1 Introduction

Many dialogue applications are for telephony domains such as train schedules, call centers and telematic systems (see Brietzmann et al. (1994), Heisterkamp & McGlashan (1996), Ehrlich et al. (1997), Boros et al. (1998) for further information). Developing such applications involves among other things the following steps:

- The dialogue structure must be modelled (for example divide the dialogue into subdialogues).
- The application parameters must be defined, i.e. the parameters necessary to provide information to the caller or to access a database (such as the name of the caller).
- System prompts must be defined for the different dialogue situations.
- Linguistic structures (lexicon, grammar, semantics) must be defined.
- The interface to the application system must be implemented (for example an SQL interface to a database).
- The appropriate vocabulary (pronunciation) must be defined.
- The language models for the speech recognizer must be trained.[2]

Figure 1 shows the knowledge bases which have to be modelled by the dialogue developer.
Currently, many of these knowledge bases have to be programmed in conventional general purpose programming languages such as Prolog and C, which makes maintenance and modification difficult. This is the case for *Speedis*, a DaimlerChrysler research dialogue system, and dialogue systems from many other companies and research groups as well. Usually it is nearly impossible for an untrained person to specify or modify an application easily since much experience is required.
Our approach to solving these problems is to develop a tool system which opens up the possibility of specifying the necessary knowledge bases easily by appropriate methods, tailored to the problem. The dialogue structure is modelled in a graphical form using visual languages and graphical user interfaces. This tool system is able to guide the user while defining the models, to check their consistency, and inform the user about missing or contradictory data.
Besides this there are various implementations of dialogue systems within our company and of course from other suppliers. The designed tool system may therefore not be restricted to the specific structure of *Speedis*. Instead it is configurable with little effort for similar requirements of various interactive speech dialogue systems and various users. The basic idea is to provide a *universal* approach to specifying conversational systems. The focus in this paper is on *dialogue flow management*.

2 Requirements and Goals

Currently our focus is on a tool for developing dialogue flow models for different kinds of dialogue systems, such as rule-based (see for example Cohen & Levesque (1980), Searle (1969)) or state-transition-based (see for example Failenschmid & Thornton (1998)) generic systems. These models

[2] We will not follow this here since this belongs to the field of speech recognition.

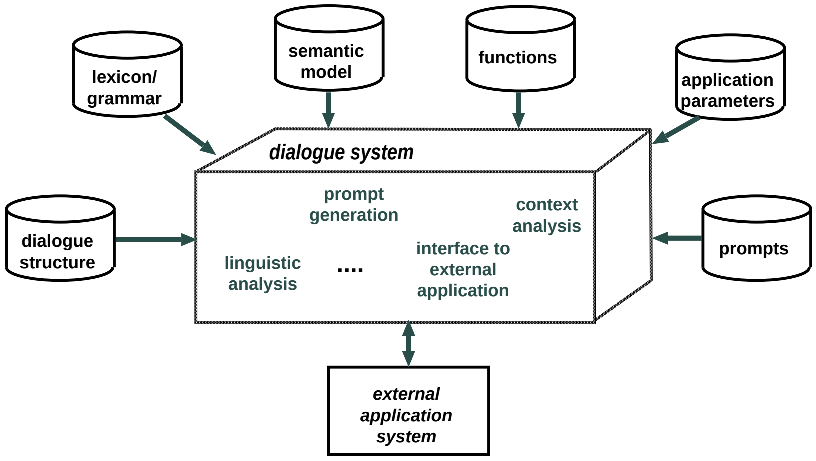

Figure 1: *Generic conversational systems and their knowledge bases. The dialogue system executes various analysis steps on the user utterances, such as linguistic analysis and context sensitive semantic interpretation. Dependent on the result, it generates system prompts or accesses an external application such as a database or the internet. In order to be able to model this, the different knowledge bases shown (cylinders) have to be specified. The dialogue structure models what the system should do first, for example which subdialogue it should start with, which prompt should be spoken and which grammar should be activated. Semantic and linguistic models (lexicon/grammar) represent the structure and the meaning of the user utterances. Usually a set of functions is needed for this interpretation and for the access to the external application for example an SQL interface to a database. Application parameters specify what is talked about such as the name and the password of the user. Predefined prompts indicate what the system can say (in case the dialogue system does not generate prompts from semantic structures).*

must be expressive enough to represent user-, mixed- and system-initiative dialogues. Our approach was especially developed for dialogue systems with a predictable dialogue structure, such as information-giving systems (for train schedules etc.) or hardware-controling systems (for video recorders etc.). Based on these requirements we decided to develop a kind of *meta dialogue model* which is universal enough to represent the needs of various systems.
A tool system had to be implemented which supports this modelling. The tool provides possibilities of editing such models rather easily and generate code – i.e. a textual description of dialogue – for a currently focussed dialogue system (see Figure 2 for an overview). This could, for example, be Prolog predicates for *Speedis*, or an XML-based language. One language, we want to support, is *VoiceXML* (AT&T et al. (2000)), which seems to be becoming a standard for commercial systems. Another is *TemicGDML*, which is also an XML-based language,[3] very similar to *VoiceXML*. Our approach can support all generic dialogue systems that provide some kind of textual dialogue description language.
Different dialogue systems work with different concepts and different features of concepts. For example *VoiceXML* and *TemicGDML* organize the dialogue structure by specifying XML documents with several dialogues

[3]development by DaimlerChrysler daughter Temic, unpublished

(called `form` in *VoiceXML* and `dialog` in *TemicGDML*) which are described in detail by dialogue steps (called `field item` in *VoiceXML* and `step` in *TemicGDML*). These dialog description languages are imperative programming languages.[4] The application developer has to model among other things the request and the confirmation of a particular parameter by defining such steps in an imperative manner, i.e. more or less as states and transitions. On the other hand *Speedis* uses a fixed set of dialogue acts and rules, and strategies for how they should follow each other (see Section 4.1 and Heisterkamp et al. (1992), McGlashan (1996)) instead of an explicit state-based step modelling. The models required for a universal representation have to support both state-transition-based as well as rule-based approaches.
The next goal is an automatic transformation of dialogue models. Since developing dialogue applications is rather costly, we want to create re-usable dialogue modules and collect them in dialogue libraries. Such modules could be a digit module which returns a number of digits (for telephone or credit cards) and a yes-no module for confirming given parameters. In order to be able to re-use such modules in different dialogue systems the tool must provide transformation algorithms: a dialogue module developed for system A must be transformed automatically into a dialogue module for system B. The tool system described here is a CASE tool (Computer Aided Software Engineering) specialized for language engineering. The following list sums up the functionality:

visual languages to specify structured dialogue data

universal dialogue model to represent all relevant data and their interdependencies

consistency checks with a formalism for defining constraints on the models and informing the user of violations

code generation such as Prolog, C, and *VoiceXML* that can be interpreted by the currently preferred generic dialogue system

dialogue module libraries supporting the re-use of formerly developed domain models

transformation algorithms for the re-use of models in different dialogue systems

3 The Structure of the Tool System

To show the principles of the tool system let us start with a popular example – a train time-table information system which can also make ticket reservation. The dialogue below shows a possible interaction in a typical information-giving dialogue system:

Example

1. System: Where do you want to depart from?
2. User: From here to Cologne.
3. System: When do you want to leave?

[4]The programmer specifies an explicit sequence of steps to follow to produce a result.

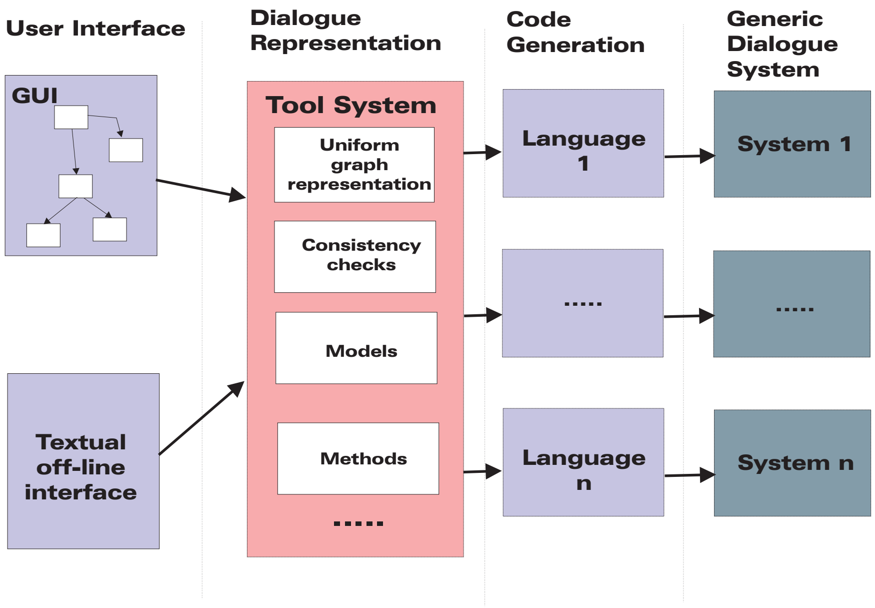

Figure 2: *Specification of dialogue with the tool system. The central unit is the tool system which provides methods for the dialogue specification, keeps the data, and checks their consistency. Data are modelled by the user on-line with the help of a graphical user interface, or off-line with a textual dialogue description language. When the specification is complete, the tool system generates the code for the dialogue system being used.*

4. User: At 10 o'clock in the morning.
5. System: You want to leave from Munich for Cologne at 10 a.m.?
6. User: Yes.
7. System: There is a train leaving Munich at 10.15 a.m. for Cologne.

The models, which can be edited with the tool system, are organized in dialogue units, which are represented as abstract states. The user can connect these dialogue units to actions, conditions, application parameters, grammars, and other concepts. The tool system can be configured for every dialogue system in use, which means that the graphical user interface will change dependent on the target dialogue system. The user only sees the information that is relevant for him. The tool system must be able to check if a current model is correct and complete dependent on a particular dialogue system. Therefore there is a set of consistency rules for every dialogue system (see Section 6.1).
Figure 2 shows how the tool system is used. The central unit is the tool system which provides methods for specifying knowledge, keeps the data and models, and checks consistency (see also Kölzer (1998)). The user modifies the models with 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 dialogue description languages. This text will be imported into the tool system with appropriate parsers. The tool system represents data in a universal graph representation (described in Section 6) and can generate code in various languages such as Prolog or *VoiceXML* dependent on the generic dialogue system currently in use. This

code output (commonly spoken textual files) is read in by the corresponding generic system and interpreted when the dialogue system is running.

4 Representing Dialogue Flow

All the concepts necessary for dialogue flow modelling as shown in Figure 1 are included in the dialogue flow tool. The dialogue structure is represented with a universal model. By this means various dialogue description languages can be automatically generated. The widely employed state machines are not able to decribe the specialties of spontaneous speech adequately. Instead we use a design method based on Harel's statecharts (Harel (1987)) which are capable of complex state modelling. We called the method *Dialog Statecharts*. The user specifies dialogue flow in the form of such dialogue states:

- The dialogue structure is specified by statechart diagrams. Every dialogue unit such as a subdialogue, a dialogue step (as in *Voice-XML*) or a turn can be represented as a state.
- States can be refined in several diagram layers. This supports a clearer dialogue structure.
- Every dialogue state can be linked to

  conditions These represent the constraints under which the state may be activated. This is especially useful for rule-based systems.

  prompts These model which prompt to speak when entering the state.

  grammars This indicates which grammars are to be activated in order to understand the next user utterance.

  actions They specify what the system should do after the user utterance has been analyzed, in error cases and so on.

  events They are used to handle exceptions such as missunderstandings or telephone line problems.

4.1 Dialogue Flow Modelling for the Rule-based System *Speedis*

Speedis is a typical generic conversational system based on concepts developed in the Sundial project (Peckham (1993)). It works with knowledge bases as shown in Figure 1. These various knowledge bases have to be adapted every time a new application is set up.
Speedis is a rule-based system (see Ehrlich (1999)) organized in so-called tasks. These represent subdialogues such as an *identification task* where a caller has to give his name and password. In our sense they are dialogue units and can be represented in the tool system by dialogue states.
Speedis only requires a rather abstract definition of the dialogue flow – the task hierarchy – as shown in Figure 3. When one task is active the system can only activate the direct daughter or mother task in this hierarchy. This is used to make dialogue handling easier and more consistent. *Speedis* has a default behavior which describes what to do in which situation, for example satisfying dialogue goals by choosing dialogue acts and a dialogue strategy (as described in Heisterkamp & McGlashan (1996)). The dialogue

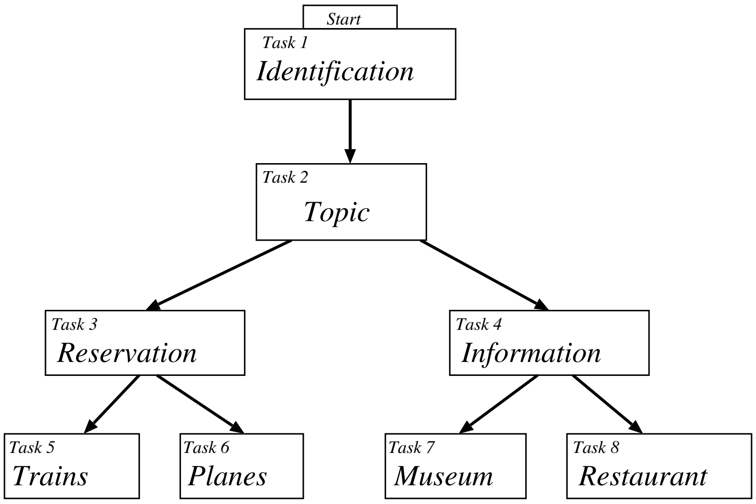

Figure 3: *Task structure graph. Each box models a task i.e. a subdialogue. The edges between the tasks show how tasks can follow each other. At any time it is possible to go back to a previous task and forward to a following task, but not to jump between siblings.*

acts are used to model system prompts. For every task the user has to specify which prompt to speak for which application parameter and dialogue act. For example confirming (dialogue act `confirm`) the name of the caller (application parameter `name`) with the prompt "Your name is <`name`>?" where <`name`> is the value of the parameter.[5]
New application domains for *Speedis* are currently developed by writing Prolog predicates as as function of a predefined set of dialogue acts.

Example – *Speedis* Prolog:

Given a dialogue application starting with an identification as shown in Figure 3. The predicate `task_type` defines a task named `Identification` with access to a database called `dbident`. The only possible successor task is `Topic`.

```
task_type('Identification',dbident,['Topic']).
```

The predicate `description` declares a list of application parameters (`name, password`) and a test predicate `test_authorized` which checks if a given tuple of these parameter values is a valid combination, and if the user is authorized to access the system.

```
description('Identification',
            [name, password],
            [test_authorized('Identification',
                         [name:N, password:P])]
            ).
```

[5]Currently *Speedis* does not use generation of system utterances from semantic structures.

Adaptation of the tool for *Speedis*

The following listing sums up the most important steps which have to be taken by the application developer using the tool system.[6] In order to support this, a special graphical editor was implemented – the task hierarchy editor, which is more adequate for specifying a hierarchy than the statechart editor described in Section 4.2. This editor includes the following functionality:

- Definition of tasks and the task structure; i.e. what does the system have to handle first and what comes next as shown in Figure 3. A start task and the successors of each task have to be defined. This is supported by the tool system by a task hierarchy editor with which diagrams such as in Figure 3 can directly be drawn. Consistency checks for *Speedis* make sure that a proper hierarchy is defined.
- Attachment of application parameters to the tasks; for example in the identification task the system must request the caller's name and password. For this step the tool system provides special masks as shown in Figure 4.
- Definition of the name of the database which is accessed in the task (special mask).
- Attachment of test predicates to the tasks, for example for checking if a user is authorized to access a database (special mask).
- Attachment of system prompts to the tasks dependent on dialogue acts (as shown in Figure 5). The prompt table can be pregenerated automatically as a function of these dialogue acts.

Once the developer has completed the specification he starts the generation of Prolog code, which specifies the application knowledge bases. Predicates as shown in the `task_type` example above are written into a file.

4.2 Dialogue Flow Modelling for State-based Systems

Many other dialogue systems do not handle dialogue flow by modelling 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.[7] Because conversational systems have a high complexity of states, state-transition models are not expressive enough to be a good means for dialogue flow modelling. The number of states is usually too large to be handled by a human with a common state-transition model (see Radtke (1999), Aust & Oerder (1995), van de Burgt et al. (1996)). Dialogue description languages such as *VoiceXML* and *TemicGDML* therefore try to find a trade-off between pure rule-based and pure state-transition-based dialogue design. They organize dialogues in dialogue units such as `dialog`, `step`, which in our sense are states. But here it is not necessary to model transitions explicitly. Instead there is a default logic which specifies when to activate which dialogue or step. for example given an active dialogue in

[6] The sequence given here is not obligatory. It is possible to design dialogues top-down as well as bottom-up.

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

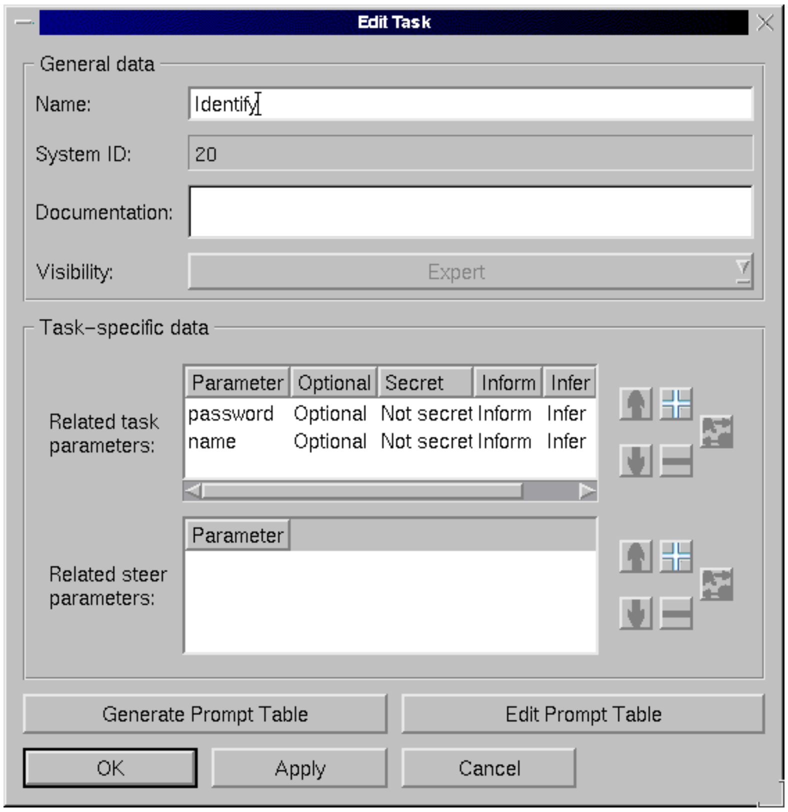

Figure 4: *Specification of tasks. This mask is only used when creating models for the Speedis dialogue system. Parameters can be linked to a task. Special Speedis specific attributes such as if a parameter is optional or obligatory can be set for the parameter. A prompt table can be generated as a function of the Speedis-specific dialogue acts. The prompts can be entered in this prompt table as shown in Figure 5.*

VoiceXML the first step (`field item`) is activated for which the application parameter is still undefined.

Example:

A dialogue named `weather_info` is defined. When it is activated a welcome prompt is played. The `block` item in this case refers to an action to be executed when entering the dialogue. Afterwards two fields are defined (`state, city`). The name of the field is always associated with an application parameter, which is called `field item variable` (for example `state` in the first field). When a field is activated the first action is normally to play a prompt. Then a grammar is activated which starts analyzing the user utterance.[8] The result found is automatically assigned to the given field item variable. The fields in the example also contain an event handler. If the user asks for help, a special help text is spoken. Once a field item variable is set successfully, the field cannot be reentered again as long as the dialogue is active. Effectively the fields

[8] Here Java Speech Grammar Format (JSGF) is used, Internet: see http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/)

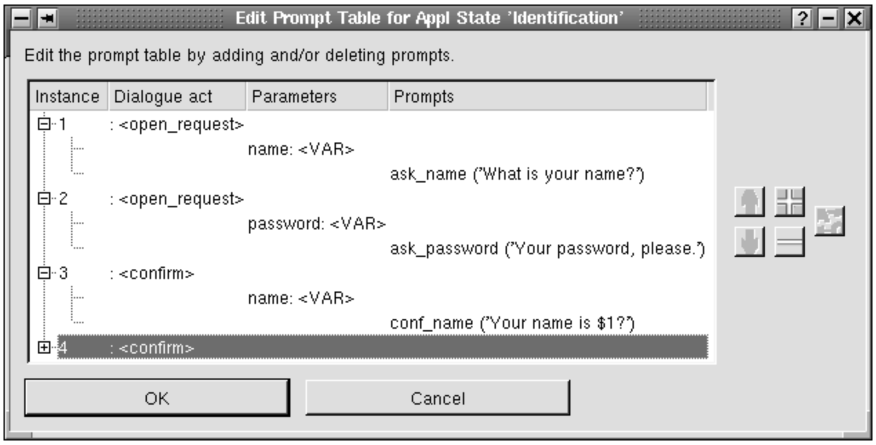

Figure 5: *Prompt table. This table is automatically pregenerated using the defined application parameters and combining them with dialogue acts. The user adds prompts such as "What is your name?" for the dialogue act* `open_request` *and the parameter* `name`. *The buttons on the right allow prompts to be added, deleted, and reordered.*

are activated one after the other in a given sequence. If mixed initiative is allowed,[9] the current user utterance determins in which sequence the fields are filled.

```
<form id="weather_info">
  <block>
    <prompt>Welcome to the weather information service.
    </prompt>
  </block>
  <field name="state">
    <prompt>What state?</prompt>
    <grammar src="state.gram" type="application/x-jsgf"/>
    <catch event="help">
      Please speak the state for which you want the weather.
    </catch>
  </field>
  <field name="city">
    <prompt>What city?</prompt>
    <grammar src="city.gram" type="application/x-jsgf"/>
    <catch event="help">
      Please speak the city for which you want the weather.
    </catch>
  </field>
  ...
</form>
```

As mentioned above, we want to model dialogues as structured states. A good approach for complex state modelling are statecharts as described in Harel (1987). They provide various means of abstraction such as concurrent states, state refinement, special event handling and action triggers.

[9]This is possible for *VoiceXML* and has to be explicitly specified by the dialogue developer

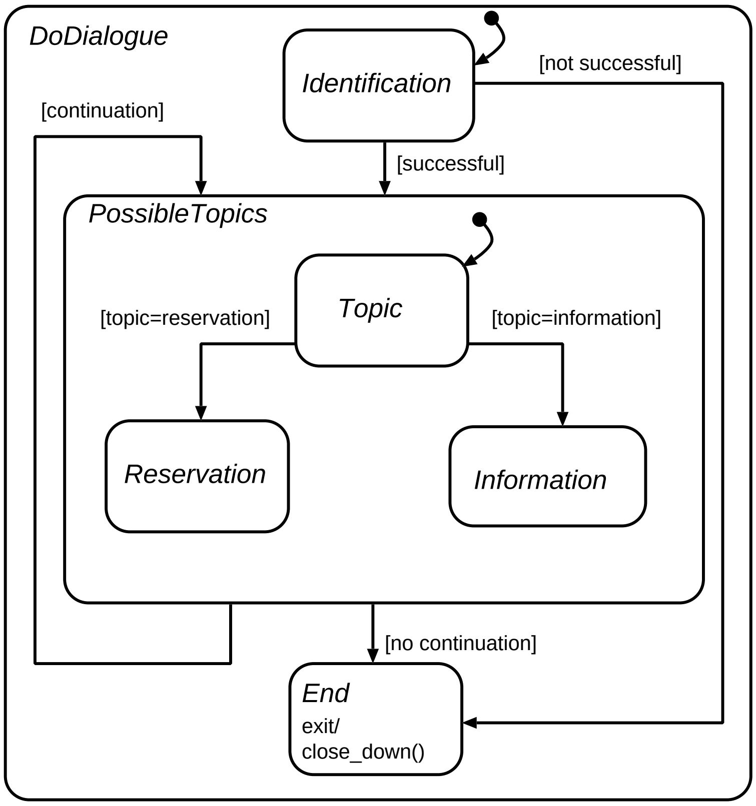

Figure 6: *Describing dialogue flow as a statechart. States are represented by boxes and can be embedded in one another. The state* `DoDialogue` *is an XOR state. This means that the system can only be in one of the states graphically inside at one point of time. The small rounded arrow at state* `Identification` *means that this is the default entry state for the* `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 made.*

By this means complex dialogue flow can be modelled rather intuitively. Figure 6 shows how the task data in Section 4.1 may be modelled in terms of states. The tasks are represented as complex states that are refined top-down to basic states where actions to be triggered are defined. The state `DoDialogue` is represented as an XOR state. This indicates that the system can only be in one of the states `Identification`, `PossibleTopics` or `End` at the same time.[10] In simple cases a task is represented by a basic state (`End`) which need not be refined any more. The complex 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 states, i.e. actions to be triggered when entering and when leaving the state. Events and conditions determin when a state is left and which transition is used. It is possible to describe actions and conditions common to several states or transitions by appropriate graphical means. For example any exit from states `Reservation` and `Information` in Figure 6 will lead to either state `End` or state `PossibleTopics` again, depending on the continuation

[10] A different graphical layout is used for concurrent states as shown in Figure 7.

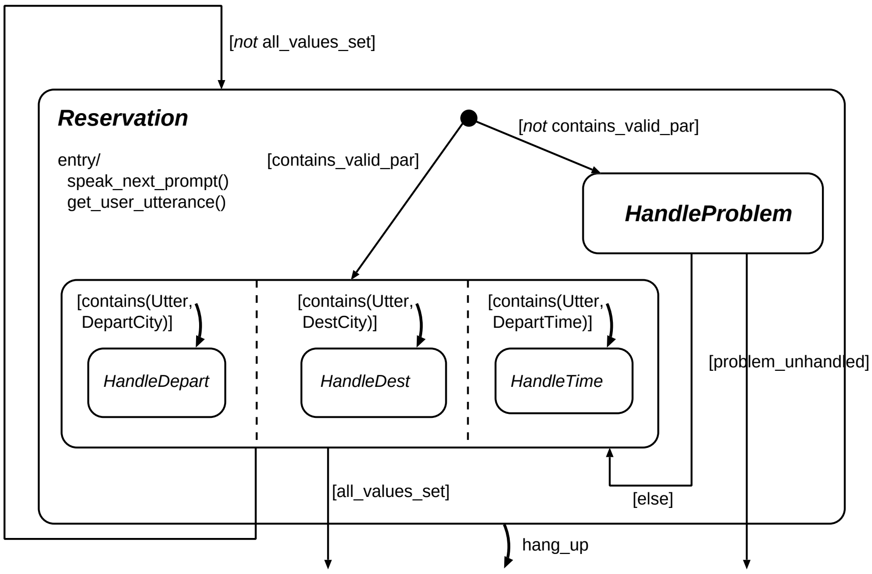

Figure 7: *Refining the state* `Reservation`*. The dialogue developer can add conditions and actions to be triggered in a given state. Entry actions are executed when entering the state, exit actions when leaving the state. The little black circle is the entry point of the state* `Reservation`*. States separated from one another by dotted lines are concurrent states. This means they can be handled in parallel and independently from each other. This state of type is used to represent mixed-initiative dialogues. In the given example the user may speak about the destination, departure city, and time in the same utterance. The* `hang_up` *arrow at the bottom of the figure shows an event (i.e. the caller hanging up) which would trigger an immediate exit from every substate of* `Reservation` *to some close-down state.*

flag.
This is only a short description of what can be done with statecharts. The figures are simplified for clarity. The statechart dialect implemented in the tool system is tailored to dialogue specification so that it is intuitive and easy to use.
For *TemicGDML* the statechart editor was adapted for the particular needs of this language. Figure 8 shows a screen shot of it.
The most important points which make *Dialog Statecharts* adequate for complex dialogue modelling and superior to simple state-transition-based modelling are:

- States can be refined. This allows for structured top-down or bottom-up design.
- States can be ordered and have entry conditions. When entering a state which contains substates the first substate is activated whose preconditions are fulfilled. So not all transitions between states have to be modelled explicitly. With this means it is possible to model rule-based systems such as *Speedis*, state-based systems such as *VoiceXML* dialogue systems and pure state-transition-based systems.
- Events can be specified which represent a break through the standard dialogue flow. This supports complex exception handling.

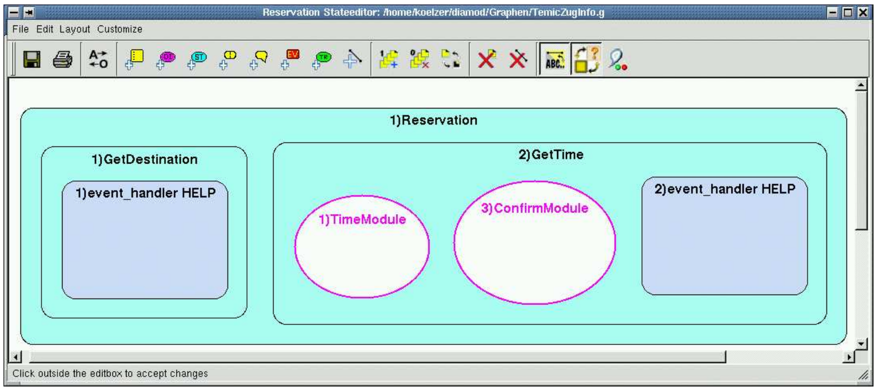

Figure 8: *Statechart editor adapted to TemicGDML. The dialogue structure is organized in one or more documents with one or more dialogues. Dialogues are composed of dialogue units such as steps and turns. There are special icons to generate such dialogue units. Every dialogue step should have an event handler for the event* `help` *which indicates that the user has asked for help. The tool system generates such event handlers automatically. The user can modify them by entering help prompts or delete them if they are not needed. The diagram shows the dialogue* `Reservation` *under development. The step* `GetTime` *calls two modules:* `TimeModule` *which requests the time from the user and* `ConfirmModule` *which confirms it after checking if this time is possible.*

– Concurrent states can be used to represent parallel dialogue flows. This can be used for mixed- and user-initiative dialogue modelling.

5 Generic versus Application Mode

The tool system can model what the system has to do in a given situation. Since it must work for different generic dialogue systems, the tool system must also consider the generic features of the dialogue system. This leads to a two-phase approach.

We distinguish between a generic expert mode (phase 1) and an application developer mode (phase 2). In the first phase an expert who knows everything about the internal structure of the generic dialogue system specifies generic data. He can specify function libraries or common dialogue patterns which can be used by the application developer in the second phase. Such a pattern is a description of how to handle dialogue parameters. For example first request a value, then try to confirm it; if there is an error, go to an error state, and so on. These generic data are used to make modelling easier for the application developer and to check consistency. They are also important for the code generation. If the target dialogue system does not provide any interfaces to its generic units (parameters, grammars, functions etc.), the generic phase will be left out.

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 in such a way that confusing data are hidden from the application developer. For example if the system works with dialogue acts and strategies it is not always necessary to present this to the user. Instead, the meaning of a particular dialogue act (`open_request`, `confirm`, `inform` etc. for

Speedis, see below) can be described with a help text. The user does not have to learn the dialogue description language itself. He is leaded by the graphical user interface of the tool system instead.
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, actions, grammars etc., while the application developer defines the application dependent data.

Overview: Phase one – generic mode:

Before any application can be specified the dialogue system expert describes the generic part of the system. In the case of *Speedis* only the generic system parameters must be specified. These are the dialogue acts and the sentence type, mentioned above, such as `statement` and `question`. They define the form of the next prompt.
The expert of the dialogue system defines all these generic parameters known by the dialogue system with the help of the tool system. With the given values the tool system calculates possible combinations such as `open_request - question` which represents the situation where a parameter is requested in the form of a question. Combinations not required by the dialogue system can be deleted by the expert. The expert adds some guide texts which comment the parameters and which will be shown to the application developer in phase two together with the calculated system states.

Overview: Phase two – application mode:

In order to specify a particular application, it is useful to follow a specific sequence of steps. Usually the developer will design the application in a top-down manner, first describing what topics the system can handle in a given sequence (task/state modelling), and then refining this structure. This is accomplished by describing the parameters, which are typical of the application, and the system prompts. The dialogue developer defines the application specific 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 supplied by the expert in phase one, the system calculates possible system states combining generic and application-dependent parameters such as `open_request - question - departure time`. The developer is asked to define the system prompt which can be used in this situation.

6 Graph-based Modelling

This section describes the basis for the implementation of the approach described in this paper. For a universal representation of all data, we chose a graph-based model (Ebert et al. (1996)). A generic graph class is defined which represents the concepts necessary for dialogue modelling in general. Every instance of this graph class, shortly spoken the graph, represents one application model.
In order to be able to handle different dialogue systems such as *Speedis* or a system of another company the generic graph class must be common enough for the required universal approach. This means that the graph class has to include every *concept* which is typical for dialogue systems.

Typical concepts in this context are dialogue states, prompts, parameters, and grammars. A vertex class is defined for every concept and an edge class for every kind of relation between concepts.
A syntax defines which classes of vertices may be attached to each other using which classes of edges. Parameters and states for example are represented by vertices in the graph. If a parameter is defined for a state, the two vertices are attached to each other by an edge. States can be related to other states in order to specify the state structure. Because the models are object-oriented it is possible to declare inheritance relations between the classes. This graph class can easily be enhanced: new vertex classes, attributes and consistency rules can be added. Thus dialogue knowledge bases for every conversational system, one likes, can be modelled, only making small adaptations.
The resulting graphs represent concrete application data. The state structure is also represented in this application graph. By examining the graph structure it is easy to inform an application developer of incomplete data such as a missing system prompt for the `confirm` state in the reservation dialogue.

6.1 Consistency checking

In order to be able to check consistency, constraints on the graphs are modelled, such as "make sure there is a system prompt for every system-initiative state".
Checking the consistency means that the object-oriented graph structure described above is analyzed. Constraints can be applied to the graph using GReQL (graph repository query language) – a special logic-based language for graph queries (see Ebert et al. (1998), Franzke (1996)). The constraints are examined by a special interpreter (Ebert et al. (1996)). By this means it is possible to guarantee that for example

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

In addition the model can be browsed using the query language. Queries can be for example:

- In which dialogue unit (for example form, field item in *VoiceXML*, task in *Speedis*) is the application parameter `DepartureTime` defined and where is it used?
- Which dialogue unit contains the prompt `AskDepartureTime` (every prompt has a name)?
- Show all dialogue units which can be reached from the dialogue unit `Reservation`.

This browsing facility is very useful for the application developer since it is easier to detect the causes of unwanted dialogue behavior.

6.2 Adaptation

The graph representation provides a universal data model which is independent of any particular dialogue system. Since the code generation is strictly separated from the models and is actually only the *interpretation* of a given graph. One dialogue model can be used for various dialogue systems. Instead of generating Prolog knowledge bases for *Speedis* for example, which describe the tasks and the task structure, it is also possible to generate any other code such as *VoiceXML*.
The tool system can be adapted to different dialogue systems by adding some concepts and attributes, if necessary, tailoring the graphical user interface, writing new code generation routines, and adding the required GReQL constraints. Mainly appropriate constraints and code generation routines have to be added when the tool is adapted to another dialogue system.

6.3 Model Transformation

Using the system-specific constraints and the different interpretation routines, it is possible to automatically transform a model which was originally developed for dialogue system A (for example the research system) to dialogue system B (for example a simple state-based command-and-control system). Usually there are concepts that are known to system A and not to system B. Therefore automatic transformation will usually not be complete. Since the completeness of the individual dialogue models is described by GReQL predicates, the tool system is able to inform an application developer of problems.
For example one must find a way to map the *Speedis* tasks and dialogue acts to *VoiceXML* forms and field items. This is realized by automatically adding missing information to a model. When a *Speedis* model is transformed into a *VoiceXML* model, one can generate one form for every task, one field for every task parameter and one turn[11] for every dialogue act.
Because not all information needed can be transformed automatically, the user of the tool system is warned where problems were detected or where he has to add information manually. It is not possible, for example, to decide automatically which field in a *VoiceXML* model should be mapped to which dialogue acts in *Speedis* when transforming a model in *VoiceXML* to *Speedis*.

7 Summary and Outlook

This paper introduced a universal approach for the specification of applications for generic conversational systems. The approach is implemented in a tool system. Details on how dialogues are modelled with the tool system and the proof that the tool system can be adapted to different kinds of dialogue systems, are given in the PhD thesis Kölzer (2002). The focus of the approach is on dialogue flow modelling. The tool system is especically useful for dialogue systems with a predictable dialogue structure. The most important features of the tool system are:

- A universal knowledge representation is used which is independent of any particular dialogue system.

[11] A *VoiceXML* field item can consist of several turns which are indicated by system prompts, usually combined with a grammar and appropriate actions.

- A model for one dialogue system can be transformed into a model for another dialogue system.
- Various aspects of dialogue with various views of the data can be modelled.
- The consistency of the models can be checked automatically.
- Additional knowledge bases can be easily adapted.

Summing up, the tool system currently supports

- task and task structure modelling with automatic prompt table generation
- state-based modelling using *Dialog Statecharts* with events, actions and conditions
- modelling of generic system parameters and application dependent parameters, and their appropriate domain types (records, enumeration types, lists, etc.)
- attaching system prompts for various natural languages to the modelled tasks or states.
- code generation of Prolog for *Speedis* and *TemicGDML*.

The task modelling and the statechart-based approach described in Section 4 are completely implemented. The transformation between different models and the re-use support of models is still under development. The approach is implemented in C++ using graphs and GReQL constraints as described in Section 6.
Up till now the tool system was adapted for *Speedis* and *TemicGDML*. It was shown that this approach is universal enough to represent these different kinds of modelling. The system is able to tell the user which data are still incomplete or inconsistent. This could be extended with guidelines of dialogue management as suggested by Bernsen et al. (1998). Our further aim is to integrate more knowledge bases such as phonetic and linguistic structures. A prototyper to immediately check the consequences of the modifications made, is also planned. The principles, described in this paper, have not yet been adapted to other kinds of dialogue systems, for example translation systems such as Verbmobil (see Wahlster et al. (2000)). This would make an interesting topic of research.
When implementing the tool system, we found that the most costly part is the development of the graphical user interface. If working with a graphical dialogue modelling tool shall be a benefit compared to using conventional dialogue description or programming languages, the graphical user interface must be really sophisticated and ergonomic. Up to date it was not possible to evaluate the benefits of the tool because of lacks in the graphical user interface. Therefore we are still improving this part of the system. Statecharts as a graphical approach, however, are very expressive and the tool prototype is very promising, so that we are confident that the approach is worth being pursued further.

References

AT&T et al. VoiceXML. 2000. `http://www.voicexml.org/`

Aust H., Oerder M. Dialogue control in automatic inquiry systems. In *ESCA Workshop on Spoken Dialogue Systems*, pp. 121–124. Vigso, Denmark, 1995

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 *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

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

Cole R. Tools for research and education in speech science. In *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 *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

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

Kölzer A. Methodical specification of knowledge bases for a speech dialogue system. In *International Workshop Speech and Computer (SPECOM) 1998*, pp. 97–100. Russian Academy of Sciences, St.Petersburg, Russia, 1998

Kölzer A. DiaMod - Ein Werkzeugsystem zur Modellierung natürlichsprachlicher Dialoge. Ph.D. thesis, University of Koblenz–Landau, Dept. Koblenz, Mensch & Buch Verlag, Berlin, 2002

McGlashan S. Multimodal dialogue management. In *Dialogue Management in Natural Language Systems, Proc. of the 11th Twente Workshop on Language Technology*. Enschede, Niederlande, 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

Radtke T. Beschreibungssprachen für Dialogsysteme. Master's thesis, Technische Universität Ilmenau, 1999

Searle J.R. Speech acts: An essay in the philosophy of language. Tech. rep., Cambridge University Press, 1969

van de Burgt S., Kloosterman H., Andernach T., et al. Building dialogue systems that sell. In *Natural Language Processing and Industrial Applications*, pp. 41–46. New Brunswick, 1996

Wahlster et al. The VERBMOBIL Project. World Wide Web, 2000. `http://verbmobil.dfki.de/`