Abstract—In the research area of spoken language dialogue systems there are many ways for modeling dialogues. The dialog model’s particular structure depends on the algorithm used to interpret it. In most cases a dialogue’s model is quite difficult to understand and to create. We present a novel technique for modeling dialogues based on ready to use open source tools in an easy and understandable way.

Making use of our approach a dialogue designer (unfamiliar with the internals of the dialogue manager) can simply develop even complex and adaptive dialogues. The dialogues are then ready to be interpreted by the dialogue management in order to integrate them seamlessly into the spoken language dialogue system.

Keywords—spoken language dialogue system; dialogue design; graphical modeling;

I. INTRODUCTION

In spoken language dialogue systems (SLDS) there are usually some key components which guarantee an efficient working system. On the system’s input side there is a recording device for transforming sound waves into a digital representation. A speech recognizer then transforms the digital representation into words and a linguistic analysis module filters out some semantic representation. This semantic representation of a user’s utterance is then fed into the dialogue management component, the heart of every dialogue system. An example of a simplified architecture is depicted in figure 1.

![Figure 1. Simplified architecture of an SLDS, according to [1].](image)

The dialogue management utilizes a dialogue model specifying how to handle the user’s input and the system’s output as well as the communication with adjacent software modules and applications.

After computing the next step, the dialogue manager sends an appropriate piece of output data to the language generation module providing some textual representation of the output. This information is then processed by the speech synthesis component in order to generate an adequate audio signal informing the user about the results of his/her input.

From this brief description of a typical dialogue system’s architecture the central dialogue model’s importance becomes apparent. The dialogue model – as a key component – incorporates domain specific knowledge in a way that allows for flexible generation of system utterances. However, there can be different types of dialogue models depending on the design of the dialogue manager.

In order to design a dialogue model there are usually two approaches:

- expert systems requiring a profound knowledge of the internal logic of the dialogue manager
- simpler systems requiring little knowledge of the internals of the dialogue manager

Expert systems on the one hand provide a more sophisticated reasoning on a user given input and may provide interfaces for reasoning on sensor data or domain knowledge. Simpler systems on the other hand normally provide a more generalized approach, maybe even standardized (cf. VoiceXML), which facilitates easier modeling. Ideally a system would feature both, sophisticated reasoning and a generalized approach for dialogue modeling, maybe even through relying on a graphical user interface.

In terms of VUI completeness (see [3]) on the one hand and adaptivity on the other hand it is logical that more prompts need to be specified in order to be adaptive to different users’ needs. Therefore, dialogue models tend to grow bigger and bigger for even short dialogues. This calls for focusing on special techniques to keep dialogue models manageable.

II. OVERVIEW

We start by giving a short survey on different dialogue management architectures in the field of SLDS and multimodal dialogue systems. This shall give an impression on the differences of several SLDS as well as their underlying dialogue models. Then we present our own approach for
dialog modelling stressing why it can be helpful in this domain. We elaborate why we chose this approach in a short discussion. Giving the reader a practical example we enable him to understand the advantages and the course of action of our approach. Finally we conclude our paper with subsuming the idea and providing some future directions.

III. RELATED WORK

As motivated above, dialogue modeling is an essential part in the design of SLDS. As there are many SLDSs there also have evolved different kinds of dialogue models. In principle the complexity of a dialogue model increases with the complexity of the purpose of the SLDS, the model refers to. According to [2], figure 2 lists different types of dialogue systems.

Due to the different application domains that the SLDSs have been designed to work in, there are also differences in the architectural design of those systems and especially in their ways to model dialogues. A well known SLDS is MALIN (cf. [4]) which stands for the classical type of SLDSs’ architecture. This kind of architecture is depicted in figure 3. There are also different agent oriented dialogue systems, most of which are based on the open agent architecture (cf. [5]). The WITAS dialogue system, for example, is a prominent SLDS that even features multi modality (cf. [6]). For the sake of being able to communicate with their mobile robots, [6] developed a dialogue model with an underlying tree-like structure, called dialogue move trees.

In the field of multi-party dialogue systems MITRE (cf. [7]) stands for an architecture, where the dialogue manager acts as a controller, which means that it is the central component in the architecture. As a consequence of the dialogue manager’s central role the model which is utilized by the dialogue manager needs to be complex enough to integrate all the components of MITRE.

An example of agenda-based dialogue modeling can be seen in [8]. Here the dialogue model combines three crucial concepts: handler, product and agenda. Handlers take care of the interaction on a small scale, the product denotes agreed information and the agenda provides structure to the conversation topics.

The Queen’s Communicator presents an approach to dialogue management which can be called object-oriented (cf. [9]). The dialogue model contains heuristic rules related
to the user as well as to the database and is divided into so called frames which give structure to the conversation.

For multimodal dialogue management there exist quite sophisticated models like the PAC-Amodeus model (cf. [10]). It makes use of the melting pot metaphor which is used to fuse different modalities. A melting pot is a two-dimensional data structure aligning input of different modalities on a time axis with structural parts of task objects on the other axis.

Another well-known dialogue system architecture is the TRAINS/TRIPS architecture (cf. [2]) which is based on different agents coordinating the behaviour of the dialogue system. It features multimodal input which is processed by an interpretation manager working with the input based on information from different other agents. The TRIPS’s architecture is depicted in figure 4.

Another sophisticated system which is especially designed for the use in intelligent environments is described in [12]. Here, the system’s underlying dialogue model is based on ontologies. One of the featured concepts is the process of on-demand generating VoiceXML in order to provide the voice user interface. Since the ontologies of the SLDS can be accessed and manipulated from all entities within the intelligent environment (IE) the capabilities and the configuration of the IE can influence the spoken dialogue during runtime. The basic idea is depicted in figure 5.

A more extensive overview over different kinds of SLDS can be found in [13].

Taking into account these given examples it is clear that SLDSs vary not only in their architectures but also in their ways of modeling dialogues. Thus it would be desirable to specify the dialogue model in a way that separates it from the internal logic of the dialogue manager. Thereby it would be possible to authorize somebody to design a dialogue without further knowledge of of the dialogue management logic, alleviating the process of dialogue design.

It would also be desirable to rely on a standardized format to enable validation, like e.g. XML, to provide a general approach for the dialogue model. Based on these points we now explain an approach to dialogue modeling separating the logic of the dialogue manager from the dialogue model, and using a standardized data format.

IV. THE CALIGRAPHI-APPROACH

For our generic approach to dialogue modeling we recommend XML as a data format in order to be flexible during editing and processing. The usual approach when aiming to define an XML-based data format would come along with the use of an XML Schema, a RELAX NG description or a DTD. It is best practice to utilize such a schema definition language in order to restrict the wide expressiveness of XML to a smaller subset that fits the needs of modeling a dialogue (cf. [14], [15]).

The shortcomings of this approach are that XML Schema as well as RELAX NG are quite complex. While DTD has a somewhat simpler syntax, it is not XML itself and its expressiveness is limited when it comes to data types and power of semantics.

With a given schema it is possible to validate the XML files. One can verify if the XML conforms to the restrictions made in the schema. Therefore, to fabricate a valid XML, the dialogue designer still needs to know the syntax as well as its semantic meaning, partly encoded in the schema, partly ascertained by common sense. So this is no trivial business.
at all.

For usability reasons it is desirable that a dialogue designer is given a simple interface, which provides the opportunity to model a dialogue that can later be interpreted by the dialogue manager – all without the the need to understand the underlying schema. In addition the interface shall also provide information about what is allowed as valid syntax for the dialogue model and what is not.

As XML editing can be tiresome and is often not as intuitive as using a graphical editing framework, we wanted to have an application which enables the designer to work and model the dialogue in a graphical way.

Facing these problems we present the CALIGRAFI-Approach. As a result of our search for a graphical editing tool we propose the use of the eclipse graphical modeling framework\(^1\). This proposal comes with the idea to use the eclipse modeling framework as a basis to generate a graphical editor. We will now describe the process in detail.

\begin{verbatim}
1 @gmf(fu="bar")
2 @namespace(uri="dialoguemodel", prefix="dialoguemodel")
3 package dialoguemodel;
4
5 @gmf.diagram(fu="bar")
6 class DialogueModel {
7   val DialogueGoal[*] topGoal;
8   val GoalLink[*] goalLinks;
9   val VariableLink[*] variableLinks;
10   val Variable[*] variables;
11 }
12
13 @gmf.node(label="name")
14 class DialogueGoal {
15   @ attr String name;
16   @ attr String description;
17   @ gmf.compartment(fu="bar")
18   val Utterance utterance;
19   @ gmf.compartment(fu="bar")
20   val Grammar grammar;
21   @ gmf.compartment(fu="bar")
22   val Effect effect;
23   @ gmf.compartment(fu="bar")
24   val Guard guard;
25   @ gmf.compartment(fu="bar")
26   val ApplicationLogic[*] applicationLogic;
27 }
28
29
30
31
32
\end{verbatim}

Figure 6. Structure of the dialogue model written in the emfat\(^2\) language. The model is based on dialogue goals which impose a certain hierarchy and flow on the dialogue. These dialogue goals can contain utterances or grammars indicating if a certain goal is used for input or output. Guards and effects act like temporal constraints and define the arrangement of the different dialogue goals within a dialogue flow.

The eclipse modeling framework (EMF) provides an opportunity to model a class structure using a subset of the UML 2 standard (cf. [16]). If necessary, a designer can make use of a diagram editor which allows to oversee the class structure. But one can also use a plugin called

\begin{verbatim}
1 http://www.eclipse.org/modeling/gmp/
2 http://www.eclipse.org/modeling/emft/?project=emfat
\end{verbatim}

...
structure of the dialogue model in order to have a graphical editor for the dialogue designer. The graphical editor is easy and intuitive to use and allows the dialogue designer to create a dialogue model without knowing about the internal matters of dialogue management. At the same time the dialogue model is restricted to a structure that is guaranteed to be interpretable by the dialogue manager.

In this way there is a clear separation of model and algorithm not only in the software design but also in the software design process. This allows to distribute the tasks for developing the dialogue system. There can be a dialogue model design team that is primarily concerned with the dialogue flow and there can be a dialogue management team that is concerned with the communication between dialogue manager and applications as well as the interpretation of the dialogue model.

V. PROOF OF CONCEPT

In order to provide a demonstration of the described method for dialogue modeling we outline the steps by an example for a multimodal dialogue model based on [17].

The exemplary dialogue model has a hierarchical structure. The dialogue flow is subdivided into dialogue goals which can contain dialogue goals themselves. Each goal can have an utterance or a grammar defining if the dialogue goal is used for user input or for system output. The goals also can contain guards which help to disambiguate the selection of the next goal if several goals can be selected for the next dialogue step. For the communication with the application the goal can contain some application logic that defines calls to application interfaces and has references on variables that may be filled by application calls or from user input.

The first step is to define this desired structure in an emfatic file. An example is illustrated in figure 6 also revealing the conceptual design of the exemplary dialogue...
For this emfatic file to be translated the emfatic plugin is needed.

Theoretically it would also be possible to directly create an ecore file (as can be seen in figure 7) in which you specify the structure of your classes accordingly but as stated above, emfatic has a more compact syntax. From the emfatic file you generate an ecore model describing the specified structure in an XML file format. With the help of this ecore XML file you can generate a diagram editor that lets you review your defined class structure in a more convenient and graphical way. An example diagram is shown in figure 8.

The structure of the dialog model has the form of a class diagram and specifies the relations between the different classes. These classes form the basis for an instance of a dialogue model which is then composed of instances of the classes specified here.

If you are satisfied with the setup of your class diagram it is now possible to make use of another eclipse plugin called eugenia\(^3\) to generate all the necessary files and classes that are needed for a graphical editing tool of the defined class structure. The eugenia plugin is not necessary but it helps a lot in simplifying the complex process to generate a graphical editor from the class diagram which represents our dialogue model. The complexity of the generation process makes it possible to customize the graphical editor to one’s own needs in many ways. But for our purposes it is sufficient to use the eugenia plugin without any special customizations. The editing tool that comes out in the end can be exported as an eclipse plugin and deployed to the dialogue designer.

![Figure 9](image)

Model data files required for the generation of the graphical editor.

We now briefly describe the generation process of the editing tool itself. First you generate a domain generator model from the ecore file. Using this model you can then generate a set of java classes that form the basis for the editor that we want to obtain. Then you generate a graphical definition model, a tooling definition model and a mapping model that combines both of the definition models. Using these, you generate a diagram editor generator model that needs to be resynchronized to your data. From this model it is then possible to generate the java code for the diagram plugin using the generated files. You can see a list of these model data files in figure 9.

From the generated java classes a deployable plugin can be exported. With this plugin it is possible to create instances of the classes defined in the ecore file in a graphical editor. The emerging diagram satisfies all constraints that are formulated in the ecore file. An example diagram file can be seen in figure 10. As defined, there are different connection types in the palette to model relations between different goals or goals and data.

At the end of the process the diagram file has an underlying data file which represents the dialogue model in XML. While the diagram file itself specifies the arrangement of the diagram for the model and is also in XML format, it is not needed for interpreting the dialogue model.

VI. CONCLUSION

In this paper we discussed a practical method to decouple the dialogue design from the internal logic of the dialogue manager. Following our advise, a dialogue designer can operate freely using a quickly and easily generated graphical editor, customized for his individual needs. This custom-tailored editor allows for the valid specification of the dialogue flow. Preceded learning to understand the internals of the dialogue management algorithms is not necessary any more. Because the graphical editor constrains the dialogue designer to build a valid dialogue model, the model is guaranteed to be interpretable by the dialogue management afterwards.

Although we provided a proof of concept for the feasibility of this method, it must still be evaluated in different dialogue domains and preferably with different dialogue management architectures.

ACKNOWLEDGMENT

This work is originated in the Transregional Collaborative Research Centre SFB/TRR 62 “Companion-Technology for Cognitive Technical Systems” funded by the German Research Foundation (DFG).

REFERENCES


\(^3\)http://www.eclipse.org/gmt/epsilon/doc/eugenia/
Figure 10. An example of the graphical editor resulting from the definition files. The dialogue model can be edited, satisfying the underlying structure defined in EMF. On the left Palette you see the parts of the dialogue model that you can create instances of in the editor. In the illustrated scenario you can create links (Connections) that mirror the relations between the instances (Objects) and arrange the whole structure on the canvas.


