The Dialog Application Metalanguage GDialogXML

Volker Schubert, Stefan W. Hamerich

TEMIC Speech Dialog Systems
Dialog Research
Ulm – Germany
{volker.schubert|stefan.hamerich}@temic-sds.com

10.21437/Interspeech.2005-364

Abstract

In this paper we present GDialogXML, which is a new powerful modeling language for multi-modal dialog applications. In contrast to other dialog description languages (e.g. VoiceXML) GDialogXML focuses on the complete development process, covering data models, dialog flow, interaction models, and many more. It even allows for the modality-independent modeling of dialog applications. Beside some innovative features like the modality refinement procedure, a main achievement of GDialogXML is its integrating architetcure.

1. Introduction

In the EC funded research project GEMINI (see [1]), an integrated development environment (IDE) for the semi-automatic generation of dialog applications was developed. A powerful modeling language was needed to represent all aspects of real-world dialog applications. Since existing dialog modeling languages like VoiceXML [2] did not prove powerful enough, a new language was developed, first presented in [3]. This language was named GDialogXML due to its origin in GEMINI. However, it can be used completely independent of the GEMINI IDE, and it can serve as a means to specify and iteratively develop large-scale dialog applications.

In the GEMINI project, GDialogXML has successfully been used for describing two multimodal and multilingual complex dialog applications. The speech version of the banking application is available since 2004 and meanwhile has been used by several thousand customers of the Egnatia Bank in Greece. For further details about the applications, refer to [4].

This paper presents the main features of GDialogXML (independently of the GEMINI context, considered in [3]), which are:

- The language allows to model all aspects of a dialog application. Besides dialog flow modeling, GDialogXML offers concepts for e.g. data modeling, I/O encapsulation, backend modeling, user modeling, and many more.
- The dialog application can be modeled in a modality-independent and language-independent way.
- The dialog flow can be developed in different layers of abstraction: first the general flow in a very abstract form, given by state transitions; second the more detailed flow, now including condition based action and branching (but still independent of the modality) and third the modality specific stamping of user interaction behavior.

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

- The language is based on virtual (expandable) states, which is a generalization of state based descriptions.
- It includes a fully featured object-oriented programming language, redundantizing the use of scripting.
- The interaction modeling includes features like mixed-initiative and overanswering, and this even in a modality independent way.
- Arbitrary services can be modeled and plugged in (like databases, Web-services, etc.)
- It is extremely modular and can easily be extended to include additional modeling concepts.
- The language is specified by a metamodel in a syntax-independent way. This metamodel is accompanied by a very easy XML meta schema. Metamodel and meta schema together determine the XML syntax of the language without the need for an explicit XML schema. This makes it also very easy to parse the content of a given XML document, since there are no syntax issues.

These points are now discussed in more detail in the following sections.

2. Dialog Modeling

In a dialog model the action flow of the dialog is represented. Many different approaches exist, from simple state based solutions to highly flexible knowledge based solutions. The GDialogXML solution is based on states, in principle, but with a lot of improvements for flexibility:

- It uses expandable hierarchical states. This reflects the desire for abstraction. In one abstraction layer a certain state may be atomic, whereas in another layer this state has to be expanded (implemented) into further sub states. A typical need for using expandable states is the abstraction from modality.
- The states are bundled into modules. In the dialog model these modules call each other in a returning fashion, i.e. as sub-modules, in contrast to an overall transition graph. This encapsulation of functionality is also a very important decoupling principle since it allows the setup of libraries with reusable modules. Note that the usage of sub-modules does not contradict the state-transition idea; it only adds an organizing principle to a flat graph. This can be seen as our realization of Reusable Dialog Components [5].
- Though GDialogXML is state based, high-level dialog managers can easily be plugged in via the integration of services. They can take over high-level decisions, where the more low-level integration logic is contained in the GDialogXML model itself.

Modeling the dialog flow in GDialogXML is possible in different ways. The straightforward way is to model the dialog in a single step, like it is done when developing dialog applications in VoiceXML (see e.g. [6]). The other possibility is to take advantage of the layer principle of GDialogXML.

When using the layer mechanism, development starts with a very simple state flow model, which is enriched by conditions, retrievals and variables to form the generic dialog model. Finally, after the I/O model has been added, a typical dialog model (as known from e.g. VoiceXML) is fully specified. Refer to Figure 1 for a schematical overview of this process. The three layers are explained in more detail in the next sections.

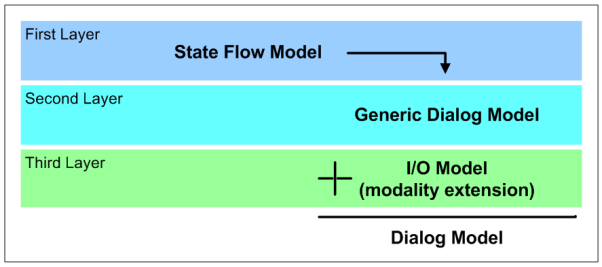

Figure 1: Development process for GDialogXML dialog models.

2.1. First Layer

The first layer is intended to specify the very basic dialog flow in the initial phase of the dialog design process. Therefore only the main states, their slots, and their successors are included into the GDialogXML model of this layer, which is named State Flow Model (SFM). The SFM does not contain any I/O logic or conditions, and certainly is modality and language independent. Since no conditions are allowed in here, multiple successors are possible for one state.

The advantage of beginning with this layer is its abstract view. So here only the most general decisions – which information is needed when from the user – are taken. This can even be done by people not being an expert in detailed dialog design. To test different dialog strategies, one SFM can be reused for several applications. This abstract view of the flow is accompanied with a static view of the data that have to be processed (see section 3.1 below).

2.2. Second Layer

In the second layer the SFM is enriched with algorithmic details, including the use of variables, conditioned behavior, procedure calls and data retrieval methods (e.g. database queries). The idea is that the modality independent dialog flow can be designed here, without the need to specify modality specific user interaction behavior. The resulting model is called Generic Dialog Model (GDM).

The GDialogXML interpretation of ”modality” is quite flexible. It perfectly makes sense to distinguish between the ”system driven speech” modality and the ”mixed initiative speech” modality, since their filling strategy is completely different. Insofar, ”mixed initiative speech” has much more similarity with page oriented modalities, like visual webpages, since there also a lot of input and output can happen in parallel (i.e. in a single state).

The GDM can be designed in a way that is independent of the way how information is gathered or presented. It is even possible to introduce the concept of optional slots, which can be used for the realization of overanswering capabilities (for further details refer to [7]). The reader should convince himself that it is far from being trivial to model overanswering in a modality-independent way, since the design has to be based on abstract filling states. The GDialogXML solution derived from the idea that a dialog module can recognize by itself whether enough information (e.g. coming from a previous overanswering step) is available.

Even if one thinks of a single modality only, this abstraction layer is helpful, since it also abstracts away from concrete wordings, and therefore from the target language. This is done by the use of input and output concepts, which are simple symbolic names (actually, interfaces). The implementation of the real input and output behavior is to be found in the I/O model, which is part of the third layer. So the GDM finally consists of the complete modality independent application logic.

2.3. Third Layer

In the third layer the realization of the input and output behavior is done, and therefore is language and modality dependent. This means, on the one hand, that the referenced I/O concepts from the GDM are defined and implemented here in the I/O model. But it also means that certain modality specific dialog modules can be added here, which allow the introduction of additional sub-states only needed for this modality (e.g. when an abstract multi-slot filling state has to be realized in sequential manner). In Figure 2 an example for the modality extension process is shown.

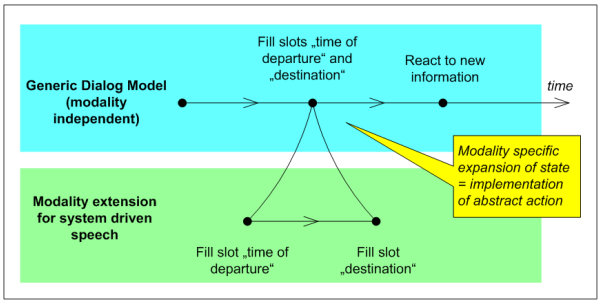

Figure 2: Example for modality extension

As a consequence, for a multi-lingual or multi-modal application several I/O models have to be written. It is important to stress that the GDialogXML layer approach does not use the ”greatest common divisor” of the modalities, which would give very weak results from the modality point of view, but tries to add the strengths of the modality in the I/O model. Insofar, the I/O model is more than a simple ”view” layer attached to the flow.

The design issues how a flow has to be divided into a modality-independent flow and an additional modality-specific flow is far from being trivial. But there are certain rules that guide the designer (for example, never to be too sequential in the abstract flow) as well as restrictions in GDialogXML.

2.4. Dialog Model

For getting the final dialog model, the GDM and an I/O model have to be merged. This is very easy, since GDialogXML expects orthogonal description of the layers, i.e. in the third layer

the second layer is not touched. This linkage process has been automated in the GEMINI IDE.

2.5. Error Handling

Another important feature of GDialogXML are its error handling capabilities. Some of these concepts are even modality independent, like the help concepts and service (database) timeout. Additional concepts for the speech modality cover timeouts, nomatch events, and different confidence scores depending on context and user level. For further details, see [8].

3. Data Modeling and Service Interfaces

3.1. Data Modeling

GDialogXML is object-oriented and allows the creation of data models. It has concepts for modeling classes with their attributes of certain types; additionally inheritance from base classes is supported. So complete data types can be introduced, which can be used for internal logic as well as for communication with arbitrary services.

One further advantage of the availability of a data model is, that if taken together with the abstract SFM it allows for the semi-automatic generation of the next layer (GDM), where the actual data flow is modeled.

3.2. Data Retrieval and Service Interfaces

GDialogXML offers the possibility of modeling APIs for service integration and data retrieval. By referencing classes or attributes from the data model, full object orientation applies. Using this mechanism arbitrary services like databases, web services and even high level dialog managers can be plugged in.

In the current version of GDialogXML predefined APIs for user identification, speaker verification, user level detection, and natural language generation components exist. Of course all of this APIs are modality and language independent.

The benefit of having these APIs is the possibility of using service functionalities directly from within the dialog without the need of knowing any special settings of the respective service. So details like HTTP address, file names or paths, database settings etc. are added later and are not needed during the dialog design process.

4. Object-oriented Programming Language

GDialogXML is also a fully featured object-oriented programming language, and offers most of the features which are available e.g. in C++. There are good reasons why an integrated programming language makes sense:

- No usage restrictions or communication/synchronization issues appear. One could argue that scripting would be a good alternative to an integrated language, but this often leads to problems in practical usage. Take, for example, VoiceXML and ECMAScript. It is true that they share the same data space, but it is not true that ECMA can be used as a control language of VoiceXML code: ECMA can be used surrounded by VoiceXML, but ECMA surrounding VoiceXML is not possible, so the ECMA control features (like loops, etc.) cannot be used, see [3].
- For each programming or scripting language a matching parser and interpreter must be available. GDialogXML, including its programming logic, is designed for easy parsing since it uses a generic syntax schema (refer to section 5). Therefore, comparing GDialogXML without programming and GDialogXML with programming, it makes only a small difference when implementing a GDialogXML interpreter. The costs for the programming capabilities are much lower than the integration costs of a parallel scripting language.

Moreover, if one does not like to code in GDialogXML directly, one still has the possibility of integrating code from other languages by calling the code as a plug-in service.

5. Metamodel

In this section we describe how the concepts of the GDialogXML language are put into form. This has to do with syntax, but is more than that.

GDialogXML is defined by a metamodel which can be expressed by a UML class diagram. It is based on the definition of concepts and their relations. In other words, everything that is covered by the language can be represented in objects and their associations, i.e. in a purely object-oriented way. This cannot be taken for granted. Most languages, from programming languages like C through modern XML-based languages like XSLT and VoiceXML, are syntax based and not concept based. This means, for example, that in a language there might be notional entities like *variables* and *function calls*, but the language specification is not based on formal representations of these concepts and their logical relation, but rather on the language grammar implicitly containing these concepts. This grammar typically is an adhoc syntax, oriented on natural languages.

In GDialogXML this is completely different since the specification is given by its metamodel. Moreover, to know how valid GDialogXML documents look like it is enough to specify the metamodel on the one hand and a generic mapping schema on the other hand, mapping objects and associations onto XML. No XML schema for the language (i.e. no grammar) is actually needed, since the schema is implicit by the generic meta schema. To give an impression of the meta schema, here are some of the mapping rules:

- An object is serialized as an XML-tree; the root tag is its class name.
- An attribute or association is serialized as an XML-tree, forming a sub-tree of the object-tree. In particular, attributes in our model are *not* represented by XML attributes (which are only string-valued), since this mechanism would be too restrictive, not allowing concepts (objects) as attribute values.
- There are two types or object associations: references and inclusions. References are realized via names and serialized in XML as <*classname* `refr="`*name*`">`.

The example in Figure 3 illustrates how the format of a concrete XML document (a *model*) is governed by the language metamodel and the generic meta schema. It shows how a *PromptCall* is realized, using a previously defined prompt and providing it with arguments. The first UML diagram shows an excerpt of the metamodel, relating the *PromptCall* class with the *Prompt* class and the *ValueExpression* class. The second diagram shows an exemplary model – presented in a graphical form – containing object instances like a previously defined

Prompt ”TellDate”. The third diagram shows the XML serialization of the model. Notice the straightforward way in which the objects and associations get represented in XML.

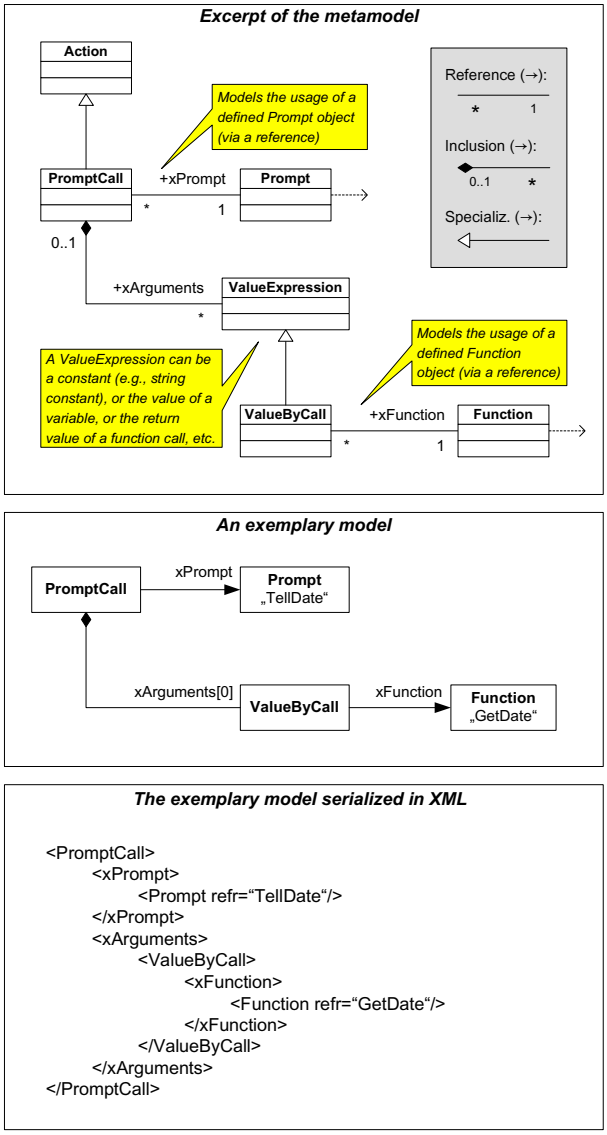

Figure 3: Exemplary models as instances of the metamodel

Advantages of this approach are:

- The language is specified in a concept-oriented grammar-independent way. The mapping onto a grammar is trivial by applying some elementary rules (the meta schema).
- The syntax is consistent per se, i.e. there cannot be any irregularities or pitfalls.
- It is trivial to write parsers. As soon as the metamodel is represented in a computational form (e.g. by C++ classes), parsing is automatic.
- Since a generic meta schema is applied, the metamodel can be changed and one still gets a consistent grammar. Therefore, the language can easily be extended by new concepts. It is highly modular and scalable.

6. Conclusion and Outlook

A new description language for dialog applications has been presented focussing on advanced and integrating features. The language covers all aspects of describing and modeling a dialog application and is intended to implement real world applications. Though being based on state-transitions, states are used in generalized way allowing for expandable states and the encapsulation into modules. The language is used in the GEMINI project where dialog applications are generated in a semi-automatic data-based way. The language supports the development of an application in different abstraction layers, allowing for top-down design. Even the simultaneous development for several modalities is supported.

The language has been submitted to a standardization organization, which currently is in the specification process of a dialog metalanguage. In workgroups the few proposals are being discussed and evaluated. There are good chances that concepts of GDialogXML will be included in upcoming standards in practical dialog design.

7. References

[1] S. W. Hamerich, R. de Córdoba, V. Schless, L. F. d’Haro, B. Kladis, V. Schubert, O. Kocsis, S. Igel, and J. M. Pardo, “The GEMINI Platform: Semi-Automatic Generation of Dialogue Applications,” in *Proceedings ICSLP*, Jeju, Korea, 2004, pp. 2629–2632.

[2] S. McGlashan, D. C. Burnett, J. Carter, P. Danielsen, J. Ferrans, A. Hunt, B. Lucas, B. Porter, K. Rehor, and S. Tryphonas, “Voice Extensible Markup Language (VoiceXML) Version 2.0,” W3C Recommendation, www.w3.org/TR/voicexml20, 2004.

[3] S. W. Hamerich, Y.-F. H. Wang, V. Schubert, V. Schless, and S. Igel, “XML-Based Dialogue Descriptions in the GEMINI Project,” in *Proceedings of the ’Berliner XML-Tage 2003’*, Berlin, Germany, 2003, pp. 404–412.

[4] S. W. Hamerich, V. Schubert, V. Schless, R. de Córdoba, J. M. Pardo, L. F. d’Haro, B. Kladis, O. Kocsis, and S. Igel, “Semi-Automatic Generation of Dialogue Applications in the GEMINI Project,” in *Proceedings of the SIGdial Workshop on Discourse and Dialogue*, Cambridge, USA, 2004, pp. 31–34.

[5] J. Huerta, D. Lubensky, D. Nahamoo, R. Pieraccini, T. V. Raman, and C. Wiecha, “Reusable Dialog Components – Mainstreaming speech-enabled Web applications,” http://www-128.ibm.com/developerworks/web/library/wa-dialogcomp/, 2004.

[6] C. Bennett, A. FontLlitjós, S. Shriver, A. Rudnicky, and A. W. Black, “Building VoiceXML-Based Applications,” in *Proceedings ICSLP*, Denver, USA, 2002, pp. 2245–2248.

[7] W. Wahlster, H. Marburger, A. Jameson, and S. Busemann, “Overanswering Yes-No Questions: Extended Responses in an Natural Language Interface to a Vision System,” in *Proceedings IJCAI*, Karlsruhe, Germany, 1983, pp. 643–646.

[8] Y.-F. H. Wang, S. W. Hamerich, and V. Schless, “Multi-Modal and Modality Specific Error Handling in the GEMINI Project,” in *Proceedings ISCA Workshop on ’Error Handling in Spoken Dialogue Systems’*, Chateau d’Oex, Switzerland, 2003, pp. 139–144.