Domain and Concept Model Based Generic Dialogue System for Practical Speech-based Database Retrieval Applications

Jaakko Hakulinen, Markku Turunen

Department of Computer and Information Sciences
University of Tampere, Finland
jh@cs.uta.fi, mturunen@cs.uta.fi

Abstract

This paper discusses generic dialogue systems and their requirements for practical speech applications. In general dialogue systems the dialogue control is separated from the domain knowledge. New applications are constructed by writing domain specific concept and domain models. We introduce our implementation of a generic database information retrieval system and present experiences from a bus-timetable information application called Busman (Bussimies). The generic dialogue system is implemented as a part of our agent-based Jaspis speech application development framework.

1. Introduction

Although many sophisticated dialogue systems have been proposed and implemented in the last decades, there exist strong need for practical spoken dialogue systems, which can be implemented using basic dialogue control strategies, such as forms and state-machines. These control strategies are suitable for many practical applications, such as popular information retrieval applications (weather, train, buss and flight information services.).

There has also been a tendency towards generic dialogue control management. Especially form-based dialogue management systems have been successfully scaled to multiple domains. The use of a form as a basis of dialogue management makes it possible to define generic rules for rather broad domain of information retrieval tasks. There have also been attempts to create a generic design tool that would work with different dialogue engines [1].

On one hand recent dialogue system development has aimed at handling more complex domains. E.g. CMU communicator [2] and several similar systems (e.g.[3]) have extended simple form filling task with dynamic tree-like dialogue structures. However, these systems are not quite domain independent and the question of scaling up to other than the original domains is still open.

The basic type of form-based information retrieval application is a system where the user gives the system a query and the system checks if the query is legal one, i.e. it has all necessary information and that the given information is not ambiguous or contradictory. The query must also be strict enough to come up with a reasonable set of results. To do the checking, the user's input is converted into a form, or semantic frame, as it is sometimes called [2]. This frame contains a set of concept-value pairs. The set of concepts in the form is often static and application specific information defines how the form should be filled. The system can retrieve information from the backend (usually a database) if the form has valid values. If it does not, the system takes the initiative to make the form complete.

The goal of making a system domain independent means that there should be no application specific program code inside the dialogue control component. The dialogue control component should be such that it can be used in any application inside the domain, for example in any database information retrieval application. The application specific material should be in the configuration files. This approach, to be a desirable one, requires that the configuration files are easy to write and maintain. In this way programming is not needed to create practical applications.

An important requirement for many practical systems is the ability to handle multiple languages. For example, our experiences with the e-mail application Mailman (Postimies) indicate that such a system must be multilingual in order to be useful [4]. To efficiently implement multilingual systems we must keep dialogue management as language independent as possible. The general dialogue management components should not include any language specific code. Instead, the handling of language specific issues should be modularized and separated from general dialogue management. Furthermore, it would be preferable that the language dependent components be domain independent.

The work described in the following sections describes an implementation of a generic dialogue system for a wide range of practical speech-based database retrieval applications and presents experiences from a local bus timetable information application Busman (Bussimies). The presented system is part of the Jaspis speech application development framework [5]. The system takes advantage of the agent and evaluator based architecture of Jaspis, but should be rather straightforward to implement outside the Jaspis framework as well.

This paper is organized as follows: first we describe the domain and concept models and then introduce how dialogue management works. A brief explanation follows on how speech inputs are parsed and how speech outputs are generated. Finally, conclusions and discussion are presented.

2. Domain and Concept Models

Information sources that a spoken dialogue system contain, can be organized as *domain model*, *concept model* and *dialogue model* [6]. Many systems use these information sources or some subset of them. This work focuses on the domain model and the concept model. The meaning of the concept model in our database query oriented system is that it defines the concepts that the system operates with. Most of the concepts relate almost 1:1 to the database fields but usually there are also concepts that cannot be found in the system database. The concept model includes the linkage to the database. It

```
<concepts>
 <concept>
  <name>email</name>
  <values><value>
   <output><type><name>email</name></type></output>
   <expression>e-mail address is <type><name>email</name></type>
   </expression>
  </value></values>
  <datasource>
   <tablename>emaildb.txt</tablename>
   <fieldname>email</fieldname>
   <datatype>string</datatype>
  </datasource>
 </concept>
 <concept>
  <name>qtype</name>
  <values><value>
   <output>what</output>
   <expression>what</expression>
  </value></values>
 </concept>
 <concept>
  <name>person_name</name>
  <values><value>
   <output><type><name>name</name></type></output>
   <expression><type><name>name</name></type></expression>
  </value></values>
  <datasource>
   <tablename>emaildb.txt</tablename>
   <fieldname>name</fieldname>
   <datatype>string</datatype>
  </datasource>
 </concept>
</concepts>
<value_types>
 <type>
  <class>Jaspis.ConceptSystem.ListDefinedConceptValueType</class>
  <name>name</name>
  <values>
   <value>Jaakko Hakulinen</value>
   <value>Markku Turunen</value>
  </values>
 </type>
</value_types>
<result_relations>
 <relation>
  <concept><name>qtype</name><value>what</value></concept>
  <result><name>email</name></result>
 </relation>
</result_relations>
<query_relations>
 <relation>
  <wanted><name>email</name></wanted>
  <needed><name>person_name</name></needed>
 </relation>
</query_relations>
<fill_rules>
 <infer>
  <concept><name>qtype</name><max_depth>5</max_depth></concept>
 </infer>
 <auto_fill>
  <concept><name>qtype</name><value>what</value></concept>
 </auto_fill>
</fill_rules>
```

Figure 1 : concept and domain models which define email addresses of the authors.

also includes information about the lingual appearances of the concepts and how it relates to the form of information presentation used internally in the dialogue management.

The domain model gives further meaning to the concepts by defining the relations between them. These relations express how the values of certain concepts can be retrieved from the database when we know some other values and what information should be retrieved in different situations. Furthermore, the domain model defines how concept values can be retrieved from the dialogue history and how the values can be automatically filled with default values (certain kind of world knowledge).

Figure 1 gives an example definition of the domain and concepts models. It contains an XML document that defines a system that serves e-mail addresses.

Our system does not include an explicit dialogue model. With the definitions given to the other models, the dialogue model would be a definition of how the dialogue flow should go. This would include definitions of when to ask question, when to make the database queries and when to serve the information. This work leaves the modeling of that information out of its focus and lets the dialogue model to be included in the program code that implements the dialogue manager.

These three models are dependent of each other only in one direction. The concept model is independent, the domain model requires a certain type of concept model and the dialogue model requires certain types of domain and concept models. Therefore, if the dialogue model is included in the program code of the dialogue manager, it should be possible to build differently behaving dialogue managers that all use the same domain model. This is of course almost trivial and a failure to do so means that the domain model is actually a mixture of domain and dialogue models. If the domain model does not include the dialogue model, it is possible to develop an application by defining the domain and concept models. Later, the application can be improved by using a better dialogue manager.

2.1. Data types

In order to use concepts efficiently we need to define 'data types', i.e. specific types of data values for the concepts that may have a limited set of values. This data is often knowledge about proper names in the application domain. Often this information can be found in the database of the system. For example, in movie database domain, the names of movies are one kind of 'data type'.

The simplest data types are straightforward. Integers and numbers in general are no problem. As they are common to many systems, their handling can be built in the generic system. Some more complex data types are also very common so that it may be reasonable to build support to them. Good examples are date and time related types. Especially a lingual form in which time can be expressed can vary a lot so speech recognition vocabulary and natural language understanding need to have more logic than just a simple list of possibilities.

Most, if not all of the applications also require their own specific data types. For example, a bus timetable system requires a data type representing the different bus lines. Often these types can be defined as a list of different values a variable or a concept of this type can take. These values are often found in a database or they can be simply listed somewhere. Therefore, a generic system only needs to be able to read this list from some source and construct a data type according to that definition. In some cases, however it is possible that we need to introduce new data types that need their own processing e.g. in language understanding. A system must therefore make it possible to introduce new data types dynamically. If possible, one should be able to add these data types to the system without recompiling the entire system.

2.2. Result relations, query relations and fill rules

The domain model consists of three parts, *result relations, query relations* and *fill rules*. The result relations define which concepts we are looking values for in the database. For example, if the user wants to know when a bus arrives at a

certain place, he/she is presenting a query-type utterance. Now we have a concept 'query' with value 'when' and the result relations tell us that we are looking a value for the concept 'arrival-time'. In other words, given a set of concepts, result relations can tell us what the user is asking for in terms of defined concepts.

The query relations define how a database can be used to query values for certain concepts. A query relation defines which concepts need to have values in order to retrieve a value for the wanted concept, i.e. present a query for a concept based on the values of other concepts. However, it does not define what these values should contain. For example, in order to retrieve a value for the concept 'arrival-time' we need to have values for the concepts 'arrival place', 'departure time' and 'departure place'. Given these, we can perform a legal query.

Finally, fill rules provide the system with an additional way to find values to the concepts. The information sources available for the system without these rules ask the user and query the database. In our current implementation, these rules give the system two additional information sources. One is dialogue history. The fill rule section of the domain model can define that values for a concept can be taken from the forms of the previous dialogue states. The other way to give a value for concept is so called autofilling. In this case the domain model defines a value that can be given to a concept if necessary (i.e. the information can not be found in other sources). This part of the domain model actually contains a kind of world knowledge as it tells the system how things usually are if no other information is available.

It is also possible to define restrictions for the fill rules. These restrictions can tell that the value of one concept may not be the same as the value of some other concept. This kind of restriction is sometimes necessary so that the dialogue history or autofill rules can't make the current concept frame inconsistent. In our applications we have not needed other kind of restrictions. However, it is easy to imagine that other kind of restrictions could be needed for certain applications. For example, in some cases certain concepts should have equal values, the value of one concept may not be larger than the value of some other, or a value should be within some specific range etc.

3. Dialogue Management

The dialogue manager (DM) handles the situation in spoken dialogue systems when the input-handling component has analyzed and conceptualized inputs from the user. The Dialogue manager also takes care of situations where the database manager has produced a reply for its query. In both cases, the results are expressed in the form of semantic frame. Next we introduce how dialogue management works in domain and concept model based information retrieval applications.

When DM starts it first checks if the input it received was from the database of from the user. If the input was from a user and the DM does not understand it, it produces a message to the user about this. If input was from the database, the result is checked to see if it contains a result that the user asked for or just some intermediate results that are required to construct the final result. If the final result is not available, or if the input was from the user, the DM proceeds to the actual processing.

First DM finds the concepts that the user queried a value for, e.g. what is the final result. The result relations are compared to the concept values in the current frame and, if necessary, concept values in the dialogue history and the autofill values. Next, DM collects a set of concepts that are required to retrieve the result concept. Each of these concepts needs to have a value, so an iteration loop goes through each of them trying to do one of the following in this order:

1. Find a value for the concept in the current frame
2. Make a database query to get a value for the concept
3. Use fill rules (dialogue history, autofill)
4. Ask the concept value from the user

If any of the concepts can be retrieved from the database, the loop is terminated, a database query is constructed and DM passes the turn to the database manager. The queries for a concept value can also recursively cause a query for some other concept whose value is required to make a query for this concept. If no database query can be made for any of the concepts, and some value is still missing, DM needs to ask values for concepts from user. The request is passed to the output presentation component and DM finishes its turn. If the user supplies the wanted information, it will be available for the DM in the next round and no special processing is necessary on DM level.

The construction of a database query is, as already noted, a recursive procedure. If necessary, the query relation rules can be several steps long so that in order to retrieve one concept, other queries need to be made. This is implemented using a recursive algorithm which receives the concept to be queried and constructs the database query if possible. If other queries are needed, the algorithm calls itself until it reaches a situation where a query can be made or it finds out that no query can be made at all. The query construction process can also use dialogue history and autofill rules to get values for concepts if queries are not possible. However, making a query is always the primary option. This makes it sure that we do not pick values from the dialogue history that are out of date, because they were retrieved from the database with old parameters.

As the query construction algorithm is recursive, it keeps a list of concepts to which it may not make a query for. This makes it sure that we do not get into a recursion loop in cases where the domain model makes it possible to query values in a loop like manner.

To be able to handle concept values correctly, each concept value pair contains also a third field called *source*. This field tells where the value for the concept was taken from. This source can be user, database, dialogue history or autofill rule. These sources must be identified so that the system can correctly prioritize different information. It is necessary in many cases to retrieve database results again when user has updated some of the values in the current frame. If a system retrieves some information just to be able to retrieve the final, wanted result, this temporary result needs to be ignored if some of the initial information changes. Of course the final results also changes in such a case.

There is one special case in the information retrieval part, where the above algorithm is not used. It is a situation where an intermediate database result contains multiple values that would lead to a too large result set. In these cases the DM asks the user to further define, which alternative is the one to follow. When the user has given an answer which is usually one of the possible values the DM algorithm can proceed normally. As this DM behavior is completely independent of the main DM algorithm, it can be implemented as a separate dialogue agent. The Jaspis architecture supports this kind of

multiple agents and depending on the case the appropriate DM agent is selected to handle the turn.

Multiple Jaspis agents can be also used to handle cases which are not part of the actual information retrieval, e.g. help request and repetitions of information can all be handled in their own agents. This system keeps each agent simple and easy to maintain.

4. Input Parsing and Output Generation

To parse spoken language utterances from the user and produce pleasant and intelligible synthesized outputs, we use the input and presentation agents of the Jaspis architecture. To implement these components for a domain and concept model based system means that input component must parse inputs to the concepts defined in a concept model and the presentation component must be able to produce natural sentences from these same concepts.

In our current implementation the input handling, i.e. natural language understanding is done comparing the given expressions to the user's input. There is no need to describe the relations between the lingual expressions in order to have a simple but working language understanding. In more detail, the process includes comparisons of wild card strings and data types of the expression to input. As there are several concepts in a single system, each of the concepts is compared to the user's input. Results of the comparisons return the position where the concept was found in the input and the output value of the concept, which can be a list of strings and values of different data types.

There is also a need to understand such expressions that do not match directly to the concepts of the system. These are, for example, such cases as greetings in the beginning of a dialogue, and help requests. These utterances are not part of the actual information retrieval task and therefore do not (necessarily) have matching concepts in the system. For such expressions, we see that it is best to write completely separate input handling components. As the system is implemented on the Jaspis platform, the solution is simply to write other input agents parallel to the one which handles known concepts. As each agent tries to work out each input, the help requests could be detected by an agent specialized in them and information retrieval request by the described concept based agent. An input evaluator can then combine the results if the input contains, for example, both a greeting and a query.

In our current implementation, system outputs are constructed using slot filling. The concept model contains natural language expressions for concepts. The slots, i.e. values of concepts are filled and output presented to the user.

It should be noted that the natural language understanding and generation parts are separate from the dialogue management and could be replaced with more advanced modules.

5. Conclusions and Future Work

In this paper, we have introduced a generic database query dialogue system for practical spoken dialogue applications. The system makes it possible to construct reasonably complex information retrieval tasks for different domains. Applications are implemented by writing concept and domain models. These models define everything that input handling, dialogue management, database interface and presentation modules need to operate.

The aim of the work is to introduce a system that makes it easy to implement new applications. By extracting the domain dependent information into the domain and concept models we can develop applications by writing a single XML file. The strength of this system is in the ease of constructing new practical applications. It will be interesting to see how easily complete novices will be able to implement applications with the system. A bus timetable query system (Bussimies) has been implemented using the presented system. Other applications include an implementation of a movie information system.

In addition, the system also includes possibility to write new modules to introduce new data types to the system. Jaspis [1], the application development platform that the system is built on, also allows the developer to easily add more functionality to the system in a modular way. This system also demonstrated the strength of the agent systems of Jaspis. For example, different special cases in dialogue management (error handling, greetings etc.) can be implement totally independent of the main dialogue management. The domain and concept models also support multilingual applications as well as the Jaspis system in general.

Future work includes expanding the query and result relation parts to include update relation rules. This would make it possible to use the system to write applications that also update the database, making the system more general in nature. One more important feature that is not supported by the system at the moment, but which would be important to include, is the automatic generation of speech recognizer vocabularies and grammars. This would make the system complete in the sense that a speech-based application could be implemented by just defining concept and domain models.

6. Acknowledgement

This work was supported by the Academy of Finland (project 163356) and by the National Technology Agency (Tekes).

7. References

[1] Kölzer, A. "Universal Dialogue Specification for Conversational Systems", Linköping Electronic Articles in Computer and Information Science, http://www.ep.liu.se/ea/cis/1999/028/.

[2] Rudnicky A. and Xu W. "An Agenda-Based Dialogue Management Architecture for Spoken Language Systems", IEEE Automatic Speech Recognition and Understanding Workshop, 1999.

[3] Niimi Y, Oku, T., Nishimoto, T. and Araki M. "A Task-Indepented Dialogue Controller Based on the Extended Frame-Driven Method", 6th International Conference of Spoken Language Processing, ICSLP 2000.

[4] Turunen, M. and Hakulinen, J., "Mailman - a Multilingual Speech-only E-mail Client Based on an Adaptive Speech Application Framework", Workshop on Multi-Lingual Speech Communication, 2000. pp: 7-12.

[5] Turunen, M. and Hakulinen, J. "Jaspis - A Framework for Multilingual Adaptive Speech Applications", 6th International Conference of Spoken Language Processing, ICSLP 2000.

[6] Dahlbäck N. and Jönsson A., "Knowledge Sources in Spoken Dialogue Systems", Eurospeech'99, Budapest, Hungary, 1999, pp. 1523-1526.