Application of backend database contents and structure to the design of spoken dialog services

Luis Fernando D'Haro , Ricardo de Córdoba, Juan Manuel Montero, Javier Ferreiros, José Manuel Pardo

A B S T R A C T

Current development platforms for designing spoken dialog services feature different kinds of strategies to help designers build, test, and deploy their applications. In general, these platforms are made up of several assistants that handle the different design stages (e.g. definition of the dialog flow, prompt and grammar definition, database connection, or to debug and test the running of the application). In spite of all the advances in this area, in general the process of designing spoken-based dialog services is a time consuming task that needs to be accelerated. In this paper we describe a complete development platform that reduces the design time by using different types of acceleration strategies based on using information from the data model structure and database contents, as well as cumulative information obtained throughout the successive steps in the design. Thanks to these accelerations, the interaction with the platform is simplified and the design is reduced, in most cases, to simple confirmations to the "proposals" that the platform automatically provides at each stage.

Different kinds of proposals are available to complete the application flow such as the possibility of selecting which information slots should be requested to the user together, predefined templates for common dialogs, the most probable actions that make up each state defined in the flow, different solutions to solve specific speech-modality problems such as the presentation of the lists of retrieved results after querying the backend database. The platform also includes accelerations for creating speech grammars and prompts, and the SQL queries for accessing the database at runtime.

Finally, we will describe the setup and results obtained in a simultaneous summative, subjective and objective evaluations with different designers used to test the usability of the proposed accelerations as well as their contribution to reducing the design time and interaction.

1. Introduction

The growing interest from companies in using new information technologies as a means to getting closer to the final users has led to the quick growth of and improvement in automatic dialog systems for tasks involving a database query. In this kind of task, users interact with an automatic system to retrieve or exchange information that is available in a backend database. Examples of this kind of service are flight reservations, information retrieval about entertainment events, or customer care services, etc., 24 h a day; 7 days a week.

One of the main difficulties with these systems is the process of designing them in a fast and flexible way, so that the time needed by the designer to design the service and the time that it will take the user to obtain the desired information in the real-time system can both be reduced. Given the different characteristics and requisites of the final users, the service is also expected to be available in several languages and modalities. It is also expected that the same design can be reused for new applications with minimal modifications.

Fortunately, the increasing demand for automatic dialog systems has resulted in several companies and academic institutions working in the development of fully integrated platforms that provide all of the aforementioned requirements. In general, all current development platforms allow rapid development, debugging, maintenance and deployment of the service; at the same time, thanks to their being made of different and independent assistant modules, they allow different teams of developers to work on the same project at the same time (i.e. collaborative role-based development). Using these modules, designers can specify, among others: the application flow, grammars and system prompts, actions for error handling, integration with backend databases, etc. On the other hand, all these platforms include as their main strategy for accelerating the design, built-in components such as dialog libraries, grammars, and prompts for common situations (e.g. for requesting a phone number, an address, names, etc.). Finally, the usability of these platforms is increased thanks to a clear and fully integrated graphic user interface.

However, in spite of all these advantages, it is surprising to observe that all current development platforms lack of some kind of acceleration 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 together with their relationships). In order to cope with this, we have included several dynamic and intelligent acceleration strategies in our platform such as the prediction of the necessary information required to complete the definition of a dialog state, the definition of the database access functions, incorporation of built-in solutions for presenting lists of retrieved results after querying the database, selection of the information slots to be requested to the user at the same time given their complexity, or the possibility of creating complex dialog states, among others.

1.1. Relevant definitions

Throughout this paper we will refer to some widely used terms which do not necessarily have a general accepted definition but that we want to clarify beforehand.

Acceleration: This term will refer to the different methodologies incorporated into the assistants in order to simplify the design and reduce the design time.

Action: This term will refer to any kind of procedure required to complete the dialog flow, for example calls or jumps to other dialogs, arithmetic or string operations, programming constructs (e.g. if-conditions, while loops), etc.

Data model: This is a graphic and object-oriented representation specified by the designer to represent the fields, tables, and relationships in the database that are relevant to the service, i.e. information to show to or retrieve from the user.

Dialog, state, and slot: In general, the term *dialog* will refer to a specific action used to ask for or show information from/to the users. The term *state* will refer mainly to a set of dialogs and other actions, such as a database access, that are grouped together to achieve a given goal. The term *slot* will refer to any compulsory information that the system requests from the user. For instance, in a banking application, the designer can define a single *state* for carrying out a transaction between two accounts; in this case, the *slots* for this state would be the debit and credit accounts and the amount to be transferred, and in turn, the *state* would include several *dialogs* such as one for asking for the credit account or another one for telling the customer the available balance.

Mixed-initiative and over-answering: Throughout this paper the term *mixed initiative* refers, in accordance with the definitions of the VoiceXML standard, to the system ability for asking for two or more compulsory slots simultaneously from the user, and, where the user answer is incomplete or wrong new sub-dialogs are started to obtain the unfiled slots (for instance, in a flight reservation service, the system asks for the departure and arrival city simultaneously). The advantage of using mixed-initiatives is that they allow more natural interactions between the user and system, and if there are not too many recognizer errors the interactions and time to obtain the information are drastically reduced. On the other hand, the term *over-answering* means that the user is providing additional data – not compulsory at the current state – to the system (for instance, the system asks only for the departure city but the user provides it together with the arrival city). According to the standard, the creation of mixed-initiative dialogs is compulsory for any platform, but over-answering is not required; however, in our platform both are allowed in such as way that the running script can be used in any voice browser.

1.2. Paper organization

This paper is organized as follows: Section 2 provides a brief review of the state-of-the-art on development platforms and acceleration strategies for designing speech-based dialog systems. In Section 3 we present an overall description of the platform architecture and the main accelerations included in the assistants of the platform; then, in Section 4 we will show the setup and results of a subjective and objective evaluation to test the proposed strategies. Finally, Sections 5 and 6 show the future work and conclusions, respectively followed by acknowledgments.

2. State-of-the-art on development platforms and acceleration strategies

This section describes relevant platforms and tools for designing VoiceXML-based applications that were studied and compared with our design platform. In this study, we have made a distinction between systems developed for both research and commercial purposes, since this allows us to make a fair comparison between them and to extract relevant features. For instance, commercial platforms usually present a clear and elegant interface that reflects the great effort that companies are making in this respect, as well as the high level of use of standard languages to increase flexibility and simplify the sharing of files across platforms; features that developers often value highly when they use or compare platforms. For further and more detailed information about these or other development tools, please check the Web address of each of them or refer to López-Cózar and Araki (2005), McTear (2004), and D'Haro (2009).

2.1. Commercial platforms

Summarizing, we can say that all commercial platforms include state-of-the-art modules such as speech recognizers, language identification, speaker verification, and high quality speech synthesizers. They also allow the creation of the service using widespread standard languages and protocols such as VoiceXML,[1] X+V,[2] xHTML, Call Control XML (CCXML),[3] etc., to guarantee the integration between different vendors and platforms. These platforms are often supported by advanced hardware modules that can be used with a minimal programming effort and adapted easily to the runtime system. Interestingly, the most common acceleration in these platforms is the incorporation of a large number of predefined libraries for typical dialog states such as requesting card or social security numbers. They also include assistants for debugging and logging the service, for defining speech grammars and pronunciation dictionaries, or for obtaining service metrics. Finally, they present a very friendly graphical user interface that simplifies the development of very complex dialogs. Since these features are common to most platforms, many of them included in ours, we will only focus on those that are relevant to our system.

Speechdraw[4]**:** Allows the development of very complex dialog applications such as "How may I help you" (Gorin, Riccardi, & Wright, 1997) without requiring any knowledge about VoiceXML. For complex dialog applications, the platform incorporates a graphical flow editor that can switch between the application logic (i.e. the dialog flow with its states, actions and transitions) and the error recovery logic (i.e. the actions and prompts that control the system behavior against typical errors in the recognizer or runtime platform). As an

[1] http://www.w3.org/TR/voicexml21/.
[2] http://www.voicexml.org/specs/multimodal/x+v/12/.
[3] http://www.w3.org/TR/ccxml/.
[4] http://www.speechvillage.com/home/.

acceleration strategy, the error recovery is automatically drawn by using pre-defined rules specified by the designer (following a similar approach as the one we follow in our platform, see Section 3.3). The platform also provides mechanisms to enable different designer profiles (i.e. flow design, grammar and backend design) to work on the same project at the same time. Finally, the IDE allows database emulation by simply entering debugging data into precompiled tables and defining the parameters that must be sent and returned when communicating with the backend. This way, the platform avoids access delays and the strong interdependence between the designer interface and the integration layer when debugging the application (in Section 3.1.2 we describe a similar assistant in our platform that follows the same underlying idea but incorporating new features).

OpenVXML Studio[5]**:** This platform includes configurable built-in modules, similar to some of the templates included in our development platform, which provide the main functionalities supported by the VoiceXML language. Each time a module is used, the visual interface allows the designer to set it up. However, the system only proposes default values when configuring the modules and not state dependent values as we do. In order to reduce the amount of information displayed in the workspace, the GUI uses multiple parallel canvases and a special kind of connector between canvases called wormholes. Our platform includes a similar functionality, as well as the possibility of encapsulating dialog actions in order to provide basic or detailed information.

On the other hand, the platform incorporates the concept of Business Objects that represent the fields of the database that the designer defines as necessary for the current service (a similar concept to the classes and attributes used in our platform, see Section 3.1.1). These objects are used later, among others, during the definition of the database queries to retrieve complex objects. At design time, the platform asks the developer to match each particular attribute of the object to a dialog variable that is used to provide the information retrieved from the database to the user. In our platform, we follow a similar approach but we go a step forward by creating a semi-automatic procedure that proposes the best matching, which automatically creates the dialog variables (see Section 3.2.2). Finally, the platform accelerates the creation of system prompts allowing the designer to type in the words that make up the message and then complete it by using dynamic values stored in the dialog variables following a procedure similar to the one we have implemented in our platform (see Section 3.3).

Vocalocity App Center[6]**:** This platform provides several configurable object-models that can be connected to each other through conditioned or direct transitions. Each object includes its own kind of configurable parameter and may have one or several outputs depending on its configuration for error handling or if there are different output results. The most interesting feature of the platform is that the design of almost any basic service can be specified in four steps, i.e. dragging and dropping three objects into the application canvas and connecting them sequentially. The first step, called the Ask step, consists of the creation of an action for requesting information from the user. The second step, called the Data step, represents the process of accessing and retrieving information from the backend database. The third step, called the Tell step, corresponds to the action of providing the retrieved information to the user. Finally, during the fourth step, called the Publish step, the designer creates the VoiceXML script and configures the platform in order to make the service available. In our platform we have developed a very similar approach where most of the actions required to define a dialog correspond to the first three steps described above (see Section 3.2.2) and the final script is also automatically created.

2.2. Academic and research platforms

In contrast to most commercial platforms, academic and research platforms allow designers to create more complex dialog interactions by providing features not included in any standard description language. They also allow the creation of more complex dialogs, some are freely available as open source, and their functionalities can be extended by using proprietary or third party modules. The following are noteworthy examples of tools developed in academic environments.

DialogStudio: Described in detail in Jung, Lee, Kima, and Geunbae Lee (2008), this platform integrates several tools that cover the different steps in designing a data-driven spoken dialog system, i.e. from preparing the data to testing the service. One of the main objectives of the platform is to provide a complete set of functionalities for preparing the input files to be used by the speech recognizer, language understanding, and dialog manager modules. The platform also provides an annotation environment for tagging semantic and knowledge information, as well as dialog examples; in this case, the platform uses a meta-model language that allows the quick definition and adaptation of semantic and dialog structures to domain specific knowledge. Thanks to these accelerations the platform also provides an average time reduction of 30% when compared to other editors on the creation and annotation of different tasks.

CSLU RAD Toolkit[7]**:** Allows the development of multimodal system initiative dialogs (combining voice, DTMF, interactive images, and animated agents), by using a representation based on state-transition networks (McTear, 1998) that describe the different functions and actions in the dialog. The states and transitions of the dialog flow are created using a toolbar with objects that can be dragged and dropped into the canvas and connected with arrows to other objects. The toolkit reduces the information displayed on the canvas by grouping repetitive or common actions. The toolkit includes embedded configurable speech recognition and text-to-speech systems, and an animated agent with configurable and synchronizable facial expressions (i.e. the face/lip movements of the agent can be aligned with the speech prompt).

Finally, Paternò and Sisti (2010) describe an integrated environment for designing ubiquitous multi-device dialog applications. One of the main advantages of this platform is the use of a high-level definition language that can be automatically transformed using XLST sheets into the corresponding script for each modality (e.g. VoiceXML for speech or any Web-based standard). For the speech modality, this language allows designers to specify all of the typical actions and elements included in the VoiceXML standard, for instance: menu-based dialogs, grammars, error recovery, prompts, recordings, fields, scripts, etc. In our platform we have developed a similar high level XML syntax that is also automatically transformed into the final VoiceXML script used by the runtime system (see the beginning of Section 3). They also describe in this paper interesting solutions to support the same kind of user interaction provided in visual interfaces (e.g. Web) such as: spin boxes or multiple vocal selections by using only elements supported in the VoiceXML standard. In this paper, we also describe similar solutions in our platform to allow over-answering dialogs or to jump back to previous dialogs in the flow.

2.3. Research platforms that provide assisted dialog design

As we have already said, surprisingly most of the aforementioned commercial and academic platforms do not include any kind of acceleration based on the content and structure of the

[5] http://www.eclipse.org/vtp/openvxml-announce/content/html/index.htm.

[6] http://www.vocalocity.com/products/productdetail.cfm?productid=100007.

[7] http://cslu.cse.ogi.edu/toolkit/.

backend database, which provide important information to accelerate the design when developing the dialog service. This section describes examples of the most relevant systems and strategies that we can find in the literature based on using the database contents to accelerate the design of the dialog applications.

In Polifroni, Chung, and Seneff (2003) and Polifroni and Walker (2006) a rapid development environment for creating spoken dialog applications using online content is described. The development process starts by extracting knowledge from various Web applications and composing a dynamic database from it. The dialog flow is determined at runtime depending on the content of this database. The authors propose a methodology for creating automatic clusters that group together and organize numerical data into symbolic data. Thus a symbolic concept such as cheap/medium/expensive in the domain of a hotel reservation is automatically created according to the information in the database (i.e. hotel rates vary depending on the city). At runtime, the system also summarizes the partially retrieved information and dynamically creates the prompts to present it to the users, determining at the same time the order in which they appear based on the most useful set of attributes to narrow down the current data subset.

In Chung (2004), the database content 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 understanding module, as well as diagnosing the system behavior against problematic user interactions or unexpected user answers, etc. In Wang and Acero (2006), the database contents are used to accelerate the creation of grammars for the speech recognition and spoken language understanding modules. The system uses the database to generate a large number of artificial sentences that are integrated into semantic frames in order to create customized grammars for different scenarios.

Feng, Bangalore, and Rahim (2003) present a different approach. In this case, they do not extract information from a backend database but they apply data mining techniques to the content of corporate websites for automatically creating spoken and text-based dialog applications for customer care. The process is carried out through a Website analyzer that exploits the content and structure of the site in order to generate structured and semi-structured task data. Then, the generated data is classified according to predefined information units (e.g. menu, question-answer, topic-explanation, etc.). With this information, the dialog manager, at the runtime system, will identify the focus or expectation of the user question and will provide a concise answer. Although the dialog flow is not defined using a GUI, it checks that important knowledge for the different modules of a dialog system can be extracted from well-designed contents.

3. Platform architecture

Fig. 1 shows the architecture and information flow between the modules of the platform described in this paper. The platform includes several independent modules that have been integrated into a common GUI to guide the designer in the design, step by step, and at the same time, let him go back and forth. The platform is divided into three main layers in order to separate the aspects that are service specific, those corresponding to the high-level dialog flow of the application (modality and language independent), and the specific details imposed by the speech modality and languages.

On the other hand, it is important to mention that in order to allow the sharing of information between modules in the platform and to facilitate the process of converting the generated model into the corresponding script required for different modalities, we have developed an object oriented XML language named GDialogXML (Schubert & Hamerich, 2005). The main feature of this specification is its flexibility, allowing the modeling of all application data, database access functions, definition of dialog variables and actions needed at each state, system prompts, grammars, user models, etc. The syntax also allows the updating to new versions of the VoiceXML standard with little effort since the platform includes an automatic translator module that can be modified accordingly. Throughout the paper we will include fragments of the generated code for the assistants giving suitable explanations of them; however, the complete specification can be consulted in Web page of the GEMINI Project (2010). In order to clarify the design process, the assistant functionalities, and the proposed acceleration strategies throughout this section we are also going to use a running example to show the process of creating a typical dialog where one of the goals is to perform a bank transfer between accounts by asking the user for a known and customized alias of the source account and destination account, and the amount of money to be transferred.

3.1. Accelerations to the definition of the data model and database access

Following the diagram in Fig. 1, first the designer needs to specify the overall aspects of the service and data. For instance, in the

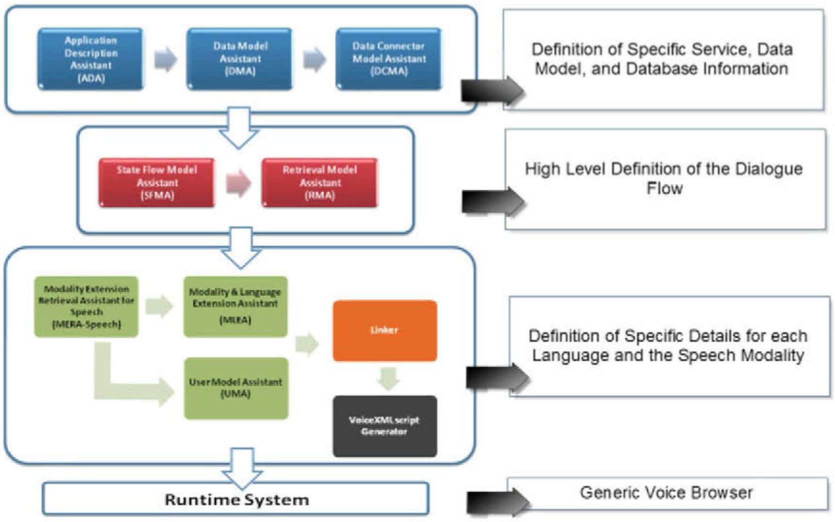

Application Description Assistant (ADA)
Data Model Assistant (DMA)
Data Connector Model Assistant (DCMA)
Definition of Specific Service, Data Model, and Database Information
State Flow Model Assistant (SFMA)
Retrieval Model Assistant (RMA)
High Level Definition of the Dialogue Flow
Modality Extension Retrieval Assistant for Speech (MERA-Speech)
Modality & Language Extension Assistant (MLEA)
Linker
User Model Assistant (UMA)
VoiceXML script Generator
Definition of Specific Details for each Language and the Speech Modality
Runtime System
Generic Voice Browser

Fig. 1. Summarized version of the platform architecture.

first assistant, the designer defines general default values for the speech modality such as the number of results to be provided, number of times to repeat a question before redirecting the call to an operator, confidence values for different kinds of confirmations, languages of the service, etc.

3.1.1. Creation of the data model structure

The next step is to specify the database structure by using an object-oriented Data Model representation using classes, attributes, and relationships (see Fig. 2). The idea of these classes and attributes is to allow designers to provide information to the assistants as to which fields in the database are relevant for the service (i.e. to provide or request information to/from the users), as well as the relationships between the attributes and the table and field that they refer to in the backend database.

In the figure we can see that the *Transaction* class includes three attributes corresponding to the information required to carry out the transfer: the *TransactionAmount* (defined using a basic type, i.e. Float) and two object references to the class *Account* that the system will use to have access to the account alias (*AccountAlias*) and the available balance before or after carrying out the transfer (*AvailableBalance*). The class *Account* also includes other information such as references to other complex classes not shown in the figure. The figure also shows a section of the GDialogXML code for the classes (number 1) and attributes in the example, as well as information on the database table and its related fields (number 2, i.e. table *accounts* and field *account_alias*).

In order to define the classes and attributes, the assistant includes a wizard window that allows the designer to select from all the tables and fields in the database and which ones to use. After selecting the fields, the assistant automatically sets the field type, name, and relationship to the database for each new attribute.

As well as the definition of the data model structure, the assistant uses an open SQL statement to extract information from the database contents such as the name and number of the tables, fields, and records. In addition, the following heuristic information for each field is also calculated: (a) field type, (b) average length in characters, (c) number of empty records, (d) language dependent fields, (e) proportion of records that are different, (f) number of different words (i.e. vocabulary), (g) average number of words, (h) average length of the words, and (i) number of different records. This information is used afterwards to simplify the design or to improve the presentation of information in later assistants. For instance, when using the wizard window to define the data model classes, the system uses these heuristics to reduce and sort by relevance the fields that can be used to define the class attributes. Thus, if we have a field in a table with a high number of empty values or if the field type is considered complex (e.g. URL addresses, bitmaps, auto-incremented fields), or the field has a long number of words (e.g. as in a memo type field) they will not be shown since they are not usually used in speech applications. We also use them to propose which slots can be unified in order to be requested at the same time by the user (see Section 3.2.1), or for creating automatic dialog proposals (see Section 3.2.2).

3.1.2. Creation of database access functions

The next assistant in the platform is used to define the prototypes of the database access functions needed for the real-time system. The advantage of using prototypes is that their actual implementation is not required during the design of the dialog flow. The main acceleration strategy included in this assistant is the possibility of relating the input/output arguments to the attributes and classes of the data model.

Fig. 3 shows the XML code and a section of the assistant window to define the input and output arguments of the function prototype. In the figure, number 1 corresponds to the input arguments (i.e. account aliases and amount to transfer) and number 2 the output argument (i.e. the available balance after carrying out the transfer). Number 3 and 4 are the information on the related class and attribute in the data model (*Account.AccountAlias*) and the relationship to the table and field in the database (*accounts.account_alias*). This information is then used to create dialog/state proposals and to propose database access functions automatically for a given dialog in the design (see Section 3.2.2).

We have also included a wizard window that semi-automatically creates the SQL statements for the given prototype and provides a preview of the results that the system would retrieve in the real-time system (see Fig. 4), thus reducing the need to learn a new programming language (SQL), and simplifying the inclusion of the generated query in the Java servlet created for accessing the database at runtime. Currently, few development platforms include this kind of help. The few platforms that include a similar SQL wizard only provide debugging tools or support for many query language standards but no automatic query proposals.

For creating the SQL query, the assistant uses the output arguments (defined in the function prototype) as returned fields for the SELECT/UPDATE clause, and the input arguments as constraints for the WHERE clause. During this process, the wizard uses the heuristic field type in order to propose and debug the SQL statement correctly (i.e. the statement to retrieve the information from a string field is different from the one used to retrieve a number). Since the input/output arguments could be defined using different

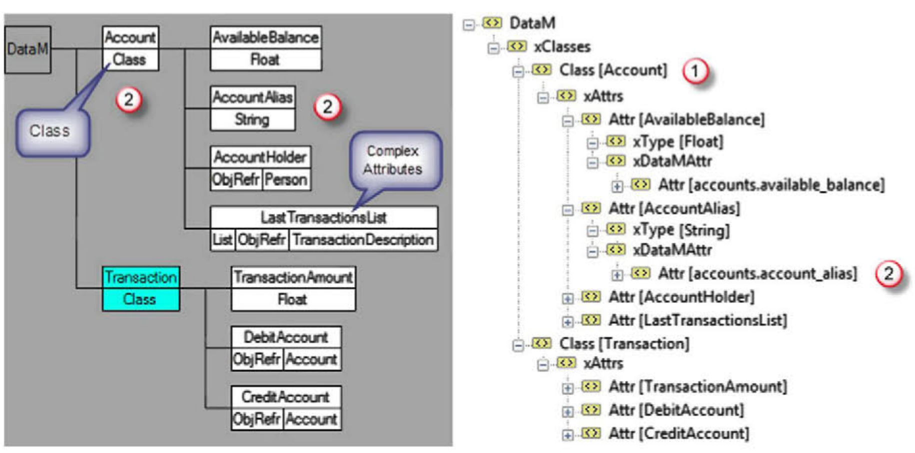
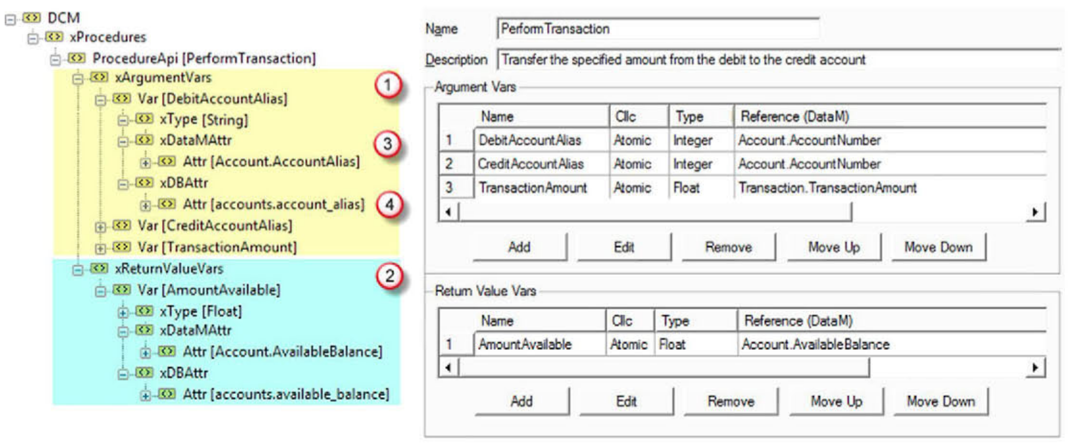


Fig. 2. Example of object oriented classes in the data model.



```
DCM
  xProcedures
    ProcedureApi [PerformTransaction]
      xArgumentVars
        Var [DebitAccountAlias]
          xType [String]
          xDataMAttr
            Attr [Account.AccountAlias]
          xDBAttr
            Attr [accounts.account_alias]
        Var [CreditAccountAlias]
        Var [TransactionAmount]
      xReturnValueVars
        Var [AmountAvailable]
          xType [Float]
          xDataMAttr
            Attr [Account.AvailableBalance]
          xDBAttr
            Attr [accounts.available_balance]
```

Name: PerformTransaction

Description: Transfer the specified amount from the debit to the credit account

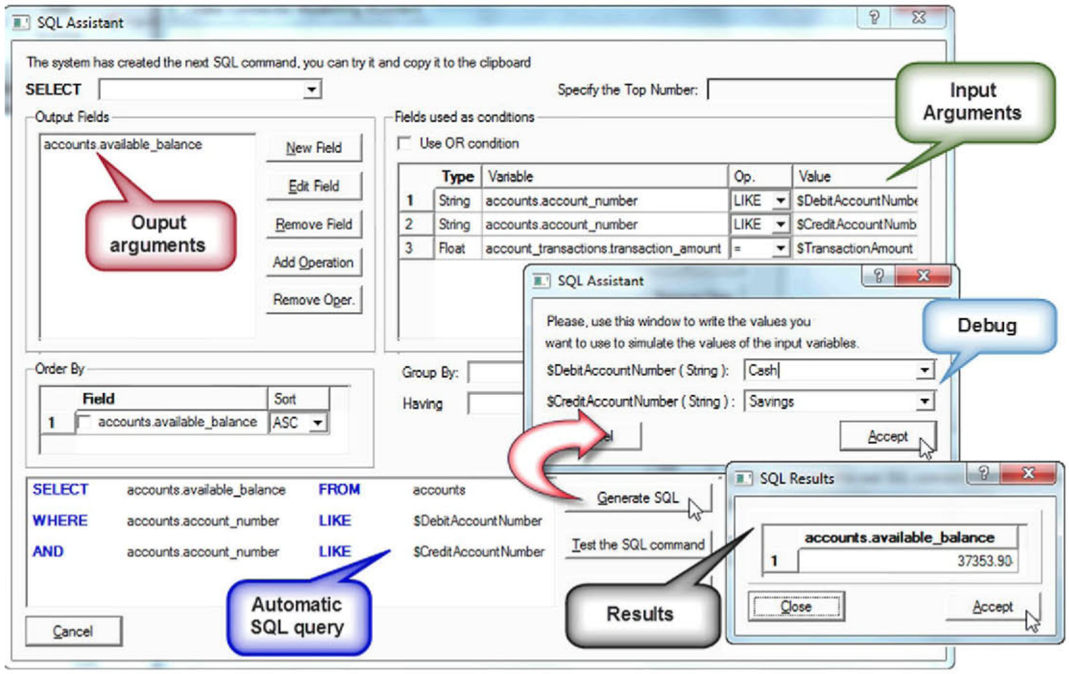
Argument Vars

	Name	Cllc	Type	Reference (DataM)
1	DebitAccountAlias	Atomic	Integer	Account.AccountNumber
2	CreditAccountAlias	Atomic	Integer	Account.AccountNumber
3	TransactionAmount	Atomic	Float	Transaction.TransactionAmount

Return Value Vars

	Name	Cllc	Type	Reference (DataM)
1	AmountAvailable	Atomic	Float	Account.AvailableBalance

Fig. 3. Example of a database access function prototype.



SQL Assistant

The system has created the next SQL command, you can try it and copy it to the clipboard

Specify the Top Number:

Output Fields: accounts.available_balance

Fields used as conditions

	Type	Variable	Op.	Value
1	String	accounts.account_number	LIKE	$DebitAccountNumbe
2	String	accounts.account_number	LIKE	$CreditAccountNumb
3	Float	account_transactions.transaction_amount	=	$TransactionAmount

Order By

	Field	Sort
1	accounts.available_balance	ASC

```
SELECT   accounts.available_balance   FROM   accounts
WHERE    accounts.account_number      LIKE   $DebitAccountNumber
AND      accounts.account_number      LIKE   $CreditAccountNumber
```

SQL Assistant

Please, use this window to write the values you want to use to simulate the values of the input variables.

$DebitAccountNumber (String): Cash

$CreditAccountNumber (String): Savings

SQL Results

	accounts.available_balance
1	37353.90

Fig. 4. Wizard window for the automatic creation and testing of SQL queries.

types (i.e. atomic or object oriented), several strategies are applied in order to create SQL queries that can use such kind of parameters. For instance, if the argument is atomic the query uses the argument directly, but if the argument is an object the system flattens it and provides a list with only the atomic elements allowing the designer to select which ones to use in the SQL query from that class.

Moreover, the assistant allows the inclusion of new input or output arguments if the function prototype is not complete or if the designer wants to test new combinations of arguments. The assistant also allows the inclusion of several constraints supported by the SQL language such as math functions (average, max, min, etc.), sorting, selection (Top or Distinct), clustering (Group By), Boolean operators (And, Or) for combining the query restrictions, among others. Finally, the wizard allows the designer to preview the records that the proposed SQL statement will retrieve in real-time. In order to debug the query, the designer specifies the values for the input arguments of the function, using a pop-up window, to test the query (for acceleration, the wizard automatically proposes the most common values for the given field in the database using the heuristic information).

3.2. Accelerations to the definition of the dialog flow

The following step in the design is to define the dialog flow at a high-level, i.e. without specific language or modality information. The first assistant is used to define the states, transitions between states, and the compulsory information that has to be asked to the users at each state (which we call slots). Then, in the second assistant, the designer defines all the actions (e.g., variables, 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. Thanks to this two-step process, the specification of all the detailed actions in the last assistant is accelerated considerably. Since these assistants are the most complex ones, here we have incorporated the most important accelerations. Below we will describe the most interesting ones. For further details please refer to D'Haro et al. (2006) and D'Haro, Cordoba, Lucas, Barra-Chicote, and San-Segundo (2009).

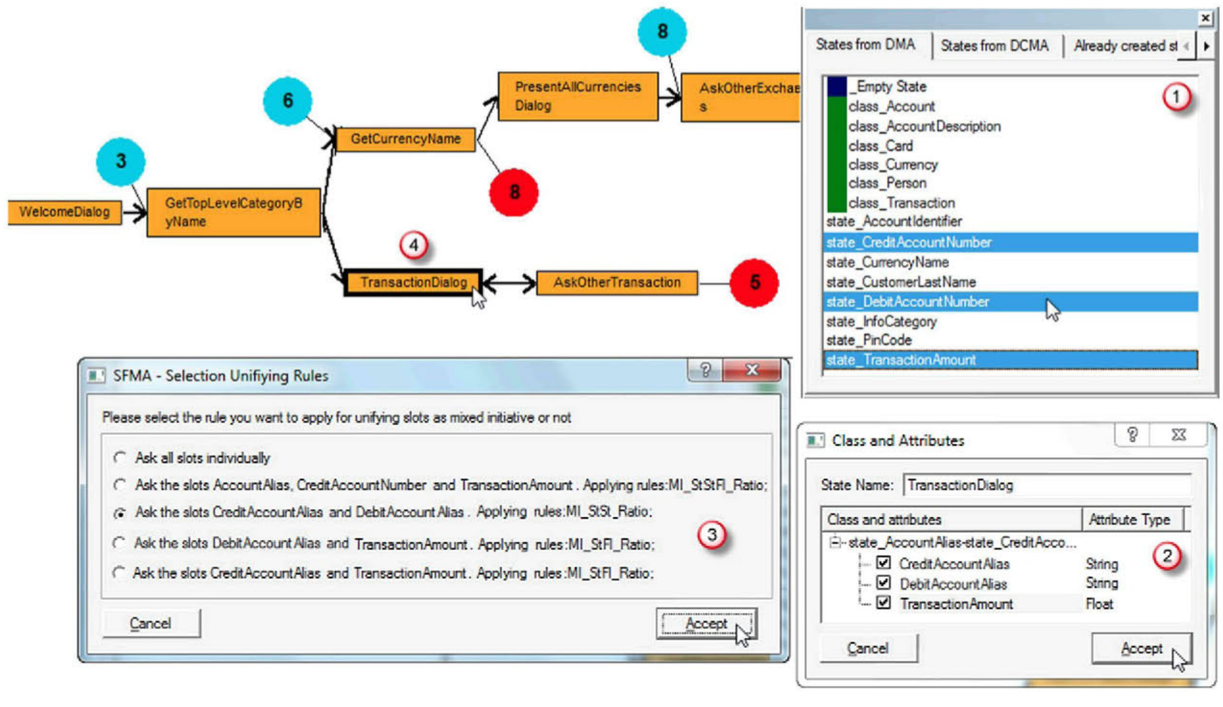
3.2.1. Creation of the dialog flow

There are currently two major kinds of dialog models: Non-structured and Structured. In the former, the model allows the highest degree of freedom interaction between the user and the system, since both of them are able to reason and to negotiate the goals and information. In the latter, the dialog follows predefined paths; here the richness of the dialog is limited because of the current limitations of the technology (i.e. speech recognizers, understanding modules, and language generation). On the other hand, structured models can be divided into two major categories: Finite-state (where the dialog is expressed as a state-transition network that is naturally and easily represented using a graphical interface) and Frame-based models (that follow the form-filling concept in which predetermined information is required to be filled in from the users) which is the only one supported by VoiceXML.

In our platform we have followed, in order to display and define the dialog flow, the same kind of representation used in most of the current development platforms, i.e. a state transition network that is made up of different states (forms or frames in VoiceXML) consisting of several pieces of information that have to be filled in (i.e. slots). The GUI allows the definition of new states by using wizard-driven steps and a drag-and-drop interface. Below, we describe the most important accelerations included these assistants.

Reduction of the confusability in the canvas: In this assistant we have implemented an automatic algorithm that helps the designer place the objects on the canvas in order to reduce the visualization problems produced when all the transitions between states are displayed. Here we have used the main idea incorporated into the OpenVXML Studio platform (see Section 2.1), i.e. using connectors between states when they are far from each other (i.e. circles in Fig. 5). The proposed algorithm detects when to use connectors or solid lines to represent the transition between states. The final decision depends on two factors: (a) the distance in pixels between the connected states in the GUI, and (b) the number and size of the objects that are along the path of the connection line. Thus, for instance, if the distance is longer than one third of the size of the canvas or if when placing a solid line the number of objects that collide with the candidate line is greater than two, then the system uses the connectors (see circles with number 8 in Fig. 5). The direction of the flow is indicated using arrows; in number 4, we can see that the *TransactionDialog* and *AskOther-Transaction* states are linked together with a forward and backward arrow that indicates the possibility of returning back from the latter to the former. Finally, the GUI allows the possibility of quickly finding states by name or by clicking the connector to go to the connecting state.

Automatic State Proposals: In order to accelerate the creation of the states we have included several configurable state templates, available in a floating window through the GUI, that include the information as to the slots to be requested to the user at each state. Three kinds of proposal state templates are available: (a) created from the classes in the data model, (b) created from attributes with database dependency, and (c) created from the database access functions. In detail, the first kind of template is created for each class defined in the data model. When these templates are used, the assistant allows the designer to select which attributes to use as definitive slots in the new state. Italso expands complex attributes (with inheritance and objects) allowing only the selection of atomic attributes because only these can be asked to the user in the real time system. The second kind of template is created from any attribute defined in the data model related to a database field only if it has been used as an input argument in any of the database access functions. The main motivation for proposing these states is that these attributes are very likely to be asked to the user. 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 doing the drag and drop and allowing the creation of states with multiple slots (as we can see in Fig. 5 in numbers 1 and 2). Finally, for the third kind of template the system analyzes all the prototypes of the database functions containing input arguments defined as atomic



_Empty State
class_Account
class_AccountDescription
class_Card
class_Currency
class_Person
class_Transaction
state_AccountIdentifier
state_CreditAccountNumber
state_CurrencyName
state_CustomerLastName
state_DebitAccountNumber
state_InfoCategory
state_PinCode
state_TransactionAmount

SFMA - Selection Unifiying Rules

Please select the rule you want to apply for unifying slots as mixed initiative or not

- Ask all slots individually
- Ask the slots AccountAlias, CreditAccountNumber and TransactionAmount . Applying rules:MI_StStFl_Ratio;
- [✓ Ask the slots CreditAccountAlias and DebitAccountAlias . Applying rules:MI_StSt_Ratio;]
- Ask the slots DebitAccountAlias and TransactionAmount . Applying rules:MI_StFl_Ratio;
- Ask the slots CreditAccountAlias and TransactionAmount . Applying rules:MI_StFl_Ratio;

Cancel   Accept

Class and Attributes

State Name: TransactionDialog

Class and attributes	Attribute Type
state_AccountAlias-state_CreditAcco...	
☑ CreditAccountAlias	String
☑ DebitAccountAlias	String
☑ TransactionAmount	Float

Cancel   Accept

Fig. 5. Example of creation of a state with mixed-initiative.

types. Then, the system uses the name of the function as proposal for the name of the state, and the input arguments as slots for that state. Fig. 5 shows an example of the floating window with the template proposals (number 1) and how the designer, for the current example, selects three attributes with database dependency (i.e. *state_TransactionAmount*, *state_DebitAccountAlias*, and *state_CreditAccountAlias*), and then the three attributes to be used as slots (number 2) in the new state *TransactionDialog* (number 4).

Automatic proposal of unifying slots: This is one of the most innovative accelerations since the system suggests when two or more slots in the state must be requested one by one (using directed forms) or together (using mixed initiative forms) according to the VoiceXML standard. The main goals of this functionality are: (a) to improve the performance of the speech recognition system by avoiding difficult data to be asked simultaneously (e.g. two dates, long number, or fields with a high vocabulary), therefore increasing the time users spend with the system due to confirmations, (b) to increase user satisfaction by allowing them to answer system prompts by using more natural expressions, and (c) to provide an objective criteria for selecting the slots that can be especially useful for designers with a reduced knowledge of speech technologies.

The proposal is based on the heuristic information extracted from the database contents related to the corresponding slots to ask to the user and on a set of predefined, but editable, rules included in the platform from our knowledge by carrying out dialog applications and from known guidelines in this area (Balentine & 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 from two to three slots with different field types (e.g. three strings, one string and one integer, two dates, two floats). Thus, for instance, if we need to ask for two numeric data with a proportion of different values close to one (heuristic e in Section 3.1.1), and the total number of records of both fields is high (heuristic i), then the system determines that these slots have a large vocabulary and a high probability of misrecognition, therefore it is better to ask one slot at a time (i.e. directed forms). In another rule, mixed initiative is not allowed if there are two slots defined as strings (heuristic a) and the sum of the average length of both is longer than 30 characters (heuristic b). In this case, the system tries to avoid the recognition of very long sentences. Finally, an example of a rule that allows mixed-initiative is when we have two slots (one string and the other float) but the vocabulary size of both fields is no larger than 2000 words (configurable value). The assistant includes a rule editor that the designer can use to add, edit, delete or disable rules. In addition, the editor allows the designer to export or import a rule file in order to be able to include rules that have been successfully used in previous applications.

In Fig. 5 we see an example where the designer is trying to create a state for carrying out a banking transaction, where three slots are needed: *debitAccountAlias* (string), *creditAccountAlias* (string), and *transactionAmount* (float) in accordance with the heuristics of each one and the predefined set of rules, the system shows a list with different kind of unifications for these slots (number 3). By default the assistant selects the option that covers the higher number of slots at the same time (i.e. second option in the figure); however, the designer can choose another one (i.e. selecting the third option as in the figure).

3.2.2. Definition of dialog actions

The next step in the design is to define the actions to be carried out at each of the previously-defined states. Here, for instance, the designer specifies the control logic within each state and the conditions to jump to other dialogs, the dialogs to ask for or to show information from or to the users, the slots that can be filled by using over-answering dialogs, the functions to be used each time to retrieve or modify information in the database, the procedures to modify dialog variables (e.g. using math or string operations), etc. Below only three of the most important accelerations in this assistant are described; for other strategies or further details please refer to D'Haro et al. (2006).

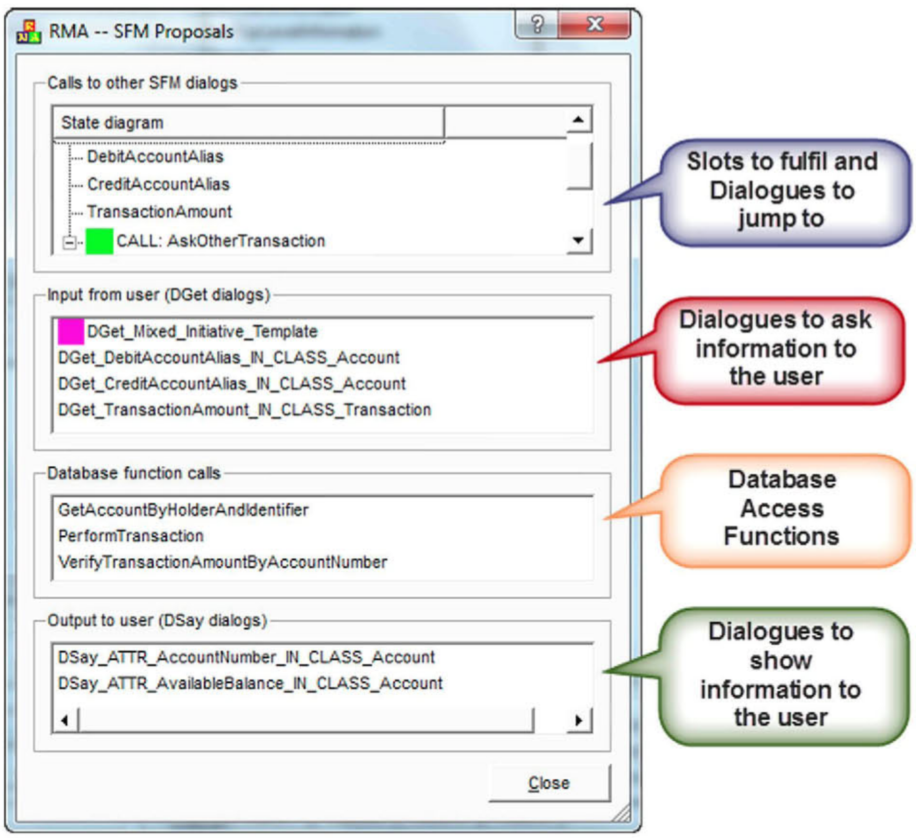
Automatic dialog template proposals: The first time the assistant is started it analyzes the data model information in order to create automatically, from all attributes defined as atomic types, configurable and generic dialog templates that can be used to request or provide information to the user (with prefix DGets and DSays respectively) at any time in the design. If the attributes are complex or include object inheritance, the assistant provides configurable DSay dialogs by using a template that shows the class and its attributes, expanding the complex attributes (with inheritance and objects) and selecting any attribute that will form part of the prompt. The assistant does not generate DGet dialogs from these complex classes since they cannot be requested from the users in real-time (i.e. users can only fill in information about atomic attributes). Other DSay dialog templates are also available: generic DSay to provide concepts (useful in case of providing help to the users), configurable DSay to present variables from a dialog, DSay to present lists of objects retrieved from the database, and predefined DSay such as: Welcome, Goodbye, Transfer to operator, etc. All these dialogs are available to the designer to be used at any time through a dockable toolbar near to the workspace. The list of all the generated dialogs is then filtered in the next strategy in order to provide relevant actions for defining each state.

Automatic action proposals for each state: The idea of this acceleration is to show the designer in a single window all of the typical actions in an information retrieval sequence required to complete a state, i.e. the dialogs to fill in the slots (DGet), the function to carry out the database access, the dialog used to show the results (DSay) and the conditions to jump to following states (similar to the steps proposed in the Vocalocity platform, Section 2.1).

Fig. 6 shows the proposals for the state that carries out a transaction between two accounts. By using this window the designer would only need to drag and drop into the state edition window (not shown in the figure) first the mixed-initiative template (*DGet_Mixed_Initiative_Template*) in order to request both account aliases, then the dialog to ask for the amount (*DGet_TransactionAmount*), then the database access function (*PerformTransaction*), and finally the dialog to inform the user of the available balance (*DSay_AvailableBalance*).

In order to decide which actions are relevant, the wizard applies the following rules:

A. For the selection of the DGet dialogs: the system filters out the list of automatic dialog templates (mentioned above) by selecting only those that have the same data model relationship (i.e. from the same attribute or database information) as the slots to be filled in the current state; if the filtered list is empty, the system searches similarities in the name or attribute type.
B. For selecting the database access functions: the system first considers functions with the same number and type of input parameters as the defined slots for the current state. The next criterion is to apply a partial matching in the number of arguments but maintaining the same relationship with the data model. Finally, the relaxed criterion is to find functions with similar names or arguments to the slots and dialog. If even with the relaxed filter no function is selected/proposed, it would probably mean that there is no database access function suitable for that state. Therefore, the assistant offers the possibility of going back to the assistant for creating a database function and then reload the proposals.


RMA -- SFM Proposals

Calls to other SFM dialogs

State diagram
- DebitAccountAlias
- CreditAccountAlias
- TransactionAmount
- 🟩 CALL: AskOtherTransaction

Input from user (DGet dialogs)

🟪 DGet_Mixed_Initiative_Template
DGet_DebitAccountAlias_IN_CLASS_Account
DGet_CreditAccountAlias_IN_CLASS_Account
DGet_TransactionAmount_IN_CLASS_Transaction

Database function calls

GetAccountByHolderAndIdentifier
PerformTransaction
VerifyTransactionAmountByAccountNumber

Output to user (DSay dialogs)

DSay_ATTR_AccountNumber_IN_CLASS_Account
DSay_ATTR_AvailableBalance_IN_CLASS_Account

Close
Slots to fulfil and Dialogues to jump to
Dialogues to ask information to the user
Database Access Functions
Dialogues to show information to the user

Fig. 6. Window with proposal of actions to complete a state.

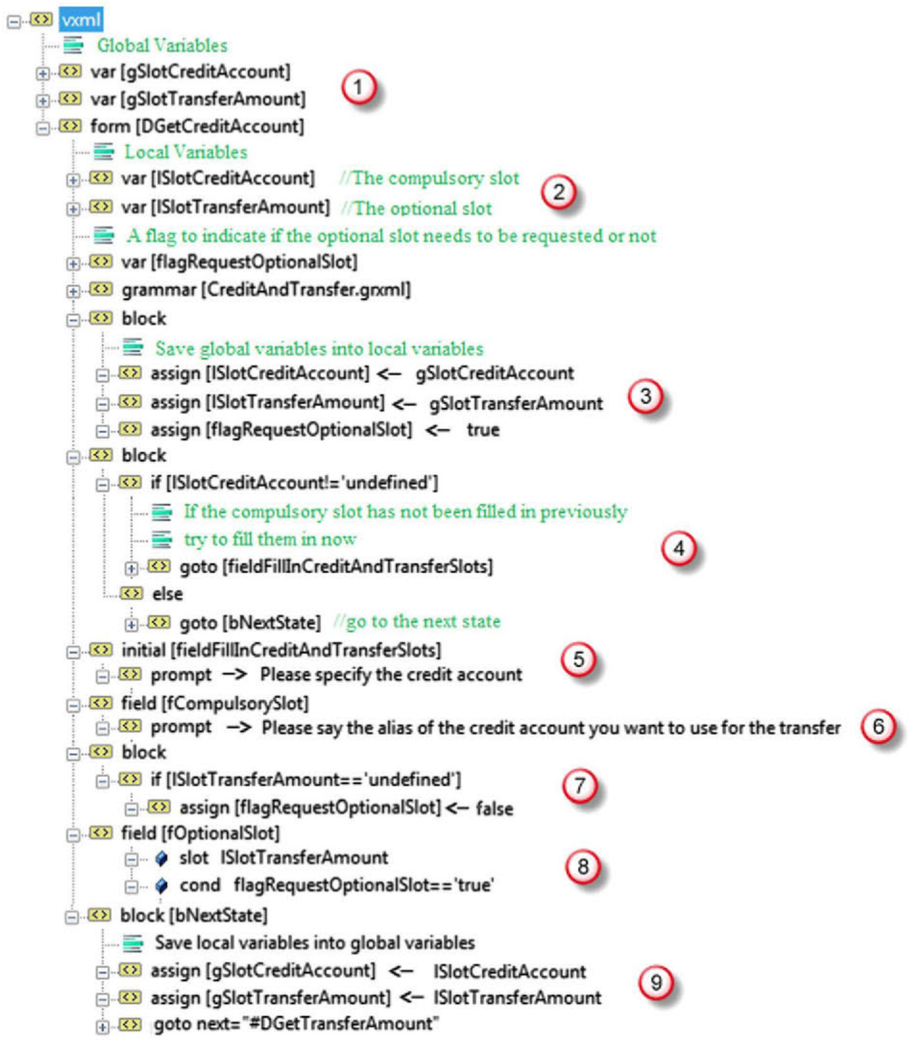
C. For selecting the DSay dialogs the system uses the same procedure as for the DGet dialogs; here we also include DSays specific to the values returned by the database functions selected in the previous step.
D. For selecting the dialogs and slots to jump to: the system directly uses the information from the dialog flow. The information about slots in the following states is used to allow the designer to create dialogs with over-answering.

Creation of over-answering and mixed-initiative dialogs: As we mentioned in Section 2, current development platforms include the possibility of creating mixed-initiative dialogs since this is compulsory in the VoiceXML standard. However, in our platform we have included the possibility of creating over-answering dialogs by creating a special flow using standard elements in VoiceXML (i.e. form, initial, block, goto, and if conditions). First we will briefly explain the procedure to create these dialogs and then the special flow designed to overcome the limitations of VoiceXML.

A. Mixed-Initiative: the system offers a configurable template (see the square item in the field for dialogs to ask information to the user in Fig. 6) which the designer can drag and drop over the dialog that is being edited. The template shows the slots specified to be requested using mixed-initiative for the current state. In addition, the template gives the possibility of adding optional slots to be used for over-answering at the same time. With this information, the system generates the same flow proposed in the VoiceXML standard for mixed initiative dialogs by using the GDialogXML syntax (i.e. a form tag containing one initial element for asking for several slots at the same time, then one field tag for each slot in order to handle the situation in which the user answers partially or only some of the slots are filled after the recognition, so the system has to ask again for unsolved slots, and handle the keeping of the optional slots when they are input by the user).
B. Over-Answering: when the designer drops any DGet dialog, the system shows a pop-up window to allow additional slots to be selected as over-answering from the current state and slots from the following states in the flow (with a limit of two in the hierarchy). By default, the slots defined as optional in the previous assistant are automatically converted into over-answering slots here.

In order to define the special logic for allowing over-answering in the VoiceXML script the system checks, before any call to a DGet dialog, whether the data to be asked for has been already obtained in a previous state in the flow (as would be the case with over-answering).

Fig. 7 shows an example of the VoiceXML script and the logic created to support over-answering dialogs. As we can see, the proposed flow only uses elements supported by the standard although the code is a little more elaborated. In the example, the system requests the credit account alias as a compulsory slot and the transfer amount as an optional slot. In number 1 we define both slots as global variables in order to allow the slots to be filled in at any state and document, or to allow jumping back to previous states in the flow. Then in the form (number 2 and 3) we declare local variables to have a copy of the values stored in the global variables. In number 4 we define a block that analyzes whether the compulsory slot has been previously filled in or not (i.e. it is possible that in the previous dialog we allowed this compulsory slot to be an optional one at that state and the user filled in at that time). If that the slot is empty, the system will try to fill it in by using the same



```
vxml
  Global Variables
  var [gSlotCreditAccount]                                        1
  var [gSlotTransferAmount]
  form [DGetCreditAccount]
    Local Variables
    var [lSlotCreditAccount]    //The compulsory slot             2
    var [lSlotTransferAmount]   //The optional slot
    A flag to indicate if the optional slot needs to be requested or not
    var [flagRequestOptionalSlot]
    grammar [CreditAndTransfer.grxml]
    block
      Save global variables into local variables
      assign [lSlotCreditAccount] <-  gSlotCreditAccount
      assign [lSlotTransferAmount] <-  gSlotTransferAmount        3
      assign [flagRequestOptionalSlot] <-  true
    block
      if [lSlotCreditAccount!='undefined']
        If the compulsory slot has not been filled in previously
        try to fill them in now                                   4
        goto [fieldFillInCreditAndTransferSlots]
      else
        goto [bNextState]  //go to the next state
    initial [fieldFillInCreditAndTransferSlots]                   5
      prompt  ->  Please specify the credit account
    field [fCompulsorySlot]
      prompt  ->  Please say the alias of the credit account you want to use for the transfer   6
    block
      if [lSlotTransferAmount=='undefined']                       7
        assign [flagRequestOptionalSlot] <- false
    field [fOptionalSlot]
      slot  lSlotTransferAmount                                   8
      cond  flagRequestOptionalSlot=='true'
    block [bNextState]
      Save local variables into global variables
      assign [gSlotCreditAccount]  <-  lSlotCreditAccount         9
      assign [gSlotTransferAmount] <-  lSlotTransferAmount
      goto next="#DGetTransferAmount"
```

Fig. 7. Example of VoiceXML generated for a dialog with over-answering.

approach as a mixed initiative dialog (number 5, using the initial tag); if the slot is already filled in then we exit the form. In number 6 the system tries to fill in the slot by using a direct dialog if the mixed-initiative one fails. In number 7 we can see a block that analyzes whether the optional slot is empty or not. If it is, a Boolean variable is set to false in order to forbid this slot to be filled in at this state (number 8, use a conditional field based on the value of the Boolean variable). Finally, in number 9, the global variables are set using the local variables and the state jumps to the next state (i.e. *DGetTransferAmmount*).

Passing of arguments between actions is automated: Finally, another important action that the designer has to carry out in this assistant is the connection of input and output parameters for database access functions or dialogs with existing dialog variables. For instance, the designer needs to connect the variable that returns the available amount after carrying out the transaction with the local variable of the dialog that shows this information to the user. In order to automate this connection, the assistant detects the input/output variables required in each action and, by using a popup window, it offers the most suitable existing variables of a compatible type; if there is more than one variable to offer, the assistant sorts them according to the name similarity between variable and dialog. If there is no compatible variable already defined or the name proposed by the assistant is not desired, the system allows the creation of a new local or global variable. The assistant includes an edition window in case the designer makes a mistake or needs to modify the matching made in the previous steps.

Other accelerations and capabilities: In addition to the strategies mentioned above, the platform provides four basic dialog types that cover the usual possibilities in programming: based on a loop, on a sequence of actions (or sub-dialogs), a switch construct based on information input by the user (i.e. menu-based dialog), or a switch construct based on the value of a variable. Empty dialogs, with no action inside, can also be created (used to specify the call to a dialog that will be defined completely afterwards) so that a top-down design of dialogs can be made; in this case, the dialog type is selected whenever the designer tries to edit the empty dialog. Another possibility is dialog cloning, useful when the dialog to be defined is very similar to an existing one. On the other hand, given the large amount of actions that can be carried out in each state, the assistant allows the creation/edition local/global variables/constants, the creation of if-then-else structures, selection structures (switch-case), loops (for, while), assignments between simple and complex (objects) variables, and an assistant for mathematical operations and another one for strings.

We have also included some useful characteristics in the GUI, such as hotkeys for accessing the most common functionalities of the assistant, different colors for distinguishing each kind of dialog (i.e. already filled in, empty, DSay or DGet dialogs, etc.). In order to reduce the number of dialogs shown in the canvas the designer can also switch between a basic presentation of the dialog or a more detailed visual/textual flow (i.e., including internal information about variables, dialogs that are called from or call the current one, type, etc.). There is also a method to display the contents of complex or nested actions contained in a dialog using tooltips, which helps the designer in their interpretation, thus avoiding the need to open or edit them.

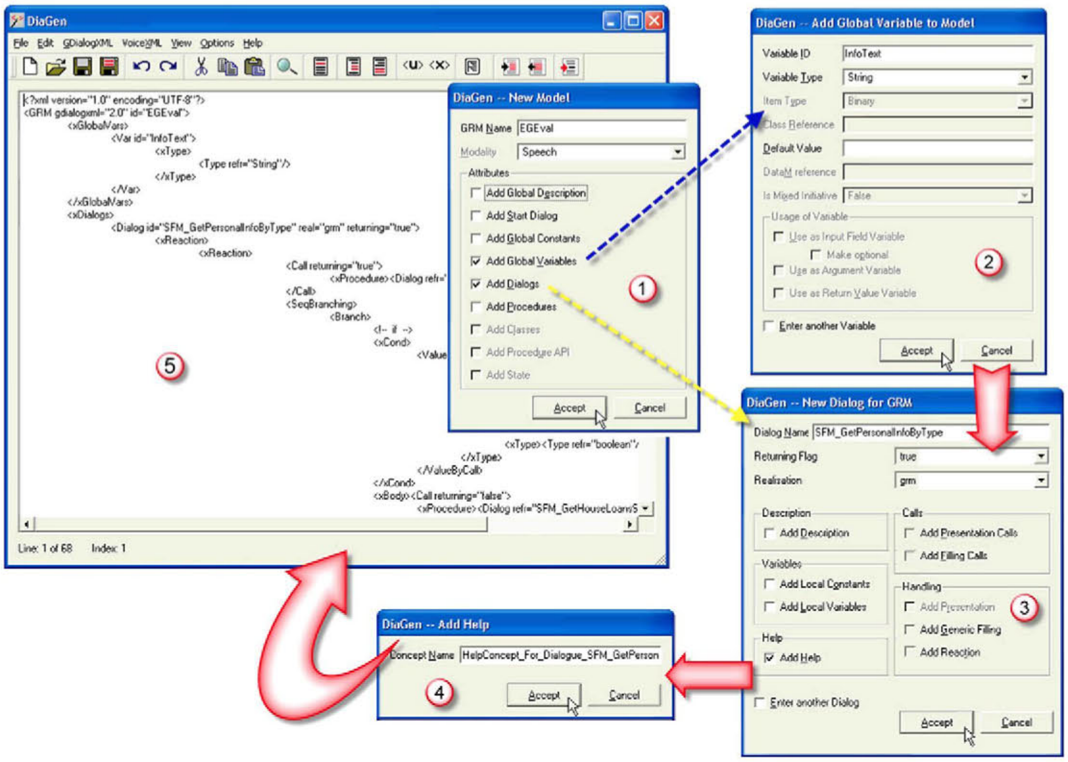
3.3. Accelerations to the definition of specific details for the speech modality

The next assistants in the platform allow the designer to specify detailed information for each language and modality. For instance, it is necessary to specify the dialog flow used to present, using the speech modality, the list of retrieved results after querying the database (e.g. the last movements in an account or the last credit card bill), the dialogs to confirm user answers, the speech recognition grammars, or prompts in different languages. In order to accelerate some of these steps we have defined the following strategies:

Presentation of object lists: These lists are the result of retrieving information from the database that need to be provided to the user in small groups since in the speech modality it is not convenient to present all of the items at the same time. In our platform we have incorporated a wizard window to specify the actions that have to be carried out by considering the different cases that can result depending on the number of items on the list (i.e. the list is empty, the list has one item, the list has more than one item and less than a maximum allowed, and the list has more items than the maximum allowed). For each case, a simple form-filling window allows the designer to specify the actions that have to be carried out. After filling out the four forms, all the actions and new dialogs needed are automatically generated. The assistant also proposes the most reasonable default values to fill in the forms.

Confirmation handling: One of the main problems in a spoken dialog system is how to cope with the speech recognition errors due to differences in the speaker voices, environments, channels, noises, etc. One possible solution would be to request the user to confirm the recognized information each time. However, this is not practical since it would extend the dialog time and it would decrease user satisfaction. One solution is to use the confidence level provided by the speech recognition. Thus, if the value is near to 1 then the recognition is very reliable and if it is near to 0 then the recognition should be rejected. It is usual to define four confirmation types: (a) no confirmation (i.e. the confidence is high, the recognition is accepted as valid), (b) implicit confirmation (i.e. the system provides the recognition result to the user as a fact in the next dialog turn, speeding up the dialog, e.g. "You want to travel to London. When do you intend to leave?" The user can say "no" or "cancel" at that point to go back in the dialog if London was not the intended destination), (c) explicit confirmation (i.e. the best option is to ask the user if the result is correct, e.g. "Do you mean London?" to confirm that London is the intended destination), and (d) reject. (i.e. The confidence level is extremely low, so the result is clearly unreliable. The system rejects it and asks it again.) Although VoiceXML specifies all the required variables and flow to access the confidence values and to handle each situation, in practice designers do not use all of them, i.e. only supporting no confirmation and rejection, since the definition of the flow for each case is a laborious task. In order to allow them all, in our platform, we have incorporated an assistant that analyses the flow and automatically provides the logic for the four confirmation types. The assistant also detects the states that cannot allow the four confirmation types. For instance, for requesting a yes or no answer, we only need the no confirmation or rejection level (i.e. we need a high confidence level only), or in the states that carry out the database access the system does not allow implicit confirmation since it cannot wait for a user confirmation in the following state since the access has to be carried out in the current one.

Setting of system prompts: For the speech modality, a high number of prompts for each kind of dialog (e.g. DGet, DSay, or



```
<?xml version="1.0" encoding="UTF-8"?>
<GRM gdialogxml="2.0" id="EGEval">
        <xGlobalVars>
                <Var id="InfoText">
                        <xType>
                                <Type refr="String"/>
                        </xType>
                </Var>
        </xGlobalVars>
        <xDialogs>
                <Dialog id="SFM_GetPersonalInfoByType" real="grm" returning="true">
                        <xReaction>
                                <xReaction>
                                                <Call returning="true">
                                                        <xProcedure><Dialog refr="
                                                </Call>
                                                <SeqBranching>
                                                        <Branch>
                                                                <!-- if -->
                                                                <xCond>
                                                                        <Value
                                                                                <xType><Type refr="boolean"/
                                                                        </xType>
                                                                </ValueByCall>
                                                                </xCond>
                                                                <xBody><Call returning="false">
                                                                        <xProcedure><Dialog refr="SFM_GetHouseLoanS
```

Line: 1 of 68   Index: 1

GRM Name: EGEval
Modality: Speech
☐ Add Global Description
☐ Add Start Dialog
☐ Add Global Constants
☑ Add Global Variables
☑ Add Dialogs
☐ Add Procedures
☐ Add Classes
☐ Add Procedure API
☐ Add State

Variable ID: InfoText
Variable Type: String
Item Type: Binary
Class Reference:
Default Value:
DataM reference:
Is Mixed Initiative: False
☐ Use as Input Field Variable
☐ Make optional
☐ Use as Argument Variable
☐ Use as Return Value Variable
☐ Enter another Variable

Dialog Name: SFM_GetPersonalInfoByType
Returning Flag: true
Realization: grm
☐ Add Description
☐ Add Presentation Calls
☐ Add Filling Calls
☐ Add Local Constants
☐ Add Local Variables
☐ Add Presentation
☐ Add Generic Filling
☐ Add Reaction
☑ Add Help
☐ Enter another Dialog

Concept Name: HelpConcept_For_Dialogue_SFM_GetPerson

Fig. 8. Process for creating a dialog in GDialogXML using Diagen.

for providing contextual help) have to be defined. The designer also needs to take into account the different user levels (i.e. novice, intermediate, advance), all possible recognition errors (no input, no match, service timeout, etc.), and the number of times that the assistant allows every error to occur (e.g. 1, 2, 3) before transferring the call to an operator or exiting. To speed up the process of typing all these prompts, the assistant offers three possibilities: (1) reuse prompts already available for the current application, (2) reuse prompts generated in previous applications and saved as libraries, or (3) reuse wording libraries saved from previous applications. The assistant also allows the designer to use SSML tags to control the voice (e.g. break duration, pitch, rate, volume), as well as the possibility of including audio files in order to allow hybrid prompts. Finally, the system allows the designer to select the arguments of the dialog to be inserted into the prompt.

3.4. Alternative system

In order to allow the manual creation and fine tuning edition of the different models and libraries generated by the assistants, the platform includes a built-in editor called Diagen (Hamerich, 2008). The main acceleration included is the possibility of creating any section of the specification with minimum effort, so instead of typing all the tags nodes and children in the XML code, the assistant uses a set of pop-up windows that are sequentially displayed in accordance with the information that the designer needs to specify.

Fig. 8 shows an example of the process for adding a dialog and a global variable into the dialog flow. According to this figure, in (number 1) the designer specifies that two elements are going to be added. After accepting, a new pop-up window is displayed (number 2) allowing the designer to specify the information for the global variable to be added. In order to add new variables, the designer only needs to check the corresponding checkbox (i.e. enter another variable) and a similar pop-up window will be displayed after accepting the current one. The next step (number 3) is the definition of the dialog state and the relevant information for this dialog. In the example, the designer selects the specification of the help concept defined by using the pop-up window marked as number 4. A similar procedure allows the incorporation of different elements such as variables, calls, help messages, reactions, etc. After finishing the process, the system automatically pastes the GDialogXML code into the editor (number 5).

4. Evaluation of the acceleration techniques

In order to implement the acceleration strategies described above successfully, two kinds of evaluations were carried out: a formative and a summative one. In order to carry out the former, since the platform is made up of different assistants and several programmers were working on them, each member of the programmer team was given the role of evaluating one or two assistants different to the one that they were working on. Thus, every time that one of the teams released a new version of the assistants, the other team responsible for evaluating it spent some time testing the GUI, the new accelerations, discovering bugs, and providing the respective feedback. As this process was continuously done throughout the platform development, as well as the periodical changes in the members of the teams, we could guarantee diversity of feedbacks and permanent improvements. As regards the second kind of evaluation, we carried out two: one at the end of the GEMINI project, and another one after incorporating the new accelerations based on the database content together with some improvements that we made following the advice from the first evaluation. Below we explain both, providing further details for the latter.

Right at the end of the GEMINI project, we carried out a summative evaluation with more than 40 developers in order to test (a) the friendliness of each assistant in the platform and the whole platform interface, (b) the complexity and time required to learn to use each assistant and the whole platform, (c) the level of functionality of each assistant, (d) the level of consistency, transparency, and intuitiveness of each assistant, and (e) the willingness of the evaluators to use the platform to develop dialog applications. 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 instance, the testers suggested a new GUI for the assistant used to define the states of the dialog in order to allow the quick inspection of the whole flow as well as the creation of the states and transitions, or to include a new wizard window to create complex classes in the data model quickly (for further details please refer to D'Haro et al. (2006) and D'Haro (2009)).

Taking into account these suggestions and after incorporating the accelerations based on the database content information in certain assistants, we decided to do a new summative evaluation by introducing some modifications into the survey in order to give more weight to the evaluation of the acceleration techniques this time instead of the whole platform or a final service since it was checked in the previous evaluation. The subjects for the new evaluation were undergraduate students and teachers from our university with experience in at least one programming language and a minimum knowledge of the technologies and processes required for designing dialog applications. The total number of participants was 9 who were classified, according to their knowledge and experience on developing spoken dialog services, in the following three levels: novice (4), intermediate (3), and expert (2). From this group, only three participants had some knowledge of the platform.

In order to provide the same information about the platform and the test all of the participants, the evaluation was carried out at the same time for all of them. Then, each evaluator was placed alone at a computer with all of the software required for running the platform and collecting the information. In addition, each evaluator received a short handbook with more information about the platform, assistants, accelerations and examples that they could consult at any time.

The evaluation was made in two sections of 4 h each. During the first session, the evaluators received a complete explanation of the whole platform, the goals of the evaluation, and the interfaces used to get the statistics. Finally, they also received instructions and evaluated the assistants used to specify the data model structure (DMA), database access functions (DCMA), and state flow model (SFMA). During the second session, the evaluators finished the evaluation of the SFMA and learnt how to use and evaluate the assistants to specify the actions to be carried out at each state (RMA). In turn, the evaluation of each assistant was divided into three main blocks: in the first one, the evaluators received instructions as to the capabilities and accelerations included in the corresponding assistant through examples of use. In the second block, the evaluators carried out an example task that was useful to answer their doubts about the assistant and to let them gain some practice. Finally, during the third block, the objective evaluation was carried out and eventually the evaluators were requested to fill in the subjective evaluation survey. The next sub-sections present full details of both evaluations.

4.1. Objective evaluation

For this evaluation we proposed that the participants perform predefined typical tasks when designing dialog applications using our assistants and then to compare them with an alternative assistant that included fewer accelerations. In order to compare the performance of the evaluators using any of the v, we defined a set of quantitative measures that were collected using a background process during the time that the evaluators carried out the proposed tasks.

Up to the best of our knowledge, there is no current standard method for evaluating and comparing this kind of development platforms and their acceleration strategies. In Jung et al. (2008) they describe DialogStudio, a development platform with some similarities to ours (see Section 2.2); in order to evaluate this platform and its accelerations, they proposed a set of quantitative measures when performing different tasks that the evaluators had to carry out and then to compare it when the same evaluators build the same tasks but using an open text editor chosen by each participant. During the evaluation, different objective metrics were measured such as mouse clicks, keystrokes, and elapsed time.

For evaluating our platform, we decided to do a similar assessment by introducing the following changes: (1) First, we included as new objective metric, the number of keystroke errors, i.e. the number of times the evaluator used any delete keys for correcting mistakes in the design. The goal of this metric is twofold: (a) to measure how much the strategies help to reduce human mistakes, and (b) to measure the difficulty of introducing information into the assistants or writing the GDialogXML code. (2) Since the development of a complete dialog application requires a lot of time, different areas of knowledge, and dealing with many details (e.g. database, runtime platform, debugger) we decided to propose the evaluators to perform only a reduced number of tasks specifically selected for testing the most important accelerations and the most complex assistants. (3) We decided to compare the quantitative measures by using our platform with those obtained when annotating the same tasks in the internal language format used by our platform by using our built-in editor (Diagen, see Section 3.4). In this case, the motivation for using the built-in editor instead of allowing the evaluators to use any XML editor of their liking was to make a fairest comparison between both applications. The advantages of using this editor are: (a) it includes several accelerations especially designed for writing the platform files, (b) it also reduces the time required to write the models in XML, and (c) it reduces the need to memorize the XML specification.

On the other hand, it is also important to mention that the main reasons for making the comparison between the assistants and Diagen, instead of comparing them with other development platforms, were that we could not find any commercial or academic platform comparable to our platform. For instance, most of these platforms do not take into account the database information nor do they include all of the accelerations that we wanted to evaluate. Most of the commercial platforms have an advanced graphical interface which we were not interested in evaluating (although it is well known that the appearance of the GUI has had a great influence on the evaluators). Finally, because these platforms are optimized for building speech-based applications only, in our platform, thanks to the separation of the assistants into the three layers, we have the possibility of incorporating new modalities such as Web or avatars in future releases.

4.1.1. Experimental setup

We have to take into account that since the evaluation that we did at the end of the GEMINI project not all the assistants in the platform were improved with the incorporation of the database content information. So, in the new evaluation we decided to include only the following: DMA (definition of data model), DCMA (definition of database access functions), SFMA (definition of states), and RMA (definition of dialog actions). As we mentioned before, during this evaluation we collected several objective measures that were obtained using a background process that could be started, paused, or stopped (see Fig. 9) to start testing the assistant or to pause/repeat the test. The objective of this system was to capture all mouse and keyboard events, but only when the mouse focus was on the platform assistants (i.e. we avoided counting as events, for instance, the process of checking the tutorial manuals during the evaluation). Finally, this system also started a parallel screen recorder application that recorded and saved all the interaction of the evaluators with the application during the evaluation. These videos were later reviewed in order to obtain a visual feedback of the process that the testers followed to complete the tasks and to find out the main problems the testers found. We also used them to find out whether the testers used the accelerations or not, together with the steps that took most of the elapsed time during the evaluation.

As we can see in the Fig. 9, the control interface allowed users to select the task to evaluate (1), to start or stop the test (2 and 4), to save the measured metrics when the evaluator was confident of having evaluated the step correctly (number 3), and to display the collected metrics at each task (5, 6, and 7).

4.1.2. Proposed tasks

In order to test the accelerations, we defined a set of typical tasks that covers those that have to be done at each assistant when creating a full application. Depending on the number of accelerations to be evaluated, each assistant could have one or more tasks. Below, we provide a detailed list of all the proposed tasks and accelerations involved.

For the process of creating the data model (DMA), we proposed that the evaluators test two different tasks or cases: (a) in the first one, they were requested to create a class model with two atomic attributes. Both attributes were related to the database, and (b) to create a class structure, including two atomic attributes (both related to the database and with language dependency) and one complex attribute (a list of embedded objects).

For the DCMA assistant, we proposed that the evaluators do one single task. It consisted of creating a function with two input

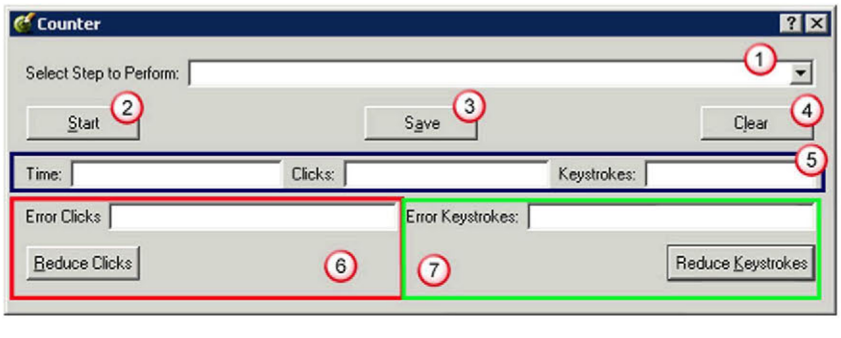


Fig. 9. Interface to control and display the metrics during the evaluation.

arguments and one output argument, where all the parameters were related to the data model. The participants were also able to test the functionality of creating and testing SQL statements.

For the assistant used to define the dialog states (SFMA), we proposed that the testers carry out three steps: (a) creation of a state with one slot related to the database. Here the testers had the option of creating the state by using the proposal of automatic states with slots or using an empty state template and then defining the slot and its relationship to the database. (b) Creation of a state with two slots, where both slots had to be set up as mixed initiative, and then to define a transition to another state. This step allowed the testers to check the automatic unification of slots to be requested by using mixed-initiative dialogs and the automatic creation of an undefined state when it is referred to as a transition state (i.e., top-down design); and (c) the creation of a connection between two states. This step allowed the testers to check some of the functionalities included in the graphical user interface (see Section 3.2.1).

For the complete definition of the actions to be carried out in each state (RMA), we proposed three tasks: (a) the creation of a typical menu-based dialog where the system requests the users to select between three different kinds of information, and according to the user selection to jump to the corresponding state. In this case, we evaluated the proposals of actions and different kinds of dialogs allowed for the platform, (b) the creation of a dialog with over-answering and an IF-Then-Else condition. Here the following accelerations were used: the dialog proposals window, the automatic matching of arguments between actions, the procedure for including compulsory and optional slots, and the possibility of defining different programming structures. And (c) the creation of the same running example in this paper, i.e. a mixed-initiative dialog for carrying out a transaction between two accounts and the amount to be transferred, and saving this amount in a global variable that was required to be created. This task allowed the accelerations provided by the assistant to be tested for defining mixed-initiative dialogs, for matching variables, the action proposals window, and the assistant for defining local/global variables.

4.1.3. Results

Before describing the results of the objective evaluation we want to mention two important factors that should be kept in mind when making the comparison between using the assistants and Diagen, the built-in editor. First, when we reviewed all the recorded videos for Diagen testing, we observed that all of the evaluators were getting used to the editor as the GUI and the procedure to use it were basically the same for all the tasks, therefore the evaluators were working faster with it, but only in the two first assistants as their model complexity is low. On the other hand, each time they were requested to test the platform assistants they had to learn to use a new interface and new accelerations. Finally, we also found out that the evaluators were spending a lot of time reviewing the final state created using the platform in order to check whether it corresponded to the one specified by the evaluation. Although this behavior is normal, we observed that for Diagen they did not spend so much time in that revision, probably because a lot of XML text was already generated.

As regards the objective evaluation, Fig. 10 provides an overview of the average improvement considering all the tasks per assistant when comparing the objective metrics obtained for Diagen and the corresponding assistant. In the figure, a positive value means that the assistants of the platform performed better than Diagen, and a negative value means that Diagen outperformed the corresponding assistant. As we can see, the accelerations proposed produced an average improvement of 65.5% for defining the data model structure (DMA), 16.6% for defining the prototypes of the database access functions (DCMA), 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 (RMA). Thus, we obtained an overall average improvement of 52.3% that corresponds to a 56.5% improvement in the elapsed time, 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 described in this paper. We can also see that the improvements were greater in the assistants where the more complex structures and actions are required; thus, we accelerated the design and guide the designer in the steps where it is more necessary.

4.2. Subjective evaluation and results

In order to rate the acceptability of the evaluated assistants and the proposed accelerations we provided the evaluators a survey form that included general questions about each assistant and specific questions about each of the evaluated strategies. In both kind of questions we used a 10-point scale (1 = minimum, 10 = maximum). The total number of general questions was 20 (including Diagen, 4 questions per assistant) and 10 for the specific questions about the strategies. The general questions were:

(1) How quickly did you learn to use the assistant?
(2) Is the assistant easy and intuitive to use? Do you know what to do at each step?
(3) Is the functionality sufficient?
(4) How do you rate the appearance of the assistant (consistent, transparent, and intuitive)?

Table 1 shows the average scores for all the general questions about the different assistants evaluated. We can see that, in general, all the assistants were considered as easy to use (question 2) and the appearance was marked with an average value of 8.5

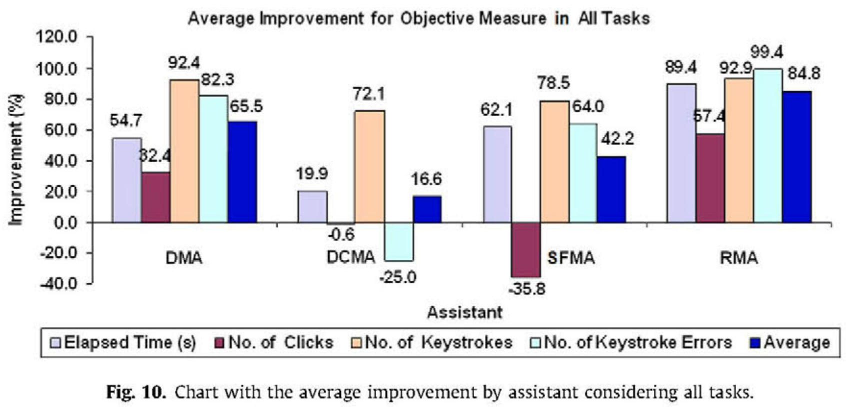


Fig. 10. Chart with the average improvement by assistant considering all tasks.

Table 1
Results of the subjective evaluation for the general questions.

Assistants	DMA		DCMA		SFMA		RMA		Diagen	
	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
Question 1	8.1	0.99	8.2	1.03	8.8	0.79	8.0	0.94	5.6	1.50
Question 2	8.6	0.83	8.2	1.23	9.0	0.82	8.8	0.79	4.6	1.42
Question 3	8.3	0.94	8.4	0.96	9.2	0.79	9.0	0.94	4.0	1.41
Question 4	8.2	1.03	8.1	0.99	9.0	0.82	8.8	0.63	4.0	1.41
Av. All	8.3	0.63	8.3	0.85	9.0	0.69	8.6	0.63	4.5	1.1

(question 4). It is important to mention that although Diagen was easy to use for the first steps, it got bad qualifications, as shown in the last column, probably because the generation of the final models was too cumbersome in comparison to our assistants.

Regarding the specific questions on the strategies, we included the following: (a) facility to create the data model structure using information from the DB, (b) facility to define dialogs with Mixed-Initiative, (c) comparison of speed development between using Diagen or the assistants of the platform, among others. Table 2 shows the results for the questions on the accelerations strategies for each assistant and evaluator levels. As we can see in the table, all the accelerations were positively assessed with an average value of 9.0, with the maximum scores in the following accelerations: automatic generation of action proposals (Section 3.2.2) and the ease in defining the state flow model (Section 3.2.1). These results are consistent, and better in all the cases, with the evaluation made at the end of the GEMINI project. The improvements are mainly due to the incorporation of the new accelerations, together with the correction of some bugs and the simplification of some procedures.

Finally, the survey included a section for comments and suggestions of the subjects with respect to the accelerations of each assistant. A summary of the main comments follows.

- Definition of data model: The evaluators considered the creation of classes using the database information as extremely useful. However, there are some aspects of the GUI that can be improved such as the possibility of copying and pasting whole classes, and the direct edition of the class attributes directly on the boxes in the workspace instead of using buttons/text-fields on the toolbar.
- Definition of database access functions: This assistant was considered easy to use since the menus were clear and simple. The most valuable accelerations were the automatic creation of SQL statements and the automatic proposal of data model classes/attributes and tables/fields when defining the arguments. One evaluator suggested improving the process of defining the function parameters using graphic objects instead of the current text-based interface.
- Definition of the dialog flow: The most valuable accelerations were the proposal of states from the classes and database functions, as well as the automatic unification of slots as mixed-initiative. Here, the testers requested the implementation of a easy mechanism for connecting states based on using anchor points and drawing the connection line with the mouse.
- Definition of dialog actions: In this assistant, the best accelerations were the window with the proposal of actions, the automatic creation of variables when passing arguments between actions, and the mechanism for creating conditional actions. The only suggestions were related to the graphical interface, for instance changing the position of some buttons in the assistants to make them easy to access, avoiding the use of modal windows that prevent the designer from carrying out other actions, since the mouse or keyboard focus cannot be redirected to other windows, etc.
- Diagen: In this assistant, two main problems were detected: 1.) the information collected through the different pop-up windows can be lost in case of problems. Sometimes it is confusing for the designer to follow the process of completing many nested items, 2) the templates were not enough for most designers. Several other features such as auto-completion and color tags were also required.

5. Future work

Considering the results obtained during the objective evaluation and taking into account the feedback provided by the evaluators in the subjective evaluation, as well as some ideas that we have been working on beforehand, we consider the following as relevant improvements in the platform.

Creation of the data model structure: To implement a mechanism for retrieving the relationships between tables in the database in order to propose complex classes and attributes automatically by selecting the most probable tables and fields to be used as attributes by default; to tag automatically (based on the heuristics) when a given attribute in a class will be used in the following assistants to provide or obtain information to/from the user thus improving the creation of the state/dialog proposals in the following assistants.

Creation of the database access functions: To extend the capabilities of generating SQL statements by improving the debugger wizard and including the possibility of offering an N-best list of SQL statements instead of only one proposal.

Table 2
Results of the subjective evaluation for questions regarding the accelerations.

Assistants	Novice		Intermediate		Experts		Av. All	
	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
DMA: Class proposals	8.5	2.38	9.33	1.15	9.0	0.0	8.89	1.62
DCMA: SQL wizard	8.75	0.96	9.33	0.58	9.00	0.00	9.0	0.71
SFMA: State proposals	9.75	0.50	8.67	1.53	9.50	0.71	9.33	1.00
SFMA + RMA: Mixed initiative	9.25	0.96	8.00	1.00	10.00	0.00	9.00	1.12
RMA: Over-answering	9.25	0.96	8.33	0.58	9.50	0.71	9.00	0.87
RMA: Pass args between actions	9.25	0.96	8.00	2.00	9.50	0.71	8.89	1.36
RMA: Action proposals	9.50	0.58	10.00	0.00	10.00	0.00	9.78	0.44
Diagen: Speed to create models	9.50	1.00	10.00	0.00	10.00	0.00	9.78	0.67

Definition of specific actions at each state: To reduce the number of automatic generated dialogs in order to simplify the main interface. The results of both evaluations showed that most of the proposed dialogs in this window were not used at all. Here, increasing the use of the heuristic information could help to identify which fields would be used for requesting information from the user or to provide information to them.

Definition of user dependent behavior: The platform currently allows the creation of different user profiles that allow the designer to modify the system behavior in three different areas: (a) the thresholds for confirming the speech recognition results in each state (e.g. novice users require higher thresholds since they usually provide longer answers than expert users), (b) the limits for different error handling (e.g. the number of times the system tries to obtain an answer before directing the call to an operator, or the number of times the system tries to access the database before reporting that an error has occurred), (c) different configuration parameters (e.g. number of items to show in group when handling a list of retrieved results, allowing barge-in or not, time in seconds allowed to record messages). In order to improve these capabilities, we propose the incorporation of new user profiles that allow the distinction between young and old people (following the indications suggested by Zajicek (2004) and Wolters et al. (2009)) or between men and women. On the other hand, the process of detecting this information could be based on using runtime modules or based on the information provided from the database after the user goes through a previous verification process.

6. Conclusions

In this paper, we have described several accelerations strategies included in a complete development platform to speed up the design of spoken-dialog systems. The proposed accelerations are, in most cases, innovative in relation to the current ones offered by any commercial or research platform. The proposed accelerations are based on using three different sources of information: (1) cumulative information shared between assistants throughout all the steps in the design, (2) heuristic information extracted from the contents of the backend database, and (3) using an object-oriented representation of the data model structure and relationships between this structure and the backend database. With all this information, the platform assistants generate different kinds of proposals that simplify the process of creating and completing the dialog flow (e.g. automatic states and dialog actions, the unification of slots to be requested using mixed-initiative dialogs, the semi-automatic creation of SQL statements), help designers to create or debug models (e.g. grammars, prompts, SQL functions) required by the runtime system, or to reduce the information displayed at each assistant. Thanks to the internal XML syntax of the platform the creation of the running scripts in VoiceXML is also accelerated, and at the same time it allows new modalities to be included in the future.

The results obtained both in a subjective and objective evaluation confirms the usability of the proposed accelerations. For the objective evaluation, different metrics were proposed and compared with those obtained when using a less-accelerated assistant for performing typical tasks when designing a dialog application. Thanks to the accelerations, the design time the evaluators spent on performing the proposed tasks was reduced by more than 56%, the number of keystrokes by 84%, and the keystroke errors by 55%. As regards the subjective evaluation all the accelerations were marked over 8.0 and the whole platform was rated with an average score of 8.0. According to this evaluation, the most appreciated accelerations were the automatic generation of action proposals for each dialog and the ease in designing the state flow model. Although not described in this paper, the platform has been used for successfully creating complex applications such as a banking application for a commercial product by a Greek bank with very good results, as well as for a second application, called CitizenCare, that offers basic voice information retrieval system functionality in the context of public authorities available in both German and English (see D'Haro et al., 2006).

Acknowledgements

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

References

Balentine, B., & Morgan, D. P. (2001). *How to build a speech recognition application. A style guide for telephony dialogs* (2nd ed.). Enterprise Integration Group (414 pages). ISBN-13: 978-0967127828.

Chung, G. 2004. Developing a flexible spoken dialog system using simulation. In *42nd Annual meeting on association for computational linguistics (ACL)* (pp. 63–70).

D'Haro, L. F. (2009). *Speed up strategies for the creation of multimodal and multilingual dialog systems*. PhD thesis, Universidad Politécnica de Madrid.

D'Haro, L. F., Cordoba, R., Ferreiros, J., Hamerich, S. W., Schless, V., Kladis, B., et al. (2006). An advanced platform to speed up the design of multilingual dialog applications for multiple modalities. *Speech Communication, 48*(8), 863–887.

D'Haro, L. F., Cordoba, R., Lucas, J. M., Barra-Chicote, R., & San-Segundo, R. (2009). Speeding up the design of dialog applications by using database contents and structure information. In *SigDial, London, UK* (pp. 160–169).

Feng, J., Bangalore, S., & Rahim, M. (2003). WEBTALK: Mining websites for automatically building dialog systems. *Workshop on automatic speech recognition and understanding (ASRU '03)* (pp. 168–173).

Web page of the GEMINI Project, October 2010. <http://www-gth.die.upm.es/projects/gemini/>.

Gorin, A. L., Riccardi, G., & Wright, J. H. (1997). How may I help you? *Speech Communication, 23*(1–2), 113–127.

Hamerich, S. W. (2008). From GEMINI to DiaGen: improving development of speech dialogs for embedded systems. In *9th SIGdial workshop on discourse and dialog* (pp. 92–95).

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

López-Cózar, R., & Araki, M. (2005). *Spoken, multilingual and multimodal dialog systems: Development and assessment* (262 pp.). John Wiley & Sons. ISBN: 0-470-02155-1.

McTear, M. (1998). Modelling spoken dialogs with state transition diagrams: Experiences with the CSLU toolkit. In *International conference on spoken language processing (ICSLP)* (pp. 1223–1226).

McTear, M. (2004). *Spoken dialog technology: Towards the conversational user interface* (432 pp.). Springer. ISBN: 1-85233-672-2.

Paternò, F., & Sisti, C. (2010). Deriving vocal interfaces from logical description in multi-device authoring environments. In B. Benatallah et al. (Eds.). *ICWE. LNCS* (Vol. 6189, pp. 204–217). Berlin, Heidelberg: Springer-Verlag.

Polifroni, J., Walker, M. (2006). Learning database content for spoken dialog system design. In *International conference on language resources and evaluation (LREC)* (pp. 143–148).

Polifroni, J., Chung, G., & Seneff, S. (2003). Towards the automatic generation of mixed-initiative dialog systems from web content. In *European conference on speech communication and technology (Eurospeech)* (pp. 193–196).

Schubert, V., & Hamerich, S. W. (2005). The dialog application metalanguage GDialogXML. In *European conference on speech communication and technology (Eurospeech)* (pp. 789–792).

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

Wolters, M., Georgila, K., Moore, J. D., Logie, R. H., MacPherson, S. E., & Watson, M. (2009). Reducing working memory load in spoken dialog systems. *Interacting with Computers, 21*(4), 276–287.

Zajicek, M. (2004). Successful and available: Interface design exemplars for older users. *Interacting with Computers, 16*(3), 411–430. Universal Usability Revisited.