Universal Dialogue Specification for Conversational Systems

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

Spoken Language Dialogue Systems which allow for spontaneous speech are not very widely spread. You find a few systems in application domains such as train time–table information or flight ticket reservation (see Bernsen et al. (1998) for an overview). 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. For every application domain different kinds of knowledge bases have to be specified. To be able to accelerate this process and to make it also transparent for non–experts we are currently working on methods and tools which support this development. Our aim is to provide specification models which are universal enough to be interpreted within different dialogue systems, i.e. different implementations of generic conversational systems. With the help of special design methods and a uniform representation of data the tools will allow a consistent specification of dialogue systems and a transformation between different models.[1]

1 Introduction

Our research group has developed a dialogue system which is able to understand spontaneous speech speaker–independently and carry on dialogues on special topics. 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 & McGlashan (1996), Ehrlich et al. (1997), Boros et al. (1998) for further information). Usually a caller/user of the system wants information on a special topic. Sometimes the user also wants to modify a data base e.g. when booking a reservation.

Every time a new application is developed similar operation steps have to be executed to obtain a structured and maintainable dialogue. Among other jobs one has to:

[1] Thus the focus here is on knowledge representation rather then knowledge inference.

– model the dialogue structure: i.e. divide the dialogue into subdialogues to handle a special part of the interaction like the identification of a caller

– define the application parameters, i.e. the parameters necessary to give information to the caller or access a database like the name of the caller

– attach system prompts to dialogue situations like what the system has to say when asking the name of the caller

– define the appropriate vocabulary (pronunciation) and train the language models

– define linguistic structures (lexicon, grammar, semantics)

– define the interface to the application system (e.g. an SQL–interface to a data base)

Currently most of these knowledge bases have to be programmed in programming languages like Prolog and *C*, which makes their maintenance (e.g. consistency checks) and modification difficult. This is the case for our dialogue system and systems from other companies as well. It is also nearly impossible for untrained persons to specify or modify an application as this requires the experience of an expert.

Besides this there are different implementations of dialogue systems within our company and of course from other suppliers.

We decided therefore to develop tools which open up the possibility to specify the necessary knowledge bases easily by appropriate methods, tailored for the problem, using visual languages and graphical user interfaces. These tools have to be able to guide the user while defining the models, to check the consistency and tell the user about missing data or contradictory information.

The designed tool concept shall not be restricted to the specific structure of the DaimlerChrysler research system but must be configurable with little effort for similar requirements of different interactive speech dialogue systems and different users. Thus the basic idea is a *universal* approach for specifying conversational systems. My focus in this paper is on *dialogue flow management*.

2 Requirements

The DaimlerChrysler research dialogue system (see section 3) allows for spontaneous speech whereas

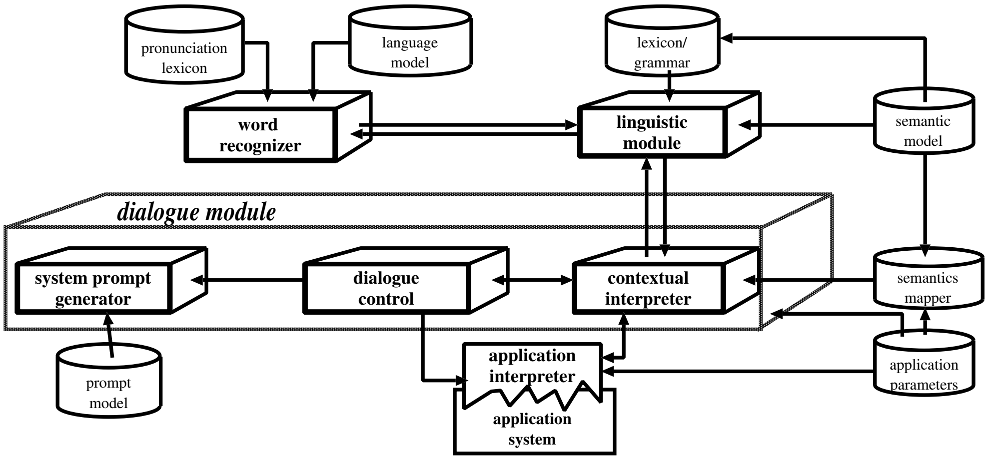

Figure 1: A typical architecture of a generic conversational system: the DaimlerChrysler research system. The boxes represent modules which process a special functionality, the cylinders show the application knowledge bases. The arrows between the modules show the dataflow, and arrows from the knowledge bases show where these data are used.

other dialogue systems are only capable of processing single commands. Currently the focus of our interest is a tool for developing dialogue flow models which can automatically be transformed in a way that a different dialogue system can interpret the result. This means that the knowledge shall be represented in a universal way so that different aspects of dialogue can be modeled and code for different dialogue systems can be generated. E.g. 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 task–based one (as will be described in section 5). 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.

The user shall be supported by a CASE–tool (Computer Aided Software Engineering) specialized for language engineering which provides all the concepts necessary for dialogue specification. To be able to develop new and modify old knowledge bases easily, the tool must support the language engineer with the following functionality:

- provide visual languages for the specification of structured dialogue data
- represent all relevant data and the dependences between them
- support a formalism for defining constraints on the models and informing the user of violations of these constraints
- generate code (Prolog, C, standardized speech API–code ...) that can be interpreted by the currently preferred generic dialogue system
- support the reuse of formerly developed domain models.

3 A Generic Dialogue System

The requirement for a tool system was raised by the fact that developing new applications for our dialogue system was rather costly. We needed a support to accelerate this process. As our dialogue system is a typical generic conversational system its architecture is presented here as an example (see figure 1). The domain dependent data are kept in special knowledge bases and application interfaces. Thus only these knowledge bases have to be changed when a new application is set up. I will not describe the knowledge bases here as most of them should be known to the reader. The structure and algorithms are based on concepts developed in the Sundial project (Peckham (1993)). As it is a generic system there are algorithms that work domain independent and interpret the knowledge bases.

The tool system described in this paper supports the development and maintenance of these knowledge bases for generic dialogue systems, not restricted to the DaimlerChrysler research system.

4 The Structure of the Tool System

Figure 2 shows the principles of working with the tool system. The central unit is the tool system which provides methods to specify 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 (described in section 6) and is able to generate code in different languages such as Prolog or *C* dependent on

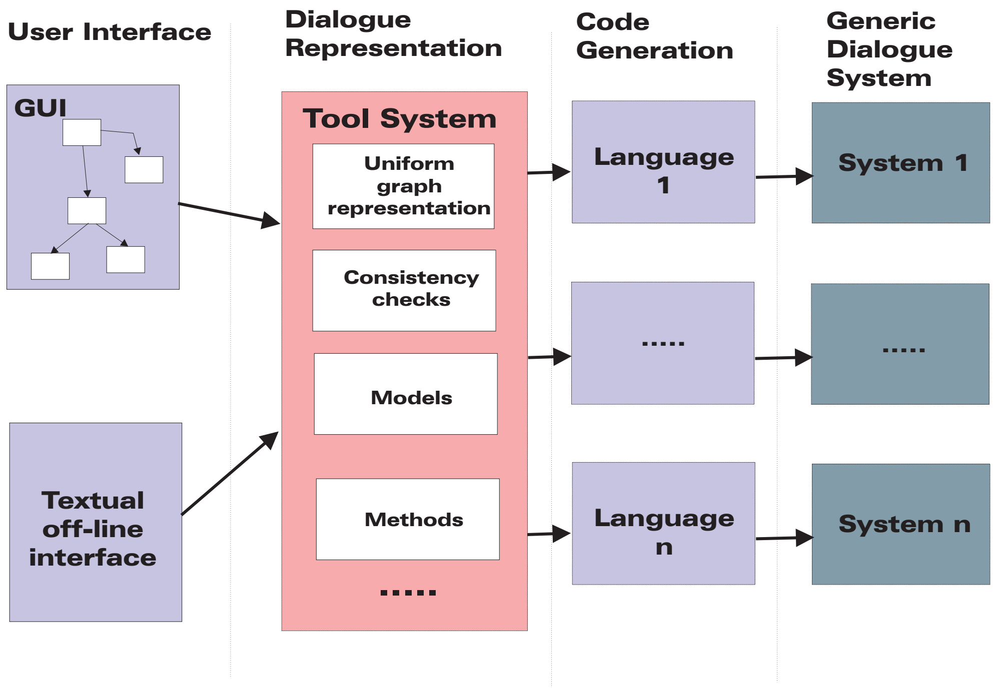

Figure 2: Specification of dialogue with the tool system. The central unit is the tool system which provides methods for the dialogue specification, keeps the data and is capable of checking the consistency on them. Data are modeled by the user on–line with the help of a graphical user interface or textual dialogue description languages for off–line specification. When the specification is complete the tool system generates the code necessary for the dialogue system in use.

the generic dialogue system currently in use. This code output (commonly spoken textual files) is read by the corresponding generic system and interpreted when the dialogue system is running.[2]

The tool system shall be capable of modeling what the system has to do in a given situation. As this must work for different generic dialogue systems, the tool system must also consider the generic features of the system (because they can be different for different dialogue systems). This leads us to a two–phase approach.

4.1 Generic versus Application Mode

The tool system is organized in a two–phase architecture and distinguishes between a generic expert mode and an application developer mode[3]. There are two phases of specifying data. In the first phase an expert who knows everything about the internal structure of the generic dialogue system specifies generic data. These data are used in the second phase to generate generic states[4] of the dialogue system automatically and to do consistency checks. They are also important for the code generation.

In the second phase – the application mode – the system supports users with only rudimentary knowledge of how the dialogue system works internally and how it is implemented. The information specified by the expert is used here, but is wrapped in a way that possibly confusing data are hidden from the application developer.

Both user groups use mainly the same concepts and functionality of the tool system, but with different views on them. The expert defines generic parameters while the application developer defines the application dependent application parameters.

To show the principles of the tool system let's start with a popular example – a train time–table information system which can also do ticket reservation. The dialogue below shows a possible interaction:

Example

1. System: Where do you want to go?
2. User: From here to Cologne.
3. System: When do you want to go?
4. User: At 10 o'clock in the morning.
5. System: You want to go from Munich to Cologne at 10 o'clock a.m.?
6. User: Yes.
7. System: There is a train at a quarter past 10 from Munich to Cologne.

Phase one – generic mode:
Before any application can be specified with the tool system the dialogue system expert describes the generic part of the system. In the case of our research system the only thing which must be done in this phase is the specification of the generic system parameters, i.e. the dialogue acts and the sentence

[2] In some cases the code will be compiled of course.

[3] In some cases the expert and the application developer will be the same person.

[4] A generic state is a description of what kind of states the system can be in at all, e.g. confirming and requesting parameters with respect to a current dialogue strategy.

type. The dialogue acts are a fixed set of application independent values. Our research system uses for example the dialogue act *inform* for the situation when the user gets an information from the system as in the interaction example given above in prompt 7 and *request* when the system requests from the user a special value as in prompt 1. Another important dialogue act is *confirm* for the confirmation of parameters (prompt 5). Besides this the dialogue system knows sentence types such as *statement* and *question* which define the form of the next prompt.

The expert of the dialogue system defines all these generic variables known by the dialogue system with the help of the tool system. With the given values the tool system calculates possible combinations like *request – question* which represent generic system states. Combinations that are senseless can be deleted by the expert. The expert adds some guide texts which comment the variables and which will be shown to the application developer in phase two together with the calculated system states.

Phase two – application mode:
In order to specify a special application it is useful to follow a specific sequence of steps. Usually the developer will design the application in a top–down manner, first of all describing what topics the system is able to handle in a given sequence (task/state modeling), and then refining this structure. This is done by describing the parameters which are typical of the application and the system prompts i.e. the utterances that must be spoken by the system dependent on the current system state. The dialogue developer defines the application specific application parameters with their domains, for example departure city with the domain *String* and departure time with a special domain *DateType*. The tool system supports the definition of arbitrary domains similar to structured types in high level programming languages. With these data and the data given by the expert in phase one the system calculates possible system states combining generic and application dependent variables like *request – question – departure time*. On this basis the developer is asked to define the system prompt which shall be used in this situation. This is done entering system prompts in the dialogue task or state model (described in section 5.1 and 5.2).

5 Representing Dialogue Flow

The user of the tool system can specify data in a task or state oriented way. The methods provided by the dialogue flow tool describe:

- *the task graph* which represents which tasks can follow each other (see figure 3). Here every subdialogue and the sequence relation between them is modeled.
- *single tasks*. Here the user refines the tasks he set up in the task graph. He describes which application parameters can be talked about in the appropriate situation and which system prompts should be uttered by the dialogue system (see figure 4) for parameter confirmation, in the case of error and so on. In the figure prompts for confirmation are shown.
- *state models* on the basis of statecharts, specially tailored for the dialogue problem
- *activity models* and *event models* to give the developer the opportunity of programming special state dependent actions such as changing to a special speech recognizer or dealing with barge–in.

As our aim is a universal approach with automatic transformation between models the tool system can automatically calculate the state model from the task model and vice versa.

5.1 Dialogue Flow for a Task–Based System

Our research system is a task–based system (see Ehrlich (1999)). It only needs a rather abstract definition of the dialogue flow as shown in the figures 3 and 4. It is not necessary to model exactly the states the system can be in and their sequence as it often has to be done for other dialogue systems. The generic research system has a default behavior concerning what to do when, e.g. choosing dialogue acts and a dialogue strategy (as described in Heisterkamp & McGlashan (1996)).

A task is described by a set of system variables and their values, such as application parameters and some internal values. System prompts are attached to the tasks in the form of templates.[5] The task represents a kind of subdialogue in which special topics (represented by application parameters) can be talked about such as an *identification task* where a caller has to give his name and password or an *end–of–dialogue–task* where the dialogue system has to close or reinitialize the internal settings. With these means one can describe how the dialogue is to be continued in a given state of a task and what has to be uttered by the system.

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 tasks i.e. the components of the dialogue (see figure 3); e.g. a subdialogue to handle the identification of the caller and find out why he calls and a subdialogue for reservation of a ticket, and one for callers who only want information.
- definition of the task structure i.e. what does the system have to handle first and what comes next which is also shown in figure 3. This is done by defining a start task and the successors of each task[6]
- attachment of application parameters to the tasks; e.g. in the identification task 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

[5] Currently we do not use generation of system utterances from semantic structures.

[6] When the graphical user interface is complete this will easily be done by drawing a task graph like the one in figure 3.

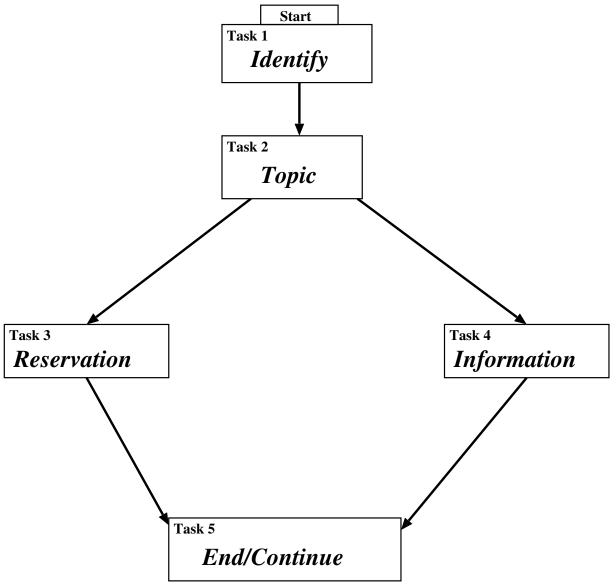

Figure 3: Task structure graph. Each rectangle models a task i.e. a subdialogue. The edges between the tasks show how tasks can follow each other. At any time it is possible to go back to a previous task.

confirm the parameter DepartCity in the reservation task (as shown in figure 4)[7]

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.

Up to now the implemented model of the tool system contains

- task and task structure modeling with automatic state generation; i.e. all combinations of generic and application parameter values are calculated automatically in order to gain all those system states, where a system prompt is needed. A typical *state* in this context is the situation where the system must execute a confirm–dialogue in a special task. For example confirming the parameter *DepartCity* in the task *Reservation* in figure 4 is described by a state.
- 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 system prompts to the modeled tasks.

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 the object–oriented graph structure described in section 6 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)

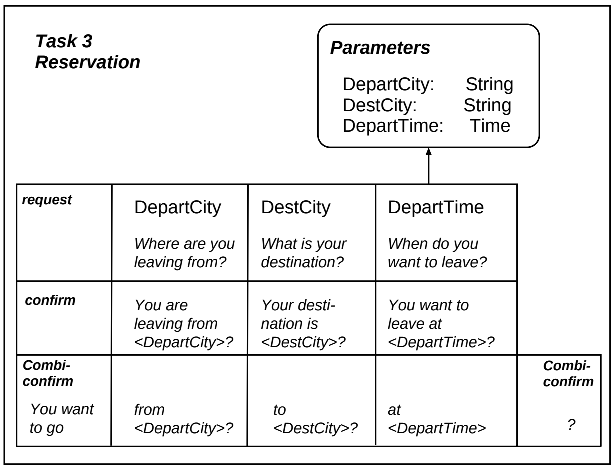

Figure 4: Defining the details of a task. The application parameters that are talked about in this task have to be attached to it, shown in the rectangle with rounded corners (*DepartCity*, *DestCity*, *DepartTime*). For every dialogue act and every application parameter there must be a system prompt defined. The table here is calculated automatically by the tool system using the generic parameters defined by the expert in phase one and the application parameters defined here. The *combi–confirmation* is used for confirming several parameters in one step. The application developer only has to fill in the system prompts.

5.2 Dialogue flow for state based systems

Many other dialogue systems do not handle dialogue flow by modeling tasks and task structure. They use a state based approach where dialogue flow is described in detail using state–transition–models combined with events (see Failenschmid & Thornton (1998), Cole (1999)). Simple state–transition–models are adequate for very simple dialogue systems such as command–and–control systems.[8] As conversational systems have a high complexity of states, the expressiveness of state–transition–models is too small to be

[7] These states can be calculated automatically out of the generic and application parameters given by the dialogue expert and the application developer (compare description of phase one and two above).

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

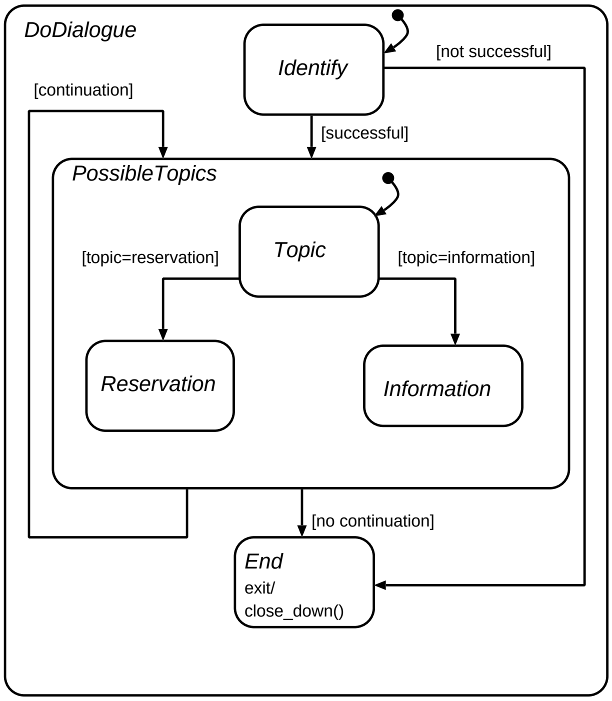

Figure 5: Describing dialogue flow in a statechart based manner. States are represented by rectangles with rounded corners and can be structured. Thus the state *DoDialogue* is an XOR–State. This indicates that the system can only be in one of the states lying graphically inside. The small rounded arrow at the state *Identify* means that this is the default entry state for *DoDialogue*. States can be refined, i.e. their structure can be described in another statechart. The transitions are labeled with conditions indicating when this transition is to be taken.

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.

Thus modeling of complex dialogue flow can be done in a rather intuitive way. Figure 5 is an example of modeling our task data described in section 5.1 in a state–based way. The tasks 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.[9] In simple cases a task is represented by a basic state (*End*) which need not be refined any more. The reservation–task–complex–state must be refined into substates, one for each dialogue act. These are refined again as shown in figure 6. 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* resp. reenter in *PossibleTopics*, according to a continuation flag.

[9]A different graphical layout is used for concurrent states.

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 implementation of the state–based modeling is still under development. The statechart dialect implemented in the tool system will not provide the complete expressiveness of the statechart class. It will be tailored to dialogue specification so that it is intuitive and easy to use. It will nevertheless offer the opportunity of adding code segments to the models.

6 Graph Based Modeling

This section describes the basis for the implementation of the approach. For a uniform representation of all data needed, I chose a graph based model (Ebert et al. (1996)). One has to distinguish between the generic graph class on the one hand, which represents the concepts necessary for dialogue modeling in general, and graph instances of this class on the other hand, one for each application. In order to be able to handle different dialogue systems such as the DaimlerChrysler research system or a system of a different company the generic graph class must be common enough for the required universal approach. This means that it has to include every *concept* which is typical for dialogue systems. Typical concepts in this context are dialogue states, prompts, parameters, grammar specific features etc.

The graph class defines the graph syntax in terms of classes of vertices and edges with constraints on how they can be related. Thus application parameters can be related to tasks and tasks can be related to other tasks in order to specify the task structure. Because the models are object–oriented it is possible to declare inheritance relations between the classes. Thus, there is one vertex class for every concept such as a class of parameters, refined (per inheritance) into application parameters and generic parameters and a class of system prompts. This is the basic idea of this universal tool system approach: defining a graph class in a way that it knows every concept typical for dialogue systems of the current state of the art and thus be able to model dialogue knowledge bases for every conversational system one likes, only making small adaptations.

Besides this, constraints on the graphs are modeled such as "make sure there is a system prompt for every system–initiative state". This is done using GReQL (graph repository query language) – a special logic based language for graph queries (see Ebert et al. (1998), Franzke (1996)).

The resulting graph instances are the models which represent concrete application data. There you find instances of e.g. application parameters such as the departure time in the train time–table application with its current value. The task structure is also in an application graph instance. By examining the graph structure it is rather easy to inform an application

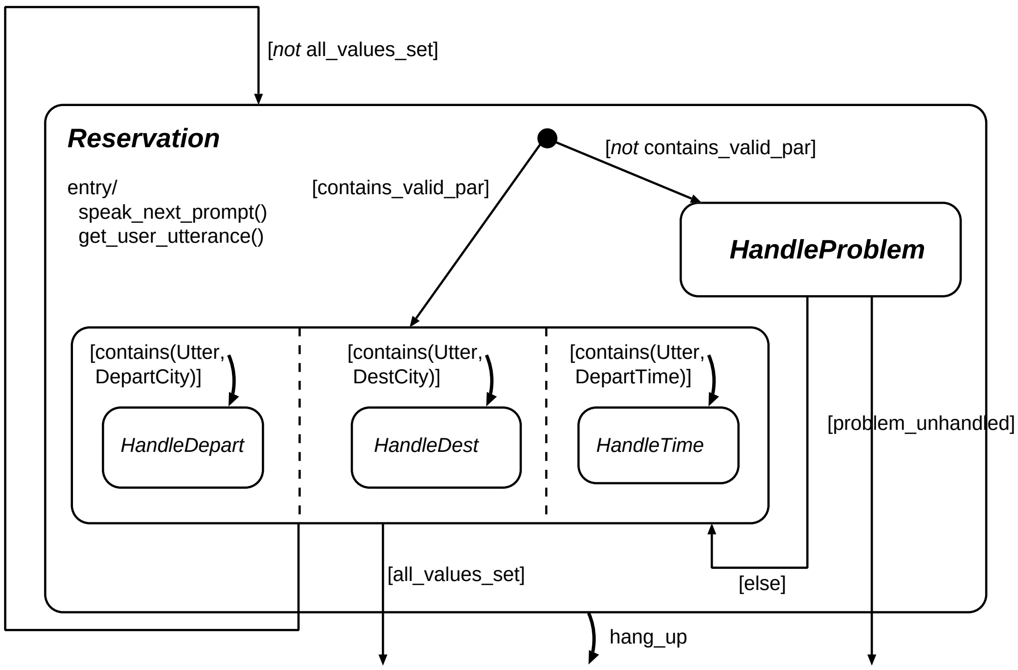

Figure 6: Refining the state *Reservation*. The dialogue developer can add actions to be triggered to the basic states. Entry actions are executed when entering the state, exit actions when leaving the state. The little black circle is the entry point of the state *Reservation*. The states separated from one another by dotted lines are concurrent states. This means that they can be handled in parallel independent from one another. The *hang_up*–Arrow at the bottom of the figure shows an event (i.e. the caller hangs up) which would trigger an immediate exit from every substate of *Reservation* to some close–down–state.

developer of incomplete data such as a missing system prompt for the *confirm*–state in the reservation task in figure 4.

The graph representation provides a uniform data model which is independent of a special dialogue system. As the code generation is strictly separated from the models and is actually only the *interpretation* of a given graph it is possible to interpret a graph in different ways. Instead of generating Prolog knowledge bases describing the tasks and the task structure for our research for example (which only needs the rather abstract task and state modeling described in 5.1) it is also possible to generate any other code such as *C*–code where every state and the state sequences are described in detail (or a certain dialogue description language). For this kind of dialogue system it is possible to specify the dialogue structure using the state based dialogue model as shown in figure 5.

The tool system can be adapted to different dialogue systems by writing new code generation routines specially for the target system and adding GReQL constraints for such a dialogue system. Thus different constraints and different code generation are the only additions the tool implementor has to make when adapting to another dialogue system. Concerning the dialogue system specific constraints and using the different interpretation routines it is possible to automatically translate a model which was originally developed for dialogue system A (e.g. the research system) to dialogue system B (e.g. a simple state based command–and–control system like those DaimlerChrysler uses in cars for controlling radio etc.). As one will always find concepts that are known by system A and not by system B this translation will usually not be complete. As the completeness of the individual dialogue models is described by GReQL predicates the tool system is able to inform an application developer of what he has to add to gain a complete model. This automatic translation will not be an easy business in any case but I am convinced that this is a useful approach to gain a universal dialogue modeling tool.

7 Summary

The paper introduced a universal approach for the specification of applications for generic conversational systems. The focus here is on dialogue flow modeling but the principle can be extended to the other knowledge bases, necessary for dialogue application development, as well. The most important features of the tool system 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 easy adaptability to additional knowledge bases.

7.1 State of work – technical realization

The task modeling described in section 5.1 is completely implemented whereas the statechart based approach is still under development. The approach is implemented in C++ using the graph classes and GReQL–constraints as described in section 6.

We are currently working on an easy to use graphical interface so that the user will be able to specify the application data easily and guided by the system. The system can tell the user what to do next and which data are still incomplete or even inconsistent. There will be help routines in every layer which support the user with instructions concerning the current situation. This could be extended with guidelines of dialogue management as suggested by Bernsen et al. (1998).

7.2 Outlook

Currently we are gathering experience in the use of the tool system. Our further aim is to integrate options for the specification of the remaining knowledge bases like phonetic and linguistic structures into the dialogue modeling tool. Further plans include the integration of a prototyper into the tool system 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

Bernsen N.O., Dybkjaer H., Dybkjaer L. Designing Interactive Speech Systems - From First Ideas to User Testing. Springer Verlag, 1998

Boros M., Ehrlich U., Heisterkamp P., et al. An evaluation framework for spoken language processing. In *Proceedings of the International Workshop Speech and Computer 1998*. Russian Academy of Sciences, St.Petersburg, Russia, 1998

Brietzmann A., Class F., Ehrlich U., et al. Robust speech understanding. In *International Conference on Spoken Language Processing*, pp. 967–970. Yokohama, 1994

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

Ebert J., Franzke A., Dahm P., et al. Graph based modeling and implementation with eer/gral. In Thalheim B., ed., *15th International Conference on Conceptual Modeling (ER'96), Proceedings*, no. 1157 in LNCS, pp. 163–178. Springer, Berlin, 1996

Ebert J., Gimnich R., Stasch H., et al. GUPRO - Generische Umgebung zum Programmverstehen, vol. 10 of *Koblenzer Schriften zur Informatik*. Fölbach, Koblenz, 1998

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

Ehrlich U., Hanrieder G., Hitzenberger L., et al. ACCeSS - automated call center through speech understanding system. In *Proc. Eurospeech '97*, pp. 1819–1822. Rhodes, Greece, 1997

Failenschmid K., Thornton J.S. End-user driven dialogue system design: The reward experience. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP) 1998*. Sydney, Australia, 1998

Franzke A. Querying graph structures with g^2ql. Fachbericht Informatik 10/96, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1996

Harel D. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8:231, 1987

Heisterkamp P., McGlashan S. Units of dialogue management: An example. In *Proc. ICSLP '96*, vol. 1, pp. 200–203. Philadelphia, PA, 1996

Peckham J. 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*, pp. 33–40. Berlin, 1993