DiaView

A Dialogue System for Video Conferencing

Andreas Wallentin
andreas.wallentin@cling.gu.se
May 2004

Master's thesis in computational linguistics
Department of linguistics, Göteborg University
Spring semester 2004

Supervisor: Staffan Larsson

GÖTEBORG
UNIVERSITY

Abstract

Communication is something that I find is essential for all people. The purpose with this thesis is to try to show that dialogue systems could be used in order to simplify the human communication. I will also try to show that communication with a computer, that is controlling another application, could be simplified by using a dialogue system. I will use GoDiS to implement my dialogue system. My aim is to get an idea of whether dialogue systems could be used in communication applications.

This thesis is written in English.

Acknowledgements

Naturally I have to thank the usual persons.

My supervisor Staffan Larsson.

David Hjelm and Jacob Hallenborg for answering questions and showing me the web interface.

I would also like to thank my family and friends because they have been very neglected lately.

Contents

1 Introduction **1**

2 Background **3**

2.1 Dialogue systems . . . 3
2.2 TrindiKit . . . 4
2.3 GoDiS . . . 5
2.3.1 Total Information State . . . 7
2.3.2 Accomodation . . . 10
2.3.3 Dialogue move . . . 11
2.3.4 Resources . . . 11
2.3.5 Menu to Dialogue . . . 16

3 Method **19**

3.1 Analysis . . . 19
3.2 Implementation . . . 20

4 Analysis **21**

4.1 Task Analysis . . . 21
4.2 User analysis . . . 25
4.3 The application . . . 30
4.3.1 Potential problems with natural language . . . 30
4.3.2 Potential problems with selective hearing . . . 31
4.4 Requirements . . . 31
4.4.1 Usability requirements . . . 31
4.4.2 Functional requirements . . . 31

5 Design **33**

5.1 Video conference applications . . . 33
5.1.1 Used application . . . 33
5.1.2 Discarded applications . . . 34
5.2 Automatic Speech Recognition . . . 35

6 Implementation of Resources **37**
6.1 Device 37
6.2 Domain 38
6.3 Lexicon 40
6.3.1 Output_form 40
6.3.2 Input_form 40
6.3.3 Difficulties 41
6.4 Semsort 42

7 Results and Discussion **45**
7.1 Results 45
7.1.1 The over all system 45
7.1.2 MyPhone 46
7.1.3 Example dialogues 46
7.2 Discussion 48

8 Summary **49**

A User's guide to starting the system **I**

B Survey form **III**

List of Figures

2.1 Hot dogs 4
2.2 SJ's time table information 5
2.3 TrindiKit 6
2.4 GoDiS-VCR 7
2.5 Information State 9
2.6 QUD 10

5.1 MSN 34
5.2 MyPhone 35

Chapter 1

Introduction

You sit in a comfortable chair, you have a cup of coffee in one hand and a piece of cake in the other. You realize that you want to call a friend of yours, just to show off your coffee and cake. This will be an excellent time to try the new communication device that enables you to see the person with whom you are speaking. Since you have used communication devices earlier, you do not find it difficult to use.

Some time later when the coffee is long since cold and the piece of cake is drying on your plate, you finally understand how to call your friend. By then there is not much to show off...

Most of us have at some time used an application that we do not quite understand how to use. We do not understand all the buttons or all the different menus. Most people who have reasonable knowledge of similar applications think; *I see, I have used something that looks like this before. Then I must be able to use this too.*

Unfortunately this is not always the case which most users notice quite fast. There could be many menus that one has to go through, often menus that have one or more lower levels. This is something that could make the application hard to learn without reading a manual of some sort. And as most will admit, reading a manual is not something that is often done.

Many people with different forms of physical disabilities may have problems communicating with people "out of sight". At the same time they may have difficulties moving in order to meet other people in person. One solution to this problem is the use of a computer aid, a communication application. Although this solution has many advantages it still has some drawbacks. In order to use the application the user must be able to sit by, and see, the computer. Furthermore he must be able to use a keyboard or a mouse.

These are facts that make many possible users unable to use such applications. Many people with physical disabilities find it hard, if not impossible, to handle a keyboard or mouse.

The idea of using spoken dialogue systems as communiation aids is something that I was convinced could be done. The use of the voice is something that I find will be very useful in future computer applications. However, problems do not only arise with physical disabilities. One major problem is the language itself. Natural language, spoken and written both, is still far from being understood by computers.

In the beginning my idea was to make an application primarily for people with physical disabilities. Since it was the dialogue system I was interested in, I changed my approach. I wanted an application to be used as a communication aid.

In this thesis I will implement a small application of a dialogue system that could ontrol a video conference application[1]. It will be implemented in GoDiS using the TrindiKit toolkit.

First I will give some background information to this thesis. Then I describe the methods I used. Following that is the analysis part, consisting of a user analysis and a task analysis. After that is the design chapter where I describe a few of the programs I will use in the thesis. Then I get into the implementation description, followed by discussion and conlusion.

I believe the thesis would be most appriciated by those with some programming background and also with some knowledge of pragmatics. You do not have to be able to make your own programming applications. The examples of the code is rather straight forward and easy to follow, even for those who cannot program very well.

[1]By video conference application I mean a program that can transfer both sound and vision/video between two computers.

Chapter 2

Background

Dialogue systems are a way for users to use natural language in order to communicate with computers. The user should not be forced to use only short command like phrases. It may seem like science fiction when you say this and in many ways it is. However, in domain specific dialogues the above statement could be true; you can have a dialogue with your computer. Unfortunately we are a long way from a natural dialogue with the machines but research is progressing all the time. If the computers are to be integrated totally in our society I believe they must be easy to use and easy to communicate with. The user should not have to adapt too much to the computer in order to use its various applications.

In this chapter I will give some scientific background to the thesis.

2.1 Dialogue systems

A real, natural dialogue often have many overlaps of information and often "much information at the time". That is one participant in the dialogue may give several pieces of information at the time. For example; *It is raining but I am happy.* This gives the information on the weather *and* the speaker's mood. This is something that is completely natural for humans. If, for example, one wants to order a hot dog with mustard but no ketchup, that is what one says; see figure 2.1. Providing the hot dog seller heard correctly, one gets the desired snack. One of the few counter questions one could get is what type of hot dog one wants; thin, fried and so forth.

Getting a computer to "understand" the same type of statement is much more complicated than one might think at a first glance. The "normal" order in which this type of computer application works is based on a question-answer order. This means that the system asks a direct question and then waits for the user to answer. Then if the application is an old one, perhaps

Hot dog seller = HD
Hot dog customer = CU

```
HD: What would you like?
CU: Hot dog without ketchup and with some mustard, please.
HD: What sort of hot dog?
CU: A thin one, please.
[...]
```

Figure 2.1: Hot dogs

the only answers available are yes and no.

In Sweden there are some dialogue systems that are available on a "commercial" basis. I believe that the most well known is the time table information for trains from SJ[1]. As seen in figure 2.2 the system is not perfect. One sees both the good and the bad in SJ's system. It handles the question of place of departure and destination quite well. Even if the system did not understand or hear the first time, it handles the information the right way. However, the first time the system gets the desired time of departure, it just ignores it. The reason for this is that the system cannot handle much information at a time. It can only handle the question it wants answered at a time. This leads to the user having to answer more questions and having to give the same answer all over again.

This kind of extra information is something a well formed dialogue system must be able to handle.

2.2 TrindiKit

TRINDI (The Trindi Group, 2004a), Task Oriented Instructional Dialogue, was a project that was started in order to handle complex dialogue interaction by building computational models of dialogue moves.

TRINDIKIT is a toolkit that was developed in the TRINDI and SIRIDUS projects. It is used for building and experimenting with *Dialogue Move Engines*(DME), and *Information States*(IS), see sections 2.3.3 and 2.3.1. TRINDIKIT is not a dialogue system in itself whereas it is used as a tool for building one, providing that you have a formalized dialogue theory. In addition to a general structure TRINDIKIT specifies formats for defining in-

[1]Swedish railroad company

Translated from an actual call

The SJ system = SJ
The customer = CU

```
SJ: From where and where to do you want to go?
CU: I want to go from Göteborg to Karlskrona tomorrow at 12
o'clock.
SJ: You want to go from Göteborg to Kramfors.  Is that correct?
CU: No.
SJ: You want to go from Göteborg to Karlskrona.  Is that correct?
CU: Yes.
SJ: At what time do you want to go?
CU: Tomorrow at 12 o'clock.
[...]
```

Figure 2.2: SJ's time table information

formation states, dialogue move engines and update rules. The update rules handle basic rules for dealing with dialogue moves. These rules are based on Ginzburg's protocols for raising and resolve issues in dialogues (Larsson, 2002). The TrindiKit manual (Larsson et al., 2002) states that a DME forms the core of a complete dialogue system. TrindiKit also offers simple modules for input/output. These modules handle interpretation and generation of text strings. That is the information the system gets from the user and the one that the user gets from the system.

A graphical overview of TrindiKit showing its various parts, could look like figure 2.3. It shows how it is all connected. The parts that are marked *module* are the modules that TrindiKit provides. For example an interpret module and a generation moudule. The DME part constists of two other modules; the update and selection modules which are included in the modules mentioned above. The resource part is made of resources that the system uses. These are the resources that are implemented to build an application for GoDiS.

2.3 GoDiS

GoDiS, Gothenburg Dialogue System, is a plan based dialogue system implemented using the TrindiKit toolkit. It was developed for research at Göteborg University where it exists in two protytype versions; GoDiS-IOD

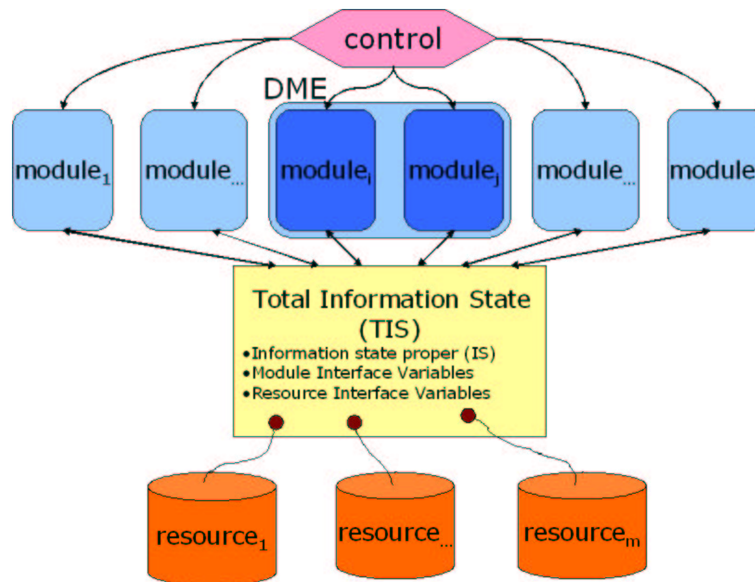

Figure 2.3: Graphical overview of TrindiKit

(Issue Oriented Dialogue) and GoDiS-AOD (Action Oriented Dialogue). The IOD version is basically an information seeking dialogue system, using a database to search for information. The AOD version is supposed to be able to control an external device, hence the name action oriented. In the first prototype of GoDiS-IOD a travel agency domain was implemented. In that of GoDiS-AOD an interface for controlling a VCR was implemented.

In order to handle the mapping from input to dialogue moves correctly, GoDiS uses keyword spotting. The idea of having keyword spotting is to trigger commands depending on the words of the input. Keyword spotting is much more stable than a full parser since the user can have very short input. He does not have to worry about sentence structure and so forth. In the travel agency domain for example, the word *to* states that the following word is a destination city, *by* is followed by means of transportation and so forth.

The version of GoDiS I looked closer upon for this thesis was the VCR domain of GoDiS-AOD; an application which should be able to control a VCR. The corner stones of any GoDiS application are basically three resources; the domain, device and lexicon resources. In the domain resource are the system's plans, that is what questions to ask, and in what order, in order to fullfill a goal. In the case of the VCR domain it could be finding out what channel to record. The device resource is an interface between the system and the external device. In case there is no external device, this re-

source is an interface to a simulated one. The lexicon resource contains the system's various output strings. Depending on the system's internal state, the system has different outputs. It also contains the input strings which are mapped to different moves.

The VCR-application is basically implemented to handle requests. The system asks a question, wanting the user to request an action. In figure 2.4 the user wants the GoDiS application to record a program (on TV). The user's answer is a request to the application to record a program. Since a VCR must know *when* and *what channel* to record, the user is obliged to provide those answers too.

```
System = $S>
User = $U>

$S> What can I do for you?
$U> I want to record a program
$S> What time do you want to start recording?
$U> From 3.15 pm.  Channel 5
$S> What time do you want to stop recording?
$U> Until 3.45 pm
$S> OK. What can I do for you?
[...]
```

Figure 2.4: Example from GoDiS-VCR

As mentioned earlier a dialogue system should be able to handle much information at a time. When handling this the system gets answers to unasked questions, and this is something that GoDiS handles well, see section 2.3.2.

2.3.1 Total Information State

The total information state (TIS), together with the update rules, is considered to be the brain of the dialogue system. It consists of three parts; the information state proper (IS), the resource interface variables (RIV) and the module interface variables (MIV). The MIV are just variables pointing to the various modules that are part of the system, for example the interpretation and generation modules. The RIV are just variables from the different resources.

Information State

The information state is the main component of the TIS. It could be said that the information state is the information that is stored internally by an agent (Bohlin et al., 1999). It represents information available to a dialogue participant at any given stage of the dialogue. The IS is structured as an abstract data structure which can be inspected and updated by dialogue system modules.

Figure 2.5 shows what this could look like in GoDiS. As one can see the IS is divided into two parts; one private and one shared. The private information is the system's internal "knowledge". It is used for updating the system and to select what actions to perform and so forth. The shared part is the common knowledge shared between the system and the user.

One does not have to fully understand the information state in figure 2.5. We will see a very short explanation to some fields in the IS. The /SHARED/COM field contains the set of propositions that the system and the user have mutually agreed to be true. The important thing is that the dialogue participants (DP) have committed to these propositions, not if the believe them. /SHARED/ISSUES, /SHARED/QUD and /SHARED/ACTIONS all have the same stack structure where the current questions and action, respectivly, are on the top.

Once the questions are answered or the actions are done, the stack is popped; that is, the top element of the stack is removed. The stack structure is appropriate in the sense that a question could be reraised, enabling the user to have a nested dialogue.

The raised questions are pushed onto the QUD and hopefully they are resolved by suitable answers. The QUD is also used in accomodation like in figure 2.6. When the system is asking the first question, it has pushed that question onto the QUD. The user answers but is not sure whether the VCR is stopped or if it is recording. Since his favorite show starts now, he wants to know whether he can record it or not. Thus the second question, asking about the VCR's status. If the system has a plan of which the new question is a part, then that question is pushed onto the QUD. The system resolves the new plan first before returning to the old one. Since the VCR was already recording, the user had to abort.

$$
\begin{bmatrix}
\textsc{private} & \begin{bmatrix}
\textsc{agenda} & : & \textsc{OpenQueue}\big(\textsc{Action}\big) \\
\textsc{plan} & : & \textsc{OpenStack}\big(\textsc{PlanConstruct}\big) \\
\textsc{bel} & : & \textsc{Set}\big(\textsc{Prop}\big) \\
\textsc{tmp} & : & \begin{bmatrix} \textsc{usr} & : & \mathit{Tmp} \\ \textsc{sys} & : & \mathit{Tmp} \end{bmatrix} \\
\textsc{nim} & : & \textsc{OpenQueue}\Big(\textsc{Pair}\big(\text{DP},\textsc{Move}\big)\Big)
\end{bmatrix} \\
\textsc{shared} & \begin{bmatrix}
\textsc{com} & : & \textsc{Set}\big(\textsc{Prop}\big) \\
\textsc{issues} & : & \textsc{OpenStack}\big(\textsc{Question}\big) \\
\textsc{actions} & : & \textsc{OpenStack}\big(\textsc{Action}\big) \\
\textsc{qud} & : & \textsc{OpenStack}\big(\textsc{Question}\big) \\
\textsc{pm} & : & \textsc{OpenStack}\big(\textsc{Move}\big) \\
\textsc{lu} & : & \begin{bmatrix} \textsc{speaker} & : & \textsc{Participant} \\ \textsc{move} & : & \textsc{Set}\big(\textsc{Move}\big) \end{bmatrix}
\end{bmatrix}
\end{bmatrix}
$$

$$
\boldsymbol{Tmp}= \begin{bmatrix}
\textsc{com} & : & \textsc{Set}\big(\textsc{Prop}\big) \\
\textsc{issues} & : & \textsc{OpenStack}\big(\textsc{Question}\big) \\
\textsc{actions} & : & \textsc{OpenStack}\big(\textsc{Action}\big) \\
\textsc{qud} & : & \textsc{OpenStack}\big(\textsc{Question}\big) \\
\textsc{agenda} & : & \textsc{OpenStack}\big(\textsc{Action}\big) \\
\textsc{plan} & : & \textsc{OpenStack}\big(\textsc{PlanConstruct}\big)
\end{bmatrix}
$$

Figure 2.5: Information state in GoDiS

System = $S>
User = $U>

```
$S>   What do you want to do?
$U-1> Record channel two.
$U-2> What status is the VCR?
$S-2> The VCR is recording.
$U-1> OK. Abort.
[...]
```

Figure 2.6: Example of accomodation onto the QUD

2.3.2 Accomodation

In order to cope with the un-asked questions, the system uses accomodation which is based on presuppositon. One of the most used examples of presupposition is *The king of France is bald*. This means that if there is a proposition that the king is bald, then there must exist a king. In this case there must also be the case that France indeed has a king, and that very king is bald.

Applying this theory to GoDiS may look like; *Record channel two at three fifteen pm*. This utterance answers the following unasked questions (the first one is also a request):

- what the system is to do (record a program)
- what channel to record (2)
- what time to record (3.15 pm)

However, the example above is something all dialogue systems *should* handle. Another important feature of GoDiS is the possibility to have alternative questions. Assume that there are two ways of interpreting a point in time; as a starting time for a recording or a stopping time. If the user only gives the time as input, without any surrounding context, GoDiS should ask a question to clarify if the user meant the starting or stopping time. The below example shows what this could look like.

```
$S> What can I do for you?
$U> Ten pm
$S> Do you want to start recording or stop recording?
$U> Start
```

[...]

2.3.3 Dialogue move

The type of dialogue move realized by an utterance is determined by the relation between the content of the utterance , and the activity in which the utterance oocurs (Larsson, 2002). Larsson also says that the central dialogue moves in an inquiry-oriented dialogue concern raising and addressing issues. This is done by *ask* and *answer* moves.

In the mentioned domain of the VCR, we are not only interested in issues but also in actions. The same relation is still there though. In order to realize an answer move, the answer must be a *relevant* answer. That is, the answer must be an answer to a *question* in the domain. The same relevance applies to the *ask* move; the question must be a relevant one. This relevance mapping, from input to move, is implemented in the lexicon module, see section 6.3.

An utterance like *Start recording at five pm* could be interpreted as answer(start-time(1700)). This should be understood as 1700 being the answer to what time time the recording should start. Furthermore 1700 must be an actual point in time, not some other number.

Dialogue Move Engine

The dialogue move engine (DME) consists of an update module and a selection module. Its main function is to update the information state based on dialogue moves that were and to select new ones to be performed by the system.

The DME can be regarded as a dialogue manager based on the concepts of dialogue moves and information states. It is a certain kind of dialogue manager which accesses an information state and whose input and output are dialogue moves.

2.3.4 Resources

In this section I will give a more detailed description of the resources. These are the components that are the most important when making one's own application of GoDiS. The recources are components that the system uses in the TIS.

In this section we will see examples of code that must be implemented in order to get working resources.

Device

The device resource could be said to be an interface to a device. It handles the different variables in order define the states of the system. The device resource connects external devices to the system. If there is no external device, it simulates one.

An application based on GoDiS-AOD must have some actions specified in the device module. Otherwise one cannot connect GoDiS to an application and make it "do things". This is done with the action/2 predicate which takes an action as first argument and a list of parameters as the second. In order to perform an action, one must also specify the parameters to the action. These parameters could be seen as help variables which are set by interacting with the system. As seen below all actions do not need parameters. The action "SetChannel" must have a parameter in order to know what channel to set. The "Play" action does not need one since there is no need to have any specification when starting a VCR.

```
action( 'SetChannel', [ new_channel ] ).
action( 'Play', [] ).
```

There are two kind of variables in the device resource. Those which are temporary and those that have a longer life expectancy. The long lived ones could be implemented as a default value. As seen below the play status of a VCR is *stopped* by default.

```
default_value( play_status, stopped ).
```

In the device resource there are also predicates to control the device. The variables needed to control it are set when the device resource interact with the other resources (also see the domain section below). The *dev_do*/2 predicate interacts with the domain resource in order to set variables to new values. The *dev_set*/2 predicate sets the variables to a value using variable_value/2. When setting a new value, the old one is naturally removed. The system must also be able to get hold of the value of the variables. This is done with the *dev_get*/2 predicate. The most important predicate to make a device do something is the *perform_command*/1 predicate.

Domain

If the device resource could be said to be a device interface and containing predicates to control it, the domain could be said to control the flow of the system. The domain resource has *plans* to manage it. They are plans for the system to follow to "get the job done". It takes a question or an action as the first arguement and a list of actions to do as the second.

More often than not, there are postconditions to the plans. In order for an action to be considered done, the propositions specified by the postconditions must be true. That is, they must be part of /SHARED/COM in the information state (section 2.3.1).

```
plan( top,
      [ forget_all,
        raise( X^action(X) ),
        findout( set( [ action(vcr_change_play_status),
                        action(vcr_new_channel),
                        action(vcr_timer_recording),
                        action(vcr_settings) ] ) ) ] ).
postcond( top, none ).
```

The plan above is a standard plan that is called when the system is started or restarted. The *forget_all* is just a way of clearing the IS of the system. The *findout*/1 predicate raises a question to the user, placing it on the QUD. The raising of a question starts an ask move in the system. For more detailed information on how this works, see (Larsson, 2002). In the lexicon section below I will show how these raised questions may look like. In this particular case above, the system wants the user to tell it which action it should perform.

Here it is crucial that the findout(X1^contactName(X1)) predicate is asking for something that has the exact same name as the parameters in the device file. That is, the device file must have some help variable named *contactName*.

In the implementation of GoDiS findout(q) means that the system is looking for an answer to the question q. The answer is then stored in /SHARED/COM. As mentioned before, the /SHARED/COM field contains the set of propostions that the user and the system have mutually agreed to during the dialogue.

Since one proposition is meaningful in one activity, it does not mean that it is meaningful in all activities. To distinguish them there is a sortal system in the domain resource implemented as below.

```
sort_restr( new_channel( X ) ) :- sem_sort( X, number ).
```

For this proposition to be correct, then X must *semantically* be a number.

Lexicon

The lexicon file consists of mappings between either strings to moves or moves to strings. There are two main predicates in the module that handle those input and output strings, the *input_form*/2 and *output_form*/2.

The system has a specific output to every move from the system. The predicate output_form takes as first argument the current move. As the second argument we find the string representing the output relating to the move. If the system wants to ask the user something it uses the *output_form* like below.

```
output_form( ask(X^new_channel(X)), ['What channel do you want?'] ).
```

In order to make a dialogue system to appear more “natural”, it has to be able to handle feedback. GoDiS is able to do this, using the output form. An example of this could be the following, showing the output when the system has not been able to understand the user. That is, it could not map the input to any move.

```
output_form( icm:und*neg, ['I dont quite understand.']  ).
```

The *input_form* predicate works more or less the opposite of the output_form. It selects moves from the given input by using keyword spotting with some surrounding context. This is where the mapping from intput to moves is defined. The example below shows that when the user gives the input “play”, it is mapped to the *request(vcr_play)* move.

```
input_form( [play], request(vcr_play) ).
```

Sometimes the user may use different words or word forms in the input. The lexicon file has an easy, simple solution to this. In the example below the system wants an answer to a “What domain do you want?”-question. Since there could be different input strings answering this question, the lexicon has the predicate *synset*/1. This is a synonomy set, mapping the synonyms to the same concept word.

```
input_form( S, answer(domain(vcr)) ):-
        lexsem( S, C ),
        sem_sort( C, domain ).

lexsem( Word, Concept ):-
        synset( Words, Concept ),
        member( Word, Words ).

synset( [[vcr],[video],[v,c,r]], vcr ).
```

If not using the above mentioned synset, there would have to be an input alternative for each and every synonym.

Semsort

The *semsort* file contains the semantical representation of specific words. It decides what category/type a word belongs to.

```
sem_sort( vcr_play, action ).
sem_sort( vcr_stop, action ).

sem_sort( N, number ) :-
        integer(N).
```

In the examples above we see that *vcr_play* and *vcr_stop* are both actions. There must be a semantic represenation of an action for every one implemented in the system. The semantic representation of a number is approved only if it is an integer.

A very important semantic representation is the predicate representing an *isa-hierarchy*. This defines what "class" a type belongs to.

```
isa( person, name ).
isa( contactName, name ).
isa( newContactName, name ).
```

Using this predicate GoDiS can disambiguate what question that *name(X)* could be an answer to. It could all start with the system looking for a *contactName* in the domain file. The relevant answer to this is of course an instanciation of contactName. However, the mapping from input to a move in the lexicon file is looking for a *name*. Since the lexicon is using the semsort file to see whether the input really is a name, this hierarchy is made to specify that the *name* could be, semantically, the same thing as

contactName.

2.3.5 Menu to Dialogue

When using menus to control an application, the menus are made in (many) levels. This gives the user an overview of what lies beneath; that is, what operations the system can do. This overview is something that can be difficult to notice in a dialogue system, if not constantly repeated by the system, when using it for the first times.

In addtion to what is mentioned in the resources' section, there is a way of implementing plans in GoDiS to make them behave like a menu interface. The plans below could be seen as multi-choice lists waiting for an alternative. When the user starts the dialogue system he is asked what he want to do. Assume that the user wants to add a program to the timer recordings. He says that he wants to work with timer recordings and the system loads the *vcr_timer_recording* plan accordingly. Since there are more than one alternative action in handling timer recordings, the system will ask the user to specify which one he would like to perform. In this example the user wants to add a program to the timer recordings.

Naturally the sub-plans could have sub-plans of their own. There is no limit to this in GoDiS other than what feels intuitive and natural.

Level 1

```
plan( top,
      [ forget_all,
        raise( X^action(X) ),
        findout( set( [ action(vcr_change_play_status),
                        action(vcr_new_channel),
                        action(vcr_timer_recording),
                        action(vcr_settings) ] ) ) ] ).
postcond( top, none ).
```

Level 2

```
plan( vcr_timer_recording,
      [ findout( set( [ action(vcr_add_program),
                        action(vcr_delete_program) ]) ) ] ).
postcond( vcr_timer_recording, done( vcr_add_program ) ).
postcond( vcr_timer_recording, done( vcr_delete_program ) ).
```

Level 3

```
plan( vcr_add_program,
      [ findout(X1^channel_to_store(X1)),
        findout(X2^date_to_store(X2)),
        findout(X3^start_time_to_store(X3)),
        findout(X4^stop_time_to_store(X4)),
        dev_do(vcr, 'AddProgram') ] ).
postcond( vcr_add_program, done( 'AddProgram' ) ).
```

Only in the last plan above are the variables set in order to let the application add a program to the timer recordings.

The plans implemented to behave like a menu interface provide help if the user is not accustomed to the system. For example, if it is not clear what the user could do with the application, he just takes one step at a time. This way the user will be helped since the application will ask what alternative the user wants to do.

However, if the user knows what the system is capable of doing, he can just say so in the top plan; for example, "Add program". This will take the user directly to the *vcr_add_program* plan. The user will not first have to specify that he wants to handle timer recordings and only after that what he wants to do with the recordings.

Chapter 3

Method

In the beginning my idea was to make an application primarily for people with physical disabilities. After giving this some more thoughts I decided to change my point of approach. Since it was the dialogue system I was interested in, I wanted an application to be used as a communication aid.

The idea of using spoken dialogue systems as communiation aids is something that I was convinced could be done. The use of the voice is something that I find will be very useful in future computer applications.

3.1 Analysis

In addition to conducting a task analysis that focuses on the software production, I also conducted a user analysis.

I started to take notes on how I would use a dialogue system to control a video conference application. I had an idea on how relevant questions would look like, and how the answers could be phrased. Since the domain, controlling a video conference application, is quite narrow, I believed that little variation would be needed in the different dialogue moves.

There was also the question of user habits and user computer skill - if the user uses a computer every day, or maybe once a week and so forth. I believe that different kind of users may have different ideas on what dialogues to use.

I used a web based survey (see appendix B) in order to get information on what users might say when communicating with these types of applications (dialogue systems).

3.2 Implementation

I started by downloading the latest versions of both GoDiS and TrindiKit. I installed them and checked if they were compatible and worked as they should.

I started to look at GoDiS-AOD in order to get a feeling of the system. There were many similarities to the IOD, inquiry-oriented dialogue, version I had used before (Ailomaa et al., 2003). It had changed quite a lot since I used it last time so I had to reacquaint myself with it.

A good reason for using GoDiS is that it is very modular and offers features for advanced dialogue management. When using GoDiS to work with a new application one does not need to solve general problems with dialogue management; these are already solved by GoDiS.

Many commercially used products often have few possibilities to be modified and changed.

Chapter 4

Analysis

Software engineering is primarily about the production of software solutions for a given application. The use of a task analysis is supposed to investigate an existing situation. In the case of this application, I used scenarios to identify what people are doing when using a video conference application.

When developping an application that is supposed to interact with users, the task analysis mentioned above is often not enough. In order to have a more "intuitive" product we must look elsewhere for a solution. The idea of using interaction design is to design products to support people in their everyday and working lives (Preece et al., 2002). Since it is about finding ways to support people it is different from the standard software engineering. Most of the time the users are not involved in the process of developping an application. In the worst case scenario, if the users are not involved in the process, the product will not be used. The conducted user analysis is a way to confirm or revise the ideas behind the task analysis.

The different requirements are destilled from the two mentioned analyses.

Naturally there are a number of different analysis methods that could be used in a thesis like this. One common and good method that is to use corpus studies. That is, the study of actual dialogues gathered from domains within the boundaries of the theses. Due to little time this was not an option in this thesis and the main reason for choosing the methods mentioned.

4.1 Task Analysis

Communication between people is often taken for granted regardless if it is spoken, written or performed in some other way. If one wants to communicate through the computer, the user will need some sort of application that

relays the communication. There are quite a few such applications today that are easy enough to understand and control even it is one's first try. The application (more often than not) has a set of buttons in order to control the application. It may a window where one can write and edit one's written message. If the user has a camera installed one has additional functions to control it. In addition to this one can have some sort of address book where one stores information on contacts, like number, address and so forth.

The following scenarios are examples of dialogues that could take place between user and system.

System = `$S>`
User = `$U>`

Call a friend

Naturally one musts be able to call someone with the help of the system. The user tells the system what he wants to do; call a friend.

The user should just say to whom he wants to call.

```
$S>:  What do you want to do?
$U>:  I want to call X
$S>:  OK, calling X
[...]
```

Hang up

When the call is to end, the user should just as easy be able to disconnect and hang up.

```
$S>:  What do you want to do?
$U>:  Hang up
$S>:  OK, disconnecting
[...]
```

Receive a call

Just like a regular phone, the user must be able to receive a call and answer it.

```
$S>:  What do you want to do?
$U>:  Answer
$S>:  OK, connecting call
[...]
```

Do not answer

Likewise when you want to be left alone you should not have to answer. When you decide to not take the call, it is just disconnected.

Add contact to address book

An address book is something that an application like this must have. In order to keep record of names and number, the user must be able to add contacts to the address book. The input to the address book consists of the contact name and the number to use. In this case it should be an IP number[1].

```
$S>:  What do you want to do?
$U>:  Add contact
$S>:  What user name?
$U>:  X
$S>:  What number?
$U>:  Y
$S>:  User X with number Y added to address book
[...]
```

Delete contact in address book

As well as adding entries in an address book there must of course be a way to delete one. This should be quite simple, the user only states the name that is to be deleted.

As of now I do not want to have any "safety mode" attached to the delete command. That is the system should not ask *Do you really want to delete X?*.

```
$S>:  What do you want to do?
$U>:  Delete X
$S>:  User X deleted from address book
```

[1] Short for Internet Protocol. This number must be unique on a network. For more information see: http://en.wikipedia.org/wiki/IP_number

```
[...]
```

Modify contact in address book

One feature that I find as important as adding to the address book is being able to modify it. It is useful in two ways. First if the user is adding people with the same name, perhaps he wants to change the names of some entries. Second, being able to change the IP number is necessary because they could often change.

```
$S>:  What do you want to do?
$U>:  Change contact information
$S>:  What user name?
$U>:  X
$S>:  What do you want to change?
$U>:  The user name/number
$S>:  State the new name/number
$U>:  Y
$S>:  User X's name/number is Y
[...]
```

Look up contact in address book

If the user is not sure of what information is stored about a certain person in the address book, he must be able to check this. As it is now the user can only check what number an entry has but as the application grows the user should be able to find out more things from the address book. For example phone number, street address and so forth.

```
$S>:  What do you want to do?
$U>:  Check the number for X
$S>:  User X has number Y
[...]
```

Close application

When using a voice controlled system the user must naturally be able to shut it down using the voice. There are problems with this that I do not make any attempt to solve, see section 4.3.2.

```
$S>:  What do you want to do?
$U>:  End/quit application
```

```
$S>:  OK, good bye!
[...]
```

4.2 User analysis

My initial user group was people with some physical disability. This I changed in the very beginning to a user group of people who wanted to communicate through some device via the internet/web. This regardless of disability or not. The users will use the system indoors. They must also, naturally, have access to a computer.

Since the system is supposed to be controlled by voice, the surroundings are important. Depending on the functionality of the ASR-system, there can be more or less amount of background noise. This is naturally something that is a problem, but in this thesis I will not make any attempt to give some solution to it.

I had an idea of how the system should work and how to communicate with it. Naturally all possible solutions cannot be found by one person alone. Opinions from other people would be needed in order to find out how people would like to communicate with the system. I wanted all sort of people to answer this, so I performed a survey that did not focus on any particular domain of people. However, the users should have some knowledge of computers. They should know the basic operations; opening an application, open close files and so forth.

The survey was a free web based one. The survey was sent to undergraduate students in computational linguistics and to some companies which were producing companies with no reference to computer technology. The reason for this was that different points of reference was desirable.

There were some 25-30 test persons who answered; men/women between the ages of under 20 up to 45. The computer usage varies quite a lot. I also wanted to know whether the user had used a dialogue system before, and if so which one. The answers from the survey was used as a base to create alternative input forms in the application.

Some questions were answered by multiple choice, other by giving the user the possibility to use his own words. The survey was made in Swedish and the following answers are the translations in English. If the free text answers differed only in some small way, for example only in word order, I interpreted them as one and the same.

Some questions were perhaps ambiguous but all answers to the questions are supposed to be said to the system.

The following tables are the results of the survey. As we will see, the anwers are basically the same within any given question. I do not believe that there are a lot of different possibilities here.

For the complete survey with questions and alternatives, see appendix B.

Start conversation

What do you say when you want to start a conversation with X?
1. I want to talk to X
2. Call X or Computer, call X
3. Start X
4. I want to start a video call with X
5. Hello, my name is NN and I want X
6. X (just the name)
7. Hello

When it comes to starting a conversation, there is not much that differs between the various answers. Most of them are inputs containg the name to call, and maybe some additional string. I received only one answer that did not contain some name to call.

End conversation

What do you say when you want to end the current conversation?
1. (Computer) Hang up
2. End call
3. End/stop
4. Good bye
5. I want to stop talking to X
6. I must end it now
7. See you!

Ending the current conversation received some answers I did not expect. I thought it would only be *Hang* up or *End call*.

Accept incoming call

What do you say when you want to accept an incoming call?
1. (Computer) Answer
2. I want to answer
3. Hello
4. Accept call
5. I'll take it
6. Yes
7. This is <own name>
8. I want to hear

As with the *End call* scenario, I thought to receive only a few variants. As we see there are eight. I thought that *Answer*, with some variants, would be the only answers.

Rejecting incoming call

What do you say when you DO NOT want to accept an incoming call?
1. (Computer) Don't answer
2. I don't want to answer now
3. Don't accept call
4. Deny incoming call
5. Occupied-busy
6. No
7. Ignore
8. Hello, I don't have time now. See you later
9. I don't want to hear

Here I expected few differences. As the example before, there are quite many. Looking from a dialogue system perspective though, I think this is good. You have both the *command* alternative and the slightly longer *phrase* alternative. Some, like the last two, are perhaps not the standard way of rejecting an incoming call, but still it should be handled in a correct way.

Make a call using address book

What do you say when you want to use the address book to make a call?
1. (Computer) Call X
2. Call X from address book
3. I want to call X
4. Get address book, call X
5. Call

This is almost the same as the first question. I deliberatly specified that the call should be made using the address book. The reason for this is that I wanted to see if people used answers with a menu like structure or if they used more direct requests. As it were, only one answer, number four above, did not use the more direct approach.

Check information in address book

What do you say when you want to check information on someone in address book?
1. (Computer) Check X (in the address book)
2. What number has X got?
3. Is X in the address book?
4. Show X
5. Find/look up X
6. (Open) Information on X in address book
7. I want to look up X in the address book
8. Info

Some users want to specify that they want to use the address book. Other ones just demand to see a certain number or person.

Add contact to address book

What do you say when you want to add a person to the address book? **What information do want to add?**
1. (Computer) Add X
2. Add new contact
3. Add X to address book

The information people want to add to the address book are, quite naturally, name and number. Some want to add both name and family name, and sometimes an alias in addition to the regular names. It should be possible to add multiple numbers: work, home, cell and so forth. The user should be able to add a street address. Also to make the address book more complete, being able to add an e-mail address or a web page address is desirable. To make the address book even more complete, some want to be able to add a birthday to the contact.

Delete contact

What do you say when you want to delete an entry in the address book?
1. Delete/remove X
2. (Computer) Delete/remove X from address book
3. Delete/remove

Here the result was quite expected. There were few alternatives to what people would say to delete an entry in the address book. The only thing that differed was the fact that some wanted to specifically say that the system should delete something in the address book.

Abort input

What do you say when you want to abort the input or similar?
1. Abort
2. Regret latest input
3. Erase
4. Redo

The most common answer by far, was the first one. It seems quite natural to abort something by saying *Abort*.

End video application

What do you say when you want to end the video application?
1. Abort
2. (Computer) Quit
3. End application

The test group was in agreement on how to end the video application. There should not be any problems with the answers except for *Abort*. Since some people would use the same word for ending the application as for aborting the current input, there could be a conflict in the system interpretation.

Comments and opinions on additional features

It seems that people would rather use short command like utterances when interacting with the system. According to some of the answers from the survey they feel silly when they have a dialogue with a computer. Those who commented on this did not see any need to use "natural language"[2] to control this type of application. They felt more comfortable using short commands.

The possibilty to save recordings of conversations, person to person, was another thing that some wanted. Whether this was only sound, or sound and image both was not stated. A feature one person wanted was the possibility to get a notification of an incoming call while having another.

[2]In this case, the use of full sentences and being able to express more complex phrases.

The last comment on additional features was the possibility to interact further with the video application. One person wanted to be able to control the camera device; right/left, up/down and so forth. She also wanted to be able to control the volume.

User analysis - conclusion

There were not any differences whether the supposed user was a man or a woman. Most of the answers were a confirmation of my ideas and thoughts in that they were all quite alike. They did not result in any real additions to my ideas of the standard dialogues to control the application.

The more time the user spends by a computer, the shorter and more command like the dialogue tends to be. On the other hand, if the user is someone who rarely uses a computer, the dialogue tends to be more like natural dialogue. The user does not think whether the system can handle the input or not, he just makes "conversation".

4.3 The application

An application should be easy to use and handle. The dialogue system is one solution to this. It enables the user to use as natural a language as possible. When controlling an application with your voice, you do not want to limit yourself to commands to control the system. That is, the system should not *only* have a menu like structure. The example below[3], although perhaps not a very good one, shows an example that could be called menu like in the extreme. You should be able to just say *Call NN*. You should not have to know specifically where and how to access the number to NN.

User: Go to address book
User: Find NN
User: Find number
User: Make call

4.3.1 Potential problems with natural language

Natural language is always a problem when dealing with dialogue systems. Since my application uses a form of keyword spotting, this problem becomes less emphasized. The text interpretation module seeks keywords with some surrounding context. See section 6.3 for details and examples.

[3] This example is purely fiction. It is not used in any application that I know of.

4.3.2 Potential problems with selective hearing

A problem that arrises if the system should use ASR is when to should "listen" and when not to. How will the system know if the user is addressing it or if the user is using the microphone to something else? One solution is that the user always uses keywords when addressing the system - for example "Computer - do this". Another solution could be to let the user click on the appliction that is to listen.

In this thesis I will not try to solve this problem since I do not find it to be problem that I have to deal with at this stage. In the scenarios it can be assumed that the system is listening to the user.

4.4 Requirements

4.4.1 Usability requirements

The dialogue system must be easy to use and easy to understand. The controls and functions must be "intuitivly correct" meaning that the application should to a certain degree adapt to the user, rather than the other way around.

An important requirement on the user is that the user must be interested in using the application. If not he will not learn enough to use it well.

4.4.2 Functional requirements

The user should succeed in connecting to the internet via the application. Naturally he must have an internet account.

The user should be able to use microphone only. Naturally the user could also use keyboard and mouse.

The user must be able to have some sort of address book which could be modified by the user.

The underlying program should work to a certain degree. The GoDiS system should be able to handle the input correctly and map the dialogue moves correctly.

The system must be able to handle all the input, not just the "correct" one. It is most essential that the system does not fail.

The system should be able to set the internal variables in order to use them to control the video application. For example storing variables locally before making callls to the video application.

The user should be able to handle and update the address book via the dialogue system.

Chapter 5

Design

In this chapter I will state how I would like the design of the system to be. I will also give some examples of free applications that could be used to engage in video conference.

5.1 Video conference applications

I wanted to use as small a video application as possible. It should be easy to use and easy to understand even if one uses it for the first time. There are many applications that could be used to this end, both ones that are free and ones that cost money.

Wanting to use a free program, I looked to Sourceforge[1] and similar communities. These communities have many programs under development that are available for research.

There are also some well known applications that are used commercially but are still free to use. Two of the best known to people are ICQ[2] and MSN Messenger[3] (Figure 5.1). These are mainly used for chatting, that is, short written messages between users. However, they have the possibility to let users engage in video conferences.

5.1.1 Used application

The video conference application I finally chose for my dialogue system was an open source[4] application, MyPhone, seen in figure 5.1.1. This looked

[1] http://sourceforge.net
[2] http://web.icq.com
[3] http://messenger.msn.com
[4] For information and explanation to open source, see the homepage http://www.opensource.org/docs/definition.php

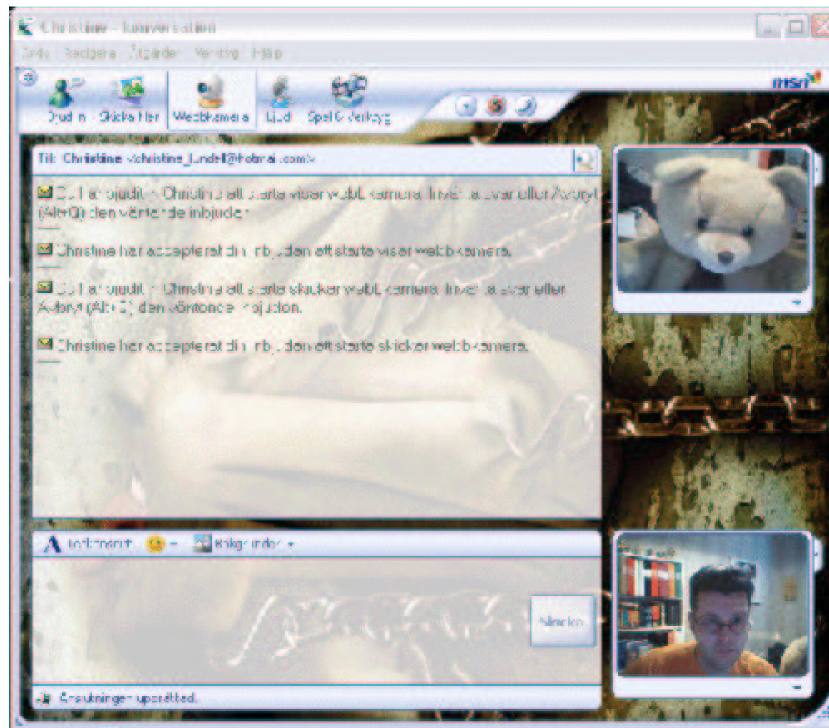

Figure 5.1: MSN Messenger, a common program for communication over the internet

both easy to use and it was a small application, not in need of an extensive installation.

MyPhone has several interfaces. One for each feature; video conference, sound only and chat interface.

5.1.2 Discarded applications

In the very beginning of this thesis I had three video conference applications that I found to be open source; MyPhone, Privaria and FFmpeg.

I hoped to be able to use all three and then decide which one was easiest to use. Unfortunately I was only able to use on of them. I could not get Privaria or FFmpeg to work.

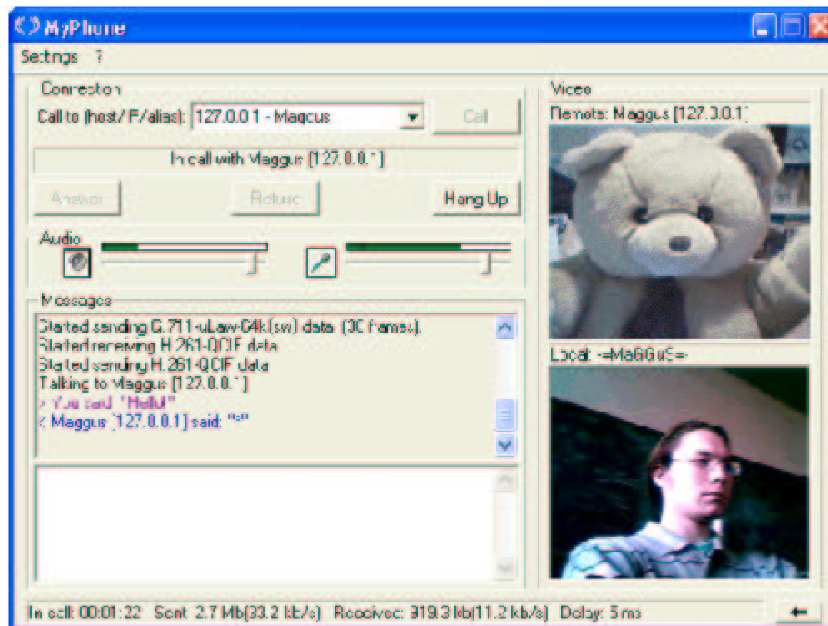

Figure 5.2: The video conference application MyPhone

5.2 Automatic Speech Recognition

Since I wanted my system to voice controlled, I would have to have an automatic speech recognizer. This was not something that was critical to have, but more of a bonus.

I tried an application from Philips, FreeSpeech 2000, which is a speech recognition program . The reason for choosing FreeSpeech 2000 was that it has proven to work satisfyingly for people; not only ordinary users but also communication-disabled people. Since it is quite old, about five years, it is rather slow working. However, this application satisfied the demands at this point.

Chapter 6

Implementation of Resources

In this chapter there will be an overview of my application and how it is implemented. I will give a description of the most important files and components of the system. I will also give small examples of the code and explain how they work.

Since the application is supposed to be in Swedish, some examples will be in Swedish - specially the examples of input and output.

Here I will give an overview of my implemented resources. We will see some small examples of the code and some comments to them.

6.1 Device

Below is a default entry in the address book. It uses the predicate variable_value/2 which takes an atom stating what type of value it is and a list specifying the value. The value is a list consisting of the contact name and the IP number.

```
variable_value( book, [contactName(andreas),number('192.168.0.171')] ).
```

As of now the address book is in this resource, consisting of entries like the above, since I only try the system so far. You can add, delete and modify entries in the address book.

If the user wants to take a call, the system does not need any specifying parameter. If, on the other hand, the user wants to add a contact to the address book, he must know what contact name to use and what IP number. The examples below show the actions and their necessary parameters.

```
action( 'CallPerson', [contactToCall] ).
```

```
action( 'TakeCall', [] ).
action( 'AddContact', [contactName,number] ).
```

In order to update the device resource the variables must change accordingly. In the example below is where the action "happens". The variables are set in order to, for example, get the address book updated.

```
perform_command( 'AddContact' ):-
        dev_get( contactName, Name ),
        dev_get( number, Number ),
        dev_set( book, [contactName(Name),number(Number)] ),
        retract( variable_value(contactName, Name) ),
        retract( variable_value(number, Number) ).
```

Here we can see the *AddContact* predicate being performed. First we can see that it finds the instantiations of th *contactName* and the *number*. Then it justs add the new information into the *book*. The *retract*/1 predicate deletes the temporary variables in order to clear the device resource. There could be problems later in the system if they are not removed.

6.2 Domain

The plans are crucial to maintain a flow in the system. Below we see the starting plan in the application. It raises the standard question, asking the user what he wants to do.

```
plan( top,
      [ forget_all,
        raise( X^action(X) ),
        findout( set( [action(vid_call_person),
                       action(vid_hang_up),
                       action(vid_take_call),
                       action(vid_add_contact),
                       action(vid_delete_contact),
                       action(vid_change_info),
                       action(vid_find_info),
                       action(vid_end_application)
                     ] ) )
      ] ).
postcond( top, none ).
```

However, the above example is not very good considering the menu like structure mentioned in section 2.3.5. If the application's alternatives are

"on the same level", it will be difficult to get an overview of what the application is capable of. It will be difficult for the the user to control the device if he is unsure of how the application works.

As of now the application looks like this but the next version will have a slightly different look; like the following example. This will give the application a better menu like structure. The "sub-menus", *vid_handle_call* and *vid_handle_addressBook*, will handle the interaction instead of letting the top plan handle everything. Even though the plans will become more menu like, the user can naturally still go "right to the point" if he knows the system.

```
plan( top,
      [ forget_all,
        raise( X^action(X) ),
        findout( set( [action(vid_handle_call),
                       action(vid_handle_addressBook),
                       action(vid_end_application)
                     ] ) )
      ] ).
postcond( top, none ).
```

In the example below the system has got an answer from the user that he wants to add a contact. In order to satisfy the new plan, the system needs to have a contact name and a number. When the user has provided the name and number, the system uses the *dev_do*/2 predicate to start the AddContact process in the device file.

```
plan( vid_add_contact,
      [ findout(X1^contactName(X1)),
        findout(X2^number(X2)),
        dev_do( vid, 'AddContact' )
      ]).
postcond( vid_add_contact, done( 'AddContact' ) ).
```

As described earlier the system handles alternative questions if there is more than one alternative. If the user wants to "Change information" in the top plan, he is first asked to give the name of the person to be changed. Next he receives a counter question whether he wants to change the user name or the number.

```
plan( vid_change_info,
      [ findout(X1^person(X1)),
        findout( set( [action(vid_change_name),
                       action(vid_change_number) ] ) )
```

```
      ] ).
postcond( vid_change_info, done(vid_change_name) ).
postcond( vid_change_info, done(vid_change_number) ).
```

In the example below the proposition that the contact name really is a name must be true. The domain resource interact with the semsort resource.

```
sort_restr( contactName( X ) ) :- sem_sort( X, name ).
```

6.3 Lexicon

The lexicon is all about mapping strings to moves and vice verse. This is all that occurs in the lexicon.

6.3.1 Output_form

The example below is the system asking the user to provide an IP number.
`output_form( ask(X^number(X)), ['Vilket nummer?']).`[1]

The lexicon file also has many *output_form* predicates regarding feedback from the system. This is something that a dialogue system should handle too. In order to make a dialogue system more "natural", the user will require some feedback. The following example is a confirmation of an addition to the address book.

`output_form( confirm(vid_add_contact), ['Ny person har lagts till i adressboken'] ).`[2]

6.3.2 Input_form

In the example below there is a "clean" keyword spotter. If it finds the two words in the list, next to each other, it will request an action, *request(vid_add_contact)*. It does not care what the surrounding words are, if any at all.

`input_form( [lägga,till]`[3]`, request(vid_add_contact) ).`

If the user did have more relevant input, the system treats that in due order. Expanding the input above to `[lägga,till,stina]`[4] the follwing mappings

[1] What number?
[2] New person added to address book
[3] Add
[4] Add stina, where *stina* is a name

are used.

```
input_form( [lägga,till], request(vid_add_contact) ).

input_form( [X], answer(contactName(X)) ):-
        sem_sort(X,name).
```

In order to enter the correct IP number, the user has to be a little careful. The system only approves of an input string that could be interpreted as an IP number. A correct IP number is a sequence of four number groups with periods (".") in between. Each number group consists of one to three digits. The input though must look as the example below.

```
thirteen point nine point onehundredandtwo point fourteen
```

The help predicate *longNum*/1 below takes the relevant input and returns an atom corresponding to the input. In this example it would return '13.9.102.14' which will be used by the system in some relevant way. For example to contact the computer that has the IP number or perhaps store it in the address book.

```
input_form( Num, answer(num(Dig)) ):-
        longNum(Num,Dig),
        sem_sort(Dig,number).
```

6.3.3 Difficulties

Some problems could arrise when there are contextually ambiguous inputs. If, for example, the user gives only a name as input the system could interpret it as a relevant answer to many questions.

The *input_form* could look like the following example. The user gave a only name as input, and the system has more than one alternative.

```
input_form( [X], answer(person(X)) ):-
        sem_sort(X,name).

input_form( [X], answer(contactName(X)) ):-
        sem_sort(X,name).

input_form( [X], answer(contactToCall(X)) ):-
        sem_sort(X,name).
```

In this case the system only finds the first one, regardless if the user really meant one of the other alternatives. The reason for this is the Prolog search method. It searches from top to bottom in the code. Then the internal system engine takes control. It finds that the name is a relevant answer to the first question. The system is satisfied and never checks the other alternatives.

These difficulties with keyword spotting could be solved by disambiguing the input by using the surrounding words. An alternative to the above *contactToCall* example, the user could help the system by specifying the input:

```
input_form( [till,X][5],answer(contactToCall(X)...
```

This makes it possible for the system to separate the different "name alternatives".

A better solution is to let the input concerning names be represented by something neutral like:

```
input_form( [X], answer(name(X)) ).
```

This soulution gives the more intuitively correct answer. All ambiguous inputs regarding names are mapped as a "name". This is more natural too. In a real conversation the participants would never add surrounding "help words" in order to simplify the dialogue.

6.4 Semsort

The *semsort* file contains the semantical representation of specific words. It decides what category/type a word belongs to.

```
sem_sort( vid_call_person, action ).
sem_sort( vid_add_contact, action ).

sem_sort( X, name ):-
        name(X).

sem_sort( number, informationType ).
```

[5] to X

In the examples above we see that *vid_call_person* and *vid_add_contact* are both actions. There must be an action for every one implemented in the system. The semantic representation of a name is approved only if it checks out to be a name by the *name*/1 predicate.

A very important semantic representation is the predicate representing an *isa-hierarchy*. This defines what "class" a type belongs to.

```
isa( person, name ).
isa( contactName, name ).
isa( newContactName, name ).
```

Using the `input_form( [X], answer(name(X)) ).` example from before we can see the solution above. The isa predicates states that the name the system is looking for could be either a person, a contactName or a newContactName.

Chapter 7

Results and Discussion

In this chapter I will give some comments on the result of my implementation. I will give my own opinions on dialogue systems and their use. Do they have any use and if so, in what areas?

7.1 Results

7.1.1 The over all system

I had a dream!

My intentions was to have a system that could control a video conference application. I did not find it necessary to have a working video application but it would be a nice bonus. Another bonus would be to have a good automatic speech recognizer. The one I have works, but it is slow working and it is not to any particular use.

The GoDiS system works and could be run as explained in appendix A. Even if the system is not attached to an application, the user should be able to see that the internal state/variables are correct.

When running my system via the web interface it would be quite possible to use a speech recognizer. However, in order to modify the Java applet enough to make it work nicely, I would have been forced to rebuild it. I did not have the time to make it work "as it should"; that is, the user should not have to click any buttons to send the input to the system. Now it works as a normal text input. The user types what he wants to say and then presses the send button.

I found using a dialogue system to control a video conference system was not only possible but also quite nice. The domain is small enough to model

in the system. As I thought, and as the survey showed, the user interaction was rather straight forward. The answers do not differ much form each other.

7.1.2 MyPhone

My idea was to attach my GoDiS application to the video conference application MyPhone. MyPhone looked easy enough to use. I did use its chat interface as a "stand alone" application and it had the necessary features I wanted.

Therefore I was quite disappointed when I discovered that it did not work with newer webcameras. The person/-s who implemented the application used old video codecs, thus it does not work with my camera. I noticed that the signal got through between the different applications, but I could not see the image.

Another problem, apart from not working, was that it did not have a reasonable API[1] or some guide on how to control it. That is what signals did the system need. I got only the source code and even if I have not used C++ or Visual-C before, it is possible to read the code with some difficultiy.

I decided not to work any further with this since it was not crucial for this thesis. It would of course have been nice to try it out but that was more of a bonus. However, I am convinced that it would be possible to control it with a dialogue system.

7.1.3 Example dialogues

I tried to use as many of the command alternatives from the survey as possible when implementing. Although if only one or two persons in the survey wanted an alternative I did not try to implement it.

Below I will give some examples of dialogues and commands that work in the system and some that do not. There will be some pseudo code in order to briefly show how the application sets variables in the device resource. Since my application is in Swedish the examples are translated to English.

[1] An abbreviation of Appllication Program Interface. A set of protocols, routines and tools for building software applications.

System = $S>
User = $U>

Test dialogues

In the first dialogue there is not much to see in the device file. There are no parameters to be set in this action. The only thing relevant that sets is the status of the call.

```
$S>:  What do you want to do?
$U>:  Answer


       dev_do(vid,'TakeCall')
       dev_set( call_status, online )

$S>:  Answering
$S>:  What do you want to do?
```

In the next example, there are more interesting actions to perform. In order to get the "AddContact" action done, the application must have two parameters; a contact name and an IP number. In this example the user gives all the information at one time.

```
$S>:  What do you want to do?
$U>:  Add Pelle two point two point one point one


      dev_do(vid,'AddContact')
      dev_set(variable_value(contactName,pelle)) \ setting
      dev_set(variable_value(number,'2.2.1.1'))  / temporary variables
      perform_command('AddContact')
      dev_get( contactName, Name )  where Name=pelle
      dev_get( number, Number )     where Number='2.2.1.1'
      dev_set( book, [contactName(pelle),number('2.2.1.1')] )

      variable_value( book, [contactName(pelle),number('2.2.1.1')])
         %% finally an entry added to address book

$S>:  Name added to address book
$S>:  What do you want to do?
```

These examples are only two short ones to give some notion of how the application works. This is the common procedure in order to set variables in the device resource in order to control the external device.

7.2 Discussion

Naturally the work with my system does not stop here. My plan is to get it to work as well as possible in accordance with my first thoughts. That is, I want it to really control an application.

A continuation of my system should naturally be to get a working video conference application and attach it to GoDiS. Since I do not like phones this is something that I would use in the everyday life. The desire of seeing the conversation partner is often greater then the desire of *having* a conversation.

Naturally the application is not enough in itself. There should be some working features in addition to the standard ones. A possible additional feature is an improved address book where the user has the possibility to add information about addresses, phone numbers and so forth.

Yet another possible development is to integrate an ASR into the system. The problem is not to get the input into the system. The problem is rather to have an ASR that knows when the user is addressing the dialogue system or just chatting with a friend.

As mentioned early in this thesis there are several ways to handle this. If the user will use microphone only to control his applications, then I believe that there must be some keywords for the system to listen to. When the dialogue system hears the word "Computer", it will know that the user is addressing it. Otherwise it should just listen to the conversation and ignore it.

Using dialogue systems as disability aids

An area that is of great interest to me is disability aids. Mainly those for communication. In this area I believe that dialogue systems could be a revolution. It shuold be a natural step to take when dialogue systems become more and more common. My opinion is that many users who are not capable of using an ordinary computer could learn to use a dialogue system. A dialogue system could be of assistance in many different situations.

A dialogue system used as communication aid could take different forms. It could be used to control a video conference application as in this thesis. Another use of a communication aid could be to help the user in the day to day conversations. The system could consist of several different domains. For example one domain for going to the bank, another when going shopping.

Chapter 8

Summary

The purpose with this thesis was to see whether it was possible to use a dialogue system to control a video conference application. Even though I could only do the underlying system, since the video application did not work properly, I found it to be very possible.

I am convinced that dialogue systems have a place in applications used for communication. It should make the applications easy to use even if one has not used it before. Using dialogue systems in this way is something that many people could benefit from; both those physically disabled and those not.

Bibliography

Ailomaa, M., Lundell, C., Wallentin, A., and Zsedényi, P. (2003). GoDiS.vma. Technical report, Department of Linguistics, Göteborg University.

Bohlin, P., Robin Cooper, Elisabet Engdahl, and Staffan Larsson (1999). Information states and dialogue move engines. *Gothenburg papers in computaional linguistics*.

Larsson, S. (2002). *Issue-based Dialogue Management*. PhD thesis, Department of Linguistics, Göteborg University.

Larsson, S., Alexander Berman, Leif Grönqvist, and Fredrik Kronlid (2002). *TrindiKit 3.0 Manual*. http://www.ling.gu.se/projekt/trindi//trindikit/docs/manual3.0.pdf.

Philips (2004). Philips speech processing. URL: `http://www.speech.philips.com/`.

Preece, J., Rogers, Y., and Sharp, H. (2002). *Interaction Design: beyond human-computer interaction*. John Wiley & Sons.

Suominen, E. (2004). Privaria. URL: `http://www.privaria.org`.

The FFmpeg Group (2004). FFmpeg Multimedia System. URL: `http://sourceforge.net/projects/ffmpeg/`.

The OpenH323 Project (2004). MyPhone. URL: `http://myphone.sourceforge.net/`.

The SIRIDUS Group (2004). The SIRIDUS homepage. URL: `http://www.ling.gu.se/projekt/siridus/`.

The Trindi Group (2004a). The Trindi project information. URL: `http://www.ling.gu.se/projekt/trindi/`.

The Trindi Group (2004b). TrindiKit homepage. URL: `http://www.ling.gu.se/projekt/trindi/trindikit/`.

Appendix A

User's guide to starting the system

This is only a very short guide on how to run the system. There is a slight difference on how to start your own system the first time and how to start the one I used in the thesis.

Find your web browser of choice and open the following URL:
`http://www.cling.gu.se/~cl0awall/webinterface/wia.html`

If the applet does not open or connect, it is because my application server is not up and running.

Click either VID or VID(bin). The latter is faster to start since it is compiled. Now you should be able to "control" a video conference application.

Sometimes you cannot type the Swedish "å,ä,ö", but it seems to be a computer related problem. It seems to depend on where you open the browser from, which browser it is and so forth.

Appendix B

Survey form

The survey(in Swedish) can be found at the following URL:
`http://fs16.formsite.com/Talsor_Inc/form719968246/index.html`

There were some multiple choice questions, then some free text questions referring to specific features of my application.

- Gender
  - o male
  - o female
- Age
  - o - 20
  - o 21 - 25
  - o 26 - 30
  - o 31 - 35
  - o 36 - 45
  - o 46 - 55
  - o 56 -
- Computer Skill
  - o Beginner
  - o Uses a computer now and then
  - o Uses the coputer often (in private or at work)
  - o Advanced user
  - o Able to make my own applications and programs

- Have you used any similar application? (dialogue system)
  If yes, which one?
  - o Yes
  - o No

I would appriciate if you answered the questions below. Try not to think in "computer terms" but instead how you *would like* to say.

- What do you say when you want to start conversation with X?
- What do you say when you want to end the current conversation?
- What do you say when you want to accept an incomming call?
- What do you say when you do NOT want to accept an incomming call?
- What do you say when you want to use the address book to:
  - call from?
  - check information about a person?
- What do you say when you want to add a person to the address book?
- What information do you want to have?
- What do you say when you want to delete an entry in the address book?
- What do you say when to abort the latest input?
- What do you say when you want to quit the video application?
- Comments and opinions on what the system could need. Own ideas on what you could say?