An Integrated Development Environment for Spoken Dialogue Systems

Matthias Denecke
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
denecke@cs.cmu.edu

Abstract

Development environments for spoken dialogue processing systems are of particular interest because the turn-around time for a dialogue system is high while at the same time a considerable amount of components can be reused with little or no modifications. We describe an Integrated Development Environment (IDE) for spoken dialogue systems. The IDE allows application designers to interactively specify reusable building blocks called *dialogue packages* for dialogue systems. Each dialogue package consists of an assembly of data sources, including an object-oriented domain model, a task model and grammars. We show how the dialogue packages can be specified through a graphical user interface with the help of a wizard.

1 Introduction

The specification and design of interactive spoken language systems has become the focus of research recently. Partly fueled by the increasing demand of spoken language applications and telephony-based services, the deployment of development environments has increased. At the time of writing, at least three main types of dialogue tools can be distinguished. One approach to development environments consists of graphical editors for Finite State Automata (FSA) [Sutton et al, 1996], [Cole, 1999]. These systems equate a dialogue with a possible path from the start state to one of the accepting states. Possible actions of the application are specified by annotations on states or arcs or both. Besides relying on a dialogue model that has been considered as problematic in the past, finite-state automata based dialogue editors do not exploit the desirable characteristics of software engineering, such as reusability and orthogonality of the components. For example, recovery strategies need to be duplicated for each state in which they should be applied. Moreover, they require a system designer to anticipate every single possible path through the system, a fact that leads to an explosion of dialogue states.

Another approach to development environments emphasizes reusability of the domain model over graphical design interfaces. Here, object-oriented features of the underlying programming language such as JAVA or C++ are used to design a class hierarchy of *speech objects* or *dialogue modules* that can be assembled and re-assembled for new applications. These modules are often used for basic data types, such as *date, time, credit card numbers*, etc. This approach has proven its practicability in numerous commercial applications. Since the modules can be reused, this is an improvement over finite-state based dialogue machines. However, fine tuning of recovery strategies requires separate fine-tuning in each module. Moreover, the dialogue flow is partly defined by an FSA whose nodes consist of the dialogue modules. When a node is reached, the dialogue module determines the dialogue control until it gives up control and an adjacent arc is traversed.

A third approach consists in designing a library of reusable dialogue strategies based on the observation that the behavior of a dialogue manager should be predictable in similar situations across several domains. Araki et al [Araki et al, 1999] proposed a library of dialogue strategies to be reused. Koelzer [Koelzer, 1999] proposed a reusable dialogue system architecture based on specifications of knowledge sources for the different components.

In this paper, we identify knowledge sources such as grammars, task models and database conversion rules, that characterize our dialogue manager for a given application. Each of the knowledge sources can be composed of smaller, modular knowledge sources. A collection of these knowledge source modules, called a *dialogue package*, specifies a subdomain of a dialogue application. We borrow techniques known from object oriented programming languages to combine partial specifications of knowledge sources to form the knowledge sources for a new application. The specifications are mostly declarative rather than procedural, leaving to the dialogue manager the decision how best to interpret them in the context of the dialogue. We describe the implementation of a wizard-based integrated development environment called Chapeau Clac that allows the specification of the knowledge sources, their integration and testing.

2 The Architecture of the IDE and the Dialogue System

2.1 The Architecture of the Dialogue System

The dialogue manager makes use of different knowledge sources. First, it contains a set of task descriptions or task models. A task description can be considered as a form to be filled in through the dialogue, together with constraints stating the minimum amount of information necessary to execute the task. The dialogue strategy is specified in a declarative programming language similar to PROLOG that can be easily adapted to the task at hand should the need arise.

The state of the dialogue system at any given time is determined implicitly by the relations of the forms with the information available in the discourse at that time. For example, a task description whose constraints are inconsistent with information in the discourse can not be a description of the intent of the user. The elements the forms can be populated with are descriptions of objects, actions and properties of objects and actions drawn from a domain model. The domain model can loosely be compared to a class hierarchy in object oriented programming languages. In addition to task model and domain model, the dialogue manager uses data base conversion rules to generate SQL queries and to transform the result sets. As the domain model is dependent on the particular speech application, it belongs to the knowledge sources to be specified through the wizard.

As the semantics of the utterances are expressed in terms of the domain model, we need to provide a mechanism to translate the text input from the speech recognizer into a canonical representation. Attributed grammar rules provide transformation between text input and semantic representations.

The place of the dialogue manager in the system is similar in spirit to, but different in functionality from, the design of a Graphical User Interface for a back-end application. In the case of the GUI, the design of windows, dialog boxes and menus is independent from the design of the back-end application that uses these graphical display elements. Similarly, in our approach, the design of dialogue grammars, dialogue goals and domain models is independent of the design of the back-end application. As in GUIs, the back-end application is notified of manipulations through events and callback functions. This approach separates clearly the speech user interface from the back-end application. The callbacks and events constitute one integration point between speech user interface and back-end application whose form and content needs to be specified for each new speech application.

It should be noted, however, that the analogy between graphical and speech user interfaces ends here. Reference in GUIs is extensional. For example, the click of the button or a menu, together with the state of the application and the focus, determines the intended action. In spoken dialogue systems, the need to resolve reference of noun phrases or ellipsis forces us to provide one more integration point with the back-end application in order to allow database retrieval.

Consequently, we argue that a dialogue manager for a given speech application can be characterized by the specification of four knowledge sources, namely (i) the domain model to characterize the semantic content of the utterances, (ii) the conversion from the text input into a canonical semantic representation and vice versa, (iii) the task model to describe the event stream from the speech user interface to the back-end application, and (iv) the conversion from semantic representation into database retrieval requests. Figure 1 shows the place of the knowledge sources in the dialogue manager. As can be seen, the knowledge sources (ii) to (iv) encapsulate entirely the dialogue manager from the remaining components of the system.

Note that we make no assumptions as to how the dialogue manager might make use of these knowledge sources. In particular, we do not make any assumptions as to how the dialogue strategy might determine the actions of the dialogue manager. As long as the provided knowledge sources are sufficient for the dialogue manager to determine its actions, the dialogue manager could implement a simple information seeking dialogue system or a more sophisticated system based on speech act or discourse theories.

All four knowledge sources can be modularized more or less straightforwardly. The domain model can be composed of different subdomain [Denecke and Waibel, 1999]; new concepts may use multiple inheritance of abstract base types. Grammar rules containing generic semantic information can be specialized and adapted to the given domain. Database conversion and dialogue goal specification modules may simply be joined; but see section 6 for potential problems. It is the task of the wizard to help the user in specifying and reusing these knowledge sources.

2.2 Requirements for the IDE

The requirements for the IDE's functionality comprise three main items. First, it should guide the application designer to specify and modify the spoken language part of an entire application through a GUI. The data sources relevant for the spoken language interface currently include grammars, domain model, data bases, task model and input/output channels. Moreover, conversions back and forth

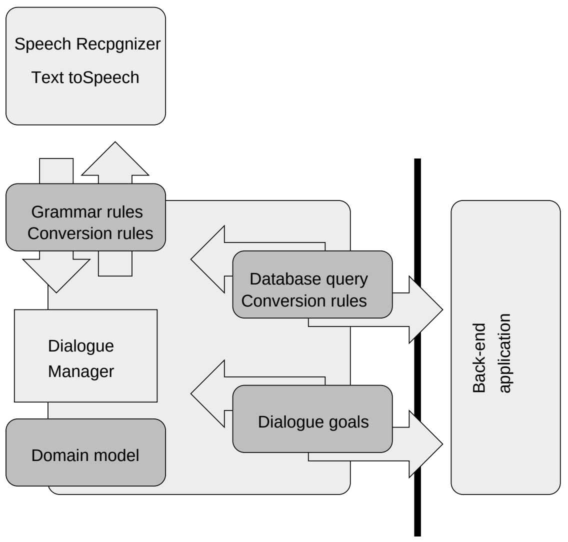


Figure 1: The place of the knowledge sources in the system architecture.

between semantic representation of utterances and database queries and results on one hand and text on the other need to be specified. The object model of the data sources used in the dialogue system is shown in figure 2. Second, the IDE should support a developer by adapting and modifying the existing dialogue strategy through the usual debugging tools such as tracer, walk through, call stacks, breakpoints and variable dumps. Third, it should support an application designer in testing the final application using batch tests and single utterance tests.

In addition, since experienced users may obtain results faster using a keyboard rather than a wizard interface, the system designer should be able switch between a standard text editor and the wizard interface at any time in the design process. Surprisingly, this design requirement had a more thorough impact on the layout of the system implementation than anticipated. For each data source to be specified, we need two classes that implement the data source: one class implementing the data source itself, and a second class implementing a description of the data source. The second class consists only of primitive data types such as strings and integers that can easily be manipulated by a wizard interface and can also be easily parsed from a file. When the final data sources are instantiated, the constructor of the data source, taking a description as its only argument, creates the data source according to the specification.

In addition to the decoupling of the GUI with the dialogue system itself, the description objects also introduce an additional level of abstraction that allows the replacement of similar data source implementations (such as different grammar formalisms as required by JSAPI and SAPI). Figure 3 depicts the relationship between the different entities.

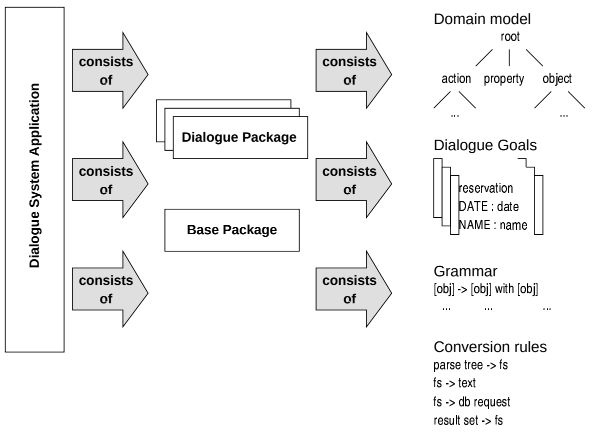


Figure 2: The object model for the dialogue system.

The data source specifications are organized in a modular fashion in dialogue packages. Each dialogue package consists of at least one and possibly all the mentioned data sources. A final application is then composed of several dialogue packages. In order to avoid naming conflicts, each dialogue package introduces its own namespace. As an example of a dialogue package, consider the task of a hotel reservation. The implementation of the hotel reservation package may contain several tasks, such as calculating the price of a stay or displaying the hotel's location on a map. The interface between the implementation of the package and the dialogue system is regulated by the knowledge sources in the package description. For example, the hotel reservation package may consist of several concepts such as hotel, room, reservation, and all possible actions that go with it. The dialogue system notifies the dialogue package in case an event related to the package occurs. It is then the responsibility of the dialogue package to process the event properly.

Similar to class libraries in object-oriented programming languages, the dialogue packages may be reused in different applications. The hotel reservation package may be reused in an information booth application (which uses another dialogue package concurrently offering services related to current events) and in a travel agency setting (which, in turn, allows the user to book flights through the use of a third dialogue package). The intention of this level of granularity is it to have each package cover all aspects of an entire subdomain.

3 The Specifications

The IDE offers a wizard-style GUI to specify the data sources described above. The wizard guides the user through the process of specifying a dialogue package. In this section, we describe the steps the

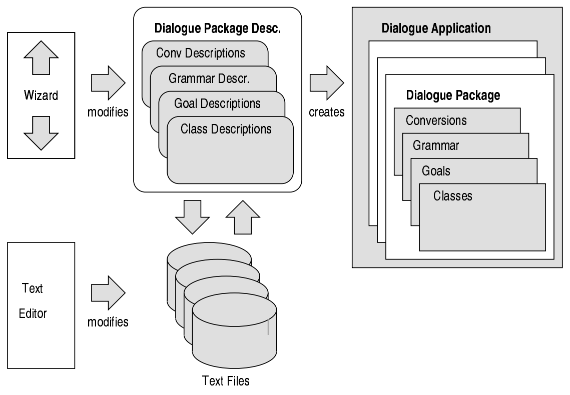


Figure 3: Descriptions specify the content of the data sources to be used in the application. Descriptions of data sources can be created and modified using the IDE or a plain text editor.

wizard guides a user through for each dialogue package. In each step, one of the four knowledge sources described above is interactively specified.

3.1 The Domain Model Specification

The domain model employed in the dialogue system uses a simple *class hierarchy*. A class hierarchy is a type hierarchy [Carpenter, 1992] extended by method descriptions. Class specifications may contain variables (whose type is a class from the ontology) and methods (whose arguments are classes from the ontology). In addition, class specifications may be related through multiple inheritance. While in conventional object-oriented design, objects in the domain correspond to classes, actions of the objects correspond to methods, and properties correspond to variables, we chose to model these elements by classes. First, this allows us to uniformly express mappings from noun phrases, verbal phrases and adjuncts to classes (see section 3.3). Second, any constituent of a spoken utterance may be underspecified. Our approach allows us to select through the inheritance mechanism the most specific class from the ontology whose informational content can be warranted in the absence of complete information.

A method specification does not implement any particular behavior of the class it belongs to. Rather, it can be seen as a constraint specification that generates an event to the back-end application as soon as it is satisfied. It is then the task of the back-end application to carry out the functionality associated with the method. Consider a class obj_displayable with an associated method display() and the constraint string < obj_displayable.name,int < obj_displayable.x,int < obj_displayable.y (read: the variable obj_displayable.name contains more information than the fact that it is a string,i.e. it is instantiated). As soon as the position and the name of the object become known to the dialogue system (e.g. through database retrieval), an event is generated and sent to the implementation of the dialogue package, providing class information as well as the values of the three variables expressed in the constraint. Should a description of an object refer ambiguously, an event is generated for each retrieved object that verifies the constraint. Not only does this approach provide a declarative way of specifying behavior and abstract over the form of the dialogue, it also decouples the natural language understanding component from the application itself in a natural way.

This form of method invocation interacts nicely with another characteristic of our approach to object-oriented design. While traditionally an instance of a class is an object, in dialogue processing an instance of a class can only be a (possibly incomplete) description of an object. Necessary information for object instantiation may be missing and can only be acquired through dialogue. Since descriptions of objects do not need to refer uniquely to objects, procedural method invocations become more complicated. For this reason, we chose the declarative approach to method invocation over a procedural one.

The domain model is the backbone of the specification process. Not only does the dialogue manager use the domain model for inferences at runtime, but other knowledge sources such as grammar and database access specifications can partly be derived from the domain model. Moreover, the type information helps to restrict choices and to verify the consistency of the specification at the design stage. Consequently, the design of the domain model is the first step in the design process. This is in contrast to many other design tools whose first step is to design the information flow of the dialogue.

Example

By way of an example, we describe the design of a fast food order service. The service offers pizza with different toppings and different pasta with different sauces. Pizzas and pasta come in different sizes. The price of the items varies as a function of the size and the toppings or the sauce, respectively. The user should be able to query properties of the items, such as price, add and remove items from a virtual shopping list, and finalize the purchase. We introduce one abstract base type *obj_priceable* with the real valued feature BASEPRICE and a feature SIZE, the value being one of *small, medium, large.* As toppings and sauces may not be purchased separately, a second abstract base type *obj_buyable*, inheriting from *obj_priceable*, allows to distinguish the dishes from its ingredients. *obj_buyable* serves then as a base type for *obj_pizza* and *obj_pasta* while *obj_topping* and *obj_sauce* are derived from *obj_priceable.* As the calculation of the price is a task specific to the back-end application, we introduce a method

$$obj\_buyable.calcprice : real \times set(real) \times real$$

with the constraints

$$\begin{array}{lcl} obj\_buyable.baseprice & > & real, \\ obj\_buyable.ingredients.\{baseprice\} & > & real, \\ obj\_buyable.price & \geq & real \end{array}$$

As soon as an *obj_buyable* whose values of the BASEPRICE features is defined appears in the discourse, all values are passed on to the back-end application. It is the task of the back-end application to determine the price of the dish and to return the result in the third argument of the method description. Since the third argument is described by the constraint $obj\_buyable.price \geq real$ (a constraint that is always satisfied due to the feature definition), the dialogue manager places the result returned from the back-end application at the appropriate place in the feature structure.

3.2 The Dialogue Goal Specifications

The application designer needs to design a description of a dialogue goal for each task the back-end system can execute. A dialogue goal can be considered as the description of a form that is filled out through the spoken dialogue with the system [Papineni et al, 1999]. The goal description consists of a typed feature structure [Carpenter, 1992] whose types are drawn from the class hierarchy designed in step 3.1. It serves as an informational lower bound, guaranteeing that the back-end application is notified if and only if the information acquired through the dialogue is at least as specific as the specification in the dialogue goal.

Note that the dialogue goal specification does not make any assumptions as to how this information is acquired, nor as to how the acquired information is to be processed. Thus, the dialogue goals form the specification of a task model that is orthogonal to any dialogue strategy specification and independent from the implementation of the back-end system. Furthermore, it should be noted that the specification of dialogue goals in typed feature structures does not restrict the dialogue strategy to be a simple form filling strategy. Rather, the dialogue goal specification is an encapsulation of a method invocation which, when triggered, causes the back-end application to do what the user intended the system to do. The assumptions made here are similar to those in the general PARADISE framework [Walker et al, 1997] for dialogue evaluation where the task model for dialogue managers is equally described in attribute value matrices.

Example (continued)

We continue the fast food service example. We concentrate on the dialogue goals relevant to the pizza and pasta objects, as we assume that we have recourse to a dialogue package *Shopping Cart* that defines the knowledge sources relevant to the virtual shopping list. We thus need to introduce only one dialogue goal, namely the one allowing the user to seek information on the buyable objects.

3.3 The Grammar Specification

It is the task of the grammar specification to map an utterance onto a feature structure. We use the robust spoken language parser described in [Gavalda and Waibel, 1998] for context free parsing. In addition to the grammar rule specification, a set of conversion rules needs to be created to declare the way a parse tree is mapped onto a semantic representation. A parse tree generated by this parser contains semantic concepts as nonterminal symbols.

Grammar rules can be either lexical rules, i.e. rules whose right hand side consists entirely of lexical entries, or phrasal rules, i.e. rules whose right hand side consists entirely of nonterminal symbols. A grammar nonterminal symbol consists of three part $\langle sem, syn_{maj}, syn_{min} \rangle$ where sem is a type drawn from the type hierarchy, syn_{maj} is the name of the major syntactic category, currently one of N, V, A or their phrasal projections NP, AP, VP, and syn_{min} is the name of their minor syntactic category. Minor categories depend on the major categories. For example, minor categories for adjectives are *predicative, comparative* and *superlative.* The purpose of separating syntactic and semantic information in the nonterminal symbols is threefold. First, it allows the technique of multiple inheritance to be applied during grammar design and parsing. For example, a nonterminal symbol $\langle sem, syn_{maj}, syn_{min} \rangle$ might be expanded by a rule with a left hand symbol $\langle sem', syn_{maj}, syn_{min} \rangle$, provided that sem subsumes sem' in the type hierarchy. Second, it provides more information to compare nonterminal symbols during parsing than plain slot names. Third, the semantic information is helpful in ensuring the semantic constructions associated with the grammar rules is well-typed. Please refer to [Denecke, 2000] for more information on the first two points. In this paper, we will concentrate on the third point as it is relevant to the design of the wizard interface.

As the syntactic structure of the input sentences might vary, it is not sufficient to rely on the names of the concept to extract the meaning of the utterances. Rather, we pursue an approach that is resembles the one found in *attributed grammars* used in compiler construction or *Montague grammars* in that the grammar rules contain an annotation describing how to construct the semantics. Consider a rule

$$\langle sem, syn_{maj}, syn_{min}\rangle \rightarrow \langle sem^1, syn^1_{maj}, syn^1_{min}\rangle \ldots \langle sem^n, syn^n_{maj}, syn^n_{min}\rangle$$

for an expression describing an object of type *sem*. We assume by induction that the constituents described by $\langle sem^i, syn^i_{maj}, syn^i_{min}\rangle$ are expressions describing objects of type sem^i. As the semantic representation of the phrases covered by $\langle sem, syn_{maj}, syn_{min}\rangle$ needs to be a feature structure of type *sem*, all that remains to be done is to define n feature paths $\pi^i = f^i_1 \ldots f^i_{m_i}$ for each of the right hand symbols such that $sem.\pi^i$ is allowable according to the type hierarchy specification and $sem.\pi^i$ takes a value that is compatible with sem^i. This sort of type information restricts the number of possible feature paths. Only allowable feature paths are offered through the wizard interface so as to ensure that the resulting structures correspond to the domain model.

As an application designer sets out to develop a new application, he can take recourse to a base ontology and a base grammar. We make the assumptions that the base grammar and the base ontology already cover a wide variety of surface forms of the input sentences. The application designer simply needs to provide the lexical rules and to specialize existing generic rules. The nonterminals in the base grammar do not contain any domain-specific semantic information, but only rather general information such as object, or location. It is then only necessary for the application designer to specialize the predefined rules and to provide the "ontological" part of the grammar.

Robust Parsing

The fact that syntactic and semantic information are represented separately in the nonterminal symbols enables a more fine grained comparison of nonterminal symbols. This can be exploited for robust parsing. For example, two symbols differing only in their minor syntactic category could be matched, with an appropriate penalty, to allow for robust parsing. At the time of writing, a standard context free grammar to be used in the parser is created from the rule specifications. Additional rules covering close matches are created for robustness.

The well-typed constraint imposed on the rules by the conversion information does not render the parsing more brittle as robustness is achieved by loosely matching the input and by a fuzzy matching of nonterminal symbols. The form of the rules can be expected to be unaltered.

Clarification Questions

The need to generate a clarification question arises in the case of ambiguous reference. The dialogue manager determines discriminating information of a set of representations using a technique described in [Denecke and Waibel, 1997]. As the grammar rules contain syntactic and semantic information, they are reversible to a limited extent. Thus, the rules can be used to generate phrases describing the discriminating information.

Example (continued)

In the fast food application, phrases such as *a pizza with salami* or *tortellini with cream sauce* need to be covered. The generic grammar provides an abstract rule of the form $\langle obj, N\rangle \rightarrow \langle obj, N\rangle\langle p, with\rangle\langle obj, N\rangle$ which is specialized to

$\langle obj\_pizza, N\rangle \rightarrow$
$\langle obj\_pizza, N\rangle\langle p, with\rangle\langle obj\_topping, N\rangle$ and
$\langle obj\_pasta, N\rangle \rightarrow$
$\langle obj\_pasta, N\rangle\langle p, with\rangle\langle obj\_sauce, N\rangle$

respectively (minor categories are omitted for clarity). Each nonterminal symbol on the right hand side is assigned a part of the resulting semantic representation. The first right hand symbol gets assigned an empty feature path, since its relation to the left hand symbol relation needs to be an is − a relation. The semantics of the second nonterminal symbol is ignored. We concentrate on the third nonterminal symbol in both rules. In this example, TOPPINGS and SAUCE, respectively, are the only feature paths that express an is − part − of relation between *obj_pizza* and *obj_toppings*, and *obj_pasta* and *obj_sauce*, respectively. This yields the following annotated rules.

$\langle obj\_pizza, N\rangle \rightarrow$
$\langle obj\_pizza, N\rangle$ $[obj\_pizza]$
$\langle p, with\rangle$
$\langle obj\_topping, N\rangle$ $[obj\_pizza$ TOP'S $obj\_topping]$
and
$\langle obj\_pasta, N\rangle \rightarrow$
$\langle obj\_pasta, N\rangle$ $[obj\_pasta]$
$\langle p, with\rangle$
$\langle obj\_sauce, N\rangle$ $[obj\_pasta$ SAUCE $obj\_sauce]$

The type information serves to restrict the number of admissible feature paths for the semantic construction. Only admissible feature paths are offered as choices in the wizard, thus reducing the burden on the grammar designer. Had the designer erroneously specialized the abstract rule to

$\langle obj\_pizza, N\rangle \rightarrow$
$\langle obj\_pizza, N\rangle\langle p, with\rangle\langle obj\_sauce, N\rangle$

the wizard would not be able to offer any consistent semantic interpretation, thus uncovering inconsistencies in the specification early in the design process.

3.4 The Database Access Conversion Rules

The IDE provides an interface to SQL databases. The tables of SQL databases are self-describing in that the form, the datatypes and the relations between the tables can be determined at run-time. If the user wishes to create a new database for some of the objects specified in step 3.1, then the corresponding SQL data definition query is generated from the domain model automatically. In this case, there is a one-to-one relationship between a type description and a table, and conversion rules are created automatically. However, it is more probable that application designers are faced with the design requirement that existing databases be reused. In this case, the wizard interface allows the user to establish a conversion between features and entries in tables. Please note that in this case there is not necessarily a one-to-one correspondence between type descriptions and tables. Here, the databases consist typically of multiple tables T that are linked via primary keys E.

The dialogue strategy executes database requests at appropriate times during the dialogue with the goal being to fill in missing feature values. It is then the responsibility of the database manager to determine the database that needs to be queried and to generate the query itself based on the information available. This is done in the following manner. First, by examining the partly filled form and scanning the conversion rules, the set of tables $t_1^1, \ldots, t_n^1 \in T_1$ for which keys are given are determined. Then, we need to obtain all pairs of primary keys that establish the links between the tables in T_1. However, a link between two tables can be given through a chain of tables not all of which need to be in T_1. Thus, we need to determine the set $t_1^2, \ldots, t_m^2 \in T_2$ of all tables involved in the query by calculating a minimal subtree $\langle T_2, E_2 \rangle$ of the graph $\langle T, E \rangle$ that spans over all tables from T_1. The information in T_1, T_2 and E_2 together with the partially filled form is then sufficient to arrive at a query of the form

$$\begin{array}{l}
\textsc{Select} \\
\quad \begin{array}{lll} t_1^2.e_1, & \cdots, & t_1^2.e_{n_1} \\ \vdots & & \vdots \\ t_m^2.e_1, & \cdots, & t_m^2.e_{n_m} \end{array} \\
\textsc{From}\ t_1^2, \ldots, t_m^2 \\
\textsc{Where} \\
\quad \begin{array}{lll} t_1^1.e_1 & = & v_1\,\textsc{And} \\ \vdots & & \vdots \\ t_1^n.e_p & = & v_p\,\textsc{And} \\ t_i^2.e_k & = & t_j^2.e_l\,\textsc{And} \quad \forall \langle t_i^2.e_k, t_j^2.e_l \rangle \in E_2 \end{array}
\end{array}$$

where the v_i are the values provided by the partly filled form. The result set returned from the query engine is then converted back to feature structures corresponding to the domain model. There exist additional constraints on the size of the result set that are verified before converting in order to avoid time consuming conversion operations in the case of large result sets.

Example (continued)

In the fast food application, the data is stored in four tables, namely *pizza, pasta, sauces* and *toppings*. The tables are assigned to the types *obj_pizza, obj_pasta, obj_sauces* and *obj_toppings* in the same order; additional assignments exist between feature names and table entries. The tables *pizza* and *toppings*, and *pasta* and *sauces*, respectively, are linked in the database through unique IDs. As the relationships between the tables is is-part-of, the links are assigned the path prefixes SAUCES and TOPPINGS. A feature structure

$$\begin{bmatrix} \textit{obj\_pasta} & \\ \textsc{sauce} & \textit{obj\_cheesesauce} \end{bmatrix}$$

is then converted to the query

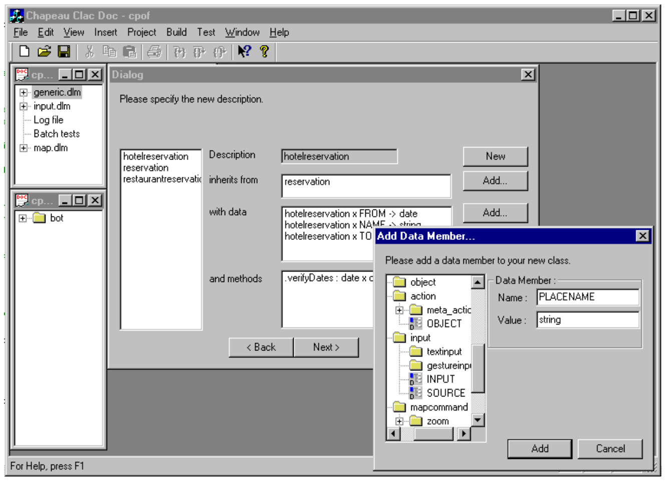
$$\begin{array}{l}
\textsc{Select} \\
\quad \textit{pasta.name}, \textit{pasta.baseprice}, \textit{pasta.size}, \\
\quad \textit{sauces.name}, \textit{sauces.baseprice} \\
\textsc{From}\ \textit{pasta}, \textit{sauces} \\
\textsc{Where} \\
\quad \textit{sauces} = \textit{cheesesauce}\,\textsc{And} \\
\quad \textit{pasta.ID} = \textit{sauce.ID}
\end{array}$$

Using the same conversion rules backwards, underspecified feature structures are constructed from the resulting table. Note that the database as a relational database cannot express inheritance relationships. This means that although *tortellini, greennoodles* and *spaghetti* all are derived from *pasta*, a query containing the constraint *pasta.name* = "*pasta*" would return the empty set, as the database does not know about the inheritance relationships. For this reason, the conversion rules associated with the table entries also contain a type restricting the constraint generation. Only types that are more specific than the restriction are taken into consideration for query generation. In this example, the types taken into consideration for query generation would need to be more specific than *obj_pasta*. This is to ensure extensionality for database access. Alternatively, one could employ extensional feature structures as described by Carpenter [Carpenter, 1992] and make sure that only extensional types are used for queries.

3.5 Interfacing the Wizard with the Knowledge Sources

A wizard-style GUI guides the application designer through the design process of the dialogue package. The knowledge sources are introduced in the order in which they are described in this section. The result of the process is a prototypical system that needs to be refined interactively using test sets. Figure 4 shows a screenshot of the wizard in step 1 at the point of specifying the domain model.

In order to abstract over different input and output modalities, the dialogue system contains an entity to maintain input and output channels. For each channel, there is a channel specification that allows



Please specify the new description.

hotelreservation
reservation
restaurantreservatic

Description: hotelreservation

inherits from: reservation

with data:
hotelreservation x FROM -> date
hotelreservation x NAME -> string
hotelreservation x TO

and methods: .verifyDates : date x c

Please add a data member to your new class.

Data Member:
Name: PLACENAME
Value: string

Figure 4: The wizard in action. Currently, a class hotelreservation is being specified. The list boxes in the larger dialog box display the base class, the member variables and the methods associated with the class. A new member variable is being added through the smaller dialog box in the foreground. The tree-shaped interface item provides a view on the domain model.

to transform an array of strings into a feature structure (for an input channel) or a feature structure into an array of strings (for an output channel). Input and output devices communicate with the dialogue system only through these channels. The intention of this approach is it to abstract away the particular form of input and output events, thus achieving modularity and extensibility.

4 Debugging of the Dialogue Strategy

The dialogue manager is driven by a Prolog style program which contains the dialogue strategy. As long as a user is engaged with the system in a dialogue, it is then the task of the dialogue system

1. to determine if the user intends to have the system perform one of the tasks known to the system, and if so,
2. to interactively acquire all the information that is needed for the system to uniquely determine the task to be executed and all its parameters, and
3. finally to notify and pass control to the subsystem responsible for the task execution once this state has been reached.

For that purpose, each task description has an internal state that can take one of the following values: Neutral, Selected, Deselected, Determined and Finalized. The state transitions are as shown in figure 5. Each state transition is passed on to the implementation of the dialogue package in the

$$\begin{tikzcd}
\text{Neutral} \arrow[r] \arrow[dr] & \text{Selected} \arrow[r] \arrow[d] & \text{Determined} \arrow[r] \arrow[dl] & \text{Finalized} \\
& \text{Deselected} & &
\end{tikzcd}$$

Figure 5: State transitions of the dialogue goals

back-end application which may or may not choose to make use of this information.

The state of the dialogue system is implicitly represented by the vector of dialogue goal states. The states of the dialogue goals are updated by a set of rules that compare the representations of the utterances with the representations in the dialogue goals. The state of a goal incompatible with the current representation becomes Deselected. A goal in the state Selected becomes Determined as soon as it is the only goal in the selected state. A Determined goal becomes Finalized as soon as the information acquired in the dialogue is at least as specific as it is required by the goal.

There is a generic dialogue strategy that serves as a starting point for system development. As possible domains may be very distinct, it becomes necessary to adapt the strategy to the domain at hand. For this reason, the IDE offers an interactive debugger interface to the rule program. It allows for single step execution, display of call stacks and variable substitutions as well as a direct query interface to evaluate the effects of single rules.

5 Testing

The cycle of grammar maintenance, testing and evaluation is a tedious and time consuming part of the development of a new application. The IDE offers a set of utilities simplifying the task.

5.1 Batch Testing

Grammar Testing

The IDE offers a tool for batch testing of grammar coverage. Here, a text string is passed through the semantic parser and conversion routine. The resulting feature structure is then presented graphically to the user. The designer is then prompted to evaluate the semantic representation of the utterance. Current choices are those defined by the partial order of feature structures. In other words, the system's designer can specify if the semantic content contains information that is equal to, less specific than, more specific than or inconsistent with the information the sentence conveys. The text string, the feature structure and the evaluation are then automatically entered to the batch test set. The system designer can then run this batch test set later in the development process and receive notification should the resulting feature structures differ in informational content. This procedure assumes, however, that the domain model is not changed between the tests. Alternatively, the system designer can enter the desired feature structure directly.

Testing for Goal State Transitions

In addition to the grammar coverage batch test, there is a dialogue goal batch test. As mentioned above, the state of the dialogue manager is implicitly described by the vector of goal states. Each utterance is assumed to represent a speech act that performs a state transition in some of the dialogue goals. Here, we store together with the utterance two vectors of dialogue goal states: before the utterance has been processed and after the utterance has been processed. During batch testing, the dialogue goals are set to the states specified in the first vector. Subsequently, the utterance is passed through the dialogue system. Then the actual goal states after processing of the utterance are compared with those in the batch test and differences are prompted to the application designer.

Testing for Orthogonality between Modules

Testing for dialogue goal state transitions requires the configuration of dialogue packages to be constant between tests. However, there are several utterances whose meaning can unambiguously be attributed to one dialogue package. For this reason, the IDE offers an additional batch test. Here, the utterances are assigned a dialogue package as well as vectors of goal states. In contrast to the state transition test, we only represent goal states from dialogue goals in the package in question. As above, the application developer is notified if the desired goal configuration in the package differs from the calculated one. Moreover, any goals not in the assigned dialogue package whose state differs from DESELECTED are displayed to the user.

5.2 Dialogue Goal Activation and WOZ

Since the IDE contains a detailed description of the dialogue goals, it is possible to present the dialogue goals to the application designer in form that needs to be filled in through the standard graphical user interface rather than through speech. Once the back-end application is in place, the application designer may proceed to test the interface of the dialogue system with the back-end application. Another possibility would be to use this feature as a poor man's Wizard of Oz interface, in which case only the domain model and the task model need to be in place (although additional support from the database would be desirable). This feature is currently under development.

6 Discussion

We are currently using the described system to prototype two spoken language applications. While it is too early to arrive at any conclusive results, our preliminary experience shows that a substantial amount of time is saved simply by using the wizard to avoid formatting errors and typographic errors in the several specification files. Moreover, as the wizard displays the options available for the user to choose from, it is easier to arrive at consistent specifications. This is particularly true in the instances where type information from the domain model can be used to reduce the number of options.

Another characteristic of the system is its integrated architecture. The entire system runs as a single thread in a single process. Comparing to an earlier version of the system in which a client/server architecture was employed, we find debugging and testing easier.

From a domain model perspective, the dialogue packages as a primary building block offer a coarse granularity compared to dialogue states, speech objects or dialogue libraries. We feel it is for this reason more comprehensive. Whether this characteristic is of benefit and whether the specifications in the different packages are sufficiently orthogonal to not interact when building the final system remains to be seen.

Although the specifications of knowledge sources in separate modules can be independent of each other, undesired interaction may not be excluded. In particular, the informational content of the dialogue goal specifications need pairwise inconsistent. The reason is that the dialogue manager bases its decision on the compatibility of the dialogue goals with the information in the discourse. If one dialogue goal were less specific than another, the second dialogue goal could never be reached as the first is satisfied first. For this reason, the dialogue manager checks for pairwise inconsistency of the goals at runtime.

Future work includes the integration of a speech recognizer directly into the development environment and improvements of the graphical user interface to speed up the design process. These improvements can only be made by experiences gained through continuous use of the wizard.

Acknowledgements

This research is supported by the Defense Advanced Research Projects Agency under contract number DAAD17-99-C-0061. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the DARPA, or any other party. I would like to thank the members of the dialogue working group in the DARPA *Command Post of the Future* project for valuable discussions.

References

M. Araki, K. Komatani, T. Hirata and S. Doshita. *A Dialogue Library for Task-Oriented Spoken Dialogue Systems* Workshop on Knowledge and Reasoning in Practical Dialogue Systems. Stockholm, Sweden, 1999. Available from http://www.ida.liu.se/ext/etai.

B. Carpenter. *The Logic of Typed Feature Structures.* Cambridge University Press, 1992.

R. Cole. *Tools for Research and Education in Speech Science.* Proceedings of the International Conference of Phonetic Sciences, San Francisco, USA, 1999.

M. Denecke and A. Waibel, *Dialogue Strategies Guiding Users to Their Communicative Goals.* Proceedings of Eurospeech, Rhodos, Greece,1997. Available from http://www.is.cs.cmu.edu.

M. Denecke and A.H. Waibel, *Integrating Knowledge Sources for a Task-Oriented Dialogue System.* Workshop on Knowledge and Reasoning in Practical Dialogue Systems, Stockholm, Sweden,1999 Available from http://www.is.cs.cmu.edu.

M. Denecke. *Modularity in Grammar and Ontology Specification.* Proceedings of the MSC 2000 Workshop, Kyoto, 2000. Available from http://www.is.cs.cmu.edu.

M. Gavalda and A. Waibel. *Growing Semantic Grammars.* Proceedings of the COLING/ACL, Montreal, Canada. Available from http://www.is.cs.cmu.edu.

Anke Koelzer. *Universal Dialogue Specification for Conversational Systems* Workshop on Knowledge and Reasoning in Practical Dialogue Systems. Stockholm, Sweden, 1999. Available from http://www.ida.liu.se/ext/etai.

K.A. Papineni, S. Roukos and R.T. Ward. *Free-Flow Dialogue Management Using Forms.* Proceedings of EUROSPEECH 99, Budapest, Ungarn, 1999.

S. Sutton, D. G. Novick, R. A. Cole, and M. Fanty. *Building 10,000 spoken-dialogue systems.* Proceedings of the International Conference on Spoken Language Processing, Philadelphia, PA, October 1996.

Walker, M.A. and Litman, D.J. and Kamm, C.A. and Abella, A. *PARADISE: A Framework for Evaluating Spoken Dialogue Agents* Proceedings of the 35th Annual Meeting of the Association of Computational Linguistics, 1997. Available from http://www.research.att.com/ walker.