dialog Name = SFM_GetTransactionAmount Returning = true Realisation = grm
      ⊟-xReaction
         ⊟-Inline Call Returning = true
            ⊟-xProcedure
               └-Dialog Reference = DGet_TransactionAmount_IN_CLASS_Transaction_IN_SFM_GetTransactionAmount
         ⊟-Inline Call
            ⊟-xProcedure
               └-Procedure Reference = PerformTransactionFromDebitAccountToCreditAccount Implementation Location = external
            ⊟-xArgs
               ⊟-ValueOfVar
                  └-Variable Reference = DebitAccountIdentifier_IN_CLASS_Account
               ⊟-ValueOfVar
                  └-Variable Reference = CreditAccountIdentifier_IN_CLASS_Account
               ⊟-ValueOfVar
                  └-Variable Reference = TransactionAmount_IN_CLASS_Transaction
            ⊟-xAcceptors
               └-Variable Reference = TransactionPerformed
         ⊟-Inline Call Returning = true
            ⊟-xProcedure
               └-Dialog Reference = SFM_VerifyTransactionData
         ⊟-Inline Call Returning = false
            ⊟-xProcedure
               └-Dialog Reference = SFM_AskOtherTransaction
   ..........
```

Fig. 7. Code generated by the RMA for the bank transfer example.

4.2.4. RMA output

We can see the output generated by this assistant in Fig. 7. The first section shows the global variables of the application, which store the slots defined for the application which may need to be accessed in all dialogs (for over-answering, as we will see in Section 5.5.1). The actions executed in the two dialogs that form the bank transfer (included in the xReaction tag) are also shown: SFM_TransactionDialog (number 1) and SFM_GetTransactionAmount (number 4). In the first one, there is a call to a subdialog (marked as number 2) that fills the source and destination accounts data using mixed initiative; then, there is a call (marked as number 3) to the second dialog. In the second dialog there is a call to a subdialog that collects the amount to be transferred (number 5), a call to the database access function (number 6), using as input parameters the three items already collected, which returns a boolean variable called TransactionPerformed, and a call to the next two dialogs specified in the SFMA (numbers 7 and 8).

5. Dialogs layer

In this layer, the dialog is completed with all modality and language dependent aspects. It has four main assistants that are dedicated to the following tasks:

- To define the user levels and their settings (UMA, see Section 5.1), to complete the modality dependent aspects of dialog design (MERA, see Section 5.2),
- To complete the language dependent aspects and the input and output concepts for each modality (MEA, see Section 5.3) and finally,
- To unify all the information and generate the execution scripts according to the modality (see Sections 5.4 and 5.5).

Finally, in Section 5.6, we will briefly outline some additional assistants.

5.1. User modeling assistant (UMA)

This assistant allows the specification of different user levels and settings for each dialog in the application in order to provide a more personalized attention to the final user. It uses as input the default confidence and error values defined in the ADA, and all dialogs defined in the RMA.

To start with, all the values are specified first for some specific user levels, but they later can be customized for each specific dialog state, so that all settings can be user-level dependent and dialog state dependent. This way, the designer may impose, for example, a stricter confirmation for some critical data such as the amount in a banking transaction. To speed up the process, defaults are used: user-level settings take the values defined in the ADA and dialog dependent settings inherit the user-level settings.

The designer can specify different settings as the possibility of barge-in for a particular user level, the maximum number of retries if there is an error (considering several error types), the maximum timeouts for several events, etc. Besides, the confidence levels that should be used in recognition for each user level are specified as they determine the confirmation type that should be used (see Section 5.2.2): no confirmation (confidence between the specified value and 1.0), implicit (confidence between the specified value and the value for 'no confirmation'), explicit (between the specified value and the value for 'implicit') and repeat (between 0 and the value for 'explicit').

The decision as to the current user level is made by a runtime component that is called after each interaction in the script generated by the platform and that sets the common internal variable that is used in the final script. This way, the platform is independent of the user modeling technology that is used.

5.2. Modality extension retrieval assistant for speech (MERA)

In this assistant, the modality extension ends up by adding special subdialogs that complement the dialogs already defined for the application in the RMA. This way, the designer can include complex dialogs to deal with modality specific problems. We have focused on researching semiautomatic solutions for two basic problems that are specific to the speech modality: the presentation of object lists in several steps (applied to DSay dialogs concerning a list) and confirmation handling, i.e. how to handle recognition errors in dialogs that obtain information from the user (applied to DGet dialogs). The input is the database model specified in the DMA and the dialogs defined in the RMA (especially those marked as DGet and DSay for lists).

5.2.1. Presentation of object lists

Object lists are the result of a database query, so there is usually a lot of information to be provided to the user. We will consider four different cases as a function of the number of items in the list. For each case, a simple form allows the designer to specify the actions that have to be carried out. After filling the forms, the actions and new dialogs needed to provide/obtain information to/from the user are automatically generated. Furthermore, the assistant provides the most reasonable default values for all dialog and slot names after the analysis of the input files. The four different cases and their actions are as follows:

1. *The list is empty.* The system tells the user that there is no available information and then jumps back to a state selected by the designer, where the user is asked again for some selected slots defined by the designer, looking for a less restrictive query.
2. *The list has one item.* The designer defines a configurable DSay that provides complete or partial info from the item found.
3. *More than one item and less than a maximum allowed.* This is a more complex situation, as the items have to be provided in groups. After playing the info in each group, the user is asked if he/she wants to continue, repeat the group, begin from scratch, exit or select a specific item to receive more detailed information. Here, we have used some command names proposed by the Universal Speech Interface project (Toth et al., 2002). When the user selects the item he/she desires, the system provides detailed information of the object using a new selection of attributes specified by the designer. Another situation that we must face is when the system finishes reading the whole list and the user does not like any item or has cancelled before the end of the list. In this case, the system informs the user and repeats the same process as for case 1. The designer also has the option of informing the user how many items there are in the list, and the user may choose how many he/she wants to listen to.
4. *More items than the maximum allowed.* As there are too many items, the search has to be more restrictive. We can have two different situations. First, if all slots of the application are already filled, the user has to change some of them to make them more restrictive (e.g., he/she wants last month's transactions but there are too many). The designer specifies these slots (they will be cleared) and the questions will be repeated, in a similar way as in case 1. In the second situation, if there are still some slots to be asked the system continues with the normal dialog flow until the next database query. We have also considered a simplified case: when the list only depends on one slot input by the user, e.g., when he/she asks for a list of banking transactions. In this case, we present a simplified version of the previous windows where the designer does not need to specify the slots to be cleared.

5.2.2. Confirmation handling

The result of speech recognition has to be confirmed before making a database query. We confirm both normal slots, mixed initiative slots and over-answering slots. We have considered two types of confirmation in this assistant: Simple and Complete. Simple is recommended for dialogs that need a very high confidence, such as Yes/No or passwords questions; in this case only two levels are allowed: none and repeat the question (like in a no-match situation). However, Complete uses several levels of confidence to determine the confirmation type: none, implicit, explicit or repeat the question.

The MERA uses the same common internal variable mentioned in the UMA assistant to store the confidence value returned by the last recognition call. Then, the final script tells the system to compare this value with the current confidence limits, stored in four fixed name variables, as defined in the UMA for each user level and specific dialog.

The assistant automatically selects the input dialogs (DGet) that need confirmation and analyzes their flow to propose the most suitable confirmation type (Simple or Complete), but the designer can change that proposal and the assistant checks whether the type is feasible. The algorithm that analyzes the dialog is as follows: first, the system examines the number and type of slots to be retrieved by the DGet dialog, and if there is only one slot, its type is Boolean or string (as used to contain an alphanumeric password) and the number of actions in the calling dialog is not too high, the system selects the simple case; if not, it selects the Complete case. The assistant also controls whether implicit confirmation can be allowed. For example, if the next step in the dialog flow is the database access, explicit confirmation should be used regardless of the confidence level. Moreover, the assistant stores the name of this dialog and with this information sets the place where to jump back in case the user rejects the implicit confirmation in the following state in the flow (the rejection is detected in this state).

Finally, the assistant automatically generates the dialog flow (consisting of calls to internal automatically generated dialogs for each type of confirmation and dialog state) to carry out all the confirmation and subsequent correction. These internal dialogs are named in a smart way (using the USI project) so that the designer can easily identify them when he/she has to define grammars and prompts in the next assistant.

5.3. Modality and language extension assistant (MEA)

Here, the language dependent aspects of an application (input and output prompts/concepts) are specified for each modality. For speech modality, the extensions consist of links to grammar and prompt concepts, while for Web modality the extensions consist of links to input and output concepts. In both cases, the extensions are language independent. In addition, language dependent information, specifically wording for both speech prompts and Web output concepts, is also set here. All this information is saved in different files for each language and modality, whose content and organization is explained at the end of this section. As input, it uses all dialogs defined in the RMA and MERA, together with the specification of the user levels from the UMA.The assistant detects the input/output dialogs (DGets and DSays) defined in previous assistants and asks the designer to define prompts and recognition grammars for them. In addition, the assistant lets the designer define the help prompts for high level dialogs that are not classified as input/output.

For the speech modality, several prompts for each input dialog have to be defined: the default one, for the different user levels and for all possible recognition errors. In all dialogs, the input/output parameters and the global variables can be used as part of the prompt. To speed up the process of typing all these prompts, the assistant offers two possibilities: reuse prompts already available for the current application or reuse prompts generated in previous applications and saved as libraries. Prompts are set using three alternatives: text-to-speech (TTS) prompts, prerecorded audio files, or generated by a Natural Language Generation (NLG) module in the runtime system.

In case of TTS prompts, the SSML markup language can be optionally used. The tags that we have considered for the runtime system are as follows:

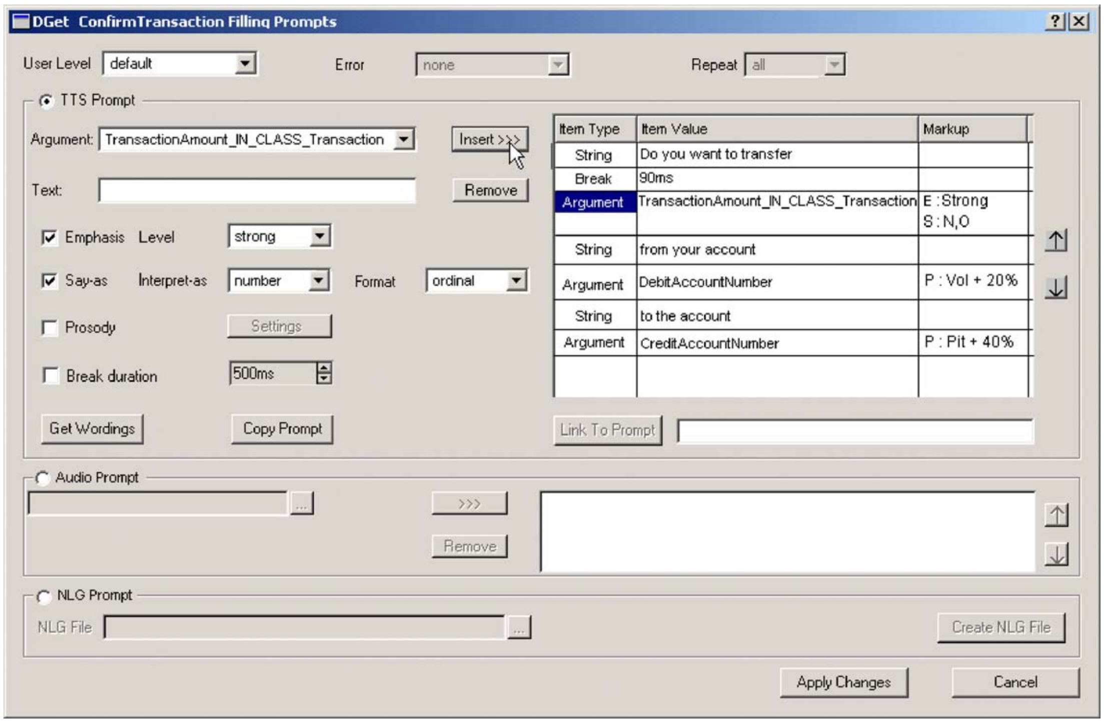
1. "emphasis" (to emphasize specific fragments), with the following values for the "level" attribute: "strong", "moderate", "none", and "reduced",
2. "break" (a break of a specific duration in ms), with the "time" attribute,
3. "prosody", with "pitch", "rate" and "volume" attributes. To specify them, in the platform we have used a relative value as a positive or negative percentage, e.g., "+10%".

In Fig. 8 we can see how this information is input using the assistant, together with the SSML tags used and the slots that are used as arguments for the prompt (e.g., DebitAccountNumber). We can also see in the bottom part the sections dedicated to the specification of prerecorded audio files (Audio Prompt) and the file used by the Natural Language Generation (NLG) to generate the prompt in real time. In the Natural Language Generation (NLG), the prompts can be defined using the Language Modeling Toolkit (see Section 5.6), so they will be coded in JSGF format.

Once the prompts for the main language have been specified, the designer has to specify them for the additional languages. This process is accelerated by using the main language prompt as a template to edit the string parts of a prompt. These prompts can be specified either at once for one language for all dialogs, or for each dialog for all additional languages.

For the Web modality, the procedure is somewhat different because of different concepts for user interaction. On the one hand, each output concept corresponds to some xHTML markup code, optionally parametrized. On the other hand, each input concept corresponds to a Web form control like a text input field, a text area, a select or a choice box, etc. In addition, a set of attributes can be defined for each component: text elements for a label, a hint, an alert or an error message can be set and the rendering behavior of the control can be defined.

As output, the assistant generates four different files for each modality. The first file contains infor-



User Level: default   Error: none   Repeat: all

(●) TTS Prompt

Argument: TransactionAmount_IN_CLASS_Transaction

Text:

[✓] Emphasis Level: strong

[✓] Say-as Interpret-as: number Format: ordinal

[ ] Prosody

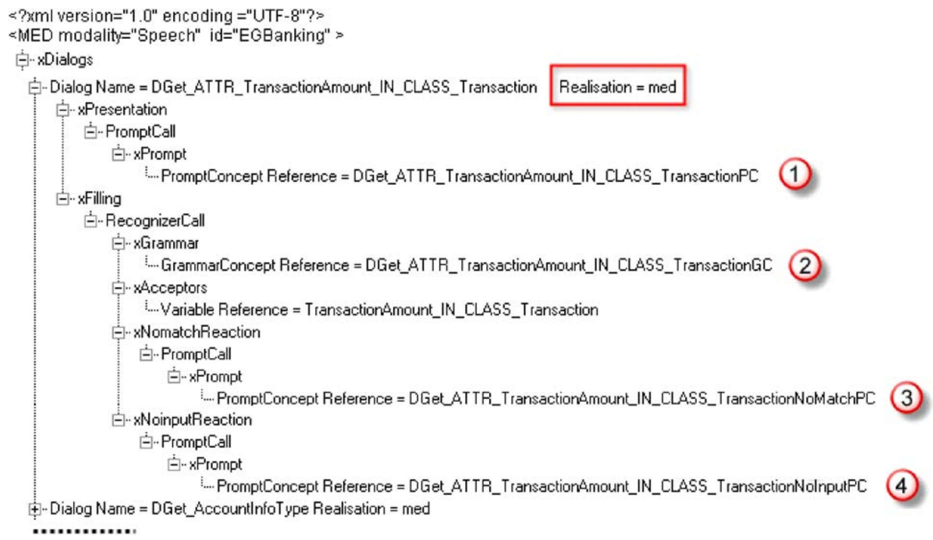
[ ] Break duration 500ms

Item Type	Item Value	Markup
String	Do you want to transfer	
Break	90ms	
Argument	TransactionAmount_IN_CLASS_Transaction	E :Strong S :N,O
String	from your account	
Argument	DebitAccountNumber	P : Vol + 20%
String	to the account	
Argument	CreditAccountNumber	P : Pit + 40%

Fig. 8. Example of the definition of a TTS prompt using SSLM tags.

mation regarding every dialog in the application and references to the input/output concepts used in each one. In Fig. 9 we can see the code for this file for the speech modality and for the dialog that collects the amount to be transferred in the bank example. We highlight the use of the tag Realisation because it tells the linker that this code is an extension of a dialog already defined in the RMA. Besides, the tag xPresentation holds the information related to the system prompt concepts (marked as number 1); the tag xFilling gives the information related to the behavior of the recognizer, i.e., the prompts used to inform the user of an unrecognized utterance (number 3) and no input detected (number 4), together with the grammar to be used in the recognition (number 2). As we have already mentioned, multilinguality is achieved using concepts, so all definitions here for prompts and grammars refer to the concepts (PC suffix for prompts and GC for grammars). These references are solved in auxiliary files that we describe below. Finally, we should mention that for the Web modality the same tags are used (xFilling and xPresentation) but, instead of prompts and grammars, input (using the tag InputControlCall) and output (tag OutputControlCall) concept references are used.

The second file, for the speech modality case, is called 'grammar concept file', and it contains the association between 'grammar concept' (GC) and the filename of the grammar(s) that will be used in the real-time system, so it is language independent. As we have different grammars for each language, to achieve multilinguality the grammars in all languages have the same name but are in separate directories; the directory name is the language code, so the real-time application just concatenates the language code with the filename to retrieve the correct grammar. The third file is the 'prompt concept file' and it contains, for each input/output dialog, the association between 'prompt concept' (PC) and the name of the text concept or the audio file that has to be used for it (not the prompts for each language). The fourth file, called 'text concept file', holds the actual prompts (the real texts in SSML format as we mentioned above) that correspond to the text concepts defined in the 'prompt concept file'. Therefore, this file is language dependent and is repeated for every language and, again, is kept with the same name in separate directories. This separation might seem complicated, but it is the only way to ensure multilinguality and the flexibility to handle audio files, prompts, etc., in the same application.



```
<?xml version="1.0" encoding ="UTF-8"?>
<MED modality="Speech" id="EGBanking" >
 xDialogs
   Dialog Name = DGet_ATTR_TransactionAmount_IN_CLASS_Transaction  Realisation = med
     xPresentation
       PromptCall
         xPrompt
           PromptConcept Reference = DGet_ATTR_TransactionAmount_IN_CLASS_TransactionPC   1
     xFilling
       RecognizerCall
         xGrammar
           GrammarConcept Reference = DGet_ATTR_TransactionAmount_IN_CLASS_TransactionGC   2
         xAcceptors
           Variable Reference = TransactionAmount_IN_CLASS_Transaction
         xNomatchReaction
           PromptCall
             xPrompt
               PromptConcept Reference = DGet_ATTR_TransactionAmount_IN_CLASS_TransactionNoMatchPC   3
         xNoinputReaction
           PromptCall
             xPrompt
               PromptConcept Reference = DGet_ATTR_TransactionAmount_IN_CLASS_TransactionNoInputPC   4
   Dialog Name = DGet_AccountInfoType Realisation = med
   ..............
```

Fig. 9. GDialogXML code generated by the MEA for speech modality.

For the Web modality, similar files are generated, but now, instead of the 'grammar concept file' and 'prompt concept file', the files are the 'input concept file' and 'output concept file', again language independent, which describe the appearance of each input or output item (radio button, submit button, secret text, labels, combo boxes, lists, etc.) and also include a reference to the text concepts that they use, which are also specified in a 'text concept file' just like the speech modality.

5.4. Dialog model linker (DML)

This module generates one file for each selected modality where all the information from previous assistants is automatically linked together: dialogs, actions, input/output concepts, prompts and grammars, etc.

The final dialog model is a combination of the files produced by the RMA, the MERA and the MEA. Both kinds of model are linked together by filling different sections of GDialogXML dialog units, see (Hamerich et al., 2003) for further information.

5.5. Script generators

In this section, the modules that convert the dialogs coded in GDialogXML syntax into the execution scripts needed for each modality (VoiceXML and xHTML) are described. To carry out the process, they solve the problems and limitations of each standard and manage those issues regarding the handling of multilinguality, database access, the preparation of prompts or Web text, and the handling of concepts in the language-independent specification of the dialog.

One important issue is how the system handles in real-time a prompt that includes information returned by a database query. To solve this, the script generators include one variable in the script called *xLanguageId_global*, that codes the language of the service with an identifier in ISO639-1 format (language code) followed by an identifier ISO3166 (country code). This variable is set by the language identification module at the beginning of the session and is always passed as an argument to *all* database access functions specified in the DCMA, where it is concatenated with the field name that is going to be retrieved. Obviously, the database (there is a single database for all languages) should contain the same information for each language used in the service using fields with the same base name but different codes as suffix, e.g., info_text_en_UK for the field with the information in English, info_text_es_ES with the information in Spanish, etc. Once the query is made, the right variables are filled and the information is provided to the user in the correct language.

5.5.1. VoiceXML Generator and connection with the runtime platform

Using the file created by the linker (DML) in the previous step for the speech modality, this module generates a file in VoiceXML format for each language used in the service.

The script generator for VoiceXML has to overcome the limitations imposed by this language in its version 2.0. The main limitation is probably that VoiceXML does not allow returning calls (subroutines), which are needed to solve the problem of presenting lists of objects (see Section 5.2.1), as ordinary statements (returning calls are only allowed at certain positions). Therefore, all complex statements and value expressions have to be 'flattened' into simpler operations and into calls to intermediate dialogs that allow jumps to other states in dialog flow.

Besides, VoiceXML does not allow input fields to be global variables; however, we used global variables for over-answering so that they can be filled in previous states and do not lose their contents when jumping to other states. Therefore, a synchronizing strategy had to be implemented to map global variables to local input fields and vice versa. To handle over-answering it is also necessary that, if a slot is optional or is already filled, the recognition process may be omitted. In VoiceXML all the slots in a form must be filled, so we introduced additional intermediate variables and conditional blocks to find out whether the slots associated to over-answering are filled or not. Another issue is how to clear the content of a slot in a form and jump back to previous dialogs or states, which is another behavior needed for list handling, as the VoiceXML manager would automatically repeat the filling process for that slot and that is not the desired behavior. To solve this, the VoiceXML generator creates intermediate variables so that, when the slot has been cleared intentionally, the filling process is not repeated.

GDialogXML supports the idea of connecting services during runtime, e.g., services providing access to databases, services generating prompts on the fly, and so on. The VoiceXML generator implements these calls as HTTP requests from a CGI script. This CGI script works as a data bridge and contacts the actual services. It is needed because it has to produce VoiceXML code, since this is the only way to integrate dynamic data (result values) into the dialog flow. By using the bridge, the services are freed from the burden of producing VoiceXML themselves.

To generate prompts on the fly, we decided to use language-dependent JSGF grammars, in which the correspondence between prompt and concept is specified, and the recognizer would return in real-time the concept specified in the grammar instead of the prompt.

To increase performance, the VoiceXML generator uses a reference resolution strategy for result values of the runtime services. This means that if the result of a service request is a reference to a complex named object (e.g., a person), only the reference (consisting of the identifying attributes) is transmitted. At the time when details of the object are needed, the complete data structure of attribute values is transmitted. This is particularly important, when the result of a service request is a large list of references to complex data objects, which happens a lot when navigating through information bases.

Finally, the VoiceXML generator automatically creates global variables in the final script, where the dynamic runtime values returned by the corresponding modules, are kept to handle several aspects of the runtime system: the user level, the speaker Id, the confidence value from the last recognition, the current language, etc. The user level variable, for example, is needed for switching prompts depending on the user level, which is set by calling the User-Level-Detector runtime service.

In (Córdoba et al., 2004; Hamerich et al., 2003) other limitations of VoiceXML are described, together with some recommendations to improve the standard.

5.5.2. Web script generator

Using the file created by the linker (DML) in the previous step for the Web modality, this module generates a file in xHTML format for each language used in the service.

Unlike the voice modality, for Web the distinction between the flow control, the data and the presentation (buttons, images, frames, etc.) is not too clear. Nowadays, there are a lot of integrated development environments for the presentation part (Web editors), whereas for the control and data access they have to be specified using script languages (perl, php, python,...) or usual programming languages (Java, .Net,...). In other cases, the overall flow control is supported by frameworks (Jakarta Struts,...), but in general there is no widespread language, so the task of integrating all of them is difficult. Therefore, the objective of this assistant is not to compete with widespread Web editors but to provide a complementary support to try to facilitate the separation between the modeling of the flow control, the data and the presentation. To this end, the assistant automatically transforms the GDialogXML models related to input and output concepts into xHTML files with embedded xForms elements. This way, the generated files can be used as templates for Web designers who can add additional design elements (xHTML tags, images, styles, etc.), while the dialog flow is preserved separately. A runtime interpreter for GDialogXML may execute the dialog model "as is" in the Webserver environment to control the dialog flow and take care of the database transactions.

Thanks to this separation, the final script is platform independent and easily adaptable to multiple display devices (browsers, PDA, public terminals, etc.). Although there are some limitations with xForms as not all browsers support them, the use of plugins or rendering programs in the server provides that support. Besides, it is expected that the use of xForms becomes a Web standard for forms design, as part of the xHTML 2.0 specification, so its popularity will grow. The use of xHTML tags could favor the integration of the two modalities, in a future development, so that they can work at the same time using the standard X + V (xHTML, 2004).

5.6. Auxiliary assistants

Besides the assistants described above, there are some other assistants that complement the work needed for voice modality.

The first one is the vocabulary builder (VB) which prepares the vocabularies that will be used by the recognizer. Thus, this component gets input from the language model resources and produces the lexicon. The lexicon contains the phonetic transcription of each word and in most cases the phonetic alternatives. There are equivalent dictionaries for each of the different languages allowed in the platform.

A second assistant is the Language Modeling Toolkit (LMT) that allows the designer to specify the language models that will be used in the runtime system to "understand" the different user answers to the system questions. The assistant allows the creation and edition of grammars in JSGF (Hunt, 2000) both for recognition and for prompt generation using the Natural Language Generation (NLG) module.

Finally, in order to be able to edit the different GDialogXML models as produced by the assistants of the AGP, a separate component has been developed. This assistant is called DiaGen and allows the creation and edition of GDialogXML models and libraries by providing auto-complete templates and other editing functionalities.

6. Portability and use of standards

Amongst the main objectives of the platform were portability, meaning independence of the operating system and the runtime platform, scalability, widespread use of standards and feasibility to use existing or new technologies. The main efforts made in this direction are mentioned below.

6.1. OpenVXI

To test the VoiceXML script generated by the platform, an interpreter to execute it in a runtime platform is needed. We selected OpenVXI 2.0.1 from Scansoft (OpenVXI Web page, 2004) because it is an open source solution and, thanks to its portability, it does not impose any recognition or text-to-speech engine nor any specific telephone platform, so it can be adapted to any runtime platform. Besides, it provides an important part of the functionality required to execute dialog applications, namely an XML interface that processes the VoiceXML script, JavaScript API, WWW functions API and Register API (Eberman et al., 2002). Regarding the functions related to input/output (recognition, text-to-speech and telephone control) it provides interfaces that can be modified and completed to adapt them to the needs of any platform.

An additional advantage of this interpreter was that it allowed us to work with our own technology in all respects, so the tests could be controlled even further. In (Córdoba et al., 2004) some proposals of improvements and adaptations are described, which we have had to make to the interpreter to adapt it to our runtime system.

6.2. Programming environment and other standards

All the graphic components of the platform have been programmed and generated using Qt from Trolltech, which is a multi-platform (Linux, Windows, Mac, X11) integrated development environment, compatible with C++, with which the designer can write code that can be executed in different operating systems and development environments, e.g., for Visual Studio, Visual.Net, Borland, just by recompiling. Moreover, Qt provides several methods and tools to translate all texts in the graphical interface to adapt them to another language.

We have also used the UTF-8 format (8-bit Unicode Transformation Format), which is a variable-length character coding for the Unicode standard in multiple languages. Besides, it is the default coding in XML and it is used by all Internet protocols. This format is crucial in the definition of prompts/grammars for the multilingual service.

The platform also uses some ideas extracted from the USI project (Universal Speech Interface) (Toth et al., 2002) that are applied in the generation of names in automatic dialogs, as they are derived from the attributes in the database, so that they are easily recognizable. Furthermore, we also use keywords for some universal commands available in the runtime system.

Finally, as we mentioned in Section 5.3, to generate the prompts used by the text-to-speech system in the voice modality we have adapted the platform and the runtime system so that they could process the SSML format (Speech Synthesis Markup Language) (Burnett et al., 2002).

7. Evaluation and test applications

7.1. Subjective evaluation of the platform

To rate the acceptance and friendliness of the platform for the rapid and efficient generation of multilingual/multimodal dialog applications, we carried out a subjective evaluation with 41 subjects (24 novice in dialog applications design, 11 intermediate and 6 experts) from Greece, Germany and Spain, with ages ranging from 21 to 49 years and, in general, with basic knowledge of programming.

A short introductory tutorial was given to the participants and then they were asked to carry out predefined tasks of dialog design in each assistant. These tasks were part of the design of an overall application, e.g., get the information about a house loan, do a transaction of a certain amount between two accounts using mixed initiative, obtain the current value of a currency, etc., and covered almost 90% of the functionality of the AGP, obviously including all the AGP assistants. The participants had to use all of them in turn: they created the data model for three complex classes, prepared at least three database access functions, designed at least four states in the SFM, completed the states in the RMA to carry out the mentioned tasks, they created and reused some libraries, etc. The reason not to make a complete system from scratch was to reduce the evaluation time to a limit of 3–4 h, including time for the tutorials.

Finally, they had to answer a questionnaire that consisted of two parts: one to evaluate each assistant individually and the other to evaluate the platform as a whole, with scores between 1 and 10 representing very poor and excellent rates respectively. In Table 1, the results are shown.

The overall score was an average of 8.37, with the maximum scores in the following aspects:

- Speeding up the development time of an application
- Over-answering and Mixed initiative functionality
- Lists handling for speech applications

Table 1
Subjective evaluation of the platform

Question	Average rating
The provision of data modeling and connecting to external data sources	8.7
The provision of application state flow modeling	8.6
Easy adaptability to other languages	8.2
Easy adaptability to other modalities	7.9
Ready-made error-handling (nomatch, noinput)	8.1
Speed up of development time as compared to writing VoiceXML/+xHTML code by hand	9.0
Provision of user modeling	7.8
Provision of mixed-initiative dialog handling	8.5
Provision of list handling	8.6
Provision of over-answering	9.0
Provision of easy connection to run-time modules	7.9

Regarding the results for the specific assistants, in general, all assistants have average qualifications between 7 and 8.5, which are very homogeneous. We should mention some remarks from the evaluators regarding the RMA and the MEA. The RMA was rated as having a very good overall appearance and extremely good functionality. However, it had some difficulties in learning and was perceived as less intuitive. The most probable reason is that it is the assistant that provides the biggest functionality for dialog design, as it is the place where detailed design is made, and the evaluating designers were given very little time to learn each assistant and practice them, so in practice evaluators only read part of the documentation written about the designs they were asked to do. In the case of the MEA, the low results are probably the result of the time required to set the prompts and grammars (text typing), especially for different language definition where machine translation and more prompt and grammar libraries would be desirable.

7.2. Services tested in the runtime system

To evaluate and validate the efficiency of the platform and its service independence, two main test applications were set up.

The first one is a banking application called EGBanking, with several services: general information (deposit products, loans, cards, e-banking or exchange rates), an authentication process where the user inputs the account number and the PIN, personal information (personal accounts and cards, last transactions, balance) and a transaction dialog to transfer money between accounts or to make payments (VAT, Social Security, public pension, TAX Income, credit card and other funds payments). It has been implemented in four languages: Greek, German, English, and Spanish. It has been running since September 2003 and is used as a commercial product by a Greek bank (Egnatia bank Web page, 2004) for its registered clients with very good results.

The second application, called CitizenCare, offers basic voice information retrieval system functionality in the context of public authorities. The caller can either ask for solutions for dedicated concerns or for general information about certain departments such as addresses, phone numbers or opening hours. Specific information may be retrieved in one call. This application was developed for German and English and for both modalities, voice and Web.

8. Future plans

Considering the good results that we have obtained in the evaluation and test applications, the growing demand of better and more complex dialog systems, and to broaden the functionality of this kind of tool, we are considering the following improvements to the platform:

- Increase the number of automatically generated dialogs making a more complex and exhaustive analysis of database structure. These dialogs would consist of multiple actions grouped by the assistant, but with the possibility of being edited by the designer.
- Increase the number of libraries available with the platform.
- Provide more flexibility in the confirmation handling for the voice modality by using more options adapted to a specific data type and providing more fine-tuning possibilities.
- Implement new strategies to reduce design time, e.g., in the generation of the structure and access functions of the Data Model, and in the generation of grammars and prompts.
- Integration of the current two modalities so that they can work at the same time through the X + V standard (xHTML, 2004).

9. Conclusions

Throughout the Gemini project we have studied different systems and strategies for the design of human-computer dialog applications and as a result we have developed an application generation platform which is both powerful and flexible, with a high degree of automation, allowing the generation of state of the art speech and Web based applications. The platform architecture is able to generate in a semiautomatic way valid dialogs for multiple languages and two modalities using just a description of the database, a basic state flow of the application and a simplified user-friendly interaction with the designer. In this paper we have shown some strategies to speed up the design process, as the possibility to handle mixed initiative and over-answering dialogs using the same framework. Detailed procedures to handle list and confirmation handling have been presented too. Features like user modeling, speaker verification, language identification can be included easily through runtime modules included in the platform. Moreover, the use of standards like VoiceXML and xHTML keeps the platform open for further development, and allows the use of other tools and existing technology. Another important result of the project is the newly designed abstract dialog description language, called GDialogXML. Furthermore, we have carried out a subjective evaluation and developed two test applications that demonstrate the user and designer-friendliness and robustness of the platform, as well as the fulfillment of all objectives initially planned, together with a proposal for improvements and future plans for the platform.

Acknowledgements

We want to thank the following people for their contribution in the design and development of this platform. To Fernando Fernández, Valentín Sama-Rojo, Ignacio Ibarz, and Javier Morante from the Universidad Politécnica de Madrid (GTH, Spain) for their contribution in the coding of this platform and the runtime system. To Yu-Fang Helena Wang from TEMIC Speech Dialog Systems (Germany) for evaluating all the CitizenCare applications, to Thomas Eppelt, Manfred Paula and Robert Stegmann from FAW (Germany) for setting up the backend database of the CitizenCare application, to Nikos Katsaounos and Alex Sarafian from Knowledge S.A., and to Georgila Kalliroi, Todor Ganchev and Nikos Fakotakis from Patras University (Greece) for their work in several assistants. And we cannot forget Gianna Tsakou and Anastasios Tsopanoglou for their excellent job as project coordinators. Finally, we want to thank the reviewers of this paper for their comments and appropriate suggestions.

References

Allen, J., Guinn, C., Horvitz, E., 1999. Mixed-initiative interaction. IEEE Intelligent Syst. 14 (5), 14–23.

Allen, J., Byron, D., Dzikovska, M., Ferguson, G., Galescu, L., 2001. Towards conversational human–computer interaction. AI Magazine 22 (4), 27–37.

Almeida, L. et al., 2002. Implementing and evaluating a multimodal and multilingual tourist guide. In: Proc. Internat. CLASS Workshop on Natural, Intelligent and Effective Interaction in Multimodal Dialogue Systems, pp. 1–7.

Bennett, C., Llitjós, A.F., Shriver, S., Rudnicky, A., Black, A.W., 2002. Building VoiceXML-based applications. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. 2245–2248.

Bohus, D., Rudnicky, A.I., 2003. RavenClaw: dialog management using hierarchical task decomposition and an expectation agenda. In: Proc. 8th European Conf. on Speech Communication and Technology (Eurospeech), pp. 597–600.

Burnett, D.C., Walker, M.R., Hunt, A., 2002. Speech Synthesis Markup Language Version 1.0. W3C Working Draft. Available from: <http://www.w3.org/TR/speech-synthesis>.

Cole, R., 1999. Tools for research and education in speech science. In: Proc. Internat. Conf. of Phonetic Sciences (ICPhS), pp. 1277–1280.

Córdoba, R., San-Segundo, R., Montero, J.M., Colás, J., Ferreiros, J., Macías-Guarasa, J., Pardo, J.M., 2001. An interactive directory assistance service for Spanish with large-vocabulary recognition. In: Proc. 7th European Conf. on Speech Communication and Technology (Eurospeech), Vol. II, pp. 1279–1282.

Córdoba, R., Fernández, F., Sama, V., D'Haro, L.F., San-Segundo, R., Montero, J.M., 2004. Implementation of dialog applications in an open-source VoiceXML platform. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. I-257–I-260.

Denecke, M., 2002. Rapid prototyping for spoken dialogue systems. In: Proc. 19th Internat. Conf. on Computational Linguistic (COLING'02).

D'Haro, L.F., de Córdoba, R., San-Segundo, R., Montero, J.M., Macías-Guarasa, J., Pardo, J.M., 2004. Strategies to reduce design time in multimodal/multilingual dialog applications. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. IV-3057–3060.

Eberman, B., Carter, J., Goddeau, D., 2002. Building VoiceXML browsers with OpenVXI. In: Proc. 11th Internat. Conf. on World Wide Web, pp. 713–717.

Egnatia bank Web page, 2004. Available from: <http://egnatiasite.egnatiabank.gr/EN/default.htm>.

Ehrlich, U., Hanrieder, G., Hitzenberger, L., Heisterkamp, P., Mecklenburg, K., Regel-Brietzmann, P., 1997. ACCeSS—automated call center through speech understanding system. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 1819–1822.

Flippo, F., Krebs, A., Marsic, I., 2003. A framework for rapid development and multimodal interfaces. In: Proc. Internat. Conf. on Multimodal Interfaces, pp. 109–116.

Gemini Project Homepage, 2004. Available from: <http://www-gth.die.upm.es/projects/gemini/>.

Glass, J., Weinstein, E., 2001. SPEECHBUILDER: facilitating spoken dialogue system development. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 1335–1339.

Gustafson, J., Elmberg, P., Carlson, R., Jonsson, A., 1998. An educational dialogue system with a user controllable dialogue manager. In: Proc. Internat. Conf. on Spoken Language Processing (ICSLP), pp. 33–37.

Gustafson, J., Bell, L., Beskow, J., Boye, J., Carlson, R., Edlund, J., Granström, B., House, D., Wiren, M., 2000. AdApt—a multimodal conversational dialogue system in an apartment domain. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. II-134–137.

Hamerich, S.W., Wang, Y.-F., Schubert, V., Schless, V., Igel, S., 2003. XML-based dialogue descriptions in the Gemini Project. In: Proc. Berliner XML-Tage, pp. 404–412.

Hamerich, S.W., Schubert, V., Schless, V., Córdoba, R., Pardo, J.M., D'Haro, L.F., Kladis, B., Kocsis, O., Igel, S., 2004a.

Semi-automatic generation of dialogue applications in the Gemini Project. In: Proc. Workshop for Discourse and Dialogue (SigDial), pp. 31–34.

Hamerich, S.W., de Córdoba, R., Schless, V., D'Haro, L.F., Kladis, B., Schubert, V., Kocsis, O., Igel, S., Pardo, J.M., 2004b. The Gemini Platform: semi-automatic generation of dialogue applications. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. IV-2629–2632.

Hunt, A., 2000. JSpeech Grammar Format. W3C Note. Available from: <http://www.w3.org/TR/jsgf>.

Johnston, M., Bangalore, S., et al., 2002. MATCH: an architecture for multimodal dialogue systems. In: Proc. 40th Annual Meeting of the ACL, pp. 376–383.

Katsurada, K., Ootani, Y., Nakamura, Y., Kobayashi, S., Yamada, H., Nitta, T., 2002. A modality independent MMI system architecture. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. 2549–2552.

Klemmer, S.R., Sinha, A K., Chen, J., Landay, J.A., Aboobaker, N., Wang, A., 2000. SUEDE: a Wizard of Oz prototyping tool for speech user interfaces. In: CHI Letters, ACM Symposium on User Interface Software and Technology (UIST), Vol. 2 (2), pp. 1–10.

Komatani, K., Ueno, S., Kawahara, T., Okuno, H.G., 2003. User modeling in spoken dialogue systems for flexible guidance generation. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 745–748.

Lamel, L., Rosset, S., Gauvain, J.L., Bennacef, S., Garnier-Rizet, M., Prouts, B., 2000. The LIMSI ARISE system. Speech Commun. 31 (4), 339–354.

Lehtinen, G., Safra, S., Gauger, M., Cochard, J.-L., Kaspar, B., Hennecke, M.E., Pardo, J.M., Córdoba, R., San-Segundo, R., Tsopanoglou, A., Louloudis, D., Mantazas, M., 2000. IDAS: interactive directory assistance service. In: Proc. COST249 ISCA Workshop on Voice Operated Telecom Services (VOTS-2000), pp. 51–54.

Levin, E., Narayanan, S. Pieraccini, R., Biatov, K., Bocchieri, E., Di Fabbrizio, G., Eckert, W., Lee, S. Pokrovsky, A., Rahim, M., Ruscitti, P., Walker, M., 2000. The AT&T-DARPA communicator mixed-initiative spoken dialog system. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), Vol. 2, pp. 122–125.

McGlashan, S., Burnett, D.C., Carter, J., Danielsen, P., Ferrans, J., Hunt, A., Lucas, B., Porter, B., Rehor, K., Tryphonas, S., 2004. Voice Extensible Markup Language (VoiceXML) Version 2.0. W3C Recommendation. Available from: <www.w3.org/TR/voicexml20>.

McTear, M., 1998. Modelling spoken dialogues with state transition diagrams: experiences with the CSLU toolkit. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. 1223–1226.

McTear, M., 1999. Software to support research and development of spoken dialogue systems. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 339–342.

McTear, M., 2002. Spoken dialogue technology: enabling the conversational user interface. ACM Comput. Surveys 34 (1), 90–169.

Meng, H.M., Lee, S., Wai, C., 2002. Intelligent speech for information systems: towards biliteracy and trilingualism. Interacting with Computers 14 (4), 327–339.

Nigay, L., Coutaz, J., 1993. A design space for multimodal systems—concurrent processing and data fusion. In: Proc. INTERCHI—Conf. on Human Factors in Computing Systems, pp. 172–178.

OpenVXI Web page, 2004. Available from: <http://fife.speech.cs.cmu.edu/openvxi/>.

Oviatt, S.L. et al., 2000. Designing the user interface for multimodal speech and gesture applications: state-of-the-art systems and research directions. J. Human Comput. Interact. 15 (4), 263–322.

Pargellis, A.N., Kuo, H.J., Lee, C., 2004. An automatic dialogue generation platform for personalized dialogue applications. Speech Commun. 42, 329–351.

Polifroni, J., Seneff, S., 2000. Galaxy-II as an architecture for spoken dialogue evaluation. In: Proc. Internat. Conf. on Language Resources and Evaluation (LREC), pp. 725–730.

Polifroni, J., Chung, G., Seneff, S., 2003. Towards the automatic generation of mixed-initiative dialogue systems from Web content. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 193–196.

Rudnicky, A., Xu, W., 1999. An agenda based dialog management architecture for spoken language systems. In: IEEE Automatic Speech Recognition and Understanding Workshop, pp. 337–340.

San Segundo, R., Montero, J.M., Colás, J., Gutiérrez, J.M., Ramos, J.M., Pardo, J.M., 2001. Methodology for dialogue design in telephone-based spoken dialogue systems: a Spanish train information system. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 2165–2168.

Schubert, V., Hamerich, S.W., 2005. The dialog application metalanguage GDialogXML. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 789–792.

Seneff, S., Polifroni, J., 2000. Dialogue management in the mercury flight reservation system. In: Proc. ANLP-NAACL Satellite Workshop, pp. 1–6.

Strik, H., Russel, A., van den Heuvel, H., Cucchiarini, C., Boves, L., 1997. A spoken dialog system for the Dutch public transport information service. Int. J. Speech Technol. 2 (2), 121–131.

Toth, A.R., Harris, T.K., et al., 2002. Towards every-citizen's speech interfaces: an application generator for speech interfaces to databases. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. 1497–1500.

Turunen, M. et al., 2004. AthosMail—a multilingual adaptive spoken dialogue system for e-mail domain. In: Proc. Workshop on Robust and Adaptive Information Processing for Mobile Speech Interfaces.

Uebler, U., 2001. Multilingual speech recognition in seven languages. Speech Commun. 35 (1), 53–69.

W3C, 1999. Available from: <http://www.w3.org/TR/voice-dialog-reqs/>.

Wahlster, W., Reithinger, N., Blocher, A., 2001. SmartKom: Multimodal communication with a life-like character. In: Proc. European Conf. on Speech Communication and Technology (Eurospeech), pp. 1547–1550.

Wang, K., 2000. Implementation of a multimodal dialog system using extended markup languages. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), Vol. 2, pp. 138–141.

Wang, K., 2002. Salt: a spoken language interface for Web-based multimodal dialog systems. In: Proc. Internat. Conf. of Spoken Language Processing (ICSLP), pp. 2241–2244.

Wang, Y.-F.H., Hamerich, S.W., Schless, V., 2003. Multi-modal and modality specific error handling in the Gemini Project. In:

Proc. Workshop on Error Handling in Spoken Dialogue Systems, pp. 139–144.

xHTML+Voice, 2004. Available from: <http://www.voice-xml.org/specs/multimodal/x+v/12/spec.html>.

Zue, V., Seneff, S., Glass, J., Polifroni, J., Pao, C., Hazen, T.J., Hetherington, L., 2000. JUPITER: a telephone-based conversational interface for weather information. IEEE Trans. Speech Audio Process. 8 (1), 85–96.