

Development of a voice-driven dictation application

Ulrika Olsson

`ulrikol@stp.ling.uu.se`

Master's thesis in Computational Linguistics

Språkteknologiprogrammet

(Language Engineering Programme)

Uppsala University · Department of Linguistics and Philology

4th June 2004

Supervisors:

Mats Dahllöf, Uppsala University

Jaan Kaja, Icepeak AB

Abstract

Applications combining speech recognition and telephony are becoming more and more common as tools for helping people with such things as timetable information and directory assistance. This master's thesis describes the development of one such application for dictation, embedded in a system, developed by Icepeak AB, with several other voice driven services used over the phone.

In a dialogue with the system the user dictates and edits her/his message. Several commands are available to the user, such as removing or adding phrases, getting parts or the whole message read by a speech synthesizer etc. The result of the dictation is then put in a file, named after the time the message was created and what kind of message it was. A message is of a certain type, called a template. This can be regarded as a kind of lexicon, a phrase collection, from which the phrases used to compose the message are gathered. This approach makes the set of utterances which have to be recognized much smaller and therefore the recognition process is more stable.

The thesis also describes the design of a web interface where the user can view and control the contents of the database: the templates and the phrases belonging to them.

Contents

List of Figures	iii
Acknowledgements	iv
1 Introduction	1
1.1 Purpose	1
1.2 Outline	1
2 Dialogue design	2
2.1 Speech recognition software	2
2.2 Defining the dialogue	2
2.3 Nuance grammars	3
2.3.1 Grammar syntax	3
2.3.2 Different kinds of grammars	4
2.4 Testing grammars	5
3 Grammars for dictation	8
3.1 Grammar development	8
3.1.1 Main grammars	8
3.1.2 Grammars for time and date	9
3.1.3 Dynamic grammars and the database	10
3.2 Dialogue state analysis	12
3.2.1 The states	12
3.2.2 Dialogue state interaction	13
4 Coding the dialogue engine	14
4.1 Program design	14
4.2 Structure of state functions	15
4.3 Slots filled by phrases	16
4.4 Inserting dynamic grammars	17
4.5 The post function	18
5 Evaluation of the dictation function	19
5.1 Approach	19
5.2 Users' comments	19
5.2.1 General impression and remarks on the dialogue	19
5.2.2 Problems	20
5.2.3 Future changes and development	20
5.3 Summary of the evaluation results	21

6	Design of a web interface to the database	22
6.1	Template overview	22
6.1.1	Renaming and removing a template	23
6.1.2	Adding a template	23
6.1.3	Editing the contents of a template	23
6.2	Copying phrases between templates	24
6.3	Modifying phrases in multiple templates	25
6.3.1	Selecting the phrase	25
6.3.2	Editing the phrase	26
7	Concluding remarks	28
7.1	The dictation component	28
7.2	Discussion	28
7.3	Future development	29
	References	30
A	Dialogue flow charts	31
B	Evaluation questions	33
C	Interface images in English	34

List of Figures

2.1	The last dictation state	3
2.2	Xapp, a graphical grammar testing tool	7
3.1	Table with templates	11
3.2	Table with phrases	11
3.3	State interaction view	13
4.1	General state function structure	15
5.1	Example of system-user interaction	20
6.1	Showing template overview	23
6.2	Adding a new template	24
6.3	Editing a template's contents	25
6.4	Copying phrases	26
6.5	Selecting a phrase	27
6.6	Editing a phrase	27
C.1	Showing template overview	34
C.2	Adding a new template	35
C.3	Editing a template's contents	35
C.4	Copying phrases between templates	36
C.5	Selecting a phrase	36
C.6	Editing a phrase	37

Acknowledgements

First of all I would like to thank my supervisors Jaan Kaja and Mats Dahllöf for their support and advise throughout the process of writing the report and doing the work upon which it is based. I am also very grateful to the others at Icepeak who have helped me develop this application. Some I would like to thank in particular: Tim Hansen for giving me the idea, Rehan Mirza for explaining the Icepeak Attendant code and above all Jörgen Hartikainen, who helped with everything from recovering crashed data to saving the breakfast.

I would also like to express my thanks to Annlouise Olsson for spending days proofreading this text and for her insightful comments not only on the structure of the text but also on the facts expressed in it.

Finally I would like to thank my boyfriend Klas for his support.

1 Introduction

Speech recognition is used more and more in applications designed for supporting people in their everyday life, as professionals as well as private persons. A majority of these voice-driven products combine speech recognition with telephony, providing different kinds of service like timetable information, booking of tickets, directory assistance etc.

This thesis describes the development of a dictation service to be integrated into a system called Icepeak Attendant.

1.1 Purpose

The aim of this thesis is to add a dictation application to Icepeak Attendant. This is a product developed by Icepeak AB, a company with computerized business telephony as its main interest. Icepeak Attendant is an automatic operator and a personal assistant, with several services available to the customer: calling, booking of conference rooms or cars, directing their phones etc. Each service can be reached by phone or by using a web interface. A dictation application would extend the service given to the customers and further promote the modern idea of the *mobile office*.

The present task includes development of a program controlling the dialogue between computer and user, with technology for speech recognition developed by Nuance Communications Inc. The task also involves constructing a web interface design, where the user can manipulate the database contents which form the basis for recognition.

The dictation is based on templates with contents decided by the user. A phrase must be defined as part of a template in order to be recognized. This makes the recognition more stabile, since the vocabulary is finite and fewer cases of ambiguity will occur. It also enables a high degree of user friendliness, since it can be adapted accordingly to the user's own wishes. Templates can be added, removed or renamed and their contents changed.

1.2 Outline

This thesis is divided into seven parts. Chapter 2 will give a brief account of how dialogues are designed. It will also give an introduction to grammar writing and testing in a system using Nuance technology. Chapter 3 will give a detailed view of the grammars developed for this application, describe the dynamic grammars and present the states of the dialogue engine. The latter will then lead to chapter 4 where the actual application, the coding of the dialogue engine, is described. An evaluation of the telephone interface of the dictation application is described in chapter 5. In chapter 6 the design of the web interface will be accounted for. The concluding chapter 7 will complete this thesis by giving a summary of the work presented here and discuss future developments of the dictation application.

2 Dialogue design

2.1 Speech recognition software

In this application the software used for recognition is developed by Nuance Communications Inc. (see www.nuance.com for more information). The Nuance system is a collection of software used for dialogue applications with speech recognition as the core component. Other features are speaker verification, web-server solutions and telephony control software. All these components are described in Nuance System (1998).

Nuance technology is used all over the world and supports over 20 languages or language varieties, such as American English, Canadian French, Swedish and Cantonese. The software is one of the most widely used in products and applications incorporating speech recognition over the telephone. It consists of several parts working together, but in sum the system recognizes and analyzes voice input to make an interpretation of the speaker utterance. This core meaning is then used for determining the appropriate action.

Special rule collections called grammars, written for each application, form the basis for recognition. Their appearance and the process of building them is described later on in this chapter.

2.2 Defining the dialogue

Before any grammar can be written, some kind of plan for the dialogue must be made. It must be firmly set what kind of information the system will need from the user. The user must also know what choices are available based on an utterance from the system. Such an utterance is called a *prompt*, and consists of a .wav file containing a pre-recorded audio segment.

Designing the prompts is an important part of this work. If this isn't done properly the prompts might cause problems, both for the developer and for the user. To avoid this it is important to make them as unambiguous as possible. This makes it easy for the user to know which information is requested, thus minimizing the different ways the prompts can be responded to. The grammar work will be less demanding, since you have to predict fewer cases. It also lightens the burden on the user, since the prompts hopefully will lead to effective comprehension and minimal level of misunderstanding.

The prompts should be designed before the grammar writing begins. One reason for this is that the wording of a prompt affects the caller responses predicted in the grammar. Also, if the prompts are changed in the midst of grammar development you might have to do a lot of extra work rewriting grammars.

Prompts are designed to fit either a *mixed-initiative dialogue* or a *directed dialogue*. In a mixed-initiative dialogue several information items are requested at the same time with a prompt like "What would you like to do?", which has many possible answers. A similar prompt in a directed dialogue would be "Do you want to continue or finish?", narrowing the choices available to the user.

A mixed-initiative dialogue feels more like a natural conversation, while a directed dialogue may be perceived by the user as less fluent. When using the latter however (as in this project), the dialogue is

considerably less complex and so is the grammar writing.

A graphical help in defining the dialogue is to construct a *dialogue flow chart*. The prompts and user responses are grouped together, forming states as in a finite automaton. The prompts and user utterances are connected within a state and there are also transitions to other states. With this tool it is possible to produce an overview of the dialogue. It helps the developer to see what responses a particular prompt can give rise to and if additional prompts may be needed to take care of some user utterances.

The chart is also an asset later on in the application development process, since it is a visual description of the main program. Figure 2.1 shows the last state from the appended dictation flow chart, translated to English. Prompts are represented by squares and user utterances by ovals:

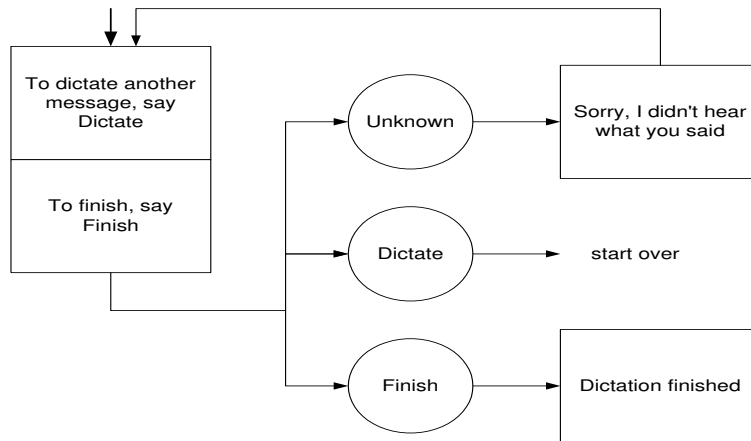


Figure 2.1: The last dictation state

2.3 Nuance grammars

2.3.1 Grammar syntax

Writing grammars is a procedure all about predicting what the caller will say to the system. They are coded in *GSL*, Grammar Specification Language. A grammar consists of a set of rules and these too are called grammars, which can be confusing for readers used to other usage of this word. However, in this essay it will be used in the Nuance sense: *grammar* refers to both grammars and the rules within them.

The syntax of GSL is quite simple, as this example shows. Details not explained here will be discussed later on:

```

Continue
[
    (fortsätt ?tack)
    (?I_WANT fortsätta)
]

```

- A grammar name must begin with at least one uppercase letter. In the example, the grammars are `Continue` and `I_WANT`.
- Terminal symbols, i.e. words like *fortsätt* (*continue*), must contain only lowercase letters.
- Square brackets, `[]`, represent disjunction. In the above example, it means that the command `Continue` can be expressed either as *(fortsätt ?tack)* or *(?I_WANT fortsätta)*.

- Parentheses, (), define a sequence of grammar elements, i.e words and grammars.
- Question mark, ?, stands for optionality. In the example, we see that both *fortsätt* and *fortsätt tack* are valid ways for the user to express a will to continue.

Other characters allowed in grammar names are underscore, dash, single quote, "at"-sign and period. The period has a special quality however, namely to distinguish between *subgrammars* and *top-level grammars*. The meaning of this will be explained in the next section.

2.3.2 Different kinds of grammars

In the example in section 2.3.1, both `Continue` and `I_WANT` are subgrammars. Top-level grammars, on the form `.NameOfGrammar`, can be referenced to by an application only. Subgrammars, on the other hand, can be referenced to by top-level grammars or by other subgrammars. The reference to `I_WANT` in `Continue` is an example of the latter. `I_WANT` is a grammar defined in another grammar file, which is included at the top of the file where `Continue` is defined in a way similar to the C programming language:

```
#include NameOfGrammar.grammar
```

When this file is compiled, `I_WANT` is replaced by its contents:

```
(jag [(vill ?gärna) (skulle ?gärna vilja)])
```

This means that for example *jag skulle vilja fortsätta (i would like to continue)* is accepted by the `Continue` grammar.

The difference between the grammars `I_WANT` and `Continue` is that `I_WANT` describes a *filler*. A filler is something that doesn't carry any information the system needs to know for interpretation. It is words or often polite expressions wrapping the *core*. The contents of the core are important to the system, in the sense that they tell the system what the user wants to do.

The information of the core is used to fill slots with the information necessary for the system to perform its tasks. As an example, let us look at the grammar for all commands in the dictation dialogue, where we also see a filler for hesitant starts of utterances:

```
Command
( (HESITATION)
[
    Abort          {<command abort>}
    Add            {<command add>}
    Continue       {<command continue>}
    Dictate        {<command dictate>}
    Done           {<command done>}
    Finish         {<command finish>}
    Help           {<command help>}
    Next           {<command next>}
    ReadAll        {<command read_all>}
    ReadLatest     {<command read_latest>}
    RemoveAll      {<command remove_all>}
    RemoveLatest   {<command remove_latest>}
    Repeat         {<command repeat>}
    Replace        {<command replace>}
    Yes_Words      {<confirmation yes>}
    No_Words       {<confirmation no>}
])
```

The slots we can see here are `command` and `confirmation`, which get different values depending on which grammar is activated. This is an example of assigning different values to one slot. Another slot-filling technique is using variables:

```
Call
(
  (ring ?till)
  (?HESITATION)
  DynamicPersonalName:n
  ?[[TimeDate DateTime] ([TimeDate DateTime]
    ?([eller och][TimeDate DateTime]))]
) {<personal_name $n>}

DynamicPersonalName:dynamic
```

In this example the variable *n* gets the value of what is returned by `DynamicPersonalName`, and the expression at the bottom within curly brackets fills the slot `personal_name` with the value of the variable *n*.

`DynamicPersonalName` in the example is a *dynamic* grammar, in contrast to the other grammars that are called *static*. A dynamic grammar can be created and modified at runtime, when the application is used. The recognition package doesn't have to be recompiled, the grammar can be used directly. Dynamic grammars are used when caller responses depend on something outside the program. `DynamicPersonalName` for example, is used to recognize the names in the user's personal phone book, whose contents naturally cannot be described by a static grammar, since it is often modified and changed.

The expression in the example containing the grammars `TimeDate` and `DateTime` in the example may seem complicated but will be described in section 3.1.2.

2.4 Testing grammars

Compiling a grammar is done with `nuance-compile`:

```
nuance-compile Dictation Swedish -auto_pron
```

Dictation is the name of the grammar and *Swedish* is what the master package is called, where language specific pronunciation information is defined. There are several options that can be used with `nuance-compile` and here we see the option *auto_pron*. The effect of this is that all words not found by the compiler in the dictionary are assigned a probable pronunciation and put in a file that in this case will be called `Dictation.missing`. If this option isn't used and the compiler finds out-of-dictionary words the compilation will fail.

As a developer you want your grammar to be as correct as possible and there are several ways of testing what you have produced. One tool for conducting coverage tests, i.e. that the grammar correctly handles phrases you want it to handle, is `parse-tool`. An example:

```
parse-tool -package Dictation -print-trees
Ready
jag vill avbryta
Sentence: "jag vill avbryta"
```

```
.Dictate
Command
  Abort
    I_WANT
      jag
      vill
      avbryta

ring ullis den andre december
Sentence: "ring ullis den andre december"
```

```
.Dictate
GeneralPhrase
  Call
    ring
    ullis
    TimeDate
      SODate
        SODate_CORE
          den
            SODate_DayOrd
              andre
                SODate_Month
                  december
```

With this tool you can see if the grammars are able to parse given phrases successfully. If so, the option *-print-trees* displays which grammars are activated and how. Phrases not covered by the grammar are simply marked with *no parses*. In the above example the selected phrases activate the top level grammar from the Dictation package: *.Dictate*, that called *Command* and *GeneralPhrase*. These in turn trigger a number of other subgrammars, as can be seen in the parse trees. The most important ones of these subgrammars will be more carefully described in the following chapter.

One of the best ways to find out how your grammar works is to test what it can generate. In this way you can easily detect the case of overgeneration, i.e. that the grammar accepts phrases outside the set of phrases it is meant to cover, e.g. phrases that are incorrect, ungrammatical or inappropriate.

```
generate -package Dictation -grammar .Dictate -num 5
```

The tool will generate possible sentences randomly. This can be done until the program is stopped, but here the option *-num* and its value specifies that only five phrases should be generated:

```
lägg till tack
upprepa
um jag vill ha allt uppläst
um det är bra
ehh jag vill ersätta
```

If something like **jag är klar färdig* (**i'm ready done*) would be generated, it would point at a problem in the grammar that needs attention.

The tools described above both operate on standard input and standard output as default, with text as input. If you want to test your grammar in an environment more like the one it will be used in when your application is finished, there is a program called *Xapp*.

This graphical tool is used in the way that you produce audio input, i.e. speak phrases, to the system that tries to recognize it with the help of the grammar you specify. If this is successful, a transcription of the phrase and a confidence score are displayed in the application window. The score is a number between 0 and 100, which marks the system's certainty that the recognized utterance actually is what was said. If the recognized phrase matches a natural language command which fills a slot, this is also shown. Figure 2.2 shows the result after the application has received and recognized the phrase *ja tack* (*yes please*):

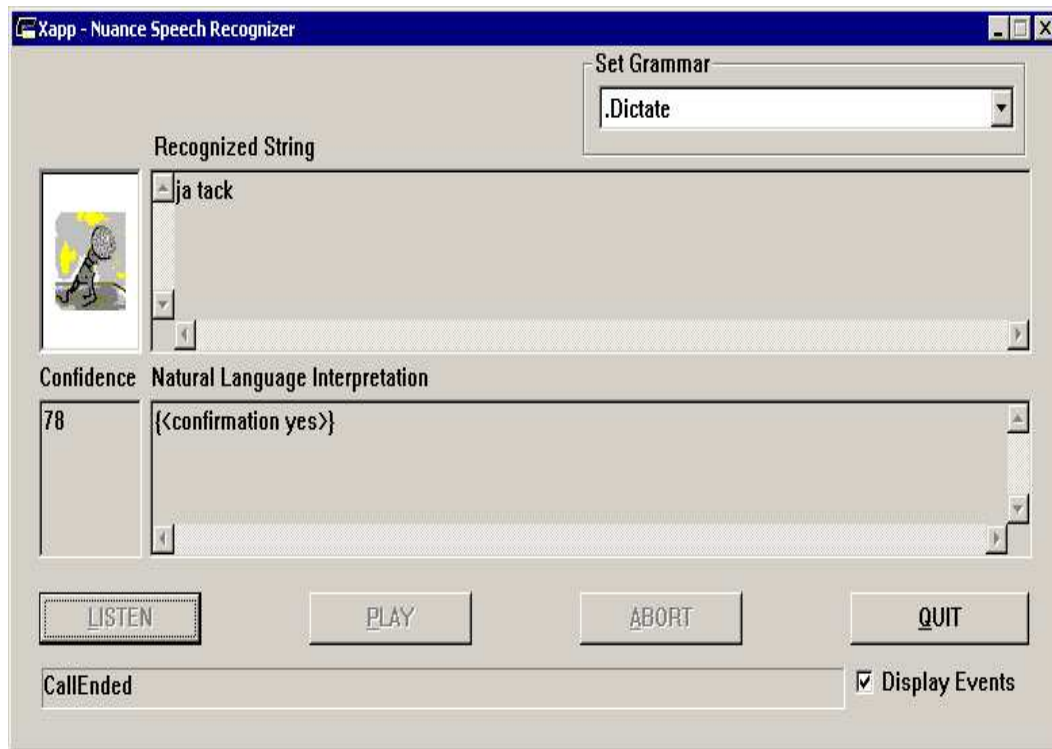


Figure 2.2: Xapp, a graphical grammar testing tool

This tool enables the grammar developer to find out if the uttered phrases are described by the grammar. Evaluation by using the voice is easier and comes more naturally than trying to make an exhaustive list of what the user might say; some parts are always forgotten and this is an excellent way to discover them.

3 Grammars for dictation

3.1 Grammar development

3.1.1 Main grammars

For this dictation project several grammars were needed for the commands and the phrases users could want to dictate.

To get from the main menu to the dictation dialogue required a voice command, and a grammar describing this was developed:

```

DICTATION
[
    diktering
    (?I_WANT diktera)
]

```

There is a reference to this grammar in Icepeak Attendant's main grammar, and thereby the connection between the applications is made.

All user utterances within the dictation program are described by the top-level grammar `.Dictate`:

```

.Dictate
[
    Command
    Template
    GeneralPhrase
]

```

This grammar consists of three subgrammars developed for capturing different kinds of user input. These are `Command`, `Template` and `GeneralPhrase`.

The user navigates through the process of dictating a message with commands like *abort*, *add*, *replace* etc. `Command` handles, as indicated by the name, all such commands in the dictation dialogue and also positive or negative confirmation from the user, i.e. *yes* and *no*. This is done with subgrammars for every kind of command that are put together in this grammar. `Command` is shown in section 2.3.2 and one of its subgrammars, `Continue`, is displayed in 2.3.1.

`Template` is a grammar whose real contents are dynamic, collected from the database where all template names the user has entered are saved.

```

Template
((?HESITATION)
    (DynamicTemplate:t)
) {<template $t>}

```

```
DynamicTemplate:dynamic
```

GeneralPhrase treats the actual contents: words and phrases of the dictated message. It consists of four subgrammars: Call, Tender, Meeting and DynamicPhrase. The latter is a dynamic grammar, which captures all phrases the user has put in the vocabulary for the current template used to create the message. The other three grammars define the phrases present from the beginning in all templates, about calling, booking a meeting or submitting a tender to someone with the option to attach some expression of time. These grammars also have a dynamic element through DynamicPersonalName, described in the paragraph about static and dynamic grammars at the end of section 2.3.2. The Call grammar is also shown there.

```
GeneralPhrase
( (?HESITATION)
[
    Call
    Tender
    Meeting
    DynamicPhrase:p
]) {<phrase $p>}
```

```
DynamicPhrase:dynamic
```

DynamicPhrase is a dynamic grammar that gets its contents from a database, updated in the web interface. All phrases that belong to a certain template, chosen by the user in the beginning of the dictation dialogue, are selected from the database and form this grammar.

Unlike DynamicPhrase, the grammars Call, Tender and Meeting don't fill the phrase slot although they are regarded as phrases. Why they don't and how they do work is shown with an example:

```
Tender
(
    ([skicka lämna] offert till)
    (?HESITATION)
    DynamicPersonalName:n
    ?[[TimeDate DateTime] ([TimeDate DateTime]
    ?([eller och][TimeDate DateTime]))]
) {<personal_name $n>}
```

As can be seen, the return value in the variable *n* from DynamicPersonalName fills the *personal_name* slot. Therefore the Tender grammar cannot return anything to fill the phrase slot. In the actual code this have to be solved in a special way, described in section 4.3.

3.1.2 Grammars for time and date

In section 2.3.2 the grammars TimeDate and DateTime were part of an example but got no further explanation or description there. They use modified grammars for time and date, called Time and Date:

```
TimeDate
(
    Time ?Date
)
```

This grammar describes phrases consisting of a time expression followed by an optional date expression, like *klockan fem på tisdag* (five o'clock on tuesday) or *på eftermiddagen* (in the afternoon). It can be seen as a concatenation of Time and Date, as well as DateTime:

```

DateTime
(
    Date ?Time
)

```

Phrases described by this grammar are those that incorporate a date expression and an optional time expression, like *om tre veckor på eftermiddagen* (in three weeks in the afternoon) or *den andra mars* (2nd of March). The two grammars `TimeDate` and `DateTime` described here, are mirrors of one another and when they are used together they enable flexible phrase construction. An example of this is an example displaying the last of the phrase grammars to be shown:

```

Meeting
(
    ([boka avboka (ställ in)][(?ett möte) mötet] med)
    (?HESITATION)
    DynamicPersonalName:n
    ?[[TimeDate DateTime] ([TimeDate DateTime]
    ?([eller och][TimeDate DateTime]))]
) {<personal_name $n>}

```

This combination of the grammars describes utterances like *på torsdag nästa vecka eller på fredag cirka klockan tre* (thursday next week or friday around three o'clock). `TimeDate` and `DateTime` are just abbreviations of combinations of the `Time` and `Date` grammars; if the original ones would be used the expression above would be twice as long and the complication factor multiplied.

The grammars `Time` and `Date` are only slightly modified versions of existing grammars built by other developers for use in different applications within Icepeak Attendant. It is therefore impossible to show pieces of them, but the modifications are described below.

The time and date grammars had to be slightly modified to suit their new purpose. Earlier they were used to describe utterances in situations like directing a user's phone or booking of conference rooms. This meant that phrases like *återkommer om en vecka* (back in a week) were accepted, but would be ungrammatical in the dictation dialogue: **ring ullis återkommer om en vecka* (*call ullis back in a week).

Modification of the time grammar also enabled phrases like *på kvällen* (in the evening), without its previously compulsory time specification like *sju på kvällen* (seven in the evening). Of course, such expressions are also accepted by the modified grammar.

Earlier there was a filler that described phrases like *tre nollnoll* (three hundred hours) and *fyra snåret* (about four o'clock), but in the new grammar *nollnoll* was separated from *snåret* and its fellow expressions *draget*, *bläcket* and *tiden*. This was done because one wish was to use more pre-fillers like *vid* (at) and *ungefär vid* (at about) in addition to the existing *cirka klockan* (around ... o'clock) and similar ones: *cirka klockan sju* (around seven o'clock) is a valid expression but **cirka klockan tre snåret* (*at about around three o'clock) isn't.

3.1.3 Dynamic grammars and the database

All dynamic grammars depend on some data stored in the icepeak Attendant database. Two new tables were needed for storing templates and phrases. This SQL query creates the table *templatetable*:

```

CREATE TABLE public."templatetable"
(
    templatename varchar(100),
    CONSTRAINT "templatetable_pkey" PRIMARY KEY (templatename)
) WITH OIDS;

```


Here the dynamic grammar `DynamicTemplate` will find its contents in the column *templatename*, where some data can be seen in the figure.



	oid	templatename	varchar
1	17726	anteckning	
2	17696	att göra-lista	
*			

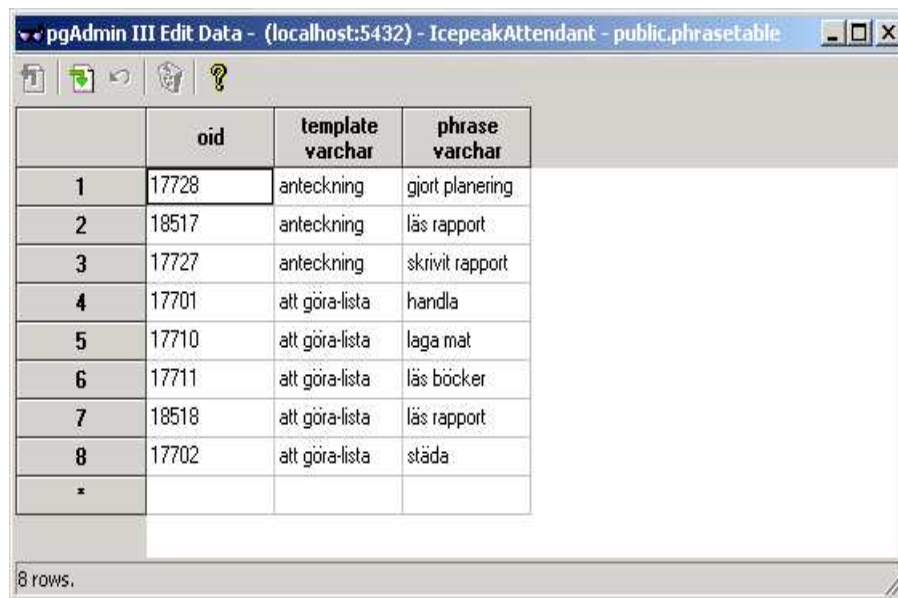
2 rows.

Figure 3.1: Table with templates

The table *phrasetable* is created in this way:

```
CREATE TABLE public."phrasetable"
(
    template varchar(100),
    phrase varchar(300),
    CONSTRAINT "phrasetable_pkey" PRIMARY KEY (template, phrase)
) WITH OIDS;
```

`DynamicPhrase` consists of the phrases stored in the column *phrase*. In this table there is also a column *templatename*. Here the template is found, to which the phrase belongs.



	oid	template	phrase	varchar
1	17728	anteckning	gjort planering	
2	18517	anteckning	läs rapport	
3	17727	anteckning	skrivit rapport	
4	17701	att göra-lista	handla	
5	17710	att göra-lista	laga mat	
6	17711	att göra-lista	läs böcker	
7	18518	att göra-lista	läs rapport	
8	17702	att göra-lista	städa	
*				

8 rows.

Figure 3.2: Table with phrases

The dynamic grammar `DynamicPersonalName` collects its contents from the column *name* in the table *phonebook*. This table existed earlier in the database and contains proper nouns and information about a person's company, departement, telephone number etc.

3.2 Dialogue state analysis

3.2.1 The states

The dictation dialogue is divided into eighteen states, named after what their function or task is:

- ① dictationStartState
- ② dictationTellTemplatesState
- ③ dictationConfirmTemplateState
- ④ dictationPrePhraseState
- ⑤ dictationFunctionsState
- ⑥ dictationPhraseState
- ⑦ dictationDoneState
- ⑧ dictationAbortState
- ⑨ dictationReadLatestState
- ⑩ dictationReadAllState
- ⑪ dictationRemoveLatestState
- ⑫ dictationRemoveAllState
- ⑬ dictationPreEditState
- ⑭ dictationEditState
- ⑮ dictationReplaceState
- ⑯ dictationAddState
- ⑰ dictationPostEditState
- ⑱ dictationFinishState

Each state consists of one or several prompts and possible user responses. In the appendix there are images of the complete dialogue flow charts, with the states numbered. All prompts and expected user responses are shown. Arrows represent both connections within a state between prompts (squares) and user utterances (circles), as well as transitions between states. To contrast the two kinds of arrows from another the latter kind is drawn with a thicker line.

These charts are in Swedish, but figure 3.3 shows a simplified view in English, displaying transitions between the states and by what command etc. they are triggered.

In a majority of the states there is a possibility for the user to get the prompt repeated by the voice command *repeat*. Sometimes *help* has the same function. In most of the cases though, this option leads to another state where the user is provided with more information about the choice and available options.

If a user utterance isn't recognized or understood by the system, an excusing message is played, followed by a transition back to the same state. This results in a repetition of the previous prompt and the user can try again.

3.2.2 Dialogue state interaction

In state ① the user is greeted by a prompt asking for the name of the template that will be the basis of the message. If help is requested the user is taken to ②, where the available templates are synthesized and she/he is urged to make a choice. Speaking a correct template name leads to ③, where the user answers *yes* or *no* when the system synthesizes the recognized template.

Positive confirmation results in transition to ④, that initiates the dictation process. If the user produces a valid phrase here she/he is taken to ⑥, where most of the dictation work is done. The program will loop here as long as the user produces phrases. It will keep the valid ones and alert the user if incorrect expressions are produced.

The user can at any time ask for help, something that takes her/him to ⑤. Here the different commands available during dictation are prompted. From here the transitions lead either back to ⑥ or to the states corresponding to the commands: ⑦, ⑧, ⑨, ⑩, ⑪ and ⑫.

State ⑦ is activated when the user has finished the message. The user is asked if she/he wants to edit the message. If the answer is *no*, the next state is the finishing state ⑮, if *yes* it is ⑬. The function of state ⑬ is only to play the prompt containing information about the editing commands and doing the transition to state ⑭, where the actual editing process begins.

In this state the phrases are synthesized by the system and the user must after each one decide what to do with it, keep it as it is or remove it. Other options are also available. If the user decides to replace the phrase with another she/he is taken to ⑮. A decision to accept the phrase and add a new one leads to ⑯.

In ⑮ the user must produce a valid phrase. This is then confirmed and the application continues to state ⑭ if there are any phrases left to edit, otherwise to ⑰. State ⑯ has a function similar to ⑮ and transitions go to ⑭ if there are phrases left and to ⑰ if that isn't the case.

State ⑰ ends the dictation process and contains a prompt asking if the user wants to listen to the message and edit again, or if she/he is satisfied. If the answer is *yes*, the process starts over again in state ⑬. If *no*, the user is taken to the last state, ⑮.

In ⑮, the user can choose between dictating another message (transition to ①) or leaving the dictation dialogue for Icepeak Attendant's main menu.

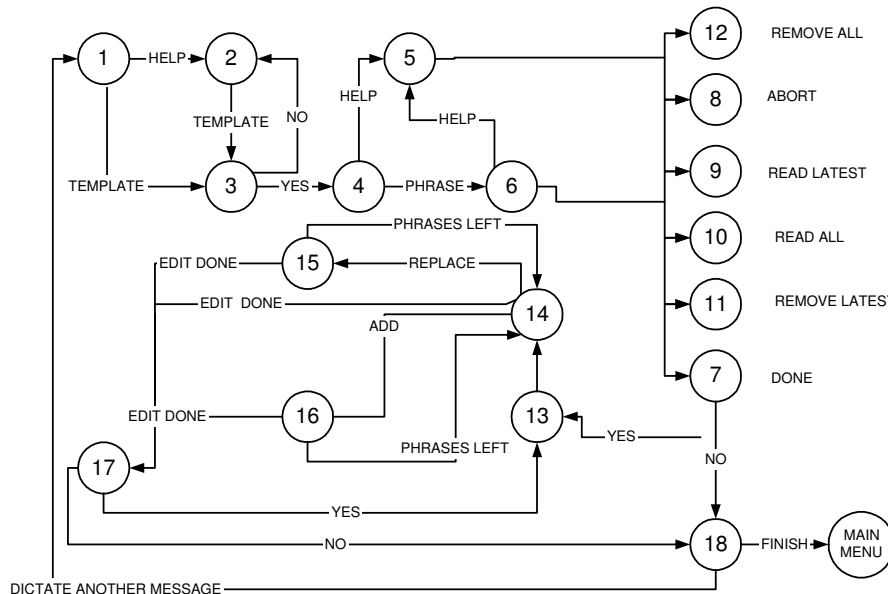


Figure 3.3: State interaction view

4 Coding the dialogue engine

4.1 Program design

The main program is actually the dialogue flow chart coded in C++ (see Skansholm (1996) for information about C++). The dialogue engine consists of eighteen states and these are represented in the code by functions with names similar to those of the states mentioned in section 3.2.1. For example, the task of `dictationStartState` is carried out by the function `fn_dictation_start`. The functions can be regarded as being the concrete implementations of the abstract states.

The structure of the program is quite simple: the incoming call is handled by the main program of Icepeak Attendant, which transfers it to the dictation program when the dictation command is used. In the same moment the dynamic grammars `DynamicPersonalName` and `DynamicTemplate` are constructed and inserted to last the length of the call. The first function is done and then the other states' tasks are carried out if and when they are activated.

The prompts played in a particular state are prepared in a state preceding it, where they are put in a prompt queue. Speech synthesis and pauses can also be added to the list, which is then played in the next state in connection to the recognition. Recognition is done while the prompts are played, to allow so called *barge-in*. This means that the user doesn't have to listen to the whole prompt before responding but can interrupt at any time, enabling better flexibility in the handling of users at different skill levels.

An experienced user, who is familiar with the options and commands, can thus save time and avoid frustration when using the program. A user with less experience, on the other hand, can listen to all information and then make a choice.

Something which makes the code rather compact is the use of macros, i.e. functions or commands available for use in every part of the Icepeak Attendant application. Macros are marked with capital letters. Examples of such a common macro function is `TRY_QUEUE_PROMPT`, which adds a prompt to be played. Another is `TRY_SET_NEXT_STATE`, which handles the transitions between the states.

Error handling is of crucial importance. In this application functions with a return value often return a pointer to some kind of error object. If this object is `NULL`, the function succeeded and if it isn't, it failed and the nature of the error must be checked. You could say that the code is built around a view that may seem backwards for some but is really effective. The focal point isn't if the functions succeed, but if they don't and why this happens. If it is a minor error, a report is written to the log file, but in the case of more serious errors or technical difficulties the program is aborted.

The call can also be aborted if the user produces too many incorrect utterances. This could be due to the fact that the user has a voice or a dialect, which the system has problem recognizing or interpreting. It could also be the case that the user calls from a noisy environment or uses a bad phone line, creating technical difficulties for the recognition software.

4.2 Structure of state functions

The states all share a similar design, shown by figure 4.1. The structure will be extensively described and exemplified in this section.

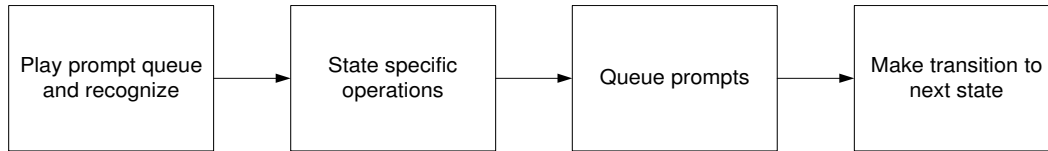


Figure 4.1: General state function structure

The general structure is exemplified below by parts of the `fn_dictation_functions`, representing `dictationFunctionsState`.

```
grammar = ".Dictate" + m_grammarSuffix;
setGrammar( grammar );
recError = recognize( ivrChannel, &recResult, stateMachine, state );
```

First it is decided which grammar should be used. In this case there is only one top level grammar, which is used all the time: `.Dictate`. After this the function `recognize` plays the prompt list (in this function containing a prompt with information of the available commands) to the user via the outgoing audio channel `ivrChannel` and recognizes the user utterance. It is then stored in the Nuance object `recResult` to be analyzed further down in the code.

Depending on what natural language command slot is filled by the recognition result, prompts are queued and transitions to the next state are made:

```
if( strcmp( pRecResult, "repeat" ) == 0 || strcmp( pRecResult, "help" ) == 0 ){
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_fortsätt" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_klar" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_avbryt" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_läs_upp_senaste" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_läs_upp" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ta_bort_senaste" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ta_bort" );
    TRY_SET_NEXT_STATE( state );
    return;
}
else if( strcmp( pRecResult, "done" ) == 0 ) {
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_lyssna_redigera" );
    TRY_SET_NEXT_STATE( dictationDoneState );
    return;
}
else if( strcmp( pRecResult, "abort" ) == 0 ) {
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_diktering_avbröts" );
    TRY_SET_NEXT_STATE( dictationAbortState );
    return;
}
else if( strcmp( pRecResult, "read_latest" ) == 0 ) {
    TRY_SET_NEXT_STATE( dictationReadLatestState );
    return;
}
```

```

else if( strcmp( pRecResult, "read_all" ) == 0 ) {
    TRY_SET_NEXT_STATE( dictationReadAllState );
    return;
}
else if( strcmp( pRecResult, "remove_latest" ) == 0 ) {
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_raderad" );
    TRY_SET_NEXT_STATE( dictationRemoveLatestState );
    return;
}
else if( strcmp( pRecResult, "remove_all" ) == 0 ) {
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_meddelandet_raderat" );
    TRY_SET_NEXT_STATE( dictationRemoveAllState );
    return;
}
else if( strcmp( pRecResult, "continue" ) == 0 ) {
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_varsågod_diktera" );
    TRY_SET_NEXT_STATE( dictationPhraseState );
}
else {
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ursäkta" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_fortsätt" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_klar" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_avbryt" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_läs_upp_senaste" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_läs_upp" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ta_bort_senaste" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ta_bort" );
    TRY_SET_NEXT_STATE( state );
    return;
}

```

The string `pRecResult` is what the system believes the user uttered, after having analyzed the raw material of the object `recResult` (seen in the last example). In a number of `if ... else` clauses the string `pRecResult` is compared to strings which constitute the desired values of the slot in question. If the difference is zero, the strings are identical and this means that the recognized result is classified as a valid value of the slot.

The user utterance may fill another slot aside from the desired ones. In this state the user might try to dictate a phrase. The utterance would then be interpreted as filling the `phrase` slot, which is unavailable in this state. The treatment of other slots being filled is seen at the end of the example, in the last `else` clause. A prompt “Ursäkta, jag hörde inte vad du sa” (“Sorry, I didn’t hear what you said”) is put in the prompt queue together with the command instructions. The user must try again in the same state.

Utterances may be incorrect for other reasons. If the user for example says something that the recognizer cannot match against any grammar or if the user is in an environment with noise disturbing the recognition, the utterance may be rendered a low confidence score. This results in the utterance being marked as *rejected*. An utterance may also be noted as *STN* (Silence Too long Nothing understood). These kind of utterances are taken care of by the post function described in section 4.5.

4.3 Slots filled by phrases

In `dictationFunctionsState`, the main task is to play prompts, but in other states more work is done. In the `dictationPhraseState` for example, the dictated phrases are stored in an array. Earlier, in section 3.1.1, the grammatical differences between the `DynamicPhrase` grammar on the one hand and `Call`, `Tender` and `Meeting` on the other hand were described. This is how phrases covered by `DynamicPhrase`, filling the slot `phrase`, are handled:

```

if( doSlotIsFilled( interp, "phrase" ) ) {
    ERR_REPORT_AND_IGNORE(ivrRecInterpGetSlotValue(interp, "phrase", &pRecResult),
        "Failed to get phrase value" );

    CString dictatedPhrase = pRecResult;
    dictatedPhrases.Add( dictatedPhrase );

    [remaining code segments omitted]
}

```

First the interpretation `interp` is matched against the valid values of the slot. It is accepted if it is part of the phrases described by `DynamicPhrase`. The value is then transferred to a string, which is then saved in `dictatedPhrases`. This is the array where the dictated phrases are stored.

The phrases described by `Call`, `Tender` and `Meeting` are dealt with in a more complex and somewhat lengthy way:

```

else if( doSlotIsFilled( interp, "personal_name" ) ) {
    char *pHypText;
    ERR_REPORT_AND_IGNORE( ivrRecHypGetText( hyp, &pHypText ),
        "Failed to get hypothesis text" );
    CString hypText = pHypText;

    hypText.Remove( '0' );
    hypText.Remove( ';' );
    int hesStart = hypText.Find( '@' );
    if ( hesStart != -1 ) {
        hypText.Delete( hesStart, 6 );
    }

    CString dictatedPhrase = hypText;
    dictatedPhrases.Add( dictatedPhrase );

    [remaining code segments omitted]
}

```

When the slot `personal_name` is filled, this means that one of grammars `Call`, `Tender` or `Meeting` describes this phrase. The problem is that the interpretation only holds the name of a person since that is what fills the slot. What could be seen as raw material for a complete phrase is the hypothesis `hyp` used to get the interpretation. The actual text of the hypothesis, `pHypText`, is fetched and stored as the string `hypText`. The string looks a bit peculiar: *0;ring fredrik söderberg*. The initial symbols have to be removed by built-in functions to get a clean string. If there is hesitation in the utterance, it is transcribed as *@hes@* and the string would look like *0;@hes@ ring fredrik söderberg*. The hesitation has to be deleted together with the following space. After these operations the string correlates to a correct phrase, which is then stored in the phrase array.

4.4 Inserting dynamic grammars

As said in section 4.1 two of the three dynamic grammars are constructed in the main program of `Icepeak Attendant`, but `DynamicPhrase` depends on which template is chosen and cannot be put in until the template is known. This dynamic grammar is therefore created in the state where it is first used and where the dictation begins, `dictationPrePhraseState`.

The construction of a dynamic grammar is done in several steps. The example shows the creation and insertion of the grammar `DynamicPhrase`:

```
switch( GetPhrases( phrases, currentTemplateName ) ) {
case RET_DB_ERROR:
    break;
case RET_OK:
    phraseGSL = MakeGSLForPhrases( phrases );
    insertCallDurationDynamicGrammar( "DynamicPhrase", "dyn_phrase" );

    [remaining code segments omitted]
}
```

The function `GetPhrases` collects the data, phrases belonging to the given template, from the database with an SQL query. The phrases are then put in a string array, naturally called `phrases`. If the database operation succeeded the function `MakeGSLForPhrases` converts the elements of the array into a GSL string, a grammar. This then inserted into Icepeak Attendant's dynamic grammar database by `insertCallDurationDynamicGrammar`.

The grammars `DynamicPhrase`, `DynamicTemplate` and `DynamicPersonalName` can be used without being set in every state function, in comparison to `.Dictate`.

Dynamic grammars can be inserted in different ways, to last permanently or for the duration of the call. Since each caller has his or her own database, new dynamic grammars will be created for every user. In this application therefore, dynamic grammars are constructed during the call and cease to exist when the call ends.

4.5 The post function

After each state, before the transition to the next state is made, a post function is run. This is where faulty user responses are handled after the status of the utterance has been checked. If it is normal, the transition is made, but if it is marked as erroneous, e.g. rejected or silent, it is taken care of. The function consists of `if ... else` clauses, each describing a state and its specific error handling.

```
if( state == dictationFunctionsState ) {
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ursäkta" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_fortsätt" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_klar" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_avbryt" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_läs_upp_senaste" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_läs_upp" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ta_bort_senaste" );
    TRY_QUEUE_PROMPT( ivrChannel, "dictation_ta_bort" );
    TRY_SET_NEXT_STATE( state );
    return;
}
```

In this case, `dictationFunctionsState`, a prompt “Ursäkta, jag hörde inte vad du sa” (“Sorry, I didn’t hear what you said”) is queued, together with the prompts reading the dictation commands. The next state is set as the same state, with the result that the caller is taken back to the state to try again.

5 Evaluation of the dictation function

The central component in the dictation application is the dialogue engine, which is used over the phone and controls the interaction between user and system. A small-scale empirical evaluation was done to assess user satisfaction.

5.1 Approach

The evaluation was done by two users, both female and 25-30 years old. Their computer literacy was high and they were rather used to technical gadgets and interested in the technological evolution. They had tried voice-driven applications like timetable information but had not used a lot.

The users were given the assignment of dictating a number of messages in the telephone interface and testing the different commands etc. Thereafter they were asked about their experience, their general impression of the application, the smoothness of the dialogue and possible problems. If they encountered any problems, they were asked to describe the difficulties and account for how they perceived the system's ability to help them out of the troublesome situation. Further, they were questioned if they wanted to change something in the system's behaviour and what they would like to see as future developments of the application.

The exact questions asked can be seen in the appendix, both the Swedish original and an English translation.

5.2 Users' comments

The users' remarks are presented in this section. They are divided into three groups, presented below.

To give the readers a better understanding of the interaction between user and system, an authentic example of a simple dictation translated into English, is given in figure 5.1. Prompts are typed within quotation marks, speech synthesis as bold text and user utterances as italic.

5.2.1 General impression and remarks on the dialogue

The users' impression was that the application was easy to use, that the instructions were clear and that the application was simple and easy to understand. They also found it rather easy to remember the different commands and thought it was good that a help menu was at hand, with assistance and reminders of the available commands.

According to the users, the prompts gave enough information to solve the task, but one user thought that a beginner might want a little more information. The possibility of barge-in was perceived as very good for a more experienced user.

One user chose an unavailable command when editing a phrase, leading to the post function being activated. The prompt "Ursäkta, jag hörde inte vad du sa" ("Sorry, I didn't hear what you said") was played together with the editing instructions and then the same phrase to be edited was played again. The user thought this was good since this meant a second chance at editing the phrase.

“Who would you like to call?”
Dictation
 “To choose dictation template, say its name”
To do list
 “You want to dictate **to do list**, is that correct?”
Yes
 “Please dictate”
Read books
 “Please dictate”
Done
 “Would you like to listen to the message and edit it?”
Yes
 “The message will be read phrase by phrase.
 To accept the phrase, say Next.
 To remove the phrase, say Remove.
 To Add a new phrase, say Add.
 To replace the phrase with a new one, say Replace.”
Read books
Replace
 “Please dictate the new phrase.”
Read report
 “The phrase has been replaced.
 The message has been read. Do you want to listen to it again and edit it?”
No
 “The message has been dictated.
 To dictate another message, say Dictate.
 To finish, say Finish.”
Finish
 “Who would you like to call?”
[Hangup]

Figure 5.1: Example of system-user interaction

5.2.2 Problems

In the loop in dictationPhraseState, where the phrases are dictated, the users thought that another prompt would be more appropriate. The present prompt is “Varsågod att diktera” (“Please dictate”), which is the same as the one initiating the loop. The users commented on this to be a little confusing. It could lead to misunderstandings, such that the user would think that the system didn’t recognize the phrase dictated and therefore keeps repeating the same prompt over and over again. The users’ suggestion for a new prompt would be something like “Phrase understood, please go on”.

One of the users had a dialect which the system had a little trouble recognizing. This user thought that the three tries, which the system gives you before the call is aborted, were not enough. At least six tries would be desirable.

5.2.3 Future changes and development

The users could see themselves using the product, if it wasn’t too expensive. They thought it would be very interesting and useful, especially if the service would be connected to a calendar, where dictated appointments etc. would be automatically scheduled. They wouldn’t want to see the dictation application

and its dialogue get too complicated, but to be kept as simple as possible if other types of templates etc. would be added.

One feature the users would like to have was to be able to dictate all phrases of the message at once, without having to wait for a confirming prompt. However, the users understood the problem of segmentation, that it would be difficult for the recognition software to tell where one phrase ends and another begins.

5.3 Summary of the evaluation results

The users thought that the dictation application was rather easy to use. It had some flaws though, but no larger problems were encountered. If any problems came up, the users asked the system for help and solved it through the assistance given by the system prompts. They thought that the information given by the prompts were sufficient to solve the task and that the system seemed adapted to both beginners and more experienced users.

6 Design of a web interface to the database

As every other component of Icepeak Attendant, the dictation application consists of two interfaces: one controlled by the voice and one accessed from the Internet. The web interface manipulates the database, whose contents are the basis for the operations performed in the voice interface. The functions are yet to be implemented, but descriptions of the future use of the interface will be given in connection to the interface images.

A design for the interface has been developed, based on the already existing interfaces connected to other Icepeak Attendant applications. Its structure should be efficient and compact without being cluttered.

The presentation pages are coded in HTML, JavaScript and JSP (JavaServer Pages) because of its dynamic contents (see Wilton (2000) and Bergsten (2001) for information about the programming languages).

The interface images presented in this chapter show the dictation interface design integrated with the main interface for other services in Icepeak Attendant.

All phrases, which the user can dictate, must be defined as belonging to a template. The basis of recognition is what is in the database, whose contents the user controls and modifies through the web interface. Available activities are:

- Add, remove, rename templates and edit the contents
- Copy phrases between templates
- Edit and remove a phrase in multiple templates

Each of the views embodying these activities will be presented in the following sections. The interface images here are all in Swedish, but in the appendix are the equivalent images from the English version of the interface. The template names and phrases though are in Swedish since they are the actual contents of the database.

In the menu to the left (shown in the interface images) the user can choose between the different options and views. There are three main views connected to dictation here, template overview shown in figure 6.1, copying phrases displayed in figure 6.4 and editing a phrase in multiple templates shown in figure 6.5. The other views shown in this chapter are initiated from these views and to make it simple, the sections group and describe the connected views together.

6.1 Template overview

The dictation application comes with two templates: att göra-lista (to do list) and anteckning (notes). The user may want to add other templates, remove templates if they aren't used and possibly change the

names of the templates.

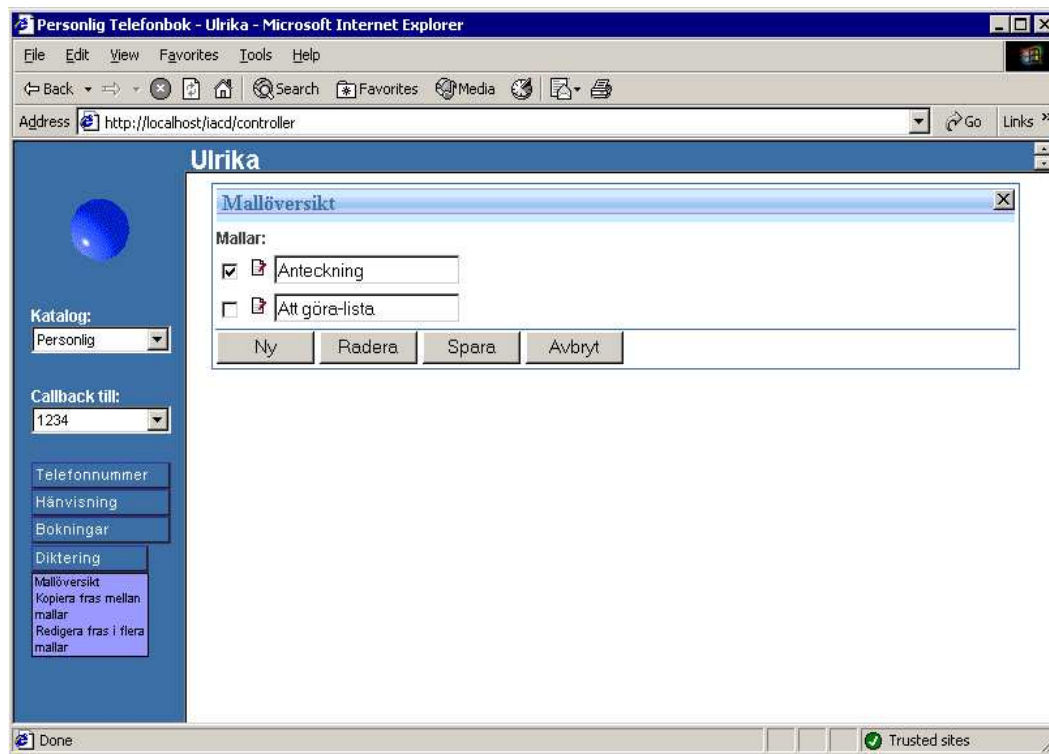


Figure 6.1: Showing template overview

The user is presented with a list of her/his templates and several functions are now available: remove, rename or add a template and also edit the contents of a template. These activities are described in the next three sections, but they are initiated from the interface presented in figure 6.1.

6.1.1 Renaming and removing a template

The name of a template is changed in two steps: the user replaces the template's name in the textfield with the new one and then selects *Spara* (Save).

If the user wants to delete a template and its contents, she/he selects the template by marking the corresponding checkbox and then selecting *Radera* (Erase). The template is removed from the overview and its contents are deleted from the database.

The button *Avbryt* (Abort) cancels all changes that has not been saved yet.

6.1.2 Adding a template

To add a new template the user chooses *Ny* (New) and is shown the view in figure 6.2.

She/he can then enter the name of the template. When the template is saved, it will be shown in the overview together with the others.

6.1.3 Editing the contents of a template

If the user wants to modify the phrases belonging to a template, she/he selects the little icon next to the template's name, leading to the editing view in figure 6.3.

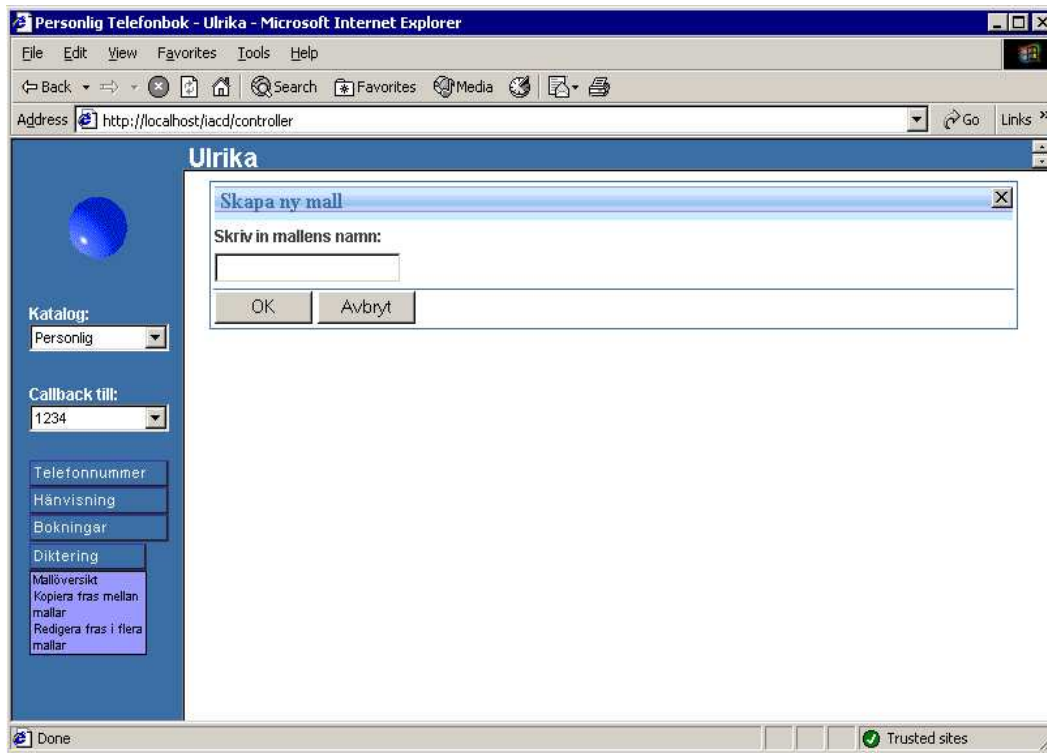


Figure 6.2: Adding a new template

Editing the template's contents includes removing a phrase, changing a phrase and adding new phrases. Users may need some guidance through this part of the interface and therefore a pop-up window with instructions and tips can be reached by pressing the button *Hjälp* (*Help*).

A phrase is removed from the template when the user doubleclicks on it in the phrase list. This causes the phrase to be marked grey. If the user wants the phrase back in the template, she/he doubleclicks on it again. The contents of the template remains unchanged until *Spara* is selected. The button *Avbryt* cancels any changes and leaves the contents of the template as they were when the template was last saved. The user may want to edit a phrase because it is spelled incorrectly etc. The phrase is selected from the phrase list when the user marks it and selects *Ändra* (*Edit*). It is then transferred to the textfield where the changes are done before it's being transferred back to the phrase list by a click on the button *Lägg till* (*Add*).

The same textfield is used when a new phrase is added: the phrase is entered in the field and then added to the list. Phrases can also be imported from a file in the frame on the right. The user simply selects a text file (by typing the file name in the textfield or by browsing the file hierarchy and selecting it from there) and clicks *OK*. The phrases in the file are now added to the template and are shown in the list.

6.2 Copying phrases between templates

It is almost a necessity that phrases can be part of several templates, especially if they are function words or other frequently used words. It would be very tiresome if the user had to add the same phrase to each and everyone of the templates, so to avoid this there is an interface for copying phrases between templates, displayed in figure 6.4.

The user chooses a template in the scrollable list up in the right corner and all phrases belonging to this one are displayed. She/he selects phrases for copying by marking them in the list and clicking on

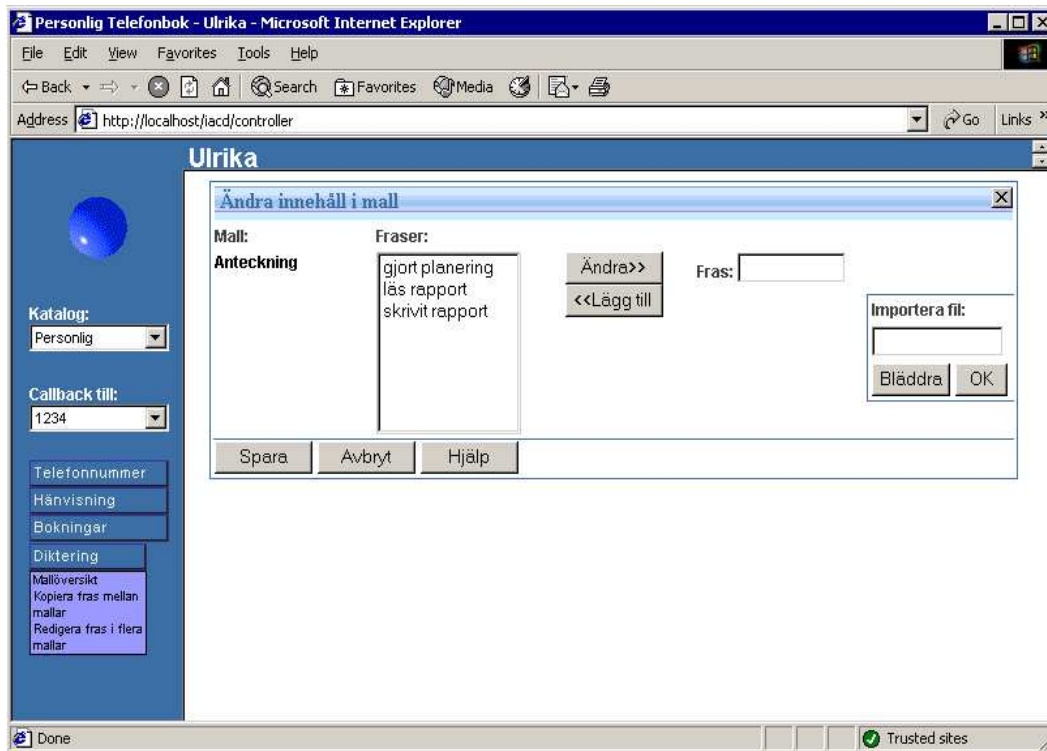


Figure 6.3: Editing a template's contents

Lägg till. The chosen phrases are shown in the list in the middle. Phrases can be moved between this list and the one to the left with the help of *Lägg till* and *Ta bort* (Remove).

When the user is satisfied with the list of phrases and wants to copy, she/he selects the target template(s) and selects *OK*. The phrases are now added to the chosen templates.

If the user should choose to add a phrase to a template to which it already belongs, the particular phrase isn't copied to this template since the database doesn't accept identical rows.

This view might present some difficulties to inexperienced users and there are guidelines available through *Hjälp*, in the same way as described earlier in section 6.1.3. The process can at any time be aborted by *Avbryt*, which cancels all changes.

6.3 Modifying phrases in multiple templates

If the same phrase is part of several templates and the user decides to delete or change it, it is most efficient if the phrase can be modified in all templates at once.

6.3.1 Selecting the phrase

The first thing the user has to do, is picking the phrase to be modified (figure 6.5).

All phrases of all templates are listed here and the user can select a phrase by marking the corresponding checkbox and clicking on *OK*. She/he can also search for the phrase by entering it in the textfield up to the left and selecting *Sök* (Search). Another possibility is to show all phrases in a particular template by selecting the template in the drop-down menu. This makes it easier for the user if the template is known, but not the exact phrase, since she/he doesn't have to look through the whole list. In this case though, no particular template has been chosen, all phrases are shown regardless of what template they belong to.

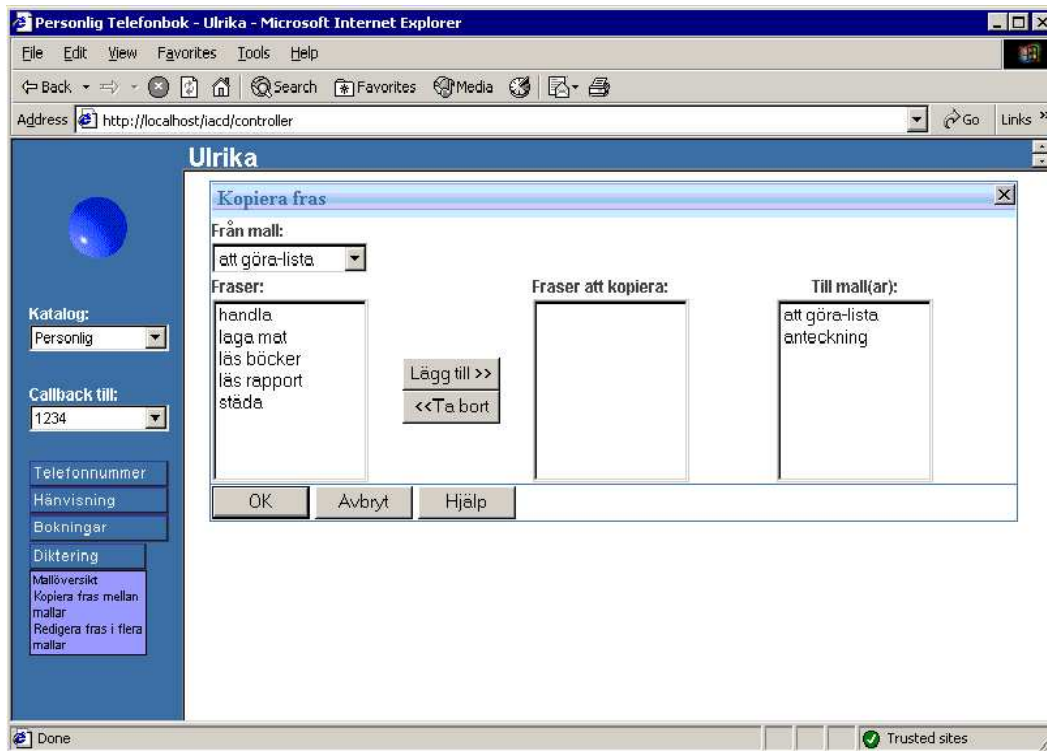


Figure 6.4: Copying phrases

6.3.2 Editing the phrase

Once the phrase is chosen it can be altered (figure 6.6).

The phrase is shown in the textfield to the right. The templates, to which it belongs, are shown to the left. The user edits the phrase (by correcting spelling mistakes for example), selects which templates the changes will apply to and clicks on *Spara* to save the changes.

If the phrase is to be removed, the process is repeated up to the last step, where *Ta bort* is chosen. The phrase is now removed from the selected templates.

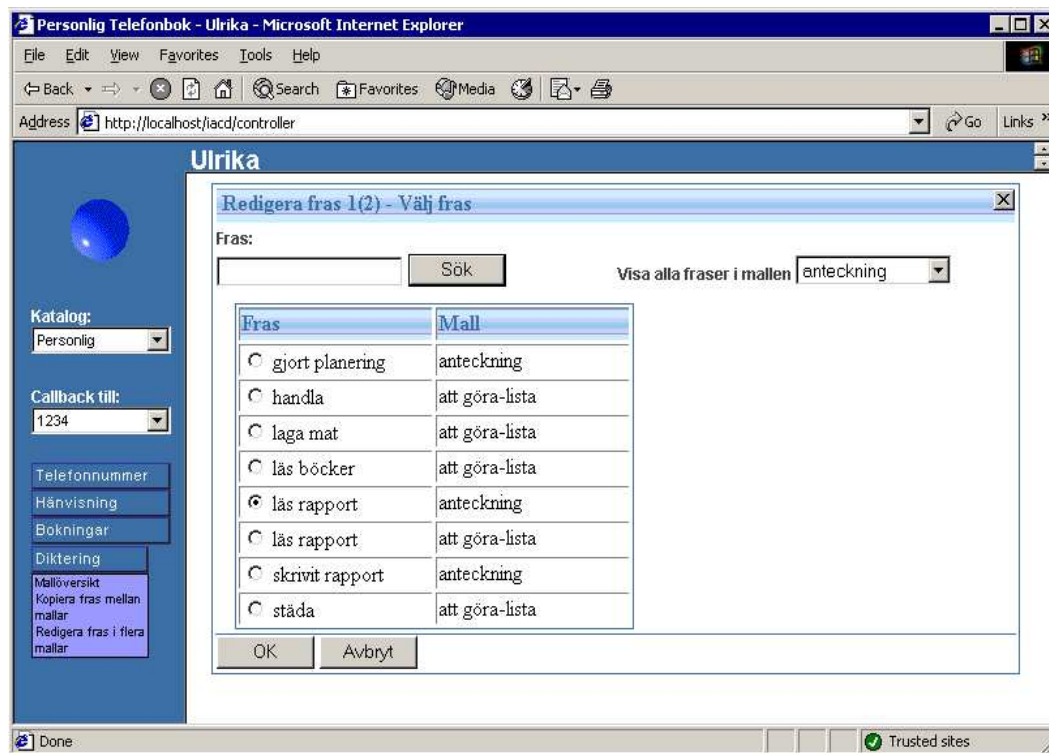


Figure 6.5: Selecting a phrase

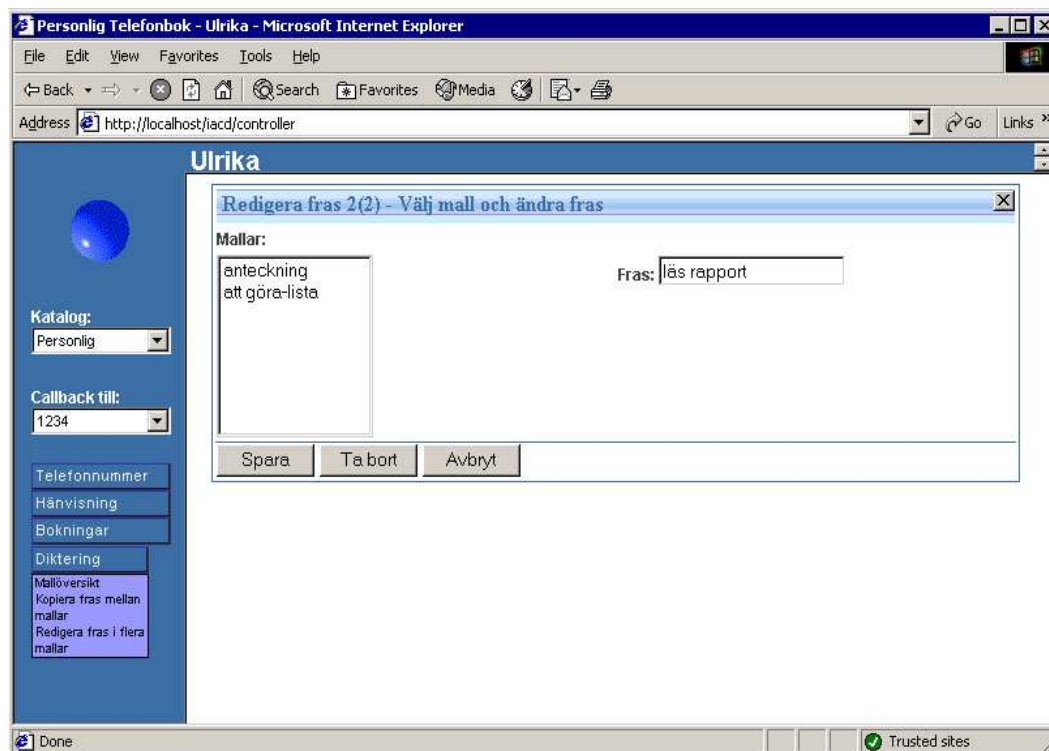


Figure 6.6: Editing a phrase

7 Concluding remarks

A small informal evaluation of the telephone interface has been done, where two users gave their points of view on the application today and possible changes, improvements and future developments. Their opinions were mostly positive, but they had some comments on the wording of a prompt and also the number of chances given to the user before the call is aborted due to too many faulty tries. On the whole though, they thought that the interaction between user and system worked well and they were able to dictate messages successfully.

The finished system enables users to dictate messages using their most natural tool, the voice. This means that messages can be dictated in a time-saving manner anywhere where the user has access to a telephone. The application can thus be used in environments where paper and pen fail, as in cars etc. through the use of hands-free mobile phones. The dictated messages are kept available to the user in the computer, in front of which a lot of people spend their day.

7.1 The dictation component

This thesis describes the development of a dictation application using Nuance technology. The system is divided in two parts: a telephone interface and a web interface. The main part, the telephone interface with the interaction between system and user, consists of a dialogue engine where the system utterances are pre-recorded audio segments and the user responses are recognized through being predicted in grammars.

The grammars are collections of rules, also called grammars. They are coded in a special language for Nuance grammars, GSL (Grammar Definition Language). The top level grammar `.Dictate` contains subgrammars, which describes all possible user utterances, commands and other expressions such as phrases etc. The dynamic grammars `DynamicTemplate`, `DynamicPhrase` and `PersonalName` are made out of the contents of the database tables *templatetable*, *phrasetable* and *phonebook*.

The process of building the dialogue engine consists of several steps and these have been described, from writing the grammars, designing flow charts and preparing the wording of the prompts to actually coding the program in C++. The main program can be compared to a finite automaton and all states are represented in the program by functions. They share the same general structure and code snippets have been given to illustrate this structure and to show how the functions work. The connections and transitions between the states have been presented in the text and a more detailed view can be seen in the appendix.

The phrases to be recognized are stored in a database and this is modified through a web interface. A design for this interface has been developed and displayed through screen images. The functions yet to be implemented have been described.

7.2 Discussion

The evaluation shows that the dictation dialogue in the telephone interface works well. The users were able to dictate their messages and test the different functions. They thought it was rather simple to use,

but they had some comments on changes that would make the application even more easy to use. One of the users' suggestions on future developments is very similar to one of those already considered by the developer. The idea is about creating a connection between the dialogue engine and a calendar, perhaps the one in Microsoft Outlook. A dictated message containing booking of an appointment etc. could then be automatically scheduled in the calendar. Since both developer and user had this idea, it seems to be a promising development that would appeal to many people. This and several other ideas for future versions of the dictation application are described in the next section.

Building a system in this way with Nuance technology works well. The software had some problems recognizing one of the evaluating user's dialect, but it was no major problem.

Several tools for assisting in the development process are provided by Nuance, as the tools for grammar testing described in section 2.4. They are very well suited for detecting problems a developer could encounter when writing grammars etc. and proved to be of great help.

7.3 Future development

The dictation application is something with great future potential. Developments could mean anything from a simple expansion of the functions available in the dictation dialogue to more complex applications where the dictation function plays a big part. The potential for this kind of program seems almost endless.

Tools that could benefit from the dictation application are voice controlled programs for ordering goods over the phone or applications about sending and receiving e-mails.

The expansion of dictation could include such functions as choosing format of the dictated message. It could be written to a file, delivered as an e-mail or as an SMS, stored in a database or integrated with Microsoft Outlook's Notes or Calendar. It could also be possible for the user to choose several different formats for the same message.

The dictation tool could be equipped with more powerful templates for dictation of meeting minutes, inspection reports, installation records, sales meeting follow-ups etc. One idea is to provide each template with a type, which will decide the appropriate dialogue. Dictation of a report is more complicated than dictation of some notes and will therefore need a more extensive dialogue and editing function. By having different kinds of template types the dialogue will adjust to the message dictated and the system will be more flexible.

In the present application, there are three phrases with variable contents of names and date/time, described by the grammars `Call`, `Meeting` and `Tender`. One development of this could be to let the user create her/his own phrases with variables for names, numbers, time/date, measurements etc. This would be done via the web interface, perhaps with checkboxes for the different variables for the user to select. The choices would then be translated to GSL strings representing grammars like `DateTime`, described in 3.1.2.

As things stand today, phrases not recognized by the system are discarded, but one possible development could be that these are stored as audio files. These files, containing the uninterpreted phrases, can be used as a help for completing the message manually. The complete dictated message itself could also be stored as an audio file to help in this work.

In the present application the messages dictated, stored in files, are unreachable from the telephone interface as well as the web interface. In the future though, it could be possible for the user to edit the messages already dictated through the telephone interface.

References

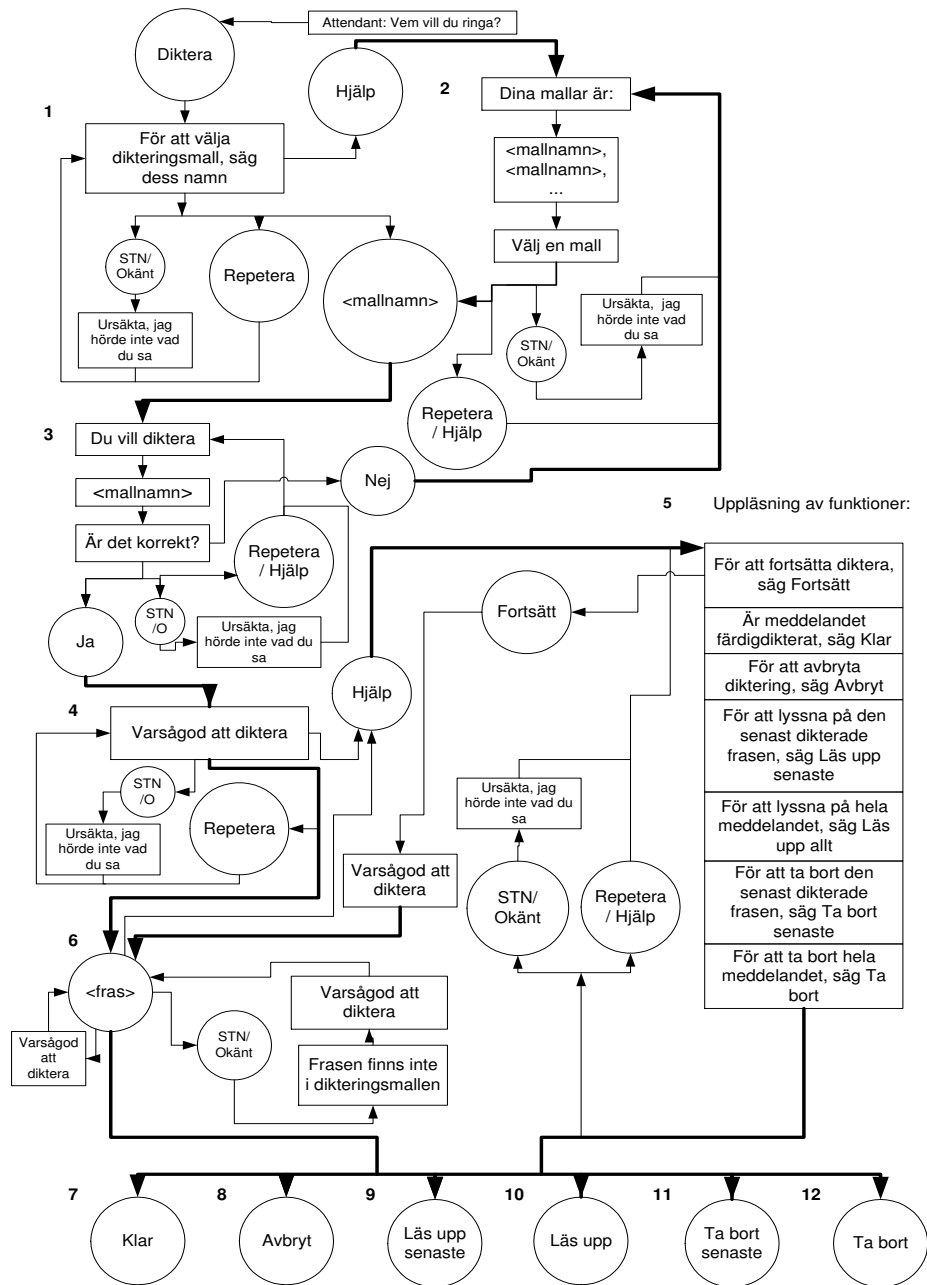
Bergsten, Hans. (2001). *JavaServer Pages*. Sebastopol: O'Reilly.

Nuance System. (1998). *Developer's Manual*. Nuance Communications Inc., Menlo Park, California.

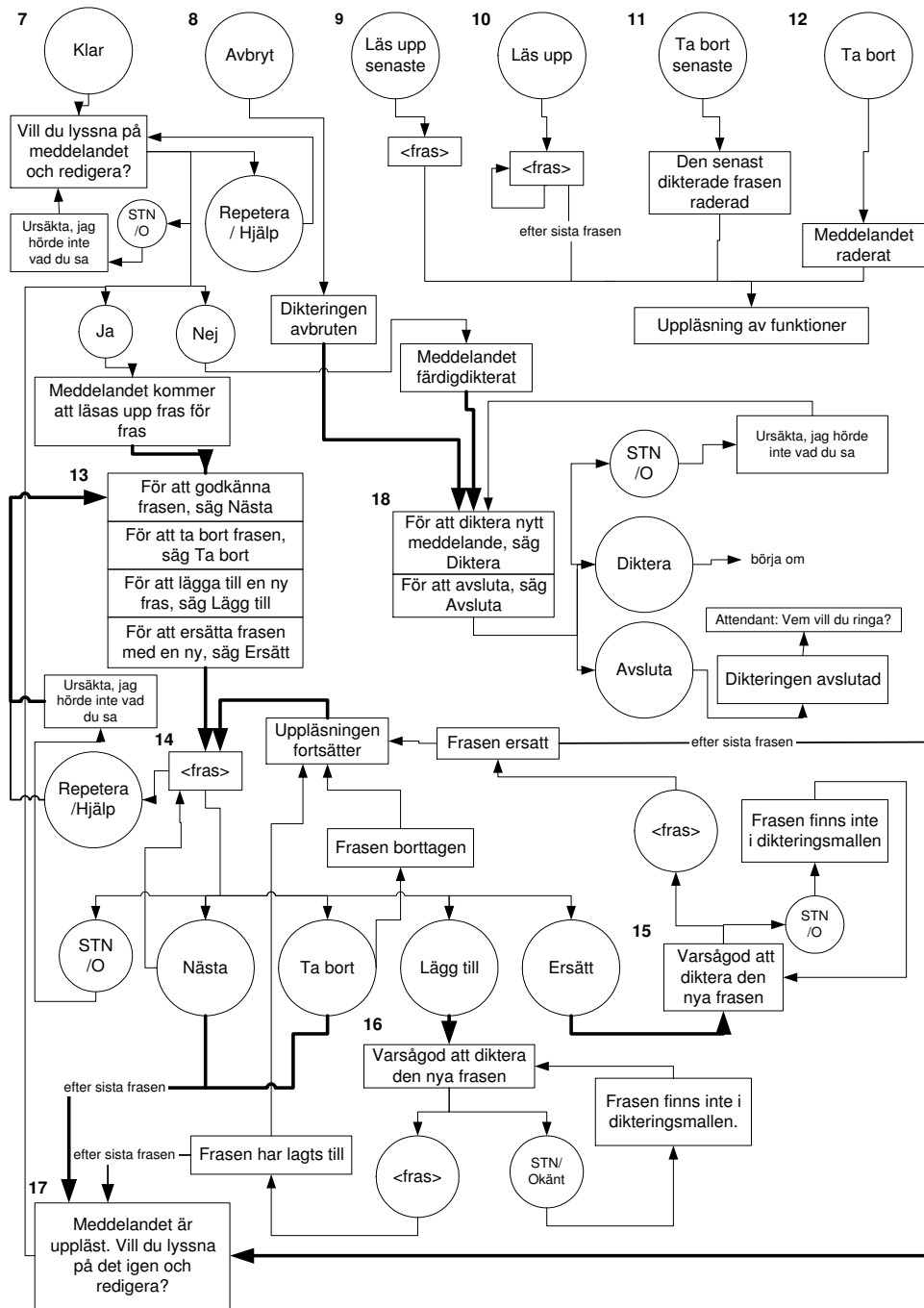
Skansholm, Jan. (1996). *C++ direkt*. Lund: Studentlitteratur.

Wilton, Paul. (2000). *Beginning JavaScript*. Birmingham: Wrox Press Ltd.

A Dialogue flow charts



continues on next page →



B Evaluation questions

1. Vad är ditt generella intryck av dikteringstjänsten?
2. Var applikationen lätt eller svår att använda? Vad var det som gjorde att det var lätt/svårt?
3. Hur fungerade det att diktera meddelandet, flöt det på eller fastnade du nån gång?
4. Stötte du på några problem? Om ja, vilken sorts problem och lyckades du lösa dem?
5. Var hjälpen som systemet kunde erbjuda bra eller kunde den ha varit strukturerad på något annat sätt?
6. Verkar applikationen lätt eller svår att lära sig?
7. Vad tyckte du om tiden det tog att diktera ett meddelande, tog det för lång tid eller hade du behövt mer?
8. Hur upplevde du promptarna, fick du den information du behövde, var det för mycket eller hade du velat ha mer?
9. Var det någonting som inte fungerade så bra som du skulle vilja ändra på?
10. Skulle du kunna tänka dig att använda en sådan här produkt?
11. Hur skulle du vilja att applikationen utvecklas i framtiden? Vilka funktioner skulle du vilja ha, som kanske saknas nu?

-
1. What is your general impression of the dictation service?
 2. Was the application easy or difficult to use? What made it easy/difficult?
 3. How did it work to dictate the message, was it smooth or did you at any time get stuck?
 4. Did you encounter any problems? If yes, what kind of problems and did you manage to solve them?
 5. Was the assistance the system could offer helpful or could it have been structured in another way?
 6. Does the application seem easy or difficult to learn?
 7. What did you think of the time required for dictating a message, did it take too long or was it too hasty?
 8. What was your experience of the prompts, did you get the amount of information you needed, was it too much or would you have wanted more?
 9. Was there something that didn't work so well that you would like to change?
 10. Could you see yourself using this product?
 11. How would you like to see the application develop in the future? What functions would you like to have, perhaps missing now?

C Interface images in English

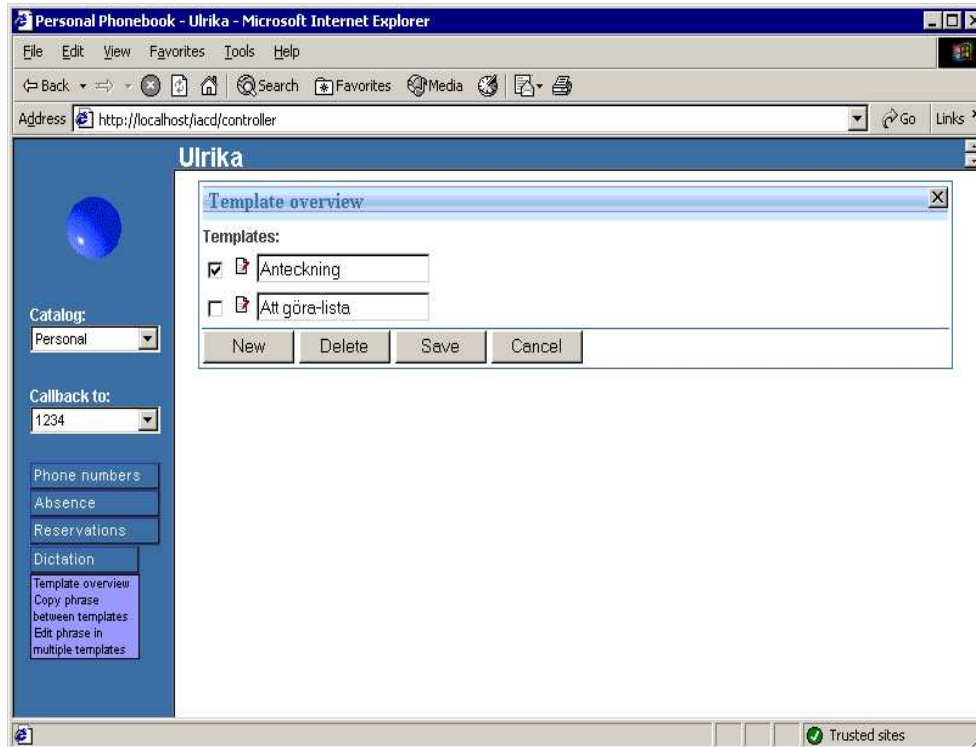


Figure C.1: Showing template overview

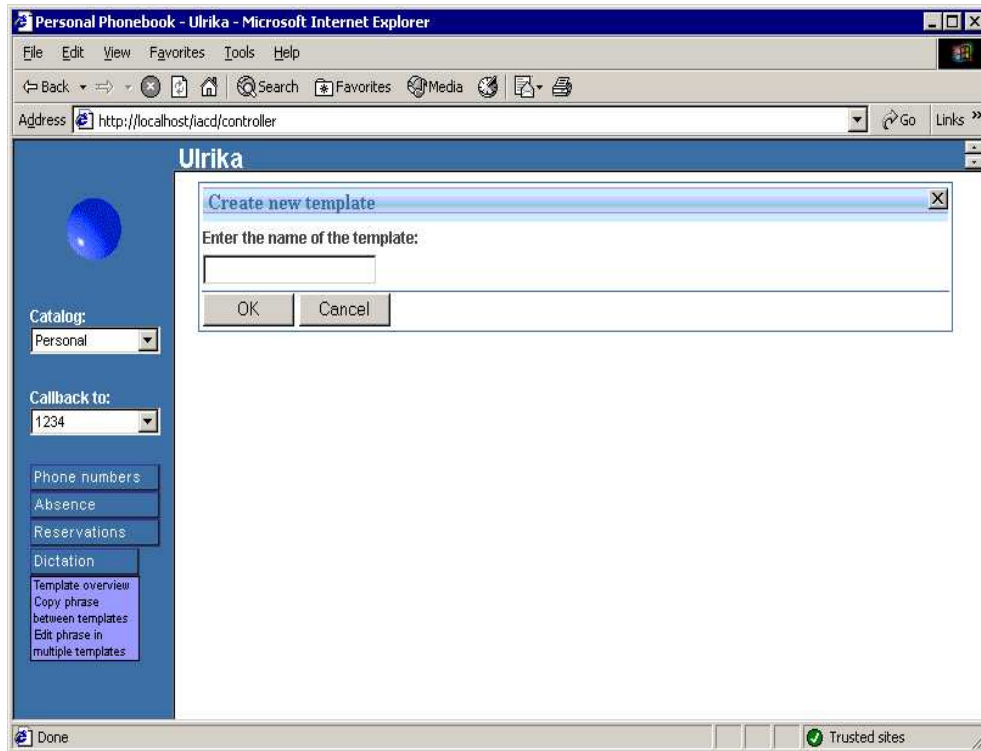


Figure C.2: Adding a new template

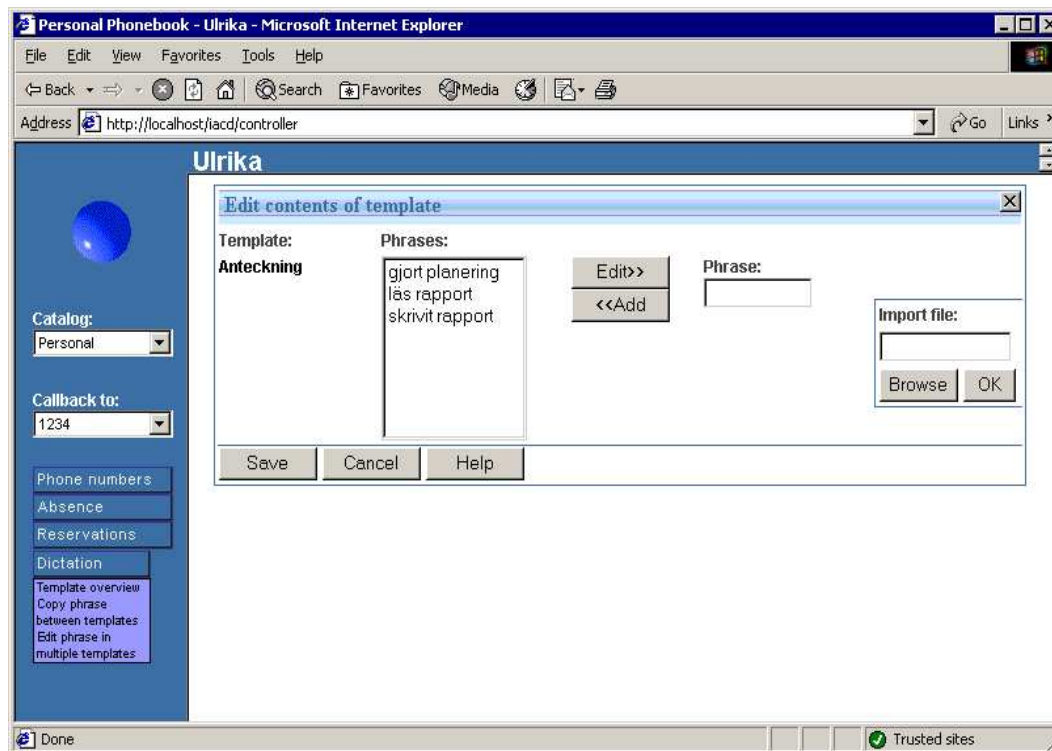


Figure C.3: Editing a template's contents

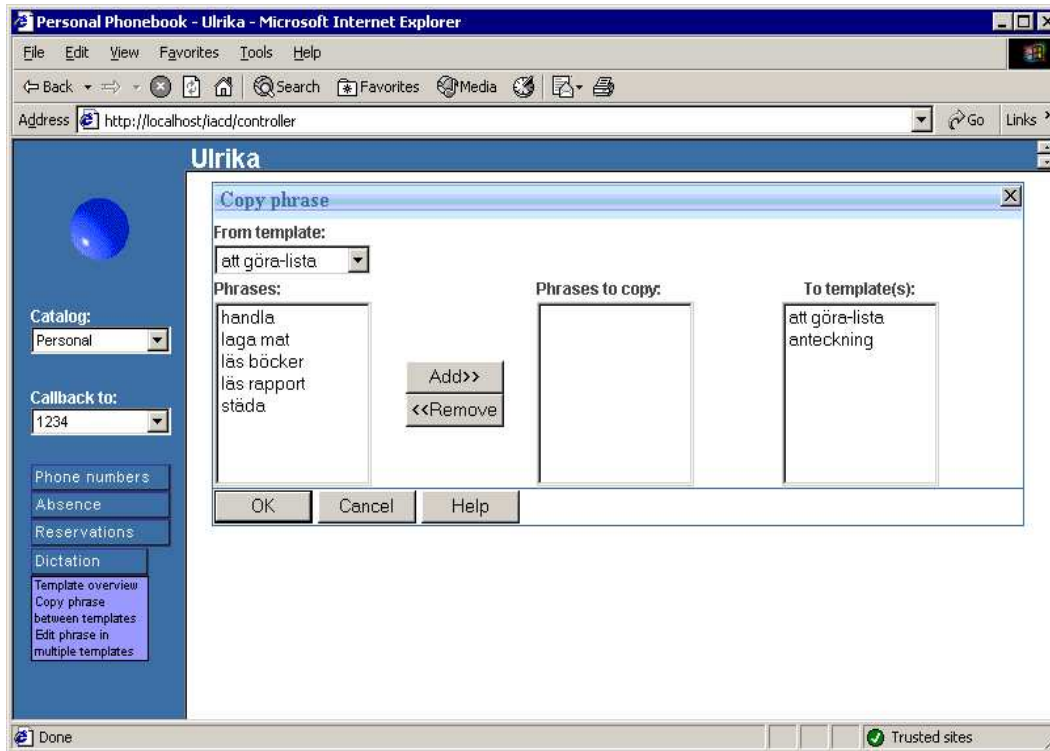


Figure C.4: Copying phrases between templates

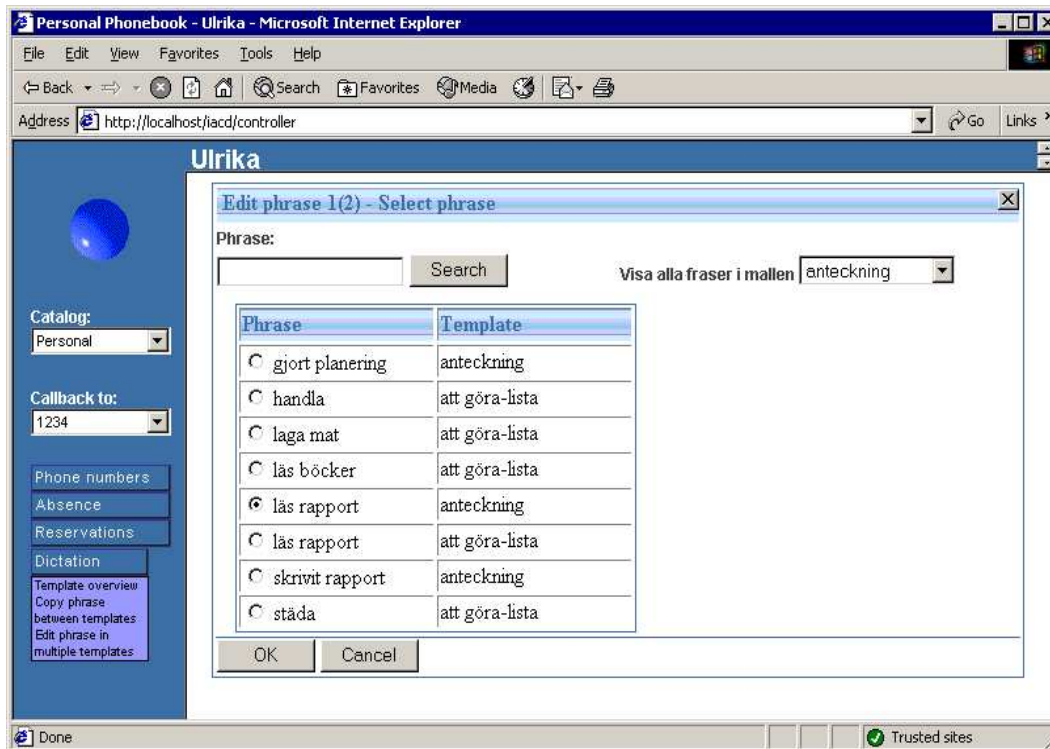


Figure C.5: Selecting a phrase

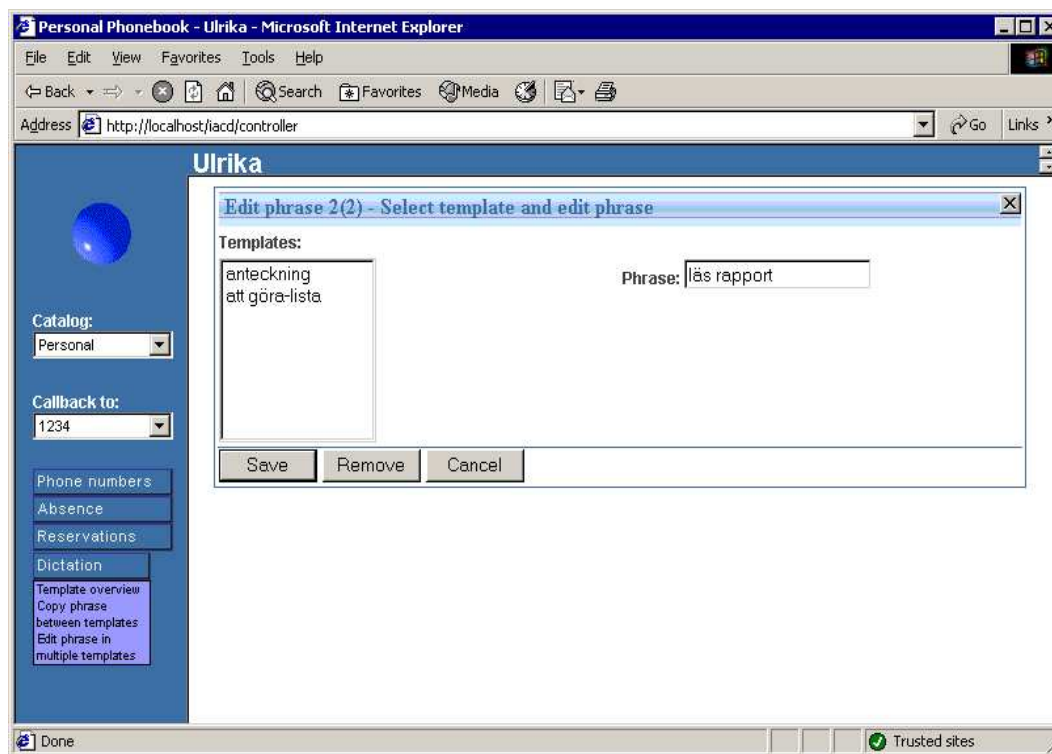


Figure C.6: Editing a phrase