Bilar

Daniel Grönlund
Master's thesis in Computational Linguistics
Göteborg University
February 2003

Supervisors:
Robin Cooper and Staffan Larsson
Department of Linguistics
Göteborg University

Abstract

In most dialogue systems the focus of a conversation consists of one object at a time. In some domains it would be advantageous to focus on sets of objects. In this thesis I investigate some strategies that can be used to handle changes of focus when sets of objects are involved. I also describe the implementation of a system that uses one of these strategies. The system makes it possible to communicate with a database, containing information about used cars, using natural language.

Contents

1 Introduction **4**

2 Background **6**
2.1 Dialogue theory . . . 6
2.1.1 Speech Acts . . . 6
2.1.2 Co-operative principle . . . 7
2.1.3 Dialogue Games . . . 8
2.2 Dialogue systems . . . 9
2.2.1 Parts of a Dialogue system . . . 9
2.2.2 Managing dialogues . . . 10
2.3 Existing systems . . . 12
2.3.1 CSLU . . . 12
2.3.2 Philips/SJ . . . 12
2.3.3 LINLIN . . . 13
2.3.4 Trindikit . . . 16
2.3.5 Godis . . . 18

3 System Description **24**
3.1 Requirements . . . 24
3.1.1 General . . . 24
3.1.2 Dialogue Structure . . . 24
3.1.3 Focus Structure . . . 24
3.1.4 Customisation . . . 25
3.2 Strategies . . . 25
3.2.1 Intension and extension . . . 25
3.2.2 Question Revision . . . 26
3.2.3 The Set . . . 27
3.2.4 The Stack . . . 28
3.2.5 The Subdialogue . . . 29
3.3 Implementation . . . 30
3.3.1 Information State . . . 30
3.3.2 Algorithms . . . 31
3.3.3 Update Rules . . . 31
3.3.4 Selection Rules . . . 32
3.3.5 Revision . . . 32
3.3.6 Lexicon . . . 33
3.3.7 Database . . . 34

4 Discussion **35**
4.1 Strategies . . . 35
4.1.1 General . . . 35
4.1.2 Comparison of strategies . . . 35
4.2 Extensions . . . 38
4.2.1 Focus . . . 38
4.2.2 Meta communication . . . 40
4.2.3 Godis and Bilar . . . 40

A Code **42**
A.1 Database . . . 42
A.2 Domain . . . 45
A.3 Lexicon . . . 47
A.4 Selection Rules . . . 50
A.5 Update Rules . . . 52

Chapter 1

Introduction

Dialogue systems make it possible to communicate with machines using natural language. Instead of forcing humans to learn artificial modes of communication machines are adapted to understand humans. The ideal dialogue system would be able to communicate at least as well as a human regardless of the domain in which it is used. This will not happen in a long time, but in order to get there eventually we need to explore the many small parts that need to be improved.

The purpose of this thesis is to present my implementation of a dialogue system that imitates parts of the system described in Arne Jönsson's Ph.D thesis, LINLIN (Jönsson, 1993). The LINLIN system makes it possible to access a database containing information about used cars using natural language.

My system is implemented with Trindikit and Prolog, LINLIN was implemented with Lisp. The basic communication between the user and the system will look something like this:

```
$U> show volvo
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |volvo          |240       |1985  |
    |volvo          |740       |1989  |
    |volvo          |850       |1993  |
    |volvo          |440       |1990  |
    |volvo          |v70       |2000  |
    +---------------+----------+------+
```

The user can then interact with the system to obtain modified versions of the table or new tables with completely different information.

When developing a dialogue system there are many different strategies to choose from for each of the different parts of the system. While choosing between different strategies there are two things to consider. The first is the inherent properties and limitations of a strategy. The second is how these properties relate to the actual communication that will take place. The way the communication will work can be examined empirically before designing the system. If you have extensive knowledge about inherent properties of different strategies it becomes easier to choose and adapt the right ones in the right ways to build a system that complies with empirical test results.

In this thesis I will investigate the inherent properties of some strategies for handling the focus structure of dialogues involving tables with several objects that are discussed at any given time. I will also describe the implementation of a system that uses one of these strategies.

Chapter 2

Background

2.1 Dialogue theory

Theories concerning dialogues cover segments of speech that are larger than a single sentence/utterance and involve more than one dialogue participant. The treatment of context is often important as contributions in a dialogue tend to be more context dependent than written language. I will give a brief account of the Speech act theory and cooperative principle. I will also write about the Dialogue game approach of Carlson (Carlson, 1983). Then I will mention some existing toolkits and systems and discuss how dialogue systems are implemented.

2.1.1 Speech Acts

Speech act theory was developed as a reaction against the earlier logical positivism according to which everything could be reduced to a matter of truth and falsity. Instead an utterance had to be viewed in relation to the activity in which it was involved. Austin ((Austin, 1962) as seen in (Levinson, 1983)) introduced felicity conditions which are used to determine whether an utterance is used successfully or not (as opposed to it being true or false). The conditions are:

A. (i) There must be a conventional procedure having a conventional effect

(ii) The circumstances and persons must be appropriate, as specified in the procedure

B. The procedure must be executed (i) correctly and (ii) completely

C. Often, (i) the persons must have the requisite thoughts, feelings and intentions, as specified in the procedure, and (ii) if consequent conduct is specified, then the relevant parties must so do

Austin then went on to define three types of actions one can perform by saying something, locutionary acts, illocutionary acts and perlocutionary acts. Later the term speech act has come to refer exclusively to illocutionary acts.

An illocutionary act is viewed as an act that is defined by its conventional force. An example of this is:

I hereby pronounce you husband and wife.

This act will be felicitous if it is performed by a priest at a wedding. The same conditions can be applied to acts performed at less formal occasions. For example:

I'm thinking about buying a new car.

In order for this act to be felicitous you still have to fulfill the conditions. You need to actually want to buy a car or want someone else to believe so. This act cannot be performed, for example, by a priest during a wedding ceremony.

Note that in the first example a performative is used, and in the second a constative. The difference between the two is that a performative cannot be said to be either true or false according to Austin. A constative can have truth values attached to it. The felicity conditions are still the same for both kinds of acts.

2.1.2 Co-operative principle

Grice from (Levinson, 1983) suggested that there are rules that are used to make conversation as efficient as possible. He defined the co-operative principle which in turn relies on the maxims of conversation:

The co-operative principle:

Make your contribution such as required, at the stage at which it occurs, by the accepted purpose or direction of the talk exchange in which you are engaged.

The maxim of Quality:

Try to make your contribution one that is true, specifically:

1. do not say what you believe is false
2. do not say that for which you lack adequate evidence

The maxim of Quantity:

1. make your contribution as informative as is required for the current purposes of the exchange
2. do not make your contribution more informative than is required

The maxim of Relevance:

Make your contributions relevant

The maxim of manner:

Be perspicuous, and specifically:

1. avoid obscurity
2. avoid ambiguity
3. be brief
4. be orderly

While these maxims are not always followed literally, they are followed on a deeper level. For example:

A: There seems to be something wrong with the gearbox on my car.

B: Here is an ad for the new Mercedes.

B's answer does not directly address A's problem. What B really means is something like this: 'Is your car broken again? It is very expensive to change gearbox. It would be cheaper in the long run to buy a new car. Also, you wouldn't have to wait for it to be repaired every time. Look at this ad for the new Mercedes. I have heard that it is very reliable. And it looks good. You should buy one.'

B uses the most efficient way to say this. This kind of inference is called implicature.

2.1.3 Dialogue Games

The notion of dialogue games was first explored by Wittgenstein (Carlson, 1983). They have later been studied further by, among others, Carlson.

When analysing dialogues according to the Dialogue Games approach the first thing to do is to define some basic unit to which you can reduce the different parts (parts of a dialogue can be utterances, subdialogues etc.). Then you need rules that define what constitutes a well formed dialogue in terms of the basic units. The third thing you need is a strategy which specifies how to build a well formed dialogue in the most effective way.

The basic units in a dialogue are often represented by speech acts. Often the speech act is the only defined unit and therefore also becomes the primary unit. In Carlson's approach the primary unit is the dialogue game itself. The basic units are represented by moves. Examples of different kinds of moves are questions, answers and greetings. The moves are connected to rules which specify when they can be used. Moves also have parameters such as participant, audience etc. With this approach each speech act is validated within the context of a dialogue rather than by itself.

The structure is defined by rules which are applied to allow moves to be made by participants. The participants each have certain knowledge that is checked by the rules. This knowledge is modelled as epistemic alternatives by Carlson. This basically means that the participants do not only have knowledge of their own, they can also make guesses about the other participants' knowledge (including guesses about each other's knowledge). A dialogue game is always played against

Nature as well as any other participants. Nature knows everything, but is always passive. Nature cannot put forward moves, nor can the other participants directly address Nature.

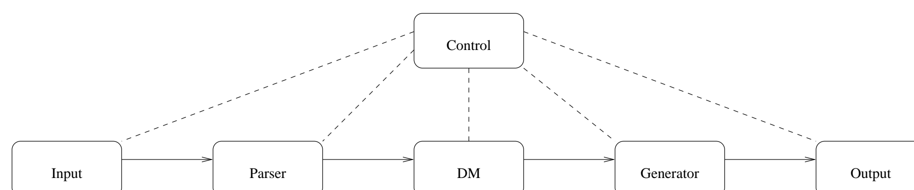
The goal of a dialogue is that the participants should reach an agreement. They do this by sharing relevant information with each other through the use of dialogue moves. The participants always try to reach the goal in the most efficient way possible. Each move in the dialogue game does not have to have a specific aim in itself, as long as it is strategically advantageous to use it. When you reach the goal you win the game. Sometimes strategies become so conventional that they can be formalised. Such strategies are implemented as rules.

The advantage of using dialogue games is that as long as you follow the rules of the game, there are several valid strategies that can be used depending on how much time/resources you want to spend.

2.2 Dialogue systems

2.2.1 Parts of a Dialogue system

Below is a picture that describes how a typical dialogue system is constructed.



There must be some kind of control module or built in strategy that determines what the system should do and when it should do it. It also takes care of the information flow through the system.

The input to a system is usually (if it is not a multimodal system) speech or text. Speech needs to be analysed and converted to some kind of textual representation. Text needs to be filtered in some way to find keywords (if you are using keyword spotting) and take care of spelling mistakes and so on. The converted speech/filtered text is then sent to the parser.

The parser converts a string from the input module to a semantic representation, for example a speech act or a dialogue move. The parsing can be done in two parts, first a syntactic analysis is made and then an interpreter takes care of the semantic analysis. It is also possible to use a semantic parser that takes care of everything in one pass.

The dialogue manager works on a pragmatic level, and is what makes the system a dialogue system rather than just a question-answering system. The dialogue manager keeps track of previous statements, makes sure the dialogue is well formed and is responsible for the strategy that is used. It decides what the system should do based on previous statements.

When the dialogue manager has decided on an action the generator will convert it to (usually) text which is then sent to the output module.

The output module presents the output to the user. If the system is speech based it will use speech synthesis, if text based it formats the text and prints it on the screen.

2.2.2 Managing dialogues

The dialogue manager keeps track of the structure of the dialogues and focus. Using this information it decides what the system should do next after receiving input from the user.

Dialogue structure

We have already seen how dialogue games can be used to model the dialogue structure. Some other commonly used ways to handle dialogue structure are: Dialogue grammars (usually Finite state automata (Fsa) or context free grammars), frames and plans. When Fsa's are used states are connected to each other with utterances or classes of utterances (for example questions of a certain kind). If a context free grammar is used a parse tree is built during the dialogue. This parse tree can then be traversed when the system needs to analyse the dialogue, for example for anaphora resolution. In the frame based approach the system wants certain information in order to fill empty slots. When these slots are filled the system can use this information to provide a relevant answer. In a plan based approach classic planning theory is used.

Grammars and frames have trouble handling complex dialogues, but are efficient. Plan-based systems can handle more complex dialogues but can get problems with efficiency. When creating a dialogue system one can sometimes assume that the dialogues it will be required to handle will not be as complex as most conversation between humans. In those cases a grammar or frame based approach will probably suffice. If the system should be able to communicate more like a human and not be restricted to a certain domain the plan based approach is the only one of these that could have a chance of working.

Focus

The structure of a dialogue is built using moves or utterances, for example a question. Usually a question is followed by an answer. But the question and answer contain additional information that is not necessarily relevant to the dialogue structure. It is, for instance, not relevant for the structure whether a question was about Volvos or Saabs, but the system must still be able to identify the difference in order to give a correct response. It must also remember the content so the user can refer to it later. For example, it does not matter for the structure whether a constituent is

```
ask([X]^[manufacturer(X,volvo)]
```

or

```
ask([X]^[manufacturer(X,saab)]
```

These moves could correspond to utterances where the user asks the system to show Volvos or Saabs. Both are the same kind of object. The answer that

will follow is also probably of the same kind in both cases but the content will be different. This kind of information is stored in a focus structure. The user can then refer back to objects that where introduced earlier in the conversation. Example:

```
$U> show volvo
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |volvo          |240       |1985  |
    |volvo          |740       |1989  |
    |volvo          |850       |1993  |
    |volvo          |440       |1990  |
    |volvo          |v70       |2000  |
    +---------------+----------+------+

<>
$U> show newer than 1990
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |volvo          |850       |1993  |
    |volvo          |440       |1990  |
    |volvo          |v70       |2000  |
    +---------------+----------+------+

<>
```

The user's second utterance does not mention Volvos, but the system remembers that the user asked about them before and therefore assumes that is what the user wants to see. The user could also be referring to things further back in a conversation:

```
U> show older
S> +---------------+----------+------+
   |manufacturer   |model     |year  |
   +---------------+----------+------+
   |volvo          |240       |1985  |
   |volvo          |740       |1989  |
   +---------------+----------+------+
```

Here the system will know that the focus is Volvos even though it was not mentioned in either this or the previous utterance by the user.

Initiative

There are basically three approaches that can be used regarding initiative. The system can keep the initiative, the user can have it or you can have a system with mixed initiative. Who has the initiative is determined by who is the active participant in a conversation. If the system has the initiative it will ask the user questions in order to aquire information about what the user wants to know. If

the user has the initiative he will ask the system questions and the system will just respond to whatever the user brings up. In a system with mixed initiative both the system and the user will bring up new topics.

Grounding

Grounding has to do with how to know if other participants in a dialogue have understood and/or accepted your utterances. This can be signalled with different kinds of feedback. Different kinds of feedback can range from simple one word acknowledgements such as 'mm', 'yes' or 'no' to repeating entire utterances from the other users (for instance if the dialogue takes place in a noisy environment). Utterances can also be considered grounded if the other participant does not explicitly give any feedback but for instance if he answers a question he probably understood the question and therefore gave implicit feedback when he answered it.

2.3 Existing systems

In this section I will describe two kinds of systems: Toolkits, which can be used to create dialogue systems, and dialogue systems themselves. First I will give short descriptions of the CSLU toolkit and the SJ dialogue system. Then I will give a more in depth description of the LINLIN dialogue system and the Trindikit toolkit, and also of the Godis dialogue system which was developed with Trindikit.

2.3.1 CSLU

The CSLU Speech Toolkit is designed to enable rapid prototyping of simple dialogue systems (Bohlin et al., 1999). It provides speech analysis and synthesis. It also provides a graphical user interface that is used to design a model for the dialogue structure. This is achieved by connecting recogniser objects (states) to each other. The states have output and input connected to them. The result will be a finite state model dialogue system. It is also possible to include Tcl code to make more advanced systems. This code is connected to the states and executed when the state is accessed in a dialogue. It is possible to build systems with a more advanced structure this way.

2.3.2 Philips/SJ

The Philips system was developed by Philips (Bohlin et al., 1999) and adapted for SJ (Statens Järnvägar, the Swedish railway company). To use the system you call it over the phone. By talking to the system you can get information about time tables for trains.

The Philips system builds on the principle of slot filling (frames). This means that the system aquires information from the user which it then uses to fill in predefind slots. When all the slots are filled in the system gives an answer. The system always has the initiative. In practice, what happens is that the system first asks about start and destination and then goes on to ask about departure time, arrival time and so on. When the system has aquired all the information it needs it will present an answer. Since the system is used with a telephone

it is entirely voice based and can only give one object as an answer at a time. Since it is a commercial system a lot is required of the speech analysis. The system must also be very robust. This means that it will be more simple than most experimental systems.

2.3.3 LINLIN

Overview

LINLIN was developed in Linköping (Jönsson, 1993). It is a multi modal system that uses text input and output in the form of text, tables and pictures.

An important aspect of NL-interaction that was investigated while making LINLIN is that humans adapt to the system. They do not use the same kind of language they would with another human, instead they use a sublanguage. A sublanguage is a language that is simpler than its human to human language counterpart. The reason they use a sublanguage is that humans tend to think that a machine is unable to understand complicated reasoning and therefore they simplify their language. Also, since they think that they are communicating with a machine they do not have to take care not to hurt its feelings, be polite etc. It is easier to implement a system that communicates in this sublanguage than it is to implement one that mimicks human behaviour. Also, since the user expects the system to behave as a machine, it could actually make the system less usable if it was more human-like.

Another aspect of dialogue systems that was covered in LINLIN is customisability. Two domains were implemented: a Cars domain, where you could get information about second hand cars, and a Travel domain, where you could get information about and book trips. Different domains do not only require different databases and lexicons for the system to access. Users also tend to use different sublanguages depending on the domain. This means the Dialogue Manager also needs to be customized.

In order to find out what a sublanguage looks like they conducted a series of Wizard of Oz experiments. In a Wizard of Oz experiment a user thinks he is using a real system, but instead what output the system gives is decided by a human sitting in another room. The result of these experiments could then be used to determine the behaviour of the system.

Dialogue structure in LINLIN

Dialogue structure is modelled with a context free grammar. The constituents of the grammar are objects. Objects are divided hierarchally in three levels: dialogue, discourse segment and speech act. The objects also contain parameters for focal content. When the Dialogue manager is customised a new grammar is applied. It is also possible to define new kinds of objects (or to not use all available ones). The dialogue grammar for Cars looks like this:

$D ::= IR^+$
$IR ::= Q_T/A_T \mid Q_T/A_S \mid Q_T/A_D \mid Q_S/A_S \mid Q_D/A_S \mid Gr/-$
$Q_T/A_T ::= Q_T\ Q_D/A_D*\ A_T$
$Q_T/A_T ::= Q_T\ Q_D/A_D\ (DC_D)$
$Q_D/A_D ::= Q_D A_D$

$Q_T/A_S ::= Q_T A_S$
$Q_S/A_S ::= Q_S A_S$
$Q_D/A_S ::= Q_D A_S$
$Q_T/A_D ::= Q_T A_D$
$Gr/- ::= Greet-$

D is a dialogue object. IR is a discourse segment. Q_X and A_X are speech acts. D can consist of several IR. Each IR can consist of several Q_X and A_X. IR stands for Initiative/Response. A participant can, for example, initiate a discourse segment with a question. The segment will span the part of the dialogue that starts with the question and ends when the question has been answered. The answer can come at once or there could be a clarification subdialogue in between. Q_T and A_T are task related questions and answers, for example Q_T can be 'what does the volvo 240 cost?' and A_T can be '23000 kr'. Q_D and A_D are clarification questions and answers, for example 'The answer contains 389 objects. Do you really want to see all of them at once?' - 'no, just the ones that cost less than 100000'. Q_S and A_S are system information questions and answers, 'what does the rust value mean?' - 'Rust stands for how rusty a car is. 1 means it is very rusty, 5 means it is not rusty at all.'.

Focus

There are two parameters for focus, objects and properties. Objects in this case means entries from the database that are displayed. Properties specify which fields belonging to these entries should be shown. The focus structure is maintained by storing and copying values in and between dialogue objects. In the Cars system objects are specified by manufacturer, model and year. Those three parameters function as a primary key in the database. Objects also have properties attached to them. A property consists of an aspect and a value. The value can be instantiated or empty. Here is an example of how it looks (Jönsson, 1993): The user utterance "show service costs, repair costs and maintenance costs for volkswagen passat, opel ascona and bmw 518/520" will get the following objects and properties:

$$Objects: \left\{ \begin{bmatrix} Manuf: Volkswagen \\ Model: Passat \\ Year: \end{bmatrix} \begin{bmatrix} Manuf: Opel \\ Model: Ascona \\ Year: \end{bmatrix} \begin{bmatrix} Manuf: BMW \\ Model: 518/520 \\ Year: \end{bmatrix} \right\}$$

$$Properties: \left\{ \begin{bmatrix} Aspect: Service \\ Value: \end{bmatrix} \begin{bmatrix} Aspect: Repair \\ Value: \end{bmatrix} \begin{bmatrix} Aspect: Maintenance \\ Value: \end{bmatrix} \right\}$$

The heuristics for the focus structure differ between the applications. In the Cars system it works as follows:

If the user specifies a set of cars that are the same or a subset of the previous set the set of properties provided in the new user initiative are added to the old set of properties. The new properties will often restrict the set of cars further. If the user specifies a new set of cars the old set of properties is replaced with the ones from the new user initiative. I will show some examples that will show how it should work:

```
$U> show volvo
```

```
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |volvo          |240       |1985  |
    |volvo          |740       |1989  |
    |volvo          |850       |1993  |
    |volvo          |440       |1990  |
    |volvo          |v70       |2000  |
    +---------------+----------+------+

$U> show cheaper than 50000
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |440       |1990  |15000     |
    +---------------+----------+------+----------+
```

Price is added to the second table since the set of cars in the second table is a subset of the cars in the first table. A second example:

```
$U> show volvo with price
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |850       |1993  |50900     |
    |volvo          |440       |1990  |15000     |
    |volvo          |v70       |2000  |180000    |
    +---------------+----------+------+----------+

$U> show saab
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |saab           |900       |1985  |
    |saab           |99        |1982  |
    |saab           |9000      |1999  |
    |saab           |93        |2001  |
    +---------------+----------+------+
```

Price is not shown in the second table since the set of cars in the second table is not a subset of the cars in the first table.

2.3.4 Trindikit

Overview

Trindikit (Larsson et al., 2000) is a toolkit for developing dialogue systems. Its focus is on dialogue move engines and information states.

The architecture of a system developed in Trindikit will be similar to the model we saw earlier in section 2.2. The dialogue manager (DM) part consists of an information state and a dialogue move engine (DME). The information state could be said to contain the declarative part of the DM and the DME the procedural part.

To develop a dialogue system with Trindikit you need to specify a number of modules (at least an update module is required), an Information State type declaration and a controller. To get a usable system modules for input, parsing, generation and output are also needed. Simple input and output modules are provided with Trindikit. It is also possible to attach resources such as databases, dialogue grammars and plan libraries.

In this section I will describe some important concepts in Trindikit. In the following section there will be an example of a system that has been implemented with Trindikit, Godis.

Some theory

The information state models a dialogue at a certain point in time. It contains information derived from the dialogue up to that point. This information can then be used by the system to determine what it should do next. The information state can be seen as a gameboard of a dialogue game.

An information state is used to model the internal states of the participants of a dialogue. The internal states can contain beliefs, knowledge, intentions, wishes etc. This knowledge can be divided into two parts: static and dynamic knowledge. Static knowledge does not change during a dialogue. Examples of static knowledge are domain knowledge and knowledge about dialogue conventions. Dynamic knowledge changes during a dialogue. If you ask a question and it gets answered you will add the content of the answer to your dynamic knowledge. The dynamic knowledge is updated.

Communication with the information state is made with dialogue moves. Dialogue moves are an abstraction between utterances and types of updates. If someone asks a question it will become a question move which will lead to an update of the information state, probably in the form of an intention to answer the question.

Moves are handled with update and selection rules. Update rules change the information state when moves are made by other participants. Selection rules select which moves should be put forward based on the information state. The update and selection rules constitute the Dialogue Move Engine.

Information State

The Total Information State (TIS) consists of three parts. The information state variable (IS), the module interface variables and the resource interface variables. Usually when the information state is mentioned it is the information

state variable that is referred to. The module and resource interface variables are used for communication with modules and resources.

The information state is represented as a record. The fields of the record can have a number of types (defined in Trindikit), for instance they can be records, stacks, sets, stacksets etc.

Accessing the TIS

There are three ways to access the TIS, checks, queries and updates.

Checks: use datatype conditions (usually) or operations. Checks are either true or false.

Queries: these work like checks but fail or succeed. Queries are used for querying Prolog predicates from user-defined modules.

Updates: use datatype operations to change the TIS. Unlike checks and queries, updates do not bind Prolog variables.

Checks, queries and updates are applied to the TIS with the help of update rules. Rules look like this:

rule(RuleName, PrecondList, EffectsList)
of_class(RuleName, RuleClass)

RuleName is the name of the rule. PrecondList is a list of checks. EffectsList is a list of queries and updates. Rules also have a class which can be used in algorithms which are described below.

Datatypes

The information state is specified with abstract datatypes. A number of datatype definitions come with Trindikit. It is also possible to define your own datatypes. There are complex and simple datatypes. As opposed to simple datatypes the definitions of complex datatypes include conditions and operations on objects. The definition of a datatype includes a name, and if it is a complex datatype operations and conditions. Operations modify objects, conditions are true or false for objects.

Algorithms

There are two kinds of algorithms that can be used with Trindikit: Control algorithms and Update algorithms. Control algorithms are used for specifying the general flow of information through the system. Update algorithms are used for specifying the order in which update rules are applied.

Algorithms are written with the DME-ADL and Control-ADL languages. ADL stands for Algorithm Definition Language. DME-ADL is used for writing algorithms for updating the TIS and Control-ADL is used for writing the control algorithm.

DME-ADL

DME-ADL is written with expressions of the following kind:

Rule means apply the update rule Rule

RuleClass means apply a rule of the class RuleClass. If there are several rules of the same class the rules are tried in the order they are declared

[**R1,...,Rn**] means execute R1,...,Rn in sequence

and so on. There are a number of common constructs such as while and if-then-else expressions.

Control-ADL

Control-ADL is similar to DME-ADL. The difference is that instead of invoking rules or rule classes, modules such as lexicon, update and so on are called. Control algorithms can be serial or asynchronous. Serial algorithms are less complicated than asynchronous ones and thus the syntax for asynchronous algorithms is more complicated.

2.3.5 Godis

Godis is a dialogue system that was developed with Trindikit. It was built to explore the notion of Questions Under Discussion (QUD). Within this framework it applied grounding and question and task accommodation. It also uses dialogue plans.

Questions under discussion can be questions that have been asked but not yet answered. They can also be questions that are possible to ask in a given context. If, for example, you are at a travel agency a question about how much it costs to fly to London is a question under discussion even before it has been asked, and in fact the question may be answered without it being asked explicitly.

Sometimes questions can be answered without having been asked first. In these cases the implicit question is seen as a presupposition to the answer. The question is accommodated, that is it is added to the information state as if it had been asked just before the answer is made. If we have the dialogue fragment

```
S: When do you want to go?
U: In April. As cheap as possible.
```

U's utterance does not only contain an answers to S's question. It also answers the implicit question 'How much can it cost?'. This question can be accommodated, and then U's utterance will be valid.

Information State

The information state in Godis looks like this (Larsson, 2000):

$$
\text{is} : \left[\begin{array}{ll}
\text{PRIVATE} : & \left[\begin{array}{ll}
\text{PLAN} & : \text{StackSet(Action)} \\
\text{AGENDA} & : \text{Stack(Action)} \\
\text{BEL} & : \text{Set(Proposition)} \\
\text{TMP} & : \left[\begin{array}{ll}
\text{BEL} & : \text{Set(Proposition)} \\
\text{QUD} & : \text{Stack(Question)} \\
\text{LU} & : \left[\begin{array}{ll} \text{SPEAKER} & : \text{Participant} \\ \text{MOVES} & : \text{assocSet(Move,Bool)} \end{array}\right] \\
\text{NIM} & : \text{StackSet(Move)}
\end{array}\right]
\end{array}\right] \\
\text{SHARED} : & \left[\begin{array}{ll}
\text{BEL} & : \text{Set(Proposition)} \\
\text{QUD} & : \text{Stack(Question)} \\
\text{LU} & : \left[\begin{array}{ll} \text{SPEAKER} & : \text{Participant} \\ \text{MOVES} & : \text{assocSet(Move,Bool)} \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

The IS is a record with two parts. PRIVATE contains information that only the system has access to. SHARED contains information that is considered grounded, that is the information has been established previously in the conversation. This structure reflects the idea behind the dialogue games approach, where the system keeps track of what it thinks that the other participant (the user) knows.

The field PLAN in PRIVATE contains a stack of actions that the system intends to carry out. It is possible that the user raises one of these actions before the system. Then this action is removed from the stack, the task is accommodated. This is possible since PLAN is not a regular stack but a stackset (a stack that you can perform set operations on). The AGENDA field contains short term goals and obligations. BEL contains a set of propositions that the system believes is true. The TMP record is a mirror of SHARED and is used for storing information that has not yet been grounded.

BEL in SHARED contains a set of propositions that the system assumes is true for the user as well. QUD is a stack of questions under discussion. LU stands for last utterance. Speaker is either system or user and MOVES contains the move or moves connected to the latest utterance. The moves are connected to a boolean value that indicates whether the move has been integrated or not.

Moves

The following moves are used:

ask(Q) , Q is a question

answer(A) , A is an answer

reqRep(R) , R can be relevance or understanding

repeat(M) , M is a move

greet

quit

Each move is connected to a set of phrases via the lexicon module. The user's utterances are converted to moves, and the system's moves are converted to utterances. If the user for example says "What does it cost?" this could be converted to a move ask(X^price(X)) and the system could answer answer(price(2300)) which will be converted to "It will cost 2300kr". It is possible to run the system and communicate directly with moves instead of text. Example from the Godis manual, in a cell phone domain:

```
S: [greet]
U: [greet]
S: [ask([task(phonebook),task(messages)])]
U: [answer(task(phonebook))]
S: [ask([task(search_phonebook),task(add_new_number)])]
U: [answer(task(search_phonebook))]
S: [ask(X^name(X))]
U: [answer(name(pelle))]
```

Plans and actions

A plan consists of several actions that need to be carried out in a certain order. A plan can also have constructs such as if-then and case. A task is accomplished by carrying out a plan. Plans, tasks and actions are all part of the domain knowledge of Godis.

Actions are connected to dialogue moves. Here are some examples:

findout(Q), find the answer to the question Q. This action is connected to an ask-move. The system asks a question to find out the answer.

respond(Q), answer the question Q. This action is connected to an answer-move.

inform(P), inform the user of something with the help of an inform-move.

greet, greet-move.

An example of a plan from the Godis manual:

```
plan(search_phonebook,
     [findout(X^name(X)),
      findout(call),
      if_then(call,
              if_then(name(N),
                      [call_name(N))])),
      forget,
      exec(phonebook)
     ]).
```

Rules

There are nine classes of rules that are invoked by the update algorithm. There is also one class of selection rules, select. Some interesting classes are grounding, integrate and accommodate. The rules that are described below have the following components:

Rule is the name of the rule. *Class* is the name of the rule class that the rule belongs to. *Pre* stands for preconditions. The preconditions consist of a number of checks that are performed on the information state. *Eff* stands for Effects. The effects are queries and updates that are performed on the information state.

Grounding rules are used to decide whether the user has understood the system's output or not. There are two strategies that can be employed here. The system can optimistically assume that as soon as output is given, the user has understood unless he says he did not. The other strategy is to not assume anything unless feedback has been received from the user. A rule that is used for the first strategy looks like this:

$$
\begin{array}{l}
Rule : assumeSysMovesGrounded \\
Class : grounding \\
Pre : \left\{ \; latest_speaker\$ == sys \right. \\
Eff : \left\{ \begin{array}{l} set\#rec(SHARED.LU.SPEAKER, sys) \\ clear\#rec(SHARED.LU.MOVES) \\ forall(in(LATEST_MOVES, Move) \\ add\#rec(SHARED.LU.MOVES, Move, false)) \end{array} \right.
\end{array}
$$

As soon as the system makes an utterance, add the relevant information to SHARED.

Integration rules are used to integrate moves into the information state. An integration rule can look like this:

$$
\begin{array}{l}
Rule : integrateUsrAnswer \\
Class : Integrate \\
Pre : \left\{ \begin{array}{l} valRec(SHARED.LU.SPEAKER, usr) \\ assocRec(SHARED.LU.MOVES, answer(R), false) \\ fstRec(SHARED.QUD, Q) \\ domain :: relevant_answer(Q, R) \\ reduce(Q, R, P) \end{array} \right. \\
Eff : \left\{ \begin{array}{l} popRec(SHARED.QUD) \\ addRec(SHARED.BEL, P) \\ setAssocRec(SHARED.LU.MOVES, answer(R), true) \end{array} \right.
\end{array}
$$

If the latest speaker was the user, the last move was an answer and it was relevant, pop the question preceding the answer from the QUD and add the answer to SHARED.BEL.

Accommodation rules are used when the user raises a topic that is a part of the system's dialogue plan but not at the top of the stack. The system will push the topic on to the QUD, delete it from the plan and then continue with the top of the stack.

$$
\begin{array}{l}
Rule : accommodateQuestion \\
Class : accommodate \\
Pre : \left\{ \begin{array}{l}
valRec(SHARE.LU.SPEAKER, usr) \\
moveInRec(SHARED.LU.MOVES, answer(A)) \\
not(lexicon :: yn_answer(A)) \\
valIntegrateFlag(answer(A), false) \\
inRec(PRIVATE.PLAN, raise(Q) \\
domain :: relevant_answer(Q, A)
\end{array} \right. \\
Eff : \left\{ \begin{array}{l}
delRec(PRIVATE.PLAN, raise(Q)) \\
pushRec(SHARED.QUD, Q)
\end{array} \right.
\end{array}
$$

Algorithms

There are three algorithms in Godis, the Control, Update and Select algorithms. The control algorithm is written in Control-ADL and the other two in DME-ADL. The Select algorithm just calls one rule class, select, when the select algorithm is run it applies the first select rule that fulfills its preconditions. The Control and Update algorithms are a bit more interesting.

Control algorithm

```
repeat ( [reset,
          repeat (  [ select,
                      generate,
                      output,
                      update,
                      print_state,
                      test( program_state == run ),
                      input,
                      interpret,
                      print_state,
                      update,
                      print_state
                      ] )
                  ] )
```

select, generate, output, input and interpret invoke the respective modules. print_state prints the information state (if enabled) for debugging purposes. test is used to stop the algorithm if the program has shut down (the user quit). The basic flow of the system is that first the system selects a move depending on the information state, generate and print an utterance, update the IS, wait for input from the user, parse it when it comes, update again and then start over with select. When the system is started the select module will select a greet move. Later the selection will depend on previous input from the user.

Update algorithm

This algorithm is run every time the update module is called in the control algorithm.

```
if (latest_moves == failed)
                 then ([ repeat refill_agenda ])
                 else
                (   [ grounding,
                      repeat ( integrate or accomodate ),
                      if (latest_speaker == usr)
                         then [ repeat refill,
                                try database ]
                      else store ] )
```

Here rule classes are invoked. This algorithm is run every time the update module is called in the control algorithm.

Chapter 3

System Description

3.1 Requirements

3.1.1 General

The input to the system is text. The output is also text, mainly in the form of tables, but also as regular sentences, for example in clarification subdialogues.

3.1.2 Dialogue Structure

It should be possible for the system to handle subdialogues and meta communication to some extent. It should also be easy to extend this aspect of the system later. There will not be an explicit mechanism to handle the dialogue structure. The limited domain, the focus mechanisms and the fact that the human users tend to adapt to the system should ensure that the dialogue structure will be fairly simple. Also, the fact that the user usually has the initiative means that the structure is less complicated. If it were to be modelled formally it would be possible to use a finite state automaton. Now the structure is modelled implicitly through update and selection rules. The focus mechanisms could probably be used successfully with most strategies for dialogue structure.

3.1.3 Focus Structure

The system should be able to keep track of the focus structure of a dialogue and be able to deal with simple anaphora resolution. In this kind of system the focus often includes several objects as the subject of a dialogue is represented by tables. As the interaction usually involves tables containing several objects it can sometimes be difficult to know what the user is referring to.

Example:

```
U> show saab cheaper than 100000
S> +---------------+----------+------+----------+
   |manufacturer   |model     |year  |price     |
```

```
+---------------+----------+------+----------+
|saab           |900       |1985  |80000     |
|saab           |99        |1982  |20000     |
|saab           |9000      |1999  |95000     |
+---------------+----------+------+----------+
```

```
<>
U> show volvo
```

Which Volvos should be shown? All of them or just those that cost less than 100000? This is the kind of problems that the strategies discussed below deal with in different ways.

3.1.4 Customisation

It should be possible to customise the system to work with other similar domains, where the user is requesting information from a database in the form of tables, and the contents of the database has a structure that is based around predefined objects and their properties. The database and lexicon would be replaced, and a different strategy for the focus structure might be used.

3.2 Strategies

3.2.1 Intension and extension

The different strategies that are proposed reflect different ways to deal with focus structure. The queries sent to the database will be altered by focus changes. This means that the database will be queried every time the system needs to come up with an answer, and no objects need to be stored anywhere else.

A different way to handle focus is to save sets of objects and copy them between different parts of the dialogue. The difference between these two methods is that the former uses the intention of the user's utterance, while the latter uses the extension. Using the extension to determine the focus can be problematic when the focus contains several objects. A user could ask for objects that are not in the database. Then he could refer back to those (non-existing) objects to get some other objects with similar properties. If the extensions of the user's questions are used the system will not know that the first question is connected to the second since the system uses the relation between the sets of objects to determine this connection. For example, the database could contain one Volvo that costs 5000 and a few other that cost more than 10000. The user could then ask:

```
U> show volvos that cost less than 2000
S> There are no objects that match the query
<>
U> cheaper than 10000
```

the system should reply with

```
S> +---------------+----------+------+----------+
   |manufacturer   |model     |year  |price     |
   +---------------+----------+------+----------+
   |volvo          |240       |1985  |5000      |
   +---------------+----------+------+----------+
```

In order to be able to give this answer the system must now that the user is referring to Volvo even though there are no Volvos in the answer to the first question.

3.2.2 Question Revision

A question here is a request for information from the user, not necessarily in the form of a question. It could be an order to show the information. This request is then turned into a database query.

Questions are represented in the following way:

```
[X,Y,Z]^[property1(X,Y),property2(X,Z)]
```

X represents the id of a car and Y and Z represent properties of a car. The property-predicates always take two arguments, an object id and a value for the property. Property values can be instantiated:

```
[X,Y]^[manufacturer(X,saab),price(X,Y)]
```

This corresponds to a question that requests all cars of the make Saab and the price of those cars. The second argument of manufacturer is instantiated, which means that the contents of the answer will be limited to those objects that have the property of being Saabs. The second argument of price is uninstantiated, which means that the contents of the answer will not be limited, but the price property will be shown. The answer to this question is given in the form of a table:

```
+---------------+----------+------+----------+
|manufacturer   |model     |year  |price     |
+---------------+----------+------+----------+
|saab           |900       |1985  |80000     |
|saab           |99        |1982  |20000     |
|saab           |9000      |1999  |95000     |
|saab           |93        |2001  |205000    |
+---------------+----------+------+----------+
```

When the system receives another question it must determine how the new question is related to the previous one. This is where the focus structure is constructed.

Example:

Question 1 (current question), a request for all Saabs:

```
[X]^[manufacturer(X,saab)]
```

Gives the table

```
+---------------+----------+------+
|manufacturer   |model     |year  |
+---------------+----------+------+
|saab           |900       |1985  |
|saab           |99        |1982  |
|saab           |9000      |1999  |
|saab           |93        |2001  |
+---------------+----------+------+
```

Question 2 (new question), a request for price:

```
[X]^[price(X,Y)]
```

Revised question:

```
[X,Y]^[manufacturer(X,saab),price(X,Y)]
```

Gives

```
+---------------+----------+------+----------+
|manufacturer   |model     |year  |price     |
+---------------+----------+------+----------+
|saab           |900       |1985  |80000     |
|saab           |99        |1982  |20000     |
|saab           |9000      |1999  |95000     |
|saab           |93        |2001  |205000    |
+---------------+----------+------+----------+
```

In the example above it seems reasonable to add information about the price of the cars to the answer. But what if the next question is manufacturer (X,volvo)? Should the Saabs still be a part of the answer? Should the answer only contain Volvo? Volvo with price?

The trivial way to handle question revision is to always add all new information to the answer and never remove anything. This is not very practical, since the answers would become bigger and bigger, while the reason for making a system like this is to narrow the answers down for the user.

Below I will present three different strategies for question revision that will deal with focus structure. The first one, the set strategy is implemented in the Bilar system.

3.2.3 The Set

The main question is seen as a set of properties. When a new question is integrated new properties are added and existing ones replaced.

Example:

Question 1:

```
[X]^[manufacturer(X,saab)]
```

Question 2:

```
[X,Y]^[price(X,Y)]
```

Since price is a new property it is added to the question.

Revised question:

```
[X,Y]^[manufacturer(X,saab),price(X,Y)]
```

The answer to this question contains the same set of objects as the answer to Question 1, but also shows the price. Now let us add a third question:
Question 3:

```
[X]^[manufacturer(X,volvo)]
```

Since manufacturer is already in the question the property is replaced.

New revised question:

```
[X,Y]^[manufacturer(X,volvo),price(X,Y)]
```

The answer to this question contains a different set of objects than the answer to Question 2.

3.2.4 The Stack

Each new question is put on a stack. When an instantiated question about an instantiated property that already exists is asked all questions in the stack that came after the previous question about that property are removed, and the current question replaces the old one of the same kind. Uninstantiated questions are treated in the same way as previously.

Example:

Question 1:

```
[X]^[manufacturer(X,saab)]
```

The question is pushed onto the stack.
Question 2:

```
[X]^[price(X,50000)]
```

This means that price is smaller than 50000. Price is instantiated, but there are no previous questions about price, so this question is also pushed onto the stack which now looks like this:

```
[[X]^[price(X,50000)],[X]^[manufacturer(X,saab)]]
```

Revised question:

```
[X]^[manufacturer(X,saab),price(X,50000)]
```

Question 3:

```
[X]^[manufacturer(X,volvo)]
```

Since this question is instantiated and there already is an instantiated question about manufacturer on the stack the first element (`[X]^[price(X,50000)]`) will be popped from the stack which now looks like this:

```
[[X]^[manufacturer(X,saab)]]
```

Since the popped question is not of the same kind as the current question the next question on the stack (`[X]^[manufacturer(X,saab)]`) is popped (the stack is now empty). Since this question is of the same kind we stop popping question and instead push the current question (`[X]^[manufacturer(X,volvo)]`) onto the stack.

New revised question:

```
[X]^[manufacturer(X,volvo)]
```

3.2.5 The Subdialogue

When a question about an existing property is asked the system asks the user what he wants to know.

Example:

Question 1:

```
[X]^[manufacturer(X,saab)]
```

Question 2:

```
[X]^[price(X,50000)]
```

Revised question:

```
[X,Y]^[manufacturer(X,saab),price(X,50000)]
```

Question 3:

```
[X]^[manufacturer(X,volvo)]
```

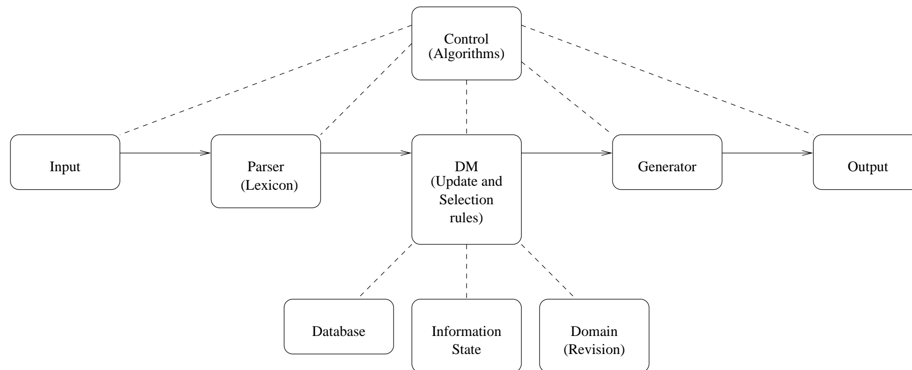
System output:

Do you want to see all volvos?

3.3 Implementation

From the start the implementation was supposed to be adapted from Godis. However, the nature of the kind of dialogues that were supposed to take place turned out to require a quite different approach to many aspects of dialogue management than that taken in Godis. A lot of the mechanisms for dealing with these aspects are implemented in the domain and lexicon modules. The information state and update and selection modules show traces from Godis that might not be necessary to make the system work as intended. On the other hand, since this is not a fully fledged dialogue system but rather a prototype for experimenting with focus structure the seemingly unnecessarily complex DME modules could be useful if one would like to expand the system in the future in order to include more complete functionality. The next chapter contains a more extensive discussion of this subject.

Below is an overview of the architecture of the system:



3.3.1 Information State

The information state looks like this:

$$
\text{is} : \left[\begin{array}{lll}
\text{PRIVATE} & : & \left[\begin{array}{ll} \text{AGENDA} & : \text{Stack(Action)} \\ \text{BEL} & : \text{Set(Proposition)} \end{array}\right] \\
\text{SHARED} & : & \left[\begin{array}{ll} \text{ANSWERS} & : \text{Set(Proposition)} \\ \text{QUESTIONS} & : \text{Stack(Question)} \\ \text{QUD} & : \text{Stack(Question)} \\ \text{LU} & : \left[\begin{array}{ll} \text{SPEAKER} & : \text{Participant} \\ \text{MOVES} & : \text{assocSet(Move,Bool)} \end{array}\right] \end{array}\right]
\end{array}\right]
$$

Like in Godis the information state is divided into two records, Private and Shared. Unlike in Godis, there are no plans.

ANSWERS: the latest answer from the system.

QUESTIONS: a stack of questions from the user. This is not used in the implementation of the set strategy, but could be used with the stack- and sub-dialogue strategies (if they were to be implemented).

QUD: the latest question that was sent to the database, this is the question that will be revised.

AGENDA: what the system wants to do next. There are no plans (since it is a user-initiative system), just one action.

BEL: what the system believes the user wanted to know with the latest question. This is the same information as is stored in ANSWER, but here it can be stored before it has been shown to the user.

SPEAKER: the latest speaker, can be user or system.

MOVES: move(s) in the latest utterance, and information about whether they are integrated or not.

3.3.2 Algorithms

The control algorithm is basically the same as in Godis. The general sequence output, manage, input, manage etc. seems to be the most logical one to use for any kind of dialogue system (at least with two participants). Selection also works in the same way as in Godis. There is only a call for the select class. The update algorithm is very simple:

```
[grounding,
 integrate,
 if (latest_speaker == usr)
    then ([try database])
    else ([])
]
```

First, take care of grounding, then integrate the move and last, if it was the user who made the last contribution check the database. Note the try, if the user for some reason said something that was not meant as a request for information from the database the query will fail but the system will continue running as intended.

3.3.3 Update Rules

The most important update rules are the ones for integrating questions from the user and for accessing the database.

Integration

$$
\begin{array}{l}
Rule : integrateUsrAsk \\
Class : Integrate \\
Pre : \left\{ \begin{array}{l} val\#rec(SHARED.LU.SPEAKER, usr) \\ assoc\#rec(SHARED.LU.MOVES, ask(Q), false) \\ fst\#rec(SHARED.QUD, B) \end{array} \right. \\
Eff : \left\{ \begin{array}{l} set_assoc\#rec(SHARED.LU.MOVES, ask(Q), true) \\ push\#rec(SHARED.QUESTIONS, Q) \\ !(domain :: breviseq(B, Q, R)) \\ push\#rec(PRIVATE.AGENDA, respond(R)) \\ clear\#rec(SHARED.QUD) \\ push\#rec(SHARED.QUD, R) \end{array} \right.
\end{array}
$$

The conditions for applying this rule are that the latest speaker is the user and that the user's utterance was a question. The third condition is used for unification for the revision predicate operation. There is always a question on the QUD, so the condition always succeeds.

When the rule is applied first the move is integrated. Then the content of the users question is pushed on SHARED.QUESTIONS. The QUD is revised with the question Q. This is done by querying the Prolog-predicate breviseq/3 in the domain module. The system puts on its agenda a respond action in order to answer the (revised) question. Then the revised question takes the place of the old QUD.

Database

$$
\begin{array}{l}
Rule : queryDB \\
Class : Database \\
Pre : \left\{ \begin{array}{l} fst\#rec(SHARED.QUD, C) \end{array} \right. \\
Eff : \left\{ \begin{array}{l} clear\#rec(PRIVATE.BEL) \\ !(database :: consultDB(C, R)) \\ add\#rec(PRIVATE.BEL, R) \end{array} \right.
\end{array}
$$

The condition is for unification. C is unified with the question in QUD so that it can be sent to the database with the predicate ConsultDB. The answer is put in PRIVATE.BEL.

3.3.4 Selection Rules

The selection rules check the private agenda. If the top element is a respond action they check what is in the private belief and then answer with content of bel. There is also functionality for checking the size of the answers. If an answer contains many objects the system can ask the user if he wants to further specify his question. There are also selection rules for greet and quit.

3.3.5 Revision

Revision is used in two cases. The first is when an utterance is parsed. The second is when the QUD is updated. The basic revision mechanism works like this:

The two questions,

```
[X]^[manufacturer(X,saab)]
```

and

```
[X,Y]^[price(X,Y)]
```

will be revised into

```
[X,Y]^[manufacturer(X,saab),price(X,Y)]
```

Here all that is needed is to put the two predicates in a list and to put X and Y in a list ahead of the predicates. You have to make sure the X is the same in both predicates. It gets a little more complicated if a third question is introduced:

```
[X,Y]^[rust(X,Y)]
```

The X is still the same in all predicates. The Y in rust(X,Y) is not the same variable as in price(X,Y). It will have to be renamed and added to the list of variables:

```
[X,Y,Z]^[manufacturer(X,saab),price(X,Y),rust(X,Z)]
```

3.3.6 Lexicon

The parser identifies keywords/phrases. Each word is then exchanged for a question. Then the questions are appended using the revision mechanism. Each keyword or phrase can be connected to either an instantiated or uninstantiated question. The keyword manufacturer is connected to the question

```
[X,Y]^[manufacturer(X,Y)]
```

while the keyword saab is connected to the question

```
[X]^[manufacturer(X,saab)]
```

When numerical values are involved there are more words than one that need to be connected to one question. For example, the question

```
[X]^[rust(X,4)]
```

will be connected to the keyword rust or rust protection or something similar and 4 will be connected to the number 4 which would have appeared directly after (after all non-keywords have been filtered out). It is assumed that the user wants all objects with 4 or higher.

3.3.7 Database

The information in the database is stored as predicates. When a query is sent to the database, it extracts all objects that match the query. The cars are stored in the following form:

```
car(1,volvo,240,1985,large,3,3,2,5000).
```

The first number is the id of the car. The rest of the values are properties of the car. The properties are accessed with their respective predicates:

```
size(X,Y):- car(X,_,_,_,Y,_,_,_,_).
```

There is one predicate of this kind for each property that a car can have. The database is accessed from the update module with a predicate of this kind:

```
consultDB([X,Y]^For,Ans):-
        setof((X,Y),(satisfy(For)),Ans).
```

There are several of these predicates, which one is called depends on the number of uninstantiated variables that are in the query. The output of the predicate is a list of n-tuples, one tuple for each car that satisfies the conditions in For, where the size of n is determined by the number of variables. In the case above n would equal 2 since there are two variables (X and Y). The variable For consists of a number of predicates which match properties.

Chapter 4

Discussion

4.1 Strategies

Here I will discuss the strategies that were presented in the previous chapter. I will give an account of the strengths and weaknesses of the respective strategies.

4.1.1 General

In order to know which strategies work best in which cases data would need to be collected and analysed, for instance using Wizard of Oz experiments. It would probably be a good idea to use a combination of strategies in a large scale system. Another possibility is to implement all the strategies and then test them to see which ones work best. Some properties are intrinsic for the strategies.

4.1.2 Comparison of strategies

The Set strategy

The set strategy is easiest to implement. A problem is that the user may not see all objects he wants to see in some cases:

```
S> Welcome to Cars.
<>
U> show volvo with price
S>  +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |850       |1993  |50900     |
    |volvo          |440       |1990  |15000     |
    |volvo          |v70       |2000  |180000    |
    +---------------+----------+------+----------+

<>
U> show saab
```

```
S>  +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |saab           |900       |1985  |80000     |
    |saab           |99        |1982  |20000     |
    |saab           |9000      |1999  |95000     |
    |saab           |93        |2001  |205000    |
    +---------------+----------+------+----------+

<>
U> show cheaper than 30000
S>  +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |saab           |99        |1982  |20000     |
    +---------------+----------+------+----------+

<>
U> show volvo
S>  +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |440       |1990  |15000     |
    +---------------+----------+------+----------+

<>
U> bye
S> Thank you for visiting.
```

When the user requests Volvo the second time he might want to see all Volvos. Now only those costing less than 30000 are shown.

The Stack strategy

The stack strategy is a bit harder to implement. It is safer, the user will not miss objects. On the other hand the user may need to request information more often than with the set strategy:

```
$U> show volvo
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |volvo          |240       |1985  |
    |volvo          |740       |1989  |
    |volvo          |850       |1993  |
    |volvo          |440       |1990  |
    |volvo          |v70       |2000  |
    +---------------+----------+------+
```

```
<>
$U> show those that cost less than  30000
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |440       |1990  |15000     |
    +---------------+----------+------+----------+

<>

$U> show saab
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |saab           |900       |1985  |
    |saab           |99        |1982  |
    |saab           |9000      |1999  |
    |saab           |93        |2001  |
    +---------------+----------+------+

<>

$U> show price
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |saab           |900       |1985  |80000     |
    |saab           |99        |1982  |20000     |
    |saab           |9000      |1999  |95000     |
    |saab           |93        |2001  |205000    |
    +---------------+----------+------+----------+
```

When the user requests Saabs he might want price to be shown at once.

The Subdialogue strategy

The subdialogue strategy is always safe, but could lead to an unwieldy interface:

```
$U> show volvo
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |volvo          |240       |1985  |
    |volvo          |740       |1989  |
    |volvo          |850       |1993  |
    |volvo          |440       |1990  |
    |volvo          |v70       |2000  |
    +---------------+----------+------+
```

```
<>
$U> show price
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |850       |1993  |50900     |
    |volvo          |440       |1990  |15000     |
    |volvo          |v70       |2000  |180000    |
    +---------------+----------+------+----------+

<>
$U> show saab
$S> do you want to see the price?
$U> yes
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |saab           |900       |1985  |80000     |
    |saab           |99        |1982  |20000     |
    |saab           |9000      |1999  |95000     |
    |saab           |93        |2001  |205000    |
    +---------------+----------+------+----------+

<>
```

4.2 Extensions

4.2.1 Focus

Strategies

As was mentioned earlier, only the set strategy has been implemented in the Bilar system. The set strategy is the simplest one. This is because you do not have to keep track of earlier utterances as in the stack strategy, and you do not have to add any checks and questions to the user as in the subdialogue strategy.

If the other two strategies, the stack and the subdialogue strategy, were to be implemented it could be done in a couple of different ways: Either the domain predicates could be changed. This would mean that the revision mechanisms would change, but the update rules could remain the same. Else the update rules could be changed and the revision could remain intact.

References

The system can not handle references to objects that were requested two or more steps back in a dialogue. It would be possible to implement this functionality. The system will need to recognise keywords that signal that the user is referring

to objects more than one step back (for example 'the other'). When such a keyword is spotted the system should go back the right number of steps in the questions stack in the information state (pop questions). The system should go back to the second to last of the instantiated questions. Example:

```
$U> show volvo
$S> +---------------+----------+------+
    |manufacturer   |model     |year  |
    +---------------+----------+------+
    |volvo          |240       |1985  |
    |volvo          |740       |1989  |
    |volvo          |850       |1993  |
    |volvo          |440       |1990  |
    |volvo          |v70       |2000  |
    +---------------+----------+------+

<>
$U> show price
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |850       |1993  |50900     |
    |volvo          |440       |1990  |15000     |
    |volvo          |v70       |2000  |180000    |
    +---------------+----------+------+----------+

<>
$U> show saab
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |saab           |900       |1985  |80000     |
    |saab           |99        |1982  |20000     |
    |saab           |9000      |1999  |95000     |
    |saab           |93        |2001  |205000    |
    +---------------+----------+------+----------+

<>
$U> show the other ones again
$S> +---------------+----------+------+----------+
    |manufacturer   |model     |year  |price     |
    +---------------+----------+------+----------+
    |volvo          |240       |1985  |5000      |
    |volvo          |740       |1989  |25000     |
    |volvo          |850       |1993  |50900     |
    |volvo          |440       |1990  |15000     |
    |volvo          |v70       |2000  |180000    |
    +---------------+----------+------+----------+
```

<>

This should work the same regardless of the strategy that is used, there might be a difference in which properties are shown though.

4.2.2 Meta communication

A lot of metacommunication is not implemented, questions about how the system works, error messages and so on. These kinds of interaction could be represented by their own moves.

4.2.3 Godis and Bilar

The main difference between Godis and Bilar is that in Godis the system usually takes the initiative, and the system tries to help the user to find one specific object that matches the users requirements. In Bilar the user almost always has the initiative, and the user specifies a number of properties that can match several objects. There are no plans in Bilar.

In Bilar there is a direct connection between the lexicon and the queries that are sent to the database. The database queries are built dynamically during the course of the conversation. There is no set goal that the system is trying to reach. As long as the user is asking questions the system tries to answer them.

It would be interesting to combine the strategies of Godis and Bilar. Instead of trying to find one specific object that matches the users requirements (Godis) the goal for the system could be to present a table with no more than five (or any number of) objects. If the user is not very active the system could use plans to find out what the user wants, or it could use accommodation if the user takes initiative, and when a suitable table has been constructed the system could scrap the rest of the plan. This would have the advantages of both Godis and Bilar. The more advanced lexicon and focus functionality of Bilar would be integrated with the more advanced dialogue management of Godis.

Bibliography

Austin, J. L. (1962). *How to do things with words.* Clarendon Press.

Bohlin, P., Bos, J., Larsson, S., Lewin, I., Matheson, C., and Milward, D. (1999). Survey of existing interactive systems. Technical report, Trindi. Deliverable D1.3.

Carlson, L. (1983). *Dialogue Games.* D. Reidel Publishing Company.

Jönsson, A. (1993). *Dialogue Management for Natural Language Interfaces.* PhD thesis, Linköping University.

Larsson, S. (2000). *GoDiS 1.2 Developers Manual Draft version.*

Larsson, S. (2002). *Issue-based Dialogue Management.* PhD thesis, Gothenburg University.

Larsson, S., Berman, A., Bos, J., Grönkvist, L., Ljunglöf, P., and Traum, D. (2000). *TrindiKit 2.0 Manual revised version.* Trindi. Deliverable D5.3.

Levinson, S. C. (1983). *Pragmatics.* Cambridge University Press.

Appendix A

Code

A.1 Database

```
/**************************************************************************

         name: database_bilar.pl


**************************************************************************/

:- module( database_bilar, [consultDB/2]).

% loaded by module database.pl

:- use_module( library(lists), [ member/2, select/3, append/3 ] ).

:- dynamic domain/2.

satisfy([P|Ps]):-
        P=..[pris,A1,A2],
        var(A1),
        integer(A2),
        Q=..[pris,B1,B2],
        Q,
        Q=..[pris,A1,B2],
        B2=<A2,
        satisfy(Ps).

satisfy([P|Ps]):-
        P=..[pris,A1,A2],
        nonvar(A1),
        integer(A2),!,
        Q=..[pris,A1,B2],
        Q,
```

```
        B2=<A2,
        satisfy(Ps).

satisfy([P|Ps]):-
        P=..[arsmodell,A1,A2],
        var(A1),
        integer(A2),
        Q=..[arsmodell,B1,B2],
        Q,
        Q=..[arsmodell,A1,B2],
        B2>=A2,
        satisfy(Ps).

satisfy([P|Ps]):-
        P=..[arsmodell,A1,A2],
        nonvar(A1),
        integer(A2),!,
        Q=..[arsmodell,A1,B2],
        Q,
        B2>=A2,
        satisfy(Ps).

satisfy([P|Ps]):-
        P=..[P1,A1,A2],
        var(A1),
        integer(A2),
        A2=<5,
        Q=..[P1,B1,B2],
        Q,
        Q=..[P1,A1,B2],
        B2>=A2,
        satisfy(Ps).

satisfy([P|Ps]):-
        P=..[P1,A1,A2],
        nonvar(A1),
        integer(A2),
        A2=<5,
        Q=..[P1,A1,B2],
        Q,
        B2>=A2,
        satisfy(Ps).

satisfy([P|Ps]):-
        P,
        satisfy(Ps).

consultDB([X,Y,Z,W,R]^For,Ans) :-
        setof((X,Y,Z,W,R),(satisfy(For)),Ans).
```

```
consultDB([X,Y,Z,W]^For,Ans) :-
        setof((X,Y,Z,W),(satisfy(For)),Ans).

consultDB([X,Y,Z]^For,Ans) :-
        setof((X,Y,Z),(satisfy(For)),Ans).

consultDB([X,Y]^For,Ans):-
        setof((X,Y),(satisfy(For)),Ans).

consultDB([X]^For,Ans):-
        setof((X,Y),(satisfy(For)),Ans1),
        rensa(Ans1,Ans).

consultDB([X]^For,Ans):-
        setof(X,(satisfy(For)),Ans).

rensa([(A,B)|As],[A|Ans]):-
        rensa(As,Ans).
rensa([(A,B)],[A]).

%databas

marke(volvo).
marke(opel).
marke(saab).

bil(1,volvo,240,1985,stor,3,3,2,5000).
bil(2,volvo,740,1989,stor,3,4,3,25000).
bil(3,volvo,850,1993,stor,4,4,3,50900).
bil(4,volvo,440,1990,mellan,2,3,2,15000).
bil(5,volvo,v70,2000,stor,5,5,4,180000).

bil(6,opel,kadett,1983,mellan,1,1,1,30000).
bil(7,opel,astra,1996,mellan,3,3,3,45000).
bil(8,opel,corsa,1999,sma,3,2,3,60000).
bil(9,opel,omega,1998,stor,3,4,2,76000).

bil(10,saab,900,1985,stor,5,5,5,80000).
bil(11,saab,99,1982,mellan,2,2,2,20000).
bil(12,saab,9000,1999,stor,4,4,4,95000).
bil(13,saab,93,2001,mellan,4,5,4,205000).

bil(14,bmw,320,1994,mellan,5,4,4,56000).
bil(15,bmw,520,1999,stor,5,5,4,189000).
bil(16,bmw,m5,2000,stor,5,5,4,450000).

bil(17,mercedes,190,1984,stor,4,3,4,23000).
bil(18,mercedes,230,1999,stor,5,5,4,123000).
bil(19,mercedes,aklass,2000,sma,4,3,4,99000).
```

```
bil(20,toyota,corolla,1973,mellan,2,1,1,12000).
bil(21,toyota,starlet,1999,sma,4,3,3,67000).
bil(22,toyota,lexus,2000,stor,4,5,4,267000).

bil(23,mazda,323,1994,mellan,4,3,3,23000).
bil(24,mazda,121,1995,sma,3,2,2,20000).

storlek(X,Y):- bil(X,_,_,_,Y,_,_,_,_).

drift(X,Y):- bil(X,_,_,_,_,Y,_,_,_).

krock(X,Y):- bil(X,_,_,_,_,_,Y,_,_).

marke(X,Y):- bil(X,Y,_,_,_,_,_,_,_).

pris(X,Y):- bil(X,_,_,_,_,_,_,_,Y).

modell(X,Y):- bil(X,_,Y,_,_,_,_,_,_).

arsmodell(X,Y):- bil(X,_,_,Y,_,_,_,_,_).

rost(X,Y):- bil(X,_,_,_,_,_,_,Y,_).
```

A.2 Domain

```
/**************************************************************************

         name: domain_bilar.pl


**************************************************************************/

:- module( domain_bilar, [breviseq/3, bigans/2] ).

:- use_module( library(lists), [member/2, select/3, append/3,
substitute/4, last/2, delete/3] ).

/*
Revision of q
*/

breviseq(B,X^[],B).

breviseq(B,X^[C|Cs],Rs):-
        reviseq(B,X^[C],R),
```

```
        fixvar(X^Cs,Y^Cs),
        breviseq(R,Y^Cs,R2),
        fixvar(R2,Rs).

reviseq(B,C,R):-
        inq(B,C,O,N),
        subq(O,B,N,R).

reviseq([X|Xs]^B,[Y1,Y2]^[C],Ys^R):-
        C =..[P,A1,A2],
        N =..[P,X,A2],
        append(B,[N],R),
        append([X|Xs],[Y2],Ys).

reviseq([X|Xs]^B,[Y]^[C],[X|Xs]^R):-
        C =..[P,A1,A2],
        N =..[P,X,A2],
        append(B,[N],R).


inq([X|Xs]^[Q|Qs],Y^[C],Q,N):-
        Q =..[P,X,Arg2],
        C =..[P,Arg3,Arg4],
        N =..[P,X,Arg4].

inq(X^[Q|Qs],Y^[C],P,N):-
        inq(X^Qs,Y^[C],P,N).

subq(O,X^B,N,X^R):-
        substitute(O,B,N,R).

fixvar(F1,F):-
        fixvar1(F1,F2),
        fixvar2(F2,F).

fixvar1([X|L]^S,[X|Y]^S):-
        vq(L,S,Y).

fixvar2([X|L]^S,[X|Y]^S):-
        va(S,Y).

va([],[]).

va([S|Ss],[V|Y]):-
        S=..[P,X,V],
        var(V),
        va(Ss,Y).

va([S|Ss],Y):-
        va(Ss,Y).
```

```
vq([],S,[]).

vq([V|Vs],S,[V|L]):-
        vqs(V,S),
        vq(Vs,S,L).

vq([V|Vs],S,L):-
        \+vqs(V,S),
        vq(Vs,S,L).

vqs(V,[S|Ss]):-
        S=..[P,X,V2],
        V\==X,
        V==V2.

vqs(V,[S|Ss]):-
        vqs(V,Ss).

/*
Big answers
*/

bigans(R,L):-
        length(R,L),
        L>20.
```

A.3 Lexicon

```
/***************************************************************************

         name: lexicon_bilar_svenska.pl
      version:
  description:
       author:

***************************************************************************/

:- module( lexicon_bilar_svenska, [output_form/2, input_form/2]).

:- use_module( library(lists), [ member/2, select/3, append/3 ] ).
:- use_module(library(charsio)).

parse([W1,W2|Ws],L):-
        integer(W2),
        n(N,W1,W2),
        parse(Ws,L1),
```

```
        breviseq(N,L1,L).

parse([],[]^[]).

parse([W|Ws],L):-
        n(N,W),
        parse(Ws,L1),
        breviseq(N,L1,L).

n([X,Y]^[storlek(X,Y)],storlek).
n([X,Y]^[storlek(X,Y)],storleksklass).
n([X,Y]^[storlek(X,Y)],stora).
n([X,Y]^[storlek(X,Y)],stor).

n([X]^[storlek(X,stor)],storbilar).

n([X]^[storlek(X,mellan)],mellanklassbilar).

n([X]^[storlek(X,sma)],småbilar).

n([X,Y]^[drift(X,Y)],driftsäkerhet).
n([X,Y]^[drift(X,Y)],tillförlitlighet).
n([X,Y]^[drift(X,Y)],drift).
n([X,Y]^[drift(X,Y)],driftsäkra).
n([X,Y]^[drift(X,Y)],driftsäker).
n([X,Y]^[drift(X,Y)],tillförlitlig).
n([X,Y]^[drift(X,Y)],tillförlitliga).

n([X]^[drift(X,Y)],drift,Y).
n([X]^[drift(X,Y)],driftsäkerhet,Y).
n([X]^[drift(X,Y)],tillförlitlighet,Y).

n([X,Y]^[krock(X,Y)],krocksäkerhet).
n([X,Y]^[krock(X,Y)],krockskydd).
n([X,Y]^[krock(X,Y)],krock).
n([X,Y]^[krock(X,Y)],krocksäker).
n([X,Y]^[krock(X,Y)],krocksäkra).
n([X,Y]^[krock(X,Y)],säker).
n([X,Y]^[krock(X,Y)],säkra).

n([X]^[krock(X,Y)],krocksäkerhet,Y).
n([X]^[krock(X,Y)],krockskydd,Y).
n([X]^[krock(X,Y)],krock,Y).

n([X,Y]^[marke(X,Y)],märke).
n([X,Y]^[marke(X,Y)],tillverkare).

n([X]^[marke(X,saab)],saab).
n([X]^[marke(X,volvo)],volvo).
n([X]^[marke(X,opel)],opel).
```

```
n([X]^[marke(X,bmw)],bmw).
n([X]^[marke(X,mercedes)],mercedes).
n([X]^[marke(X,toyota)],toyota).
n([X]^[marke(X,mazda)],mazda).

n([X,Y]^[pris(X,Y)],pris).
n([X,Y]^[pris(X,Y)],kostnad).
n([X,Y]^[pris(X,Y)],kostar).

n([X]^[pris(X,Y)],pris,Y).
n([X]^[pris(X,Y)],kostar,Y).
n([X]^[pris(X,Y)],billigare,Y).

n([X,Y]^[modell(X,Y)],modell).
n([X,Y]^[modell(X,Y)],modellbeteckning).

n([X]^[modell(X,240)],240).
n([X]^[modell(X,740)],740).
n([X]^[modell(X,850)],850).
n([X]^[modell(X,440)],440).
n([X]^[modell(X,v70)],v70).
n([X]^[modell(X,kadett)],kadett).
n([X]^[modell(X,astra)],astra).
n([X]^[modell(X,corsa)],corsa).
n([X]^[modell(X,omega)],omega).
n([X]^[modell(X,900)],900).
n([X]^[modell(X,99)],99).
n([X]^[modell(X,9000)],9000).
n([X]^[modell(X,93)],93).
n([X]^[modell(X,320)],320).
n([X]^[modell(X,520)],520).
n([X]^[modell(X,m5)],m5).
n([X]^[modell(X,190)],190).
n([X]^[modell(X,230)],230).
n([X]^[modell(X,aklass)],aklass).
n([X]^[modell(X,corolla)],corolla).
n([X]^[modell(X,starlet)],starlet).
n([X]^[modell(X,lexus)],lexus).
n([X]^[modell(X,323)],323).
n([X]^[modell(X,121)],121).

n([X,Y]^[arsmodell(X,Y)],årsmodell).
n([X,Y]^[arsmodell(X,Y)],år).
n([X,Y]^[arsmodell(X,Y)],tillverkningsår).
n([X,Y]^[arsmodell(X,Y)],gammal).
n([X,Y]^[arsmodell(X,Y)],gamla).

n([X]^[arsmodell(X,Y)],årsmodell,Y).
n([X]^[arsmodell(X,Y)],år,Y).
n([X]^[arsmodell(X,Y)],tillverkningsår,Y).
```

```
n([X]^[arsmodell(X,Y)],nyare,Y).

n([X,Y]^[rost(X,Y)],rostskydd).
n([X,Y]^[rost(X,Y)],rost).
n([X,Y]^[rost(X,Y)],rostbenägenhet).
n([X,Y]^[rost(X,Y)],rostig).
n([X,Y]^[rost(X,Y)],rostiga).

n([X]^[rost(X,Y)],rostskydd,Y).
n([X]^[rost(X,Y)],rost,Y).
n([X]^[rost(X,Y)],rostbenägenhet,Y).

keywords([W|Ws],[W|Ws2]):-
         n(_,W),
         keywords(Ws,Ws2).

keywords([W|Ws],[W|Ws2]):-
         n(_,W,_),
         keywords(Ws,Ws2).

keywords([W|Ws],[Y|Ws2]):-
         name(W,X),
         name(Y,X),
         integer(Y),
         keywords(Ws,Ws2).

keywords([W|Ws],Ws2):-
        keywords(Ws,Ws2).

keywords([],[]).

%input
input_form([hejdå],quit).

input_form([ja],answer(ja)).
input_form([nej],answer(nej)).

input_form(S,ask(Q)):-
        keywords(S,S1),
        parse(S1,Q).
```

A.4 Selection Rules

```
/*************************************************************************

         name: selection_rules.pl
```

```
*************************************************************************/

:- op(800, fx, '!').
:- op(850, xfx, and ).

rule_class( selectAnswer, select ).

rule_class( selectQuit, select ).
rule_class( selectGreet, select ).
rule_class( selectThank, select ).

%%% answer a question

rule( selectAnswer,
      [ fst#rec( private^agenda, respond(Q) ),
        not(in#rec(shared^answers,ja)),
        not(in#rec(shared^answers,nej)),
        in#rec( private^bel, R ),
        domain :: bigans(R,B)],
      [ set( next_moves, set( [ ask(B) ] ) ) ]
    ).

rule( selectAnswer,
      [ fst#rec( private^agenda, respond(Q) ),
        in#rec( private^bel, R ),
        not(domain :: bigans(R,B))],
      [ set( next_moves, set( [ answer(Q,R) ] ) ) ]
    ).

rule( selectAnswer,
      [ fst#rec( private^agenda, respond(Q) ),
        in#rec( private^bel, R ),
        in#rec(shared^answers,ja)],
      [ set( next_moves, set( [ answer(Q,R) ] ) ),
        clear#rec(shared^answers)]
    ).

rule(selectAnswer,
     [in#rec(shared^answers,nej)],
     [set(next_moves, set([answer(nej)])),
      clear#rec(shared^answers)]
    ).

%%% quit
rule( selectQuit,
       [ fst#rec( private^agenda, quit ) ],
       [ set( next_moves, set( [ quit ] ) )]
     ).
```

```
%%% greet
rule( selectGreet,
      [ fst#rec( private^agenda, greet ) ],
      [ set(next_moves, set([greet]) ) ]
     ).

%%% thank
rule( selectThank,
       [ fst#rec( private^agenda, thank ) ],
       [ set(next_moves, set([ thank ]) ) ]
     ).
```

A.5 Update Rules

```
/**************************************************************************

         name: update_rules.pl

**************************************************************************/

:- module(update_rules, [rule/3, rule_class/2]).

:- op(800, fx, ['!', not]).
:- op(850, xfx, ['$=', '$==', and, or] ).

rule_class( assumeSysMovesGrounded, grounding ).
rule_class( assumeUsrMovesGrounded, grounding ).

rule_class( integrateSysAnswer, integrate ).
rule_class( integrateSysAsk, integrate).

rule_class( integrateSysGreet, integrate ).
rule_class( integrateSysQuit, integrate ).
rule_class( integrateSysThank, integrate ).

rule_class( integrateUsrAsk, integrate ).
rule_class( integrateUsrAnswer, integrate ).

rule_class( integrateUsrGreet, integrate ).
rule_class( integrateUsrQuit, integrate ).

rule_class( queryDB, database ).

/*------------------------------------------------------------------------
       Grounding (cautious)
```

```
----------------------------------------------------------------------*/

% optimism about user recognizing system contributions

rule( assumeSysMovesGrounded,
       [ latest_speaker $== sys ],
       [ set#rec( shared^lu^speaker, sys),
         clear#rec( shared^lu^moves ),
         forall_do( in( latest_moves, Move),
                 add#rec( shared^lu^moves, Move, false) ) ] ).

% optimism about system recognizing user contributions
% (assume usr is optimistic about sys recognizing usr contrib.)

rule( assumeUsrMovesGrounded,
       [ latest_speaker $==  usr ],
       [ set#rec( shared^lu^speaker, usr ),
         clear#rec( shared^lu^moves ),
         forall_do( in(latest_moves, Move ),
                 add#rec( shared^lu^moves, Move, false) ) ] ).


/*----------------------------------------------------------------------
       Integration of the system's moves
       */

rule( integrateSysAnswer,
       [ val#rec( shared^lu^speaker, sys ),
         assoc#rec( shared^lu^moves,  answer(Q,R), false ),
         fst#rec( private^agenda, respond(Q) )],
       [ pop#rec( private^agenda ),
         add#rec( shared^answers, R ),
         set_assoc#rec( shared^lu^moves, answer(Q,R), true) ] ).

rule( integrateSysAnswer,
      [val#rec( shared^lu^speaker, sys ),
       assoc#rec( shared^lu^moves,  answer(nej), false )],
      [set_assoc#rec( shared^lu^moves, answer(nej), true) ]).

rule( integrateSysAsk,
      [val#rec( shared^lu^speaker, sys ),
       assoc#rec( shared^lu^moves, ask(Q), false )],
      [set_assoc#rec( shared^lu^moves, ask(Q), true)]).

rule( integrateSysGreet,
       [ val#rec( shared^lu^speaker, sys ),
         assoc#rec( shared^lu^moves,  greet, false ),
         fst#rec( private^agenda, greet ) ],
       [ pop#rec( private^agenda ),
         set_assoc#rec( shared^lu^moves, greet, true ),
```

```
         % forget previous information, if any
         clear#rec( shared^answers ) ] ).

rule( integrateSysQuit,
       [ val#rec( shared^lu^speaker, sys ),
         assoc#rec( shared^lu^moves,  quit, false ),
         fst#rec( private^agenda, quit ) ],
       [ pop#rec( private^agenda ),
         set( program_state, quit ),
         set_assoc#rec( shared^lu^moves, quit, true ) ] ).

rule( integrateSysThank,
       [ val#rec( shared^lu^speaker, sys ),
         assoc#rec( shared^lu^moves,  thank, false ),
         fst#rec( private^agenda, thank ) ],
       [ pop#rec( private^agenda ),
         set_assoc#rec( shared^lu^moves, thank, true ) ] ).


/*-----------------------------------------------------------------------
       Integration of the user's moves
           ask(Q), answer(R), reqRep, greet, quit
-----------------------------------------------------------------------*/

% normal question

rule(integrateUsrAsk,
     [val#rec( shared^lu^speaker, usr ),
      assoc#rec( shared^lu^moves,  ask(Q), false ),
      fst#rec(shared^qud,B)
     ],
     [set_assoc#rec( shared^lu^moves, ask(Q), true ),
      push#rec(shared^questions,Q),
      ! (domain :: breviseq(B,Q,R)),
      push#rec(private^agenda,respond(R)),
      clear#rec(shared^qud),
      push#rec(shared^qud,R)]).

rule(integrateUsrAnswer,
     [val#rec( shared^lu^speaker, usr ),
      assoc#rec( shared^lu^moves,  answer( R ), false )],
     [ set_assoc#rec( shared^lu^moves, answer( R ), true ),
       clear#rec(shared^answers),
       add#rec(shared^answers,R)]).

rule( integrateUsrGreet,
       [ val#rec( shared^lu^speaker, usr ),
         assoc#rec( shared^lu^moves,  greet, false ) ],
       [ set_assoc#rec( shared^lu^moves, greet, true ) ] ).
```

```
rule( integrateUsrQuit,
       [ val#rec( shared^lu^speaker, usr ),
         assoc#rec( shared^lu^moves,  quit, false ) ],
       [ push#rec( private^agenda, quit ),
         set_assoc#rec( shared^lu^moves,  quit, true ) ] ).

/*------------------------------------------------------------------------
       Database
------------------------------------------------------------------------*/

%%% ask the database a query

rule(queryDB,
     [fst#rec(shared^qud,C)
     ],
     [clear#rec(private^bel),
      ! (database :: consultDB(C,R)),
      add#rec(private^bel,R)]).
```