Diamod - a Tool for Modeling Dialogue Applications

Anke Kölzer

Speech Understanding Systems (FT3/AV)
DaimlerChrysler AG – Research and Technology
P.O.Box 2360
D–89013 Ulm (Germany)
e–mail: anke.koelzer@daimlerchrysler.com

Abstract

Speech dialogue systems are currently becoming state–of–the–art for different kinds of applications, but they are still weak in the support of spontaneous speech and correct interpretation of what was said. One reason for the lack of good interactive dialogue systems is their complexity. To develop a system which is able to handle more than simple commands and phrases requires a lot of experience and time. To be able to accelerate and improve this process we are currently working on methods and tools which support this development. A new method called *Dialogue Statecharts* was defined for the graphical specification of complex dialogues. It is capable of representing parallel dialogue steps which is e.g. necessary for mixed–initiative dialogues. Our tool system named *Diamod* provides editors for different dialogue concepts, such as dialogue structures, grammars and parameters. The modeling is supported by graphical editors for *Dialogue Statecharts* and *Task Hierarchies*. *Diamod* is able to check for the completeness and consistency of dialogue models. One goal when developing *Diamod* was to provide specification models which are universal enough to be interpreted within different dialogue systems, i.e. different implementations of generic conversational systems.[1] With the help of a uniform representation of data a transformation between different models and different dialogue description languages (DDL) such as VoiceXML (AT&T et al., 2000) and some in–house–DDLs, such as Temic–DDL and Dialogue-Prolog, will be possible.

[1]By this we mean systems which are implemented application independently and are easily adapted to different applications.

1 Introduction

You find different dialogue system approaches on the market place and in research. One has been developed by the DaimlerChrysler research and is able to understand spontaneous speech speaker–independently and carry on dialogues on special topics. The structure and algorithms used are based on concepts developed in the Sundial project (Peckham, 1993). Most applications are made for telephony domains. Thus, up to now we gathered experience in applications like train time–table information, call centers for insurances and telematic systems for traffic data (see (Brietzmann et al., 1994), (Heisterkamp and McGlashan, 1996), (Ehrlich et al., 1997), (Boros et al., 1998) for further information).

We made the experience that developing new applications is very expensive concerning time and staff and needed tools to accelerate the process. Another goal was to make dialogue application modeling possible even for non–experts and help the expert to achieve consistent reusable applications. As there are different dialogue systems all over the world and many steps of application development are similar or even the same for all of them we decided to create tools which are system independent resp. easily adaptable to different needs and different dialogue systems. Our focus was on modeling dialogue structure for information–extracting resp. –processing systems of the slot–filling kind (Bilange, 1991).

In order to find out what functionality a tool must provide to be helpful we analyzed the way we construct dialogue applications and the different knowledge bases that are needed. Similar operation steps have to be executed for every new application in order to obtain a structured and maintainable dialogue. Typical tasks are:

- modeling of the dialogue structure: i.e. divide the dialogue into subdialogues to handle a special part of the interaction like the identification of a caller
- definition of the application parameters, i.e. the parameters necessary to give information to the caller or access a database like the name of the caller
- attachment of system prompts to dialogue situations like what the system has to say when asking the name of the caller
- definition of the appropriate vocabulary (pronunciation) and training of the language models
- definition of linguistic structures (lexicon, grammar, semantics)
- definition of the interface to the application system (e.g. an SQL–interface to a data base)

Diamod supports the specification of all of these dialogue application concepts (some are still under construction) and generates code which is interpreted by the target dialogue system.

2 Requirements

Dialogue systems which allow for spontaneous speech are much more difficult to handle than those which are only capable of processing single commands. *Diamod* has to support different ways of modeling dialogue structure and to transform one into another regarding special consistency requirements.

Thus the knowledge – i.e. the dialogue concepts – has to be represented in a universal way so that different aspects of dialogue can be modeled and code for different dialogue systems can be generated. A transformation from a spontaneous speech dialogue model to a rather restrictive command–and–control one and vice versa should be possible or a transformation from a state–based dialogue flow model to a rule–based one which is organized in tasks (as will be described in section 3.1). The approach must be extensible with little effort for specifying the additional knowledge bases, necessary for conversational systems, such as grammar models.

All the concepts necessary for dialogue flow modeling are to be integrated in the dialogue flow tool. Thus the dialogue flow tool must provide concepts such as application parameters, system prompts, state and task modeling. The state logic has to be described in a rather abstract way so that an automatic transformation for different dialogue systems is possible. Therefore it is not sufficient to use the widely employed state machines with which the specialties of spontaneous speech cannot be described adequately. Instead we use a design method based on Harel's statecharts (Harel, 1987) which are capable of describing concurrency and provide special event mechanisms and called it *Dialogue Statecharts*.

2.1 Properties of *Diamod*

Diamod is a CASE–tool (Computer Aided Software Engineering) specialized for language engineering which provides the concepts necessary for dialogue specification. To be able to develop new and modify old knowledge bases easily, the tool supports the language engineer with the following functionality:

Graphical editors for visual languages such as *Dialogue Statecharts* for the specification of structured dialogue data. The graphical interface shall enable the user to specify his models in a rather easy and intuitive way.

Data representation of all relevant information and the dependences between them.

Consistency checking by a formalism for defining constraints on the models and informing the user of violations of these constraints.

Code–generation (Prolog, VoiceXML, standardized speech API–code, ...) that can be interpreted by the currently preferred generic dialogue system.

Reuse support of formerly developed application models.

Two-phase modeling in order to be able to specify generic data independently of application specific data.

Easy adaptability to further dialogue systems and needs.

The principles of working with *Diamod* are described in the following sections.

3 The Tool System *Diamod*

Figure 1 shows the workflow in *Diamod*. The central unit is the tool system which provides methods for specifying knowledge, keeps the data and models, and does consistency checks. The user modifies the models with the help of a graphical user interface. A second possibility in future editions will be a textual interface for off–line specification where the user can model the dialogue with the help of a dialogue description language. The tool system represents data in a uniform graph representation and is able to generate code in different dialogue description languages such as Prolog[2] or VoiceXML dependent on the generic dialogue system currently in use. This code output (commonly spoken textual files) is read and interpreted by the corresponding generic dialogue system at runtime.

3.1 Dialogue flow models

With *Diamod* the application developer models what the system has to do in a given situation. As this must work for different generic dialogue systems, *Diamod* must also consider the generic features of the system (because they can be different for different dialogue systems). Therefore a two–phase approach is supported where in the first phase a dialogue expert (usually the developer of the generic dialogue system) models application independent data. In a second phase an application developer models application dependent data using the data which was modeled by the expert (Kölzer, 1999).

Another feature of *Diamod* is the support of different dialogue structure models. Our research system is a rule–based system (Ehrlich, 1999) which can be modeled in *Diamod* using tasks and task–hierarchy–diagrams. A rather state–based system can be modeled using the *Dialogue–Statecharts*–editor.

The following listing sums up the most important steps which have to be done by the application developer in order to specify the dialogue flow of a new application:

- definition of the components of the dialogue; e.g. a subdialogue for handling the identification of the caller and finding out why he calls, a subdialogue for reservation of a ticket, and one for callers who only want information.
- definition of the dialogue structure i.e. what the system has to handle first and what comes next. This is done by defining a start dialogue and the successors of each dialogue.
- attachment of application parameters to the dialogues; e.g. in the identification dialogue the system must request the caller's name and password.
- attachment of system prompts to the states where the system has to say something such as *confirm the parameter "Source"* in the reservation dialogue.

The following sections describe how *Diamod* supports the modeling for different approaches. With *Diamod* the user can model every concept by entering a name, a comment and information on the specific structure of the concept.

3.1.1 Task–Based Approach

The DaimlerChrysler research system is organized in tasks. Every task represents a subdialogue, e.g. a caller identification or a hotel reservation. The task structure is organized in a task hierarchy as shown in figure 2, which can be modeled with *Diamod* using the task–hierarchy–editor. At runtime the dialogue system can only activate the direct daughter– or mother–task of a currently active task in this hierarchy. This is used to make dialogue handling easier and more consistent. It is not necessary to model exactly the system states and their sequence as it often has to be done for other dialogue systems. The dialogue system uses a set of dialogue acts (Gazdar, 1981), (Heisterkamp et al., 1992) such as `confirm`, `request` and `inform` in order to distinguish between different dialogue situations. Every task has different application concepts attached to it. Among others these are:

Task (application) parameters: These are the concepts which model what values must be found out in order to reach the goals of the task, e.g. to be able to make a database access. This is usually what you

[2] A predefined sublanguage of Prolog is used to model applications for the DaimlerChrysler research system.

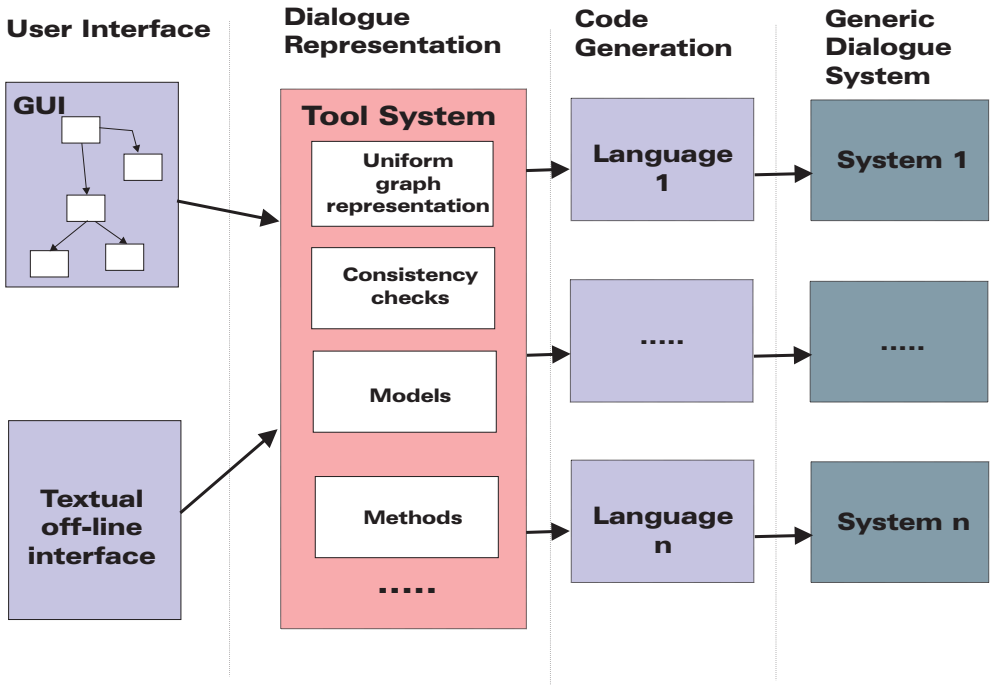

Figure 1: Specification of dialogues with *Diamod*. The central unit is the tool system which provides methods for the dialogue specification, keeps the data and is capable of checking the consistency. Data are modeled by the user on–line with the help of a graphical user interface or offline with textual dialogue description languages. When the specification is complete, the tool system generates the code necessary for the dialogue system in use.

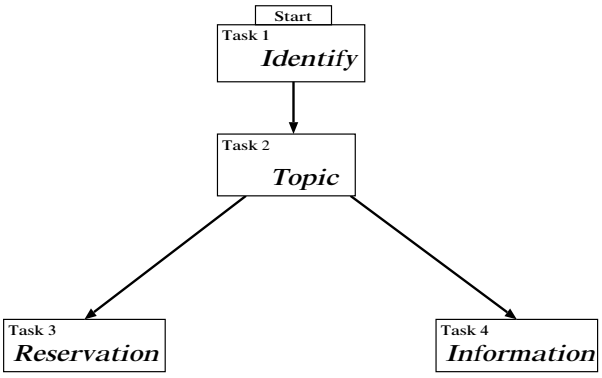

Figure 2: Task hierarchy diagram. Each rectangle models a task i.e. a subdialogue. The edges between the tasks show how tasks can follow each other.

have to request from the user such as a caller name or address. Task parameters can have attributes such as if they are optional or obligatory and if the system may repeat them or not (like passwords). *Diamod* supports the specification of such parameters with user definable types such as records and lists. The user can enter defaults and set the mentioned task parameter specific attributes using masks as shown in figure 3.

Databases and database parameters: If the dialogue system uses databases every task can declare a set of databases and database parameters that it wants to access. Task parameters can be mapped to database parameters. E.g. if the user speaks of *tomorrow*, this must be mapped to a concrete database date like *03.03.2000*. This is supported by *Diamod* with special masks.

System prompts: Given a dialogue act and an application parameter this concept models what the system has to say in that situation. With *Diamod* dialogue acts can be modeled and combined with task parameters in order to model the appropriate prompt. References to the values of task parameters can be used in a prompt such as in "'Your name is <*value name*>?"'. This

prompt is an example for confirming (dialogue act `confirm`) a task parameter `name`. *Diamod* is able to check if a used parameter value reference is feasible. This is the case when the appropriate task parameter was declared for this task. Prompts can be entered for different languages and *Diamod* can check if there is a prompt for every situation in every language. Figure 4 shows the prompt table mask of *Diamod*.

The prompt table can be calculated automatically. I.e. all combinations of dialogue acts and application parameter values are generated in order to gain all those system states, where a system prompt is needed. The result of such a generation is shown in figure 4. The user only has to fill in the prompts or delete table entries which are not needed.

Language models, grammars and lexicons: They can be declared for a task in order to switch between different ones and improve speech recognition this way. This is still under development (see section 4.1).

Actions: The application developer can model typed actions which should be performed on entering, resp. exiting the task. They can be related to task parameters using *Diamod*–masks which offer the user a list of accessable parameters and functions.

The transitions between tasks are realized using rules and conditions which are generic. This means that they are implemented in the dialogue system and do not have to be modeled by the application developer. Such a rule is for example that a task can only be exited successfully if all obligatory task parameters are known. In order to determine the next task to be activated the user's utterance is interpreted, like if he wants a hotel reservation. This together with preconditions for entering possible successor tasks is considered to control the dialogue flow.

When the developer has finished the specification he or she starts the code generation. The code produced can then be interpreted by the dialogue system. For our research system this is Prolog–code specifying the application knowledge bases.

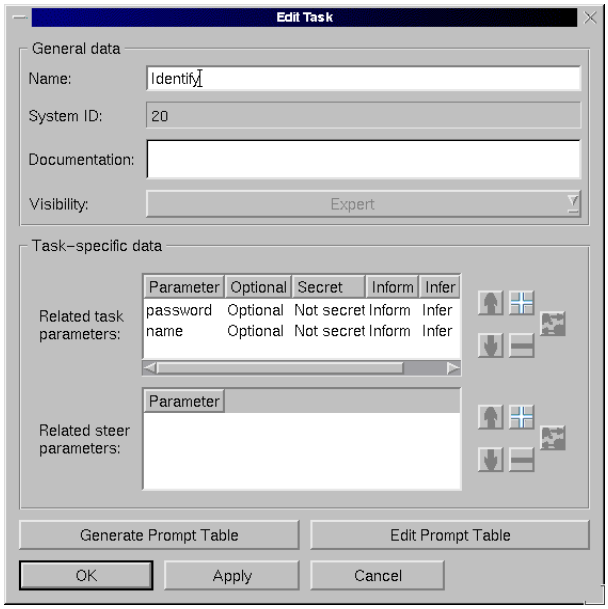

Figure 3: A mask for the description of a dialogue for the DaimlerChrysler research system. In some dialogue systems parameters can have attributes like if they are obligatory or optional. Therefore the masks have to be configured for the dialogue system. Clicking the button `Generate Prompt Table` will generate the possible prompts. Clicking the button `Edit Prompt Table` will open the mask shown in figure 4.

3.1.2 State–Based Approach

Many dialogue systems use a state based approach where dialogue flow is described in detail using state–transition–models combined with events (Failenschmid and Thornton, 1998), (Cole, 1999). Simple state–transition–models are adequate for very simple dialogue systems such as command–and–control systems.[3] As conversational systems have a high complexity of states, the expressiveness of state–transition–models is too small to be a good means for dialogue flow modeling. The number of states is usually too big to be handled by a human.

A good alternative for complex state modeling are statecharts as described by Harel (1987). They provide different means of abstraction such as concurrent states, state refinement, special event handling and action triggers.

[3]These are systems where a speaker may only say special commands like "radio louder" and not speak spontaneously.

Thus modeling of complex dialogue flow can be done in a rather intuitive way. Figure 6 is an example of modeling the task data shown in figure 2 in a state–based way. The dialogues are represented as complex states that are refined top–down to basic states where actions to be triggered are defined. Thus the state *DoDialogue* is represented as an XOR–State. This indicates that the system can only be in one of the states *Identify*, *PossibleTopics* or *End* at the same time. In simple cases a dialogue is represented by a basic state (*End*) which need not be refined any more. The *Reservation*–state must be refined into substates, one for each dialogue act. These are refined again as shown in figure 7. The developer defines entry and exit actions for basic states, i.e. actions to be triggered when entering and when leaving the state. The preconditions for changing the state taking an outgoing transition are described by events and conditions which have to occur. It is possible to describe actions and conditions common for several states or transitions by special means. E.g. any exit from the states *Reservation* and *Information* will lead to the state *End*.

There were already some state–based approaches for graphical dialogue representation. They were never used for complex systems such

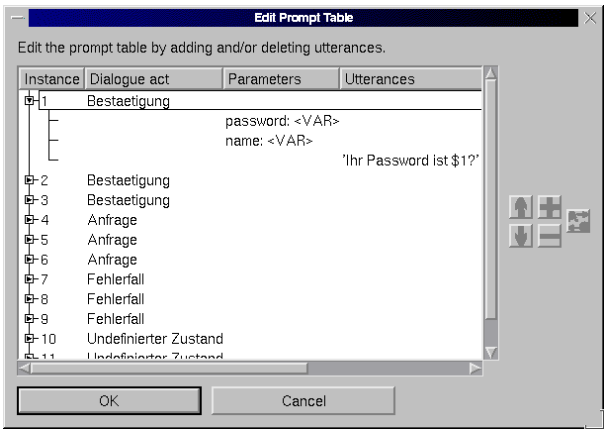

Figure 4: Defining the prompts for a dialogue. The application parameters that are talked about in this dialogue have to be declared for it before. For every dialogue act and every application parameter there must be a system prompt defined. The table here is calculated automatically by *Diamod* using the generic parameters (in this case the dialogue acts) defined by the expert and the application parameters defined here. The application developer only has to fill in the system prompts.

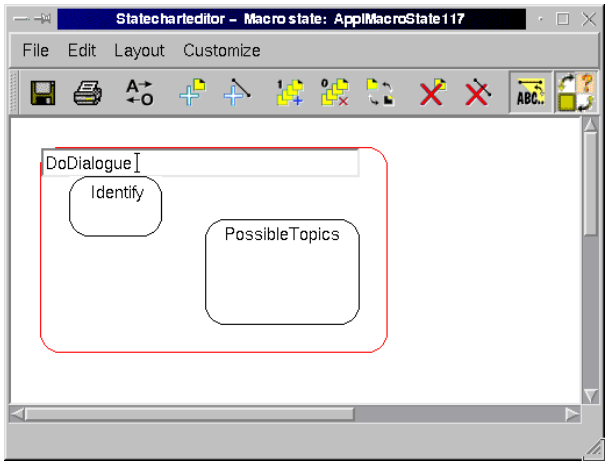

Figure 5: The Dialogue–Statecharts–editor.

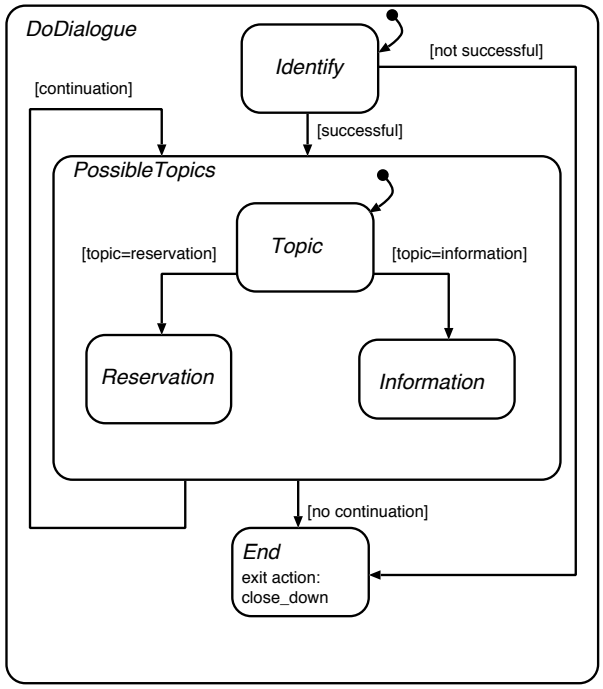

Figure 6: Describing dialogue flow in a statechart based manner. States are represented by rectangles with rounded corners and can be structured. Thus the state *DoDialogue* is an XOR–State. This indicates that the system can only be in one of the states lying graphically inside. The small rounded arrow at the state *Identify* means that this is the default entry state for *DoDialogue*. The transitions are labeled with conditions indicating when this transition is to be taken.

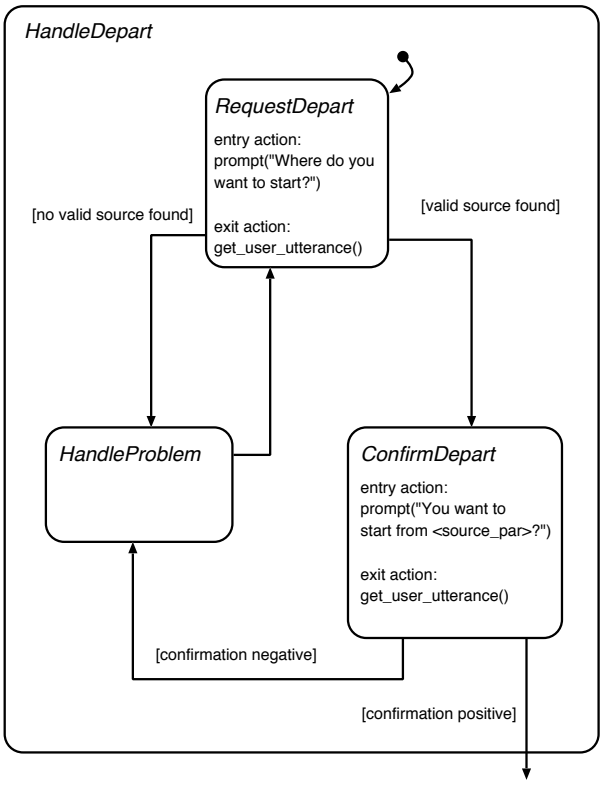

Figure 7: Refining states: the confirmation subdialogue in the reservation dialogue. The dialogue developer can add to the basic states actions to be triggered. Entry actions are executed when entering the state, exit actions when leaving the state.

as mixed– and user–initiative dialogues because there expressiveness was too small. With *Dialogue Statecharts* we think that we found a way to handle even such complex structures using the concurrency concepts of Harel's statecharts. Figure 8 shows an example for the representation of a mixed initiative dialogue. All the topics that a speaker may talk about in one sentence are represented as parallel slots of a concurrent dialogue state. All the slots represent parallel[4] substates of the dialogue system. If the speaker can tell the departure city, the destination city and the departure time in one sentence in a train time table information, there will be one slot for every parameter. The action which the dialogue system has to perform are described inside these slots. E.g. if the utterance contains a value for the departure time the value of a dialogue parameter concerned with the departure time must be set and analogous for the other parameters.

[4]This does not mean, that they really have to be processed in parallel, but that they are independent of one another.

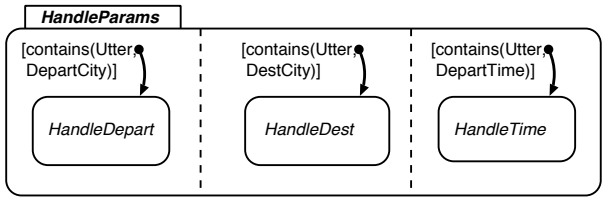

Figure 8: Concurrent states: dialogue parameters which the speaker can talk about in one sentence are handled in parallel. The picture represents e.g. for a train time table information that the speaker can tell the departure city, the destination city and the departure time at one time. The statechart in figure 7 is a possible refinement of the state *HandleDepart*.

Diamod supports the state–based modeling with graphical editors which can check consistency concerning the depth of the state–hierarchy, unwanted cycles, completeness of the system prompts etc. The rules which indicate when the model is consistent must be entered for every dialogue system, as they can be different according to the given system. States can be described in detail using masks similar to the ones used for task modeling (see figure 3). Here also prompts, conditions, actions and so on can be related to the state.

This is only a short description of what can be done with statecharts. The figures are simplified for reasons of clarity. Statecharts offer many features of abstraction which makes them capable of complex state modeling.

The models specified by the user of *Diamod* are internally represented as graphs which are also used as the basis for the model transformation. In order to do this, rules have to be specified how one graph can be automatically transformed into another. As different dialogue systems work with different concepts this transformation cannot be completely automatic. The approach here is to use defaults where possible and ask the user to make some additional editing, where needed. Some information can be lost by such a transformation. *Diamod* must warn the user about this.

3.1.3 Rule–based approach

Advanced dialogue systems are often not modeled using states and transitions but rules and conditions. *Diamod* can support this, too, as states can be used as abstract dialogue units. Thus states can represent subdialogues and dialogue steps. Every state can be modeled by a set of preconditions which indicate when the state may be entered and postconditions which represent when the state can be exited successfully. Rules can be specified to model how the next state to be activated has to be selected. There is a default order on the states which supports this selection. Some of these concepts are used for the application modeling of the DaimlerChrysler research dialogue system.

The benefits of *Diamod* in this context has not been investigated yet as one needs a well defined dialogue description language as interface to such a rule–based dialogue system.[5] Thus this is work for the future.

3.1.4 Concistency checking

An important point is that the tool is capable of checking the completeness of the models and their consistency. This is done using an object–oriented graph structure which represents all required concepts and the dependences between them. Consistency checks can be executed by formulating constraints on the graph using path expressions and having them examined by a special path interpreter (Ebert et al., 1996). Thus, it is possible to guarantee that for example

- there are no problematic cycles in the model
- there is a system prompt defined for every system initiative state (i.e. states where the system has to speak an utterance) and every parameter, so that the system never runs in a situation where it is 'speechless'.
- domains are defined properly for all parameters
- there is a following state in every situation (or the end of the dialogue)

4 Summary

The paper introduces the tool system *Diamod* which implements a universal approach for the specification of dialogue applications with a focus on task–oriented dialogue systems of the slot–filling kind. The tool system supports dialogue flow modeling in terms of tasks and states which can be specified in detail by describing parameters, actions, prompts and other typical concepts of dialogue models. The most important features of *Diamod* are

- a uniform knowledge representation which allows for automatic transformation of data for different generic dialogue systems
- the possibility of modeling different aspects of dialogue with different views on the data
- the capability of checking the consistency of the models automatically
- the support of the reuse of models
- the easy adaptability to additional knowledge bases and different dialogue systems.

4.1 State of work – technical realization

The task and statechart modeling are completely implemented as described in section 3. The following summary gives an overview over what *Diamod* contains up to now:

- task structure modeling as shown in figure 2
- *Dialogue Statecharts* modeling as shown in figure 5; this includes relating prompts, conditions, actions and events etc. to the dialogues. These are described in masks as shown in figure 3
- automatic prompt table generation
- system parameters and application dependent application parameters which represent the dialogue state
- mapping from application parameters to data base parameters; e.g. if the caller talks about "tomorrow" this has to be mapped into the actual date in a form that can be handled by the database such as 03.02.99
- attaching multilingual system prompts to the modeled dialogues.

The system is implemented in C++ using graphs and one set of constraints per dialogue

[5] This would be a quite interesting project and we would be grateful for suggestions of collaboration here.

system, which represents the consistency rules for this system.

We are currently working on adapting the system to the needs of Temic–DDL (a dialogue description language developed by Temic) and VoiceXML (AT&T et al., 2000) and on the automatic transformation of models. The integration of a grammar specification tool (work in progress) is planned for the end of the year. This module will provide different grammar formalisms such as UCG (Zeevat, 1988), PSG (Boros, 1997) and Java Speech API (Sun microsystems, 2000). The conversion between these grammar types will be supported.

The implementation of the system has just been finished so far that it can be used by application developers. But as it is completely new and the graphical user interface is still being improved in order to make it more intuitive, we have not made any experience yet how much the win of using *Diamod* will be for realistic dialogues. We are currently starting the evaluation and we are optimistic after the first tests.

4.2 Outlook

The dialogue systems we aimed at when we developed *Diamod* were mainly task–oriented systems, i.e. systems giving information on special topics or modifying databases. The benefits of *Diamod* in another context like translation systems (e.g. Verbmobil (Wahlster et al., 2000)) has not been investigated so far, but this is one of our goals in the future.

Another interesting topic would be the adaptation of *Diamod* to dialogue systems using dialogue grammars (Reichman, 1981) or plan–based systems (Cohen and Levesque, 1980).

Further plans include the integration of a prototyper into the tool system to be able to immediately check the consequences of the modifications made. With these different means it will be possible even for an untrained user to specify new applications for his or her own requirements.

References

AT&T et al. 2000. *VoiceXML*. World Wide Web, http://www.voicexml.org/.

Eric Bilange. 1991. A task independent oral dialogue model. In *Proceedings of the Fifth Conference of the European Chapter of the Association for Computational Linguistics*, pages 83–88, Congress Hall, Alexanderplatz, Berlin, Germany.

Manuela Boros, Ute Ehrlich, Paul Heisterkamp, and Heinrich Niemann. 1998. An evaluation framework for spoken language processing. In *Proceedings of the International Workshop Speech and Computer 1998*, Russian Academy of Sciences, St.Petersburg, Russia, October.

Manuela Boros. 1997. Gepard - dokumentation des parsers f"ur phrasenstrukturgrammatiken. Projektbericht, FORWISS, Juni.

Astrid Brietzmann, Fritz Class, Ute Ehrlich, Paul Heisterkamp, Alfred Kaltenmeier, Klaus Mecklenburg, Peter Regel-Brietzmann, Gerhard Hanrieder, and Waltraud Hiltl. 1994. Robust speech understanding. In *International Conference on Spoken Language Processing*, pages 967–970, Yokohama.

Philip R. Cohen and Hector J. Levesque. 1980. Speech acts and the recognition of shared plans. In *Proceedings of the Third Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 263–271.

Ron Cole. 1999. Tools for research and education in speech science. In *Proceedings of the International Conference of Phonetic Sciences*, San Francisco, USA, August.

Jürgen Ebert, Angelika Franzke, Peter Dahm, Andreas Winter, and Roger Sttenbach. 1996. Graph based modeling and implementation with eer/gral. In B. Thalheim, editor, *15th International Conference on Conceptual Modeling (ER'96), Proceedings*, number 1157 in LNCS, pages 163–178, Berlin. Springer.

Ute Ehrlich, Gerhard Hanrieder, Ludwig Hitzenberger, Paul Heisterkamp, Klaus Mecklenburg, and Peter Regel-Brietzmann. 1997. ACCeSS - automated call center through speech understanding system. In *Proc. Eurospeech '97*, pages 1819–1822, Rhodes, Greece, September.

Ute Ehrlich. 1999. Task hierarchies - representing sub-dialogs in speech dialog systems. In *6th European Conference on Speech Communication and Technology (EUROSPEECH)*, Budapest, Hungary, September.

Klaus Failenschmid and J.H. Simon Thornton. 1998. End-user driven dialogue system design: The reward experience. In *Proceedings*

of the International Conference on Spoken Language Processing (ICSLP) 1998, Sydney, Australia, November.

Gerald Gazdar. 1981. Speech act assignment. In Aravind K. Joshi, Bonnie Lynn Webber, and Ivan Sag, editors, *Elements of Discourse Understanding*, pages 63–83. Cambridge University Press, Cambridge.

David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Paul Heisterkamp and Scott McGlashan. 1996. Units of dialogue management: An example. In *Proc. ICSLP '96*, volume 1, pages 200–203, Philadelphia, PA, October.

Paul Heisterkamp, Scott McGlashan, and N. Youd. 1992. Dialogue semantics for an oral dialogue system. In *International Conference on Spoken Language Processing (ICSLP), Volume1*, pages 643–646, Banff, Alberta, Canada.

Anke Kölzer. 1999. Universal dialogue specification for conversational systems. In *Proceedings of the International Workshop: Knowledge and Reasoning in Practical Dialogue Systems, IJCAI 1999*, pages 65–72, Stockholm, Sweden, August.

Jeremy Peckham. 1993. A new generation of spoken dialogue systems: Results and lessons from the sundial project. In *3rd European Conference on Speech Communication and Technology (EUROSPEECH'93); Vol.1*, pages 33–40, Berlin, September.

Rachel Reichman. 1981. *Plain-speaking: A theory and grammar of spontaneous discourse*. Ph.D. thesis, Department of Computer Science, Harvard University, Cambridge, Massachusetts.

Sun microsystems. 2000. *Java Speech API*. World Wide Web, http://java.sun.com/products/java-media/speech/index.html.

Wahlster et al. 2000. *Project Verbmobil*. World Wide Web, http://www.coli.uni-sb.de/~vm/.

Henk Zeevat. 1988. Combining categorial grammar and unification. In *Reyle, Rohrer: Natural Language Parsing and Linguistic Theories*, pages 202–229, Dordrecht. D. Reidel Publishing Company.