Design and evaluation of acceleration strategies for speeding up the development of dialog applications

Luis Fernando D'Haro *, Ricardo de Córdoba, Rubén San-Segundo, Javier Ferreiros, José Manuel Pardo

Grupo de Tecnología del Habla, Universidad Politécnica de Madrid, Madrid, Spain

Abstract

In this paper, we describe a complete development platform that features different innovative acceleration strategies, not included in any other current platform, that simplify and speed up the definition of the different elements required to design a spoken dialog service. The proposed accelerations are mainly based on using the information from the backend database schema and contents, as well as cumulative information produced throughout the different steps in the design. Thanks to these accelerations, the interaction between the designer and the platform is improved, and in most cases the design is reduced to simple confirmations of the "proposals" that the platform dynamically provides at each step.

In addition, the platform provides several other accelerations such as configurable templates that can be used to define the different tasks in the service or the dialogs to obtain or show information to the user, automatic proposals for the best way to request slot contents from the user (i.e. using mixed-initiative forms or directed forms), an assistant that offers the set of more probable actions required to complete the definition of the different tasks in the application, or another assistant for solving specific modality details such as confirmations of user answers or how to present them the lists of retrieved results after querying the backend database. Additionally, the platform also allows the creation of speech grammars and prompts, database access functions, and the possibility of using mixed initiative and over-answering dialogs. In the paper we also describe in detail each assistant in the platform, emphasizing the different kind of methodologies followed to facilitate the design process at each one.

Finally, we describe the results obtained in both a subjective and an objective evaluation with different designers that confirm the viability, usefulness, and functionality of the proposed accelerations. Thanks to the accelerations, the design time is reduced in more than 56% and the number of keystrokes by 84%.

Keywords: Development tools; Automatic design; VoiceXML; Data mining; Speech-based dialogs

1. Introduction

The current increasing demand of automatic dialog systems for different domains and user requirements has resulted in several companies and academic institutions working on the development of fully integrated platforms that need to provide the maximum number of features to designers and final users, a high level of portability, standardization and scalability in order to minimize design time and costs. Moreover, these platforms have to enable the rapid development and maintenance of automatic dialog services, as well as being flexible enough to allow the creation of a wide range of services and to be adapted to the special characteristics of each one. In general, these platforms are made up of different and independent assistants

* Corresponding author. Address: ETSI Telecomunicación, Ciudad Universitaria s/n, 28040 Madrid, Spain. Tel.: +34 91 453 35 43; fax: +34 91 3367323.

E-mail addresses: lfdharo@die.upm.es (L.F. D'Haro), cordoba@die.upm.es (R. de Córdoba), lapiz@die.upm.es (R. San-Segundo), jfl@die.upm.es (J. Ferreiros), pardo@die.upm.es (J.M. Pardo).

that allow collaborative role-based development so that different developers teams can work on the same project at the same time. Finally, the usability of these platforms is increased thanks to a clear and fully integrated graphical user interface, as well as the incorporation of built-in libraries and out-of-the-box dialog components that allows previous knowledge to be reused and an easy deployment of the service.

1.1. Strengths and weaknesses of commercial and academic platforms

In their effort to speed up the design of dialog applications, most of the commercial platforms (e.g. Nuance V-builder,[1] IBM Web-Sphere,[2] Audium Studio,[3] Envox,[4] etc.) include state-of-the-art modules such as speech recognizers, high quality speech synthesizers, language identification modules, etc., as well as using widespread standards such as VoiceXML, SALT, CCXML, etc. These platforms also include a large number of predefined libraries for typical dialogs such as requesting addresses or social security numbers. In addition, they incorporate assistants for debugging and logging the service. Finally, these platforms provide user-friendly graphical interfaces that simplify the development of very complex applications. On the other hand, a large drawback they present is that the behavior of the service may change across different platforms because of the use of attributes or features not supported in most platforms (e.g. including non-standard tags in the VoiceXML script to allow sending faxes or playing videos) or because they use advanced runtime modules (e.g. automatic speech recognizers, text-to-speech, language identification or speaker identification) that can reduce the necessity of coding many actions in the scripts. In addition, it is difficult to integrate proprietary modules and they do not provide automatic proposals for defining the dialog flow. Finally, it is difficult to integrate new modalities, create the service in multiple languages, adapt the service according to predefined user profiles, or obtain the same functionalities on different operating systems.

In contrast to commercial platforms, academic and research platforms (e.g. CSLU-RAD,[5] DialogDesigner,[6] Trindikit,[7] RavenClaw,[8] etc.) do not necessarily incorporate all of the aforementioned features. However, they allow more complex dialog interactions (e.g. incorporating the possibility of changing the dialog goal at any moment and then recovering it later (Bohus and Rudnicky, 2009), allowing users to interact with the final system using several different modalities at the same time (Tsai, 2006), or allowing complex confirmation strategies for error handling (McTear et al., 2005)); in addition, some of them are available as open source and can be extended using third party modules. The main drawback is that they may have serious limitations such as a low portability level as they are tied to specific runtime platforms which make them difficult to integrate with other systems and/or architectures; besides, many of their interesting features are not easily available, therefore they are only used by advanced developers. The number of different services and capabilities that they can offer to the final users and programmers is also usually low. They also require the designer to know several programming languages and non-standard formats thus reducing their usability. Finally, they may present limitations for implementing dialog strategies that take into account the user experience, different modalities, and languages required by the service.

In spite of these features, interestingly, both kinds of platform lack accelerations (i.e. mechanisms to automate or simplify the design of the dialog service) based on basic business intelligence and data mining methodologies applied to the contents of the task database and from the data model structure (i.e. the set of object-oriented classes and attributes that model the database tables and fields and their relationships). To cope with this issue, our objective was to define and use dynamic and intelligent acceleration strategies so that we can, among other things, predict the necessary information required to complete the definition of a state, accelerate the specification of the application flow, the definition of the database access functions, and to help designers with built-in solutions, not forcing them to define all this information from scratch. For a more detailed description of the capabilities provided by current commercial and academic platforms please refer to Section 2.1 and Appendix B in (D'Haro et al., 2004).

1.2. Incorporation of database contents information in the design

Although the database content or structure is rarely used for accelerating the definition of the dialog flow, in the literature we can find examples of use in other stages in the design.

In (Polifroni and Walker, 2006) a rapid development environment for speech dialogs from online resources is described. Here the goal is to reduce the need to specify a pre-defined dialog flow. Therefore, the flow is dynamically built based on an analysis of the retrieved data at every turn, as the user provides new constraints. For instance, here the database contents are used to create clusters of numeric fields in order to establish subjective ranges that the users can use in their answers such as "near" or "cheap/expensive", in the domain of a hotel reservation, that change depending on the city. This way, if the database contains information about the average price of a room for each hotel and for different cities, it is possible to automatically classify which hotels are "cheap" from those that are "expensive" and include this

[1] http://www.nuance.com.
[2] http://www-01.ibm.com/software/voice/.
[3] http://www.audiumcorp.com/Audium_Studio/.
[4] http://www.nuxiba.com/envox.html.
[5] http://cslu.cse.ogi.edu/toolkit/.
[6] http://spokendialog.dk/DialogDesigner/.
[7] http://www.ling.gu.se/projekt/trindi/trindikit/.
[8] http://wiki.speech.cs.cmu.edu/olympus/index.php/RavenClaw.

information in the database. At each turn the system also uses the retrieved results to generate and select, on the fly, the prompts to summarize the retrieved results or to suggest new constraints.

In (Pargellis et al., 2004) the dialog flow is dynamically modified through a set of templates adapted to the final user of the system, as well as with the available information and services. The system uses the dynamic contents of the database to create, on the fly, new grammars and prompts, as well as the dialog flow for presenting information to the user, or for solving errors, through predefined templates and according to the user profile.

In (Chung, 2004) the database is used together with a simulation system in order to generate thousands of unique dialogs that can be used to train the speech recognizer and the understanding module, as well as diagnosing the system behavior against problematic user interactions or for unexpected user answers. In (Wang and Acero, 2006) the system generates a large number of artificial sentences using the database contents and sentences from other domains by applying syntactic and semantic information that are used to improve and create new language models for the speech recognition system.

Feng et al. (2003) proposes a very different approach, not using a database but mining the contents of corporate websites for automatically creating spoken and text-based dialog applications for customer care. After analyzing the content and structure of the website, the dialog manager, at runtime, will identify the focus or expectations of the user question and will provide a concise answer. Although the dialog flow is not defined using any GUI, the paper proves that important knowledge can be extracted from well-designed contents as we have done.

In (D'Haro et al., 2006), we described our initial steps to include several acceleration strategies to the design, based mainly on exploiting the structure of the backend database and with a special emphasis in proposing accelerations for the assistant used to define the dialog flow at a high level (i.e. modality and language independent, see Section 3.5). In the current paper, we describe new strategies that exploit the database contents and schema incorporating them in diverse ways. For instance: (a) for creating different kind of templates that can be used to define the dialog flow (Section 3.4.2) or the actions to be done at each state (Section 3.5). (b) To propose which slots should be requested at the same time to the users or one by one considering mainly the difficulty of the speech recognizer to correctly recognize them (Section 3.4.3). (c) To reduce the information displayed to the designer in the different assistants of the platform (Section 3.2). (d) To simplify the process of debugging the database access functions used by the real-time system and automatically proposed by the platform (Section 3.3.2).

1.3. Platform background and limitations

Taking into account the limitations of the best commercial and research platforms, the scant use of database content information in the design, as well as the limited number of research projects for creating, accelerating, and improving these design platforms, we undertook the GEMINI European Project (GEMINI, 2011). The final result was a complete, flexible, and highly automated development platform consisting of a set of tools and agents that guide the design process and allow the definition of the different levels of knowledge needed to complete and run the state-of-the-art speech and Web-based services. The platform allows the creation of a wide range of applications to access database centered services such as the ones provided in banking transactions, transport reservations, information kiosks, etc. through a Web browser or a telephone.

In (D'Haro et al., 2006, 2004) we describe in detail the initial platform, our efforts in separating the general and high-level definition of the dialog flow from the specific details imposed by each modality, language and user profile, as well as the differences between operating systems and runtime platforms by using several standard languages. Finally, we also describe our first attempts to accelerate the design using only information from the data model structure and by proposing different kinds of actions for completing the dialog flow.

After finishing the project, we decided to continue working on the platform in order to propose new accelerations strategies and improving its capabilities. The main new improvements described in this paper can be summarized as follows:

(1) Incorporation of heuristic information extracted from analyzing the contents of the backend database. This information is used later onto speed up the design of the database schema (Section 3.2), or to suggest when two or more data (slots) should be requested to the users together or one by one (Section 3.4.3).
(2) Incorporation of two new wizard windows to help designers to automate/eliminate repetitive or common procedures in the design. The first one allows the creation of complex classes and attributes when defining the database schema (Section 3.2), and the second one provides automatic proposals of SQL queries to access the backend database at runtime (Section 3.3.2). Finally, we have also redesigned the GUI of the assistant used to define the application flow, including also some algorithms and strategies to improve the visualization of the workspace used to show the states and transitions in the dialog application (Section 3.4.1).
(3) Integration of the runtime system into a distributed platform allowing the use of third party modules for the ASR, TTS, or voice browser (Section 2.4).
(4) Finally, we have also incorporated new several configurable templates based on the database schema and access functions to accelerate the creation of the states in the dialog flow (Section 3.4.2).

It is important to mention that we have focused a lot on proposing generic strategies that could be useful for a great variety of services and tasks where the users can modify or obtain information stored in a database. For example, the platform allows the creation of applications such as a banking application, a travel agency, a remote access to an agenda or phone directory, a command control device, or for appointment reservation, among others. In general, these are the kind of services that can be created considering the capabilities and limitations of the VoiceXML and xHTML standards generated by the platform. On the other hand, since many of the new strategies are based on using heuristic information from the backend database contents, it is clear that these strategies will be limited by the number of tables and records available in the database. In order to increase the robustness of the proposed accelerations, some of them allow the configuration of different parameters that the designer can adjust according to the requirements of each task (e.g. number of relevant tables, capabilities and expected performance of the speech recognizer, vocabulary size, etc.). Finally, we want to mention one current limitation of our platform is that we do not consider the possibility of using key semantic terms (such as "cheap", "near", etc., as used in (Polifroni and Walker, 2006)). As we describe in Section 3.1 this limitation can be solved in a future version of the platform.

1.4. Relevant definitions

Throughout this paper we are going to use some terms that we want to clarify beforehand from the perspective of our platform since they do not necessarily present a generally accepted definition.

Slot: This term will refer to any compulsory information that the system requests from the user.
Action: This term will refer to any kind of procedure (e.g. calls to other dialogs, calls to database access functions, arithmetic or string operations, programming constructs, etc.) required to complete the '*states*' in the application.
Dialog: This term will refer, as in VoiceXML, to the specific form or turn where the information is provided or requested to/from the user.
State: This term will refer, like in the dialog and automata theory, to one of all the possible nodes or states in a finite state based dialog system. However, in our platform we have extended this concept considering that a state does not represent a single dialog or action but that it is a group of dialogs or actions. This extension to the concept allows us to reduce the complexity of understanding and visualizing the whole application flow to a reduced number of '*states*' instead of hundreds or thousands of actions.
Acceleration: This term will refer to the different methodologies implemented in the assistants of the platform in order to reduce the design time and facilitate the definition of the different actions required to design and run the service.
Mixed-initiative and Over-answering: Following the definition of the VoiceXML standard (McGlashan et al., 2004), the term *mixed initiative* will indicate the system's ability to ask for two or more compulsory data from the user simultaneously, and, if the user's answer is incomplete or wrong new sub-dialogs are started in order to obtain the corresponding data. *Over-answering* will indicate the user's ability to provide additional data – not compulsory at the current state – to the system.

1.5. Paper organization

The paper is organized as follows: in Section 2 we present an overall description of the platform architecture, the main assistants and layers that makes it up, its scope and limitations. Section 3 describes the main accelerations in the platform and the assistants that include them; then, in Section 4 we will show the results of a subjective and objective evaluation of the platform carried out with different designers. Finally we will show our conclusions and future work in Section 5.

2. Platform structure

Fig. 1 shows the architecture and main assistants and tools that make up the Application Generation Platform (AGP). The platform consists of three main layers integrated into a common graphical user interface (GUI) that guides the designer step-by-step and lets him go back and forth. The three layers separate the aspects that are service specific (general characteristics of the application, database structure and access), those corresponding to the high level dialog flow of the application (modality and language independent), and the specific details imposed by each modality and language. This distribution also helps the designer to create several versions of the same service (for different modalities and languages) in a single step at the intermediate level. In the figure, the assistants in yellow are those that have been recently modified or extended in relation to previous versions of the platform, described in (D'Haro et al., 2006), and in those white have not been modified at all. Detailed information will be provided for the former.

In order to ease the communication and sharing of information between all the assistants, the platform uses an object oriented abstract language called GDialogXML[9] (Gemini Dialog XML) (see Schubert and Hamerich, 2005; Hamerich et al., 2003). This XML language allows the definition of all the application data, e.g. database access functions, variables and actions needed in each dialog, prompts and grammars, user models, Web graphical interfaces, etc. After finishing the design, the platform uses all the gener-

[9] http://www-gth.die.upm.es/projects/gemini/.

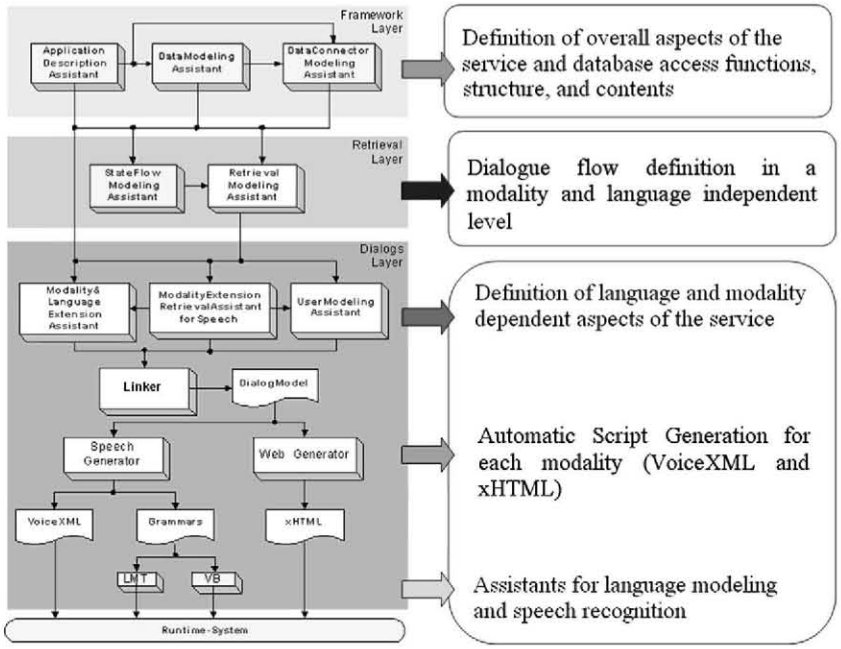

Fig. 1. Platform architecture.

ated XML files to convert them into the languages used for the runtime scripts according to the modality (VoiceXML and/or xHTML).

Before starting to describe the layers and assistants in detail, we want to emphasize their goal and the current limitations. As we mentioned in the introduction, 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 transactions, transport reservations, information kiosks, etc.) through a Web browser or telephone separately, although it should be possible to execute them simultaneously by incorporating new code elements for synchronization in our XML syntax and a new code generator (e.g. for $X+V$). It is also also important to consider the limitations imposed mainly by the VoiceXML 2.0 and xHTML scripts generated by the platform.

2.1. Framework layer

In the framework layer, the designer specifies the overall aspects related to the application and the data involved. This layer includes the *Application Description Assistant* (ADA) that is used to define the overall aspects of the service such as the number of modalities and languages, the database connection settings (e.g. total number of connection errors, timeouts, URL of the database server). For the speech modality the following information is defined: the timeout values for events such as no input, default confidence levels for speech recognition, maximum number of repetitions/errors before transferring the call to the operator, etc.; and for the Web modality, handling of errors such as page not found, non-authorized, or timeouts. Finally, the designer specifies the libraries that will be used throughout the design process, e.g. database access functions, list of prompts and grammars for each language.

In the *Data Model Assistant* (DMA) the designer defines the data structure (i.e. data model or schema) of the service specifying the classes, including inheritance, attributes and types that make up the database; the assistant also extracts heuristic information from the database contents. The objective of these classes is to provide information about which tables and fields in the database are relevant for the service and how the fields can be grouped together into classes. Therefore, we can think that the attributes in a class correspond to the possible database fields that can be requested or presented to the user, as well as how these attributes relate to the actual database tables and fields.

Fig. 2 shows an example of some classes and attributes defined for a banking application. As we can see, 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. Here, the *Transaction* class has been defined with one basic attribute: *TransactionAmount* and two object type attributes from the class *Account*: *DebitAccount* to specify the source account and *CreditAccount* to specify the destination account. In addition, the class *Account* has two atomic type attributes (i.e. *AvailableBalance* and *AccountNumber*) and two complex ones (i.e. *AccountHolder* and *LastTransactionsList*).

Finally, the *data connector model assistant* (DCMA) is used to specify the database access functions needed for the real-time system to provide the information to the user. These functions are specified as interface definitions

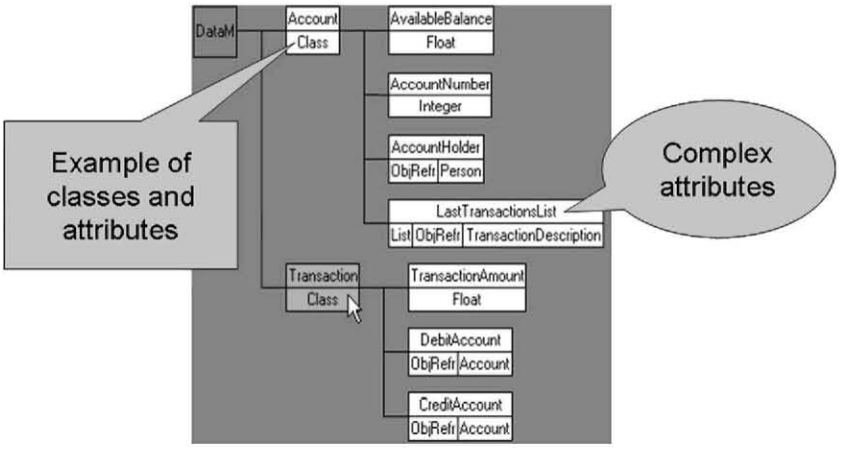

Fig. 2. Graphical details of data model classes and attributes.

including only their input and output parameters allowing their use by dialog designers, without needing to know much about database programming, and leaving the dialog flow to any changes in the system backend unaffected as long as the interface remains stable. For instance, a function that performs a money transfer between two accounts, the designer can indicate here as input arguments two integer variables for storing the account numbers and a float variable for the amount to transfer, and as output argument a Boolean variable to know if the operation was successful. Another example, in this case for the domain of a travel agency, could be a function to make a reservation; in this case, the input arguments could be two "String" variables, one for the departure city and the other for the arrival city, as well as two "Date" variables for storing the corresponding departure and returning dates. The returning variable for this function could be an "Integer" that stores the number of available flights retrieved by the search and an array with all flights information.

2.2. Retrieval layer

In the retrieval layer, the general flow of the application – in a language and modality independent way – is modeled, including all the actions that make it up (transitions and calls between dialogs, input/output information, procedures, etc.). It includes the state flow model assistant (SFMA) and the retrieval model assistant (RMA).

The *flow model assistant* is used to create the dialog flow at an abstract level, by specifying the states of the application, plus the slots to ask to the user and the transitions among states. It is also possible to specify which slots are optional (for over-answering) and which ones can be asked for by using mixed-initiatives (see Section 3.4.3). For instance, in the case of a banking application, the designer specifies the different tasks that can be accomplished in the service (e.g. welcome state, initial menu state to access available items in the service, a state for performing transactions between accounts, for providing information about account movements, and so on). Then, the designer specifies as slots the credit and debit account numbers and the amount to transfer in the transaction state.

The *retrieval model assistant* is used afterwards to include all the low-level detailed actions (e.g. conditions for making transitions between states, definition of variables and assignments, math or string operations, calls to dialogs to provide/obtain information to/from the user) to be done in each state defined in the previous assistant. For example, for the state where the user performs the transaction between accounts, the designer can define the following sequential actions (see example in Section 3.5):

(1) A call to a sub-dialog for requesting the account numbers and the amount to be transferred,
(2) An access to the database in order to perform the transaction,
(3) Then, report to the user if the transfer was successful or not,
(4) Finally, a jump to the next state in the application.

The assistant allows the designer to include complex actions such as making conditional transitions, performing mathematical or string operations, creation of variables, inclusion of programming loops (useful in case of requiring a user authentication procedure), as well as the possibility of using different kind of form templates (e.g. menu-based or sequential). Since this layer is modality and language independent all the input/output data provided by/to the user are managed using concepts.

2.3. Dialog layer

Finally, the dialog layer contains the assistants that complete the application flow specifying the details that are modality and language dependent for each dialog. The platform includes the following assistants:

The *User Modeling Assistant* (UMA) that allows the specification of different user levels and settings for each dialog in the application. Here, the designer specifies, for instance, the system behavior at runtime for confirming

the users' answers. This way, if the speech recognizer returns a low confidence in the recognition result then the system could request an explicit confirmation through a direct question or by asking a new one. On the other hand, the possibility of modifying the confidence levels according to the user profile allows the designer to change the behavior of the system to (1) permit advanced users to interact more naturally with the system by allowing additional confirmation strategies (e.g. implicit confirmations and not only the explicit confirmations available to the novice users), or (2) impose a stricter confirmation for critical data such as the amount in a banking transaction.

The *modality extension retrieval assistant* for Speech (MERA-Speech, see Section 3.6) adds special sub-dialogs that complete the dialogs already defined for the application considering the specific issues of using speech. Thus, the designer can create a complex dialog flow in order to deal with modality specific problems. Here we have dealt with the two basic problems that are specific to the speech modality: (1) the presentation of list of results retrieved from the database to the users in several steps depending on the number of retrieved items (i.e. zero, one, from two up to a maximum number, or more items than the maximum allowed, and (2) handling recognition errors by using different confirmation strategies (i.e. none, implicit, explicit, and repeat) in the dialogs that obtain information from the user. In the first case, the assistant allows the designer to define the different dialogs to show or request information to the user as well as the dialog flow for each of the four situations; in the second case, the assistant analyzes the dialog flow and automatically creates the sub-dialogs to provide the four kind of confirmation strategies and it also analyzes when each confirmation can be used or not (e.g. it is not possible to do an implicit confirmation if the next action in the flow is the access to the database since the system will not have the opportunity to confirm the information in the next turn).

In the *modality and language extension assistant* (MLEA) the language dependent aspects of the service are specified for each modality and language. For the speech modality, the extensions consist of links to the grammar and prompts for each language and dialog defined in the previous assistants for obtaining or presenting information to the user, while for the Web modality they are links to the input and output objects to interact with the user (e.g. textboxes, radio buttons, lists of results).

The *dialog model linker* (DML) is the responsible for generating one file for each selected modality where all the information from previous assistants is automatically linked together, i.e. dialogs, actions, input/output concepts, prompts and grammars, etc. by filling in different sections of GDialogXML dialog units. Then, the unified file for each language and modality is converted into the corresponding runtime script using the script generators of the next step.

The *script generators* convert the file generated by the dialog model linker into the execution scripts needed for each modality (VoiceXML and xHTML). Therefore, these modules solve the problems and limitations of each standard (Hamerich et al., 2003) and manage those issues regarding the handling of multilinguality (López-Cózar and Araki, 2005), database access, preparation of prompts or Web text, etc.

Finally, there are three other assistants that complement the platform. The first one is the *Vocabulary Builder* which prepares the vocabularies that will be used by the speech recognizer (i.e. the phonetic transcriptions of each word and phonetic alternatives for each language). The second one is the *Language Modeling Toolkit* that allows the designer to specify and debug the grammar files (in JSGF format or *n*-gram based) that will be used in the runtime system for recognition and for prompt generation using the *Natural Language Generation* (NLG) module (Georgila et al., 2004).

Finally, the third assistant, called *Diagen*, allows the manual creation from scratch or the fine tuning edition of all the different GDialogXML models and libraries generated by the assistants of the AGP. In contrast to most current editors available in other platforms, this assistant allows the possibility of creating any section of the GDialogXML specification with minimum effort (Hamerich, 2008). In this case, instead of forcing the designer to type in the XML tree (i.e. all the nodes and attributes), the assistant uses a set of pop-up windows that are sequentially displayed according to the information that the designer needs to specify. This way, in case the designer needs to create a state, the assistant shows a form window for obtaining the name of the state and the system strategy at that state (i.e. mixed initiative or system initiative), then several consecutive windows for defining the information (e.g. name, type) about each slot to ask in that state, then another pop-up window for defining the information about the transitions, and finally optional windows for defining help prompts, etc. Thanks to these features, the designer does not need to memorize the whole XML specification and thanks to the simple mechanism for defining the information the process is made easy.

2.4. Runtime system

Finally, another important component in order to run the VoiceXML script generated by the AGP is the interpreter or browser that executes the script and performs the connections with the other modules (recognizer, synthesizer, database access, telephonic interface, etc.). The selected interpreter for our platform was the open source library OpenVXI (Eberman et al., 2002) supported by Vocalocity Inc. The platform includes basic telephony functionalities, an XML parser to process VoiceXML and JavaScript files, processing user input, a complete implementation of the Form Interpretation Algorithm (FIA), debugging functionalities, simulated speech recognition, etc. Since the source files are available, there were no restrictions in adapting, mainly, the TTS and ASR

-interfaces to our proprietary modules and platform (see Cordoba et al., 2004; Hamerich et al., 2003 for detailed information).

As a mechanism for allowing the use of third party modules instead of ours (e.g. TTS or ASR), we worked on the integration of the runtime system into a distributed platform developed during the EDECAN[10] and SD-TEAM[11] projects. The platform is made up of seven modules that carry out the different processes in a dialog system. The current modules are Automatic Speech Recognition (ASR), Audio server, text-to-speech (TTS), Natural Language Understanding (NLU), Natural Language Generator (NLG), the dialog manager (DM), and the hub. The architecture defines different messages that the modules can use to share information between them. Since all information is passed between modules using XML messages via a central hub, it is possible to include new modules or new messages as required for new modalities or system capabilities.

3. Smart strategies to accelerate the design and improve human–computer interaction

In this section, all the strategies and mechanisms to accelerate the dialog design and improve the interaction between the platform and the designer are explained in detail. The main goal is to reduce the design time by simplifying the definition of the different dialogs, actions, and elements required to specify and run the service. Moreover, the proposed mechanisms help to guarantee that the generated models are well formed and optimized, as well as contributing to minimizing mistakes in the design.

The proposed accelerations can be classified into four classes: Heuristic-based, Rule-based, Context-based, and Wizards for simplifying the design process.

The first one corresponds to accelerations that use the database contents and data model structure. These accelerations are used to reduce the information displayed to the designer in the assistant for creating the database schema (Section 3.2), for proposing the SQL statements to access the database at real time (Section 3.3.2), for defining the database function prototypes (Section 3.3.1), and for automatically proposing states and dialogs templates that can be use to define the application flow (Sections 3.4.2 and 3.5).

Rule-based accelerations correspond to the application of the configurable domain knowledge rules that we have incorporated from our experience in designing dialog systems. Here, we use configurable rules that allow the assistant to propose which slots should be requested together, using mixed initiative dialogs or one by one using directed dialogs; the proposals are made depending on the difficulty of the data to request according to some configurable rules and the heuristic information from the database associated to each slot (Sections 3.1 and 3.4.3).

Context based accelerations correspond to strategies that use the information generated from previous assistants throughout the design. For instance, the relationships between the input/output arguments of the prototypes of the database functions with the attributes and classes in the database schema (Section 3.3.1) are used later onto automatically create state templates (Section 3.4.2) or dialog templates (Section 3.5). In addition, we use the high-level definition of the flow states and slots in order to propose the set of most probable actions required to complete the definition of each state (Section 3.5 point 3). In addition, the assistant uses the sequence of actions defined for each state in order to detect when it is possible to use implicit confirmations or not at real-time for the speech modality (Section 3.6).

Finally, the fourth one corresponds to accelerations mainly based on the incorporation of different wizard windows that automate/eliminate repetitive or common procedures in the design. For instance, we have included different form windows to define the dialog variables, for including conditional structures in the dialog flow (e.g. for, if-else, while), for creating mixed-initiative dialogs, for automatically proposing SQL statements (Section 3.3.2), or for defining the dialog flow used to show lists of retrieved results to the user when using the speech modality (Section 3.6).

Most of these accelerations are innovative and do not exist, to the best of our knowledge, in any commercial or research platform. When a similar acceleration is available, we have tried to go one-step further by incorporating new automation mechanisms. For instance, currently there are some development platforms that include assistants for defining and debugging SQL statements, but none of them propose the SQL statement to use; In addition, our platform is unique since it allows the creation of dialogs with over-answering, and over-answering plus mixed-initiative (Section 3.5), which are not included in the VoiceXML specification but that were accomplished by using standard elements at the expense of generating a more elaborated final script.

3.1. Heuristic information

Since many of the accelerations rely on using heuristic information from the database contents, we have implemented a new module that automatically extracts this information from the backend database. These heuristic features are obtained using an open SQL query that retrieves all the information from every table and field in the database. The system automatically collects information regarding the name and the number of the different tables and fields, and the number of records for every table. In addition, for each field the following numerical features are also collected:

(a) The average length in characters.
(b) The average number of words.

[10] http://www.edecan.es/.
[11] http://www.sd-team.es/all/Welcome.html.

(c) The vocabulary size (number of words that are different).
(d) The proportion of values that are different.
(e) The field type.
(f) The number of empty values.
(g) The number of different values.
(h) Whether the field is language dependent or not.

These features, grouped or individual, are used in different ways to improve the assistants and the design. For instance: (e) and (h) are used to accelerate the creation of the data model structure (Section 3.2) and to create and debug SQL statements (Section 3.3.2), (f) is used in the wizard window to define the data model classes (Section 3.2), in order to reduce and sort by relevance the fields that can be used to define the class attributes and when proposing dialogs to retrieve information from the user in the RMA (Section 3.5). Finally, (a)–(d) and (g) have been used to detect candidate slots that can be requested using mixed-initiative dialogs or one-by-one (Section 3.4.3); here the idea was to use these heuristic features in combination with predefined configurable rules in order to improve the performance of the speech recognition system by avoiding difficult data to be asked simultaneously (e.g. two long number or dates, or two string fields with a high vocabulary).

During the extraction of the heuristic features, we have incorporated a correction mechanism based on regular expressions in order to change the type returned by the metadata information in the SQL query for a given field. Thus, if the designer of the database defined a field using a generic type such as string or float when they actually corresponded, for instance, to dates or integers, then the system sets the right type. Besides, the analysis of each field is used to avoid or warn the designer about using mostly empty fields since they do not provide relevant information. One current limitation in our approach, as we mentioned in Section 1.3, is that we only collect numerical values for the heuristic information, instead of grouping them using associated key semantic terms (e.g. cheap, expensive, high, far, etc.). The possibility of including them in a future work would increase the robustness of the accelerations, as well as their understanding. The required modifications would be to implement some kind of automatic clustering in topics or ranges of the database contents and then introduce modifications in the different assistants in order to replace the semantic term for the corresponding threshold value.

3.2. Strategies applied to the data model assistant (DMA)

In this assistant the data model structure or scheme of the service is created through the definition of object oriented classes. As we have mentioned before, the objective of these classes is to provide information about the information in the database that are relevant for the service. Therefore, using as example the database schema depicted in Fig. 2, we can see that the designer defines two classes: *Transaction* and *Account*, and several attributes that are related between them and with the database (i.e. information about the relationship between each attribute and tables and fields). Considering the organization of the class *Transaction*, it is possible to infer that in order to perform a transaction three elements are required: the *TransactionAmount*, the *DebitAccount* and the *CreditAccount*. Since the last two are not atomic attributes but object references (ObjRefr to the class *Account*), we are required to go one level deeper into the class *Account* in order to find the corresponding atomic attribute that the system will request from the user (i.e., the attribute *AccountNumber*). Additionally, other dialog goals could be possible from analyzing these two classes, e.g. obtaining information about the last account movements (using the attribute *LastTransactionList*), to access the information about the account owner (through the class *Person*), information regarding the available balance, etc.

The main acceleration in this assistant is the incorporation of a wizard window that uses the heuristic features to propose full custom classes and attributes that the designer can use when creating the structure (see Fig. 3). The wizard uses the heuristic (e), the field type, for correctly setting the corresponding information in the window. The assistant also sorts the most important or relevant fields for each table in the database by relevance, using the heuristic (f), i.e. the number of empty values. Thus, if the heuristic is high (i.e. there are a large number of empty values), then the system considers that it is unlikely that it will be used to request information from the user and it will be placed at the bottom of the list. Moreover, the assistant accelerates the design proposing automatic names when a new class or attribute is being created. Finally, the assistant allows already defined classes to be used for creating new ones. There are also other interesting accelerations such as:

(a) Re-utilization of libraries with previously created models, which can be copied totally or partially. In this way, it would be possible to take advantage from previous models of the same application in order to add a new goal or service. Besides, the assistant allows the possibility of creating new libraries by selecting several classes and attributes in the current model.
(b) Automatic creation of a non-existing class when it is referenced as an attribute within another one. For instance, consider the case that the designer is starting the definitions of the complex attributes for the class Transaction in the schema shown in Fig. 2. In this case, when the complex attribute *DebitAccount* is included into the class, the assistant automatically searches the referenced object class, i.e. the class *Account*, in the internal list of already defined classes. If this class has not been defined previously, the assistant automatically creates it as an empty class that can be edited afterwards to include the attributes that belong to it (i.e. a top-down design). In the example,

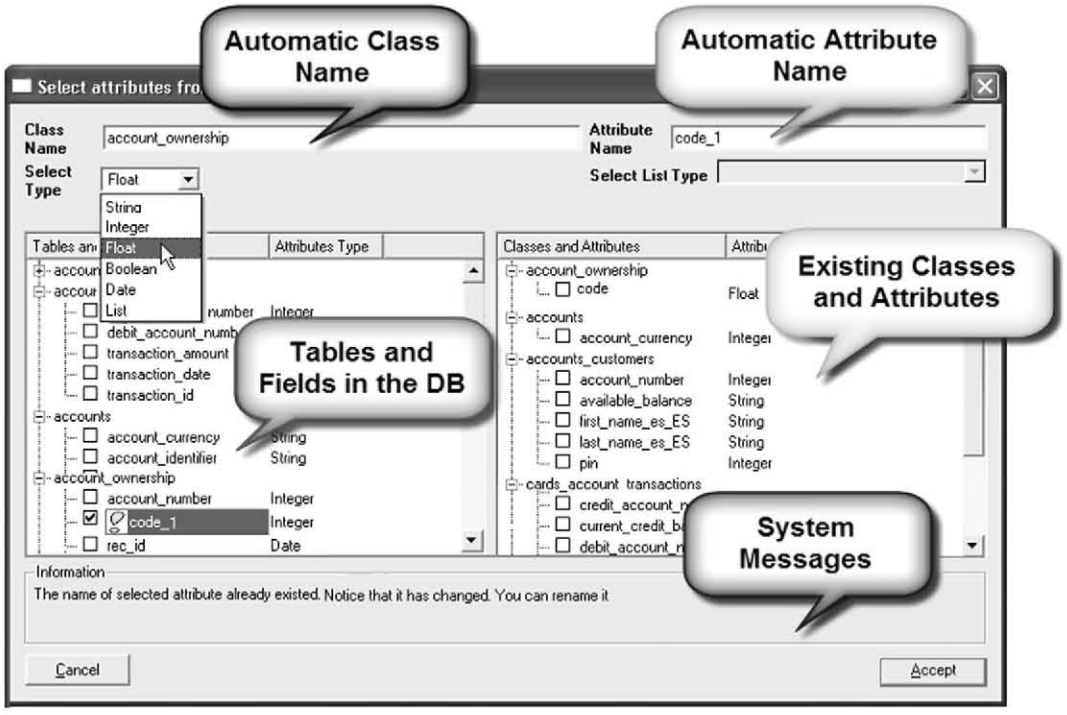

Fig. 3. Form fill-in window that allows the creation of custom classes (from the database and classes from the current model) in the DMA.

the same process can be done for the referenced class *TransactionDescription* when the *LastTransactionList* attribute is defined.

(c) Definition of classes inheriting the attributes of a base class (i.e. parent classes). In this case, when defining a new class, the designer can specify all the classes required to be used as base classes. Then, the assistant automatically displays all the attributes defined in the selected base classes and include the selected ones into the new class. This way, the platform uses concepts inherited from object-oriented programming.

3.3. Strategies applied to the data connector model assistant (DCMA)

The goal of this assistant is to allow the definition of the prototypes (i.e. the input and output parameters) of the database access functions that are called from the runtime system. Although the platform only requires the prototypes, we take advantage of this assistant in order to create the actual implementation of these functions and to include meta-information to accelerate the dialog design in subsequent assistants.

3.3.1. Definition of relationship between arguments and data model

The main acceleration strategy included in this assistant is the possibility of defining the relationship between the input/output arguments of the database access functions and the attributes and classes defined in the data model.

Fig. 4 shows an example of the GDialogXML code generated by the assistant for a database access function in the domain of the banking application. In this case, the function *PerformTransaction* has three input argument variables that collect the information regarding the account numbers and the quantity to transfer, and one returning variable defined as Float. In the code, the tag xArgumentVars (highlighted in yellow color) contains the information regarding the input parameters: the debit account number (*DebitAccountNumber,* letter A), the destination account (*CreditAccountNumber,* letter B), the amount to transfer (*TransactionAmount,* letter C), and the tag xReturnValueVars (highlighted in yellow color) contains the return argument *AvailableAmount* (in this case, the available amount after performing the transaction). In the figure, we can also see the information about the dependencies with the classes and attributes of the database schema defined in the previous assistant (i.e. with the tag XDataMAttr, highlighted in blue color) and the dependencies with the database tables and fields (i.e. with the tag xDBAttr, highlighted in green color). The usefulness of this acceleration is that these dependencies will be used in subsequent assistants (i.e., SFMA and RMA) to create state proposals (Section 3.4.2) and the automatic proposal of actions at each state (Section 3.5). As acceleration, during the definition of the arguments, the assistant automatically proposes the class and attribute which is more likely to be related to the given argument, as well as the database table and field. The mechanism is to use the name of the argument being edited to search for similar classes or attributes in the data model structure, whereas the table and field of the database is extracted from the data model since this information has been already defined in the previous assistant.

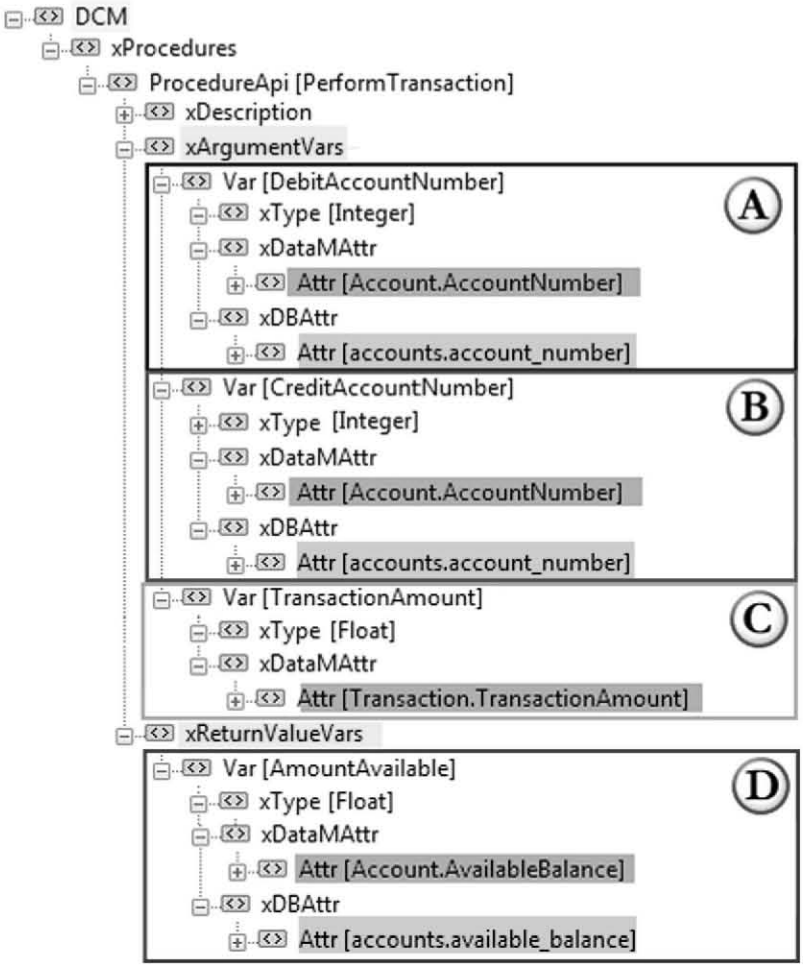

Fig. 4. Example of GDialogXML code for a Database access function.

3.3.2. Automatic generation of SQL queries

Fig. 5 shows the wizard window that generates the SQL query automatically for a given function. The assistant allows the inclusion of several constraints supported by the SQL language such as math functions (average, max, min, ln, exp, etc.), sorting, selection (Top or Distinct), clustering (Group By), Boolean operators (AND, OR) for combining the query restrictions, among others. In order to create the query automatically, the assistant uses the input arguments (defined in the function prototype, see number 2) as constraints for the WHERE clause, and the information of the output arguments as returned fields for the SELECT clause (number 1). The wizard also uses the heuristic (e), the field type, in order to create and debug the SQL statement correctly. New input or output arguments can be added if the function prototype is not complete or if the designer wants to test new argument combinations. The proposed SQL query is presented in a textbox (number 3) that the designer can edit. In addition, the assistant has a debug window (number 5) that allows a pre-viewing of the retrieved records when using the proposed query. In order to debug the query, the assistant first asks for specific values for the input arguments of the function (see number 4) proposing the value that appears the most in the database by default.

3.4. Strategies applied to the state flow model assistant (SFMA)

In this assistant the designer defines the state transition network that represents the dialog flow at an abstract level. The main accelerations are the automatic generation of state proposals, the possibility of specifying the slots through attributes offered automatically from the data model, the automatic unification of the slots to be requested to the user using mixed initiative dialogs, and the possibility of editing or generating new rules for controlling the unification. In addition, a new GUI allows the definition of new states using wizard driven steps and a drag-and-drop interface.

3.4.1. Functionalities included in the graphical user interface

One of the first conditions imposed to be successful in the interaction with the designer is a clear, intuitive, and flexible GUI. This is especially relevant in this assistant since it has to allow several editing and visualization

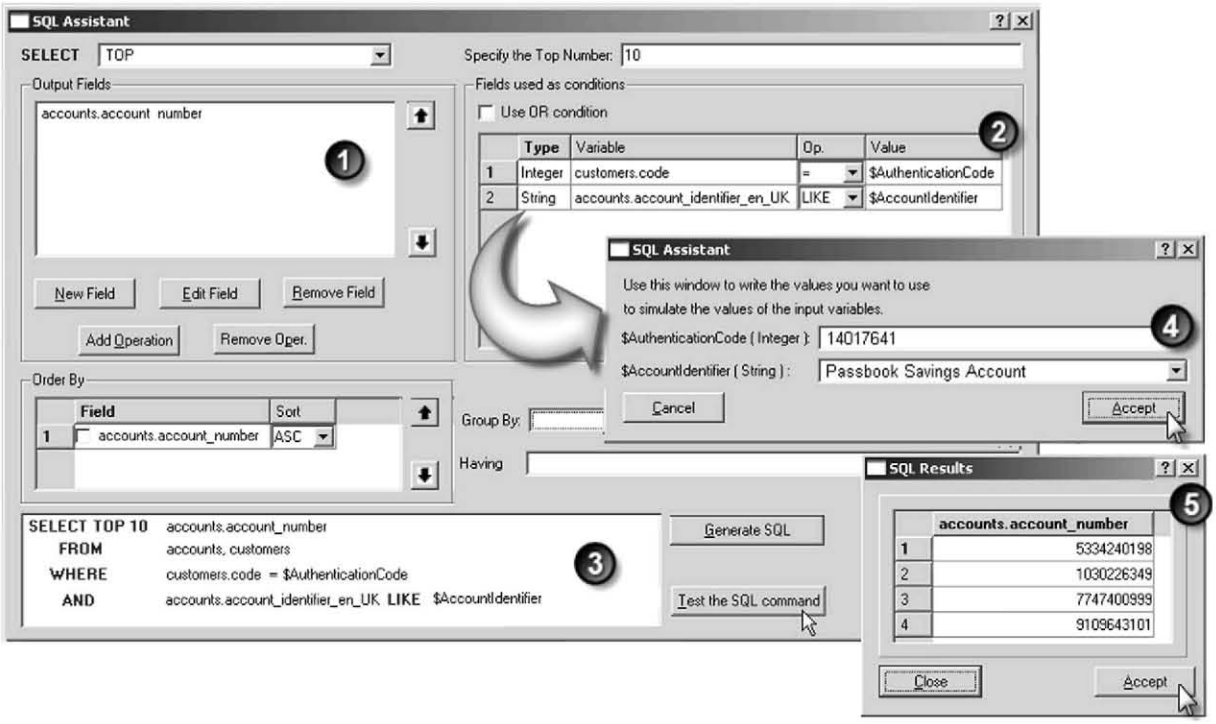

Fig. 5. Form fill-in window for the automatic creation and testing of SQL queries.

capabilities such as the possibility of creating the flow diagram easily. Basically, there are two visualization strategies: tree-based form-filling object modeling (e.g. like that used by VoiceObjects Desktop[12]) or state-based dialog modeling (e.g. like that used by the CSLU RAD toolkit or the Avaya Dialog Designer[13]). In our case, we have used the state-based dialog modeling or tree-structured description. In this kind of representation, each leaf and branch represents a state and a corresponding transition. Our main motivation for selecting this kind of visual representation was twofold: it is common in most commercial and research platforms (McTear, 1998), and it simplifies the visualization of the flow thanks to its different states and transitions. Although it is limited by the complexity of the task, since as the number of states grows the visualization degrades, several strategies have been proposed to solve this problem. In our case, we have followed two solutions: (a) allowing the designer to show detailed or minimum information on the states, as well as some degree of encapsulation using libraries and complex dialogs, and (b) implementing an automatic algorithm that helps the designer to place the objects on the canvas avoiding the creation of a confusing network of crossed lines between the states, and that reduces the visualization problems by using connector symbols so the designer is not forced to follow long lines beyond the area of visualization of the canvas (see Fig. 6). Finally, the main window also allows the creation of new states just dragging and dropping them from the floating window with the proposal of states, or using contextual right click commands. At the same time, it allows the creation of several connections (N:1, 1:M or N:M states) in few steps.

3.4.2. *Automatic state proposals for defining the dialog flow*

This is one of the most important accelerations in the assistant. Here, the system automatically generates an automatic proposal for the dialog states that include the slots to be requested to the user. The advantage of these proposals is that they can be used directly by the designer with little or no modification. In order to create these proposals, the assistant uses the information from the database structure and the prototypes of the access functions from the database. The proposed states are available as a sidebar for the workspace (see Fig. 7). The following sub-sections explain these state proposals.

3.4.2.1. Class dependent states. For each class defined in the DMA, the assistant creates a class template in which the designer can drag and drop into the workspace. The pop-up window, on the left-hand side of the figure, allows the designer to select the attributes to be used as slots in the new state. The assistant allows the selection of multiple templates/classes in order to create the new state. In this case, the pop-up window shows all atomic attributes that belong to the selected class. The assistant expands the complex attributes (with inheritance and objects) allowing only the selection of atomic attributes since only these attributes can be asked to the user in the real time system (numbers 3 and 4). A proposed name for the new state is automatically generated from the selected classes, but the designer can change it. Finally, the new state is inserted into the

[12] http://developers.voiceobjects.com/.
[13] http://www.avaya.com/usa/product/dialog-designer.

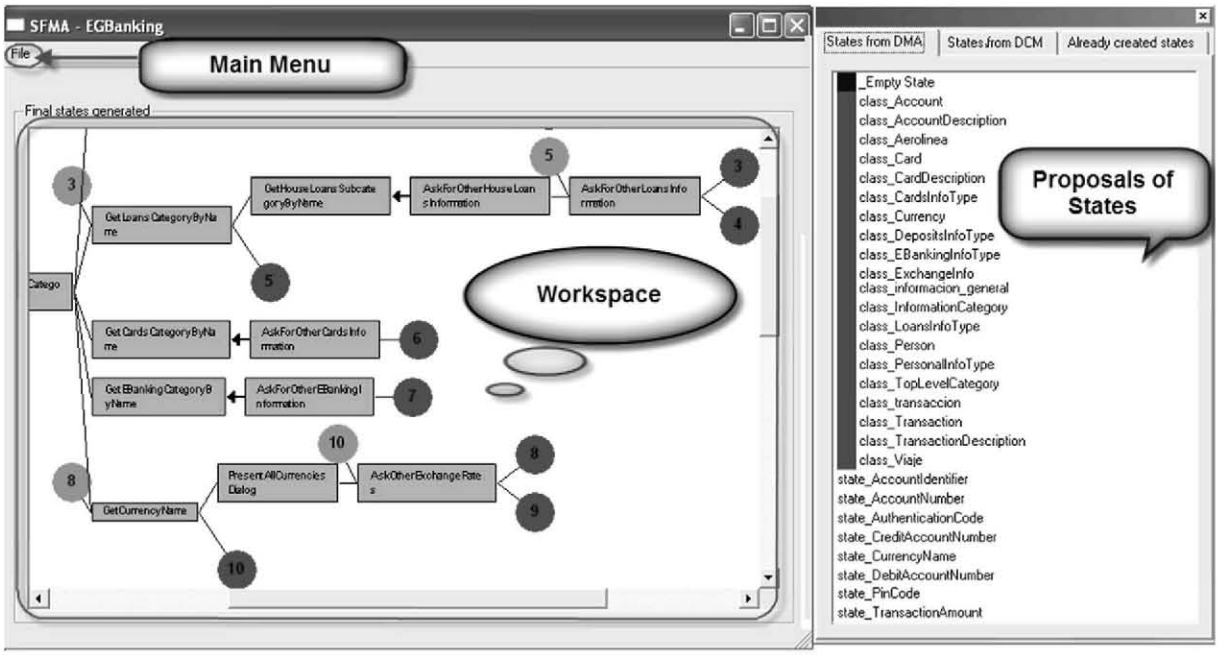

Fig. 6. Appearance of the SFMA main window: states, connectors, and proposals of states.

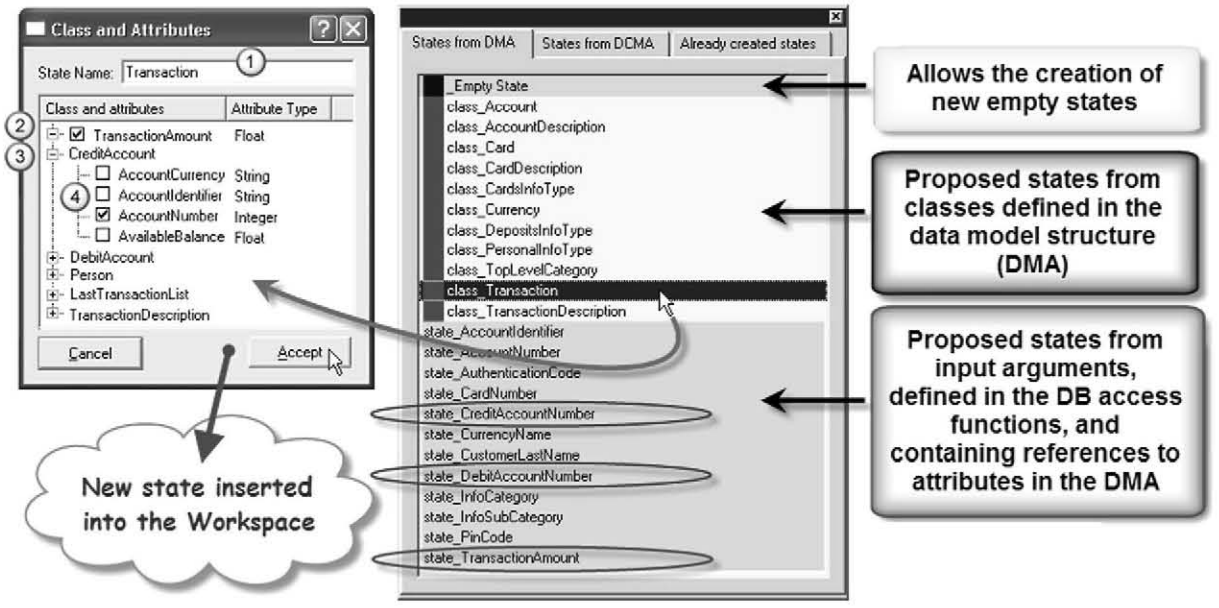

Fig. 7. Generation of new states using a pop-up window with state proposals and slots from classes defined in the data model structure (DMA).

workspace allowing the designer to define the transitions to other states. During this process, the mechanism for proposing the unification of slots for mixed-initiative or form filling dialogs is applied (see Section 3.4.3).

3.4.2.2. States from attributes with database dependency. This kind of state is created from any attribute defined in the database model (DMA) that refers to a database field and also used as an input argument in any database access function. For instance, considering the database schema of Fig. 2 and the input arguments of the database access function depicted in Fig. 4, the assistant would create three state proposals: one for the attribute *TransactionAmount*, one for the attribute *CreditAccountNumber*, and another one for the attribute *DebitAccountNumber* (see the marked states with ellipses in Fig. 7).The proposed states contain only one slot and its name corresponds to the name of the attribute in the data model. However, the designer can select several states before making the drag and drop allowing the creation of states with multiple slots.

3.4.2.3. States from database access functions. In this case, the system analyzes all the prototypes of the database functions containing input arguments defined as atomic types. Then, the system uses the name of the function as a proposal for the name of the state, and the input arguments as slots for that state. The main motivation for proposing these states is that they are likely to be asked to the user since, in general, the arguments of the database functions will be filled in with the information provided by the users in real-time. For instance, in the case of the database access function *PerformTransaction,* the assistant detects that it contains three input arguments (*CreditAccountNumber*, *DebitAccountNumber*, and *TransactionAmount*), therefore it creates three states, one for each input argument. and adds them to the list in the dock window. Moreover, the platform allows the designer to select several of these proposed states in order to create a unified state. The proposed states are available to the designer in the main window through the second tab in Fig. 7 (named "States from DCMA").

3.4.2.4. Empty state template and already created states. The first one allows the creation of a new empty state, with no defined slots inside, that the designer can completely define afterwards. Thus, we allow a top-down design. The second one allows the designer to re-use already defined states to create new ones (e.g. to create a new state based on our example *Transaction* state where the user has to provide the credit and debit account numbers but instead of returning the available amount, in this case the system will return the available credit or the new amount of monthly installments).

Example. In order to demonstrate the usefulness of the proposed states, consider the following case (all numbers refer to Fig. 7): the designer needs to create a state where the user will be able to perform a money transfer between two accounts (i.e., *Transaction* in number 1). Here, it will be necessary to define three slots: two for requesting the credit and debit accounts, and another one for the amount to be transferred. As we can see in Fig. 7, the assistant proposes this state through the template *Transaction* created from the corresponding class in the database schema (i.e. class *Transaction* in Fig. 2). From this proposed state, the designer could select the attributes *TransactionAmount* (number 2) and *AccountNumber* (number 4, from the attribute *CreditAccount*) (the debit account number is specified in the same way from the attribute *DebitAccount*, but it is not shown in the figure) to be used as slots in the new state *Transaction*. After closing the pop-up window, the system will analyze the three defined slots and will decide which ones should be asked together based on the heuristic information and unification rules described in the next section. In this case, the system will propose to ask them one by one (because three long numbers asked together would be very difficult to recognize). Finally, the system will create the new state and draw it into the workspace where the designer can edit the transitions. On the other hand, if instead of selecting the templates proposed in (A), the designer selects the three states marked with ellipses in Fig. 7 (i.e. proposals type (B)) or selects the template created from the SQL function *PerformTransaction* defined in the previous assistant (proposals type (C)), the system will create the state and analyze the slot unification as before, and the result would be the same.

3.4.3. Automatic unification of slots for mixed-initiative dialogs

This acceleration helps the designer to decide when two or more slots are good candidates to be requested at the same time (using mixed-initiative forms) or one by one (using direct dialogs) only when a mixed-initiative is not advisable. This is an interesting and innovative feature that we offer and distinguish our platform from others, where they leave this decision up to the designer. Since this functionality relies on using heuristic information it is only available when the slots in a given state have been related to a field/table in the backend database.

The assistant uses the average length, the vocabulary size, the proportion of different values, and the field type as main heuristic features obtained for the candidate fields (Section 3.1) and applies a set of customizable rules to decide which slots can be unified and which ones cannot. The rules included in the platform were defined from our knowledge on deploying dialog applications and from known guidelines in this area (Balentine and Morgan, 2001). In total, we provide a list of 30 different rules (16 for allowing mixed-initiative and 14 for using directed forms) that ranges from analyzing combinations of more than two slots with different field types (e.g. three strings, one string and one integer, two dates, two floats). Table 1 shows some examples of rules provided for allowing mixed-initiative (MI) or directed-form (i.e. one by one, DF). In the table, the terms long, short, high, etc. are defined according to the thresholds set by the designer for each heuristic.

For instance, according to the predefined rules included in the platform, the system does not propose using mixed-initiative dialogs if: (a) there are two slots defined as strings and the sum of the average length of both is longer than 30 characters. In this case, the system tries to avoid the recognition of very long sentences, (b) one of the slots is defined as a string with an average length greater than 10 characters, and the other slot is an integer/float number greater than 4 digits. The rule tries to avoid the recognition of long strings, e.g. an address plus long numeric quantities, e.g. phone or social security numbers, etc., in the same sentence, which again is very likely to fail, or (c) there are two numeric slots with a proportion of different fields for a given attribute which is close to one, and the vocabulary size of both fields is high (configurable value). Again, there is a high probability of misrecognition. Therefore, in all

Table 1
Example of default rules for unification or separation of slots provided by the platform (ranges and thresholds are application dependent).

Description and Justification	Float	Int	String	Date	MI	DF
Two or more "Date" slots: since dates include too many words we avoid to recognize them together	☐	☐	☐	☒	☐	☒
Two "Strings", with a high number of characters or words, and related to fields with a high vocabulary size: e.g. name of airports or cities and states	☐	☐	☒	☐	☐	☒
One single "String" and one long "Integer": Avoids the recognition of a long sentence generated by expanding the number into words	☐	☒	☒	☐	☐	☒
Two "Floats" with a high number of different values (ratio) and a high total number of values: Since both are floats, we have to consider the recognition of the decimal part	☒	☐	☐	☐	☐	☒
Three "String" slots each one with more than two words length and a medium vocabulary size: We avoid the recognition of long sentences	☐	☐	☒	☐	☐	☒
One short "Integer" and one "String" with low vocabulary size: e.g. channel and number in a TV recorder system	☐	☒	☒	☐	☒	☐
Two "Strings" with low vocabulary size: e.g. play the cassette	☐	☐	☒	☐	☒	☐
Two small "Integers" with low or medium ratio and low vocabulary size: e.g. asking for a year, day, and month	☐	☒	☐	☐	☒	☐
Two low vocabulary "Strings" and one short "Integer": E.g. two currencies and the amount in a currency conversion system	☐	☒	☒	☐	☒	☐
One short "Float" and one "String": allows asking for a command and quantity (e.g. set cursor position to three point five)	☒	☐	☒	☐	☒	☐

three cases, the system decides that it is better to ask one slot at a time using direct dialogs.

The configuration window, Fig. 8, allows the creation (number 2), edition, deletion, or activation of the rules and conditions (number 3). It is also possible to create rules for detecting direct dialogs (number 1).

3.5. Strategies applied to the retrieval model assistant (RMA)

This is the most complex assistant in the platform since this is where the designer describes each dialog in detail, i.e. all the actions (e.g. variables, loops, if-conditions, math or string operations, conditions for making transitions between states, calls to dialogs to provide/obtain information to/from the user) to be done in each state defined previously. The assistant is highly automated and intuitive, so it reduces the designer effort. In (D'Haro et al., 2006, 2004), we describe all the available acceleration strategies and capabilities in detail. Briefly, the most important ones are:

(1) Automatic creation of configurable and generic dialogs for obtaining or showing information from/to the user (with prefix DGet and DSay, respectively for easy identification). These dialog templates are created for each class and attribute defined in the data model. For instance, using the database schema in Fig. 2, the system will automatically propose a configurable DSay dialog for the class *Account* and another for class *Transaction*.Fig. 9 shows the form window to customize the proposed DSay dialog template allowing the selection of which information is

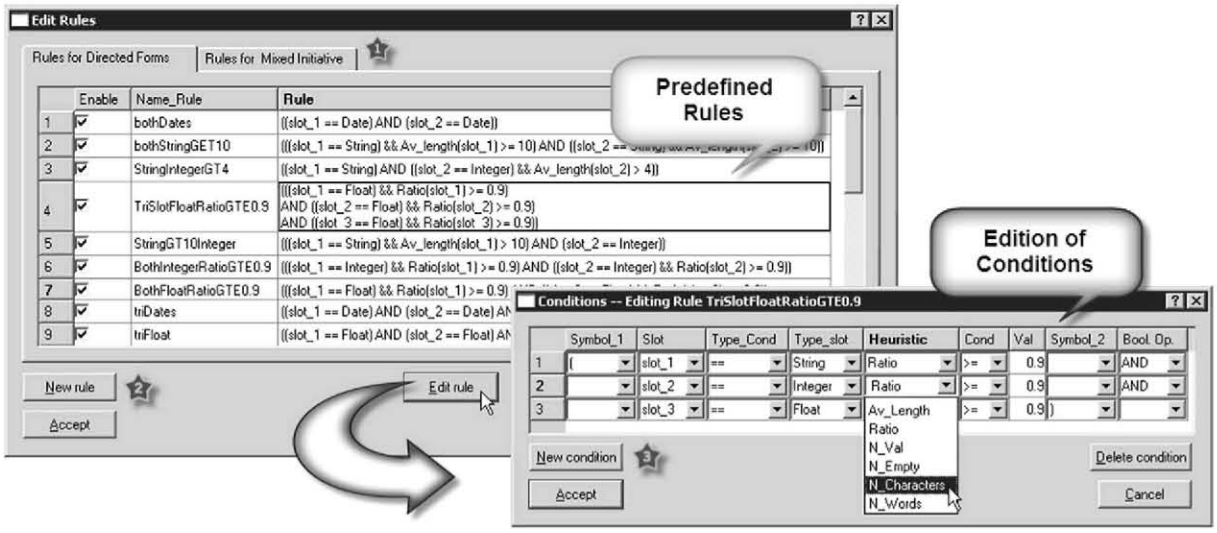

Fig. 8. Configuration window for creating or editing rules for automatic detection of direct or mixed-initiative dialogs.

going to be provided using it: the *AvailableBalance* and *TransactionAmount* in this example. Then, the resulting dialog can be set as the posterior turn in the dialog flow after performing the transaction, in order to inform the user about how much money was transferred and what is the available balance in the credit account. The figure also shows that the assistant allows the selection of other inherited attributes mentioned in the data model (in this case, from the class *Person* in Fig. 2).On the other hand, the assistant generates additional DSay and DGet dialogs for all the atomic attributes defined in the database schema (e.g. two dialogs: one to show and another one for obtaining the *AvailableBalance*, two more for the *AccountNumber* attribute, and two more for the *TransactionAmmount*). Finally, other common dialogs are also available such as Welcome, Goodbye, Transfer to operator, etc.

(2) Automate the process of passing information between actions/dialogs by proposing the variables that best match the connections or allowing the creation of new variables where no match exists. Since it is very common in dialog applications that several actions and states have to be 'connected' as they use the information from the preceding dialogs, we considered a highly valuable acceleration. In general, most current design platforms allow the same kind of functionality, offering the user a selectable list of all the available variables in the dialog. In other cases, especially considering the connections with database access functions, some platforms only allow the designer to define the matching by modifying the script code by hand. In our platform, we provide a better solution by automating the connection through automatic proposals.For example, suppose that the designer is defining a state to perform a transaction between two accounts and then to inform the user about the available amount. In case that the designer had previously defined a database function to perform this action, and that the function prototype requires three input arguments (i.e. credit account number, debit account number, and amount) and returns a float value (i.e. the available amount), the designer here needs to connect the current state variables containing the two accounts (e.g. *debitAccountNumber* and *creditAccountNumber*) and the transfer amount provided by the user (e.g. *TransactionAmount*), as well as the variable to save the final available amount, with the input and output arguments of the database function. In this case, the assistant detects the input/output variables required and offers the designer the most suitable already defined variable of a compatible type; if there is more than one candidate variable to be shown, the assistant sorts them according to the name similarity between function argument and current variables. If there is no compatible variable to offer, the assistant allows the creation of a new local/global variable. Since the sys-

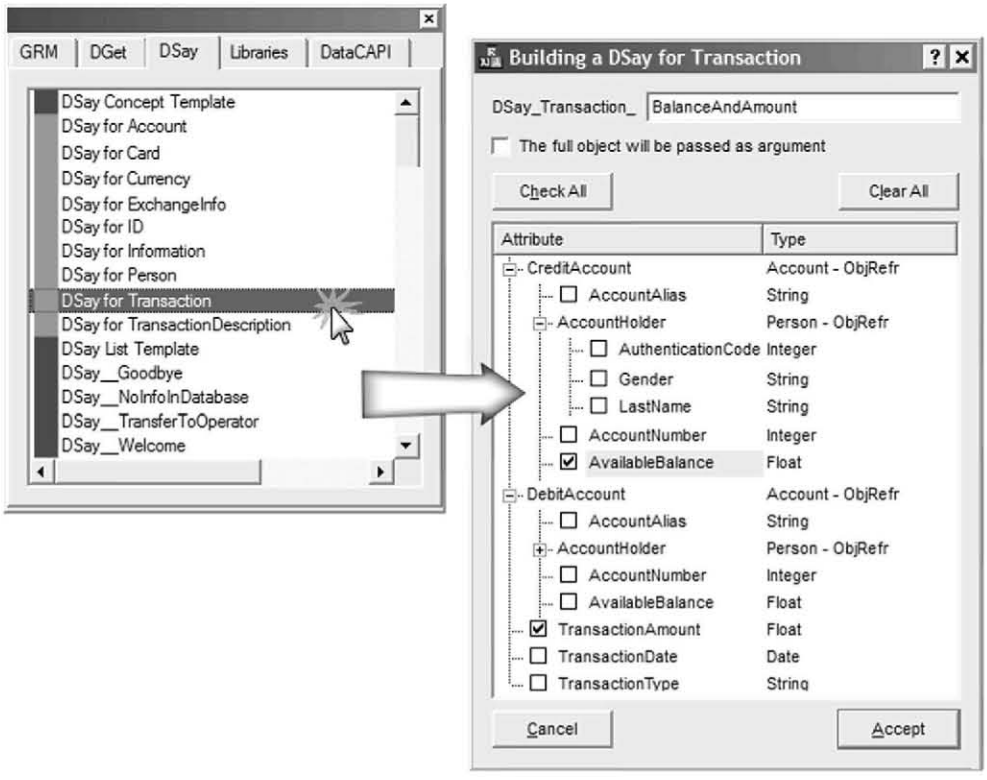

Fig. 9. Example of an edited DSay dialog that provides the user the available balance and the transaction transferred amount.

tem automatically proposes the values and options presented in the forms, the designer only needs to click the accept button and continue with the design. Additionally, the assistant includes a window where all the matching can be edited.

(3) Automatically propose the actions required for completing the information for each state of the dialog flow; the assistant proposes the dialogs to ask information from the user, the database access functions, and the dialogs to show information to the user. Fig. 10 shows an example of the proposals for a dialog where given a currency name the system provides its specific information (e.g. buy and sell price, general information, etc.) in the context of a banking application. Using the proposal window, all the designer would need to do is to select the corresponding dialog to ask the currency name (i.e. *DGet_CurrencyName_IN_CLASS_Currency*), then the database access function for retrieving the information (i.e. *GetCurrencyByName*), and finally the dialog to show the information to the user about the currency (i.e. *DSay_ATTR_BuyPrice_IN_CLASS_Currency*). To provide these proposals the assistant uses the information of the relationships between slots and arguments of the database functions and the attributes and classes in the data model. When there is no relationship specified, we apply relaxed filters such as matching in types, similarity of names, or same number of arguments and slots in the state.

(4) The platform provides five basic dialog types that cover the usual possibilities in programming: based on a loop, based on a sequence of actions (e.g. calls to sub-dialogs), a switch construct based on information input by the user (i.e. menu-based dialog), a switch construct based on the value of a variable, or empty dialogs, with no action within, that can be edited afterwards.

(5) The platform allows the quick creation of mixed-initiative dialogs, dialogs with over-answering (that do not exist in any current development platform), the quick view of dialog actions using tooltips, among others.

(6) Finally, the platform allows the quick creation/deletion of dialog variables and constants, the creation of if-then-else or loop (for, while) structures that allow the designer to test one or more conditions before doing other actions or proceeding with the dialog (e.g. to ask the user a pin code and then try to obtain it until a defined number of tries is reached, in case the pin is incorrect the system can provide an error message and finish the service), selection structures (switch-case), assignments between simple and complex (objects) variables, and assistants for carrying out mathematical or string operations. In all these cases, the assistant uses form-fill windows to allow the designer to define them and then to include the corresponding embedded code to perform them at real time.

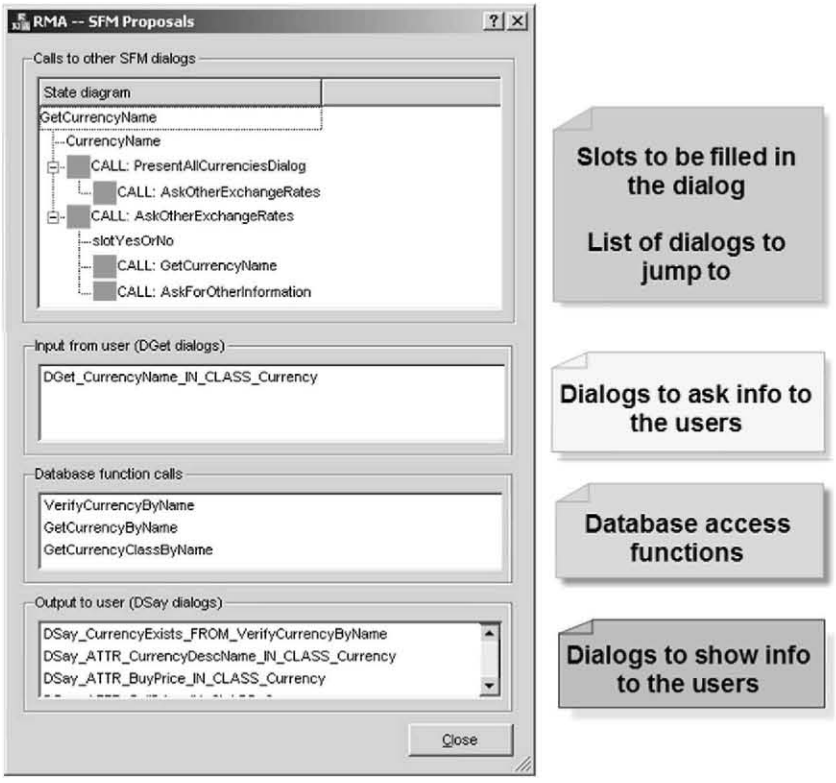

Fig. 10. Example with automatic dialogs and database access function proposals.

3.6. Strategies applied to the modality extension retrieval assistant for speech

In this assistant we considered solutions for two important and specific problems for the speech modality: (a) the presentation of results to the user after accessing the database, and (b) the confirmation of the user's answers. The common mechanism offered by current platforms to deal with these problems is to force the designer to specify the complete dialog flow or to leave the problem to some predefined actions provided by the ASR engine. These solutions are not satisfactory since they imply the codification of too many situations and conditions by hand, and because there will be restrictions on the confirmation handling that the designer could not take into account. Our solution relies on providing automatic proposals for the different data that the designer has to specify, by automatically generating all the dialog flow according to the designer selections, and by using predefined configurable templates and built-in dialogs (please refer to Section 4.6 and Appendix C in (D'Haro, 2009) for further information).

For the dialogs that provide the list of retrieved results after a database query, the assistant allows to specify the dialog flow for showing the information depending on the size of the list. Four cases were considered: (a) when there is no retrieved result, (b) when the list has only one item, (c) when the number of items lies within a defined range, or (d) when there are too many items, so it is difficult to say all of them using speech.

On the other hand, for the dialogs to obtain information from the users, the assistant automatically generates the flow for confirmation handling (i.e. what to do when the user does not provide an answer after a system query, to ask direct questions, etc.). We consider the following cases: (a) confirmation for dialogs with one slot, (b) dialogs with mixed-initiative, (c) dialogs with one compulsory slot plus slots with over-answering, and (d) the most complex case, dialogs with mixed-initiative and over-answering slots.

3.7. Strategies applied to the script generator

As we mentioned in the description of the platform structure, the platform automatically generates a standard compliant VoiceXML script required to run the service. However, several tasks were carried out, both in the platform and the runtime system, in order to support and overcome some of the limitations of VoiceXML and to increase the portability and functionality of the platform. Below, we briefly describe our efforts in this area. For further information please refer to D'Haro et al. (2006), D'Haro (2009) and Hamerich et al. (2003).

It is well known that the current VoiceXML standard specification limits the naturalness of the interaction of the user with the system. One of the main problems happens when the speaker wants to go back in the flow. In this case, the VoiceXML allows the designer to introduce a dialog to ask if the user wants to try again or repeat the same action. In our platform, we have applied a more general solution to this problem by using a "switch-case" dialog that the designer can use to reset the corresponding slots in the state and jump back to a previous state to allow the user to repeat the process. Since we use global variables to keep the information of each slot, it is easy to reset them according to the user selection at any state.

Finally, another problem occurs if the user wants to change an earlier piece of information before querying the database. In this case, the VoiceXML standard does not define an easy mechanism to implement this kind of behavior; therefore, it is responsibility of the designer to design it. In this case, the platform allows the designer to select the following options: (a) to use a confirmation sub-dialog just before retrieving the results from the database, or (b) to use a special token word such as: "abort" in order to allow the user to restart the state or "agent" in order to redirect the call to a human agent. As future work, we plan to include an automatic dialog template for confirming the dialog slots that the designer can easily use.

4. Evaluation

In order to estimate the performance of the platform, its assistants, and the different acceleration techniques, two evaluations were carried out: (a) an objective evaluation, where different designers, using our platform, carried out predefined typical tasks when designing dialog applications, and then compared the same tasks but carried out with an alternative assistant with fewer accelerations, and (b) a subjective evaluation where the designers rated the assistants and accelerations after using the platform.

In order to understand the scope and goals of the current evaluation, it is important to mention that right at the end of the GEMINI project, we carried out a subjective and an objective evaluation with more than 40 developers, where we tested the level of functionality of each assistant and their integration in the platform. During this evaluation, a complete dialog application was carried out, allowing us to know the amount of time the evaluators spent on using and learning the application, as well as different recommended improvements in terms of accelerations and GUI (for further details please refer to D'Haro et al. (2006)). Besides, as part of the project, the development platform was used for successfully creating two complex applications: (1) a banking application for a commercial product by a Greek bank (one of our partners), and (2) an application called CitizenCare to offer a voice information retrieval system in the context of public authorities available in both German and English languages, as part of a government supported application. It is important to highlight that both applications were evaluated with actual callers, showing that the resulting dialog application and the design platform worked properly. In the next section, we will provide a short description of the evaluation results

for both applications. For additional details please refer to GEMINI (2011), in the section "Public test evaluation report".

4.1. Evaluation results for the runtime platform

For the banking application, a total number of 143,653 calls (with more than 2000 different customers) were answered by the VoiceBanking system. The calls were recorded for a total of 6 months with an average of 22 thousand calls per month. The distribution of the population using the service was: Male: 70.5% and Female: 29.5%, without any limitation on age or profession. The users' language was Greek with the following dialect variations: Northern, Aegean, and Cretan. One of the most important results from the evaluation was that the percentage of customers that actually chose to be served by the automatic system was almost 45%, although they knew, from the first prompt, that they could reach the human operator at any time. In addition, from the total number of calls, more than 40% of them were served totally by the system without any operator intervention. On the other hand, dialog performance in terms of transaction success was 92.23%. The task completion rate was 93.51%, and the average duration of the interaction was 107.4 s considering the nine main tasks available in the application (the result also includes the time spent on performing the user authentication and the prompts used to provide the information to the users). The hang-up rate was 22.08% (where 20.08% of them occurred before the 1st answer), the average number of turns was 6.35, and the operator fallback was 2.81%. Finally, a subjective survey about the system was done among a users control group, i.e. bank employees and call centre agents. The results show that 74% of the young users (20–40 years old) were willing to use the automatic system in comparison with the 60% of older users. Moreover, 76% of the young users changed the way of speaking in order to increase the quality of system–user interaction and only 60% of older people accepted such a change.

Regarding the CitizenCare application, the evaluation was carried out on seven male and three female German subjects with ages ranging from 27 to 44. When asked about their user experience with automatic systems, three considered themselves as 'novices' while the other seven considered themselves as 'intermediate' users. The results showed that most of the subjects (80%) rated the system easy to use, and 30% stressed the system's capability to react on shortcuts. 10% rated the system 'partly easy to use' since it sometimes presented too much information at once (when selecting all information). Finally, 10% did not find easy to get the desired information, mainly due to the poor recognition rate of the ASR used. Seventy percent of the subjects had no complaints about the dialog flow. The other 30% criticized mainly the recognition failures of single words and the overall poor recognition quality.

Finally, we want to highlight that many aspects of the runtime behavior of the application were not considered in this evaluation for the following reasons: (1) because the final result for the voice modality is a VoiceXML compliant script that can be run at any voice browser, therefore the quality of the final script was assumed to be right, except for minor bugs or mistakes made by the designers, and (2) because the final dialog application is constrained by the self-limitations of the VoiceXML standard, although some of them, such as incorporating over-answering dialogs by using a more elaborated flow logic with standard elements, using global variables for allowing transitions between different states and keeping the dialog information available to all the states, using a special switch dialog in order to be able to go back in the dialog flow, or the ones described in Section 3.7, were tested during the creation of the GEMINI applications (for further details about the improvements made to the VoiceXML standard please refer to D'Haro et al. (2004).

4.2. Experimental setup

The evaluation was made in two sessions of 4 h each by 9 testers which were classified into three levels: 4 novices, 3 intermediates, and 2 experts. All the evaluators had some experience in at least one programming language but little experience in designing dialog applications. Most of the evaluators were undergraduate students at our university. The average age for all testers was 27. From this group, only three participants had some knowledge of the platform.

During the first session, the evaluators received a complete explanation of the whole platform, the goals of the evaluation, and the interface used to obtain the statistics. Finally, they also received instructions and evaluated the three first assistants: DMA, DCMA, and SFMA. During the second session, the evaluators learnt how to use and evaluate the RMA and MERA-Speech assistants. In general, each assistant evaluation was divided into three main blocks: (a) the evaluators received instructions on the capabilities and accelerations included in the corresponding assistant through examples of use, (b) the evaluators were asked to carry out an example task in order to consolidate the knowledge and to answer questions. (c) Finally, the evaluation was carried out and the evaluators were later requested to fill in the subjective survey to measure the acceptance, usability, intuitiveness, and most interesting features of each assistant.

4.3. Objective evaluation

The goal was to evaluate the proposed accelerations in our platform against using a similar tool with different or less accelerations. In order to do so, we collected a set of quantitative measures obtained by the testers when they were requested to carry out different tasks using the platform and a parallel tool. Although there are currently no

standard metrics for making the comparison, in (Jung et al., 2008), for a similar evaluation, they proposed different tasks that the evaluators had to carry out using their platform and an open text editor chosen by each participant. Here, different metrics were collected such as mouse clicks, keystrokes, and elapsed time. Agah and Tanie (2000) carried out a similar evaluation, proposing the same metrics when evaluating their intelligent interface. Given both cases, we decided to use these metrics too but proposing a new one: the number of times the user presses the delete key when typing. The goal of this new metric was to provide an additional measure of the difficulty of introducing information into the assistants or writing the GDialogXML code. Besides, since the assistants reduce the number of keystrokes needed, this fact could also be reflected in the number of mistakes made by the designers.

For our evaluation, we followed a similar approach than (Jung et al., 2008), i.e. proposing different tasks for each assistant and comparing the quantitative measures in each case with those obtained when annotating the same tasks using the semi-automatic editor included in the platform called Diagen. Like the other tools in the platform, Diagen also includes interesting accelerations to facilitate the process of writing or editing the GDialogXML models. The most important features are: (a) the XML is automatically created and pasted onto the workspace by using of a set of pop-up windows that are sequentially displayed according to the information that the designer needs to specify, thus it is not necessary to type in all the tags nodes and children, (b) incorporation of a large number of templates for defining the whole set of possible actions and information allowed by the XML syntax for each kind of model and assistant, and (c) the visualization and validation of the data. For further details see Hamerich (2008) and D'Haro (2009).

The reasons for using Diagen, instead of allowing the evaluators to use any text editor of their liking, were: (a) to make the fairest comparison between both evaluations. It is well known that writing any information in any XML-based language is a tedious and difficult task; (b) Diagen reduces the need to memorize the XML specification, (c) almost all developers and development platforms use some kind of tool for writing from scratch or fine-tuning the code generated by the main application, and Diagen is a representative example of this kind of application, and (d) because we could not find any commercial or academic platform comparable to ours. For instance, most of the platforms create only VoiceXML applications instead of multimodal services as in our case (Speech using VoiceXML and Web using xHTML pages), or they do not take into account the Database information nor include the accelerations that we needed to evaluate. Finally, most of the commercial platforms have an advanced graphical interface which we did not want to evaluate as it is well known that the appearance of the GUI has a great influence on the evaluators.

Finally, it is also important to mention that the database used during the evaluation was a modified version of the database used for developing the Greek bank application at the end of the GEMINI project. The reason for not using the original one was because of the sensible data about the customers contained on it. In this case, the critical information such as names, account numbers, pin codes, etc. were completely modify by similar ones; however, the database schema was preserved without any modification. In addition, the selection of the same database for all the participants was considered as necessary in order to compare the different metrics obtained for each evaluator.

4.3.1. Description of the evaluated tasks

In general, for each of the evaluated assistants we defined a set of two or three different tasks that were carefully chosen to test the different possibilities and accelerations allowed by the assistants, as well as the different kinds of problem that a designer could find when developing a real application. Below, we provide a brief description of each of the evaluated tasks as well as information about the time the evaluators spent on completing them. For a complete description, please refer to D'Haro (2009).

To evaluate the creation of the data model structure (DMA, Section 3.2), we asked the evaluators to test two different sub-tasks:

(a) In the assistant for creating complex classes: The definition of the class *Account* with two atomic attributes (i.e. account number and available balance, both related to the corresponding database fields).
(b) In the automatic creation of non-existing classes (see Section 3.2): The creation of a mixed class structure (in this case, the class *Person*) including two atomic attributes (i.e. first name and last name, both related to the corresponding database fields and with language dependency) and one complex attribute (i.e. a list of accounts defined as an embedded class).

For the first task, the average elapsed time was 45 s. For the second task, it was 65 s.

To evaluate the creation of the database access functions (DCMA, Section 3.3), we proposed the creation of a function with two input arguments and one output argument, as well as to check the results retrieved for the proposed SQL statement (Section 3.3.2). In this case, the function proposed for testing had to return the account number given the authentication code and account alias. The average time needed in the evaluation was 125 s.

For the definition of the states, slots and transitions at a high-level (using the SFMA), we proposed three sub-tasks:

(a) The creation of a state with one slot related to the database (using the proposal of automatic states with slots or the empty state template and then define the slot, Section 3.4.2). The objective of the proposed

state was to ask the user for the target service and to define the transitions to the next dialog. The average time for this task was 33 s.

(b) The definition of a state with two slots, where both slots had to be set as a mixed initiative, and the transition to other state (using the automatic unification of slots to be requested using mixed-initiative dialogs and the automatic creation of an undefined state when it is referred as a transition state, Section 3.4.3). The proposed task was to create a state for requesting the pin code and alias of the account and then to make the transition to a new state where the user would be asked to select the available tasks after performing the authentication step (e.g. transactions, obtain account information, and buy or sell shares). The time spent on this evaluation was 58 s in average.

(c) The creation of a connection between two states (in this case, this task was included for evaluating some of the functionalities included in the graphical user interface). The average time was 10 s.

For the complete definition of the actions to be carried out in each state (RMA), we proposed three tasks:

(a) The creation of a menu-based dialog where users are required to select between three options (i.e. personal information, general information, and transactions), and according to the user selection to jump to a different state. In this case, the dialog flow was designed in less than 90 s thanks to the different kinds of dialogs provided by the platform (Section 3.5), the action proposals window, and the automatic DGet dialog templates.

(b) The creation of a dialog with over-answering and an IF-Then-Else condition. The proposed task was to use a DGet dialog to obtain the alias of the account to make a transfer and optionally to provide the transfer amount. Then, depending on the selected account (i.e. if it was the favorite one or not) to jump to the dialog to ask the transfer amount or to another dialog to request additional information about the account to be used. Here, the designers spent less than 2½ min thanks to the dialog proposals window, the automatic matching of arguments between actions, the procedure for including compulsory and optional slots, and the possibility of defining programming structures.

(c) Finally, the creation of a mixed-initiative dialog to perform a transfer between two accounts (requesting the aliases of the debit and credit accounts), then calling the dialog that asks for the amount, then calling the function that accesses the database and, finally, confirming the user if the transfer was successful or not. This task allowed testing the accelerations provided by the assistant for defining mixed-initiative dialogs, matching variables, the action proposals window, and the assistant for defining local/global variables. The average time spent on this task was close to 90 s.

Finally, for the MERA-Speech assistant, we proposed two tasks. In this case, thanks to the available accelerations, the assistant automatically proposes the strategy to be followed and automatically creates all the internal actions for handling the speech recognizer errors.

(a) The definition of a dialog for presenting a list of retrieved results. In this case, for providing information on the rates for buying or selling different international currencies. The elapsed time was in this case nearly 1½ min.

(b) Finally, to automatically fill-in the confirmation handling for all the dialogs to ask for information from the user included in the design. Here, the time spent was only 4 s, since all the evaluators used the automatic proposal of the application, although, as it was expected, the expert developers spent a little more time (around 7 s) on reviewing the proposals.

4.3.2. Evaluation results and observations

During the evaluation we observed some factors that must be considered in order to understand the results. The first one was that in some cases the time that experts and novices/intermediates spent on solving the same task was very different since the former used the available strategies and accelerations but the latter used an alternative method, not using the accelerations but a manual method. In order to avoid this behavior, we reinforced the explanation of the accelerations and spent some more time solving questions; (b) considering the increasing complexity of the XML language for coding the more complex tasks, we should expect greater improvements in the elapsed time when using the assistants instead of Diagen. However, as the testers used Diagen continuously during all of the tasks they soon got used to its interface and therefore worked faster with it; (c) finally, we also saw that the evaluators, when using the assistants, spent a lot of time reviewing the final result to check whether it corresponded to the expected result, however when using Diagen, since a lot of XML text was generated, they did not spend so much time on the revision.

In general, all tasks using the assistants or Diagen were carried out in just a few seconds to two minutes (Diagen being, on average, two or three times slower). The exception were the tasks for the RMA, where the average time elapsed using Diagen was 1493 s (around 25 min), in comparison to the 140 s (2½ min) using the platform. In this case, the time elapsed is one order of magnitude greater than that using the assistant. The main reasons for these values are the extensive complexity of the GDialogXML syntax when codifying the optional and compulsory slots, and the low number of accelerations included in Diagen to codify the conditional actions.

Fig. 11 shows an overview of the average improvements, in percentage, of using the assistants instead of Diagen, for each quantitative measure and the average improvement considering all the metrics and evaluators. In the figure, a positive value means that the assistants perform better than Diagen, and a negative value means the opposite. As we can see, the accelerations proposed in this paper produce an average improvement of 65.5% for defining the data model structure, 16.6% for defining the prototypes of the database access functions, 42.2% in the definition of the finite state model of the application (SFMA), and 84.8% for defining all the actions of each state of the dialog flow. Thus, we obtained an overall average improvement of 52.3% which corresponds to 56.5% improvement in the time elapsed, 13.4% for the number of clicks, 84% in the number of keystrokes, and 55.2% in the number of keystroke errors. These results are consistent with the number and scope of the accelerations provided by each assistant. Besides, the improvements are greater in the assistants where the more complex structures and actions are required; thus, we accelerate the design and guide the designer in the steps where it is more necessary.

4.4. Subjective survey

At the end of the two sessions of the objective evaluation, the evaluators were requested to fill in a subjective survey regarding the different assistants and accelerations. They were asked to answer a 4-item questionnaire per assistant with general questions about the appearance of the assistant, its level of intuitiveness, how fast it took to learn it, and whether the functionality of the assistant was enough. Then, they also answered to a 12-item questionnaire with specific questions about the accelerations included in the AGP. In most questions the users had to rate the relevant attribute or characteristic using a 10-point scale (1 = minimum, 10 = maximum). Finally, the survey also included open questions to provide comments and suggestions.

The left-hand side of the chart in Fig. 12 shows the results of the general questions on the different assistants evaluated. In this case, we observed that these results confirm the designer-friendliness of the assistants, as well as their usability, since all the assistants obtained an overall score of more than 8.0, which is a satisfactory result. It is

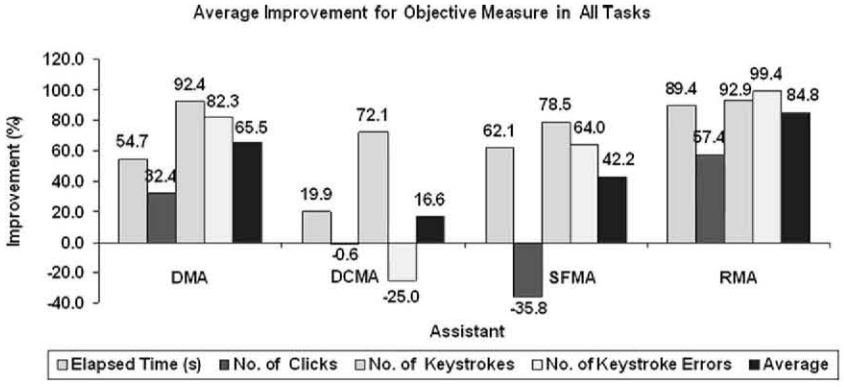

Fig. 11. Chart with the average improvement by assistant considering all tasks for the objective evaluation.

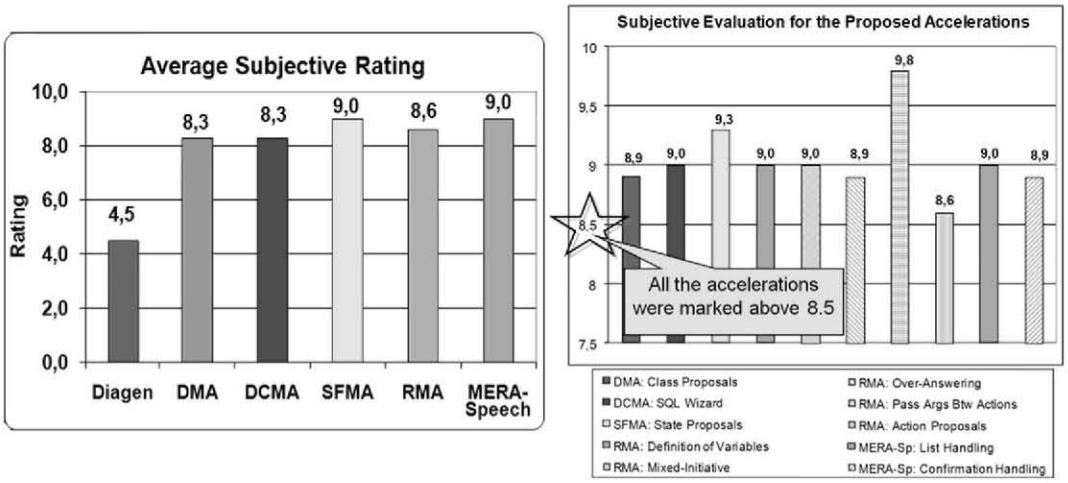

Fig. 12. Average results of the subjective evaluation for general questions on the assistants (left) and for the accelerations (right).

important to mention that although Diagen was easy to use for the first tasks, it got a bad qualification of 4.5, probably because the generation of the final tasks was too cumbersome in comparison to using the platform assistants.

The right-hand side of the chart in Fig. 12 corresponds to the results for the accelerations used during the objective evaluation. Thus, the participants had the possibility of using and experimenting with them, therefore their results are relevant since they are given in the heat of the moment. In this case, evaluators scored the automatic states in the SFMA with 9.3, the SQL generation and the unification of slots for mixed initiative with 9.0, and the class proposals for the DMA with 8.9. As regards the RMA, the passing of information between actions/dialogs and the proposal of actions to define the states obtained 9.8 and 8.6 respectively.

5. Conclusions and future work

In this paper, we have described the main accelerations included in a multimodal and multilingual design platform in order to speed up the design and guide the designer through all the steps required to create dialog services. The proposed accelerations are, in most cases, innovative without a direct correspondence to those offered by any of the current commercial and research platforms. Different types of accelerations have been proposed according to the requirements, capabilities, and available information at each assistant that makes up the platform. Most of these accelerations take advantage of heuristic information extracted from the contents of the backend database and from an object-oriented representation of the data model structure, in order to generate different kinds of proposals that simplify the process of creating and completing the dialog flow. Other accelerations consist of different wizard windows or simplified processes that help designers to complete, create, or debug models required by the design and runtime platform in order to provide the service.

In order to study the usability and acceptability of the assistants, as well as the proposed accelerations we carried out both subjective and objective evaluations with designers with different levels of experience in programming dialog applications. The results showed that the proposed accelerations improve the interaction with the platform, help to generate better services, reduce the design time by more than 56%, and were highly appreciated (between 8.0 and 9.0) by the designers as proved by the subjective evaluation. In addition, the whole platform was rated with an average score of 8.0 that also confirmed the high performance of the platform and its assistants.

In spite of the good results that we obtained during the subjective and objective evaluations, several interesting ideas can be considered in order to extend the functionalities of the platform, as well as increasing the usability of the information extracted from the database contents:

- *DMA*: Allows the automatic creation of complex data model structures created for each table in the database, allowing the possibility of including complex attributes using the relationships defined in the database between different fields and tables. The assistant could also use the heuristic features in order to select the most probable tables and fields to be used as attributes in the new classes.
- *DCMA*: Extends the capabilities of generating SQL statements and improve the process of defining the input/output parameters of the function prototypes through a graphical interface.
- *UMA*: Incorporation of an innovative methodology for proposing the default values for the confidence levels to ask for information from the users. In this case, we will use the heuristic information of the database and a set of rules to modify the default values specified by the designer in the first stages of the design. Another idea is to extend the user profiles (for instance to young/old people), in order to modify the values of several parameters for confirmation/presentation of information following the results reported in (Wolters et al., 2009).
- *MLEA*: Extends the generation of vocabulary files for the speech recognizer by automatically creating them from the database contents and heuristic information.
- *General*: Finally, we also consider important to improve the evaluation by incorporating new tasks and databases from other domains such as a travel agency or tourism information kiosk.

Acknowledgements

This work has been supported by ROBONAUTA (DPI2007-66846-c02-02) and SD-TEAM (TIN2008-06856-C05-03). We want to thank the following people for their contribution in the coding of the platform and runtime system: to Rosalía Ramos, José Ramón Jimenez, Javier Morante, Ignacio Ibarz, and Rubén Martín from the Universidad Politécnica de Madrid, and to all the members of the GEMINI project for making possible the creation of the platform described in this paper.

References

Agah, A., Tanie, K., 2000. Intelligent graphical user interface design utilizing multiple fuzzy agents. Interact. Comput. 12 (5), 529–542.

Balentine, B., Morgan, D.P., 2001. How to Build a Speech Recognition Application: Second Edition: A Style Guide for Telephony Dialogs, second ed. Enterprise Integration Group, p. 414. ISBN-13: 978-0967127828.

Bohus, D., Rudnicky, A., 2009. The RavenClaw dialog management framework: architecture and systems. Comput. Speech Lang. 23 (3), , 332–361.

Chung, G., 2004. Developing a flexible spoken dialog system using simulation. Assoc. Comput. Linguist. (ACL), 63–70.

Cordoba, R., Fernández, F., Sama, V., D'Haro, L.F. et al. 2004. Implementation of dialog applications in an open-source VoiceXML

platform. In: Internat. Conf. on Spoken Language Processing (ICSLP), pp. I-257–260.

D'Haro, L.F., 2009. Speed Up Strategies for the Creation of Multimodal and Multilingual Dialog Applications. Ph.D. Dissertation thesis. Universidad Politécnica de Madrid. Available at <http://www-gth.die.upm.es/~lfdharo/index_en.php?status=publications>.

D'Haro, L.F., Cordoba, R., Ferreiros, J., Hamerich, S.W., Schless, V., Kladis, B., Schubert, V., Kocsis, O., Igel, S., Pardo, J.M., 2006. An advanced platform to speed up the design of multilingual dialog applications for multiple modalities. Speech Comm. 48 (8), 863–887.

D'Haro, L.F., de Cordoba, R., San-Segundo et al. 2004. Strategies to reduce design time in multimodal/multilingual dialog applications. In: Internat. Conf. on Spoken Language Processing (ICSLP), pp. IV-3057–3060.

Eberman, B., Carter, J., Goddeau, D., 2002. Building VoiceXML Browsers with OpenVXI. In: 11th Internat. Conf. on WWW, pp. 713–717.

Feng, J., Bangalore, S., Rahim, M., 2003. WEBTALK: Mining websites for automatically building dialog systems. In: Workshop on Automatic Speech Recognition and Understanding (ASRU '03). pp. 168–173.

Web page of the GEMINI Project. <http://www-gth.die.upm.es/projects/gemini/> (04.11).

Georgila, K., Fakotakis, N., Kokkinakis, G., 2004. A graphical tool for handling rule grammars in Java speech grammar format. In: Fourth Internat. Conf. on Language Resources and Evaluation.

Hamerich, S.W., 2008. From GEMINI to DiaGen: Improving development of speech dialogs for embedded systems. In: 9th SIGdial Workshop on Discourse and Dialog – Association for Computational Linguistics (ACL), pp. 92–95.

Hamerich, S.W., Wang, Y.-F., Schubert, V. et al. 2003. XML-based dialog descriptions in the gemini project. Berliner XML-Tage, pp. 404–412.

Jung, S., Lee, C., Kima, S., Geunbae Lee, G., 2008. DialogStudio: A workbench for data-driven spoken dialog system development and management. Speech Comm. 50 (8-9), 683–697.

López-Cózar, R., Araki, M. 2005. Spoken, Multilingual and Multimodal Dialog Systems: Development and Assessment. John Wiley & Sons, 262 pp., ISBN: 0-470-02155-1.

McGlashan, S., Burnett, D.C., Carter, J., et al. 2004. Voice Extensible Markup Language (VoiceXML) Version 2.0. W3C recommendation. Available at <http://www.w3.org/TR/voicexml20>.

McTear, M., O'Neill, I., Hanna, P., Liu, X., 2005. Handling errors and determining confirmation strategies—an object-based approach. Speech Comm. 45 (3), 249–269.

McTear, M., 1998. Modelling spoken dialogs with state transition diagrams: experiences with the CSLU Toolkit. In: Internat. Conference on Spoken Language Processing (ICSLP), pp. 1223–1226.

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

Polifroni, J., Walker, M. 2006. Learning database content for spoken dialog system design. In: Internat. Conf. on Language Resources and Evaluation (LREC), pp. 143–148.

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

Tsai, M.J., 2006. VoiceXML dialog system of the multimodal IP-telephony – the application for voice ordering service. Experts Systems Appl. 31, 684–696.

Wang, Y., Acero, A., 2006. Rapid development of spoken language understanding grammars. Speech Comm. 48 (3–4), 390–416.

Wolters, M., Georgila, K., Moore, J., et al., 2009. Reducing working memory load in spoken dialog systems. Interact. Comput. 21 (4), 276–287.